
PART III

Sorting and Searching

Internal Sorting

We sort many things in our everyday lives: A handful of cards when playing Bridge; bills and other piles of paper; jars of spices; and so on. And we have many intuitive strategies that we can use to do the sorting, depending on how many objects we have to sort and how hard they are to move around. Sorting is also one of the most frequently performed computing tasks. We might sort the records in a database so that we can search the collection efficiently. We might sort the records by zip code so that we can print and mail them more cheaply. We might use sorting as an intrinsic part of an algorithm to solve some other problem, such as when computing the minimum-cost spanning tree (see Section 11.5).

Because sorting is so important, naturally it has been studied intensively and many algorithms have been devised. Some of these algorithms are straightforward adaptations of schemes we use in everyday life. Others are totally alien to how humans do things, having been invented to sort thousands or even millions of records stored on the computer. After years of study, there are still unsolved problems related to sorting. New algorithms are still being developed and refined for special-purpose applications.

While introducing this central problem in computer science, this chapter has a secondary purpose of illustrating issues in algorithm design and analysis. For example, this collection of sorting algorithms shows multiple approaches to using divide-and-conquer. In particular, there are multiple ways to do the dividing: Mergesort divides a list in half; Quicksort divides a list into big values and small values; and Radix Sort divides the problem by working on one digit of the key at a time. Sorting algorithms can also illustrate a wide variety of analysis techniques. We'll find that it is possible for an algorithm to have an average case whose growth rate is significantly smaller than its worse case (Quicksort). We'll see how it is possible to speed up sorting algorithms (both Shellsort and Quicksort) by taking advantage of the best case behavior of another algorithm (Insertion sort). We'll see several examples of how we can tune an algorithm for better performance. We'll see that special case behavior by some algorithms makes them a good solution for

special niche applications (Heapsort). Sorting provides an example of a significant technique for analyzing the lower bound for a problem. Sorting will also be used to motivate the introduction to file processing presented in Chapter 8.

The present chapter covers several standard algorithms appropriate for sorting a collection of records that fit in the computer's main memory. It begins with a discussion of three simple, but relatively slow, algorithms requiring $\Theta(n^2)$ time in the average and worst cases. Several algorithms with considerably better performance are then presented, some with $\Theta(n \log n)$ worst-case running time. The final sorting method presented requires only $\Theta(n)$ worst-case time under special conditions. The chapter concludes with a proof that sorting in general requires $\Omega(n \log n)$ time in the worst case.

7.1 Sorting Terminology and Notation

Except where noted otherwise, input to the sorting algorithms presented in this chapter is a collection of records stored in an array. Records are compared to one another by means of a comparator class, as introduced in Section 4.4. To simplify the discussion we will assume that each record has a key field whose value is extracted from the record by the comparator. The key method of the comparator class is **prior**, which returns true when its first argument should appear prior to its second argument in the sorted list. We also assume that for every record type there is a **swap** function that can interchange the contents of two records in the array (see the Appendix).

Given a set of records r_1, r_2, \dots, r_n with key values k_1, k_2, \dots, k_n , the **Sorting Problem** is to arrange the records into any order s such that records $r_{s_1}, r_{s_2}, \dots, r_{s_n}$ have keys obeying the property $k_{s_1} \leq k_{s_2} \leq \dots \leq k_{s_n}$. In other words, the sorting problem is to arrange a set of records so that the values of their key fields are in non-decreasing order.

As defined, the Sorting Problem allows input with two or more records that have the same key value. Certain applications require that input not contain duplicate key values. The sorting algorithms presented in this chapter and in Chapter 8 can handle duplicate key values unless noted otherwise.

When duplicate key values are allowed, there might be an implicit ordering to the duplicates, typically based on their order of occurrence within the input. It might be desirable to maintain this initial ordering among duplicates. A sorting algorithm is said to be **stable** if it does not change the relative ordering of records with identical key values. Many, but not all, of the sorting algorithms presented in this chapter are stable, or can be made stable with minor changes.

When comparing two sorting algorithms, the most straightforward approach would seem to be simply program both and measure their running times. An example of such timings is presented in Figure 7.20. However, such a comparison

can be misleading because the running time for many sorting algorithms depends on specifics of the input values. In particular, the number of records, the size of the keys and the records, the allowable range of the key values, and the amount by which the input records are “out of order” can all greatly affect the relative running times for sorting algorithms.

When analyzing sorting algorithms, it is traditional to measure the number of comparisons made between keys. This measure is usually closely related to the running time for the algorithm and has the advantage of being machine and data-type independent. However, in some cases records might be so large that their physical movement might take a significant fraction of the total running time. If so, it might be appropriate to measure the number of swap operations performed by the algorithm. In most applications we can assume that all records and keys are of fixed length, and that a single comparison or a single swap operation requires a constant amount of time regardless of which keys are involved. Some special situations “change the rules” for comparing sorting algorithms. For example, an application with records or keys having widely varying length (such as sorting a sequence of variable length strings) will benefit from a special-purpose sorting technique. Some applications require that a small number of records be sorted, but that the sort be performed frequently. An example would be an application that repeatedly sorts groups of five numbers. In such cases, the constants in the runtime equations that are usually ignored in an asymptotic analysis now become crucial. Finally, some situations require that a sorting algorithm use as little memory as possible. We will note which sorting algorithms require significant extra memory beyond the input array.

7.2 Three $\Theta(n^2)$ Sorting Algorithms

This section presents three simple sorting algorithms. While easy to understand and implement, we will soon see that they are unacceptably slow when there are many records to sort. Nonetheless, there are situations where one of these simple algorithms is the best tool for the job.

7.2.1 Insertion Sort

Imagine that you have a stack of phone bills from the past two years and that you wish to organize them by date. A fairly natural way to do this might be to look at the first two bills and put them in order. Then take the third bill and put it into the right order with respect to the first two, and so on. As you take each bill, you would add it to the sorted pile that you have already made. This naturally intuitive process is the inspiration for our first sorting algorithm, called **Insertion Sort**. Insertion Sort iterates through a list of records. Each record is inserted in turn at the correct position within a sorted list composed of those records already processed. The

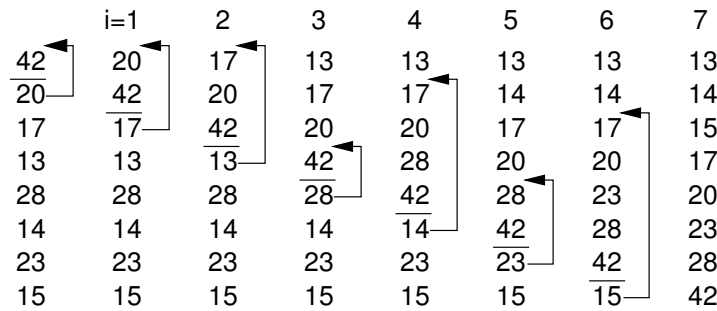


Figure 7.1 An illustration of Insertion Sort. Each column shows the array after the iteration with the indicated value of **i** in the outer **for** loop. Values above the line in each column have been sorted. Arrows indicate the upward motions of records through the array.

following is a C++ implementation. The input is an array of n records stored in array **A**.

```
template <typename E, typename Comp>
void inssort(E A[], int n) { // Insertion Sort
    for (int i=1; i<n; i++)    // Insert i'th record
        for (int j=i; (j>0) && (Comp::prior(A[j], A[j-1])); j--)
            swap(A, j, j-1);
}
```

Consider the case where **inssort** is processing the i th record, which has key value X . The record is moved upward in the array as long as X is less than the key value immediately above it. As soon as a key value less than or equal to X is encountered, **inssort** is done with that record because all records above it in the array must have smaller keys. Figure 7.1 illustrates how Insertion Sort works.

The body of **inssort** is made up of two nested **for** loops. The outer **for** loop is executed $n - 1$ times. The inner **for** loop is harder to analyze because the number of times it executes depends on how many keys in positions 1 to $i - 1$ have a value less than that of the key in position i . In the worst case, each record must make its way to the top of the array. This would occur if the keys are initially arranged from highest to lowest, in the reverse of sorted order. In this case, the number of comparisons will be one the first time through the **for** loop, two the second time, and so on. Thus, the total number of comparisons will be

$$\sum_{i=2}^n i \approx n^2/2 = \Theta(n^2).$$

In contrast, consider the best-case cost. This occurs when the keys begin in sorted order from lowest to highest. In this case, every pass through the inner **for** loop will fail immediately, and no values will be moved. The total number

of comparisons will be $n - 1$, which is the number of times the outer **for** loop executes. Thus, the cost for Insertion Sort in the best case is $\Theta(n)$.

While the best case is significantly faster than the worst case, the worst case is usually a more reliable indication of the “typical” running time. However, there are situations where we can expect the input to be in sorted or nearly sorted order. One example is when an already sorted list is slightly disordered by a small number of additions to the list; restoring sorted order using Insertion Sort might be a good idea if we know that the disordering is slight. Examples of algorithms that take advantage of Insertion Sort’s near-best-case running time are the Shellsort algorithm of Section 7.3 and the Quicksort algorithm of Section 7.5.

What is the average-case cost of Insertion Sort? When record i is processed, the number of times through the inner **for** loop depends on how far “out of order” the record is. In particular, the inner **for** loop is executed once for each key greater than the key of record i that appears in array positions 0 through $i - 1$. For example, in the leftmost column of Figure 7.1 the value 15 is preceded by five values greater than 15. Each such occurrence is called an **inversion**. The number of inversions (i.e., the number of values greater than a given value that occur prior to it in the array) will determine the number of comparisons and swaps that must take place. We need to determine what the average number of inversions will be for the record in position i . We expect on average that half of the keys in the first $i - 1$ array positions will have a value greater than that of the key at position i . Thus, the average case should be about half the cost of the worst case, or around $n^2/4$, which is still $\Theta(n^2)$. So, the average case is no better than the worst case in asymptotic complexity.

Counting comparisons or swaps yields similar results. Each time through the inner **for** loop yields both a comparison and a swap, except the last (i.e., the comparison that fails the inner **for** loop’s test), which has no swap. Thus, the number of swaps for the entire sort operation is $n - 1$ less than the number of comparisons. This is 0 in the best case, and $\Theta(n^2)$ in the average and worst cases.

7.2.2 Bubble Sort

Our next sorting algorithm is called **Bubble Sort**. Bubble Sort is often taught to novice programmers in introductory computer science courses. This is unfortunate, because Bubble Sort has no redeeming features whatsoever. It is a relatively slow sort, it is no easier to understand than Insertion Sort, it does not correspond to any intuitive counterpart in “everyday” use, and it has a poor best-case running time. However, Bubble Sort can serve as the inspiration for a better sorting algorithm that will be presented in Section 7.2.3.

Bubble Sort consists of a simple double **for** loop. The first iteration of the inner **for** loop moves through the record array from bottom to top, comparing adjacent keys. If the lower-indexed key’s value is greater than its higher-indexed

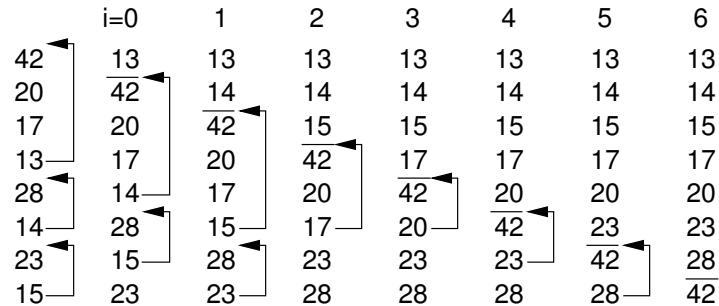


Figure 7.2 An illustration of Bubble Sort. Each column shows the array after the iteration with the indicated value of *i* in the outer **for** loop. Values above the line in each column have been sorted. Arrows indicate the swaps that take place during a given iteration.

neighbor, then the two values are swapped. Once the smallest value is encountered, this process will cause it to “bubble” up to the top of the array. The second pass through the array repeats this process. However, because we know that the smallest value reached the top of the array on the first pass, there is no need to compare the top two elements on the second pass. Likewise, each succeeding pass through the array compares adjacent elements, looking at one less value than the preceding pass. Figure 7.2 illustrates Bubble Sort. A C++ implementation is as follows:

```
template <typename E, typename Comp>
void bubsort(E A[], int n) { // Bubble Sort
    for (int i=0; i<n-1; i++) // Bubble up i'th record
        for (int j=n-1; j>i; j--)
            if (Comp::prior(A[j], A[j-1]))
                swap(A, j, j-1);
}
```

Determining Bubble Sort’s number of comparisons is easy. Regardless of the arrangement of the values in the array, the number of comparisons made by the inner **for** loop is always *i*, leading to a total cost of

$$\sum_{i=1}^n i \approx n^2/2 = \Theta(n^2).$$

Bubble Sort’s running time is roughly the same in the best, average, and worst cases.

The number of swaps required depends on how often a value is less than the one immediately preceding it in the array. We can expect this to occur for about half the comparisons in the average case, leading to $\Theta(n^2)$ for the expected number of swaps. The actual number of swaps performed by Bubble Sort will be identical to that performed by Insertion Sort.

	i=0	1	2	3	4	5	6
42	13	13	13	13	13	13	13
20	20	14	14	14	14	14	14
17	17	17	15	15	15	15	15
13	42	42	42	17	17	17	17
28	28	28	28	28	20	20	20
14	14	20	20	20	28	23	23
23	23	23	23	23	23	28	28
15	15	15	17	42	42	42	42

Figure 7.3 An example of Selection Sort. Each column shows the array after the iteration with the indicated value of i in the outer **for** loop. Numbers above the line in each column have been sorted and are in their final positions.

7.2.3 Selection Sort

Consider again the problem of sorting a pile of phone bills for the past year. Another intuitive approach might be to look through the pile until you find the bill for January, and pull that out. Then look through the remaining pile until you find the bill for February, and add that behind January. Proceed through the ever-shrinking pile of bills to select the next one in order until you are done. This is the inspiration for our last $\Theta(n^2)$ sort, called **Selection Sort**. The i th pass of Selection Sort “selects” the i th smallest key in the array, placing that record into position i . In other words, Selection Sort first finds the smallest key in an unsorted list, then the second smallest, and so on. Its unique feature is that there are few record swaps. To find the next smallest key value requires searching through the entire unsorted portion of the array, but only one swap is required to put the record in place. Thus, the total number of swaps required will be $n - 1$ (we get the last record in place “for free”).

Figure 7.3 illustrates Selection Sort. Below is a C++ implementation.

```
template <typename E, typename Comp>
void selsort(E A[], int n) { // Selection Sort
    for (int i=0; i<n-1; i++) { // Select i'th record
        int lowindex = i;      // Remember its index
        for (int j=n-1; j>i; j--) // Find the least value
            if (Comp::prior(A[j], A[lowindex]))
                lowindex = j;    // Put it in place
        swap(A, i, lowindex);
    }
}
```

Selection Sort (as written here) is essentially a Bubble Sort, except that rather than repeatedly swapping adjacent values to get the next smallest record into place, we instead remember the position of the element to be selected and do one swap at the end. Thus, the number of comparisons is still $\Theta(n^2)$, but the number of swaps is much less than that required by bubble sort. Selection Sort is particularly

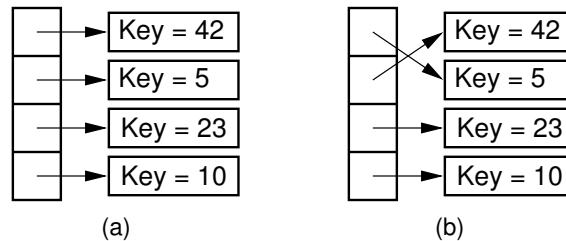


Figure 7.4 An example of swapping pointers to records. (a) A series of four records. The record with key value 42 comes before the record with key value 5. (b) The four records after the top two pointers have been swapped. Now the record with key value 5 comes before the record with key value 42.

advantageous when the cost to do a swap is high, for example, when the elements are long strings or other large records. Selection Sort is more efficient than Bubble Sort (by a constant factor) in most other situations as well.

There is another approach to keeping the cost of swapping records low that can be used by any sorting algorithm even when the records are large. This is to have each element of the array store a pointer to a record rather than store the record itself. In this implementation, a swap operation need only exchange the pointer values; the records themselves do not move. This technique is illustrated by Figure 7.4. Additional space is needed to store the pointers, but the return is a faster swap operation.

7.2.4 The Cost of Exchange Sorting

Figure 7.5 summarizes the cost of Insertion, Bubble, and Selection Sort in terms of their required number of comparisons and swaps¹ in the best, average, and worst cases. The running time for each of these sorts is $\Theta(n^2)$ in the average and worst cases.

The remaining sorting algorithms presented in this chapter are significantly better than these three under typical conditions. But before continuing on, it is instructive to investigate what makes these three sorts so slow. The crucial bottleneck is that only *adjacent* records are compared. Thus, comparisons and moves (in all but Selection Sort) are by single steps. Swapping adjacent records is called an **exchange**. Thus, these sorts are sometimes referred to as **exchange sorts**. The cost of any exchange sort can be at best the total number of steps that the records in the

¹There is a slight anomaly with Selection Sort. The supposed advantage for Selection Sort is its low number of swaps required, yet Selection Sort's best-case number of swaps is worse than that for Insertion Sort or Bubble Sort. This is because the implementation given for Selection Sort does not avoid a swap in the case where record i is already in position i . One could put in a test to avoid swapping in this situation. But it usually takes more time to do the tests than would be saved by avoiding such swaps.

	Insertion	Bubble	Selection
Comparisons:			
Best Case	$\Theta(n)$	$\Theta(n^2)$	$\Theta(n^2)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$
Swaps:			
Best Case	0	0	$\Theta(n)$
Average Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
Worst Case	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$

Figure 7.5 A comparison of the asymptotic complexities for three simple sorting algorithms.

array must move to reach their “correct” location (i.e., the number of inversions for each record).

What is the average number of inversions? Consider a list \mathbf{L} containing n values. Define \mathbf{L}_R to be \mathbf{L} in reverse. \mathbf{L} has $n(n-1)/2$ distinct pairs of values, each of which could potentially be an inversion. Each such pair must either be an inversion in \mathbf{L} or in \mathbf{L}_R . Thus, the total number of inversions in \mathbf{L} and \mathbf{L}_R together is exactly $n(n-1)/2$ for an average of $n(n-1)/4$ per list. We therefore know with certainty that any sorting algorithm which limits comparisons to adjacent items will cost at least $n(n-1)/4 = \Omega(n^2)$ in the average case.

7.3 Shellsort

The next sorting algorithm that we consider is called **Shellsort**, named after its inventor, D.L. Shell. It is also sometimes called the **diminishing increment** sort. Unlike Insertion and Selection Sort, there is no real life intuitive equivalent to Shellsort. Unlike the exchange sorts, Shellsort makes comparisons and swaps between non-adjacent elements. Shellsort also exploits the best-case performance of Insertion Sort. Shellsort’s strategy is to make the list “mostly sorted” so that a final Insertion Sort can finish the job. When properly implemented, Shellsort will give substantially better performance than $\Theta(n^2)$ in the worst case.

Shellsort uses a process that forms the basis for many of the sorts presented in the following sections: Break the list into sublists, sort them, then recombine the sublists. Shellsort breaks the array of elements into “virtual” sublists. Each sublist is sorted using an Insertion Sort. Another group of sublists is then chosen and sorted, and so on.

During each iteration, Shellsort breaks the list into disjoint sublists so that each element in a sublist is a fixed number of positions apart. For example, let us assume for convenience that n , the number of values to be sorted, is a power of two. One possible implementation of Shellsort will begin by breaking the list into $n/2$

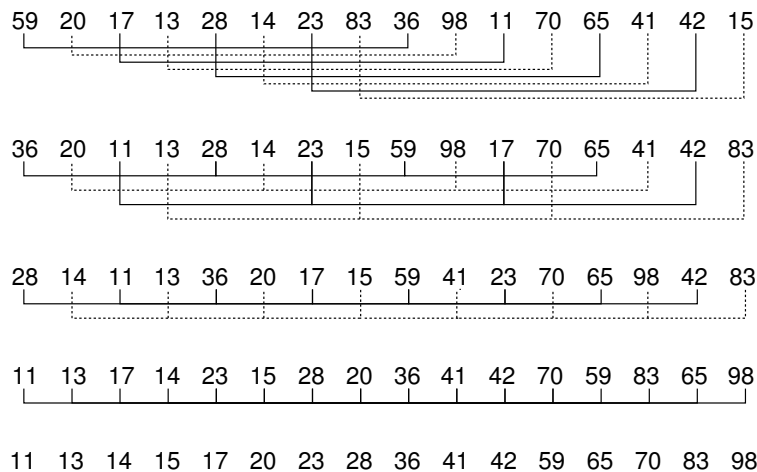


Figure 7.6 An example of Shellsort. Sixteen items are sorted in four passes. The first pass sorts 8 sublists of size 2 and increment 8. The second pass sorts 4 sublists of size 4 and increment 4. The third pass sorts 2 sublists of size 8 and increment 2. The fourth pass sorts 1 list of size 16 and increment 1 (a regular Insertion Sort).

sublists of 2 elements each, where the array index of the 2 elements in each sublist differs by $n/2$. If there are 16 elements in the array indexed from 0 to 15, there would initially be 8 sublists of 2 elements each. The first sublist would be the elements in positions 0 and 8, the second in positions 1 and 9, and so on. Each list of two elements is sorted using Insertion Sort.

The second pass of Shellsort looks at fewer, bigger lists. For our example the second pass would have $n/4$ lists of size 4, with the elements in the list being $n/4$ positions apart. Thus, the second pass would have as its first sublist the 4 elements in positions 0, 4, 8, and 12; the second sublist would have elements in positions 1, 5, 9, and 13; and so on. Each sublist of four elements would also be sorted using an Insertion Sort.

The third pass would be made on two lists, one consisting of the odd positions and the other consisting of the even positions.

The culminating pass in this example would be a “normal” Insertion Sort of all elements. Figure 7.6 illustrates the process for an array of 16 values where the sizes of the increments (the distances between elements on the successive passes) are 8, 4, 2, and 1. Figure 7.7 presents a C++ implementation for Shellsort.

Shellsort will work correctly regardless of the size of the increments, *provided that the final pass has increment 1* (i.e., provided the final pass is a regular Insertion Sort). If Shellsort will always conclude with a regular Insertion Sort, then how can it be any improvement on Insertion Sort? The expectation is that each of the (relatively cheap) sublist sorts will make the list “more sorted” than it was before.

```

// Modified version of Insertion Sort for varying increments
template <typename E, typename Comp>
void inssort2(E A[], int n, int incr) {
    for (int i=incr; i<n; i+=incr)
        for (int j=i; (j>=incr) &&
              (Comp::prior(A[j], A[j-incr]))); j-=incr)
            swap(A, j, j-incr);
}

template <typename E, typename Comp>
void shellsort(E A[], int n) { // Shellsort
    for (int i=n/2; i>2; i/=2) // For each increment
        for (int j=0; j<i; j++) // Sort each sublist
            inssort2<E,Comp>(&A[j], n-j, i);
    inssort2<E,Comp>(A, n, 1);
}

```

Figure 7.7 An implementation for Shell Sort.

It is not necessarily the case that this will be true, but it is almost always true in practice. When the final Insertion Sort is conducted, the list should be “almost sorted,” yielding a relatively cheap final Insertion Sort pass.

Some choices for increments will make Shellsort run more efficiently than others. In particular, the choice of increments described above ($2^k, 2^{k-1}, \dots, 2, 1$) turns out to be relatively inefficient. A better choice is the following series based on division by three: ($\dots, 121, 40, 13, 4, 1$).

The analysis of Shellsort is difficult, so we must accept without proof that the average-case performance of Shellsort (for “divisions by three” increments) is $O(n^{1.5})$. Other choices for the increment series can reduce this upper bound somewhat. Thus, Shellsort is substantially better than Insertion Sort, or any of the $\Theta(n^2)$ sorts presented in Section 7.2. In fact, Shellsort is not terrible when compared with the asymptotically better sorts to be presented whenever n is of medium size (thought is tends to be a little slower than these other algorithms when they are well implemented). Shellsort illustrates how we can sometimes exploit the special properties of an algorithm (in this case Insertion Sort) even if in general that algorithm is unacceptably slow.

7.4 Mergesort

A natural approach to problem solving is divide and conquer. In terms of sorting, we might consider breaking the list to be sorted into pieces, process the pieces, and then put them back together somehow. A simple way to do this would be to split the list in half, sort the halves, and then merge the sorted halves together. This is the idea behind **Mergesort**.

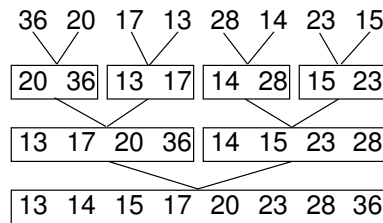


Figure 7.8 An illustration of Mergesort. The first row shows eight numbers that are to be sorted. Mergesort will recursively subdivide the list into sublists of one element each, then recombine the sublists. The second row shows the four sublists of size 2 created by the first merging pass. The third row shows the two sublists of size 4 created by the next merging pass on the sublists of row 2. The last row shows the final sorted list created by merging the two sublists of row 3.

Mergesort is one of the simplest sorting algorithms conceptually, and has good performance both in the asymptotic sense and in empirical running time. Surprisingly, even though it is based on a simple concept, it is relatively difficult to implement in practice. Figure 7.8 illustrates Mergesort. A pseudocode sketch of Mergesort is as follows:

```

List mergesort(List inlist) {
    if (inlist.length() <= 1) return inlist;;
    List L1 = half of the items from inlist;
    List L2 = other half of the items from inlist;
    return merge(mergesort(L1), mergesort(L2));
}

```

Before discussing how to implement Mergesort, we will first examine the merge function. Merging two sorted sublists is quite simple. Function **merge** examines the first element of each sublist and picks the smaller value as the smallest element overall. This smaller value is removed from its sublist and placed into the output list. Merging continues in this way, comparing the front elements of the sublists and continually appending the smaller to the output list until no more input elements remain.

Implementing Mergesort presents a number of technical difficulties. The first decision is how to represent the lists. Mergesort lends itself well to sorting a singly linked list because merging does not require random access to the list elements. Thus, Mergesort is the method of choice when the input is in the form of a linked list. Implementing **merge** for linked lists is straightforward, because we need only remove items from the front of the input lists and append items to the output list. Breaking the input list into two equal halves presents some difficulty. Ideally we would just break the lists into front and back halves. However, even if we know the length of the list in advance, it would still be necessary to traverse halfway down the linked list to reach the beginning of the second half. A simpler method, which does not rely on knowing the length of the list in advance, assigns elements of the

```

template <typename E, typename Comp>
void mergesort(E A[], E temp[], int left, int right) {
    if (left == right) return;          // List of one element
    int mid = (left+right)/2;
    mergesort<E,Comp>(A, temp, left, mid);
    mergesort<E,Comp>(A, temp, mid+1, right);
    for (int i=left; i<=right; i++)    // Copy subarray to temp
        temp[i] = A[i];
    // Do the merge operation back to A
    int i1 = left; int i2 = mid + 1;
    for (int curr=left; curr<=right; curr++) {
        if (i1 == mid+1)                // Left sublist exhausted
            A[curr] = temp[i2++];
        else if (i2 > right)            // Right sublist exhausted
            A[curr] = temp[i1++];
        else if (Comp::prior(temp[i1], temp[i2]))
            A[curr] = temp[i1++];
        else A[curr] = temp[i2++];
    }
}

```

Figure 7.9 Standard implementation for Mergesort.

input list alternating between the two sublists. The first element is assigned to the first sublist, the second element to the second sublist, the third to first sublist, the fourth to the second sublist, and so on. This requires one complete pass through the input list to build the sublists.

When the input to Mergesort is an array, splitting input into two subarrays is easy if we know the array bounds. Merging is also easy if we merge the subarrays into a second array. Note that this approach requires twice the amount of space as any of the sorting methods presented so far, which is a serious disadvantage for Mergesort. It is possible to merge the subarrays without using a second array, but this is extremely difficult to do efficiently and is not really practical. Merging the two subarrays into a second array, while simple to implement, presents another difficulty. The merge process ends with the sorted list in the auxiliary array. Consider how the recursive nature of Mergesort breaks the original array into subarrays, as shown in Figure 7.8. Mergesort is recursively called until subarrays of size 1 have been created, requiring $\log n$ levels of recursion. These subarrays are merged into subarrays of size 2, which are in turn merged into subarrays of size 4, and so on. We need to avoid having each merge operation require a new array. With some difficulty, an algorithm can be devised that alternates between two arrays. A much simpler approach is to copy the sorted sublists to the auxiliary array first, and then merge them back to the original array. Figure 7.9 shows a complete implementation for mergesort following this approach.

An optimized Mergesort implementation is shown in Figure 7.10. It reverses the order of the second subarray during the initial copy. Now the current positions of the two subarrays work inwards from the ends, allowing the end of each subarray

```

template <typename E, typename Comp>
void mergesort(E A[], E temp[], int left, int right) {
    if ((right-left) <= THRESHOLD) { // Small list
        inssort<E,Comp>(&A[left], right-left+1);
        return;
    }
    int i, j, k, mid = (left+right)/2;
    mergesort<E,Comp>(A, temp, left, mid);
    mergesort<E,Comp>(A, temp, mid+1, right);
    // Do the merge operation. First, copy 2 halves to temp.
    for (i=mid; i>=left; i--) temp[i] = A[i];
    for (j=1; j<=right-mid; j++) temp[right-j+1] = A[j+mid];
    // Merge sublists back to A
    for (i=left, j=right, k=left; k<=right; k++)
        if (Comp::prior(temp[i], temp[j])) A[k] = temp[i++];
        else A[k] = temp[j--];
}

```

Figure 7.10 Optimized implementation for Mergesort.

to act as a sentinel for the other. Unlike the previous implementation, no test is needed to check for when one of the two subarrays becomes empty. This version also uses Insertion Sort to sort small subarrays.

Analysis of Mergesort is straightforward, despite the fact that it is a recursive algorithm. The merging part takes time $\Theta(i)$ where i is the total length of the two subarrays being merged. The array to be sorted is repeatedly split in half until subarrays of size 1 are reached, at which time they are merged to be of size 2, these merged to subarrays of size 4, and so on as shown in Figure 7.8. Thus, the depth of the recursion is $\log n$ for n elements (assume for simplicity that n is a power of two). The first level of recursion can be thought of as working on one array of size n , the next level working on two arrays of size $n/2$, the next on four arrays of size $n/4$, and so on. The bottom of the recursion has n arrays of size 1. Thus, n arrays of size 1 are merged (requiring $\Theta(n)$ total steps), $n/2$ arrays of size 2 (again requiring $\Theta(n)$ total steps), $n/4$ arrays of size 4, and so on. At each of the $\log n$ levels of recursion, $\Theta(n)$ work is done, for a total cost of $\Theta(n \log n)$. This cost is unaffected by the relative order of the values being sorted, thus this analysis holds for the best, average, and worst cases.

7.5 Quicksort

While Mergesort uses the most obvious form of divide and conquer (split the list in half then sort the halves), it is not the only way that we can break down the sorting problem. And we saw that doing the merge step for Mergesort when using an array implementation is not so easy. So perhaps a different divide and conquer strategy might turn out to be more efficient?

Quicksort is aptly named because, when properly implemented, it is the fastest known general-purpose in-memory sorting algorithm in the average case. It does not require the extra array needed by Mergesort, so it is space efficient as well. Quicksort is widely used, and is typically the algorithm implemented in a library sort routine such as the UNIX **qsort** function. Interestingly, Quicksort is hampered by exceedingly poor worst-case performance, thus making it inappropriate for certain applications.

Before we get to Quicksort, consider for a moment the practicality of using a Binary Search Tree for sorting. You could insert all of the values to be sorted into the BST one by one, then traverse the completed tree using an inorder traversal. The output would form a sorted list. This approach has a number of drawbacks, including the extra space required by BST pointers and the amount of time required to insert nodes into the tree. However, this method introduces some interesting ideas. First, the root of the BST (i.e., the first node inserted) splits the list into two sublists: The left subtree contains those values in the list less than the root value while the right subtree contains those values in the list greater than or equal to the root value. Thus, the BST implicitly implements a “divide and conquer” approach to sorting the left and right subtrees. Quicksort implements this concept in a much more efficient way.

Quicksort first selects a value called the **pivot**. (This is conceptually like the root node’s value in the BST.) Assume that the input array contains k values less than the pivot. The records are then rearranged in such a way that the k values less than the pivot are placed in the first, or leftmost, k positions in the array, and the values greater than or equal to the pivot are placed in the last, or rightmost, $n - k$ positions. This is called a **partition** of the array. The values placed in a given partition need not (and typically will not) be sorted with respect to each other. All that is required is that all values end up in the correct partition. The pivot value itself is placed in position k . Quicksort then proceeds to sort the resulting subarrays now on either side of the pivot, one of size k and the other of size $n - k - 1$. How are these values sorted? Because Quicksort is such a good algorithm, using Quicksort on the subarrays would be appropriate.

Unlike some of the sorts that we have seen earlier in this chapter, Quicksort might not seem very “natural” in that it is not an approach that a person is likely to use to sort real objects. But it should not be too surprising that a really efficient sort for huge numbers of abstract objects on a computer would be rather different from our experiences with sorting a relatively few physical objects.

The C++ code for Quicksort is shown in Figure 7.11. Parameters **i** and **j** define the left and right indices, respectively, for the subarray being sorted. The initial call to Quicksort would be **qsort(array, 0, n-1)**.

Function **partition** will move records to the appropriate partition and then return **k**, the first position in the right partition. Note that the pivot value is initially

```

template <typename E, typename Comp>
void qsort(E A[], int i, int j) { // Quicksort
    if (j <= i) return; // Don't sort 0 or 1 element
    int pivotindex = findpivot(A, i, j);
    swap(A, pivotindex, j); // Put pivot at end
    // k will be the first position in the right subarray
    int k = partition<E,Comp>(A, i-1, j, A[j]);
    swap(A, k, j); // Put pivot in place
    qsort<E,Comp>(A, i, k-1);
    qsort<E,Comp>(A, k+1, j);
}

```

Figure 7.11 Implementation for Quicksort.

placed at the end of the array (position j). Thus, **partition** must not affect the value of array position j . After partitioning, the pivot value is placed in position k , which is its correct position in the final, sorted array. By doing so, we guarantee that at least one value (the pivot) will not be processed in the recursive calls to **qsort**. Even if a bad pivot is selected, yielding a completely empty partition to one side of the pivot, the larger partition will contain at most $n - 1$ elements.

Selecting a pivot can be done in many ways. The simplest is to use the first key. However, if the input is sorted or reverse sorted, this will produce a poor partitioning with all values to one side of the pivot. It is better to pick a value at random, thereby reducing the chance of a bad input order affecting the sort. Unfortunately, using a random number generator is relatively expensive, and we can do nearly as well by selecting the middle position in the array. Here is a simple **findpivot** function:

```

template <typename E>
inline int findpivot(E A[], int i, int j)
{ return (i+j)/2; }

```

We now turn to function **partition**. If we knew in advance how many keys are less than the pivot, **partition** could simply copy elements with key values less than the pivot to the low end of the array, and elements with larger keys to the high end. Because we do not know in advance how many keys are less than the pivot, we use a clever algorithm that moves indices inwards from the ends of the subarray, swapping values as necessary until the two indices meet. Figure 7.12 shows a C++ implementation for the partition step.

Figure 7.13 illustrates **partition**. Initially, variables **l** and **r** are immediately outside the actual bounds of the subarray being partitioned. Each pass through the outer **do** loop moves the counters **l** and **r** inwards, until eventually they meet. Note that at each iteration of the inner **while** loops, the bounds are moved prior to checking against the pivot value. This ensures that progress is made by each **while** loop, even when the two values swapped on the last iteration of the **do** loop were equal to the pivot. Also note the check that $r > l$ in the second **while**

```

template <typename E, typename Comp>
inline int partition(E A[], int l, int r, E& pivot) {
    do {
        // Move the bounds inward until they meet
        while (Comp::prior(A[++l], pivot)); // Move l right and
        while ((l < r) && Comp::prior(pivot, A[--r])); // r left
        swap(A, l, r); // Swap out-of-place values
    } while (l < r); // Stop when they cross
    return l; // Return first position in right partition
}

```

Figure 7.12 The Quicksort partition implementation.

Initial	72	6	57	88	85	42	83	73	48	60
										r
Pass 1	72	6	57	88	85	42	83	73	48	60
										r
Swap 1	48	6	57	88	85	42	83	73	72	60
										r
Pass 2	48	6	57	88	85	42	83	73	72	60
							r			
Swap 2	48	6	57	42	85	88	83	73	72	60
							r			
Pass 3	48	6	57	42	85	88	83	73	72	60
					,r					

Figure 7.13 The Quicksort partition step. The first row shows the initial positions for a collection of ten key values. The pivot value is 60, which has been swapped to the end of the array. The **do** loop makes three iterations, each time moving counters **l** and **r** inwards until they meet in the third pass. In the end, the left partition contains four values and the right partition contains six values. Function **qsort** will place the pivot value into position 4.

loop. This ensures that **r** does not run off the low end of the partition in the case where the pivot is the least value in that partition. Function **partition** returns the first index of the right partition so that the subarray bound for the recursive calls to **qsort** can be determined. Figure 7.14 illustrates the complete Quicksort algorithm.

To analyze Quicksort, we first analyze the **findpivot** and **partition** functions operating on a subarray of length k . Clearly, **findpivot** takes constant time. Function **partition** contains a **do** loop with two nested **while** loops. The total cost of the partition operation is constrained by how far **l** and **r** can move inwards. In particular, these two bounds variables together can move a

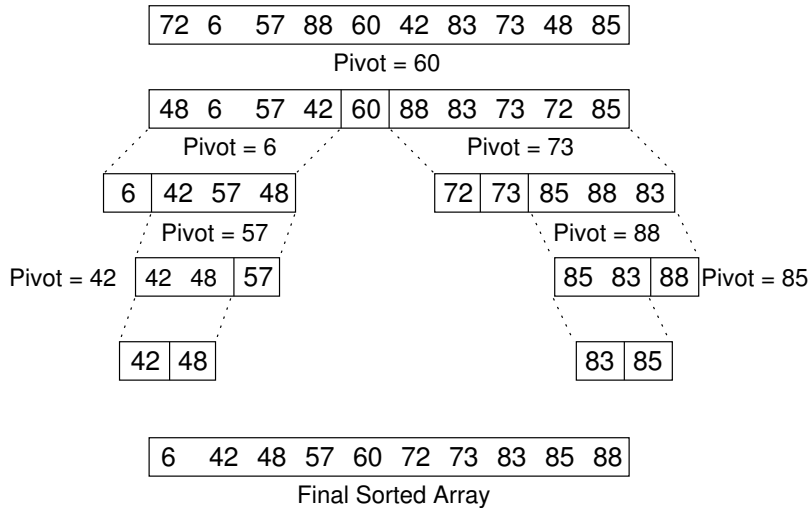


Figure 7.14 An illustration of Quicksort.

total of s steps for a subarray of length s . However, this does not directly tell us how much work is done by the nested **while** loops. The **do** loop as a whole is guaranteed to move both **l** and **r** inward at least one position on each first pass. Each **while** loop moves its variable at least once (except in the special case where **r** is at the left edge of the array, but this can happen only once). Thus, we see that the **do** loop can be executed at most s times, the total amount of work done moving **l** and **r** is s , and each **while** loop can fail its test at most s times. The total work for the entire **partition** function is therefore $\Theta(s)$.

Knowing the cost of **findpivot** and **partition**, we can determine the cost of Quicksort. We begin with a worst-case analysis. The worst case will occur when the pivot does a poor job of breaking the array, that is, when there are no elements in one partition, and $n - 1$ elements in the other. In this case, the divide and conquer strategy has done a poor job of dividing, so the conquer phase will work on a subproblem only one less than the size of the original problem. If this happens at each partition step, then the total cost of the algorithm will be

$$\sum_{k=1}^n k = \Theta(n^2).$$

In the worst case, Quicksort is $\Theta(n^2)$. This is terrible, no better than Bubble Sort.² When will this worst case occur? Only when each pivot yields a bad partitioning of the array. If the pivot values are selected at random, then this is extremely unlikely to happen. When selecting the middle position of the current subarray, it

²The worst insult that I can think of for a sorting algorithm.

is still unlikely to happen. It does not take many good partitionings for Quicksort to work fairly well.

Quicksort's best case occurs when **findpivot** always breaks the array into two equal halves. Quicksort repeatedly splits the array into smaller partitions, as shown in Figure 7.14. In the best case, the result will be $\log n$ levels of partitions, with the top level having one array of size n , the second level two arrays of size $n/2$, the next with four arrays of size $n/4$, and so on. Thus, at each level, all partition steps for that level do a total of n work, for an overall cost of $n \log n$ work when Quicksort finds perfect pivots.

Quicksort's average-case behavior falls somewhere between the extremes of worst and best case. Average-case analysis considers the cost for all possible arrangements of input, summing the costs and dividing by the number of cases. We make one reasonable simplifying assumption: At each partition step, the pivot is equally likely to end in any position in the (sorted) array. In other words, the pivot is equally likely to break an array into partitions of sizes 0 and $n - 1$, or 1 and $n - 2$, and so on.

Given this assumption, the average-case cost is computed from the following equation:

$$T(n) = cn + \frac{1}{n} \sum_{k=0}^{n-1} [T(k) + T(n-1-k)], \quad T(0) = T(1) = c.$$

This equation is in the form of a recurrence relation. Recurrence relations are discussed in Chapters 2 and 14, and this one is solved in Section 14.2.4. This equation says that there is one chance in n that the pivot breaks the array into subarrays of size 0 and $n - 1$, one chance in n that the pivot breaks the array into subarrays of size 1 and $n - 2$, and so on. The expression " $T(k) + T(n - 1 - k)$ " is the cost for the two recursive calls to Quicksort on two arrays of size k and $n - 1 - k$. The initial cn term is the cost of doing the **findpivot** and **partition** steps, for some constant c . The closed-form solution to this recurrence relation is $\Theta(n \log n)$. Thus, Quicksort has average-case cost $\Theta(n \log n)$.

This is an unusual situation that the average case cost and the worst case cost have asymptotically different growth rates. Consider what "average case" actually means. We compute an average cost for inputs of size n by summing up for every possible input of size n the product of the running time cost of that input times the probability that that input will occur. To simplify things, we assumed that every permutation is equally likely to occur. Thus, finding the average means summing up the cost for every permutation and dividing by the number of inputs ($n!$). We know that some of these $n!$ inputs cost $O(n^2)$. But the sum of all the permutation costs has to be $(n!)(O(n \log n))$. Given the extremely high cost of the worst inputs, there must be very few of them. In fact, there cannot be a constant fraction of the inputs with cost $O(n^2)$. Even, say, 1% of the inputs with cost $O(n^2)$ would lead to

an average cost of $O(n^2)$. Thus, as n grows, the fraction of inputs with high cost must be going toward a limit of zero. We can conclude that Quicksort will have good behavior if we can avoid those very few bad input permutations.

The running time for Quicksort can be improved (by a constant factor), and much study has gone into optimizing this algorithm. The most obvious place for improvement is the **findpivot** function. Quicksort's worst case arises when the pivot does a poor job of splitting the array into equal size subarrays. If we are willing to do more work searching for a better pivot, the effects of a bad pivot can be decreased or even eliminated. One good choice is to use the "median of three" algorithm, which uses as a pivot the middle of three randomly selected values. Using a random number generator to choose the positions is relatively expensive, so a common compromise is to look at the first, middle, and last positions of the current subarray. However, our simple **findpivot** function that takes the middle value as its pivot has the virtue of making it highly unlikely to get a bad input by chance, and it is quite cheap to implement. This is in sharp contrast to selecting the first or last element as the pivot, which would yield bad performance for many permutations that are nearly sorted or nearly reverse sorted.

A significant improvement can be gained by recognizing that Quicksort is relatively slow when n is small. This might not seem to be relevant if most of the time we sort large arrays, nor should it matter how long Quicksort takes in the rare instance when a small array is sorted because it will be fast anyway. But you should notice that Quicksort itself sorts many, many small arrays! This happens as a natural by-product of the divide and conquer approach.

A simple improvement might then be to replace Quicksort with a faster sort for small numbers, say Insertion Sort or Selection Sort. However, there is an even better — and still simpler — optimization. When Quicksort partitions are below a certain size, do nothing! The values within that partition will be out of order. However, we do know that all values in the array to the left of the partition are smaller than all values in the partition. All values in the array to the right of the partition are greater than all values in the partition. Thus, even if Quicksort only gets the values to "nearly" the right locations, the array will be close to sorted. This is an ideal situation in which to take advantage of the best-case performance of Insertion Sort. The final step is a single call to Insertion Sort to process the entire array, putting the elements into final sorted order. Empirical testing shows that the subarrays should be left unordered whenever they get down to nine or fewer elements.

The last speedup to be considered reduces the cost of making recursive calls. Quicksort is inherently recursive, because each Quicksort operation must sort two sublists. Thus, there is no simple way to turn Quicksort into an iterative algorithm. However, Quicksort can be implemented using a stack to imitate recursion, as the amount of information that must be stored is small. We need not store copies of a

subarray, only the subarray bounds. Furthermore, the stack depth can be kept small if care is taken on the order in which Quicksort's recursive calls are executed. We can also place the code for **findpivot** and **partition** inline to eliminate the remaining function calls. Note however that by not processing sublists of size nine or less as suggested above, about three quarters of the function calls will already have been eliminated. Thus, eliminating the remaining function calls will yield only a modest speedup.

7.6 Heapsort

Our discussion of Quicksort began by considering the practicality of using a binary search tree for sorting. The BST requires more space than the other sorting methods and will be slower than Quicksort or Mergesort due to the relative expense of inserting values into the tree. There is also the possibility that the BST might be unbalanced, leading to a $\Theta(n^2)$ worst-case running time. Subtree balance in the BST is closely related to Quicksort's partition step. Quicksort's pivot serves roughly the same purpose as the BST root value in that the left partition (subtree) stores values less than the pivot (root) value, while the right partition (subtree) stores values greater than or equal to the pivot (root).

A good sorting algorithm can be devised based on a tree structure more suited to the purpose. In particular, we would like the tree to be balanced, space efficient, and fast. The algorithm should take advantage of the fact that sorting is a special-purpose application in that all of the values to be stored are available at the start. This means that we do not necessarily need to insert one value at a time into the tree structure.

Heapsort is based on the heap data structure presented in Section 5.5. Heapsort has all of the advantages just listed. The complete binary tree is balanced, its array representation is space efficient, and we can load all values into the tree at once, taking advantage of the efficient **buildheap** function. The asymptotic performance of Heapsort is $\Theta(n \log n)$ in the best, average, and worst cases. It is not as fast as Quicksort in the average case (by a constant factor), but Heapsort has special properties that will make it particularly useful when sorting data sets too large to fit in main memory, as discussed in Chapter 8.

A sorting algorithm based on max-heaps is quite straightforward. First we use the heap building algorithm of Section 5.5 to convert the array into max-heap order. Then we repeatedly remove the maximum value from the heap, restoring the heap property each time that we do so, until the heap is empty. Note that each time we remove the maximum element from the heap, it is placed at the end of the array. Assume the n elements are stored in array positions 0 through $n - 1$. After removing the maximum value from the heap and readjusting, the maximum value will now be placed in position $n - 1$ of the array. The heap is now considered to be

of size $n - 1$. Removing the new maximum (root) value places the second largest value in position $n - 2$ of the array. After removing each of the remaining values in turn, the array will be properly sorted from least to greatest. This is why Heapsort uses a max-heap rather than a min-heap as might have been expected. Figure 7.15 illustrates Heapsort. The complete C++ implementation is as follows:

```
template <typename E, typename Comp>
void heapsort(E A[], int n) { // Heapsort
    E maxval;
    heap<E, Comp> H(A, n, n);    // Build the heap
    for (int i=0; i<n; i++)      // Now sort
        maxval = H.removefirst(); // Place maxval at end
}
```

Because building the heap takes $\Theta(n)$ time (see Section 5.5), and because n deletions of the maximum element each take $\Theta(\log n)$ time, we see that the entire Heapsort operation takes $\Theta(n \log n)$ time in the worst, average, and best cases. While typically slower than Quicksort by a constant factor, Heapsort has one special advantage over the other sorts studied so far. Building the heap is relatively cheap, requiring $\Theta(n)$ time. Removing the maximum element from the heap requires $\Theta(\log n)$ time. Thus, if we wish to find the k largest elements in an array, we can do so in time $\Theta(n + k \log n)$. If k is small, this is a substantial improvement over the time required to find the k largest elements using one of the other sorting methods described earlier (many of which would require sorting all of the array first). One situation where we are able to take advantage of this concept is in the implementation of Kruskal's minimum-cost spanning tree (MST) algorithm of Section 11.5.2. That algorithm requires that edges be visited in ascending order (so, use a min-heap), but this process stops as soon as the MST is complete. Thus, only a relatively small fraction of the edges need be sorted.

7.7 Binsort and Radix Sort

Imagine that for the past year, as you paid your various bills, you then simply piled all the paperwork onto the top of a table somewhere. Now the year has ended and its time to sort all of these papers by what the bill was for (phone, electricity, rent, etc.) and date. A pretty natural approach is to make some space on the floor, and as you go through the pile of papers, put the phone bills into one pile, the electric bills into another pile, and so on. Once this initial assignment of bills to piles is done (in one pass), you can sort each pile by date relatively quickly because they are each fairly small. This is the basic idea behind a Binsort.

Section 3.9 presented the following code fragment to sort a permutation of the numbers 0 through $n - 1$:

```
for (i=0; i<n; i++)
    B[A[i]] = A[i];
```

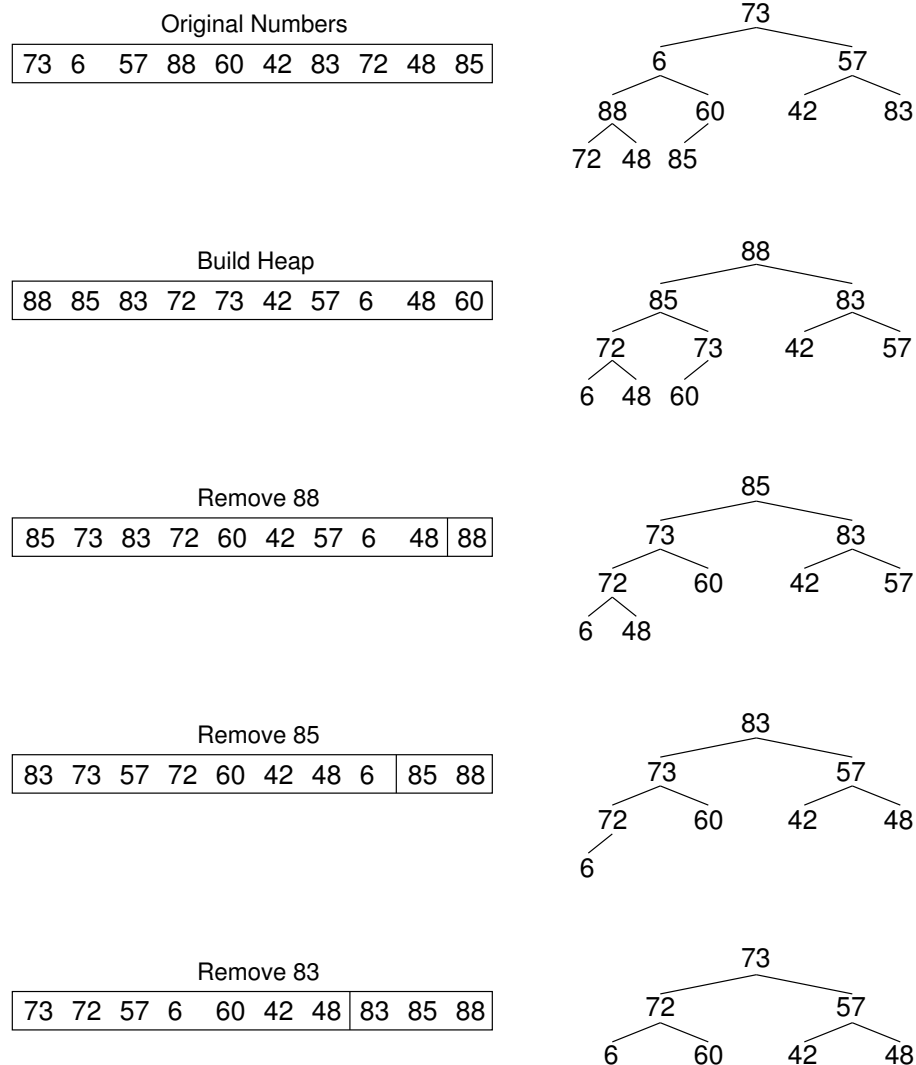



Figure 7.15 An illustration of Heapsort. The top row shows the values in their original order. The second row shows the values after building the heap. The third row shows the result of the first **removefirst** operation on key value 88. Note that 88 is now at the end of the array. The fourth row shows the result of the second **removefirst** operation on key value 85. The fifth row shows the result of the third **removefirst** operation on key value 83. At this point, the last three positions of the array hold the three greatest values in sorted order. Heapsort continues in this manner until the entire array is sorted.

```

template <typename E, class getKey>
void binsort(E A[], int n) {
    List<E> B[MaxKeyValue];
    E item;
    for (int i=0; i<n; i++) B[A[i]].append(getKey::key(A[i]));
    for (int i=0; i<MaxKeyValue; i++)
        for (B[i].setStart(); B[i].getValue(item); B[i].next())
            output(item);
}

```

Figure 7.16 The extended Binsort algorithm.

Here the key value is used to determine the position for a record in the final sorted array. This is the most basic example of a **Binsort**, where key values are used to assign records to **bins**. This algorithm is extremely efficient, taking $\Theta(n)$ time regardless of the initial ordering of the keys. This is far better than the performance of any sorting algorithm that we have seen so far. The only problem is that this algorithm has limited use because it works only for a permutation of the numbers from 0 to $n - 1$.

We can extend this simple Binsort algorithm to be more useful. Because Binsort must perform direct computation on the key value (as opposed to just asking which of two records comes first as our previous sorting algorithms did), we will assume that the records use an integer key type. We further assume that it can be extracted from a record using the **key** method supplied by a template parameter class named **getKey**.

The simplest extension is to allow for duplicate values among the keys. This can be done by turning array slots into arbitrary-length bins by turning **B** into an array of linked lists. In this way, all records with key value i can be placed in bin **B[i]**. A second extension allows for a key range greater than n . For example, a set of n records might have keys in the range 1 to $2n$. The only requirement is that each possible key value have a corresponding bin in **B**. The extended Binsort algorithm is shown in Figure 7.16.

This version of Binsort can sort any collection of records whose key values fall in the range from 0 to **MaxKeyValue**−1. The total work required is simply that needed to place each record into the appropriate bin and then take all of the records out of the bins. Thus, we need to process each record twice, for $\Theta(n)$ work.

Unfortunately, there is a crucial oversight in this analysis. Binsort must also look at each of the bins to see if it contains a record. The algorithm must process **MaxKeyValue** bins, regardless of how many actually hold records. If **MaxKeyValue** is small compared to n , then this is not a great expense. Suppose that **MaxKeyValue** = n^2 . In this case, the total amount of work done will be $\Theta(n + n^2) = \Theta(n^2)$. This results in a poor sorting algorithm, and the algorithm becomes even worse as the disparity between n and **MaxKeyValue** increases. In addition,

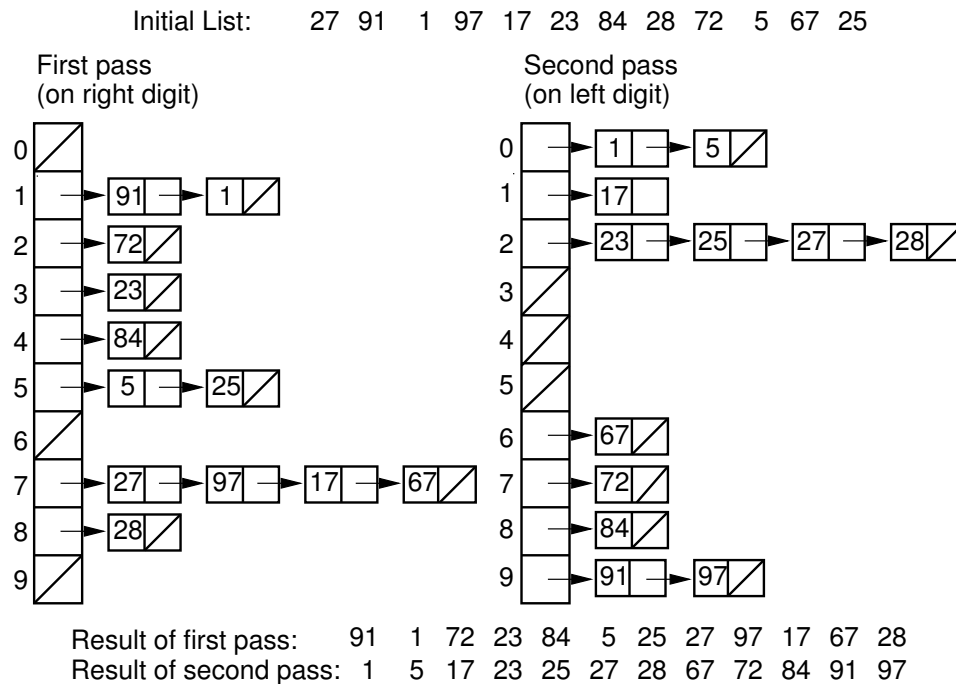


Figure 7.17 An example of Radix Sort for twelve two-digit numbers in base ten. Two passes are required to sort the list.

a large key range requires an unacceptably large array **B**. Thus, even the extended Binsort is useful only for a limited key range.

A further generalization to Binsort yields a **bucket sort**. Each bin is associated with not just one key, but rather a range of key values. A bucket sort assigns records to bins and then relies on some other sorting technique to sort the records within each bin. The hope is that the relatively inexpensive bucketing process will put only a small number of records in each bin, and that a “cleanup sort” within the bins will then be relatively cheap.

There is a way to keep the number of bins and the related processing small while allowing the cleanup sort to be based on Binsort. Consider a sequence of records with keys in the range 0 to 99. If we have ten bins available, we can first assign records to bins by taking their key value modulo 10. Thus, every key will be assigned to the bin matching its rightmost decimal digit. We can then take these records from the bins *in order* and reassign them to the bins on the basis of their leftmost (10’s place) digit (define values in the range 0 to 9 to have a leftmost digit of 0). In other words, assign the i th record from array **A** to a bin using the formula $\mathbf{A}[i] / 10$. If we now gather the values from the bins in order, the result is a sorted list. Figure 7.17 illustrates this process.

```

template <typename E, typename getKey>
void radix(E A[], E B[],
           int n, int k, int r, int cnt[]) {
    // cnt[i] stores number of records in bin[i]
    int j;

    for (int i=0, rtoi=1; i<k; i++, rtoi*=r) { // For k digits
        for (j=0; j<r; j++) cnt[j] = 0;        // Initialize cnt

        // Count the number of records for each bin on this pass
        for (j=0; j<n; j++) cnt[(getKey::key(A[j])/rtoi)%r]++;

        // Index B: cnt[j] will be index for last slot of bin j.
        for (j=1; j<r; j++) cnt[j] = cnt[j-1] + cnt[j];

        // Put records into bins, work from bottom of each bin.
        // Since bins fill from bottom, j counts downwards
        for (j=n-1; j>=0; j--)
            B[--cnt[(getKey::key(A[j])/rtoi)%r]] = A[j];

        for (j=0; j<n; j++) A[j] = B[j];    // Copy B back to A
    }
}

```

Figure 7.18 The Radix Sort algorithm.

In this example, we have $r = 10$ bins and $n = 12$ keys in the range 0 to $r^2 - 1$. The total computation is $\Theta(n)$, because we look at each record and each bin a constant number of times. This is a great improvement over the simple Binsort where the number of bins must be as large as the key range. Note that the example uses $r = 10$ so as to make the bin computations easy to visualize: Records were placed into bins based on the value of first the rightmost and then the leftmost decimal digits. Any number of bins would have worked. This is an example of a **Radix Sort**, so called because the bin computations are based on the **radix** or the **base** of the key values. This sorting algorithm can be extended to any number of keys in any key range. We simply assign records to bins based on the keys' digit values working from the rightmost digit to the leftmost. If there are k digits, then this requires that we assign keys to bins k times.

As with Mergesort, an efficient implementation of Radix Sort is somewhat difficult to achieve. In particular, we would prefer to sort an array of values and avoid processing linked lists. If we know how many values will be in each bin, then an auxiliary array of size r can be used to hold the bins. For example, if during the first pass the 0 bin will receive three records and the 1 bin will receive five records, then we could simply reserve the first three array positions for the 0 bin and the next five array positions for the 1 bin. Exactly this approach is taken by the C++ implementation of Figure 7.18. At the end of each pass, the records are copied back to the original array.

The first inner **for** loop initializes array **cnt**. The second loop counts the number of records to be assigned to each bin. The third loop sets the values in **cnt** to their proper indices within array **B**. Note that the index stored in **cnt[j]** is the *last* index for bin **j**; bins are filled from high index to low index. The fourth loop assigns the records to the bins (within array **B**). The final loop simply copies the records back to array **A** to be ready for the next pass. Variable **rtoi** stores r^i for use in bin computation on the i 'th iteration. Figure 7.19 shows how this algorithm processes the input shown in Figure 7.17.

This algorithm requires k passes over the list of n numbers in base r , with $\Theta(n + r)$ work done at each pass. Thus the total work is $\Theta(nk + rk)$. What is this in terms of n ? Because r is the size of the base, it might be rather small. One could use base 2 or 10. Base 26 would be appropriate for sorting character strings. For now, we will treat r as a constant value and ignore it for the purpose of determining asymptotic complexity. Variable k is related to the key range: It is the maximum number of digits that a key may have in base r . In some applications we can determine k to be of limited size and so might wish to consider it a constant. In this case, Radix Sort is $\Theta(n)$ in the best, average, and worst cases, making it the sort with best asymptotic complexity that we have studied.

Is it a reasonable assumption to treat k as a constant? Or is there some relationship between k and n ? If the key range is limited and duplicate key values are common, there might be no relationship between k and n . To make this distinction clear, use N to denote the number of distinct key values used by the n records. Thus, $N \leq n$. Because it takes a minimum of $\log_r N$ base r digits to represent N distinct key values, we know that $k \geq \log_r N$.

Now, consider the situation in which no keys are duplicated. If there are n unique keys ($n = N$), then it requires n distinct code values to represent them. Thus, $k \geq \log_r n$. Because it requires *at least* $\Omega(\log n)$ digits (within a constant factor) to distinguish between the n distinct keys, k is in $\Omega(\log n)$. This yields an asymptotic complexity of $\Omega(n \log n)$ for Radix Sort to process n distinct key values.

It is possible that the key range is much larger; $\log_r n$ bits is merely the best case possible for n distinct values. Thus, the $\log_r n$ estimate for k could be overly optimistic. The moral of this analysis is that, for the general case of n distinct key values, Radix Sort is at best a $\Omega(n \log n)$ sorting algorithm.

Radix Sort can be much improved by making base r be as large as possible. Consider the case of an integer key value. Set $r = 2^i$ for some i . In other words, the value of r is related to the number of bits of the key processed on each pass. Each time the number of bits is doubled, the number of passes is cut in half. When processing an integer key value, setting $r = 256$ allows the key to be processed one byte at a time. Processing a 32-bit key requires only four passes. It is not unreasonable on most computers to use $r = 2^{16} = 64K$, resulting in only two passes for

Initial Input: Array A

27	91	1	97	17	23	84	28	72	5	67	25
----	----	---	----	----	----	----	----	----	---	----	----

First pass values for Count.
rtol = 1.

0	1	2	3	4	5	6	7	8	9
0	2	1	1	1	2	0	4	1	0

Count array:
Index positions for Array B.

0	1	2	3	4	5	6	7	8	9
0	2	3	4	5	7	7	11	12	12

91	1	72	23	84	5	25	27	97	17	67	28
0	1	2	3	4	5	6	7	8	9	10	11

End of Pass 1: Array A.

Second pass values for Count.
rtol = 10.

0	1	2	3	4	5	6	7	8	9
2	1	4	0	0	0	1	1	1	2

Count array:
Index positions for Array B.

0	1	2	3	4	5	6	7	8	9
2	3	7	7	7	7	8	9	10	12

1	5	17	23	25	27	28	67	72	84	91	97
0	1	2	3	4	5	6	7	8	9	10	11

End of Pass 2: Array A.

Figure 7.19 An example showing function **radix** applied to the input of Figure 7.17. Row 1 shows the initial values within the input array. Row 2 shows the values for array **cnt** after counting the number of records for each bin. Row 3 shows the index values stored in array **cnt**. For example, **cnt[0]** is 0, indicating no input values are in bin 0. **Cnt[1]** is 2, indicating that array **B** positions 0 and 1 will hold the values for bin 1. **Cnt[2]** is 3, indicating that array **B** position 2 will hold the (single) value for bin 2. **Cnt[7]** is 11, indicating that array **B** positions 7 through 10 will hold the four values for bin 7. Row 4 shows the results of the first pass of the Radix Sort. Rows 5 through 7 show the equivalent steps for the second pass.

a 32-bit key. Of course, this requires a `cnt` array of size 64K. Performance will be good only if the number of records is close to 64K or greater. In other words, the number of records must be large compared to the key size for Radix Sort to be efficient. In many sorting applications, Radix Sort can be tuned in this way to give good performance.

Radix Sort depends on the ability to make a fixed number of multiway choices based on a digit value, as well as random access to the bins. Thus, Radix Sort might be difficult to implement for certain key types. For example, if the keys are real numbers or arbitrary length strings, then some care will be necessary in implementation. In particular, Radix Sort will need to be careful about deciding when the “last digit” has been found to distinguish among real numbers, or the last character in variable length strings. Implementing the concept of Radix Sort with the trie data structure (Section 13.1) is most appropriate for these situations.

At this point, the perceptive reader might begin to question our earlier assumption that key comparison takes constant time. If the keys are “normal integer” values stored in, say, an integer variable, what is the size of this variable compared to n ? In fact, it is almost certain that 32 (the number of bits in a standard `int` variable) is greater than $\log n$ for any practical computation. In this sense, comparison of two long integers requires $\Omega(\log n)$ work.

Computers normally do arithmetic in units of a particular size, such as a 32-bit word. Regardless of the size of the variables, comparisons use this native word size and require a constant amount of time since the comparison is implemented in hardware. In practice, comparisons of two 32-bit values take constant time, even though 32 is much greater than $\log n$. To some extent the truth of the proposition that there are constant time operations (such as integer comparison) is in the eye of the beholder. At the gate level of computer architecture, individual bits are compared. However, constant time comparison for integers is true in practice on most computers (they require a fixed number of machine instructions), and we rely on such assumptions as the basis for our analyses. In contrast, Radix Sort must do several arithmetic calculations on key values (each requiring constant time), where the number of such calculations is proportional to the key length. Thus, Radix Sort truly does $\Omega(n \log n)$ work to process n distinct key values.

7.8 An Empirical Comparison of Sorting Algorithms

Which sorting algorithm is fastest? Asymptotic complexity analysis lets us distinguish between $\Theta(n^2)$ and $\Theta(n \log n)$ algorithms, but it does not help distinguish between algorithms with the same asymptotic complexity. Nor does asymptotic analysis say anything about which algorithm is best for sorting small lists. For answers to these questions, we can turn to empirical testing.

Sort	10	100	1K	10K	100K	1M	Up	Down
Insertion	.00023	.007	0.66	64.98	7381.0	674420	0.04	129.05
Bubble	.00035	.020	2.25	277.94	27691.0	2820680	70.64	108.69
Selection	.00039	.012	0.69	72.47	7356.0	780000	69.76	69.58
Shell	.00034	.008	0.14	1.99	30.2	554	0.44	0.79
Shell/O	.00034	.008	0.12	1.91	29.0	530	0.36	0.64
Merge	.00050	.010	0.12	1.61	19.3	219	0.83	0.79
Merge/O	.00024	.007	0.10	1.31	17.2	197	0.47	0.66
Quick	.00048	.008	0.11	1.37	15.7	162	0.37	0.40
Quick/O	.00031	.006	0.09	1.14	13.6	143	0.32	0.36
Heap	.00050	.011	0.16	2.08	26.7	391	1.57	1.56
Heap/O	.00033	.007	0.11	1.61	20.8	334	1.01	1.04
Radix/4	.00838	.081	0.79	7.99	79.9	808	7.97	7.97
Radix/8	.00799	.044	0.40	3.99	40.0	404	4.00	3.99

Figure 7.20 Empirical comparison of sorting algorithms run on a 3.4-GHz Intel Pentium 4 CPU running Linux. Shellsort, Quicksort, Mergesort, and Heapsort each are shown with regular and optimized versions. Radix Sort is shown for 4- and 8-bit-per-pass versions. All times shown are milliseconds.

Figure 7.20 shows timing results for actual implementations of the sorting algorithms presented in this chapter. The algorithms compared include Insertion Sort, Bubble Sort, Selection Sort, Shellsort, Quicksort, Mergesort, Heapsort and Radix Sort. Shellsort shows both the basic version from Section 7.3 and another with increments based on division by three. Mergesort shows both the basic implementation from Section 7.4 and the optimized version (including calls to Insertion Sort for lists of length below nine). For Quicksort, two versions are compared: the basic implementation from Section 7.5 and an optimized version that does not partition sublists below length nine (with Insertion Sort performed at the end). The first Heapsort version uses the class definitions from Section 5.5. The second version removes all the class definitions and operates directly on the array using inlined code for all access functions.

Except for the rightmost columns, the input to each algorithm is a random array of integers. This affects the timing for some of the sorting algorithms. For example, Selection Sort is not being used to best advantage because the record size is small, so it does not get the best possible showing. The Radix Sort implementation certainly takes advantage of this key range in that it does not look at more digits than necessary. On the other hand, it was not optimized to use bit shifting instead of division, even though the bases used would permit this.

The various sorting algorithms are shown for lists of sizes 10, 100, 1000, 10,000, 100,000, and 1,000,000. The final two columns of each table show the performance for the algorithms on inputs of size 10,000 where the numbers are in ascending (sorted) and descending (reverse sorted) order, respectively. These columns demonstrate best-case performance for some algorithms and worst-case

performance for others. They also show that for some algorithms, the order of input has little effect.

These figures show a number of interesting results. As expected, the $O(n^2)$ sorts are quite poor performers for large arrays. Insertion Sort is by far the best of this group, unless the array is already reverse sorted. Shellsort is clearly superior to any of these $O(n^2)$ sorts for lists of even 100 elements. Optimized Quicksort is clearly the best overall algorithm for all but lists of 10 elements. Even for small arrays, optimized Quicksort performs well because it does one partition step before calling Insertion Sort. Compared to the other $O(n \log n)$ sorts, unoptimized Heapsort is quite slow due to the overhead of the class structure. When all of this is stripped away and the algorithm is implemented to manipulate an array directly, it is still somewhat slower than mergesort. In general, optimizing the various algorithms makes a noticeable improvement for larger array sizes.

Overall, Radix Sort is a surprisingly poor performer. If the code had been tuned to use bit shifting of the key value, it would likely improve substantially; but this would seriously limit the range of element types that the sort could support.

7.9 Lower Bounds for Sorting

This book contains many analyses for algorithms. These analyses generally define the upper and lower bounds for algorithms in their worst and average cases. For many of the algorithms presented so far, analysis has been easy. This section considers a more difficult task — an analysis for the cost of a *problem* as opposed to an *algorithm*. The upper bound for a problem can be defined as the asymptotic cost of the fastest known algorithm. The lower bound defines the best possible efficiency for *any* algorithm that solves the problem, including algorithms not yet invented. Once the upper and lower bounds for the problem meet, we know that no future algorithm can possibly be (asymptotically) more efficient.

A simple estimate for a problem's lower bound can be obtained by measuring the size of the input that must be read and the output that must be written. Certainly no algorithm can be more efficient than the problem's I/O time. From this we see that the sorting problem cannot be solved by *any* algorithm in less than $\Omega(n)$ time because it takes at least n steps to read and write the n values to be sorted. Alternatively, any sorting algorithm must at least look at every input value to recognize whether the input values are in sort order. So, based on our current knowledge of sorting algorithms and the size of the input, we know that the *problem* of sorting is bounded by $\Omega(n)$ and $O(n \log n)$.

Computer scientists have spent much time devising efficient general-purpose sorting algorithms, but no one has ever found one that is faster than $O(n \log n)$ in the worst or average cases. Should we keep searching for a faster sorting algorithm?

Or can we prove that there is no faster sorting algorithm by finding a tighter lower bound?

This section presents one of the most important and most useful proofs in computer science: No sorting algorithm based on key comparisons can possibly be faster than $\Omega(n \log n)$ in the worst case. This proof is important for three reasons. First, knowing that widely used sorting algorithms are asymptotically optimal is reassuring. In particular, it means that you need not bang your head against the wall searching for an $O(n)$ sorting algorithm (or at least not one in any way based on key comparisons). Second, this proof is one of the few non-trivial lower-bounds proofs that we have for any problem; that is, this proof provides one of the relatively few instances where our lower bound is tighter than simply measuring the size of the input and output. As such, it provides a useful model for proving lower bounds on other problems. Finally, knowing a lower bound for sorting gives us a lower bound in turn for other problems whose solution could be used as the basis for a sorting algorithm. The process of deriving asymptotic bounds for one problem from the asymptotic bounds of another is called a **reduction**, a concept further explored in Chapter 17.

Except for the Radix Sort and Binsort, all of the sorting algorithms presented in this chapter make decisions based on the direct comparison of two key values. For example, Insertion Sort sequentially compares the value to be inserted into the sorted list until a comparison against the next value in the list fails. In contrast, Radix Sort has no direct comparison of key values. All decisions are based on the value of specific digits in the key value, so it is possible to take approaches to sorting that do not involve key comparisons. Of course, Radix Sort in the end does not provide a more efficient sorting algorithm than comparison-based sorting. Thus, empirical evidence suggests that comparison-based sorting is a good approach.³

The proof that any comparison sort requires $\Omega(n \log n)$ comparisons in the worst case is structured as follows. First, comparison-based decisions can be modeled as the branches in a tree. This means that any sorting algorithm based on comparisons between records can be viewed as a binary tree whose nodes correspond to the comparisons, and whose branches correspond to the possible outcomes. Next, the minimum number of leaves in the resulting tree is shown to be the factorial of n . Finally, the minimum depth of a tree with $n!$ leaves is shown to be in $\Omega(n \log n)$.

Before presenting the proof of an $\Omega(n \log n)$ lower bound for sorting, we first must define the concept of a **decision tree**. A decision tree is a binary tree that can model the processing for any algorithm that makes binary decisions. Each (binary) decision is represented by a branch in the tree. For the purpose of modeling sorting algorithms, we count all comparisons of key values as decisions. If two keys are

³The truth is stronger than this statement implies. In reality, Radix Sort relies on comparisons as well and so can be modeled by the technique used in this section. The result is an $\Omega(n \log n)$ bound in the general case even for algorithms that look like Radix Sort.

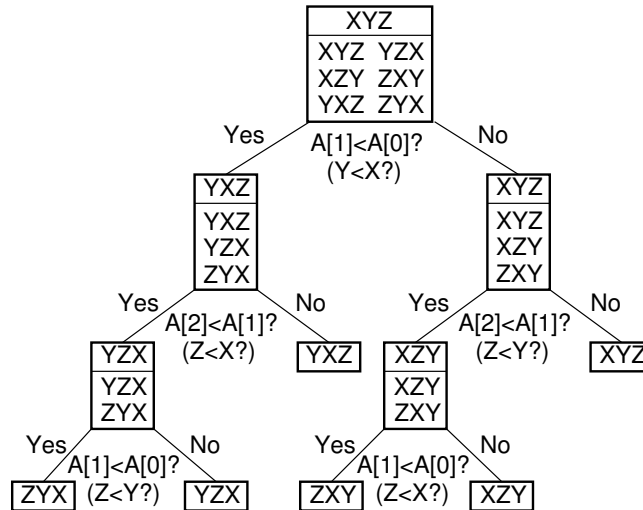


Figure 7.21 Decision tree for Insertion Sort when processing three values labeled X, Y, and Z, initially stored at positions 0, 1, and 2, respectively, in input array A.

compared and the first is less than the second, then this is modeled as a left branch in the decision tree. In the case where the first value is greater than the second, the algorithm takes the right branch.

Figure 7.21 shows the decision tree that models Insertion Sort on three input values. The first input value is labeled X, the second Y, and the third Z. They are initially stored in positions 0, 1, and 2, respectively, of input array **A**. Consider the possible outputs. Initially, we know nothing about the final positions of the three values in the sorted output array. The correct output could be any permutation of the input values. For three values, there are $n! = 6$ permutations. Thus, the root node of the decision tree lists all six permutations that might be the eventual result of the algorithm.

When $n = 3$, the first comparison made by Insertion Sort is between the second item in the input array (Y) and the first item in the array (X). There are two possibilities: Either the value of Y is less than that of X, or the value of Y is *not* less than that of X. This decision is modeled by the first branch in the tree. If Y is less than X, then the left branch should be taken and Y must appear before X in the final output. Only three of the original six permutations have this property, so the left child of the root lists the three permutations where Y appears before X: YXZ, YZX, and ZYX. Likewise, if Y were not less than X, then the right branch would be taken, and only the three permutations in which Y appears after X are possible outcomes: XYZ, XZY, and ZXY. These are listed in the right child of the root.

Let us assume for the moment that Y is less than X and so the left branch is taken. In this case, Insertion Sort swaps the two values. At this point the array

stores YXZ. Thus, in Figure 7.21 the left child of the root shows YXZ above the line. Next, the third value in the array is compared against the second (i.e., Z is compared with X). Again, there are two possibilities. If Z is less than X, then these items should be swapped (the left branch). If Z is not less than X, then Insertion Sort is complete (the right branch).

Note that the right branch reaches a leaf node, and that this leaf node contains only one permutation: YXZ. This means that only permutation YXZ can be the outcome based on the results of the decisions taken to reach this node. In other words, Insertion Sort has “found” the single permutation of the original input that yields a sorted list. Likewise, if the second decision resulted in taking the left branch, a third comparison, regardless of the outcome, yields nodes in the decision tree with only single permutations. Again, Insertion Sort has “found” the correct permutation that yields a sorted list.

Any sorting algorithm based on comparisons can be modeled by a decision tree in this way, regardless of the size of the input. Thus, all sorting algorithms can be viewed as algorithms to “find” the correct permutation of the input that yields a sorted list. Each algorithm based on comparisons can be viewed as proceeding by making branches in the tree based on the results of key comparisons, and each algorithm can terminate once a node with a single permutation has been reached.

How is the worst-case cost of an algorithm expressed by the decision tree? The decision tree shows the decisions made by an algorithm for all possible inputs of a given size. Each path through the tree from the root to a leaf is one possible series of decisions taken by the algorithm. The depth of the deepest node represents the longest series of decisions required by the algorithm to reach an answer.

There are many comparison-based sorting algorithms, and each will be modeled by a different decision tree. Some decision trees might be well-balanced, others might be unbalanced. Some trees will have more nodes than others (those with more nodes might be making “unnecessary” comparisons). In fact, a poor sorting algorithm might have an arbitrarily large number of nodes in its decision tree, with leaves of arbitrary depth. There is no limit to how slow the “worst” possible sorting algorithm could be. However, we are interested here in knowing what the *best* sorting algorithm could have as its minimum cost in the worst case. In other words, we would like to know what is the *smallest* depth possible for the *deepest* node in the tree for any sorting algorithm.

The smallest depth of the deepest node will depend on the number of nodes in the tree. Clearly we would like to “push up” the nodes in the tree, but there is limited room at the top. A tree of height 1 can only store one node (the root); the tree of height 2 can store three nodes; the tree of height 3 can store seven nodes, and so on.

Here are some important facts worth remembering:

- A binary tree of height n can store at most $2^n - 1$ nodes.

- Equivalently, a tree with n nodes requires at least $\lceil \log(n+1) \rceil$ levels.

What is the minimum number of nodes that must be in the decision tree for any comparison-based sorting algorithm for n values? Because sorting algorithms are in the business of determining which unique permutation of the input corresponds to the sorted list, the decision tree for any sorting algorithm must contain at least one leaf node for each possible permutation. There are $n!$ permutations for a set of n numbers (see Section 2.2).

Because there are at least $n!$ nodes in the tree, we know that the tree must have $\Omega(\log n!)$ levels. From Stirling's approximation (Section 2.2), we know $\log n!$ is in $\Omega(n \log n)$. The decision tree for any comparison-based sorting algorithm must have nodes $\Omega(n \log n)$ levels deep. Thus, in the worst case, any such sorting algorithm must require $\Omega(n \log n)$ comparisons.

Any sorting algorithm requiring $\Omega(n \log n)$ comparisons in the worst case requires $\Omega(n \log n)$ running time in the worst case. Because any sorting algorithm requires $\Omega(n \log n)$ running time, the problem of sorting also requires $\Omega(n \log n)$ time. We already know of sorting algorithms with $O(n \log n)$ running time, so we can conclude that the problem of sorting requires $\Theta(n \log n)$ time. As a corollary, we know that no comparison-based sorting algorithm can improve on existing $\Theta(n \log n)$ time sorting algorithms by more than a constant factor.

7.10 Further Reading

The definitive reference on sorting is Donald E. Knuth's *Sorting and Searching* [Knu98]. A wealth of details is covered there, including optimal sorts for small size n and special purpose sorting networks. It is a thorough (although somewhat dated) treatment on sorting. For an analysis of Quicksort and a thorough survey on its optimizations, see Robert Sedgewick's *Quicksort* [Sed80]. Sedgewick's *Algorithms* [Sed11] discusses most of the sorting algorithms described here and pays special attention to efficient implementation. The optimized Mergesort version of Section 7.4 comes from Sedgewick.

While $\Omega(n \log n)$ is the theoretical lower bound in the worst case for sorting, many times the input is sufficiently well ordered that certain algorithms can take advantage of this fact to speed the sorting process. A simple example is Insertion Sort's best-case running time. Sorting algorithms whose running time is based on the amount of disorder in the input are called **adaptive**. For more information on adaptive sorting algorithms, see "A Survey of Adaptive Sorting Algorithms" by Estivill-Castro and Wood [ECW92].

7.11 Exercises

7.1 Using induction, prove that Insertion Sort will always produce a sorted array.

7.2 Write an Insertion Sort algorithm for integer key values. However, here's the catch: The input is a stack (*not* an array), and the only variables that your algorithm may use are a fixed number of integers and a fixed number of stacks. The algorithm should return a stack containing the records in sorted order (with the least value being at the top of the stack). Your algorithm should be $\Theta(n^2)$ in the worst case.

7.3 The Bubble Sort implementation has the following inner **for** loop:

```
for (int j=n-1; j>i; j--)
```

Consider the effect of replacing this with the following statement:

```
for (int j=n-1; j>0; j--)
```

Would the new implementation work correctly? Would the change affect the asymptotic complexity of the algorithm? How would the change affect the running time of the algorithm?

7.4 When implementing Insertion Sort, a binary search could be used to locate the position within the first $i - 1$ elements of the array into which element i should be inserted. How would this affect the number of comparisons required? How would using such a binary search affect the asymptotic running time for Insertion Sort?

7.5 Figure 7.5 shows the best-case number of swaps for Selection Sort as $\Theta(n)$. This is because the algorithm does not check to see if the i th record is already in the i th position; that is, it might perform unnecessary swaps.

- (a) Modify the algorithm so that it does not make unnecessary swaps.
- (b) What is your prediction regarding whether this modification actually improves the running time?
- (c) Write two programs to compare the actual running times of the original Selection Sort and the modified algorithm. Which one is actually faster?

7.6 Recall that a sorting algorithm is said to be stable if the original ordering for duplicate keys is preserved. Of the sorting algorithms Insertion Sort, Bubble Sort, Selection Sort, Shellsort, Mergesort, Quicksort, Heapsort, Binsort, and Radix Sort, which of these are stable, and which are not? For each one, describe either why it is or is not stable. If a minor change to the implementation would make it stable, describe the change.

7.7 Recall that a sorting algorithm is said to be stable if the original ordering for duplicate keys is preserved. We can make any algorithm stable if we alter the input keys so that (potentially) duplicate key values are made unique in a way that the first occurrence of the original duplicate value is less than the second occurrence, which in turn is less than the third, and so on. In the worst case, it is possible that all n input records have the same key value. Give an

algorithm to modify the key values such that every modified key value is unique, the resulting key values give the same sort order as the original keys, the result is stable (in that the duplicate original key values remain in their original order), and the process of altering the keys is done in linear time using only a constant amount of additional space.

7.8 The discussion of Quicksort in Section 7.5 described using a stack instead of recursion to reduce the number of function calls made.

- (a) How deep can the stack get in the worst case?
- (b) Quicksort makes two recursive calls. The algorithm could be changed to make these two calls in a specific order. In what order should the two calls be made, and how does this affect how deep the stack can become?

7.9 Give a permutation for the values 0 through 7 that will cause Quicksort (as implemented in Section 7.5) to have its worst case behavior.

7.10 Assume **L** is an array, **length(L)** returns the number of records in the array, and **qsort(L, i, j)** sorts the records of **L** from **i** to **j** (leaving the records sorted in **L**) using the Quicksort algorithm. What is the average-case time complexity for each of the following code fragments?

(a) `for (i=0; i<length(L); i++)
 qsort(L, 0, i);`

(b) `for (i=0; i<length(L); i++)
 qsort(L, 0, length(L)-1);`

7.11 Modify Quicksort to find the smallest k values in an array of records. Your output should be the array modified so that the k smallest values are sorted in the first k positions of the array. Your algorithm should do the minimum amount of work necessary, that is, no more of the array than necessary should be sorted.

7.12 Modify Quicksort to sort a sequence of variable-length strings stored one after the other in a character array, with a second array (storing pointers to strings) used to index the strings. Your function should modify the index array so that the first pointer points to the beginning of the lowest valued string, and so on.

7.13 Graph $f_1(n) = n \log n$, $f_2(n) = n^{1.5}$, and $f_3(n) = n^2$ in the range $1 \leq n \leq 1000$ to visually compare their growth rates. Typically, the constant factor in the running-time expression for an implementation of Insertion Sort will be less than the constant factors for Shellsort or Quicksort. How many times greater can the constant factor be for Shellsort to be faster than Insertion Sort when $n = 1000$? How many times greater can the constant factor be for Quicksort to be faster than Insertion Sort when $n = 1000$?

- 7.14** Imagine that there exists an algorithm **SPLITk** that can split a list **L** of n elements into k sublists, each containing one or more elements, such that sublist i contains only elements whose values are less than all elements in sublist j for $i < j \leq k$. If $n < k$, then $k - n$ sublists are empty, and the rest are of length 1. Assume that **SPLITk** has time complexity $O(\text{length of } L)$. Furthermore, assume that the k lists can be concatenated again in constant time. Consider the following algorithm:

```

List SORTk(List L) {
    List sub[k]; // To hold the sublists
    if (L.length() > 1) {
        SPLITk(L, sub); // SPLITk places sublists into sub
        for (i=0; i<k; i++)
            sub[i] = SORTk(sub[i]); // Sort each sublist
        L = concatenation of k sublists in sub;
        return L;
    }
}

```

- (a) What is the worst-case asymptotic running time for **SORTk**? Why?
 - (b) What is the average-case asymptotic running time of **SORTk**? Why?
- 7.15** Here is a variation on sorting. The problem is to sort a collection of n nuts and n bolts by size. It is assumed that for each bolt in the collection, there is a corresponding nut of the same size, but initially we do not know which nut goes with which bolt. The differences in size between two nuts or two bolts can be too small to see by eye, so you cannot rely on comparing the sizes of two nuts or two bolts directly. Instead, you can only compare the sizes of a nut and a bolt by attempting to screw one into the other (assume this comparison to be a constant time operation). This operation tells you that either the nut is bigger than the bolt, the bolt is bigger than the nut, or they are the same size. What is the minimum number of comparisons needed to sort the nuts and bolts in the worst case?
- 7.16** (a) Devise an algorithm to sort three numbers. It should make as few comparisons as possible. How many comparisons and swaps are required in the best, worst, and average cases?
- (b) Devise an algorithm to sort five numbers. It should make as few comparisons as possible. How many comparisons and swaps are required in the best, worst, and average cases?
- (c) Devise an algorithm to sort eight numbers. It should make as few comparisons as possible. How many comparisons and swaps are required in the best, worst, and average cases?
- 7.17** Devise an efficient algorithm to sort a set of numbers with values in the range 0 to 30,000. There are no duplicates. Keep memory requirements to a minimum.

- 7.18** Which of the following operations are best implemented by first sorting the list of numbers? For each operation, briefly describe an algorithm to implement it, and state the algorithm's asymptotic complexity.
- (a) Find the minimum value.
 - (b) Find the maximum value.
 - (c) Compute the arithmetic mean.
 - (d) Find the median (i.e., the middle value).
 - (e) Find the mode (i.e., the value that appears the most times).
- 7.19** Consider a recursive Mergesort implementation that calls Insertion Sort on sublists smaller than some threshold. If there are n calls to Mergesort, how many calls will there be to Insertion Sort? Why?
- 7.20** Implement Mergesort for the case where the input is a linked list.
- 7.21** Counting sort (assuming the input key values are integers in the range 0 to $m - 1$) works by counting the number of records with each key value in the first pass, and then uses this information to place the records in order in a second pass. Write an implementation of counting sort (see the implementation of radix sort for some ideas). What can we say about the relative values of m and n for this to be effective? If $m < n$, what is the running time of this algorithm?
- 7.22** Use an argument similar to that given in Section 7.9 to prove that $\log n$ is a worst-case lower bound for the problem of searching for a given value in a sorted array containing n elements.
- 7.23** A simpler way to do the Quicksort partition step is to set index **split** to the position of the first value greater than the pivot. Then from position **split+1** have another index **curr** move to the right until it finds a value less than a pivot. Swap the values at **split** and **next**, and increment **split**. Continue in this way, swapping the smaller values to the left side. When **curr** reaches the end of the subarray, **split** will be at the split point between the two partitions. Unfortunately, this approach requires more swaps than does the version presented in Section 7.5, resulting in a slower implementation. Give an example and explain why.

7.12 Projects

- 7.1** One possible improvement for Bubble Sort would be to add a flag variable and a test that determines if an exchange was made during the current iteration. If no exchange was made, then the list is sorted and so the algorithm can stop early. This makes the best case performance become $O(n)$ (because if the list is already sorted, then no iterations will take place on the first pass, and the sort will stop right there).

Modify the Bubble Sort implementation to add this flag and test. Compare the modified implementation on a range of inputs to determine if it does or does not improve performance in practice.

- 7.2** Double Insertion Sort is a variation on Insertion Sort that works from the middle of the array out. At each iteration, some middle portion of the array is sorted. On the next iteration, take the two adjacent elements to the sorted portion of the array. If they are out of order with respect to each other, then swap them. Now, push the left element toward the right in the array so long as it is greater than the element to its right. And push the right element toward the left in the array so long as it is less than the element to its left. The algorithm begins by processing the middle two elements of the array if the array is even. If the array is odd, then skip processing the middle item and begin with processing the elements to its immediate left and right.

First, explain what the cost of Double Insertion Sort will be in comparison to standard Insertion sort, and why. (Note that the two elements being processed in the current iteration, once initially swapped to be sorted with respect to each other, cannot cross as they are pushed into sorted position.) Then, implement Double Insertion Sort, being careful to properly handle both when the array is odd and when it is even. Compare its running time in practice against standard Insertion Sort. Finally, explain how this speedup might affect the threshold level and running time for a Quicksort implementation.

- 7.3** Perform a study of Shellsort, using different increments. Compare the version shown in Section 7.3, where each increment is half the previous one, with others. In particular, try implementing “division by 3” where the increments on a list of length n will be $n/3$, $n/9$, etc. Do other increment schemes work as well?
- 7.4** The implementation for Mergesort given in Section 7.4 takes an array as input and sorts that array. At the beginning of Section 7.4 there is a simple pseudocode implementation for sorting a linked list using Mergesort. Implement both a linked list-based version of Mergesort and the array-based version of Mergesort, and compare their running times.
- 7.5** Starting with the C++ code for Quicksort given in this chapter, write a series of Quicksort implementations to test the following optimizations on a wide range of input data sizes. Try these optimizations in various combinations to try and develop the fastest possible Quicksort implementation that you can.
- (a) Look at more values when selecting a pivot.
 - (b) Do not make a recursive call to `qsort` when the list size falls below a given threshold, and use Insertion Sort to complete the sorting process. Test various values for the threshold size.
 - (c) Eliminate recursion by using a stack and inline functions.

- 7.6** It has been proposed that Heapsort can be optimized by altering the heap's sift-down function. Call the value being sifted down X . Sift-down does two comparisons per level: First the children of X are compared, then the winner is compared to X . If X is too small, it is swapped with its larger child and the process repeated. The proposed optimization dispenses with the test against X . Instead, the larger child automatically replaces X , until X reaches the bottom level of the heap. At this point, X might be too large to remain in that position. This is corrected by repeatedly comparing X with its parent and swapping as necessary to “bubble” it up to its proper level. The claim is that this process will save a number of comparisons because most nodes when sifted down end up near the bottom of the tree anyway. Implement both versions of sift-down, and do an empirical study to compare their running times.
- 7.7** Radix Sort is typically implemented to support only a radix that is a power of two. This allows for a direct conversion from the radix to some number of bits in an integer key value. For example, if the radix is 16, then a 32-bit key will be processed in 8 steps of 4 bits each. This can lead to a more efficient implementation because bit shifting can replace the division operations shown in the implementation of Section 7.7. Re-implement the Radix Sort code given in Section 7.7 to use bit shifting in place of division. Compare the running time of the old and new Radix Sort implementations.
- 7.8** Write your own collection of sorting programs to implement the algorithms described in this chapter, and compare their running times. Be sure to implement optimized versions, trying to make each program as fast as possible. Do you get the same relative timings as shown in Figure 7.20? If not, why do you think this happened? How do your results compare with those of your classmates? What does this say about the difficulty of doing empirical timing studies?

File Processing and External Sorting

Earlier chapters presented basic data structures and algorithms that operate on data stored in main memory. Some applications require that large amounts of information be stored and processed — so much information that it cannot all fit into main memory. In that case, the information must reside on disk and be brought into main memory selectively for processing.

You probably already realize that main memory access is much faster than access to data stored on disk or other storage devices. The relative difference in access times is so great that efficient disk-based programs require a different approach to algorithm design than most programmers are used to. As a result, many programmers do a poor job when it comes to file processing applications.

This chapter presents the fundamental issues relating to the design of algorithms and data structures for disk-based applications.¹ We begin with a description of the significant differences between primary memory and secondary storage. Section 8.2 discusses the physical aspects of disk drives. Section 8.3 presents basic methods for managing buffer pools. Section 8.4 discusses the C++ model for random access to data stored on disk. Section 8.5 discusses the basic principles for sorting collections of records too large to fit in main memory.

8.1 Primary versus Secondary Storage

Computer storage devices are typically classified into **primary** or **main** memory and **secondary** or **peripheral** storage. Primary memory usually refers to **Random**

¹ Computer technology changes rapidly. I provide examples of disk drive specifications and other hardware performance numbers that are reasonably up to date as of the time when the book was written. When you read it, the numbers might seem out of date. However, the basic principles do not change. The approximate ratios for time, space, and cost between memory and disk have remained surprisingly steady for over 20 years.

Medium	1996	1997	2000	2004	2006	2008	2011
RAM	\$45.00	7.00	1.500	0.3500	0.1500	0.0339	0.0138
Disk	0.25	0.10	0.010	0.0010	0.0005	0.0001	0.0001
USB drive	—	—	—	0.1000	0.0900	0.0029	0.0018
Floppy	0.50	0.36	0.250	0.2500	—	—	—
Tape	0.03	0.01	0.001	0.0003	—	—	—
Solid State	—	—	—	—	—	—	0.0021

Figure 8.1 Price comparison table for some writable electronic data storage media in common use. Prices are in US Dollars/MB.

Access Memory (RAM), while secondary storage refers to devices such as hard disk drives, solid state drives, removable “USB” drives, CDs, and DVDs. Primary memory also includes registers, cache, and video memories, but we will ignore them for this discussion because their existence does not affect the principal differences between primary and secondary memory.

Along with a faster CPU, every new model of computer seems to come with more main memory. As memory size continues to increase, is it possible that relatively slow disk storage will be unnecessary? Probably not, because the desire to store and process larger files grows at least as fast as main memory size. Prices for both main memory and peripheral storage devices have dropped dramatically in recent years, as demonstrated by Figure 8.1. However, the cost per unit of disk drive storage is about two orders of magnitude less than RAM and has been for many years.

There is now a wide range of removable media available for transferring data or storing data offline in relative safety. These include floppy disks (now largely obsolete), writable CDs and DVDs, “flash” drives, and magnetic tape. Optical storage such as CDs and DVDs costs roughly half the price of hard disk drive space per megabyte, and have become practical for use as backup storage within the past few years. Tape used to be much cheaper than other media, and was the preferred means of backup, but are not so popular now as other media have decreased in price. “Flash” drives cost the most per megabyte, but due to their storage capacity and flexibility, quickly replaced floppy disks as the primary storage device for transferring data between computer when direct network transfer is not available.

Secondary storage devices have at least two other advantages over RAM memory. Perhaps most importantly, disk, “flash,” and optical media are **persistent**, meaning that they are not erased from the media when the power is turned off. In contrast, RAM used for main memory is usually **volatile** — all information is lost with the power. A second advantage is that CDs and “USB” drives can easily be transferred between computers. This provides a convenient way to take information from one computer to another.

In exchange for reduced storage costs, persistence, and portability, secondary storage devices pay a penalty in terms of increased access time. While not all accesses to disk take the same amount of time (more on this later), the typical time required to access a byte of storage from a disk drive in 2011 is around 9 ms (i.e., 9 *thousandths* of a second). This might not seem slow, but compared to the time required to access a byte from main memory, this is fantastically slow. Typical access time from standard personal computer RAM in 2011 is about 5-10 nanoseconds (i.e., 5-10 *billionths* of a second). Thus, the time to access a byte of data from a disk drive is about six orders of magnitude greater than that required to access a byte from main memory. While disk drive and RAM access times are both decreasing, they have done so at roughly the same rate. The relative speeds have remained the same for over several decades, in that the difference in access time between RAM and a disk drive has remained in the range between a factor of 100,000 and 1,000,000.

To gain some intuition for the significance of this speed difference, consider the time that it might take for you to look up the entry for disk drives in the index of this book, and then turn to the appropriate page. Call this your “primary memory” access time. If it takes you about 20 seconds to perform this access, then an access taking 500,000 times longer would require months.

It is interesting to note that while processing speeds have increased dramatically, and hardware prices have dropped dramatically, disk and memory access times have improved by less than an order of magnitude over the past 15 years. However, the situation is really much better than that modest speedup would suggest. During the same time period, the size of both disk and main memory has increased by over three orders of magnitude. Thus, the access times have actually decreased in the face of a massive increase in the density of these storage devices.

Due to the relatively slow access time for data on disk as compared to main memory, great care is required to create efficient applications that process disk-based information. The million-to-one ratio of disk access time versus main memory access time makes the following rule of paramount importance when designing disk-based applications:

Minimize the number of disk accesses!

There are generally two approaches to minimizing disk accesses. The first is to arrange information so that if you do access data from secondary memory, you will get what you need in as few accesses as possible, and preferably on the first access. **File structure** is the term used for a data structure that organizes data stored in secondary memory. File structures should be organized so as to minimize the required number of disk accesses. The other way to minimize disk accesses is to save information previously retrieved (or retrieve additional data with each access at little additional cost) that can be used to minimize the need for future accesses.

This requires the ability to guess accurately what information will be needed later and store it in primary memory now. This is referred to as **caching**.

8.2 Disk Drives

A C++ programmer views a random access file stored on disk as a contiguous series of bytes, with those bytes possibly combining to form data records. This is called the **logical** file. The **physical** file actually stored on disk is usually not a contiguous series of bytes. It could well be in pieces spread all over the disk. The **file manager**, a part of the operating system, is responsible for taking requests for data from a logical file and mapping those requests to the physical location of the data on disk. Likewise, when writing to a particular logical byte position with respect to the beginning of the file, this position must be converted by the file manager into the corresponding physical location on the disk. To gain some appreciation for the approximate time costs for these operations, you need to understand the physical structure and basic workings of a disk drive.

Disk drives are often referred to as **direct access** storage devices. This means that it takes roughly equal time to access any record in the file. This is in contrast to **sequential access** storage devices such as tape drives, which require the tape reader to process data from the beginning of the tape until the desired position has been reached. As you will see, the disk drive is only approximately direct access: At any given time, some records are more quickly accessible than others.

8.2.1 Disk Drive Architecture

A hard disk drive is composed of one or more round **platters**, stacked one on top of another and attached to a central **spindle**. Platters spin continuously at a constant rate. Each usable surface of each platter is assigned a **read/write head** or **I/O head** through which data are read or written, somewhat like the arrangement of a phonograph player's arm "reading" sound from a phonograph record. Unlike a phonograph needle, the disk read/write head does not actually touch the surface of a hard disk. Instead, it remains slightly above the surface, and any contact during normal operation would damage the disk. This distance is very small, much smaller than the height of a dust particle. It can be likened to a 5000-kilometer airplane trip across the United States, with the plane flying at a height of one meter!

A hard disk drive typically has several platters and several read/write heads, as shown in Figure 8.2(a). Each head is attached to an **arm**, which connects to the **boom**.² The boom moves all of the heads in or out together. When the heads are in some position over the platters, there are data on each platter directly accessible

² This arrangement, while typical, is not necessarily true for all disk drives. Nearly everything said here about the physical arrangement of disk drives represents a typical engineering compromise, not a fundamental design principle. There are many ways to design disk drives, and the engineering

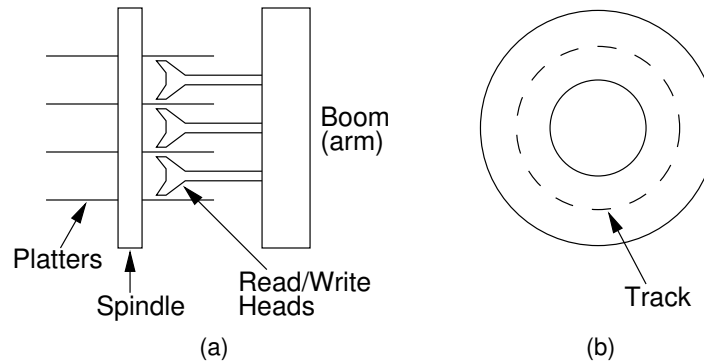


Figure 8.2 (a) A typical disk drive arranged as a stack of platters. (b) One track on a disk drive platter.

to each head. The data on a single platter that are accessible to any one position of the head for that platter are collectively called a **track**, that is, all data on a platter that are a fixed distance from the spindle, as shown in Figure 8.2(b). The collection of all tracks that are a fixed distance from the spindle is called a **cylinder**. Thus, a cylinder is all of the data that can be read when the arms are in a particular position.

Each track is subdivided into **sectors**. Between each sector there are **inter-sector gaps** in which no data are stored. These gaps allow the read head to recognize the end of a sector. Note that each sector contains the same amount of data. Because the outer tracks have greater length, they contain fewer bits per inch than do the inner tracks. Thus, about half of the potential storage space is wasted, because only the innermost tracks are stored at the highest possible data density. This arrangement is illustrated by Figure 8.3a. Disk drives today actually group tracks into “zones” such that the tracks in the innermost zone adjust their data density going out to maintain the same radial data density, then the tracks of the next zone reset the data density to make better use of their storage ability, and so on. This arrangement is shown in Figure 8.3b.

In contrast to the physical layout of a hard disk, a CD-ROM consists of a single spiral track. Bits of information along the track are equally spaced, so the information density is the same at both the outer and inner portions of the track. To keep the information flow at a constant rate along the spiral, the drive must speed up the rate of disk spin as the I/O head moves toward the center of the disk. This makes for a more complicated and slower mechanism.

Three separate steps take place when reading a particular byte or series of bytes of data from a hard disk. First, the I/O head moves so that it is positioned over the track containing the data. This movement is called a **seek**. Second, the sector containing the data rotates to come under the head. When in use the disk is always

compromises change over time. In addition, most of the description given here for disk drives is a simplified version of the reality. But this is a useful working model to understand what is going on.

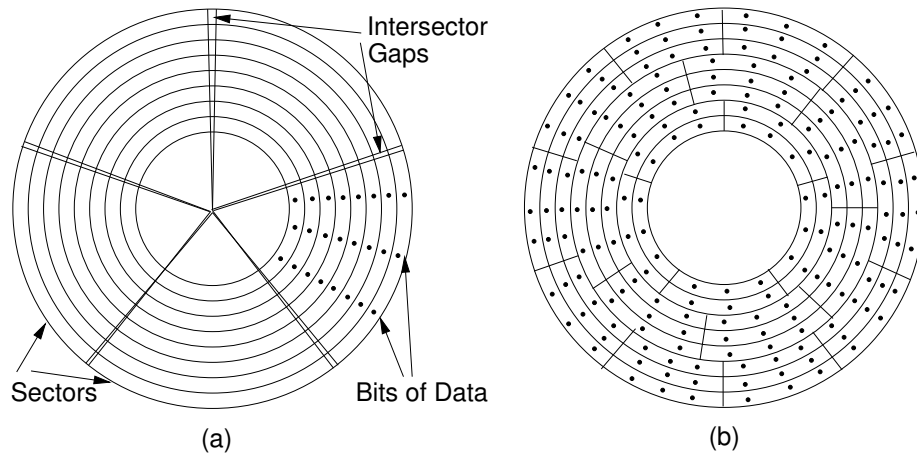


Figure 8.3 The organization of a disk platter. Dots indicate density of information. (a) Nominal arrangement of tracks showing decreasing data density when moving outward from the center of the disk. (b) A “zoned” arrangement with the sector size and density periodically reset in tracks further away from the center.

spinning. At the time of this writing, typical disk spin rates are 7200 rotations per minute (rpm). The time spent waiting for the desired sector to come under the I/O head is called **rotational delay** or **rotational latency**. The third step is the actual transfer (i.e., reading or writing) of data. It takes relatively little time to read information once the first byte is positioned under the I/O head, simply the amount of time required for it all to move under the head. In fact, disk drives are designed not to read one byte of data, but rather to read an entire sector of data at each request. Thus, a sector is the minimum amount of data that can be read or written at one time.

In general, it is desirable to keep all sectors for a file together on as few tracks as possible. This desire stems from two assumptions:

1. Seek time is slow (it is typically the most expensive part of an I/O operation), and
2. If one sector of the file is read, the next sector will probably soon be read.

Assumption (2) is called **locality of reference**, a concept that comes up frequently in computer applications.

Contiguous sectors are often grouped to form a **cluster**. A cluster is the smallest unit of allocation for a file, so all files are a multiple of the cluster size. The cluster size is determined by the operating system. The file manager keeps track of which clusters make up each file.

In Microsoft Windows systems, there is a designated portion of the disk called the **File Allocation Table**, which stores information about which sectors belong to which file. In contrast, UNIX does not use clusters. The smallest unit of file

allocation and the smallest unit that can be read/written is a sector, which in UNIX terminology is called a **block**. UNIX maintains information about file organization in certain disk blocks called **i-nodes**.

A group of physically contiguous clusters from the same file is called an **extent**. Ideally, all clusters making up a file will be contiguous on the disk (i.e., the file will consist of one extent), so as to minimize seek time required to access different portions of the file. If the disk is nearly full when a file is created, there might not be an extent available that is large enough to hold the new file. Furthermore, if a file grows, there might not be free space physically adjacent. Thus, a file might consist of several extents widely spaced on the disk. The fuller the disk, and the more that files on the disk change, the worse this file fragmentation (and the resulting seek time) becomes. File fragmentation leads to a noticeable degradation in performance as additional seeks are required to access data.

Another type of problem arises when the file's logical record size does not match the sector size. If the sector size is not a multiple of the record size (or vice versa), records will not fit evenly within a sector. For example, a sector might be 2048 bytes long, and a logical record 100 bytes. This leaves room to store 20 records with 48 bytes left over. Either the extra space is wasted, or else records are allowed to cross sector boundaries. If a record crosses a sector boundary, two disk accesses might be required to read it. If the space is left empty instead, such wasted space is called **internal fragmentation**.

A second example of internal fragmentation occurs at cluster boundaries. Files whose size is not an even multiple of the cluster size must waste some space at the end of the last cluster. The worst case will occur when file size modulo cluster size is one (for example, a file of 4097 bytes and a cluster of 4096 bytes). Thus, cluster size is a tradeoff between large files processed sequentially (where a large cluster size is desirable to minimize seeks) and small files (where small clusters are desirable to minimize wasted storage).

Every disk drive organization requires that some disk space be used to organize the sectors, clusters, and so forth. The layout of sectors within a track is illustrated by Figure 8.4. Typical information that must be stored on the disk itself includes the File Allocation Table, **sector headers** that contain address marks and information about the condition (whether usable or not) for each sector, and gaps between sectors. The sector header also contains error detection codes to help verify that the data have not been corrupted. This is why most disk drives have a "nominal" size that is greater than the actual amount of user data that can be stored on the drive. The difference is the amount of space required to organize the information on the disk. Even more space will be lost due to fragmentation.

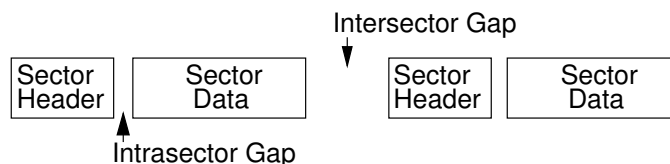


Figure 8.4 An illustration of sector gaps within a track. Each sector begins with a sector header containing the sector address and an error detection code for the contents of that sector. The sector header is followed by a small intra-sector gap, followed in turn by the sector data. Each sector is separated from the next sector by a larger inter-sector gap.

8.2.2 Disk Access Costs

When a seek is required, it is usually the primary cost when accessing information on disk. This assumes of course that a seek is necessary. When reading a file in sequential order (if the sectors comprising the file are contiguous on disk), little seeking is necessary. However, when accessing a random disk sector, seek time becomes the dominant cost for the data access. While the actual seek time is highly variable, depending on the distance between the track where the I/O head currently is and the track where the head is moving to, we will consider only two numbers. One is the track-to-track cost, or the minimum time necessary to move from a track to an adjacent track. This is appropriate when you want to analyze access times for files that are well placed on the disk. The second number is the average seek time for a random access. These two numbers are often provided by disk manufacturers. A typical example is the Western Digital Caviar serial ATA drive. The manufacturer's specifications indicate that the track-to-track time is 2.0 ms and the average seek time is 9.0 ms. In 2008 a typical drive in this line might be 120GB in size. In 2011, that same line of drives had sizes of up to 2 or 3TB. In both years, the advertised track-to-track and average seek times were identical.

For many years, typical rotation speed for disk drives was 3600 rpm, or one rotation every 16.7 ms. Most disk drives in 2011 had a rotation speed of 7200 rpm, or 8.3 ms per rotation. When reading a sector at random, you can expect that the disk will need to rotate halfway around to bring the desired sector under the I/O head, or 4.2 ms for a 7200-rpm disk drive.

Once under the I/O head, a sector of data can be transferred as fast as that sector rotates under the head. If an entire track is to be read, then it will require one rotation (8.3 ms at 7200 rpm) to move the full track under the head. If only part of the track is to be read, then proportionately less time will be required. For example, if there are 16,000 sectors on the track and one sector is to be read, this will require a trivial amount of time (1/16,000 of a rotation).

Example 8.1 Assume that an older disk drive has a total (nominal) capacity of 16.8GB spread among 10 platters, yielding 1.68GB/platter. Each

platter contains 13,085 tracks and each track contains (after formatting) 256 sectors of 512 bytes/sector. Track-to-track seek time is 2.2 ms and average seek time for random access is 9.5 ms. Assume the operating system maintains a cluster size of 8 sectors per cluster (4KB), yielding 32 clusters per track. The disk rotation rate is 5400 rpm (11.1 ms per rotation). Based on this information we can estimate the cost for various file processing operations.

How much time is required to read the track? On average, it will require half a rotation to bring the first sector of the track under the I/O head, and then one complete rotation to read the track.

How long will it take to read a file of 1MB divided into 2048 sector-sized (512 byte) records? This file will be stored in 256 clusters, because each cluster holds 8 sectors. The answer to the question depends largely on how the file is stored on the disk, that is, whether it is all together or broken into multiple extents. We will calculate both cases to see how much difference this makes.

If the file is stored so as to fill all of the sectors of eight adjacent tracks, then the cost to read the first sector will be the time to seek to the first track (assuming this requires a random seek), then a wait for the initial rotational delay, and then the time to read (which is the same as the time to rotate the disk again). This requires

$$9.5 + 11.1 \times 1.5 = 26.2 \text{ ms.}$$

At this point, because we assume that the next seven tracks require only a track-to-track seek because they are adjacent. Each requires

$$2.2 + 11.1 \times 1.5 = 18.9 \text{ ms.}$$

The total time required is therefore

$$26.2\text{ms} + 7 \times 18.9\text{ms} = 158.5\text{ms.}$$

If the file's clusters are spread randomly across the disk, then we must perform a seek for each cluster, followed by the time for rotational delay. Once the first sector of the cluster comes under the I/O head, very little time is needed to read the cluster because only 8/256 of the track needs to rotate under the head, for a total time of about 5.9 ms for latency and read time. Thus, the total time required is about

$$256(9.5 + 5.9) \approx 3942\text{ms}$$

or close to 4 seconds. This is much longer than the time required when the file is all together on disk!

This example illustrates why it is important to keep disk files from becoming fragmented, and why so-called “disk defragmenters” can speed up file processing time. File fragmentation happens most commonly when the disk is nearly full and the file manager must search for free space whenever a file is created or changed.

8.3 Buffers and Buffer Pools

Given the specifications of the disk drive from Example 8.1, we find that it takes about $9.5 + 11.1 \times 1.5 = 26.2$ ms to read one track of data on average. It takes about $9.5 + 11.1/2 + (1/256) \times 11.1 = 15.1$ ms on average to read a single sector of data. This is a good savings (slightly over half the time), but less than 1% of the data on the track are read. If we want to read only a single byte, it would save us effectively no time over that required to read an entire sector. For this reason, nearly all disk drives automatically read or write an entire sector’s worth of information whenever the disk is accessed, even when only one byte of information is requested.

Once a sector is read, its information is stored in main memory. This is known as **buffering** or **caching** the information. If the next disk request is to that same sector, then it is not necessary to read from disk again because the information is already stored in main memory. Buffering is an example of one method for minimizing disk accesses mentioned at the beginning of the chapter: Bring off additional information from disk to satisfy future requests. If information from files were accessed at random, then the chance that two consecutive disk requests are to the same sector would be low. However, in practice most disk requests are close to the location (in the logical file at least) of the previous request. This means that the probability of the next request “hitting the cache” is much higher than chance would indicate.

This principle explains one reason why average access times for new disk drives are lower than in the past. Not only is the hardware faster, but information is also now stored using better algorithms and larger caches that minimize the number of times information needs to be fetched from disk. This same concept is also used to store parts of programs in faster memory within the CPU, using the CPU cache that is prevalent in modern microprocessors.

Sector-level buffering is normally provided by the operating system and is often built directly into the disk drive controller hardware. Most operating systems maintain at least two buffers, one for input and one for output. Consider what would happen if there were only one buffer during a byte-by-byte copy operation. The sector containing the first byte would be read into the I/O buffer. The output operation would need to destroy the contents of the single I/O buffer to write this byte. Then the buffer would need to be filled again from disk for the second byte,

only to be destroyed during output. The simple solution to this problem is to keep one buffer for input, and a second for output.

Most disk drive controllers operate independently from the CPU once an I/O request is received. This is useful because the CPU can typically execute millions of instructions during the time required for a single I/O operation. A technique that takes maximum advantage of this micro-parallelism is **double buffering**. Imagine that a file is being processed sequentially. While the first sector is being read, the CPU cannot process that information and so must wait or find something else to do in the meantime. Once the first sector is read, the CPU can start processing while the disk drive (in parallel) begins reading the second sector. If the time required for the CPU to process a sector is approximately the same as the time required by the disk controller to read a sector, it might be possible to keep the CPU continuously fed with data from the file. The same concept can also be applied to output, writing one sector to disk while the CPU is writing to a second output buffer in memory. Thus, in computers that support double buffering, it pays to have at least two input buffers and two output buffers available.

Caching information in memory is such a good idea that it is usually extended to multiple buffers. The operating system or an application program might store many buffers of information taken from some **backing storage** such as a disk file. This process of using buffers as an intermediary between a user and a disk file is called **buffering** the file. The information stored in a buffer is often called a **page**, and the collection of buffers is called a **buffer pool**. The goal of the buffer pool is to increase the amount of information stored in memory in hopes of increasing the likelihood that new information requests can be satisfied from the buffer pool rather than requiring new information to be read from disk.

As long as there is an unused buffer available in the buffer pool, new information can be read in from disk on demand. When an application continues to read new information from disk, eventually all of the buffers in the buffer pool will become full. Once this happens, some decision must be made about what information in the buffer pool will be sacrificed to make room for newly requested information.

When replacing information contained in the buffer pool, the goal is to select a buffer that has “unnecessary” information, that is, the information least likely to be requested again. Because the buffer pool cannot know for certain what the pattern of future requests will look like, a decision based on some **heuristic**, or best guess, must be used. There are several approaches to making this decision.

One heuristic is “first-in, first-out” (FIFO). This scheme simply orders the buffers in a queue. The buffer at the front of the queue is used next to store new information and then placed at the end of the queue. In this way, the buffer to be replaced is the one that has held its information the longest, in hopes that this information is no longer needed. This is a reasonable assumption when processing moves along the file at some steady pace in roughly sequential order. However,

many programs work with certain key pieces of information over and over again, and the importance of information has little to do with how long ago the information was first accessed. Typically it is more important to know how many times the information has been accessed, or how recently the information was last accessed.

Another approach is called “least frequently used” (LFU). LFU tracks the number of accesses to each buffer in the buffer pool. When a buffer must be reused, the buffer that has been accessed the fewest number of times is considered to contain the “least important” information, and so it is used next. LFU, while it seems intuitively reasonable, has many drawbacks. First, it is necessary to store and update access counts for each buffer. Second, what was referenced many times in the past might now be irrelevant. Thus, some time mechanism where counts “expire” is often desirable. This also avoids the problem of buffers that slowly build up big counts because they get used just often enough to avoid being replaced. An alternative is to maintain counts for all sectors ever read, not just the sectors currently in the buffer pool. This avoids immediately replacing the buffer just read, which has not yet had time to build a high access count.

The third approach is called “least recently used” (LRU). LRU simply keeps the buffers in a list. Whenever information in a buffer is accessed, this buffer is brought to the front of the list. When new information must be read, the buffer at the back of the list (the one least recently used) is taken and its “old” information is either discarded or written to disk, as appropriate. This is an easily implemented approximation to LFU and is often the method of choice for managing buffer pools unless special knowledge about information access patterns for an application suggests a special-purpose buffer management scheme.

The main purpose of a buffer pool is to minimize disk I/O. When the contents of a block are modified, we could write the updated information to disk immediately. But what if the block is changed again? If we write the block’s contents after every change, that might be a lot of disk write operations that can be avoided. It is more efficient to wait until either the file is to be closed, or the contents of the buffer containing that block is to be flushed from the buffer pool.

When a buffer’s contents are to be replaced in the buffer pool, we only want to write the contents to disk if it is necessary. That would be necessary only if the contents have changed since the block was read in originally from the file. The way to insure that the block is written when necessary, but only when necessary, is to maintain a Boolean variable with the buffer (often referred to as the **dirty bit**) that is turned on when the buffer’s contents are modified by the client. At the time when the block is flushed from the buffer pool, it is written to disk if and only if the dirty bit has been turned on.

Modern operating systems support **virtual memory**. Virtual memory is a technique that allows the programmer to write programs as though there is more of the faster main memory (such as RAM) than actually exists. Virtual memory makes use

of a buffer pool to store data read from blocks on slower, secondary memory (such as on the disk drive). The disk stores the complete contents of the virtual memory. Blocks are read into main memory as demanded by memory accesses. Naturally, programs using virtual memory techniques are slower than programs whose data are stored completely in main memory. The advantage is reduced programmer effort because a good virtual memory system provides the appearance of larger main memory without modifying the program.

Example 8.2 Consider a virtual memory whose size is ten sectors, and which has a buffer pool of five buffers (each one sector in size) associated with it. We will use a LRU replacement scheme. The following series of memory requests occurs.

9017668135171

After the first five requests, the buffer pool will store the sectors in the order 6, 7, 1, 0, 9. Because Sector 6 is already at the front, the next request can be answered without reading new data from disk or reordering the buffers. The request to Sector 8 requires emptying the contents of the least recently used buffer, which contains Sector 9. The request to Sector 1 brings the buffer holding Sector 1's contents back to the front. Processing the remaining requests results in the buffer pool as shown in Figure 8.5.

Example 8.3 Figure 8.5 illustrates a buffer pool of five blocks mediating a virtual memory of ten blocks. At any given moment, up to five sectors of information can be in main memory. Assume that Sectors 1, 7, 5, 3, and 8 are currently in the buffer pool, stored in this order, and that we use the LRU buffer replacement strategy. If a request for Sector 9 is then received, then one sector currently in the buffer pool must be replaced. Because the buffer containing Sector 8 is the least recently used buffer, its contents will be copied back to disk at Sector 8. The contents of Sector 9 are then copied into this buffer, and it is moved to the front of the buffer pool (leaving the buffer containing Sector 3 as the new least-recently used buffer). If the next memory request were to Sector 5, no data would need to be read from disk. Instead, the buffer already containing Sector 5 would be moved to the front of the buffer pool.

When implementing buffer pools, there are two basic approaches that can be taken regarding the transfer of information between the user of the buffer pool and the buffer pool class itself. The first approach is to pass “messages” between the two. This approach is illustrated by the following abstract class:

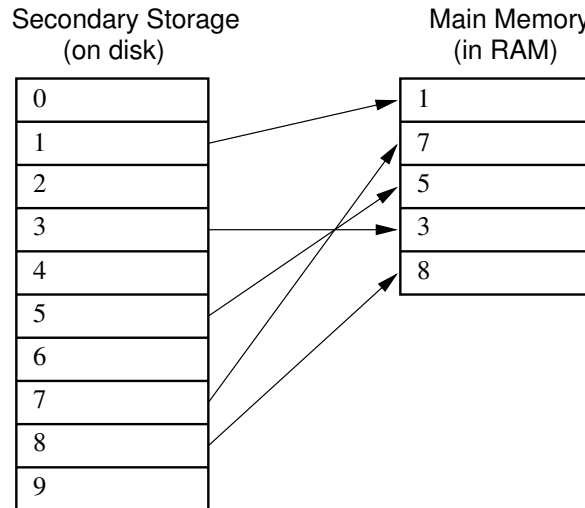


Figure 8.5 An illustration of virtual memory. The complete collection of information resides in the slower, secondary storage (on disk). Those sectors recently accessed are held in the fast main memory (in RAM). In this example, copies of Sectors 1, 7, 5, 3, and 8 from secondary storage are currently stored in the main memory. If a memory access to Sector 9 is received, one of the sectors currently in main memory must be replaced.

```
// ADT for buffer pools using the message-passing style
class BufferPool {
public:
    // Copy "sz" bytes from "space" to position "pos" in the
    //   buffered storage.
    virtual void insert(void* space, int sz, int pos) = 0;

    // Copy "sz" bytes from position "pos" of the buffered
    //   storage to "space".
    virtual void getbytes(void* space, int sz, int pos) = 0;
};
```

This simple class provides an interface with two member functions, **insert** and **getbytes**. The information is passed between the buffer pool user and the buffer pool through the **space** parameter. This is storage space, provided by the bufferpool client and at least **sz** bytes long, which the buffer pool can take information from (the **insert** function) or put information into (the **getbytes** function). Parameter **pos** indicates where the information will be placed in the buffer pool's logical storage space. Physically, it will actually be copied to the appropriate byte position in some buffer in the buffer pool. This ADT is similar to the **read** and **write** methods of the **RandomAccessFile** class discussed in Section 8.4.

Example 8.4 Assume each sector of the disk file (and thus each block in the buffer pool) stores 1024 bytes. Assume that the buffer pool is in the state shown in Figure 8.5. If the next request is to copy 40 bytes beginning at position 6000 of the file, these bytes should be placed into Sector 5 (whose bytes go from position 5120 to position 6143). Because Sector 5 is currently in the buffer pool, we simply copy the 40 bytes contained in **space** to byte positions 880-919. The buffer containing Sector 5 is then moved to the buffer pool ahead of the buffer containing Sector 1.

An alternative interface is to have the buffer pool provide to the user a direct pointer to a buffer that contains the requested information. Such an interface might look as follows:

```
// ADT for buffer pools using the buffer-passing style
class BufferPool {
public:
    // Return pointer to the requested block
    virtual void* getblock(int block) = 0;

    // Set the dirty bit for the buffer holding "block"
    virtual void dirtyblock(int block) = 0;

    // Tell the size of a buffer
    virtual int blocksize() = 0;
};
```

In this approach, the buffer pool user is made aware that the storage space is divided into blocks of a given size, where each block is the size of a buffer. The user requests specific blocks from the buffer pool, with a pointer to the buffer holding the requested block being returned to the user. The user might then read from or write to this space. If the user writes to the space, the buffer pool must be informed of this fact. The reason is that, when a given block is to be removed from the buffer pool, the contents of that block must be written to the backing storage if it has been modified. If the block has not been modified, then it is unnecessary to write it out.

Example 8.5 We wish to write 40 bytes beginning at logical position 6000 in the file. Assume that the buffer pool is in the state shown in Figure 8.5. Using the second ADT, the client would need to know that blocks (buffers) are of size 1024, and therefore would request access to Sector 5. A pointer to the buffer containing Sector 5 would be returned by the call to **getblock**. The client would then copy 40 bytes to positions 880-919 of the buffer, and call **dirtyblock** to warn the buffer pool that the contents of this block have been modified.

A variation on this approach is to have the **getblock** function take another parameter to indicate the “mode” of use for the information. If the mode is **READ** then the buffer pool assumes that no changes will be made to the buffer’s contents (and so no write operation need be done when the buffer is reused to store another block). If the mode is **WRITE** then the buffer pool assumes that the client will not look at the contents of the buffer and so no read from the file is necessary. If the mode is **READ AND WRITE** then the buffer pool would read the existing contents of the block in from disk, and write the contents of the buffer to disk when the buffer is to be reused. Using the “mode” approach, the **dirtyblock** method is avoided.

One problem with the buffer-passing ADT is the risk of **stale pointers**. When the buffer pool user is given a pointer to some buffer space at time **T1**, that pointer does indeed refer to the desired data at that time. As further requests are made to the buffer pool, it is possible that the data in any given buffer will be removed and replaced with new data. If the buffer pool user at a later time **T2** then refers to the data referred to by the pointer given at time **T1**, it is possible that the data are no longer valid because the buffer contents have been replaced in the meantime. Thus the pointer into the buffer pool’s memory has become “stale.” To guarantee that a pointer is not stale, it should not be used if intervening requests to the buffer pool have taken place.

We can solve this problem by introducing the concept of a user (or possibly multiple users) gaining access to a buffer, and then releasing the buffer when done. We will add method **acquireBuffer** and **releaseBuffer** for this purpose. Method **acquireBuffer** takes a block ID as input and returns a pointer to the buffer that will be used to store this block. The buffer pool will keep a count of the number of requests currently active for this block. Method **releaseBuffer** will reduce the count of active users for the associated block. Buffers associated with active blocks will not be eligible for flushing from the buffer pool. This will lead to a problem if the client neglects to release active blocks when they are no longer needed. There would also be a problem if there were more total active blocks than buffers in the buffer pool. However, the buffer pool should always be initialized to include more buffers than should ever be active at one time.

An additional problem with both ADTs presented so far comes when the user intends to completely overwrite the contents of a block, and does not need to read in the old contents already on disk. However, the buffer pool cannot in general know whether the user wishes to use the old contents or not. This is especially true with the message-passing approach where a given message might overwrite only part of the block. In this case, the block will be read into memory even when not needed, and then its contents will be overwritten.

This inefficiency can be avoided (at least in the buffer-passing version) by separating the assignment of blocks to buffers from actually reading in data for the

block. In particular, the following revised buffer-passing ADT does not actually read data in the **acquireBuffer** method. Users who wish to see the old contents must then issue a **readBlock** request to read the data from disk into the buffer, and then a **getDataPointer** request to gain direct access to the buffer's data contents.

```
// A single buffer in the buffer pool
class Buffer {
public:
    // Read the associated block from disk (if necessary) and
    // return a pointer to the data
    void *readBlock() = 0;

    // Return a pointer to the buffer's data array
    // (without reading from disk)
    void *getDataPointer() = 0;

    // Flag the buffer's contents as having changed, so that
    // flushing the block will write it back to disk
    void markDirty() = 0;

    // Release the block's access to this buffer. Further
    // accesses to this buffer are illegal.
    void releaseBuffer() = 0;
}

// The bufferpool
class BufferPool {
public:
    // Constructor: The bufferpool has "numbuff" buffers that
    // each contain "buffsize" bytes of data.
    BufferPool(int numbuff, int buffsize) = 0;

    // Relate a block to a buffer, returning a pointer to a
    // buffer object
    Buffer *acquireBuffer(int block) = 0;
}
```

Again, a mode parameter could be added to the **acquireBuffer** method, eliminating the need for the **readBlock** and **markDirty** methods.

Clearly, the buffer-passing approach places more obligations on the user of the buffer pool. These obligations include knowing the size of a block, not corrupting the buffer pool's storage space, and informing the buffer pool both when a block has been modified and when it is no longer needed. So many obligations make this approach prone to error. An advantage is that there is no need to do an extra copy step when getting information from the user to the buffer. If the size of the records stored is small, this is not an important consideration. If the size of the records is large (especially if the record size and the buffer size are the same, as typically is the case when implementing B-trees, see Section 10.5), then this efficiency issue might

become important. Note however that the in-memory copy time will always be far less than the time required to write the contents of a buffer to disk. For applications where disk I/O is the bottleneck for the program, even the time to copy lots of information between the buffer pool user and the buffer might be inconsequential. Another advantage to buffer passing is the reduction in unnecessary read operations for data that will be overwritten anyway.

You should note that these implementations for the buffer pool ADT do not use templates. Instead, the **space** parameter and the buffer pointer are declared to be **void***. When a class uses a template, that means that the record type is arbitrary, but that the class knows what the record type is. In contrast, using a **void*** pointer for the space means that not only is the record type arbitrary, but also the buffer pool does not even know what the user's record type is. In fact, a given buffer pool might have many users who store many types of records.

In a buffer pool, the user decides where a given record will be stored but has no control over the precise mechanism by which data are transferred to the backing storage. This is in contrast to the memory manager described in Section 12.3 in which the user passes a record to the manager and has no control at all over where the record is stored.

8.4 The Programmer's View of Files

The C++ programmer's logical view of a random access file is a single stream of bytes. Interaction with a file can be viewed as a communications channel for issuing one of three instructions: read bytes from the current position in the file, write bytes to the current position in the file, and move the current position within the file. You do not normally see how the bytes are stored in sectors, clusters, and so forth. The mapping from logical to physical addresses is done by the file system, and sector-level buffering is done automatically by the disk controller.

When processing records in a disk file, the order of access can have a great effect on I/O time. A **random access** procedure processes records in an order independent of their logical order within the file. **Sequential access** processes records in order of their logical appearance within the file. Sequential processing requires less seek time if the physical layout of the disk file matches its logical layout, as would be expected if the file were created on a disk with a high percentage of free space.

C++ provides several mechanisms for manipulating disk files. One of the most commonly used is the **fstream** class. The following methods can be used to manipulate information in the file.

- **open(char *name, openmode flags)**: Open file name **name** for processing. **Flags** control various details such as whether the file permits

reading, writing, or both; and whether its pre-existing contents should be deleted.

- **read(char *buff, int count)**: Read **count** bytes from the current position in the file. The current file position moves forward as the bytes are read. The bytes are read into array **buff**, which must be at least **count** bytes long.
- **write(char *buff, int count)**: Write **count** bytes at the current position in the file (overwriting the bytes already at that position). The current file position moves forward as these bytes are written. The bytes to be written come from array **buff**.
- **seekg(int pos)** and **seekp(int pos)**: Move the current position in the file to **pos**. This allows bytes at arbitrary places within the file to be read or written. There are actually two “current” positions: one for reading and one for writing. Function **seekg** changes the “get” or read position, while function **seekp** changes the “put” or write position.
- **close()**: Close a file at the end of processing.

Note that the spirit of this ADT is similar to the “message passing” version of the ADT for buffer pools described in Section 8.3.

8.5 External Sorting

We now consider the problem of sorting collections of records too large to fit in main memory. Because the records must reside in peripheral or external memory, such sorting methods are called **external sorts**. This is in contrast to the internal sorts discussed in Chapter 7 which assume that the records to be sorted are stored in main memory. Sorting large collections of records is central to many applications, such as processing payrolls and other large business databases. As a consequence, many external sorting algorithms have been devised. Years ago, sorting algorithm designers sought to optimize the use of specific hardware configurations, such as multiple tape or disk drives. Most computing today is done on personal computers and low-end workstations with relatively powerful CPUs, but only one or at most two disk drives. The techniques presented here are geared toward optimized processing on a single disk drive. This approach allows us to cover the most important issues in external sorting while skipping many less important machine-dependent details. Readers who have a need to implement efficient external sorting algorithms that take advantage of more sophisticated hardware configurations should consult the references in Section 8.6.

When a collection of records is too large to fit in main memory, the only practical way to sort it is to read some records from disk, do some rearranging, then write them back to disk. This process is repeated until the file is sorted, with each record read perhaps many times. Given the high cost of disk I/O, it should come as

no surprise that the primary goal of an external sorting algorithm is to minimize the number of times information must be read from or written to disk. A certain amount of additional CPU processing can profitably be traded for reduced disk access.

Before discussing external sorting techniques, consider again the basic model for accessing information from disk. The file to be sorted is viewed by the programmer as a sequential series of fixed-size **blocks**. Assume (for simplicity) that each block contains the same number of fixed-size data records. Depending on the application, a record might be only a few bytes — composed of little or nothing more than the key — or might be hundreds of bytes with a relatively small key field. Records are assumed not to cross block boundaries. These assumptions can be relaxed for special-purpose sorting applications, but ignoring such complications makes the principles clearer.

As explained in Section 8.2, a sector is the basic unit of I/O. In other words, all disk reads and writes are for one or more complete sectors. Sector sizes are typically a power of two, in the range 512 to 16K bytes, depending on the operating system and the size and speed of the disk drive. The block size used for external sorting algorithms should be equal to or a multiple of the sector size.

Under this model, a sorting algorithm reads a block of data into a buffer in main memory, performs some processing on it, and at some future time writes it back to disk. From Section 8.1 we see that reading or writing a block from disk takes on the order of one million times longer than a memory access. Based on this fact, we can reasonably expect that the records contained in a single block can be sorted by an internal sorting algorithm such as Quicksort in less time than is required to read or write the block.

Under good conditions, reading from a file in sequential order is more efficient than reading blocks in random order. Given the significant impact of seek time on disk access, it might seem obvious that sequential processing is faster. However, it is important to understand precisely under what circumstances sequential file processing is actually faster than random access, because it affects our approach to designing an external sorting algorithm.

Efficient sequential access relies on seek time being kept to a minimum. The first requirement is that the blocks making up a file are in fact stored on disk in sequential order and close together, preferably filling a small number of contiguous tracks. At the very least, the number of extents making up the file should be small. Users typically do not have much control over the layout of their file on disk, but writing a file all at once in sequential order to a disk drive with a high percentage of free space increases the likelihood of such an arrangement.

The second requirement is that the disk drive's I/O head remain positioned over the file throughout sequential processing. This will not happen if there is competition of any kind for the I/O head. For example, on a multi-user time-shared computer the sorting process might compete for the I/O head with the processes

of other users. Even when the sorting process has sole control of the I/O head, it is still likely that sequential processing will not be efficient. Imagine the situation where all processing is done on a single disk drive, with the typical arrangement of a single bank of read/write heads that move together over a stack of platters. If the sorting process involves reading from an input file, alternated with writing to an output file, then the I/O head will continuously seek between the input file and the output file. Similarly, if two input files are being processed simultaneously (such as during a merge process), then the I/O head will continuously seek between these two files.

The moral is that, with a single disk drive, there often is no such thing as efficient sequential processing of a data file. Thus, a sorting algorithm might be more efficient if it performs a smaller number of non-sequential disk operations rather than a larger number of logically sequential disk operations that require a large number of seeks in practice.

As mentioned previously, the record size might be quite large compared to the size of the key. For example, payroll entries for a large business might each store hundreds of bytes of information including the name, ID, address, and job title for each employee. The sort key might be the ID number, requiring only a few bytes. The simplest sorting algorithm might be to process such records as a whole, reading the entire record whenever it is processed. However, this will greatly increase the amount of I/O required, because only a relatively few records will fit into a single disk block. Another alternative is to do a **key sort**. Under this method, the keys are all read and stored together in an **index file**, where each key is stored along with a pointer indicating the position of the corresponding record in the original data file. The key and pointer combination should be substantially smaller than the size of the original record; thus, the index file will be much smaller than the complete data file. The index file will then be sorted, requiring much less I/O because the index records are smaller than the complete records.

Once the index file is sorted, it is possible to reorder the records in the original database file. This is typically not done for two reasons. First, reading the records in sorted order from the record file requires a random access for each record. This can take a substantial amount of time and is only of value if the complete collection of records needs to be viewed or processed in sorted order (as opposed to a search for selected records). Second, database systems typically allow searches to be done on multiple keys. For example, today's processing might be done in order of ID numbers. Tomorrow, the boss might want information sorted by salary. Thus, there might be no single "sorted" order for the full record. Instead, multiple index files are often maintained, one for each sort key. These ideas are explored further in Chapter 10.

8.5.1 Simple Approaches to External Sorting

If your operating system supports virtual memory, the simplest “external” sort is to read the entire file into virtual memory and run an internal sorting method such as Quicksort. This approach allows the virtual memory manager to use its normal buffer pool mechanism to control disk accesses. Unfortunately, this might not always be a viable option. One potential drawback is that the size of virtual memory is usually limited to something much smaller than the disk space available. Thus, your input file might not fit into virtual memory. Limited virtual memory can be overcome by adapting an internal sorting method to make use of your own buffer pool.

A more general problem with adapting an internal sorting algorithm to external sorting is that it is not likely to be as efficient as designing a new algorithm with the specific goal of minimizing disk I/O. Consider the simple adaptation of Quicksort to use a buffer pool. Quicksort begins by processing the entire array of records, with the first partition step moving indices inward from the two ends. This can be implemented efficiently using a buffer pool. However, the next step is to process each of the subarrays, followed by processing of sub-subarrays, and so on. As the subarrays get smaller, processing quickly approaches random access to the disk drive. Even with maximum use of the buffer pool, Quicksort still must read and write each record $\log n$ times on average. We can do much better. Finally, even if the virtual memory manager can give good performance using a standard Quicksort, this will come at the cost of using a lot of the system’s working memory, which will mean that the system cannot use this space for other work. Better methods can save time while also using less memory.

Our approach to external sorting is derived from the Mergesort algorithm. The simplest form of external Mergesort performs a series of sequential passes over the records, merging larger and larger sublists on each pass. The first pass merges sublists of size 1 into sublists of size 2; the second pass merges the sublists of size 2 into sublists of size 4; and so on. A sorted sublist is called a **run**. Thus, each pass is merging pairs of runs to form longer runs. Each pass copies the contents of the file to another file. Here is a sketch of the algorithm, as illustrated by Figure 8.6.

1. Split the original file into two equal-sized **run files**.
2. Read one block from each run file into input buffers.
3. Take the first record from each input buffer, and write a run of length two to an output buffer in sorted order.
4. Take the next record from each input buffer, and write a run of length two to a second output buffer in sorted order.
5. Repeat until finished, alternating output between the two output run buffers. Whenever the end of an input block is reached, read the next block from the appropriate input file. When an output buffer is full, write it to the appropriate output file.

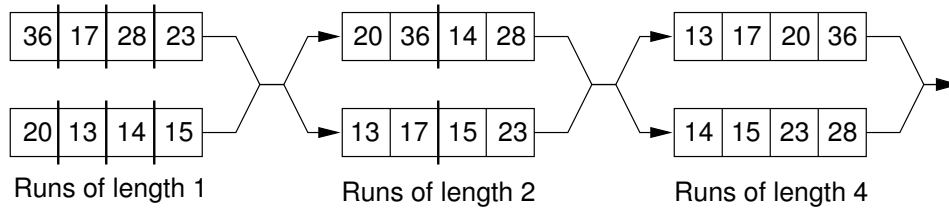


Figure 8.6 A simple external Mergesort algorithm. Input records are divided equally between two input files. The first runs from each input file are merged and placed into the first output file. The second runs from each input file are merged and placed in the second output file. Merging alternates between the two output files until the input files are empty. The roles of input and output files are then reversed, allowing the runlength to be doubled with each pass.

6. Repeat steps 2 through 5, using the original output files as input files. On the second pass, the first two records of each input run file are already in sorted order. Thus, these two runs may be merged and output as a single run of four elements.
7. Each pass through the run files provides larger and larger runs until only one run remains.

Example 8.6 Using the input of Figure 8.6, we first create runs of length one split between two input files. We then process these two input files sequentially, making runs of length two. The first run has the values 20 and 36, which are output to the first output file. The next run has 13 and 17, which is output to the second file. The run 14, 28 is sent to the first file, then run 15, 23 is sent to the second file, and so on. Once this pass has completed, the roles of the input files and output files are reversed. The next pass will merge runs of length two into runs of length four. Runs 20, 36 and 13, 17 are merged to send 13, 17, 20, 36 to the first output file. Then runs 14, 28 and 15, 23 are merged to send run 14, 15, 23, 28 to the second output file. In the final pass, these runs are merged to form the final run 13, 14, 15, 17, 20, 23, 28, 36.

This algorithm can easily take advantage of the double buffering techniques described in Section 8.3. Note that the various passes read the input run files sequentially and write the output run files sequentially. For sequential processing and double buffering to be effective, however, it is necessary that there be a separate I/O head available for each file. This typically means that each of the input and output files must be on separate disk drives, requiring a total of four disk drives for maximum efficiency.

The external Mergesort algorithm just described requires that $\log n$ passes be made to sort a file of n records. Thus, each record must be read from disk and written to disk $\log n$ times. The number of passes can be significantly reduced by observing that it is not necessary to use Mergesort on small runs. A simple modification is to read in a block of data, sort it in memory (perhaps using Quicksort), and then output it as a single sorted run.

Example 8.7 Assume that we have blocks of size 4KB, and records are eight bytes with four bytes of data and a 4-byte key. Thus, each block contains 512 records. Standard Mergesort would require nine passes to generate runs of 512 records, whereas processing each block as a unit can be done in one pass with an internal sort. These runs can then be merged by Mergesort. Standard Mergesort requires eighteen passes to process 256K records. Using an internal sort to create initial runs of 512 records reduces this to one initial pass to create the runs and nine merge passes to put them all together, approximately half as many passes.

We can extend this concept to improve performance even further. Available main memory is usually much more than one block in size. If we process larger initial runs, then the number of passes required by Mergesort is further reduced. For example, most modern computers can provide tens or even hundreds of megabytes of RAM to the sorting program. If all of this memory (excepting a small amount for buffers and local variables) is devoted to building initial runs as large as possible, then quite large files can be processed in few passes. The next section presents a technique for producing large runs, typically twice as large as could fit directly into main memory.

Another way to reduce the number of passes required is to increase the number of runs that are merged together during each pass. While the standard Mergesort algorithm merges two runs at a time, there is no reason why merging needs to be limited in this way. Section 8.5.3 discusses the technique of multiway merging.

Over the years, many variants on external sorting have been presented, but all are based on the following two steps:

1. Break the file into large initial runs.
2. Merge the runs together to form a single sorted file.

8.5.2 Replacement Selection

This section treats the problem of creating initial runs as large as possible from a disk file, assuming a fixed amount of RAM is available for processing. As mentioned previously, a simple approach is to allocate as much RAM as possible to a large array, fill this array from disk, and sort the array using Quicksort. Thus, if

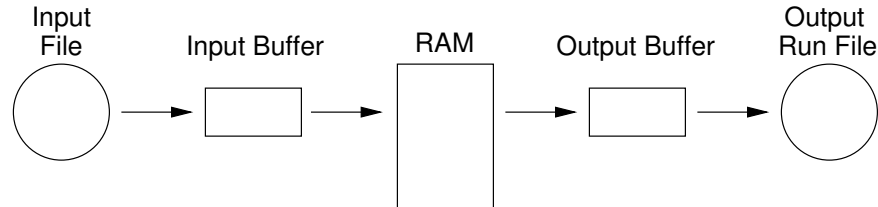


Figure 8.7 Overview of replacement selection. Input records are processed sequentially. Initially RAM is filled with M records. As records are processed, they are written to an output buffer. When this buffer becomes full, it is written to disk. Meanwhile, as replacement selection needs records, it reads them from the input buffer. Whenever this buffer becomes empty, the next block of records is read from disk.

the size of memory available for the array is M records, then the input file can be broken into initial runs of length M . A better approach is to use an algorithm called **replacement selection** that, on average, creates runs of $2M$ records in length. Replacement selection is actually a slight variation on the Heapsort algorithm. The fact that Heapsort is slower than Quicksort is irrelevant in this context because I/O time will dominate the total running time of any reasonable external sorting algorithm. Building longer initial runs will reduce the total I/O time required.

Replacement selection views RAM as consisting of an array of size M in addition to an input buffer and an output buffer. (Additional I/O buffers might be desirable if the operating system supports double buffering, because replacement selection does sequential processing on both its input and its output.) Imagine that the input and output files are streams of records. Replacement selection takes the next record in sequential order from the input stream when needed, and outputs runs one record at a time to the output stream. Buffering is used so that disk I/O is performed one block at a time. A block of records is initially read and held in the input buffer. Replacement selection removes records from the input buffer one at a time until the buffer is empty. At this point the next block of records is read in. Output to a buffer is similar: Once the buffer fills up it is written to disk as a unit. This process is illustrated by Figure 8.7.

Replacement selection works as follows. Assume that the main processing is done in an array of size M records.

1. Fill the array from disk. Set $\text{LAST} = M - 1$.
2. Build a min-heap. (Recall that a min-heap is defined such that the record at each node has a key value *less* than the key values of its children.)
3. Repeat until the array is empty:
 - (a) Send the record with the minimum key value (the root) to the output buffer.

- (b) Let R be the next record in the input buffer. If R 's key value is greater than the key value just output ...
 - i. Then place R at the root.
 - ii. Else replace the root with the record in array position LAST, and place R at position LAST. Set $\text{LAST} = \text{LAST} - 1$.
- (c) Sift down the root to reorder the heap.

When the test at step 3(b) is successful, a new record is added to the heap, eventually to be output as part of the run. As long as records coming from the input file have key values greater than the last key value output to the run, they can be safely added to the heap. Records with smaller key values cannot be output as part of the current run because they would not be in sorted order. Such values must be stored somewhere for future processing as part of another run. However, because the heap will shrink by one element in this case, there is now a free space where the last element of the heap used to be! Thus, replacement selection will slowly shrink the heap and at the same time use the discarded heap space to store records for the next run. Once the first run is complete (i.e., the heap becomes empty), the array will be filled with records ready to be processed for the second run. Figure 8.8 illustrates part of a run being created by replacement selection.

It should be clear that the minimum length of a run will be M records if the size of the heap is M , because at least those records originally in the heap will be part of the run. Under good conditions (e.g., if the input is sorted), then an arbitrarily long run is possible. In fact, the entire file could be processed as one run. If conditions are bad (e.g., if the input is reverse sorted), then runs of only size M result.

What is the expected length of a run generated by replacement selection? It can be deduced from an analogy called the **snowplow argument**. Imagine that a snowplow is going around a circular track during a heavy, but steady, snowstorm. After the plow has been around at least once, snow on the track must be as follows. Immediately behind the plow, the track is empty because it was just plowed. The greatest level of snow on the track is immediately in front of the plow, because this is the place least recently plowed. At any instant, there is a certain amount of snow S on the track. Snow is constantly falling throughout the track at a steady rate, with some snow falling “in front” of the plow and some “behind” the plow. (On a circular track, everything is actually “in front” of the plow, but Figure 8.9 illustrates the idea.) During the next revolution of the plow, all snow S on the track is removed, plus half of what falls. Because everything is assumed to be in steady state, after one revolution S snow is still on the track, so $2S$ snow must fall during a revolution, and $2S$ snow is removed during a revolution (leaving S snow behind).

At the beginning of replacement selection, nearly all values coming from the input file are greater (i.e., “in front of the plow”) than the latest key value output for this run, because the run's initial key values should be small. As the run progresses,

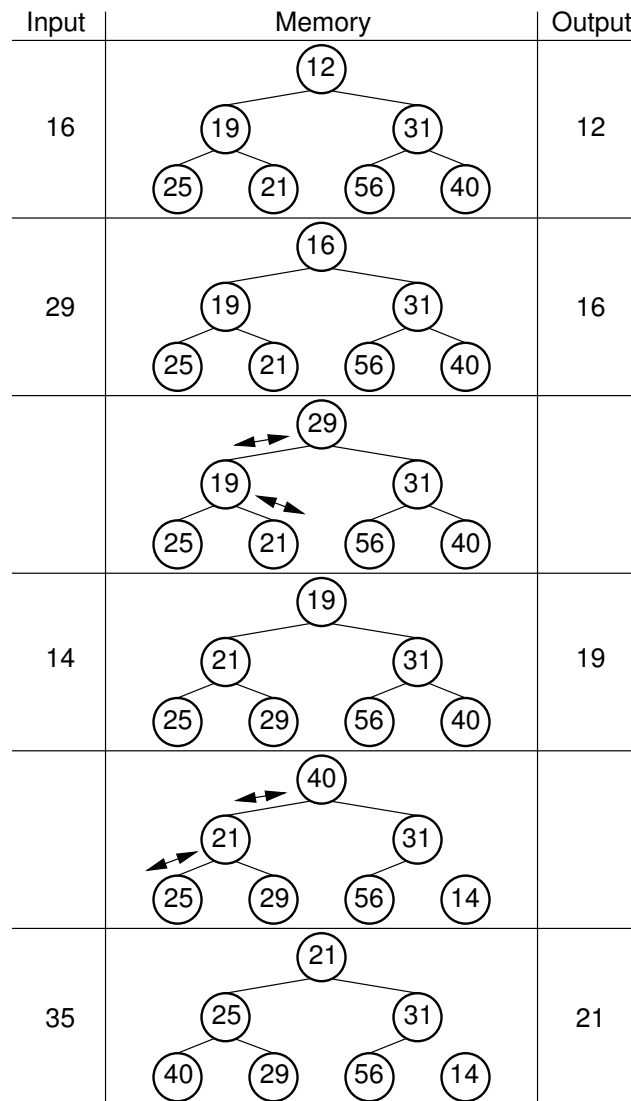


Figure 8.8 Replacement selection example. After building the heap, root value 12 is output and incoming value 16 replaces it. Value 16 is output next, replaced with incoming value 29. The heap is reordered, with 19 rising to the root. Value 19 is output next. Incoming value 14 is too small for this run and is placed at end of the array, moving value 40 to the root. Reordering the heap results in 21 rising to the root, which is output next.

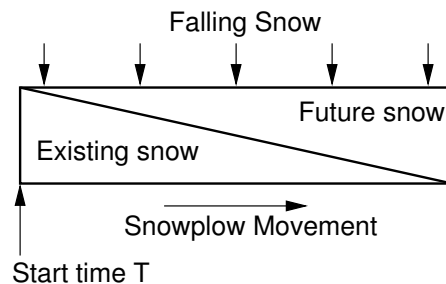


Figure 8.9 The snowplow analogy showing the action during one revolution of the snowplow. A circular track is laid out straight for purposes of illustration, and is shown in cross section. At any time T , the most snow is directly in front of the snowplow. As the plow moves around the track, the same amount of snow is always in front of the plow. As the plow moves forward, less of this is snow that was in the track at time T ; more is snow that has fallen since.

the latest key value output becomes greater and so new key values coming from the input file are more likely to be too small (i.e., “after the plow”); such records go to the bottom of the array. The total length of the run is expected to be twice the size of the array. Of course, this assumes that incoming key values are evenly distributed within the key range (in terms of the snowplow analogy, we assume that snow falls evenly throughout the track). Sorted and reverse sorted inputs do not meet this expectation and so change the length of the run.

8.5.3 Multiway Merging

The second stage of a typical external sorting algorithm merges the runs created by the first stage. Assume that we have R runs to merge. If a simple two-way merge is used, then R runs (regardless of their sizes) will require $\log R$ passes through the file. While R should be much less than the total number of records (because the initial runs should each contain many records), we would like to reduce still further the number of passes required to merge the runs together. Note that two-way merging does not make good use of available memory. Because merging is a sequential process on the two runs, only one block of records per run need be in memory at a time. Keeping more than one block of a run in memory at any time will not reduce the disk I/O required by the merge process (though if several blocks are read from a file at once time, at least they take advantage of sequential access). Thus, most of the space just used by the heap for replacement selection (typically many blocks in length) is not being used by the merge process.

We can make better use of this space and at the same time greatly reduce the number of passes needed to merge the runs if we merge several runs at a time. Multiway merging is similar to two-way merging. If we have B runs to merge, with a block from each run available in memory, then the B -way merge algorithm

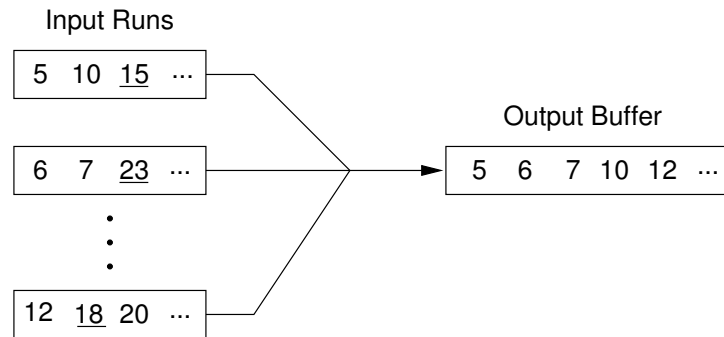


Figure 8.10 Illustration of multiway merge. The first value in each input run is examined and the smallest sent to the output. This value is removed from the input and the process repeated. In this example, values 5, 6, and 12 are compared first. Value 5 is removed from the first run and sent to the output. Values 10, 6, and 12 will be compared next. After the first five values have been output, the “current” value of each block is the one underlined.

simply looks at B values (the front-most value for each input run) and selects the smallest one to output. This value is removed from its run, and the process is repeated. When the current block for any run is exhausted, the next block from that run is read from disk. Figure 8.10 illustrates a multiway merge.

Conceptually, multiway merge assumes that each run is stored in a separate file. However, this is not necessary in practice. We only need to know the position of each run within a single file, and use `seekg` to move to the appropriate block whenever we need new data from a particular run. Naturally, this approach destroys the ability to do sequential processing on the input file. However, if all runs were stored on a single disk drive, then processing would not be truly sequential anyway because the I/O head would be alternating between the runs. Thus, multiway merging replaces several (potentially) sequential passes with a single random access pass. If the processing would not be sequential anyway (such as when all processing is on a single disk drive), no time is lost by doing so.

Multiway merging can greatly reduce the number of passes required. If there is room in memory to store one block for each run, then all runs can be merged in a single pass. Thus, replacement selection can build initial runs in one pass, and multiway merging can merge all runs in one pass, yielding a total cost of two passes. However, for truly large files, there might be too many runs for each to get a block in memory. If there is room to allocate B blocks for a B -way merge, and the number of runs R is greater than B , then it will be necessary to do multiple merge passes. In other words, the first B runs are merged, then the next B , and so on. These super-runs are then merged by subsequent passes, B super-runs at a time.

How big a file can be merged in one pass? Assuming B blocks were allocated to the heap for replacement selection (resulting in runs of average length $2B$ blocks), followed by a B -way merge, we can process on average a file of size $2B^2$ blocks in a single multiway merge. $2B^{k+1}$ blocks on average can be processed in k B -way merges. To gain some appreciation for how quickly this grows, assume that we have available 0.5MB of working memory, and that a block is 4KB, yielding 128 blocks in working memory. The average run size is 1MB (twice the working memory size). In one pass, 128 runs can be merged. Thus, a file of size 128MB can, on average, be processed in two passes (one to build the runs, one to do the merge) with only 0.5MB of working memory. As another example, assume blocks are 1KB long and working memory is 1MB = 1024 blocks. Then 1024 runs of average length 2MB (which is about 2GB) can be combined in a single merge pass. A larger block size would reduce the size of the file that can be processed in one merge pass for a fixed-size working memory; a smaller block size or larger working memory would increase the file size that can be processed in one merge pass. Two merge passes allow much bigger files to be processed. With 0.5MB of working memory and 4KB blocks, a file of size 16 gigabytes could be processed in two merge passes, which is big enough for most applications. Thus, this is a very effective algorithm for single disk drive external sorting.

Figure 8.11 shows a comparison of the running time to sort various-sized files for the following implementations: (1) standard Mergesort with two input runs and two output runs, (2) two-way Mergesort with large initial runs (limited by the size of available memory), and (3) R -way Mergesort performed after generating large initial runs. In each case, the file was composed of a series of four-byte records (a two-byte key and a two-byte data value), or 256K records per megabyte of file size. We can see from this table that using even a modest memory size (two blocks) to create initial runs results in a tremendous savings in time. Doing 4-way merges of the runs provides another considerable speedup, however large-scale multi-way merges for R beyond about 4 or 8 runs does not help much because a lot of time is spent determining which is the next smallest element among the R runs.

We see from this experiment that building large initial runs reduces the running time to slightly more than one third that of standard Mergesort, depending on file and memory sizes. Using a multiway merge further cuts the time nearly in half.

In summary, a good external sorting algorithm will seek to do the following:

- Make the initial runs as long as possible.
- At all stages, overlap input, processing, and output as much as possible.
- Use as much working memory as possible. Applying more memory usually speeds processing. In fact, more memory will have a greater effect than a faster disk. A faster CPU is unlikely to yield much improvement in running time for external sorting, because disk I/O speed is the limiting factor.

File Size (Mb)	Sort 1	Sort 2				Sort 3		
		Memory size (in blocks)				Memory size (in blocks)		
		2	4	16	256	2	4	16
1	0.61	0.27	0.24	0.19	0.10	0.21	0.15	0.13
	4,864	2,048	1,792	1,280	256	2,048	1,024	512
4	2.56	1.30	1.19	0.96	0.61	1.15	0.68	0.66*
	21,504	10,240	9,216	7,168	3,072	10,240	5,120	2,048
16	11.28	6.12	5.63	4.78	3.36	5.42	3.19	3.10
	94,208	49,152	45,056	36,864	20,480	49,152	24,516	12,288
256	220.39	132.47	123.68	110.01	86.66	115.73	69.31	68.71
	1,769K	1,048K	983K	852K	589K	1,049K	524K	262K

Figure 8.11 A comparison of three external sorts on a collection of small records for files of various sizes. Each entry in the table shows time in seconds and total number of blocks read and written by the program. File sizes are in Megabytes. For the third sorting algorithm, on a file size of 4MB, the time and blocks shown in the last column are for a 32-way merge (marked with an asterisk). 32 is used instead of 16 because 32 is a root of the number of blocks in the file (while 16 is not), thus allowing the same number of runs to be merged at every pass.

- If possible, use additional disk drives for more overlapping of processing with I/O, and to allow for sequential file processing.

8.6 Further Reading

A good general text on file processing is Folk and Zoellick's *File Structures: A Conceptual Toolkit* [FZ98]. A somewhat more advanced discussion on key issues in file processing is Betty Salzberg's *File Structures: An Analytical Approach* [Sal88]. A great discussion on external sorting methods can be found in Salzberg's book. The presentation in this chapter is similar in spirit to Salzberg's.

For details on disk drive modeling and measurement, see the article by Ruemmler and Wilkes, "An Introduction to Disk Drive Modeling" [RW94]. See Andrew S. Tanenbaum's *Structured Computer Organization* [Tan06] for an introduction to computer hardware and organization. An excellent, detailed description of memory and hard disk drives can be found online at "The PC Guide," by Charles M. Kozierok [Koz05] (www.pcguide.com). The PC Guide also gives detailed descriptions of the Microsoft Windows and UNIX (Linux) file systems.

See "Outperforming LRU with an Adaptive Replacement Cache Algorithm" by Megiddo and Modha [MM04] for an example of a more sophisticated algorithm than LRU for managing buffer pools.

The snowplow argument comes from Donald E. Knuth's *Sorting and Searching* [Knu98], which also contains a wide variety of external sorting algorithms.

8.7 Exercises

- 8.1** Computer memory and storage prices change rapidly. Find out what the current prices are for the media listed in Figure 8.1. Does your information change any of the basic conclusions regarding disk processing?
- 8.2** Assume a disk drive from the late 1990s is configured as follows. The total storage is approximately 675MB divided among 15 surfaces. Each surface has 612 tracks; there are 144 sectors/track, 512 bytes/sector, and 8 sectors/cluster. The disk turns at 3600 rpm. The track-to-track seek time is 20 ms, and the average seek time is 80 ms. Now assume that there is a 360KB file on the disk. On average, how long does it take to read all of the data in the file? Assume that the first track of the file is randomly placed on the disk, that the entire file lies on adjacent tracks, and that the file completely fills each track on which it is found. A seek must be performed each time the I/O head moves to a new track. Show your calculations.
- 8.3** Using the specifications for the disk drive given in Exercise 8.2, calculate the expected time to read one entire track, one sector, and one byte. Show your calculations.
- 8.4** Using the disk drive specifications given in Exercise 8.2, calculate the time required to read a 10MB file assuming
- (a) The file is stored on a series of contiguous tracks, as few tracks as possible.
 - (b) The file is spread randomly across the disk in 4KB clusters.
- Show your calculations.
- 8.5** Assume that a disk drive is configured as follows. The total storage is approximately 1033MB divided among 15 surfaces. Each surface has 2100 tracks, there are 64 sectors/track, 512 bytes/sector, and 8 sectors/cluster. The disk turns at 7200 rpm. The track-to-track seek time is 3 ms, and the average seek time is 20 ms. Now assume that there is a 512KB file on the disk. On average, how long does it take to read all of the data on the file? Assume that the first track of the file is randomly placed on the disk, that the entire file lies on contiguous tracks, and that the file completely fills each track on which it is found. Show your calculations.
- 8.6** Using the specifications for the disk drive given in Exercise 8.5, calculate the expected time to read one entire track, one sector, and one byte. Show your calculations.
- 8.7** Using the disk drive specifications given in Exercise 8.5, calculate the time required to read a 10MB file assuming
- (a) The file is stored on a series of contiguous tracks, as few tracks as possible.
 - (b) The file is spread randomly across the disk in 4KB clusters.

Show your calculations.

8.8 A typical disk drive from 2004 has the following specifications.³ The total storage is approximately 120GB on 6 platter surfaces or 20GB/platter. Each platter has 16K tracks with 2560 sectors/track (a sector holds 512 bytes) and 16 sectors/cluster. The disk turns at 7200 rpm. The track-to-track seek time is 2.0 ms, and the average seek time is 10.0 ms. Now assume that there is a 6MB file on the disk. On average, how long does it take to read all of the data on the file? Assume that the first track of the file is randomly placed on the disk, that the entire file lies on contiguous tracks, and that the file completely fills each track on which it is found. Show your calculations.

8.9 Using the specifications for the disk drive given in Exercise 8.8, calculate the expected time to read one entire track, one sector, and one byte. Show your calculations.

8.10 Using the disk drive specifications given in Exercise 8.8, calculate the time required to read a 10MB file assuming

(a) The file is stored on a series of contiguous tracks, as few tracks as possible.

(b) The file is spread randomly across the disk in 8KB clusters.

Show your calculations.

8.11 At the end of 2004, the fastest disk drive I could find specifications for was the Maxtor Atlas. This drive had a nominal capacity of 73.4GB using 4 platters (8 surfaces) or 9.175GB/surface. Assume there are 16,384 tracks with an average of 1170 sectors/track and 512 bytes/sector.⁴ The disk turns at 15,000 rpm. The track-to-track seek time is 0.4 ms and the average seek time is 3.6 ms. How long will it take on average to read a 6MB file, assuming that the first track of the file is randomly placed on the disk, that the entire file lies on contiguous tracks, and that the file completely fills each track on which it is found. Show your calculations.

8.12 Using the specifications for the disk drive given in Exercise 8.11, calculate the expected time to read one entire track, one sector, and one byte. Show your calculations.

8.13 Using the disk drive specifications given in Exercise 8.11, calculate the time required to read a 10MB file assuming

(a) The file is stored on a series of contiguous tracks, as few tracks as possible.

³To make the exercise doable, this specification is completely fictitious with respect to the track and sector layout. While sectors do have 512 bytes, and while the number of platters and amount of data per track is plausible, the reality is that all modern drives use a zoned organization to keep the data density from inside to outside of the disk reasonably high. The rest of the numbers are typical for a drive from 2004.

⁴Again, this track layout does not account for the zoned arrangement on modern disk drives.

(b) The file is spread randomly across the disk in 8KB clusters.

Show your calculations.

- 8.14** Prove that two tracks selected at random from a disk are separated on average by one third the number of tracks on the disk.
- 8.15** Assume that a file contains one million records sorted by key value. A query to the file returns a single record containing the requested key value. Files are stored on disk in sectors each containing 100 records. Assume that the average time to read a sector selected at random is 10.0 ms. In contrast, it takes only 2.0 ms to read the sector adjacent to the current position of the I/O head. The “batch” algorithm for processing queries is to first sort the queries by order of appearance in the file, and then read the entire file sequentially, processing all queries in sequential order as the file is read. This algorithm implies that the queries must all be available before processing begins. The “interactive” algorithm is to process each query in order of its arrival, searching for the requested sector each time (unless by chance two queries in a row are to the same sector). Carefully define under what conditions the batch method is more efficient than the interactive method.
- 8.16** Assume that a virtual memory is managed using a buffer pool. The buffer pool contains five buffers and each buffer stores one block of data. Memory accesses are by block ID. Assume the following series of memory accesses takes place:

5 2 5 12 3 6 5 9 3 2 4 1 5 9 8 15 3 7 2 5 9 10 4 6 8 5

For each of the following buffer pool replacement strategies, show the contents of the buffer pool at the end of the series, and indicate how many times a block was found in the buffer pool (instead of being read into memory). Assume that the buffer pool is initially empty.

- (a) First-in, first out.
 - (b) Least frequently used (with counts kept only for blocks currently in memory, counts for a page are lost when that page is removed, and the oldest item with the smallest count is removed when there is a tie).
 - (c) Least frequently used (with counts kept for all blocks, and the oldest item with the smallest count is removed when there is a tie).
 - (d) Least recently used.
 - (e) Most recently used (replace the block that was most recently accessed).
- 8.17** Suppose that a record is 32 bytes, a block is 1024 bytes (thus, there are 32 records per block), and that working memory is 1MB (there is also additional space available for I/O buffers, program variables, etc.). What is the *expected* size for the largest file that can be merged using replacement selection followed by a *single* pass of multiway merge? Explain how you got your answer.

- 8.18** Assume that working memory size is 256KB broken into blocks of 8192 bytes (there is also additional space available for I/O buffers, program variables, etc.). What is the *expected* size for the largest file that can be merged using replacement selection followed by *two* passes of multiway merge? Explain how you got your answer.
- 8.19** Prove or disprove the following proposition: Given space in memory for a heap of M records, replacement selection will completely sort a file if no record in the file is preceded by M or more keys of greater value.
- 8.20** Imagine a database containing ten million records, with each record being 100 bytes long. Provide an estimate of the time it would take (in seconds) to sort the database on a typical desktop or laptop computer.
- 8.21** Assume that a company has a computer configuration satisfactory for processing their monthly payroll. Further assume that the bottleneck in payroll processing is a sorting operation on all of the employee records, and that an external sorting algorithm is used. The company's payroll program is so good that it plans to hire out its services to do payroll processing for other companies. The president has an offer from a second company with 100 times as many employees. She realizes that her computer is not up to the job of sorting 100 times as many records in an acceptable amount of time. Describe what impact each of the following modifications to the computing system is likely to have in terms of reducing the time required to process the larger payroll database.
- (a) A factor of two speedup to the CPU.
 - (b) A factor of two speedup to disk I/O time.
 - (c) A factor of two speedup to main memory access time.
 - (d) A factor of two increase to main memory size.
- 8.22** How can the external sorting algorithm described in this chapter be extended to handle variable-length records?

8.8 Projects

- 8.1** For a database application, assume it takes 10 ms to read a block from disk, 1 ms to search for a record in a block stored in memory, and that there is room in memory for a buffer pool of 5 blocks. Requests come in for records, with the request specifying which block contains the record. If a block is accessed, there is a 10% probability for each of the next ten requests that the request will be to the same block. What will be the expected performance improvement for each of the following modifications to the system?
- (a) Get a CPU that is twice as fast.
 - (b) Get a disk drive that is twice as fast.

(c) Get enough memory to double the buffer pool size.

Write a simulation to analyze this problem.

- 8.2** Pictures are typically stored as an array, row by row, on disk. Consider the case where the picture has 16 colors. Thus, each pixel can be represented using 4 bits. If you allow 8 bits per pixel, no processing is required to unpack the pixels (because a pixel corresponds to a byte, the lowest level of addressing on most machines). If you pack two pixels per byte, space is saved but the pixels must be unpacked. Which takes more time to read from disk and access every pixel of the image: 8 bits per pixel, or 4 bits per pixel with 2 pixels per byte? Program both and compare the times.
- 8.3** Implement a disk-based buffer pool class based on the LRU buffer pool replacement strategy. Disk blocks are numbered consecutively from the beginning of the file with the first block numbered as 0. Assume that blocks are 4096 bytes in size, with the first 4 bytes used to store the block ID corresponding to that buffer. Use the first **BufferPool** abstract class given in Section 8.3 as the basis for your implementation.
- 8.4** Implement an external sort based on replacement selection and multiway merging as described in this chapter. Test your program both on files with small records and on files with large records. For what size record do you find that key sorting would be worthwhile?
- 8.5** Implement a Quicksort for large files on disk by replacing all array access in the normal Quicksort application with access to a virtual array implemented using a buffer pool. That is, whenever a record in the array would be read or written by Quicksort, use a call to a buffer pool function instead. Compare the running time of this implementation with implementations for external sorting based on mergesort as described in this chapter.
- 8.6** Section 8.5.1 suggests that an easy modification to the basic 2-way mergesort is to read in a large chunk of data into main memory, sort it with Quicksort, and write it out for initial runs. Then, a standard 2-way merge is used in a series of passes to merge the runs together. However, this makes use of only two blocks of working memory at a time. Each block read is essentially random access, because the various files are read in an unknown order, even though each of the input and output files is processed sequentially on each pass. A possible improvement would be, on the merge passes, to divide working memory into four equal sections. One section is allocated to each of the two input files and two output files. All reads during merge passes would be in full sections, rather than single blocks. While the total number of blocks read and written would be the same as a regular 2-way Mergesort, it is possible that this would speed processing because a series of blocks that are logically adjacent in the various input and output files would be read/written each time. Implement this variation, and compare its running time against

a standard series of 2-way merge passes that read/write only a single block at a time. Before beginning implementation, write down your hypothesis on how the running time will be affected by this change. After implementing, did you find that this change has any meaningful effect on performance?

Searching

Organizing and retrieving information is at the heart of most computer applications, and searching is surely the most frequently performed of all computing tasks. Search can be viewed abstractly as a process to determine if an element with a particular value is a member of a particular set. The more common view of searching is an attempt to find the record within a collection of records that has a particular key value, or those records in a collection whose key values meet some criterion such as falling within a range of values.

We can define searching formally as follows. Suppose that we have a collection **L** of n records of the form

$$(k_1, I_1), (k_2, I_2), \dots, (k_n, I_n)$$

where I_j is information associated with key k_j from record j for $1 \leq j \leq n$. Given a particular key value K , the **search problem** is to locate a record (k_j, I_j) in **L** such that $k_j = K$ (if one exists). **Searching** is a systematic method for locating the record (or records) with key value $k_j = K$.

A **successful** search is one in which a record with key $k_j = K$ is found. An **unsuccessful** search is one in which no record with $k_j = K$ is found (and no such record exists).

An **exact-match query** is a search for the record whose key value matches a specified key value. A **range query** is a search for all records whose key value falls within a specified range of key values.

We can categorize search algorithms into three general approaches:

1. Sequential and list methods.
2. Direct access by key value (hashing).
3. Tree indexing methods.

This and the following chapter treat these three approaches in turn. Any of these approaches are potentially suitable for implementing the Dictionary ADT

introduced in Section 4.4. However, each has different performance characteristics that make it the method of choice in particular circumstances.

The current chapter considers methods for searching data stored in lists. List in this context means any list implementation including a linked list or an array. Most of these methods are appropriate for sequences (i.e., duplicate key values are allowed), although special techniques applicable to sets are discussed in Section 9.3. The techniques from the first three sections of this chapter are most appropriate for searching a collection of records stored in RAM. Section 9.4 discusses hashing, a technique for organizing data in an array such that the location of each record within the array is a function of its key value. Hashing is appropriate when records are stored either in RAM or on disk.

Chapter 10 discusses tree-based methods for organizing information on disk, including a commonly used file structure called the B-tree. Nearly all programs that must organize large collections of records stored on disk use some variant of either hashing or the B-tree. Hashing is practical for only certain access functions (exact-match queries) and is generally appropriate only when duplicate key values are not allowed. B-trees are the method of choice for dynamic disk-based applications anytime hashing is not appropriate.

9.1 Searching Unsorted and Sorted Arrays

The simplest form of search has already been presented in Example 3.1: the sequential search algorithm. Sequential search on an unsorted list requires $\Theta(n)$ time in the worst case.

How many comparisons does linear search do on average? A major consideration is whether K is in list \mathbf{L} at all. We can simplify our analysis by ignoring everything about the input except the position of K if it is found in \mathbf{L} . Thus, we have $n + 1$ distinct possible events: That K is in one of positions 0 to $n - 1$ in \mathbf{L} (each position having its own probability), or that it is not in \mathbf{L} at all. We can express the probability that K is not in \mathbf{L} as

$$\mathbf{P}(K \notin \mathbf{L}) = 1 - \sum_{i=1}^n \mathbf{P}(K = \mathbf{L}[i])$$

where $\mathbf{P}(x)$ is the probability of event x .

Let p_i be the probability that K is in position i of \mathbf{L} (indexed from 0 to $n - 1$). For any position i in the list, we must look at $i + 1$ records to reach it. So we say that the cost when K is in position i is $i + 1$. When K is not in \mathbf{L} , sequential search will require n comparisons. Let p_n be the probability that K is not in \mathbf{L} . Then the average cost $\mathbf{T}(n)$ will be

$$\mathbf{T}(n) = np_n + \sum_{i=0}^{n-1} (i+1)p_i.$$

What happens to the equation if we assume all the p_i 's are equal (except p_0)?

$$\begin{aligned} \mathbf{T}(n) &= p_n n + \sum_{i=0}^{n-1} (i+1)p \\ &= p_n n + p \sum_{i=1}^n i \\ &= p_n n + p \frac{n(n+1)}{2} \\ &= p_n n + \frac{1-p_n}{n} \frac{n(n+1)}{2} \\ &= \frac{n+1+p_n(n-1)}{2} \end{aligned}$$

Depending on the value of p_n , $\frac{n+1}{2} \leq \mathbf{T}(n) \leq n$.

For large collections of records that are searched repeatedly, sequential search is unacceptably slow. One way to reduce search time is to preprocess the records by sorting them. Given a sorted array, an obvious improvement over simple linear search is to test if the current element in \mathbf{L} is greater than K . If it is, then we know that K cannot appear later in the array, and we can quit the search early. But this still does not improve the worst-case cost of the algorithm.

We can also observe that if we look first at position 1 in sorted array \mathbf{L} and find that K is bigger, then we rule out position 0 as well as position 1. Because more is often better, what if we look at position 2 in \mathbf{L} and find that K is bigger yet? This rules out positions 0, 1, and 2 with one comparison. What if we carry this to the extreme and look first at the last position in \mathbf{L} and find that K is bigger? Then we know in one comparison that K is not in \mathbf{L} . This is very useful to know, but what is wrong with the conclusion that we should always start by looking at the last position? The problem is that, while we learn a lot sometimes (in one comparison we might learn that K is not in the list), usually we learn only a little bit (that the last element is not K).

The question then becomes: What is the right amount to jump? This leads us to an algorithm known as **Jump Search**. For some value j , we check every j 'th element in \mathbf{L} , that is, we check elements $\mathbf{L}[j]$, $\mathbf{L}[2j]$, and so on. So long as K is greater than the values we are checking, we continue on. But when we reach a

value in \mathbf{L} greater than K , we do a linear search on the piece of length $j - 1$ that we know brackets K if it is in the list.

If we define m such that $mj \leq n < (m + 1)j$, then the total cost of this algorithm is at most $m + j - 1$ 3-way comparisons. (They are 3-way because at each comparison of K with some $\mathbf{L}[i]$ we need to know if K is less than, equal to, or greater than $\mathbf{L}[i]$.) Therefore, the cost to run the algorithm on n items with a jump of size j is

$$\mathbf{T}(n, j) = m + j - 1 = \left\lfloor \frac{n}{j} \right\rfloor + j - 1.$$

What is the best value that we can pick for j ? We want to minimize the cost:

$$\min_{1 \leq j \leq n} \left\{ \left\lfloor \frac{n}{j} \right\rfloor + j - 1 \right\}$$

Take the derivative and solve for $f'(j) = 0$ to find the minimum, which is $j = \sqrt{n}$. In this case, the worst case cost will be roughly $2\sqrt{n}$.

This example invokes a basic principle of algorithm design. We want to balance the work done while selecting a sublist with the work done while searching a sublist. In general, it is a good strategy to make subproblems of equal effort. This is an example of a **divide and conquer** algorithm.

What if we extend this idea to three levels? We would first make jumps of some size j to find a sublist of size $j - 1$ whose end values bracket value K . We would then work through this sublist by making jumps of some smaller size, say j_1 . Finally, once we find a bracketed sublist of size $j_1 - 1$, we would do sequential search to complete the process.

This probably sounds convoluted to do two levels of jumping to be followed by a sequential search. While it might make sense to do a two-level algorithm (that is, jump search jumps to find a sublist and then does sequential search on the sublist), it almost never seems to make sense to do a three-level algorithm. Instead, when we go beyond two levels, we nearly always generalize by using recursion. This leads us to the most commonly used search algorithm for sorted arrays, the binary search described in Section 3.5.

If we know nothing about the distribution of key values, then binary search is the best algorithm available for searching a sorted array (see Exercise 9.22). However, sometimes we do know something about the expected key distribution. Consider the typical behavior of a person looking up a word in a large dictionary. Most people certainly do not use sequential search! Typically, people use a modified form of binary search, at least until they get close to the word that they are looking for. The search generally does not start at the middle of the dictionary. A person looking for a word starting with 'S' generally assumes that entries beginning with 'S' start about three quarters of the way through the dictionary. Thus, he or

she will first open the dictionary about three quarters of the way through and then make a decision based on what is found as to where to look next. In other words, people typically use some knowledge about the expected distribution of key values to “compute” where to look next. This form of “computed” binary search is called a **dictionary search** or **interpolation search**. In a dictionary search, we search \mathbf{L} at a position p that is appropriate to the value of K as follows.

$$p = \frac{K - \mathbf{L}[1]}{\mathbf{L}[n] - \mathbf{L}[1]}$$

This equation is computing the position of K as a fraction of the distance between the smallest and largest key values. This will next be translated into that position which is the same fraction of the way through the array, and this position is checked first. As with binary search, the value of the key found eliminates all records either above or below that position. The actual value of the key found can then be used to compute a new position within the remaining range of the array. The next check is made based on the new computation. This proceeds until either the desired record is found, or the array is narrowed until no records are left.

A variation on dictionary search is known as **Quadratic Binary Search** (QBS), and we will analyze this in detail because its analysis is easier than that of the general dictionary search. QBS will first compute p and then examine $\mathbf{L}[\lceil pn \rceil]$. If $K < \mathbf{L}[\lceil pn \rceil]$ then QBS will sequentially probe to the left by steps of size \sqrt{n} , that is, we step through

$$\mathbf{L}[\lceil pn - i\sqrt{n} \rceil], i = 1, 2, 3, \dots$$

until we reach a value less than or equal to K . Similarly for $K > \mathbf{L}[\lceil pn \rceil]$ we will step to the right by \sqrt{n} until we reach a value in \mathbf{L} that is greater than K . We are now within \sqrt{n} positions of K . Assume (for now) that it takes a constant number of comparisons to bracket K within a sublist of size \sqrt{n} . We then take this sublist and repeat the process recursively. That is, at the next level we compute an interpolation to start somewhere in the subarray. We then step to the left or right (as appropriate) by steps of size $\sqrt{\sqrt{n}}$.

What is the cost for QBS? Note that $\sqrt{c^n} = c^{n/2}$, and we will be repeatedly taking square roots of the current sublist size until we find the item that we are looking for. Because $n = 2^{\log n}$ and we can cut $\log n$ in half only $\log \log n$ times, the cost is $\Theta(\log \log n)$ if the number of probes on jump search is constant.

Say that the number of comparisons needed is i , in which case the cost is i (since we have to do i comparisons). If \mathbf{P}_i is the probability of needing exactly i probes, then

$$\begin{aligned} & \sum_{i=1}^{\sqrt{n}} i \mathbf{P}(\text{need exactly } i \text{ probes}) \\ &= 1\mathbf{P}_1 + 2\mathbf{P}_2 + 3\mathbf{P}_3 + \dots + \sqrt{n}\mathbf{P}_{\sqrt{n}} \end{aligned}$$

We now show that this is the same as

$$\begin{aligned}
 & \sum_{i=1}^{\sqrt{n}} \mathbf{P}(\text{need at least } i \text{ probes}) \\
 &= 1 + (1 - \mathbf{P}_1) + (1 - \mathbf{P}_1 - \mathbf{P}_2) + \cdots + \mathbf{P}_{\sqrt{n}} \\
 &= (\mathbf{P}_1 + \cdots + \mathbf{P}_{\sqrt{n}}) + (\mathbf{P}_2 + \cdots + \mathbf{P}_{\sqrt{n}}) + \\
 &\quad (\mathbf{P}_3 + \cdots + \mathbf{P}_{\sqrt{n}}) + \cdots \\
 &= 1\mathbf{P}_1 + 2\mathbf{P}_2 + 3\mathbf{P}_3 + \cdots + \sqrt{n}\mathbf{P}_{\sqrt{n}}
 \end{aligned}$$

We require at least two probes to set the bounds, so the cost is

$$2 + \sum_{i=3}^{\sqrt{n}} \mathbf{P}(\text{need at least } i \text{ probes}).$$

We now make take advantage of a useful fact known as Čebyšev's Inequality. Čebyšev's inequality states that $\mathbf{P}(\text{need exactly } i \text{ probes})$, or \mathbf{P}_i , is

$$\mathbf{P}_i \leq \frac{p(1-p)n}{(i-2)^2n} \leq \frac{1}{4(i-2)^2}$$

because $p(1-p) \leq 1/4$ for any probability p . This assumes uniformly distributed data. Thus, the expected number of probes is

$$2 + \sum_{i=3}^{\sqrt{n}} \frac{1}{4(i-2)^2} < 2 + \frac{1}{4} \sum_{i=1}^{\infty} \frac{1}{i^2} = 2 + \frac{1}{4} \frac{\pi}{6} \approx 2.4112$$

Is QBS better than binary search? Theoretically yes, because $O(\log \log n)$ grows slower than $O(\log n)$. However, we have a situation here which illustrates the limits to the model of asymptotic complexity in some practical situations. Yes, $c_1 \log n$ does grow faster than $c_2 \log \log n$. In fact, it is exponentially faster! But even so, for practical input sizes, the absolute cost difference is fairly small. Thus, the constant factors might play a role. First we compare $\lg \lg n$ to $\lg n$.

n	$\lg n$	$\lg \lg n$	Factor
			Difference
16	4	2	2
256	8	3	2.7
2^{16}	16	4	4
2^{32}	32	5	6.4

It is not always practical to reduce an algorithm's growth rate. There is a "practicality window" for every problem, in that we have a practical limit to how big an input we wish to solve for. If our problem size never grows too big, it might not matter if we can reduce the cost by an extra log factor, because the constant factors in the two algorithms might differ by more than the log of the log of the input size.

For our two algorithms, let us look further and check the actual number of comparisons used. For binary search, we need about $\lg n - 1$ total comparisons. Quadratic binary search requires about $2.4 \lg \lg n$ comparisons. If we incorporate this observation into our table, we get a different picture about the relative differences.

n	$\lg n - 1$	$2.4 \lg \lg n$	Factor Difference
16	3	4.8	worse
256	7	7.2	\approx same
64K	15	9.6	1.6
2^{32}	31	12	2.6

But we still are not done. This is only a count of raw comparisons. Binary search is inherently much simpler than QBS, because binary search only needs to calculate the midpoint position of the array before each comparison, while quadratic binary search must calculate an interpolation point which is more expensive. So the constant factors for QBS are even higher.

Not only are the constant factors worse on average, but QBS is far more dependent than binary search on good data distribution to perform well. For example, imagine that you are searching a telephone directory for the name "Young." Normally you would look near the back of the book. If you found a name beginning with 'Z,' you might look just a little ways toward the front. If the next name you find also begins with 'Z,' you would look a little further toward the front. If this particular telephone directory were unusual in that half of the entries begin with 'Z,' then you would need to move toward the front many times, each time eliminating relatively few records from the search. In the extreme, the performance of interpolation search might not be much better than sequential search if the distribution of key values is badly calculated.

While it turns out that QBS is not a practical algorithm, this is not a typical situation. Fortunately, algorithm growth rates are usually well behaved, so that asymptotic algorithm analysis nearly always gives us a practical indication for which of two algorithms is better.

9.2 Self-Organizing Lists

While ordering of lists is most commonly done by key value, this is not the only viable option. Another approach to organizing lists to speed search is to order the

records by expected frequency of access. While the benefits might not be as great as when organized by key value, the cost to organize (at least approximately) by frequency of access can be much cheaper, and thus can speed up sequential search in some situations.

Assume that we know, for each key k_i , the probability p_i that the record with key k_i will be requested. Assume also that the list is ordered so that the most frequently requested record is first, then the next most frequently requested record, and so on. Search in the list will be done sequentially, beginning with the first position. Over the course of many searches, the expected number of comparisons required for one search is

$$\overline{C}_n = 1p_0 + 2p_1 + \dots + np_{n-1}.$$

In other words, the cost to access the record in $L[0]$ is 1 (because one key value is looked at), and the probability of this occurring is p_0 . The cost to access the record in $L[1]$ is 2 (because we must look at the first and the second records' key values), with probability p_1 , and so on. For n records, assuming that all searches are for records that actually exist, the probabilities p_0 through p_{n-1} must sum to one.

Certain probability distributions give easily computed results.

Example 9.1 Calculate the expected cost to search a list when each record has equal chance of being accessed (the classic sequential search through an unsorted list). Setting $p_i = 1/n$ yields

$$\overline{C}_n = \sum_{i=1}^n i/n = (n+1)/2.$$

This result matches our expectation that half the records will be accessed on average by normal sequential search. If the records truly have equal access probabilities, then ordering records by frequency yields no benefit. We saw in Section 9.1 the more general case where we must consider the probability (labeled p_n) that the search key does not match that for any record in the array. In that case, in accordance with our general formula, we get

$$(1-p_n)\frac{n+1}{2} + p_nn = \frac{n+1 - np_nn - p_n + 2p_n}{2} = \frac{n+1 + p_0(n-1)}{2}.$$

Thus, $\frac{n+1}{2} \leq \overline{C}_n \leq n$, depending on the value of p_0 .

A geometric probability distribution can yield quite different results.

Example 9.2 Calculate the expected cost for searching a list ordered by frequency when the probabilities are defined as

$$p_i = \begin{cases} 1/2^i & \text{if } 0 \leq i \leq n-2 \\ 1/2^n & \text{if } i = n-1. \end{cases}$$

Then,

$$\bar{C}_n \approx \sum_{i=0}^{n-1} (i+1)/2^{i+1} = \sum_{i=1}^n (i/2^i) \approx 2.$$

For this example, the expected number of accesses is a constant. This is because the probability for accessing the first record is high (one half), the second is much lower (one quarter) but still much higher than for the third record, and so on. This shows that for some probability distributions, ordering the list by frequency can yield an efficient search technique.

In many search applications, real access patterns follow a rule of thumb called the **80/20 rule**. The 80/20 rule says that 80% of the record accesses are to 20% of the records. The values of 80 and 20 are only estimates; every data access pattern has its own values. However, behavior of this nature occurs surprisingly often in practice (which explains the success of caching techniques widely used by web browsers for speeding access to web pages, and by disk drive and CPU manufacturers for speeding access to data stored in slower memory; see the discussion on buffer pools in Section 8.3). When the 80/20 rule applies, we can expect considerable improvements to search performance from a list ordered by frequency of access over standard sequential search in an unordered list.

Example 9.3 The 80/20 rule is an example of a **Zipf distribution**. Naturally occurring distributions often follow a Zipf distribution. Examples include the observed frequency for the use of words in a natural language such as English, and the size of the population for cities (i.e., view the relative proportions for the populations as equivalent to the “frequency of use”). Zipf distributions are related to the Harmonic Series defined in Equation 2.10. Define the Zipf frequency for item i in the distribution for n records as $1/(i\mathcal{H}_n)$ (see Exercise 9.4). The expected cost for the series whose members follow this Zipf distribution will be

$$\bar{C}_n = \sum_{i=1}^n i/i\mathcal{H}_n = n/\mathcal{H}_n \approx n/\log_e n.$$

When a frequency distribution follows the 80/20 rule, the average search looks at about 10-15% of the records in a list ordered by frequency.

This is potentially a useful observation that typical “real-life” distributions of record accesses, if the records were ordered by frequency, would require that we visit on average only 10-15% of the list when doing sequential search. This means that if we had an application that used sequential search, and we wanted to make it go a bit faster (by a constant amount), we could do so without a major rewrite to the system to implement something like a search tree. But that is only true if there is an easy way to (at least approximately) order the records by frequency.

In most applications, we have no means of knowing in advance the frequencies of access for the data records. To complicate matters further, certain records might be accessed frequently for a brief period of time, and then rarely thereafter. Thus, the probability of access for records might change over time (in most database systems, this is to be expected). **Self-organizing lists** seek to solve both of these problems.

Self-organizing lists modify the order of records within the list based on the actual pattern of record access. Self-organizing lists use a heuristic for deciding how to reorder the list. These heuristics are similar to the rules for managing buffer pools (see Section 8.3). In fact, a buffer pool is a form of self-organizing list. Ordering the buffer pool by expected frequency of access is a good strategy, because typically we must search the contents of the buffers to determine if the desired information is already in main memory. When ordered by frequency of access, the buffer at the end of the list will be the one most appropriate for reuse when a new page of information must be read. Below are three traditional heuristics for managing self-organizing lists:

1. The most obvious way to keep a list ordered by frequency would be to store a count of accesses to each record and always maintain records in this order. This method will be referred to as **count**. Count is similar to the least frequently used buffer replacement strategy. Whenever a record is accessed, it might move toward the front of the list if its number of accesses becomes greater than a record preceding it. Thus, count will store the records in the order of frequency that has actually occurred so far. Besides requiring space for the access counts, count does not react well to changing frequency of access over time. Once a record has been accessed a large number of times under the frequency count system, it will remain near the front of the list regardless of further access history.
2. Bring a record to the front of the list when it is found, pushing all the other records back one position. This is analogous to the least recently used buffer replacement strategy and is called **move-to-front**. This heuristic is easy to implement if the records are stored using a linked list. When records are stored in an array, bringing a record forward from near the end of the array will result in a large number of records (slightly) changing position. Move-to-front’s cost is bounded in the sense that it requires at most twice the num-

ber of accesses required by the **optimal static ordering** for n records when at least n searches are performed. In other words, if we had known the series of (at least n) searches in advance and had stored the records in order of frequency so as to minimize the total cost for these accesses, this cost would be at least half the cost required by the move-to-front heuristic. (This will be proved using amortized analysis in Section 14.3.) Finally, move-to-front responds well to local changes in frequency of access, in that if a record is frequently accessed for a brief period of time it will be near the front of the list during that period of access. Move-to-front does poorly when the records are processed in sequential order, especially if that sequential order is then repeated multiple times.

3. Swap any record found with the record immediately preceding it in the list. This heuristic is called **transpose**. Transpose is good for list implementations based on either linked lists or arrays. Frequently used records will, over time, move to the front of the list. Records that were once frequently accessed but are no longer used will slowly drift toward the back. Thus, it appears to have good properties with respect to changing frequency of access. Unfortunately, there are some pathological sequences of access that can make transpose perform poorly. Consider the case where the last record of the list (call it X) is accessed. This record is then swapped with the next-to-last record (call it Y), making Y the last record. If Y is now accessed, it swaps with X . A repeated series of accesses alternating between X and Y will continually search to the end of the list, because neither record will ever make progress toward the front. However, such pathological cases are unusual in practice. A variation on transpose would be to move the accessed record forward in the list by some fixed number of steps.

Example 9.4 Assume that we have eight records, with key values A to H , and that they are initially placed in alphabetical order. Now, consider the result of applying the following access pattern:

$$F D F G E G F A D F G E.$$

Assume that when a record's frequency count goes up, it moves forward in the list to become the last record with that value for its frequency count. After the first two accesses, F will be the first record and D will be the second. The final list resulting from these accesses will be

$$F G D E A B C H,$$

and the total cost for the twelve accesses will be 45 comparisons.

If the list is organized by the move-to-front heuristic, then the final list will be

$$E G F D A B C H,$$

and the total number of comparisons required is 54.

Finally, if the list is organized by the transpose heuristic, then the final list will be

$$A B F D G E C H,$$

and the total number of comparisons required is 62.

While self-organizing lists do not generally perform as well as search trees or a sorted list, both of which require $O(\log n)$ search time, there are many situations in which self-organizing lists prove a valuable tool. Obviously they have an advantage over sorted lists in that they need not be sorted. This means that the cost to insert a new record is low, which could more than make up for the higher search cost when insertions are frequent. Self-organizing lists are simpler to implement than search trees and are likely to be more efficient for small lists. Nor do they require additional space. Finally, in the case of an application where sequential search is “almost” fast enough, changing an unsorted list to a self-organizing list might speed the application enough at a minor cost in additional code.

As an example of applying self-organizing lists, consider an algorithm for compressing and transmitting messages. The list is self-organized by the move-to-front rule. Transmission is in the form of words and numbers, by the following rules:

1. If the word has been seen before, transmit the current position of the word in the list. Move the word to the front of the list.
2. If the word is seen for the first time, transmit the word. Place the word at the front of the list.

Both the sender and the receiver keep track of the position of words in the list in the same way (using the move-to-front rule), so they agree on the meaning of the numbers that encode repeated occurrences of words. Consider the following example message to be transmitted (for simplicity, ignore case in letters).

The car on the left hit the car I left.

The first three words have not been seen before, so they must be sent as full words. The fourth word is the second appearance of “the,” which at this point is the third word in the list. Thus, we only need to transmit the position value “3.” The next two words have not yet been seen, so must be sent as full words. The seventh word is the third appearance of “the,” which coincidentally is again in the third position. The eighth word is the second appearance of “car,” which is now in the fifth position of the list. “I” is a new word, and the last word “left” is now in the fifth position. Thus the entire transmission would be

The car on 3 left hit 3 5 I 5.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	0	1	1	0	1	0	1	0	0	0	1	0	1	0	0

Figure 9.1 The bit array for the set of primes in the range 0 to 15. The bit at position i is set to 1 if and only if i is prime.

This approach to compression is similar in spirit to Ziv-Lempel coding, which is a class of coding algorithms commonly used in file compression utilities. Ziv-Lempel coding replaces repeated occurrences of strings with a pointer to the location in the file of the first occurrence of the string. The codes are stored in a self-organizing list in order to speed up the time required to search for a string that has previously been seen.

9.3 Bit Vectors for Representing Sets

Determining whether a value is a member of a particular set is a special case of searching for keys in a sequence of records. Thus, any of the search methods discussed in this book can be used to check for set membership. However, we can also take advantage of the restricted circumstances imposed by this problem to develop another representation.

In the case where the set values fall within a limited range, we can represent the set using a bit array with a bit position allocated for each potential member. Those members actually in the set store a value of 1 in their corresponding bit; those members not in the set store a value of 0 in their corresponding bit. For example, consider the set of primes between 0 and 15. Figure 9.1 shows the corresponding bit array. To determine if a particular value is prime, we simply check the corresponding bit. This representation scheme is called a **bit vector** or a **bitmap**. The mark array used in several of the graph algorithms of Chapter 11 is an example of such a set representation.

If the set fits within a single computer word, then set union, intersection, and difference can be performed by logical bit-wise operations. The union of sets A and B is the bit-wise OR function (whose symbol is `|` in C++). The intersection of sets A and B is the bit-wise AND function (whose symbol is `&` in C++). For example, if we would like to compute the set of numbers between 0 and 15 that are both prime and odd numbers, we need only compute the expression

$$0011010100010100 \ \& \ 0101010101010101.$$

The set difference $A - B$ can be implemented in C++ using the expression `A & ~B` (`~` is the symbol for bit-wise negation). For larger sets that do not fit into a single computer word, the equivalent operations can be performed in turn on the series of words making up the entire bit vector.

This method of computing sets from bit vectors is sometimes applied to document retrieval. Consider the problem of picking from a collection of documents those few which contain selected keywords. For each keyword, the document retrieval system stores a bit vector with one bit for each document. If the user wants to know which documents contain a certain three keywords, the corresponding three bit vectors are AND'ed together. Those bit positions resulting in a value of 1 correspond to the desired documents. Alternatively, a bit vector can be stored for each document to indicate those keywords appearing in the document. Such an organization is called a **signature file**. The signatures can be manipulated to find documents with desired combinations of keywords.

9.4 Hashing

This section presents a completely different approach to searching arrays: by direct access based on key value. The process of finding a record using some computation to map its key value to a position in the array is called **hashing**. Most hashing schemes place records in the array in whatever order satisfies the needs of the address calculation, thus the records are not ordered by value or frequency. The function that maps key values to positions is called a **hash function** and will be denoted by **h**. The array that holds the records is called the **hash table** and will be denoted by **HT**. A position in the hash table is also known as a **slot**. The number of slots in hash table **HT** will be denoted by the variable M , with slots numbered from 0 to $M - 1$. The goal for a hashing system is to arrange things such that, for any key value K and some hash function **h**, $i = \mathbf{h}(K)$ is a slot in the table such that $0 \leq \mathbf{h}(K) < M$, and we have the key of the record stored at **HT**[i] equal to K .

Hashing is not good for applications where multiple records with the same key value are permitted. Hashing is not a good method for answering range searches. In other words, we cannot easily find all records (if any) whose key values fall within a certain range. Nor can we easily find the record with the minimum or maximum key value, or visit the records in key order. Hashing is most appropriate for answering the question, "What record, if any, has key value K ?" For applications where access involves only exact-match queries, hashing is usually the search method of choice because it is extremely efficient when implemented correctly. As you will see in this section, however, there are many approaches to hashing and it is easy to devise an inefficient implementation. Hashing is suitable for both in-memory and disk-based searching and is one of the two most widely used methods for organizing large databases stored on disk (the other is the B-tree, which is covered in Chapter 10).

As a simple (though unrealistic) example of hashing, consider storing n records each with a unique key value in the range 0 to $n - 1$. In this simple case, a record

with key k can be stored in $\mathbf{HT}[k]$, and the hash function is simply $\mathbf{h}(k) = k$. To find the record with key value k , simply look in $\mathbf{HT}[k]$.

Typically, there are many more values in the key range than there are slots in the hash table. For a more realistic example, suppose that the key can take any value in the range 0 to 65,535 (i.e., the key is a two-byte unsigned integer), and that we expect to store approximately 1000 records at any given time. It is impractical in this situation to use a hash table with 65,536 slots, because most of the slots will be left empty. Instead, we must devise a hash function that allows us to store the records in a much smaller table. Because the possible key range is larger than the size of the table, at least some of the slots must be mapped to from multiple key values. Given a hash function \mathbf{h} and two keys k_1 and k_2 , if $\mathbf{h}(k_1) = \beta = \mathbf{h}(k_2)$ where β is a slot in the table, then we say that k_1 and k_2 have a **collision** at slot β under hash function \mathbf{h} .

Finding a record with key value K in a database organized by hashing follows a two-step procedure:

1. Compute the table location $\mathbf{h}(K)$.
2. Starting with slot $\mathbf{h}(K)$, locate the record containing key K using (if necessary) a **collision resolution policy**.

9.4.1 Hash Functions

Hashing generally takes records whose key values come from a large range and stores those records in a table with a relatively small number of slots. Collisions occur when two records hash to the same slot in the table. If we are careful—or lucky—when selecting a hash function, then the actual number of collisions will be few. Unfortunately, even under the best of circumstances, collisions are nearly unavoidable.¹ For example, consider a classroom full of students. What is the probability that some pair of students shares the same birthday (i.e., the same day of the year, not necessarily the same year)? If there are 23 students, then the odds are about even that two will share a birthday. This is despite the fact that there are 365 days in which students can have birthdays (ignoring leap years), on most of which no student in the class has a birthday. With more students, the probability of a shared birthday increases. The mapping of students to days based on their

¹The exception to this is **perfect hashing**. Perfect hashing is a system in which records are hashed such that there are no collisions. A hash function is selected for the specific set of records being hashed, which requires that the entire collection of records be available before selecting the hash function. Perfect hashing is efficient because it always finds the record that we are looking for exactly where the hash function computes it to be, so only one access is required. Selecting a perfect hash function can be expensive, but might be worthwhile when extremely efficient search performance is required. An example is searching for data on a read-only CD. Here the database will never change, the time for each access is expensive, and the database designer can build the hash table before issuing the CD.

birthday is similar to assigning records to slots in a table (of size 365) using the birthday as a hash function. Note that this observation tells us nothing about *which* students share a birthday, or on *which* days of the year shared birthdays fall.

To be practical, a database organized by hashing must store records in a hash table that is not so large that it wastes space. Typically, this means that the hash table will be around half full. Because collisions are extremely likely to occur under these conditions (by chance, any record inserted into a table that is half full will have a collision half of the time), does this mean that we need not worry about the ability of a hash function to avoid collisions? Absolutely not. The difference between a good hash function and a bad hash function makes a big difference in practice. Technically, any function that maps all possible key values to a slot in the hash table is a hash function. In the extreme case, even a function that maps all records to the same slot is a hash function, but it does nothing to help us find records during a search operation.

We would like to pick a hash function that stores the actual records in the collection such that each slot in the hash table has equal probability of being filled. Unfortunately, we normally have no control over the key values of the actual records, so how well any particular hash function does this depends on the distribution of the keys within the allowable key range. In some cases, incoming data are well distributed across their key range. For example, if the input is a set of random numbers selected uniformly from the key range, any hash function that assigns the key range so that each slot in the hash table receives an equal share of the range will likely also distribute the input records uniformly within the table. However, in many applications the incoming records are highly clustered or otherwise poorly distributed. When input records are not well distributed throughout the key range it can be difficult to devise a hash function that does a good job of distributing the records throughout the table, especially if the input distribution is not known in advance.

There are many reasons why data values might be poorly distributed.

1. Natural frequency distributions tend to follow a common pattern where a few of the entities occur frequently while most entities occur relatively rarely. For example, consider the populations of the 100 largest cities in the United States. If you plot these populations on a number line, most of them will be clustered toward the low side, with a few outliers on the high side. This is an example of a Zipf distribution (see Section 9.2). Viewed the other way, the home town for a given person is far more likely to be a particular large city than a particular small town.
2. Collected data are likely to be skewed in some way. Field samples might be rounded to, say, the nearest 5 (i.e., all numbers end in 5 or 0).
3. If the input is a collection of common English words, the beginning letter will be poorly distributed.

Note that in examples 2 and 3, either high- or low-order bits of the key are poorly distributed.

When designing hash functions, we are generally faced with one of two situations.

1. We know nothing about the distribution of the incoming keys. In this case, we wish to select a hash function that evenly distributes the key range across the hash table, while avoiding obvious opportunities for clustering such as hash functions that are sensitive to the high- or low-order bits of the key value.
2. We know something about the distribution of the incoming keys. In this case, we should use a distribution-dependent hash function that avoids assigning clusters of related key values to the same hash table slot. For example, if hashing English words, we should *not* hash on the value of the first character because this is likely to be unevenly distributed.

Below are several examples of hash functions that illustrate these points.

Example 9.5 Consider the following hash function used to hash integers to a table of sixteen slots:

```
int h(int x) {  
    return x % 16;  
}
```

The value returned by this hash function depends solely on the least significant four bits of the key. Because these bits are likely to be poorly distributed (as an example, a high percentage of the keys might be even numbers, which means that the low order bit is zero), the result will also be poorly distributed. This example shows that the size of the table M can have a big effect on the performance of a hash system because this value is typically used as the modulus to ensure that the hash function produces a number in the range 0 to $M - 1$.

Example 9.6 A good hash function for numerical values comes from the **mid-square** method. The mid-square method squares the key value, and then takes the middle r bits of the result, giving a value in the range 0 to $2^r - 1$. This works well because most or all bits of the key value contribute to the result. For example, consider records whose keys are 4-digit numbers in base 10. The goal is to hash these key values to a table of size 100 (i.e., a range of 0 to 99). This range is equivalent to two digits in base 10. That is, $r = 2$. If the input is the number 4567, squaring yields an 8-digit number, 20857489. The middle two digits of this result are 57. All digits

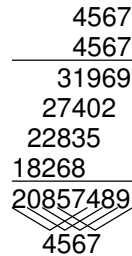


Figure 9.2 An illustration of the mid-square method, showing the details of long multiplication in the process of squaring the value 4567. The bottom of the figure indicates which digits of the answer are most influenced by each digit of the operands.

(equivalently, all bits when the number is viewed in binary) contribute to the middle two digits of the squared value. Figure 9.2 illustrates the concept. Thus, the result is not dominated by the distribution of the bottom digit or the top digit of the original key value.

Example 9.7 Here is a hash function for strings of characters:

```
int h(char* x) {
    int i, sum;
    for (sum=0, i=0; x[i] != '\0'; i++)
        sum += (int) x[i];
    return sum % M;
}
```

This function sums the ASCII values of the letters in a string. If the hash table size M is small, this hash function should do a good job of distributing strings evenly among the hash table slots, because it gives equal weight to all characters. This is an example of the **folding** approach to designing a hash function. Note that the order of the characters in the string has no effect on the result. A similar method for integers would add the digits of the key value, assuming that there are enough digits to (1) keep any one or two digits with bad distribution from skewing the results of the process and (2) generate a sum much larger than M . As with many other hash functions, the final step is to apply the modulus operator to the result, using table size M to generate a value within the table range. If the sum is not sufficiently large, then the modulus operator will yield a poor distribution. For example, because the ASCII value for “A” is 65 and “Z” is 90, **sum** will always be in the range 650 to 900 for a string of ten upper case letters. For a hash table of size 100 or less, a reasonable distribution results. For a hash table of size 1000, the distribution is terrible because only slots 650 to 900

can possibly be the home slot for some key value, and the values are not evenly distributed even within those slots.

Example 9.8 Here is a much better hash function for strings.

```
// Use folding on a string, summed 4 bytes at a time
int sfold(char* key) {
    unsigned int *lkey = (unsigned int *)key;
    int intlength = strlen(key)/4;
    unsigned int sum = 0;
    for(int i=0; i<intlength; i++)
        sum += lkey[i];

    // Now deal with the extra chars at the end
    int extra = strlen(key) - intlength*4;
    char temp[4];
    lkey = (unsigned int *)temp;
    lkey[0] = 0;
    for(int i=0; i<extra; i++)
        temp[i] = key[intlength*4+i];
    sum += lkey[0];

    return sum % M;
}
```

This function takes a string as input. It processes the string four bytes at a time, and interprets each of the four-byte chunks as a single (unsigned) long integer value. The integer values for the four-byte chunks are added together. In the end, the resulting sum is converted to the range 0 to $M - 1$ using the modulus operator.²

For example, if the string “aaaabbbb” is passed to **sfold**, then the first four bytes (“aaaa”) will be interpreted as the integer value 1,633,771,873 and the next four bytes (“bbbb”) will be interpreted as the integer value 1,650,614,882. Their sum is 3,284,386,755 (when viewed as an unsigned integer). If the table size is 101 then the modulus function will cause this key to hash to slot 75 in the table. Note that for any sufficiently long string, the sum for the integer quantities will typically cause a 32-bit integer to overflow (thus losing some of the high-order bits) because the resulting values are so large. But this causes no problems when the goal is to compute a hash function.

²Recall from Section 2.2 that the implementation for $n \bmod m$ on many C++ and Java compilers will yield a negative number if n is negative. Implementors for hash functions need to be careful that their hash function does not generate a negative number. This can be avoided either by insuring that n is positive when computing $n \bmod m$, or adding m to the result if $n \bmod m$ is negative. All computation in **sfold** is done using unsigned long values in part to protect against taking the modulus of an negative number.

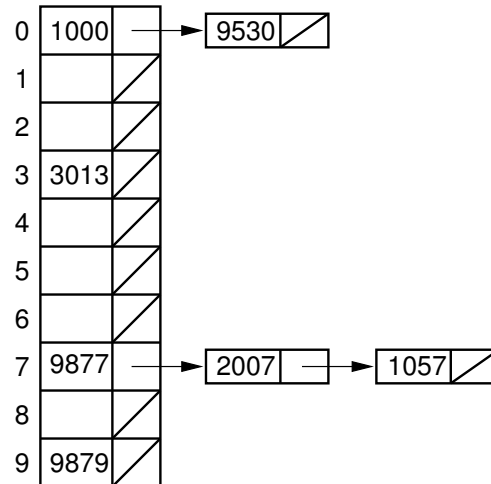


Figure 9.3 An illustration of open hashing for seven numbers stored in a ten-slot hash table using the hash function $h(K) = K \bmod 10$. The numbers are inserted in the order 9877, 2007, 1000, 9530, 3013, 9879, and 1057. Two of the values hash to slot 0, one value hashes to slot 2, three of the values hash to slot 7, and one value hashes to slot 9.

9.4.2 Open Hashing

While the goal of a hash function is to minimize collisions, some collisions are unavoidable in practice. Thus, hashing implementations must include some form of collision resolution policy. Collision resolution techniques can be broken into two classes: **open hashing** (also called **separate chaining**) and **closed hashing** (also called **open addressing**).³ The difference between the two has to do with whether collisions are stored outside the table (open hashing), or whether collisions result in storing one of the records at another slot in the table (closed hashing). Open hashing is treated in this section, and closed hashing in Section 9.4.3.

The simplest form of open hashing defines each slot in the hash table to be the head of a linked list. All records that hash to a particular slot are placed on that slot's linked list. Figure 9.3 illustrates a hash table where each slot stores one record and a link pointer to the rest of the list.

Records within a slot's list can be ordered in several ways: by insertion order, by key value order, or by frequency-of-access order. Ordering the list by key value provides an advantage in the case of an unsuccessful search, because we know to stop searching the list once we encounter a key that is greater than the one being

³Yes, it is confusing when “open hashing” means the opposite of “open addressing,” but unfortunately, that is the way it is.

searched for. If records on the list are unordered or ordered by frequency, then an unsuccessful search will need to visit every record on the list.

Given a table of size M storing N records, the hash function will (ideally) spread the records evenly among the M positions in the table, yielding on average N/M records for each list. Assuming that the table has more slots than there are records to be stored, we can hope that few slots will contain more than one record. In the case where a list is empty or has only one record, a search requires only one access to the list. Thus, the average cost for hashing should be $\Theta(1)$. However, if clustering causes many records to hash to only a few of the slots, then the cost to access a record will be much higher because many elements on the linked list must be searched.

Open hashing is most appropriate when the hash table is kept in main memory, with the lists implemented by a standard in-memory linked list. Storing an open hash table on disk in an efficient way is difficult, because members of a given linked list might be stored on different disk blocks. This would result in multiple disk accesses when searching for a particular key value, which defeats the purpose of using hashing.

There are similarities between open hashing and Binsort. One way to view open hashing is that each record is simply placed in a bin. While multiple records may hash to the same bin, this initial binning should still greatly reduce the number of records accessed by a search operation. In a similar fashion, a simple Binsort reduces the number of records in each bin to a small number that can be sorted in some other way.

9.4.3 Closed Hashing

Closed hashing stores all records directly in the hash table. Each record R with key value k_R has a **home position** that is $h(k_R)$, the slot computed by the hash function. If R is to be inserted and another record already occupies R 's home position, then R will be stored at some other slot in the table. It is the business of the collision resolution policy to determine which slot that will be. Naturally, the same policy must be followed during search as during insertion, so that any record not found in its home position can be recovered by repeating the collision resolution process.

Bucket Hashing

One implementation for closed hashing groups hash table slots into **buckets**. The M slots of the hash table are divided into B buckets, with each bucket consisting of M/B slots. The hash function assigns each record to the first slot within one of the buckets. If this slot is already occupied, then the bucket slots are searched sequentially until an open slot is found. If a bucket is entirely full, then the record is stored in an **overflow bucket** of infinite capacity at the end of the table. All

Hash Table		Overflow
0	1000	1057
	9530	
1		
2	9877	
	2007	
3	3013	
4	9879	

Figure 9.4 An illustration of bucket hashing for seven numbers stored in a five-bucket hash table using the hash function $h(K) = K \bmod 5$. Each bucket contains two slots. The numbers are inserted in the order 9877, 2007, 1000, 9530, 3013, 9879, and 1057. Two of the values hash to bucket 0, three values hash to bucket 2, one value hashes to bucket 3, and one value hashes to bucket 4. Because bucket 2 cannot hold three values, the third one ends up in the overflow bucket.

buckets share the same overflow bucket. A good implementation will use a hash function that distributes the records evenly among the buckets so that as few records as possible go into the overflow bucket. Figure 9.4 illustrates bucket hashing.

When searching for a record, the first step is to hash the key to determine which bucket should contain the record. The records in this bucket are then searched. If the desired key value is not found and the bucket still has free slots, then the search is complete. If the bucket is full, then it is possible that the desired record is stored in the overflow bucket. In this case, the overflow bucket must be searched until the record is found or all records in the overflow bucket have been checked. If many records are in the overflow bucket, this will be an expensive process.

A simple variation on bucket hashing is to hash a key value to some slot in the hash table as though bucketing were not being used. If the home position is full, then the collision resolution process is to move down through the table toward the end of the bucket while searching for a free slot in which to store the record. If the bottom of the bucket is reached, then the collision resolution routine wraps around to the top of the bucket to continue the search for an open slot. For example, assume that buckets contain eight records, with the first bucket consisting of slots 0 through 7. If a record is hashed to slot 5, the collision resolution process will attempt to insert the record into the table in the order 5, 6, 7, 0, 1, 2, 3, and finally 4. If all slots in this bucket are full, then the record is assigned to the overflow bucket.

Hash Table		Overflow
0	1000	1057
1	9530	
2		
3	3013	
4		
5		
6	2007	
7	9877	
8		
9	9879	

Figure 9.5 An variant of bucket hashing for seven numbers stored in a 10-slot hash table using the hash function $h(K) = K \bmod 10$. Each bucket contains two slots. The numbers are inserted in the order 9877, 2007, 1000, 9530, 3013, 9879, and 1057. Value 9877 first hashes to slot 7, so when value 2007 attempts to do likewise, it is placed in the other slot associated with that bucket which is slot 6. When value 1057 is inserted, there is no longer room in the bucket and it is placed into overflow. The other collision occurs after value 1000 is inserted to slot 0, causing 9530 to be moved to slot 1.

The advantage of this approach is that initial collisions are reduced, Because any slot can be a home position rather than just the first slot in the bucket. Figure 9.5 shows another example for this form of bucket hashing.

Bucket methods are good for implementing hash tables stored on disk, because the bucket size can be set to the size of a disk block. Whenever search or insertion occurs, the entire bucket is read into memory. Because the entire bucket is then in memory, processing an insert or search operation requires only one disk access, unless the bucket is full. If the bucket is full, then the overflow bucket must be retrieved from disk as well. Naturally, overflow should be kept small to minimize unnecessary disk accesses.

Linear Probing

We now turn to the most commonly used form of hashing: closed hashing with no bucketing, and a collision resolution policy that can potentially use any slot in the hash table.

During insertion, the goal of collision resolution is to find a free slot in the hash table when the home position for the record is already occupied. We can view any collision resolution method as generating a sequence of hash table slots that can

```

// Insert e into hash table HT
template <typename Key, typename E>
void hashdict<Key, E>::
hashInsert(const Key& k, const E& e) {
    int home;                                // Home position for e
    int pos = home = h(k);                    // Init probe sequence
    for (int i=1; EMPTYKEY != (HT[pos]).key(); i++) {
        pos = (home + p(k, i)) % M; // probe
        Assert(k != (HT[pos]).key(), "Duplicates not allowed");
    }
    KVpair<Key,E> temp(k, e);
    HT[pos] = temp;
}

```

Figure 9.6 Insertion method for a dictionary implemented by a hash table.

potentially hold the record. The first slot in the sequence will be the home position for the key. If the home position is occupied, then the collision resolution policy goes to the next slot in the sequence. If this is occupied as well, then another slot must be found, and so on. This sequence of slots is known as the **probe sequence**, and it is generated by some **probe function** that we will call **p**. The insert function is shown in Figure 9.6.

Method **hashInsert** first checks to see if the home slot for the key is empty. If the home slot is occupied, then we use the probe function, $p(k, i)$ to locate a free slot in the table. Function **p** has two parameters, the key k and a count i for where in the probe sequence we wish to be. That is, to get the first position in the probe sequence after the home slot for key K , we call $p(K, 1)$. For the next slot in the probe sequence, call $p(K, 2)$. Note that the probe function returns an offset from the original home position, rather than a slot in the hash table. Thus, the **for** loop in **hashInsert** is computing positions in the table at each iteration by adding the value returned from the probe function to the home position. The i th call to **p** returns the i th offset to be used.

Searching in a hash table follows the same probe sequence that was followed when inserting records. In this way, a record not in its home position can be recovered. A C++ implementation for the search procedure is shown in Figure 9.7.

The insert and search routines assume that at least one slot on the probe sequence of every key will be empty. Otherwise, they will continue in an infinite loop on unsuccessful searches. Thus, the dictionary should keep a count of the number of records stored, and refuse to insert into a table that has only one free slot.

The discussion on bucket hashing presented a simple method of collision resolution. If the home position for the record is occupied, then move down the bucket until a free slot is found. This is an example of a technique for collision resolution known as **linear probing**. The probe function for simple linear probing is

$$p(K, i) = i.$$

```

// Search for the record with Key K
template <typename Key, typename E>
E hashdict<Key, E>::
hashSearch(const Key& k) const {
    int home;           // Home position for k
    int pos = home = h(k); // Initial position is home slot
    for (int i = 1; (k != (HT[pos]).key()) &&
          (EMPTYKEY != (HT[pos]).key()); i++)
        pos = (home + p(k, i)) % M; // Next on probe sequence
    if (k == (HT[pos]).key()) // Found it
        return (HT[pos]).value();
    else return NULL;        // k not in hash table
}

```

Figure 9.7 Search method for a dictionary implemented by a hash table.

That is, the i th offset on the probe sequence is just i , meaning that the i th step is simply to move down i slots in the table.

Once the bottom of the table is reached, the probe sequence wraps around to the beginning of the table. Linear probing has the virtue that all slots in the table will be candidates for inserting a new record before the probe sequence returns to the home position.

While linear probing is probably the first idea that comes to mind when considering collision resolution policies, it is not the only one possible. Probe function p allows us many options for how to do collision resolution. In fact, linear probing is one of the worst collision resolution methods. The main problem is illustrated by Figure 9.8. Here, we see a hash table of ten slots used to store four-digit numbers, with hash function $h(K) = K \bmod 10$. In Figure 9.8(a), five numbers have been placed in the table, leaving five slots remaining.

The ideal behavior for a collision resolution mechanism is that each empty slot in the table will have equal probability of receiving the next record inserted (assuming that every slot in the table has equal probability of being hashed to initially). In this example, assume that the hash function gives each slot (roughly) equal probability of being the home position for the next key. However, consider what happens to the next record if its key has its home position at slot 0. Linear probing will send the record to slot 2. The same will happen to records whose home position is at slot 1. A record with home position at slot 2 will remain in slot 2. Thus, the probability is 3/10 that the next record inserted will end up in slot 2. In a similar manner, records hashing to slots 7 or 8 will end up in slot 9. However, only records hashing to slot 3 will be stored in slot 3, yielding one chance in ten of this happening. Likewise, there is only one chance in ten that the next record will be stored in slot 4, one chance in ten for slot 5, and one chance in ten for slot 6. Thus, the resulting probabilities are not equal.

To make matters worse, if the next record ends up in slot 9 (which already has a higher than normal chance of happening), then the following record will end up

0	9050
1	1001
2	
3	
4	
5	
6	
7	9877
8	2037
9	

(a)

0	9050
1	1001
2	
3	
4	
5	
6	
7	9877
8	2037
9	1059

(b)

Figure 9.8 Example of problems with linear probing. (a) Four values are inserted in the order 1001, 9050, 9877, and 2037 using hash function $h(K) = K \bmod 10$. (b) The value 1059 is added to the hash table.

in slot 2 with probability 6/10. This is illustrated by Figure 9.8(b). This tendency of linear probing to cluster items together is known as **primary clustering**. Small clusters tend to merge into big clusters, making the problem worse. The objection to primary clustering is that it leads to long probe sequences.

Improved Collision Resolution Methods

How can we avoid primary clustering? One possible improvement might be to use linear probing, but to skip slots by a constant c other than 1. This would make the probe function

$$p(K, i) = ci,$$

and so the i th slot in the probe sequence will be $(h(K) + ic) \bmod M$. In this way, records with adjacent home positions will not follow the same probe sequence. For example, if we were to skip by twos, then our offsets from the home slot would be 2, then 4, then 6, and so on.

One quality of a good probe sequence is that it will cycle through all slots in the hash table before returning to the home position. Clearly linear probing (which “skips” slots by one each time) does this. Unfortunately, not all values for c will make this happen. For example, if $c = 2$ and the table contains an even number of slots, then any key whose home position is in an even slot will have a probe sequence that cycles through only the even slots. Likewise, the probe sequence for a key whose home position is in an odd slot will cycle through the odd slots. Thus, this combination of table size and linear probing constant effectively divides

the records into two sets stored in two disjoint sections of the hash table. So long as both sections of the table contain the same number of records, this is not really important. However, just from chance it is likely that one section will become fuller than the other, leading to more collisions and poorer performance for those records. The other section would have fewer records, and thus better performance. But the overall system performance will be degraded, as the additional cost to the side that is more full outweighs the improved performance of the less-full side.

Constant c must be relatively prime to M to generate a linear probing sequence that visits all slots in the table (that is, c and M must share no factors). For a hash table of size $M = 10$, if c is any one of 1, 3, 7, or 9, then the probe sequence will visit all slots for any key. When $M = 11$, any value for c between 1 and 10 generates a probe sequence that visits all slots for every key.

Consider the situation where $c = 2$ and we wish to insert a record with key k_1 such that $h(k_1) = 3$. The probe sequence for k_1 is 3, 5, 7, 9, and so on. If another key k_2 has home position at slot 5, then its probe sequence will be 5, 7, 9, and so on. The probe sequences of k_1 and k_2 are linked together in a manner that contributes to clustering. In other words, linear probing with a value of $c > 1$ does not solve the problem of primary clustering. We would like to find a probe function that does not link keys together in this way. We would prefer that the probe sequence for k_1 after the first step on the sequence should not be identical to the probe sequence of k_2 . Instead, their probe sequences should diverge.

The ideal probe function would select the next position on the probe sequence at random from among the unvisited slots; that is, the probe sequence should be a random permutation of the hash table positions. Unfortunately, we cannot actually select the next position in the probe sequence at random, because then we would not be able to duplicate this same probe sequence when searching for the key. However, we can do something similar called **pseudo-random probing**. In pseudo-random probing, the i th slot in the probe sequence is $(h(K) + r_i) \bmod M$ where r_i is the i th value in a random permutation of the numbers from 1 to $M - 1$. All insertion and search operations use the same random permutation. The probe function is

$$p(K, i) = \text{Perm}[i - 1],$$

where **Perm** is an array of length $M - 1$ containing a random permutation of the values from 1 to $M - 1$.

Example 9.9 Consider a table of size $M = 101$, with $\text{Perm}[1] = 5$, $\text{Perm}[2] = 2$, and $\text{Perm}[3] = 32$. Assume that we have two keys k_1 and k_2 where $h(k_1) = 30$ and $h(k_2) = 35$. The probe sequence for k_1 is 30, then 35, then 32, then 62. The probe sequence for k_2 is 35, then 40, then 37, then 67. Thus, while k_2 will probe to k_1 's home position as its second choice, the two keys' probe sequences diverge immediately thereafter.

Another probe function that eliminates primary clustering is called **quadratic probing**. Here the probe function is some quadratic function

$$p(K, i) = c_1 i^2 + c_2 i + c_3$$

for some choice of constants c_1 , c_2 , and c_3 . The simplest variation is $p(K, i) = i^2$ (i.e., $c_1 = 1$, $c_2 = 0$, and $c_3 = 0$). Then the i th value in the probe sequence would be $(h(K) + i^2) \bmod M$. Under quadratic probing, two keys with different home positions will have diverging probe sequences.

Example 9.10 Given a hash table of size $M = 101$, assume for keys k_1 and k_2 that $h(k_1) = 30$ and $h(k_2) = 29$. The probe sequence for k_1 is 30, then 31, then 34, then 39. The probe sequence for k_2 is 29, then 30, then 33, then 38. Thus, while k_2 will probe to k_1 's home position as its second choice, the two keys' probe sequences diverge immediately thereafter.

Unfortunately, quadratic probing has the disadvantage that typically not all hash table slots will be on the probe sequence. Using $p(K, i) = i^2$ gives particularly inconsistent results. For many hash table sizes, this probe function will cycle through a relatively small number of slots. If all slots on that cycle happen to be full, then the record cannot be inserted at all! For example, if our hash table has three slots, then records that hash to slot 0 can probe only to slots 0 and 1 (that is, the probe sequence will never visit slot 2 in the table). Thus, if slots 0 and 1 are full, then the record cannot be inserted even though the table is not full. A more realistic example is a table with 105 slots. The probe sequence starting from any given slot will only visit 23 other slots in the table. If all 24 of these slots should happen to be full, even if other slots in the table are empty, then the record cannot be inserted because the probe sequence will continually hit only those same 24 slots.

Fortunately, it is possible to get good results from quadratic probing at low cost. The right combination of probe function and table size will visit many slots in the table. In particular, if the hash table size is a prime number and the probe function is $p(K, i) = i^2$, then at least half the slots in the table will be visited. Thus, if the table is less than half full, we can be certain that a free slot will be found. Alternatively, if the hash table size is a power of two and the probe function is $p(K, i) = (i^2 + i)/2$, then every slot in the table will be visited by the probe function.

Both pseudo-random probing and quadratic probing eliminate primary clustering, which is the problem of keys sharing substantial segments of a probe sequence. If two keys hash to the same home position, however, then they will always follow the same probe sequence for every collision resolution method that we have seen so far. The probe sequences generated by pseudo-random and quadratic probing (for example) are entirely a function of the home position, not the original key value.

This is because function \mathbf{p} ignores its input parameter K for these collision resolution methods. If the hash function generates a cluster at a particular home position, then the cluster remains under pseudo-random and quadratic probing. This problem is called **secondary clustering**.

To avoid secondary clustering, we need to have the probe sequence make use of the original key value in its decision-making process. A simple technique for doing this is to return to linear probing by a constant step size for the probe function, but to have that constant be determined by a second hash function, \mathbf{h}_2 . Thus, the probe sequence would be of the form $\mathbf{p}(K, i) = i * \mathbf{h}_2(K)$. This method is called **double hashing**.

Example 9.11 Assume a hash table has size $M = 101$, and that there are three keys k_1 , k_2 , and k_3 with $\mathbf{h}(k_1) = 30$, $\mathbf{h}(k_2) = 28$, $\mathbf{h}(k_3) = 30$, $\mathbf{h}_2(k_1) = 2$, $\mathbf{h}_2(k_2) = 5$, and $\mathbf{h}_2(k_3) = 5$. Then, the probe sequence for k_1 will be 30, 32, 34, 36, and so on. The probe sequence for k_2 will be 28, 33, 38, 43, and so on. The probe sequence for k_3 will be 30, 35, 40, 45, and so on. Thus, none of the keys share substantial portions of the same probe sequence. Of course, if a fourth key k_4 has $\mathbf{h}(k_4) = 28$ and $\mathbf{h}_2(k_4) = 2$, then it will follow the same probe sequence as k_1 . Pseudo-random or quadratic probing can be combined with double hashing to solve this problem.

A good implementation of double hashing should ensure that all of the probe sequence constants are relatively prime to the table size M . This can be achieved easily. One way is to select M to be a prime number, and have \mathbf{h}_2 return a value in the range $1 \leq \mathbf{h}_2(K) \leq M - 1$. Another way is to set $M = 2^m$ for some value m and have \mathbf{h}_2 return an odd value between 1 and 2^m .

Figure 9.9 shows an implementation of the dictionary ADT by means of a hash table. The simplest hash function is used, with collision resolution by linear probing, as the basis for the structure of a hash table implementation. A suggested project at the end of this chapter asks you to improve the implementation with other hash functions and collision resolution policies.

9.4.4 Analysis of Closed Hashing

How efficient is hashing? We can measure hashing performance in terms of the number of record accesses required when performing an operation. The primary operations of concern are insertion, deletion, and search. It is useful to distinguish between successful and unsuccessful searches. Before a record can be deleted, it must be found. Thus, the number of accesses required to delete a record is equivalent to the number required to successfully search for it. To insert a record, an empty slot along the record's probe sequence must be found. This is equivalent to

```

// Dictionary implemented with a hash table
template <typename Key, typename E>
class hashdict : public Dictionary<Key,E> {
private:
    KVpair<Key,E>* HT;    // The hash table
    int M;                // Size of HT
    int currnt;           // The current number of elements in HT
    Key EMPTYKEY;        // User-supplied key value for an empty slot

    int p(Key K, int i) const // Probe using linear probing
    { return i; }

    int h(int x) const { return x % M; } // Poor hash function
    int h(char* x) const { // Hash function for character keys
        int i, sum;
        for (sum=0, i=0; x[i] != '\0'; i++) sum += (int) x[i];
        return sum % M;
    }

    void hashInsert(const Key&, const E&);
    E hashSearch(const Key&) const;

public:
    hashdict(int sz, Key k){ // "k" defines an empty slot
        M = sz;
        EMPTYKEY = k;
        currnt = 0;
        HT = new KVpair<Key,E>[sz]; // Make HT of size sz
        for (int i=0; i<M; i++)
            (HT[i]).setKey(EMPTYKEY); // Initialize HT
    }

    ~hashdict() { delete HT; }

    // Find some record with key value "K"
    E find(const Key& k) const
    { return hashSearch(k); }
    int size() { return currnt; } // Number stored in table

    // Insert element "it" with Key "k" into the dictionary.
    void insert(const Key& k, const E& it) {
        Assert(currnt < M, "Hash table is full");
        hashInsert(k, it);
        currnt++;
    }
}

```

Figure 9.9 A partial implementation for the dictionary ADT using a hash table. This uses a poor hash function and a poor collision resolution policy (linear probing), which can easily be replaced. Member functions **hashInsert** and **hashSearch** appear in Figures 9.6 and 9.7, respectively.

an unsuccessful search for the record (recall that a successful search for the record during insertion should generate an error because two records with the same key are not allowed to be stored in the table).

When the hash table is empty, the first record inserted will always find its home position free. Thus, it will require only one record access to find a free slot. If all records are stored in their home positions, then successful searches will also require only one record access. As the table begins to fill up, the probability that a record can be inserted into its home position decreases. If a record hashes to an occupied slot, then the collision resolution policy must locate another slot in which to store it. Finding records not stored in their home position also requires additional record accesses as the record is searched for along its probe sequence. As the table fills up, more and more records are likely to be located ever further from their home positions.

From this discussion, we see that the expected cost of hashing is a function of how full the table is. Define the **load factor** for the table as $\alpha = N/M$, where N is the number of records currently in the table.

An estimate of the expected cost for an insertion (or an unsuccessful search) can be derived analytically as a function of α in the case where we assume that the probe sequence follows a random permutation of the slots in the hash table. Assuming that every slot in the table has equal probability of being the home slot for the next record, the probability of finding the home position occupied is α . The probability of finding both the home position occupied and the next slot on the probe sequence occupied is $\frac{N(N-1)}{M(M-1)}$. The probability of i collisions is

$$\frac{N(N-1) \cdots (N-i+1)}{M(M-1) \cdots (M-i+1)}.$$

If N and M are large, then this is approximately $(N/M)^i$. The expected number of probes is one plus the sum over $i \geq 1$ of the probability of i collisions, which is approximately

$$1 + \sum_{i=1}^{\infty} (N/M)^i = 1/(1 - \alpha).$$

The cost for a successful search (or a deletion) has the same cost as originally inserting that record. However, the expected value for the insertion cost depends on the value of α not at the time of deletion, but rather at the time of the original insertion. We can derive an estimate of this cost (essentially an average over all the insertion costs) by integrating from 0 to the current value of α , yielding a result of

$$\frac{1}{\alpha} \int_0^{\alpha} \frac{1}{1-x} dx = \frac{1}{\alpha} \log_e \frac{1}{1-\alpha}.$$

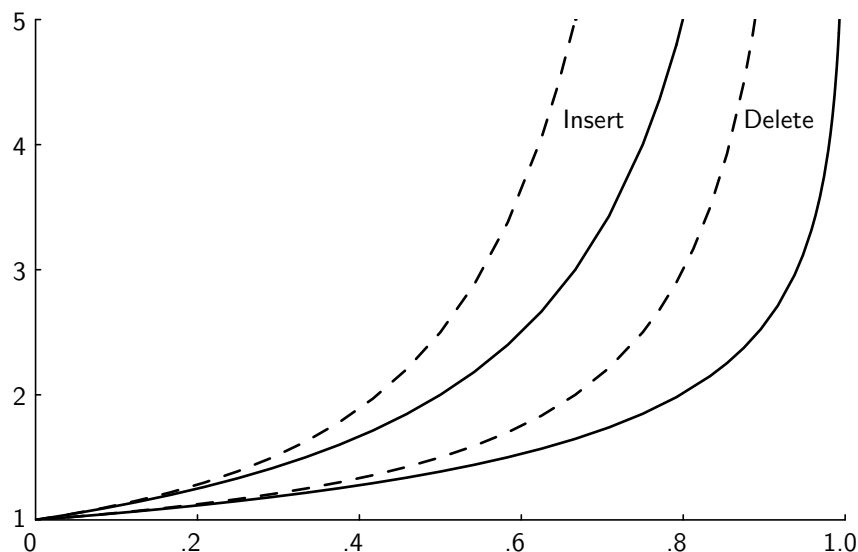


Figure 9.10 Growth of expected record accesses with α . The horizontal axis is the value for α , the vertical axis is the expected number of accesses to the hash table. Solid lines show the cost for “random” probing (a theoretical lower bound on the cost), while dashed lines show the cost for linear probing (a relatively poor collision resolution strategy). The two leftmost lines show the cost for insertion (equivalently, unsuccessful search); the two rightmost lines show the cost for deletion (equivalently, successful search).

It is important to realize that these equations represent the expected cost for operations using the unrealistic assumption that the probe sequence is based on a random permutation of the slots in the hash table (thus avoiding all expense resulting from clustering). Thus, these costs are lower-bound estimates in the average case. The true average cost under linear probing is $\frac{1}{2}(1 + 1/(1 - \alpha)^2)$ for insertions or unsuccessful searches and $\frac{1}{2}(1 + 1/(1 - \alpha))$ for deletions or successful searches. Proofs for these results can be found in the references cited in Section 9.5.

Figure 9.10 shows the graphs of these four equations to help you visualize the expected performance of hashing based on the load factor. The two solid lines show the costs in the case of a “random” probe sequence for (1) insertion or unsuccessful search and (2) deletion or successful search. As expected, the cost for insertion or unsuccessful search grows faster, because these operations typically search further down the probe sequence. The two dashed lines show equivalent costs for linear probing. As expected, the cost of linear probing grows faster than the cost for “random” probing.

From Figure 9.10 we see that the cost for hashing when the table is not too full is typically close to one record access. This is extraordinarily efficient, much better than binary search which requires $\log n$ record accesses. As α increases, so does

the expected cost. For small values of α , the expected cost is low. It remains below two until the hash table is about half full. When the table is nearly empty, adding a new record to the table does not increase the cost of future search operations by much. However, the additional search cost caused by each additional insertion increases rapidly once the table becomes half full. Based on this analysis, the rule of thumb is to design a hashing system so that the hash table never gets above half full. Beyond that point performance will degrade rapidly. This requires that the implementor have some idea of how many records are likely to be in the table at maximum loading, and select the table size accordingly.

You might notice that a recommendation to never let a hash table become more than half full contradicts the disk-based space/time tradeoff principle, which strives to minimize disk space to increase information density. Hashing represents an unusual situation in that there is no benefit to be expected from locality of reference. In a sense, the hashing system implementor does everything possible to eliminate the effects of locality of reference! Given the disk block containing the last record accessed, the chance of the next record access coming to the same disk block is no better than random chance in a well-designed hash system. This is because a good hashing implementation breaks up relationships between search keys. Instead of improving performance by taking advantage of locality of reference, hashing trades increased hash table space for an improved chance that the record will be in its home position. Thus, the more space available for the hash table, the more efficient hashing should be.

Depending on the pattern of record accesses, it might be possible to reduce the expected cost of access even in the face of collisions. Recall the 80/20 rule: 80% of the accesses will come to 20% of the data. In other words, some records are accessed more frequently. If two records hash to the same home position, which would be better placed in the home position, and which in a slot further down the probe sequence? The answer is that the record with higher frequency of access should be placed in the home position, because this will reduce the total number of record accesses. Ideally, records along a probe sequence will be ordered by their frequency of access.

One approach to approximating this goal is to modify the order of records along the probe sequence whenever a record is accessed. If a search is made to a record that is not in its home position, a self-organizing list heuristic can be used. For example, if the linear probing collision resolution policy is used, then whenever a record is located that is not in its home position, it can be swapped with the record preceding it in the probe sequence. That other record will now be further from its home position, but hopefully it will be accessed less frequently. Note that this approach will not work for the other collision resolution policies presented in this section, because swapping a pair of records to improve access to one might remove the other from its probe sequence.

Another approach is to keep access counts for records and periodically rehash the entire table. The records should be inserted into the hash table in frequency order, ensuring that records that were frequently accessed during the last series of requests have the best chance of being near their home positions.

9.4.5 Deletion

When deleting records from a hash table, there are two important considerations.

1. Deleting a record must not hinder later searches. In other words, the search process must still pass through the newly emptied slot to reach records whose probe sequence passed through this slot. Thus, the delete process cannot simply mark the slot as empty, because this will isolate records further down the probe sequence. For example, in Figure 9.8(a), keys 9877 and 2037 both hash to slot 7. Key 2037 is placed in slot 8 by the collision resolution policy. If 9877 is deleted from the table, a search for 2037 must still pass through Slot 7 as it probes to slot 8.
2. We do not want to make positions in the hash table unusable because of deletion. The freed slot should be available to a future insertion.

Both of these problems can be resolved by placing a special mark in place of the deleted record, called a **tombstone**. The tombstone indicates that a record once occupied the slot but does so no longer. If a tombstone is encountered when searching along a probe sequence, the search procedure continues with the search. When a tombstone is encountered during insertion, that slot can be used to store the new record. However, to avoid inserting duplicate keys, it will still be necessary for the search procedure to follow the probe sequence until a truly empty position has been found, simply to verify that a duplicate is not in the table. However, the new record would actually be inserted into the slot of the first tombstone encountered.

The use of tombstones allows searches to work correctly and allows reuse of deleted slots. However, after a series of intermixed insertion and deletion operations, some slots will contain tombstones. This will tend to lengthen the average distance from a record's home position to the record itself, beyond where it could be if the tombstones did not exist. A typical database application will first load a collection of records into the hash table and then progress to a phase of intermixed insertions and deletions. After the table is loaded with the initial collection of records, the first few deletions will lengthen the average probe sequence distance for records (it will add tombstones). Over time, the average distance will reach an equilibrium point because insertions will tend to decrease the average distance by filling in tombstone slots. For example, after initially loading records into the database, the average path distance might be 1.2 (i.e., an average of 0.2 accesses per search beyond the home position will be required). After a series of insertions and deletions, this average distance might increase to 1.6 due to tombstones. This

seems like a small increase, but it is three times longer on average beyond the home position than before deletions.

Two possible solutions to this problem are

1. Do a local reorganization upon deletion to try to shorten the average path length. For example, after deleting a key, continue to follow the probe sequence of that key and swap records further down the probe sequence into the slot of the recently deleted record (being careful not to remove any key from its probe sequence). This will not work for all collision resolution policies.
2. Periodically rehash the table by reinserting all records into a new hash table. Not only will this remove the tombstones, but it also provides an opportunity to place the most frequently accessed records into their home positions.

9.5 Further Reading

For a comparison of the efficiencies for various self-organizing techniques, see Bentley and McGeoch, “Amortized Analysis of Self-Organizing Sequential Search Heuristics” [BM85]. The text compression example of Section 9.2 comes from Bentley et al., “A Locally Adaptive Data Compression Scheme” [BSTW86]. For more on Ziv-Lempel coding, see *Data Compression: Methods and Theory* by James A. Storer [Sto88]. Knuth covers self-organizing lists and Zipf distributions in Volume 3 of *The Art of Computer Programming* [Knu98].

Introduction to Modern Information Retrieval by Salton and McGill [SM83] is an excellent source for more information about document retrieval techniques.

See the paper “Practical Minimal Perfect Hash Functions for Large Databases” by Fox et al. [FHCD92] for an introduction and a good algorithm for perfect hashing.

For further details on the analysis for various collision resolution policies, see Knuth, Volume 3 [Knu98] and *Concrete Mathematics: A Foundation for Computer Science* by Graham, Knuth, and Patashnik [GKP94].

The model of hashing presented in this chapter has been of a fixed-size hash table. A problem not addressed is what to do when the hash table gets half full and more records must be inserted. This is the domain of dynamic hashing methods. A good introduction to this topic is “Dynamic Hashing Schemes” by R.J. Enbody and H.C. Du [ED88].

9.6 Exercises

- 9.1 Create a graph showing expected cost versus the probability of an unsuccessful search when performing sequential search (see Section 9.1). What

can you say qualitatively about the rate of increase in expected cost as the probability of unsuccessful search grows?

- 9.2 Modify the binary search routine of Section 3.5 to implement interpolation search. Assume that keys are in the range 1 to 10,000, and that all key values within the range are equally likely to occur.
- 9.3 Write an algorithm to find the K th smallest value in an unsorted array of n numbers ($K \leq n$). Your algorithm should require $\Theta(n)$ time in the average case. Hint: Your algorithm should look similar to Quicksort.
- 9.4 Example 9.9.3 discusses a distribution where the relative frequencies of the records match the harmonic series. That is, for every occurrence of the first record, the second record will appear half as often, the third will appear one third as often, the fourth one quarter as often, and so on. The actual probability for the i th record was defined to be $1/(i\mathcal{H}_n)$. Explain why this is correct.
- 9.5 Graph the equations $\mathbf{T}(n) = \log_2 n$ and $\mathbf{T}(n) = n/\log_e n$. Which gives the better performance, binary search on a sorted list, or sequential search on a list ordered by frequency where the frequency conforms to a Zipf distribution? Characterize the difference in running times.
- 9.6 Assume that the values A through H are stored in a self-organizing list, initially in ascending order. Consider the three self-organizing list heuristics: count, move-to-front, and transpose. For count, assume that the record is moved ahead in the list passing over any other record that its count is now greater than. For each, show the resulting list and the total number of comparisons required resulting from the following series of accesses:

D H H G H E G H G H E C E H G.

- 9.7 For each of the three self-organizing list heuristics (count, move-to-front, and transpose), describe a series of record accesses for which it would require the greatest number of comparisons of the three.
- 9.8 Write an algorithm to implement the frequency count self-organizing list heuristic, assuming that the list is implemented using an array. In particular, write a function **FreqCount** that takes as input a value to be searched for and which adjusts the list appropriately. If the value is not already in the list, add it to the end of the list with a frequency count of one.
- 9.9 Write an algorithm to implement the move-to-front self-organizing list heuristic, assuming that the list is implemented using an array. In particular, write a function **MoveToFront** that takes as input a value to be searched for and which adjusts the list appropriately. If the value is not already in the list, add it to the beginning of the list.
- 9.10 Write an algorithm to implement the transpose self-organizing list heuristic, assuming that the list is implemented using an array. In particular, write

a function **Transpose** that takes as input a value to be searched for and which adjusts the list appropriately. If the value is not already in the list, add it to the end of the list.

- 9.11** Write functions for computing union, intersection, and set difference on arbitrarily long bit vectors used to represent set membership as described in Section 9.3. Assume that for each operation both vectors are of equal length.
- 9.12** Compute the probabilities for the following situations. These probabilities can be computed analytically, or you may write a computer program to generate the probabilities by simulation.
- (a) Out of a group of 23 students, what is the probability that 2 students share the same birthday?
 - (b) Out of a group of 100 students, what is the probability that 3 students share the same birthday?
 - (c) How many students must be in the class for the probability to be at least 50% that there are 2 who share a birthday in the same month?
- 9.13** Assume that you are hashing key K to a hash table of n slots (indexed from 0 to $n - 1$). For each of the following functions $h(K)$, is the function acceptable as a hash function (i.e., would the hash program work correctly for both insertions and searches), and if so, is it a good hash function? Function **Random(n)** returns a random integer between 0 and $n - 1$, inclusive.
- (a) $h(k) = k/n$ where k and n are integers.
 - (b) $h(k) = 1$.
 - (c) $h(k) = (k + \text{Random}(n)) \bmod n$.
 - (d) $h(k) = k \bmod n$ where n is a prime number.
- 9.14** Assume that you have a seven-slot closed hash table (the slots are numbered 0 through 6). Show the final hash table that would result if you used the hash function $h(k) = k \bmod 7$ and linear probing on this list of numbers: 3, 12, 9, 2. After inserting the record with key value 2, list for each empty slot the probability that it will be the next one filled.
- 9.15** Assume that you have a ten-slot closed hash table (the slots are numbered 0 through 9). Show the final hash table that would result if you used the hash function $h(k) = k \bmod 10$ and quadratic probing on this list of numbers: 3, 12, 9, 2, 79, 46. After inserting the record with key value 46, list for each empty slot the probability that it will be the next one filled.
- 9.16** Assume that you have a ten-slot closed hash table (the slots are numbered 0 through 9). Show the final hash table that would result if you used the hash function $h(k) = k \bmod 10$ and pseudo-random probing on this list of numbers: 3, 12, 9, 2, 79, 44. The permutation of offsets to be used by the pseudo-random probing will be: 5, 9, 2, 1, 4, 8, 6, 3, 7. After inserting the record with key value 44, list for each empty slot the probability that it will be the next one filled.

- 9.17** What is the result of running **sfold** from Section 9.4.1 on the following strings? Assume a hash table size of 101 slots.
- (a) HELLO WORLD
 - (b) NOW HEAR THIS
 - (c) HEAR THIS NOW
- 9.18** Using closed hashing, with double hashing to resolve collisions, insert the following keys into a hash table of thirteen slots (the slots are numbered 0 through 12). The hash functions to be used are H1 and H2, defined below. You should show the hash table after all eight keys have been inserted. Be sure to indicate how you are using H1 and H2 to do the hashing. Function $\text{Rev}(k)$ reverses the decimal digits of k , for example, $\text{Rev}(37) = 73$; $\text{Rev}(7) = 7$.
- $H1(k) = k \bmod 13$.
- $H2(k) = (\text{Rev}(k + 1) \bmod 11)$.
- Keys: 2, 8, 31, 20, 19, 18, 53, 27.
- 9.19** Write an algorithm for a deletion function for hash tables that replaces the record with a special value indicating a tombstone. Modify the functions **hashInsert** and **hashSearch** to work correctly with tombstones.
- 9.20** Consider the following permutation for the numbers 1 to 6:

2, 4, 6, 1, 3, 5.

Analyze what will happen if this permutation is used by an implementation of pseudo-random probing on a hash table of size seven. Will this permutation solve the problem of primary clustering? What does this say about selecting a permutation for use when implementing pseudo-random probing?

9.7 Projects

- 9.1** Implement a binary search and the quadratic binary search of Section 9.1. Run your implementations over a large range of problem sizes, timing the results for each algorithm. Graph and compare these timing results.
- 9.2** Implement the three self-organizing list heuristics count, move-to-front, and transpose. Compare the cost for running the three heuristics on various input data. The cost metric should be the total number of comparisons required when searching the list. It is important to compare the heuristics using input data for which self-organizing lists are reasonable, that is, on frequency distributions that are uneven. One good approach is to read text files. The list should store individual words in the text file. Begin with an empty list, as was done for the text compression example of Section 9.2. Each time a word is encountered in the text file, search for it in the self-organizing list. If the

word is found, reorder the list as appropriate. If the word is not in the list, add it to the end of the list and then reorder as appropriate.

- 9.3** Implement the text compression system described in Section 9.2.
- 9.4** Implement a system for managing document retrieval. Your system should have the ability to insert (abstract references to) documents into the system, associate keywords with a given document, and to search for documents with specified keywords.
- 9.5** Implement a database stored on disk using bucket hashing. Define records to be 128 bytes long with a 4-byte key and 120 bytes of data. The remaining 4 bytes are available for you to store necessary information to support the hash table. A bucket in the hash table will be 1024 bytes long, so each bucket has space for 8 records. The hash table should consist of 27 buckets (total space for 216 records with slots indexed by positions 0 to 215) followed by the overflow bucket at record position 216 in the file. The hash function for key value K should be $K \bmod 213$. (Note that this means the last three slots in the table will not be home positions for any record.) The collision resolution function should be linear probing with wrap-around within the bucket. For example, if a record is hashed to slot 5, the collision resolution process will attempt to insert the record into the table in the order 5, 6, 7, 0, 1, 2, 3, and finally 4. If a bucket is full, the record should be placed in the overflow section at the end of the file.
- Your hash table should implement the dictionary ADT of Section 4.4. When you do your testing, assume that the system is meant to store about 100 or so records at a time.
- 9.6** Implement the dictionary ADT of Section 4.4 by means of a hash table with linear probing as the collision resolution policy. You might wish to begin with the code of Figure 9.9. Using empirical simulation, determine the cost of insert and delete as α grows (i.e., reconstruct the dashed lines of Figure 9.10). Then, repeat the experiment using quadratic probing and pseudo-random probing. What can you say about the relative performance of these three collision resolution policies?

Indexing

Many large-scale computing applications are centered around data sets that are too large to fit into main memory. The classic example is a large database of records with multiple search keys, requiring the ability to insert, delete, and search for records. Hashing provides outstanding performance for such situations, but only in the limited case in which all searches are of the form “find the record with key value K .” Many applications require more general search capabilities. One example is a range query search for all records whose key lies within some range. Other queries might involve visiting all records in order of their key value, or finding the record with the greatest key value. Hash tables are not organized to support any of these queries efficiently.

This chapter introduces file structures used to organize a large collection of records stored on disk. Such file structures support efficient insertion, deletion, and search operations, for exact-match queries, range queries, and largest/smallest key value searches.

Before discussing such file structures, we must become familiar with some basic file-processing terminology. An **entry-sequenced file** stores records in the order that they were added to the file. Entry-sequenced files are the disk-based equivalent to an unsorted list and so do not support efficient search. The natural solution is to sort the records by order of the search key. However, a typical database, such as a collection of employee or customer records maintained by a business, might contain multiple search keys. To answer a question about a particular customer might require a search on the name of the customer. Businesses often wish to sort and output the records by zip code order for a bulk mailing. Government paperwork might require the ability to search by Social Security number. Thus, there might not be a single “correct” order in which to store the records.

Indexing is the process of associating a key with the location of a corresponding data record. Section 8.5 discussed the concept of a key sort, in which an **index file** is created whose records consist of key/pointer pairs. Here, each key is associated with a pointer to a complete record in the main database file. The index file

could be sorted or organized using a tree structure, thereby imposing a logical order on the records without physically rearranging them. One database might have several associated index files, each supporting efficient access through a different key field.

Each record of a database normally has a unique identifier, called the **primary key**. For example, the primary key for a set of personnel records might be the Social Security number or ID number for the individual. Unfortunately, the ID number is generally an inconvenient value on which to perform a search because the searcher is unlikely to know it. Instead, the searcher might know the desired employee's name. Alternatively, the searcher might be interested in finding all employees whose salary is in a certain range. If these are typical search requests to the database, then the name and salary fields deserve separate indices. However, key values in the name and salary indices are not likely to be unique.

A key field such as salary, where a particular key value might be duplicated in multiple records, is called a **secondary key**. Most searches are performed using a secondary key. The secondary key index (or more simply, **secondary index**) will associate a secondary key value with the primary key of each record having that secondary key value. At this point, the full database might be searched directly for the record with that primary key, or there might be a primary key index (or **primary index**) that relates each primary key value with a pointer to the actual record on disk. In the latter case, only the primary index provides the location of the actual record on disk, while the secondary indices refer to the primary index.

Indexing is an important technique for organizing large databases, and many indexing methods have been developed. Direct access through hashing is discussed in Section 9.4. A simple list sorted by key value can also serve as an index to the record file. Indexing disk files by sorted lists are discussed in the following section. Unfortunately, a sorted list does not perform well for insert and delete operations.

A third approach to indexing is the tree index. Trees are typically used to organize large databases that must support record insertion, deletion, and key range searches. Section 10.2 briefly describes ISAM, a tentative step toward solving the problem of storing a large database that must support insertion and deletion of records. Its shortcomings help to illustrate the value of tree indexing techniques. Section 10.3 introduces the basic issues related to tree indexing. Section 10.4 introduces the 2-3 tree, a balanced tree structure that is a simple form of the B-tree covered in Section 10.5. B-trees are the most widely used indexing method for large disk-based databases, and for implementing file systems. Since they have such great practical importance, many variations have been invented. Section 10.5 begins with a discussion of the variant normally referred to simply as a "B-tree." Section 10.5.1 presents the most widely implemented variant, the B⁺-tree.

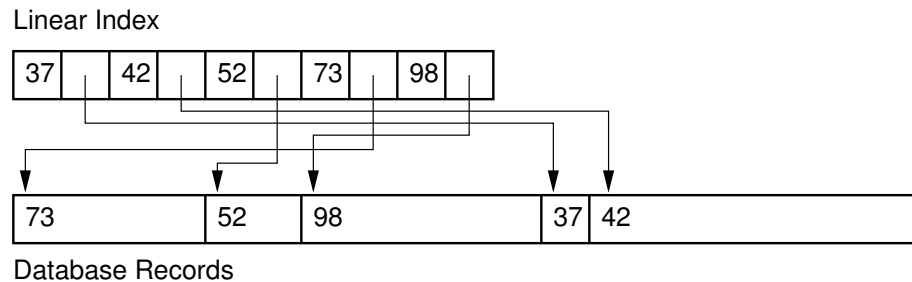


Figure 10.1 Linear indexing for variable-length records. Each record in the index file is of fixed length and contains a pointer to the beginning of the corresponding record in the database file.

10.1 Linear Indexing

A **linear index** is an index file organized as a sequence of key/pointer pairs where the keys are in sorted order and the pointers either (1) point to the position of the complete record on disk, (2) point to the position of the primary key in the primary index, or (3) are actually the value of the primary key. Depending on its size, a linear index might be stored in main memory or on disk. A linear index provides a number of advantages. It provides convenient access to variable-length database records, because each entry in the index file contains a fixed-length key field and a fixed-length pointer to the beginning of a (variable-length) record as shown in Figure 10.1. A linear index also allows for efficient search and random access to database records, because it is amenable to binary search.

If the database contains enough records, the linear index might be too large to store in main memory. This makes binary search of the index more expensive because many disk accesses would typically be required by the search process. One solution to this problem is to store a second-level linear index in main memory that indicates which disk block in the index file stores a desired key. For example, the linear index on disk might reside in a series of 1024-byte blocks. If each key/pointer pair in the linear index requires 8 bytes (a 4-byte key and a 4-byte pointer), then 128 key/pointer pairs are stored per block. The second-level index, stored in main memory, consists of a simple table storing the value of the key in the first position of each block in the linear index file. This arrangement is shown in Figure 10.2. If the linear index requires 1024 disk blocks (1MB), the second-level index contains only 1024 entries, one per disk block. To find which disk block contains a desired search key value, first search through the 1024-entry table to find the greatest value less than or equal to the search key. This directs the search to the proper block in the index file, which is then read into memory. At this point, a binary search within this block will produce a pointer to the actual record in the database. Because the

1	2003	5894	10528
---	------	------	-------

Second Level Index

1	2001	2003	5688	5894	9942	10528	10984
---	------	------	------	------	------	-------	-------

Linear Index: Disk Blocks

Figure 10.2 A simple two-level linear index. The linear index is stored on disk. The smaller, second-level index is stored in main memory. Each element in the second-level index stores the first key value in the corresponding disk block of the index file. In this example, the first disk block of the linear index stores keys in the range 1 to 2001, and the second disk block stores keys in the range 2003 to 5688. Thus, the first entry of the second-level index is key value 1 (the first key in the first block of the linear index), while the second entry of the second-level index is key value 2003.

second-level index is stored in main memory, accessing a record by this method requires two disk reads: one from the index file and one from the database file for the actual record.

Every time a record is inserted to or deleted from the database, all associated secondary indices must be updated. Updates to a linear index are expensive, because the entire contents of the array might be shifted. Another problem is that multiple records with the same secondary key each duplicate that key value within the index. When the secondary key field has many duplicates, such as when it has a limited range (e.g., a field to indicate job category from among a small number of possible job categories), this duplication might waste considerable space.

One improvement on the simple sorted array is a two-dimensional array where each row corresponds to a secondary key value. A row contains the primary keys whose records have the indicated secondary key value. Figure 10.3 illustrates this approach. Now there is no duplication of secondary key values, possibly yielding a considerable space savings. The cost of insertion and deletion is reduced, because only one row of the table need be adjusted. Note that a new row is added to the array when a new secondary key value is added. This might lead to moving many records, but this will happen infrequently in applications suited to using this arrangement.

A drawback to this approach is that the array must be of fixed size, which imposes an upper limit on the number of primary keys that might be associated with a particular secondary key. Furthermore, those secondary keys with fewer records than the width of the array will waste the remainder of their row. A better approach is to have a one-dimensional array of secondary key values, where each secondary key is associated with a linked list. This works well if the index is stored in main memory, but not so well when it is stored on disk because the linked list for a given key might be scattered across several disk blocks.

Jones	AA10	AB12	AB39	FF37
Smith	AX33	AX35	ZX45	
Zukowski	ZQ99			

Figure 10.3 A two-dimensional linear index. Each row lists the primary keys associated with a particular secondary key value. In this example, the secondary key is a name. The primary key is a unique four-character code.

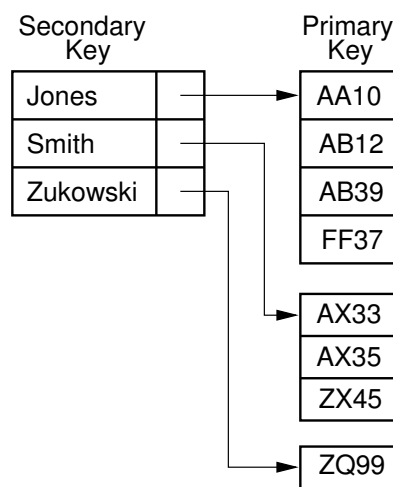


Figure 10.4 Illustration of an inverted list. Each secondary key value is stored in the secondary key list. Each secondary key value on the list has a pointer to a list of the primary keys whose associated records have that secondary key value.

Consider a large database of employee records. If the primary key is the employee's ID number and the secondary key is the employee's name, then each record in the name index associates a name with one or more ID numbers. The ID number index in turn associates an ID number with a unique pointer to the full record on disk. The secondary key index in such an organization is also known as an **inverted list** or **inverted file**. It is inverted in that searches work backwards from the secondary key to the primary key to the actual data record. It is called a list because each secondary key value has (conceptually) a list of primary keys associated with it. Figure 10.4 illustrates this arrangement. Here, we have last names as the secondary key. The primary key is a four-character unique identifier.

Figure 10.5 shows a better approach to storing inverted lists. An array of secondary key values is shown as before. Associated with each secondary key is a pointer to an array of primary keys. The primary key array uses a linked-list implementation. This approach combines the storage for all of the secondary key lists into a single array, probably saving space. Each record in this array consists of a

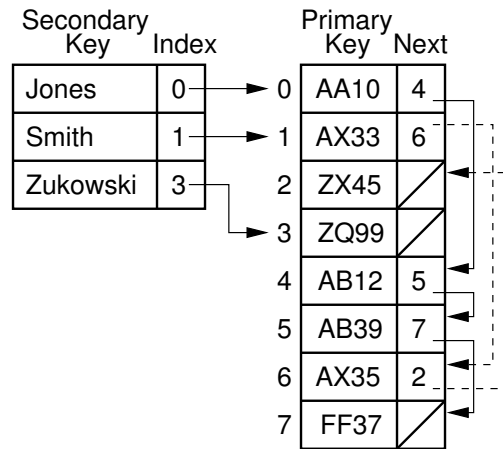


Figure 10.5 An inverted list implemented as an array of secondary keys and combined lists of primary keys. Each record in the secondary key array contains a pointer to a record in the primary key array. The **next** field of the primary key array indicates the next record with that secondary key value.

primary key value and a pointer to the next element on the list. It is easy to insert and delete secondary keys from this array, making this a good implementation for disk-based inverted files.

10.2 ISAM

How do we handle large databases that require frequent update? The main problem with the linear index is that it is a single, large array that does not adjust well to updates because a single update can require changing the position of every key in the index. Inverted lists reduce this problem, but they are only suitable for secondary key indices with many fewer secondary key values than records. The linear index would perform well as a primary key index if it could somehow be broken into pieces such that individual updates affect only a part of the index. This concept will be pursued throughout the rest of this chapter, eventually culminating in the B⁺-tree, the most widely used indexing method today. But first, we begin by studying ISAM, an early attempt to solve the problem of large databases requiring frequent update. Its weaknesses help to illustrate why the B⁺-tree works so well.

Before the invention of effective tree indexing schemes, a variety of disk-based indexing methods were in use. All were rather cumbersome, largely because no adequate method for handling updates was known. Typically, updates would cause the index to degrade in performance. ISAM is one example of such an index and was widely used by IBM prior to adoption of the B-tree.

ISAM is based on a modified form of the linear index, as illustrated by Figure 10.6. Records are stored in sorted order by primary key. The disk file is divided

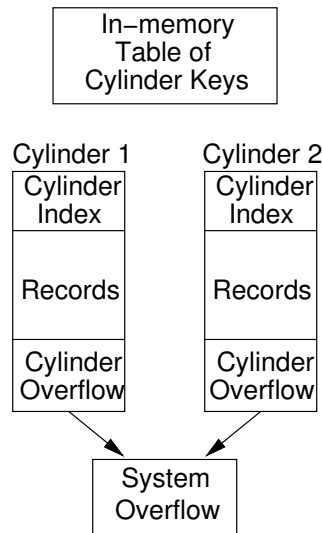


Figure 10.6 Illustration of the ISAM indexing system.

among a number of cylinders on disk.¹ Each cylinder holds a section of the list in sorted order. Initially, each cylinder is not filled to capacity, and the extra space is set aside in the **cylinder overflow**. In memory is a table listing the lowest key value stored in each cylinder of the file. Each cylinder contains a table listing the lowest key value for each block in that cylinder, called the **cylinder index**. When new records are inserted, they are placed in the correct cylinder's overflow area (in effect, a cylinder acts as a bucket). If a cylinder's overflow area fills completely, then a system-wide overflow area is used. Search proceeds by determining the proper cylinder from the system-wide table kept in main memory. The cylinder's block table is brought in from disk and consulted to determine the correct block. If the record is found in that block, then the search is complete. Otherwise, the cylinder's overflow area is searched. If that is full, and the record is not found, then the system-wide overflow is searched.

After initial construction of the database, so long as no new records are inserted or deleted, access is efficient because it requires only two disk fetches. The first disk fetch recovers the block table for the desired cylinder. The second disk fetch recovers the block that, under good conditions, contains the record. After many inserts, the overflow list becomes too long, resulting in significant search time as the cylinder overflow area fills up. Under extreme conditions, many searches might eventually lead to the system overflow area. The "solution" to this problem is to periodically reorganize the entire database. This means re-balancing the records

¹Recall from Section 8.2.1 that a cylinder is all of the tracks readable from a particular placement of the heads on the multiple platters of a disk drive.

among the cylinders, sorting the records within each cylinder, and updating both the system index table and the within-cylinder block table. Such reorganization was typical of database systems during the 1960s and would normally be done each night or weekly.

10.3 Tree-based Indexing

Linear indexing is efficient when the database is static, that is, when records are inserted and deleted rarely or never. ISAM is adequate for a limited number of updates, but not for frequent changes. Because it has essentially two levels of indexing, ISAM will also break down for a truly large database where the number of cylinders is too great for the top-level index to fit in main memory.

In their most general form, database applications have the following characteristics:

1. Large sets of records that are frequently updated.
2. Search is by one or a combination of several keys.
3. Key range queries or min/max queries are used.

For such databases, a better organization must be found. One approach would be to use the binary search tree (BST) to store primary and secondary key indices. BSTs can store duplicate key values, they provide efficient insertion and deletion as well as efficient search, and they can perform efficient range queries. When there is enough main memory, the BST is a viable option for implementing both primary and secondary key indices.

Unfortunately, the BST can become unbalanced. Even under relatively good conditions, the depth of leaf nodes can easily vary by a factor of two. This might not be a significant concern when the tree is stored in main memory because the time required is still $\Theta(\log n)$ for search and update. When the tree is stored on disk, however, the depth of nodes in the tree becomes crucial. Every time a BST node B is visited, it is necessary to visit all nodes along the path from the root to B . Each node on this path must be retrieved from disk. Each disk access returns a block of information. If a node is on the same block as its parent, then the cost to find that node is trivial once its parent is in main memory. Thus, it is desirable to keep subtrees together on the same block. Unfortunately, many times a node is not on the same block as its parent. Thus, each access to a BST node could potentially require that another block to be read from disk. Using a buffer pool to store multiple blocks in memory can mitigate disk access problems if BST accesses display good locality of reference. But a buffer pool cannot eliminate disk I/O entirely. The problem becomes greater if the BST is unbalanced, because nodes deep in the tree have the potential of causing many disk blocks to be read. Thus, there are two significant issues that must be addressed to have efficient search from a disk-based

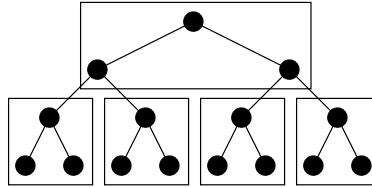


Figure 10.7 Breaking the BST into blocks. The BST is divided among disk blocks, each with space for three nodes. The path from the root to any leaf is contained on two blocks.

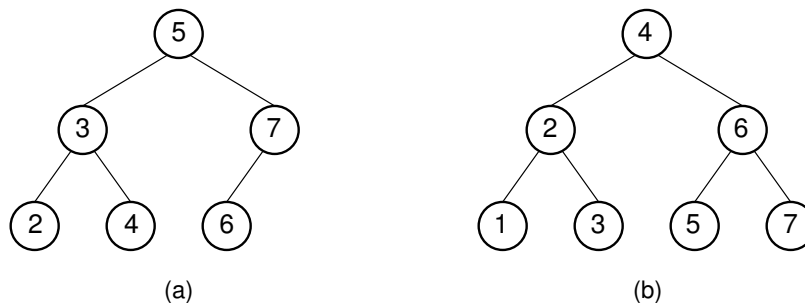


Figure 10.8 An attempt to re-balance a BST after insertion can be expensive. (a) A BST with six nodes in the shape of a complete binary tree. (b) A node with value 1 is inserted into the BST of (a). To maintain both the complete binary tree shape and the BST property, a major reorganization of the tree is required.

BST. The first is how to keep the tree balanced. The second is how to arrange the nodes on blocks so as to keep the number of blocks encountered on any path from the root to the leaves at a minimum.

We could select a scheme for balancing the BST and allocating BST nodes to blocks in a way that minimizes disk I/O, as illustrated by Figure 10.7. However, maintaining such a scheme in the face of insertions and deletions is difficult. In particular, the tree should remain balanced when an update takes place, but doing so might require much reorganization. Each update should affect only a few blocks, or its cost will be too high. As you can see from Figure 10.8, adopting a rule such as requiring the BST to be complete can cause a great deal of rearranging of data within the tree.

We can solve these problems by selecting another tree structure that automatically remains balanced after updates, and which is amenable to storing in blocks. There are a number of balanced tree data structures, and there are also techniques for keeping BSTs balanced. Examples are the AVL and splay trees discussed in Section 13.2. As an alternative, Section 10.4 presents the **2-3 tree**, which has the property that its leaves are always at the same level. The main reason for discussing the 2-3 tree here in preference to the other balanced search trees is that it naturally

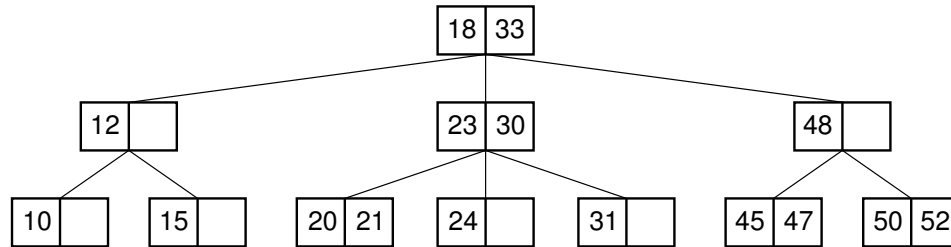


Figure 10.9 A 2-3 tree.

leads to the B-tree of Section 10.5, which is by far the most widely used indexing method today.

10.4 2-3 Trees

This section presents a data structure called the 2-3 tree. The 2-3 tree is not a binary tree, but instead its shape obeys the following definition:

1. A node contains one or two keys.
2. Every internal node has either two children (if it contains one key) or three children (if it contains two keys). Hence the name.
3. All leaves are at the same level in the tree, so the tree is always height balanced.

In addition to these shape properties, the 2-3 tree has a search tree property analogous to that of a BST. For every node, the values of all descendants in the left subtree are less than the value of the first key, while values in the center subtree are greater than or equal to the value of the first key. If there is a right subtree (equivalently, if the node stores two keys), then the values of all descendants in the center subtree are less than the value of the second key, while values in the right subtree are greater than or equal to the value of the second key. To maintain these shape and search properties requires that special action be taken when nodes are inserted and deleted. The 2-3 tree has the advantage over the BST in that the 2-3 tree can be kept height balanced at relatively low cost.

Figure 10.9 illustrates the 2-3 tree. Nodes are shown as rectangular boxes with two key fields. (These nodes actually would contain complete records or pointers to complete records, but the figures will show only the keys.) Internal nodes with only two children have an empty right key field. Leaf nodes might contain either one or two keys. Figure 10.10 is an implementation for the 2-3 tree node. Class **TTNode** is assumed to be a private class of the 2-3 tree class **TTTree**, and thus the data members of **TTNode** will be made public to simplify the presentation.

Note that this sample declaration does not distinguish between leaf and internal nodes and so is space inefficient, because leaf nodes store three pointers each. The

```

template <typename Key, typename E>
class TTreeNode {          // 2-3 tree node structure
public:
    E lval;                // The node's left record
    Key lkey;              // Left record's key
    E rval;                // The node's right record
    Key rkey;              // Right record's key
    TTreeNode* left;       // Pointer to left child
    TTreeNode* center;     // Pointer to middle child
    TTreeNode* right;      // Pointer to right child
    TTreeNode() {
        center = left = right = NULL;
        lkey = rkey = EMPTYKEY;
    }
    TTreeNode(Key lk, E lv, Key rk, E rv, TTreeNode<Key,E>* p1,
              TTreeNode<Key,E>* p2, TTreeNode<Key,E>* p3) {
        lkey = lk; rkey = rk;
        lval = lv; rval = rv;
        left = p1; center = p2; right = p3;
    }
    ~TTreeNode() { }
    bool isLeaf() { return left == NULL; }
    TTreeNode<Key,E>* add(TTreeNode<Key,E>* it);
};

```

Figure 10.10 The 2-3 tree node implementation.

techniques of Section 5.3.1 can be applied here to implement separate internal and leaf node types.

From the defining rules for 2-3 trees we can derive relationships between the number of nodes in the tree and the depth of the tree. A 2-3 tree of height k has at least 2^{k-1} leaves, because if every internal node has two children it degenerates to the shape of a complete binary tree. A 2-3 tree of height k has at most 3^{k-1} leaves, because each internal node can have at most three children.

Searching for a value in a 2-3 tree is similar to searching in a BST. Search begins at the root. If the root does not contain the search key K , then the search progresses to the only subtree that can possibly contain K . The value(s) stored in the root node determine which is the correct subtree. For example, if searching for the value 30 in the tree of Figure 10.9, we begin with the root node. Because 30 is between 18 and 33, it can only be in the middle subtree. Searching the middle child of the root node yields the desired record. If searching for 15, then the first step is again to search the root node. Because 15 is less than 18, the first (left) branch is taken. At the next level, we take the second branch to the leaf node containing 15. If the search key were 16, then upon encountering the leaf containing 15 we would find that the search key is not in the tree. Figure 10.11 is an implementation for the 2-3 tree search method.

```

// Find the record that matches a given key value
template <typename Key, typename E>
E TTree<Key, E>::
findhelp(TTNode<Key,E>* root, Key k) const {
    if (root == NULL) return NULL;           // value not found
    if (k == root->lkey) return root->lval;
    if (k == root->rkey) return root->rval;
    if (k < root->lkey)                        // Go left
        return findhelp(root->left, k);
    else if (root->rkey == EMPTYKEY)           // 2 child node
        return findhelp(root->center, k);     // Go center
    else if (k < root->rkey)
        return findhelp(root->center, k);     // Go center
    else return findhelp(root->right, k);     // Go right
}

```

Figure 10.11 Implementation for the 2-3 tree search method.

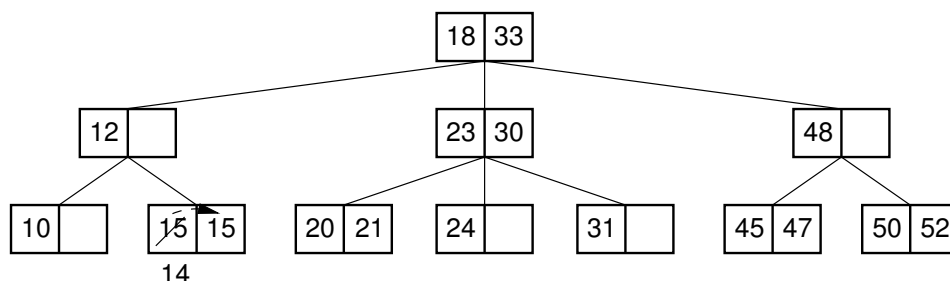


Figure 10.12 Simple insert into the 2-3 tree of Figure 10.9. The value 14 is inserted into the tree at the leaf node containing 15. Because there is room in the node for a second key, it is simply added to the left position with 15 moved to the right position.

Insertion into a 2-3 tree is similar to insertion into a BST to the extent that the new record is placed in the appropriate leaf node. Unlike BST insertion, a new child is not created to hold the record being inserted, that is, the 2-3 tree does not grow downward. The first step is to find the leaf node that would contain the record if it were in the tree. If this leaf node contains only one value, then the new record can be added to that node with no further modification to the tree, as illustrated in Figure 10.12. In this example, a record with key value 14 is inserted. Searching from the root, we come to the leaf node that stores 15. We add 14 as the left value (pushing the record with key 15 to the rightmost position).

If we insert the new record into a leaf node L that already contains two records, then more space must be created. Consider the two records of node L and the record to be inserted without further concern for which two were already in L and which is the new record. The first step is to split L into two nodes. Thus, a new node — call it L' — must be created from free store. L receives the record with the least of the three key values. L' receives the greatest of the three. The record

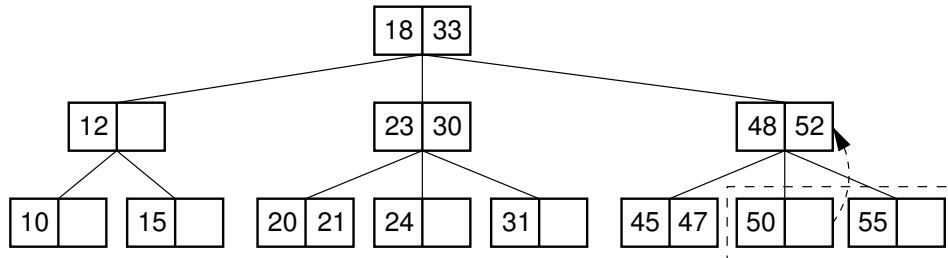


Figure 10.13 A simple node-splitting insert for a 2-3 tree. The value 55 is added to the 2-3 tree of Figure 10.9. This makes the node containing values 50 and 52 split, promoting value 52 to the parent node.

with the middle of the three key value is passed up to the parent node along with a pointer to L' . This is called a **promotion**. The promoted key is then inserted into the parent. If the parent currently contains only one record (and thus has only two children), then the promoted record and the pointer to L' are simply added to the parent node. If the parent is full, then the split-and-promote process is repeated. Figure 10.13 illustrates a simple promotion. Figure 10.14 illustrates what happens when promotions require the root to split, adding a new level to the tree. In either case, all leaf nodes continue to have equal depth. Figures 10.15 and 10.16 present an implementation for the insertion process.

Note that **inserthelp** of Figure 10.15 takes three parameters. The first is a pointer to the root of the current subtree, named **rt**. The second is the key for the record to be inserted, and the third is the record itself. The return value for **inserthelp** is a pointer to a 2-3 tree node. If **rt** is unchanged, then a pointer to **rt** is returned. If **rt** is changed (due to the insertion causing the node to split), then a pointer to the new subtree root is returned, with the key value and record value in the leftmost fields, and a pointer to the (single) subtree in the center pointer field. This revised node will then be added to the parent, as illustrated in Figure 10.14.

When deleting a record from the 2-3 tree, there are three cases to consider. The simplest occurs when the record is to be removed from a leaf node containing two records. In this case, the record is simply removed, and no other nodes are affected. The second case occurs when the only record in a leaf node is to be removed. The third case occurs when a record is to be removed from an internal node. In both the second and the third cases, the deleted record is replaced with another that can take its place while maintaining the correct order, similar to removing a node from a BST. If the tree is sparse enough, there is no such record available that will allow all nodes to still maintain at least one record. In this situation, sibling nodes are merged together. The delete operation for the 2-3 tree is excessively complex and will not be described further. Instead, a complete discussion of deletion will be postponed until the next section, where it can be generalized for a particular variant of the B-tree.

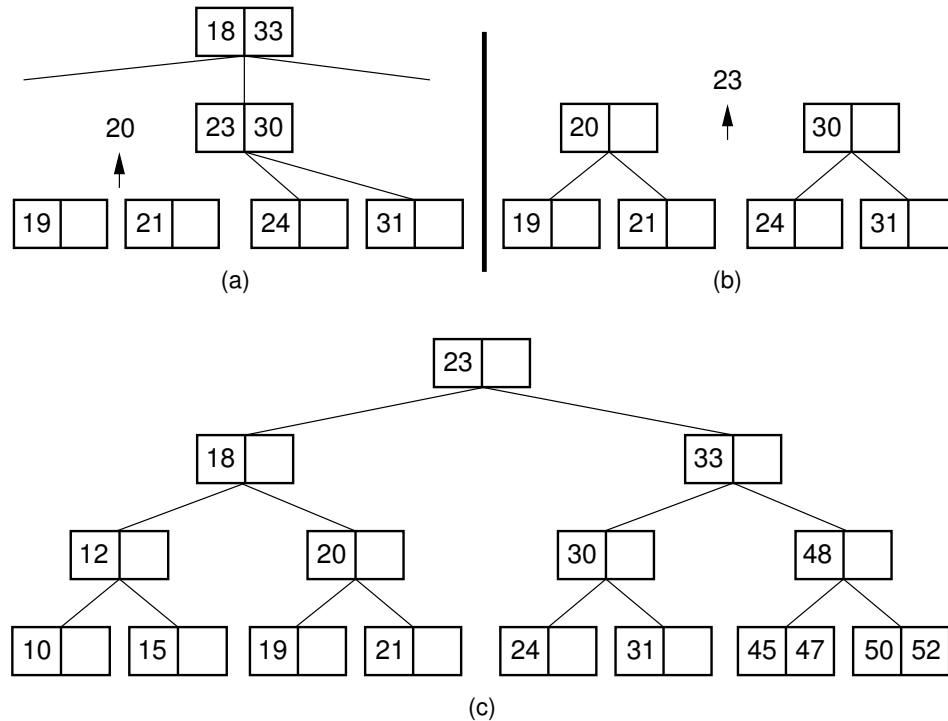


Figure 10.14 Example of inserting a record that causes the 2-3 tree root to split. (a) The value 19 is added to the 2-3 tree of Figure 10.9. This causes the node containing 20 and 21 to split, promoting 20. (b) This in turn causes the internal node containing 23 and 30 to split, promoting 23. (c) Finally, the root node splits, promoting 23 to become the left record in the new root. The result is that the tree becomes one level higher.

The 2-3 tree insert and delete routines do not add new nodes at the bottom of the tree. Instead they cause leaf nodes to split or merge, possibly causing a ripple effect moving up the tree to the root. If necessary the root will split, causing a new root node to be created and making the tree one level deeper. On deletion, if the last two children of the root merge, then the root node is removed and the tree will lose a level. In either case, all leaf nodes are always at the same level. When all leaf nodes are at the same level, we say that a tree is **height balanced**. Because the 2-3 tree is height balanced, and every internal node has at least two children, we know that the maximum depth of the tree is $\log n$. Thus, all 2-3 tree insert, find, and delete operations require $\Theta(\log n)$ time.

10.5 B-Trees

This section presents the B-tree. B-trees are usually attributed to R. Bayer and E. McCreight who described the B-tree in a 1972 paper. By 1979, B-trees had re-


```

template <typename Key, typename E>
TTNode<Key,E>* TTTTree<Key, E>::
inserthelp(TTNode<Key,E>* rt, const Key k, const E e) {
    TTNode<Key,E>* retval;
    if (rt == NULL) // Empty tree: create a leaf node for root
        return new TTNode<Key,E>(k, e, EMPTYKEY, NULL,
                                   NULL, NULL, NULL);
    if (rt->isLeaf()) // At leaf node: insert here
        return rt->add(new TTNode<Key,E>(k, e, EMPTYKEY, NULL,
                                           NULL, NULL, NULL));

    // Add to internal node
    if (k < rt->lkey) {
        retval = inserthelp(rt->left, k, e);
        if (retval == rt->left) return rt;
        else return rt->add(retval);
    }
    else if((rt->rkey == EMPTYKEY) || (k < rt->rkey)) {
        retval = inserthelp(rt->center, k, e);
        if (retval == rt->center) return rt;
        else return rt->add(retval);
    }
    else { // Insert right
        retval = inserthelp(rt->right, k, e);
        if (retval == rt->right) return rt;
        else return rt->add(retval);
    }
}

```

Figure 10.15 The 2-3 tree insert routine.

placed virtually all large-file access methods other than hashing. B-trees, or some variant of B-trees, are *the* standard file organization for applications requiring insertion, deletion, and key range searches. They are used to implement most modern file systems. B-trees address effectively all of the major problems encountered when implementing disk-based search trees:

1. B-trees are always height balanced, with all leaf nodes at the same level.
2. Update and search operations affect only a few disk blocks. The fewer the number of disk blocks affected, the less disk I/O is required.
3. B-trees keep related records (that is, records with similar key values) on the same disk block, which helps to minimize disk I/O on searches due to locality of reference.
4. B-trees guarantee that every node in the tree will be full at least to a certain minimum percentage. This improves space efficiency while reducing the typical number of disk fetches necessary during a search or update operation.

A B-tree of order m is defined to have the following shape properties:

- The root is either a leaf or has at least two children.
- Each internal node, except for the root, has between $\lceil m/2 \rceil$ and m children.

```

// Add a new key/value pair to the node. There might be a
// subtree associated with the record being added. This
// information comes in the form of a 2-3 tree node with
// one key and a (possibly NULL) subtree through the
// center pointer field.
template <typename Key, typename E>
TTNode<Key,E>* TTNode<Key, E>::add(TTNode<Key,E>* it) {
    if (rkey == EMPTYKEY) { // Only one key, add here
        if (lkey < it->lkey) {
            rkey = it->lkey; rval = it->lval;
            right = center; center = it->center;
        }
        else {
            rkey = lkey; rval = lval; right = center;
            lkey = it->lkey; lval = it->lval;
            center = it->center;
        }
        return this;
    }
    else if (lkey >= it->lkey) { // Add left
        center = new TTNode<Key,E>(rkey, rval, EMPTYKEY, NULL,
                                   center, right, NULL);
        rkey = EMPTYKEY; rval = NULL; right = NULL;
        it->left = left; left = it;
        return this;
    }
    else if (rkey < it->lkey) { // Add center
        it->center = new TTNode<Key,E>(rkey, rval, EMPTYKEY,
                                     NULL, it->center, right, NULL);
        it->left = this;
        rkey = EMPTYKEY; rval = NULL; right = NULL;
        return it;
    }
    else { // Add right
        TTNode<Key,E>* N1 = new TTNode<Key,E>(rkey, rval,
                                               EMPTYKEY, NULL, this, it, NULL);
        it->left = right;
        right = NULL; rkey = EMPTYKEY; rval = NULL;
        return N1;
    }
}

```

Figure 10.16 The 2-3 tree node **add** method.

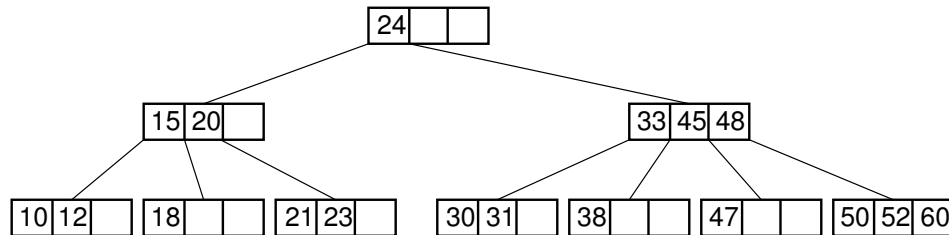


Figure 10.17 A B-tree of order four.

- All leaves are at the same level in the tree, so the tree is always height balanced.

The B-tree is a generalization of the 2-3 tree. Put another way, a 2-3 tree is a B-tree of order three. Normally, the size of a node in the B-tree is chosen to fill a disk block. A B-tree node implementation typically allows 100 or more children. Thus, a B-tree node is equivalent to a disk block, and a “pointer” value stored in the tree is actually the number of the block containing the child node (usually interpreted as an offset from the beginning of the corresponding disk file). In a typical application, the B-tree’s access to the disk file will be managed using a buffer pool and a block-replacement scheme such as LRU (see Section 8.3).

Figure 10.17 shows a B-tree of order four. Each node contains up to three keys, and internal nodes have up to four children.

Search in a B-tree is a generalization of search in a 2-3 tree. It is an alternating two-step process, beginning with the root node of the B-tree.

1. Perform a binary search on the records in the current node. If a record with the search key is found, then return that record. If the current node is a leaf node and the key is not found, then report an unsuccessful search.
2. Otherwise, follow the proper branch and repeat the process.

For example, consider a search for the record with key value 47 in the tree of Figure 10.17. The root node is examined and the second (right) branch taken. After examining the node at level 1, the third branch is taken to the next level to arrive at the leaf node containing a record with key value 47.

B-tree insertion is a generalization of 2-3 tree insertion. The first step is to find the leaf node that should contain the key to be inserted, space permitting. If there is room in this node, then insert the key. If there is not, then split the node into two and promote the middle key to the parent. If the parent becomes full, then it is split in turn, and its middle key promoted.

Note that this insertion process is guaranteed to keep all nodes at least half full. For example, when we attempt to insert into a full internal node of a B-tree of order four, there will now be five children that must be dealt with. The node is split into

two nodes containing two keys each, thus retaining the B-tree property. The middle of the five children is promoted to its parent.

10.5.1 B⁺-Trees

The previous section mentioned that B-trees are universally used to implement large-scale disk-based systems. Actually, the B-tree as described in the previous section is almost never implemented, nor is the 2-3 tree as described in Section 10.4. What is most commonly implemented is a variant of the B-tree, called the B⁺-tree. When greater efficiency is required, a more complicated variant known as the B*-tree is used.

When data are static, a linear index provides an extremely efficient way to search. The problem is how to handle those pesky inserts and deletes. We could try to keep the core idea of storing a sorted array-based list, but make it more flexible by breaking the list into manageable chunks that are more easily updated. How might we do that? First, we need to decide how big the chunks should be. Since the data are on disk, it seems reasonable to store a chunk that is the size of a disk block, or a small multiple of the disk block size. If the next record to be inserted belongs to a chunk that hasn't filled its block then we can just insert it there. The fact that this might cause other records in that chunk to move a little bit in the array is not important, since this does not cause any extra disk accesses so long as we move data within that chunk. But what if the chunk fills up the entire block that contains it? We could just split it in half. What if we want to delete a record? We could just take the deleted record out of the chunk, but we might not want a lot of near-empty chunks. So we could put adjacent chunks together if they have only a small amount of data between them. Or we could shuffle data between adjacent chunks that together contain more data. The big problem would be how to find the desired chunk when processing a record with a given key. Perhaps some sort of tree-like structure could be used to locate the appropriate chunk. These ideas are exactly what motivate the B⁺-tree. The B⁺-tree is essentially a mechanism for managing a sorted array-based list, where the list is broken into chunks.

The most significant difference between the B⁺-tree and the BST or the standard B-tree is that the B⁺-tree stores records only at the leaf nodes. Internal nodes store key values, but these are used solely as placeholders to guide the search. This means that internal nodes are significantly different in structure from leaf nodes. Internal nodes store keys to guide the search, associating each key with a pointer to a child B⁺-tree node. Leaf nodes store actual records, or else keys and pointers to actual records in a separate disk file if the B⁺-tree is being used purely as an index. Depending on the size of a record as compared to the size of a key, a leaf node in a B⁺-tree of order m might have enough room to store more or less than m records. The requirement is simply that the leaf nodes store enough records to remain at least half full. The leaf nodes of a B⁺-tree are normally linked together

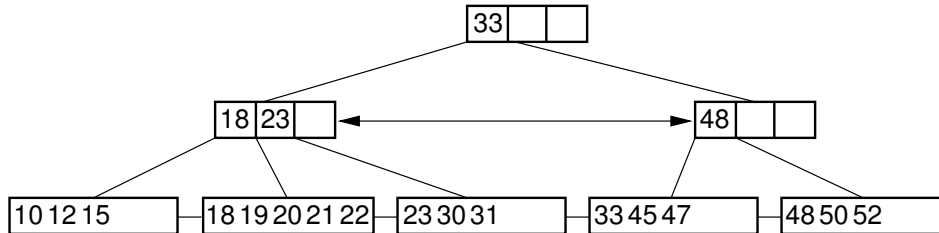


Figure 10.18 Example of a B^+ -tree of order four. Internal nodes must store between two and four children. For this example, the record size is assumed to be such that leaf nodes store between three and five records.

to form a doubly linked list. Thus, the entire collection of records can be traversed in sorted order by visiting all the leaf nodes on the linked list. Here is a C++-like pseudocode representation for the B^+ -tree node class. Leaf node and internal node subclasses would implement this base class.

```
// Abstract class definition for B+-trees
template <typename Key, typename E>class BPNode {
public:
    BPNode* lftpnt; BPNode* rhtpnt; // Links to siblings
    virtual ~BPNode() {} // Base destructor
    virtual bool isLeaf() const =0; // True if node is a leaf
    virtual bool isFull() const =0; // True if node is full
    virtual int numrecs() const =0; // Current num of records
    virtual Key* keys() const=0; // Return array of keys
};
```

An important implementation detail to note is that while Figure 10.17 shows internal nodes containing three keys and four pointers, class **BPNode** is slightly different in that it stores key/pointer pairs. Figure 10.17 shows the B^+ -tree as it is traditionally drawn. To simplify implementation in practice, nodes really do associate a key with each pointer. Each internal node should be assumed to hold in the leftmost position an additional key that is less than or equal to any possible key value in the node's leftmost subtree. B^+ -tree implementations typically store an additional dummy record in the leftmost leaf node whose key value is less than any legal key value.

B^+ -trees are exceptionally good for range queries. Once the first record in the range has been found, the rest of the records with keys in the range can be accessed by sequential processing of the remaining records in the first node, and then continuing down the linked list of leaf nodes as far as necessary. Figure 10.18 illustrates the B^+ -tree.

Search in a B^+ -tree is nearly identical to search in a regular B-tree, except that the search must always continue to the proper leaf node. Even if the search-key value is found in an internal node, this is only a placeholder and does not provide

```

template <typename Key, typename E>
E BPTree<Key, E>::findhelp(BPNode<Key,E>* rt, const Key k)
    const {
    int currec = binaryle(rt->keys(), rt->numrecs(), k);
    if (rt->isLeaf())
        if (((BPLeaf<Key,E>*)rt)->keys())[currec] == k)
            return ((BPLeaf<Key,E>*)rt)->recs(currec);
        else return NULL;
    else
        return findhelp(((BPInternal<Key,E>*)rt)->
                        pointers(currec), k);
    }

```

Figure 10.19 Implementation for the B^+ -tree search method.

access to the actual record. To find a record with key value 33 in the B^+ -tree of Figure 10.18, search begins at the root. The value 33 stored in the root merely serves as a placeholder, indicating that keys with values greater than or equal to 33 are found in the second subtree. From the second child of the root, the first branch is taken to reach the leaf node containing the actual record (or a pointer to the actual record) with key value 33. Figure 10.19 shows a pseudocode sketch of the B^+ -tree search algorithm.

B^+ -tree insertion is similar to B-tree insertion. First, the leaf L that should contain the record is found. If L is not full, then the new record is added, and no other B^+ -tree nodes are affected. If L is already full, split it in two (dividing the records evenly among the two nodes) and promote a copy of the least-valued key in the newly formed right node. As with the 2-3 tree, promotion might cause the parent to split in turn, perhaps eventually leading to splitting the root and causing the B^+ -tree to gain a new level. B^+ -tree insertion keeps all leaf nodes at equal depth. Figure 10.20 illustrates the insertion process through several examples. Figure 10.21 shows a C++-like pseudocode sketch of the B^+ -tree insert algorithm.

To delete record R from the B^+ -tree, first locate the leaf L that contains R . If L is more than half full, then we need only remove R , leaving L still at least half full. This is demonstrated by Figure 10.22.

If deleting a record reduces the number of records in the node below the minimum threshold (called an **underflow**), then we must do something to keep the node sufficiently full. The first choice is to look at the node's adjacent siblings to determine if they have a spare record that can be used to fill the gap. If so, then enough records are transferred from the sibling so that both nodes have about the same number of records. This is done so as to delay as long as possible the next time when a delete causes this node to underflow again. This process might require that the parent node has its placeholder key value revised to reflect the true first key value in each node. Figure 10.23 illustrates the process.

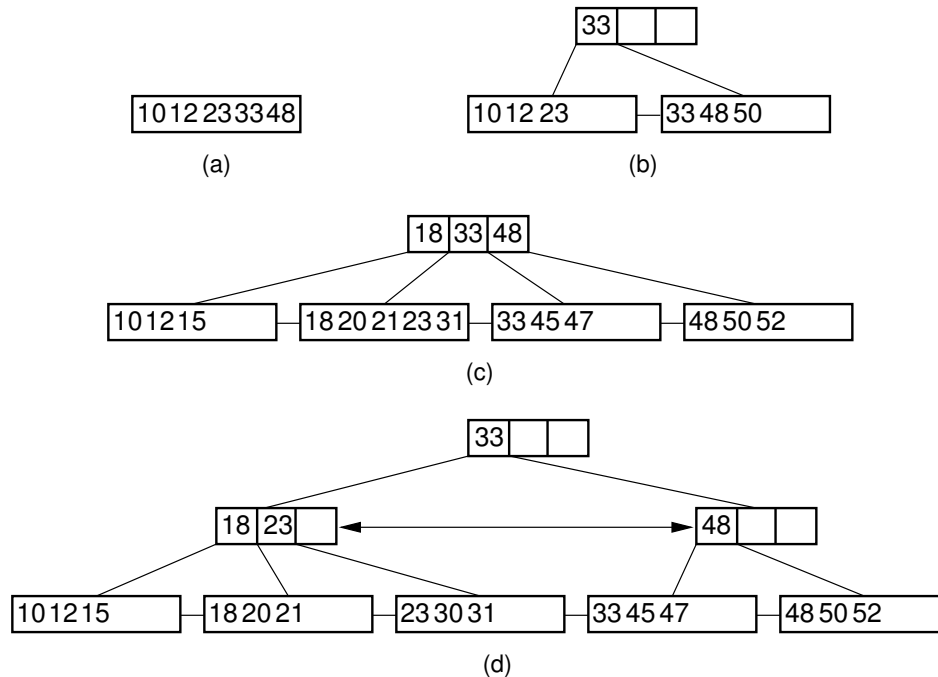


Figure 10.20 Examples of B⁺-tree insertion. (a) A B⁺-tree containing five records. (b) The result of inserting a record with key value 50 into the tree of (a). The leaf node splits, causing creation of the first internal node. (c) The B⁺-tree of (b) after further insertions. (d) The result of inserting a record with key value 30 into the tree of (c). The second leaf node splits, which causes the internal node to split in turn, creating a new root.

```

template <typename Key, typename E>
BPNode<Key,E>* BPTree<Key, E>::inserthelp(BPNode<Key,E>* rt,
      const Key& k, const E& e) {
    if (rt->isLeaf()) // At leaf node: insert here
        return ((BPLeaf<Key,E>*)rt)->add(k, e);
    // Add to internal node
    int currec = binaryle(rt->keys(), rt->numrecs(), k);
    BPNode<Key,E>* temp = inserthelp(
        ((BPInternal<Key,E>*)rt)->pointers(currec), k, e);
    if (temp != ((BPInternal<Key,E>*)rt)->pointers(currec))
        return ((BPInternal<Key,E>*)rt)->
            add(k, ((BPInternal<Key,E>*)temp));
    else
        return rt;
}

```

Figure 10.21 A C++-like pseudocode sketch of the B⁺-tree insert algorithm.

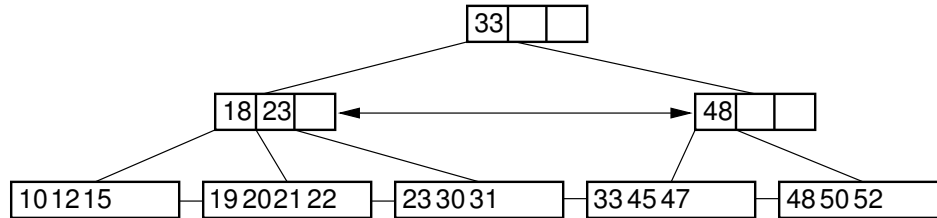


Figure 10.22 Simple deletion from a B^+ -tree. The record with key value 18 is removed from the tree of Figure 10.18. Note that even though 18 is also a placeholder used to direct search in the parent node, that value need not be removed from internal nodes even if no record in the tree has key value 18. Thus, the leftmost node at level one in this example retains the key with value 18 after the record with key value 18 has been removed from the second leaf node.

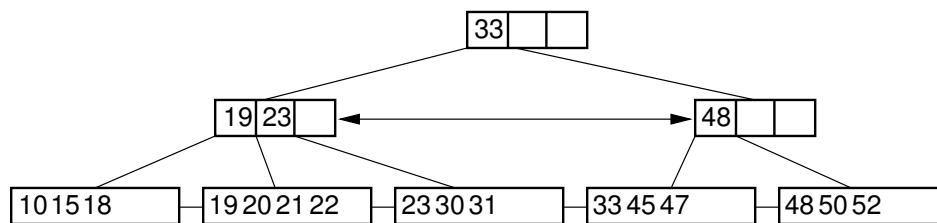


Figure 10.23 Deletion from the B^+ -tree of Figure 10.18 via borrowing from a sibling. The key with value 12 is deleted from the leftmost leaf, causing the record with key value 18 to shift to the leftmost leaf to take its place. Note that the parent must be updated to properly indicate the key range within the subtrees. In this example, the parent node has its leftmost key value changed to 19.

If neither sibling can lend a record to the under-full node (call it N), then N must give its records to a sibling and be removed from the tree. There is certainly room to do this, because the sibling is at most half full (remember that it had no records to contribute to the current node), and N has become less than half full because it is under-flowing. This merge process combines two subtrees of the parent, which might cause it to underflow in turn. If the last two children of the root merge together, then the tree loses a level. Figure 10.24 illustrates the node-merge deletion process. Figure 10.25 shows $C++$ -like pseudocode for the B^+ -tree delete algorithm.

The B^+ -tree requires that all nodes be at least half full (except for the root). Thus, the storage utilization must be at least 50%. This is satisfactory for many implementations, but note that keeping nodes fuller will result both in less space required (because there is less empty space in the disk file) and in more efficient processing (fewer blocks on average will be read into memory because the amount of information in each block is greater). Because B-trees have become so popular, many algorithm designers have tried to improve B-tree performance. One method

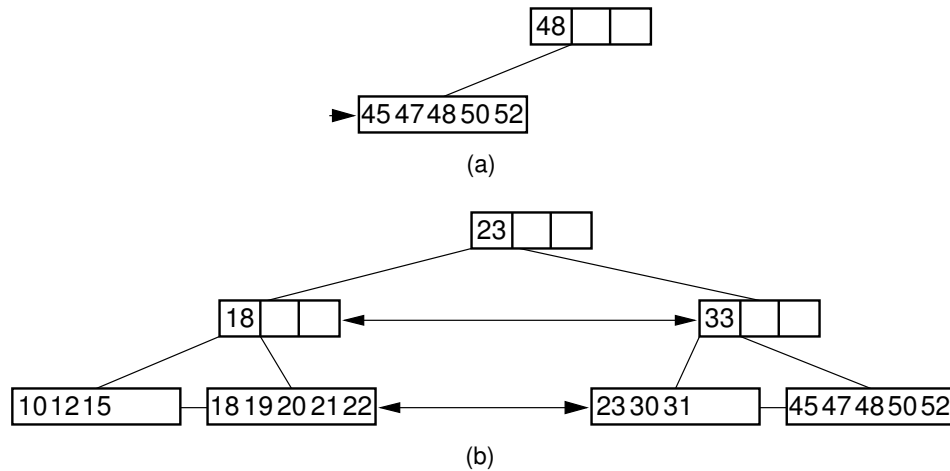


Figure 10.24 Deleting the record with key value 33 from the B⁺-tree of Figure 10.18 via collapsing siblings. (a) The two leftmost leaf nodes merge together to form a single leaf. Unfortunately, the parent node now has only one child. (b) Because the left subtree has a spare leaf node, that node is passed to the right subtree. The placeholder values of the root and the right internal node are updated to reflect the changes. Value 23 moves to the root, and old root value 33 moves to the rightmost internal node.

```

/** Delete a record with the given key value, and
    return true if the root underflows */
template <typename Key, typename E>
bool BPTree<Key, E>::removehelp(BPNode<Key,E>* rt,
                                const Key& k) {
    int currec = binaryle(rt->keys(), rt->numrecs(), k);
    if (rt->isLeaf())
        if (((BPLeaf<Key,E>*)rt)->keys()[currec] == k)
            return ((BPLeaf<Key,E>*)rt)->del(currec);
        else return false;
    else // Process internal node
        if (removehelp(((BPInternal<Key,E>*)rt)->
                        pointers(currec), k))
            // Child will merge if necessary
            return ((BPInternal<Key,E>*)rt)->underflow(currec);
        else return false;
}

```

Figure 10.25 C++-like pseudocode for the B⁺-tree delete algorithm.

for doing so is to use the B^+ -tree variant known as the B^* -tree. The B^* -tree is identical to the B^+ -tree, except for the rules used to split and merge nodes. Instead of splitting a node in half when it overflows, the B^* -tree gives some records to its neighboring sibling, if possible. If the sibling is also full, then these two nodes split into three. Similarly, when a node underflows, it is combined with its two siblings, and the total reduced to two nodes. Thus, the nodes are always at least two thirds full.²

10.5.2 B-Tree Analysis

The asymptotic cost of search, insertion, and deletion of records from B-trees, B^+ -trees, and B^* -trees is $\Theta(\log n)$ where n is the total number of records in the tree. However, the base of the log is the (average) branching factor of the tree. Typical database applications use extremely high branching factors, perhaps 100 or more. Thus, in practice the B-tree and its variants are extremely shallow.

As an illustration, consider a B^+ -tree of order 100 and leaf nodes that contain up to 100 records. A B^+ -tree with height one (that is, just a single leaf node) can have at most 100 records. A B^+ -tree with height two (a root internal node whose children are leaves) must have at least 100 records (2 leaves with 50 records each). It has at most 10,000 records (100 leaves with 100 records each). A B^+ -tree with height three must have at least 5000 records (two second-level nodes with 50 children containing 50 records each) and at most one million records (100 second-level nodes with 100 full children each). A B^+ -tree with height four must have at least 250,000 records and at most 100 million records. Thus, it would require an *extremely* large database to generate a B^+ -tree of more than height four.

The B^+ -tree split and insert rules guarantee that every node (except perhaps the root) is at least half full. So they are on average about $3/4$ full. But the internal nodes are purely overhead, since the keys stored there are used only by the tree to direct search, rather than store actual data. Does this overhead amount to a significant use of space? No, because once again the high fan-out rate of the tree structure means that the vast majority of nodes are leaf nodes. Recall (from Section 6.4) that a full K -ary tree has approximately $1/K$ of its nodes as internal nodes. This means that while half of a full binary tree's nodes are internal nodes, in a B^+ -tree of order 100 probably only about $1/75$ of its nodes are internal nodes. This means that the overhead associated with internal nodes is very low.

We can reduce the number of disk fetches required for the B-tree even more by using the following methods. First, the upper levels of the tree can be stored in

² This concept can be extended further if higher space utilization is required. However, the update routines become much more complicated. I once worked on a project where we implemented 3-for-4 node split and merge routines. This gave better performance than the 2-for-3 node split and merge routines of the B^* -tree. However, the spitting and merging routines were so complicated that even their author could no longer understand them once they were completed!

main memory at all times. Because the tree branches so quickly, the top two levels (levels 0 and 1) require relatively little space. If the B-tree is only height four, then at most two disk fetches (internal nodes at level two and leaves at level three) are required to reach the pointer to any given record.

A buffer pool could be used to manage nodes of the B-tree. Several nodes of the tree would typically be in main memory at one time. The most straightforward approach is to use a standard method such as LRU to do node replacement. However, sometimes it might be desirable to “lock” certain nodes such as the root into the buffer pool. In general, if the buffer pool is even of modest size (say at least twice the depth of the tree), no special techniques for node replacement will be required because the upper-level nodes will naturally be accessed frequently.

10.6 Further Reading

For an expanded discussion of the issues touched on in this chapter, see a general file processing text such as *File Structures: A Conceptual Toolkit* by Folk and Zoellick [FZ98]. In particular, Folk and Zoellick provide a good discussion of the relationship between primary and secondary indices. The most thorough discussion on various implementations for the B-tree is the survey article by Comer [Com79]. Also see [Sal88] for further details on implementing B-trees. See Shaffer and Brown [SB93] for a discussion of buffer pool management strategies for B⁺-tree-like data structures.

10.7 Exercises

- 10.1** Assume that a computer system has disk blocks of 1024 bytes, and that you are storing records that have 4-byte keys and 4-byte data fields. The records are sorted and packed sequentially into the disk file.
- (a) Assume that a linear index uses 4 bytes to store the key and 4 bytes to store the block ID for the associated records. What is the greatest number of records that can be stored in the file if a linear index of size 256KB is used?
 - (b) What is the greatest number of records that can be stored in the file if the linear index is also stored on disk (and thus its size is limited only by the second-level index) when using a second-level index of 1024 bytes (i.e., 256 key values) as illustrated by Figure 10.2? Each element of the second-level index references the smallest key value for a disk block of the linear index.
- 10.2** Assume that a computer system has disk blocks of 4096 bytes, and that you are storing records that have 4-byte keys and 64-byte data fields. The records are sorted and packed sequentially into the disk file.

- (a) Assume that a linear index uses 4 bytes to store the key and 4 bytes to store the block ID for the associated records. What is the greatest number of records that can be stored in the file if a linear index of size 2MB is used?
- (b) What is the greatest number of records that can be stored in the file if the linear index is also stored on disk (and thus its size is limited only by the second-level index) when using a second-level index of 4096 bytes (i.e., 1024 key values) as illustrated by Figure 10.2? Each element of the second-level index references the smallest key value for a disk block of the linear index.

10.3 Modify the function **binary** of Section 3.5 so as to support variable-length records with fixed-length keys indexed by a simple linear index as illustrated by Figure 10.1.

10.4 Assume that a database stores records consisting of a 2-byte integer key and a variable-length data field consisting of a string. Show the linear index (as illustrated by Figure 10.1) for the following collection of records:

397	Hello world!
82	XYZ
1038	This string is rather long
1037	This is shorter
42	ABC
2222	Hello new world!

10.5 Each of the following series of records consists of a four-digit primary key (with no duplicates) and a four-character secondary key (with many duplicates).

3456	DEER
2398	DEER
2926	DUCK
9737	DEER
7739	GOAT
9279	DUCK
1111	FROG
8133	DEER
7183	DUCK
7186	FROG

- (a) Show the inverted list (as illustrated by Figure 10.4) for this collection of records.
- (b) Show the improved inverted list (as illustrated by Figure 10.5) for this collection of records.

- 10.6** Under what conditions will ISAM be more efficient than a B^+ -tree implementation?
- 10.7** Prove that the number of leaf nodes in a 2-3 tree with height k is between 2^{k-1} and 3^{k-1} .
- 10.8** Show the result of inserting the values 55 and 46 into the 2-3 tree of Figure 10.9.
- 10.9** You are given a series of records whose keys are letters. The records arrive in the following order: C, S, D, T, A, M, P, I, B, W, N, G, U, R, K, E, H, O, L, J. Show the 2-3 tree that results from inserting these records.
- 10.10** You are given a series of records whose keys are letters. The records are inserted in the following order: C, S, D, T, A, M, P, I, B, W, N, G, U, R, K, E, H, O, L, J. Show the tree that results from inserting these records when the 2-3 tree is modified to be a $2\text{-}3^+$ tree, that is, the internal nodes act only as placeholders. Assume that the leaf nodes are capable of holding up to two records.
- 10.11** Show the result of inserting the value 55 into the B-tree of Figure 10.17.
- 10.12** Show the result of inserting the values 1, 2, 3, 4, 5, and 6 (in that order) into the B^+ -tree of Figure 10.18.
- 10.13** Show the result of deleting the values 18, 19, and 20 (in that order) from the B^+ -tree of Figure 10.24b.
- 10.14** You are given a series of records whose keys are letters. The records are inserted in the following order: C, S, D, T, A, M, P, I, B, W, N, G, U, R, K, E, H, O, L, J. Show the B^+ -tree of order four that results from inserting these records. Assume that the leaf nodes are capable of storing up to three records.
- 10.15** Assume that you have a B^+ -tree whose internal nodes can store up to 100 children and whose leaf nodes can store up to 15 records. What are the minimum and maximum number of records that can be stored by the B^+ -tree with heights 1, 2, 3, 4, and 5?
- 10.16** Assume that you have a B^+ -tree whose internal nodes can store up to 50 children and whose leaf nodes can store up to 50 records. What are the minimum and maximum number of records that can be stored by the B^+ -tree with heights 1, 2, 3, 4, and 5?

10.8 Projects

- 10.1** Implement a two-level linear index for variable-length records as illustrated by Figures 10.1 and 10.2. Assume that disk blocks are 1024 bytes in length. Records in the database file should typically range between 20 and 200 bytes, including a 4-byte key value. Each record of the index file should store a key value and the byte offset in the database file for the first byte of the

corresponding record. The top-level index (stored in memory) should be a simple array storing the lowest key value on the corresponding block in the index file.

- 10.2** Implement the $2\text{-}3^+$ tree, that is, a 2-3 tree where the internal nodes act only as placeholders. Your $2\text{-}3^+$ tree should implement the dictionary interface of Section 4.4.
- 10.3** Implement the dictionary ADT of Section 4.4 for a large file stored on disk by means of the B^+ -tree of Section 10.5. Assume that disk blocks are 1024 bytes, and thus both leaf nodes and internal nodes are also 1024 bytes. Records should store a 4-byte (**int**) key value and a 60-byte data field. Internal nodes should store key value/pointer pairs where the “pointer” is actually the block number on disk for the child node. Both internal nodes and leaf nodes will need room to store various information such as a count of the records stored on that node, and a pointer to the next node on that level. Thus, leaf nodes will store 15 records, and internal nodes will have room to store about 120 to 125 children depending on how you implement them. Use a buffer pool (Section 8.3) to manage access to the nodes stored on disk.

PART IV

Advanced Data Structures

Graphs

Graphs provide the ultimate in data structure flexibility. Graphs can model both real-world systems and abstract problems, so they are used in hundreds of applications. Here is a small sampling of the range of problems that graphs are routinely applied to.

1. Modeling connectivity in computer and communications networks.
2. Representing a map as a set of locations with distances between locations; used to compute shortest routes between locations.
3. Modeling flow capacities in transportation networks.
4. Finding a path from a starting condition to a goal condition; for example, in artificial intelligence problem solving.
5. Modeling computer algorithms, showing transitions from one program state to another.
6. Finding an acceptable order for finishing subtasks in a complex activity, such as constructing large buildings.
7. Modeling relationships such as family trees, business or military organizations, and scientific taxonomies.

We begin in Section 11.1 with some basic graph terminology and then define two fundamental representations for graphs, the adjacency matrix and adjacency list. Section 11.2 presents a graph ADT and simple implementations based on the adjacency matrix and adjacency list. Section 11.3 presents the two most commonly used graph traversal algorithms, called depth-first and breadth-first search, with application to topological sorting. Section 11.4 presents algorithms for solving some problems related to finding shortest routes in a graph. Finally, Section 11.5 presents algorithms for finding the minimum-cost spanning tree, useful for determining lowest-cost connectivity in a network. Besides being useful and interesting in their own right, these algorithms illustrate the use of some data structures presented in earlier chapters.

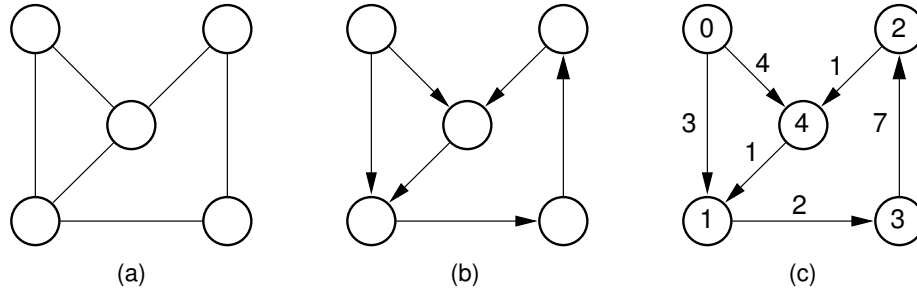


Figure 11.1 Examples of graphs and terminology. (a) A graph. (b) A directed graph (digraph). (c) A labeled (directed) graph with weights associated with the edges. In this example, there is a simple path from Vertex 0 to Vertex 3 containing Vertices 0, 1, and 3. Vertices 0, 1, 3, 2, 4, and 1 also form a path, but not a simple path because Vertex 1 appears twice. Vertices 1, 3, 2, 4, and 1 form a simple cycle.

11.1 Terminology and Representations

A graph $G = (V, E)$ consists of a set of vertices V and a set of edges E , such that each edge in E is a connection between a pair of vertices in V .¹ The number of vertices is written $|V|$, and the number of edges is written $|E|$. $|E|$ can range from zero to a maximum of $|V|^2 - |V|$. A graph with relatively few edges is called **sparse**, while a graph with many edges is called **dense**. A graph containing all possible edges is said to be **complete**.

A graph with edges directed from one vertex to another (as in Figure 11.1(b)) is called a **directed graph** or **digraph**. A graph whose edges are not directed is called an **undirected graph** (as illustrated by Figure 11.1(a)). A graph with labels associated with its vertices (as in Figure 11.1(c)) is called a **labeled graph**. Two vertices are **adjacent** if they are joined by an edge. Such vertices are also called **neighbors**. An edge connecting Vertices U and V is written (U, V) . Such an edge is said to be **incident** on Vertices U and V . Associated with each edge may be a cost or **weight**. Graphs whose edges have weights (as in Figure 11.1(c)) are said to be **weighted**.

A sequence of vertices v_1, v_2, \dots, v_n forms a **path** of length $n - 1$ if there exist edges from v_i to v_{i+1} for $1 \leq i < n$. A path is **simple** if all vertices on the path are distinct. The **length** of a path is the number of edges it contains. A **cycle** is a path of length three or more that connects some vertex v_1 to itself. A cycle is **simple** if the path is simple, except for the first and last vertices being the same.

¹Some graph applications require that a given pair of vertices can have multiple or parallel edges connecting them, or that a vertex can have an edge to itself. However, the applications discussed in this book do not require either of these special cases, so for simplicity we will assume that they cannot occur.

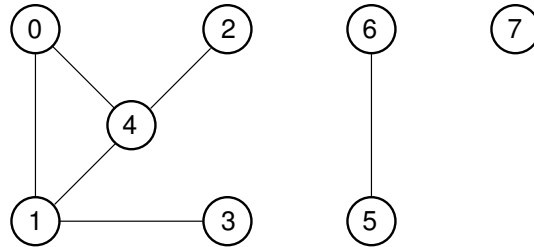


Figure 11.2 An undirected graph with three connected components. Vertices 0, 1, 2, 3, and 4 form one connected component. Vertices 5 and 6 form a second connected component. Vertex 7 by itself forms a third connected component.

A **subgraph** S is formed from graph G by selecting a subset V_s of G 's vertices and a subset E_s of G 's edges such that for every edge E in E_s , both of E 's vertices are in V_s .

An undirected graph is **connected** if there is at least one path from any vertex to any other. The maximally connected subgraphs of an undirected graph are called **connected components**. For example, Figure 11.2 shows an undirected graph with three connected components.

A graph without cycles is called **acyclic**. Thus, a directed graph without cycles is called a **directed acyclic graph** or DAG.

A **free tree** is a connected, undirected graph with no simple cycles. An equivalent definition is that a free tree is connected and has $|V| - 1$ edges.

There are two commonly used methods for representing graphs. The **adjacency matrix** is illustrated by Figure 11.3(b). The adjacency matrix for a graph is a $|V| \times |V|$ array. Assume that $|V| = n$ and that the vertices are labeled from v_0 through v_{n-1} . Row i of the adjacency matrix contains entries for Vertex v_i . Column j in row i is marked if there is an edge from v_i to v_j and is not marked otherwise. Thus, the adjacency matrix requires one bit at each position. Alternatively, if we wish to associate a number with each edge, such as the weight or distance between two vertices, then each matrix position must store that number. In either case, the space requirements for the adjacency matrix are $\Theta(|V|^2)$.

The second common representation for graphs is the **adjacency list**, illustrated by Figure 11.3(c). The adjacency list is an array of linked lists. The array is $|V|$ items long, with position i storing a pointer to the linked list of edges for Vertex v_i . This linked list represents the edges by the vertices that are adjacent to Vertex v_i . The adjacency list is therefore a generalization of the “list of children” representation for trees described in Section 6.3.1.

Example 11.1 The entry for Vertex 0 in Figure 11.3(c) stores 1 and 4 because there are two edges in the graph leaving Vertex 0, with one going

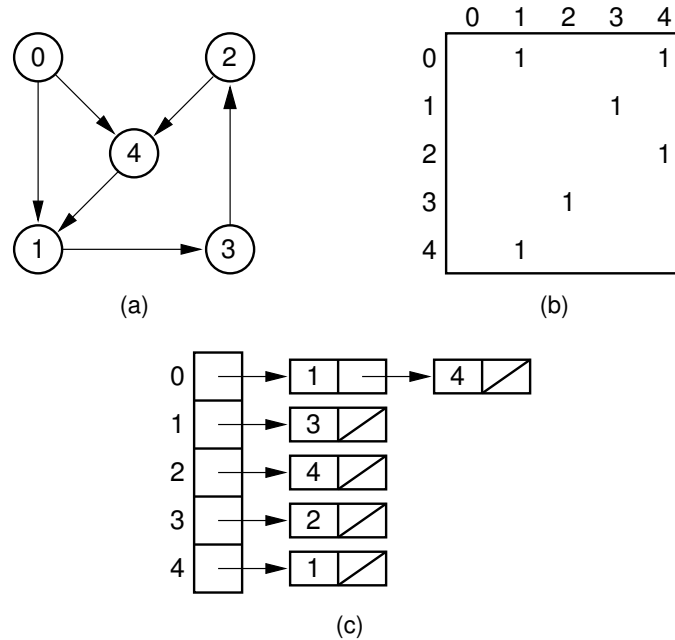


Figure 11.3 Two graph representations. (a) A directed graph. (b) The adjacency matrix for the graph of (a). (c) The adjacency list for the graph of (a).

to Vertex 1 and one going to Vertex 4. The list for Vertex 2 stores an entry for Vertex 4 because there is an edge from Vertex 2 to Vertex 4, but no entry for Vertex 3 because this edge comes into Vertex 2 rather than going out.

The storage requirements for the adjacency list depend on both the number of edges and the number of vertices in the graph. There must be an array entry for each vertex (even if the vertex is not adjacent to any other vertex and thus has no elements on its linked list), and each edge must appear on one of the lists. Thus, the cost is $\Theta(|V| + |E|)$.

Both the adjacency matrix and the adjacency list can be used to store directed or undirected graphs. Each edge of an undirected graph connecting Vertices U and V is represented by two directed edges: one from U to V and one from V to U . Figure 11.4 illustrates the use of the adjacency matrix and the adjacency list for undirected graphs.

Which graph representation is more space efficient depends on the number of edges in the graph. The adjacency list stores information only for those edges that actually appear in the graph, while the adjacency matrix requires space for each potential edge, whether it exists or not. However, the adjacency matrix requires no overhead for pointers, which can be a substantial cost, especially if the only

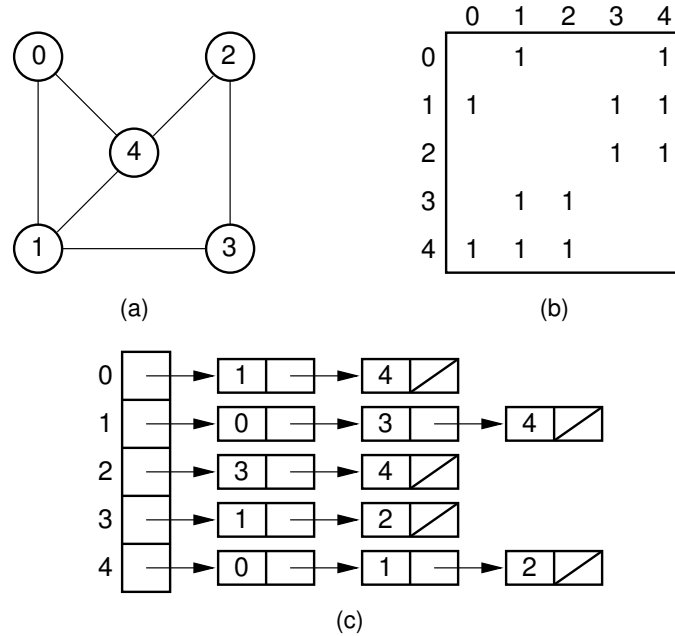


Figure 11.4 Using the graph representations for undirected graphs. (a) An undirected graph. (b) The adjacency matrix for the graph of (a). (c) The adjacency list for the graph of (a).

information stored for an edge is one bit to indicate its existence. As the graph becomes denser, the adjacency matrix becomes relatively more space efficient. Sparse graphs are likely to have their adjacency list representation be more space efficient.

Example 11.2 Assume that a vertex index requires two bytes, a pointer requires four bytes, and an edge weight requires two bytes. Then the adjacency matrix for the graph of Figure 11.3 requires $2|\mathbf{V}^2| = 50$ bytes while the adjacency list requires $4|\mathbf{V}| + 6|\mathbf{E}| = 56$ bytes. For the graph of Figure 11.4, the adjacency matrix requires the same space as before, while the adjacency list requires $4|\mathbf{V}| + 6|\mathbf{E}| = 92$ bytes (because there are now 12 edges instead of 6).

The adjacency matrix often requires a higher asymptotic cost for an algorithm than would result if the adjacency list were used. The reason is that it is common for a graph algorithm to visit each neighbor of each vertex. Using the adjacency list, only the actual edges connecting a vertex to its neighbors are examined. However, the adjacency matrix must look at each of its $|\mathbf{V}|$ potential edges, yielding a total cost of $\Theta(|\mathbf{V}^2|)$ time when the algorithm might otherwise require only $\Theta(|\mathbf{V}| + |\mathbf{E}|)$

time. This is a considerable disadvantage when the graph is sparse, but not when the graph is closer to full.

11.2 Graph Implementations

We next turn to the problem of implementing a general-purpose graph class. Figure 11.5 shows an abstract class defining an ADT for graphs. Vertices are defined by an integer index value. In other words, there is a Vertex 0, Vertex 1, and so on. We can assume that a graph application stores any additional information of interest about a given vertex elsewhere, such as a name or application-dependent value. Note that this ADT is not implemented using a template, because it is the **Graph** class users' responsibility to maintain information related to the vertices themselves. The **Graph** class need have no knowledge of the type or content of the information associated with a vertex, only the index number for that vertex.

Abstract class **Graph** has methods to return the number of vertices and edges (methods **n** and **e**, respectively). Function **weight** returns the weight of a given edge, with that edge identified by its two incident vertices. For example, calling **weight(0, 4)** on the graph of Figure 11.1 (c) would return 4. If no such edge exists, the weight is defined to be 0. So calling **weight(0, 2)** on the graph of Figure 11.1 (c) would return 0.

Functions **setEdge** and **delEdge** set the weight of an edge and remove an edge from the graph, respectively. Again, an edge is identified by its two incident vertices. **setEdge** does not permit the user to set the weight to be 0, because this value is used to indicate a non-existent edge, nor are negative edge weights permitted. Functions **getMark** and **setMark** get and set, respectively, a requested value in the **Mark** array (described below) for Vertex *V*.

Nearly every graph algorithm presented in this chapter will require visits to all neighbors of a given vertex. Two methods are provided to support this. They work in a manner similar to linked list access functions. Function **first** takes as input a vertex *V*, and returns the edge to the first neighbor for *V* (we assume the neighbor list is sorted by vertex number). Function **next** takes as input Vertices *V1* and *V2* and returns the index for the vertex forming the next edge with *V1* after *V2* on *V1*'s edge list. Function **next** will return a value of $n = |V|$ once the end of the edge list for *V1* has been reached. The following line appears in many graph algorithms:

```
for (w = G->first(v); w < G->n(); w = G->next(v,w))
```

This **for** loop gets the first neighbor of **v**, then works through the remaining neighbors of **v** until a value equal to **G->n()** is returned, signaling that all neighbors of **v** have been visited. For example, **first(1)** in Figure 11.4 would return 0. **next(1, 0)** would return 3. **next(0, 3)** would return 4. **next(1, 4)** would return 5, which is not a vertex in the graph.

```

// Graph abstract class. This ADT assumes that the number
// of vertices is fixed when the graph is created.
class Graph {
private:
    void operator =(const Graph&) {}      // Protect assignment
    Graph(const Graph&) {}                // Protect copy constructor

public:
    Graph() {}                            // Default constructor
    virtual ~Graph() {}                  // Base destructor

    // Initialize a graph of n vertices
    virtual void Init(int n) =0;

    // Return: the number of vertices and edges
    virtual int n() =0;
    virtual int e() =0;

    // Return v's first neighbor
    virtual int first(int v) =0;

    // Return v's next neighbor
    virtual int next(int v, int w) =0;

    // Set the weight for an edge
    // i, j: The vertices
    // wgt: Edge weight
    virtual void setEdge(int v1, int v2, int wgt) =0;

    // Delete an edge
    // i, j: The vertices
    virtual void delEdge(int v1, int v2) =0;

    // Determine if an edge is in the graph
    // i, j: The vertices
    // Return: true if edge i,j has non-zero weight
    virtual bool isEdge(int i, int j) =0;

    // Return an edge's weight
    // i, j: The vertices
    // Return: The weight of edge i,j, or zero
    virtual int weight(int v1, int v2) =0;

    // Get and Set the mark value for a vertex
    // v: The vertex
    // val: The value to set
    virtual int getMark(int v) =0;
    virtual void setMark(int v, int val) =0;
};

```

Figure 11.5 A graph ADT. This ADT assumes that the number of vertices is fixed when the graph is created, but that edges can be added and removed. It also supports a mark array to aid graph traversal algorithms.

It is reasonably straightforward to implement our graph and edge ADTs using either the adjacency list or adjacency matrix. The sample implementations presented here do not address the issue of how the graph is actually created. The user of these implementations must add functionality for this purpose, perhaps reading the graph description from a file. The graph can be built up by using the **setEdge** function provided by the ADT.

Figure 11.6 shows an implementation for the adjacency matrix. Array **Mark** stores the information manipulated by the **setMark** and **getMark** functions. The edge matrix is implemented as an integer array of size $n \times n$ for a graph of n vertices. Position (i, j) in the matrix stores the weight for edge (i, j) if it exists. A weight of zero for edge (i, j) is used to indicate that no edge connects Vertices i and j .

Given a vertex V , function **first** locates the position in **matrix** of the first edge (if any) of V by beginning with edge $(V, 0)$ and scanning through row V until an edge is found. If no edge is incident on V , then **first** returns n .

Function **next** locates the edge following edge (i, j) (if any) by continuing down the row of Vertex i starting at position $j + 1$, looking for an edge. If no such edge exists, **next** returns n . Functions **setEdge** and **delEdge** adjust the appropriate value in the array. Function **weight** returns the value stored in the appropriate position in the array.

Figure 11.7 presents an implementation of the adjacency list representation for graphs. Its main data structure is an array of linked lists, one linked list for each vertex. These linked lists store objects of type **Edge**, which merely stores the index for the vertex pointed to by the edge, along with the weight of the edge. Because the **Edge** class is assumed to be private to the **Graph1** class, its data members have been made public for convenience.

```
// Edge class for Adjacency List graph representation
class Edge {
    int vert, wt;
public:
    Edge() { vert = -1; wt = -1; }
    Edge(int v, int w) { vert = v; wt = w; }
    int vertex() { return vert; }
    int weight() { return wt; }
};
```

Implementation for **Graph1** member functions is straightforward in principle, with the key functions being **setEdge**, **delEdge**, and **weight**. They simply start at the beginning of the adjacency list and move along it until the desired vertex has been found. Note that **isEdge** checks to see if j is already the current neighbor in i 's adjacency list, since this will often be true when processing the neighbors of each vertex in turn.


```

// Implementation for the adjacency matrix representation
class Graphm : public Graph {
private:
    int numVertex, numEdge; // Store number of vertices, edges
    int **matrix;           // Pointer to adjacency matrix
    int *mark;              // Pointer to mark array
public:
    Graphm(int numVert)     // Constructor
    { Init(numVert); }

    ~Graphm() {           // Destructor
        delete [] mark; // Return dynamically allocated memory
        for (int i=0; i<numVertex; i++)
            delete [] matrix[i];
        delete [] matrix;
    }

    void Init(int n) { // Initialize the graph
        int i;
        numVertex = n;
        numEdge = 0;
        mark = new int[n]; // Initialize mark array
        for (i=0; i<numVertex; i++)
            mark[i] = UNVISITED;
        matrix = (int**) new int*[numVertex]; // Make matrix
        for (i=0; i<numVertex; i++)
            matrix[i] = new int[numVertex];
        for (i=0; i< numVertex; i++) // Initialize to 0 weights
            for (int j=0; j<numVertex; j++)
                matrix[i][j] = 0;
    }

    int n() { return numVertex; } // Number of vertices
    int e() { return numEdge; }   // Number of edges

    // Return first neighbor of "v"
    int first(int v) {
        for (int i=0; i<numVertex; i++)
            if (matrix[v][i] != 0) return i;
        return numVertex; // Return n if none
    }

    // Return v's next neighbor after w
    int next(int v, int w) {
        for(int i=w+1; i<numVertex; i++)
            if (matrix[v][i] != 0)
                return i;
        return numVertex; // Return n if none
    }
}

```

Figure 11.6 An implementation for the adjacency matrix implementation.

```

// Set edge (v1, v2) to "wt"
void setEdge(int v1, int v2, int wt) {
    Assert(wt>0, "Illegal weight value");
    if (matrix[v1][v2] == 0) numEdge++;
    matrix[v1][v2] = wt;
}

void delEdge(int v1, int v2) { // Delete edge (v1, v2)
    if (matrix[v1][v2] != 0) numEdge--;
    matrix[v1][v2] = 0;
}

bool isEdge(int i, int j) // Is (i, j) an edge?
{ return matrix[i][j] != 0; }

int weight(int v1, int v2) { return matrix[v1][v2]; }
int getMark(int v) { return mark[v]; }
void setMark(int v, int val) { mark[v] = val; }
};

```

Figure 11.6 (continued)

11.3 Graph Traversals

Often it is useful to visit the vertices of a graph in some specific order based on the graph's topology. This is known as a **graph traversal** and is similar in concept to a tree traversal. Recall that tree traversals visit every node exactly once, in some specified order such as preorder, inorder, or postorder. Multiple tree traversals exist because various applications require the nodes to be visited in a particular order. For example, to print a BST's nodes in ascending order requires an inorder traversal as opposed to some other traversal. Standard graph traversal orders also exist. Each is appropriate for solving certain problems. For example, many problems in artificial intelligence programming are modeled using graphs. The problem domain may consist of a large collection of states, with connections between various pairs of states. Solving the problem may require getting from a specified start state to a specified goal state by moving between states only through the connections. Typically, the start and goal states are not directly connected. To solve this problem, the vertices of the graph must be searched in some organized manner.

Graph traversal algorithms typically begin with a start vertex and attempt to visit the remaining vertices from there. Graph traversals must deal with a number of troublesome cases. First, it may not be possible to reach all vertices from the start vertex. This occurs when the graph is not connected. Second, the graph may contain cycles, and we must make sure that cycles do not cause the algorithm to go into an infinite loop.

```

class Graph1 : public Graph {
private:
    List<Edge>** vertex;           // List headers
    int numVertex, numEdge;       // Number of vertices, edges
    int *mark;                   // Pointer to mark array
public:
    Graph1(int numVert)
    { Init(numVert); }

    ~Graph1() {                // Destructor
        delete [] mark; // Return dynamically allocated memory
        for (int i=0; i<numVertex; i++) delete [] vertex[i];
        delete [] vertex;
    }

    void Init(int n) {
        int i;
        numVertex = n;
        numEdge = 0;
        mark = new int[n]; // Initialize mark array
        for (i=0; i<numVertex; i++) mark[i] = UNVISITED;
        // Create and initialize adjacency lists
        vertex = (List<Edge>**) new List<Edge>*[numVertex];
        for (i=0; i<numVertex; i++)
            vertex[i] = new LList<Edge>();
    }

    int n() { return numVertex; } // Number of vertices
    int e() { return numEdge; }   // Number of edges

    int first(int v) { // Return first neighbor of "v"
        if (vertex[v]->length() == 0)
            return numVertex; // No neighbor
        vertex[v]->moveToStart();
        Edge it = vertex[v]->getValue();
        return it.vertex();
    }

    // Get v's next neighbor after w
    int next(int v, int w) {
        Edge it;
        if (isEdge(v, w)) {
            if ((vertex[v]->currPos()+1) < vertex[v]->length()) {
                vertex[v]->next();
                it = vertex[v]->getValue();
                return it.vertex();
            }
        }
        return n(); // No neighbor
    }
}

```

Figure 11.7 An implementation for the adjacency list.

```

// Set edge (i, j) to "weight"
void setEdge(int i, int j, int weight) {
    Assert(weight>0, "May not set weight to 0");
    Edge currEdge(j, weight);
    if (isEdge(i, j)) { // Edge already exists in graph
        vertex[i]->remove();
        vertex[i]->insert(currEdge);
    }
    else { // Keep neighbors sorted by vertex index
        numEdge++;
        for (vertex[i]->moveToStart();
             vertex[i]->currPos() < vertex[i]->length();
             vertex[i]->next()) {
            Edge temp = vertex[i]->getValue();
            if (temp.vertex() > j) break;
        }
        vertex[i]->insert(currEdge);
    }
}

void delEdge(int i, int j) { // Delete edge (i, j)
    if (isEdge(i, j)) {
        vertex[i]->remove();
        numEdge--;
    }
}

bool isEdge(int i, int j) { // Is (i, j) an edge?
    Edge it;
    for (vertex[i]->moveToStart();
         vertex[i]->currPos() < vertex[i]->length();
         vertex[i]->next()) { // Check whole list
        Edge temp = vertex[i]->getValue();
        if (temp.vertex() == j) return true;
    }
    return false;
}

int weight(int i, int j) { // Return weight of (i, j)
    Edge curr;
    if (isEdge(i, j)) {
        curr = vertex[i]->getValue();
        return curr.weight();
    }
    else return 0;
}

int getMark(int v) { return mark[v]; }
void setMark(int v, int val) { mark[v] = val; }
};

```

Figure 11.7 (continued)

Graph traversal algorithms can solve both of these problems by maintaining a **mark bit** for each vertex on the graph. At the beginning of the algorithm, the mark bit for all vertices is cleared. The mark bit for a vertex is set when the vertex is first visited during the traversal. If a marked vertex is encountered during traversal, it is not visited a second time. This keeps the program from going into an infinite loop when it encounters a cycle.

Once the traversal algorithm completes, we can check to see if all vertices have been processed by checking the mark bit array. If not all vertices are marked, we can continue the traversal from another unmarked vertex. Note that this process works regardless of whether the graph is directed or undirected. To ensure visiting all vertices, **graphTraverse** could be called as follows on a graph **G**:

```
void graphTraverse(Graph* G) {
    int v;
    for (v=0; v<G->n(); v++)
        G->setMark(v, UNVISITED); // Initialize mark bits
    for (v=0; v<G->n(); v++)
        if (G->getMark(v) == UNVISITED)
            doTraverse(G, v);
}
```

Function “**doTraverse**” might be implemented by using one of the graph traversals described in this section.

11.3.1 Depth-First Search

The first method of organized graph traversal is called **depth-first search** (DFS). Whenever a vertex V is visited during the search, DFS will recursively visit all of V 's unvisited neighbors. Equivalently, DFS will add all edges leading out of v to a stack. The next vertex to be visited is determined by popping the stack and following that edge. The effect is to follow one branch through the graph to its conclusion, then it will back up and follow another branch, and so on. The DFS process can be used to define a **depth-first search tree**. This tree is composed of the edges that were followed to any new (unvisited) vertex during the traversal, and leaves out the edges that lead to already visited vertices. DFS can be applied to directed or undirected graphs. Here is an implementation for the DFS algorithm:

```
void DFS(Graph* G, int v) { // Depth first search
    PreVisit(G, v);         // Take appropriate action
    G->setMark(v, VISITED);
    for (int w=G->first(v); w<G->n(); w = G->next(v,w))
        if (G->getMark(w) == UNVISITED)
            DFS(G, w);
    PostVisit(G, v);        // Take appropriate action
}
```

This implementation contains calls to functions **PreVisit** and **PostVisit**. These functions specify what activity should take place during the search. Just

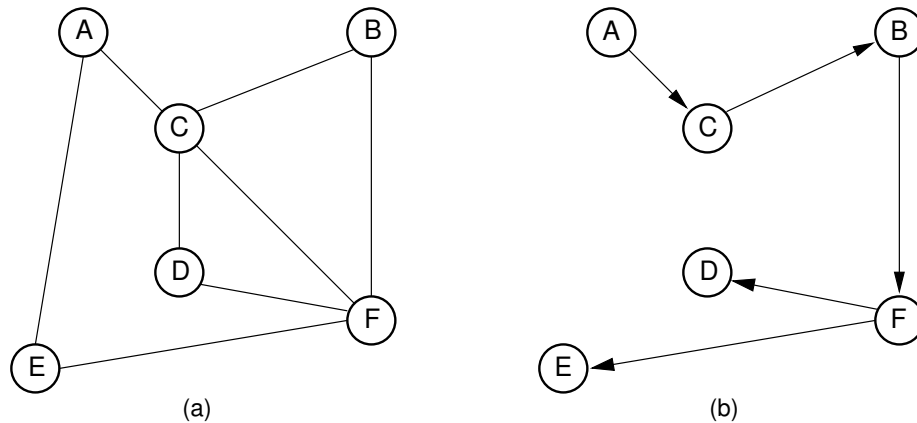


Figure 11.8 (a) A graph. (b) The depth-first search tree for the graph when starting at Vertex A.

as a preorder tree traversal requires action before the subtrees are visited, some graph traversals require that a vertex be processed before ones further along in the DFS. Alternatively, some applications require activity *after* the remaining vertices are processed; hence the call to function **PostVisit**. This would be a natural opportunity to make use of the visitor design pattern described in Section 1.3.2.

Figure 11.8 shows a graph and its corresponding depth-first search tree. Figure 11.9 illustrates the DFS process for the graph of Figure 11.8(a).

DFS processes each edge once in a directed graph. In an undirected graph, DFS processes each edge from both directions. Each vertex must be visited, but only once, so the total cost is $\Theta(|V| + |E|)$.

11.3.2 Breadth-First Search

Our second graph traversal algorithm is known as a **breadth-first search** (BFS). BFS examines all vertices connected to the start vertex before visiting vertices further away. BFS is implemented similarly to DFS, except that a queue replaces the recursion stack. Note that if the graph is a tree and the start vertex is at the root, BFS is equivalent to visiting vertices level by level from top to bottom. Figure 11.10 provides an implementation for the BFS algorithm. Figure 11.11 shows a graph and the corresponding breadth-first search tree. Figure 11.12 illustrates the BFS process for the graph of Figure 11.11(a).

11.3.3 Topological Sort

Assume that we need to schedule a series of tasks, such as classes or construction jobs, where we cannot start one task until after its prerequisites are completed. We wish to organize the tasks into a linear order that allows us to complete them one

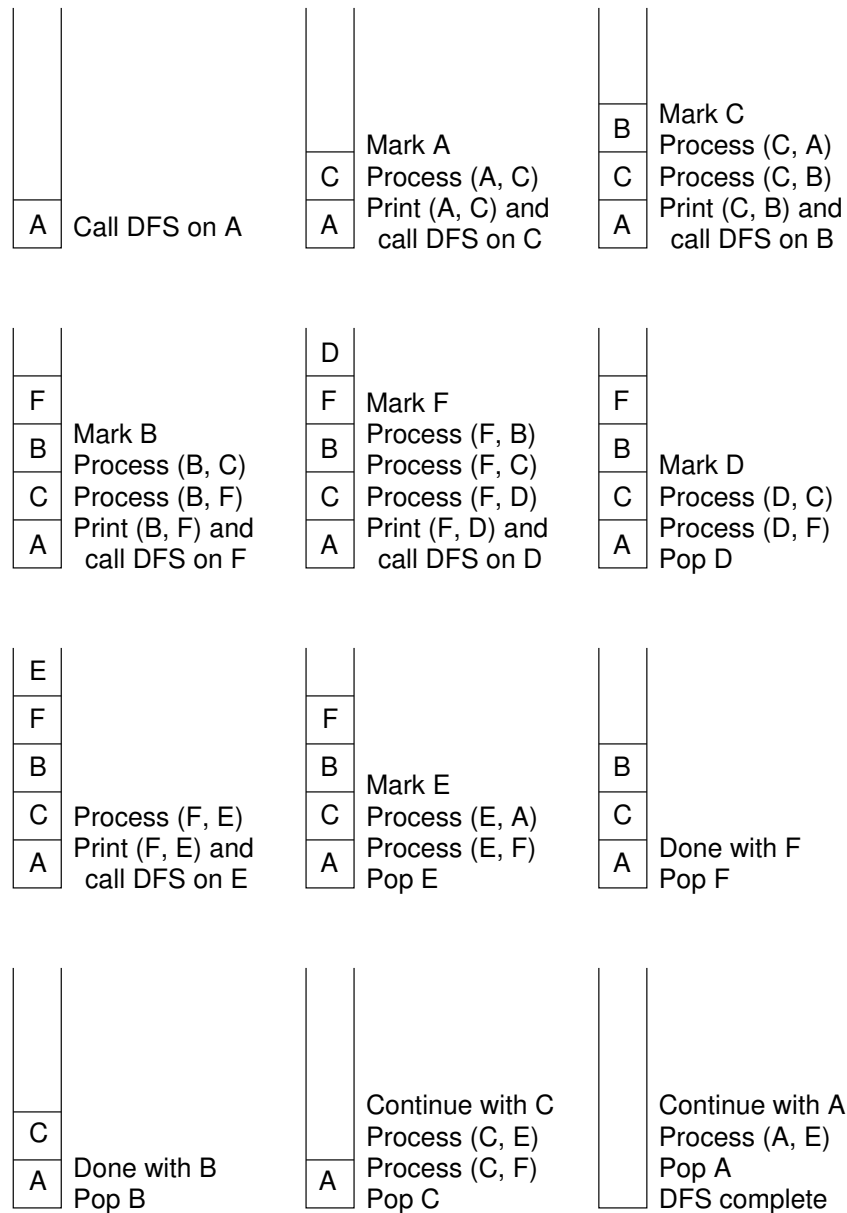


Figure 11.9 A detailed illustration of the DFS process for the graph of Figure 11.8(a) starting at Vertex A. The steps leading to each change in the recursion stack are described.

```

void BFS(Graph* G, int start, Queue<int>* Q) {
    int v, w;
    Q->enqueue(start);           // Initialize Q
    G->setMark(start, VISITED);
    while (Q->length() != 0) { // Process all vertices on Q
        v = Q->dequeue();
        PreVisit(G, v);         // Take appropriate action
        for (w=G->first(v); w<G->n(); w = G->next(v,w))
            if (G->getMark(w) == UNVISITED) {
                G->setMark(w, VISITED);
                Q->enqueue(w);
            }
    }
}

```

Figure 11.10 Implementation for the breadth-first graph traversal algorithm

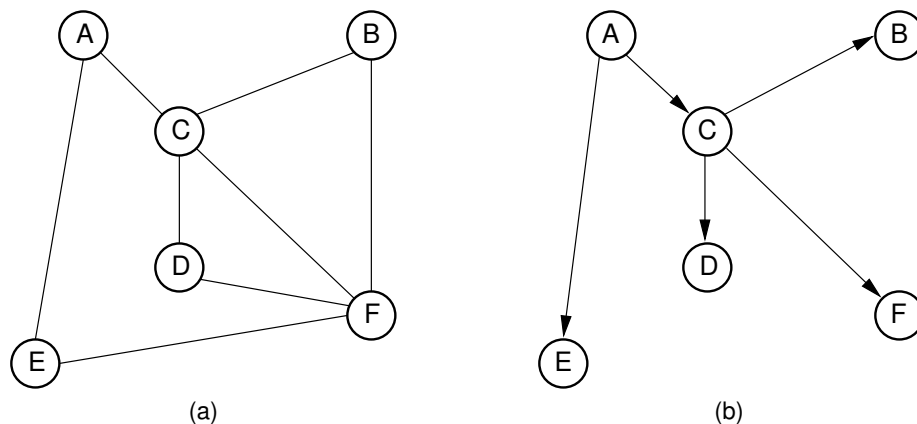


Figure 11.11 (a) A graph. (b) The breadth-first search tree for the graph when starting at Vertex A.

at a time without violating any prerequisites. We can model the problem using a DAG. The graph is directed because one task is a prerequisite of another — the vertices have a directed relationship. It is acyclic because a cycle would indicate a conflicting series of prerequisites that could not be completed without violating at least one prerequisite. The process of laying out the vertices of a DAG in a linear order to meet the prerequisite rules is called a **topological sort**. Figure 11.14 illustrates the problem. An acceptable topological sort for this example is *J1, J2, J3, J4, J5, J6, J7*.

A topological sort may be found by performing a DFS on the graph. When a vertex is visited, no action is taken (i.e., function **PreVisit** does nothing). When the recursion pops back to that vertex, function **PostVisit** prints the vertex. This yields a topological sort in reverse order. It does not matter where the sort starts, as long as all vertices are visited in the end. Figure 11.13 shows an implementation for the DFS-based algorithm.

A	
---	--

Initial call to BFS on A.
Mark A and put on the queue.

C	E	
---	---	--

Dequeue A.
Process (A, C).
Mark and enqueue C. Print (A, C)
Process (A, E).
Mark and enqueue E. Print(A, E).

E	B	D	F
---	---	---	---

Dequeue C.
Process (C, A). Ignore.
Process (C, B).
Mark and enqueue B. Print (C, B).
Process (C, D).
Mark and enqueue D. Print (C, D).
Process (C, F).
Mark and enqueue F. Print (C, F).

B	D	F	
---	---	---	--

Dequeue E.
Process (E, A). Ignore.
Process (E, F). Ignore.

D	F	
---	---	--

Dequeue B.
Process (B, C). Ignore.
Process (B, F). Ignore.

F	
---	--

Dequeue D.
Process (D, C). Ignore.
Process (D, F). Ignore.

--	--

Dequeue F.
Process (F, B). Ignore.
Process (F, C). Ignore.
Process (F, D). Ignore.
BFS is complete.

Figure 11.12 A detailed illustration of the BFS process for the graph of Figure 11.11(a) starting at Vertex A. The steps leading to each change in the queue are described.

```

void topsort(Graph* G) {    // Topological sort: recursive
    int i;
    for (i=0; i<G->n(); i++) // Initialize Mark array
        G->setMark(i, UNVISITED);
    for (i=0; i<G->n(); i++) // Process all vertices
        if (G->getMark(i) == UNVISITED)
            tophelp(G, i);    // Call recursive helper function
}

void tophelp(Graph* G, int v) { // Process vertex v
    G->setMark(v, VISITED);
    for (int w=G->first(v); w<G->n(); w = G->next(v,w))
        if (G->getMark(w) == UNVISITED)
            tophelp(G, w);
    printout(v);                // PostVisit for Vertex v
}

```

Figure 11.13 Implementation for the recursive topological sort.

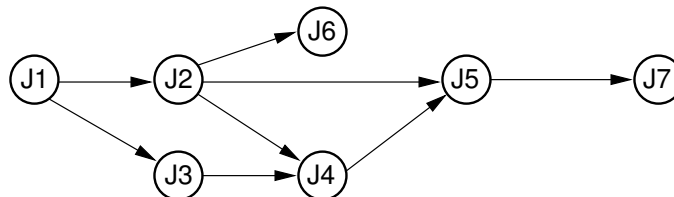


Figure 11.14 An example graph for topological sort. Seven tasks have dependencies as shown by the directed graph.

Using this algorithm starting at *J1* and visiting adjacent neighbors in alphabetic order, vertices of the graph in Figure 11.14 are printed out in the order *J7, J5, J4, J6, J2, J3, J1*. Reversing this yields the topological sort *J1, J3, J2, J6, J4, J5, J7*.

We can implement topological sort using a queue instead of recursion, as follows. First visit all edges, counting the number of edges that lead to each vertex (i.e., count the number of prerequisites for each vertex). All vertices with no prerequisites are placed on the queue. We then begin processing the queue. When Vertex *V* is taken off of the queue, it is printed, and all neighbors of *V* (that is, all vertices that have *V* as a prerequisite) have their counts decremented by one. Place on the queue any neighbor whose count becomes zero. If the queue becomes empty without printing all of the vertices, then the graph contains a cycle (i.e., there is no possible ordering for the tasks that does not violate some prerequisite). The printed order for the vertices of the graph in Figure 11.14 using the queue version of topological sort is **J1, J2, J3, J6, J4, J5, J7**. Figure 11.15 shows an implementation for the algorithm.

```

// Topological sort: Queue
void topsort(Graph* G, Queue<int>* Q) {
    int Count[G->n()];
    int v, w;
    for (v=0; v<G->n(); v++) Count[v] = 0; // Initialize
    for (v=0; v<G->n(); v++) // Process every edge
        for (w=G->first(v); w<G->n(); w = G->next(v,w))
            Count[w]++; // Add to v's prereq count
    for (v=0; v<G->n(); v++) // Initialize queue
        if (Count[v] == 0) // Vertex has no prerequisites
            Q->enqueue(v);
    while (Q->length() != 0) { // Process the vertices
        v = Q->dequeue();
        printout(v); // PreVisit for "v"
        for (w=G->first(v); w<G->n(); w = G->next(v,w)) {
            Count[w]--; // One less prerequisite
            if (Count[w] == 0) // This vertex is now free
                Q->enqueue(w);
        }
    }
}

```

Figure 11.15 A queue-based topological sort algorithm.

11.4 Shortest-Paths Problems

On a road map, a road connecting two towns is typically labeled with its distance. We can model a road network as a directed graph whose edges are labeled with real numbers. These numbers represent the distance (or other cost metric, such as travel time) between two vertices. These labels may be called **weights**, **costs**, or **distances**, depending on the application. Given such a graph, a typical problem is to find the total length of the shortest path between two specified vertices. This is not a trivial problem, because the shortest path may not be along the edge (if any) connecting two vertices, but rather may be along a path involving one or more intermediate vertices. For example, in Figure 11.16, the cost of the path from *A* to *B* to *D* is 15. The cost of the edge directly from *A* to *D* is 20. The cost of the path from *A* to *C* to *B* to *D* is 10. Thus, the shortest path from *A* to *D* is 10 (not along the edge connecting *A* to *D*). We use the notation $d(A, D) = 10$ to indicate that the shortest distance from *A* to *D* is 10. In Figure 11.16, there is no path from *E* to *B*, so we set $d(E, B) = \infty$. We define $w(A, D) = 20$ to be the weight of edge (A, D) , that is, the weight of the direct connection from *A* to *D*. Because there is no edge from *E* to *B*, $w(E, B) = \infty$. Note that $w(D, A) = \infty$ because the graph of Figure 11.16 is directed. We assume that all weights are positive.

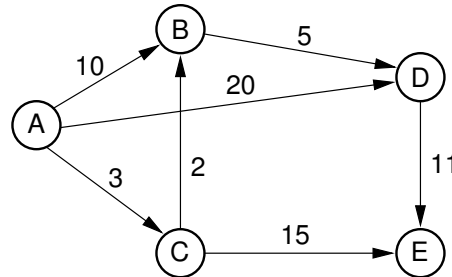


Figure 11.16 Example graph for shortest-path definitions.

11.4.1 Single-Source Shortest Paths

This section presents an algorithm to solve the **single-source shortest-paths** problem. Given Vertex S in Graph G , find a shortest path from S to every other vertex in G . We might want only the shortest path between two vertices, S and T . However in the worst case, while finding the shortest path from S to T , we might find the shortest paths from S to every other vertex as well. So there is no better algorithm (in the worst case) for finding the shortest path to a single vertex than to find shortest paths to all vertices. The algorithm described here will only compute the distance to every such vertex, rather than recording the actual path. Recording the path requires modifications to the algorithm that are left as an exercise.

Computer networks provide an application for the single-source shortest-paths problem. The goal is to find the cheapest way for one computer to broadcast a message to all other computers on the network. The network can be modeled by a graph with edge weights indicating time or cost to send a message to a neighboring computer.

For unweighted graphs (or whenever all edges have the same cost), the single-source shortest paths can be found using a simple breadth-first search. When weights are added, BFS will not give the correct answer.

One approach to solving this problem when the edges have differing weights might be to process the vertices in a fixed order. Label the vertices v_0 to v_{n-1} , with $S = v_0$. When processing Vertex v_1 , we take the edge connecting v_0 and v_1 . When processing v_2 , we consider the shortest distance from v_0 to v_2 and compare that to the shortest distance from v_0 to v_1 to v_2 . When processing Vertex v_i , we consider the shortest path for Vertices v_0 through v_{i-1} that have already been processed. Unfortunately, the true shortest path to v_i might go through Vertex v_j for $j > i$. Such a path will not be considered by this algorithm. However, the problem would not occur if we process the vertices in order of distance from S . Assume that we have processed in order of distance from S to the first $i - 1$ vertices that are closest to S ; call this set of vertices S . We are now about to process the i th closest vertex;

```

// Compute shortest path distances from "s".
// Return these distances in "D".
void Dijkstra(Graph* G, int* D, int s) {
    int i, v, w;
    for (int i=0; i<G->n(); i++)          // Initialize
        D[i] = INFINITY;
    D[0] = 0;
    for (i=0; i<G->n(); i++) {            // Process the vertices
        v = minVertex(G, D);
        if (D[v] == INFINITY) return; // Unreachable vertices
        G->setMark(v, VISITED);
        for (w=G->first(v); w<G->n(); w = G->next(v,w))
            if (D[w] > (D[v] + G->weight(v, w)))
                D[w] = D[v] + G->weight(v, w);
    }
}

```

Figure 11.17 An implementation for Dijkstra's algorithm.

call it X . A shortest path from S to X must have its next-to-last vertex in S . Thus,

$$d(S, X) = \min_{U \in S} (d(S, U) + w(U, X)).$$

In other words, the shortest path from S to X is the minimum over all paths that go from S to U , then have an edge from U to X , where U is some vertex in S .

This solution is usually referred to as Dijkstra's algorithm. It works by maintaining a distance estimate $D(X)$ for all vertices X in V . The elements of D are initialized to the value **INFINITE**. Vertices are processed in order of distance from S . Whenever a vertex V is processed, $D(X)$ is updated for every neighbor X of V . Figure 11.17 shows an implementation for Dijkstra's algorithm. At the end, array D will contain the shortest distance values.

There are two reasonable solutions to the key issue of finding the unvisited vertex with minimum distance value during each pass through the main **for** loop. The first method is simply to scan through the list of $|V|$ vertices searching for the minimum value, as follows:

```

int minVertex(Graph* G, int* D) { // Find min cost vertex
    int i, v = -1;
    // Initialize v to some unvisited vertex
    for (i=0; i<G->n(); i++)
        if (G->getMark(i) == UNVISITED) { v = i; break; }
    for (i++; i<G->n(); i++) // Now find smallest D value
        if ((G->getMark(i) == UNVISITED) && (D[i] < D[v]))
            v = i;
    return v;
}

```

Because this scan is done $|V|$ times, and because each edge requires a constant-time update to D , the total cost for this approach is $\Theta(|V|^2 + |E|) = \Theta(|V|^2)$, because $|E|$ is in $O(|V|^2)$.

The second method is to store unprocessed vertices in a min-heap ordered by distance values. The next-closest vertex can be found in the heap in $\Theta(\log |\mathbf{V}|)$ time. Every time we modify $\mathbf{D}(X)$, we could reorder X in the heap by deleting and reinserting it. This is an example of a priority queue with priority update, as described in Section 5.5. To implement true priority updating, we would need to store with each vertex its array index within the heap. A simpler approach is to add the new (smaller) distance value for a given vertex as a new record in the heap. The smallest value for a given vertex currently in the heap will be found first, and greater distance values found later will be ignored because the vertex will already be marked as **VISITED**. The only disadvantage to repeatedly inserting distance values is that it will raise the number of elements in the heap from $\Theta(|\mathbf{V}|)$ to $\Theta(|\mathbf{E}|)$ in the worst case. The time complexity is $\Theta((|\mathbf{V}| + |\mathbf{E}|) \log |\mathbf{E}|)$, because for each edge we must reorder the heap. Because the objects stored on the heap need to know both their vertex number and their distance, we create a simple class for the purpose called **DijkElem**, as follows. **DijkElem** is quite similar to the **Edge** class used by the adjacency list representation.

```
class DijkElem {
public:
    int vertex, distance;
    DijkElem() { vertex = -1; distance = -1; }
    DijkElem(int v, int d) { vertex = v; distance = d; }
};
```

Figure 11.18 shows an implementation for Dijkstra's algorithm using the priority queue.

Using **MinVertex** to scan the vertex list for the minimum value is more efficient when the graph is dense, that is, when $|\mathbf{E}|$ approaches $|\mathbf{V}|^2$. Using a priority queue is more efficient when the graph is sparse because its cost is $\Theta((|\mathbf{V}| + |\mathbf{E}|) \log |\mathbf{E}|)$. However, when the graph is dense, this cost can become as great as $\Theta(|\mathbf{V}|^2 \log |\mathbf{E}|) = \Theta(|V|^2 \log |V|)$.

Figure 11.19 illustrates Dijkstra's algorithm. The start vertex is *A*. All vertices except *A* have an initial value of ∞ . After processing Vertex *A*, its neighbors have their *D* estimates updated to be the direct distance from *A*. After processing *C* (the closest vertex to *A*), Vertices *B* and *E* are updated to reflect the shortest path through *C*. The remaining vertices are processed in order *B*, *D*, and *E*.

11.5 Minimum-Cost Spanning Trees

The **minimum-cost spanning tree** (MST) problem takes as input a connected, undirected graph **G**, where each edge has a distance or weight measure attached. The MST is the graph containing the vertices of **G** along with the subset of **G**'s edges that (1) has minimum total cost as measured by summing the values for all of

```

// Dijkstra's shortest paths algorithm with priority queue
void Dijkstra(Graph* G, int* D, int s) {
    int i, v, w;                // v is current vertex
    DijkElem temp;
    DijkElem E[G->e()];        // Heap array with lots of space
    temp.distance = 0; temp.vertex = s;
    E[0] = temp;                // Initialize heap array
    heap<DijkElem, DDComp> H(E, 1, G->e()); // Create heap
    for (int i=0; i<G->n(); i++) // Initialize
        D[i] = INFINITY;
    D[0] = 0;
    for (i=0; i<G->n(); i++) {    // Now, get distances
        do {
            if (H.size() == 0) return; // Nothing to remove
            temp = H.removefirst();
            v = temp.vertex;
        } while (G->getMark(v) == VISITED);
        G->setMark(v, VISITED);
        if (D[v] == INFINITY) return; // Unreachable vertices
        for (w=G->first(v); w<G->n(); w = G->next(v,w))
            if (D[w] > (D[v] + G->weight(v, w))) { // Update D
                D[w] = D[v] + G->weight(v, w);
                temp.distance = D[w]; temp.vertex = w;
                H.insert(temp); // Insert new distance in heap
            }
        }
    }
}

```

Figure 11.18 An implementation for Dijkstra's algorithm using a priority queue.

	A	B	C	D	E
Initial	0	∞	∞	∞	∞
Process A	0	10	3	20	∞
Process C	0	5	3	20	18
Process B	0	5	3	10	18
Process D	0	5	3	10	18
Process E	0	5	3	10	18

Figure 11.19 A listing for the progress of Dijkstra's algorithm operating on the graph of Figure 11.16. The start vertex is A.

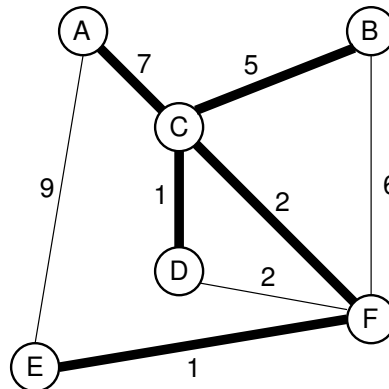


Figure 11.20 A graph and its MST. All edges appear in the original graph. Those edges drawn with heavy lines indicate the subset making up the MST. Note that edge (C, F) could be replaced with edge (D, F) to form a different MST with equal cost.

the edges in the subset, and (2) keeps the vertices connected. Applications where a solution to this problem is useful include soldering the shortest set of wires needed to connect a set of terminals on a circuit board, and connecting a set of cities by telephone lines in such a way as to require the least amount of cable.

The MST contains no cycles. If a proposed MST did have a cycle, a cheaper MST could be had by removing any one of the edges in the cycle. Thus, the MST is a free tree with $|V| - 1$ edges. The name “minimum-cost spanning tree” comes from the fact that the required set of edges forms a tree, it spans the vertices (i.e., it connects them together), and it has minimum cost. Figure 11.20 shows the MST for an example graph.

11.5.1 Prim’s Algorithm

The first of our two algorithms for finding MSTs is commonly referred to as Prim’s algorithm. Prim’s algorithm is very simple. Start with any Vertex N in the graph, setting the MST to be N initially. Pick the least-cost edge connected to N . This edge connects N to another vertex; call this M . Add Vertex M and Edge (N, M) to the MST. Next, pick the least-cost edge coming from either N or M to any other vertex in the graph. Add this edge and the new vertex it reaches to the MST. This process continues, at each step expanding the MST by selecting the least-cost edge from a vertex currently in the MST to a vertex not currently in the MST.

Prim’s algorithm is quite similar to Dijkstra’s algorithm for finding the single-source shortest paths. The primary difference is that we are seeking not the next closest vertex to the start vertex, but rather the next closest vertex to any vertex currently in the MST. Thus we replace the lines

```
if (D[w] > (D[v] + G->weight(v, w)))
    D[w] = D[v] + G->weight(v, w);
```



```

void Prim(Graph* G, int* D, int s) { // Prim's MST algorithm
    int V[G->n()]; // Store closest vertex
    int i, w;
    for (int i=0; i<G->n(); i++) // Initialize
        D[i] = INFINITY;
    D[0] = 0;
    for (i=0; i<G->n(); i++) { // Process the vertices
        int v = minVertex(G, D);
        G->setMark(v, VISITED);
        if (v != s)
            AddEdgeToMST(V[v], v); // Add edge to MST
        if (D[v] == INFINITY) return; // Unreachable vertices
        for (w=G->first(v); w<G->n(); w = G->next(v,w))
            if (D[w] > G->weight(v,w)) {
                D[w] = G->weight(v,w); // Update distance
                V[w] = v; // Where it came from
            }
    }
}

```

Figure 11.21 An implementation for Prim's algorithm.

in Dijkstra's algorithm with the lines

```

if (D[w] > G->weight(v, w))
    D[w] = G->weight(v, w);

```

in Prim's algorithm.

Figure 11.21 shows an implementation for Prim's algorithm that searches the distance matrix for the next closest vertex. For each vertex I , when I is processed by Prim's algorithm, an edge going to I is added to the MST that we are building. Array $V[I]$ stores the previously visited vertex that is closest to Vertex I . This information lets us know which edge goes into the MST when Vertex I is processed. The implementation of Figure 11.21 also contains calls to **AddEdgeToMST** to indicate which edges are actually added to the MST.

Alternatively, we can implement Prim's algorithm using a priority queue to find the next closest vertex, as shown in Figure 11.22. As with the priority queue version of Dijkstra's algorithm, the heap's **Elem** type stores a **DijkElem** object.

Prim's algorithm is an example of a greedy algorithm. At each step in the **for** loop, we select the least-cost edge that connects some marked vertex to some unmarked vertex. The algorithm does not otherwise check that the MST really should include this least-cost edge. This leads to an important question: Does Prim's algorithm work correctly? Clearly it generates a spanning tree (because each pass through the **for** loop adds one edge and one unmarked vertex to the spanning tree until all vertices have been added), but does this tree have minimum cost?

Theorem 11.1 *Prim's algorithm produces a minimum-cost spanning tree.*

```

// Prim's MST algorithm: priority queue version
void Prim(Graph* G, int* D, int s) {
    int i, v, w;           // "v" is current vertex
    int V[G->n()];         // V[I] stores I's closest neighbor
    DijkElem temp;
    DijkElem E[G->e()];    // Heap array with lots of space
    temp.distance = 0; temp.vertex = s;
    E[0] = temp;           // Initialize heap array
    heap<DijkElem, DDComp> H(E, 1, G->e()); // Create heap
    for (int i=0; i<G->n(); i++) // Initialize
        D[i] = INFINITY;
    D[0] = 0;
    for (i=0; i<G->n(); i++) { // Now build MST
        do {
            if(H.size() == 0) return; // Nothing to remove
            temp = H.removefirst();
            v = temp.vertex;
        } while (G->getMark(v) == VISITED);
        G->setMark(v, VISITED);
        if (v != s) AddEdgetoMST(V[v], v); // Add edge to MST
        if (D[v] == INFINITY) return;      // Ureachable vertex
        for (w=G->first(v); w<G->n(); w = G->next(v,w))
            if (D[w] > G->weight(v, w)) { // Update D
                D[w] = G->weight(v, w);
                V[w] = v; // Update who it came from
                temp.distance = D[w]; temp.vertex = w;
                H.insert(temp); // Insert new distance in heap
            }
        }
    }
}

```

Figure 11.22 An implementation of Prim's algorithm using a priority queue.

Proof: We will use a proof by contradiction. Let $G = (V, E)$ be a graph for which Prim's algorithm does *not* generate an MST. Define an ordering on the vertices according to the order in which they were added by Prim's algorithm to the MST: v_0, v_1, \dots, v_{n-1} . Let edge e_i connect (v_x, v_i) for some $x < i$ and $i \geq 1$. Let e_j be the lowest numbered (first) edge added by Prim's algorithm such that the set of edges selected so far *cannot* be extended to form an MST for G . In other words, e_j is the first edge where Prim's algorithm "went wrong." Let T be the "true" MST. Call v_p ($p < j$) the vertex connected by edge e_j , that is, $e_j = (v_p, v_j)$.

Because T is a tree, there exists some path in T connecting v_p and v_j . There must be some edge e' in this path connecting vertices v_u and v_w , with $u < j$ and $w \geq j$. Because e_j is not part of T , adding edge e_j to T forms a cycle. Edge e' must be of lower cost than edge e_j , because Prim's algorithm did not generate an MST. This situation is illustrated in Figure 11.23. However, Prim's algorithm would have selected the least-cost edge available. It would have selected e' , not e_j . Thus, it is a contradiction that Prim's algorithm would have selected the wrong edge, and thus, Prim's algorithm must be correct. \square

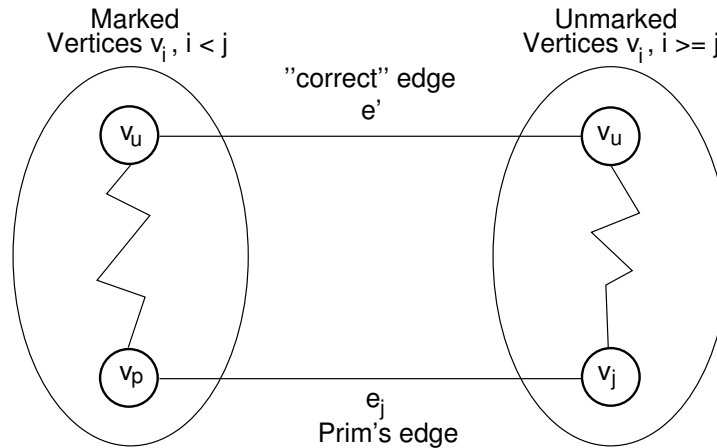


Figure 11.23 Prim's MST algorithm proof. The left oval contains that portion of the graph where Prim's MST and the "true" MST T agree. The right oval contains the rest of the graph. The two portions of the graph are connected by (at least) edges e_j (selected by Prim's algorithm to be in the MST) and e' (the "correct" edge to be placed in the MST). Note that the path from v_u to v_j cannot include any marked vertex v_i , $i \leq j$, because to do so would form a cycle.

Example 11.3 For the graph of Figure 11.20, assume that we begin by marking Vertex A. From A, the least-cost edge leads to Vertex C. Vertex C and edge (A, C) are added to the MST. At this point, our candidate edges connecting the MST (Vertices A and C) with the rest of the graph are (A, E), (C, B), (C, D), and (C, F). From these choices, the least-cost edge from the MST is (C, D). So we add Vertex D to the MST. For the next iteration, our edge choices are (A, E), (C, B), (C, F), and (D, F). Because edges (C, F) and (D, F) happen to have equal cost, it is an arbitrary decision as to which gets selected. Say we pick (C, F). The next step marks Vertex E and adds edge (F, E) to the MST. Following in this manner, Vertex B (through edge (C, B)) is marked. At this point, the algorithm terminates.

11.5.2 Kruskal's Algorithm

Our next MST algorithm is commonly referred to as Kruskal's algorithm. Kruskal's algorithm is also a simple, greedy algorithm. First partition the set of vertices into $|V|$ equivalence classes (see Section 6.2), each consisting of one vertex. Then process the edges in order of weight. An edge is added to the MST, and two equivalence classes combined, if the edge connects two vertices in different equivalence classes. This process is repeated until only one equivalence class remains.

Example 11.4 Figure 11.24 shows the first three steps of Kruskal’s Algorithm for the graph of Figure 11.20. Edge (C, D) has the least cost, and because C and D are currently in separate MSTs, they are combined. We next select edge (E, F) to process, and combine these vertices into a single MST. The third edge we process is (C, F) , which causes the MST containing Vertices C and D to merge with the MST containing Vertices E and F . The next edge to process is (D, F) . But because Vertices D and F are currently in the same MST, this edge is rejected. The algorithm will continue on to accept edges (B, C) and (A, C) into the MST.

The edges can be processed in order of weight by using a min-heap. This is generally faster than sorting the edges first, because in practice we need only visit a small fraction of the edges before completing the MST. This is an example of finding only a few smallest elements in a list, as discussed in Section 7.6.

The only tricky part to this algorithm is determining if two vertices belong to the same equivalence class. Fortunately, the ideal algorithm is available for the purpose — the UNION/FIND algorithm based on the parent pointer representation for trees described in Section 6.2. Figure 11.25 shows an implementation for the algorithm. Class **KruskalElem** is used to store the edges on the min-heap.

Kruskal’s algorithm is dominated by the time required to process the edges. The **differ** and **UNION** functions are nearly constant in time if path compression and weighted union is used. Thus, the total cost of the algorithm is $\Theta(|\mathbf{E}| \log |\mathbf{E}|)$ in the worst case, when nearly all edges must be processed before all the edges of the spanning tree are found and the algorithm can stop. More often the edges of the spanning tree are the shorter ones, and only about $|\mathbf{V}|$ edges must be processed. If so, the cost is often close to $\Theta(|\mathbf{V}| \log |\mathbf{E}|)$ in the average case.

11.6 Further Reading

Many interesting properties of graphs can be investigated by playing with the programs in the Stanford Graphbase. This is a collection of benchmark databases and graph processing programs. The Stanford Graphbase is documented in [Knu94].

11.7 Exercises

11.1 Prove by induction that a graph with n vertices has at most $n(n-1)/2$ edges.

11.2 Prove the following implications regarding free trees.

- (a) IF an undirected graph is connected and has no simple cycles, THEN the graph has $|\mathbf{V}| - 1$ edges.

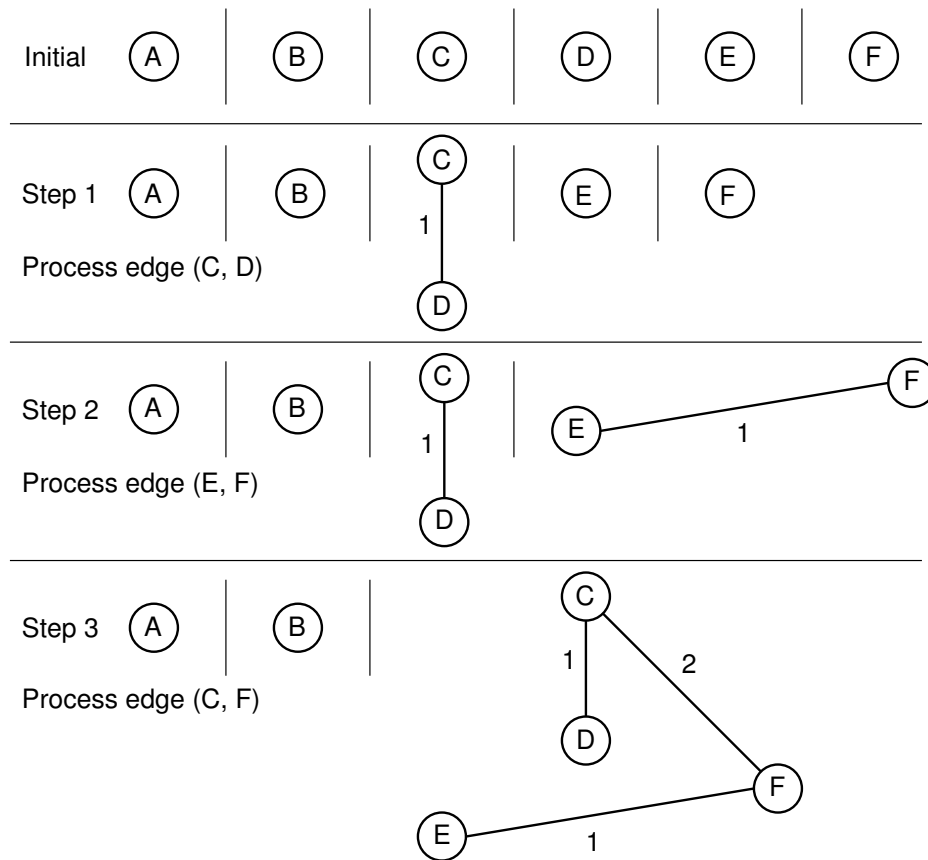


Figure 11.24 Illustration of the first three steps of Kruskal's MST algorithm as applied to the graph of Figure 11.20.

- (b) IF an undirected graph has $|V| - 1$ edges and no cycles, THEN the graph is connected.
- 11.3** (a) Draw the adjacency matrix representation for the graph of Figure 11.26.
 (b) Draw the adjacency list representation for the same graph.
 (c) If a pointer requires four bytes, a vertex label requires two bytes, and an edge weight requires two bytes, which representation requires more space for this graph?
 (d) If a pointer requires four bytes, a vertex label requires one byte, and an edge weight requires two bytes, which representation requires more space for this graph?
- 11.4** Show the DFS tree for the graph of Figure 11.26, starting at Vertex 1.
- 11.5** Write a pseudocode algorithm to create a DFS tree for an undirected, connected graph starting at a specified vertex V .

```

class KruskElem {          // An element for the heap
public:
    int from, to, distance; // The edge being stored
    KruskElem() { from = -1; to = -1; distance = -1; }
    KruskElem(int f, int t, int d)
        { from = f; to = t; distance = d; }
};

void Kruskal(Graph* G) {    // Kruskal's MST algorithm
    ParPtrTree A(G->n());   // Equivalence class array
    KruskElem E[G->e()];    // Array of edges for min-heap
    int i;
    int edgecnt = 0;
    for (i=0; i<G->n(); i++) // Put the edges on the array
        for (int w=G->first(i); w<G->n(); w = G->next(i,w)) {
            E[edgecnt].distance = G->weight(i, w);
            E[edgecnt].from = i;
            E[edgecnt++].to = w;
        }
    // Heapify the edges
    heap<KruskElem, Comp> H(E, edgecnt, edgecnt);
    int numMST = G->n();     // Initially n equiv classes
    for (i=0; numMST>1; i++) { // Combine equiv classes
        KruskElem temp;
        temp = H.removefirst(); // Get next cheapest edge
        int v = temp.from; int u = temp.to;
        if (A.differ(v, u)) { // If in different equiv classes
            A.UNION(v, u);    // Combine equiv classes
            AddEdgetoMST(temp.from, temp.to); // Add edge to MST
            numMST--;         // One less MST
        }
    }
}

```

Figure 11.25 An implementation for Kruskal's algorithm.

- 11.6** Show the BFS tree for the graph of Figure 11.26, starting at Vertex 1.
- 11.7** Write a pseudocode algorithm to create a BFS tree for an undirected, connected graph starting at a specified vertex V .
- 11.8** The BFS topological sort algorithm can report the existence of a cycle if one is encountered. Modify this algorithm to print the vertices possibly appearing in cycles (if there are any cycles).
- 11.9** Explain why, in the worst case, Dijkstra's algorithm is (asymptotically) as efficient as any algorithm for finding the shortest path from some vertex I to another vertex J .
- 11.10** Show the shortest paths generated by running Dijkstra's shortest-paths algorithm on the graph of Figure 11.26, beginning at Vertex 4. Show the D values as each vertex is processed, as in Figure 11.19.
- 11.11** Modify the algorithm for single-source shortest paths to actually store and return the shortest paths rather than just compute the distances.

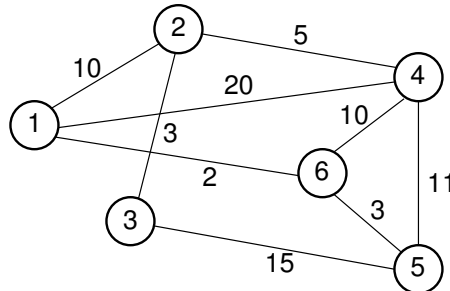


Figure 11.26 Example graph for Chapter 11 exercises.

- 11.12** The root of a DAG is a vertex R such that every vertex of the DAG can be reached by a directed path from R . Write an algorithm that takes a directed graph as input and determines the root (if there is one) for the graph. The running time of your algorithm should be $\Theta(|V| + |E|)$.
- 11.13** Write an algorithm to find the longest path in a DAG, where the length of the path is measured by the number of edges that it contains. What is the asymptotic complexity of your algorithm?
- 11.14** Write an algorithm to determine whether a directed graph of $|V|$ vertices contains a cycle. Your algorithm should run in $\Theta(|V| + |E|)$ time.
- 11.15** Write an algorithm to determine whether an undirected graph of $|V|$ vertices contains a cycle. Your algorithm should run in $\Theta(|V|)$ time.
- 11.16** The **single-destination shortest-paths** problem for a directed graph is to find the shortest path *from* every vertex to a specified vertex V . Write an algorithm to solve the single-destination shortest-paths problem.
- 11.17** List the order in which the edges of the graph in Figure 11.26 are visited when running Prim's MST algorithm starting at Vertex 3. Show the final MST.
- 11.18** List the order in which the edges of the graph in Figure 11.26 are visited when running Kruskal's MST algorithm. Each time an edge is added to the MST, show the result on the equivalence array, (e.g., show the array as in Figure 6.7).
- 11.19** Write an algorithm to find a **maximum** cost spanning tree, that is, the spanning tree with highest possible cost.
- 11.20** When can Prim's and Kruskal's algorithms yield different MSTs?
- 11.21** Prove that, if the costs for the edges of Graph G are distinct, then only one MST exists for G .
- 11.22** Does either Prim's or Kruskal's algorithm work if there are negative edge weights?
- 11.23** Consider the collection of edges selected by Dijkstra's algorithm as the shortest paths to the graph's vertices from the start vertex. Do these edges form

a spanning tree (not necessarily of minimum cost)? Do these edges form an MST? Explain why or why not.

- 11.24** Prove that a tree is a bipartite graph.
- 11.25** Prove that any tree (i.e., a connected, undirected graph with no cycles) can be two-colored. (A graph can be two colored if every vertex can be assigned one of two colors such that no adjacent vertices have the same color.)
- 11.26** Write an algorithm that determines if an arbitrary undirected graph is a bipartite graph. If the graph is bipartite, then your algorithm should also identify the vertices as to which of the two partitions each belongs to.

11.8 Projects

- 11.1** Design a format for storing graphs in files. Then implement two functions: one to read a graph from a file and the other to write a graph to a file. Test your functions by implementing a complete MST program that reads an undirected graph in from a file, constructs the MST, and then writes to a second file the graph representing the MST.
- 11.2** An undirected graph need not explicitly store two separate directed edges to represent a single undirected edge. An alternative would be to store only a single undirected edge (I, J) to connect Vertices I and J . However, what if the user asks for edge (J, I) ? We can solve this problem by consistently storing the edge such that the lesser of I and J always comes first. Thus, if we have an edge connecting Vertices 5 and 3, requests for edge $(5, 3)$ and $(3, 5)$ both map to $(3, 5)$ because $3 < 5$.
Looking at the adjacency matrix, we notice that only the lower triangle of the array is used. Thus we could cut the space required by the adjacency matrix from $|V|^2$ positions to $|V|(|V| - 1)/2$ positions. Read Section 12.2 on triangular matrices. The re-implement the adjacency matrix representation of Figure 11.6 to implement undirected graphs using a triangular array.
- 11.3** While the underlying implementation (whether adjacency matrix or adjacency list) is hidden behind the graph ADT, these two implementations can have an impact on the efficiency of the resulting program. For Dijkstra's shortest paths algorithm, two different implementations were given in Section 11.4.1 that provide different ways for determining the next closest vertex at each iteration of the algorithm. The relative costs of these two variants depend on who sparse or dense the graph is. They might also depend on whether the graph is implemented using an adjacency list or adjacency matrix.

Design and implement a study to compare the effects on performance for three variables: (i) the two graph representations (adjacency list and adjacency matrix); (ii) the two implementations for Dijkstra's shortest paths algorithm (searching the table of vertex distances or using a priority queue to

track the distances), and (iii) sparse versus dense graphs. Be sure to test your implementations on a variety of graphs that are sufficiently large to generate meaningful times.

- 11.4** The example implementations for DFS and BFS show calls to functions **PreVisit** and **PostVisit**. Re-implement the BFS and DFS functions to make use of the visitor design pattern to handle the pre/post visit functionality.
- 11.5** Write a program to label the connected components for an undirected graph. In other words, all vertices of the first component are given the first component's label, all vertices of the second component are given the second component's label, and so on. Your algorithm should work by defining any two vertices connected by an edge to be members of the same equivalence class. Once all of the edges have been processed, all vertices in a given equivalence class will be connected. Use the UNION/FIND implementation from Section 6.2 to implement equivalence classes.

Lists and Arrays Revisited

Simple lists and arrays are the right tools for the many applications. Other situations require support for operations that cannot be implemented efficiently by the standard list representations of Chapter 4. This chapter presents a range of topics, whose unifying thread is that the data structures included are all list- or array-like. These structures overcome some of the problems of simple linked list and contiguous array representations. This chapter also seeks to reinforce the concept of logical representation versus physical implementation, as some of the “list” implementations have quite different organizations internally.

Section 12.1 describes a series of representations for multilists, which are lists that may contain sublists. Section 12.2 discusses representations for implementing sparse matrices, large matrices where most of the elements have zero values. Section 12.3 discusses memory management techniques, which are essentially a way of allocating variable-length sections from a large array.

12.1 Multilists

Recall from Chapter 4 that a list is a finite, ordered sequence of items of the form $\langle x_0, x_1, \dots, x_{n-1} \rangle$ where $n \geq 0$. We can represent the empty list by **NULL** or $\langle \rangle$. In Chapter 4 we assumed that all list elements had the same data type. In this section, we extend the definition of lists to allow elements to be arbitrary in nature. In general, list elements are one of two types.

1. An **atom**, which is a data record of some type such as a number, symbol, or string.
2. Another list, which is called a **sublist**.

A list containing sublists will be written as

$$\langle x1, \langle y1, \langle a1, a2 \rangle, y3 \rangle, \langle z1, z2 \rangle, x4 \rangle.$$

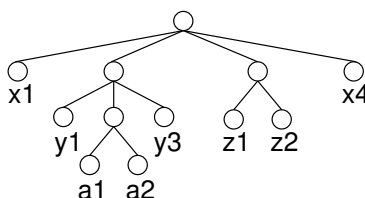


Figure 12.1 Example of a multilist represented by a tree.

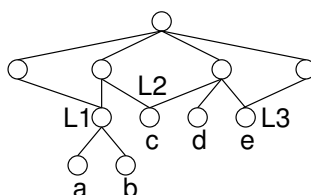


Figure 12.2 Example of a reentrant multilist. The shape of the structure is a DAG (all edges point downward).

In this example, the list has four elements. The second element is the sublist $\langle y1, \langle a1, a2 \rangle, y3 \rangle$ and the third is the sublist $\langle z1, z2 \rangle$. The sublist $\langle y1, \langle a1, a2 \rangle, y3 \rangle$ itself contains a sublist. If a list **L** has one or more sublists, we call **L** a **multilist**. Lists with no sublists are often referred to as **linear lists** or **chains**. Note that this definition for multilist fits well with our definition of sets from Definition 2.1, where a set's members can be either primitive elements or sets.

We can restrict the sublists of a multilist in various ways, depending on whether the multilist should have the form of a tree, a DAG, or a generic graph. A **pure list** is a list structure whose graph corresponds to a tree, such as in Figure 12.1. In other words, there is exactly one path from the root to any node, which is equivalent to saying that no object may appear more than once in the list. In the pure list, each pair of angle brackets corresponds to an internal node of the tree. The members of the list correspond to the children for the node. Atoms on the list correspond to leaf nodes.

A **reentrant list** is a list structure whose graph corresponds to a DAG. Nodes might be accessible from the root by more than one path, which is equivalent to saying that objects (including sublists) may appear multiple times in the list as long as no cycles are formed. All edges point downward, from the node representing a list or sublist to its elements. Figure 12.2 illustrates a reentrant list. To write out this list in bracket notation, we can duplicate nodes as necessary. Thus, the bracket notation for the list of Figure 12.2 could be written

$$\langle \langle \langle a, b \rangle \rangle, \langle \langle a, b \rangle, c \rangle, \langle c, d, e \rangle, \langle e \rangle \rangle.$$

For convenience, we will adopt a convention of allowing sublists and atoms to be labeled, such as “L1:”. Whenever a label is repeated, the element corresponding to

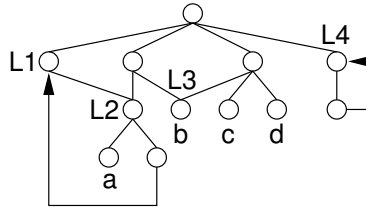


Figure 12.3 Example of a cyclic list. The shape of the structure is a directed graph.

that label will be substituted when we write out the list. Thus, the bracket notation for the list of Figure 12.2 could be written

$$\langle \langle L1 : \langle a, b \rangle \rangle, \langle L1, L2 : c \rangle, \langle L2, d, L3 : e \rangle, \langle L3 \rangle \rangle.$$

A **cyclic list** is a list structure whose graph corresponds to any directed graph, possibly containing cycles. Figure 12.3 illustrates such a list. Labels are required to write this in bracket notation. Here is the bracket notation for the list of Figure 12.3.

$$\langle L1 : \langle L2 : \langle a, L1 \rangle \rangle, \langle L2, L3 : b \rangle, \langle L3, c, d \rangle, L4 : \langle L4 \rangle \rangle.$$

Multilists can be implemented in a number of ways. Most of these should be familiar from implementations suggested earlier in the book for list, tree, and graph data structures.

One simple approach is to use a simple array to represent the list. This works well for chains with fixed-length elements, equivalent to the simple array-based list of Chapter 4. We can view nested sublists as variable-length elements. To use this approach, we require some indication of the beginning and end of each sublist. In essence, we are using a sequential tree implementation as discussed in Section 6.5. This should be no surprise, because the pure list is equivalent to a general tree structure. Unfortunately, as with any sequential representation, access to the n th sublist must be done sequentially from the beginning of the list.

Because pure lists are equivalent to trees, we can also use linked allocation methods to support direct access to the list of children. Simple linear lists are represented by linked lists. Pure lists can be represented as linked lists with an additional tag field to indicate whether the node is an atom or a sublist. If it is a sublist, the data field points to the first element on the sublist. This is illustrated by Figure 12.4.

Another approach is to represent all list elements with link nodes storing two pointer fields, except for atoms. Atoms just contain data. This is the system used by the programming language LISP. Figure 12.5 illustrates this representation. Either the pointer contains a tag bit to identify what it points to, or the object being pointed to stores a tag bit to identify itself. Tags distinguish atoms from list nodes. This

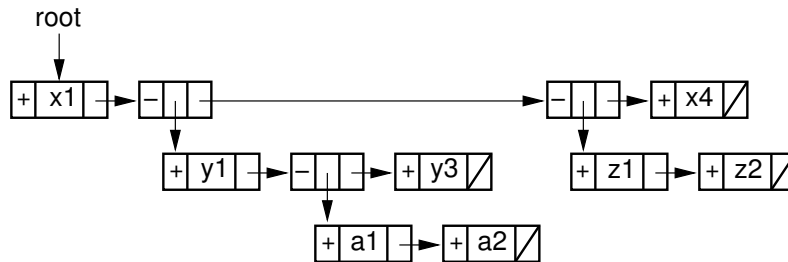


Figure 12.4 Linked representation for the pure list of Figure 12.1. The first field in each link node stores a tag bit. If the tag bit stores “+,” then the data field stores an atom. If the tag bit stores “-,” then the data field stores a pointer to a sublist.

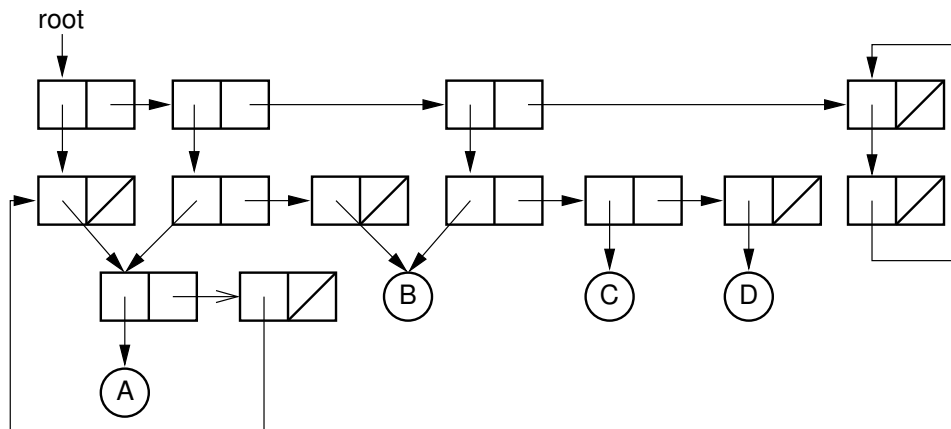


Figure 12.5 LISP-like linked representation for the cyclic multilist of Figure 12.3. Each link node stores two pointers. A pointer either points to an atom, or to another link node. Link nodes are represented by two boxes, and atoms by circles.

implementation can easily support reentrant and cyclic lists, because non-atoms can point to any other node.

12.2 Matrix Representations

Sometimes we need to represent a large, two-dimensional matrix where many of the elements have a value of zero. One example is the lower triangular matrix that results from solving systems of simultaneous equations. A lower triangular matrix stores zero values at all positions $[r, c]$ such that $r < c$, as shown in Figure 12.6(a). Thus, the upper-right triangle of the matrix is always zero. Another example is representing undirected graphs in an adjacency matrix (see Project 11.2). Because all edges between Vertices i and j go in both directions, there is no need to store both. Instead we can just store one edge going from the higher-indexed vertex to

a ₀₀	0	0	0
a ₁₀	a ₁₁	0	0
a ₂₀	a ₂₁	a ₂₂	0
a ₃₀	a ₃₁	a ₃₂	a ₃₃

(a)

a ₀₀	a ₀₁	a ₀₂	a ₀₃
0	a ₁₁	a ₁₂	a ₁₃
0	0	a ₂₂	a ₂₃
0	0	0	a ₃₃

(b)

Figure 12.6 Triangular matrices. (a) A lower triangular matrix. (b) An upper triangular matrix.

the lower-indexed vertex. In this case, only the lower triangle of the matrix can have non-zero values.

We can take advantage of this fact to save space. Instead of storing $n(n+1)/2$ pieces of information in an $n \times n$ array, it would save space to use a list of length $n(n+1)/2$. This is only practical if some means can be found to locate within the list the element that would correspond to position $[r, c]$ in the original matrix.

We will derive an equation to convert position $[r, c]$ to a position in a one-dimensional list to store the lower triangular matrix. Note that row 0 of the matrix has one non-zero value, row 1 has two non-zero values, and so on. Thus, row r is preceded by r rows with a total of $\sum_{k=1}^r k = (r^2 + r)/2$ non-zero elements. Adding c to reach the c th position in the r th row yields the following equation to convert position $[r, c]$ in the original matrix to the correct position in the list.

$$\text{matrix}[r, c] = \text{list}[(r^2 + r)/2 + c].$$

A similar equation can be used to convert coordinates in an upper triangular matrix, that is, a matrix with zero values at positions $[r, c]$ such that $r > c$, as shown in Figure 12.6(b). For an $n \times n$ upper triangular matrix, the equation to convert from matrix coordinates to list positions would be

$$\text{matrix}[r, c] = \text{list}[rn - (r^2 + r)/2 + c].$$

A more difficult situation arises when the vast majority of values stored in an $n \times m$ matrix are zero, but there is no restriction on which positions are zero and which are non-zero. This is known as a **sparse matrix**.

One approach to representing a sparse matrix is to concatenate (or otherwise combine) the row and column coordinates into a single value and use this as a key in a hash table. Thus, if we want to know the value of a particular position in the matrix, we search the hash table for the appropriate key. If a value for this position is not found, it is assumed to be zero. This is an ideal approach when all queries to the matrix are in terms of access by specified position. However, if we wish to find the first non-zero element in a given row, or the next non-zero element below the current one in a given column, then the hash table requires us to check sequentially through all possible positions in some row or column.

Another approach is to implement the matrix as an **orthogonal list**. Consider the following sparse matrix:

10	23	0	0	0	0	19
45	5	0	93	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
40	0	0	0	0	0	0
0	0	0	0	0	0	0
0	0	0	0	0	0	0
0	32	0	12	0	0	7

The corresponding orthogonal array is shown in Figure 12.7. Here we have a list of row headers, each of which contains a pointer to a list of matrix records. A second list of column headers also contains pointers to matrix records. Each non-zero matrix element stores pointers to its non-zero neighbors in the row, both following and preceding it. Each non-zero element also stores pointers to its non-zero neighbors following and preceding it in the column. Thus, each non-zero element stores its own value, its position within the matrix, and four pointers. Non-zero elements are found by traversing a row or column list. Note that the first non-zero element in a given row could be in any column; likewise, the neighboring non-zero element in any row or column list could be at any (higher) row or column in the array. Thus, each non-zero element must also store its row and column position explicitly.

To find if a particular position in the matrix contains a non-zero element, we traverse the appropriate row or column list. For example, when looking for the element at Row 7 and Column 1, we can traverse the list either for Row 7 or for Column 1. When traversing a row or column list, if we come to an element with the correct position, then its value is non-zero. If we encounter an element with a higher position, then the element we are looking for is not in the sparse matrix. In this case, the element's value is zero. For example, when traversing the list for Row 7 in the matrix of Figure 12.7, we first reach the element at Row 7 and Column 1. If this is what we are looking for, then the search can stop. If we are looking for the element at Row 7 and Column 2, then the search proceeds along the Row 7 list to next reach the element at Column 3. At this point we know that no element at Row 7 and Column 2 is stored in the sparse matrix.

Insertion and deletion can be performed by working in a similar way to insert or delete elements within the appropriate row and column lists.

Each non-zero element stored in the sparse matrix representation takes much more space than an element stored in a simple $n \times n$ matrix. When is the sparse matrix more space efficient than the standard representation? To calculate this, we need to determine how much space the standard matrix requires, and how much

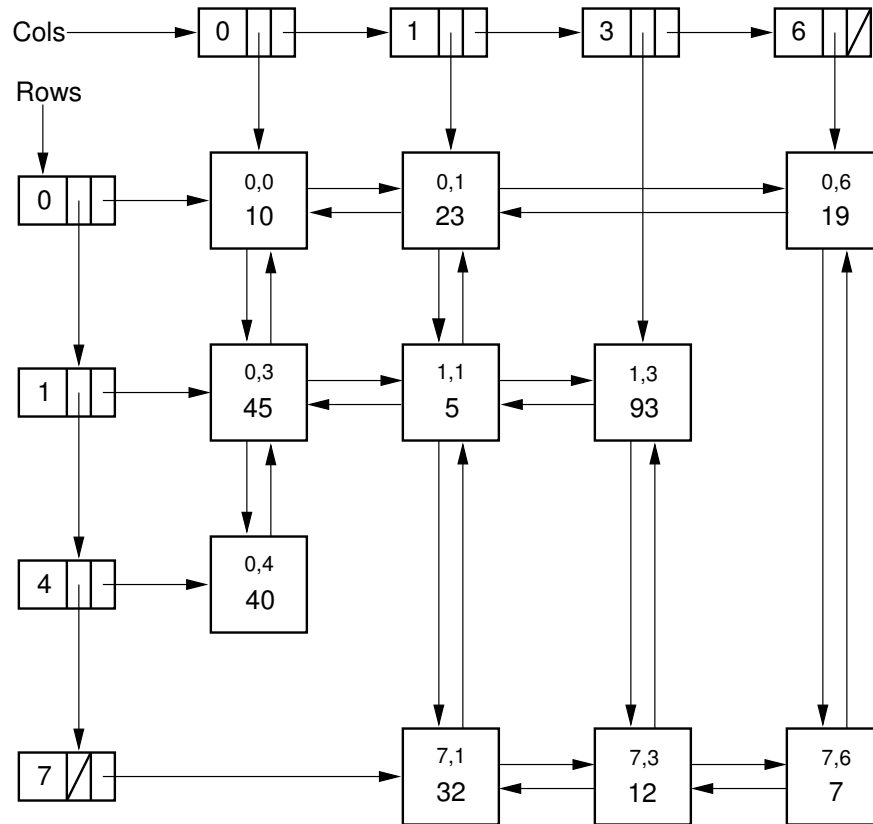


Figure 12.7 The orthogonal list sparse matrix representation.

the sparse matrix requires. The size of the sparse matrix depends on the number of non-zero elements (we will refer to this value as NNZ), while the size of the standard matrix representation does not vary. We need to know the (relative) sizes of a pointer and a data value. For simplicity, our calculation will ignore the space taken up by the row and column header (which is not much affected by the number of elements in the sparse array).

As an example, assume that a data value, a row or column index, and a pointer each require four bytes. An $n \times m$ matrix requires $4nm$ bytes. The sparse matrix requires 28 bytes per non-zero element (four pointers, two array indices, and one data value). If we set X to be the percentage of non-zero elements, we can solve for the value of X below which the sparse matrix representation is more space efficient. Using the equation

$$28X = 4mn$$

and solving for X , we find that the sparse matrix using this implementation is more space efficient when $X < 1/7$, that is, when less than about 14% of the elements

are non-zero. Different values for the relative sizes of data values, pointers, or matrix indices can lead to a different break-even point for the two implementations.

The time required to process a sparse matrix should ideally depend on NNZ . When searching for an element, the cost is the number of elements preceding the desired element on its row or column list. The cost for operations such as adding two matrices should be $\Theta(n + m)$ in the worst case when the one matrix stores n non-zero elements and the other stores m non-zero elements.

Another representation for sparse matrices is sometimes called the Yale representation. Matlab uses a similar representation, with a primary difference being that the Matlab representation uses column-major order.¹ The Matlab representation stores the sparse matrix using three lists. The first is simply all of the non-zero element values, in column-major order. The second list stores the start position within the first list for each column. The third list stores the row positions for each of the corresponding non-zero values. In the Yale representation, the matrix of Figure 12.7 would appear as:

```

Values:      10 45 40 23 5 32 93 12 19 7
Column starts: 0 3 5 5 7 7 7 7
Row positions: 0 1 4 0 1 7 1 7 0 7

```

If the matrix has c columns, then the total space required will be proportional to $c + 2NNZ$. This is good in terms of space. It allows fairly quick access to any column, and allows for easy processing of the non-zero values along a column. However, it does not do a good job of providing access to the values along a row, and is terrible when values need to be added or removed from the representation. Fortunately, when doing computations such as adding or multiplying two sparse matrices, the processing of the input matrices and construction of the output matrix can be done reasonably efficiently.

12.3 Memory Management

Most data structures are designed to store and access objects of uniform size. A typical example would be an integer stored in a list or a queue. Some applications require the ability to store variable-length records, such as a string of arbitrary length. One solution is to store in the list or queue fixed-length pointers to the variable-length strings. This is fine for data structures stored in main memory. But if the collection of strings is meant to be stored on disk, then we might need to worry about where exactly these strings are stored. And even when stored in main memory, something has to figure out where there are available bytes to hold the string. We could easily store variable-size records in a queue or stack, where

¹Scientific packages tend to prefer column-oriented representations for matrices since this the dominant access need for the operations to be performed.

```
// Memory Manager abstract class
class MemManager {
public:
    virtual ~MemManager() {} // Base destructor

    // Store a record and return a handle to it
    virtual MemHandle insert(void* info, int length) =0;

    // Get back a copy of a stored record
    virtual int get(void* info, MemHandle h) =0;

    // Release the space associated with a record
    virtual void release(MemHandle h) =0;
};
```

Figure 12.8 A simple ADT for a memory manager.

the restricted order of insertions and deletions makes this easy to deal with. But in a language like C++ or Java, programmers can allocate and deallocate space in complex ways through use of **new**. Where does this space come from? This section discusses memory management techniques for the general problem of handling space requests of variable size.

The basic model for memory management is that we have a (large) block of contiguous memory locations, which we will call the **memory pool**. Periodically, memory requests are issued for some amount of space in the pool. The **memory manager** has the job of finding a contiguous block of locations of at least the requested size from somewhere within the memory pool. Honoring such a request is called a **memory allocation**. The memory manager will typically return some piece of information that the requester can hold on to so that later it can recover the record that was just stored by the memory manager. This piece of information is called a **handle**. At some point, space that has been requested might no longer be needed, and this space can be returned to the memory manager so that it can be reused. This is called a **memory deallocation**. The memory manager should then be able to reuse this space to satisfy later memory requests. We can define an ADT for the memory manager as shown in Figure 12.8.

The user of the **MemManager** ADT provides a pointer (in parameter **info**) to space that holds some record or message to be stored or retrieved. This is similar to the C++ basic file read/write methods presented in Section 8.4. The fundamental idea is that the client gives messages to the memory manager for safe keeping. The memory manager returns a “receipt” for the message in the form of a **MemHandle** object. Of course to be practical, a **MemHandle** must be much smaller than the typical message to be stored. The client holds the **MemHandle** object until it wishes to get the message back.

Method **insert** lets the client tell the memory manager the length and contents of the message to be stored. This ADT assumes that the memory manager will



Figure 12.9 Dynamic storage allocation model. Memory is made up of a series of variable-size blocks, some allocated and some free. In this example, shaded areas represent memory currently allocated and unshaded areas represent unused memory available for future allocation.

remember the length of the message associated with a given handle (perhaps in the handle itself), thus method **get** does not include a length parameter but instead returns the length of the message actually stored. Method **release** allows the client to tell the memory manager to release the space that stores a given message.

When all inserts and releases follow a simple pattern, such as last requested, first released (stack order), or first requested, first released (queue order), memory management is fairly easy. We are concerned here with the general case where blocks of any size might be requested and released in any order. This is known as **dynamic storage allocation**. One example of dynamic storage allocation is managing free store for a compiler's runtime environment, such as the system-level **new** and **delete** operations in **C++**. Another example is managing main memory in a multitasking operating system. Here, a program might require a certain amount of space, and the memory manager must keep track of which programs are using which parts of the main memory. Yet another example is the file manager for a disk drive. When a disk file is created, expanded, or deleted, the file manager must allocate or deallocate disk space.

A block of memory or disk space managed in this way is sometimes referred to as a **heap**. The term "heap" is being used here in a different way than the heap data structure discussed in Section 5.5. Here "heap" refers to the memory controlled by a dynamic memory management scheme.

In the rest of this section, we first study techniques for dynamic memory management. We then tackle the issue of what to do when no single block of memory in the memory pool is large enough to honor a given request.

12.3.1 Dynamic Storage Allocation

For the purpose of dynamic storage allocation, we view memory as a single array which, after a series of memory requests and releases tends to become broken into a series of variable-size blocks, where some of the blocks are **free** and some are **reserved** or already allocated to store messages. The memory manager typically uses a linked list to keep track of the free blocks, called the **freelist**, which is used for servicing future memory requests. Figure 12.9 illustrates the situation that can arise after a series of memory allocations and deallocations.

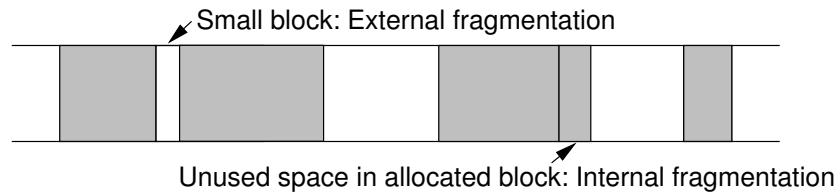


Figure 12.10 An illustration of internal and external fragmentation. The small white block labeled "External fragmentation" is too small to satisfy typical memory requests. The small grey block labeled "Internal fragmentation" was allocated as part of the grey block to its left, but it does not actually store information.

When a memory request is received by the memory manager, some block on the freelist must be found that is large enough to service the request. If no such block is found, then the memory manager must resort to a **failure policy** such as discussed in Section 12.3.2.

If there is a request for m words, and no block exists of exactly size m , then a larger block must be used instead. One possibility in this case is that the entire block is given away to the memory allocation request. This might be desirable when the size of the block is only slightly larger than the request. This is because saving a tiny block that is too small to be useful for a future memory request might not be worthwhile. Alternatively, for a free block of size k , with $k > m$, up to $k - m$ space may be retained by the memory manager to form a new free block, while the rest is used to service the request.

Memory managers can suffer from two types of **fragmentation**, which refers to unused space that is too small to be useful. **External fragmentation** occurs when a series of memory requests and releases results in small free blocks. **Internal fragmentation** occurs when more than m words are allocated to a request for m words, wasting free storage. This is equivalent to the internal fragmentation that occurs when files are allocated in multiples of the cluster size. The difference between internal and external fragmentation is illustrated by Figure 12.10.

Some memory management schemes sacrifice space to internal fragmentation to make memory management easier (and perhaps reduce external fragmentation). For example, external fragmentation does not happen in file management systems that allocate file space in clusters. Another example of sacrificing space to internal fragmentation so as to simplify memory management is the **buddy method** described later in this section.

The process of searching the memory pool for a block large enough to service the request, possibly reserving the remaining space as a free block, is referred to as a **sequential fit** method.

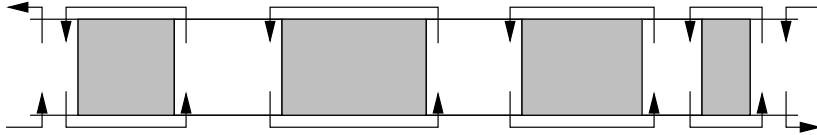


Figure 12.11 A doubly linked list of free blocks as seen by the memory manager. Shaded areas represent allocated memory. Unshaded areas are part of the freelist.

Sequential Fit Methods

Sequential-fit methods attempt to find a “good” block to service a storage request. The three sequential-fit methods described here assume that the free blocks are organized into a doubly linked list, as illustrated by Figure 12.11.

There are two basic approaches to implementing the freelist. The simpler approach is to store the freelist separately from the memory pool. In other words, a simple linked-list implementation such as described in Chapter 4 can be used, where each node of the linked list contains a pointer to a single free block in the memory pool. This is fine if there is space available for the linked list itself, separate from the memory pool.

The second approach to storing the freelist is more complicated but saves space. Because the free space is free, it can be used by the memory manager to help it do its job. That is, the memory manager can temporarily “borrow” space within the free blocks to maintain its doubly linked list. To do so, each unallocated block must be large enough to hold these pointers. In addition, it is usually worthwhile to let the memory manager add a few bytes of space to each reserved block for its own purposes. In other words, a request for m bytes of space might result in slightly more than m bytes being allocated by the memory manager, with the extra bytes used by the memory manager itself rather than the requester. We will assume that all memory blocks are organized as shown in Figure 12.12, with space for tags and linked list pointers. Here, free and reserved blocks are distinguished by a tag bit at both the beginning and the end of the block, for reasons that will be explained. In addition, both free and reserved blocks have a size indicator immediately after the tag bit at the beginning of the block to indicate how large the block is. Free blocks have a second size indicator immediately preceding the tag bit at the end of the block. Finally, free blocks have left and right pointers to their neighbors in the free block list.

The information fields associated with each block permit the memory manager to allocate and deallocate blocks as needed. When a request comes in for m words of storage, the memory manager searches the linked list of free blocks until it finds a “suitable” block for allocation. How it determines which block is suitable will be discussed below. If the block contains exactly m words (plus space for the tag and size fields), then it is removed from the freelist. If the block (of size k) is large

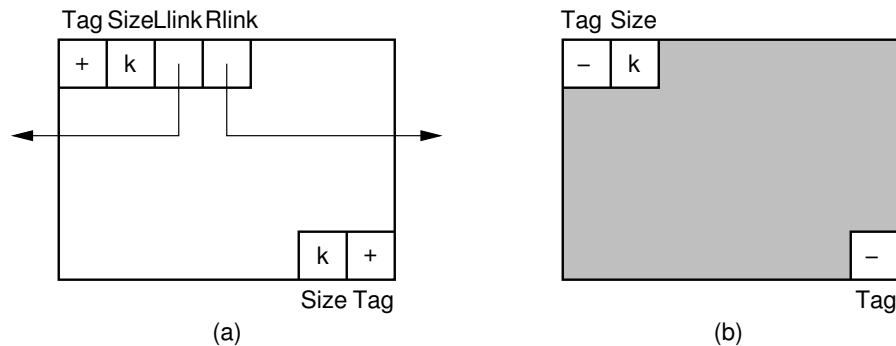


Figure 12.12 Blocks as seen by the memory manager. Each block includes additional information such as freelist link pointers, start and end tags, and a size field. (a) The layout for a free block. The beginning of the block contains the tag bit field, the block size field, and two pointers for the freelist. The end of the block contains a second tag field and a second block size field. (b) A reserved block of k bytes. The memory manager adds to these k bytes an additional tag bit field and block size field at the beginning of the block, and a second tag field at the end of the block.

enough, then the remaining $k - m$ words are reserved as a block on the freelist, in the current location.

When a block F is freed, it must be merged into the freelist. If we do not care about merging adjacent free blocks, then this is a simple insertion into the doubly linked list of free blocks. However, we would like to merge adjacent blocks, because this allows the memory manager to serve requests of the largest possible size. Merging is easily done due to the tag and size fields stored at the ends of each block, as illustrated by Figure 12.13. Here, the memory manager first checks the unit of memory immediately preceding block F to see if the preceding block (call it P) is also free. If it is, then the memory unit before P 's tag bit stores the size of P , thus indicating the position for the beginning of the block in memory. P can then simply have its size extended to include block F . If block P is not free, then we just add block F to the freelist. Finally, we also check the bit following the end of block F . If this bit indicates that the following block (call it S) is free, then S is removed from the freelist and the size of F is extended appropriately.

We now consider how a "suitable" free block is selected to service a memory request. To illustrate the process, assume that we have a memory pool with 200 units of storage. After some series of allocation requests and releases, we have reached a point where there are four free blocks on the freelist of sizes 25, 35, 32, and 45 (in that order). Assume that a request is made for 30 units of storage. For our examples, we ignore the overhead imposed for the tag, link, and size fields discussed above.

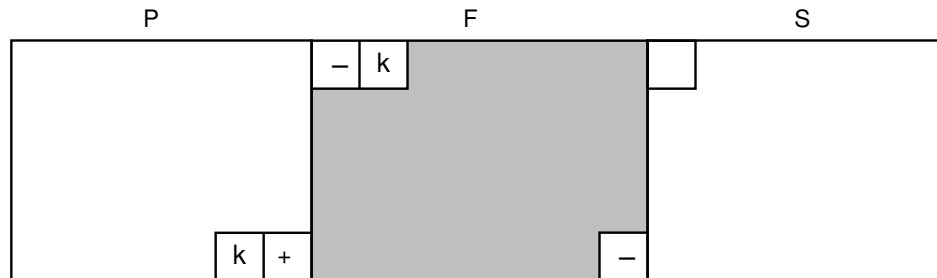


Figure 12.13 Adding block *F* to the freelist. The word immediately preceding the start of *F* in the memory pool stores the tag bit of the preceding block *P*. If *P* is free, merge *F* into *P*. We find the end of *F* by using *F*'s size field. The word following the end of *F* is the tag field for block *S*. If *S* is free, merge it into *F*.

The simplest method for selecting a block would be to move down the free block list until a block of size at least 30 is found. Any remaining space in this block is left on the freelist. If we begin at the beginning of the list and work down to the first free block at least as large as 30, we select the block of size 35. 30 units of storage will be allocated, leaving a free block with 5 units of space. Because this approach selects the first block with enough space, it is called **first fit**. A simple variation that will improve performance is, instead of always beginning at the head of the freelist, remember the last position reached in the previous search and start from there. When the end of the freelist is reached, search begins again at the head of the freelist. This modification reduces the number of unnecessary searches through small blocks that were passed over by previous requests.

There is a potential disadvantage to first fit: It might “waste” larger blocks by breaking them up, and so they will not be available for large requests later. A strategy that avoids using large blocks unnecessarily is called **best fit**. Best fit looks at the entire list and picks the smallest block that is at least as large as the request (i.e., the “best” or closest fit to the request). Continuing with the preceding example, the best fit for a request of 30 units is the block of size 32, leaving a remainder of size 2. Best fit has the disadvantage that it requires that the entire list be searched. Another problem is that the remaining portion of the best-fit block is likely to be small, and thus useless for future requests. In other words, best fit tends to maximize problems of external fragmentation while it minimizes the chance of not being able to service an occasional large request.

A strategy contrary to best fit might make sense because it tends to minimize the effects of external fragmentation. This is called **worst fit**, which always allocates the largest block on the list hoping that the remainder of the block will be useful for servicing a future request. In our example, the worst fit is the block of size 45, leaving a remainder of size 15. If there are a few unusually large requests, this approach will have less chance of servicing them. If requests generally tend

to be of the same size, then this might be an effective strategy. Like best fit, worst fit requires searching the entire freelist at each memory request to find the largest block. Alternatively, the freelist can be ordered from largest to smallest free block, possibly by using a priority queue implementation.

Which strategy is best? It depends on the expected types of memory requests. If the requests are of widely ranging size, best fit might work well. If the requests tend to be of similar size, with rare large and small requests, first or worst fit might work well. Unfortunately, there are always request patterns that one of the three sequential fit methods will service, but which the other two will not be able to service. For example, if the series of requests 600, 650, 900, 500, 100 is made to a freelist containing blocks 500, 700, 650, 900 (in that order), the requests can all be serviced by first fit, but not by best fit. Alternatively, the series of requests 600, 500, 700, 900 can be serviced by best fit but not by first fit on this same freelist.

Buddy Methods

Sequential-fit methods rely on a linked list of free blocks, which must be searched for a suitable block at each memory request. Thus, the time to find a suitable free block would be $\Theta(n)$ in the worst case for a freelist containing n blocks. Merging adjacent free blocks is somewhat complicated. Finally, we must either use additional space for the linked list, or use space within the memory pool to support the memory manager operations. In the second option, both free and reserved blocks require tag and size fields. Fields in free blocks do not cost any space (because they are stored in memory that is not otherwise being used), but fields in reserved blocks create additional overhead.

The buddy system solves most of these problems. Searching for a block of the proper size is efficient, merging adjacent free blocks is simple, and no tag or other information fields need be stored within reserved blocks. The buddy system assumes that memory is of size 2^N for some integer N . Both free and reserved blocks will always be of size 2^k for $k \leq N$. At any given time, there might be both free and reserved blocks of various sizes. The buddy system keeps a separate list for free blocks of each size. There can be at most N such lists, because there can only be N distinct block sizes.

When a request comes in for m words, we first determine the smallest value of k such that $2^k \geq m$. A block of size 2^k is selected from the free list for that block size if one exists. The buddy system does not worry about internal fragmentation: The entire block of size 2^k is allocated.

If no block of size 2^k exists, the next larger block is located. This block is split in half (repeatedly if necessary) until the desired block of size 2^k is created. Any other blocks generated as a by-product of this splitting process are placed on the appropriate freelists.

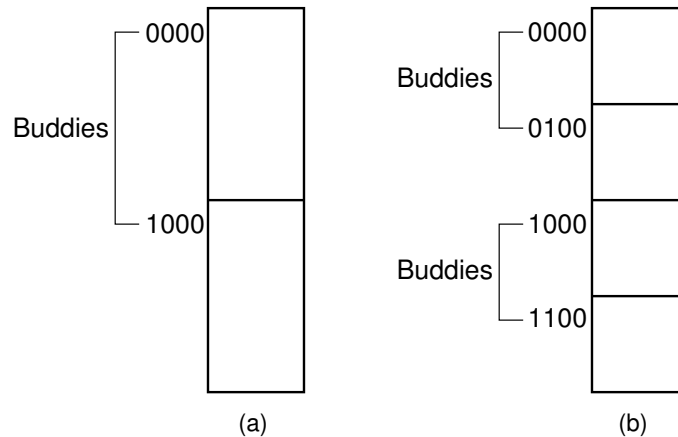


Figure 12.14 Example of the buddy system. (a) Blocks of size 8. (b) Blocks of size 4.

The disadvantage of the buddy system is that it allows internal fragmentation. For example, a request for 257 words will require a block of size 512. The primary advantages of the buddy system are (1) there is less external fragmentation; (2) search for a block of the right size is cheaper than, say, best fit because we need only find the first available block on the block list for blocks of size 2^k ; and (3) merging adjacent free blocks is easy.

The reason why this method is called the buddy system is because of the way that merging takes place. The **buddy** for any block of size 2^k is another block of the same size, and with the same address (i.e., the byte position in memory, read as a binary value) except that the k th bit is reversed. For example, the block of size 8 with beginning address 0000 in Figure 12.14(a) has buddy with address 1000. Likewise, in Figure 12.14(b), the block of size 4 with address 0000 has buddy 0100. If free blocks are sorted by address value, the buddy can be found by searching the correct block-size list. Merging simply requires that the address for the combined buddies be moved to the freelist for the next larger block size.

Other Memory Allocation Methods

In addition to sequential-fit and buddy methods, there are many ad hoc approaches to memory management. If the application is sufficiently complex, it might be desirable to break available memory into several memory **zones**, each with a different memory management scheme. For example, some zones might have a simple memory access pattern of first-in, first-out. This zone can therefore be managed efficiently by using a simple queue. Another zone might allocate only records of fixed size, and so can be managed with a simple freelist as described in Section 4.1.2. Other zones might need one of the general-purpose memory allocation methods discussed in this section. The advantage of zones is that some portions of memory

can be managed more efficiently. The disadvantage is that one zone might fill up while other zones have excess free memory if the zone sizes are chosen poorly.

Another approach to memory management is to impose a standard size on all memory requests. We have seen an example of this concept already in disk file management, where all files are allocated in multiples of the cluster size. This approach leads to internal fragmentation, but managing files composed of clusters is easier than managing arbitrarily sized files. The cluster scheme also allows us to relax the restriction that the memory request be serviced by a contiguous block of memory. Most disk file managers and operating system main memory managers work on a cluster or page system. Block management is usually done with a buffer pool to allocate available blocks in main memory efficiently.

12.3.2 Failure Policies and Garbage Collection

At some point when processing a series of requests, a memory manager could encounter a request for memory that it cannot satisfy. In some situations, there might be nothing that can be done: There simply might not be enough free memory to service the request, and the application may require that the request be serviced immediately. In this case, the memory manager has no option but to return an error, which could in turn lead to a failure of the application program. However, in many cases there are alternatives to simply returning an error. The possible options are referred to collectively as **failure policies**.

In some cases, there might be sufficient free memory to satisfy the request, but it is scattered among small blocks. This can happen when using a sequential-fit memory allocation method, where external fragmentation has led to a series of small blocks that collectively could service the request. In this case, it might be possible to **compact** memory by moving the reserved blocks around so that the free space is collected into a single block. A problem with this approach is that the application must somehow be able to deal with the fact that its data have now been moved to different locations. If the application program relies on the absolute positions of the data in any way, this would be disastrous. One approach for dealing with this problem involves the handles returned by the memory manager. A handle works as a second level of indirection to a memory location. The memory allocation routine does not return a pointer to the block of storage, but rather a pointer to a handle that in turn gives access to the storage. The handle never moves its position, but the position of the block might be moved and the value of the handle updated. Of course, this requires that the memory manager keep track of the handles and how they associate with the stored messages. Figure 12.15 illustrates the concept.

Another failure policy that might work in some applications is to defer the memory request until sufficient memory becomes available. For example, a multi-tasking operating system could adopt the strategy of not allowing a process to run until there is sufficient memory available. While such a delay might be annoying

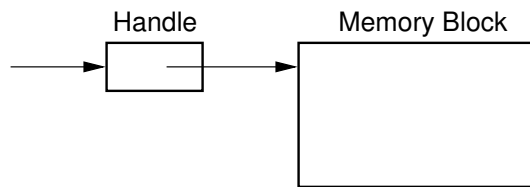


Figure 12.15 Using handles for dynamic memory management. The memory manager returns the address of the handle in response to a memory request. The handle stores the address of the actual memory block. In this way, the memory block might be moved (with its address updated in the handle) without disrupting the application program.

to the user, it is better than halting the entire system. The assumption here is that other processes will eventually terminate, freeing memory.

Another option might be to allocate more memory to the memory manager. In a zoned memory allocation system where the memory manager is part of a larger system, this might be a viable option. In a C++ program that implements its own memory manager, it might be possible to get more memory from the system-level **new** operator, such as is done by the freelist of Section 4.1.2.

The last failure policy that we will consider is **garbage collection**. Consider the following series of statements.

```
int* p = new int[5];
int* q = new int[10];
p = q;
```

While in Java this would be no problem (due to automatic garbage collection), in languages such as C++, this would be considered bad form because the original space allocated to **p** is lost as a result of the third assignment. This space cannot be used again by the program. Such lost memory is referred to as **garbage**, also known as a **memory leak**. When no program variable points to a block of space, no future access to that space is possible. Of course, if another variable had first been assigned to point to **p**'s space, then reassigning **p** would not create garbage.

Some programming languages take a different view towards garbage. In particular, the LISP programming language uses the multilist representation of Figure 12.5, and all storage is in the form either of internal nodes with two pointers or atoms. Figure 12.16 shows a typical collection of LISP structures, headed by variables named **A**, **B**, and **C**, along with a freelist.

In LISP, list objects are constantly being put together in various ways as temporary variables, and then all reference to them is lost when the object is no longer needed. Thus, garbage is normal in LISP, and in fact cannot be avoided during routine program behavior. When LISP runs out of memory, it resorts to a garbage collection process to recover the space tied up in garbage. Garbage collection consists of examining the managed memory pool to determine which parts are still

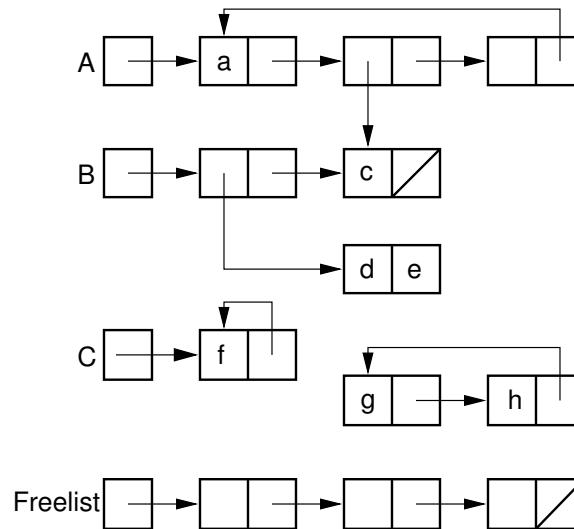


Figure 12.16 Example of LISP list variables, including the system freelist.

The Java programming language also makes use of garbage collection. As in LISP, it is common practice in Java to allocate dynamic memory as needed, and to later drop all references to that memory. The garbage collector is responsible for reclaiming such unused space as necessary. This might require extra time when running the program, but it makes life considerably easier for the programmer. In contrast, many large applications written in C++ (even commonly used commercial software) contain memory leaks that will in time cause the program to fail.

Several algorithms have been used for garbage collection. One is the **reference count** algorithm. Here, every dynamically allocated memory block includes space for a count field. Whenever a pointer is directed to a memory block, the reference count is increased. Whenever a pointer is directed away from a memory block, the reference count is decreased. If the count ever becomes zero, then the memory block is considered garbage and is immediately placed in free store. This approach has the advantage that it does not require an explicit garbage collection phase, because information is put in free store immediately when it becomes garbage.

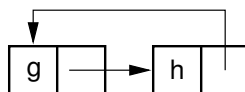


Figure 12.17 Garbage cycle example. All memory elements in the cycle have non-zero reference counts because each element has one pointer to it, even though the entire cycle is garbage (i.e., no static variable in the program points to it).

Reference counts are used by the UNIX file system. Files can have multiple names, called links. The file system keeps a count of the number of links to each file. Whenever a file is “deleted,” in actuality its link field is simply reduced by one. If there is another link to the file, then no space is recovered by the file system. When the number of links goes to zero, the file’s space becomes available for reuse.

Reference counts have several major disadvantages. First, a reference count must be maintained for each memory object. This works well when the objects are large, such as a file. However, it will not work well in a system such as LISP where the memory objects typically consist of two pointers or a value (an atom). Another major problem occurs when garbage contains cycles. Consider Figure 12.17. Here each memory object is pointed to once, but the collection of objects is still garbage because no pointer points to the collection. Thus, reference counts only work when the memory objects are linked together without cycles, such as the UNIX file system where files can only be organized as a DAG.

Another approach to garbage collection is the **mark/sweep** strategy. Here, each memory object needs only a single mark bit rather than a reference counter field. When free store is exhausted, a separate garbage collection phase takes place as follows.

1. Clear all mark bits.
2. Perform depth-first search (DFS) following pointers beginning with each variable on the system’s list of static variables. Each memory element encountered during the DFS has its mark bit turned on.
3. A “sweep” is made through the memory pool, visiting all elements. Unmarked elements are considered garbage and placed in free store.

The advantages of the mark/sweep approach are that it needs less space than is necessary for reference counts, and it works for cycles. However, there is a major disadvantage. This is a “hidden” space requirement needed to do the processing. DFS is a recursive algorithm: Either it must be implemented recursively, in which case the compiler’s runtime system maintains a stack, or else the memory manager can maintain its own stack. What happens if all memory is contained in a single linked list? Then the depth of the recursion (or the size of the stack) is the number of memory cells! Unfortunately, the space for the DFS stack must be available at the worst conceivable time, that is, when free memory has been exhausted.

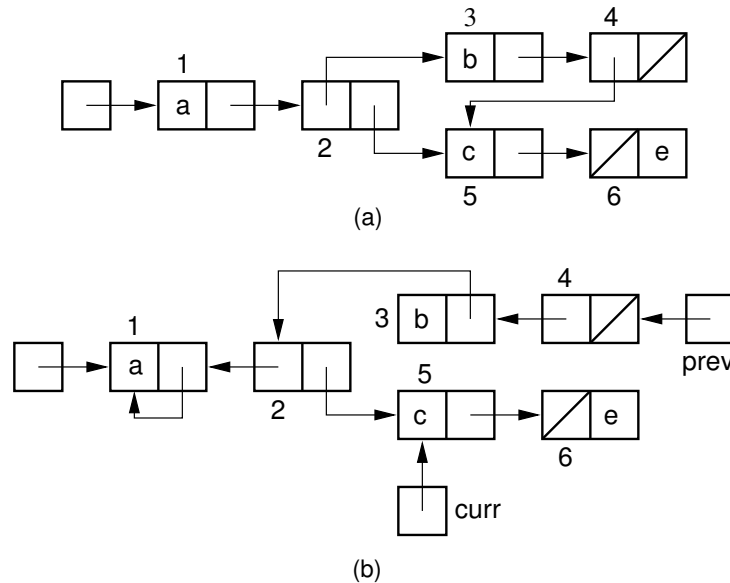


Figure 12.18 Example of the Deutsch-Schorr-Waite garbage collection algorithm. (a) The initial multilist structure. (b) The multilist structure of (a) at the instant when link node 5 is being processed by the garbage collection algorithm. A chain of pointers stretching from variable **prev** to the head node of the structure has been (temporarily) created by the garbage collection algorithm.

Fortunately, a clever technique allows DFS to be performed without requiring additional space for a stack. Instead, the structure being traversed is used to hold the stack. At each step deeper into the traversal, instead of storing a pointer on the stack, we “borrow” the pointer being followed. This pointer is set to point back to the node we just came from in the previous step, as illustrated by Figure 12.18. Each borrowed pointer stores an additional bit to tell us whether we came down the left branch or the right branch of the link node being pointed to. At any given instant we have passed down only one path from the root, and we can follow the trail of pointers back up. As we return (equivalent to popping the recursion stack), we set the pointer back to its original position so as to return the structure to its original condition. This is known as the Deutsch-Schorr-Waite garbage collection algorithm.

12.4 Further Reading

For information on LISP, see *The Little LISP* by Friedman and Felleisen [FF89]. Another good LISP reference is *Common LISP: The Language* by Guy L. Steele [Ste90]. For information on Emacs, which is both an excellent text editor and a programming environment, see the *GNU Emacs Manual* by Richard Stallman

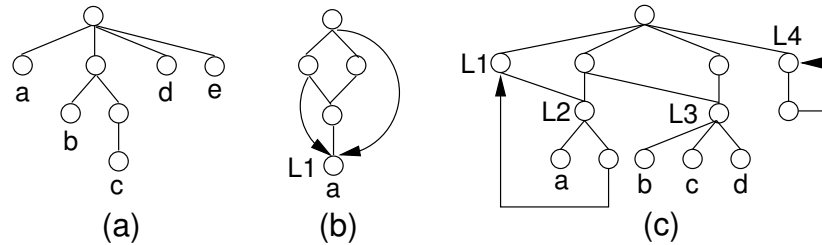


Figure 12.19 Some example multilists.

[Sta11b]. You can get more information about Java's garbage collection system from *The Java Programming Language* by Ken Arnold and James Gosling [AG06].

For more details on sparse matrix representations, the Yale representation is described by Eisenstat, Schultz and Sherman [ESS81]. The MATLAB sparse matrix representation is described by Gilbert, Moler, and Schreiber [GMS91].

An introductory text on operating systems covers many topics relating to memory management issues, including layout of files on disk and caching of information in main memory. All of the topics covered here on memory management, buffer pools, and paging are relevant to operating system implementation. For example, see *Operating Systems* by William Stallings [Sta11a].

12.5 Exercises

12.1 For each of the following bracket notation descriptions, draw the equivalent multilist in graphical form such as shown in Figure 12.2.

- (a) $\langle a, b, \langle c, d, e \rangle, \langle f, \langle g, h \rangle \rangle$
- (b) $\langle a, b, \langle c, d, L1 : e \rangle, L1 \rangle$
- (c) $\langle L1 : a, L1, \langle L2 : b \rangle, L2, \langle L1 \rangle \rangle$

12.2 (a) Show the bracket notation for the list of Figure 12.19(a).

(b) Show the bracket notation for the list of Figure 12.19(b).

(c) Show the bracket notation for the list of Figure 12.19(c).

12.3 Given the linked representation of a pure list such as

$$\langle x_1, \langle y_1, y_2, \langle z_1, z_2 \rangle, y_4 \rangle, \langle w_1, w_2 \rangle, x_4 \rangle,$$

write an in-place reversal algorithm to reverse the sublists at all levels including the topmost level. For this example, the result would be a linked representation corresponding to

$$\langle x_4, \langle w_2, w_1 \rangle, \langle y_4, \langle z_2, z_1 \rangle, y_2, y_1 \rangle, x_1 \rangle.$$

12.4 What fraction of the values in a matrix must be zero for the sparse matrix representation of Section 12.2 to be more space efficient than the standard two-dimensional matrix representation when data values require eight bytes, array indices require two bytes, and pointers require four bytes?

- 12.5** Write a function to add an element at a given position to the sparse matrix representation of Section 12.2.
- 12.6** Write a function to delete an element from a given position in the sparse matrix representation of Section 12.2.
- 12.7** Write a function to transpose a sparse matrix as represented in Section 12.2.
- 12.8** Write a function to add two sparse matrices as represented in Section 12.2.
- 12.9** Write memory manager allocation and deallocation routines for the situation where all requests and releases follow a last-requested, first-released (stack) order.
- 12.10** Write memory manager allocation and deallocation routines for the situation where all requests and releases follow a last-requested, last-released (queue) order.
- 12.11** Show the result of allocating the following blocks from a memory pool of size 1000 using first fit for each series of block requests. State if a given request cannot be satisfied.
- (a) Take 300 (call this block A), take 500, release A, take 200, take 300.
 - (b) Take 200 (call this block A), take 500, release A, take 200, take 300.
 - (c) Take 500 (call this block A), take 300, release A, take 300, take 200.
- 12.12** Show the result of allocating the following blocks from a memory pool of size 1000 using best fit for each series of block requests. State if a given request cannot be satisfied.
- (a) Take 300 (call this block A), take 500, release A, take 200, take 300.
 - (b) Take 200 (call this block A), take 500, release A, take 200, take 300.
 - (c) Take 500 (call this block A), take 300, release A, take 300, take 200.
- 12.13** Show the result of allocating the following blocks from a memory pool of size 1000 using worst fit for each series of block requests. State if a given request cannot be satisfied.
- (a) Take 300 (call this block A), take 500, release A, take 200, take 300.
 - (b) Take 200 (call this block A), take 500, release A, take 200, take 300.
 - (c) Take 500 (call this block A), take 300, release A, take 300, take 200.
- 12.14** Assume that the memory pool contains three blocks of free storage. Their sizes are 1300, 2000, and 1000. Give examples of storage requests for which
- (a) first-fit allocation will work, but not best fit or worst fit.
 - (b) best-fit allocation will work, but not first fit or worst fit.
 - (c) worst-fit allocation will work, but not first fit or best fit.

12.6 Projects

- 12.1** Implement the orthogonal list sparse matrix representation of Section 12.2. Your implementation should support the following operations on the matrix:

- insert an element at a given position,
- delete an element from a given position,
- return the value of the element at a given position,
- take the transpose of a matrix,
- add two matrices, and
- multiply two matrices.

12.2 Implement the Yale model for sparse matrices described at the end of Section 12.2. Your implementation should support the following operations on the matrix:

- insert an element at a given position,
- delete an element from a given position,
- return the value of the element at a given position,
- take the transpose of a matrix,
- add two matrices, and
- multiply two matrices.

12.3 Implement the **MemManager** ADT shown at the beginning of Section 12.3. Use a separate linked list to implement the freelist. Your implementation should work for any of the three sequential-fit methods: first fit, best fit, and worst fit. Test your system empirically to determine under what conditions each method performs well.

12.4 Implement the **MemManager** ADT shown at the beginning of Section 12.3. Do not use separate memory for the free list, but instead embed the free list into the memory pool as shown in Figure 12.12. Your implementation should work for any of the three sequential-fit methods: first fit, best fit, and worst fit. Test your system empirically to determine under what conditions each method performs well.

12.5 Implement the **MemManager** ADT shown at the beginning of Section 12.3 using the buddy method of Section 12.3.1. Your system should support requests for blocks of a specified size and release of previously requested blocks.

12.6 Implement the Deutsch-Schorr-Waite garbage collection algorithm that is illustrated by Figure 12.18.

Advanced Tree Structures

This chapter introduces several tree structures designed for use in specialized applications. The trie of Section 13.1 is commonly used to store and retrieve strings. It also serves to illustrate the concept of a key space decomposition. The AVL tree and splay tree of Section 13.2 are variants on the BST. They are examples of self-balancing search trees and have guaranteed good performance regardless of the insertion order for records. An introduction to several spatial data structures used to organize point data by xy -coordinates is presented in Section 13.3.

Descriptions of the fundamental operations are given for each data structure. One purpose for this chapter is to provide opportunities for class programming projects, so detailed implementations are left to the reader.

13.1 Tries

Recall that the shape of a BST is determined by the order in which its data records are inserted. One permutation of the records might yield a balanced tree while another might yield an unbalanced tree, with the extreme case becoming the shape of a linked list. The reason is that the value of the key stored in the root node splits the key range into two parts: those key values less than the root's key value, and those key values greater than the root's key value. Depending on the relationship between the root node's key value and the distribution of the key values for the other records in the tree, the resulting BST might be balanced or unbalanced. Thus, the BST is an example of a data structure whose organization is based on an **object space decomposition**, so called because the decomposition of the key range is driven by the objects (i.e., the key values of the data records) stored in the tree.

The alternative to object space decomposition is to predefine the splitting position within the key range for each node in the tree. In other words, the root could be predefined to split the key range into two equal halves, regardless of the particular values or order of insertion for the data records. Those records with keys in the lower half of the key range will be stored in the left subtree, while those records

with keys in the upper half of the key range will be stored in the right subtree. While such a decomposition rule will not necessarily result in a balanced tree (the tree will be unbalanced if the records are not well distributed within the key range), at least the shape of the tree will not depend on the order of key insertion. Furthermore, the depth of the tree will be limited by the resolution of the key range; that is, the depth of the tree can never be greater than the number of bits required to store a key value. For example, if the keys are integers in the range 0 to 1023, then the resolution for the key is ten bits. Thus, two keys can be identical only until the tenth bit. In the worst case, two keys will follow the same path in the tree only until the tenth branch. As a result, the tree will never be more than ten levels deep. In contrast, a BST containing n records could be as much as n levels deep.

Splitting based on predetermined subdivisions of the key range is called **key space decomposition**. In computer graphics, the technique is known as **image space** decomposition, and this term is sometimes used to describe the process for data structures as well. A data structure based on key space decomposition is called a **trie**. Folklore has it that “trie” comes from “retrieval.” Unfortunately, that would imply that the word is pronounced “tree,” which would lead to confusion with regular use of the word “tree.” “Trie” is actually pronounced as “try.”

Like the B^+ -tree, a trie stores data records only in leaf nodes. Internal nodes serve as placeholders to direct the search process, but since the split points are predetermined, internal nodes need not store “traffic-directing” key values. Figure 13.1 illustrates the trie concept. Upper and lower bounds must be imposed on the key values so that we can compute the middle of the key range. Because the largest value inserted in this example is 120, a range from 0 to 127 is assumed, as 128 is the smallest power of two greater than 120. The binary value of the key determines whether to select the left or right branch at any given point during the search. The most significant bit determines the branch direction at the root. Figure 13.1 shows a **binary trie**, so called because in this example the trie structure is based on the value of the key interpreted as a binary number, which results in a binary tree.

The Huffman coding tree of Section 5.6 is another example of a binary trie. All data values in the Huffman tree are at the leaves, and each branch splits the range of possible letter codes in half. The Huffman codes are actually reconstructed from the letter positions within the trie.

These are examples of binary tries, but tries can be built with any branching factor. Normally the branching factor is determined by the alphabet used. For binary numbers, the alphabet is $\{0, 1\}$ and a binary trie results. Other alphabets lead to other branching factors.

One application for tries is to store a dictionary of words. Such a trie will be referred to as an **alphabet trie**. For simplicity, our examples will ignore case in letters. We add a special character (\$) to the 26 standard English letters. The \$ character is used to represent the end of a string. Thus, the branching factor for

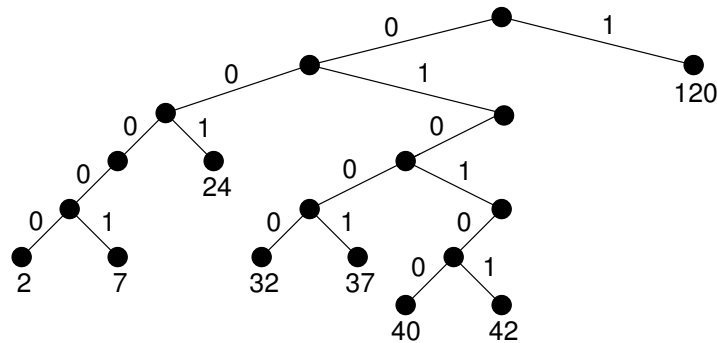


Figure 13.1 The binary trie for the collection of values 2, 7, 24, 31, 37, 40, 42, 120. All data values are stored in the leaf nodes. Edges are labeled with the value of the bit used to determine the branching direction of each node. The binary form of the key value determines the path to the record, assuming that each key is represented as a 7-bit value representing a number in the range 0 to 127.

each node is (up to) 27. Once constructed, the alphabet trie is used to determine if a given word is in the dictionary. Consider searching for a word in the alphabet trie of Figure 13.2. The first letter of the search word determines which branch to take from the root, the second letter determines which branch to take at the next level, and so on. Only the letters that lead to a word are shown as branches. In Figure 13.2(b) the leaf nodes of the trie store a copy of the actual words, while in Figure 13.2(a) the word is built up from the letters associated with each branch.

One way to implement a node of the alphabet trie is as an array of 27 pointers indexed by letter. Because most nodes have branches to only a small fraction of the possible letters in the alphabet, an alternate implementation is to use a linked list of pointers to the child nodes, as in Figure 6.9.

The depth of a leaf node in the alphabet trie of Figure 13.2(b) has little to do with the number of nodes in the trie, or even with the length of the corresponding string. Rather, a node's depth depends on the number of characters required to distinguish this node's word from any other. For example, if the words "anteater" and "antelope" are both stored in the trie, it is not until the fifth letter that the two words can be distinguished. Thus, these words must be stored at least as deep as level five. In general, the limiting factor on the depth of nodes in the alphabet trie is the length of the words stored.

Poor balance and clumping can result when certain prefixes are heavily used. For example, an alphabet trie storing the common words in the English language would have many words in the "th" branch of the tree, but none in the "zq" branch.

Any multiway branching trie can be replaced with a binary trie by replacing the original trie's alphabet with an equivalent binary code. Alternatively, we can use the techniques of Section 6.3.4 for converting a general tree to a binary tree without modifying the alphabet.

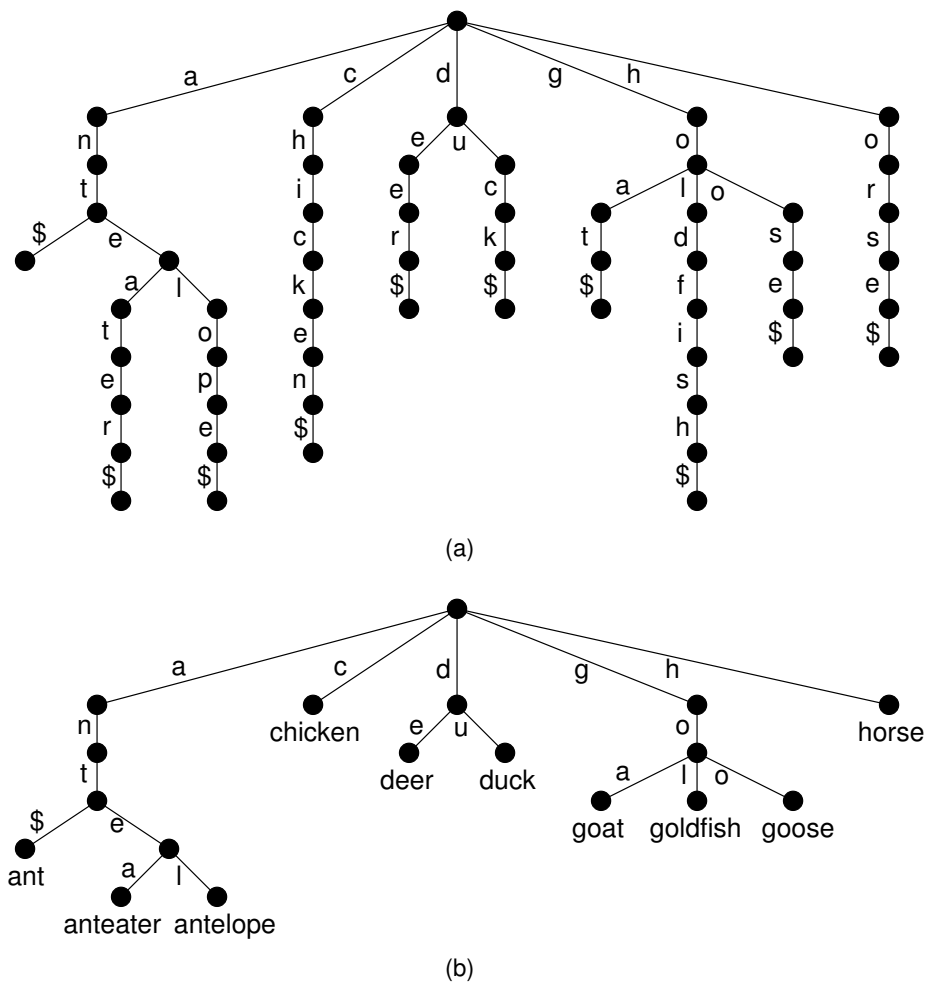


Figure 13.2 Two variations on the alphabet trie representation for a set of ten words. (a) Each node contains a set of links corresponding to single letters, and each letter in the set of words has a corresponding link. “\$” is used to indicate the end of a word. Internal nodes direct the search and also spell out the word one letter per link. The word need not be stored explicitly. “\$” is needed to recognize the existence of words that are prefixes to other words, such as ‘ant’ in this example. (b) Here the trie extends only far enough to discriminate between the words. Leaf nodes of the trie each store a complete word; internal nodes merely direct the search.

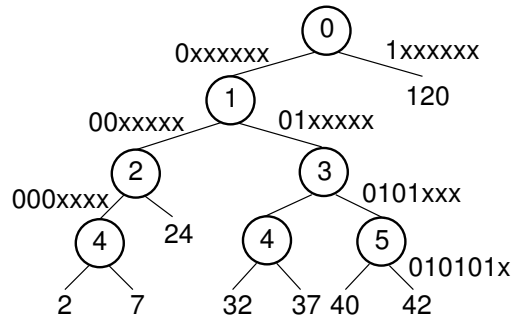


Figure 13.3 The PAT trie for the collection of values 2, 7, 24, 32, 37, 40, 42, 120. Contrast this with the binary trie of Figure 13.1. In the PAT trie, all data values are stored in the leaf nodes, while internal nodes store the bit position used to determine the branching decision, assuming that each key is represented as a 7-bit value representing a number in the range 0 to 127. Some of the branches in this PAT trie have been labeled to indicate the binary representation for all values in that subtree. For example, all values in the left subtree of the node labeled 0 must have value 0xxxxxx (where x means that bit can be either a 0 or a 1). All nodes in the right subtree of the node labeled 3 must have value 0101xxx. However, we can skip branching on bit 2 for this subtree because all values currently stored have a value of 0 for that bit.

The trie implementations illustrated by Figures 13.1 and 13.2 are potentially quite inefficient as certain key sets might lead to a large number of nodes with only a single child. A variant on trie implementation is known as PATRICIA, which stands for “Practical Algorithm To Retrieve Information Coded In Alphanumeric.” In the case of a binary alphabet, a PATRICIA trie (referred to hereafter as a PAT trie) is a full binary tree that stores data records in the leaf nodes. Internal nodes store only the position within the key’s bit pattern that is used to decide on the next branching point. In this way, internal nodes with single children (equivalently, bit positions within the key that do not distinguish any of the keys within the current subtree) are eliminated. A PAT trie corresponding to the values of Figure 13.1 is shown in Figure 13.3.

Example 13.1 When searching for the value 7 (0000111 in binary) in the PAT trie of Figure 13.3, the root node indicates that bit position 0 (the leftmost bit) is checked first. Because the 0th bit for value 7 is 0, take the left branch. At level 1, branch depending on the value of bit 1, which again is 0. At level 2, branch depending on the value of bit 2, which again is 0. At level 3, the index stored in the node is 4. This means that bit 4 of the key is checked next. (The value of bit 3 is irrelevant, because all values stored in that subtree have the same value at bit position 3.) Thus, the single branch that extends from the equivalent node in Figure 13.1 is just skipped. For key value 7, bit 4 has value 1, so the rightmost branch is taken. Because

this leads to a leaf node, the search key is compared against the key stored in that node. If they match, then the desired record has been found.

Note that during the search process, only a single bit of the search key is compared at each internal node. This is significant, because the search key could be quite large. Search in the PAT trie requires only a single full-key comparison, which takes place once a leaf node has been reached.

Example 13.2 Consider the situation where we need to store a library of DNA sequences. A DNA sequence is a series of letters, usually many thousands of characters long, with the string coming from an alphabet of only four letters that stand for the four amino acids making up a DNA strand. Similar DNA sequences might have long sections of their string that are identical. The PAT trie would avoid making multiple full key comparisons when searching for a specific sequence.

13.2 Balanced Trees

We have noted several times that the BST has a high risk of becoming unbalanced, resulting in excessively expensive search and update operations. One solution to this problem is to adopt another search tree structure such as the 2-3 tree or the binary trie. An alternative is to modify the BST access functions in some way to guarantee that the tree performs well. This is an appealing concept, and it works well for heaps, whose access functions maintain the heap in the shape of a complete binary tree. Unfortunately, requiring that the BST always be in the shape of a complete binary tree requires excessive modification to the tree during update, as discussed in Section 10.3.

If we are willing to weaken the balance requirements, we can come up with alternative update routines that perform well both in terms of cost for the update and in balance for the resulting tree structure. The AVL tree works in this way, using insertion and deletion routines altered from those of the BST to ensure that, for every node, the depths of the left and right subtrees differ by at most one. The AVL tree is described in Section 13.2.1.

A different approach to improving the performance of the BST is to not require that the tree always be balanced, but rather to expend some effort toward making the BST more balanced every time it is accessed. This is a little like the idea of path compression used by the UNION/FIND algorithm presented in Section 6.2. One example of such a compromise is called the **splay tree**. The splay tree is described in Section 13.2.2.

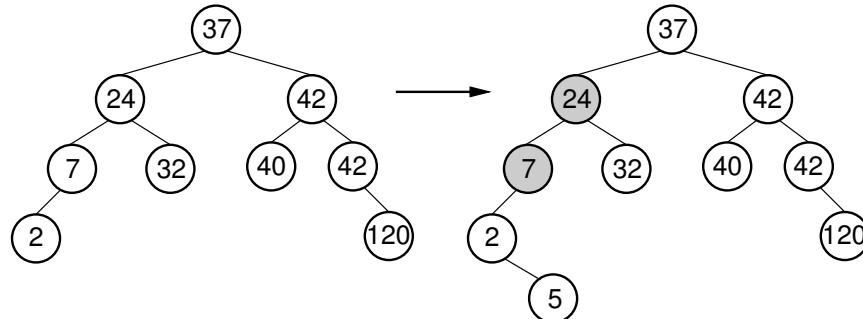


Figure 13.4 Example of an insert operation that violates the AVL tree balance property. Prior to the insert operation, all nodes of the tree are balanced (i.e., the depths of the left and right subtrees for every node differ by at most one). After inserting the node with value 5, the nodes with values 7 and 24 are no longer balanced.

13.2.1 The AVL Tree

The AVL tree (named for its inventors Adelson-Velskii and Landis) should be viewed as a BST with the following additional property: For every node, the heights of its left and right subtrees differ by at most 1. As long as the tree maintains this property, if the tree contains n nodes, then it has a depth of at most $O(\log n)$. As a result, search for any node will cost $O(\log n)$, and if the updates can be done in time proportional to the depth of the node inserted or deleted, then updates will also cost $O(\log n)$, even in the worst case.

The key to making the AVL tree work is to alter the insert and delete routines so as to maintain the balance property. Of course, to be practical, we must be able to implement the revised update routines in $\Theta(\log n)$ time.

Consider what happens when we insert a node with key value 5, as shown in Figure 13.4. The tree on the left meets the AVL tree balance requirements. After the insertion, two nodes no longer meet the requirements. Because the original tree met the balance requirement, nodes in the new tree can only be unbalanced by a difference of at most 2 in the subtrees. For the bottommost unbalanced node, call it S , there are 4 cases:

1. The extra node is in the left child of the left child of S .
2. The extra node is in the right child of the left child of S .
3. The extra node is in the left child of the right child of S .
4. The extra node is in the right child of the right child of S .

Cases 1 and 4 are symmetrical, as are cases 2 and 3. Note also that the unbalanced nodes must be on the path from the root to the newly inserted node.

Our problem now is how to balance the tree in $O(\log n)$ time. It turns out that we can do this using a series of local operations known as **rotations**. Cases 1 and

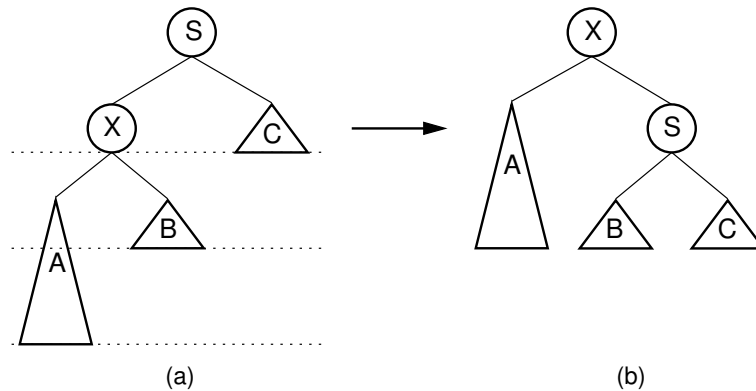


Figure 13.5 A single rotation in an AVL tree. This operation occurs when the excess node (in subtree A) is in the left child of the left child of the unbalanced node labeled S. By rearranging the nodes as shown, we preserve the BST property, as well as re-balance the tree to preserve the AVL tree balance property. The case where the excess node is in the right child of the right child of the unbalanced node is handled in the same way.

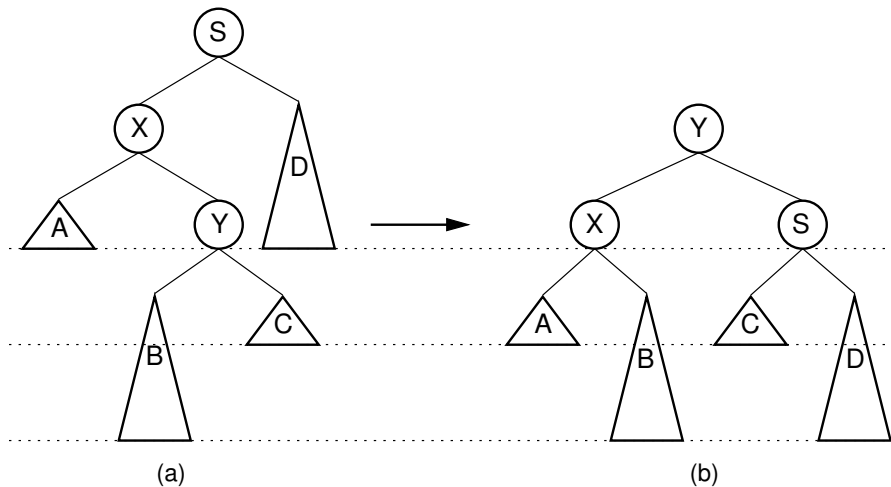


Figure 13.6 A double rotation in an AVL tree. This operation occurs when the excess node (in subtree B) is in the right child of the left child of the unbalanced node labeled S. By rearranging the nodes as shown, we preserve the BST property, as well as re-balance the tree to preserve the AVL tree balance property. The case where the excess node is in the left child of the right child of S is handled in the same way.

4 can be fixed using a **single rotation**, as shown in Figure 13.5. Cases 2 and 3 can be fixed using a **double rotation**, as shown in Figure 13.6.

The AVL tree insert algorithm begins with a normal BST insert. Then as the recursion unwinds up the tree, we perform the appropriate rotation on any node

that is found to be unbalanced. Deletion is similar; however, consideration for unbalanced nodes must begin at the level of the **deletemin** operation.

Example 13.3 In Figure 13.4 (b), the bottom-most unbalanced node has value 7. The excess node (with value 5) is in the right subtree of the left child of 7, so we have an example of Case 2. This requires a double rotation to fix. After the rotation, 5 becomes the left child of 24, 2 becomes the left child of 5, and 7 becomes the right child of 5.

13.2.2 The Splay Tree

Like the AVL tree, the splay tree is not actually a distinct data structure, but rather reimplements the BST insert, delete, and search methods to improve the performance of a BST. The goal of these revised methods is to provide guarantees on the time required by a series of operations, thereby avoiding the worst-case linear time behavior of standard BST operations. No single operation in the splay tree is guaranteed to be efficient. Instead, the splay tree access rules guarantee that a series of m operations will take $O(m \log n)$ time for a tree of n nodes whenever $m \geq n$. Thus, a single insert or search operation could take $O(n)$ time. However, m such operations are guaranteed to require a total of $O(m \log n)$ time, for an average cost of $O(\log n)$ per access operation. This is a desirable performance guarantee for any search-tree structure.

Unlike the AVL tree, the splay tree is not guaranteed to be height balanced. What is guaranteed is that the total cost of the entire series of accesses will be cheap. Ultimately, it is the cost of the series of operations that matters, not whether the tree is balanced. Maintaining balance is really done only for the sake of reaching this time efficiency goal.

The splay tree access functions operate in a manner reminiscent of the move-to-front rule for self-organizing lists from Section 9.2, and of the path compression technique for managing parent-pointer trees from Section 6.2. These access functions tend to make the tree more balanced, but an individual access will not necessarily result in a more balanced tree.

Whenever a node S is accessed (e.g., when S is inserted, deleted, or is the goal of a search), the splay tree performs a process called **splaying**. Splaying moves S to the root of the BST. When S is being deleted, splaying moves the parent of S to the root. As in the AVL tree, a splay of node S consists of a series of **rotations**. A rotation moves S higher in the tree by adjusting its position with respect to its parent and grandparent. A side effect of the rotations is a tendency to balance the tree. There are three types of rotation.

A **single rotation** is performed only if S is a child of the root node. The single rotation is illustrated by Figure 13.7. It basically switches S with its parent in a

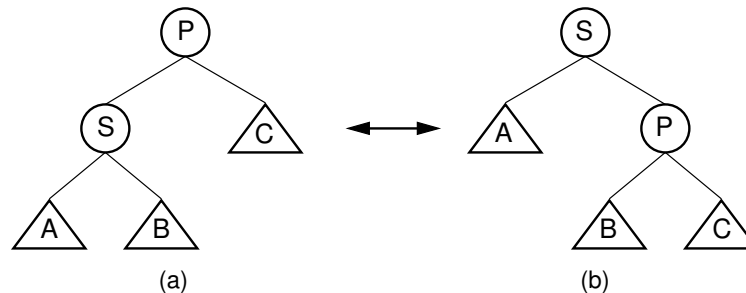


Figure 13.7 Splay tree single rotation. This rotation takes place only when the node being splayed is a child of the root. Here, node S is promoted to the root, rotating with node P . Because the value of S is less than the value of P , P must become S 's right child. The positions of subtrees A , B , and C are altered as appropriate to maintain the BST property, but the contents of these subtrees remains unchanged. (a) The original tree with P as the parent. (b) The tree after a rotation takes place. Performing a single rotation a second time will return the tree to its original shape. Equivalently, if (b) is the initial configuration of the tree (i.e., S is at the root and P is its right child), then (a) shows the result of a single rotation to splay P to the root.

way that retains the BST property. While Figure 13.7 is slightly different from Figure 13.5, in fact the splay tree single rotation is identical to the AVL tree single rotation.

Unlike the AVL tree, the splay tree requires two types of double rotation. Double rotations involve S , its parent (call it P), and S 's grandparent (call it G). The effect of a double rotation is to move S up two levels in the tree.

The first double rotation is called a **zigzag rotation**. It takes place when either of the following two conditions are met:

1. S is the left child of P , and P is the right child of G .
2. S is the right child of P , and P is the left child of G .

In other words, a zigzag rotation is used when G , P , and S form a zigzag. The zigzag rotation is illustrated by Figure 13.8.

The other double rotation is known as a **zigzig rotation**. A zigzig rotation takes place when either of the following two conditions are met:

1. S is the left child of P , which is in turn the left child of G .
2. S is the right child of P , which is in turn the right child of G .

Thus, a zigzig rotation takes place in those situations where a zigzag rotation is not appropriate. The zigzig rotation is illustrated by Figure 13.9. While Figure 13.9 appears somewhat different from Figure 13.6, in fact the zigzig rotation is identical to the AVL tree double rotation.

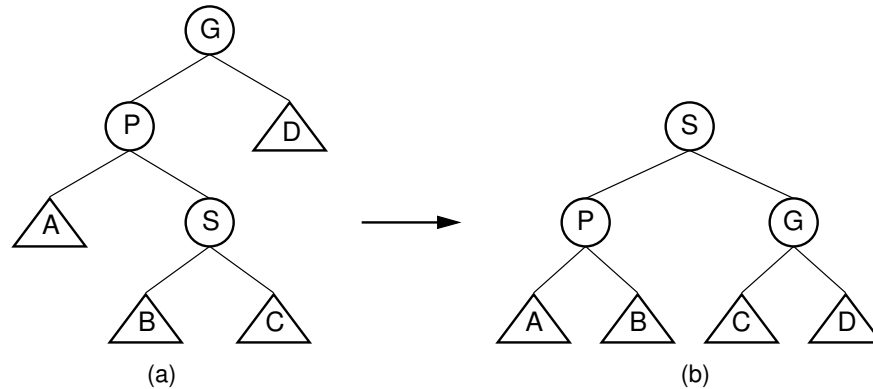


Figure 13.8 Splay tree zigzag rotation. (a) The original tree with S , P , and G in zigzag formation. (b) The tree after the rotation takes place. The positions of subtrees A , B , C , and D are altered as appropriate to maintain the BST property.

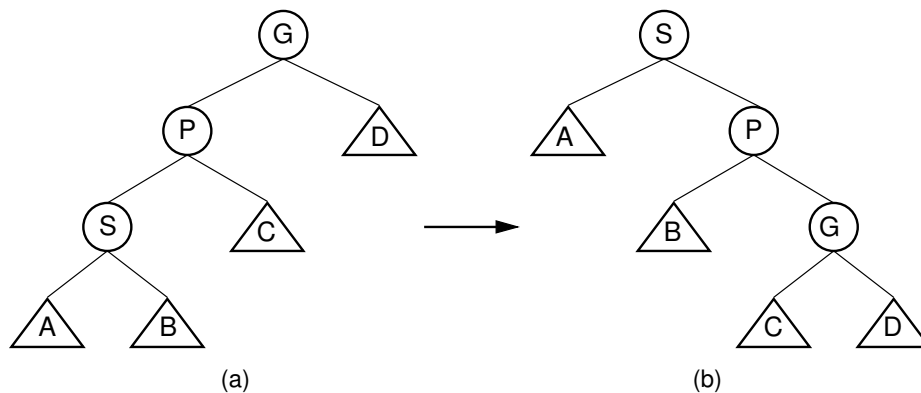


Figure 13.9 Splay tree zigzag rotation. (a) The original tree with S , P , and G in zigzag formation. (b) The tree after the rotation takes place. The positions of subtrees A , B , C , and D are altered as appropriate to maintain the BST property.

Note that zigzag rotations tend to make the tree more balanced, because they bring subtrees B and C up one level while moving subtree D down one level. The result is often a reduction of the tree's height by one. Zigzag promotions and single rotations do not typically reduce the height of the tree; they merely bring the newly accessed record toward the root.

Splaying node S involves a series of double rotations until S reaches either the root or the child of the root. Then, if necessary, a single rotation makes S the root. This process tends to re-balance the tree. Regardless of balance, splaying will make frequently accessed nodes stay near the top of the tree, resulting in reduced access cost. Proof that the splay tree meets the guarantee of $O(m \log n)$ is beyond the scope of this book. Such a proof can be found in the references in Section 13.4.

Example 13.4 Consider a search for value 89 in the splay tree of Figure 13.10(a). The splay tree's search operation is identical to searching in a BST. However, once the value has been found, it is splayed to the root. Three rotations are required in this example. The first is a zigzig rotation, whose result is shown in Figure 13.10(b). The second is a zigzag rotation, whose result is shown in Figure 13.10(c). The final step is a single rotation resulting in the tree of Figure 13.10(d). Notice that the splaying process has made the tree shallower.

13.3 Spatial Data Structures

All of the search trees discussed so far — BSTs, AVL trees, splay trees, 2-3 trees, B-trees, and tries — are designed for searching on a one-dimensional key. A typical example is an integer key, whose one-dimensional range can be visualized as a number line. These various tree structures can be viewed as dividing this one-dimensional number line into pieces.

Some databases require support for multiple keys. In other words, records can be searched for using any one of several key fields, such as name or ID number. Typically, each such key has its own one-dimensional index, and any given search query searches one of these independent indices as appropriate.

A multidimensional search key presents a rather different concept. Imagine that we have a database of city records, where each city has a name and an xy -coordinate. A BST or splay tree provides good performance for searches on city name, which is a one-dimensional key. Separate BSTs could be used to index the x - and y -coordinates. This would allow us to insert and delete cities, and locate them by name or by one coordinate. However, search on one of the two coordinates is not a natural way to view search in a two-dimensional space. Another option is to combine the xy -coordinates into a single key, say by concatenating the two coordinates, and index cities by the resulting key in a BST. That would allow search by coordinate, but would not allow for efficient two-dimensional **range queries** such as searching for all cities within a given distance of a specified point. The problem is that the BST only works well for one-dimensional keys, while a coordinate is a two-dimensional key where neither dimension is more important than the other.

Multidimensional range queries are the defining feature of a **spatial application**. Because a coordinate gives a position in space, it is called a **spatial attribute**. To implement spatial applications efficiently requires the use of **spatial data structures**. Spatial data structures store data objects organized by position and are an important class of data structures used in geographic information systems, computer graphics, robotics, and many other fields.

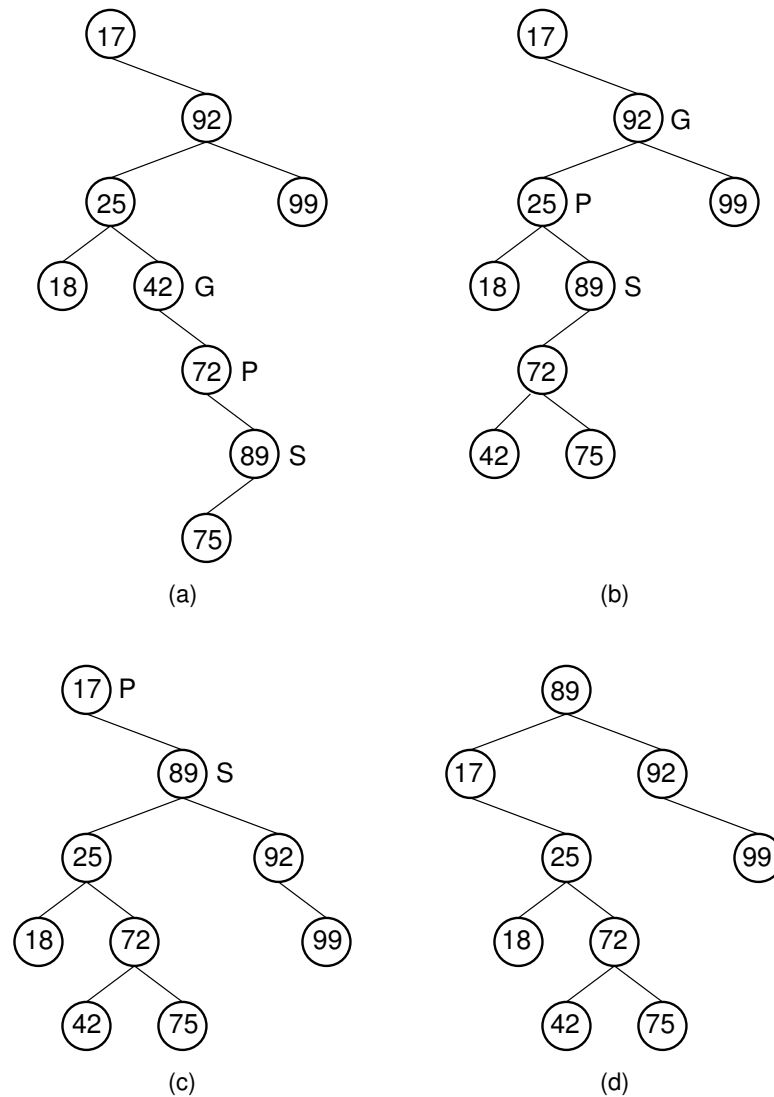


Figure 13.10 Example of splaying after performing a search in a splay tree. After finding the node with key value 89, that node is splayed to the root by performing three rotations. (a) The original splay tree. (b) The result of performing a zigzig rotation on the node with key value 89 in the tree of (a). (c) The result of performing a zigzag rotation on the node with key value 89 in the tree of (b). (d) The result of performing a single rotation on the node with key value 89 in the tree of (c). If the search had been for 91, the search would have been unsuccessful with the node storing key value 89 being that last one visited. In that case, the same splay operations would take place.

This section presents two spatial data structures for storing point data in two or more dimensions. They are the **k-d tree** and the **PR quadtree**. The k-d tree is a natural extension of the BST to multiple dimensions. It is a binary tree whose splitting decisions alternate among the key dimensions. Like the BST, the k-d tree uses object space decomposition. The PR quadtree uses key space decomposition and so is a form of trie. It is a binary tree only for one-dimensional keys (in which case it is a trie with a binary alphabet). For d dimensions it has 2^d branches. Thus, in two dimensions, the PR quadtree has four branches (hence the name “quadtree”), splitting space into four equal-sized quadrants at each branch. Section 13.3.3 briefly mentions two other variations on these data structures, the **bintree** and the **point quadtree**. These four structures cover all four combinations of object versus key space decomposition on the one hand, and multi-level binary versus 2^d -way branching on the other. Section 13.3.4 briefly discusses spatial data structures for storing other types of spatial data.

13.3.1 The K-D Tree

The k-d tree is a modification to the BST that allows for efficient processing of multidimensional keys. The k-d tree differs from the BST in that each level of the k-d tree makes branching decisions based on a particular search key associated with that level, called the **discriminator**. In principle, the k-d tree could be used to unify key searching across any arbitrary set of keys such as name and zipcode. But in practice, it is nearly always used to support search on multidimensional coordinates, such as locations in 2D or 3D space. We define the discriminator at level i to be $i \bmod k$ for k dimensions. For example, assume that we store data organized by xy -coordinates. In this case, k is 2 (there are two coordinates), with the x -coordinate field arbitrarily designated key 0, and the y -coordinate field designated key 1. At each level, the discriminator alternates between x and y . Thus, a node N at level 0 (the root) would have in its left subtree only nodes whose x values are less than N_x (because x is search key 0, and $0 \bmod 2 = 0$). The right subtree would contain nodes whose x values are greater than N_x . A node M at level 1 would have in its left subtree only nodes whose y values are less than M_y . There is no restriction on the relative values of M_x and the x values of M 's descendants, because branching decisions made at M are based solely on the y coordinate. Figure 13.11 shows an example of how a collection of two-dimensional points would be stored in a k-d tree.

In Figure 13.11 the region containing the points is (arbitrarily) restricted to a 128×128 square, and each internal node splits the search space. Each split is shown by a line, vertical for nodes with x discriminators and horizontal for nodes with y discriminators. The root node splits the space into two parts; its children further subdivide the space into smaller parts. The children's split lines do not cross the root's split line. Thus, each node in the k-d tree helps to decompose the

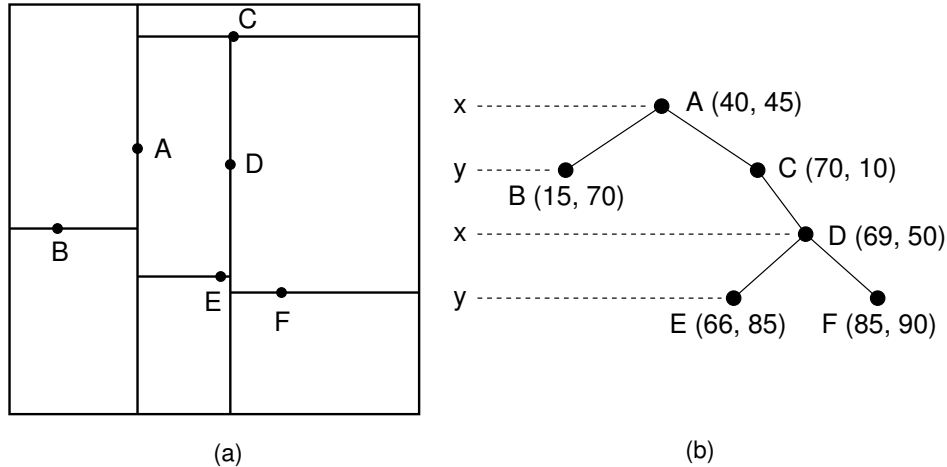


Figure 13.11 Example of a k-d tree. (a) The k-d tree decomposition for a 128×128 -unit region containing seven data points. (b) The k-d tree for the region of (a).

space into rectangles that show the extent of where nodes can fall in the various subtrees.

Searching a k-d tree for the record with a specified xy -coordinate is like searching a BST, except that each level of the k-d tree is associated with a particular discriminator.

Example 13.5 Consider searching the k-d tree for a record located at $P = (69, 50)$. First compare P with the point stored at the root (record A in Figure 13.11). If P matches the location of A, then the search is successful. In this example the positions do not match (A 's location $(40, 45)$ is not the same as $(69, 50)$), so the search must continue. The x value of A is compared with that of P to determine in which direction to branch. Because A_x 's value of 40 is less than P 's x value of 69, we branch to the right subtree (all cities with x value greater than or equal to 40 are in the right subtree). A_y does not affect the decision on which way to branch at this level. At the second level, P does not match record C's position, so another branch must be taken. However, at this level we branch based on the relative y values of point P and record C (because $1 \bmod 2 = 1$, which corresponds to the y -coordinate). Because C_y 's value of 10 is less than P_y 's value of 50, we branch to the right. At this point, P is compared against the position of D. A match is made and the search is successful.

If the search process reaches a **NULL** pointer, then that point is not contained in the tree. Here is a k-d tree search implementation, equivalent to the **findhelp** function of the BST class. **KD** class private member **D** stores the key's dimension.

```

// Find the record with the given coordinates
bool findhelp(BinNode<E>* root, int* coord,
              E& e, int discrim) const {
    // Member "coord" of a node is an integer array storing
    // the node's coordinates.
    if (root == NULL) return false;    // Empty tree
    int* currcoord = (root->val())->coord();
    if (EqualCoord(currcoord, coord)) { // Found it
        e = root->val();
        return true;
    }
    if (currcoord[discrim] < coord[discrim])
        return findhelp(root->left(), coord, e, (discrim+1)%D);
    else
        return findhelp(root->right(), coord, e, (discrim+1)%D);
}

```

Inserting a new node into the k-d tree is similar to BST insertion. The k-d tree search procedure is followed until a **NULL** pointer is found, indicating the proper place to insert the new node.

Example 13.6 Inserting a record at location (10, 50) in the k-d tree of Figure 13.11 first requires a search to the node containing record *B*. At this point, the new record is inserted into *B*'s left subtree.

Deleting a node from a k-d tree is similar to deleting from a BST, but slightly harder. As with deleting from a BST, the first step is to find the node (call it *N*) to be deleted. It is then necessary to find a descendant of *N* which can be used to replace *N* in the tree. If *N* has no children, then *N* is replaced with a **NULL** pointer. Note that if *N* has one child that in turn has children, we cannot simply assign *N*'s parent to point to *N*'s child as would be done in the BST. To do so would change the level of all nodes in the subtree, and thus the discriminator used for a search would also change. The result is that the subtree would no longer be a k-d tree because a node's children might now violate the BST property for that discriminator.

Similar to BST deletion, the record stored in *N* should be replaced either by the record in *N*'s right subtree with the least value of *N*'s discriminator, or by the record in *N*'s left subtree with the greatest value for this discriminator. Assume that *N* was at an odd level and therefore *y* is the discriminator. *N* could then be replaced by the record in its right subtree with the least *y* value (call it Y_{\min}). The problem is that Y_{\min} is not necessarily the leftmost node, as it would be in the BST. A modified search procedure to find the least *y* value in the left subtree must be used to find it instead. The implementation for **findmin** is shown in Figure 13.12. A recursive call to the delete routine will then remove Y_{\min} from the tree. Finally, Y_{\min} 's record is substituted for the record in node *N*.

Note that we can replace the node to be deleted with the least-valued node from the right subtree only if the right subtree exists. If it does not, then a suitable

```

// Return a pointer to the node with the least value in root
// for the selected discriminator
BinNode<E>* findmin(BinNode<E>* root,
                    int discrim, int currdis) const {
    // discrim: discriminator key used for minimum search;
    // currdis: current level (mod D);
    if (root == NULL) return NULL;
    BinNode<E> *minnode = findmin(root->left(), discrim,
                                   (currdis+1)%D);
    if (discrim != currdis) { // If not at discrim's level,
                             // we must search both subtrees
        BinNode<E> *rightmin =
            findmin(root->right(), discrim, (currdis+1)%D);
        // Check if right side has smaller key value
        minnode = min(minnode, rightmin, discrim);
    } // Now, minnode has the smallest value in children
    return min(minnode, root, discrim);
}

```

Figure 13.12 The k-d tree **findmin** method. On levels using the minimum value's discriminator, branching is to the left. On other levels, both children's subtrees must be visited. Helper function **min** takes two nodes and a discriminator as input, and returns the node with the smaller value in that discriminator.

replacement must be found in the left subtree. Unfortunately, it is not satisfactory to replace N 's record with the record having the greatest value for the discriminator in the left subtree, because this new value might be duplicated. If so, then we would have equal values for the discriminator in N 's left subtree, which violates the ordering rules for the k-d tree. Fortunately, there is a simple solution to the problem. We first move the left subtree of node N to become the right subtree (i.e., we simply swap the values of N 's left and right child pointers). At this point, we proceed with the normal deletion process, replacing the record of N to be deleted with the record containing the *least* value of the discriminator from what is now N 's right subtree.

Assume that we want to print out a list of all records that are within a certain distance d of a given point P . We will use Euclidean distance, that is, point P is defined to be within distance d of point N if¹

$$\sqrt{(P_x - N_x)^2 + (P_y - N_y)^2} \leq d.$$

If the search process reaches a node whose key value for the discriminator is more than d above the corresponding value in the search key, then it is not possible that any record in the right subtree can be within distance d of the search key because all key values in that dimension are always too great. Similarly, if the current node's key value in the discriminator is d less than that for the search key value,

¹A more efficient computation is $(P_x - N_x)^2 + (P_y - N_y)^2 \leq d^2$. This avoids performing a square root function.

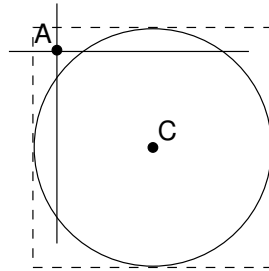


Figure 13.13 Function **InCircle** must check the Euclidean distance between a record and the query point. It is possible for a record *A* to have *x*- and *y*-coordinates each within the query distance of the query point *C*, yet have *A* itself lie outside the query circle.

then no record in the left subtree can be within the radius. In such cases, the subtree in question need not be searched, potentially saving much time. In the average case, the number of nodes that must be visited during a range query is linear on the number of data records that fall within the query circle.

Example 13.7 We will now find all cities in the *k*-d tree of Figure 13.14 within 25 units of the point (25, 65). The search begins with the root node, which contains record *A*. Because (40, 45) is exactly 25 units from the search point, it will be reported. The search procedure then determines which branches of the tree to take. The search circle extends to both the left and the right of *A*'s (vertical) dividing line, so both branches of the tree must be searched. The left subtree is processed first. Here, record *B* is checked and found to fall within the search circle. Because the node storing *B* has no children, processing of the left subtree is complete. Processing of *A*'s right subtree now begins. The coordinates of record *C* are checked and found not to fall within the circle. Thus, it should not be reported. However, it is possible that cities within *C*'s subtrees could fall within the search circle even if *C* does not. As *C* is at level 1, the discriminator at this level is the *y*-coordinate. Because $65 - 25 > 10$, no record in *C*'s left subtree (i.e., records above *C*) could possibly be in the search circle. Thus, *C*'s left subtree (if it had one) need not be searched. However, cities in *C*'s right subtree could fall within the circle. Thus, search proceeds to the node containing record *D*. Again, *D* is outside the search circle. Because $25 + 25 < 69$, no record in *D*'s right subtree could be within the search circle. Thus, only *D*'s left subtree need be searched. This leads to comparing record *E*'s coordinates against the search circle. Record *E* falls outside the search circle, and processing is complete. So we see that we only search subtrees whose rectangles fall within the search circle.

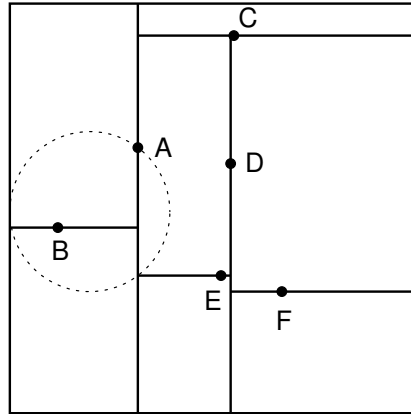


Figure 13.14 Searching in the k-d tree of Figure 13.11. (a) The k-d tree decomposition for a 128×128 -unit region containing seven data points. (b) The k-d tree for the region of (a).

```
// Print all points within distance "rad" of "coord"
void regionhelp(BinNode<E>* root, int* coord,
               int rad, int discrim) const {
    if (root == NULL) return;          // Empty tree
    // Check if record at root is in circle
    if (InCircle((root->val())->coord(), coord, rad))
        cout << root->val() << endl; // Do what is appropriate
    int* currcoord = (root->val())->coord();
    if (currcoord[discrim] > (coord[discrim] - rad))
        regionhelp(root->left(), coord, rad, (discrim+1)%D);
    if (currcoord[discrim] < (coord[discrim] + rad))
        regionhelp(root->right(), coord, rad, (discrim+1)%D);
}
```

Figure 13.15 The k-d tree region search method.

Figure 13.15 shows an implementation for the region search method. When a node is visited, function **InCircle** is used to check the Euclidean distance between the node's record and the query point. It is not enough to simply check that the differences between the x - and y -coordinates are each less than the query distances because the record could still be outside the search circle, as illustrated by Figure 13.13.

13.3.2 The PR quadtree

In the Point-Region Quadtree (hereafter referred to as the PR quadtree) each node either has exactly four children or is a leaf. That is, the PR quadtree is a full four-way branching (4-ary) tree in shape. The PR quadtree represents a collection of data points in two dimensions by decomposing the region containing the data points into four equal quadrants, subquadrants, and so on, until no leaf node contains more

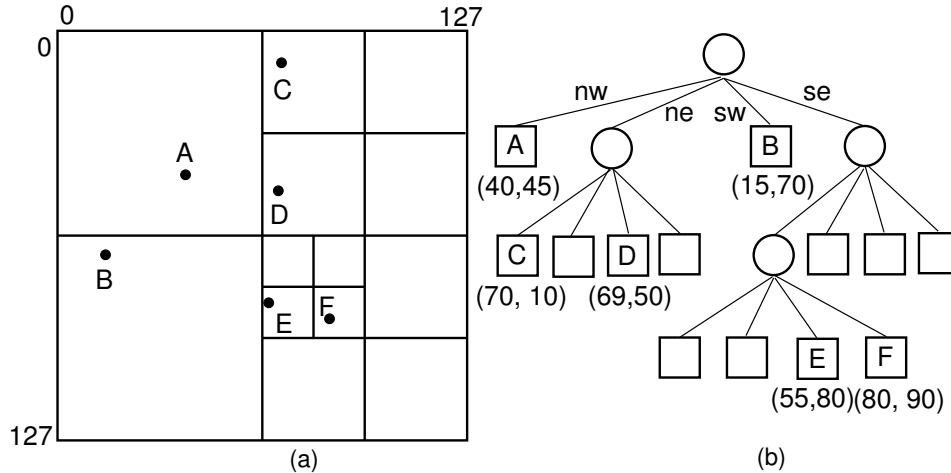


Figure 13.16 Example of a PR quadtree. (a) A map of data points. We define the region to be square with origin at the upper-left-hand corner and sides of length 128. (b) The PR quadtree for the points in (a). (a) also shows the block decomposition imposed by the PR quadtree for this region.

than a single point. In other words, if a region contains zero or one data points, then it is represented by a PR quadtree consisting of a single leaf node. If the region contains more than a single data point, then the region is split into four equal quadrants. The corresponding PR quadtree then contains an internal node and four subtrees, each subtree representing a single quadrant of the region, which might in turn be split into subquadrants. Each internal node of a PR quadtree represents a single split of the two-dimensional region. The four quadrants of the region (or equivalently, the corresponding subtrees) are designated (in order) NW, NE, SW, and SE. Each quadrant containing more than a single point would in turn be recursively divided into subquadrants until each leaf of the corresponding PR quadtree contains at most one point.

For example, consider the region of Figure 13.16(a) and the corresponding PR quadtree in Figure 13.16(b). The decomposition process demands a fixed key range. In this example, the region is assumed to be of size 128×128 . Note that the internal nodes of the PR quadtree are used solely to indicate decomposition of the region; internal nodes do not store data records. Because the decomposition lines are predetermined (i.e., key-space decomposition is used), the PR quadtree is a trie.

Search for a record matching point Q in the PR quadtree is straightforward. Beginning at the root, we continuously branch to the quadrant that contains Q until our search reaches a leaf node. If the root is a leaf, then just check to see if the node's data record matches point Q . If the root is an internal node, proceed to the child that contains the search coordinate. For example, the NW quadrant of Figure 13.16 contains points whose x and y values each fall in the range 0 to 63.

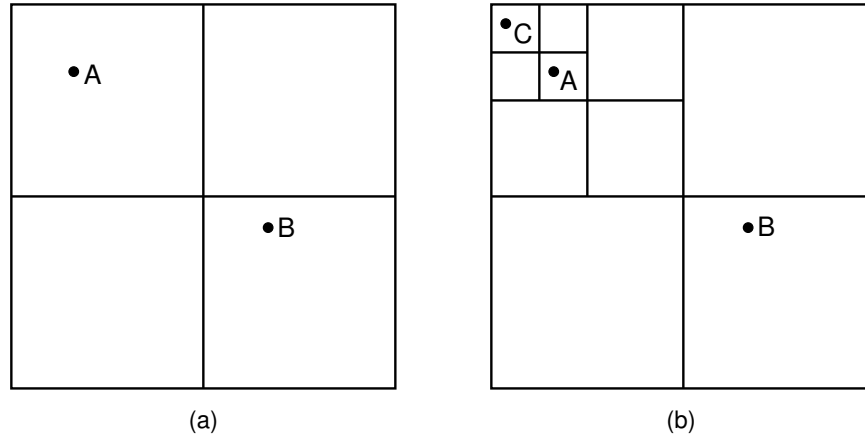


Figure 13.17 PR quadtree insertion example. (a) The initial PR quadtree containing two data points. (b) The result of inserting point *C*. The block containing *A* must be decomposed into four sub-blocks. Points *A* and *C* would still be in the same block if only one subdivision takes place, so a second decomposition is required to separate them.

The NE quadrant contains points whose x value falls in the range 64 to 127, and whose y value falls in the range 0 to 63. If the root's child is a leaf node, then that child is checked to see if Q has been found. If the child is another internal node, the search process continues through the tree until a leaf node is found. If this leaf node stores a record whose position matches Q then the query is successful; otherwise Q is not in the tree.

Inserting record P into the PR quadtree is performed by first locating the leaf node that contains the location of P . If this leaf node is empty, then P is stored at this leaf. If the leaf already contains P (or a record with P 's coordinates), then a duplicate record should be reported. If the leaf node already contains another record, then the node must be repeatedly decomposed until the existing record and P fall into different leaf nodes. Figure 13.17 shows an example of such an insertion.

Deleting a record P is performed by first locating the node N of the PR quadtree that contains P . Node N is then changed to be empty. The next step is to look at N 's three siblings. N and its siblings must be merged together to form a single node N' if only one point is contained among them. This merging process continues until some level is reached at which at least two points are contained in the subtrees represented by node N' and its siblings. For example, if point C is to be deleted from the PR quadtree representing Figure 13.17(b), the resulting node must be merged with its siblings, and that larger node again merged with its siblings to restore the PR quadtree to the decomposition of Figure 13.17(a).

Region search is easily performed with the PR quadtree. To locate all points within radius r of query point Q , begin at the root. If the root is an empty leaf node,

then no data points are found. If the root is a leaf containing a data record, then the location of the data point is examined to determine if it falls within the circle. If the root is an internal node, then the process is performed recursively, but *only* on those subtrees containing some part of the search circle.

Let us now consider how the structure of the PR quadtree affects the design of its node representation. The PR quadtree is actually a trie (as defined in Section 13.1). Decomposition takes place at the mid-points for internal nodes, regardless of where the data points actually fall. The placement of the data points does determine *whether* a decomposition for a node takes place, but not *where* the decomposition for the node takes place. Internal nodes of the PR quadtree are quite different from leaf nodes, in that internal nodes have children (leaf nodes do not) and leaf nodes have data fields (internal nodes do not). Thus, it is likely to be beneficial to represent internal nodes differently from leaf nodes. Finally, there is the fact that approximately half of the leaf nodes will contain no data field.

Another issue to consider is: How does a routine traversing the PR quadtree get the coordinates for the square represented by the current PR quadtree node? One possibility is to store with each node its spatial description (such as upper-left corner and width). However, this will take a lot of space — perhaps as much as the space needed for the data records, depending on what information is being stored.

Another possibility is to pass in the coordinates when the recursive call is made. For example, consider the search process. Initially, the search visits the root node of the tree, which has origin at (0, 0), and whose width is the full size of the space being covered. When the appropriate child is visited, it is a simple matter for the search routine to determine the origin for the child, and the width of the square is simply half that of the parent. Not only does passing in the size and position information for a node save considerable space, but avoiding storing such information in the nodes enables a good design choice for empty leaf nodes, as discussed next.

How should we represent empty leaf nodes? On average, half of the leaf nodes in a PR quadtree are empty (i.e., do not store a data point). One implementation option is to use a **NULL** pointer in internal nodes to represent empty nodes. This will solve the problem of excessive space requirements. There is an unfortunate side effect that using a **NULL** pointer requires the PR quadtree processing methods to understand this convention. In other words, you are breaking encapsulation on the node representation because the tree now must know things about how the nodes are implemented. This is not too horrible for this particular application, because the node class can be considered private to the tree class, in which case the node implementation is completely invisible to the outside world. However, it is undesirable if there is another reasonable alternative.

Fortunately, there is a good alternative. It is called the Flyweight design pattern. In the PR quadtree, a flyweight is a single empty leaf node that is reused in all places where an empty leaf node is needed. You simply have *all* of the internal nodes with

empty leaf children point to the same node object. This node object is created once at the beginning of the program, and is never removed. The node class recognizes from the pointer value that the flyweight is being accessed, and acts accordingly.

Note that when using the Flyweight design pattern, you *cannot* store coordinates for the node in the node. This is an example of the concept of intrinsic versus extrinsic state. Intrinsic state for an object is state information stored in the object. If you stored the coordinates for a node in the node object, those coordinates would be intrinsic state. Extrinsic state is state information about an object stored elsewhere in the environment, such as in global variables or passed to the method. If your recursive calls that process the tree pass in the coordinates for the current node, then the coordinates will be extrinsic state. A flyweight can have in its intrinsic state *only* information that is accurate for *all* instances of the flyweight. Clearly coordinates do not qualify, because each empty leaf node has its own location. So, if you want to use a flyweight, you must pass in coordinates.

Another design choice is: Who controls the work, the node class or the tree class? For example, on an insert operation, you could have the tree class control the flow down the tree, looking at (querying) the nodes to see their type and reacting accordingly. This is the approach used by the BST implementation in Section 5.4. An alternate approach is to have the node class do the work. That is, you have an insert method for the nodes. If the node is internal, it passes the city record to the appropriate child (recursively). If the node is a flyweight, it replaces itself with a new leaf node. If the node is a full node, it replaces itself with a subtree. This is an example of the Composite design pattern, discussed in Section 5.3.1. Use of the composite design would be difficult if **NULL** pointers are used to represent empty leaf nodes. It turns out that the PR quadtree insert and delete methods are easier to implement when using the composite design.

13.3.3 Other Point Data Structures

The differences between the k-d tree and the PR quadtree illustrate many of the design choices encountered when creating spatial data structures. The k-d tree provides an object space decomposition of the region, while the PR quadtree provides a key space decomposition (thus, it is a trie). The k-d tree stores records at all nodes, while the PR quadtree stores records only at the leaf nodes. Finally, the two trees have different structures. The k-d tree is a binary tree (and need not be full), while the PR quadtree is a full tree with 2^d branches (in the two-dimensional case, $2^2 = 4$). Consider the extension of this concept to three dimensions. A k-d tree for three dimensions would alternate the discriminator through the x , y , and z dimensions. The three-dimensional equivalent of the PR quadtree would be a tree with 2^3 or eight branches. Such a tree is called an **octree**.

We can also devise a binary trie based on a key space decomposition in each dimension, or a quadtree that uses the two-dimensional equivalent to an object

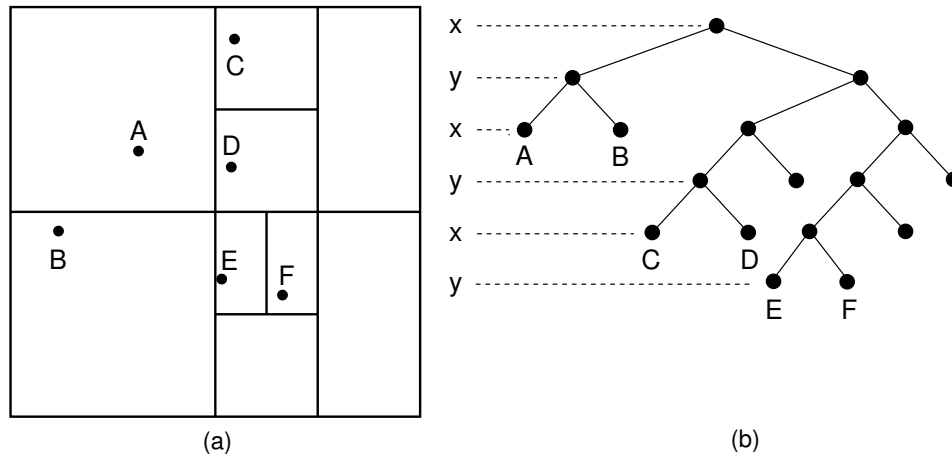


Figure 13.18 An example of the bintree, a binary tree using key space decomposition and discriminators rotating among the dimensions. Compare this with the k-d tree of Figure 13.11 and the PR quadtree of Figure 13.16.

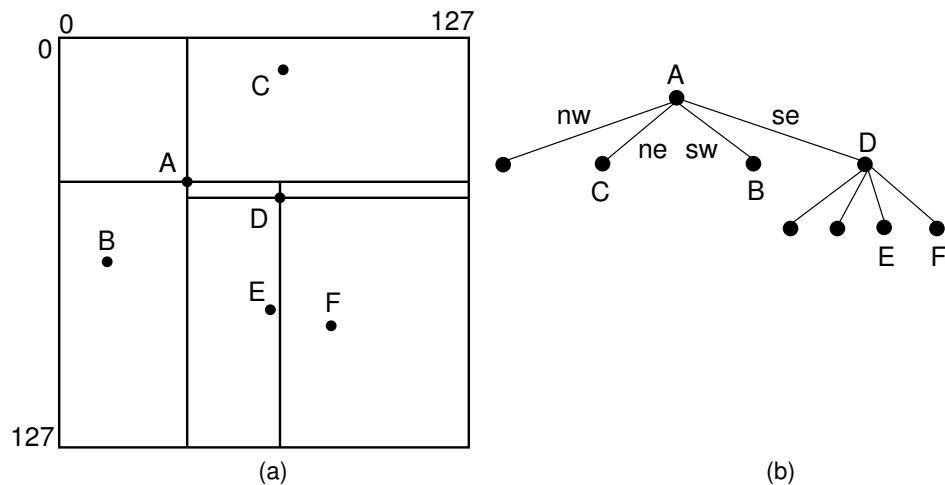


Figure 13.19 An example of the point quadtree, a 4-ary tree using object space decomposition. Compare this with the PR quadtree of Figure 13.11.

space decomposition. The **bintree** is a binary trie that uses keyspace decomposition and alternates discriminators at each level in a manner similar to the k-d tree. The bintree for the points of Figure 13.11 is shown in Figure 13.18. Alternatively, we can use a four-way decomposition of space centered on the data points. The tree resulting from such a decomposition is called a **point quadtree**. The point quadtree for the data points of Figure 13.11 is shown in Figure 13.19.

13.3.4 Other Spatial Data Structures

This section has barely scratched the surface of the field of spatial data structures. Dozens of distinct spatial data structures have been invented, many with variations and alternate implementations. Spatial data structures exist for storing many forms of spatial data other than points. The most important distinctions between are the tree structure (binary or not, regular decompositions or not) and the decomposition rule used to decide when the data contained within a region is so complex that the region must be subdivided.

One such spatial data structure is the Region Quadtree for storing images where the pixel values tend to be blocky, such as a map of the countries of the world. The region quadtree uses a four-way regular decomposition scheme similar to the PR quadtree. The decomposition rule is simply to divide any node containing pixels of more than one color or value.

Spatial data structures can also be used to store line object, rectangle object, or objects of arbitrary shape (such as polygons in two dimensions or polyhedra in three dimensions). A simple, yet effective, data structure for storing rectangles or arbitrary polygonal shapes can be derived from the PR quadtree. Pick a threshold value c , and subdivide any region into four quadrants if it contains more than c objects. A special case must be dealt with when more than c object intersect.

Some of the most interesting developments in spatial data structures have to do with adapting them for disk-based applications. However, all such disk-based implementations boil down to storing the spatial data structure within some variant on either B-trees or hashing.

13.4 Further Reading

PATRICIA tries and other trie implementations are discussed in *Information Retrieval: Data Structures & Algorithms*, Frakes and Baeza-Yates, eds. [FBY92].

See Knuth [Knu97] for a discussion of the AVL tree. For further reading on splay trees, see “Self-adjusting Binary Search” by Sleator and Tarjan [ST85].

The world of spatial data structures is rich and rapidly evolving. For a good introduction, see *Foundations of Multidimensional and Metric Data Structures* by Hanan Samet [Sam06]. This is also the best reference for more information on the PR quadtree. The k-d tree was invented by John Louis Bentley. For further information on the k-d tree, in addition to [Sam06], see [Ben75]. For information on using a quadtree to store arbitrary polygonal objects, see [SH92].

For a discussion on the relative space requirements for two-way versus multi-way branching, see “A Generalized Comparison of Quadtree and Bintree Storage Requirements” by Shaffer, Juvvadi, and Heath [SJH93].

Closely related to spatial data structures are data structures for storing multi-dimensional data (which might not necessarily be spatial in nature). A popular

data structure for storing such data is the R-tree, which was originally proposed by Guttman [Gut84].

13.5 Exercises

- 13.1** Show the binary trie (as illustrated by Figure 13.1) for the following collection of values: 42, 12, 100, 10, 50, 31, 7, 11, 99.
- 13.2** Show the PAT trie (as illustrated by Figure 13.3) for the following collection of values: 42, 12, 100, 10, 50, 31, 7, 11, 99.
- 13.3** Write the insertion routine for a binary trie as shown in Figure 13.1.
- 13.4** Write the deletion routine for a binary trie as shown in Figure 13.1.
- 13.5**
 - (a) Show the result (including appropriate rotations) of inserting the value 39 into the AVL tree on the left in Figure 13.4.
 - (b) Show the result (including appropriate rotations) of inserting the value 300 into the AVL tree on the left in Figure 13.4.
 - (c) Show the result (including appropriate rotations) of inserting the value 50 into the AVL tree on the left in Figure 13.4.
 - (d) Show the result (including appropriate rotations) of inserting the value 1 into the AVL tree on the left in Figure 13.4.
- 13.6** Show the splay tree that results from searching for value 75 in the splay tree of Figure 13.10(d).
- 13.7** Show the splay tree that results from searching for value 18 in the splay tree of Figure 13.10(d).
- 13.8** Some applications do not permit storing two records with duplicate key values. In such a case, an attempt to insert a duplicate-keyed record into a tree structure such as a splay tree should result in a failure on insert. What is the appropriate action to take in a splay tree implementation when the insert routine is called with a duplicate-keyed record?
- 13.9** Show the result of deleting point A from the k-d tree of Figure 13.11.
- 13.10**
 - (a) Show the result of building a k-d tree from the following points (inserted in the order given). A (20, 20), B (10, 30), C (25, 50), D (35, 25), E (30, 45), F (30, 35), G (55, 40), H (45, 35), I (50, 30).
 - (b) Show the result of deleting point A from the tree you built in part (a).
- 13.11**
 - (a) Show the result of deleting F from the PR quadtree of Figure 13.16.
 - (b) Show the result of deleting records E and F from the PR quadtree of Figure 13.16.
- 13.12**
 - (a) Show the result of building a PR quadtree from the following points (inserted in the order given). Assume the tree is representing a space of 64 by 64 units. A (20, 20), B (10, 30), C (25, 50), D (35, 25), E (30, 45), F (30, 35), G (45, 25), H (45, 30), I (50, 30).
 - (b) Show the result of deleting point C from the tree you built in part (a).

- (c) Show the result of deleting point F from the resulting tree in part (b).
- 13.13** On average, how many leaf nodes of a PR quadtree will typically be empty? Explain why.
- 13.14** When performing a region search on a PR quadtree, we need only search those subtrees of an internal node whose corresponding square falls within the query circle. This is most easily computed by comparing the x and y ranges of the query circle against the x and y ranges of the square corresponding to the subtree. However, as illustrated by Figure 13.13, the x and y ranges might overlap without the circle actually intersecting the square. Write a function that accurately determines if a circle and a square intersect.
- 13.15** (a) Show the result of building a bintree from the following points (inserted in the order given). Assume the tree is representing a space of 64 by 64 units. A (20, 20), B (10, 30), C (25, 50), D (35, 25), E (30, 45), F (30, 35), G (45, 25), H (45, 30), I (50, 30).
 (b) Show the result of deleting point C from the tree you built in part (a).
 (c) Show the result of deleting point F from the resulting tree in part (b).
- 13.16** Compare the trees constructed for Exercises 12 and 15 in terms of the number of internal nodes, full leaf nodes, empty leaf nodes, and total depths of the two trees.
- 13.17** Show the result of building a point quadtree from the following points (inserted in the order given). Assume the tree is representing a space of 64 by 64 units. A (20, 20), B (10, 30), C (25, 50), D (35, 25), E (30, 45), F (31, 35), G (45, 26), H (44, 30), I (50, 30).

13.6 Projects

- 13.1** Use the trie data structure to devise a program to sort variable-length strings. The program's running time should be proportional to the total number of letters in all of the strings. Note that some strings might be very long while most are short.
- 13.2** Define the set of **suffix strings** for a string S to be S , S without its first character, S without its first two characters, and so on. For example, the complete set of suffix strings for "HELLO" would be

$$\{\text{HELLO}, \text{ELLO}, \text{LLO}, \text{LO}, \text{O}\}.$$

A **suffix tree** is a PAT trie that contains all of the suffix strings for a given string, and associates each suffix with the complete string. The advantage of a suffix tree is that it allows a search for strings using "wildcards." For example, the search key "TH*" means to find all strings with "TH" as the first two characters. This can easily be done with a regular trie. Searching for "*TH" is not efficient in a regular trie, but it is efficient in a suffix tree.

Implement the suffix tree for a dictionary of words or phrases, with support for wildcard search.

- 13.3** Revise the BST class of Section 5.4 to use the AVL tree rotations. Your new implementation should not modify the original BST class ADT. Compare your AVL tree against an implementation of the standard BST over a wide variety of input data. Under what conditions does the splay tree actually save time?
- 13.4** Revise the BST class of Section 5.4 to use the splay tree rotations. Your new implementation should not modify the original BST class ADT. Compare your splay tree against an implementation of the standard BST over a wide variety of input data. Under what conditions does the splay tree actually save time?
- 13.5** Implement a city database using the k-d tree. Each database record contains the name of the city (a string of arbitrary length) and the coordinates of the city expressed as integer x - and y -coordinates. Your database should allow records to be inserted, deleted by name or coordinate, and searched by name or coordinate. You should also support region queries, that is, a request to print all records within a given distance of a specified point.
- 13.6** Implement a city database using the PR quadtree. Each database record contains the name of the city (a string of arbitrary length) and the coordinates of the city expressed as integer x - and y -coordinates. Your database should allow records to be inserted, deleted by name or coordinate, and searched by name or coordinate. You should also support region queries, that is, a request to print all records within a given distance of a specified point.
- 13.7** Implement and test the PR quadtree, using the composite design to implement the insert, search, and delete operations.
- 13.8** Implement a city database using the bintree. Each database record contains the name of the city (a string of arbitrary length) and the coordinates of the city expressed as integer x - and y -coordinates. Your database should allow records to be inserted, deleted by name or coordinate, and searched by name or coordinate. You should also support region queries, that is, a request to print all records within a given distance of a specified point.
- 13.9** Implement a city database using the point quadtree. Each database record contains the name of the city (a string of arbitrary length) and the coordinates of the city expressed as integer x - and y -coordinates. Your database should allow records to be inserted, deleted by name or coordinate, and searched by name or coordinate. You should also support region queries, that is, a request to print all records within a given distance of a specified point.
- 13.10** Use the PR quadtree to implement an efficient solution to Problem 6.5. That is, store the set of points in a PR quadtree. For each point, the PR quadtree is used to find those points within distance D that should be equivalenced. What is the asymptotic complexity of this solution?

- 13.11** Select any two of the point representations described in this chapter (i.e., the k-d tree, the PR quadtree, the bintree, and the point quadtree). Implement your two choices and compare them over a wide range of data sets. Describe which is easier to implement, which appears to be more space efficient, and which appears to be more time efficient.
- 13.12** Implement a representation for a collection of (two dimensional) rectangles using a quadtree based on regular decomposition. Assume that the space being represented is a square whose width and height are some power of two. Rectangles are assumed to have integer coordinates and integer width and height. Pick some value c , and use as a decomposition rule that a region is subdivided into four equal-sized regions whenever it contains more than c rectangles. A special case occurs if all of these rectangles intersect at some point within the current region (because decomposing such a node would never reach termination). In this situation, the node simply stores pointers to more than c rectangles. Try your representation on data sets of rectangles with varying values of c .

PART V

Theory of Algorithms

Analysis Techniques

Often it is easy to invent an equation to model the behavior of an algorithm or data structure. Often it is easy to derive a closed-form solution for the equation should it contain a recurrence or summation. But sometimes analysis proves more difficult. It may take a clever insight to derive the right model, such as the snowplow argument for analyzing the average run length resulting from Replacement Selection (Section 8.5.2). In this example, once the snowplow argument is understood, the resulting equations follow naturally. Sometimes, developing the model is straightforward but analyzing the resulting equations is not. An example is the average-case analysis for Quicksort. The equation given in Section 7.5 simply enumerates all possible cases for the pivot position, summing corresponding costs for the recursive calls to Quicksort. However, deriving a closed-form solution for the resulting recurrence relation is not as easy.

Many analyses of iterative algorithms use a summation to model the cost of a loop. Techniques for finding closed-form solutions to summations are presented in Section 14.1. The cost for many algorithms based on recursion are best modeled by recurrence relations. A discussion of techniques for solving recurrences is provided in Section 14.2. These sections build on the introduction to summations and recurrences provided in Section 2.4, so the reader should already be familiar with that material.

Section 14.3 provides an introduction to the topic of **amortized analysis**. Amortized analysis deals with the cost of a series of operations. Perhaps a single operation in the series has high cost, but as a result the cost of the remaining operations is limited. Amortized analysis has been used successfully to analyze several of the algorithms presented in previous sections, including the cost of a series of UNION/FIND operations (Section 6.2), the cost of partition in Quicksort (Section 7.5), the cost of a series of splay tree operations (Section 13.2), and the cost of a series of operations on self-organizing lists (Section 9.2). Section 14.3 discusses the topic in more detail.

14.1 Summation Techniques

Consider the following simple summation.

$$\sum_{i=1}^n i.$$

In Section 2.6.3 it was proved by induction that this summation has the well-known closed form $n(n+1)/2$. But while induction is a good technique for proving that a proposed closed-form expression is correct, how do we find a candidate closed-form expression to test in the first place? Let us try to think through this problem from first principles, as though we had never seen it before.

A good place to begin analyzing a summation is to give an estimate of its value for a given n . Observe that the biggest term for this summation is n , and there are n terms being summed up. So the total must be less than n^2 . Actually, most terms are much less than n , and the sizes of the terms grows linearly. If we were to draw a picture with bars for the size of the terms, their heights would form a line, and we could enclose them in a box n units wide and n units high. It is easy to see from this that a closer estimate for the summation is about $(n^2)/2$. Having this estimate in hand helps us when trying to determine an exact closed-form solution, because we will hopefully recognize if our proposed solution is badly wrong.

Let us now consider some ways that we might hit upon an exact equation for the closed form solution to this summation. One particularly clever approach we can take is to observe that we can “pair up” the first and last terms, the second and $(n-1)$ th terms, and so on. Each pair sums to $n+1$. The number of pairs is $n/2$. Thus, the solution is $n(n+1)/2$. This is pretty, and there is no doubt about it being correct. The problem is that it is not a useful technique for solving many other summations.

Now let us try to do something a bit more general. We already recognized that, because the largest term is n and there are n terms, the summation is less than n^2 . If we are lucky, the closed form solution is a polynomial. Using that as a working assumption, we can invoke a technique called **guess-and-test**. We will guess that the closed-form solution for this summation is a polynomial of the form $c_1n^2 + c_2n + c_3$ for some constants c_1 , c_2 , and c_3 . If this is true, then we can plug in the answers to small cases of the summation to solve for the coefficients. For this example, substituting 0, 1, and 2 for n leads to three simultaneous equations. Because the summation when $n = 0$ is just 0, c_3 must be 0. For $n = 1$ and $n = 2$ we get the two equations

$$\begin{aligned} c_1 + c_2 &= 1 \\ 4c_1 + 2c_2 &= 3, \end{aligned}$$

which in turn yield $c_1 = 1/2$ and $c_2 = 1/2$. Thus, if the closed-form solution for the summation is a polynomial, it can only be

$$1/2n^2 + 1/2n + 0$$

which is more commonly written

$$\frac{n(n+1)}{2}.$$

At this point, we still must do the “test” part of the guess-and-test approach. We can use an induction proof to verify whether our candidate closed-form solution is correct. In this case it is indeed correct, as shown by Example 2.11. The induction proof is necessary because our initial assumption that the solution is a simple polynomial could be wrong. For example, it might have been that the true solution includes a logarithmic term, such as $c_1n^2 + c_2n \log n$. The process shown here is essentially fitting a curve to a fixed number of points. Because there is always an n -degree polynomial that fits $n + 1$ points, we have not done enough work to be sure that we know the true equation without the induction proof.

Guess-and-test is useful whenever the solution is a polynomial expression. In particular, similar reasoning can be used to solve for $\sum_{i=1}^n i^2$, or more generally $\sum_{i=1}^n i^c$ for c any positive integer. Why is this not a universal approach to solving summations? Because many summations do not have a polynomial as their closed form solution.

A more general approach is based on the **subtract-and-guess** or **divide-and-guess** strategies. One form of subtract-and-guess is known as the **shifting method**. The shifting method subtracts the summation from a variation on the summation. The variation selected for the subtraction should be one that makes most of the terms cancel out. To solve sum f , we pick a known function g and find a pattern in terms of $f(n) - g(n)$ or $f(n)/g(n)$.

Example 14.1 Find the closed form solution for $\sum_{i=1}^n i$ using the divide-and-guess approach. We will try two example functions to illustrate the divide-and-guess method: dividing by n and dividing by $f(n-1)$. Our goal is to find patterns that we can use to guess a closed-form expression as our candidate for testing with an induction proof. To aid us in finding such patterns, we can construct a table showing the first few numbers of each function, and the result of dividing one by the other, as follows.

n	1	2	3	4	5	6	7	8	9	10
$f(n)$	1	3	6	10	15	21	28	36	46	57
n	1	2	3	4	5	6	7	8	9	10
$f(n)/n$	2/2	3/2	4/2	5/2	6/2	7/2	8/2	9/2	10/2	11/2
$f(n-1)$	0	1	3	6	10	15	21	28	36	46
$f(n)/f(n-1)$		3/1	4/2	5/3	6/4	7/5	8/6	9/7	10/8	11/9

Dividing by both n and $f(n-1)$ happen to give us useful patterns to work with. $\frac{f(n)}{n} = \frac{n+1}{2}$, and $\frac{f(n)}{f(n-1)} = \frac{n+1}{n-1}$. Of course, lots of other guesses for function g do not work. For example, $f(n) - n = f(n-1)$. Knowing that $f(n) = f(n-1) + n$ is not useful for determining the closed form solution to this summation. Or consider $f(n) - f(n-1) = n$. Again, knowing that $f(n) = f(n-1) + n$ is not useful. Finding the right combination of equations can be like finding a needle in a haystack.

In our first example, we can see directly what the closed-form solution should be. Since $\frac{f(n)}{n} = \frac{n+1}{2}$, obviously $f(n) = n(n+1)/2$.

Dividing $f(n)$ by $f(n-1)$ does not give so obvious a result, but it provides another useful illustration.

$$\begin{aligned}\frac{f(n)}{f(n-1)} &= \frac{n+1}{n-1} \\ f(n)(n-1) &= (n+1)f(n-1) \\ f(n)(n-1) &= (n+1)(f(n)-n) \\ nf(n) - f(n) &= nf(n) + f(n) - n^2 - n \\ 2f(n) &= n^2 + n = n(n+1) \\ f(n) &= \frac{n(n+1)}{2}\end{aligned}$$

Once again, we still do not have a proof that $f(n) = n(n+1)/2$. Why? Because we did not prove that $f(n)/n = (n+1)/2$ nor that $f(n)/f(n-1) = (n+1)/(n-1)$. We merely hypothesized patterns from looking at a few terms. Fortunately, it is easy to check our hypothesis with induction.

Example 14.2 Solve the summation

$$\sum_{i=1}^n 1/2^i.$$

We will begin by writing out a table listing the first few values of the summation, to see if we can detect a pattern.

n	1	2	3	4	5	6
$f(n)$	$\frac{1}{2}$	$\frac{3}{4}$	$\frac{7}{8}$	$\frac{15}{16}$	$\frac{31}{32}$	$\frac{63}{64}$
$1 - f(n)$	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	$\frac{1}{32}$	$\frac{1}{64}$

By direct inspection of the second line of the table, we might recognize the pattern $f(n) = \frac{2^n - 1}{2^n}$. A simple induction proof can then prove that this always holds true. Alternatively, consider if we hadn't noticed the pattern for the form of $f(n)$. We might observe that $f(n)$ appears to be reaching an asymptote at one. In which case, we might consider looking at the difference between $f(n)$ and the expected asymptote. This result is shown in the last line of the table, which has a clear pattern since the i th entry is of $1/2^i$. From this we can easily deduce a guess that $f(n) = 1 - \frac{1}{2^n}$. Again, a simple induction proof will verify the guess.

Example 14.3 Solve the summation

$$f(n) = \sum_{i=0}^n ar^i = a + ar + ar^2 + \cdots + ar^n.$$

This is called a geometric series. Our goal is to find some function $g(n)$ such that the difference between $f(n)$ and $g(n)$ one from the other leaves us with an easily manipulated equation. Because the difference between consecutive terms of the summation is a factor of r , we can shift terms if we multiply the entire expression by r :

$$rf(n) = r \sum_{i=0}^n ar^i = ar + ar^2 + ar^3 + \cdots + ar^{n+1}.$$

We can now subtract the one equation from the other, as follows:

$$\begin{aligned} f(n) - rf(n) &= a + ar + ar^2 + ar^3 + \cdots + ar^n \\ &\quad - (ar + ar^2 + ar^3 + \cdots + ar^n) - ar^{n+1}. \end{aligned}$$

The result leaves only the end terms:

$$\begin{aligned} f(n) - rf(n) &= \sum_{i=0}^n ar^i - r \sum_{i=0}^n ar^i. \\ (1 - r)f(n) &= a - ar^{n+1}. \end{aligned}$$

Thus, we get the result

$$f(n) = \frac{a - ar^{n+1}}{1 - r}$$

where $r \neq 1$.

Example 14.4 For our second example of the shifting method, we solve

$$f(n) = \sum_{i=1}^n i2^i = 1 \cdot 2^1 + 2 \cdot 2^2 + 3 \cdot 2^3 + \cdots + n \cdot 2^n.$$

We can achieve our goal if we multiply by two:

$$2f(n) = 2 \sum_{i=1}^n i2^i = 1 \cdot 2^2 + 2 \cdot 2^3 + 3 \cdot 2^4 + \cdots + (n-1) \cdot 2^n + n \cdot 2^{n+1}.$$

The i th term of $2f(n)$ is $i \cdot 2^{i+1}$, while the $(i+1)$ th term of $f(n)$ is $(i+1) \cdot 2^{i+1}$. Subtracting one expression from the other yields the summation of 2^i and a few non-canceled terms:

$$\begin{aligned} 2f(n) - f(n) &= 2 \sum_{i=1}^n i2^i - \sum_{i=1}^n i2^i \\ &= \sum_{i=1}^n i2^{i+1} - \sum_{i=1}^n i2^i. \end{aligned}$$

Shift i 's value in the second summation, substituting $(i+1)$ for i :

$$= n2^{n+1} + \sum_{i=0}^{n-1} i2^{i+1} - \sum_{i=0}^{n-1} (i+1)2^{i+1}.$$

Break the second summation into two parts:

$$= n2^{n+1} + \sum_{i=0}^{n-1} i2^{i+1} - \sum_{i=0}^{n-1} i2^{i+1} - \sum_{i=0}^{n-1} 2^{i+1}.$$

Cancel like terms:

$$= n2^{n+1} - \sum_{i=0}^{n-1} 2^{i+1}.$$

Again shift i 's value in the summation, substituting i for $(i+1)$:

$$= n2^{n+1} - \sum_{i=1}^n 2^i.$$

Replace the new summation with a solution that we already know:

$$= n2^{n+1} - (2^{n+1} - 2).$$

Finally, reorganize the equation:

$$= (n-1)2^{n+1} + 2.$$

14.2 Recurrence Relations

Recurrence relations are often used to model the cost of recursive functions. For example, the standard Mergesort (Section 7.4) takes a list of size n , splits it in half, performs Mergesort on each half, and finally merges the two sublists in n steps. The cost for this can be modeled as

$$T(n) = 2T(n/2) + n.$$

In other words, the cost of the algorithm on input of size n is two times the cost for input of size $n/2$ (due to the two recursive calls to Mergesort) plus n (the time to merge the sublists together again).

There are many approaches to solving recurrence relations, and we briefly consider three here. The first is an estimation technique: Guess the upper and lower bounds for the recurrence, use induction to prove the bounds, and tighten as required. The second approach is to expand the recurrence to convert it to a summation and then use summation techniques. The third approach is to take advantage of already proven theorems when the recurrence is of a suitable form. In particular, typical divide and conquer algorithms such as Mergesort yield recurrences of a form that fits a pattern for which we have a ready solution.

14.2.1 Estimating Upper and Lower Bounds

The first approach to solving recurrences is to guess the answer and then attempt to prove it correct. If a correct upper or lower bound estimate is given, an easy induction proof will verify this fact. If the proof is successful, then try to tighten the bound. If the induction proof fails, then loosen the bound and try again. Once the upper and lower bounds match, you are finished. This is a useful technique when you are only looking for asymptotic complexities. When seeking a precise closed-form solution (i.e., you seek the constants for the expression), this method will probably be too much work.

Example 14.5 Use the guessing technique to find the asymptotic bounds for Mergesort, whose running time is described by the equation

$$T(n) = 2T(n/2) + n; \quad T(2) = 1.$$

We begin by guessing that this recurrence has an upper bound in $O(n^2)$. To be more precise, assume that

$$T(n) \leq n^2.$$

We prove this guess is correct by induction. In this proof, we assume that n is a power of two, to make the calculations easy. For the base case,

$\mathbf{T}(2) = 1 \leq 2^2$. For the induction step, we need to show that $\mathbf{T}(n) \leq n^2$ implies that $\mathbf{T}(2n) \leq (2n)^2$ for $n = 2^N$, $N \geq 1$. The induction hypothesis is

$$\mathbf{T}(i) \leq i^2, \text{ for all } i \leq n.$$

It follows that

$$\mathbf{T}(2n) = 2\mathbf{T}(n) + 2n \leq 2n^2 + 2n \leq 4n^2 \leq (2n)^2$$

which is what we wanted to prove. Thus, $\mathbf{T}(n)$ is in $O(n^2)$.

Is $O(n^2)$ a good estimate? In the next-to-last step we went from $n^2 + 2n$ to the much larger $4n^2$. This suggests that $O(n^2)$ is a high estimate. If we guess something smaller, such as $\mathbf{T}(n) \leq cn$ for some constant c , it should be clear that this cannot work because $c2n = 2cn$ and there is no room for the extra n cost to join the two pieces together. Thus, the true cost must be somewhere between cn and n^2 .

Let us now try $\mathbf{T}(n) \leq n \log n$. For the base case, the definition of the recurrence sets $\mathbf{T}(2) = 1 \leq (2 \cdot \log 2) = 2$. Assume (induction hypothesis) that $\mathbf{T}(n) \leq n \log n$. Then,

$$\mathbf{T}(2n) = 2\mathbf{T}(n) + 2n \leq 2n \log n + 2n \leq 2n(\log n + 1) \leq 2n \log 2n$$

which is what we seek to prove. In similar fashion, we can prove that $\mathbf{T}(n)$ is in $\Omega(n \log n)$. Thus, $\mathbf{T}(n)$ is also $\Theta(n \log n)$.

Example 14.6 We know that the factorial function grows exponentially. How does it compare to 2^n ? To n^n ? Do they all grow “equally fast” (in an asymptotic sense)? We can begin by looking at a few initial terms.

n	1	2	3	4	5	6	7	8	9
$n!$	1	2	6	24	120	720	5040	40320	362880
2^n	2	4	8	16	32	64	128	256	512
n^n	1	4	9	256	3125	46656	823543	16777216	387420489

We can also look at these functions in terms of their recurrences.

$$n! = \begin{cases} 1 & n = 1 \\ n(n-1)! & n > 1 \end{cases}$$

$$2^n = \begin{cases} 2 & n = 1 \\ 2(2^{n-1}) & n > 1 \end{cases}$$

$$n^n = \begin{cases} n & n = 1 \\ n(n^{n-1}) & n > 1 \end{cases}$$

At this point, our intuition should be telling us pretty clearly the relative growth rates of these three functions. But how do we prove formally which grows the fastest? And how do we decide if the differences are significant in an asymptotic sense, or just constant factor differences?

We can use logarithms to help us get an idea about the relative growth rates of these functions. Clearly, $\log 2^n = n$. Equally clearly, $\log n^n = n \log n$. We can easily see from this that 2^n is $o(n^n)$, that is, n^n grows asymptotically faster than 2^n .

How does $n!$ fit into this? We can again take advantage of logarithms. Obviously $n! \leq n^n$, so we know that $\log n!$ is $O(n \log n)$. But what about a lower bound for the factorial function? Consider the following.

$$\begin{aligned} n! &= n \times (n-1) \times \cdots \times \frac{n}{2} \times \left(\frac{n}{2} - 1\right) \times \cdots \times 2 \times 1 \\ &\geq \frac{n}{2} \times \frac{n}{2} \times \cdots \times \frac{n}{2} \times 1 \times \cdots \times 1 \times 1 \\ &= \left(\frac{n}{2}\right)^{n/2} \end{aligned}$$

Therefore

$$\log n! \geq \log \left(\frac{n}{2}\right)^{n/2} = \left(\frac{n}{2}\right) \log \left(\frac{n}{2}\right).$$

In other words, $\log n!$ is in $\Omega(n \log n)$. Thus, $\log n! = \Theta(n \log n)$.

Note that this does **not** mean that $n! = \Theta(n^n)$. Because $\log n^2 = 2 \log n$, it follows that $\log n = \Theta(\log n^2)$ but $n \neq \Theta(n^2)$. The log function often works as a “flattener” when dealing with asymptotics. That is, whenever $\log f(n)$ is in $O(\log g(n))$ we know that $f(n)$ is in $O(g(n))$. But knowing that $\log f(n) = \Theta(\log g(n))$ does not necessarily mean that $f(n) = \Theta(g(n))$.

Example 14.7 What is the growth rate of the Fibonacci sequence? We define the Fibonacci sequence as $f(n) = f(n-1) + f(n-2)$ for $n \geq 2$; $f(0) = f(1) = 1$.

In this case it is useful to compare the ratio of $f(n)$ to $f(n-1)$. The following table shows the first few values.

n	1	2	3	4	5	6	7
$f(n)$	1	2	3	5	8	13	21
$f(n)/f(n-1)$	1	2	1.5	1.666	1.625	1.615	1.619

If we continue for more terms, the ratio appears to converge on a value slightly greater than 1.618. Assuming $f(n)/f(n-1)$ really does converge to a fixed value as n grows, we can determine what that value must be.

$$\frac{f(n)}{f(n-2)} = \frac{f(n-1)}{f(n-2)} + \frac{f(n-2)}{f(n-2)} \rightarrow x + 1$$

For some value x . This follows from the fact that $f(n) = f(n-1) + f(n-2)$. We divide by $f(n-2)$ to make the second term go away, and we also get something useful in the first term. Remember that the goal of such manipulations is to give us an equation that relates $f(n)$ to something without recursive calls.

For large n , we also observe that:

$$\frac{f(n)}{f(n-2)} = \frac{f(n)}{f(n-1)} \frac{f(n-1)}{f(n-2)} \rightarrow x^2$$

as n gets big. This comes from multiplying $f(n)/f(n-2)$ by $f(n-1)/f(n-1)$ and rearranging.

If x exists, then $x^2 - x - 1 \rightarrow 0$. Using the quadratic equation, the only solution greater than one is

$$x = \frac{1 + \sqrt{5}}{2} \approx 1.618.$$

This expression also has the name ϕ . What does this say about the growth rate of the Fibonacci sequence? It is exponential, with $f(n) = \Theta(\phi^n)$. More precisely, $f(n)$ converges to

$$\frac{\phi^n - (1 - \phi)^n}{\sqrt{5}}.$$

14.2.2 Expanding Recurrences

Estimating bounds is effective if you only need an approximation to the answer. More precise techniques are required to find an exact solution. One approach is called **expanding** the recurrence. In this method, the smaller terms on the right side of the equation are in turn replaced by their definition. This is the expanding step. These terms are again expanded, and so on, until a full series with no recurrence results. This yields a summation, and techniques for solving summations can then be used. A couple of simple expansions were shown in Section 2.4. A more complex example is given below.

Example 14.8 Find the solution for

$$\mathbf{T}(n) = 2\mathbf{T}(n/2) + 5n^2; \quad \mathbf{T}(1) = 7.$$

For simplicity we assume that n is a power of two, so we will rewrite it as $n = 2^k$. This recurrence can be expanded as follows:

$$\begin{aligned} \mathbf{T}(n) &= 2\mathbf{T}(n/2) + 5n^2 \\ &= 2(2\mathbf{T}(n/4) + 5(n/2)^2) + 5n^2 \\ &= 2(2(2\mathbf{T}(n/8) + 5(n/4)^2) + 5(n/2)^2) + 5n^2 \\ &= 2^k \mathbf{T}(1) + 2^{k-1} \cdot 5 \left(\frac{n}{2^{k-1}} \right)^2 + \cdots + 2 \cdot 5 \left(\frac{n}{2} \right)^2 + 5n^2. \end{aligned}$$

This last expression can best be represented by a summation as follows:

$$\begin{aligned} &7n + 5 \sum_{i=0}^{k-1} n^2 / 2^i \\ &= 7n + 5n^2 \sum_{i=0}^{k-1} 1/2^i. \end{aligned}$$

From Equation 2.6, we have:

$$\begin{aligned} &= 7n + 5n^2 \left(2 - 1/2^{k-1} \right) \\ &= 7n + 5n^2(2 - 2/n) \\ &= 7n + 10n^2 - 10n \\ &= 10n^2 - 3n. \end{aligned}$$

This is the *exact* solution to the recurrence for n a power of two. At this point, we should use a simple induction proof to verify that our solution is indeed correct.

Example 14.9 Our next example models the cost of the algorithm to build a heap. Recall from Section 5.5 that to build a heap, we first heapify the two subheaps, then push down the root to its proper position. The cost is:

$$f(n) \leq 2f(n/2) + 2 \log n.$$

Let us find a closed form solution for this recurrence. We can expand the recurrence a few times to see that

$$\begin{aligned}
f(n) &\leq 2f(n/2) + 2\log n \\
&\leq 2[2f(n/4) + 2\log n/2] + 2\log n \\
&\leq 2[2(2f(n/8) + 2\log n/4) + 2\log n/2] + 2\log n
\end{aligned}$$

We can deduce from this expansion that this recurrence is equivalent to following summation and its derivation:

$$\begin{aligned}
f(n) &\leq \sum_{i=0}^{\log n - 1} 2^{i+1} \log(n/2^i) \\
&= 2 \sum_{i=0}^{\log n - 1} 2^i (\log n - i) \\
&= 2 \log n \sum_{i=0}^{\log n - 1} 2^i - 4 \sum_{i=0}^{\log n - 1} i 2^{i-1} \\
&= 2n \log n - 2 \log n - 2n \log n + 4n - 4 \\
&= 4n - 2 \log n - 4.
\end{aligned}$$

14.2.3 Divide and Conquer Recurrences

The third approach to solving recurrences is to take advantage of known theorems that provide the solution for classes of recurrences. Of particular practical use is a theorem that gives the answer for a class known as **divide and conquer** recurrences. These have the form

$$\mathbf{T}(n) = a\mathbf{T}(n/b) + cn^k; \quad \mathbf{T}(1) = c$$

where a , b , c , and k are constants. In general, this recurrence describes a problem of size n divided into a subproblems of size n/b , while cn^k is the amount of work necessary to combine the partial solutions. Mergesort is an example of a divide and conquer algorithm, and its recurrence fits this form. So does binary search. We use the method of expanding recurrences to derive the general solution for any divide and conquer recurrence, assuming that $n = b^m$.

$$\begin{aligned}
\mathbf{T}(n) &= a\mathbf{T}(n/b) + cn^k \\
&= a(a\mathbf{T}(n/b^2) + c(n/b)^k) + cn^k \\
&= a(a[a\mathbf{T}(n/b^3) + c(n/b^2)^k] + c(n/b)^k) + cn^k
\end{aligned}$$

$$\begin{aligned}
&= a^m \mathbf{T}(1) + a^{m-1} c(n/b^{m-1})^k + \cdots + ac(n/b)^k + cn^k \\
&= a^m c + a^{m-1} c(n/b^{m-1})^k + \cdots + ac(n/b)^k + cn^k \\
&= c \sum_{i=0}^m a^{m-i} b^{ik} \\
&= ca^m \sum_{i=0}^m (b^k/a)^i.
\end{aligned}$$

Note that

$$a^m = a^{\log_b n} = n^{\log_b a}. \quad (14.1)$$

The summation is a geometric series whose sum depends on the ratio $r = b^k/a$. There are three cases.

1. $r < 1$. From Equation 2.4,

$$\sum_{i=0}^m r^i < 1/(1-r), \text{ a constant.}$$

Thus,

$$\mathbf{T}(n) = \Theta(a^m) = \Theta(n^{\log_b a}).$$

2. $r = 1$. Because $r = b^k/a$, we know that $a = b^k$. From the definition of logarithms it follows immediately that $k = \log_b a$. We also note from Equation 14.1 that $m = \log_b n$. Thus,

$$\sum_{i=0}^m r = m + 1 = \log_b n + 1.$$

Because $a^m = n \log_b a = n^k$, we have

$$\mathbf{T}(n) = \Theta(n^{\log_b a} \log n) = \Theta(n^k \log n).$$

3. $r > 1$. From Equation 2.5,

$$\sum_{i=0}^m r = \frac{r^{m+1} - 1}{r - 1} = \Theta(r^m).$$

Thus,

$$\mathbf{T}(n) = \Theta(a^m r^m) = \Theta(a^m (b^k/a)^m) = \Theta(b^{km}) = \Theta(n^k).$$

We can summarize the above derivation as the following theorem, sometimes referred to as the **Master Theorem**.

Theorem 14.1 (The Master Theorem) For any recurrence relation of the form $\mathbf{T}(n) = a\mathbf{T}(n/b) + cn^k$, $\mathbf{T}(1) = c$, the following relationships hold.

$$\mathbf{T}(n) = \begin{cases} \Theta(n^{\log_b a}) & \text{if } a > b^k \\ \Theta(n^k \log n) & \text{if } a = b^k \\ \Theta(n^k) & \text{if } a < b^k. \end{cases}$$

This theorem may be applied whenever appropriate, rather than re-deriving the solution for the recurrence.

Example 14.10 Apply the Master Theorem to solve

$$\mathbf{T}(n) = 3\mathbf{T}(n/5) + 8n^2.$$

Because $a = 3$, $b = 5$, $c = 8$, and $k = 2$, we find that $3 < 5^2$. Applying case (3) of the theorem, $\mathbf{T}(n) = \Theta(n^2)$.

Example 14.11 Use the Master Theorem to solve the recurrence relation for Mergesort:

$$\mathbf{T}(n) = 2\mathbf{T}(n/2) + n; \quad \mathbf{T}(1) = 1.$$

Because $a = 2$, $b = 2$, $c = 1$, and $k = 1$, we find that $2 = 2^1$. Applying case (2) of the theorem, $\mathbf{T}(n) = \Theta(n \log n)$.

14.2.4 Average-Case Analysis of Quicksort

In Section 7.5, we determined that the average-case analysis of Quicksort had the following recurrence:

$$\mathbf{T}(n) = cn + \frac{1}{n} \sum_{k=0}^{n-1} [\mathbf{T}(k) + \mathbf{T}(n-1-k)], \quad \mathbf{T}(0) = \mathbf{T}(1) = c.$$

The cn term is an upper bound on the **findpivot** and **partition** steps. This equation comes from assuming that the partitioning element is equally likely to occur in any position k . It can be simplified by observing that the two recurrence terms $\mathbf{T}(k)$ and $\mathbf{T}(n-1-k)$ are equivalent, because one simply counts up from $T(0)$ to $T(n-1)$ while the other counts down from $T(n-1)$ to $T(0)$. This yields

$$\mathbf{T}(n) = cn + \frac{2}{n} \sum_{k=0}^{n-1} \mathbf{T}(k).$$

This form is known as a recurrence with **full history**. The key to solving such a recurrence is to cancel out the summation terms. The shifting method for summations provides a way to do this. Multiply both sides by n and subtract the result from the formula for $n\mathbf{T}(n+1)$:

$$\begin{aligned} n\mathbf{T}(n) &= cn^2 + 2 \sum_{k=1}^{n-1} \mathbf{T}(k) \\ (n+1)\mathbf{T}(n+1) &= c(n+1)^2 + 2 \sum_{k=1}^n \mathbf{T}(k). \end{aligned}$$

Subtracting $n\mathbf{T}(n)$ from both sides yields:

$$\begin{aligned} (n+1)\mathbf{T}(n+1) - n\mathbf{T}(n) &= c(n+1)^2 - cn^2 + 2\mathbf{T}(n) \\ (n+1)\mathbf{T}(n+1) - n\mathbf{T}(n) &= c(2n+1) + 2\mathbf{T}(n) \\ (n+1)\mathbf{T}(n+1) &= c(2n+1) + (n+2)\mathbf{T}(n) \\ \mathbf{T}(n+1) &= \frac{c(2n+1)}{n+1} + \frac{n+2}{n+1}\mathbf{T}(n). \end{aligned}$$

At this point, we have eliminated the summation and can now use our normal methods for solving recurrences to get a closed-form solution. Note that $\frac{c(2n+1)}{n+1} < 2c$, so we can simplify the result. Expanding the recurrence, we get

$$\begin{aligned} \mathbf{T}(n+1) &\leq 2c + \frac{n+2}{n+1}\mathbf{T}(n) \\ &= 2c + \frac{n+2}{n+1} \left(2c + \frac{n+1}{n}\mathbf{T}(n-1) \right) \\ &= 2c + \frac{n+2}{n+1} \left(2c + \frac{n+1}{n} \left(2c + \frac{n}{n-1}\mathbf{T}(n-2) \right) \right) \\ &= 2c + \frac{n+2}{n+1} \left(2c + \cdots + \frac{4}{3} \left(2c + \frac{3}{2}\mathbf{T}(1) \right) \right) \\ &= 2c \left(1 + \frac{n+2}{n+1} + \frac{n+2}{n+1} \frac{n+1}{n} + \cdots + \frac{n+2}{n+1} \frac{n+1}{n} \cdots \frac{3}{2} \right) \\ &= 2c \left(1 + (n+2) \left(\frac{1}{n+1} + \frac{1}{n} + \cdots + \frac{1}{2} \right) \right) \\ &= 2c + 2c(n+2)(\mathcal{H}_{n+1} - 1) \end{aligned}$$

for \mathcal{H}_{n+1} , the Harmonic Series. From Equation 2.10, $\mathcal{H}_{n+1} = \Theta(\log n)$, so the final solution is $\Theta(n \log n)$.

14.3 Amortized Analysis

This section presents the concept of **amortized analysis**, which is the analysis for a series of operations taken as a whole. In particular, amortized analysis allows us to deal with the situation where the worst-case cost for n operations is less than n times the worst-case cost of any one operation. Rather than focusing on the individual cost of each operation independently and summing them, amortized analysis looks at the cost of the entire series and “charges” each individual operation with a share of the total cost.

We can apply the technique of amortized analysis in the case of a series of sequential searches in an unsorted array. For n random searches, the average-case cost for each search is $n/2$, and so the *expected* total cost for the series is $n^2/2$. Unfortunately, in the worst case all of the searches would be to the last item in the array. In this case, each search costs n for a total worst-case cost of n^2 . Compare this to the cost for a series of n searches such that each item in the array is searched for precisely once. In this situation, some of the searches *must* be expensive, but also some searches *must* be cheap. The total number of searches, in the best, average, and worst case, for this problem must be $\sum_{i=1}^n i \approx n^2/2$. This is a factor of two better than the more pessimistic analysis that charges each operation in the series with its worst-case cost.

As another example of amortized analysis, consider the process of incrementing a binary counter. The algorithm is to move from the lower-order (rightmost) bit toward the high-order (leftmost) bit, changing 1s to 0s until the first 0 is encountered. This 0 is changed to a 1, and the increment operation is done. Below is C++ code to implement the increment operation, assuming that a binary number of length n is stored in array **A** of length n .

```
for (i=0; ((i<n) && (A[i] == 1)); i++)
    A[i] = 0;
if (i < n)
    A[i] = 1;
```

If we count from 0 through $2^n - 1$, (requiring a counter with at least n bits), what is the average cost for an increment operation in terms of the number of bits processed? Naive worst-case analysis says that if all n bits are 1 (except for the high-order bit), then n bits need to be processed. Thus, if there are 2^n increments, then the cost is $n2^n$. However, this is much too high, because it is rare for so many bits to be processed. In fact, half of the time the low-order bit is 0, and so only that bit is processed. One quarter of the time, the low-order two bits are 01, and so only the low-order two bits are processed. Another way to view this is that the low-order bit is always flipped, the bit to its left is flipped half the time, the next bit one quarter of the time, and so on. We can capture this with the summation

(charging costs to bits going from right to left)

$$\sum_{i=0}^{n-1} \frac{1}{2^i} < 2.$$

In other words, the average number of bits flipped on each increment is 2, leading to a total cost of only $2 \cdot 2^n$ for a series of 2^n increments.

A useful concept for amortized analysis is illustrated by a simple variation on the stack data structure, where the **pop** function is slightly modified to take a second parameter k indicating that k pop operations are to be performed. This revised pop function, called **multipop**, might look as follows:

```
// pop k elements from stack
void multipop(int k);
```

The “local” worst-case analysis for **multipop** is $\Theta(n)$ for n elements in the stack. Thus, if there are m_1 calls to **push** and m_2 calls to **multipop**, then the naive worst-case cost for the series of operation is $m_1 + m_2 \cdot n = m_1 + m_2 \cdot m_1$. This analysis is unreasonably pessimistic. Clearly it is not really possible to pop m_1 elements each time **multipop** is called. Analysis that focuses on single operations cannot deal with this global limit, and so we turn to amortized analysis to model the entire series of operations.

The key to an amortized analysis of this problem lies in the concept of **potential**. At any given time, a certain number of items may be on the stack. The cost for **multipop** can be no more than this number of items. Each call to **push** places another item on the stack, which can be removed by only a single **multipop** operation. Thus, each call to **push** raises the potential of the stack by one item. The sum of costs for all calls to **multipop** can never be more than the total potential of the stack (aside from a constant time cost associated with each call to **multipop** itself).

The amortized cost for any series of **push** and **multipop** operations is the sum of three costs. First, each of the **push** operations takes constant time. Second, each **multipop** operation takes a constant time in overhead, regardless of the number of items popped on that call. Finally, we count the sum of the potentials expended by all **multipop** operations, which is at most m_1 , the number of **push** operations. This total cost can therefore be expressed as

$$m_1 + (m_2 + m_1) = \Theta(m_1 + m_2).$$

A similar argument was used in our analysis for the partition function in the Quicksort algorithm (Section 7.5). While on any given pass through the while loop the left or right pointers might move all the way through the remainder of the

partition, doing so would reduce the number of times that the while loop can be further executed.

Our final example uses amortized analysis to prove a relationship between the cost of the move-to-front self-organizing list heuristic from Section 9.2 and the cost for the optimal static ordering of the list.

Recall that, for a series of search operations, the minimum cost for a static list results when the list is sorted by frequency of access to its records. This is the optimal ordering for the records if we never allow the positions of records to change, because the most-frequently accessed record is first (and thus has least cost), followed by the next most frequently accessed record, and so on.

Theorem 14.2 *The total number of comparisons required by any series S of n or more searches on a self-organizing list of length n using the move-to-front heuristic is never more than twice the total number of comparisons required when series S is applied to the list stored in its optimal static order.*

Proof: Each comparison of the search key with a record in the list is either successful or unsuccessful. For m searches, there must be exactly m successful comparisons for both the self-organizing list and the static list. The total number of unsuccessful comparisons in the self-organizing list is the sum, over all pairs of distinct keys, of the number of unsuccessful comparisons made between that pair.

Consider a particular pair of keys A and B . For any sequence of searches S , the total number of (unsuccessful) comparisons between A and B is identical to the number of comparisons between A and B required for the subsequence of S made up only of searches for A or B . Call this subsequence S_{AB} . In other words, including searches for other keys does not affect the relative position of A and B and so does not affect the relative contribution to the total cost of the unsuccessful comparisons between A and B .

The number of unsuccessful comparisons between A and B made by the move-to-front heuristic on subsequence S_{AB} is at most twice the number of unsuccessful comparisons between A and B required when S_{AB} is applied to the optimal static ordering for the list. To see this, assume that S_{AB} contains i A s and j B s, with $i \leq j$. Under the optimal static ordering, i unsuccessful comparisons are required because B must appear before A in the list (because its access frequency is higher). Move-to-front will yield an unsuccessful comparison whenever the request sequence changes from A to B or from B to A . The total number of such changes possible is $2i$ because each change involves an A and each A can be part of at most two changes.

Because the total number of unsuccessful comparisons required by move-to-front for any given pair of keys is at most twice that required by the optimal static ordering, the total number of unsuccessful comparisons required by move-to-front for all pairs of keys is also at most twice as high. Because the number of successful

comparisons is the same for both methods, the total number of comparisons required by move-to-front is less than twice the number of comparisons required by the optimal static ordering. \square

14.4 Further Reading

A good introduction to solving recurrence relations appears in *Applied Combinatorics* by Fred S. Roberts [Rob84]. For a more advanced treatment, see *Concrete Mathematics* by Graham, Knuth, and Patashnik [GKP94].

Cormen, Leiserson, and Rivest provide a good discussion on various methods for performing amortized analysis in *Introduction to Algorithms* [CLRS09]. For an amortized analysis that the splay tree requires $m \log n$ time to perform a series of m operations on n nodes when $m > n$, see “Self-Adjusting Binary Search Trees” by Sleator and Tarjan [ST85]. The proof for Theorem 14.2 comes from “Amortized Analysis of Self-Organizing Sequential Search Heuristics” by Bentley and McGeoch [BM85].

14.5 Exercises

- 14.1** Use the technique of guessing a polynomial and deriving the coefficients to solve the summation

$$\sum_{i=1}^n i^2.$$

- 14.2** Use the technique of guessing a polynomial and deriving the coefficients to solve the summation

$$\sum_{i=1}^n i^3.$$

- 14.3** Find, and prove correct, a closed-form solution for

$$\sum_{i=a}^b i^2.$$

- 14.4** Use subtract-and-guess or divide-and-guess to find the closed form solution for the following summation. You must first find a pattern from which to deduce a potential closed form solution, and then prove that the proposed solution is correct.

$$\sum_{i=1}^n i/2^i$$

14.5 Use the shifting method to solve the summation

$$\sum_{i=1}^n i^2.$$

14.6 Use the shifting method to solve the summation

$$\sum_{i=1}^n 2^i.$$

14.7 Use the shifting method to solve the summation

$$\sum_{i=1}^n i2^{n-i}.$$

14.8 Consider the following code fragment.

```
sum = 0; inc = 0;
for (i=1; i<=n; i++)
    for (j=1; j<=i; j++) {
        sum = sum + inc;
        inc++;
    }
```

- (a) Determine a summation that defines the final value for variable `sum` as a function of n .
- (b) Determine a closed-form solution for your summation.

14.9 A chocolate company decides to promote its chocolate bars by including a coupon with each bar. A bar costs a dollar, and with c coupons you get a free bar. So depending on the value of c , you get more than one bar of chocolate for a dollar when considering the value of the coupons. How much chocolate is a dollar worth (as a function of c)?

14.10 Write and solve a recurrence relation to compute the number of times `Fibr` is called in the `Fibr` function of Exercise 2.11.

14.11 Give and prove the closed-form solution for the recurrence relation $T(n) = T(n-1) + 1$, $T(1) = 1$.

14.12 Give and prove the closed-form solution for the recurrence relation $T(n) = T(n-1) + c$, $T(1) = c$.

14.13 Prove by induction that the closed-form solution for the recurrence relation

$$T(n) = 2T(n/2) + n; \quad T(2) = 1$$

is in $\Omega(n \log n)$.

- 14.14** For the following recurrence, give a closed-form solution. You should not give an exact solution, but only an asymptotic solution (i.e., using Θ notation). You may assume that n is a power of 2. Prove that your answer is correct.

$$T(n) = T(n/2) + \sqrt{n} \text{ for } n > 1; \quad T(1) = 1.$$

- 14.15** Using the technique of expanding the recurrence, find the exact closed-form solution for the recurrence relation

$$T(n) = 2T(n/2) + n; \quad T(2) = 2.$$

You may assume that n is a power of 2.

- 14.16** Section 5.5 provides an asymptotic analysis for the worst-case cost of function **buildHeap**. Give an exact worst-case analysis for **buildHeap**.

- 14.17** For each of the following recurrences, find and then prove (using induction) an exact closed-form solution. When convenient, you may assume that n is a power of 2.

(a) $T(n) = T(n-1) + n/2$ for $n > 1$; $T(1) = 1$.

(b) $T(n) = 2T(n/2) + n$ for $n > 2$; $T(2) = 2$.

- 14.18** Use Theorem 14.1 to prove that binary search requires $\Theta(\log n)$ time.

- 14.19** Recall that when a hash table gets to be more than about one half full, its performance quickly degrades. One solution to this problem is to reinsert all elements of the hash table into a new hash table that is twice as large. Assuming that the (expected) average case cost to insert into a hash table is $\Theta(1)$, prove that the average cost to insert is still $\Theta(1)$ when this re-insertion policy is used.

- 14.20** Given a 2-3 tree with N nodes, prove that inserting M additional nodes requires $O(M + N)$ node splits.

- 14.21** One approach to implementing an array-based list where the list size is unknown is to let the array grow and shrink. This is known as a **dynamic array**. When necessary, we can grow or shrink the array by copying the array's contents to a new array. If we are careful about the size of the new array, this copy operation can be done rarely enough so as not to affect the amortized cost of the operations.

- (a) What is the amortized cost of inserting elements into the list if the array is initially of size 1 and we double the array size whenever the number of elements that we wish to store exceeds the size of the array? Assume that the insert itself cost $O(1)$ time per operation and so we are just concerned with minimizing the copy time to the new array.

- (b) Consider an underflow strategy that cuts the array size in half whenever the array falls below half full. Give an example where this strategy leads to a bad amortized cost. Again, we are only interested in measuring the time of the array copy operations.
- (c) Give a better underflow strategy than that suggested in part (b). Your goal is to find a strategy whose amortized analysis shows that array copy requires $O(n)$ time for a series of n operations.

14.22 Recall that two vertices in an undirected graph are in the same connected component if there is a path connecting them. A good algorithm to find the connected components of an undirected graph begins by calling a DFS on the first vertex. All vertices reached by the DFS are in the same connected component and are so marked. We then look through the vertex **mark** array until an unmarked vertex i is found. Again calling the DFS on i , all vertices reachable from i are in a second connected component. We continue working through the **mark** array until all vertices have been assigned to some connected component. A sketch of the algorithm is as follows:

```
void DFS_component(Graph* G, int v, int component) {
    G->setMark(v, component);
    for (int w=G->first(v); w<G->n(); w = G->next(v,w))
        if (G->getMark(w) == 0)
            DFS_component(G, w, component);
}

void concom(Graph* G) {
    int i;
    int component = 1;    // Counter for current component
    for (i=0; i<G->n(); i++) // For n vertices in graph
        G->setMark(i, 0); // Vertices start in no component
    for (i=0; i<G->n(); i++)
        if (G->getMark(i) == 0) // Start a new component
            DFS_component(G, i, component++);
}
```

Use the concept of potential from amortized analysis to explain why the total cost of this algorithm is $\Theta(|V| + |E|)$. (Note that this will not be a true amortized analysis because this algorithm does not allow an arbitrary series of DFS operations but rather is fixed to do a single call to DFS from each vertex.)

14.23 Give a proof similar to that used for Theorem 14.2 to show that the total number of comparisons required by any series of n or more searches S on a self-organizing list of length n using the count heuristic is never more than twice the total number of comparisons required when series S is applied to the list stored in its optimal static order.

14.24 Use mathematical induction to prove that

$$\sum_{i=1}^n \text{Fib}(i) = \text{Fib}(n+2) - 1, \text{ for } n \geq 1.$$

14.25 Use mathematical induction to prove that $\text{Fib}(i)$ is even if and only if i is divisible by 3.

14.26 Use mathematical induction to prove that for $n \geq 6$, $\text{fib}(n) > (3/2)^{n-1}$.

14.27 Find closed forms for each of the following recurrences.

- (a) $F(n) = F(n-1) + 3; F(1) = 2.$
- (b) $F(n) = 2F(n-1); F(0) = 1.$
- (c) $F(n) = 2F(n-1) + 1; F(1) = 1.$
- (d) $F(n) = 2nF(n-1); F(0) = 1.$
- (e) $F(n) = 2^n F(n-1); F(0) = 1.$
- (f) $F(n) = 2 + \sum_{i=1}^{n-1} F(i); F(1) = 1.$

14.28 Find Θ for each of the following recurrence relations.

- (a) $T(n) = 2T(n/2) + n^2.$
- (b) $T(n) = 2T(n/2) + 5.$
- (c) $T(n) = 4T(n/2) + n.$
- (d) $T(n) = 2T(n/2) + n^2.$
- (e) $T(n) = 4T(n/2) + n^3.$
- (f) $T(n) = 4T(n/3) + n.$
- (g) $T(n) = 4T(n/3) + n^2.$
- (h) $T(n) = 2T(n/2) + \log n.$
- (i) $T(n) = 2T(n/2) + n \log n.$

14.6 Projects

14.1 Implement the UNION/FIND algorithm of Section 6.2 using both path compression and the weighted union rule. Count the total number of node accesses required for various series of equivalences to determine if the actual performance of the algorithm matches the expected cost of $\Theta(n \log^* n)$.

Lower Bounds

How do I know if I have a good algorithm to solve a problem? If my algorithm runs in $\Theta(n \log n)$ time, is that good? It would be if I were sorting the records stored in an array. But it would be terrible if I were searching the array for the largest element. The value of an algorithm must be determined in relation to the inherent complexity of the problem at hand.

In Section 3.6 we defined the upper bound for a problem to be the upper bound of the best algorithm we know for that problem, and the lower bound to be the tightest lower bound that we can prove over all algorithms for that problem. While we usually can recognize the upper bound for a given algorithm, finding the tightest lower bound for all possible algorithms is often difficult, especially if that lower bound is more than the “trivial” lower bound determined by measuring the amount of input that must be processed.

The benefits of being able to discover a strong lower bound are significant. In particular, when we can make the upper and lower bounds for a problem meet, this means that we truly understand our problem in a theoretical sense. It also saves us the effort of attempting to discover more (asymptotically) efficient algorithms when no such algorithm can exist.

Often the most effective way to determine the lower bound for a problem is to find a reduction to another problem whose lower bound is already known. This is the subject of Chapter 17. However, this approach does not help us when we cannot find a suitable “similar problem.” Our focus in this chapter is discovering and proving lower bounds from first principles. Our most significant example of a lower bounds argument so far is the proof from Section 7.9 that the problem of sorting is $O(n \log n)$ in the worst case.

Section 15.1 reviews the concept of a lower bound for a problem and presents the basic “algorithm” for finding a good algorithm. Section 15.2 discusses lower bounds on searching in lists, both those that are unordered and those that are ordered. Section 15.3 deals with finding the maximum value in a list, and presents a model for selection based on building a partially ordered set. Section 15.4 presents

the concept of an adversarial lower bounds proof. Section 15.5 illustrates the concept of a state space lower bound. Section 15.6 presents a linear time worst-case algorithm for finding the i th biggest element on a list. Section 15.7 continues our discussion of sorting with a quest for the algorithm that requires the absolute fewest number of comparisons needed to sort a list.

15.1 Introduction to Lower Bounds Proofs

The lower bound for the problem is the tightest (highest) lower bound that we can prove *for all possible algorithms* that solve the problem.¹ This can be a difficult bar, given that we cannot possibly know all algorithms for any problem, because there are theoretically an infinite number. However, we can often recognize a simple lower bound based on the amount of input that must be examined. For example, we can argue that the lower bound for any algorithm to find the maximum-valued element in an unsorted list must be $\Omega(n)$ because any algorithm must examine all of the inputs to be sure that it actually finds the maximum value.

In the case of maximum finding, the fact that we know of a simple algorithm that runs in $O(n)$ time, combined with the fact that any algorithm needs $\Omega(n)$ time, is significant. Because our upper and lower bounds meet (within a constant factor), we know that we do have a “good” algorithm for solving the problem. It is possible that someone can develop an implementation that is a “little” faster than an existing one, by a constant factor. But we know that its not possible to develop one that is asymptotically better.

We must be careful about how we interpret this last statement, however. The world is certainly better off for the invention of Quicksort, even though Mergesort was available at the time. Quicksort is not asymptotically faster than Mergesort, yet is not merely a “tuning” of Mergesort either. Quicksort is a substantially different approach to sorting. So even when our upper and lower bounds for a problem meet, there are still benefits to be gained from a new, clever algorithm.

So now we have an answer to the question “How do I know if I have a good algorithm to solve a problem?” An algorithm is good (asymptotically speaking) if its upper bound matches the problem’s lower bound. If they match, we know to stop trying to find an (asymptotically) faster algorithm. What if the (known) upper bound for our algorithm does not match the (known) lower bound for the problem? In this case, we might not know what to do. Is our upper bound flawed, and the algorithm is really faster than we can prove? Is our lower bound weak, and the true lower bound for the problem is greater? Or is our algorithm simply not the best?

¹Throughout this discussion, it should be understood that any mention of bounds must specify what class of inputs are being considered. Do we mean the bound for the worst case input? The average cost over all inputs? Regardless of which class of inputs we consider, all of the issues raised apply equally.

Now we know precisely what we are aiming for when designing an algorithm: We want to find an algorithm whose upper bound matches the lower bound of the problem. Putting together all that we know so far about algorithms, we can organize our thinking into the following “algorithm for designing algorithms.”²

```

If the upper and lower bounds match,
then stop,
else if the bounds are close or the problem isn't important,
then stop,
else if the problem definition focuses on the wrong thing,
then restate it,
else if the algorithm is too slow,
then find a faster algorithm,
else if lower bound is too weak,
then generate a stronger bound.

```

We can repeat this process until we are satisfied or exhausted.

This brings us smack up against one of the toughest tasks in analysis. Lower bounds proofs are notoriously difficult to construct. The problem is coming up with arguments that truly cover all of the things that *any* algorithm possibly *could* do. The most common fallacy is to argue from the point of view of what some good algorithm actually *does* do, and claim that any algorithm must do the same. This simply is not true, and any lower bounds proof that refers to specific behavior that must take place should be viewed with some suspicion.

Let us consider the Towers of Hanoi problem again. Recall from Section 2.5 that our basic algorithm is to move $n - 1$ disks (recursively) to the middle pole, move the bottom disk to the third pole, and then move $n - 1$ disks (again recursively) from the middle to the third pole. This algorithm generates the recurrence $T(n) = 2T(n - 1) + 1 = 2^n - 1$. So, the upper bound for our algorithm is $2^n - 1$. But is this the best algorithm for the problem? What is the lower bound for the problem?

For our first try at a lower bounds proof, the “trivial” lower bound is that we must move every disk at least once, for a minimum cost of n . Slightly better is to observe that to get the bottom disk to the third pole, we must move every other disk at least twice (once to get them off the bottom disk, and once to get them over to the third pole). This yields a cost of $2n - 1$, which still is not a good match for our algorithm. Is the problem in the algorithm or in the lower bound?

We can get to the correct lower bound by the following reasoning: To move the biggest disk from first to the last pole, we must first have all of the other $n - 1$ disks out of the way, and the only way to do that is to move them all to the middle pole (for a cost of at least $T(n - 1)$). We then must move the bottom disk (for a cost of

²This is a minor reformulation of the “algorithm” given by Gregory J.E. Rawlins in his book “Compared to What?”

at least one). After that, we must move the $n - 1$ remaining disks from the middle pole to the third pole (for a cost of at least $T(n - 1)$). Thus, no possible algorithm can solve the problem in less than $2^n - 1$ steps. Thus, our algorithm is optimal.³

Of course, there are variations to a given problem. Changes in the problem definition might or might not lead to changes in the lower bound. Two possible changes to the standard Towers of Hanoi problem are:

- Not all disks need to start on the first pole.
- Multiple disks can be moved at one time.

The first variation does not change the lower bound (at least not asymptotically). The second one does.

15.2 Lower Bounds on Searching Lists

In Section 7.9 we presented an important lower bounds proof to show that the problem of sorting is $\Theta(n \log n)$ in the worst case. In Chapter 9 we discussed a number of algorithms to search in sorted and unsorted lists, but we did not provide any lower bounds proofs to this important problem. We will extend our pool of techniques for lower bounds proofs in this section by studying lower bounds for searching unsorted and sorted lists.

15.2.1 Searching in Unsorted Lists

Given an (unsorted) list L of n elements and a search key K , we seek to identify one element in L which has key value K , if any exists. For the rest of this discussion, we will assume that the key values for the elements in L are unique, that the set of all possible keys is totally ordered (that is, the operations $<$, $=$, and $>$ are defined for all pairs of key values), and that comparison is our only way to find the relative ordering of two keys. Our goal is to solve the problem using the minimum number of comparisons.

Given this definition for searching, we can easily come up with the standard sequential search algorithm, and we can also see that the lower bound for this problem is “obviously” n comparisons. (Keep in mind that the key K might not actually appear in the list.) However, lower bounds proofs are a bit slippery, and it is instructive to see how they can go wrong.

Theorem 15.1 *The lower bound for the problem of searching in an unsorted list is n comparisons.*

³Recalling the advice to be suspicious of any lower bounds proof that argues a given behavior “must” happen, this proof should be raising red flags. However, in this particular case the problem is so constrained that there really is no (better) alternative to this particular sequence of events.

Here is our first attempt at proving the theorem.

Proof 1: We will try a proof by contradiction. Assume an algorithm A exists that requires only $n - 1$ (or less) comparisons of K with elements of \mathbf{L} . Because there are n elements of \mathbf{L} , A must have avoided comparing K with $\mathbf{L}[i]$ for some value i . We can feed the algorithm an input with K in position i . Such an input is legal in our model, so the algorithm is incorrect. \square

Is this proof correct? Unfortunately no. First of all, any given algorithm need not necessarily consistently skip any given position i in its $n - 1$ searches. For example, it is not necessary that all algorithms search the list from left to right. It is not even necessary that all algorithms search the same $n - 1$ positions first each time through the list.

We can try to dress up the proof as follows: **Proof 2:** On any given run of the algorithm, if $n - 1$ elements are compared against K , then *some* element position (call it position i) gets skipped. It is possible that K is in position i at that time, and will not be found. Therefore, n comparisons are required. \square

Unfortunately, there is another error that needs to be fixed. It is not true that all algorithms for solving the problem must work by comparing elements of \mathbf{L} against K . An algorithm might make useful progress by comparing elements of \mathbf{L} against each other. For example, if we compare two elements of \mathbf{L} , then compare the greater against K and find that this element is less than K , we know that the other element is also less than K . It seems intuitively obvious that such comparisons won't actually lead to a faster algorithm, but how do we know for sure? We somehow need to generalize the proof to account for this approach.

We will now present a useful abstraction for expressing the state of knowledge for the value relationships among a set of objects. A **total order** defines relationships within a collection of objects such that for every pair of objects, one is greater than the other. A **partially ordered set** or **poset** is a set on which only a partial order is defined. That is, there can be pairs of elements for which we cannot decide which is "greater". For our purpose here, the partial order is the state of our current knowledge about the objects, such that zero or more of the order relations between pairs of elements are known. We can represent this knowledge by drawing directed acyclic graphs (DAGs) showing the known relationships, as illustrated by Figure 15.1.

Proof 3: Initially, we know nothing about the relative order of the elements in \mathbf{L} , or their relationship to K . So initially, we can view the n elements in \mathbf{L} as being in n separate partial orders. Any comparison between two elements in \mathbf{L} can affect the structure of the partial orders. This is somewhat similar to the UNION/FIND algorithm implemented using parent pointer trees, described in Section 6.2.

Now, every comparison between elements in \mathbf{L} can at best combine two of the partial orders together. Any comparison between K and an element, say A , in \mathbf{L} can at best eliminate the partial order that contains A . Thus, if we spend m comparisons

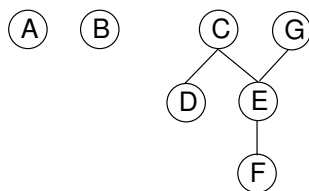


Figure 15.1 Illustration of using a poset to model our current knowledge of the relationships among a collection of objects. A directed acyclic graph (DAG) is used to draw the poset (assume all edges are directed downward). In this example, our knowledge is such that we don't know how A or B relate to any of the other objects. However, we know that both C and G are greater than E and F . Further, we know that C is greater than D , and that E is greater than F .

comparing elements in \mathbf{L} we have at least $n - m$ partial orders. Every such partial order needs at least one comparison against K to make sure that K is not somewhere in that partial order. Thus, any algorithm must make at least n comparisons in the worst case. \square

15.2.2 Searching in Sorted Lists

We will now assume that list \mathbf{L} is sorted. In this case, is linear search still optimal? Clearly no, but why not? Because we have additional information to work with that we do not have when the list is unsorted. We know that the standard binary search algorithm has a worst case cost of $O(\log n)$. Can we do better than this? We can prove that this is the best possible in the worst case with a proof similar to that used to show the lower bound on sorting.

Again we use the decision tree to model our algorithm. Unlike when searching an unsorted list, comparisons between elements of \mathbf{L} tell us nothing new about their relative order, so we consider only comparisons between K and an element in \mathbf{L} . At the root of the decision tree, our knowledge rules out no positions in \mathbf{L} , so all are potential candidates. As we take branches in the decision tree based on the result of comparing K to an element in \mathbf{L} , we gradually rule out potential candidates. Eventually we reach a leaf node in the tree representing the single position in \mathbf{L} that can contain K . There must be at least $n + 1$ nodes in the tree because we have $n + 1$ distinct positions that K can be in (any position in \mathbf{L} , plus not in \mathbf{L} at all). Some path in the tree must be at least $\log n$ levels deep, and the deepest node in the tree represents the worst case for that algorithm. Thus, any algorithm on a sorted array requires at least $\Omega(\log n)$ comparisons in the worst case.

We can modify this proof to find the average cost lower bound. Again, we model algorithms using decision trees. Except now we are interested not in the depth of the deepest node (the worst case) and therefore the tree with the least-deepest node. Instead, we are interested in knowing what the minimum possible is

for the “average depth” of the leaf nodes. Define the **total path length** as the sum of the levels for each node. The cost of an outcome is the level of the corresponding node plus 1. The average cost of the algorithm is the average cost of the outcomes (total path length/ n). What is the tree with the least average depth? This is equivalent to the tree that corresponds to binary search. Thus, binary search is optimal in the average case.

While binary search is indeed an optimal algorithm for a sorted list in the worst and average cases when searching a sorted array, there are a number of circumstances that might lead us to select another algorithm instead. One possibility is that we know something about the distribution of the data in the array. We saw in Section 9.1 that if each position in L is equally likely to hold X (equivalently, the data are well distributed along the full key range), then an interpolation search is $\Theta(\log \log n)$ in the average case. If the data are not sorted, then using binary search requires us to pay the cost of sorting the list in advance, which is only worthwhile if many (at least $O(\log n)$) searches will be performed on the list. Binary search also requires that the list (even if sorted) be implemented using an array or some other structure that supports random access to all elements with equal cost. Finally, if we know all search requests in advance, we might prefer to sort the list by frequency and do linear search in extreme search distributions, as discussed in Section 9.2.

15.3 Finding the Maximum Value

How can we find the i th largest value in a sorted list? Obviously we just go to the i th position. But what if we have an unsorted list? Can we do better than to sort it? If we are looking for the minimum or maximum value, certainly we can do better than sorting the list. Is this true for the second biggest value? For the median value? In later sections we will examine those questions. For this section, we will continue our examination of lower bounds proofs by reconsidering the simple problem of finding the maximum value in an unsorted list.

Here is a simple algorithm for finding the largest value.

```
// Return position of largest value in "A" of size "n"
int largest(int A[], int n) {
    int currlarge = 0; // Holds largest element position
    for (int i=1; i<n; i++) // For each array element
        if (A[currlarge] < A[i]) // if A[i] is larger
            currlarge = i; // remember its position
    return currlarge; // Return largest position
}
```

Obviously this algorithm requires n comparisons. Is this optimal? It should be intuitively obvious that it is, but let us try to prove it. (Before reading further you might try writing down your own proof.)

Proof 1: The winner must compare against all other elements, so there must be $n - 1$ comparisons. \square

This proof is clearly wrong, because the winner does not need to explicitly compare against all other elements to be recognized. For example, a standard single-elimination playoff sports tournament requires only $n - 1$ comparisons, and the winner does not play every opponent. So let's try again.

Proof 2: Only the winner does not lose. There are $n - 1$ losers. A single comparison generates (at most) one (new) loser. Therefore, there must be $n - 1$ comparisons. \square

This proof is sound. However, it will be useful later to abstract this by introducing the concept of posets as we did in Section 15.2.1. We can view the maximum-finding problem as starting with a poset where there are no known relationships, so every member of the collection is in its own separate DAG of one element.

Proof 2a: To find the largest value, we start with a poset of n DAGs each with a single element, and we must build a poset having all elements in one DAG such that there is one maximum value (and by implication, $n - 1$ losers). We wish to connect the elements of the poset into a single DAG with the minimum number of links. This requires at least $n - 1$ links. A comparison provides at most one new link. Thus, a minimum of $n - 1$ comparisons must be made. \square

What is the average cost of **largest**? Because it always does the same number of comparisons, clearly it must cost $n - 1$ comparisons. We can also consider the number of assignments that **largest** must do. Function **largest** might do an assignment on any iteration of the **for** loop.

Because this event does happen, or does not happen, if we are given no information about distribution we could guess that an assignment is made after each comparison with a probability of one half. But this is clearly wrong. In fact, **largest** does an assignment on the i th iteration if and only if $\mathbf{A}[i]$ is the biggest of the first i elements. Assuming all permutations are equally likely, the probability of this being true is $1/i$. Thus, the average number of assignments done is

$$1 + \sum_{i=2}^n \frac{1}{i} = \sum_{i=1}^n \frac{1}{i}$$

which is the Harmonic Series \mathcal{H}_n . $\mathcal{H}_n = \Theta(\log n)$. More exactly, \mathcal{H}_n is close to $\log_e n$.

How “reliable” is this average? That is, how much will a given run of the program deviate from the mean cost? According to Čebyšev's Inequality, an observation will fall within two standard deviations of the mean at least 75% of the time. For **Largest**, the variance is

$$\mathcal{H}_n - \frac{\pi^2}{6} = \log_e n - \frac{\pi^2}{6}.$$

The standard deviation is thus about $\sqrt{\log_e n}$. So, 75% of the observations are between $\log_e n - 2\sqrt{\log_e n}$ and $\log_e n + 2\sqrt{\log_e n}$. Is this a narrow spread or a wide spread? Compared to the mean value, this spread is pretty wide, meaning that the number of assignments varies widely from run to run of the program.

15.4 Adversarial Lower Bounds Proofs

Our next problem will be finding the second largest in a collection of objects. Consider what happens in a standard single-elimination tournament. Even if we assume that the “best” team wins in every game, is the second best the one that loses in the finals? Not necessarily. We might expect that the second best must lose to the best, but they might meet at any time.

Let us go through our standard “algorithm for finding algorithms” by first proposing an algorithm, then a lower bound, and seeing if they match. Unlike our analysis for most problems, this time we are going to count the exact number of comparisons involved and attempt to minimize this count. A simple algorithm for finding the second largest is to first find the maximum (in $n - 1$ comparisons), discard it, and then find the maximum of the remaining elements (in $n - 2$ comparisons) for a total cost of $2n - 3$ comparisons. Is this optimal? That seems doubtful, but let us now proceed to the step of attempting to prove a lower bound.

Theorem 15.2 *The lower bound for finding the second largest value is $2n - 3$.*

Proof: Any element that loses to anything other than the maximum cannot be second. So, the only candidates for second place are those that lost to the maximum. Function **largest** might compare the maximum element to $n - 1$ others. Thus, we might need $n - 2$ additional comparisons to find the second largest. \square

This proof is wrong. It exhibits the **necessity fallacy**: “Our algorithm does something, therefore all algorithms solving the problem must do the same.”

This leaves us with our best lower bounds argument at the moment being that finding the second largest must cost at least as much as finding the largest, or $n - 1$. Let us take another try at finding a better algorithm by adopting a strategy of divide and conquer. What if we break the list into halves, and run **largest** on each half? We can then compare the two winners (we have now used a total of $n - 1$ comparisons), and remove the winner from its half. Another call to **largest** on the winner’s half yields its second best. A final comparison against the winner of the other half gives us the true second place winner. The total cost is $\lceil 3n/2 \rceil - 2$. Is this optimal? What if we break the list into four pieces? The best would be $\lceil 5n/4 \rceil$. What if we break the list into eight pieces? Then the cost would be about $\lceil 9n/8 \rceil$. Notice that as we break the list into more parts, comparisons among the winners of the parts becomes a larger concern.

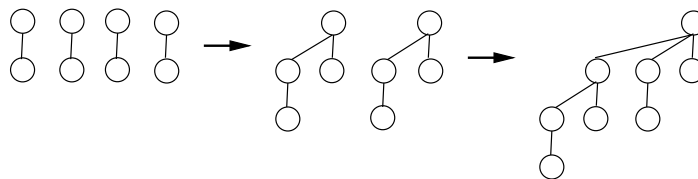


Figure 15.2 An example of building a binomial tree. Pairs of elements are combined by choosing one of the parents to be the root of the entire tree. Given two trees of size four, one of the roots is chosen to be the root for the combined tree of eight nodes.

Looking at this another way, the only candidates for second place are losers to the eventual winner, and our goal is to have as few of these as possible. So we need to keep track of the set of elements that have lost in direct comparison to the (eventual) winner. We also observe that we learn the most from a comparison when both competitors are known to be larger than the same number of other values. So we would like to arrange our comparisons to be against “equally strong” competitors. We can do all of this with a **binomial tree**. A binomial tree of height m has 2^m nodes. Either it is a single node (if $m = 0$), or else it is two height $m - 1$ binomial trees with one tree’s root becoming a child of the other. Figure 15.2 illustrates how a binomial tree with eight nodes would be constructed.

The resulting algorithm is simple in principle: Build the binomial tree for all n elements, and then compare the $\lceil \log n \rceil$ children of the root to find second place. We could store the binomial tree as an explicit tree structure, and easily build it in time linear on the number of comparisons as each comparison requires one link be added. Because the shape of a binomial tree is heavily constrained, we can also store the binomial tree implicitly in an array, much as we do for a heap. Assume that two trees, each with 2^k nodes, are in the array. The first tree is in positions 1 to 2^k . The second tree is in positions $2^k + 1$ to 2^{k+1} . The root of each subtree is in the final array position for that subtree.

To join two trees, we simply compare the roots of the subtrees. If necessary, swap the subtrees so that tree with the the larger root element becomes the second subtree. This trades space (we only need space for the data values, no node pointers) for time (in the worst case, all of the data swapping might cost $O(n \log n)$, though this does not affect the number of comparisons required). Note that for some applications, this is an important observation that the array’s data swapping requires no comparisons. If a comparison is simply a check between two integers, then of course moving half the values within the array is too expensive. But if a comparison requires that a competition be held between two sports teams, then the cost of a little bit (or even a lot) of book keeping becomes irrelevant.

Because the binomial tree’s root has $\log n$ children, and building the tree requires $n - 1$ comparisons, the number of comparisons required by this algorithm is $n + \lceil \log n \rceil - 2$. This is clearly better than our previous algorithm. Is it optimal?

We now go back to trying to improve the lower bounds proof. To do this, we introduce the concept of an **adversary**. The adversary's job is to make an algorithm's cost as high as possible. Imagine that the adversary keeps a list of all possible inputs. We view the algorithm as asking the adversary for information about the algorithm's input. The adversary may never lie, in that its answer must be consistent with the previous answers. But it is permitted to "rearrange" the input as it sees fit in order to drive the total cost for the algorithm as high as possible. In particular, when the algorithm asks a question, the adversary must answer in a way that is consistent with at least one remaining input. The adversary then crosses out all remaining inputs inconsistent with that answer. Keep in mind that there is not really an entity within the computer program that is the adversary, and we don't actually modify the program. The adversary operates merely as an analysis device, to help us reason about the program.

As an example of the adversary concept, consider the standard game of Hangman. Player *A* picks a word and tells player *B* how many letters the word has. Player *B* guesses various letters. If *B* guesses a letter in the word, then *A* will indicate which position(s) in the word have the letter. Player *B* is permitted to make only so many guesses of letters not in the word before losing.

In the Hangman game example, the adversary is imagined to hold a dictionary of words of some selected length. Each time the player guesses a letter, the adversary consults the dictionary and decides if more words will be eliminated by accepting the letter (and indicating which positions it holds) or saying that its not in the word. The adversary can make any decision it chooses, so long as at least one word in the dictionary is consistent with all of the decisions. In this way, the adversary can hope to make the player guess as many letters as possible.

Before explaining how the adversary plays a role in our lower bounds proof, first observe that at least $n - 1$ values must lose at least once. This requires at least $n - 1$ compares. In addition, at least $k - 1$ values must lose to the second largest value. That is, k direct losers to the winner must be compared. There must be at least $n + k - 2$ comparisons. The question is: How low can we make k ?

Call the **strength** of element $A[i]$ the number of elements that $A[i]$ is (known to be) bigger than. If $A[i]$ has strength a , and $A[j]$ has strength b , then the winner has strength $a + b + 1$. The algorithm gets to know the (current) strengths for each element, and it gets to pick which two elements are compared next. The adversary gets to decide who wins any given comparison. What strategy by the adversary would cause the algorithm to learn the least from any given comparison? It should minimize the rate at which any element improves its strength. It can do this by making the element with the greater strength win at every comparison. This is a "fair" use of an adversary in that it represents the results of providing a worst-case input for that given algorithm.

To minimize the effects of worst-case behavior, the algorithm's best strategy is to maximize the minimum improvement in strength by balancing the strengths of any two competitors. From the algorithm's point of view, the best outcome is that an element doubles in strength. This happens whenever $a = b$, where a and b are the strengths of the two elements being compared. All strengths begin at zero, so the winner must make at least k comparisons when $2^{k-1} < n \leq 2^k$. Thus, there must be at least $n + \lceil \log n \rceil - 2$ comparisons. So our algorithm is optimal.

15.5 State Space Lower Bounds Proofs

We now consider the problem of finding both the minimum and the maximum from an (unsorted) list of values. This might be useful if we want to know the range of a collection of values to be plotted, for the purpose of drawing the plot's scales. Of course we could find them independently in $2n - 2$ comparisons. A slight modification is to find the maximum in $n - 1$ comparisons, remove it from the list, and then find the minimum in $n - 2$ further comparisons for a total of $2n - 3$ comparisons. Can we do better than this?

Before continuing, think a moment about how this problem of finding the minimum and the maximum compares to the problem of the last section, that of finding the second biggest value (and by implication, the maximum). Which of these two problems do you think is harder? It is probably not at all obvious to you that one problem is harder or easier than the other. There is intuition that argues for either case. On the one hand intuition might argue that the process of finding the maximum should tell you something about the second biggest value, more than that process should tell you about the minimum value. On the other hand, any given comparison tells you something about which of two can be a candidate for maximum value, and which can be a candidate for minimum value, thus making progress in both directions.

We will start by considering a simple divide-and-conquer approach to finding the minimum and maximum. Split the list into two parts and find the minimum and maximum elements in each part. Then compare the two minimums and maximums to each other with a further two comparisons to get the final result. The algorithm is shown in Figure 15.3.

The cost of this algorithm can be modeled by the following recurrence.

$$\mathbf{T}(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ \mathbf{T}(\lfloor n/2 \rfloor) + \mathbf{T}(\lceil n/2 \rceil) + 2 & n > 2 \end{cases}$$

This is a rather interesting recurrence, and its solution ranges between $3n/2 - 2$ (when $n = 2^i$ or $n = 2^1 \pm 1$) and $5n/3 - 2$ (when $n = 3 \times 2^i$). We can infer from this behavior that how we divide the list affects the performance of the algorithm.

```

// Return the minimum and maximum values in A
// between positions l and r
template <typename E>
void MinMax(E A[], int l, int r, E& Min, E& Max) {
    if (l == r) { // n=1
        Min = A[l];
        Max = A[l];
    }
    else if (l+1 == r) { // n=2
        Min = min(A[l], A[r]);
        Max = max(A[l], A[r]);
    }
    else { // n>2
        int Min1, Min2, Max1, Max2;
        int mid = (l + r)/2;
        MinMax(A, l, mid, Min1, Max1);
        MinMax(A, mid+1, r, Min2, Max2);
        Min = min(Min1, Min2);
        Max = max(Max1, Max2);
    }
}

```

Figure 15.3 Recursive algorithm for finding the minimum and maximum values in an array.

For example, what if we have six items in the list? If we break the list into two sublists of three elements, the cost would be 8. If we break the list into a sublist of size two and another of size four, then the cost would only be 7.

With divide and conquer, the best algorithm is the one that minimizes the work, not necessarily the one that balances the input sizes. One lesson to learn from this example is that it can be important to pay attention to what happens for small sizes of n , because any division of the list will eventually produce many small lists.

We can model all possible divide-and-conquer strategies for this problem with the following recurrence.

$$T(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ \min_{1 \leq k \leq n-1} \{T(k) + T(n-k)\} + 2 & n > 2 \end{cases}$$

That is, we want to find a way to break up the list that will minimize the total work. If we examine various ways of breaking up small lists, we will eventually recognize that breaking the list into a sublist of size 2 and a sublist of size $n-2$ will always produce results as good as any other division. This strategy yields the following recurrence.

$$T(n) = \begin{cases} 0 & n = 1 \\ 1 & n = 2 \\ T(n-2) + 3 & n > 2 \end{cases}$$

This recurrence (and the corresponding algorithm) yields $\mathbf{T}(n) = \lceil 3n/2 \rceil - 2$ comparisons. Is this optimal? We now introduce yet another tool to our collection of lower bounds proof techniques: The state space proof.

We will model our algorithm by defining a **state** that the algorithm must be in at any given instant. We can then define the start state, the end state, and the transitions between states that any algorithm can support. From this, we will reason about the minimum number of states that the algorithm must go through to get from the start to the end, to reach a state space lower bound.

At any given instant, we can track the following four categories of elements:

- Untested: Elements that have not been tested.
- Winners: Elements that have won at least once, and never lost.
- Losers: Elements that have lost at least once, and never won.
- Middle: Elements that have both won and lost at least once.

We define the current state to be a vector of four values, (U, W, L, M) for untested, winners, losers, and middles, respectively. For a set of n elements, the initial state of the algorithm is $(n, 0, 0, 0)$ and the end state is $(0, 1, 1, n - 2)$. Thus, every run for any algorithm must go from state $(n, 0, 0, 0)$ to state $(0, 1, 1, n - 2)$. We also observe that once an element is identified to be a middle, it can then be ignored because it can neither be the minimum nor the maximum.

Given that there are four types of elements, there are 10 types of comparison. Comparing with a middle cannot be more efficient than other comparisons, so we should ignore those, leaving six comparisons of interest. We can enumerate the effects of each comparison type as follows. If we are in state (i, j, k, l) and we have a comparison, then the state changes are as follows.

$$\begin{array}{ll}
 U : U & (i - 2, \quad j + 1, \quad k + 1, \quad l) \\
 W : W & (i, \quad j - 1, \quad k, \quad l + 1) \\
 L : L & (i, \quad j, \quad k - 1, \quad l + 1) \\
 L : U & (i - 1, \quad j + 1, \quad k, \quad l) \\
 & \text{or} \quad (i - 1, \quad j, \quad k, \quad l + 1) \\
 W : U & (i - 1, \quad j, \quad k + 1, \quad l) \\
 & \text{or} \quad (i - 1, \quad j, \quad k, \quad l + 1) \\
 W : L & (i, \quad j, \quad k, \quad l) \\
 & \text{or} \quad (i, \quad j - 1, \quad k - 1, \quad l + 2)
 \end{array}$$

Now, let us consider what an adversary will do for the various comparisons. The adversary will make sure that each comparison does the least possible amount of work in taking the algorithm toward the goal state. For example, comparing a winner to a loser is of no value because the worst case result is always to learn nothing new (the winner remains a winner and the loser remains a loser). Thus, only the following five transitions are of interest:

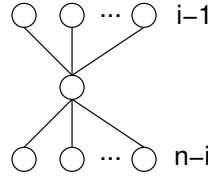


Figure 15.4 The poset that represents the minimum information necessary to determine the i th element in a list. We need to know which element has $i - 1$ values less and $n - i$ values more, but we do not need to know the relationships among the elements with values less or greater than the i th element.

$$\begin{array}{lcl}
 U : U & (i - 2, & j + 1, \quad k + 1, \quad l) \\
 L : U & (i - 1, & j + 1, \quad k, \quad l) \\
 W : U & (i - 1, & j, \quad k + 1, \quad l) \\
 \hline
 W : W & (i, & j - 1, \quad k, \quad l + 1) \\
 L : L & (i, & j, \quad k - 1, \quad l + 1)
 \end{array}$$

Only the last two transition types increase the number of middles, so there must be $n - 2$ of these. The number of untested elements must go to 0, and the first transition is the most efficient way to do this. Thus, $\lceil n/2 \rceil$ of these are required. Our conclusion is that the minimum possible number of transitions (comparisons) is $n + \lceil n/2 \rceil - 2$. Thus, our algorithm is optimal.

15.6 Finding the i th Best Element

We now tackle the problem of finding the i th best element in a list. As observed earlier, one solution is to sort the list and simply look in the i th position. However, this process provides considerably more information than we need to solve the problem. The minimum amount of information that we actually need to know can be visualized as shown in Figure 15.4. That is, all we need to know is the $i - 1$ items less than our desired value, and the $n - i$ items greater. We do not care about the relative order within the upper and lower groups. So can we find the required information faster than by first sorting? Looking at the lower bound, can we tighten that beyond the trivial lower bound of n comparisons? We will focus on the specific question of finding the median element (i.e., the element with rank $n/2$), because the resulting algorithm can easily be modified to find the i th largest value for any i .

Looking at the Quicksort algorithm might give us some insight into solving the median problem. Recall that Quicksort works by selecting a pivot value, partitioning the array into those elements less than the pivot and those greater than the pivot, and moving the pivot to its proper location in the array. If the pivot is in position i , then we are done. If not, we can solve the subproblem recursively by only considering one of the sublists. That is, if the pivot ends up in position $k > i$, then we

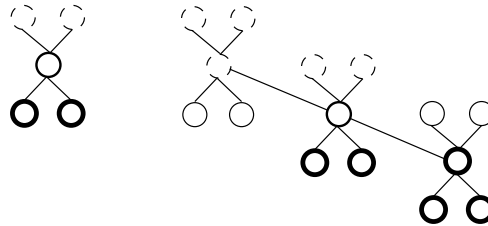


Figure 15.5 A method for finding a pivot for partitioning a list that guarantees at least a fixed fraction of the list will be in each partition. We divide the list into groups of five elements, and find the median for each group. We then recursively find the median of these $n/5$ medians. The median of five elements is guaranteed to have at least two in each partition. The median of three medians from a collection of 15 elements is guaranteed to have at least five elements in each partition.

simply solve by finding the i th best element in the left partition. If the pivot is at position $k < i$, then we wish to find the $i - k$ th element in the right partition.

What is the worst case cost of this algorithm? As with Quicksort, we get bad performance if the pivot is the first or last element in the array. This would lead to possibly $O(n^2)$ performance. However, if the pivot were to always cut the array in half, then our cost would be modeled by the recurrence $T(n) = T(n/2) + n = 2n$ or $O(n)$ cost.

Finding the average cost requires us to use a recurrence with full history, similar to the one we used to model the cost of Quicksort. If we do this, we will find that $T(n)$ is in $O(n)$ in the average case.

Is it possible to modify our algorithm to get worst-case linear time? To do this, we need to pick a pivot that is guaranteed to discard a fixed fraction of the elements. We cannot just choose a pivot at random, because doing so will not meet this guarantee. The ideal situation would be if we could pick the median value for the pivot each time. But that is essentially the same problem that we are trying to solve to begin with.

Notice, however, that if we choose any constant c , and then if we pick the median from a sample of size n/c , then we can guarantee that we will discard at least $n/2c$ elements. Actually, we can do better than this by selecting small subsets of a constant size (so we can find the median of each in constant time), and then taking the median of these medians. Figure 15.5 illustrates this idea. This observation leads directly to the following algorithm.

- Choose the $n/5$ medians for groups of five elements from the list. Choosing the median of five items can be done in constant time.
- Recursively, select M , the median of the $n/5$ medians-of-fives.
- Partition the list into those elements larger and smaller than M .

While selecting the median in this way is guaranteed to eliminate a fraction of the elements (leaving at most $\lceil (7n-5)/10 \rceil$ elements left), we still need to be sure that our recursion yields a linear-time algorithm. We model the algorithm by the following recurrence.

$$\mathbf{T}(n) \leq \mathbf{T}(\lceil n/5 \rceil) + \mathbf{T}(\lceil (7n-5)/10 \rceil) + 6\lceil n/5 \rceil + n - 1.$$

The $\mathbf{T}(\lceil n/5 \rceil)$ term comes from computing the median of the medians-of-fives, the $6\lceil n/5 \rceil$ term comes from the cost to calculate the median-of-fives (exactly six comparisons for each group of five element), and the $\mathbf{T}(\lceil (7n-5)/10 \rceil)$ term comes from the recursive call of the remaining (up to) 70% of the elements that might be left.

We will prove that this recurrence is linear by assuming that it is true for some constant r , and then show that $\mathbf{T}(n) \leq rn$ for all n greater than some bound.

$$\begin{aligned} \mathbf{T}(n) &\leq \mathbf{T}(\lceil \frac{n}{5} \rceil) + \mathbf{T}(\lceil \frac{7n-5}{10} \rceil) + 6\lceil \frac{n}{5} \rceil + n - 1 \\ &\leq r(\frac{n}{5} + 1) + r(\frac{7n-5}{10} + 1) + 6(\frac{n}{5} + 1) + n - 1 \\ &\leq (\frac{r}{5} + \frac{7r}{10} + \frac{11}{5})n + \frac{3r}{2} + 5 \\ &\leq \frac{9r+22}{10}n + \frac{3r+10}{2}. \end{aligned}$$

This is true for $r \geq 23$ and $n \geq 380$. This provides a base case that allows us to use induction to prove that $\forall n \geq 380, \mathbf{T}(n) \leq 23n$.

In reality, this algorithm is not practical because its constant factor costs are so high. So much work is being done to guarantee linear time performance that it is more efficient on average to rely on chance to select the pivot, perhaps by picking it at random or picking the middle value out of the current subarray.

15.7 Optimal Sorting

We conclude this section with an effort to find the sorting algorithm with the absolute fewest possible comparisons. It might well be that the result will not be practical for a general-purpose sorting algorithm. But recall our analogy earlier to sports tournaments. In sports, a “comparison” between two teams or individuals means doing a competition between the two. This is fairly expensive (at least compared to some minor book keeping in a computer), and it might be worth trading a fair amount of book keeping to cut down on the number of games that need to be played. What if we want to figure out how to hold a tournament that will give us the exact ordering for all teams in the fewest number of total games? Of course, we are assuming that the results of each game will be “accurate” in that we assume

not only that the outcome of A playing B would always be the same (at least over the time period of the tournament), but that transitivity in the results also holds. In practice these are unrealistic assumptions, but such assumptions are implicitly part of many tournament organizations. Like most tournament organizers, we can simply accept these assumptions and come up with an algorithm for playing the games that gives us some rank ordering based on the results we obtain.

Recall Insertion Sort, where we put element i into a sorted sublist of the first $i - 1$ elements. What if we modify the standard Insertion Sort algorithm to use binary search to locate where the i th element goes in the sorted sublist? This algorithm is called **binary insert sort**. As a general-purpose sorting algorithm, this is not practical because we then have to (on average) move about $i/2$ elements to make room for the newly inserted element in the sorted sublist. But if we count *only* comparisons, binary insert sort is pretty good. And we can use some ideas from binary insert sort to get closer to an algorithm that uses the absolute minimum number of comparisons needed to sort.

Consider what happens when we run binary insert sort on five elements. How many comparisons do we need to do? We can insert the second element with one comparison, the third with two comparisons, and the fourth with 2 comparisons. When we insert the fifth element into the sorted list of four elements, we need to do three comparisons in the worst case. Notice exactly what happens when we attempt to do this insertion. We compare the fifth element against the second. If the fifth is bigger, we have to compare it against the third, and if it is bigger we have to compare it against the fourth. In general, when is binary search most efficient? When we have $2^i - 1$ elements in the list. It is least efficient when we have 2^i elements in the list. So, we can do a bit better if we arrange our insertions to avoid inserting an element into a list of size 2^i if possible.

Figure 15.6 illustrates a different organization for the comparisons that we might do. First we compare the first and second element, and the third and fourth elements. The two winners are then compared, yielding a binomial tree. We can view this as a (sorted) chain of three elements, with element A hanging off from the root. If we then insert element B into the sorted chain of three elements, we will end up with one of the two posets shown on the right side of Figure 15.6, at a cost of 2 comparisons. We can then merge A into the chain, for a cost of two comparisons (because we already know that it is smaller than either one or two elements, we are actually merging it into a list of two or three elements). Thus, the total number of comparisons needed to sort the five elements is at most seven instead of eight.

If we have ten elements to sort, we can first make five pairs of elements (using five compares) and then sort the five winners using the algorithm just described (using seven more compares). Now all we need to do is to deal with the original losers. We can generalize this process for any number of elements as:

- Pair up all the nodes with $\lfloor \frac{n}{2} \rfloor$ comparisons.

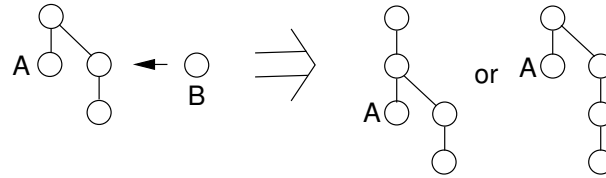


Figure 15.6 Organizing comparisons for sorting five elements. First we order two pairs of elements, and then compare the two winners to form a binomial tree of four elements. The original loser to the root is labeled A, and the remaining three elements form a sorted chain. We then insert element B into the sorted chain. Finally, we put A into the resulting chain to yield a final sorted list.

- Recursively sort the winners.
- Fold in the losers.

We use binary insert to place the losers. However, we are free to choose the best ordering for inserting, keeping in mind the fact that binary search has the same cost for 2^i through $2^{i+1} - 1$ items. For example, binary search requires three comparisons in the worst case for lists of size 4, 5, 6, or 7. So we pick the order of inserts to optimize the binary searches, which means picking an order that avoids growing a sublist size such that it crosses the boundary on list size to require an additional comparison. This sort is called **merge insert sort**, and also known as the Ford and Johnson sort.

For ten elements, given the poset shown in Figure 15.7 we fold in the last four elements (labeled 1 to 4) in the order Element 3, Element 4, Element 1, and finally Element 2. Element 3 will be inserted into a list of size three, costing two comparisons. Depending on where Element 3 then ends up in the list, Element 4 will now be inserted into a list of size 2 or 3, costing two comparisons in either case. Depending on where Elements 3 and 4 are in the list, Element 1 will now be inserted into a list of size 5, 6, or 7, all of which requires three comparisons to place in sort order. Finally, Element 2 will be inserted into a list of size 5, 6, or 7.

Merge insert sort is pretty good, but is it optimal? Recall from Section 7.9 that no sorting algorithm can be faster than $\Omega(n \log n)$. To be precise, the **information theoretic lower bound** for sorting can be proved to be $\lceil \log n! \rceil$. That is, we can prove a lower bound of exactly $\lceil \log n! \rceil$ comparisons. Merge insert sort gives us a number of comparisons equal to this information theoretic lower bound for all values up to $n = 12$. At $n = 12$, merge insert sort requires 30 comparisons while the information theoretic lower bound is only 29 comparisons. However, for such a small number of elements, it is possible to do an exhaustive study of every possible arrangement of comparisons. It turns out that there is in fact no possible arrangement of comparisons that makes the lower bound less than 30 comparisons when $n = 12$. Thus, the information theoretic lower bound is an underestimate in this case, because 30 really is the best that can be done.

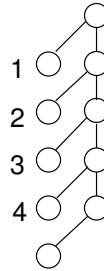


Figure 15.7 Merge insert sort for ten elements. First five pairs of elements are compared. The five winners are then sorted. This leaves the elements labeled 1-4 to be sorted into the chain made by the remaining six elements.

Call the optimal worst cost for n elements $S(n)$. We know that $S(n+1) \leq S(n) + \lceil \log(n+1) \rceil$ because we could sort n elements and use binary insert for the last one. For all n and m , $S(n+m) \leq S(n) + S(m) + M(m, n)$ where $M(m, n)$ is the best time to merge two sorted lists. For $n = 47$, it turns out that we can do better by splitting the list into pieces of size 5 and 42, and then merging. Thus, merge sort is not quite optimal. But it is extremely good, and nearly optimal for smallish numbers of elements.

15.8 Further Reading

Much of the material in this book is also covered in many other textbooks on data structures and algorithms. The biggest exception is that not many other textbooks cover lower bounds proofs in any significant detail, as is done in this chapter. Those that do focus on the same example problems (search and selection) because it tells such a tight and compelling story regarding related topics, while showing off the major techniques for lower bounds proofs. Two examples of such textbooks are “Computer Algorithms” by Baase and Van Gelder [BG00], and “Compared to What?” by Gregory J.E. Rawlins [Raw92]. “Fundamentals of Algorithmics” by Brassard and Bratley [BB96] also covers lower bounds proofs.

15.9 Exercises

- 15.1** Consider the so-called “algorithm for algorithms” in Section 15.1. Is this really an algorithm? Review the definition of an algorithm from Section 1.4. Which parts of the definition apply, and which do not? Is the “algorithm for algorithms” a heuristic for finding a good algorithm? Why or why not?
- 15.2** Single-elimination tournaments are notorious for their scheduling difficulties. Imagine that you are organizing a tournament for n basketball teams (you may assume that $n = 2^i$ for some integer i). We will further simplify

things by assuming that each game takes less than an hour, and that each team can be scheduled for a game every hour if necessary. (Note that everything said here about basketball courts is also true about processors in a parallel algorithm to solve the maximum-finding problem).

- (a) How many basketball courts do we need to insure that every team can play whenever we want to minimize the total tournament time?
 - (b) How long will the tournament be in this case?
 - (c) What is the total number of “court-hours” available? How many total hours are courts being used? How many total court-hours are unused?
 - (d) Modify the algorithm in such a way as to reduce the total number of courts needed, by perhaps not letting every team play whenever possible. This will increase the total hours of the tournament, but try to keep the increase as low as possible. For your new algorithm, how long is the tournament, how many courts are needed, how many total court-hours are available, how many court-hours are used, and how many unused?
- 15.3** Explain why the cost of splitting a list of six into two lists of three to find the minimum and maximum elements requires eight comparisons, while splitting the list into a list of two and a list of four costs only seven comparisons.
- 15.4** Write out a table showing the number of comparisons required to find the minimum and maximum for all divisions for all values of $n \leq 13$.
- 15.5** Present an adversary argument as a lower bounds proof to show that $n - 1$ comparisons are necessary to find the maximum of n values in the worst case.
- 15.6** Present an adversary argument as a lower bounds proof to show that n comparisons are necessary in the worst case when searching for an element with value X (if one exists) from among n elements.
- 15.7** Section 15.6 claims that by picking a pivot that always discards at least a fixed fraction c of the remaining array, the resulting algorithm will be linear. Explain why this is true. Hint: The Master Theorem (Theorem 14.1) might help you.
- 15.8** Show that any comparison-based algorithm for finding the median must use at least $n - 1$ comparisons.
- 15.9** Show that any comparison-based algorithm for finding the second-smallest of n values can be extended to find the smallest value also, without requiring any more comparisons to be performed.
- 15.10** Show that any comparison-based algorithm for sorting can be modified to remove all duplicates without requiring any more comparisons to be performed.
- 15.11** Show that any comparison-based algorithm for removing duplicates from a list of values must use $\Omega(n \log n)$ comparisons.
- 15.12** Given a list of n elements, an element of the list is a *majority* if it appears more than $n/2$ times.

- (a) Assume that the input is a list of integers. Design an algorithm that is linear in the number of integer-integer comparisons in the worst case that will find and report the majority if one exists, and report that there is no majority if no such integer exists in the list.
 - (b) Assume that the input is a list of elements that have no relative ordering, such as colors or fruit. So all that you can do when you compare two elements is ask if they are the same or not. Design an algorithm that is linear in the number of element-element comparisons in the worst case that will find a majority if one exists, and report that there is no majority if no such element exists in the list.
- 15.13** Given an undirected graph G , the problem is to determine whether or not G is connected. Use an adversary argument to prove that it is necessary to look at all $(n^2 - n)/2$ potential edges in the worst case.
- 15.14** (a) Write an equation that describes the average cost for finding the median.
(b) Solve your equation from part (a).
- 15.15** (a) Write an equation that describes the average cost for finding the i th-smallest value in an array. This will be a function of both n and i , $T(n, i)$.
(b) Solve your equation from part (a).
- 15.16** Suppose that you have n objects that have identical weight, except for one that is a bit heavier than the others. You have a balance scale. You can place objects on each side of the scale and see which collection is heavier. Your goal is to find the heavier object, with the minimum number of weighings. Find and prove matching upper and lower bounds for this problem.
- 15.17** Imagine that you are organizing a basketball tournament for 10 teams. You know that the merge insert sort will give you a full ranking of the 10 teams with the minimum number of games played. Assume that each game can be played in less than an hour, and that any team can play as many games in a row as necessary. Show a schedule for this tournament that also attempts to minimize the number of total hours for the tournament and the number of courts used. If you have to make a tradeoff between the two, then attempt to minimize the total number of hours that basketball courts are idle.
- 15.18** Write the complete algorithm for the merge insert sort sketched out in Section 15.7.
- 15.19** Here is a suggestion for what might be a truly optimal sorting algorithm. Pick the best set of comparisons for input lists of size 2. Then pick the best set of comparisons for size 3, size 4, size 5, and so on. Combine them together into one program with a big case statement. Is this an algorithm?

15.10 Projects

- 15.1** Implement the median-finding algorithm of Section 15.6. Then, modify this algorithm to allow finding the i th element for any value $i < n$.

Patterns of Algorithms

This chapter presents several fundamental topics related to the theory of algorithms. Included are dynamic programming (Section 16.1), randomized algorithms (Section 16.2), and the concept of a transform (Section 16.3.5). Each of these can be viewed as an example of an “algorithmic pattern” that is commonly used for a wide variety of applications. In addition, Section 16.3 presents a number of numerical algorithms. Section 16.2 on randomized algorithms includes the Skip List (Section 16.2.2). The Skip List is a probabilistic data structure that can be used to implement the dictionary ADT. The Skip List is no more complicated than the BST. Yet it often outperforms the BST because the Skip List’s efficiency is not tied to the values or insertion order of the dataset being stored.

16.1 Dynamic Programming

Consider again the recursive function for computing the n th Fibonacci number.

```
long fibr(int n) { // Recursive Fibonacci generator
    // fibr(46) is largest value that fits in a long
    Assert((n > 0) && (n < 47), "Input out of range");
    if ((n == 1) || (n == 2)) return 1; // Base cases
    return fibr(n-1) + fibr(n-2);      // Recursion
}
```

The cost of this algorithm (in terms of function calls) is the size of the n th Fibonacci number itself, which our analysis of Section 14.2 showed to be exponential (approximately $n^{1.62}$). Why is this so expensive? Primarily because two recursive calls are made by the function, and the work that they do is largely redundant. That is, each of the two calls is recomputing most of the series, as is each sub-call, and so on. Thus, the smaller values of the function are being recomputed a huge number of times. If we could eliminate this redundancy, the cost would be greatly reduced. The approach that we will use can also improve any algorithm that spends most of its time recomputing common subproblems.

One way to accomplish this goal is to keep a table of values, and first check the table to see if the computation can be avoided. Here is a straightforward example of doing so.

```
int fibrt(int n) {
    // Assume Values has at least n slots, and all
    // slots are initialized to 0
    if (n <= 2) return 1;           // Base case
    if (Values[n] == 0)
        Values[n] = fibrt(n-1) + fibrt(n-2);
    return Values[n];
}
```

This version of the algorithm will not compute a value more than once, so its cost should be linear. Of course, we didn't actually need to use a table storing all of the values, since future computations do not need access to all prior subproblems. Instead, we could build the value by working from 0 and 1 up to n rather than backwards from n down to 0 and 1. Going up from the bottom we only need to store the previous two values of the function, as is done by our iterative version.

```
long fibi(int n) { // Iterative Fibonacci generator
    // fibi(46) is largest value that fits in a long
    Assert((n > 0) && (n < 47), "Input out of range");
    long past, prev, curr; // Store temporary values
    past = prev = curr = 1; // initialize
    for (int i=3; i<=n; i++) { // Compute next value
        past = prev;           // past holds fibi(i-2)
        prev = curr;           // prev holds fibi(i-1)
        curr = past + prev;     // curr now holds fibi(i)
    }
    return curr;
}
```

Recomputing of subproblems comes up in many algorithms. It is not so common that we can store only a few prior results as we did for **fibi**. Thus, there are many times where storing a complete table of subresults will be useful.

This approach to designing an algorithm that works by storing a table of results for subproblems is called dynamic programming. The name is somewhat arcane, because it doesn't bear much obvious similarity to the process that is taking place when storing subproblems in a table. However, it comes originally from the field of dynamic control systems, which got its start before what we think of as computer programming. The act of storing precomputed values in a table for later reuse is referred to as "programming" in that field.

Dynamic programming is a powerful alternative to the standard principle of divide and conquer. In divide and conquer, a problem is split into subproblems, the subproblems are solved (independently), and then recombined into a solution for the problem being solved. Dynamic programming is appropriate whenever (1)

subproblems are solved repeatedly, and (2) we can find a suitable way of doing the necessary bookkeeping. Dynamic programming algorithms are usually not implemented by simply using a table to store subproblems for recursive calls (i.e., going backwards as is done by **fibrt**). Instead, such algorithms are typically implemented by building the table of subproblems from the bottom up. Thus, **fibi** better represents the most common form of dynamic programming than does **fibrt**, even though it doesn't use the complete table.

16.1.1 The Knapsack Problem

We will next consider a problem that appears with many variations in a variety of commercial settings. Many businesses need to package items with the greatest efficiency. One way to describe this basic idea is in terms of packing items into a knapsack, and so we will refer to this as the Knapsack Problem. We will first define a particular formulation of the knapsack problem, and then we will discuss an algorithm to solve it based on dynamic programming. We will see other versions of the knapsack problem in the exercises and in Chapter 17.

Assume that we have a knapsack with a certain amount of space that we will define using integer value K . We also have n items each with a certain size such that that item i has integer size k_i . The problem is to find a subset of the n items whose sizes exactly sum to K , if one exists. For example, if our knapsack has capacity $K = 5$ and the two items are of size $k_1 = 2$ and $k_2 = 4$, then no such subset exists. But if we add a third item of size $k_3 = 1$, then we can fill the knapsack exactly with the second and third items. We can define the problem more formally as: Find $S \subset \{1, 2, \dots, n\}$ such that

$$\sum_{i \in S} k_i = K.$$

Example 16.1 Assume that we are given a knapsack of size $K = 163$ and 10 items of sizes 4, 9, 15, 19, 27, 44, 54, 68, 73, 101. Can we find a subset of the items that exactly fills the knapsack? You should take a few minutes and try to do this before reading on and looking at the answer.

One solution to the problem is: 19, 27, 44, 73.

Example 16.2 Having solved the previous example for knapsack of size 163, how hard is it now to solve for a knapsack of size 164?

Unfortunately, knowing the answer for 163 is of almost no use at all when solving for 164. One solution is: 9, 54, 101.

If you tried solving these examples, you probably found yourself doing a lot of trial-and-error and a lot of backtracking. To come up with an algorithm, we want an organized way to go through the possible subsets. Is there a way to make the problem smaller, so that we can apply divide and conquer? We essentially have two parts to the input: The knapsack size K and the n items. It probably will not do us much good to try and break the knapsack into pieces and solve the sub-pieces (since we already saw that knowing the answer for a knapsack of size 163 did nothing to help us solve the problem for a knapsack of size 164).

So, what can we say about solving the problem with or without the n th item? This seems to lead to a way to break down the problem. If the n th item is not needed for a solution (that is, if we can solve the problem with the first $n - 1$ items) then we can also solve the problem when the n th item is available (we just ignore it). On the other hand, if we do include the n th item as a member of the solution subset, then we now would need to solve the problem with the first $n - 1$ items and a knapsack of size $K - k_n$ (since the n th item is taking up k_n space in the knapsack).

To organize this process, we can define the problem in terms of two parameters: the knapsack size K and the number of items n . Denote a given instance of the problem as $P(n, K)$. Now we can say that $P(n, K)$ has a solution if and only if there exists a solution for either $P(n - 1, K)$ or $P(n - 1, K - k_n)$. That is, we can solve $P(n, K)$ only if we can solve one of the sub problems where we use or do not use the n th item. Of course, the ordering of the items is arbitrary. We just need to give them some order to keep things straight.

Continuing this idea, to solve any subproblem of size $n - 1$, we need only to solve two subproblems of size $n - 2$. And so on, until we are down to only one item that either fills the knapsack or not. This naturally leads to a cost expressed by the recurrence relation $T(n) = 2T(n - 1) + c = \Theta(2^n)$. That can be pretty expensive!

But... we should quickly realize that there are only $n(K + 1)$ subproblems to solve! Clearly, there is the possibility that many subproblems are being solved repeatedly. This is a natural opportunity to apply dynamic programming. We simply build an array of size $n \times K + 1$ to contain the solutions for all subproblems $P(i, k)$, $1 \leq i \leq n$, $0 \leq k \leq K$.

There are two approaches to actually solving the problem. One is to start with our problem of size $P(n, K)$ and make recursive calls to solve the subproblems, each time checking the array to see if a subproblem has been solved, and filling in the corresponding cell in the array whenever we get a new subproblem solution. The other is to start filling the array for row 1 (which indicates a successful solution only for a knapsack of size k_1). We then fill in the succeeding rows from $i = 2$ to n , left to right, as follows.

if $P(n - 1, K)$ has a solution,

then $P(n, K)$ has a solution
else if $P(n - 1, K - k_n)$ has a solution
 then $P(n, K)$ has a solution
 else $P(n, K)$ has no solution.

In other words, a new slot in the array gets its solution by looking at two slots in the preceding row. Since filling each slot in the array takes constant time, the total cost of the algorithm is $\Theta(nK)$.

Example 16.3 Solve the Knapsack Problem for $K = 10$ and five items with sizes 9, 2, 7, 4, 1. We do this by building the following array.

	0	1	2	3	4	5	6	7	8	9	10
$k_1=9$	O	—	—	—	—	—	—	—	—	I	—
$k_2=2$	O	—	I	—	—	—	—	—	—	O	—
$k_3=7$	O	—	O	—	—	—	—	I	—	I/O	—
$k_4=4$	O	—	O	—	I	—	I	O	—	O	—
$k_5=1$	O	I	O	I	O	I	O	I/O	I	O	I

Key:

- : No solution for $P(i, k)$.
- O: Solution(s) for $P(i, k)$ with i omitted.
- I: Solution(s) for $P(i, k)$ with i included.
- I/O: Solutions for $P(i, k)$ with i included AND omitted.

For example, $P(3, 9)$ stores value I/O. It contains O because $P(2, 9)$ has a solution. It contains I because $P(2, 2) = P(2, 9 - 7)$ has a solution. Since $P(5, 10)$ is marked with an I, it has a solution. We can determine what that solution actually is by recognizing that it includes the 5th item (of size 1), which then leads us to look at the solution for $P(4, 9)$. This in turn has a solution that omits the 4th item, leading us to $P(3, 9)$. At this point, we can either use the third item or not. We can find a solution by taking one branch. We can find all solutions by following all branches when there is a choice.

16.1.2 All-Pairs Shortest Paths

We next consider the problem of finding the shortest distance between all pairs of vertices in the graph, called the **all-pairs shortest-paths** problem. To be precise, for every $u, v \in \mathbf{V}$, calculate $d(u, v)$.

One solution is to run Dijkstra's algorithm for finding the single-source shortest path (see Section 11.4.1) $|\mathbf{V}|$ times, each time computing the shortest path from a different start vertex. If \mathbf{G} is sparse (that is, $|\mathbf{E}| = \Theta(|\mathbf{V}|)$) then this is a good

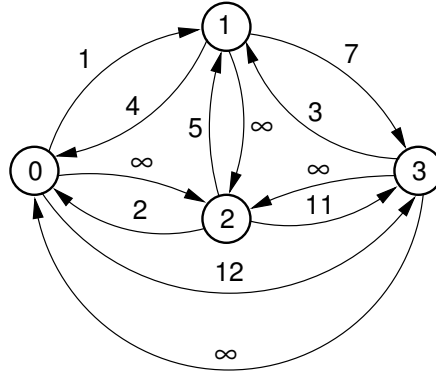


Figure 16.1 An example of k -paths in Floyd's algorithm. Path 1, 3 is a 0-path by definition. Path 3, 0, 2 is not a 0-path, but it is a 1-path (as well as a 2-path, a 3-path, and a 4-path) because the largest intermediate vertex is 0. Path 1, 3, 2 is a 4-path, but not a 3-path because the intermediate vertex is 3. All paths in this graph are 4-paths.

solution, because the total cost will be $\Theta(|V|^2 + |V||E| \log |V|) = \Theta(|V|^2 \log |V|)$ for the version of Dijkstra's algorithm based on priority queues. For a dense graph, the priority queue version of Dijkstra's algorithm yields a cost of $\Theta(|V|^3 \log |V|)$, but the version using **MinVertex** yields a cost of $\Theta(|V|^3)$.

Another solution that limits processing time to $\Theta(|V|^3)$ regardless of the number of edges is known as Floyd's algorithm. It is an example of dynamic programming. The chief problem with solving this problem is organizing the search process so that we do not repeatedly solve the same subproblems. We will do this organization through the use of the k -path. Define a **k -path** from vertex v to vertex u to be any path whose intermediate vertices (aside from v and u) all have indices less than k . A 0-path is defined to be a direct edge from v to u . Figure 16.1 illustrates the concept of k -paths.

Define $D_k(v, u)$ to be the length of the shortest k -path from vertex v to vertex u . Assume that we already know the shortest k -path from v to u . The shortest $(k+1)$ -path either goes through vertex k or it does not. If it does go through k , then the best path is the best k -path from v to k followed by the best k -path from k to u . Otherwise, we should keep the best k -path seen before. Floyd's algorithm simply checks all of the possibilities in a triple loop. Here is the implementation for Floyd's algorithm. At the end of the algorithm, array **D** stores the all-pairs shortest distances.


```
// Floyd's all-pairs shortest paths algorithm
// Store the pair-wise distances in "D"
void Floyd(Graph* G) {
    for (int i=0; i<G->n(); i++) // Initialize D with weights
        for (int j=0; j<G->n(); j++)
            if (G->weight(i, j) != 0) D[i][j] = G->weight(i, j);
    for (int k=0; k<G->n(); k++) // Compute all k paths
        for (int i=0; i<G->n(); i++)
            for (int j=0; j<G->n(); j++)
                if (D[i][j] > (D[i][k] + D[k][j]))
                    D[i][j] = D[i][k] + D[k][j];
}
```

Clearly this algorithm requires $\Theta(|V|^3)$ running time, and it is the best choice for dense graphs because it is (relatively) fast and easy to implement.

16.2 Randomized Algorithms

In this section, we will consider how introducing randomness into our algorithms might speed things up, although perhaps at the expense of accuracy. But often we can reduce the possibility for error to be as low as we like, while still speeding up the algorithm.

16.2.1 Randomized algorithms for finding large values

In Section 15.1 we determined that the lower bound cost of finding the maximum value in an unsorted list is $\Omega(n)$. This is the least time needed to be certain that we have found the maximum value. But what if we are willing to relax our requirement for certainty? The first question is: What do we mean by this? There are many aspects to “certainty” and we might relax the requirement in various ways.

There are several possible guarantees that we might require from an algorithm that produces X as the maximum value, when the true maximum is Y . So far we have assumed that we require X to equal Y . This is known as an exact or deterministic algorithm to solve the problem. We could relax this and require only that X 's rank is “close to” Y 's rank (perhaps within a fixed distance or percentage). This is known as an approximation algorithm. We could require that X is “usually” Y . This is known as a probabilistic algorithm. Finally, we could require only that X 's rank is “usually” “close” to Y 's rank. This is known as a heuristic algorithm.

There are also different ways that we might choose to sacrifice reliability for speed. These types of algorithms also have names.

1. **Las Vegas Algorithms:** We always find the maximum value, and “usually” we find it fast. Such algorithms have a guaranteed result, but do not guarantee fast running time.

- 2. Monte Carlo Algorithms:** We find the maximum value fast, or we don't get an answer at all (but fast). While such algorithms have good running time, their result is not guaranteed.

Here is an example of an algorithm for finding a large value that gives up its guarantee of getting the best value in exchange for an improved running time. This is an example of a **probabilistic** algorithm, since it includes steps that are affected by **random** events. Choose m elements at random, and pick the best one of those as the answer. For large n , if $m \approx \log n$, the answer is pretty good. The cost is $m - 1$ compares (since we must find the maximum of m values). But we don't know for sure what we will get. However, we can estimate that the rank will be about $\frac{mn}{m+1}$. For example, if $n = 1,000,000$ and $m = \log n = 20$, then we expect that the largest of the 20 randomly selected values be among the top 5% of the n values.

Next, consider a slightly different problem where the goal is to pick a number in the upper half of n values. We would pick the maximum from among the first $\frac{n+1}{2}$ values for a cost of $n/2$ comparisons. Can we do better than this? Not if we want to guarantee getting the correct answer. But if we are willing to accept near certainty instead of absolute certainty, we can gain a lot in terms of speed.

As an alternative, consider this probabilistic algorithm. Pick 2 numbers and choose the greater. This will be in the upper half with probability $3/4$ (since it is not in the upper half only when both numbers we choose happen to be in the lower half). Is a probability of $3/4$ not good enough? Then we simply pick more numbers! For k numbers, the greatest is in upper half with probability $1 - \frac{1}{2^k}$, regardless of the number n that we pick from, so long as n is much larger than k (otherwise the chances might become even better). If we pick ten numbers, then the chance of failure is only one in $2^{10} = 1024$. What if we really want to be sure, because lives depend on drawing a number from the upper half? If we pick 30 numbers, we can fail only one time in a billion. If we pick enough numbers, then the chance of picking a small number is less than the chance of the power failing during the computation. Picking 100 numbers means that we can fail only one time in 10^{100} which is less chance than any disaster that you can imagine disrupting the process.

16.2.2 Skip Lists

This section presents a probabilistic search structure called the Skip List. Like BSTs, Skip Lists are designed to overcome a basic limitation of array-based and linked lists: Either search or update operations require linear time. The Skip List is an example of a **probabilistic data structure**, because it makes some of its decisions at random.

Skip Lists provide an alternative to the BST and related tree structures. The primary problem with the BST is that it may easily become unbalanced. The 2-3 tree of Chapter 10 is guaranteed to remain balanced regardless of the order in which data

values are inserted, but it is rather complicated to implement. Chapter 13 presents the AVL tree and the splay tree, which are also guaranteed to provide good performance, but at the cost of added complexity as compared to the BST. The Skip List is easier to implement than known balanced tree structures. The Skip List is not guaranteed to provide good performance (where good performance is defined as $\Theta(\log n)$ search, insertion, and deletion time), but it will provide good performance with extremely high probability (unlike the BST which has a good chance of performing poorly). As such it represents a good compromise between difficulty of implementation and performance.

Figure 16.2 illustrates the concept behind the Skip List. Figure 16.2(a) shows a simple linked list whose nodes are ordered by key value. To search a sorted linked list requires that we move down the list one node at a time, visiting $\Theta(n)$ nodes in the average case. What if we add a pointer to every other node that lets us skip alternating nodes, as shown in Figure 16.2(b)? Define nodes with a single pointer as level 0 Skip List nodes, and nodes with two pointers as level 1 Skip List nodes.

To search, follow the level 1 pointers until a value greater than the search key has been found, go back to the previous level 1 node, then revert to a level 0 pointer to travel one more node if necessary. This effectively cuts the work in half. We can continue adding pointers to selected nodes in this way — give a third pointer to every fourth node, give a fourth pointer to every eighth node, and so on — until we reach the ultimate of $\log n$ pointers in the first and middle nodes for a list of n nodes as illustrated in Figure 16.2(c). To search, start with the bottom row of pointers, going as far as possible and skipping many nodes at a time. Then, shift up to shorter and shorter steps as required. With this arrangement, the worst-case number of accesses is $\Theta(\log n)$.

We will store with each Skip List node an array named **forward** that stores the pointers as shown in Figure 16.2(c). Position **forward[0]** stores a level 0 pointer, **forward[1]** stores a level 1 pointer, and so on. The Skip List object includes data member **level** that stores the highest level for any node currently in the Skip List. The Skip List stores a header node named **head** with **level** pointers. The **find** function is shown in Figure 16.3.

Searching for a node with value 62 in the Skip List of Figure 16.2(c) begins at the header node. Follow the header node's pointer at **level**, which in this example is level 2. This points to the node with value 31. Because 31 is less than 62, we next try the pointer from **forward[2]** of 31's node to reach 69. Because 69 is greater than 62, we cannot go forward but must instead decrement the current level counter to 1. We next try to follow **forward[1]** of 31 to reach the node with value 58. Because 58 is smaller than 62, we follow 58's **forward[1]** pointer to 69. Because 69 is too big, follow 58's level 0 pointer to 62. Because 62 is not less than 62, we fall out of the **while** loop and move one step forward to the node with value 62.

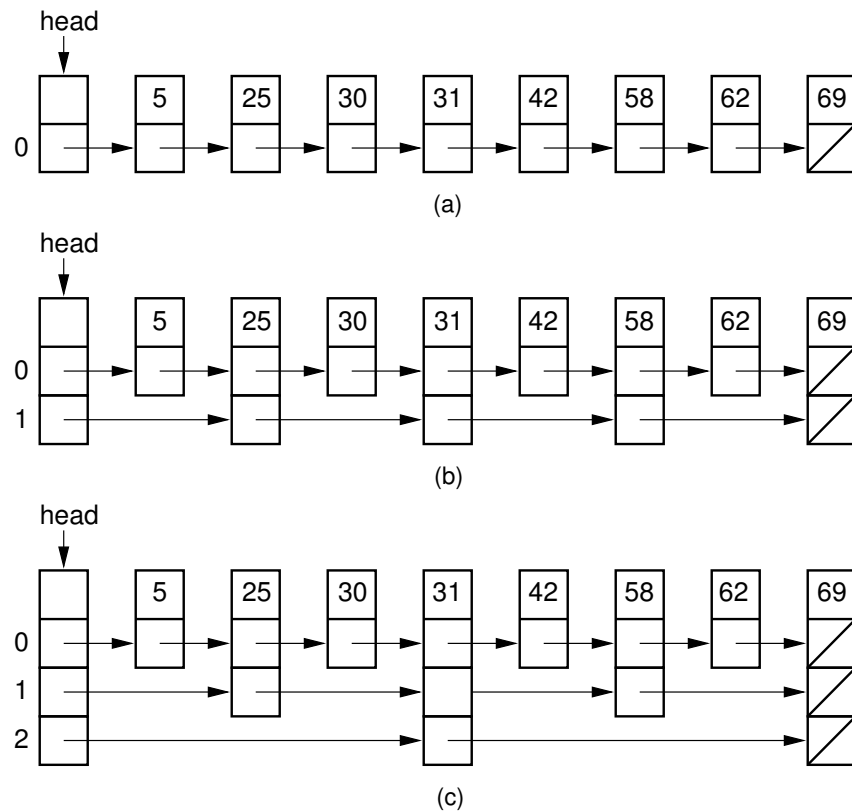


Figure 16.2 Illustration of the Skip List concept. (a) A simple linked list. (b) Augmenting the linked list with additional pointers at every other node. To find the node with key value 62, we visit the nodes with values 25, 31, 58, and 69, then we move from the node with key value 58 to the one with value 62. (c) The ideal Skip List, guaranteeing $O(\log n)$ search time. To find the node with key value 62, we visit nodes in the order 31, 69, 58, then 69 again, and finally, 62.

```

E find(const Key& k) const {
    SkipNode<Key,E> *x = head;           // Dummy header node
    for (int i=level; i>=0; i--)
        while ((x->forward[i] != NULL) &&
               (k > x->forward[i]->k))
            x = x->forward[i];
    x = x->forward[0]; // Move to actual record, if it exists
    if ((x != NULL) && (k == x->k)) return x->it;
    return NULL;
}

```

Figure 16.3 Implementation for the Skip List **find** function.

```

void insert(const Key& k, const E& it) {
    int i;
    SkipNode<Key,E> *x = head;    // Start at header node
    int newLevel = randomLevel(); // Select level for new node
    if (newLevel > level) {        // New node is deepest in list
        AdjustHead(newLevel);     // Add null pointers to header
        level = newLevel;
    }
    SkipNode<Key,E>* update[level+1]; // Track level ends
    for(i=level; i>=0; i--) {      // Search for insert position
        while((x->forward[i] != NULL) && (x->forward[i]->k < k))
            x = x->forward[i];
        update[i] = x;            // Keep track of end at level i
    }
    x = new SkipNode<Key,E>(k, it, newLevel); // New node
    for (i=0; i<=newLevel; i++) {   // Splice into list
        x->forward[i] = update[i]->forward[i]; // Where x points
        update[i]->forward[i] = x;      // What points to x
    }
    reccount++;
}

```

Figure 16.4 Implementation for the Skip List **Insert** function.

The ideal Skip List of Figure 16.2(c) has been organized so that (if the first and last nodes are not counted) half of the nodes have only one pointer, one quarter have two, one eighth have three, and so on. The distances are equally spaced; in effect this is a “perfectly balanced” Skip List. Maintaining such balance would be expensive during the normal process of insertions and deletions. The key to Skip Lists is that we do not worry about any of this. Whenever inserting a node, we assign it a level (i.e., some number of pointers). The assignment is random, using a geometric distribution yielding a 50% probability that the node will have one pointer, a 25% probability that it will have two, and so on. The following function determines the level based on such a distribution:

```

// Pick a level using an exponential distribution
int randomLevel(void) {
    int level = 0;
    while (Random(2) == 0) level++;
    return level;
}

```

Once the proper level for the node has been determined, the next step is to find where the node should be inserted and link it in as appropriate at all of its levels. Figure 16.4 shows an implementation for inserting a new value into the Skip List.

Figure 16.5 illustrates the Skip List insertion process. In this example, we begin by inserting a node with value 10 into an empty Skip List. Assume that **randomLevel** returns a value of 1 (i.e., the node is at level 1, with 2 pointers). Because the empty Skip List has no nodes, the level of the list (and thus the level

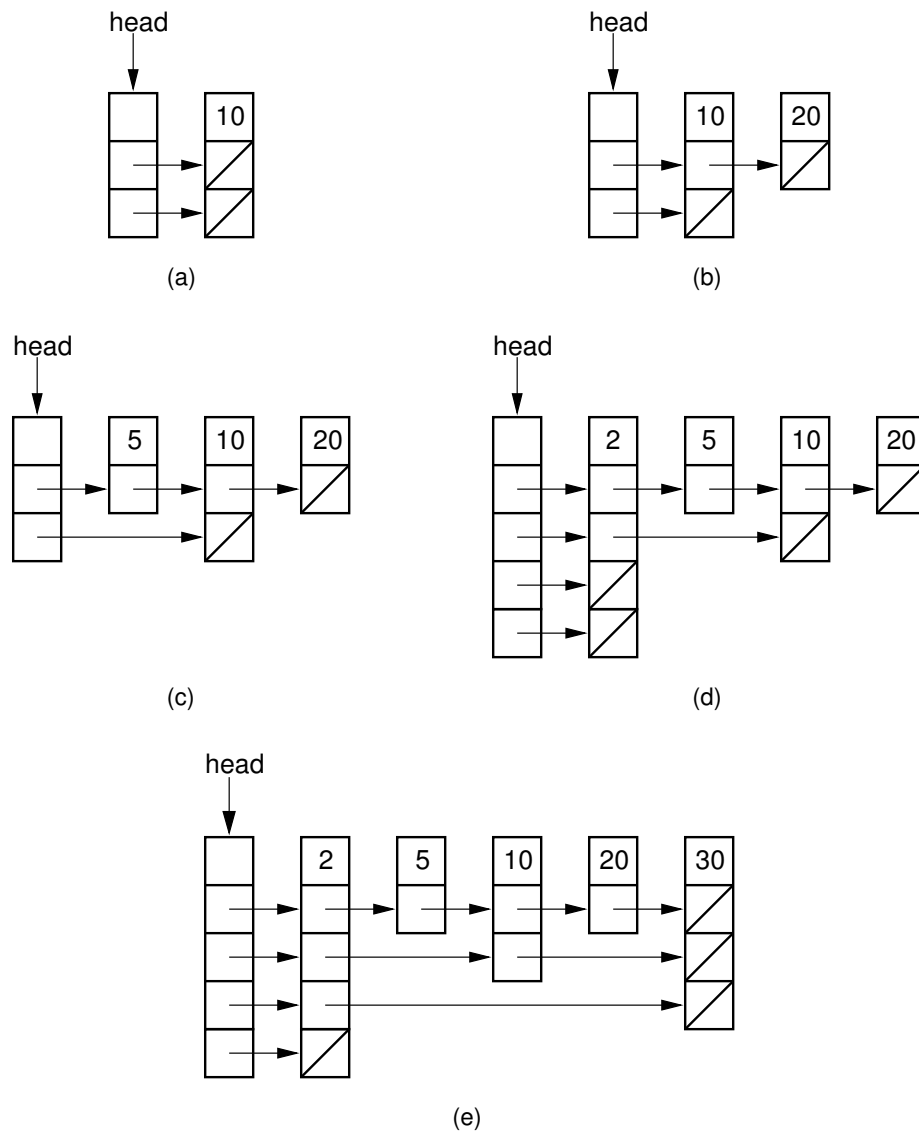


Figure 16.5 Illustration of Skip List insertion. (a) The Skip List after inserting initial value 10 at level 1. (b) The Skip List after inserting value 20 at level 0. (c) The Skip List after inserting value 5 at level 0. (d) The Skip List after inserting value 2 at level 3. (e) The final Skip List after inserting value 30 at level 2.

of the header node) must be set to 1. The new node is inserted, yielding the Skip List of Figure 16.5(a).

Next, insert the value 20. Assume this time that **randomLevel** returns 0. The search process goes to the node with value 10, and the new node is inserted after, as shown in Figure 16.5(b). The third node inserted has value 5, and again assume that **randomLevel** returns 0. This yields the Skip List of Figure 16.5(c).

The fourth node inserted has value 2, and assume that **randomLevel** returns 3. This means that the level of the Skip List must rise, causing the header node to gain an additional two (**NULL**) pointers. At this point, the new node is added to the front of the list, as shown in Figure 16.5(d).

Finally, insert a node with value 30 at level 2. This time, let us take a close look at what array **update** is used for. It stores the farthest node reached at each level during the search for the proper location of the new node. The search process begins in the header node at level 3 and proceeds to the node storing value 2. Because **forward[3]** for this node is **NULL**, we cannot go further at this level. Thus, **update[3]** stores a pointer to the node with value 2. Likewise, we cannot proceed at level 2, so **update[2]** also stores a pointer to the node with value 2. At level 1, we proceed to the node storing value 10. This is as far as we can go at level 1, so **update[1]** stores a pointer to the node with value 10. Finally, at level 0 we end up at the node with value 20. At this point, we can add in the new node with value 30. For each value **i**, the new node's **forward[i]** pointer is set to be **update[i] → forward[i]**, and the nodes stored in **update[i]** for indices 0 through 2 have their **forward[i]** pointers changed to point to the new node. This “splices” the new node into the Skip List at all levels.

The **remove** function is left as an exercise. It is similar to insertion in that the **update** array is built as part of searching for the record to be deleted. Then those nodes specified by the update array have their forward pointers adjusted to point around the node being deleted.

A newly inserted node could have a high level generated by **randomLevel**, or a low level. It is possible that many nodes in the Skip List could have many pointers, leading to unnecessary insert cost and yielding poor (i.e., $\Theta(n)$) performance during search, because not many nodes will be skipped. Conversely, too many nodes could have a low level. In the worst case, all nodes could be at level 0, equivalent to a regular linked list. If so, search will again require $\Theta(n)$ time. However, the probability that performance will be poor is quite low. There is only one chance in 1024 that ten nodes in a row will be at level 0. The motto of probabilistic data structures such as the Skip List is “Don’t worry, be happy.” We simply accept the results of **randomLevel** and expect that probability will eventually work in our favor. The advantage of this approach is that the algorithms are simple, while requiring only $\Theta(\log n)$ time for all operations in the average case.

In practice, the Skip List will probably have better performance than a BST. The BST can have bad performance caused by the order in which data are inserted. For example, if n nodes are inserted into a BST in ascending order of their key value, then the BST will look like a linked list with the deepest node at depth $n - 1$. The Skip List's performance does not depend on the order in which values are inserted into the list. As the number of nodes in the Skip List increases, the probability of encountering the worst case decreases geometrically. Thus, the Skip List illustrates a tension between the theoretical worst case (in this case, $\Theta(n)$ for a Skip List operation), and a rapidly increasing probability of average-case performance of $\Theta(\log n)$, that characterizes probabilistic data structures.

16.3 Numerical Algorithms

This section presents a variety of algorithms related to mathematical computations on numbers. Examples are activities like multiplying two numbers or raising a number to a given power. In particular, we are concerned with situations where built-in integer or floating-point operations cannot be used because the values being operated on are too large. Similar concerns arise for operations on polynomials or matrices.

Since we cannot rely on the hardware to process the inputs in a single constant-time operation, we are concerned with how to most effectively implement the operation to minimize the time cost. This begs a question as to how we should apply our normal measures of asymptotic cost in terms of growth rates on input size. First, what is an instance of addition or multiplication? Each value of the operands yields a different problem instance. And what is the input size when multiplying two numbers? If we view the input size as two (since two numbers are input), then any non-constant-time algorithm has a growth rate that is infinitely high compared to the growth of the input. This makes no sense, especially in light of the fact that we know from grade school arithmetic that adding or multiplying numbers does seem to get more difficult as the value of the numbers involved increases. In fact, we know from standard grade school algorithms that the cost of standard addition is linear on the number of digits being added, and multiplication has cost $n \times m$ when multiplying an m -digit number by an n -digit number.

The number of digits for the operands does appear to be a key consideration when we are performing a numeric algorithm that is sensitive to input size. The number of digits is simply the log of the value, for a suitable base of the log. Thus, for the purpose of calculating asymptotic growth rates of algorithms, we will consider the “size” of an input value to be the log of that value. Given this view, there are a number of features that seem to relate such operations.

- Arithmetic operations on large values are not cheap.
- There is only one instance of value n .

- There are 2^k instances of length k or less.
- The size (length) of value n is $\log n$.
- The cost of a particular algorithm can decrease when n increases in value (say when going from a value of $2^k - 1$ to 2^k to $2^k + 1$), but generally increases when n increases in length.

16.3.1 Exponentiation

We will start our examination of standard numerical algorithms by considering how to perform exponentiation. That is, how do we compute m^n ? We could multiply by m a total of $n - 1$ times. Can we do better? Yes, there is a simple divide and conquer approach that we can use. We can recognize that, when n is even, $m^n = m^{n/2}m^{n/2}$. If n is odd, then $m^n = m^{\lfloor n/2 \rfloor}m^{\lfloor n/2 \rfloor}m$. This leads to the following recursive algorithm

```
int Power(base, exp) {
    if exp = 0 return 1;
    int half = Power(base, exp/2); // integer division of exp
    half = half * half;
    if (odd(exp)) then half = half * base;
    return half;
}
```

Function **Power** has recurrence relation

$$f(n) = \begin{cases} 0 & n = 1 \\ f(\lfloor n/2 \rfloor) + 1 + n \bmod 2 & n > 1 \end{cases}$$

whose solution is

$$f(n) = \lfloor \log n \rfloor + \beta(n) - 1$$

where β is the number of 1's in the binary representation of n .

How does this cost compare with the problem size? The original problem size is $\log m + \log n$, and the number of multiplications required is $\log n$. This is far better (in fact, exponentially better) than performing $n - 1$ multiplications.

16.3.2 Largest Common Factor

We will next present Euclid's algorithm for finding the largest common factor (LCF) for two integers. The LCF is the largest integer that divides both inputs evenly.

First we make this observation: If k divides n and m , then k divides $n - m$. We know this is true because if k divides n then $n = ak$ for some integer a , and if k divides m then $m = bk$ for some integer b . So, $LCF(n, m) = LCF(n - m, n) = LCF(m, n - m) = LCF(m, n)$.

Now, for any value n there exists k and l such that

$$n = km + l \text{ where } m > l \geq 0.$$

From the definition of the mod function, we can derive the fact that

$$n = \lfloor n/m \rfloor m + n \bmod m.$$

Since the LCF is a factor of both n and m , and since $n = km + l$, the LCF must therefore be a factor of both km and l , and also the largest common factor of each of these terms. As a consequence, $LCF(n, m) = LCF(m, l) = LCF(m, n \bmod m)$.

This observation leads to a simple algorithm. We will assume that $n \geq m$. At each iteration we replace n with m and m with $n \bmod m$ until we have driven m to zero.

```
int LCF(int n, int m) {
    if (m == 0) return n;
    return LCF(m, n % m);
}
```

To determine how expensive this algorithm is, we need to know how much progress we are making at each step. Note that after two iterations, we have replaced n with $n \bmod m$. So the key question becomes: How big is $n \bmod m$ relative to n ?

$$\begin{aligned} n \geq m &\Rightarrow n/m \geq 1 \\ &\Rightarrow 2\lfloor n/m \rfloor > n/m \\ &\Rightarrow m\lfloor n/m \rfloor > n/2 \\ &\Rightarrow n - n/2 > n - m\lfloor n/m \rfloor = n \bmod m \\ &\Rightarrow n/2 > n \bmod m \end{aligned}$$

Thus, function LCF will halve its first parameter in no more than 2 iterations. The total cost is then $O(\log n)$.

16.3.3 Matrix Multiplication

The standard algorithm for multiplying two $n \times n$ matrices requires $\Theta(n^3)$ time. It is possible to do better than this by rearranging and grouping the multiplications in various ways. One example of this is known as Strassen's matrix multiplication algorithm.

For simplicity, we will assume that n is a power of two. In the following, A and B are $n \times n$ arrays, while A_{ij} and B_{ij} refer to arrays of size $n/2 \times n/2$. Using

this notation, we can think of matrix multiplication using divide and conquer in the following way:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}.$$

Of course, each of the multiplications and additions on the right side of this equation are recursive calls on arrays of half size, and additions of arrays of half size, respectively. The recurrence relation for this algorithm is

$$T(n) = 8T(n/2) + 4(n/2)^2 = \Theta(n^3).$$

This closed form solution can easily be obtained by applying the Master Theorem 14.1.

Strassen's algorithm carefully rearranges the way that the various terms are multiplied and added together. It does so in a particular order, as expressed by the following equation:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} s_1 + s_2 - s_4 + s_6 & s_4 + s_5 \\ s_6 + s_7 & s_2 - s_3 + s_5 - s_7 \end{bmatrix}.$$

In other words, the result of the multiplication for an $n \times n$ array is obtained by a different series of matrix multiplications and additions for $n/2 \times n/2$ arrays. Multiplications between subarrays also use Strassen's algorithm, and the addition of two subarrays requires $\Theta(n^2)$ time. The subfactors are defined as follows:

$$\begin{aligned} s_1 &= (A_{12} - A_{22}) \cdot (B_{21} + B_{22}) \\ s_2 &= (A_{11} + A_{22}) \cdot (B_{11} + B_{22}) \\ s_3 &= (A_{11} - A_{21}) \cdot (B_{11} + B_{12}) \\ s_4 &= (A_{11} + A_{12}) \cdot B_{22} \\ s_5 &= A_{11} \cdot (B_{12} - B_{22}) \\ s_6 &= A_{22} \cdot (B_{21} - B_{11}) \\ s_7 &= (A_{21} + A_{22}) \cdot B_{11} \end{aligned}$$

With a little effort, you should be able to verify that this peculiar combination of operations does in fact produce the correct answer!

Now, looking at the list of operations to compute the s factors, and then counting the additions/subtractions needed to put them together to get the final answers, we see that we need a total of seven (array) multiplications and 18 (array) additions/subtractions to do the job. This leads to the recurrence

$$\begin{aligned} T(n) &= 7T(n/2) + 18(n/2)^2 \\ T(n) &= \Theta(n^{\log_2 7}) = \Theta(n^{2.81}). \end{aligned}$$

We obtained this closed form solution again by applying the Master Theorem.

Unfortunately, while Strassen's algorithm does in fact reduce the asymptotic complexity over the standard algorithm, the cost of the large number of addition and subtraction operations raises the constant factor involved considerably. This means that an extremely large array size is required to make Strassen's algorithm practical in real applications.

16.3.4 Random Numbers

The success of randomized algorithms such as were presented in Section 16.2 depend on having access to a good random number generator. While modern compilers are likely to include a random number generator that is good enough for most purposes, it is helpful to understand how they work, and to even be able to construct your own in case you don't trust the one provided. This is easy to do.

First, let us consider what a random sequence. From the following list, which appears to be a sequence of "random" numbers?

- 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
- 1, 2, 3, 4, 5, 6, 7, 8, 9, ...
- 2, 7, 1, 8, 2, 8, 1, 8, 2, ...

In fact, all three happen to be the beginning of a some sequence in which one could continue the pattern to generate more values (in case you do not recognize it, the third one is the initial digits of the irrational constant e). Viewed as a series of digits, ideally every possible sequence has equal probability of being generated (even the three sequences above). In fact, definitions of randomness generally have features such as:

- One cannot predict the next item. The series is **unpredictable**.
- The series cannot be described more briefly than simply listing it out. This is the **equidistribution** property.

There is no such thing as a random number sequence, only "random enough" sequences. A sequence is **pseudorandom** if no future term can be predicted in polynomial time, given all past terms.

Most computer systems use a deterministic algorithm to select pseudorandom numbers.¹ The most commonly used approach historically is known as the **Linear Congruential Method** (LCM). The LCM method is quite simple. We begin by picking a **seed** that we will call $r(1)$. Then, we can compute successive terms as follows.

$$r(i) = (r(i-1) \times b) \bmod t$$

where b and t are constants.

¹Another approach is based on using a computer chip that generates random numbers resulting from "thermal noise" in the system. Time will tell if this approach replaces deterministic approaches.

By definition of the mod function, all generated numbers must be in the range 0 to $t - 1$. Now, consider what happens when $r(i) = r(j)$ for values i and j . Of course then $r(i + 1) = r(j + 1)$ which means that we have a repeating cycle.

Since the values coming out of the random number generator are between 0 and $t - 1$, the longest cycle that we can hope for has length t . In fact, since $r(0) = 0$, it cannot even be quite this long. It turns out that to get a good result, it is crucial to pick good values for both b and t . To see why, consider the following example.

Example 16.4 Given a t value of 13, we can get very different results depending on the b value that we pick, in ways that are hard to predict.

$$\begin{aligned} r(i) &= 6r(i-1) \bmod 13 = \\ &\quad \dots, 1, 6, 10, 8, 9, 2, 12, 7, 3, 5, 4, 11, 1, \dots \\ r(i) &= 7r(i-1) \bmod 13 = \\ &\quad \dots, 1, 7, 10, 5, 9, 11, 12, 6, 3, 8, 4, 2, 1, \dots \\ r(i) &= 5r(i-1) \bmod 13 = \\ &\quad \dots, 1, 5, 12, 8, 1, \dots \\ &\quad \dots, 2, 10, 11, 3, 2, \dots \\ &\quad \dots, 4, 7, 9, 6, 4, \dots \\ &\quad \dots, 0, 0, \dots \end{aligned}$$

Clearly, a b value of 5 is far inferior to b values of 6 or 7 in this example.

If you would like to write a simple LCM random number generator of your own, an effective one can be made with the following formula.

$$r(i) = 16807r(i-1) \bmod 2^{31} - 1.$$

16.3.5 The Fast Fourier Transform

As noted at the beginning of this section, multiplication is considerably more difficult than addition. The cost to multiply two n -bit numbers directly is $O(n^2)$, while addition of two n -bit numbers is $O(n)$.

Recall from Section 2.3 that one property of logarithms is

$$\log nm = \log n + \log m.$$

Thus, if taking logarithms and anti-logarithms were cheap, then we could reduce multiplication to addition by taking the log of the two operands, adding, and then taking the anti-log of the sum.

Under normal circumstances, taking logarithms and anti-logarithms is expensive, and so this reduction would not be considered practical. However, this reduction is precisely the basis for the slide rule. The slide rule uses a logarithmic scale to measure the lengths of two numbers, in effect doing the conversion to logarithms automatically. These two lengths are then added together, and the inverse

logarithm of the sum is read off another logarithmic scale. The part normally considered expensive (taking logarithms and anti-logarithms) is cheap because it is a physical part of the slide rule. Thus, the entire multiplication process can be done cheaply via a reduction to addition. In the days before electronic calculators, slide rules were routinely used by scientists and engineers to do basic calculations of this nature.

Now consider the problem of multiplying polynomials. A vector \mathbf{a} of n values can uniquely represent a polynomial of degree $n - 1$, expressed as

$$P_{\mathbf{a}}(x) = \sum_{i=0}^{n-1} \mathbf{a}_i x^i.$$

Alternatively, a polynomial can be uniquely represented by a list of its values at n distinct points. Finding the value for a polynomial at a given point is called **evaluation**. Finding the coefficients for the polynomial given the values at n points is called **interpolation**.

To multiply two $n - 1$ -degree polynomials A and B normally takes $\Theta(n^2)$ coefficient multiplications. However, if we evaluate both polynomials (at the same points), we can simply multiply the corresponding pairs of values to get the corresponding values for polynomial AB .

Example 16.5 Polynomial A: $x^2 + 1$.

Polynomial B: $2x^2 - x + 1$.

Polynomial AB: $2x^4 - x^3 + 3x^2 - x + 1$.

When we multiply the evaluations of A and B at points 0, 1, and -1, we get the following results.

$$AB(-1) = (2)(4) = 8$$

$$AB(0) = (1)(1) = 1$$

$$AB(1) = (2)(2) = 4$$

These results are the same as when we evaluate polynomial AB at these points.

Note that evaluating any polynomial at 0 is easy. If we evaluate at 1 and -1, we can share a lot of the work between the two evaluations. But we would need five points to nail down polynomial AB , since it is a degree-4 polynomial. Fortunately, we can speed processing for any pair of values c and $-c$. This seems to indicate some promising ways to speed up the process of evaluating polynomials. But, evaluating two points in roughly the same time as evaluating one point only speeds the process by a constant factor. Is there some way to generalize these

observations to speed things up further? And even if we do find a way to evaluate many points quickly, we will also need to interpolate the five values to get the coefficients of AB back.

So we see that we could multiply two polynomials in less than $\Theta(n^2)$ operations *if* a fast way could be found to do evaluation/interpolation of $2n - 1$ points. Before considering further how this might be done, first observe again the relationship between evaluating a polynomial at values c and $-c$. In general, we can write $P_a(x) = E_a(x) + O_a(x)$ where E_a is the even powers and O_a is the odd powers. So,

$$P_a(x) = \sum_{i=0}^{n/2-1} a_{2i}x^{2i} + \sum_{i=0}^{n/2-1} a_{2i+1}x^{2i+1}$$

The significance is that when evaluating the pair of values c and $-c$, we get

$$\begin{aligned} E_a(c) + O_a(c) &= E_a(c) - O_a(-c) \\ O_a(c) &= -O_a(-c) \end{aligned}$$

Thus, we only need to compute the E s and O s once instead of twice to get both evaluations.

The key to fast polynomial multiplication is finding the right points to use for evaluation/interpolation to make the process efficient. In particular, we want to take advantage of symmetries, such as the one we see for evaluating x and $-x$. But we need to find even more symmetries between points if we want to do more than cut the work in half. We have to find symmetries not just between pairs of values, but also further symmetries between pairs of pairs, and then pairs of pairs of pairs, and so on.

Recall that a **complex** number z has a real component and an imaginary component. We can consider the position of z on a number line if we use the y dimension for the imaginary component. Now, we will define a **primitive n th root of unity** if

1. $z^n = 1$ and
2. $z^k \neq 1$ for $0 < k < n$.

z^0, z^1, \dots, z^{n-1} are called the **n th roots of unity**. For example, when $n = 4$, then $z = i$ or $z = -i$. In general, we have the identities $e^{i\pi} = -1$, and $z^j = e^{2\pi i j/n} = -1^{2j/n}$. The significance is that we can find as many points on a unit circle as we would need (see Figure 16.6). But these points are special in that they will allow us to do just the right computation necessary to get the needed symmetries to speed up the overall process of evaluating many points at once.

The next step is to define how the computation is done. Define an $n \times n$ matrix A_z with row i and column j as

$$A_z = (z^{ij}).$$

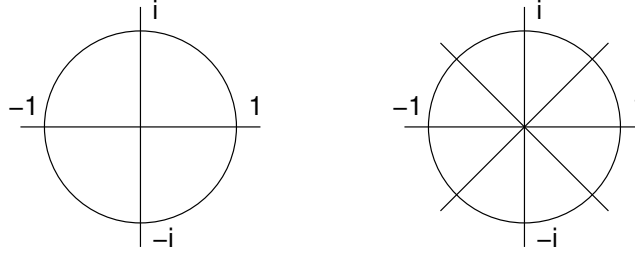


Figure 16.6 Examples of the 4th and 8th roots of unity.

The idea is that there is a row for each root (row i for z^i) while the columns correspond to the power of the exponent of the x value in the polynomial. For example, when $n = 4$ we have $z = i$. Thus, the A_z array appears as follows.

$$A_z = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

Let $a = [a_0, a_1, \dots, a_{n-1}]^T$ be a vector that stores the coefficients for the polynomial being evaluated. We can then do the calculations to evaluate the polynomial at the n th roots of unity by multiplying the A_z matrix by the coefficient vector. The resulting vector F_z is called the Discrete Fourier Transform for the polynomial.

$$F_z = A_z a = b.$$

$$b_i = \sum_{k=0}^{n-1} a_k z^{ik}.$$

When $n = 8$, then $z = \sqrt{i}$, since $\sqrt{i}^8 = 1$. So, the corresponding matrix is as follows.

$$A_z = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \sqrt{i} & i & i\sqrt{i} & -1 & -\sqrt{i} & -i & -i\sqrt{i} \\ 1 & i & -1 & -i & 1 & i & -1 & -i \\ 1 & i\sqrt{i} & -i & \sqrt{i} & -1 & -i\sqrt{i} & i & -\sqrt{i} \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -\sqrt{i} & i & -i\sqrt{i} & -1 & \sqrt{i} & -i & i\sqrt{i} \\ 1 & -i & -1 & i & 1 & -i & -1 & i \\ 1 & -i\sqrt{i} & -i & -\sqrt{i} & -1 & i\sqrt{i} & i & \sqrt{i} \end{bmatrix}$$

We still have two problems. We need to be able to multiply this matrix and the vector faster than just by performing a standard matrix-vector multiplication,

otherwise the cost is still n^2 multiplies to do the evaluation. Even if we can multiply the matrix and vector cheaply, we still need to be able to reverse the process. That is, after transforming the two input polynomials by evaluating them, and then pair-wise multiplying the evaluated points, we must interpolate those points to get the resulting polynomial back that corresponds to multiplying the original input polynomials.

The interpolation step is nearly identical to the evaluation step.

$$F_z^{-1} = A_z^{-1}b' = a'.$$

We need to find A_z^{-1} . This turns out to be simple to compute, and is defined as follows.

$$A_z^{-1} = \frac{1}{n}A_{1/z}.$$

In other words, interpolation (the inverse transformation) requires the same computation as evaluation, except that we substitute $1/z$ for z (and multiply by $1/n$ at the end). So, if we can do one fast, we can do the other fast.

If you examine the example A_z matrix for $n = 8$, you should see that there are symmetries within the matrix. For example, the top half is identical to the bottom half with suitable sign changes on some rows and columns. Likewise for the left and right halves. An efficient divide and conquer algorithm exists to perform both the evaluation and the interpolation in $\Theta(n \log n)$ time. This is called the **Discrete Fourier Transform** (DFT). It is a recursive function that decomposes the matrix multiplications, taking advantage of the symmetries made available by doing evaluation at the n th roots of unity. The algorithm is as follows.

```
Fourier_Transform(double *Polynomial, int n) {
    // Compute the Fourier transform of Polynomial
    // with degree n. Polynomial is a list of
    // coefficients indexed from 0 to n-1. n is
    // assumed to be a power of 2.
    double Even[n/2], Odd[n/2], List1[n/2], List2[n/2];

    if (n==1) return Polynomial[0];

    for (j=0; j<=n/2-1; j++) {
        Even[j] = Polynomial[2j];
        Odd[j] = Polynomial[2j+1];
    }
    List1 = Fourier_Transform(Even, n/2);
    List2 = Fourier_Transform(Odd, n/2);
    for (j=0; j<=n-1, J++) {
        Imaginary z = pow(E, 2*i*PI*j/n);
        k = j % (n/2);
        Polynomial[j] = List1[k] + z*List2[k];
    }
    return Polynomial;
}
```

Thus, the full process for multiplying polynomials A and B using the Fourier transform is as follows.

1. Represent an $n - 1$ -degree polynomial as $2n - 1$ coefficients:

$$[a_0, a_1, \dots, a_{n-1}, 0, \dots, 0]$$

2. Perform **Fourier Transform** on the representations for A and B
3. Pairwise multiply the results to get $2n - 1$ values.
4. Perform the inverse **Fourier Transform** to get the $2n - 1$ degree polynomial AB .

16.4 Further Reading

For further information on Skip Lists, see “Skip Lists: A Probabilistic Alternative to Balanced Trees” by William Pugh [Pug90].

16.5 Exercises

16.1 Solve Towers of Hanoi using a dynamic programming algorithm.

16.2 There are six possible permutations of the lines

```
for (int k=0; k<G.n(); k++)
    for (int i=0; i<G.n(); i++)
        for (int j=0; j<G.n(); j++)
```

in Floyd’s algorithm. Which ones give a correct algorithm?

16.3 Show the result of running Floyd’s all-pairs shortest-paths algorithm on the graph of Figure 11.26.

16.4 The implementation for Floyd’s algorithm given in Section 16.1.2 is inefficient for adjacency lists because the edges are visited in a bad order when initializing array **D**. What is the cost of this initialization step for the adjacency list? How can this initialization step be revised so that it costs $\Theta(|V|^2)$ in the worst case?

16.5 State the greatest possible lower bound that you can prove for the all-pairs shortest-paths problem, and justify your answer.

16.6 Show the Skip List that results from inserting the following values. Draw the Skip List after each insert. With each value, assume the depth of its corresponding node is as given in the list.

value	depth
5	2
20	0
30	0
2	0
25	1
26	3
31	0

- 16.7** If we had a linked list that would never be modified, we can use a simpler approach than the Skip List to speed access. The concept would remain the same in that we add additional pointers to list nodes for efficient access to the i th element. How can we add a second pointer to each element of a singly linked list to allow access to an arbitrary element in $O(\log n)$ time?
- 16.8** What is the expected (average) number of pointers for a Skip List node?
- 16.9** Write a function to remove a node with given value from a Skip List.
- 16.10** Write a function to find the i th node on a Skip List.

16.6 Projects

- 16.1** Complete the implementation of the Skip List-based dictionary begun in Section 16.2.2.
- 16.2** Implement both a standard $\Theta(n^3)$ matrix multiplication algorithm and Strassen's matrix multiplication algorithm (see Exercise 14.16.3.3). Using empirical testing, try to estimate the constant factors for the runtime equations of the two algorithms. How big must n be before Strassen's algorithm becomes more efficient than the standard algorithm?

Limits to Computation

This book describes data structures that can be used in a wide variety of problems, and many examples of efficient algorithms. In general, our search algorithms strive to be at worst in $O(\log n)$ to find a record, and our sorting algorithms strive to be in $O(n \log n)$. A few algorithms have higher asymptotic complexity. Both Floyd's all-pairs shortest-paths algorithm and standard matrix multiply have running times of $\Theta(n^3)$ (though for both, the amount of data being processed is $\Theta(n^2)$).

We can solve many problems efficiently because we have available (and choose to use) efficient algorithms. Given any problem for which you know *some* algorithm, it is always possible to write an inefficient algorithm to “solve” the problem. For example, consider a sorting algorithm that tests every possible permutation of its input until it finds the correct permutation that provides a sorted list. The running time for this algorithm would be unacceptably high, because it is proportional to the number of permutations which is $n!$ for n inputs. When solving the minimum-cost spanning tree problem, if we were to test every possible subset of edges to see which forms the shortest minimum spanning tree, the amount of work would be proportional to $2^{|E|}$ for a graph with $|E|$ edges. Fortunately, for both of these problems we have more clever algorithms that allow us to find answers (relatively) quickly without explicitly testing every possible solution.

Unfortunately, there are many computing problems for which the best possible algorithm takes a long time to run. A simple example is the Towers of Hanoi problem, which requires 2^n moves to “solve” a tower with n disks. It is not possible for any computer program that solves the Towers of Hanoi problem to run in less than $\Omega(2^n)$ time, because that many moves must be printed out.

Besides those problems whose solutions *must* take a long time to run, there are also many problems for which we simply do not know if there are efficient algorithms or not. The best algorithms that we know for such problems are very slow, but perhaps there are better ones waiting to be discovered. Of course, while having a problem with high running time is bad, it is even worse to have a problem that cannot be solved at all! Such problems do exist, and are discussed in Section 17.3.

This chapter presents a brief introduction to the theory of expensive and impossible problems. Section 17.1 presents the concept of a reduction, which is the central tool used for analyzing the difficulty of a problem (as opposed to analyzing the cost of an algorithm). Reductions allow us to relate the difficulty of various problems, which is often much easier than doing the analysis for a problem from first principles. Section 17.2 discusses “hard” problems, by which we mean problems that require, or at least appear to require, time exponential on the input size. Finally, Section 17.3 considers various problems that, while often simple to define and comprehend, are in fact impossible to solve using a computer program. The classic example of such a problem is deciding whether an arbitrary computer program will go into an infinite loop when processing a specified input. This is known as the **halting problem**.

17.1 Reductions

We begin with an important concept for understanding the relationships between problems, called **reduction**. Reduction allows us to solve one problem in terms of another. Equally importantly, when we wish to understand the difficulty of a problem, reduction allows us to make relative statements about upper and lower bounds on the cost of a problem (as opposed to an algorithm or program).

Because the concept of a problem is discussed extensively in this chapter, we want notation to simplify problem descriptions. Throughout this chapter, a problem will be defined in terms of a mapping between inputs and outputs, and the name of the problem will be given in all capital letters. Thus, a complete definition of the sorting problem could appear as follows:

SORTING:

Input: A sequence of integers $x_0, x_1, x_2, \dots, x_{n-1}$.

Output: A permutation $y_0, y_1, y_2, \dots, y_{n-1}$ of the sequence such that $y_i \leq y_j$ whenever $i < j$.

When you buy or write a program to solve one problem, such as sorting, you might be able to use it to help solve a different problem. This is known in software engineering as **software reuse**. To illustrate this, let us consider another problem.

PAIRING:

Input: Two sequences of integers $X = (x_0, x_1, \dots, x_{n-1})$ and $Y = (y_0, y_1, \dots, y_{n-1})$.

Output: A pairing of the elements in the two sequences such that the least value in X is paired with the least value in Y , the next least value in X is paired with the next least value in Y , and so on.

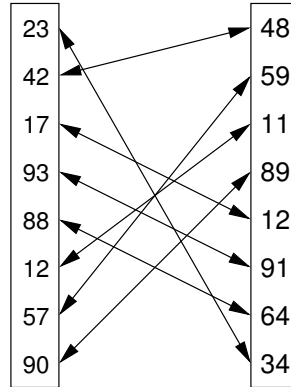


Figure 17.1 An illustration of PAIRING. The two lists of numbers are paired up so that the least values from each list make a pair, the next smallest values from each list make a pair, and so on.

Figure 17.1 illustrates PAIRING. One way to solve PAIRING is to use an existing sorting program to sort each of the two sequences, and then pair off items based on their position in sorted order. Technically we say that in this solution, PAIRING is **reduced** to SORTING, because SORTING is used to solve PAIRING.

Notice that reduction is a three-step process. The first step is to convert an instance of PAIRING into two instances of SORTING. The conversion step in this example is not very interesting; it simply takes each sequence and assigns it to an array to be passed to SORTING. The second step is to sort the two arrays (i.e., apply SORTING to each array). The third step is to convert the output of SORTING to the output for PAIRING. This is done by pairing the first elements in the sorted arrays, the second elements, and so on.

A reduction of PAIRING to SORTING helps to establish an upper bound on the cost of PAIRING. In terms of asymptotic notation, assuming that we can find one method to convert the inputs to PAIRING into inputs to SORTING “fast enough,” and a second method to convert the result of SORTING back to the correct result for PAIRING “fast enough,” then the asymptotic cost of PAIRING cannot be more than the cost of SORTING. In this case, there is little work to be done to convert from PAIRING to SORTING, or to convert the answer from SORTING back to the answer for PAIRING, so the dominant cost of this solution is performing the sort operation. Thus, an upper bound for PAIRING is in $O(n \log n)$.

It is important to note that the pairing problem does *not* require that elements of the two sequences be sorted. This is merely one possible way to solve the problem. PAIRING only requires that the elements of the sequences be paired correctly. Perhaps there is another way to do it? Certainly if we use sorting to solve PAIRING, the algorithms will require $\Omega(n \log n)$ time. But, another approach might conceivably be faster.

There is another use of reductions aside from applying an old algorithm to solve a new problem (and thereby establishing an upper bound for the new problem). That is to prove a lower bound on the cost of a new problem by showing that it could be used as a solution for an old problem with a known lower bound.

Assume we can go the other way and convert SORTING to PAIRING “fast enough.” What does this say about the minimum cost of PAIRING? We know from Section 7.9 that the cost of SORTING in the worst and average cases is in $\Omega(n \log n)$. In other words, the best possible algorithm for sorting requires at least $n \log n$ time.

Assume that PAIRING could be done in $O(n)$ time. Then, one way to create a sorting algorithm would be to convert SORTING into PAIRING, run the algorithm for PAIRING, and finally convert the answer back to the answer for SORTING. Provided that we can convert SORTING to/from PAIRING “fast enough,” this process would yield an $O(n)$ algorithm for sorting! Because this contradicts what we know about the lower bound for SORTING, and the only flaw in the reasoning is the initial assumption that PAIRING can be done in $O(n)$ time, we can conclude that there is no $O(n)$ time algorithm for PAIRING. This reduction process tells us that PAIRING must be at least as expensive as SORTING and so must itself have a lower bound in $\Omega(n \log n)$.

To complete this proof regarding the lower bound for PAIRING, we need now to find a way to reduce SORTING to PAIRING. This is easily done. Take an instance of SORTING (i.e., an array A of n elements). A second array B is generated that simply stores i in position i for $0 \leq i < n$. Pass the two arrays to PAIRING. Take the resulting set of pairs, and use the value from the B half of the pair to tell which position in the sorted array the A half should take; that is, we can now reorder the records in the A array using the corresponding value in the B array as the sort key and running a simple $\Theta(n)$ Binsort. The conversion of SORTING to PAIRING can be done in $O(n)$ time, and likewise the conversion of the output of PAIRING can be converted to the correct output for SORTING in $O(n)$ time. Thus, the cost of this “sorting algorithm” is dominated by the cost for PAIRING.

Consider any two problems for which a suitable reduction from one to the other can be found. The first problem takes an arbitrary instance of its input, which we will call I , and transforms I to a solution, which we will call SLN . The second problem takes an arbitrary instance of its input, which we will call I' , and transforms I' to a solution, which we will call SLN' . We can define reduction more formally as a three-step process:

1. Transform an arbitrary instance of the first problem to an instance of the second problem. In other words, there must be a transformation from any instance I of the first problem to an instance I' of the second problem.
2. Apply an algorithm for the second problem to the instance I' , yielding a solution SLN' .

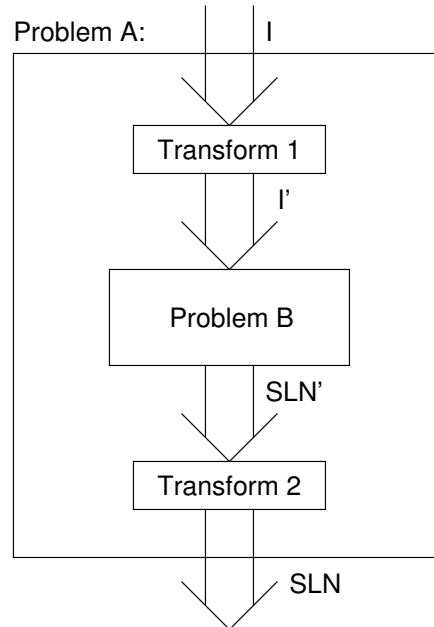


Figure 17.2 The general process for reduction shown as a “blackbox” diagram.

3. Transform SLN' to the solution of I , known as SLN . Note that SLN must in fact be the correct solution for I for the reduction to be acceptable.

Figure 17.2 shows a graphical representation of the general reduction process, showing the role of the two problems, and the two transformations. Figure 17.3 shows a similar diagram for the reduction of SORTING to PAIRING.

It is important to note that the reduction process does not give us an algorithm for solving either problem by itself. It merely gives us a method for solving the first problem given that we already have a solution to the second. More importantly for the topics to be discussed in the remainder of this chapter, reduction gives us a way to understand the bounds of one problem in terms of another. Specifically, given efficient transformations, the upper bound of the first problem is at most the upper bound of the second. Conversely, the lower bound of the second problem is at least the lower bound of the first.

As a second example of reduction, consider the simple problem of multiplying two n -digit numbers. The standard long-hand method for multiplication is to multiply the last digit of the first number by the second number (taking $\Theta(n)$ time), multiply the second digit of the first number by the second number (again taking $\Theta(n)$ time), and so on for each of the n digits of the first number. Finally, the intermediate results are added together. Note that adding two numbers of length M and N can easily be done in $\Theta(M+N)$ time. Because each digit of the first number

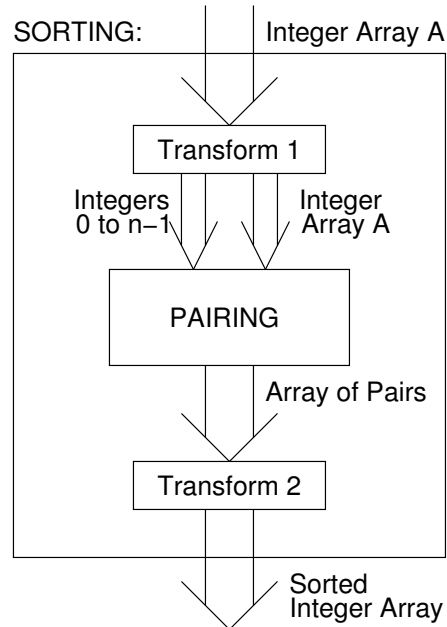


Figure 17.3 A reduction of SORTING to PAIRING shown as a “blackbox” diagram.

is multiplied against each digit of the second, this algorithm requires $\Theta(n^2)$ time. Asymptotically faster (but more complicated) algorithms are known, but none is so fast as to be in $O(n)$.

Next we ask the question: Is squaring an n -digit number as difficult as multiplying two n -digit numbers? We might hope that something about this special case will allow for a faster algorithm than is required by the more general multiplication problem. However, a simple reduction proof serves to show that squaring is “as hard” as multiplying.

The key to the reduction is the following formula:

$$X \times Y = \frac{(X + Y)^2 - (X - Y)^2}{4}.$$

The significance of this formula is that it allows us to convert an arbitrary instance of multiplication to a series of operations involving three addition/subtractions (each of which can be done in linear time), two squarings, and a division by 4. Note that the division by 4 can be done in linear time (simply convert to binary, shift right by two digits, and convert back).

This reduction shows that if a linear time algorithm for squaring can be found, it can be used to construct a linear time algorithm for multiplication.

Our next example of reduction concerns the multiplication of two $n \times n$ matrices. For this problem, we will assume that the values stored in the matrices are simple integers and that multiplying two simple integers takes constant time (because multiplication of two `int` variables takes a fixed number of machine instructions). The standard algorithm for multiplying two matrices is to multiply each element of the first matrix's first row by the corresponding element of the second matrix's first column, then adding the numbers. This takes $\Theta(n)$ time. Each of the n^2 elements of the solution are computed in similar fashion, requiring a total of $\Theta(n^3)$ time. Faster algorithms are known (see the discussion of Strassen's Algorithm in Section 16.3.3), but none are so fast as to be in $O(n^2)$.

Now, consider the case of multiplying two **symmetric** matrices. A symmetric matrix is one in which entry ij is equal to entry ji ; that is, the upper-right triangle of the matrix is a mirror image of the lower-left triangle. Is there something about this restricted case that allows us to multiply two symmetric matrices faster than in the general case? The answer is no, as can be seen by the following reduction. Assume that we have been given two $n \times n$ matrices A and B . We can construct a $2n \times 2n$ symmetric matrix from an arbitrary matrix A as follows:

$$\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}.$$

Here 0 stands for an $n \times n$ matrix composed of zero values, A is the original matrix, and A^T stands for the transpose of matrix A .¹ Note that the resulting matrix is now symmetric. We can convert matrix B to a symmetric matrix in a similar manner. If symmetric matrices could be multiplied "quickly" (in particular, if they could be multiplied together in $\Theta(n^2)$ time), then we could find the result of multiplying two arbitrary $n \times n$ matrices in $\Theta(n^2)$ time by taking advantage of the following observation:

$$\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} 0 & B^T \\ B & 0 \end{bmatrix} = \begin{bmatrix} AB & 0 \\ 0 & A^T B^T \end{bmatrix}.$$

In the above formula, AB is the result of multiplying matrices A and B together.

17.2 Hard Problems

There are several ways that a problem could be considered hard. For example, we might have trouble understanding the definition of the problem itself. At the beginning of a large data collection and analysis project, developers and their clients might have only a hazy notion of what their goals actually are, and need to work that out over time. For other types of problems, we might have trouble finding or understanding an algorithm to solve the problem. Understanding spoken English

¹The transpose operation takes position ij of the original matrix and places it in position ji of the transpose matrix. This can easily be done in n^2 time for an $n \times n$ matrix.

and translating it to written text is an example of a problem whose goals are easy to define, but whose solution is not easy to discover. But even though a natural language processing algorithm might be difficult to write, the program's running time might be fairly fast. There are many practical systems today that solve aspects of this problem in reasonable time.

None of these is what is commonly meant when a computer theoretician uses the word "hard." Throughout this section, "hard" means that the best-known algorithm for the problem is expensive in its running time. One example of a hard problem is Towers of Hanoi. It is easy to understand this problem and its solution. It is also easy to write a program to solve this problem. But, it takes an extremely long time to run for any "reasonably" large value of n . Try running a program to solve Towers of Hanoi for only 30 disks!

The Towers of Hanoi problem takes exponential time, that is, its running time is $\Theta(2^n)$. This is radically different from an algorithm that takes $\Theta(n \log n)$ time or $\Theta(n^2)$ time. It is even radically different from a problem that takes $\Theta(n^4)$ time. These are all examples of polynomial running time, because the exponents for all terms of these equations are constants. Recall from Chapter 3 that if we buy a new computer that runs twice as fast, the size of problem with complexity $\Theta(n^4)$ that we can solve in a certain amount of time is increased by the fourth root of two. In other words, there is a multiplicative factor increase, even if it is a rather small one. This is true for any algorithm whose running time can be represented by a polynomial.

Consider what happens if you buy a computer that is twice as fast and try to solve a bigger Towers of Hanoi problem in a given amount of time. Because its complexity is $\Theta(2^n)$, we can solve a problem only one disk bigger! There is no multiplicative factor, and this is true for any exponential algorithm: A constant factor increase in processing power results in only a fixed addition in problem-solving power.

There are a number of other fundamental differences between polynomial running times and exponential running times that argues for treating them as qualitatively different. Polynomials are closed under composition and addition. Thus, running polynomial-time programs in sequence, or having one program with polynomial running time call another a polynomial number of times yields polynomial time. Also, all computers known are polynomially related. That is, any program that runs in polynomial time on any computer today, when transferred to any other computer, will still run in polynomial time.

There is a practical reason for recognizing a distinction. In practice, most polynomial time algorithms are "feasible" in that they can run reasonably large inputs in reasonable time. In contrast, most algorithms requiring exponential time are not practical to run even for fairly modest sizes of input. One could argue that a program with high polynomial degree (such as n^{100}) is not practical, while an

exponential-time program with cost 1.001^n is practical. But the reality is that we know of almost no problems where the best polynomial-time algorithm has high degree (they nearly all have degree four or less), while almost no exponential-time algorithms (whose cost is $O(c^n)$) have their constant c close to one. So there is not much gray area between polynomial and exponential time algorithms in practice.

For the rest of this chapter, we define a **hard algorithm** to be one that runs in exponential time, that is, in $\Omega(c^n)$ for some constant $c > 1$. A definition for a hard *problem* will be presented in the next section.

17.2.1 The Theory of \mathcal{NP} -Completeness

Imagine a magical computer that works by guessing the correct solution from among all of the possible solutions to a problem. Another way to look at this is to imagine a super parallel computer that could test all possible solutions simultaneously. Certainly this magical (or highly parallel) computer can do anything a normal computer can do. It might also solve some problems more quickly than a normal computer can. Consider some problem where, given a guess for a solution, checking the solution to see if it is correct can be done in polynomial time. Even if the number of possible solutions is exponential, any given guess can be checked in polynomial time (equivalently, all possible solutions are checked simultaneously in polynomial time), and thus the problem can be solved in polynomial time by our hypothetical magical computer. Another view of this concept is this: If you cannot get the answer to a problem in polynomial time by guessing the right answer and then checking it, then you cannot do it in polynomial time in any other way.

The idea of “guessing” the right answer to a problem — or checking all possible solutions in parallel to determine which is correct — is called **non-determinism**. An algorithm that works in this manner is called a **non-deterministic algorithm**, and any problem with an algorithm that runs on a non-deterministic machine in polynomial time is given a special name: It is said to be a problem in \mathcal{NP} . Thus, problems in \mathcal{NP} are those problems that can be solved in polynomial time on a non-deterministic machine.

Not all problems requiring exponential time on a regular computer are in \mathcal{NP} . For example, Towers of Hanoi is *not* in \mathcal{NP} , because it must print out $O(2^n)$ moves for n disks. A non-deterministic machine cannot “guess” and print the correct answer in less time.

On the other hand, consider the TRAVELING SALESMAN problem.

TRAVELING SALESMAN (1)

Input: A complete, directed graph G with positive distances assigned to each edge in the graph.

Output: The shortest simple cycle that includes every vertex.

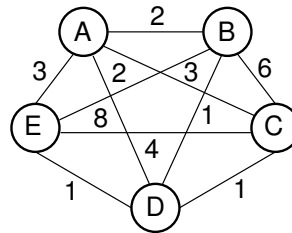


Figure 17.4 An illustration of the TRAVELING SALESMAN problem. Five vertices are shown, with edges between each pair of cities. The problem is to visit all of the cities exactly once, returning to the start city, with the least total cost.

Figure 17.4 illustrates this problem. Five vertices are shown, with edges and associated costs between each pair of edges. (For simplicity Figure 17.4 shows an undirected graph, assuming that the cost is the same in both directions, though this need not be the case.) If the salesman visits the cities in the order ABCDEA, he will travel a total distance of 13. A better route would be ABDCEA, with cost 11. The best route for this particular graph would be ABEDCA, with cost 9.

We cannot solve this problem in polynomial time with a guess-and-test non-deterministic computer. The problem is that, given a candidate cycle, while we can quickly check that the answer is indeed a cycle of the appropriate form, and while we can quickly calculate the length of the cycle, we have no easy way of knowing if it is in fact the *shortest* such cycle. However, we can solve a variant of this problem cast in the form of a **decision problem**. A decision problem is simply one whose answer is either YES or NO. The decision problem form of TRAVELING SALESMAN is as follows:

TRAVELING SALESMAN (2)

Input: A complete, directed graph G with positive distances assigned to each edge in the graph, and an integer k .

Output: YES if there is a simple cycle with total distance $\leq k$ containing every vertex in G , and NO otherwise.

We can solve this version of the problem in polynomial time with a non-deterministic computer. The non-deterministic algorithm simply checks all of the possible subsets of edges in the graph, in parallel. If any subset of the edges is an appropriate cycle of total length less than or equal to k , the answer is YES; otherwise the answer is NO. Note that it is only necessary that *some* subset meet the requirement; it does not matter how many subsets fail. Checking a particular subset is done in polynomial time by adding the distances of the edges and verifying that the edges form a cycle that visits each vertex exactly once. Thus, the checking algorithm runs in polynomial time. Unfortunately, there are $2^{|E|}$ subsets to check,

so this algorithm cannot be converted to a polynomial time algorithm on a regular computer. Nor does anybody in the world know of any other polynomial time algorithm to solve TRAVELING SALESMAN on a regular computer, despite the fact that the problem has been studied extensively by many computer scientists for many years.

It turns out that there is a large collection of problems with this property: We know efficient non-deterministic algorithms, but we do not know if there are efficient deterministic algorithms. At the same time, we have not been able to prove that any of these problems do *not* have efficient deterministic algorithms. This class of problems is called \mathcal{NP} -**complete**. What is truly strange and fascinating about \mathcal{NP} -complete problems is that if anybody ever finds the solution to any one of them that runs in polynomial time on a regular computer, then by a series of reductions, every other problem that is in \mathcal{NP} can also be solved in polynomial time on a regular computer!

Define a problem to be \mathcal{NP} -**hard** if *any* problem in \mathcal{NP} can be reduced to X in polynomial time. Thus, X is *as hard as* any problem in \mathcal{NP} . A problem X is defined to be \mathcal{NP} -complete if

1. X is in \mathcal{NP} , and
2. X is \mathcal{NP} -hard.

The requirement that a problem be \mathcal{NP} -hard might seem to be impossible, but in fact there are hundreds of such problems, including TRAVELING SALESMAN. Another such problem is called K-CLIQUE.

K-CLIQUE

Input: An arbitrary undirected graph G and an integer k .

Output: YES if there is a complete subgraph of at least k vertices, and NO otherwise.

Nobody knows whether there is a polynomial time solution for K-CLIQUE, but if such an algorithm is found for K-CLIQUE *or* for TRAVELING SALESMAN, then that solution can be modified to solve the other, or any other problem in \mathcal{NP} , in polynomial time.

The primary theoretical advantage of knowing that a problem $P1$ is \mathcal{NP} -complete is that it can be used to show that another problem $P2$ is \mathcal{NP} -complete. This is done by finding a polynomial time reduction of $P1$ to $P2$. Because we already know that all problems in \mathcal{NP} can be reduced to $P1$ in polynomial time (by the definition of \mathcal{NP} -complete), we now know that all problems can be reduced to $P2$ as well by the simple algorithm of reducing to $P1$ and then from there reducing to $P2$.

There is a practical advantage to knowing that a problem is \mathcal{NP} -complete. It relates to knowing that if a polynomial time solution can be found for *any* prob-

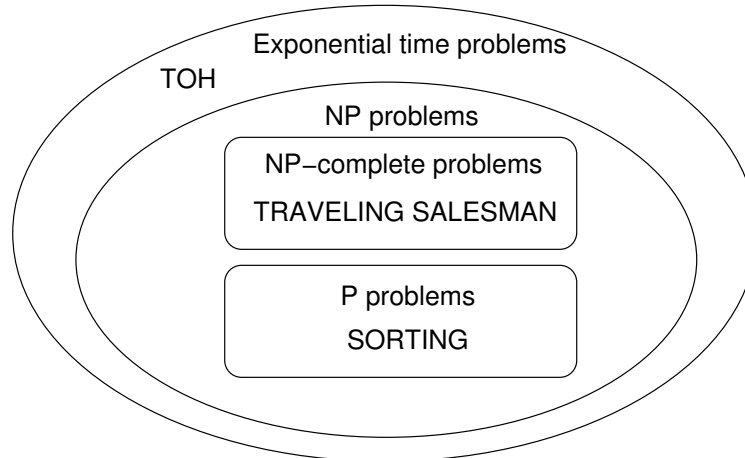


Figure 17.5 Our knowledge regarding the world of problems requiring exponential time or less. Some of these problems are solvable in polynomial time by a non-deterministic computer. Of these, some are known to be \mathcal{NP} -complete, and some are known to be solvable in polynomial time on a regular computer.

lem that is \mathcal{NP} -complete, then a polynomial solution can be found for *all* such problems. The implication is that,

1. Because no one has yet found such a solution, it must be difficult or impossible to do; and
2. Effort to find a polynomial time solution for one \mathcal{NP} -complete problem can be considered to have been expended for all \mathcal{NP} -complete problems.

How is \mathcal{NP} -completeness of practical significance for typical programmers? Well, if your boss demands that you provide a fast algorithm to solve a problem, she will not be happy if you come back saying that the best you could do was an exponential time algorithm. But, if you can prove that the problem is \mathcal{NP} -complete, while she still won't be happy, at least she should not be mad at you! By showing that her problem is \mathcal{NP} -complete, you are in effect saying that the most brilliant computer scientists for the last 50 years have been trying and failing to find a polynomial time algorithm for her problem.

Problems that are solvable in polynomial time on a regular computer are said to be in class \mathcal{P} . Clearly, all problems in \mathcal{P} are solvable in polynomial time on a non-deterministic computer simply by neglecting to use the non-deterministic capability. Some problems in \mathcal{NP} are \mathcal{NP} -complete. We can consider all problems solvable in exponential time or better as an even bigger class of problems because all problems solvable in polynomial time are solvable in exponential time. Thus, we can view the world of exponential-time-or-better problems in terms of Figure 17.5.

The most important unanswered question in theoretical computer science is whether $\mathcal{P} = \mathcal{NP}$. If they are equal, then there is a polynomial time algorithm

for TRAVELING SALESMAN and all related problems. Because TRAVELING SALESMAN is known to be \mathcal{NP} -complete, if a polynomial time algorithm were to be found for this problem, then *all* problems in \mathcal{NP} would also be solvable in polynomial time. Conversely, if we were able to prove that TRAVELING SALESMAN has an exponential time lower bound, then we would know that $\mathcal{P} \neq \mathcal{NP}$.

17.2.2 \mathcal{NP} -Completeness Proofs

To start the process of being able to prove problems are \mathcal{NP} -complete, we need to prove just one problem H is \mathcal{NP} -complete. After that, to show that any problem X is \mathcal{NP} -hard, we just need to reduce H to X . When doing \mathcal{NP} -completeness proofs, it is very important not to get this reduction backwards! If we reduce candidate problem X to known hard problem H , this means that we use H as a step to solving X . All that means is that we have found a (known) hard way to solve X . However, when we reduce known hard problem H to candidate problem X , that means we are using X as a step to solve H . And if we know that H is hard, that means X must also be hard (because if X were not hard, then neither would H be hard).

So a crucial first step to getting this whole theory off the ground is finding one problem that is \mathcal{NP} -hard. The first proof that a problem is \mathcal{NP} -hard (and because it is in \mathcal{NP} , therefore \mathcal{NP} -complete) was done by Stephen Cook. For this feat, Cook won the first Turing award, which is the closest Computer Science equivalent to the Nobel Prize. The “grand-daddy” \mathcal{NP} -complete problem that Cook used is called SATISFIABILITY (or SAT for short).

A **Boolean expression** includes Boolean variables combined using the operators AND (\cdot), OR ($+$), and NOT (to negate Boolean variable x we write \bar{x}). A **literal** is a Boolean variable or its negation. A **clause** is one or more literals OR’ed together. Let E be a Boolean expression over variables x_1, x_2, \dots, x_n . Then we define **Conjunctive Normal Form** (CNF) to be a Boolean expression written as a series of clauses that are AND’ed together. For example,

$$E = (x_5 + x_7 + \bar{x}_8 + x_{10}) \cdot (\bar{x}_2 + x_3) \cdot (x_1 + \bar{x}_3 + x_6)$$

is in CNF, and has three clauses. Now we can define the problem SAT.

SATISFIABILITY (SAT)

Input: A Boolean expression E over variables x_1, x_2, \dots in Conjunctive Normal Form.

Output: YES if there is an assignment to the variables that makes E true, NO otherwise.

Cook proved that SAT is \mathcal{NP} -hard. Explaining Cook’s proof is beyond the scope of this book. But we can briefly summarize it as follows. Any decision

problem F can be recast as some language acceptance problem L :

$$F(I) = \text{YES} \Leftrightarrow L(I') = \text{ACCEPT}.$$

That is, if a decision problem F yields YES on input I , then there is a language L containing string I' where I' is some suitable transformation of input I . Conversely, if F would give answer NO for input I , then I 's transformed version I' is not in the language L .

Turing machines are a simple model of computation for writing programs that are language acceptors. There is a “universal” Turing machine that can take as input a description for a Turing machine, and an input string, and return the execution of that machine on that string. This Turing machine in turn can be cast as a Boolean expression such that the expression is satisfiable if and only if the Turing machine yields ACCEPT for that string. Cook used Turing machines in his proof because they are simple enough that he could develop this transformation of Turing machines to Boolean expressions, but rich enough to be able to compute any function that a regular computer can compute. The significance of this transformation is that *any* decision problem that is performable by the Turing machine is transformable to SAT. Thus, SAT is \mathcal{NP} -hard.

As explained above, to show that a decision problem X is \mathcal{NP} -complete, we prove that X is in \mathcal{NP} (normally easy, and normally done by giving a suitable polynomial-time, nondeterministic algorithm) and then prove that X is \mathcal{NP} -hard. To prove that X is \mathcal{NP} -hard, we choose a known \mathcal{NP} -complete problem, say A . We describe a polynomial-time transformation that takes an *arbitrary* instance I of A to an instance I' of X . We then describe a polynomial-time transformation from SLN' to SLN such that SLN is the solution for I . The following example provides a model for how an \mathcal{NP} -completeness proof is done.

3-SATISFIABILITY (3 SAT)

Input: A Boolean expression E in CNF such that each clause contains exactly 3 literals.

Output: YES if the expression can be satisfied, NO otherwise.

Example 17.1 3 SAT is a special case of SAT. Is 3 SAT easier than SAT? Not if we can prove it to be \mathcal{NP} -complete.

Theorem 17.1 3 SAT is \mathcal{NP} -complete.

Proof: **Prove that 3 SAT is in \mathcal{NP} :** Guess (nondeterministically) truth values for the variables. The correctness of the guess can be verified in polynomial time.

Prove that 3 SAT is \mathcal{NP} -hard: We need a polynomial-time reduction from SAT to 3 SAT. Let $E = C_1 \cdot C_2 \cdot \dots \cdot C_k$ be any instance of SAT. Our

strategy is to replace any clause C_i that does not have exactly three literals with a set of clauses each having exactly three literals. (Recall that a literal can be a variable such as x , or the negation of a variable such as \bar{x} .) Let $C_i = x_1 + x_2 + \dots + x_j$ where x_1, \dots, x_j are literals.

1. $j = 1$, so $C_i = x_1$. Replace C_i with C'_i :

$$(x_1 + y + z) \cdot (x_1 + \bar{y} + z) \cdot (x_1 + y + \bar{z}) \cdot (x_1 + \bar{y} + \bar{z})$$

where y and z are variables not appearing in \mathbf{E} . Clearly, C'_i is satisfiable if and only if (x_1) is satisfiable, meaning that x_1 is **true**.

2. $j = 2$, so $C_i = (x_1 + x_2)$. Replace C_i with

$$(x_1 + x_2 + z) \cdot (x_1 + x_2 + \bar{z})$$

where z is a new variable not appearing in \mathbf{E} . This new pair of clauses is satisfiable if and only if $(x_1 + x_2)$ is satisfiable, that is, either x_1 or x_2 must be true.

3. $j > 3$. Replace $C_i = (x_1 + x_2 + \dots + x_j)$ with

$$(x_1 + x_2 + z_1) \cdot (x_3 + \bar{z}_1 + z_2) \cdot (x_4 + \bar{z}_2 + z_3) \cdot \dots \\ \cdot (x_{j-2} + \bar{z}_{j-4} + z_{j-3}) \cdot (x_{j-1} + x_j + \bar{z}_{j-3})$$

where z_1, \dots, z_{j-3} are new variables.

After appropriate replacements have been made for each C_i , a Boolean expression results that is an instance of 3 SAT. Each replacement is satisfiable if and only if the original clause is satisfiable. The reduction is clearly polynomial time.

For the first two cases it is fairly easy to see that the original clause is satisfiable if and only if the resulting clauses are satisfiable. For the case where we replaced a clause with more than three literals, consider the following.

1. If E is satisfiable, then E' is satisfiable: Assume x_m is assigned **true**. Then assign $z_t, t \leq m - 2$ as **true** and $z_k, t \geq m - 1$ as **false**. Then all clauses in Case (3) are satisfied.
2. If x_1, x_2, \dots, x_j are all **false**, then z_1, z_2, \dots, z_{j-3} are all **true**. But then $(x_{j-1} + x_{j-2} + \bar{z}_{j-3})$ is **false**.

□

Next we define the problem VERTEX COVER for use in further examples.

VERTEX COVER:

Input: A graph \mathbf{G} and an integer k .

Output: YES if there is a subset \mathbf{S} of the vertices in \mathbf{G} of size k or less such that every edge of \mathbf{G} has at least one of its endpoints in \mathbf{S} , and NO otherwise.

Example 17.2 In this example, we make use of a simple conversion between two graph problems.

Theorem 17.2 *VERTEX COVER is \mathcal{NP} -complete.*

Proof: Prove that VERTEX COVER is in \mathcal{NP} : Simply guess a subset of the graph and determine in polynomial time whether that subset is in fact a vertex cover of size k or less.

Prove that VERTEX COVER is \mathcal{NP} -hard: We will assume that K-CLIQUE is already known to be \mathcal{NP} -complete. (We will see this proof in the next example. For now, just accept that it is true.)

Given that K-CLIQUE is \mathcal{NP} -complete, we need to find a polynomial-time transformation from the input to K-CLIQUE to the input to VERTEX COVER, and another polynomial-time transformation from the output for VERTEX COVER to the output for K-CLIQUE. This turns out to be a simple matter, given the following observation. Consider a graph \mathbf{G} and a vertex cover \mathbf{S} on \mathbf{G} . Denote by \mathbf{S}' the set of vertices in \mathbf{G} but not in \mathbf{S} . There can be no edge connecting any two vertices in \mathbf{S}' because, if there were, then \mathbf{S} would not be a vertex cover. Denote by \mathbf{G}' the inverse graph for \mathbf{G} , that is, the graph formed from the edges not in \mathbf{G} . If \mathbf{S} is of size k , then \mathbf{S}' forms a clique of size $n - k$ in graph \mathbf{G}' . Thus, we can reduce K-CLIQUE to VERTEX COVER simply by converting graph \mathbf{G} to \mathbf{G}' , and asking if \mathbf{G}' has a VERTEX COVER of size $n - k$ or smaller. If YES, then there is a clique in \mathbf{G} of size k ; if NO then there is not. \square

Example 17.3 So far, our \mathcal{NP} -completeness proofs have involved transformations between inputs of the same “type,” such as from a Boolean expression to a Boolean expression or from a graph to a graph. Sometimes an \mathcal{NP} -completeness proof involves a transformation between types of inputs, as shown next.

Theorem 17.3 *K-CLIQUE is \mathcal{NP} -complete.*

Proof: K-CLIQUE is in \mathcal{NP} , because we can just guess a collection of k vertices and test in polynomial time if it is a clique. Now we show that K-CLIQUE is \mathcal{NP} -hard by using a reduction from SAT. An instance of SAT is a Boolean expression

$$B = C_1 \cdot C_2 \cdot \dots \cdot C_m$$

whose clauses we will describe by the notation

$$C_i = y[i, 1] + y[i, 2] + \dots + y[i, k_i]$$

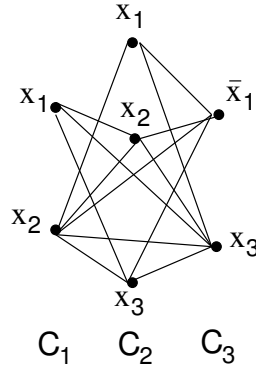


Figure 17.6 The graph generated from Boolean expression $B = (x_1 + x_2) \cdot (\overline{x_1} + x_2 + x_3) \cdot (\overline{x_1} + x_3)$. Literals from the first clause are labeled C_1 , and literals from the second clause are labeled C_2 . There is an edge between every pair of vertices except when both vertices represent instances of literals from the same clause, or a negation of the same variable. Thus, the vertex labeled $C_1 : y_1$ does not connect to the vertex labeled $C_1 : y_2$ (because they are literals in the same clause) or the vertex labeled $C_2 : \overline{y_1}$ (because they are opposite values for the same variable).

where k_i is the number of literals in Clause c_i . We will transform this to an instance of K-CLIQUE as follows. We build a graph

$$G = \{v[i, j] \mid 1 \leq i \leq m, 1 \leq j \leq k_i\},$$

that is, there is a vertex in \mathbf{G} corresponding to every literal in Boolean expression \mathbf{B} . We will draw an edge between each pair of vertices $v[i_1, j_1]$ and $v[i_2, j_2]$ unless (1) they are two literals within the same clause ($i_1 = i_2$) or (2) they are opposite values for the same variable (i.e., one is negated and the other is not). Set $k = m$. Figure 17.6 shows an example of this transformation.

\mathbf{B} is satisfiable if and only if \mathbf{G} has a clique of size k or greater. \mathbf{B} being satisfiable implies that there is a truth assignment such that at least one literal $y[i, j_i]$ is true for each i . If so, then these m literals must correspond to m vertices in a clique of size $k = m$. Conversely, if \mathbf{G} has a clique of size k or greater, then the clique must have size exactly k (because no two vertices corresponding to literals in the same clause can be in the clique) and there is one vertex $v[i, j_i]$ in the clique for each i . There is a truth assignment making each $y[i, j_i]$ true. That truth assignment satisfies \mathbf{B} .

We conclude that K-CLIQUE is \mathcal{NP} -hard, therefore \mathcal{NP} -complete. \square

17.2.3 Coping with \mathcal{NP} -Complete Problems

Finding that your problem is \mathcal{NP} -complete might not mean that you can just forget about it. Traveling salesmen need to find reasonable sales routes regardless of the complexity of the problem. What do you do when faced with an \mathcal{NP} -complete problem that you must solve?

There are several techniques to try. One approach is to run only small instances of the problem. For some problems, this is not acceptable. For example, TRAVELING SALESMAN grows so quickly that it cannot be run on modern computers for problem sizes much over 30 cities, which is not an unreasonable problem size for real-life situations. However, some other problems in \mathcal{NP} , while requiring exponential time, still grow slowly enough that they allow solutions for problems of a useful size.

Consider the Knapsack problem from Section 16.1.1. We have a dynamic programming algorithm whose cost is $\Theta(nK)$ for n objects being fit into a knapsack of size K . But it turns out that Knapsack is \mathcal{NP} -complete. Isn't this a contradiction? Not when we consider the relationship between n and K . How big is K ? Input size is typically $O(n \lg K)$ because the item sizes are smaller than K . Thus, $\Theta(nK)$ is exponential on input size.

This dynamic programming algorithm is tractable if the numbers are “reasonable.” That is, we can successfully find solutions to the problem when nK is in the thousands. Such an algorithm is called a **pseudo-polynomial** time algorithm. This is different from TRAVELING SALESMAN which cannot possibly be solved when $n = 100$ given current algorithms.

A second approach to handling \mathcal{NP} -complete problems is to solve a special instance of the problem that is not so hard. For example, many problems on graphs are \mathcal{NP} -complete, but the same problem on certain restricted types of graphs is not as difficult. For example, while the VERTEX COVER and K-CLIQUE problems are \mathcal{NP} -complete in general, there are polynomial time solutions for bipartite graphs (i.e., graphs whose vertices can be separated into two subsets such that no pair of vertices within one of the subsets has an edge between them). 2-SATISFIABILITY (where every clause in a Boolean expression has at most two literals) has a polynomial time solution. Several geometric problems require only polynomial time in two dimensions, but are \mathcal{NP} -complete in three dimensions or more. KNAPSACK is considered to run in polynomial time if the numbers (and K) are “small.” Small here means that they are polynomial on n , the number of items.

In general, if we want to guarantee that we get the correct answer for an \mathcal{NP} -complete problem, we potentially need to examine all of the (exponential number of) possible solutions. However, with some organization, we might be able to either examine them quickly, or avoid examining a great many of the possible answers in some cases. For example, Dynamic Programming (Section 16.1) attempts to

organize the processing of all the subproblems to a problem so that the work is done efficiently.

If we need to do a brute-force search of the entire solution space, we can use **backtracking** to visit all of the possible solutions organized in a solution tree. For example, SATISFIABILITY has 2^n possible ways to assign truth values to the n variables contained in the Boolean expression being satisfied. We can view this as a tree of solutions by considering that we have a choice of making the first variable **true** or **false**. Thus, we can put all solutions where the first variable is **true** on one side of the tree, and the remaining solutions on the other. We then examine the solutions by moving down one branch of the tree, until we reach a point where we know the solution cannot be correct (such as if the current partial collection of assignments yields an unsatisfiable expression). At this point we backtrack and move back up a node in the tree, and then follow down the alternate branch. If this fails, we know to back up further in the tree as necessary and follow alternate branches, until finally we either find a solution that satisfies the expression or exhaust the tree. In some cases we avoid processing many potential solutions, or find a solution quickly. In others, we end up visiting a large portion of the 2^n possible solutions.

Banch-and-Bounds is an extension of backtracking that applies to **optimization problems** such as TRAVELING SALESMAN where we are trying to find the shortest tour through the cities. We traverse the solution tree as with backtracking. However, we remember the best value found so far. Proceeding down a given branch is equivalent to deciding which order to visit cities. So any node in the solution tree represents some collection of cities visited so far. If the sum of these distances exceeds the best tour found so far, then we know to stop pursuing this branch of the tree. At this point we can immediately back up and take another branch. If we have a quick method for finding a good (but not necessarily best) solution, we can use this as an initial bound value to effectively prune portions of the tree.

Another coping strategy is to find an approximate solution to the problem. There are many approaches to finding approximate solutions. One way is to use a heuristic to solve the problem, that is, an algorithm based on a “rule of thumb” that does not always give the best answer. For example, the TRAVELING SALESMAN problem can be solved approximately by using the heuristic that we start at an arbitrary city and then always proceed to the next unvisited city that is closest. This rarely gives the shortest path, but the solution might be good enough. There are many other heuristics for TRAVELING SALESMAN that do a better job.

Some approximation algorithms have guaranteed performance, such that the answer will be within a certain percentage of the best possible answer. For example, consider this simple heuristic for the VERTEX COVER problem: Let M be a maximal (not necessarily maximum) **matching** in G . A matching pairs vertices (with connecting edges) so that no vertex is paired with more than one partner.

Maximal means to pick as many pairs as possible, selecting them in some order until there are no more available pairs to select. Maximum means the matching that gives the most pairs possible for a given graph. If OPT is the size of a minimum vertex cover, then $|M| \leq 2 \cdot \text{OPT}$ because at least one endpoint of every matched edge must be in *any* vertex cover.

A better example of a guaranteed bound on a solution comes from simple heuristics to solve the BIN PACKING problem.

BIN PACKING:

Input: Numbers x_1, x_2, \dots, x_n between 0 and 1, and an unlimited supply of bins of size 1 (no bin can hold numbers whose sum exceeds 1).

Output: An assignment of numbers to bins that requires the fewest possible bins.

BIN PACKING in its decision form (i.e., asking if the items can be packed in less than k bins) is known to be \mathcal{NP} -complete. One simple heuristic for solving this problem is to use a “first fit” approach. We put the first number in the first bin. We then put the second number in the first bin if it fits, otherwise we put it in the second bin. For each subsequent number, we simply go through the bins in the order we generated them and place the number in the first bin that fits. The number of bins used is no more than twice the sum of the numbers, because every bin (except perhaps one) must be at least half full. However, this “first fit” heuristic can give us a result that is much worse than optimal. Consider the following collection of numbers: 6 of $1/7 + \epsilon$, 6 of $1/3 + \epsilon$, and 6 of $1/2 + \epsilon$, where ϵ is a small, positive number. Properly organized, this requires 6 bins. But if done wrongly, we might end up putting the numbers into 10 bins.

A better heuristic is to use decreasing first fit. This is the same as first fit, except that we keep the bins sorted from most full to least full. Then when deciding where to put the next item, we place it in the fullest bin that can hold it. This is similar to the “best fit” heuristic for memory management discussed in Section 12.3. The significant thing about this heuristic is not just that it tends to give better performance than simple first fit. This decreasing first fit heuristic can be proven to require no more than $11/9$ the optimal number of bins. Thus, we have a guarantee on how much inefficiency can result when using the heuristic.

The theory of \mathcal{NP} -completeness gives a technique for separating tractable from (probably) intractable problems. Recalling the algorithm for generating algorithms in Section 15.1, we can refine it for problems that we suspect are \mathcal{NP} -complete. When faced with a new problem, we might alternate between checking if it is tractable (that is, we try to find a polynomial-time solution) and checking if it is intractable (we try to prove the problem is \mathcal{NP} -complete). While proving that some problem is \mathcal{NP} -complete does not actually make our upper bound for our

algorithm match the lower bound for the problem with certainty, it is nearly as good. Once we realize that a problem is \mathcal{NP} -complete, then we know that our next step must either be to redefine the problem to make it easier, or else use one of the “coping” strategies discussed in this section.

17.3 Impossible Problems

Even the best programmer sometimes writes a program that goes into an infinite loop. Of course, when you run a program that has not stopped, you do not know for sure if it is just a slow program or a program in an infinite loop. After “enough time,” you shut it down. Wouldn’t it be great if your compiler could look at your program and tell you before you run it that it will get into an infinite loop? To be more specific, given a program and a particular input, it would be useful to know if executing the program on that input will result in an infinite loop without actually running the program.

Unfortunately, the **Halting Problem**, as this is called, cannot be solved. There will never be a computer program that can positively determine, for an arbitrary program **P**, if **P** will halt for all input. Nor will there even be a computer program that can positively determine if arbitrary program **P** will halt for a specified input *I*. How can this be? Programmers look at programs regularly to determine if they will halt. Surely this can be automated. As a warning to those who believe any program can be analyzed in this way, carefully examine the following code fragment before reading on.

```
while (n > 1)
    if (ODD(n))
        n = 3 * n + 1;
    else
        n = n / 2;
```

This is a famous piece of code. The sequence of values that is assigned to *n* by this code is sometimes called the **Collatz sequence** for input value *n*. Does this code fragment halt for all values of *n*? Nobody knows the answer. Every input that has been tried halts. But does it always halt? Note that for this code fragment, because we do not know if it halts, we also do not know an upper bound for its running time. As for the lower bound, we can easily show $\Omega(\log n)$ (see Exercise 3.14).

Personally, I have faith that someday some smart person will completely analyze the Collatz function, proving once and for all that the code fragment halts for all values of *n*. Doing so may well give us techniques that advance our ability to do algorithm analysis in general. Unfortunately, proofs from **computability** — the branch of computer science that studies what is impossible to do with a computer — compel us to believe that there will always be another bit of program code that

we cannot analyze. This comes as a result of the fact that the Halting Problem is unsolvable.

17.3.1 Uncountability

Before proving that the Halting Problem is unsolvable, we first prove that not all functions can be implemented as a computer program. This must be so because the number of programs is much smaller than the number of possible functions.

A set is said to be **countable** (or **countably infinite** if it is a set with an infinite number of members) if every member of the set can be uniquely assigned to a positive integer. A set is said to be **uncountable** (or **uncountably infinite**) if it is not possible to assign every member of the set to its own positive integer.

To understand what is meant when we say “assigned to a positive integer,” imagine that there is an infinite row of bins, labeled 1, 2, 3, and so on. Take a set and start placing members of the set into bins, with at most one member per bin. If we can find a way to assign all of the set members to bins, then the set is countable. For example, consider the set of positive even integers 2, 4, and so on. We can assign an integer i to bin $i/2$ (or, if we don’t mind skipping some bins, then we can assign even number i to bin i). Thus, the set of even integers is countable. This should be no surprise, because intuitively there are “fewer” positive even integers than there are positive integers, even though both are infinite sets. But there are not really any more positive integers than there are positive even integers, because we can uniquely assign every positive integer to some positive even integer by simply assigning positive integer i to positive even integer $2i$.

On the other hand, the set of all integers is also countable, even though this set appears to be “bigger” than the set of positive integers. This is true because we can assign 0 to positive integer 1, 1 to positive integer 2, -1 to positive integer 3, 2 to positive integer 4, -2 to positive integer 5, and so on. In general, assign positive integer value i to positive integer value $2i$, and assign negative integer value $-i$ to positive integer value $2i + 1$. We will never run out of positive integers to assign, and we know exactly which positive integer every integer is assigned to. Because every integer gets an assignment, the set of integers is countably infinite.

Are the number of programs countable or uncountable? A program can be viewed as simply a string of characters (including special punctuation, spaces, and line breaks). Let us assume that the number of different characters that can appear in a program is P . (Using the ASCII character set, P must be less than 128, but the actual number does not matter). If the number of strings is countable, then surely the number of programs is also countable. We can assign strings to the bins as follows. Assign the null string to the first bin. Now, take all strings of one character, and assign them to the next P bins in “alphabetic” or ASCII code order. Next, take all strings of two characters, and assign them to the next P^2 bins, again in ASCII code order working from left to right. Strings of three characters

are likewise assigned to bins, then strings of length four, and so on. In this way, a string of any given length can be assigned to some bin.

By this process, any string of finite length is assigned to some bin. So any program, which is merely a string of finite length, is assigned to some bin. Because all programs are assigned to some bin, the set of all programs is countable. Naturally most of the strings in the bins are not legal programs, but this is irrelevant. All that matters is that the strings that *do* correspond to programs are also in the bins.

Now we consider the number of possible functions. To keep things simple, assume that all functions take a single positive integer as input and yield a single positive integer as output. We will call such functions **integer functions**. A function is simply a mapping from input values to output values. Of course, not all computer programs literally take integers as input and yield integers as output. However, everything that computers read and write is essentially a series of numbers, which may be interpreted as letters or something else. Any useful computer program's input and output can be coded as integer values, so our simple model of computer input and output is sufficiently general to cover all possible computer programs.

We now wish to see if it is possible to assign all of the integer functions to the infinite set of bins. If so, then the number of functions is countable, and it might then be possible to assign every integer function to a program. If the set of integer functions cannot be assigned to bins, then there will be integer functions that must have no corresponding program.

Imagine each integer function as a table with two columns and an infinite number of rows. The first column lists the positive integers starting at 1. The second column lists the output of the function when given the value in the first column as input. Thus, the table explicitly describes the mapping from input to output for each function. Call this a **function table**.

Next we will try to assign function tables to bins. To do so we must order the functions, but it does not matter what order we choose. For example, Bin 1 could store the function that always returns 1 regardless of the input value. Bin 2 could store the function that returns its input. Bin 3 could store the function that doubles its input and adds 5. Bin 4 could store a function for which we can see no simple relationship between input and output.² These four functions as assigned to the first four bins are shown in Figure 17.7.

Can we assign every function to a bin? The answer is no, because there is always a way to create a new function that is not in any of the bins. Suppose that somebody presents a way of assigning functions to bins that they claim includes all of the functions. We can build a new function that has not been assigned to

²There is no requirement for a function to have any discernible relationship between input and output. A function is simply a mapping of inputs to outputs, with no constraint on how the mapping is determined.

1		2		3		4		5	
x	$f_1(x)$	x	$f_2(x)$	x	$f_3(x)$	x	$f_4(x)$		
1	1	1	1	1	7	1	15		
2	1	2	2	2	9	2	1		
3	1	3	3	3	11	3	7		
4	1	4	4	4	13	4	13		
5	1	5	5	5	15	5	2		
6	1	6	6	6	17	6	7		
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots		

Figure 17.7 An illustration of assigning functions to bins.

1		2		3		4		5			
x	$f_1(x)$	x	$f_2(x)$	x	$f_3(x)$	x	$f_4(x)$			x	$f_{\text{new}}(x)$
1	①	1	1	1	7	1	15			1	→ 2
2	1	2	②	2	9	2	1			2	→ 3
3	1	3	3	3	⑪	3	7			3	→ 12
4	1	4	4	4	13	4	⑬			4	→ 14
5	1	5	5	5	15	5	2			5	→
6	1	6	6	6	17	6	7			6	
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\vdots			\vdots	\vdots

Figure 17.8 Illustration for the argument that the number of integer functions is uncountable.

any bin, as follows. Take the output value for input 1 from the function in the first bin. Call this value $F_1(1)$. Add 1 to it, and assign the result as the output of a new function for input value 1. Regardless of the remaining values assigned to our new function, it must be different from the first function in the table, because the two give different outputs for input 1. Now take the output value for 2 from the second function in the table (known as $F_2(2)$). Add 1 to this value and assign it as the output for 2 in our new function. Thus, our new function must be different from the function of Bin 2, because they will differ at least at the second value. Continue in this manner, assigning $F_{\text{new}}(i) = F_i(i) + 1$ for all values i . Thus, the new function must be different from any function F_i at least at position i . This procedure for constructing a new function not already in the table is called **diagonalization**. Because the new function is different from every other function, it must not be in the table. This is true no matter how we try to assign functions to bins, and so the number of integer functions is uncountable. The significance of this is that not all functions can possibly be assigned to programs, so there *must* be functions with no corresponding program. Figure 17.8 illustrates this argument.

17.3.2 The Halting Problem Is Unsolvable

While there might be intellectual appeal to knowing that there exists *some* function that cannot be computed by a computer program, does this mean that there is any such *useful* function? After all, does it really matter if no program can compute a “nonsense” function such as shown in Bin 4 of Figure 17.7? Now we will prove that the Halting Problem cannot be computed by any computer program. The proof is by contradiction.

We begin by assuming that there is a function named **halt** that can solve the Halting Problem. Obviously, it is not possible to write out something that does not exist, but here is a plausible sketch of what a function to solve the Halting Problem might look like if it did exist. Function **halt** takes two inputs: a string representing the source code for a program or function, and another string representing the input that we wish to determine if the input program or function halts on. Function **halt** does some work to make a decision (which is encapsulated into some fictitious function named **PROGRAM HALTS**). Function **halt** then returns **true** if the input program or function does halt on the given input, and **false** otherwise.

```
bool halt(string prog, string input) {
    if (PROGRAM_HALTS(prog, input))
        return true;
    else
        return false;
}
```

We now will examine two simple functions that clearly can exist because the complete code for them is presented here:

```
// Return true if "prog" halts when given itself as input
bool selfhalt(char *prog) {
    if (halt(prog, prog))
        return true;
    else
        return false;
}

// Return the reverse of what selfhalt returns on "prog"
void contrary(char *prog) {
    if (selfhalt(prog))
        while (true); // Go into an infinite loop
}
```

What happens if we make a program whose sole purpose is to execute the function **contrary** and run that program with itself as input? One possibility is that the call to **selfhalt** returns **true**; that is, **selfhalt** claims that **contrary** will halt when run on itself. In that case, **contrary** goes into an infinite loop (and thus does not halt). On the other hand, if **selfhalt** returns **false**, then **halt** is proclaiming that **contrary** does not halt on itself, and **contrary** then returns,

that is, it halts. Thus, **contrary** does the contrary of what **halt** says that it will do.

The action of **contrary** is logically inconsistent with the assumption that **halt** solves the Halting Problem correctly. There are no other assumptions we made that might cause this inconsistency. Thus, by contradiction, we have proved that **halt** cannot solve the Halting Problem correctly, and thus there is no program that can solve the Halting Problem.

Now that we have proved that the Halting Problem is unsolvable, we can use reduction arguments to prove that other problems are also unsolvable. The strategy is to assume the existence of a computer program that solves the problem in question and use that program to solve another problem that is already known to be unsolvable.

Example 17.4 Consider the following variation on the Halting Problem. Given a computer program, will it halt when its input is the empty string? That is, will it halt when it is given no input? To prove that this problem is unsolvable, we will employ a standard technique for computability proofs: Use a computer program to modify another computer program.

Proof: Assume that there is a function **Ehalt** that determines whether a given program halts when given no input. Recall that our proof for the Halting Problem involved functions that took as parameters a string representing a program and another string representing an input. Consider another function **combine** that takes a program P and an input string I as parameters. Function **combine** modifies P to store I as a static variable S and further modifies all calls to input functions within P to instead get their input from S . Call the resulting program P' . It should take no stretch of the imagination to believe that any decent compiler could be modified to take computer programs and input strings and produce a new computer program that has been modified in this way. Now, take P' and feed it to **Ehalt**. If **Ehalt** says that P' will halt, then we know that P would halt on input I . In other words, we now have a procedure for solving the original Halting Problem. The only assumption that we made was the existence of **Ehalt**. Thus, the problem of determining if a program will halt on no input must be unsolvable. □

Example 17.5 For arbitrary program P , does there exist *any* input for which P halts?

Proof: This problem is also uncomputable. Assume that we had a function **Ahalt** that, when given program P as input would determine if there is some input for which P halts. We could modify our compiler (or write

a function as part of a program) to take P and some input string w , and modify it so that w is hardcoded inside P , with P reading no input. Call this modified program P' . Now, P' always behaves the same way regardless of its input, because it ignores all input. However, because w is now hardwired inside of P' , the behavior we get is that of P when given w as input. So, P' will halt on any arbitrary input if and only if P would halt on input w . We now feed P' to function **Ahalt**. If **Ahalt** could determine that P' halts on some input, then that is the same as determining that P halts on input w . But we know that that is impossible. Therefore, **Ahalt** cannot exist. \square

There are many things that we would like to have a computer do that are unsolvable. Many of these have to do with program behavior. For example, proving that an arbitrary program is “correct,” that is, proving that a program computes a particular function, is a proof regarding program behavior. As such, what can be accomplished is severely limited. Some other unsolvable problems include:

- Does a program halt on every input?
- Does a program compute a particular function?
- Do two programs compute the same function?
- Does a particular line in a program get executed?

This does *not* mean that a computer program cannot be written that works on special cases, possibly even on most programs that we would be interested in checking. For example, some **C** compilers will check if the control expression for a **while** loop is a constant expression that evaluates to **false**. If it is, the compiler will issue a warning that the **while** loop code will never be executed. However, it is not possible to write a computer program that can check for *all* input programs whether a specified line of code will be executed when the program is given some specified input.

Another unsolvable problem is whether a program contains a computer virus. The property “contains a computer virus” is a matter of behavior. Thus, it is not possible to determine positively whether an arbitrary program contains a computer virus. Fortunately, there are many good heuristics for determining if a program is likely to contain a virus, and it is usually possible to determine if a program contains a particular virus, at least for the ones that are now known. Real virus checkers do a pretty good job, but, it will always be possible for malicious people to invent new viruses that no existing virus checker can recognize.

17.4 Further Reading

The classic text on the theory of \mathcal{NP} -completeness is *Computers and Intractability: A Guide to the Theory of \mathcal{NP} -completeness* by Garey and Johnston [GJ79]. *The Traveling Salesman Problem*, edited by Lawler et al. [LLKS85], discusses

many approaches to finding an acceptable solution to this particular \mathcal{NP} -complete problem in a reasonable amount of time.

For more information about the Collatz function see “On the Ups and Downs of Hailstone Numbers” by B. Hayes [Hay84], and “The $3x + 1$ Problem and its Generalizations” by J.C. Lagarias [Lag85].

For an introduction to the field of computability and impossible problems, see *Discrete Structures, Logic, and Computability* by James L. Hein [Hei09].

17.5 Exercises

- 17.1** Consider this algorithm for finding the maximum element in an array: First sort the array and then select the last (maximum) element. What (if anything) does this reduction tell us about the upper and lower bounds to the problem of finding the maximum element in a sequence? Why can we not reduce SORTING to finding the maximum element?
- 17.2** Use a reduction to prove that squaring an $n \times n$ matrix is just as expensive (asymptotically) as multiplying two $n \times n$ matrices.
- 17.3** Use a reduction to prove that multiplying two upper triangular $n \times n$ matrices is just as expensive (asymptotically) as multiplying two arbitrary $n \times n$ matrices.
- 17.4** (a) Explain why computing the factorial of n by multiplying all values from 1 to n together is an exponential time algorithm.
 (b) Explain why computing an approximation to the factorial of n by making use of Stirling’s formula (see Section 2.2) is a polynomial time algorithm.
- 17.5** Consider this algorithm for solving the K-CLIQUE problem. First, generate all subsets of the vertices containing exactly k vertices. There are $O(n^k)$ such subsets altogether. Then, check whether any subgraphs induced by these subsets is complete. If this algorithm ran in polynomial time, what would be its significance? Why is this not a polynomial-time algorithm for the K-CLIQUE problem?
- 17.6** Write the 3 SAT expression obtained from the reduction of SAT to 3 SAT described in Section 17.2.1 for the expression

$$(a + b + \bar{c} + d) \cdot (\bar{d}) \cdot (\bar{b} + \bar{c}) \cdot (\bar{a} + b) \cdot (a + c) \cdot (b).$$

Is this expression satisfiable?

- 17.7** Draw the graph obtained by the reduction of SAT to the K-CLIQUE problem given in Section 17.2.1 for the expression

$$(a + \bar{b} + c) \cdot (\bar{a} + b + \bar{c}) \cdot (\bar{a} + b + c) \cdot (a + \bar{b} + \bar{c}).$$

Is this expression satisfiable?

17.8 A **Hamiltonian cycle** in graph G is a cycle that visits every vertex in the graph exactly once before returning to the start vertex. The problem **HAMILTONIAN CYCLE** asks whether graph G does in fact contain a Hamiltonian cycle. Assuming that **HAMILTONIAN CYCLE** is \mathcal{NP} -complete, prove that the decision-problem form of **TRAVELING SALESMAN** is \mathcal{NP} -complete.

17.9 Use the assumption that **VERTEX COVER** is \mathcal{NP} -complete to prove that **K-CLIQUE** is also \mathcal{NP} -complete by finding a polynomial time reduction from **VERTEX COVER** to **K-CLIQUE**.

17.10 We define the problem **INDEPENDENT SET** as follows.

INDEPENDENT SET

Input: A graph G and an integer k .

Output: YES if there is a subset S of the vertices in G of size k or greater such that no edge connects any two vertices in S , and NO otherwise.

Assuming that **K-CLIQUE** is \mathcal{NP} -complete, prove that **INDEPENDENT SET** is \mathcal{NP} -complete.

17.11 Define the problem **PARTITION** as follows:

PARTITION

Input: A collection of integers.

Output: YES if the collection can be split into two such that the sum of the integers in each partition sums to the same amount. NO otherwise.

(a) Assuming that **PARTITION** is \mathcal{NP} -complete, prove that the decision form of **BIN PACKING** is \mathcal{NP} -complete.

(b) Assuming that **PARTITION** is \mathcal{NP} -complete, prove that **KNAPSACK** is \mathcal{NP} -complete.

17.12 Imagine that you have a problem P that you know is \mathcal{NP} -complete. For this problem you have two algorithms to solve it. For each algorithm, some problem instances of P run in polynomial time and others run in exponential time (there are lots of heuristic-based algorithms for real \mathcal{NP} -complete problems with this behavior). You can't tell beforehand for any given problem instance whether it will run in polynomial or exponential time on either algorithm. However, you do know that for every problem instance, at least one of the two algorithms will solve it in polynomial time.

(a) What should you do?

(b) What is the running time of your solution?

- (c) What does it say about the question of $\mathcal{P} = \mathcal{NP}$ if the conditions described in this problem existed?
- 17.13** Here is another version of the knapsack problem, which we will call EXACT KNAPSACK. Given a set of items each with given integer size, and a knapsack of size integer k , is there a subset of the items which fits exactly within the knapsack?
Assuming that EXACT KNAPSACK is \mathcal{NP} -complete, use a reduction argument to prove that KNAPSACK is \mathcal{NP} -complete.
- 17.14** The last paragraph of Section 17.2.3 discusses a strategy for developing a solution to a new problem by alternating between finding a polynomial time solution and proving the problem \mathcal{NP} -complete. Refine the “algorithm for designing algorithms” from Section 15.1 to incorporate identifying and dealing with \mathcal{NP} -complete problems.
- 17.15** Prove that the set of real numbers is uncountable. Use a proof similar to the proof in Section 17.3.1 that the set of integer functions is uncountable.
- 17.16** Prove, using a reduction argument such as given in Section 17.3.2, that the problem of determining if an arbitrary program will print any output is unsolvable.
- 17.17** Prove, using a reduction argument such as given in Section 17.3.2, that the problem of determining if an arbitrary program executes a particular statement within that program is unsolvable.
- 17.18** Prove, using a reduction argument such as given in Section 17.3.2, that the problem of determining if two arbitrary programs halt on exactly the same inputs is unsolvable.
- 17.19** Prove, using a reduction argument such as given in Section 17.3.2, that the problem of determining whether there is some input on which two arbitrary programs will both halt is unsolvable.
- 17.20** Prove, using a reduction argument such as given in Section 17.3.2, that the problem of determining whether an arbitrary program halts on all inputs is unsolvable.
- 17.21** Prove, using a reduction argument such as given in Section 17.3.2, that the problem of determining whether an arbitrary program computes a specified function is unsolvable.
- 17.22** Consider a program named COMP that takes two strings as input. It returns **TRUE** if the strings are the same. It returns **FALSE** if the strings are different. Why doesn’t the argument that we used to prove that a program to solve the halting problem does not exist work to prove that COMP does not exist?

17.6 Projects

- 17.1** Implement VERTEX COVER; that is, given graph G and integer k , answer the question of whether or not there is a vertex cover of size k or less. Begin

by using a brute-force algorithm that checks all possible sets of vertices of size k to find an acceptable vertex cover, and measure the running time on a number of input graphs. Then try to reduce the running time through the use of any heuristics you can think of. Next, try to find approximate solutions to the problem in the sense of finding the smallest set of vertices that forms a vertex cover.

- 17.2** Implement KNAPSACK (see Section 16.1). Measure its running time on a number of inputs. What is the largest practical input size for this problem?
- 17.3** Implement an approximation of TRAVELING SALESMAN; that is, given a graph G with costs for all edges, find the cheapest cycle that visits all vertices in G . Try various heuristics to find the best approximations for a wide variety of input graphs.
- 17.4** Write a program that, given a positive integer n as input, prints out the Collatz sequence for that number. What can you say about the types of integers that have long Collatz sequences? What can you say about the length of the Collatz sequence for various types of integers?

PART VI

APPENDIX

A

Utility Functions

Here are various utility functions used by the C++ example programs in this text.

```
// Return true iff "x" is even
inline bool EVEN(int x) { return (x % 2) == 0; }

// Return true iff "x" is odd
inline bool ODD(int x) { return (x % 2) != 0; }

// Assert: If "val" is false, print a message and terminate
// the program
void Assert(bool val, string s) {
    if (!val) { // Assertion failed -- close the program
        cout << "Assertion Failed: " << s << endl;
        exit(-1);
    }
}

// Swap two elements in a generic array
template<typename E>
inline void swap(E A[], int i, int j) {
    E temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}

// Random number generator functions

inline void Randomize() // Seed the generator
{ srand(1); }

// Return a random value in range 0 to n-1
inline int Random(int n)
{ return rand() % (n); }
```

Bibliography

- [AG06] Ken Arnold and James Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, USA, fourth edition, 2006.
- [Aha00] Dan Aharoni. Cogito, ergo sum! cognitive processes of students dealing with data structures. In *Proceedings of SIGCSE'00*, pages 26–30, ACM Press, March 2000.
- [AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.
- [AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, Reading, MA, 1983.
- [BB96] G. Brassard and P. Bratley. *Fundamentals of Algorithmics*. Prentice Hall, Upper Saddle River, NJ, 1996.
- [Ben75] John Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975. ISSN: 0001-0782.
- [Ben82] John Louis Bentley. *Writing Efficient Programs*. Prentice Hall, Upper Saddle River, NJ, 1982.
- [Ben84] John Louis Bentley. Programming pearls: The back of the envelope. *Communications of the ACM*, 27(3):180–184, March 1984.
- [Ben85] John Louis Bentley. Programming pearls: Thanks, heaps. *Communications of the ACM*, 28(3):245–250, March 1985.
- [Ben86] John Louis Bentley. Programming pearls: The envelope is back. *Communications of the ACM*, 29(3):176–182, March 1986.
- [Ben88] John Bentley. *More Programming Pearls: Confessions of a Coder*. Addison-Wesley, Reading, MA, 1988.
- [Ben00] John Bentley. *Programming Pearls*. Addison-Wesley, Reading, MA, second edition, 2000.
- [BG00] Sara Baase and Allen Van Gelder. *Computer Algorithms: Introduction to Design & Analysis*. Addison-Wesley, Reading, MA, USA, third edition, 2000.

- [BM85] John Louis Bentley and Catherine C. McGeoch. Amortized analysis of self-organizing sequential search heuristics. *Communications of the ACM*, 28(4):404–411, April 1985.
- [Bro95] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 25th Anniversary Edition*. Addison-Wesley, Reading, MA, 1995.
- [BSTW86] John Louis Bentley, Daniel D. Sleator, Robert E. Tarjan, and Victor K. Wei. A locally adaptive data compression scheme. *Communications of the ACM*, 29(4):320–330, April 1986.
- [CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, third edition, 2009.
- [Com79] Douglas Comer. The ubiquitous B-tree. *Computing Surveys*, 11(2):121–137, June 1979.
- [DD08] H.M. Deitel and P.J. Deitel. *C++ How to Program*. Prentice Hall, Upper Saddle River, NJ, sixth edition, 2008.
- [ECW92] Vladimir Estivill-Castro and Derick Wood. A survey of adaptive sorting algorithms. *Computing Surveys*, 24(4):441–476, December 1992.
- [ED88] R.J. Enbody and H.C. Du. Dynamic hashing schemes. *Computing Surveys*, 20(2):85–113, June 1988.
- [Epp10] Susanna S. Epp. *Discrete Mathematics with Applications*. Brooks/Cole Publishing Company, Pacific Grove, CA, fourth edition, 2010.
- [ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990.
- [ESS81] S. C. Eisenstat, M. H. Schultz, and A. H. Sherman. Algorithms and data structures for sparse symmetric gaussian elimination. *SIAM Journal on Scientific Computing*, 2(2):225–237, June 1981.
- [FBY92] W.B. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures & Algorithms*. Prentice Hall, Upper Saddle River, NJ, 1992.
- [FF89] Daniel P. Friedman and Matthias Felleisen. *The Little LISPer*. Macmillan Publishing Company, New York, NY, 1989.
- [FFBS95] Daniel P. Friedman, Matthias Felleisen, Duane Bibby, and Gerald J. Sussman. *The Little Schemer*. The MIT Press, Cambridge, MA, fourth edition, 1995.
- [FHCD92] Edward A. Fox, Lenwood S. Heath, Q. F. Chen, and Amjad M. Daoud. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, 35(1):105–121, January 1992.
- [FL95] H. Scott Folger and Steven E. LeBlanc. *Strategies for Creative Problem Solving*. Prentice Hall, Upper Saddle River, NJ, 1995.
- [FZ98] M.J. Folk and B. Zoellick. *File Structures: An Object-Oriented Approach with C++*. Addison-Wesley, Reading, MA, third edition, 1998.

- [GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.
- [GI91] Zvi Galil and Giuseppe F. Italiano. Data structures and algorithms for disjoint set union problems. *Computing Surveys*, 23(3):319–344, September 1991.
- [GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, New York, NY, 1979.
- [GKP94] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, Reading, MA, second edition, 1994.
- [Gle92] James Gleick. *Genius: The Life and Science of Richard Feynman*. Vintage, New York, NY, 1992.
- [GMS91] John R. Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1991.
- [Gut84] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In B. Yormark, editor, *Annual Meeting ACM SIGMOD*, pages 47–57, Boston, MA, June 1984.
- [Hay84] B. Hayes. Computer recreations: On the ups and downs of hailstone numbers. *Scientific American*, 250(1):10–16, January 1984.
- [Hei09] James L. Hein. *Discrete Structures, Logic, and Computability*. Jones and Bartlett, Sudbury, MA, third edition, 2009.
- [Jay90] Julian Jaynes. *The Origin of Consciousness in the Breakdown of the Bicameral Mind*. Houghton Mifflin, Boston, MA, 1990.
- [Kaf98] Dennis Kafura. *Object-Oriented Software Design and Construction with C++*. Prentice Hall, Upper Saddle River, NJ, 1998.
- [Knu94] Donald E. Knuth. *The Stanford GraphBase*. Addison-Wesley, Reading, MA, 1994.
- [Knu97] Donald E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1. Addison-Wesley, Reading, MA, third edition, 1997.
- [Knu98] Donald E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison-Wesley, Reading, MA, second edition, 1998.
- [Koz05] Charles M. Kozierok. The PC guide. www.pcguide.com, 2005.
- [KP99] Brian W. Kernighan and Rob Pike. *The Practice of Programming*. Addison-Wesley, Reading, MA, 1999.
- [Lag85] J. C. Lagarias. The $3x+1$ problem and its generalizations. *The American Mathematical Monthly*, 92(1):3–23, January 1985.

- [Lev94] Marvin Levine. *Effective Problem Solving*. Prentice Hall, Upper Saddle River, NJ, second edition, 1994.
- [LLKS85] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys, editors. *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*. John Wiley & Sons, New York, NY, 1985.
- [Man89] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley, Reading, MA, 1989.
- [MM04] Nimrod Megiddo and Dharmendra S. Modha. Outperforming lru with an adaptive replacement cache algorithm. *IEEE Computer*, 37(4):58–65, April 2004.
- [MM08] Zbigniew Michalewicz and Matthew Michalewicz. *Puzzle-Based Learning: An introduction to critical thinking, mathematics, and problem solving*. Hybrid Publishers, Melbourne, Australia, 2008.
- [Pó157] George Pólya. *How To Solve It*. Princeton University Press, Princeton, NJ, second edition, 1957.
- [Pug90] W. Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, June 1990.
- [Raw92] Gregory J.E. Rawlins. *Compared to What? An Introduction to the Analysis of Algorithms*. Computer Science Press, New York, NY, 1992.
- [Rie96] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, Reading, MA, 1996.
- [Rob84] Fred S. Roberts. *Applied Combinatorics*. Prentice Hall, Upper Saddle River, NJ, 1984.
- [Rob86] Eric S. Roberts. *Thinking Recursively*. John Wiley & Sons, New York, NY, 1986.
- [RW94] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, March 1994.
- [Sal88] Betty Salzberg. *File Structures: An Analytic Approach*. Prentice Hall, Upper Saddle River, NJ, 1988.
- [Sam06] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, San Francisco, CA, 2006.
- [SB93] Clifford A. Shaffer and Patrick R. Brown. A paging scheme for pointer-based quadrees. In D. Abel and B-C. Ooi, editors, *Advances in Spatial Databases*, pages 89–104, Springer Verlag, Berlin, June 1993.
- [Sed80] Robert Sedgewick. *Quicksort*. Garland Publishing, Inc., New York, NY, 1980.
- [Sed11] Robert Sedgewick. *Algorithms*. Addison-Wesley, Reading, MA, 4th edition, 2011.
- [Sel95] Kevin Self. Technically speaking. *IEEE Spectrum*, 32(2):59, February 1995.

- [SH92] Clifford A. Shaffer and Gregory M. Herb. A real-time robot arm collision avoidance system. *IEEE Transactions on Robotics*, 8(2):149–160, 1992.
- [SJH93] Clifford A. Shaffer, Ramana Juvvadi, and Lenwood S. Heath. A generalized comparison of quadtree and bintree storage requirements. *Image and Vision Computing*, 11(7):402–412, September 1993.
- [Ski10] Steven S. Skiena. *The Algorithm Design Manual*. Springer Verlag, New York, NY, second edition, 2010.
- [SM83] Gerard Salton and Michael J. McGill. *Introduction to Modern Information Retrieval*. McGraw-Hill, New York, NY, 1983.
- [Sol09] Daniel Solow. *How to Read and Do Proofs: An Introduction to Mathematical Thought Processes*. John Wiley & Sons, New York, NY, fifth edition, 2009.
- [ST85] D.D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32:652–686, 1985.
- [Sta11a] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, Upper Saddle River, NJ, seventh edition, 2011.
- [Sta11b] Richard M. Stallman. *GNU Emacs Manual*. Free Software Foundation, Cambridge, MA, sixteenth edition, 2011.
- [Ste90] Guy L. Steele. *Common Lisp: The Language*. Digital Press, Bedford, MA, second edition, 1990.
- [Sto88] James A. Storer. *Data Compression: Methods and Theory*. Computer Science Press, Rockville, MD, 1988.
- [Str00] Bjarne Stroustrup. *The C++ Programming Language, Special Edition*. Addison-Wesley, Reading, MA, 2000.
- [SU92] Clifford A. Shaffer and Mahesh T. Ursekar. Large scale editing and vector to raster conversion via quadtree spatial indexing. In *Proceedings of the 5th International Symposium on Spatial Data Handling*, pages 505–513, August 1992.
- [SW94] Murali Sitaraman and Bruce W. Weide. Special feature: Component-based software using resolve. *Software Engineering Notes*, 19(4):21–67, October 1994.
- [SWH93] Murali Sitaraman, Lonnie R. Welch, and Douglas E. Harms. On specification of reusable software components. *International Journal of Software Engineering and Knowledge Engineering*, 3(2):207–229, June 1993.
- [Tan06] Andrew S. Tanenbaum. *Structured Computer Organization*. Prentice Hall, Upper Saddle River, NJ, fifth edition, 2006.
- [Tar75] Robert E. Tarjan. On the efficiency of a good but not linear set merging algorithm. *Journal of the ACM*, 22(2):215–225, April 1975.

- [Wel88] Dominic Welsh. *Codes and Cryptography*. Oxford University Press, Oxford, 1988.
- [Win94] Patrick Henry Winston. *On to C++*. Addison-Wesley, Reading, MA, 1994.
- [WL99] Arthur Whimbey and Jack Lochhead. *Problem Solving & Comprehension*. Lawrence Erlbaum Associates, Mahwah, NJ, sixth edition, 1999.
- [WMB99] I.H. Witten, A. Moffat, and T.C. Bell. *Managing Gigabytes*. Morgan Kaufmann, second edition, 1999.
- [Zei07] Paul Zeitz. *The Art and Craft of Problem Solving*. John Wiley & Sons, New York, NY, second edition, 2007.

Index

- 80/20 rule, 319, 343
- abstract data type (ADT), xiv, 8–12, 20, 21, 49, 95–99, 134–143, 145, 146, 155, 168, 169, 204–206, 213, 215, 223, 224, 285–290, 381, 386, 388, 421, 436, 464
- abstraction, 10
- accounting, 120, 129
- Ackermann’s function, 223
- activation record, *see* compiler, activation record
- aggregate type, 8
- algorithm analysis, xiii, 4, 55–91, 231
 - amortized, *see* amortized analysis
 - asymptotic, 4, 55, 56, 65–70, 95, 469
 - empirical comparison, 55–56, 85, 232
 - for program statements, 71–75
 - multiple parameters, 79–80
 - running time measures, 57
 - space requirements, 56, 80–82
- algorithm, definition of, 17–18
- all-pairs shortest paths, 521–523, 540, 543
- amortized analysis, 73, 114, 321, 469, 484–485, 487, 489, 490
- approximation, 561
- array
 - dynamic, 113, 489
 - implementation, 8, 9, 21
- artificial intelligence, 381
- assert**, xvi
- asymptotic analysis, *see* algorithm analysis, asymptotic
- ATM machine, 6
- average-case analysis, 61–62
- AVL tree, 196, 359, 437, 442–446, 464
- back of the envelope, napkin, *see* estimating
- backtracking, 561
- bag, 26, 49
- bank, 6–7
- basic operation, 5, 6, 20, 21, 57, 58, 63
- best fit, *see* memory management, best fit
- best-case analysis, 61–62
- big-Oh notation, *see* O notation
- bin packing, 562
- binary search, *see* search, binary
- binary search tree, *see* BST
- binary tree, 151–201, 203
 - BST, *see* BST
 - complete, 152, 153, 168, 179, 251
 - full, 152–155, 166, 167, 186, 196, 221
 - implementation, 151, 153, 196
 - node, 151, 155, 160–166
 - NULL** pointers, 155
 - overhead, 166

- parent pointer, 160
- space requirements, 154, 160, 166–167
- terminology, 151–153
- threaded, 200
- traversal, *see* traversal, binary tree
- Binsort, 81, 82, 252–259, 262, 331, 546
- bintree, 460, 464
- birthday problem, 325, 347
- block, 292, 296
- Boolean expression, 555
 - clause, 555
 - Conjunctive Normal Form, 555
 - literal, 555
- Boolean variable, 8, 30, 90, 91
- branch and bounds, 561
- breadth-first search, 381, 394, 396, 397, 410
- BST, xv, 168–178, 196–198, 200, 226, 245, 251, 358–363, 368, 437, 442–448, 450, 459, 464, 524, 530
 - efficiency, 176
 - insert, 172–174
 - remove, 174–176
 - search, 170–172
 - search tree property, 169
 - traversal, *see* traversal
- B-tree, 312, 324, 352, 356, 364–375, 377, 461
 - analysis, 374–375
 - B⁺-tree, 8, 10, 352, 356, 368–375, 377, 378, 438
 - B*-tree, 374
- Bubble Sort, 76, 235–236, 238–239, 260, 266
- buffer pool, xv, 11, 273, 282–290, 306–308, 319, 320, 358, 367, 429
 - ADT, 285–290
 - replacement schemes, 283–284, 320–322
- cache, 276, 282–290, 319
- CD-ROM, 9, 274, 277, 325
- ceiling function, 30
- city database, 149, 200, 454, 464
- class, *see* object-oriented programming, class
- clique, 553, 558–560, 571
- cluster problem, 226, 464
- cluster, file, 278, 281, 429
- code tuning, 55–57, 83–86, 250–251
- Collatz sequence, 69, 89, 563, 570, 573
- comparator, xiv
- compiler, 85
 - activation record, 125
 - efficiency, 56
 - optimization, 84
- complexity, 10
- composite, *see* design pattern, composite
- composite type, 8
- computability, 19, 563, 570
- computer graphics, 438, 448
- connected component, *see* graph, connected component
- contradiction, proof by, *see* proof, contradiction
- cost, 5
- cylinder, *see* disk drive, cylinder
- data item, 8
- data member, 9
- data structure, 4, 9
 - costs and benefits, xiii, 3, 6–8
 - definition, 9
 - philosophy, 4–6
 - physical vs. logical form, xv, 8–9, 11–12, 95, 179, 276, 286, 413
 - selecting, 5–6
 - spatial, *see* spatial data structure

- data type, 8
- decision problem, 552, 556, 571
- decision tree, 262–265, 498–499
- decomposition
 - image space, 438
 - key space, 438
 - object space, 437
- delete**, 110, 111, 422
- depth-first search, 381, 393–395, 410, 432, 490
- deque, 149
- dequeue, *see* queue, dequeue
- design pattern, xiv, 12–16, 20
 - composite, 14–15, 163, 459
 - flyweight, 13, 163, 200, 458–459
 - strategy, 15–16, 143
 - visitor, 13–14, 158, 394, 412
- Deutsch-Schorr-Waite algorithm, 433, 436
- dictionary, xiv, 169, 339, 439
 - ADT, 134–143, 311, 349, 378, 517
- Dijkstra’s algorithm, 400–402, 404, 410, 411, 521, 522
- Diminishing Increment Sort, *see* Shellsort
- directed acyclic graph (DAG), 383, 396, 410, 414, 432
- discrete mathematics, xiv, 47
- disjoint, 151
- disjoint set, *see* equivalence class
- disk drive, 9, 273, 276–304, 306
 - access cost, 280–282, 303
 - cylinder, 277, 357
 - organization, 276–279
- disk processing, *see* file processing
- divide and conquer, 245, 248, 250, 314, 475, 480–482
- document retrieval, 324, 345
- double buffering, 283, 295, 297
- dynamic array, *see* array, dynamic
- dynamic memory allocation, 103
- dynamic programming, 517–523, 540, 561
- efficiency, xiii, 3–5
- element, 25
 - homogeneity, 96, 114
 - implementation, 114–115
- Emacs text editor, 431, 433
- encapsulation, 9
- enqueue, *see* queue, enqueue
- entry-sequenced file, 351
- enumeration, *see* traversal
- equation, representation, 162
- equivalence, 27–28
 - class, 27, 203, 208–213, 223, 224, 226, 407, 408, 411, 412, 464
 - relation, 27, 48
- estimation, 25, 46–48, 52, 53, 55–57, 65
- exact-match query, *see* search, exact-match query
- exponential growth rate, *see* growth rate, exponential
- expression tree, 162–166
- extent, 279
- external sorting, *see* sorting, external
- factorial function, 29, 34, 36, 45, 49, 73, 81, 86, 127, 262, 265, 570
 - Stirling’s approximation, 29, 265
- Fibonacci sequence, 34, 49–51, 91, 477–478, 517
- FIFO list, 129
- file access, 290–291
- file manager, 276, 278, 282, 422, 423, 429
- file processing, 82, 232, 303
- file structure, 9, 275, 351, 375
- first fit, *see* memory management, first fit
- floor function, 30
- floppy disk drive, 277

- Floyd's algorithm, 521–523, 540, 543
- flyweight, *see* design pattern, flyweight
- fragmentation, 279, 282, 423, 427–429
 - external, 423
 - internal, 279, 423
- free store, 110–111
- free tree, 383, 403, 409
- freelist, 120, 124
- fstream** class, 290–291
- full binary tree theorem, 153–155, 166, 196, 221
- function, mathematical, 16
- general tree, 203–227
 - ADT, 204–205, 223
 - converting to binary tree, 218, 224
 - dynamic implementations, 224
 - implementation, 213–218
 - left-child/right-sibling, 215, 224
 - list of children, 214–215, 224, 383
 - parent pointer implementation, 207–213, 445
 - terminology, 203–204
 - traversal, *see* traversal
- Geographic Information System, 7–8
- geometric distribution, 318, 527, 530
- gigabyte, 29
- graph, xv, 22, 381–412, 415
 - adjacency list, 381, 383, 384, 391, 410
 - adjacency matrix, 381, 383, 384, 388, 389, 410, 416
 - ADT, 381, 386, 388
 - connected component, 383, 412, 490
 - edge, 382
 - implementation, 381, 386–388
 - modeling of problems, 381, 390, 396, 399, 400, 402
 - parallel edge, 382
 - representation, 383–386
 - self loop, 382
 - terminology, 381–383
 - traversal, *see* traversal, graph
 - undirected, 382, 416
 - vertex, 382
- greatest common divisor, *see* largest common factor
- greedy algorithm, 189, 405, 407
- growth rate, 55, 58–60, 86
 - asymptotic, 65
 - constant, 58, 66
 - exponential, 60, 64, 544, 549–563
 - linear, 60, 63, 65, 82
 - quadratic, 60, 63, 64, 82, 83
- halting problem, 563–569
- Hamiltonian cycle, 571
- Harmonic Series, 33, 319, 483
- hashing, 7, 10, 31, 62, 312, 324–345, 351, 352, 365, 417, 461, 489
 - analysis of, 339–344
 - bucket, 331–333, 349
 - closed, 330–339, 348
 - collision resolution, 325, 331–339, 343, 345
 - deletion, 344–345, 348
 - double, 339, 348
 - dynamic, 345
 - hash function, 325–329, 347
 - home position, 331
 - linear probing, 333–336, 339, 342, 343, 347, 349
 - load factor, 341
 - open, 330–331
 - perfect, 325, 345
 - primary clustering, 336–338
 - probe function, 334, 336–339
 - probe sequence, 334–339, 341–345
 - pseudo-random probing, 337–339
 - quadratic probing, 338, 339, 347
 - search, 334
 - table, 324
 - tombstone, 344

- header node, 124, 134
- heap, 151, 153, 168, 178–185, 196, 198, 201, 251–252, 271, 401, 407
 - building, 181–184
 - for memory management, 422
 - insert, 181, 182
 - max-heap, 179
 - min-heap, 179, 297
 - partial ordering property, 179
 - remove, 184, 185
 - siftdown, 183, 184, 271
- Heapsort, 179, 251–253, 260, 271, 297
- heuristic, 561–562
- hidden obligations, *see* obligations, hidden
- Huffman coding tree, 151, 153, 162, 185–196, 198–201, 226, 438
 - prefix property, 194
- independent set, 571
- index, 11, 267, 351–378
 - file, 293, 351
 - inverted list, 355, 376
 - linear, 8, 353–355, 376, 377
 - tree, 352, 358–375
- induction, 224
- induction, proof by, *see* proof, induction
- inheritance, xvi, 97, 100, 105, 142, 162, 164, 165, 167
- inorder traversal, *see* traversal, inorder
- input size, 57, 61
- Insertion Sort, 76, 233–236, 238–241, 250, 260, 262–267, 269, 270
 - Double, 270
- integer representation, 4, 8–9, 20, 149
- inversion, 235, 239
- inverted list, *see* index, inverted list
- ISAM, 352, 356–358, 377
- k-d tree, 450–455, 459, 461, 464
- K-ary tree, 218–219, 223, 224, 455
- key, 134–138
- kilobyte, 29
- knapsack problem, 560, 572, 573
- Kruskal's algorithm, xv, 252, 407–408, 411
- largest common factor, 50, 531–532
- latency, 278, 280, 281
- least frequently used (LFU), 284, 306, 320
- least recently used (LRU), 284, 306, 308, 320, 367
- LIFO list, 120
- linear growth, *see* growth rate, linear
- linear index, *see* index, linear
- linear search, *see* search, sequential
- link, *see* list, link class
- linked list, *see* list, linked
- LISP, 48, 415, 430, 432, 433
- list, 23, 95–151, 187, 352, 413, 415, 489
 - ADT, 9, 95–99, 146
 - append, 100, 116
 - array-based, 8, 95, 100–103, 112–114, 121, 149
 - basic operations, 96
 - circular, 147
 - comparison of space requirements, 148
 - current position, 96, 97, 104–106, 113
 - doubly linked, 115–120, 147, 149, 160
 - space, 118–120
 - element, 96, 114–115
 - freelist, 110–112, 422–432
 - head, 96, 100, 105
 - implementations compared, 112–114
 - initialization, 96, 100
 - insert, 96, 100, 102, 104–106, 109, 113, 116, 119, 151
 - link class, 103

- linked, 8, 95, 100, 103–106, 112–114, 354, 413, 437, 524, 525, 529
- node, 103–105, 116, 117
- notation, 96
- ordered by frequency, 317–323, 469
- orthogonal, 418
- remove, 96, 100, 106, 109, 113, 116, 118, 119
- search, 151, 311–323
- self-organizing, xv, 62, 320–323, 343, 345, 346, 348, 445, 486–487
- singly linked, 103, 115
- sorted, 4, 96, 140–143
- space requirements, 112–113, 147
- tail, 96
- terminology, 96
- unsorted, 96
- locality of reference, 278, 282, 343, 358, 365
- logarithm, 31–32, 49, 535–536
 - \log^* , 213, 223
- logical representation, *see* data
 - structure, physical vs. logical form
- lookup table, 81
- lower bound, 55, 67–70, 342
 - sorting, 261–265, 546
- map, 381, 399
- Master Theorem, *see* recurrence
 - relation, Master Theorem
- matching, 561
- matrix, 416–420
 - multiplication, 548, 549
 - sparse, xv, 9, 413, 417–420, 434, 435
 - triangular, 416, 417
- megabyte, 29
- member, *see* object-oriented
 - programming, member
- member function, *see* object-oriented
 - programming, member function
- memory management, 11, 413, 420–435
 - ADT, 421, 436
 - best fit, 426, 562
 - buddy method, 423, 427–428, 436
 - failure policy, 423, 429–433
 - first fit, 426, 562
 - garbage collection, 430–433
 - memory allocation, 421
 - memory pool, 421
 - sequential fit, 423–427, 435
 - worst fit, 426
- Mergesort, 127, 241–244, 256, 260, 269, 475, 480, 482
 - external, 294–296
 - multiway merging, 300–302, 306–308
- metaphor, 10, 19
- Microsoft Windows, 278, 303
- millisecond, 29
- minimum-cost spanning tree, 231, 252, 381, 402–408, 410, 411, 543
- modulus function, 28, 30
- move-to-front, 320–323, 346, 486–487
- multilist, 26, 413–416, 434
- multiway merging, *see* Mergesort,
 - multiway merging
- nested parentheses, 22, 148
- networks, 381, 400
- new**, 110–111, 116, 422
- \mathcal{NP} , *see* problem, \mathcal{NP}
- null pointer, 103
- O notation, 65–70, 87
- object-oriented programming, 9, 11–16, 19–20

- class, xvi, 9, 96
- class hierarchy, 14–15, 160–166, 458–459
- inheritance, xvi
 - members and objects, 8, 9
- obligations, hidden, 143, 158, 289
- octree, 459
- Ω notation, 67–70, 87
- one-way list, 103
- operating system, 18, 178, 276, 278, 281–284, 294, 297, 422, 429, 434
- operator overloading, xvi, 110
- overhead, 81, 112–113
 - binary tree, 197
 - matrix, 418
 - stack, 125
- pairing, 544–546
- palindrome, 148
- partial order, 28, 49, 179
 - poset, 28
- partition, 571
- path compression, 212–213, 224, 491
- permutation, 29, 49, 50, 81, 82, 252, 263–265, 341
- physical representation, *see* data
 - structure, physical vs. logical form
- Pigeonhole Principle, 51, 132
- point quadtree, 460, 464
- pop, *see* stack, pop
- postorder traversal, *see* traversal,
 - postorder
- powerset, *see* set, powerset
- PR quadtree, 13, 162, 218, 450, 455–459, 461, 463, 464
- preorder traversal, *see* traversal,
 - preorder
- prerequisite problem, 381
- Prim’s algorithm, 404–406, 410, 411
- primary index, 352
- primary key, 352
- priority queue, 151, 168, 187, 201, 402, 405
- probabilistic data structure, 517, 524–530
- problem, 6, 16, 18, 544
 - analysis of, 55, 76–77, 232, 261–265
 - hard, 549–563
 - impossible, 563–569
 - instance, 16
 - \mathcal{NP} , 551
 - \mathcal{NP} -complete, 551–563, 569
 - \mathcal{NP} -hard, 553
- problem solving, 19
- program, 3, 18
 - running time, 56–57
- programming style, 19
- proof
 - contradiction, 39–40, 51, 405, 546, 567, 568
 - direct, 39
 - induction, 34, 39–44, 48, 51, 154, 182, 192, 196, 265, 409, 470–473, 475, 476, 479, 488, 489, 491, 509
- pseudo-polynomial time algorithm, 560
- pseudocode, xvii, 17
- push, *see* stack, push
- quadratic growth, *see* growth rate,
 - quadratic
- queue, 95, 103, 129–134, 148, 394, 397, 398
 - array-based, 129–132
 - circular, 129–132, 148
 - dequeue, 129, 134
 - empty vs. full, 131–132
 - enqueue, 129
 - implementations compared, 134
 - linked, 134, 135
 - priority, *see* priority queue

- terminology, 129
- Quicksort, 127, 235, 244–252, 260, 267, 270, 292, 294, 296, 346, 469
 - analysis, 482–483
- Radix Sort, 255–260, 262, 271
- RAM, 274, 275
- Random**, 30
- range query, 352, *see* search, range query
- real-time applications, 61, 62
- recurrence relation, 34–35, 51, 249, 469, 475–483, 487, 488
 - divide and conquer, 480–482
 - estimating, 475–478
 - expanding, 478, 480, 489
 - Master Theorem, 480–482
 - solution, 35
- recursion, xiv, 34, 36–38, 40, 41, 49, 50, 73, 111, 126, 157–158, 172, 197, 200, 242–244, 267, 269, 270, 432
 - implemented by stack, 125–129, 250
 - replaced by iteration, 50, 127
- reduction, 262, 544–549, 568, 570, 572
- relation, 27–29, 48, 49
- replacement selection, 179, 296–301, 307, 308, 469
- resource constraints, 5, 6, 16, 55, 56
- run (in sorting), 294
- run file, 294, 295
- running-time equation, 58
- satisfiability, 555–561
- Scheme, 48
- search, 21, 82, 311–349, 351
 - binary, 31, 73–75, 89–91, 266, 314, 346, 353, 367, 480, 489
 - defined, 311
 - exact-match query, 7–8, 10, 311, 312, 351
 - in a dictionary, 314
 - interpolation, 314–317, 346
 - jump, 313–314
 - methods, 311
 - multi-dimensional, 448
 - range query, 7–8, 10, 311, 324, 351, 358
 - sequential, 21, 57–58, 61–62, 66, 67, 73–75, 89, 91, 312–313, 322, 346, 484
 - sets, 323–324
 - successful, 311
 - unsuccessful, 311, 341
- search trees, 62, 178, 352, 356, 359, 365, 442, 445, 448
- secondary index, 352
- secondary key, 352
- secondary storage, 273–282, 304–306
- sector, 277, 279, 281, 292
- seek, 277, 280
- Selection Sort, 237–239, 250, 260, 266
- self-organizing lists, *see* list, self-organizing
- sequence, 27, 29, 49, 96, 312, 323, 353, 544
- sequential search, *see* search, sequential
- sequential tree implementations, 219–222, 225, 226
- serialization, 220
- set, 25–29, 49
 - powerset, 26, 29
 - search, 312, 323–324
 - subset, superset, 26
 - terminology, 25–26
 - union, intersection, difference, 26, 323, 347
- Shellsort, 235, 239–241, 260, 267
- shortest paths, 381, 399–402, 410
- simulation, 85

- Skip List, xv, 524–530, 540, 541
- slide rule, 32, 535
- software engineering, xiii, 4, 19, 544
- sorting, 17, 21, 22, 57, 61, 62, 76–77, 79, 82, 231–271, 313, 322, 544–546
 - adaptive, 265
 - comparing algorithms, 232–233, 259–261, 302
 - exchange sorting, 238–239
 - external, xv, 168, 232, 251, 273, 291–303, 306–308
 - internal, xv
 - lower bound, 232, 261–265
 - small data sets, 233, 250, 265, 268
 - stable algorithms, 232, 266
 - terminology, 232–233
- spatial data structure, 437, 448–461
- splay tree, 178, 196, 359, 437, 442, 445–448, 461, 462, 464, 469, 525
- stable sorting algorithms, *see* sorting, stable algorithms
- stack, 95, 103, 120–129, 148, 197, 200, 250, 266, 267, 270, 393–395, 485
 - array-based, 121–123
 - constructor, 121
 - implementations compared, 125
 - insert, 120, 121
 - linked, 124, 125
 - pop, 121–123, 125, 150
 - push, 121–123, 125, 150
 - remove, 120, 121
 - terminology, 121
 - top, 121–123, 125
 - two in one array, 125, 148
 - variable-size elements, 149
- Strassen's algorithm, 532, 541
- strategy, *see* design pattern, strategy
- subclass, *see* object-oriented programming, class hierarchy
- subset, *see* set, subset
- suffix tree, 463
- summation, 32–34, 41, 42, 51, 72, 73, 90, 177, 184, 248, 249, 318, 319, 417, 469–474, 479, 481, 482, 484, 485, 487
 - guess and test, 487
 - list of solutions, 33, 34
 - notation, 32
 - shifting method, 471–474, 483, 488
- swap**, 30
- tape drive, 276, 291
- template, 114
- templates, xvi, 12, 97
- text compression, 151, 185–195, 322–323, 345, 349
- Θ notation, 68–70, 89
- topological sort, 381, 394–398, 410
- total order, 28, 49, 179
- Towers of Hanoi, 36–38, 127, 543, 550
- tradeoff, xiii, 3, 13, 75, 279, 292
 - disk-based space/time principle, 82, 343
 - space/time principle, 81–82, 97, 120, 185, 343
- transportation network, 381, 399
- transpose, 321, 322, 346
- traveling salesman, 551–553, 560, 561, 571, 573
- traversal
 - binary tree, 127, 151, 155–160, 163, 169, 177, 197, 390
 - enumeration, 155, 169, 220
 - general tree, 205–206, 223
 - graph, 381, 390–398
- tree
 - height balanced, 364, 365, 367, 525
 - terminology, 151

- trie, 162, 186, 196, 259, 437–442, 462, 463
 - alphabet, 438
 - binary, 438
 - PATRICIA, 439–442, 461, 462
- tuple, 27
- Turing machine, 556
- two-coloring, 44
- 2-3 tree, 178, 352, 360–364, 367, 370, 377, 378, 442, 489, 524
- type, 8
- uncountability, 564–566
- UNION/FIND, xv, 207, 408, 412, 469, 491
- units of measure, 29, 47
- UNIX, 245, 278, 303, 432
- upper bound, 55, 65–69
- variable-length record, 149, 150, 353, 377, 413, 415, 420
 - sorting, 233
- vector, 27, 113
- vertex cover, 557, 558, 560, 561, 571, 572
- virtual function, xvi, 167
- virtual memory, 284–286, 294, 306
- visitor, *see* design pattern, visitor
- weighted union rule, 210–212, 223–224, 491
- worst fit, *see* memory management, worst fit
- worst-case analysis, 61–62, 67
- Zipf distribution, 319, 326, 346
- Ziv-Lempel coding, 323, 345