BỘ GIAO THÔNG VẬN TẢI
TRƯỜNG ĐẠI HỌC HÀNG HẢI
BỘ MÔN: KHOA HỌC MÁY TÍNH
KHOA: CÔNG NGHỆ THÔNG TIN

BÀI GIẢNG

CẤU TRÚC DỮ LIỆU

TÊN HỌC PHẦN : **Cấu trúc dữ liệu**
MÃ HỌC PHẦN : 17207
TRÌNH ĐỘ ĐÀO TẠO : **ĐẠI HỌC CHÍNH QUY**
DÙNG CHO SV NGÀNH : **CÔNG NGHỆ THÔNG TIN**

HẢI PHÒNG - 2008

11.7. **Tên học phần:** Cấu trúc dữ liệu **Loại học phần:** 2
Bộ môn phụ trách giảng dạy: Khoa học Máy tính **Khoa phụ trách:** CNTT
Mã học phần: 17207 **Tổng số TC**: 3

TS tiết	Lý thuyết	Thực hành/Xemina	Tự học	Bài tập lớn	Đồ án môn học
60	30	30	0	0	0

Điều kiện tiên quyết:

Sinh viên phải học xong các học phần sau mới được đăng ký học phần này:
Toán cao cấp, Toán rời rạc, Ngôn ngữ C, Tin học đại cương.

Mục tiêu của học phần:

Cung cấp kiến thức và rèn luyện kỹ năng thực hành cấu trúc dữ liệu cho sinh viên.

Nội dung chủ yếu

- Những vấn đề cơ bản về cấu trúc dữ liệu;
- Các cấu trúc dữ liệu cơ bản
- Danh sách liên kết;
- Ngăn xếp, hàng đợi;
- Cấu trúc cây;
- Bảng băm, ...

Nội dung chi tiết của học phần:

TÊN CHƯƠNG MỤC	**PHÂN PHỐI SỐ TIẾT**				
	TS	**LT**	**TH/Xemina**	**BT**	**KT**
Chương I : Khái niệm liên quan đến CTDL	2	2	0		
1.1. Giải thuật và cấu trúc dữ liệu.					
1.2. Giải thuật và các vấn đề liên quan.					
1.3. Ngôn ngữ diễn đạt giải thuật.					
1.4. Kiểu dữ liệu, cấu trúc dữ liệu, kiểu dữ liệu trừu tượng.					
Chương II : Các kiểu dữ liệu trừu tượng cơ bản	12	6	6		
2.1. Danh sách					
2.1.1. Khái niệm danh sách					
2.1.2. Các phép toán trên danh sách					
2.1.3. Cài đặt danh sách					
2.1.4. Các dạng danh sách liên kết (DSLK): DSLK đơn, vòng, kép, …					
2.2. Ngăn xếp (stack)					
2.2.1. Khái niệm					
2.2.2. Cài đặt ngăn xếp bởi mảng, DSLK					
2.2.3. Ứng dụng					
2.3. Hàng đợi (queue)					
2.3.1. Khái niệm					
2.3.2. Cài đặt hàng đợi bởi mảng, DSLK					
2.3.3. Ứng dụng					
2.4. Bài tập áp dụng					
Chương III: Cây (tree).	18	9	8		1
3.1. Khái niệm.					

TÊN CHƯƠNG MỤC	PHÂN PHỐI SỐ TIẾT				
	TS	LT	TH/Xemina	BT	KT
3.2. Cây tổng quát.					
3.2.1. Biểu diễn cây tổng quát.					
3.2.2. Duyệt cây tổng quát.					
3.2.3. Vài ví dụ áp dụng.					
3.3. Cây nhị phân.					
3.3.1. Định nghĩa và tính chất					
3.3.2. Lưu trữ cây.					
3.3.3. Duyệt cây.					
3.3.4. Cây nhị phân nối vòng.					
3.4. Các phép toán thực hiện trên cây nhị phân.					
3.4.1. Dựng cây					
3.4.2. Duyệt cây để tìm kiếm					
3.4.3. Sắp xếp cây nhị phân					
3.5. Cây tìm kiếm nhị phân (binary search tree)					
3.5.1. Khái niệm, cài đặt.					
3.5.2. Cây AVL					
3.6. Bài tập					
Chương IV: Bảng băm (hash table)	14	7	6		1
5.1. Khái niệm					
5.2. Các loại hàm băm					
5.3. Các phương pháp giải quyết xung đột					
5.4. Đánh giá hiệu quả các phương pháp băm					
5.5. Bài tập áp dụng					

Nhiệm vụ của sinh viên :

Tham dự các buổi thuyết trình của giáo viên, tự học, tự làm bài tập do giáo viên giao, tham dự các bài kiểm tra định kỳ và cuối kỳ.

Tài liệu học tập :

1. Đinh Mạnh Tường, *Cấu trúc dữ liệu và thuật toán,* Nhà xuất bản ĐH QG Hà Nội, 2004.
2. Đỗ Xuân Lôi, *Cấu trúc dữ liệu và giải thuật*, Nhà xuất bản ĐH QG Hà Nội, 2004.
3. Robert Sedgewick, *Cẩm nang thuật toán*, NXB Khoa học kỹ thuật, 2000.

Hình thức và tiêu chuẩn đánh giá sinh viên:

- Hình thức thi cuối kỳ : Thi viết.
- Sinh viên phải đảm bảo các điều kiện theo Quy chế của Nhà trường và của Bộ

Thang điểm: Thang điểm chữ A, B, C, D, F

Điểm đánh giá học phần: Z = 0,3X + 0,7Y.

CHƯƠNG 1. CÁC KHÁI NIỆM MỞ ĐẦU

1.1. Giải thuật và cấu trúc dữ liệu.

Để giải một bài toán trong thực tế bằng máy tính ta phải bắt đầu từ việc xác định bài toán. Nhiều thời gian và công sức bỏ ra để xác định bài toán cần giải quyết, tức là phải trả lời rõ ràng câu hỏi "phải làm gì?" sau đó là "làm như thế nào?". Thông thường, khi khởi đầu, hầu hết các bài toán là không đon giản, không rõ ràng. Để giảm bớt sự phức tạp của bài toán thực tế, ta phải hình thức hóa nó, nghĩa là phát biểu lại bài toán thực tế thành một bài toán hình thức (hay còn gọi là mô hình toán). Có thể có rất nhiều bài toán thực tế có cùng một mô hình toán.

Ví dụ : Tô màu bản đồ thế giới.

Ta cần phải tô màu cho các nước trên bản đồ thế giới. Trong đó mỗi nước đều được tô một màu và hai nước láng giềng (cùng biên giới) thì phải được tô bằng hai màu khác nhau. Hãy tìm một phương án tô màu sao cho số màu sử dụng là ít nhất.

Ta có thể xem mỗi nước trên bản đồ thế giới là một đỉnh của đồ thị, hai nước láng giềng của nhau thì hai đỉnh ứng với nó được nối với nhau bằng một cạnh. Bài toán lúc này trở thành bài toán tô màu cho đồ thị như sau: Mỗi đỉnh đều phải được tô màu, hai đỉnh có cạnh nối thì phải tô bằng hai màu khác nhau và ta cần tìm một phương án tô màu sao cho số màu được sử dụng là ít nhất.

Đối với một bài toán đã được hình thức hoá, chúng ta có thể tìm kiếm cách giải trong thuật ngữ của mô hình đó và xác định có hay không một chương trình có sẵn để giải. Nếu không có một chương trình như vậy thì ít nhất chúng ta cũng có thể tìm được những gì đã biết về mô hình và dùng các tính chất của mô hình để xây dựng một giải thuật tốt.

Khi đã có mô hình thích hợp cho một bài toán ta cần cố gắng tìm cách giải quyết bài toán trong mô hình đó. Khởi đầu là tìm một giải thuật, đó là một chuỗi hữu hạn các chỉ thị (instruction) mà mỗi chỉ thị có một ý nghĩa rõ ràng và thực hiện được trong một lượng thời gian hữu hạn.

Nhưng xét cho cùng, giải thuật chỉ phản ánh các phép xử lý, còn đói tượng để xử lý trong máy tính chính là dữ liệu (data), chúng biểu diễn các thông tin cần thiết cho bài toán: các dữ liệu vào, các dữ liệu ra, dữ liệu trung gian, … Không thể nói tới giải thuật mà không nghĩ tới: giải thuật đó được tác động trên dữ liệu nào, còn xét tới dữ liệu thì phải biết dữ liệu ấy cần được giải thuật gì tác động để đưa ra kết quả mong muốn.. Như vậy, giữa cấu trúc dữ liệu và giải thuật có mối liên quan mật thiết với nhau.

1.2. Cấu trúc dữ liệu và các vấn đề liên quan.

Trong một bài toán, dữ liệu bao gồm một tập các phần tử cơ sở, được gọi là dữ liệu nguyên tử. Dữ liệu nguyên tử có thể là một chữ số, một ký tự, … cũng có thể là một số, một xâu, … tùy vào bài toán. Trên cơ sở các dữ liệu nguyên tử, các cung cách khả dĩ theo đó lien kết chúng lại với nhau, sẽ đãn đến các cấu trúc dữ liệu khác nhau.

Lựa chọn một cấu trúc dữ liệu thích hợp để tổ chức dữ liệu vào và trên cơ sở đó xây dựng được giải thuật xử lý hữu hiệu đưa tới kết quả mong muốn cho bài toán (dữ liệu ra), là một khâu quan trọng.

Cách biểu diễn một cấu trúc dữ liệu trong bộ nhớ được gọi là cấu trúc lưu trữ. Đây chính là cách cài đặt cấu trúc ấy trên máy tính và trên cơ sở các cấu trúc lưu trữ này mà thực hiện các phép xử lý. Có thể có nhiều cấu trúc lưu trữ khác nhau cho cùng một cấu trúc dữ liệu và ngược lại.

Khi đề cập tới cấu trúc lưu trũ, cần phân biệt: cấu trúc lưu trữ tương ứng với bộ nhớ trong – lưu trữ trong; cấu trúc lưu trữ ứng với bộ nhớ ngoài – lưu trữ ngoài. Chúng có đặc điểm và cách xử lý riêng.

1.3. Ngôn ngữ diễn đạt giải thuật.

Việc sử dụng một ngôn ngữ lập trình bậc cao để diễn đạt giải thuật, như Pascal, C, C++, … sẽ gặp một số hạn chế sau:

- Phải luôn tuân thủ các quy tắc chặt chẽ về cú pháp của ngôn ngữ khiến cho việc trình bày về giải thuật và cấu trúc dữ liệu có thiên hướng nặng nề, gò bó.

- Phải phụ thuộc vào cấu trúc dữ liệu tiền định của ngôn ngữ nên có lúc không thể hiện được đầy đủ các ý về cấu trúc mà ta muốn biểu đạt

Một khi đã có *mô hình thích hợp* cho bài toán, ta cần hình *thức hoá một giải thuật, một cấu trúc dữ liệu* trong thuật ngữ của mô hình đó. Khởi đầu là viết những mệnh đề tổng quát rồi tinh chế dần thành những chuỗi mệnh đề cụ thể hơn, cuối cùng là các chỉ thị thích hợp trong một ngôn ngữ lập trình.

Ở bước này, nói chung, ta có một giải thuật, một cấu trúc dữ liệu tương đối rõ ràng, nó gần giống như một chương trình được viết trong ngôn ngữ lập trình, nhưng nó không phải là một chương trình chạy được vì trong khi viết giải thuật ta không chú trọng nặng đến cú pháp của ngôn ngữ và các *kiểu dữ liệu* còn ở mức trừu tượng chứ không phải là các khai báo cài đặt kiểu trong ngôn ngữ lập trình.

Chẳng hạn với giải thuật tô màu đồ thị GREEDY, giả sử đồ thị là G, giải thuật sẽ xác định một tập hợp **Newclr** các đỉnh của G được tô cùng một màu, mà ta gọi là màu mới C ở trên. Để tiến hành tô màu hoàn tất cho đồ thị G thì giải thuật này phải được gọi lặp lại cho đến khi toàn thể các đỉnh đều được tô màu.

```
void GREEDY ( GRAPH *G, SET *Newclr )
{
Newclr =  ; /*1*/
for (mỗi đỉnh v chưa tô màu của G) /*2*/
if (v không được nối với một đỉnh nào trong Newclr) /*3*/
        {
                đánh dấu v đã được tô màu; /*4*/
                thêm v vào Newclr; /*5*/
        }
}
```

Trong thủ tục bằng ngôn ngữ giả này chúng ta đã dùng một số từ khoá của ngôn ngữ C xen lẫn các mệnh đề tiếng Việt. Điều đặc biệt nữa là ta dùng các kiểu GRAPH, SET có vẻ xa lạ, chúng là các "kiểu dữ liệu trừu tượng" mà sau này chúng ta sẽ viết bằng các khai báo thích hợp trong ngôn ngữ lập trình cụ thể. Dĩ nhiên, để cài đặt thủ tục này ta phải cụ thể hoá dần những mệnh đề bằng tiếng Việt ở trên cho đến khi mỗi mệnh đề tương ứng với một doạn mã thích hợp của ngôn ngữ lập trình. Chẳng hạn mệnh đề **if** ở /*3*/ có thể chi tiết hoá hơn nữa như sau:

```
void GREEDY ( GRAPH *G, SET *Newclr )
{
Newclr=  ; /*1*/
for (mỗi đỉnh v chưa tô màu của G) /*2*/
        {
        int found=0; /*3.1*/
        for (mỗi đỉnh w trong Newclr) /*3.2*/
        if (có cạnh nối giữa v và w) /*3.3*/
                found=1; /*3.4*/
        if (found==0)/*3.5*/
        {
                đánh dấu v đã được tô màu; /*4*/
                thêm v vào Newclr; /*5*/

        }

        }

}
```

GRAPH và SET ta coi như tập hợp. Có nhiều cách để biểu diễn tập hợp trong ngôn ngữ

lập trình, để đơn giản ta xem các tập hợp như là một danh sách (LIST) các số nguyên biểu diễn chỉ số của các đỉnh và kết thúc bằng một giá trị đặc biệt NULL. Với những qui ước như vậy ta có thể tinh chế giải thuật GREEDY một bước nữa như sau:

```
void GREEDY ( GRAPH *G, LIST *Newclr )
  {
  int found;
  int v,w ;
  Newclr=  ;
  v= đỉnh đầu tiên chưa được tô màu trong G;
  while (v<>null)
          {
          found=0;
          w=đỉnh đầu tiên trong newclr;
          while( w<>null) && (found=0)
                  {
                  if ( có cạnh nối giữa v và w )
                          found=1;
                  else w= đỉnh kế tiếp trong newclr;
                  }
                  if (found==0 )
                  {
                          Đánh dấu v đã được tô màu;
                          Thêm v vào Newclr;
                  }
          v= đỉnh chưa tô màu kế tiếp trong G;
          }
  }
```

1.4. Kiểu dữ liệu (data types), cấu trúc dữ liệu (data structures), kiểu dữ liệu trừu tượng (abstract data types –ADT).

Khái niệm trừu tượng hóa

Trong tin học, trừu tượng hóa nghĩa là đơn giản hóa, làm cho nó sáng sủa hơn và dễ hiểu hơn. Cụ thể trừu tượng hóa là che di những chi tiết, làm nổi bật cái tổng thể. Trừu tượng hóa có thể thực hiện trên hai khía cạnh là trừu tượng hóa dữ liệu và trừu tượng hóa chương trình.

Trừu tượng hóa chương trình

Trừu tượng hóa chương trình là sự định nghĩa các chương trình con để tạo ra các phép toán trừu tượng (sự tổng quát hóa của các phép toán nguyên thủy). Chẳng hạn ta có thể tạo ra một chương trình con Matrix_Mult để thực hiện phép toán nhân hai ma trận. Sau khi Matrix_mult đã được tạo ra, ta có thể dùng nó như một phép toán nguyên thủy (chẳng hạn phép cộng hai số).

Trừu tượng hóa chương trình cho phép phân chia chương trình thành các chương trình con. Sự phân chia này sẽ che dấu tất cả các lệnh cài đặt chi tiết trong các chương trình con. Ở cấp độ chương trình chính, ta chỉ thấy lời gọi các chương trình con và điều này được gọi là sự bao gói.

Ví dụ như một chương trình quản lý sinh viên được viết bằng trừu tượng hóa có thể là:

```
void main()
{
Nhap(Lop);
Xu_ly (Lop);
Xuat (Lop);
}
```

Trong chương trình trên, Nhap, Xu_ly, Xuat là các phép toán trừu tượng. Chúng che dấu bên trong rất nhiều lệnh phức tạp mà ở cấp độ chương trình chính ta không nhìn thấy được. Còn Lop là một biến thuộc kiểu dữ liệu trừu tượng mà ta sẽ xét sau.

Chương trình được viết theo cách gọi các phép toán trừu tượng có lệ thuộc vào cách cài đặt kiểu dữ liệu không?

Trừu tượng hóa dữ liệu

Trừu tượng hóa dữ liệu là định nghĩa các kiểu dữ liệu trừu tượng

Một kiểu dữ liệu trừu tượng là một mô hình toán học cùng với một tập hợp các phép toán (operator) trừu tượng được định nghĩa trên mô hình đó. Ví dụ tập hợp số nguyên cùng với các phép toán hợp, giao, hiệu là một kiểu dữ liệu trừu tượng.

Trong một ADT các phép toán có thể thực hiện trên các đói tượng (toán hạng) không chỉ thuộc ADT đó, cũng như kết quả không nhất thiết phải thuộc ADT. Tuy nhiên, phải có ít nhất một toán hạng hoặc kết quả phải thuộc ADT đang xét.

ADT là sự tổng quát hoá của các kiểu dữ liệu nguyên thuỷ.

Để minh hoạ ta có thể xét bản phác thảo cuối cùng của thủ tục GREEDY. Ta đã dùng một danh sách (LIST) các số nguyên và các phép toán trên danh sách newclr là:

- Tạo một danh sách rỗng.
- Lấy phần tử đầu tiên trong danh sách và trả về giá trị null nếu danh sách rỗng.
- Lấy phần tử kế tiếp trong danh sách và trả về giá trị null nếu không còn phần tử kế tiếp.
- Thêm một số nguyên vào danh sách.

Nếu chúng ta viết các chương trình con thực hiện các phép toán này, thì ta dễ dàng thay các mệnh đề hình thức trong giải thuật bằng các câu lệnh đơn giản

Câu lệnh	Mệnh đề hình thức
MAKENULL(newclr)	newclr=ϕ;
w=FIRST(newclr)	w=phần tử đầu tiên trong newclr
w=NEXT(w,newclr)	w=phần tử kế tiếp trong newclr
INSERT(v,newclr)	Thêm v vào newclr

Điều này cho thấy sự thuận lợi của ADT, đó là ta có thể định nghĩa một kiểu dữ liệu tuỳ ý cùng với các phép toán cần thiết trên nó rồi chúng ta dùng như là các đói tượng nguyên thuỷ. Hơn nữa chúng ta có thể cài đặt một ADT bằng bất kỳ cách nào, chương trình dùng chúng cũng không thay đổi, chỉ có các chương trình con biểu diễn cho các phép toán của ADT là thay đổi.

Cài đặt ADT là sự thể hiện các phép toán mong muốn (các phép toán trừu tượng) thành các câu lệnh của ngôn ngữ lập trình, bao gồm các khai báo thích hợp và các thủ tục thực hiện các phép toán trừu tượng. Để cài đặt ta chọn một **cấu trúc dữ liệu** thích hợp có trong ngôn ngữ lập trình hoặc là một cấu trúc dữ liệu phức hợp được xây dựng lên từ các kiểu dữ liệu cơ bản của ngôn ngữ lập trình.

Sự khác nhau giữa kiểu dữ liệu và kiểu dữ liệu trừu tượng là gì?

Mặc dù các thuật ngữ kiểu dữ liệu (hay kiểu - data type), cấu trúc dữ liệu (data structure), kiểu dữ liệu trừu tượng (abstract data type) nghe như nhau, nhưng chúng có ý nghĩa rất khác nhau.

Kiểu dữ liệu là một tập hợp các giá trị và một tập hợp các phép toán trên các giá trị đó.

Ví dụ kiểu Boolean là một tập hợp có 2 giá trị TRUE, FALSE và các phép toán trên nó như OR, AND, NOT …. Kiểu Integer là tập hợp các số nguyên có giá trị từ -32768 đến 32767 cùng các phép toán cộng, trừ, nhân, chia, Div, Mod…

Kiểu dữ liệu có hai loại là kiểu dữ liệu sơ cấp và kiểu dữ liệu có cấu trúc hay còn gọi là cấu trúc dữ liệu.

Kiểu dữ liệu sơ cấp là kiểu dữ liệu mà giá trị dữ liệu của nó là đơn nhất. Ví dụ: kiểu Boolean, Integer….

Kiểu dữ liệu có cấu trúc hay còn gọi là cấu trúc dữ liệu là kiểu dữ liệu mà giá trị dữ liệu của nó là sự kết hợp của các giá trị khác. Ví dụ: ARRAY là một cấu trúc dữ liệu.

Một kiểu dữ liệu trừu tượng là một mô hình toán học cùng với một tập hợp các phép toán trên nó. Có thể nói kiểu dữ liệu trừu tượng là một kiểu dữ liệu do chúng ta định nghĩa ở mức khái niệm (conceptual), nó chưa được cài đặt cụ thể bằng một ngôn ngữ lập trình.

Khi cài đặt một kiểu dữ liệu trừu tượng trên một ngôn ngữ lập trình cụ thể, chúng ta phải thực hiện hai nhiệm vụ:

1. Biểu diễn kiểu dữ liệu trừu tượng bằng một cấu trúc dữ liệu hoặc một kiểu dữ liệu trừu tượng khác đã được cài đặt.
2. Viết các chương trình con thực hiện các phép toán trên kiểu dữ liệu trừu tượng mà ta thường gọi là cài đặt các phép toán.

CHƯƠNG 2. CÁC KIỂU DỮ LIỆU TRỪU TƯỢNG CƠ BẢN

2.1. Danh sách

2.1.1. Khái niệm danh sách

Mô hình toán học của danh sách là một tập hợp hữu hạn các phần tử có cùng một kiểu, mà tổng quát ta gọi là kiểu phần tử (ElementType). Ta biểu diễn danh sách như là một chuỗi các phần tử của nó: $a_1, a_2, \ldots, a_n$ với $n \geq 0$. Nếu n=0 ta nói danh sách rỗng (empty list). Nếu $n > 0$ ta gọi a_1 là phần tử đầu tiên và a_n là phần tử cuối cùng của danh sách. Số phần tử của danh sách ta gọi là độ dài của danh sách.

Một tính chất quan trọng của danh sách là các phần tử của danh sách có thứ tự tuyến tính theo vị trí (position) xuất hiện của các phần tử. Ta nói a_i đứng trước a_{i+1}, với i từ 1 đến n-1; Tương tự ta nói a_i là phần tử đứng sau a_{i-1}, với i từ 2 đến n. Ta cũng nói a_i là phần tử tại vị trí thứ i, hay phần tử thứ i của danh sách.

Ví dụ: Tập hợp họ tên các sinh viên của lớp TINHOC 26 được liệt kê trên giấy như sau:

1. Nguyễn Trung Cang
2. Nguyễn Ngọc Chương
3. Lê Thị Lệ Sương
4. Trịnh Vu Thành
5. Nguyễn Phú Vinh

là một danh sách. Danh sách này gồm có 5 phần tử, mỗi phần tử có một vị trí trong danh sách theo thứ tự xuất hiện của nó.

2.1.2. Các phép toán trên danh sách

Để thiết lập kiểu dữ liệu trừu tượng danh sách (hay ngắn gọn là danh sách) ta phải định nghĩa các phép toán trên danh sách. Và như chúng ta sẽ thấy trong toàn bộ giáo trình, không có một tập hợp các phép toán nào thích hợp cho mọi ứng dụng (application). Vì vậy ở đây ta sẽ định nghĩa một số phép toán cơ bản nhất trên danh sách. Để thuận tiện cho việc định nghĩa ta giả sử rằng danh sách gồm các phần tử có kiểu là kiểu phần tử (ElementType); vị trí của các phần tử trong danh sách có kiểu là kiểu vị trí (position) và vị trí sau phần tử cuối cùng trong danh sách L là ENDLIST(L). Cần nhấn mạnh rằng khái niệm vị trí (position) là do ta định nghĩa, nó không phải là giá trị của các phần tử trong danh sách. Vị trí có thể là đồng nhất với vị trí lưu trữ phần tử hoặc không.

Các phép toán được định nghĩa trên danh sách là:

INSERT_LIST(x,p,L): xen phần tử x (kiểu ElementType) tại vị trí p (kiểu position) trong danh sách L. Tức là nếu danh sách là $a_1, a_2, \ldots, a_{p-1}, a_p, \ldots, a_n$ thì sau khi xen ta có kết quả $a_1, a_2, \ldots, a_{p-1}, x, a_p, \ldots, a_n$. Nếu vị trí p không tồn tại trong danh sách thì phép toán không được xác định.

LOCATE(x,L) thực hiện việc định vị phần tử có nội dung x đầu tiên trong danh sách L. Locate trả kết quả là vị trí (kiểu position) của phần tử x trong danh sách. Nếu x không có trong danh sách thì vị trí sau phần tử cuối cùng của danh sách được trả về, tức là ENDLIST(L).

RETRIEVE(p,L) lấy giá trị của phần tử ở vị trí p (kiểu position) của danh sách L; nếu vị trí p không có trong danh sách thì kết quả không xác định (có thể thông báo lỗi).

DELETE_LIST(p,L) chương trình con thực hiện việc xoá phần tử ở vị trí p (kiểu position) của danh sách. Nếu vị trí p không có trong danh sách thì phép toán không được định nghĩa và danh sách L sẽ không thay dổi

NEXT(p,L) cho kết quả là vị trí của phần tử (kiểu position) đi sau phần tử p; nếu p là phần tử cuối cùng trong danh sách L thì NEXT(p,L) cho kết quả là ENDLIST(L). Next không xác định nếu p không phải là vị trí của một phần tử trong danh sách.

PREVIOUS(p,L) cho kết quả là vị trí của phần tử đứng trước phần tử p trong danh sách. Nếu p là phần tử đầu tiên trong danh sách thì Previous(p,L) không xác định. Previous cũng không xác định trong trường hợp p không phải là vị trí của phần tử nào trong danh sách.

FIRST(L) cho kết quả là vị trí của phần tử đầu tiên trong danh sách. Nếu danh sách

rỗng thì ENDLIST(L) được trả về.

EMPTY_LIST(L) cho kết quả TRUE nếu danh sách có rỗng, ngược lại nó cho giá trị FALSE.

MAKENULL_LIST(L) khởi tạo một danh sách L rỗng.

Trong thiết kế các giải thuật sau này chúng ta dùng các phép toán trừu tượng đã được định nghĩa ở đây như là các phép toán nguyên thủy.

Ví dụ: Dùng các phép toán trừu tượng trên danh sách, viết một chương trình con nhận một tham số là danh sách rồi sắp xếp danh sách theo thứ tự tăng dần (giả sử các phần tử trong danh sách thuộc kiểu có thứ tự).

Giả sử SWAP(p,q) thực hiện việc dổi chỗ hai phần tử tại vị trí p và q trong danh sách, chương trình con sắp xếp được viết như sau:

```
void SORT(LIST L)
{
Position p,q; //kiểu vị trí của các phần tử trong danh sách
p= FIRST(L);//vị trí phần tử đầu tiên trong danh sách
while (p!=ENDLIST(L))
{
q=NEXT(p,L);//vị trí phần tử đứng ngay sau phần tử p
while (q!=ENDLIST(L))
{


if (RETRIEVE(p,L) > RETRIEVE(q,L))
swap(p,q); // dịch chuyển nội dung phần tử
q=NEXT(q,L);
}
p=NEXT(p,L);
}
}
```

Tuy nhiên, cần phải nhấn mạnh rằng, đây là các phép toán trừu tượng do chúng ta định nghĩa, nó chưa được cài đặt trong các ngôn ngữ lập trình. Do đó, để cài đặt giải thuật thành một chương trình chạy được thì ta cũng phải cài đặt các phép toán thành các chương trình con trong chương trình. Hơn nữa, trong khi cài đặt cụ thể, một số tham số hình thức trong các phép toán trừu tượng không đóng vai trò gì trong chương trình con cài đặt chúng, do vậy ta có thể bỏ qua nó trong danh sách tham số của chương trình con. Ví dụ: phép toán trừu tượng INSERT_LIST(x,p,L) có 3 tham số hình thức: phần tử muốn thêm x, vị trí thêm vào p và danh sách được thêm vào L. Nhưng khi cài đặt danh sách bằng con trỏ (danh sách liên kết đơn), tham số L là không cần thiết vì với cấu trúc này chỉ có con trỏ tại vị trí p phải thay đổi để nối kết với ô chứa phần tử mới. Trong bài giảng này, ta vẫn giữ đúng những tham số trong cách cài đặt để làm cho chương trình đồng nhất và trong suốt đói với các phương pháp cài đặt của cùng một kiểu dữ liệu trừu tượng.

2.1.3. Cài đặt danh sách

a. Cài đặt danh sách bằng mảng (danh sách đặc)

Ta có thể cài đặt danh sách bằng mảng như sau: *dùng một mảng để lưu giữ liên tiếp các phần tử của danh sách từ vị trí đầu tiên của mảng*. Với cách cài đặt này, dĩ nhiên, ta phải ước lượng số phần tử của danh sách để khai báo số phần tử của mảng cho thích hợp. Dễ thấy rằng số phần tử của mảng phải được khai báo không ít hơn số phần tử của danh sách. Nói chung là mảng còn thừa một số chỗ trống. Mặt khác ta phải lưu giữ độ dài hiện tại của danh sách, độ dài này cho biết danh sách có bao nhiêu phần tử và cho biết phần nào của mảng còn trống như trong hình II.1. *Ta định nghĩa vị trí của một phần tử trong danh sách là chỉ số của mảng tại vị trí lưu trữ phần tử đó + 1(**vì phần tử đầu tiên trong mảng là chỉ số 0**).*

Chỉ số	0	1	…	Last-1	…	Maxlength-1
Nội dung phần tử	Phần tử thứ 1	Phần tử thứ 2	…	Phần tử cuối cùng trong danh sách	…	

Cài đặt danh sách bằng mảng

Với hình ảnh minh họa trên, ta cần **các khai báo cần thiết là** #define MaxLength ...

//Số nguyên thích hợp để chỉ độ dài của danh sách typedef ... ElementType;//kiểu của phần tử trong danh sách

```
typedef int Position; //kiểu vị trí cuả các phần tử
typedef struct {
	ElementType Elements[MaxLength];
	 //mảng chứa các phần tử của danh sách
	Position Last;
	 //giữ độ dài danh sách
	} List;
```

Trên đây là sự biểu diễn kiểu dữ liệu trừu trượng danh sách bằng cấu trúc dữ liệu mảng. Phần tiếp theo là cài đặt các phép toán cơ bản trên danh sách.

Khởi tạo danh sách rỗng

Danh sách rỗng là một danh sách không chứa bất kỳ một phần tử nào (hay độ dài danh sách bằng 0). Theo cách khai báo trên, trường Last chỉ vị trí của phần tử cuối cùng trong danh sách và đó cũng độ dài hiện tại của danh sách, vì vậy để khởi tạo danh sách rỗng ta chỉ việc gán giá trị trường Last này bằng 0.

```
void MakeNull_List(List *L)
	{
	 L->Last=0;
	}
```

Câu hỏi áp dụng:

1. Hãy trình bày cách gọi thực thi chương trình con tạo danh sách rỗng trên?
2. Hãy giải thích cách khai báo tham số hình thức trong chương trình con và cách truyền tham số khi gọi chương trình con đó?

Kiểm tra danh sách rỗng

Danh sách rỗng là một danh sách mà độ dài của nó bằng 0.

```
int Empty_List(List L){
return L.Last==0;
}
```

Xen một phần tử vào danh sách

Khi xen phần tử có nội dung x vào tại vị trí p của danh sách L thì sẽ xuất hiện các khả năng sau:

- Mảng đầy: mọi phần tử của mảng đều chứa phần tử của danh sách, tức là phần tử cuối cùng của danh sách nằm ở vị trí cuối cùng trong mảng. Nói cách khác, độ dài của danh sách bằng chỉ số tối đa của mảng; Khi đó, không còn chỗ cho phần tử mới, vì vậy việc xen là không thể thực hiện được, chương trình con gặp lỗi.

- Ngược lại ta tiếp tục xét:

Nếu p không hợp lệ (p>last+1 hoặc p<1) thì chương trình con gặp lỗi; (Vị trí xen p<1 thì khi đó p không phải là một vị trí phần tử trong trong danh sách đặc. Nếu vị trí p>L.last+1 thì khi xen sẽ làm cho danh sách L không còn là một danh sách đặc nữa vì nó có một vị trí trong mảng mà chưa có nội dung.)

- Nếu vị trí p hợp lệ thì ta tiến hành xen theo các bước sau:

+ Dời các phần tử từ vị trí p đến cuối danh sách ra sau 1 vị trí.

+ Độ dài danh sách tăng 1.
+ Đua phần tử mới vào vị trí p

Chương trình con xen phần tử x vào vị trí p của danh sách L có thể viết như sau:

```
void Insert_List(ElementType X, Position P, List *L)
{
    if (L->Last==MaxLength) printf("Danh sach day");
    else if ((P<1) || (P>L->Last+1)) printf("Vi tri khong hop le");
    else{
Position Q;
/*Dời các phần tử từ vị trí p (chỉ số trong mảng là p-1) đến cuối danh sách sang
phải 1 vị trí*/
for(Q=(L->Last-1)+1;Q>P-1;Q--)
    L->Elements[Q]=L->Elements[Q-1]; //Đưa x vào vị trí p
L->Elements[P-1]=X;
L->Last++;//Tăng độ dài danh sách lên 1
}
}
```

Xóa phần tử ra khỏi danh sách

Xoá một phần tử ở vị trí p ra khỏi danh sách L ta làm công việc ngược lại với xen.

Trước tiên ta kiểm tra vị trí phần tử cần xóa xem có hợp lệ hay chưa. Nếu p>L.last hoặc p<1 thì đây không phải là vị trí của phần tử trong danh sách.

Ngược lại, vị trí đã hợp lệ thì ta phải dời các phần tử từ vị trí p+1 đến cuối danh sách ra trước một vị trí và độ dài danh sách giảm di 1 phần tử (do đã xóa bớt 1 phần tử).

```
void Delete_List(Position P,List *L)
{
if ((P<1) || (P>L->Last))
printf("Vi tri khong hop le");
else if (EmptyList(*L))
printf("Danh sach rong!");
else{
Position Q;
/*Dời các phần tử từ vị trí p+1 (chỉ số trong mảng là p) đến cuối danh sách sang
trái 1 vị trí*/
for(Q=P-1;Q<L->Last-1;Q++)
L->Elements[Q]=L->Elements[Q+1];
L->Last--;
}
}
```

Định vị một phần tử trong danh sách

Để định vị vị trí phần tử đầu tiên có nội dung x trong danh sách L, ta tiến hành dò tìm từ đầu danh sách. Nếu tìm thấy x thì vị trí của phần tử tìm thấy được trả về, nếu không tìm thấy thì hàm trả về vị trí sau vị trí của phần tử cuối cùng trong danh sách, tức là ENDLIST(L) (ENDLIST(L)= L.Last+1). Trong trường hợp có nhiều phần tử cùng giá trị x trong danh sách thì vị trí của phần tử được tìm thấy đầu tiên được trả về.

```
Position Locate(ElementType X, List L)
{
Position P;
int Found = 0;
P = First(L); //vị trí phần tử đầu tiên
/*trong khi chưa tìm thấy và chưa kết thúc danh sách thì xét phần tử kế tiếp*/
while ((P != EndList(L)) && (Found == 0))
if (Retrieve(P,L) == X) Found = 1;
```

```
else P = Next(P, L);
return P;
}
```

Lưu ý : Các phép toán sau phải thiết kế trước Locate

- *First(L)=1*
- *Retrieve(P,L)=L.Elements[P-1]*
- *EndList(L)=L.Last+1*
- *Next(P,L)=P+1*

Ví dụ : Vận dụng các phép toán trên danh sách đặc để viết chương trình nhập vào một danh sách các số nguyên và hiển thị danh sách vừa nhập ra màn hình. Thêm phần tử có nội dung x vào danh sách tại ví trí p (trong đó x và p được nhập từ bàn phím). Xóa phần tử đầu tiên có nội dung x (nhập từ bàn phím) ra khỏi danh sách.

Hướng giải quyết :

Giả sử ta đã cài đặt đầy đủ các phép toán cơ bản trên danh sách. Để thực hiện yêu cầu như trên, ta cần thiết kế thêm một số chương trình con sau :

- Nhập danh sách từ bàn phím (READ_LIST(L)) (Phép toán này chưa có trong kiểu danh sách)
- Hiển thị danh sách ra màn hình (in danh sách) (PRINT_LIST(L)) (Phép toán này chưa có trong kiểu danh sách).

Thực ra thì chúng ta chỉ cần sử dụng các phép toán MakeNull_List, Insert_List, Delete_List, Locate và các chương trình con Read_List, Print_List vừa nói trên là có thể giải quyết được bài toán. Để dáp ứng yêu cầu đặt ra, ta viết chương trình chính để nối kết những chương trình con lại với nhau như sau:

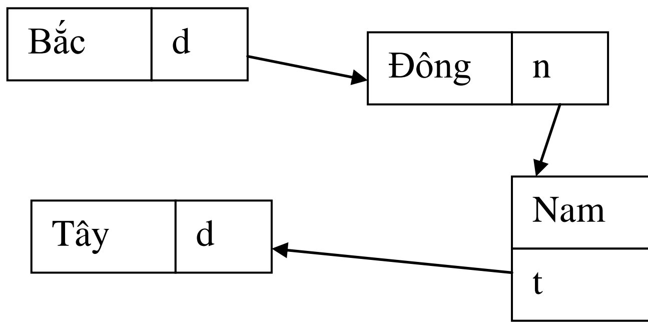
```
int main()
{
List L;
ElementType X;
Position P;
MakeNull_List(&L); //Khởi tạo danh sách rỗng
Read_List(&L);
printf("Danh sach vua nhap: ");
Print_List(L); // In danh sach len man hinh
printf("Phan tu can them: ");
scanf("%d",&X);
printf("Vi tri can them: ");
scanf("%d",&P);
Insert_List(X,P,&L);
printf("Danh sach sau khi them phan tu la: ");
PrintList(L);
printf("Noi dung phan tu can xoa: ");
scanf("%d",&X); P=Locate(X,L);
Delete_List(P,&L);
printf("Danh sach sau khi xoa %d la: ",X);
Print_List(L);
return 0;
}
```

b. Cài đặt danh sách bằng con trỏ (danh sách liên kết)

Cách khác để cài đặt danh sách là dùng con trỏ để liên kết các ô chứa các phần tử. Trong cách cài đặt này các phần tử của danh sách được lưu trữ trong các ô, mỗi ô có thể chỉ đến ô chứa phần tử kế tiếp trong danh sách. Bạn đọc có thể hình dung cơ chế này qua ví dụ sau:

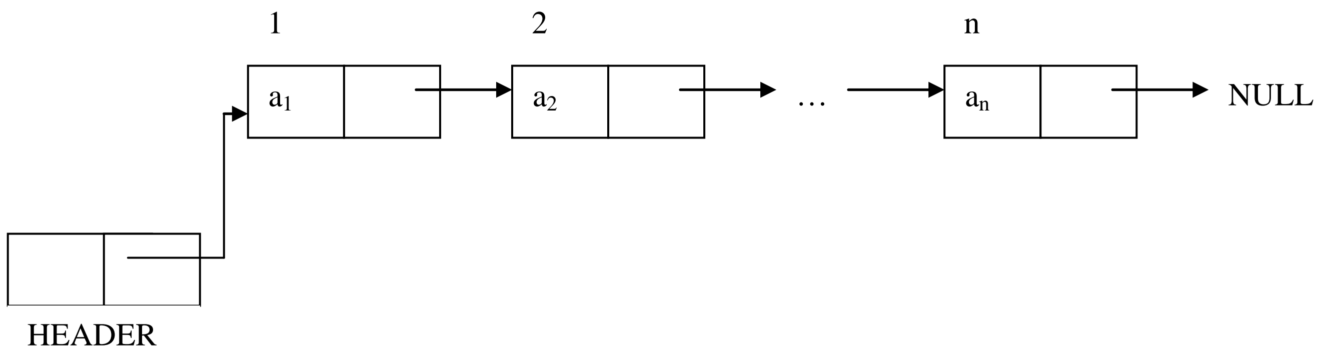
Giả sử 1 lớp có 4 bạn: Đông, Tây, Nam, Bắc có địa chỉ lần lượt là d, t, n, b. Giả sử:

Đông có địa chỉ của Nam, Tây không có địa chỉ của bạn nào, Bắc giữ địa chỉ của Đông, Nam có địa chỉ của Tây (xem hình vẽ sau).



Như vậy, nếu ta xét thứ tự các phần tử bằng cơ chế chỉ đến này thì ta có một danh sách: Bắc, Đông, Nam, Tây. Hơn nữa, để có danh sách này thì ta cần và chỉ cần giữ địa chỉ của Bắc.

Trong cài đặt, để một ô có thể *chỉ đến* ô khác ta cài đặt mỗi ô là một mẩu tin (**record**, **struct**) có hai trường: trường Element giữ giá trị của các phần tử trong danh sách; trường next là một *con trỏ* giữ địa chỉ của ô kế tiếp. Trường next của phần tử cuối trong danh sách chỉ đến một giá trị đặc biệt là **NULL**. Cấu trúc như vậy gọi là danh sách cài đặt bằng con trỏ hay *danh sách liên kết đơn* hay ngắn gọn là *danh sách liên kết*. Ví dụ (hình b1):



Để quản lý danh sách ta chỉ cần một biến giữ địa chỉ ô chứa phần tử đầu tiên của danh sách, tức là một con trỏ trỏ đến phần tử đầu tiên trong danh sách. Biến này gọi là *chỉ điểm đầu danh sách (Header)* . Để đơn giản hóa vấn đề, trong chi tiết cài đặt, Header là một biến cùng kiểu với các ô chứa các phần tử của danh sách và nó có thể được cấp phát ô nhớ y như một ô chứa phần tử của danh sách (hình b1). Tuy nhiên, Header là một ô đặc biệt nên nó không chứa phần tử nào của danh sách, trường dữ liệu của ô này là rỗng, chỉ có trường con trỏ Next trỏ tới ô chứa phần tử đầu tiên thật sự của danh sách. Nếu danh sách rỗng thì Header->next trỏ tới **NULL**. Việc cấp phát ô nhớ cho Header như là một ô chứa dữ liệu bình thường nhằm tăng tính đơn giản của các giải thuật thêm, xoá các phần tử trong danh sách.

Ở đây, ta cần phân biệt rõ giá trị của một phần tử và vị trí (position) của nó trong cấu trúc trên. Ví dụ, giá trị của phần tử đầu tiên của danh sách trong hình vẽ trên là a_1, Trong khi vị trí của nó là địa chỉ của ô chứa nó, tức là giá trị nằm ở trường next của ô Header. Giá trị và vị trí của các phần tử của danh sách trong hình trên như sau:

Phần tử thứ	Giá trị	Vị trí
1	a1	HEADER 1
2	a2	1
...	...	...
n	an	(n-1)
Sau phần tử cuối cùng	Không xác định	N và n->next có giá trị là **NULL**

Như đã thấy trong bảng trên, vị trí của phần tử thứ i là (i-1), như vậy để biết được vị trí của phần tử thứ i ta phải truy xuất vào ô thứ (i-1). Khi thêm hoặc xoá một phần tử trong danh sách liên kết tại vị trí p, ta phải cập nhật lại con trỏ trỏ tới vị trí này, tức là cập nhật lại (p-1). Nói cách khác, để thao tác vào vị trí p ta phải biết con trỏ trỏ vào p mà con trỏ này chính là (p-1). Do đó, ta định nghĩa p-1 như là vị trí của p. Có thể nói nôm na rằng vị trí của phần tử a_i là địa chỉ của ô đứng ngay phía trước ô chứa a_i. Hay chính xác hơn, ta nói, vị trí của phần tử thứ i là con trỏ trỏ tới ô có trường next trỏ tới ô chứa phần tử a_i Như vậy vị trí của phần tử thứ 1 là con trỏ trỏ đến Header, vị trí phần tử thứ 2 là con trỏ trỏ ô chứa phần tử a_1, vị trí của phần tử thứ 3 là con trỏ trỏ ô a_2, ..., vị trí phần tử thứ n là con trỏ trỏ ô chứa a_{n-1}. Vậy vị trí sau phần tử cuối trong danh sách, tức là ENDLIST, chính là con trỏ trỏ ô chứa phần tử a_n (xem hìnhvẽ).

Theo định nghĩa này ta có, nếu p là vị trí của phần tử thứ p trong danh sách thì giá trị của phần tử ở vị trí p này nằm trong trường element của ô được trỏ bởi p->next. Nói cách khác, p->next->element chứa nội dung của phần tử ở vị trí p trong danh sách.

Các khai báo cần thiết là

```
typedef ... ElementType; //kiểu của phần tử trong danh sách
typedef struct Node
{
ElementType Element;//Chứa nội dung của phần tử
Node* Next; /*con trỏ chỉ đến phần tử kế tiếp trong danh sách*/
};
typedef Node* Position; // Kiểu vị trí
typedef Position List;
```

Tạo danh sách rỗng

Như đã nói ở phần trên, ta dùng Header như là một biến con trỏ có kiểu giống như

kiểu của một ô chứa một phần tử của danh sách. Tuy nhiên, trường Element của Header không bao giờ được dùng, chỉ có trường Next dùng để trỏ tới ô chứa phần tử đầu tiên của danh sách. Vậy nếu như danh sách rỗng thì trường ô Header vẫn phải tồn tại và ô này có trường next chỉ đến **NULL** (do không có một phần tử nào). Vì vậy khi khởi tạo danh sách rỗng, ta phải cấp phát ô nhớ cho HEADER và cho con trỏ trong trường next của nó trỏ tới **NULL**.

```
void MakeNull_List(List *Header)
{
(*Header)=(Node*)malloc(sizeof(Node));
(*Header)->Next= NULL;
}
```

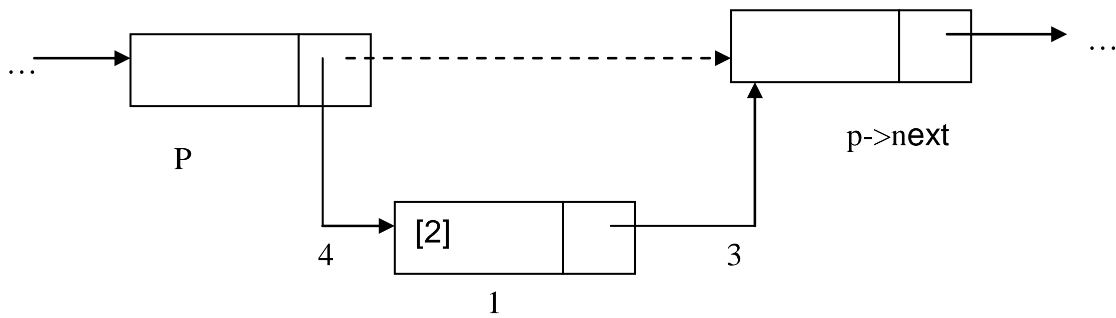
Kiểm tra một danh sách rỗng

Danh sách rỗng nếu như trường next trong ô Header trỏ tới **NULL**.

```
int Empty_List(List L)
{
return (L->Next==NULL);
}
```

Xen một phần tử vào danh sách :

Xen một phần tử có giá trị x vào danh sách L tại vị trí p ta phải cấp phát một ô mới để lưu trữ phần tử mới này và nối kết lại các con trỏ để đưa ô mới này vào vị trí p. Sơ đồ nối kết và thứ tự các thao tác được cho trong hình b2.



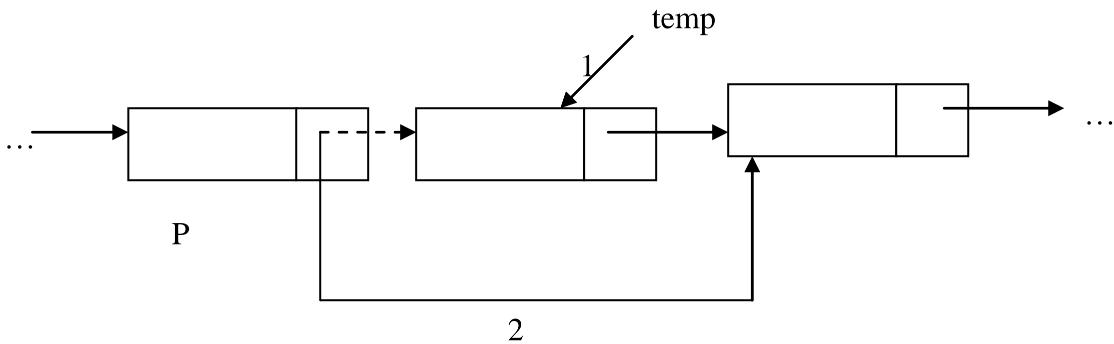
Hình b2: Thêm một phần tử vào danh sách tại vị trí p

```
void Insert_List(ElementType X, Position P, List *L)
{
Position T;
T=(Node*)malloc(sizeof(Node));
T->Element=X;
T->Next=P->Next;
P->Next=T;
}
```

Câu hỏi ôn tập:

Tham số L (danh sách) trong chương trình con trên có bỏ được không? Tại sao?

Xóa phần tử ra khỏi danh sách



Hình b3: Xóa một phần tử tại vị trí p

Tương tự như khi xen một phần tử vào danh sách liên kết, muốn xóa một phần tử khỏi danh sách ta cần biết vị trí p của phần tử muốn xóa trong danh sách L. Nối kết lại các con trỏ bằng cách cho p trỏ tới phần tử đứng sau phần tử thứ p. Trong các ngôn ngữ lập trình không có cơ chế thu hồi vùng nhớ tự động như ngôn ngữ Pascal, C thì ta phải thu hồi vùng nhớ của ô bị xóa một các tường minh trong giải thuật. Tuy nhiên, vì tính đơn giản của giải thuật cho nên đôi khi chúng ta không đề cập đến việc thư hồi vùng nhớ cho các ô bị xoá. Chi tiết và trình tự các thao tác để xoá một phần tử trong danh sách liên kết như trong hình b3. Chương trình con có thể được cài đặt như sau:

```
void Delete_List(Position P, List *L)
{
Position T;
if (P->Next!=NULL)
{
T=P->Next; /*/giữ ô chứa phần tử bị xoá để thư hồi vùng nhớ*/
P->Next=T->Next; /*nối kết con trỏ trỏ tới phần tử thứ p+1*/
free(T); //thư hồi vùng nhớ
}
}
```

Định vị một phần tử trong danh sách liên kết

Để định vị phần tử x trong danh sách L ta tiến hành tìm từ đầu danh sách (ô header) nếu tìm thấy thì vị trí của phần tử đầu tiên được tìm thấy sẽ được trả về nếu không thì ENDLIST(L) được trả về. Nếu x có trong sách sách thì hàm Locate trả về vị trí p mà trong đó ta có x = p->next->element.

```
Position Locate(ElementType X, List L)
{
Position P;
int Found = 0;
P = L;
while ((P->Next != NULL) && (Found == 0))
if (P- >Next->Element == X) Found = 1;
else P = P->Next;
return P;
}
```

Thực chất, khi gọi hàm Locate ở trên ta có thể truyền giá trị cho L là bất kỳ giá trị nào. Nếu L là Header thì chương trình con sẽ tìm x từ đầu danh sách. Nếu L là một vị trí p bất kỳ trong danh sách thì chương trình con Locate sẽ tiến hành định vị phần tử x từ vị trí p.

Xác định nội dung phần tử:

Nội dung phần tử đang lưu trữ tại vị trí p trong danh sách L là p->next->Element. Do đó, hàm sẽ trả về giá trị p->next->element nếu phần tử có tồn tại, ngược lại phần tử không tồn tại (p->next=**NULL**) thì hàm không xác định

```
ElementType Retrieve(Position P, List L)
{
if (P->Next!=NULL)
return P->Next->Element;
}
```

Hãy thiết kế hàm Locate bằng cách sử dụng các phép toán trừu tượng co bản trên danh sách?

c. So sánh hai phương pháp cài đặt

Không thể kết luận phương pháp cài đặt nào hiệu quả hơn, mà nó hoàn toàn tuỳ thuộc vào từng ứng dụng hay tuỳ thuộc vào các phép toán trên danh sách. Tuy nhiên, ta có thể tổng kết một số ưu nhược điểm của từng phương pháp làm cơ sở để lựa chọn phương pháp cài đặt thích hợp cho từng ứng dụng:

Cài đặt bằng mảng đòi hỏi phải xác định số phần tử của mảng, do đó nếu không thể ước lượng được số phần tử trong danh sách thì khó áp dụng cách cài đặt này một cách hiệu quả vì nếu khai báo thiếu chỗ thì mảng thường xuyên bị đầy, không thể làm việc được còn nếu khai báo quá thừa thì lãng phí bộ nhớ.

Cài đặt bằng con trỏ thích hợp cho sự biến động của danh sách, danh sách có thể rỗng hoặc lớn tuỳ ý chỉ phụ thuộc vào bộ nhớ tối đa của máy. Tuy nhiên, ta phải tốn thêm vùng nhớ cho các con trỏ (trường next).

Cài đặt bằng mảng thì thời gian xen hoặc xoá một phần tử tỉ lệ với số phần tử di sau vị trí xen/ xóa. Trong khi cài đặt bằng con trỏ các phép toán này mất chỉ một hằng thời gian.

Phép truy nhập vào một phần tử trong danh sách, chẳng hạn như PREVIOUS, chỉ tốn một hằng thời gian đói với cài đặt bằng mảng, trong khi đói với danh sách cài đặt bằng con trỏ ta phải tìm từ đầu danh sách cho đến vị trí trước vị trí của phần tử hiện hành.Nói chung, *danh sách liên kết thích hợp với danh sách có nhiều biến động*, tức là ta thường xuyên thêm, xoá các phần tử.

d. Cài đặt bằng con nháy

Một số ngôn ngữ lập trình không có cũng cấp kiểu con trỏ. Trong trường hợp này ta có thể "giả" con trỏ để cài đặt danh sách liên kết. Ý tuởng chính là: dùng mảng để chứa các phần

tử của danh sách, các "con trỏ" sẽ là các biến số nguyên (**int**) để giữ chỉ số của phần tử kế tiếp trong mảng. Để phân biệt giữa "con trỏ thật" và "con trỏ giả" ta gọi các con trỏ giả này là con nháy (cursor). Như vậy để cài đặt danh sách bằng con nháy ta cần một mảng mà mỗi phần tử xem như là một ô gồm có hai trường: trường Element như thông lệ giữ giá trị của phần tử trong danh sách (có kiểu Elementtype) trường Next là con nháy để chỉ tới vị trí trong mảng của phần tử kế tiếp. Chẳng hạn hình d.1 biểu diễn cho mảng SPACE đang chứa hai danh sách L_1, L_2. Để quản lí các danh sách ta cũng cần một con nháy chỉ đến phần tử đầu của mỗi danh sách (giống như header trong danh sách liên kết). Phần tử cuối cùng của danh sách ta cho chỉ tới giá trị đặc biệt **Null**, có thể xem **Null** = -1 với một giả thiết là mảng SPACE không có vị trí nào có chỉ số -1.

Trong hình d.1, danh sách L_1 gồm 3 phần tử : f, o ,r. Chỉ điểm đầu của L_1 là con nháy có giá trị 5, tức là nó trỏ vào ô lưu giữ phần tử đầu tiên của L_1, trường next của ô này có giá trị 1 là ô lưu trữ phần tử kế tiếp (tức là o). Trường next tại ô chứa o là 4 là ô lưu trữ phần tử kế tiếp trong danh sách (tức là r). Cuối cùng trường next của ô này chứa **Null** nghĩa là danh sách không còn phần tử kế tiếp.

Phân tích tương tự ta có L_2 gồm 4 phần tử : w, i, n, d

Chỉ số	Elements	Next
0	d	**Null**
1	o	4
2		
3	n	0
4	r	**Null**
5	f	1
6	i	3
7	w	6
8		
9		

Chỉ điểm của danh sách thứ nhất L_1 → (5)

Chỉ điểm của danh sách thứ hai L_2 → (7)

Mảng SPACE

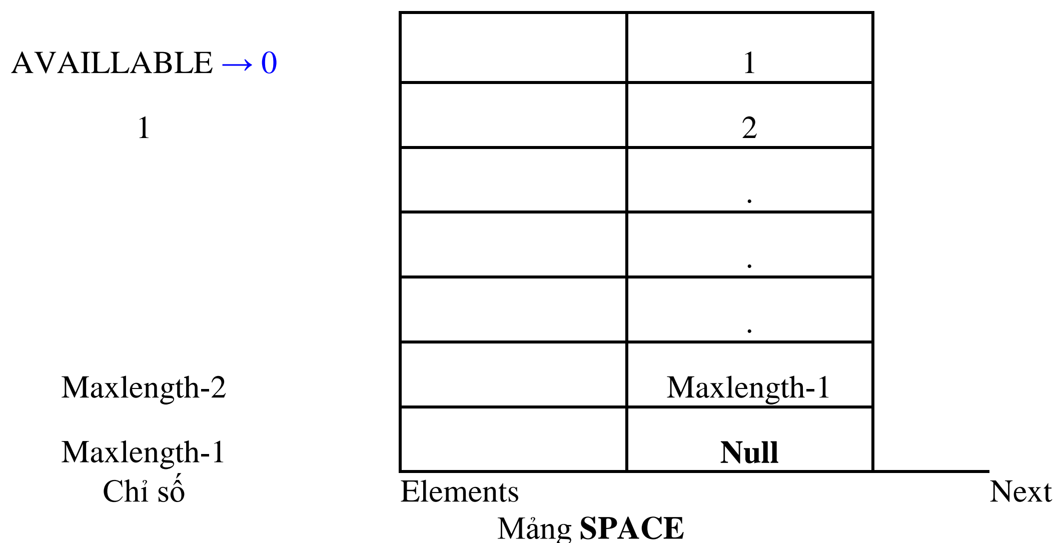
Hình d.1 Mảng đang chứa hai danh sách L_1 và L_2

Khi xen một phần tử vào danh sách ta lấy một ô trống trong mảng để chứa phần tử mới này và nối kết lại các con nháy. Ngược lại, khi xoá một phần tử khỏi danh sách ta nối kết lại các con nháy để loại phần tử này khỏi danh sách, điều này kéo theo số ô trống trong mảng tăng lên 1. Vấn đề là làm thế nào để quản lí các ô trống này để biết ô nào còn trống ô nào đã dùng? một giải pháp là *liên kết tất cả các ô trống vào một danh sách đặc biệt gọi là AVAILABLE*, khi xen một phần tử vào danh sách ta lấy ô trống đầu AVAILABLE để chứa phần tử mới này. Khi xoá một phần tử từ danh sách ta cho ô bị xoá nối vào đầu AVAILABLE. Tất nhiên khi mới khởi đầu việc xây dựng cấu trúc thì mảng chưa chứa phần tử nào của bất kỳ một danh sách nào. Lúc này tất cả các ô của mảng đều là ô trống, và như

vậy, tất cả các ô đều được liên kết vào trong AVAILABLE. Việc khởi tạo AVAILABLE ban đầu có thể thực hiện bằng cách cho phần tử thứ i của mảng trỏ tới phần tử i+1.

Các khai báo cần thiết cho danh sách

```
#define MaxLength ... //Chieu dai mang
#define Null -1 //Gia tri Null
typedef ... ElementType; /*kiểu của các phần tử trong danh sách*/
typedef struct{
ElementType Elements; /*trường chứa phần tử trong danh sách*/
int Next; //con nháy trỏ đến phần tử kế tiếp
} Node;
Node Space[MaxLength]; //Mang toan cuc
int Available;
```



Mảng **SPACE**

Hình d2: Khởi tạo Available ban đầu

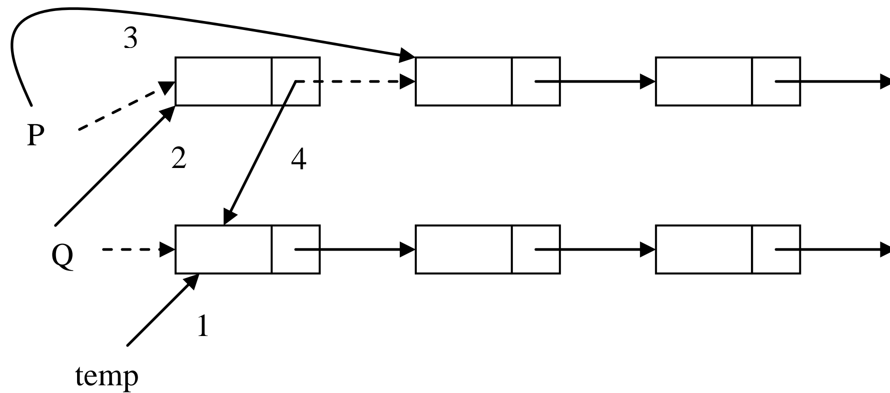
Khởi tạo cấu trúc – Thiết lập available ban đầu

Ta cho phần tử thứ 0 của mảng trỏ đến phần tử thứ 1,..., phần tử cuối cùng trỏ **Null**. Chỉ điểm đầu của AVAILABLE là 0 như trong hình II.7

```
void Initialize()
{
 int i;
for(i=0;i<MaxLength-1;i++)
Space[i].Next=i+1;
Space[MaxLength-1].Next=NULL;
Available=0;
}
```

Chuyển một ô từ danh sách này sang danh sách khác

Ta thấy thực chất của việc xen hay xoá một phần tử là thực hiện việc chuyển một ô từ danh sách này sang danh sách khác. Chẳng hạn muốn xen thêm một phần tử vào danh sách L_1 trong hình II.6 vào một vị trí p nào đó ta phải chuyển một ô từ AVAILABLE (tức là một ô trống) vào L_1 tại vị trí p; muốn xoá một phần tử tại vị trí p nào đó trong danh sách L_2, chẳng hạn, ta chuyển ô chứa phần tử đó sang AVAILABLE, thao tác này xem như là giải phóng bộ nhớ bị chiếm bởi phần tử này. Do đó, tốt nhất ta viết một hàm thực hiện thao tác chuyển một ô từ danh sách này sang danh sách khác và hàm cho kết quả kiểu int tùy theo chuyển thành công hay thất bại (là 0 nếu chuyển không thành công, 1 nếu chuyển thành công). Hàm **Move** sau đây thực hiện chuyển ô được trỏ tới bởi con nháy P vào danh sách khác được trỏ bởi con nháy Q như trong hình d3. Hình d3 trình bày các thao tác cơ bản để chuyển một ô (ô được chuyển ta tạm gọi là ô mới):



Hình d3:

Chuyển 1 ô từ danh sách này sang danh sách khác (các liên kết vẽ bằng nét dứt biểu diễn cho các liên kết cũ - trước khi giải thuật bắt đầu)

-Dùng con nháy temp để trỏ ô được trỏ bởi Q.

-Cho Q trỏ tới ô mới.

-Cập nhật lại con nháy P bằng cách cho nó trỏ tới ô kế tiếp.

-Nối con nháy trường next của ô mới (ô mà Q đang trỏ) trỏ vào ô mà temp đang trỏ.

```
int Move(int *p, int *q)
{
 int temp;
if (*p==Null)
return 0;
//Khơng co o de chuyen
else
{
temp=*q;
*q=*p;
*p=Space[*q].Next;
Space[*q].Next=temp;
return 1; //Chuyen thanh cong
}
}
```

Trong cách cài đặt này, khái niệm vị trí tương tự như khái niệm vị trí trong trường hợp cài đặt bằng con trỏ, tức là, vị trí của phần tử thứ I trong danh sách là chỉ số của ô trong mảng chứa con nháy trỏ đến ô chứa phần tử thứ i. Ví dụ xét danh sách L_1 trong hình d.1, vị trí của phần tử thứ 2 trong danh sách (phần tử có giá trị o) là 5, không phải là 1; vị trí của phần tử thứ 3 (phần tử có giá trị r) là 1, không phải là 4. Vị trí của phần tử thứ 1 (phần tử có giá trị f) được định nghĩa là -1, vì không có ô nào trong mảng chứa con nháy trỏ đến ô chứa phần tử f.

Xen một phần tử vào danh sách

Muốn xen một phần tử vào danh sách ta cần biết *vị trí xen*, gọi là *p*, *rồi ta ch*uy*ển ô đầu của available vào vị trí này*. Chú ý rằng vị trí của phần tử đầu tiên trong danh sách được định nghĩa là -1, do đó nếu p=-1 có nghĩa là thực hiện việc thêm vào đầu danh sách.

```
void Insert_List(ElementType X, int P, int *L)
{ if (P = = -1) //Xen dau danh sach
{
if (Move(&Available,L)) Space[*L].Elements=X;
else printf("Loi! Khong con bo nho trong");
}
else //Chuyen mot o tu Available vao vi tri P
{
```

```
if (Move(&Available,&Space[P].Next))
// O nhan X la o tro boi Space[p].Next
Space[Space[P].Next].Elements=X;
else printf("Loi! Khong con bo nho trong");
}
}
```

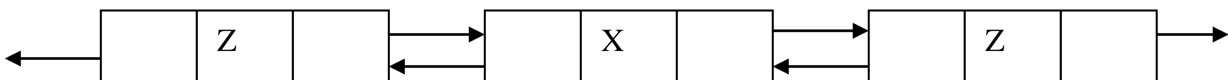
Xoá một phần tử trong danh sách

Muốn *xoá một phần tử tại vị trí p* trong danh sách ta chỉ cần *chuyển ô chứa phần tử tại vị trí này vào đầu AVAILABLE*. Tương tự như phép thêm vào, nếu p=-1 thì xoá phần tử đầu danh sách.

```
void Delete_List(int p, int *L)
{
if (p==-1) //Neu la o dau tien
{
if (!Move(L,&Available)) printf("Loi trong khi xoa");
// else Khong lam gi ca
}
else
if (!Move(&Space[p].Next,&Available))
printf("Loi trong khi xoa");
//else Khong lam gi
}
```

2.1.4. Các dạng danh sách liên kết (DSLK): DSLK đơn, vòng, kép, …

Một số ứng dụng đòi hỏi chúng ta phải duyệt danh sách theo cả hai chiều một cách hiệu quả. Chẳng hạn, cho phần tử X cần biết ngay phần tử trước X và sau X một cách mau chóng. Trong trường hợp này ta phải dùng hai con trỏ, một con trỏ chỉ đến phần tử đứng sau (next), một con trỏ chỉ đến phần tử đứng trước (previous). Với cách tổ chức này ta có một danh sách liên kết kép. Dạng của một danh sách liên kép như sau:



Hình II.15 Hình ảnh một danh sách liên kết kép

Các khai báo cần thiết

```
typedef ... ElementType;
//kiểu nội dung của các phần tử trong danh sách
typedef struct Node{
ElementType Element; //lưu trữ nội dung phần tử
//Hai con trỏ trỏ tới phần tử trước và sau
Node* Prev;
Node* Next;
};
typedef Node* Position;
typedef Position DoubleList;
```

Để quản lí một danh sách liên kết kép ta có thể dùng một con trỏ trỏ đến một ô bất kỳ trong cấu trúc. Hoàn toàn tương tự như trong danh sách liên kết đơn đã trình bày trong phần trước, con trỏ để quản lí danh sách liên kết kép có thể là một con trỏ có kiểu giống như kiểu phần tử trong danh sách và nó có thể được cấp phát ô nhớ (tương tự như Header trong danh sách liên kết đơn) hoặc không được cấp phát ô nhớ. Ta cũng có thể xem danh sách liên kết kép như là danh sách liên kết đơn, với một bổ sung duy nhất là có con trỏ previous để nối kết các ô theo chiều ngược lại. Theo quan điểm này thì chúng ta có thể cài đặt các phép toán thêm

(insert), xoá (delete) một phần tử hoàn toàn tương tự như trong danh sách liên kết đơn và con trỏ Header cũng cần thiết như trong danh sách liên kết đơn, vì nó chính là vị trí của phần tử đầu tiên trong danh sách.

Tuy nhiên, nếu tận dụng khả năng duyệt theo cả hai chiều thì ta *không cần phải cấp phát bộ nhớ cho Header* và vị trí (position) của một phần tử trong danh sách có thể định nghĩa như sau: *Vị trí của phần tử a_i là con trỏ trỏ tới ô chứa a_i, tức là địa chỉ ô nhớ chứa a_i. Nói nôm na, vị trí của a_i là ô chứa a_i*. Theo định nghĩa vị trí như vậy các phép toán trên danh sách liên kết kép sẽ được cài đặt hơi khác với danh sách liên đơn. Trong cách cài đặt này, chúng ta không sử dụng ô đầu mục như danh sách liên kết đơn mà sẽ quản lý danh sách một các trực tiếp (header chỉ ngay đến ô đầu tiên trong danh sách).

Tạo danh sách liên kết kép rỗng

Giả sử DL là con trỏ quản lí danh sách liên kết kép thì khi khởi tạo danh sách rỗng ta cho con trỏ này trỏ **NULL** (không cấp phát ô nhớ cho DL), tức là gán DL=**NULL**.

```
void MakeNull_List (DoubleList *DL)
{
(*DL)= NULL;
}
```

Kiểm tra danh sách liên kết kép rỗng

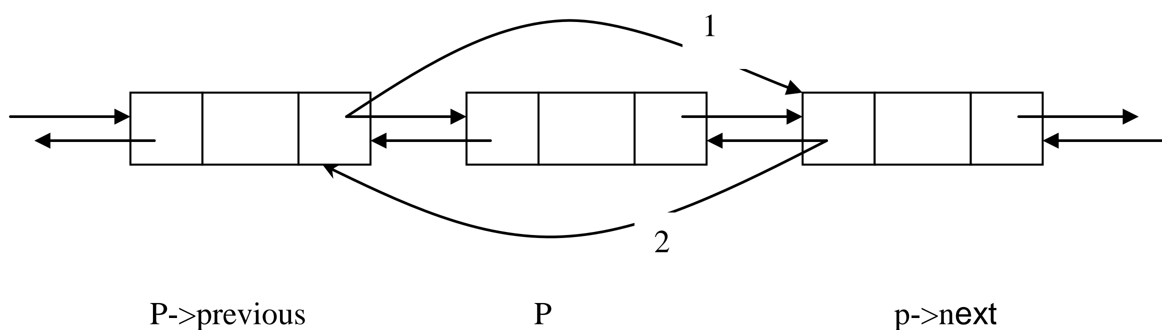
Rõ ràng, danh sách liên kết kép rỗng khi và chỉ khi chỉ điểm đầu danh sách không trỏ tới một ô xác định nào cả. Do đó ta sẽ kiểm tra DL = **NULL**.

```
int Empty (DoubleList DL)
{
return (DL==NULL);
}
```

Xóa một phần tử ra khỏi danh sách liên kết kép

Để xoá một phần tử tại vị trí p trong danh sách liên kết kép được trỏ bởi DL, ta phải chú ý đến các trường hợp sau:

- Danh sách rỗng, tức là DL=**NULL**: chương trình con dừng.
- Trường hợp danh sách khác rỗng, tức là DL!=**NULL**, ta phải phân biệt hai trường hợp Ô bị xoá không phải là ô được trỏ bởi DL, ta chỉ cần cập nhật lại các con trỏ để nối kết ô trước p với ô sau p, các thao tác cần thiết là (xem hình II.16):



P->previous P p->next

Hình II.16 Xóa phần tử tại vị trí p

Nếu (p->previous!=**NULL**) thì p->previous->next=p->next;

Nếu (p->next!=**NULL**) thì p->next->previous=p->previous;

Xoá ô đang được trỏ bởi DL, tức là p=DL: ngoài việc cập nhật lại các con trỏ để nối kết các ô trước và sau p ta còn phải cập nhật lại DL, ta có thể cho DL trỏ đến phần tử trước nó (DL = p->Previous) hoặc đến phần tử sau nó (DL = p->Next) tuỳ theo phần tử nào có mặt trong danh sách. Đặc biệt, nếu danh sách chỉ có một phần tử tức là p->Next=**NULL** và p->Previous=**NULL** thì DL=**NULL**.

void Delete_List (Position p, DoubleList *DL)

```
{
 if (*DL == NULL)
printf(”Danh sach rong”);
else
{
if (p==*DL)
(*DL)=(*DL)->Next;
//Xóa phần tử đầu tiên của danh sách nên phải thay dổi DL
else
p->Previous->Next=p->Next;
if (p->Next!=NULL)
p->Next->Previous=p->Previous;
free(p);
}
}
```

Thêm phần tử vào danh sách liên kết kép

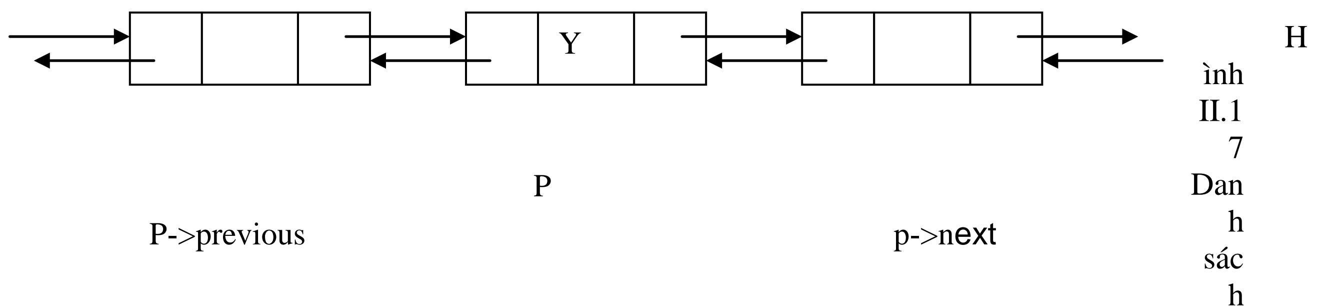
Để thêm một phần tử x vào vị trí p trong danh sách liên kết kép được trỏ bởi DL, ta cũng cần phân biệt mấy trường hợp sau:

Danh sách rỗng, tức là DL = **NULL**: trong trường hợp này ta không quan tâm đến giá trị của p. Để thêm một phần tử, ta chỉ cần cấp phát ô nhớ cho nó, gán giá trị x vào trường Element của ô nhớ này và cho hai con trỏ previous, next trỏ tới **NULL** còn DL trỏ vào ô nhớ này, các thao tác trên có thể viết như sau:

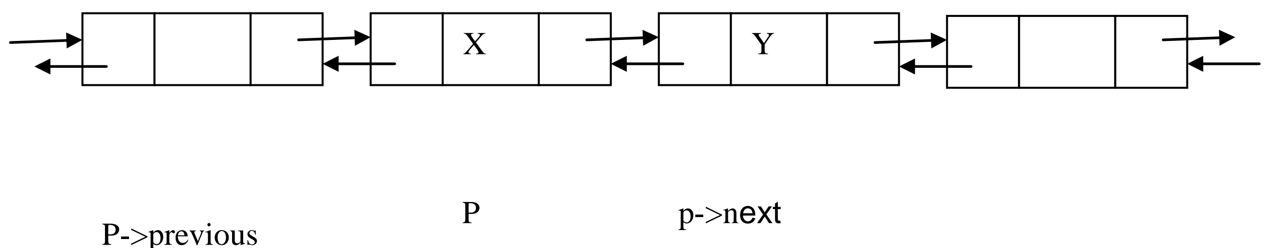
DL=(Node*)malloc(sizeof(Node));

DL->Element = x; DL->Previous=**NULL**; DL->Next =**NULL**;

Nếu DL!=**NULL**, sau khi thêm phần tử x vào vị trí p ta có kết quả như hình II.18



Hình II.17 Danh sách trước khi thêm phần tử X



Hình II.18: Danh sách sau khi thêm phần tử x vào tại vị trí p (phần tử tại vị trí p cũ trở thành phần tử "sau" của x)

Lưu ý: các kí hiệu p, p->Next, p->Previous trong hình II.18 để chỉ các ô trước khi thêm phần tử x, tức là nó chỉ các ô trong hình II.17.

Trong trường hợp p=DL, ta có thể cập nhật lại DL để DL trỏ tới ô mới thêm vào hoặc để nó trỏ đến ô tại vị trí p cu như nó đang trỏ cũng chỉ là sự lựa chọn trong chi tiết cài đặt.

void Insert_List (ElementType X,Position p, DoubleList *DL)

```
{
 if (*DL == NULL)
{
        (*DL)=(Node*)malloc(sizeof(Node));
        (*DL)->Element = X;
        (*DL)->Previous =NULL;
        (*DL)->Next =NULL;
}
else{
        Position temp;
        temp=(Node*)malloc(sizeof(Node));
        temp->Element=X;
        temp->Next=p;
        temp->Previous=p->Previous;
        if (p->Previous!=NULL)
                p->Previous->Next=temp;
        p- >Previous=temp;
}

}
```

2.2. Ngăn xếp (stack)

2.2.1. Khái niệm

Ngăn xếp (Stack) là một danh sách mà ta giới hạn việc thêm vào hoặc loại bỏ một phần tử chỉ thực hiện tại một đầu của danh sách, đầu này gọi là đỉnh (TOP) của ngăn xếp.

Ta có thể xem hình ảnh trực quan của ngăn xếp bằng một chồng đĩa đặt trên bàn. Muốn thêm vào chồng đó 1 đĩa ta để đĩa mới trên đỉnh chồng, muốn lấy các đĩa ra khỏi chồng ta cũng phải lấy đĩa trên trước. Như vậy ngăn xếp là một cấu trúc có tính chất “vào sau - ra trước” hay “vào trước – ra sau“ (**LIFO** (last in - first out) hay **FILO** (first in – last out)).

Các phép toán trên ngăn xếp

- **MAKENULL_STACK(S):** tạo một ngăn xếp rỗng.

- **TOP(S)** xem như một hàm trả về phần tử tại đỉnh ngăn xếp. Nếu ngăn xếp rỗng thì hàm không xác định. Lưu ý rằng ở dây ta dùng từ "hàm" để ngụ ý là TOP(S) có trả kết quả ra. Nó có thể không dồng nhất với khái niệm hàm trong ngôn ngữ lập trình như C chẳng hạn, vì có thể kiểu phần tử không thể là kiểu kết quả ra của hàm trong C.

- **POP(S)** chương trình con xoá một phần tử tại đỉnh ngăn xếp.

- **PUSH(x,S)** chương trình con thêm một phần tử x vào đầu ngăn xếp.

- **EMPTY_STACK(S)** kiểm tra ngăn xếp rỗng. Hàm cho kết quả 1 (true) nếu ngăn xếp rỗng và 0 (false) trong trường hợp ngược lại.

Như đã nói từ trước, khi thiết kế giải thuật ta có thể dùng các phép toán trừu tượng như là các "nguyên thủy" mà không cần phải định nghĩa lại hay giải thích thêm. Tuy nhiên, để giải thuật đó thành chương trình chạy được thì ta phải chọn một cấu trúc dữ liệu hợp lí để cài đặt các "nguyên thủy" này.

Ví dụ: Viết chương trình con Edit nhận một chuỗi kí tự từ bàn phím cho đến khi gặp kí tự @ thì kết thúc việc nhập và in kết quả theo thứ tự ngược lại.

```
void Edit(){
Stack S;
char c;
MakeNull_Stack(&S);
do{
// Lưu từng ký tự vào ngăn xếp
c=getche();
```

```
Push(c,&S);
}while (c!='@');
printf("\nChươi theo thư tu ngươc lai\n"); // In ngăn xep
while (!Empty_Stack(S))
{
printf("%c\n",Top(S));
Pop(&S);
}
}
```

Câu hỏi ôn tập:

Ta có thể truy xuất trực tiếp phần tử tại vị trí bất kỳ trong ngăn xếp được không?

2.2.2. Cài đặt ngăn xếp bởi mảng, DSLK

a. Cài đặt ngăn xếp bằng danh sách:

Do ngăn xếp là một danh sách đặc biệt nên ta có thể sử dụng kiểu dữ liệu trừu tượng danh sách để biểu diễn cách cài đặt nó. Như vậy, ta có thể khai báo ngăn xếp như sau:

```
typedef List Stack;
```

Khi chúng ta đã dùng danh sách để biểu diễn cho ngăn xếp thì ta nên sử dụng các phép toán trên danh sách để cài đặt các phép toán trên ngăn xếp. Sau dây là phần cài đặt ngăn xếp bằng danh sách.

Tạo ngăn xếp rỗng:

```
void MakeNull_Stack(Stack *S)
{
MakeNull_List(S);
}
```

Kiểm tra ngăn xếp rỗng:

```
int Empty_Stack(Stack S)
{
return Empty_List(S);
}
```

Thêm phần tử vào ngăn xếp

```
void Push(Elementtype X, Stack *S)
{
Insert_List (x, First (*S), &S);
}
```

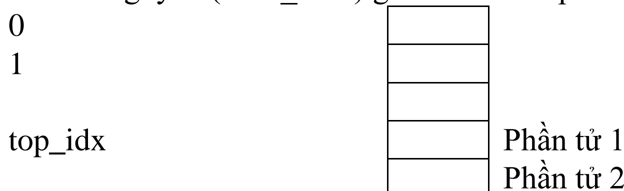
Xóa phần tử ra khỏi ngăn xếp

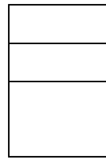
```
void Pop (Stack *S)
{
Delete_List (First (*S), &S);
}
```

Tuy nhiên để tăng tính hiệu quả của ngăn xếp ta có thể cài đặt ngăn xếp trực tiếp từ các cấu trúc dữ liệu như các phần sau.

b. Cài đặt bằng mảng

Dùng một mảng để lưu trữ liên tiếp các phần tử của ngăn xếp. Các phần tử đưa vào ngăn xếp bắt đầu từ vị trí có chỉ số cao nhất của mảng, xem hình sau. Ta còn phải dùng một biến số nguyên (TOP_IDX) giữ chỉ số của phần tử tại đỉnh ngăn xếp.





Maxlenth-1 Phần tử cuối cùng (phần tủ đầu tiên được them vào ngăn xếp)

Khai báo ngăn xếp

```
#define MaxLength ... //độ dài của mảng
typedef ... ElementType; //kiểu các phần tử trong ngăn xếp
typedef struct {
ElementType Elements[MaxLength]; //Lưu nội dung của các phần tử
int Top_idx; //giữ vị trí đỉnh ngăn xếp
} Stack;
```

Tạo ngăn xếp rỗng

Ngăn xếp rỗng là ngăn xếp không chứa bất kỳ một phần tử nào, do đó đỉnh của ngăn xếp không được phép chỉ đến bất kỳ vị trí nào trong mảng. Để tiện cho quá trình thêm và xóa phần tử ra khỏi ngăn xếp, khi tạo ngăn xếp rỗng ta cho đỉnh ngăn xếp nằm ở vị trí maxlength.

```
void MakeNull_Stack(Stack *S)
{
S->Top_idx=MaxLength;
}
```

Kiểm tra ngăn xếp rỗng

```
int Empty_Stack(Stack S)
{
return S.Top_idx==MaxLength;
}
```

Kiểm tra ngăn xếp đầy

```
int Full_Stack(Stack S)
{
return S.Top_idx==0;
}
```

Lấy nội dung phần tử tại đỉnh của ngăn xếp :

Hàm trả về nội dung phần tử tại đỉnh của ngăn xếp khi ngăn xếp không rỗng. Nếu ngăn xếp rỗng thì hàm hiển thị câu thông báo lỗi.

```
ElementType Top(Stack S)
{
if (!Empty_Stack(S))
return S.Elements[S.Top_idx];
else printf("Loi! Ngăn xep rong");
}
```

Chú ý Nếu ElementType không thể là kiểu kết quả trả về của một hàm thì ta có thể viết Hàm Top như sau:

```
void TOP(Stack S, Elementtype *X)
{
//Trong đó, x sẽ lưu trữ nội dung phần tử tại đỉnh của ngăn xếp
if (!Empty_Stack(S))
*X = S.Elements[S.Top_idx];
else printf("Loi: Ngăn xep rong ");
}
```

Chương trình con xóa phần tử ra khỏi ngăn xếp

Phần tử được xóa ra khỏi ngăn xếp là tại đỉnh của ngăn xếp. Do đó, khi xóa ta chỉ cần dịch chuyển đỉnh của ngăn xếp xuống 1 vị trí (top_idx tăng 1 đơn vị)

```
void Pop(Stack *S){ if (!Empty_Stack(*S))
S->Top_idx=S->Top_idx+1;
else printf("Loi! Ngăn xep rong!");
}
```

Chương trình con thêm phần tử vào ngăn xếp :

Khi thêm phần tử có nội dung x (kiểu ElementType) vào ngăn xếp S (kiểu Stack), trước tiên ta phải kiểm tra xem ngăn xếp có còn chỗ trống để lưu trữ phần tử mới không. Nếu không còn chỗ trống (ngăn xếp đầy) thì báo lỗi; Ngược lại, dịch chuyển Top_idx lên trên 1 vị trí và đặt x vào tại vị trí đỉnh mới.

```
void Push(ElementType X, Stack *S)
{
if (Full_Stack(*S))
printf("Loi! Ngăn xep day!");
else{
S->Top_idx=S->Top_idx-1;
S->Elements[S->Top_idx]=X;
}
}
```

Bài tập: Cài đặt ngăn xếp sử dụng con trỏ.

2.2.3. Ứng dụng

Ứng dụng ngăn xếp để loại bỏ đệ qui của chương trình

Nếu một chương trình con đệ qui P(x) được gọi từ chương trình chính ta nói chương trình con được thực hiện ở mức 1. Chương trình con này gọi chính nó, ta nói nó đi sâu vào mức 2... cho đến một mức k. Rõ ràng mức k phải thực hiện xong thì mức k-1 mới được thực hiện tiếp tục, hay ta còn nói là chương trình con quay về mức k-1.

Trong khi một chương trình con từ mức i di vào mức i+1 thì các biến cục bộ của mức i và địa chỉ của mã lệnh còn đang dở phải được lưu trữ, địa chỉ này gọi là địa chỉ trở về. Khi từ mức i+1 quay về mức i các giá trị đó được sử dụng. Như vậy những biến cục bộ và địa chỉ lưu sau được dùng trước. Tính chất này gợi ý cho ta dùng một ngăn xếp để lưu giữ các giá trị cần thiết của mỗi lần gọi tới chương trình con. Mỗi khi lùi về một mức thì các giá trị này được lấy ra để tiếp tục thực hiện mức này. Ta có thể tóm tắt quá trình như sau:

Bước 1: Lưu các biến cục bộ và địa chỉ trở về.

Bước 2: Nếu thoả điều kiện ngừng đệ qui thì chuyển sang bước 3. Nếu không thì tính toán từng phần và quay lại bước 1 (đệ qui tiếp).

Bước 3: Khôi phục lại các biến cục bộ và địa chỉ trở về.

Ví dụ sau dây minh hoạ việc dùng ngăn xếp để loại bỏ chương trình đệ qui của bài toán "tháp Hà Nội" (tower of Hanoi).

Bài toán "tháp Hà Nội" được phát biểu như sau:

Có ba cọc A,B,C. Khởi đầu cọc A có một số dĩa xếp theo thứ tự nhỏ dần lên trên đỉnh. Bài toán đặt ra là phải chuyển toàn bộ chồng đĩa từ A sang B. Mỗi lần thực hiện chuyển một đĩa từ một cọc sang một cọc khác và không được đặt đĩa lớn nằm trên đĩa nhỏ

Chương trình con đệ qui để giải bài toán tháp Hà Nội như sau:

```
void Move(int n, int A, int B, int C)
//n: số đĩa, A,B,C: cọc nguồn , đích và trung gian
{
if (n==1)
printf("Chuyen 1 đĩa tu %c sang %c\n",Temp.A,Temp.B);
else
{
Move(n-1, A,C,B);
//chuyển n-1 đĩa từ cọc nguồn sang cọc trung gian
```

Move(1,A,B,C);
//chuyển 1 đĩa từ cọc nguồn sang cọc đích Move(n-1,C,B,A);
//chuyển n-1 đĩa từ cọc trung gian sang cọc đích
}
}
Quá trình thực hiện chương trình con được minh hoạ với ba đĩa (n=3) như sau:

		Move(1,A,B,C)
	Move(2,A,C,B)	Move(1,A,C,B)
		Move(1,B,C,A)
Move(3,A,B,C)	Move(1,A,B,C)	
		Move(1,C,A,B)
	Move(2,C,B,A)	Move(1,C,B,A)
		Move(1,A,B,C)
Mức 1	mức 2	mức 3

Để khử đệ qui ta phải nắm nguyên tắc sau dây:

Mỗi khi chương trình con đệ qui được gọi, ứng với việc di từ mức i vào mức i+1, ta phải lưu trữ các biến cục bộ của chương trình con ở bước i vào ngăn xếp. Ta cũng phải lưu "địa chỉ mã lệnh" chưa được thi hành của chương trình con ở mức i. Tuy nhiên khi lập trình bằng ngôn ngữ cấp cao thì dây không phải là địa chỉ ô nhớ chứa mã lệnh của máy mà ta sẽ tổ chức sao cho khi mức i+1 hoàn thành thì lệnh tiếp theo sẽ được thực hiện là lệnh đầu tiên chưa được thi hành trong mức i.

Tập hợp các biến cục bộ của mỗi lần gọi chương trình con xem như là một mẩu tin hoạt động (activation record).

Mỗi lần thực hiện chương trình con tại mức i thì phải xoá mẩu tin lưu các biến cục bộ ở mức này trong ngăn xếp.

Như vậy nếu ta tổ chức ngăn xếp hợp lí thì các giá trị trong ngăn xếp chẳng những lưu trữ được các biến cục bộ cho mỗi lần gọi đệ qui, mà còn "điều khiển được thứ tự trở về" của các chương trình con. Ý tưởng này thể hiện trong cài đặt khử đệ qui cho bài toán tháp Hà Nội là: mẩu tin lưu trữ các biến cục bộ của chương trình con thực hiện sau thì được đưa vào ngăn xếp trước để nó được lấy ra dùng sau.

```
//Kiểu cấu trúc lưu trữ biến cục bộ
typedef struct{
int N;
int A, B, C;
} ElementType;

// Chương trình con MOVE không đệ qui

void Move(ElementType X)
{
ElementType Temp, Temp1;
Stack S;
MakeNull_Stack(&S); Push(X,&S);
do
{
Temp=Top(S); //Lay phan tu dau
Pop(&S); //Xoa phan tu dau
if (Temp.N==1)
printf("Chuyen 1 đĩa tu %c sang %c\n",Temp.A,Temp.B);
else
{
// Lưu cho loi goi Move(n-1,C,B,A)
Temp1.N=Temp.N-1;
Temp1.A=Temp.C;
Temp1.B=Temp.B;
Temp1.C=Temp.A;
Push(Temp1,&S);
//Lưu cho loi goi Move(1,A,B,C)
```

```
Temp1.N=1;
Temp1.A=Temp.A;
Temp1.B=Temp.B;
Temp1.C=Temp.C;
Push(Temp1,&S);
//Lưu cho loi goi Move(n-1,A,C,B)
Temp1.N=Temp.N-1;
Temp1.A=Temp.A;
Temp1.B=Temp.C;
Temp1.C=Temp.B;
Push(Temp1,&S);
}
} while (!Empty_Stack(S));
}
```

Minh họa cho lời gọi Move(x) với 3 đĩa, tức là x.N=3. Ngăn xếp khởi đầu:

3,A,B,C

Ngăn xếp sau lần lặp thứ nhất:

2,A,C,B
1,A,B,C
2,C,B,A

Ngăn xếp sau lần lặp thứ hai

1,A,B,C
1,A,C,B
1,B,C,A
1,A,B,C
2,C,B,A

Các lần lặp 3,4,5,6 thì chương trình con xử lý trường hợp chuyển 1 đĩa (ứng với trường hợp không gọi đệ qui), vì vậy không có mẩu tin nào được thêm vào ngăn xếp. Mỗi lần xử lý, phần tử đầu ngăn xếp bị xoá. Ta sẽ có ngăn xếp như sau.

2,C,B,A

Tiếp tục lặp bước 7 ta có ngăn xếp như sau:

1,C,A,B
1,C,B,A
1,A,B,C

Các lần lặp tiếp tục chỉ xử lý việc chuyển 1 đĩa (ứng với trường hợp không gọi đệ qui). Chương trình con in ra các phép chuyển và dẫn đến ngăn xếp rỗng.

2.3. Hàng đợi (queue)

2.3.1. Khái niệm

Hàng đợi, hay ngắn gọn là hàng (queue) cũng là một danh sách đặc biệt mà phép thêm vào chỉ thực hiện tại một đầu của danh sách, gọi là cuối hàng (REAR), còn phép loại bỏ thì thực hiện ở đầu kia của danh sách, gọi là đầu hàng (FRONT).

Xếp hàng mua vé xem phim là một hình ảnh trực quan của khái niệm trên, người mới đến thêm vào cuối hàng còn người ở đầu hàng mua vé và ra khỏi hàng, vì vậy hàng còn được gọi là cấu trúc **FIFO** (first in - first out) hay "vào trước - ra trước".

Các phép toán co bản trên hàng

- **MAKENULL_QUEUE(Q)** khởi tạo một hàng rỗng.
- **FRONT(Q)** hàm trả về phần tử đầu tiên của hàng Q.
- **ENQUEUE(x,Q)** thêm phần tử x vào cuối hàng Q.
- **DEQUEUE(Q)** xoá phần tử tại đầu của hàng Q.
- **EMPTY_QUEUE(Q)** hàm kiểm tra hàng rỗng.

- FULL_QUEUE(Q) kiểm tra hàng đầy.

2.3.2. Cài đặt hàng đợi bởi mảng, DSLK

Như đã trình bày trong phần ngăn xếp, ta hoàn toàn có thể dùng danh sách để biểu diễn cho một hàng và dùng các phép toán đã được cài đặt của danh sách để cài đặt các phép toán trên hàng. Tuy nhiên làm như vậy có khi sẽ không hiệu quả, chẳng hạn dùng danh sách cài đặt bằng mảng ta thấy lời gọi INSERT_LIST(x,ENDLIST(Q),Q) tốn một hằng thời gian trong khi lời gọi DELETE_LIST(FIRST(Q),Q) để xoá phần tử đầu hàng (phần tử ở vị trí 0 của mảng) ta phải tốn thời gian tỉ lệ với số các phần tử trong hàng để thực hiện việc dời toàn bộ hàng lên một vị trí. Để cài đặt hiệu quả hơn ta phải có một suy nghi khác dựa trên tính chất dặc biệt của phép thêm và loại bỏ một phần tử trong hàng.

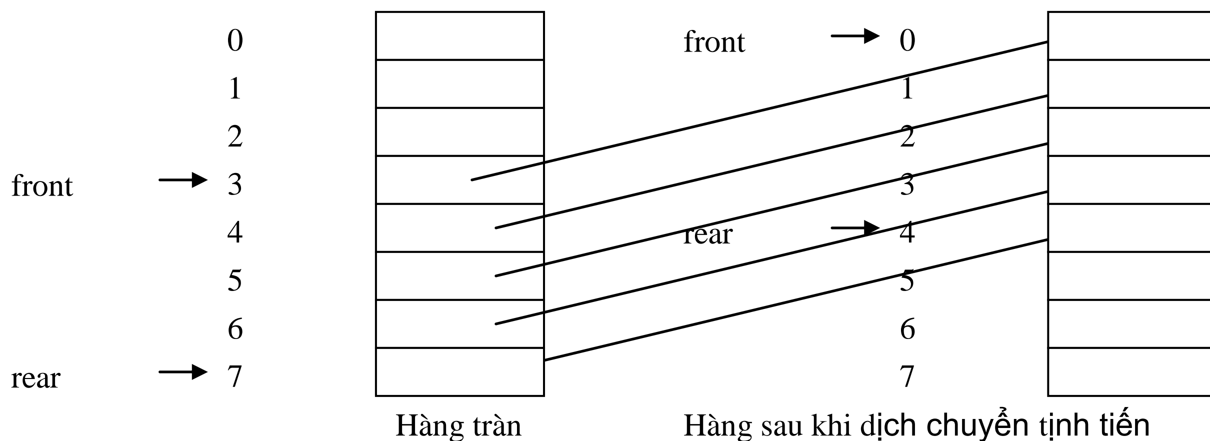
a. Cài đặt hàng bằng mảng

Ta dùng một mảng để chứa các phần tử của hàng, khởi đầu phần tử đầu tiên của hàng được đưa vào vị trí thứ 1 của mảng, phần tử thứ 2 vào vị trí thứ 2 của mảng... Giả sử hàng có n phần tử, ta có front=0 và rear=n-1. Khi xoá một phần tử front tăng lên 1, khi thêm một phần tử rear tăng lên 1. Như vậy hàng có khuynh hướng di xuống, đến một lúc nào đó ta không thể thêm vào hàng được nữa (rear=maxlength-1) dù mảng còn nhiều chỗ trống (các vị trí trước front) trường hợp này ta gọi là *hàng bị tràn* (xem hình sau).Trong trường hợp toàn bộ mảng đã chứa các phần tử của hàng ta gọi là *hàng bị đầy*.

Cách khắc phục hàng bị tràn

- Dời toàn bộ hàng lên front -1 vị trí, cách này gọi là di chuyển tịnh tiến. Trong trường hợp này ta luôn có front<=rear.

- Xem mảng như là một vòng tròn nghĩa là khi hàng bị tràn nhưng chưa đầy ta thêm phần tử mới vào vị trí 0 của mảng, thêm một phần tử mới nữa thì thêm vào vị trí 1 (nếu có thể)...Rõ ràng cách làm này front có thể lớn hơn rear. Cách khắc phục này gọi là dùng mảng xoay vòng (xem hình II.12).



Hàng tràn Hàng sau khi dịch chuyển tịnh tiến

Hình vẽ minh họa việc di chuyển tịnh tiến các phần tử khi hàng bị tràn

Cài đặt hàng bằng mảng theo phương pháp tịnh tiến

Để quản lí một hàng ta chỉ cần quản lí đầu hàng và cuối hàng. Có thể dùng 2 biến số nguyên chỉ vị trí đầu hàng và cuối hàng

Các khai báo cần thiết

```
#define MaxLength ... //chiều dài tối da của mảng
typedef ... ElementType;
//Kiểu dữ liệu của các phần tử trong hàng
typedef struct
{
ElementType Elements[MaxLength]; //Lưu trữ nội dung các phần tử
int Front, Rear; //chỉ số đầu và duôi hàng
} Queue;
```

Tạo hàng rỗng

Lúc này front và rear không trỏ đến vị trí hợp lệ nào trong mảng vậy ta có thể cho front và rear dều bằng -1.

```
void MakeNull_Queue(Queue *Q)
{
Q->Front=-1;
Q->Rear=-1;
}
```

Kiểm tra hàng rỗng

Trong quá trình làm việc ta có thể thêm và xóa các phần tử trong hàng. Rõ ràng, nếu ta có đưa vào hàng một phần tử nào đó thì front>-1. Khi xoá một phần tử ta tăng front lên 1. Hàng rỗng nếu front>rear. Hơn nữa khi mới khởi tạo hàng, tức là front = -1, thì hàng cũng rỗng. Tuy nhiên để phép kiểm tra hàng rỗng đơn giản, ta sẽ làm một phép kiểm tra khi xoá một phần tử của hàng, nếu phần tử bị xoá là phần tử duy nhất trong hàng thì ta đặt lại front=-1. Vậy hàng rỗng khi và chỉ khi front =-1.

```
int Empty_Queue(Queue Q)
{
return Q.Front==-1;
}
```

Kiểm tra đầy

Hàng đầy nếu số phần tử hiện có trong hàng bằng số phần tử trong
mảng.

```
int Full_Queue(Queue Q){
return (Q.Rear-Q.Front+1)==MaxLength;
}
```

Xóa phần tử ra khỏi hàng

Khi xóa một phần tử đầu hàng ta chỉ cần cho front tăng lên 1. Nếu front > rear thì hàng thực chất là hàng đã rỗng, nên ta sẽ khởi tạo lại hàng rỗng (tức là đặt lại giá trị front = rear =-1).

```
void DeQueue(Queue *Q)
{ if (!Empty_Queue(*Q))
{
Q->Front=Q->Front+1;
if (Q->Front>Q->Rear)
MakeNull_Queue(Q); //Dat lai hàng rong
}
else printf("Loi: Hàng rong!");
}
```

Thêm phần tử vào hàng

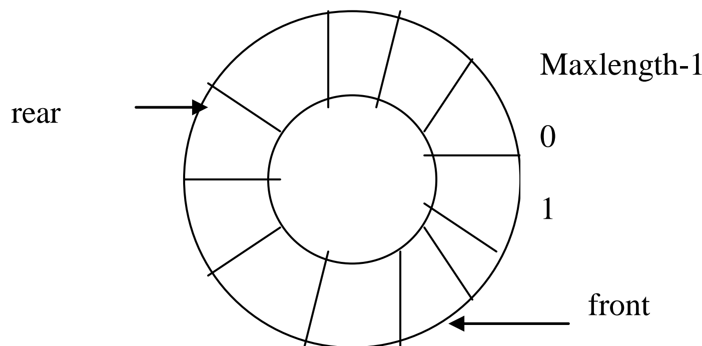
Một phần tử khi được thêm vào hàng sẽ nằm kế vị trí Rear cũ của hàng. Khi thêm một phần tử vào hàng ta phải xét các trường hợp sau:

Nếu hàng đầy thì báo lỗi không thêm được nữa.

Nếu hàng chưa đầy ta phải xét xem hàng có bị tràn không. Nếu hàng bị tràn ta di chuyển tịnh tiến rồi mới nối thêm phần tử mới vào duôi hàng (rear tăng lên 1). Đặc biệt nếu thêm vào hàng rỗng thì ta cho front=0 để front trỏ dúng phần tử đầu tiên của hàng.

```
void EnQueue(ElementType X,Queue *Q)
{
if (!Full_Queue(*Q))
{
if (Empty_Queue(*Q))
Q->Front=0;
if (Q->Rear==MaxLength-1)
{
//Di chuyen tinh tien ra trước Front -1 vi tri
for(int i=Q->Front;i<=Q->Rear;i++)
Q->Elements[i-Q->Front]=Q->Elements[i]; //Xac dinh vi tri Rear moi
Q->Rear=MaxLength - Q->Front-1;
Q->Front=0;
}
//Tăng Rear de lưu noi dung moi
Q->Rear=Q->Rear+1;
Q->Element[Q->Rear]=X;
}
else printf("Loi: Hàng day!");
}
```

b. Cài đặt hàng với mảng xoay vòng



Hình II.12 Cài đặt hàng bằng mảng xoay vòng

Với phương pháp này, khi hàng bị tràn, tức là rear=maxlength-1, nhưng chưa đầy, tức là front>0, thì ta thêm phần tử mới vào vị trí 0 của mảng và cứ tiếp tục như vậy vì từ 0 đến front-1 là các vị trí trống. Vì ta sử dụng mảng một cách xoay vòng như vậy nên phương pháp này gọi là phương pháp dùng mảng xoay vòng.

Các phần khai báo cấu trúc dữ liệu, tạo hàng rỗng, kiểm tra hàng rỗng giống như phương pháp di chuyển tịnh tiến.

Khai báo cần thiết

```
#define MaxLength ... //chiều dài tối đa của mảng
typedef ... ElementType;
```

```
//Kiểu dữ liệu của các phần tử trong hàng
typedef struct {
ElementType Elements[MaxLength];
//Lưu trữ nội dung các phần tử
int Front, Rear; //chỉ số đầu và đuôi hàng
} Queue;
```

Tạo hàng rỗng

Lúc này front và rear không trỏ đến vị trí hợp lệ nào trong mảng vậy ta có thể cho front và rear đều bằng -1.

```
void MakeNull_Queue(Queue *Q){
 Q->Front=-1;
Q->Rear=-1;
}
```

Kiểm tra hàng rỗng

```
int Empty_Queue(Queue Q){
return Q.Front==-1;
}
```

Kiểm tra hàng đầy

Hàng đầy nếu toàn bộ các ô trong mảng đang chứa các phần tử của hàng. Với phương pháp này thì front có thể lớn hơn rear. Ta có hai trường hợp hàng đầy như sau:

- Trường hợp Q.Rear=Maxlength-1 và Q.Front =0
- Trường hợp Q.Front =Q.Rear+1.

Để đơn giản ta có thể gom cả hai trường hợp trên lại theo một công thức như sau:

(Q.rear-Q.front +1) mod Maxlength =0

```
int Full_Queue(Queue Q)
{
return (Q.Rear-Q.Front+1) % MaxLength==0;
}
```

Xóa một phần tử ra khỏi hàng

Khi xóa một phần tử ra khỏi hàng, ta xóa tại vị trí đầu hàng và có thể xảy ra các trường hợp sau:

Nếu hàng rỗng thì báo lỗi không xóa;

Ngược lại, nếu hàng chỉ còn 1 phần tử thì khởi tạo lại hàng rỗng;

Ngược lại, thay dổi giá trị của Q.Front.

(Nếu Q.front != Maxlength-1 thì đặt lại Q.front = q.Front +1; Ngược lại Q.front=0)

```
void DeQueue(Queue *Q){ if (!Empty_Queue(*Q)){
//Nếu hàng chỉ chứa một phần tử thì khởi tạo hàng lại
if (Q->Front==Q->Rear)
MakeNull_Queue(Q);
else
Q->Front=(Q->Front+1) % MaxLength; //tăng Front lên 1 đơn vị
}
else printf("Loi: Hàng rong!");
}
```

Thêm một phần tử vào hàng

Khi thêm một phần tử vào hàng thì có thể xảy ra các trường hợp sau:

- Trường hợp hàng đầy thì báo lỗi và không thêm;
- Ngược lại, thay dổi giá trị của Q.rear (Nếu Q.rear =maxlength-1 thì đặt lại Q.rear=0; Ngược lại Q.rear =Q.rear+1) và đặt nội dung vào vị trí Q.rear mới.

```
void EnQueue(ElementType X,Queue *Q)
```

```
{ if (!Full_Queue(*Q))
{
if (Empty_Queue(*Q))
Q->Front=0;
Q->Rear=(Q->Rear+1) % MaxLength;
Q->Elements[Q->Rear]=X;
}
else printf("Loi: Hàng day!");
}
```

Câu hỏi ôn tập:

Cài đặt hàng bằng mảng vòng có ưu điểm gì so với bằng mảng theo phương pháp tịnh tiến? Trong ngôn ngữ lập trình có kiểu dữ liệu mảng vòng không?

c. Cài đặt hàng bằng danh sách liên kết (cài đặt bằng con trỏ)

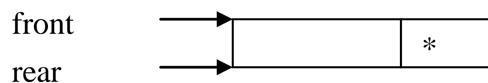
Cách tự nhiên nhất là dùng hai con trỏ front và rear để trỏ tới phần tử đầu và cuối hàng. Hàng được cài đặt như một danh sách liên kết có Header là một ô thực sự, xem hình II.13.

Khai báo cần thiết

```
typedef ... ElementType; //kiểu phần tử của hàng
typedef struct Node
{
ElementType Element;
Node* Next; //Con trỏ chỉ ô kế tiếp
};
typedef Node* Position;
typedef struct
{
Position Front, Rear;
//là hai trường chỉ đến đầu và cuối của hàng
} Queue;
```

Khởi tạo hàng rỗng

Khi hàng rỗng Front va Rear cùng trỏ về 1 vị trí đó chính là ô header



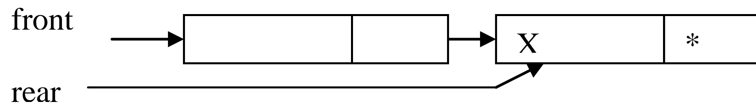
Hình II.13: Khởi tạo hàng rỗng

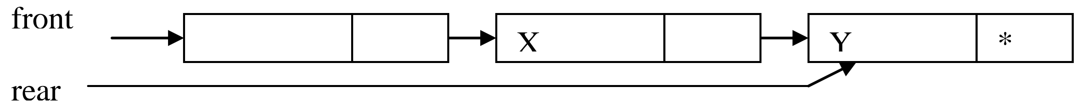
```
void MakeNullQueue(Queue *Q)
{
Position Header;
Header=(Node*)malloc(sizeof(Node)); //Cấp phát Header
Header->Next=NULL;
Q->Front=Header;
Q->Rear=Header;
}
```

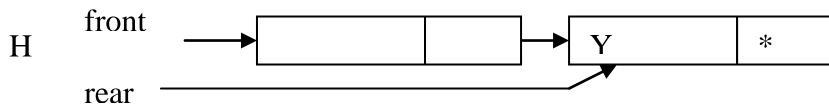
Kiểm tra hàng rỗng

Hàng rỗng nếu Front và Rear chỉ cùng một vị trí là ô Header.

```
int EmptyQueue(Queue Q)
{
return (Q.Front==Q.Rear);
}
```







Hình II.14 Hàng sau khi thêm và xóa phần tử

Thêm một phần tử vào hàng

Thêm một phần tử vào hàng ta thêm vào sau Rear (Rear->next), rồi cho Rear trỏ đến phần tử mới này, xem hình II.14. Trường next của ô mới này trỏ tới **NULL**.

```
void EnQueue(ElementType X, Queue *Q){
Q->Rear->Next=(Node*)malloc(sizeof(Node));
Q->Rear=Q->Rear->Next;
//Dat gia tri vao cho Rear
Q->Rear->Element=X;
Q->Rear->Next=NULL;
}
```

Xóa một phần tử ra khỏi hàng

Thực chất là xoá phần tử nằm ở vị trí đầu hàng do đó ta chỉ cần cho front trỏ tới vị trí kế tiếp của nó trong hàng.

```
void DeQueue(Queue *Q){ if (!Empty_Queue(Q))
{
Position T;
T=Q->Front;
Q->Front=Q->Front->Next;
free(T);
}
else printf(”Loi : Hàng rong”);
}
```

2.3.3. Ứng dụng

Hàng đợi là một cấu trúc dữ liệu được dùng khá phổ biến trong thiết kế giải thuật. Bất kỳ nơi nào ta cần quản lí dữ liệu, quá trình... theo kiểu vào trước-ra trước đều có thể ứng dụng hàng đợi.

Ví dụ rất dễ thấy là quản lí in trên mạng, nhiều máy tính yêu cầu in đồng thời và ngay cả một máy tính cũng yêu cầu in nhiều lần. Nói chung có nhiều yêu cầu in dữ liệu, nhưng máy in không thể đáp ứng tức thời tất cả các yêu cầu đó nên chương trình quản lí in sẽ thiết lập một hàng đợi để quản lí các yêu cầu. Yêu cầu nào mà chương trình quản lí in nhận trước nó sẽ giải quyết trước.

Một ví dụ khác là duyệt cây theo mức được trình bày chi tiết trong chương sau. Các giải thuật duyệt theo chiều rộng một đồ thị có hướng hoặc vô hướng cũng dùng hàng đợi để quản lí các nút đồ thị. Các giải thuật đổi biểu thức trung tố thành hậu tố, tiền tố.

2.4. Bài tập áp dụng

1. Viết khai báo và các chương trình con cài đặt danh sách bằng mảng. Dùng các chương trình con này để viết:

a. Chương trình con nhận một dãy các số nguyên nhập từ bàn phím, lưu trữ nó trong danh sách theo thứ tự nhập vào.

b. Chương trình con nhận một dãy các số nguyên nhập từ bàn phím, lưu trữ nó trong danh sách theo thứ tự ngược với thứ tự nhập vào.

c. Viết chương trình con in ra màn hình các phần tử trong danh sách theo thứ tự của nó trong danh sách.

2. Tương tự như bài tập 1. nhưng cài đặt bằng con trỏ.

3. Viết chương trình con sắp xếp một danh sách chứa các số nguyên, trong các trường hợp:

a. Danh sách được cài đặt bằng mảng (danh sách đặc).

b. Danh sách được cài đặt bằng con trỏ (danh sách liên kết).

4. Viết chương trình con thêm một phần tử trong danh sách đã có thứ tự sao cho ta vẫn có một danh sách có thứ tự bằng cách vận dụng các phép toán co bản trên danh sách

5. Viết chương trình con tìm kiếm và xóa một phần tử trong danh sách có thứ tự.

6. Viết chương trình con nhận vào từ bàn phím một dãy số nguyên, lưu trữ nó trong một danh sách có thứ tự không giảm, theo cách sau: với mỗi phần tử được nhập vào chương trình con phải tìm vị trí thích hợp để xen nó vào danh sách cho dúng thứ tự. Viết chương trình con trên cho trường hợp danh sách được cài đặt bằng mảng và cài đặt bằng con trỏ và trong trường hợp tổng quát (dùng các phép toán cơ bản trên danh sách)

7. Viết chương trình con loại bỏ các phần tử trùng nhau (giữ lại duy nhất 1 phần tử) trong một danh sách có thứ tự không giảm, trong hai trường hợp: cài đặt bằng mảng và cài đặt bằng con trỏ.

8. Viết chương trình con nhận vào từ bàn phím một dãy số nguyên, lưu trữ nó trong một danh sách có thứ tự tang không có hai phần tử trùng nhau, theo cách sau: với mỗi phần tử được nhập vào chương trình con phải tìm kiếm xem nó có trong danh sách chưa, nếu chưa có thì xen nó vào danh sách cho đúng thứ tự. Viết chương trình con trên cho trường hợp danh sách được cài đặt bằng mảng và cài đặt bằng con trỏ.

9. Viết chương trình con trộn hai danh sách liên kết chứa các số nguyên theo thứ tự tăng để được một danh sách cũng có thứ tự tăng.

10. Viết chương trình con xoá khỏi danh sách lưu trữ các số nguyên các phần tử là số nguyên lẻ, cũng trong hai trường hợp: cài đặt bằng mảng và bằng con trỏ.

11. Viết chương trình con tách một danh sách chứa các số nguyên thành hai danh sách: một danh sách gồm các số chẵn còn cái kia chứa các số lẻ.

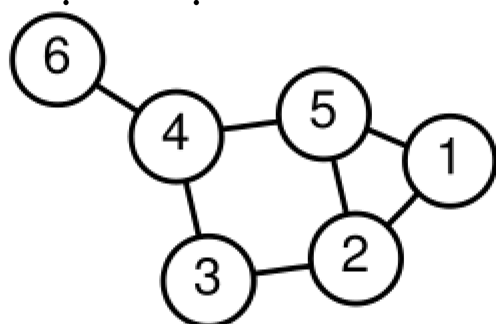
CHƯƠNG 3. CÂY (TREE).

1. Định nghĩa

1.1. Đồ thị (Graph)

Trước khi xem xét khái niệm thế nào là một cây (tree) chúng ta nhắc lại khái niệm đồ thị (graph) đã được học trong học phần Toán rời rạc: Đồ thị G bao gồm hai thành phần chính: tập các đỉnh V (Vertices) và tập các cung E (hay cạnh Edges), thường viết ở dạng G = <V, E>. Trong đó tập các đỉnh V là tập các đối tượng cùng loại, độc lập, chẳng hạn như các điểm trên mặt phẳng tọa độ, hoặc tập các thành phố, tập các trạng thái của một trò chơi, một đối tượng thực như con người, … tất cả đều có thể là các đỉnh của một đồ thị nào đó. Tập các cung E là tập các mối quan hệ hai ngôi giữa các đỉnh của đồ thị, đối với đỉnh là các điểm thì đây có thể là quan hệ về khoảng cách, tập đỉnh là các thành phố thì đây có thể là quan hệ về đường đi (có tồn tại đường đi trực tiếp nào giữa các thành phố hay không), hoặc nếu đỉnh là các trạng thái của một trò chơi thì cạnh có thể là cách biến đổi (transform) để đi từ trạng thái này sang một trạng thái khác, quá trình chơi chính là biến đổi từ trạng thái ban đầu tới trạng thái đích (có nghĩa là đi tìm một đường đi).

Ví dụ về đồ thị:



Hình 5.1. Đồ thị có 6 đỉnh và 7 cạnh, tham khảo từ wikipedia.

Có rất nhiều vấn đề liên quan tới đồ thị, ở phần này chúng ta chỉ nhắc lại một số khái niệm liên quan.

Một đồ thị được gọi là đơn đồ thị (simple graph) nếu như không có đường đi giữa hai đỉnh bất kỳ của đồ thị bị lặp lại, ngược lại nếu như có đường đi nào đó bị lặp lại hoặc tồn tại khuyên (self-loop), một dạng cung đi từ 1 đỉnh đến chính đỉnh đó, thì đồ thị được gọi là đa đồ thị (multigraph).

Giữa hai đỉ nh u, v trong đồ thị có đường đi trực tiếp thì u, v được gọi là liền kề với nhau, cạnh (u, v) được gọi là liên thuộc với hai đỉnh u, v.

Đồ thì được gọi là đồ thị có hướng (directed graph) nếu như các đường đi giữa hai đỉnh bất kỳ trong đồ thị phân biệt hướng với nhau, khi đó các quan hệ giữa các đỉnh được gọi chính xác là các cung, ngược lại đồ nếu không phân biệt hướng giữa các đỉnh trong các cạnh nối giữa hai đỉnh thì đồ thị được gọi là đồ thị vô hướng (undirected graph), khi đó ta nói tập E là tập các cạnh của đồ thị.

Các cung hay các cạnh của đồ thị có thể được gán các giá trị gọi là các trọng số (weight), một đồ thị có thể là đồ thị có trọng số hoặc không có trọng số. Ví dụ như đối với đồ thị mà các đỉnh là các thành phố ta có thể gán trọng số của các cung là độ dài đường đi nối giữa các thành phố hoặc chi phí đi trên con đường đó …

Một đường đi (path) trong đồ thị là một dãy các đỉnh $v_1, v_2, …, v_k$, trong đó các đỉnh v_i, v_{i+1} là liền kề với nhau. Đường đi có đỉnh đầu trùng với đỉnh cuối được gọi là chu trình (cycle).

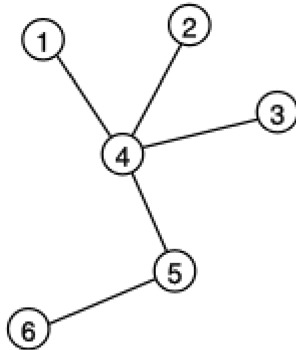
Giữa hai đỉnh của đồ thị có thể có các đường đi trực tiếp nếu chúng liền kề với nhau, hoặc nếu có một đường đi giữa chúng (gián tiếp) thì hai đỉnh đó được gọi là liên thông (connected) với nhau. Một đồ thị được gọi là liên thông nếu như hai đỉnh bất kỳ của nó đều

liên thông với nhau. Nếu đồ thị không liên thông thì luôn có thể chia nó thành các thành phần liên thông nhỏ hơn.

1.2. Cây (tree)

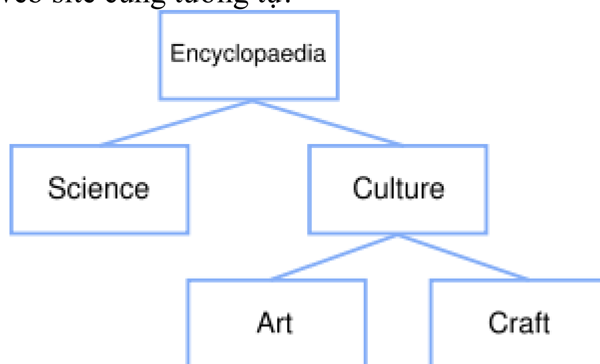
Có nhiều cách định nghĩa cây khác nhau nhưng ở đây chúng ta sẽ định nghĩa khái niệm cây theo lý thuyết đồ thị (graph theory).

Cây là một đồ thị vô hướng, không có trọng số, liên thông và không có chu trình. Ví dụ hình vẽ sau là một cây:



Hình 5.2. Cây, tham khảo từ wikipedia

Cấu trúc cây là một cấu trúc được sử dụng rất rộng rã i trong cuộc sống hàng ngày và trên máy tính, chẳng hạn cấu trúc tổ chức của một công ty là một cây phân cấp, cấu trúc của một web site cũng tương tự:



Hình 5.3. Cấu trúc web site wikipedia, tham khảo từ wikipedia.

Cấu trúc tổ chức thư mục của hệ điều hành là một cây …

Trong cây luôn có một nút đặc biệt gọi là gốc của cây (root), các đỉnh trong cây được gọi là các nút (nodes). Từ gốc của cây đi xuống tất cả các đỉnh liền kề với nó, các đỉnh này gọi là con của gốc, đến lượt các con của gốc lại có các nút con (child nodes) khác, như vậy quan hệ giữa hai nút liền kề nhau trong cây là quan hệ cha con, một nút là cha (parent), một nút là con (child), nút cha của cha của một nút được gọi là tổ tiên (ancestor) của nút đó.

Các nút trong cây được phân biệt làm nhiều loại: các nút có ít nhất 1 nút con được gọi là các nút trong (internal nodes hay inner nodes), các nút không có nút con được gọi là các nút lá (leaf nodes). Các nút lá không có các nút con nhưng để thuận tiện trong quá trình cài đặt người ta vẫn coi các nút lá có hai nút con giả, rỗng (NULL) đóng vai trò lính canh, gọi là các nút ngoài (external nodes).

Các nút trong cây được phân chia thành các tầng (level), nút gốc thuộc tầng 0 (level 0), sau đó các tầng tiếp theo sẽ được tăng lên 1 đơn vị so với tầng phía trên nó cho đến tầng cuối cùng. Độ cao (height) của cây được tính bằng số tầng của cây, độ cao của cây sẽ quyết định độ phức tạp (số thao tác) khi thực hiện các thao tác trên cây.

Mỗi nút trong của cây tổng quát có thể có nhiều nút con, tuy nhiên các nghiên cứu của ngành khoa h ọc máy tính đã cho thấy cấu trúc cây quan trọng nhất cần nghiên cứu chính là các cây nhị phân (binary tree), là các cây là mỗi nút chỉ có nhiều nhất hai nút con. Một cây tổng quát luôn có thể phân chia thành các cây nhị phân.

Các nút con của một nút trong cây nhị phân được gọi là nút con trái (**left child**) và nút con phải (**right child**).

Trong chương này chúng ta sẽ nghiên cứu một số loại cây nhị phân cơ bản và được ứng dụng rộng rãi nhất , đó là cây tìm kiếm nhị phân BST (**Binary Search Tree**), cây biểu thức (**expression tree** hay **syntax tree**) và cây cân bằng (**balanced tree**) AVL.

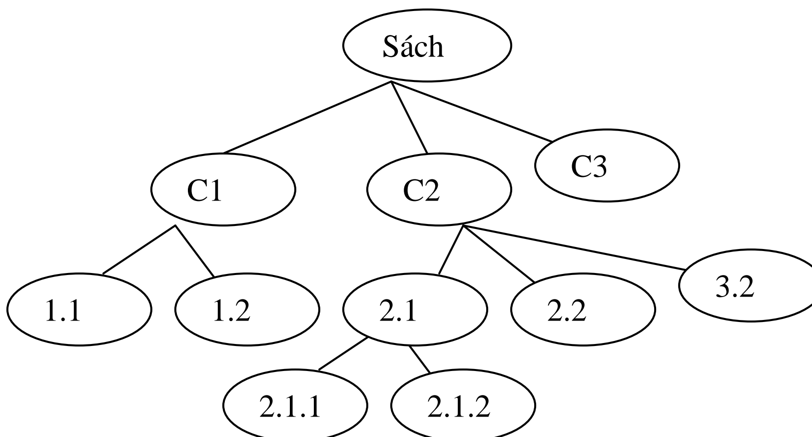
Hoặc một cách định nghĩa khác (đọc thêm)

Cây là một tập hợp các phần tử gọi là nút (nodes) trong đó có một nút được phân biệt gọi là nút gốc (root). Trên tập hợp các nút này có một quan hệ, gọi là mối quan hệ *cha - con* (parenthood), để xác dịnh hệ thống cấu trúc trên các nút. Mỗi nút, trừ nút gốc, có duy nhất một nút cha. Một nút có thể có nhiều nút con hoặc không có nút con nào. Mỗi nút biểu diễn một phần tử trong tập hợp dang xét và nó có thể có một kiểu nào đó bất kỳ, thường ta biểu diễn nút bằng một kí tự, một chuỗi hoặc một số ghi trong vòng tròn. Mối *quan hệ cha con* được biểu diễn theo qui ước *nút cha ở dòng trên nút con ở dòng dưới* và được *nối bởi một doạn thẳng*. Một cách hình thức ta có thể dịnh nghĩa cây một cách đệ qui như sau:

Định nghĩa

- Một nút đơn dộc là một cây. Nút này cũng chính là nút gốc của cây.
- Giả sử ta có n là một nút đơn độc và k cây T1,.., Tk với các nút gốc tương ứng là n1,.., nk thì có thể xây dựng một cây mới bằng cách cho nút n là cha của các nút n1,.., nk. Cây mới này có nút gốc là nút n và các cây T1,.., Tk được gọi là các cây con. Tập rỗng cũng được coi là một cây và gọi là cây rỗng kí hiệu .

Ví dụ: Xét mục lục của một quyển sách. Mục lục này có thể xem là một cây. Xét cấu trúc thư mục trong tin học, cấu trúc này cũng được xem như một cây.



Hình III.1 - Cây mục lục một quyển sách

Nếu n^1,.., n^k là một chuỗi các nút trên cây sao cho n^i là nút cha của nút n^{i+1}, với i=1..k-1, thì chuỗi này gọi là một *dường di trên cây* (hay ngắn gọn là dường di) từ n^1 đến n^k. *Độ dài dường di* được dịnh nghĩa bằng số nút trên dường di trừ 1. Như vậy dộ dài dường di từ một nút đến chính nó bằng không.

Nếu có dường di từ nút a đến nút b thì ta nói a là *tiền bối* (ancestor) của b, còn b gọi là *hậu duệ* (descendant) của nút a. Rõ ràng *một nút vừa là tiền bối vừa là hậu duệ của chính nó*. Tiền bối hoặc hậu duệ của một nút khác với chính nó gọi là tiền bối hoặc hậu duệ thực sự. Trên cây *nút gốc* không có tiền bối thực sự. Một nút không có hậu duệ thực sự gọi là *nút lá* (leaf). Nút không phải là lá ta còn gọi là *nút trung gian* (interior). Cây con của một cây là một nút cùng với tất cả các hậu duệ của nó.

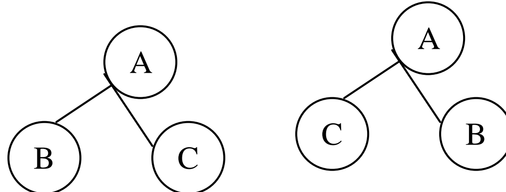
Chiều cao của một nút là dộ dài dường di lớn nhất từ nút đó tới lá. *Chiều cao của cây* là chiều cao của nút gốc. *Độ sâu của một nút* là dộ dài dường di từ nút gốc đến nút đó. Các nút có cùng một dộ sâu i ta gọi là các nút có cùng một mức i. Theo dịnh nghĩa này thì nút gốc ở mức 0, các nút con của nút gốc ở mức 1.

Ví dụ: đối với cây trong hình III.1 ta có nút C2 có chiều cao 2. Cây có chiều cao 3. nút

C3 có chiều cao 0. Nút 2.1 có dộ sâu 2. Các nút C1,C2,C3 cùng mức 1.

Thứ tự các nút trong cây

Nếu ta phân biệt thứ tự các nút con của cùng một nút thì cây gọi là cây có thứ tự, thứ tự qui ước từ trái sang phải. Như vậy, nếu kể thứ tự thì hai cây sau là hai cây khác nhau:



Hình III.2: Hai cây có thứ tự khác nhau

Trong trường hợp ta không phân biệt rõ ràng thứ tự các nút thì ta gọi là cây không có thứ tự. Các nút con cùng một nút cha gọi là các nút anh em ruột (siblings). Quan hệ "trái sang phải" của các anh em ruột có thể mở rộng cho hai nút bất kỳ theo qui tắc: nếu a, b là hai anh em ruột và a bên trái b thì các hậu duệ của a là "bên trái" mọi hậu duệ của b.

2. Cây tìm kiếm nhị phân (Binary Search Tree - BST)

2.1. Định nghĩa

Mỗi nút trong cây bất kỳ đều chứa các trường thông tin , trên một cây tìm kiếm nhị phân mỗi nút là một struct (bản ghi – record) gồm các trường: trường dữ liệu data, trường khóa key để so sánh với các nút khác, các liên kết tới các nút con của nút left và right.

Để tập trung vào các vấn đề thuật toán ta bỏ qua trường dữ liệu , chỉ xem như mỗi nút trên cây tìm kiếm nhị phân gồm có một trường khóa **key** và hai trường liên kết **left** và **right**.

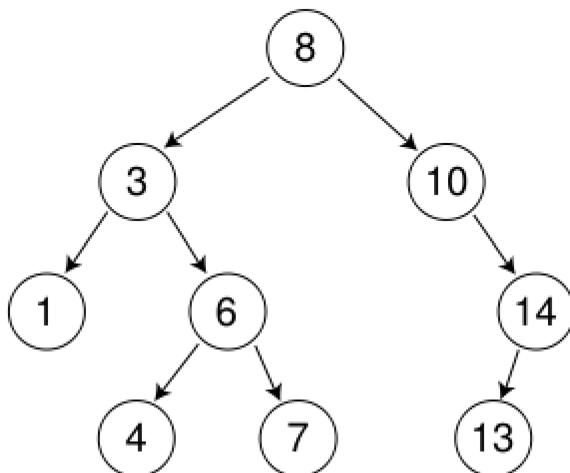
Với các giả thiết trên ta định nghĩa cây tìm kiếm nhị phân như sau:

Cây tìm kiếm nhị phân là một cây nhị phân (binary tree) mà mỗi nút x trong cây thỏa mãn bất đẳng thức kép sau:

$$key(left_child(x)) < key(x) < key(right_child(x))$$

Trong đó left_child(x), right_child(x) là các nút con trái và phải của nút x , key() là hàm trả về giá trị khóa ở nút tương ứng.

Ví dụ:



Hình 5.4. Cây tìm kiếm nhị phân BST, tham khảo từ wikipedia.

Nhận xét:

- Trên cây BST không có hai nút cùng khoá.
- Cây con của một cây BST là cây BST.

Ưu điểm chính của cây tìm kiếm nhị phân là : nó cung cấp thuật toán sắp xếp và tìm kiếm dựa trên kiểu duyệt thứ tự giữa (in-order) một cách rất hiệu quả, và là cấu trúc dữ liệu cơ bản cho các cấu trúc dữ liệu cao cấp hơn (trừu tượng hơn) như tập hợp (set), các mảng liên kết

(associative array), các ánh xạ **map**, và các cây cân bằng tối ưu như AVL, cây đỏ đen. Chúng ta sẽ xem xét tại sao cây tìm kiếm nhị phân lại hiệu quả như vậy.

2.2. Khởi tạo cây rỗng

Thao tác đầu tiên là khai báo cấu trúc cây và khởi tạo một cây rỗng để bắt đầu thực hiện các thao tác khác.

Ở đây ta giả sử cây tìm kiếm nhị phân chỉ chứa các khóa là các số nguyên dương.

Khai báo cây tìm kiếm nhị phân trong ngôn ngữ C như sau:

```
// khai bao cau truc cay tim kiem nhi phan
typedef struct  tree
{
  int       key;
  struct  tree  *left,*right;
}BSTree;
```

Để khởi tạo một cây rỗng ta khai báo gốc của cây và gán cho gốc đó bằng NULL :

```
// cay
BSTree  **root;
*root = NULL;
```

2.3. Chèn thêm một nút mới vào cây

Để chèn một nút mới vào cây ta xuất phát từ gốc của cây, ta gọi đó là nút đang xét. Nếu như nút đang xét có khóa bằng với khóa cần chèn vào cây thì xảy ra hiện tượng trùng khóa, thuật toán kết thúc với thông báo trùng khóa. Nếu như nút đang xét là một nút ngoài (external nodes) thì ta tạo một nút mới và gán các trường thông tin tương ứng cho nút đó, gán các con của nút đó bằng NULL.

```
// them mot nut moi vao cay, gia tri khoa cua nut moi luu trong bien toan cuc newkey
void insert(BSTree **root)
{
  if(*root==NULL)
  {
        *root=calloc(1,sizeof(BSTree));
        (*root)->key = newkey;
        (*root)->left=NULL;
        (*root)->right=NULL;
  }else{
        if((*root)->key>newkey)
              insert(&((*root)->left));
        else
if((*root)->key<newkey)
                    insert(&((*root)->right));
              else
                    printf("\nError: Duplicate key");
  }
}
```

Thuật toán trên sử dụng bộ nhớ Θ(log n) trong trường hợp trung bình và Ω(n) trong trường hợp tồi nhất. Độ phức tạp thuật toán bằng với độ cao của cây, tức là O(log n) trong trường hợp trung bình đối với hầu hết các cây, nhưng sẽ là Ω(n) trong trường hợp xấu nhất.

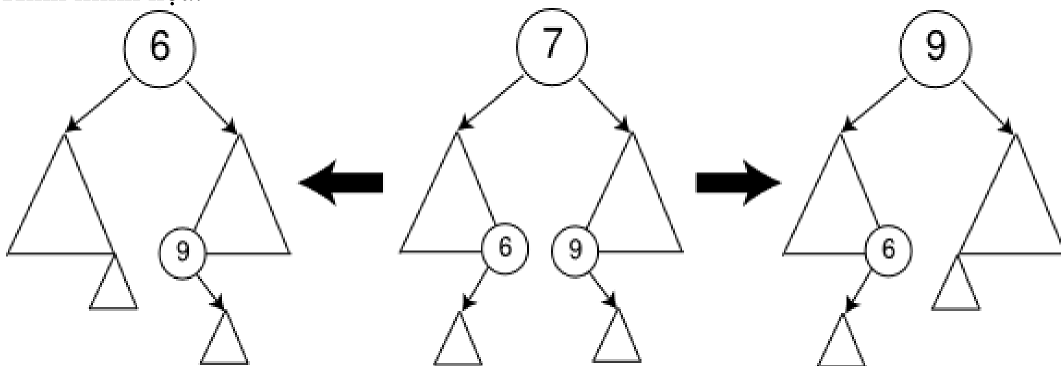
Cũng nên chú ý là các nút mới luôn được chèn vào các nút ngoài của cây tìm kiếm nhị phân, gốc của cây không thay đổi trong quá trình chèn thêm nút vào cây.

2.4. Xóa bỏ khỏi cây một nút

Khi xóa bỏ một nút X khỏi cây (dựa trên giá trị khóa), chúng ta chia ra một số trường hợp sau:

- X là một nút lá: khi đó việc xóa nút không làm ảnh hưởng tới các nút khác, ta chỉ việc xóa bỏ nút đó khỏi cây.
- X chỉ có một nút con (trái hoặc phải): khi đó ta đưa nút con duy nhất của X lên thay cho nút X và xóa bỏ X.
- Còn nếu X là một nút trong và có hai con, ta sẽ có hai lựa chọn, một là tìm nút hậu duệ nhỏ nhất bên nhánh phải của X (gọi là Y), thay khóa của Y lên X và xóa bỏ Y. Cách thứ hai là tìm nút hậu duệ lớn nhất bên nhánh trái của X (gọi là Z), thay khóa của Z lên X và xóa bỏ Z. Các thao tác với Y hoặc Z được lặp lại tương tự như đối với X.

Hình minh họa:



Hình 5.5. Xóa nút trên cây BST, tham khảo từ wikipedia

Do các nút thực sự bị xóa trong trường hợp thứ ba sẽ có thể rơi vào trường hợp 1 hoặc 2 (là các nút lá hoặc các nút chỉ có 1 con), đồng thời nút bị xóa sẽ có khóa nhỏ hơn hai con của X nên trong cài đặt ta nên tránh chỉ sử dụng một phương pháp, vì có thể dẫn tới tình huống mất tính cân bằng của cây.

Việc cài đặt thuật toán xóa một nút trên cây tìm kiếm nhị phân không đơn giản như việc mô tả thuật toán xóa ở trên. Trước hết ta sẽ xuất phát từ gốc của cây để đi tìm nút chứa khóa cần xóa trên cây. Trong quá trình này điều quan trọng là ta xác định rõ nút cần xóa (biến p trong đoạn mã chương trình bên dưới) là một nút lá, hay là một nút chỉ có một con, hay là nút có đầy đủ cả hai con. Dù trong trường hợp nào thì chúng ta cũng cần xác định nút cha của nút p (nút q), và p là con trái hay con phải của q. Để xác định các trường hợp trên ta sử dụng một biến cờ f, f bằng 0 tương ứng với việc nút cần xóa là gốc của cây, f bằng 1 tương ứng với p là con phải của q, và f bằng 2 tương ứng với p là con trái của q.

Cài đặt bằng C của thao tác xóa một nút khỏi cây BST:

```
// xoa bo mot khoa khoi cay
void del(BSTree ** root, int key)
{
  BSTree *p, *q, *r;
  int f=0;
  p = *root;
  q = NULL;
  while(p!=NULL&&p->key!=key)
  {
          q = p;
          if(p->key<key)
          {
```

```
                f = 1;
                p = p->right;
            }
            else
            {
                f = 2;
                p = p->left;
            }
    }
    if(p!=NULL)
    {
            if(p->right==NULL)
            {
                if(f==1)
                {
                    q->right=p->left;
                    free(p);
                }
                else if(f==2)
                {
                    q->left=p->left;
                    free(p);
                }else
                {
                    *root = p->left;
                    free(p);
                }
            }else
            {
                q = p->right;
                r = NULL;
                while(q->left)
                {
                    r = q;
                    q = q->left;
                }
                p->key = q->key;
                if(r==NULL)
                    p->right = q->right;
                else
                    r->left = q->right;
                free(q);
            }
    }
}
```

Mặc dù việc xóa cây không phải luôn đòi hỏi phải duyệt từ gốc xuống thực hiện ở một nút lá nhưng tình huống này luôn có thể xảy ra (duyệt qua từng nút tới một nút lá), khi đó độ phức tạp của thuật toán xóa cây tương đương với độ cao của cây (tình huống tồi nhất).

2.5. Tìm kiếm trên cây

Việc tìm kiếm trên cây n hị phân tìm kiếm giống như khi ta thêm một nút mới vào cây . Dựa trên khóa tìm kiếm key ta xuất phát từ gốc , gọi nút đang xét là X . Nếu khóa của X bằng

với key, thì kết thúc và trả về X . Nếu X là một nút lá thì kết quả trả về NULL (cũng chính là X). Nếu khóa của X nhỏ hơn key thì ta lặp lại thao tác tìm kiếm với nút con phải của X , ngược lại thì tiến hành tìm kiếm với nút con trái của X.

Độ phức tạp của thuật toán này bằng với độ phức tạp của thuật toán chèn một nút mới vào cây.

Cài đặt của thuật toán được để lại như một bài tập dành cho các bạn độc giả.

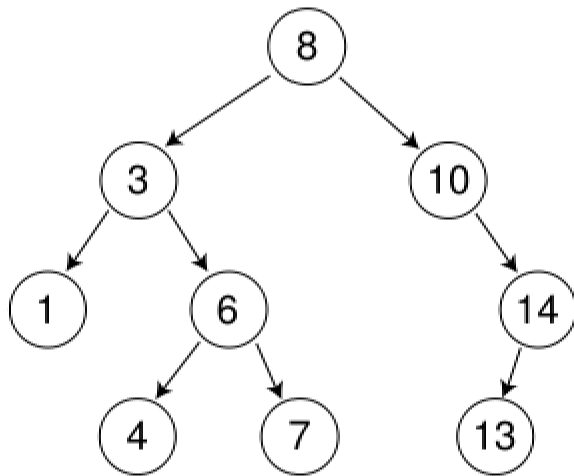
2.6. Duyệt cây

Duyệt cây (tree travel) là thao tác duyệt qua (đến thăm) tất cả các nút trên cây.

Có nhiều cách để duyệt một cây , chẳng hạn như duyệt theo chiều sâu (DFS), duyệt theo chiều rộng (BFS), nhưng ở đây ta phân chia các cách duyệt một cây BST dựa trên thứ tự đến thăm nút gốc, nút con trái, và nút con phải của gốc.

Cụ thể có ba cách duyệt một cây BST: duyệt thứ tự trước, thứ tự giữa, thứ tự sau.

Để minh họa kết quả của các cách duyệt cây ta xét cây ví dụ sau:



Hình 5.6. Cây tìm kiếm nhị phân, tham khảo từ wikipedia

Duyệt thứ tự trước (pre-order traversal):

- Thăm gốc (visit root).
- Duyệt cây con trái theo thứ tự trước
- Duyệt cây con phải theo thứ tự trước.

Cụ thể thuật toán được cài đặt như sau:

```
// duyet theo thu tu truoc
void pre_order(BSTree *node)
{
  if(node!=NULL)
  {
        visit(node); // ham tham mot nut, don gian la in gia tri khoa
        pre_order(node->left);
        pre_order(node->right);
  }
}
```

Kết quả duyệt cây theo thứ tự trước: 8, 3, 1, 6, 4, 7, 10, 14, 13.

Trong cách duyệt theo thứ tự trước, gốc của cây luôn được thăm đầu tiên.

Duyệt thứ tự giữa (in-order traversal):

- Duyệt cây con trái theo thứ tự giữa
- Thăm gốc

• Duyệt cây con phải theo thứ tự giữa.

Kết quả duyệt cây theo thứ tự trước: 1, 3, 4, 6, 7, 8, 10, 13, 14.

Một điều dễ nhận thấy là các khóa của cây khi duyệt theo thứ tự giữa xuất hiện theo thứ tự tăng dần.

Duyệt thứ tự sau (post-order traversal):

• Duyệt cây con trái theo thứ tự sau

• Duyệt cây con phải theo thứ tự sau

• Thăm gốc

Kết quả duyệt cây theo thứ tự sau: 1, 4, 7, 6, 3, 13, 14, 10, 8.

Trong cách duyệt này, gốc được thăm sau cùng.

Nhận xét: - Khi duyệt trung tự (InOrder) cây BST ta được một dãy có thứ tự tăng.

Cài đặt bằng C của hai cách duyệt sau được dành cho các bạn độc giả như một bài tập.

2.7. Cài đặt cây BST

Cây TKNP, trước hết, là một cây nhị phân. Do đó, ta có thể áp dụng các cách cài đặt như đã trình bày trong phần cây nhị phân. Sẽ không có sự khác biệt nào trong việc cài đặt cấu trúc dữ liệu cho cây TKNP so với cây nhị phân, nhưng tất nhiên, sẽ có sự khác biệt trong các giải thuật thao tác trên cây TKNP như tìm kiếm, thêm hoặc xoá một nút trên cây TKNP để luôn đảm bảo tính chất cuả cây TKNP.

Một cách cài đặt cây TKNP thường gặp là cài đặt bằng con trỏ. Mỗi nút của cây như là một mẩu tin (record) có ba trường: một trường chứa khoá, hai trường kia là hai con trỏ trỏ đến hai nút con (nếu nút con vắng mặt ta gán con trỏ bằng NIL)

Khai báo như sau

```
typedef <kiểu dữ liệu của khoá> KeyType;
typedef struct Node
{
KeyType Key;
Node* Left,Right;
}
typedef Node* Tree;
```

Khởi tạo cây TKNP rỗng

Ta cho con trỏ quản lý nút gốc (Root) của cây bằng NULL.

```
void MakeNullTree(Tree *Root)
{
(*Root)=NULL;
}
```

Tìm kiếm một nút có khóa cho trước trên cây TKNP

Để tìm kiếm 1 nút có khoá x trên cây TKNP, ta tiến hành từ nút gốc bằng cách so sánh khoá của nút gốc với khoá x.

- Nếu nút gốc bằng NULL thì không có khoá x trên cây.
- Nếu x bằng khoá của nút gốc thì giải thuật dừng và ta đã tìm được nút chứa khoá x.
- Nếu x lớn hơn khoá của nút gốc thì ta tiến hành (một cách đệ qui) việc tìm khoá x trên cây con bên phải.
- Nếu x nhỏ hơn khoá của nút gốc thì ta tiến hành (một cách đệ qui) việc tìm khoá x trên cây con bên trái.

Ví dụ: tìm nút có khoá 30 trong cây ở trong hình III.15

- So sánh 30 với khoá nút gốc là 20, vì 30 > 20 vậy ta tìm tiếp trên cây con bên phải, tức là cây có nút gốc có khoá là 35.
- So sánh 30 với khoá của nút gốc là 35, vì 30 < 35 vậy ta tìm tiếp trên cây con bên

trái, tức là cây có nút gốc có khoá là 22.

- So sánh 30 với khoá của nút gốc là 22, vì 30 > 22 vậy ta tìm tiếp trên cây con bên phải, tức là cây có nút gốc có khoá là 30.

- So sánh 30 với khoá nút gốc là 30, 30 = 30 vậy đến đây giải thuật dừng và ta tìm được nút chứa khoá cần tìm.

- Hàm dưới đây trả về kết quả là con trỏ trỏ tới nút chứa khoá x hoặc NULL nếu không tìm thấy khoá x trên cây TKNP.

```
Tree Search(KeyType x,Tree Root)
{
if (Root == NULL) return NULL;   //không tìm thấy khoá x
else if (Root->Key == x) /* tìm thấy khoá x */
return Root;
else if (Root->Key < x) //tìm tiếp trên cây bên phải
return Search(x,Root->right);
else
//tìm tiếp trên cây bên trái
return Search(x,Root->left);
}
```

Câu hỏi ôn tập:

Cây tìm kiếm nhị phân được tổ chức như thế nào để quá trình tìm kiếm được hiệu quả nhất?

Nhận xét: *giải thuật này sẽ rất hiệu quả về mặt thời gian nếu cây TKNP* được *tổ chức tốt, nghĩa là cây tương đối "cân bằng". Về chủ dề cây cân bằng các bạn có thể tham khảo thêm trong các tài liệu tham khảo của môn này.*

Thêm một nút có khóa cho trước vào cây TKNP

Theo dịnh nghĩa cây tìm kiếm nhị phân ta thấy trên cây tìm kiếm nhị phân không có hai nút có cùng một khoá. Do đó, nếu ta muốn thêm một nút có khoá x vào cây TKNP thì trước hết ta phải tìm kiếm để xác dịnh có nút nào chứa khoá x chưa. Nếu có thì giải thuật kết thúc (không làm gì cả!). Ngược lại, sẽ thêm một nút mới chứa khoá x này. Việc thêm một khoá vào cây TKNP là việc tìm kiếm và thêm một nút, tất nhiên, phải đảm bảo cấu trúc cây TKNP không bị phá vỡ. Giải thuật cụ thể như sau:

Ta tiến hành từ nút gốc bằng cách so sánh khóa cuả nút gốc với khoá x.

- Nếu nút gốc bằng NULL thì khoá x chưa có trên cây, do đó ta thêm một nút mới chứa khoá x.

- Nếu x bằng khoá của nút gốc thì giải thuật dừng, trường hợp này ta không thêm nút.

- Nếu x lớn hơn khoá của nút gốc thì ta tiến hành (một cách đệ qui) giải thuật này trên cây con bên phải.

- Nếu x nhỏ hơn khoá của nút gốc thì ta tiến hành (một cách đệ qui) giải thuật này trên cây con bên trái.

Ví dụ: thêm khoá 19 vào cây ở trong hình III.15

So sánh 19 với khoá của nút gốc là 20, vì 19 < 20 vậy ta xét tiếp đến cây bên trái, tức là cây có nút gốc có khoá là 10.

- So sánh 19 với khoá của nút gốc là 10, vì 19 > 10 vậy ta xét tiếp đến cây bên phải, tức là cây có nút gốc có khoá là 17.

- So sánh 19 với khoá của nút gốc là 17, vì 19 > 17 vậy ta xét tiếp đến cây bên phải. Nút con bên phải bằng NULL, chứng tỏ rằng khoá 19 chưa có trên cây, ta thêm nút mới chứa khoá 19 và nút mới này là con bên phải của nút có khoá là 17, xem hình III.16

Hình III.16: Thêm khoá 19 vào cây hình III.15

Thủ tục sau dây tiến hành việc thêm một khoá vào cây TKNP.

```
void InsertNode(KeyType x,Tree *Root ){
if (*Root == NULL){ /* thêm nút mới chứa khoá x */
```

```
(*Root)=(Node*)malloc(sizeof(Node));
(*Root)->Key = x;
(*Root)->left = NULL;
(*Root)->right = NULL;
}
else
if (x < (*Root)->Key)
InsertNode(x,Root->left);
else if (x>(*Root)->Key) InsertNode(x,Root->right);
}
```

Xóa một nút có khóa cho trước ra khỏi cây TKNP

Giả sử ta muốn xoá một nút có khoá x, trước hết ta phải tìm kiếm nút chứa khoá x trên cây.

Việc xoá một nút như vậy, tất nhiên, ta phải bảo đảm cấu trúc cây TKNP không bị phá vỡ. Ta có các trường hợp như hình III.17:

Hình III.17 Ví dụ về giải thuật xóa nút trên cây

- Nếu không tìm thấy nút chứa khoá x thì giải thuật kết thúc.
- Nếu tìm gặp nút N có chứa khoá x, ta có ba trường hợp sau (xem hình III.17)
- Nếu N là lá ta thay nó bởi NULL.
- N chỉ có một nút con ta thay nó bởi nút con của nó.
- N có hai nút con ta thay nó bởi nút lớn nhất trên cây con trái của nó (nút cực phải của cây con trái) hoặc là nút bé nhất trên cây con phải của nó (nút cực trái của cây con phải). Trong giải thuật sau, ta thay x bởi khoá của nút cực trái của cây con bên phải rồi ta xoá nút cực trái này. Việc xoá nút cực trái của cây con bên phải sẽ roi vào một trong hai trường hợp trên.

Giải thuật xoá một nút có khoá nhỏ nhất

Hàm dưới dây trả về khoá của nút cực trái, dồng thời xoá nút này.

```
KeyType DeleteMin (Tree *Root )
{
KeyType k;
if ((*Root)->left == NULL){
k=(*Root)->key;
(*Root) = (*Root)->right;
return k;
}
else return DeleteMin(Root->left);
}
```

Thủ tục xóa một nút có khoá cho trước trên cây TKNP

```
void DeleteNode(key X, Tree *Root)
{
if ((*Root)!=NULL)
if (x < (*Root)->Key)
        DeleteNode(x,Root->left)
else if (x > (*Root)->Key)
        DeleteNode(x,Root->right) else
if ((*Root)->left==NULL)&&((*Root)->right==NULL)
        (*Root)=NULL;
else
if ((*Root)->left == NULL)
        (*Root) = (*Root)->right ;
else if ((*Root)->right==NULL)
        (*Root) = (*Root)->left;
```

```
else (*Root)->Key = DeleteMin(Root->right);
}
```

3.Cây cân bằng – AVL

Trong khoa học máy tính, một **cây AVL** là một cây tìm kiếm nhị phân tự cân bằng, và là cấu trúc dữ liệu đầu tiên có khả năng này. Trong một cây AVL, tại mỗi nút chiều cao của hai cây con sai khác nhau không quá một. Hiệu quả là các phép chèn (insertion), và xóa (deletion) luôn chỉ tốn thời gian O(log *n*) trong cả trường hợp trung bình và trường hợp xấu nhất. Phép bổ sung và loại bỏ có thể cần đến việc tái cân bằng bằng một hoặc nhiều phép quay.

3.1. Cây nhị phân cân bằng hoàn toàn

3.1.1. Định nghĩa

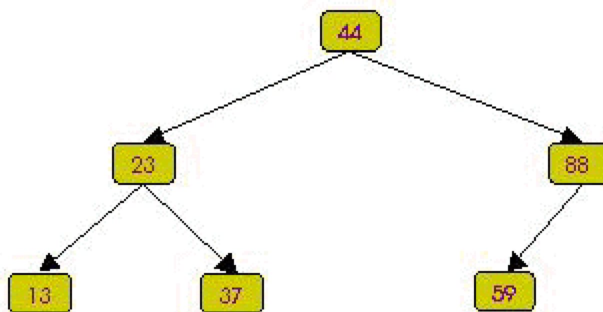
Cây cân bằng hoàn toàn là cây nhị phân tìm kiếm mà tại mỗi nút của nó, số nút của cây con trái chênh lệch không quá một so với số nút của cây con phải.

3.1.2. Đánh giá

Một cây rất khó đạt được trạng thái cân bằng hoàn toàn và cũng rất dễ mất cân bằng vì khi thêm hay hủy các nút trên cây có thể làm cây mất cân bằng (xác suất rất lớn), chi phí cân bằng lại cây lớn vì phải thao tác trên toàn bộ cây.

Tuy nhiên nếu cây cân đối thì việc tìm kiếm sẽ nhanh. Đối với cây cân bằng hoàn toàn, trong trường hợp xấu nhất ta chỉ phải tìm qua $\log_2 n$ phần tử (n là số nút trên cây).

Sau đây là ví dụ một cây cân bằng hoàn toàn (CCBHT):



CCBHT có n nút có chiều cao $h \approx \log_2 n$. Đây chính là lý do cho phép bảo đảm khả năng tìm kiếm nhanh trên CTDL này.

Do CCBHT là một cấu trúc kém ổn định nên trong thực tế không thể sử dụng. Nhưng ưu điểm của nó lại rất quan trọng. Vì vậy, cần đưa ra một CTDL khác có đặc tính giống CCBHT nhưng ổn định hơn.

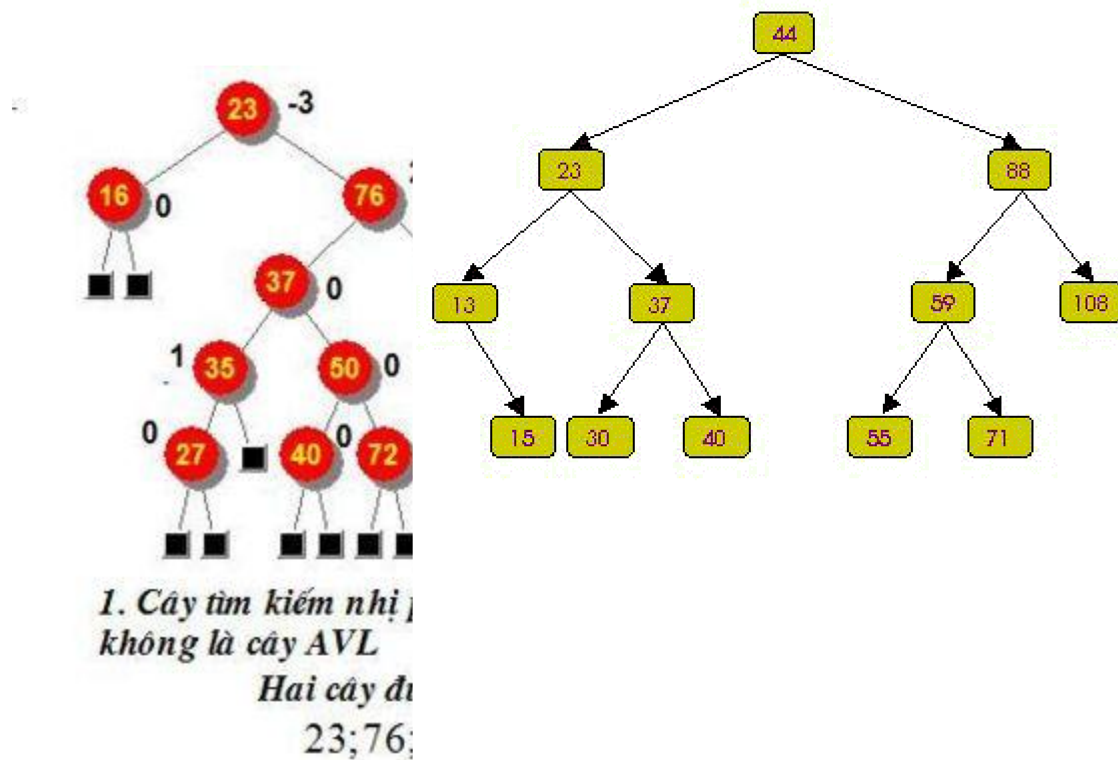
Như vậy, cần tìm cách tổ chức một cây đạt trạng thái cân bằng yếu hơn và việc cân bằng lại chỉ xảy ra ở phạm vi cục bộ nhưng vẫn phải bảo đảm chi phí cho thao tác tìm kiếm đạt ở mức $O(\log_2 n)$.

3.2. CÂY NHỊ PHÂN CÂN BẰNG (AVL)

3.2.1 Định nghĩa:

Cây nhị phân tìm kiếm cân bằng là cây mà tại mỗi nút của nó độ cao của cây con trái và của cây con phải chênh lệch không quá một.

Dưới đây là ví dụ cây cân bằng (lưu ý, cây này không phải là cây cân bằng hoàn toàn):



Dễ dàng thấy CCBHT là cây cân bằng. Điều ngược lại không đúng.

3.2.2 Lịch sử cây cân bằng (AVL Tree)

AVL là tên viết tắt của các tác giả người Nga đã đưa ra định nghĩa của cây cân bằng Adelson-Velskii và Landis (1962). Vì lý do này, người ta gọi cây nhị phân cân băng là cây AVL. Tù nay về sau, chúng ta sẽ dùng thuật ngữ cây AVL thay cho cây cân bằng.

Từ khi được giới thiệu, cây AVL đã nhanh chóng tìm thấy ứng dụng trong nhiều bài toán khác nhau. Vì vậy, nó mau chóng trở nên thịnh hành và thu hút nhiều nghiên cứu. Từ cây AVL, người ta đã phát triển thêm nhiều loại CTDL hữu dụng khác như cây đỏ-đen (Red-Black Tree), B-Tree, …

3.2.3 Chiều cao của cây AVL

Một vấn đề quan trọng, như đã đề cập đến ở phần trước, là ta pjải khẳng định cây AVL n nút phải có chiều cao khoảng $\log_2(n)$.

Để đánh giá chính xác về chiều cao của cây AVL, ta xét bài toán: cây AVL có chiều cao h sẽ phải có tối thiểu bao nhiêu nút ?

Gọi N(h) là số nút tối thiểu của cây AVL có chiều cao h.

Ta có $N(0) = 0$, $N(1) = 1$ và $N(2) = 2$.

Cây AVL tối thiểu có chiều cao h sẽ có 1 cây con AVL tối thiểu chiều cao h-1 và 1 cây con AVL tối thiểu chiều cao h-2. Như vậy:

$N(h) = 1 + N(h-1) + N(h-2)$ (1)

Ta lại có: $N(h-1) > N(h-2)$

Nên từ (1) suy ra:

$N(h) > 2N(h-2)$

$N(h) > 2^2N(h-4)$

…

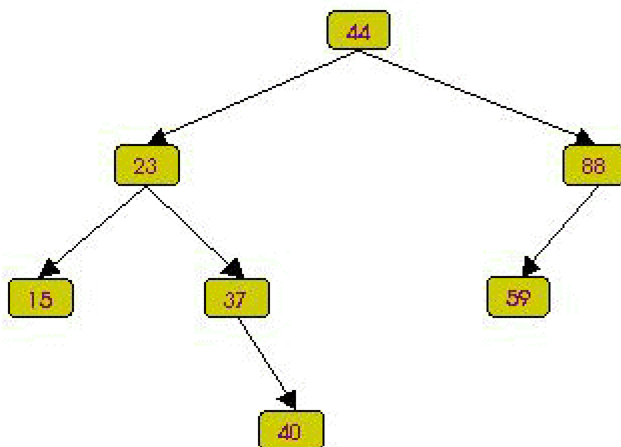
$N(h) > 2^{iN}(h-2i)$

$\Rightarrow N(h) > 2^{h/2-1}$

$\Rightarrow h < 2\log_2(N(h)) + 2$

Như vậy, cây AVL có chiều cao $O(\log_2(n))$.

Ví dụ: cây AVL tối thiểu có chiều cao h=4



3.2.4 Cấu trúc dữ liệu cho cây AVL

Chỉ số cân bằng của một nút:

Định nghĩa: Chỉ số cân bằng của một nút là hiệu của chiều cao cây con phải và cây con trái của nó.

Đối với một cây cân bằng, chỉ số cân bằng (CSCB) của mỗi nút chỉ có thể mang một trong ba giá trị sau đây:

CSCB(p) = 0 <=> Độ cao cây trái (p) = Độ cao cây phải (p)

CSCB(p) = 1 <=> Độ cao cây trái (p) < Độ cao cây phải (p)

CSCB(p) =-1 <=> Độ cao cây trái (p) > Độ cao cây phải (p)

Để tiện trong trình bày, chúng ta sẽ ký hiệu như sau:

p->balFactor = CSCB(p);

Độ cao cây trái (p) ký hiệu là hL

Độ cao cây phải(p) ký hiệu là hR

Để khảo sát cây cân bằng, ta cần lưu thêm thông tin về chỉ số cân bằng tại mỗi nút. Lúc đó, cây cân bằng có thể được khai báo như sau:

```
typedef struct tagAVLNode {

    char balFactor; //Chỉ số cân bằng

    Data key;

    struct tagAVLNode* pLeft;

    struct tagAVLNode* pRight;

}AVLNode;

typedef AVLNode *AVLTree;
```

Để tiện cho việc trình bày, ta định nghĩa một số hăng số sau:

```
#define LH -1 //Cây con trái cao hơn

#define EH -0  //Hai cây con bằng nhau

#define RH  1  //Cây con phải cao hơn
```

3.2.5 Đánh giá cây AVL

- Cây cân bằng là CTDL ổn định hơn hẳn CCBHT vì chỉ khi thêm hủy làm cây thay đổi chiều cao các trường hợp mất cân bằng mới có khả năng xảy ra.

- Cây AVL với chiều cao được khống chế sẽ cho phép thực thi các thao tác tìm thêm hủy với chi phí O ($\log_2(n)$) và bảo đảm không suy biến thành O(n).

3.3. Các thao tác cơ bản trên cây AVL

Ta nhận thấy trường hợp thêm hay hủy một phần tử trên cây có thể làm cây tăng hay giảm chiều cao, khi đó phải cân bằng lại cây. Việc cân bằng lại một cây sẽ phải thực hiện sao cho chỉ ảnh hưởng tối thiểu đến cây nhằm giảm thiểu chi phí cân bằng. Như đã nói ở trên, cây cân

bằng cho phép việc cân bằng lại chỉ xảy ra trong giới hạn cục bộ nên chúng ta có thể thực hiện được mục tiêu vừa nêu.

Như vậy, ngoài các thao tác bình thường như trên CNPTK, các thao tác đặc trưng của cây AVL gồm:

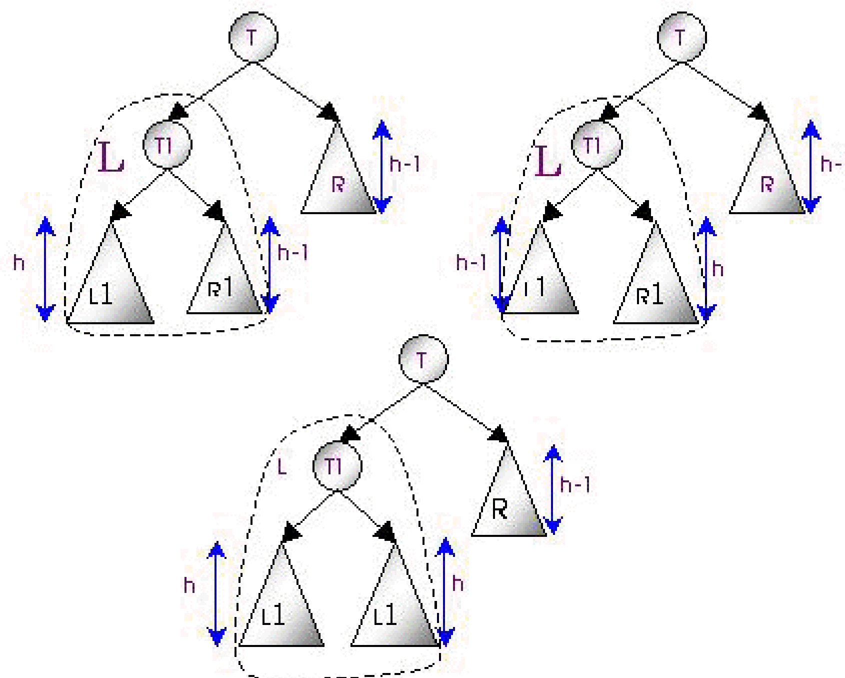
> **Thêm một phần tử vào cây AVL.**
>
> **Hủy một phần tử trên cây AVL.**
>
> **Cân bằng lại một cây vừa bị mất cân bằng.**

3.3.1. CÁC TRƯỜNG HỢP MẤT CÂN BẰNG

Ta sẽ không khảo sát tính cân bằng của 1 cây nhị phân bất kỳ mà chỉ quan tâm đến các khả năng mất cân bằng xảy rakhi thêm hoặc hủy một nút trên cây AVL.

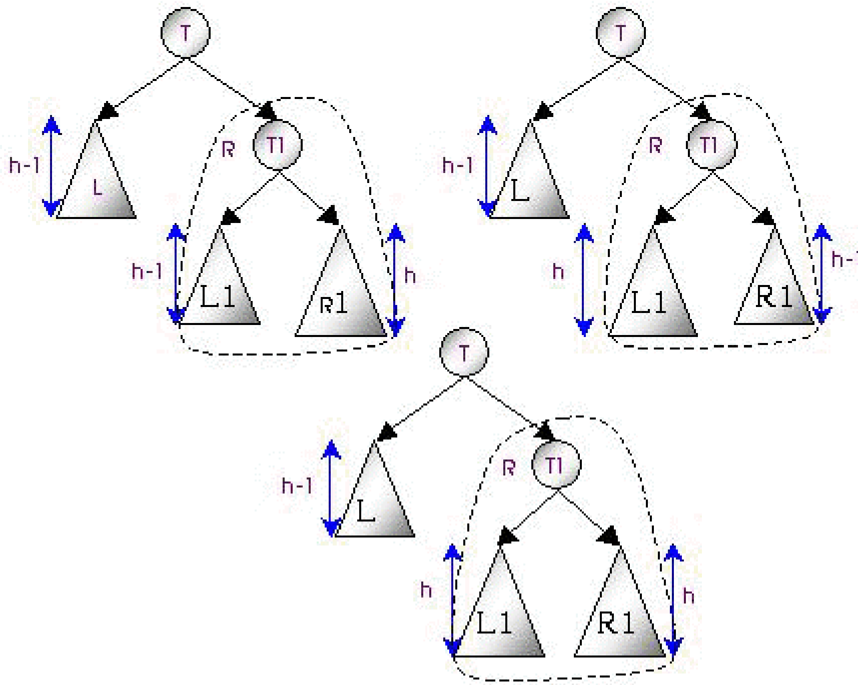
Như vậy, khi mất cân bằng, độ lệch chiều cao giữa 2 cây con sẽ là **2**. Ta có 6 khả năng sau:

Trường hợp 1: **cây T lệch về bên trái (có 3 khả năng)**



Trường hợp 2: **cây T lệch về bên phải**

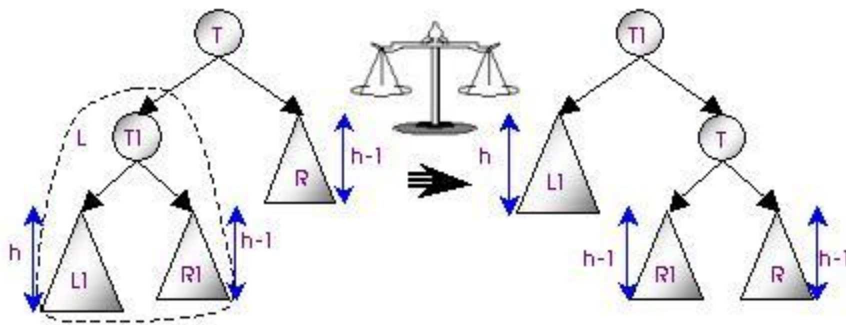
Ta có các khả năng sau:



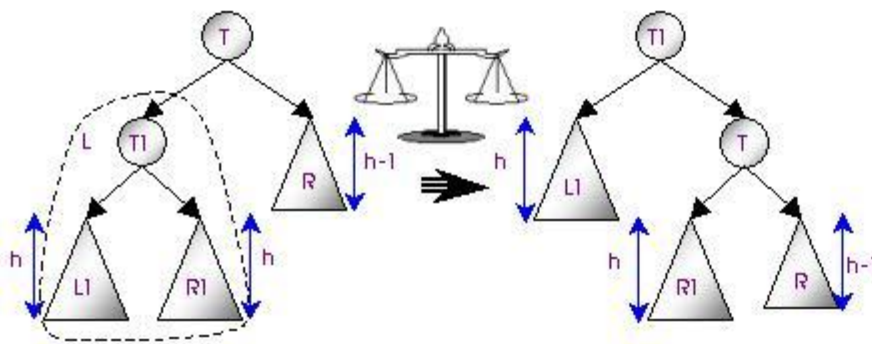
Ta có thể thấy rằng các trường hợp lệch về bên phải hoàn toàn đối xứng với các trường hợp lệch về bên trái. Vì vậy ta chỉ cần khảo sát trường hợp lệch về bên trái. Trong 3 trường hợp lệch về bên trái, trường hợp T1 lệch phải là phức tạp nhất. Các trường hợp còn lại giải quyết rất đơn giản.

Sau đây, ta sẽ khảo sát và giải quyết từng trường hợp nêu trên.

T/h 1.1: cây T1 lệch về bên trái. Ta thực hiện phép quay đơn Left-Left



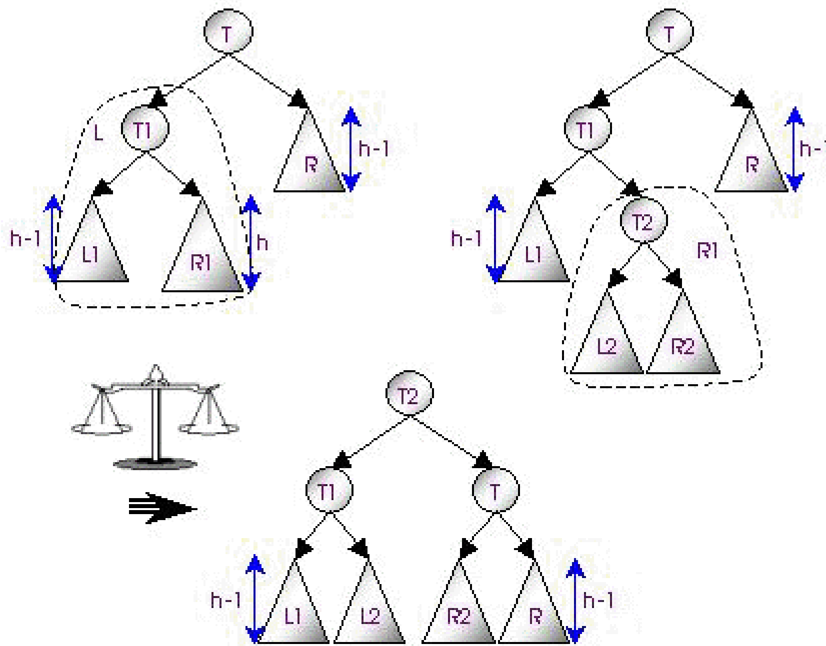
T/h 1.2: cây T1 không lệch. Ta thực hiện phép quay đơn Left-Left



T/h 1.3: cây T1 lệch về bên phải. Ta thực hiện phép quay kép Left-Right

Do T1 lệch về bên phải ta không thể áp dụng phép quay đơn đã áp dụng trong 2 trường hợp trên vì khi đó cây T sẽ chuyển từ trạng thái mất cân bằng do lệch trái thành mất cân bằng do lệch phải ⇒ cần áp dụng cách khác.

Hình vẽ dưới đây minh họa phép quay kép áp dụng cho trường hợp này:



Lưu ý rằng, trước khi cân bằng cây T có chiều cao h+2 trong cả 3 trường hợp 1.1, 1.2 và 1.3. Sau khi cân bằng, trong 2 trường hợp 1.1 và 1.3 cây có chiều cao h+1; còn ở trường hợp 1.2 cây vẫn có chiều cao h+2. Và trường hợp này cũng là trường hợp duy nhất sau khi cân bằng nút T cũ có chỉ số cân bằng ≠ 0.

Thao tác cân bằng lại trong tất cả các trường hợp đều cóù độ phức tạp O(1).

Với những xem xét trên, xét tương tự cho trường hợp cây T lệch về bên phải, ta có thể xây dựng 2 hàm quay đơn và 2 hàm quay kép sau:

```
//quay đơn Left-Left

void rotateLL(AVLTree &T)

{ AVLNode* T1 = T->pLeft;

     T->pLeft = T1->pRight;

     T1->pRight = T;

     switch(T1->balFactor) {

          case LH: T->balFactor = EH;
```

```
                        T1->balFactor = EH; break;
          case EH: T->balFactor = LH;
                        T1->balFactor = RH; break;
     }
     T = T1;
}
```

//quay đơn Right-Right

```
void rotateRR(AVLTree &T)
{ AVLNode* T1 = T->pRight;
     T->pRight = T1->pLeft;
     T1->pLeft = T;
     switch(T1->balFactor) {
               case RH: T->balFactor = EH;
                    T1->balFactor = EH; break;
               case EH: T->balFactor = RH; break;
                    T1->balFactor = LH; break;
     }
     T = T1;
}
```

//quay kép Left-Right

```
void rotateLR(AVLTree &T)
{ AVLNode* T1 = T->pLeft;
     AVLNode* T2 = T1->pRight;
     T->pLeft = T2->pRight;
     T2->pRight = T;
     T1->pRight = T2->pLeft;
```

```
    T2->pLeft = T1;
    switch(T2->balFactor) {
        case LH: T->balFactor = RH;
        T1->balFactor = EH; break;
        case EH: T->balFactor = EH;
        T1->balFactor = EH; break;
        case RH: T->balFactor = EH;
        T1->balFactor = LH; break;
    }
    T2->balFactor = EH;
    T = T2;
}
```

//quay kép Right-Left

```
void rotateRL(AVLTree &T)
{ AVLNode* T1 = T->pRight;
    AVLNode* T2 = T1->pLeft;
    T->pRight = T2->pLeft;
    T2->pLeft = T;
    T1->pLeft = T2->pRight;
    T2->pRight = T1;
    switch(T2->balFactor) {
        case RH: T->balFactor = LH;
        T1->balFactor = EH; break;
        case EH: T->balFactor = EH;
        T1->balFactor = EH; break;
        case LH: T->balFactor = EH;
```

```
                T1->balFactor = RH; break;
          }
          T2->balFactor = EH;
          T = T2;
     }
```

Để thuận tiện, ta xây dựng 2 hàm cân bằng lại khi cây bị lệch trái hay lệch phải như sau:

```
     //Cân băng khi cây bị lêch về bên trái
     int balanceLeft(AVLTree &T)
     { AVLNode* T1 = T->pLeft;
          switch(T1->balFactor) {
                case LH: rotateLL(T); return 2;
                case EH: rotateLL(T); return 1;
                case RH: rotateLR(T); return 2;
          }
          return 0;
     }
     //Cân băng khi cây bị lêch về bên phải
     int balanceRight(AVLTree &T)
     { AVLNode* T1 = T->pRight;
          switch(T1->balFactor) {
                case LH: rotateRL(T); return 2;
                case EH: rotateRR(T); return 1;
                case RH: rotateRR(T); return 2;
          }
          return 0;
     }
```

3.3.2. Thêm một phần tử trên cây AVL

Việc thêm một phần tử vào cây AVL diễn ra tương tự như trên CNPTK. Tuy nhiên, sau khi thêm xong, nếu chiều cao của cây thay đổi, từ vị trí thêm vào, ta phải lần ngược lên gốc để kiểm tra xem có nút nào bị mất cân bằng không. Nếu có, ta phải cân bằng lại ở nút này.

Việc cân bằng lại chỉ cần thực hiện 1 lần tại nơi mất cân bằng. (*Tại sao ? Hd: chú ý những khả năng mất cân bằng có thể*)

Hàm insert trả về giá trị –1, 0, 1 khi không đủ bộ nhớ, gặp nút cũ hay thành công. Nếu sau khi thêm, chiều cao cây bị tăng, giá trị 2 sẽ được trả về:

```
int insertNode(AVLTree &T, DataType X)
{ int res;
     if(T) {
          if(T->key == X) return 0; //đã có
          if(T->key > X) {
               res = insertNode(T->pLeft, X);
               if(res < 2) return res;
               switch(T->balFactor) {
                    case RH: T->balFactor = EH;
                                        return 1;
                    case EH: T->balFactor = LH;
                                        return 2;
                    case  LH:  balanceLeft(T);  return
                    1;
               }
          }else {
               res = insertNode(T-> pRight, X);
               if(res < 2) return res;
               switch(T->balFactor) {
                    case LH: T->balFactor = EH;
                                        return 1;
```

```
                    case EH: T->balFactor = RH;
                                        return 2;
                    case RH: balanceRight(T); return
                    1;
               }
          }
     }
     T = new TNode;
     if(T == NULL) return -1; //thiếu bộ nhớ
     T->key = X; T->balFactor = EH;
     T->pLeft = T->pRight = NULL;
     return 2; // thành công, chiều cao tăng
}
```

3.3.3. Hủy một phần tử trên cây AVL

Cũng giống như thao tác thêm một nút, việc hủy một phần tử X ra khỏi cây AVL thực hiện giống như trên CNPTK. Chỉ sau khi hủy, nếu tính cân bằng của cây bị vi phạm ta sẽ thực hiện việc cân bằng lại.

Tuy nhiên việc cân bằng lại trong thao tác hủy sẽ phức tạp hơn nhiều do có thể xảy ra phản ứng dây chuyền. (*Tại sao ?*)

Hàm delNode trả về giá trị 1, 0 khi hủy thành công hoặc không có X trong cây. Nếu sau khi huỷ, chiều cao cây bị giảm, giá trị 2 sẽ được trả về:

```
int delNode(AVLTree &T, DataType X)
{ int res;
     if(T==NULL) return 0;
     if(T->key > X) {
          res = delNode (T->pLeft, X);
          if(res < 2) return res;
          switch(T->balFactor) {
               case LH: T->balFactor = EH;
```

```
                                return 2;
                case EH: T->balFactor = RH;
                                return 1;
                case RH: return balanceRight(T);
        }
}
if(T->key < X) {
        res = delNode (T->pRight, X);
        if(res < 2) return res;
        switch(T->balFactor) {
                case RH: T->balFactor = EH;
                                return 2;
                case EH: T->balFactor = LH;
                                return 1;
                case LH: return balanceLeft(T);
        }
}else { //T->key == X
        AVLNode* p = T;
        if(T->pLeft == NULL) {
                T = T->pRight; res = 2;
        }else if(T->pRight == NULL) {
                        T = T->pLeft; res = 2;
                }else { //T có cả 2 con
                        res=searchStandFor(p,T->pRight);
                        if(res < 2) return res;
                        switch(T->balFactor) {
```

```
                        case RH: T->balFactor = EH;
                        return 2;
                        case EH: T->balFactor = LH;
                        return 1;
                        case        LH:        return
                        balanceLeft(T);
                    }
                }
                delete p;
                return res;
            }
        }
```

//Tìm phần tử thế mạng

```
int searchStandFor(AVLTree &p, AVLTree &q)
{ int res;
    if(q->pLeft) {
        res = searchStandFor(p, q->pLeft);
        if(res < 2) return res;
        switch(q->balFactor) {
            case LH: q->balFactor = EH;
                            return 2;
            case EH: q->balFactor = RH;
                            return 1;
            case RH: return balanceRight(T);
        }
    }else {
        p->key = q->key;
```

```
            p = q;
            q = q->pRight;
            return 2;
        }
    }
```

3.3.4. Nhận xét

▸Thao tác thêm một nút có độ phức tạp O(1).

▸Thao tác hủy một nút có độ phức tạp O(h).

▸Với cây cân bằng trung bình 2 lần thêm vào cây thì cần một lần cân bằng lại; 5 lần hủy thì cần một lần cân bằng lại.

▸Việc huỷ 1 nút có thể phải cân bằng dây chuyền các nút từ gốc cho đên phần tử bị huỷ trong khi thêm vào chỉ cần 1 lần cân bằng cục bộ.

▸Độ dài đường tìm kiếm trung bình trong cây cân bằng gần bằng cây cân bằng hoàn toàn $\log_2 n$, nhưng việc cân bằng lại đơn giản hơn nhiều.

▸Một cây cân bằng không bao giờ cao hơn 45% cây cân bằng hoàn toàn tương ứng dù số nút trên cây là bao nhiêu.

CHƯƠNG 4. BẢNG BĂM (HASH TABLE)

Phép băm là một thuật toán được đề xuất và hiện thực trên máy tính từ những năm 50 của thế kỷ 20. Thuật toán này dựa trên ý tưởng là chuyển đổi khoá thành một số và sử dụng số này để đánh chỉ số cho bảng dữ liệu.

Như chúng ta đã biết các phép toán dựa trên các cấu trúc như cây, danh sách … chủ yếu được thực hiện thông qua việc so sánh các phần tử có cấu trúc. Do vậy thời gian thực thi lâu và phụ thuộc vào kích thước các phần tử này. Để khắc phục người ta đưa ra thuật toán sử dụng bảng băm (Hash Table). Các phép toán trên bảng băm có độ phức tạp là $\boldsymbol{O}(1)$ và không phụ thuộc vào kích thước bảng. Dưới đây là một số vấn đề chính mà chúng ta cần quan tâm trong bảng băm :

- Định nghĩa bảng băm.
- Hàm băm và các loại hàm băm.
- Xung đột và cách xử lý xung đột

1. Định nghĩa bảng băm :

1.1. Định nghĩa :

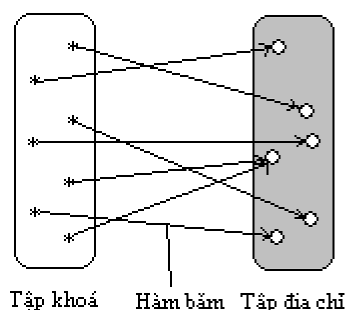
Bảng băm là một kiểu dữ liệu trừu tượng cho phép lưu trữ dữ liệu một cách nhanh chóng và hiệu quả. Về thực chất bảng băm là một mảng có chỉ số là bất cứ loại dữ liệu nào. Trong khi một mảng thông thường yêu cầu chỉ số của nó phải là số nguyên thì chỉ số bảng băm lại có thể là một số thực, một xâu, một mảng khác hay thậm chí là một dạng cấu trúc dữ liệu. Các chỉ số này người ta gọi chung là khoá (Key) và nội dung chỉ định bởi các chỉ số này gọi là các giá trị (Value).

Vậy bảng băm là một cấu trúc dữ liệu lưu trữ một cặp dữ liệu Key/Value và cho phép tìm Key một cách nhanh chóng.

Bảng băm sử dụng một hàm cho phép biến đổi bất kỳ đối tượng nào thành chỉ số phù hợp của mảng. Hàm này được gọi là hàm băm (Hash Function)

Bảng băm có thể được mô tả như sau:

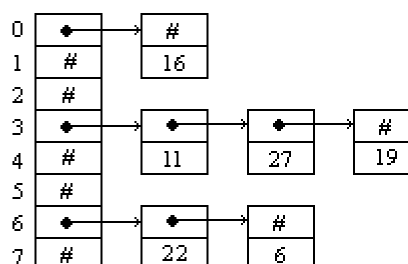
- Gọi K là tập các khoá.
- M là tập các địa chỉ.
- HF(k) là hàm băm dùng để ánh xạ một khoá k từ tập khoá k thành một chỉ số trong tập địa chỉ M.



Hình 2.1 : Mô tả về hàm băm

Sau đây là ví dụ về một bảng băm (Hàm băm trong trường hợp này có dạng : h(k) = k mod 8 trong đó k là khoá).

Hình 2.2 : Một bảng băm đơn giản



Trong bảng băm nhiều khoá có giá trị khác nhau có thể được băm thành cùng một chỉ số của mảng. Hiện tượng này gọi là xung đột và giải quyết xung đột chính là mục tiêu của bất cứ bảng băm nào. Vấn đề này chúng ta sẽ đề cập đến trong phần 3 của chương này.

<u>1.2.Kích thước của bảng băm :</u>

Kích thước một bảng băm cho biết số mục vào tối đa mà bảng băm có thể lưu trữ được. Thông thường các giá trị của khoá được lưu trữ vừa đủ lấp đầy bảng nhưng đôi khi các giá trị này lại vượt quá giới hạn của mảng. Giải pháp đưa ra là buộc các khoảng giá trị này nằm trong giới hạn kích thước của bảng.

Kích thước bảng phải được lưu trữ một cách ngẫu nhiên vì các phương pháp giải quyết xung đột trong bảng băm có một số điều kiện về kích thước bảng nhất định để đảm bảo thực thi chính xác. Tuy nhiên hầu hết các trường hợp kích thước bảng băm thường được lựa chọn là luỹ thừa của 2 (2^n)hay một số nguyên tố.

Bảng băm có kích thước là luỹ thừa của 2 chỉ là một kỳ vọng lớn. Bởi vì kích thước này cho phép việc tính toán địa chỉ được thực hiện dễ dàng hơn và kết quả có được nhanh hơn. Cách để buộc các giá trị nằm trong khoảng luỹ thừa của 2 một cách nhanh chóng là sử dụng hàm mặt nạ.

Kích thước bảng băm thường được sử dụng là một số nguyên tố. Lí do là vì các phép băm nhìn chung là khó hiểu và các phép băm yêu cầu thêm các bước chia của số nguyên tố được trộn lẫn với nhau. Mặt khác một số phương pháp xử lý xung đột cũng yêu cầu kiểu kích thước này.

<u>1.3. Phân loại :</u>

Có rất nhiều loại bảng băm khác nhau. Thông thường bảng băm được phân loại theo cấu trúc hoặc theo cách xử lý xung đột.

<u>1.3.1.Phân loại theo cấu trúc :</u>

Bảng băm phân loại theo cấu trúc gồm có :

- Bảng băm chữ nhật.
- Bảng băm tam giác (tam giác trên và tam giác dưới).
- Bảng băm đường chéo.

Gọi i, j là các khoá tương ứng với phần tử hàng i, cột j. Khi đó một phần tử trong bảng băm được xác định bởi cặp i, j.

a.Bảng băm chữ nhật :

Một phần tử của bảng được xác định bởi khoá i ở hàng i và khoá j ở hàng j. Tổng quát vị trí của phần tử này có thể xác định qua công thức :

f(i,j) = n*i + j (n là số cột của bảng chữ nhật)

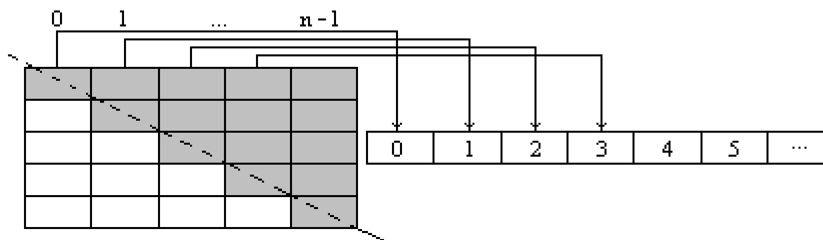
	0 - - - - - - - - - - - - - → j				
0	0	1	2	...	n
⋮	1				
⋮	2				
↓	...				
i	m				

Bảng băm hình chữ nhật được mô tả bởi một danh sách kề :

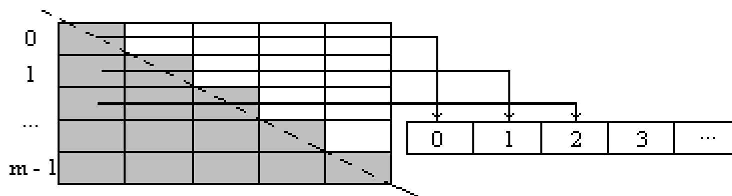
0 1 2 … n-1 n n+1 … m*n

b.Bảng băm tam giác :

- Bảng băm tam giác trên n cột:



- Bảng băm tam giác dưới m hàng:

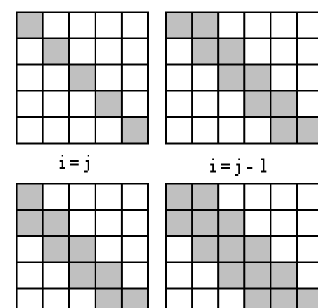


Mỗi phần tử trên bảng tam giác tương ứng với hàng i, cột j (i ≥ j) và địa chỉ của nó được xác định qua hàm băm :

f (i,j) = i*(i + 1)/2 + j

c.Bảng băm đường chéo :

Một số loại bảng băm đường chéo có dạng sau :



<u>*1.3.2.Phân loại theo cách xử lý xung đột :*</u>

Bảng băm phân loại theo cách này gồm :

- Bảng băm sử dụng phương pháp nối kết trực tiếp
- Bảng băm với phương pháp nối kết hợp nhất
- Bảng băm với phương pháp dò tuyến
- Bảng băm với phương pháp dò căn bậc 2
- Bảng băm với phương pháp băm kép

1.4.Các phép toán trên bảng băm :

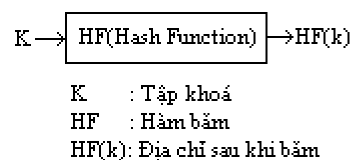
- Khởi tạo (Initialize): Khởi tạo bảng băm, cấp phát vùng nhớ, quy định số phần tử của bảng (kích thước của bảng).
- Kiểm tra rỗng (Empty): Kiểm tra liệu bảng băm có rỗng hay không.
- Lấy kích thước bảng băm (Size): Lấy số phần tử hiện thời có trong bảng băm.
- Tìm kiếm (Search): Tìm một phần tử theo một khoá k cho trước.
- Thêm mới một phần tử (Insert): Chèn thêm một phần tử vào một vị trí trống của bảng băm.
- Xoá (Delete / Removal): Loại bỏ một phần tử khỏi bảng băm.
- Sao chép (Copy): Tạo một bảng băm mới trên cơ sở một bảng băm đã có.
- Duyệt (Traverse): Duyệt các phần tử của bảng theo một thứ tự nhất định.

2.Hàm băm và các loại hàm băm :

2.1.Hàm băm (Hash Function):

Hàm băm là hàm sử dụng để ánh xạ tập các khoá đại diện cho các mục dữ liệu trong bảng thành địa chỉ nơi chứa mục dữ liệu đó.

Hình 2.3 : Mô hình hàm băm



Khoá trong bảng băm có thể là dạng số hoặc chuỗi (xâu ký tự). Nếu khoá là dạng số thì trước khi áp dụng phép băm ta phải lựa chọn các chữ số, giới hạn giá trị, áp dụng các thuật toán. Các khoá ở dạng số thường được chọn có kiểu số nguyên.

Nếu khoá ở dạng xâu ký tự thì trước khi áp dụng phép băm nó cần được biến đổi thành dạng phù hợp (Ví dụ lấy giá trị mã ASCII của các ký tự chẳng hạn), chọn lựa những phần độc lập và có ý nghĩa nhất trong khoá và lựa chọn một hàm băm phù hợp nhất với cấu trúc của khoá.

Hàm băm được chia làm hai dạng chính : Dạng bảng tra và dạng công thức.

- Hàm băm dạng bảng tra :

Giả sử có bảng tra có khoá là bộ chữ cái tiếng Anh.Bảng có 26 địa chỉ có giá trị từ 0..25. Khoá *a* ứng với địa chỉ 0, khoá *b* ứng với địa chỉ 1… khoá *z* ứng với địa chỉ 25.

Khoá	Địa chỉ	Khoá	Địa chỉ	Khoá	Địa chỉ	Khoá	Địa chỉ
a	0	h	7	o	14	v	21
b	1	i	8	p	15	w	22
c	2	j	9	q	16	x	23
d	3	k	10	r	17	y	24
e	4	l	11	s	18	z	25
f	5	m	12	t	19		
g	6	n	13	u	20		

Bảng 2.1 : Hàm băm dạng bảng tra

- Hàm băm dạng công thức : Hàm băm dạng công thức thường có dạng tổng quát là HF(*k*) trong đó *k* là khoá. Hàm băm dạng này rất đa dạng và không bị ràng buộc bởi bất cứ tiêu chuẩn nào.

2.2.Một số loại hàm băm :

Một hàm băm tốt phải thoả mãn một số điều kiện sau :

- Tính toán nhanh chóng và đơn giản.
- Các khoá phân bố đều trong bảng.
- Ít xảy ra xung đột giữa các khoá.
- Gọi P(k) là xác suất khoá k xuất hiện trong bảng. Khi đó với mỗi $i = 0, 1, \ldots, m - 1$ thì ta có :

$$\sum_{\forall k \in H(k)=i} P(k) = \frac{1}{m}$$

- Giá trị băm phải độc lập với bất cứ phần nào của dữ liệu nghĩa là nó phải phù hợp và có tính ngẫu nhiên.

Sau đây là một số hàm băm đơn giản và phổ biến.

2.2.1.Hàm băm sử dụng phương pháp chia :

Hàm băm này có các đặc điểm sau :

- Một khoá được ánh xạ vào một trong *m* ô của bảng thông qua hàm:

HF(*k*) = *k* mod *m*

Trong đó : *k* là khoá, *m* là kích thước bảng.

- Chỉ sử dụng phép chia đơn do đó tốc độ tính toán nhanh.
- Vấn đề đặt ra là phải chọn một giá trị *m* phù hợp.

o *m* chọn không tốt khi nó có một trong các giá trị sau :

+ $m = 2^P$, khi đó h(*k*) sẽ chọn cùng giá trị là *p* bit cuối của *k*.

+ $m = 10^P$, khi đó hàm băm không phụ thuộc vào tất cả các số thập phân của khoá.

+ $m = 2^P - 1$. Nếu khoá là một xâu ký tự được dịch thành các giá trị là luỹ thừa của 2, thì hai xâu có thể được băm thành cùng một giá trị địa chỉ trên bảng.

o Giá trị của *m* là tốt khi nó là một số nguyên tố và không quá gần với giá trị là luỹ thừa của 2.

• Ví dụ về cài đặt một hàm băm sử dụng phép chia :

Public Function Hash(ByVal Key As Long) As Long

Hash = Key Mod HashTableSize

End Function

2.2.2.Hàm băm sử dụng phương pháp nhân :

Phương pháp nhân có hai bước :

• Khoá *k* được nhân với hằng số A nằm trong khoảng $0 < A < 1$. Sau đó người ta sẽ sử dụng phần phân số của *k**A.

• Phần phân số nói trên được nhân với *m* sau đó lấy phần nguyên. Do đó hàm băm có dạng :

$$HF(k) = \lfloor m * (k*A \bmod 1) \rfloor$$

Trong đó : *k* là khoá, *m* là kích thước bảng, A là hằng số.

Một hàm băm sử dụng phép nhân muốn có hiệu quả cao phải lựa chọn giá trị *m* và A cho phù hợp.

• *m* thường được chọn là $m = 2^p$.

• A được chọn phụ thuộc vào đặc trưng của dữ liệu. Một giá trị A tốt được đề xuất có giá trị là :

$$A = 1/((1+\sqrt{5})/2) = (\sqrt{5}-1)/2 \approx 0.6180339887\ldots$$

• Ví dụ về cài đặt một hàm băm sử dụng phép chia :

Private Const S As Long = 64

Private Const N As Long = 1023

Public Function Hash(ByVal Key As Long) As Long

*Hash = ((K * Key) And N) \ S*

End Function

2.2.3.Hàm băm sử dụng phép nghịch đảo :

Đây là phương pháp trong đó hàm băm có dạng :

$$HF(k) = \lfloor A / (B*k + C) \rfloor \bmod m$$

Trong đó : k là khoá, m là kích thước bảng, A, B, C là các hằng số.

2.2.4.Hàm băm sử dụng phương pháp cộng xâu :

Để băm một xâu có chiều dài thay đổi, mỗi ký tự được thêm vào xâu sẽ được chia lấy dư cho 256 cho đến tận ký tự cuối cùng. Giá trị băm, nằm trong khoảng 0..255, được tính như sau :

```
Public Function Hash(ByVal S As String) As Long
Dim h As Byte
Dim i As Long
h = 0
For i = 1 to Len(S)
h = h + Asc(Mid(S, i, 1))
Next i
Hash = h
End Function
```

2.2.5.Hàm băm sử dụng phương pháp XOR xâu :

Trong các xâu thường xuất hiện một chuỗi ký tự tương tự nhau hay đảo ngữ. Do đó việc thực hiện phép XOR các Byte trong xâu sẽ giúp khắc phục hiện tượng này và giúp đạt được các giá trị băm nằm trong khoảng 0..255. Kết quả của mỗi phép XOR tạo ra một thành phần ngẫu nhiên.

```
Private Rand8(0 To 255) As Byte
Public Function Hash(ByVal S As String) As Long
Dim h As Byte
Dim i As Long
h = 0
For i = 1 To Len(S)
h = Rand8(h Xor Asc(Mid(S, i, 1)))
Next i
Hash = h
End Function
```

2.2.6.Phép băm phổ quát (Universal Hashing):

Như chúng ta thấy có nhiều loại hàm băm khác nhau. Xong chúng ta cần phải chọn được một hàm băm thích hợp để hạn chế hiện tượng xung đột giữa các khoá. Giải pháp đưa ra là sử dụng hàm băm phổ quát.

Băm phổ quát nghĩa là chúng ta chọn ngẫu nhiên một hàm băm *h* trong một tập các hàm băm ***H*** khi thuật toán bắt đầu. Hàm băm được chọn phải đảm bảo :

- Có tính chất ngẫu nhiên.
- Đảm bảo các khoá ít xảy ra xung đột.

Gọi ***H*** là tập hữu hạn các hàm băm ánh xạ một tập các khoá ***U*** thành các giá trị nằm trong khoảng $\{0, 1, \ldots, m - 1\}$. ***H*** gọi là phổ quát nếu :

• Mỗi cặp khoá riêng biệt $x, y \in U$ số hàm băm $h \in H$ cho kết quả $h(x) = h(y)$ là $|H| / m$.

• Nói cách khác với mỗi hàm băm ngẫu nhiên từ ***H*** khả năng xung đột giữa *x* và *y* ($x \neq y$) chính xác là $1/m$(*m* là kích thước bảng băm cho trước).

Tập ***H*** sẽ được xây dựng như sau :

• Chọn kích thước bảng *m* là một số nguyên tố.

• Phân tích khoá *x* thành $r + 1$ byte để *x* có dạng $x = \{x_1, x_2, ..., x_r\}$.

• Giá trị lớn nhất của chuỗi sau khi phân tích < *m*.

• Gọi $\alpha = \{\alpha_1, \alpha_2, \ldots, \alpha_r\}$ biểu thị cho một chuỗi $r + 1$ phần tử được chọn trong khoảng $\{0, 1, \ldots, m - 1\}$.

• Hàm băm $h \in H$ tương ứng được định nghĩa như sau :

$$h_\alpha(x) = \sum_{i=0}^{r} a_i x_i \mod m$$

Theo định nghĩa ở trên ***H*** có m^{r+1} phần tử.

3.Xung đột và cách xử lý xung đột :

3.1. Định nghĩa :

Xung đột trong phép băm được hiểu là trạng thái khi hai khoá khác nhau được băm thành cùng một giá trị địa chỉ. Tổng quát ta có:

$k1 \neq k2$ thì ta nói *k1* và *k2* là hai khoá xung đột khi: HF(*k1*) = HF(*k2*)

3.2.Hệ số tải (Load Factor - λ) :

Giả sử có bảng băm có kích thước *m* với *n* mục dữ liệu. Khi đó tỷ số $\lambda = n/m$ được gọi là *hệ số tải*. Hệ số tải cho biết trạng thái lấp đầy của bảng. Ví dụ một bảng băm có hệ số tải là 0.25 thì có nghĩa là bảng băm này đã sử dụng 25% kích thước bảng để lưu dữ liệu.

Hệ số tải quyết định xác suất xảy ra tương tranh của các khoá. Do đó cần phải chọn một hệ số tải thích hợp để giảm thiểu xung đột. Giá trị của hệ số tải thường được sử dụng là nhỏ hơn hoặc bằng 30%.

3.3.Một số phương pháp xử lý xung đột :

Có hai cách tiếp cận chủ yếu để giải quyết xung đột : sử dụng bảng băm địa chỉ mở và cấu trúc lại bảng băm.

Để giải quyết xung đột thông qua bảng băm địa chỉ mở người ta có các phương pháp : dò tuyến tính, dò căn bậc hai, băm kép và băm lại.

Đối với cách tiếp cận thay đổi cấu trúc bảng người ta có các phương pháp : Móc nối trực tiếp, sử dụng các Bucket.

Ngoài ra đối với trường hợp dữ liệu có kích thước lớn người ta có thể sử dụng các phương pháp băm khác như : băm lại, băm mở rộng.

Dưới đây là chi tiết về các phương pháp này.

3.3.1.Băm theo địa chỉ mở (Open-adressing hashing) :

Băm theo địa chỉ mở giải quyết xung đột bằng cách lưu tất cả các mục dữ liệu trong chính bảng băm. Phương pháp này khá thích hợp khi chúng ta có thể ước lượng được số mục vào. Khi đó chúng ta có thể có đủ các vị trí để lưu tất cả các mục trong bảng (kể cả các vị trí sử dụng để ngăn cách) và vẫn giảm được không gian lưu trữ nhiều hơn so với phương pháp móc nối.

Người ta định nghĩa một hàm băm chung cho phương pháp băm theo địa chỉ mở. Như vậy hàm băm lúc này gồm có 2 tham số : khoá k và số lần dò tìm p , trong đó $0 \leq p \leq m-1$. Tham số p sử dụng để giới hạn số lần dò và cho phép chúng ta biết khi nào thuật toán dừng.

Sau đây chúng ta xét một số phương pháp băm theo địa chỉ mở cụ thể.

3.3.1.1.Phương pháp dò tuyến tính :

Dò tuyến tính là mô hình địa chỉ mở đơn giản nhất. Phương pháp này gồm các thao tác: tìm kiếm, chèn thêm một mục dữ liệu.

Hàm băm sử dụng cho phương pháp này có dạng :

$h(k,p) = (h(k) + p) \bmod m$

❋ Thao tác tìm kiếm :

Khi xung đột xảy ra phương pháp này đơn giản là dò một vị trí trống trong bảng. Để tìm một mục dữ liệu trước hết ta phải thực hiện băm khoá của mục dữ liệu này để tìm ra chỉ số của nó trong bảng. Nếu mục dữ liệu không có tại vị trí của chỉ số mà chúng ta thu được thì chúng ta bắt đầu thực hiện dò theo tuyến tại vị trí này. Có 3 khả năng có thể xảy ra :

1. Vị trí tiếp theo có chứa mục dữ liệu và tìm kiếm kết thúc thành công.

2. Vị trí tiếp theo trống, dữ liệu không tìm thấy, quá trình tìm kiếm kết thúc không thành công.

3. Vị trí tiếp theo bị chiếm giữ nhưng các khoá lại không phù hợp vì thế vị trí tiếp theo đó sẽ được dò.

Số các vị trí cần dò trong phương pháp này phụ thuộc vào 2 yếu tố :

+ Hàm băm được chọn như thế nào.

+ Bảng đã sử dụng bao nhiêu không gian để lưu dữ liệu.

Nếu chúng ta chọn được một hàm băm thích hợp và bảng đã sử dụng khoảng 30% - 50% thì sẽ đảm bảo được số vị trí cần dò là nhỏ nhất có thể.

Chúng ta có một ví dụ về cách cài đặt thao tác tìm kiếm đó là :

```
int jsw_find ( void *key, int len )
{
  unsigned h = hash ( key, len ) % N;
  void *save = table[h];
  while ( table[h] != NULL ) {
    if ( compare ( key, table[h] ) == 0 )
      return 1;
    h = ( h + 1 ) % N;
    if ( compare ( table[h], save ) == 0 )
      return 0;
  }
  return 0;
}
```

❈ Thao tác chèn :

Để chèn thêm một mục mới chúng ta cần thực hiện :

- Tính các giá trị băm cho các khoá thông qua hàm băm đã chọn.
- Nếu vị trí có giá trị băm đã có dữ liệu thì thao tác dò được thực hiện từ vị trí này. Thao tác dò được thực hiện cho đến khi tìm được một vị trí trống. Thao tác này sẽ dò tiếp ở vị trí đầu nếu nó đạt đến vị trí cuối của tuyến.
- Khi tìm được một ô trống thì mục dữ liệu sẽ được chèn vào.

Thao tác này có thể cài đặt như sau :

```
void jsw_insert ( void *key, int len )
{
  unsigned h = hash ( key, len ) % N;
  while ( table[h] != NULL )
    h = ( h + 1 ) % N;
  table[h] = key;
}
```

❈ Thao tác xoá :

Thao tác nay không đơn giản như hai thao tác trên. Việc xoá trực tiếp một mục khỏi bảng là khôn thể vì các phép dò tiếp theo đó có thể nhận ra các khoá đã bị bỏ đi và nếu một

bucket rỗng được tạo ra trong khi một buket khác vẫn đầy thì quá trính tìm kiếm có thể không chính xác. Như vậy thao tác xoá có thể phá vỡ cấu trúc dữ liệu của bảng. Giải pháp đưa ra là khi xoá một khoá trên một đoạn của bucket thì ta lại chèn khoá vào đoạn tương tự của nó. Nhưng cách này dường như khá phức tạp.

Sau đây là một ví dụ về thao tác xoá :

```
void jsw_remove ( void *key, int len )
{
  unsigned h = hash ( key, len ) % N;
  while ( table[h] != NULL )
    h = ( h + 1 ) % N;
  table[h] = DELETED;
}
```

❈ Đánh giá :

Trong phương pháp này các khoá có khuynh hướng bị đưa vào các đoạn gọi là Cluster (bó cụm). Điều này có nghĩa là nhiều phần trong bảng có thể đầy lên nhanh chóng trong khi các phần khác vẫn còn trống. Do phương pháp này cần sử dụng một lượng lớn các Bucket rỗng nằm xen kẽ với các Bucket đã sử dụng nên việc bó cụm sẽ làm cho nhiều Bucket bị duyệt qua trước khi tìm được một Bucket rỗng. Vì vậy thao tác tìm kiếm sẽ bị chậm đi và kéo theo các thao tác chèn và xoá cũng chậm. Một bảng băm có hệ số tải càng lớn thì khả năng bó cụm xảy ra càng lớn. Do đó một hàm băm tốt và kích thước bảng là một số nguyên tố sẽ cải thiện được vấn đề này.

3.3.1.2.Phương pháp dò căn bậc 2 :

Để khắc phục vấn đề bó cụm chính người ta đưa ra phương pháp dò căn bậc hai. Phương pháp này sử dụng hàm băm có dạng :

$h(k,p) = (h(k) + c_1 p + c_2 p^2) \bmod m$

Các giá trị c_1, c_2, m xác định liệu toàn bộ bảng có được sử dụng hay không.

❈ Thao tác tìm kiếm :

Theo hàm băm như trên để tìm kiếm một mục trong bảng người ta sẽ bắt đầu từ vị trí đầu tiên trong bảng được xác định bởi hàm băm, gọi là vị trí i và tiếp tục dò tới các vị trí $i + 1^2$, $i + 2^2$, …, $i + (m - 1)^2$ (tất cả đều mod m). Cứ như vậy quá trình tìm kiếm được thực hiện cho đến khi tìm thấy mục dữ liệu trong bảng (kết thúc thành công) hoặc gặp một vị trí trống (kết thúc không thành công).

Thuật toán sử dụng cho phương pháp này có phần phức tạp hơn phương pháp dò tuyến tính. Dưới đây là ví dụ cụ thể :

```
int jsw_search ( void *key, int len )
{
  unsigned h = hash ( key, len ) % N;
  unsigned step;
  for ( step = 1; table[h] != NULL; ++step ) {
   if ( compare ( key, table[h] ) == 0 )
     return 1;
   h = ( h + step * step ) % N;
  }
  return 0;
}
```

❋ Thao tác chèn :

```
void jsw_insert ( void *key, int len )
{
  unsigned h = hash ( key, len ) % N;
  unsigned step;
  for ( step = 1; table[h] != NULL; ++step )
   h = ( h + step * step ) % N;
  table[h] = key;
}
```

❋ Thao tác xoá :

```
void jsw_remove ( void *key, int len )
{
  unsigned h = hash ( key, len ) % N;
  unsigned step;
  for ( step = 1; table[h] != NULL; ++step )
   h = ( h + step * step ) % N;
  table[h] = DELETED;
}
```

❋ Đánh giá :

Phương pháp dò theo căn bậc hai giảm đáng kể hiện tượng bó cụm chính. Tuy nhiên vì chuỗi dò tìm luôn bắt đầu ở cùng bucket (một ô của bảng) nên chúng ta lại gặp phải hiện tượng bó cụm thứ cấp (Secondary Clustering). Đây không phải là một hiện tượng đáng quan tâm như bó cụm chính. Nhưng do phương pháp dò căn bậc hai chỉ hoạt động khi hệ số tải <

0.5 và kích thước của bảng là một số nguyên tố nên hiện tượng này lại làm chậm đáng kể tốc độ tìm kiếm.

Nói chung phương pháp này nhanh và tránh được hiện tượng bó cụm chính nhưng lại ít được sử dụng trong thực tế vì sự giới hạn về thời gian. Phương pháp này chỉ đảm bảo hoạt động hiệu quả khi kích thước bảng là số nguyên tố và dung lượng bảng đã sử dụng chưa quá một nửa.

3.3.1.3.Phương pháp băm kép :

Phương pháp này là một giải pháp đáng lưu ý thay cho phương pháp dò theo căn bậc hai. Nó có thể khắc phục được hiện tượng bó cụm chính mà không chịu sự giới hạn nào.

Phương pháp này sử dụng hai hàm băm độc lập nhau. Hàm băm thứ nhất được sử dụng như bình thường (theo một trong hai phương pháp đã kể trên). Hàm băm thứ hai được sử dụng để tạo ra kích thước bước dò. Do bản thân khoá có thể xác định được bước dò nên tránh được hiện tượng bó cụm chính. Phương pháp này có thuật toán khá đơn giản nhưng có 2 điều cần chú ý để đảm bảo phương pháp này hoạt động chính xác :

- Hàm băm thứ hai không được trả về giá trị 0 để tránh hiện tượng lặp vô hạn.
- Giống như dò theo tuyến, kích thước bảng phải là một số nguyên tố hoặc là một giá trị luỹ thừa của 2 với kết quả hàm băm thứ hai trả về là một số lẻ.

Như vậy cách cài đặt cho phương pháp này cũng tương tự như các phương pháp băm theo địa chỉ mở khác. Một ví dụ minh hoạ cụ thể là :

❈ Thao tác tìm kiếm :

```
int jsw_search ( void *key, int len )
{
  unsigned h = hash ( key, len ) % N;
  unsigned h2 = hash2 ( key ) % ( N - 1 ) + 1;
  while ( table[h] != NULL ) {
    if ( compare ( key, table[h] ) == 0 )
      return 1;
    h = ( h + h2 ) % N;
  }
  return 0;
}
```

❈ Thao tác chèn :

```
void jsw_insert ( void *key, int len )
{
```

```
  unsigned h = hash ( key, len ) % N;
  unsigned h2 = hash2 ( key ) % ( N - 1 ) + 1;
  while ( table[h] != NULL )
   h = ( h + h2 ) % N;
  table[h] = key;
}
```

❋ Thao tác xoá :

```
void jsw_remove ( void *key, int len )
{
  unsigned h = hash ( key, len ) % N;
  unsigned h2 = hash2 ( key ) % ( N - 1 ) + 1;
  for ( table[h] != NULL )
   h = ( h + h2) % N;
  table[h] = DELETED;
}
```

<u>3.3.2.Cấu trúc lại bảng băm :</u>

Phương pháp này giải quyết xung đột bằng cách thay đổi cấu trúc của bảng băm. Theo đó một vị trí có thể chứa được nhiều hơn một mục dữ liệu. Sau đây là một số phương pháp cấu trúc lại bảng băm.

3.3.2.1.Sử dụng Bucket :

Trong phương pháp này bản thân mỗi phần tử trong bảng băm là một mảng được gọi là ngăn chứa(Bucket). Các khoá được băm về cùng một vị trí sẽ được đặt trong các cột tương ứng với các vị trí này. Ví dụ :

0	1	2	3	4	5	6	7	8	9	10
	K			k3	k5	k5	k7	k8		
	K			k4				k9		

Bảng 2.2 : Mô hình một Bucket

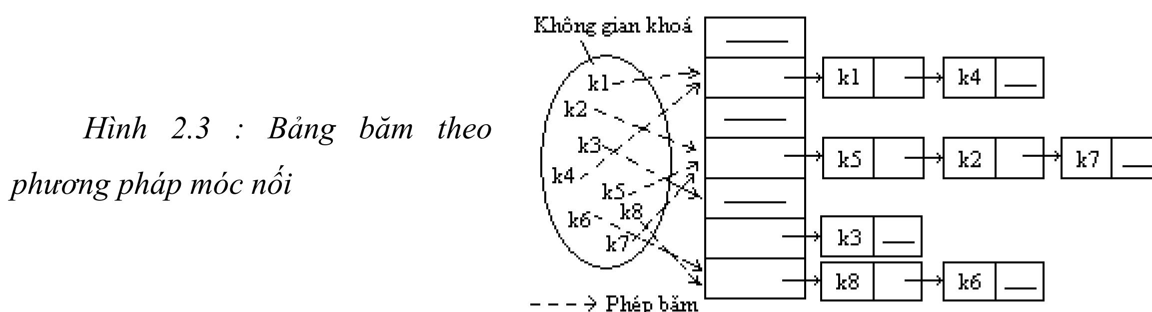
Giả sử có hai khoá *k*1, *k*2 là hai khoá xung đột. Kết quả sau khi băm hai khoá này là 2. Vậy ta xếp *k*1 và *k*2 vào cột 1 tương ứng với vị trí thứ 2. Tương tự như vậy nếu kết quả băm các khoá khác cho vị trí 2 thì các khoá sẽ lần lượt được xếp vào cột này.

Như vậy có thể thấy phương pháp này cần nhiều không gian lưu trữ hơn thực tế dữ liệu yêu cầu.

Trong phương pháp này, để tìm một khoá nhất định người ta phải kiểm tra tất cả các cột liên quan tới vị trí của khoá này (ít nhất là cho đến khi chúng ta tìm được một vị trí trống). Nếu các khoá có thứ tự, chúng ta nên lưu các khoá theo trật tự tăng dần. Như vậy chúng ta có thể loại bớt các cột cần kiểm tra. Độ phức tạp trung bình của thao tác tìm kiếm phụ thuộc vào số vị trí trong Bucket đã được điền. Do đó khi nào bảng chưa được điền đầy khi đó khả năng xung đột vẫn có thể xảy ra.

3.3.2.2.Phương pháp móc nối trực tiếp :

Theo phương pháp này bảng băm là một mảng các danh sách liên kết. Phương pháp này đặt các khoá bị xung đột vào cùng một danh sách liên kết tại vị trí là kết quả sau khi băm. Một bảng băm theo phương pháp này có thể mô tả như sau :



Hình 2.3 : Bảng băm theo phương pháp móc nối

Vấn đề đặt ra ở đây là danh sách liên kết được cấu trúc như thế nào. Khi việc tìm kiếm của bảng băm phụ thuộc vào việc tìm kiếm trên danh sách liên kết thì một danh sách được sắp xếp là một cấu trúc đáng lưu ý. Cấu trúc này có hiệu quả ít hơn với thao tác chèn và thao tác tìm kiếm thành công hoặc không thành công (tính trung bình). Khi đó một nửa số danh sách liên kết sẽ được tìm kiếm. Nếu danh sách không được sắp xếp thì một lần tìm kiếm không thành công có thể duyệt qua toàn bộ các danh sách liên kết.

Tuy nhiên điều nói trên chỉ đúng về mặt lý thuyết. Trên thực tế còn có nhiều thao tác hơn cần thiết. Vì với một hàm băm tốt các chuỗi (danh sách liên kết) phải đảm bảo đủ ngắn để thấy được sự khác nhau giữa danh sách được sắp xếp và danh sách chưa được sắp xếp.

Chúng ta giả sử rằng hầu hết thời gian thực hiện phương pháp này sử dụng để thực hiện so sánh giữa khoá tìm kiếm với các khoá khác trong danh sách. Chúng ta cũng giả sử thời gian cần thiết để băm tới danh sách thích hợp và xác định khi nào đến vị trí cuối của danh sách tương đương với một so sánh.

Như vậy tất cả các thao tác cần 1 + Số so sánh (lần).

Giả sử bảng băm có kích thước m và có n mục được chèn vào bảng. Khi đó mỗi danh sách trung bình sẽ chứa n/m mục. Ở đây ta có thể thấy số mục trung bình trên một danh sách chính là hệ số tải $\lambda = n/m$.

Sau đây là một số thao tác chính của phương pháp này.

❈ Thao tác tìm kiếm :

Khi tìm một khoá trong bảng, khoá sẽ được băm và trả về kết quả là chỉ số nơi có danh sách liên kết sẽ được tìm kiếm. Như vậy hàm băm sẽ quyết định phải tìm ở danh sách nào.

Khi tìm kiếm thành công thuật toán sẽ thực hiện băm và tìm tới danh sách thích hợp. Tính trung bình thì một nửa số mục sẽ được kiểm tra trước khi một mục chính xác được tìm thấy. Do đó thời gian tìm kiếm là : $1 + \lambda/2$.

Trong trường hợp tìm kiếm không thành công, nếu danh sách không có trật tự thì tất cả các mục sẽ được kiểm tra và thời gian tìm kiếm là : $1 + \lambda$.

Sau đây là cài đặt của thao tác này :

```
struct jsw_node {
  void *key;
  struct jsw_node *next;
};
struct jsw_node *table[N];
int jsw_find ( void *key, int len )
{
  unsigned h = hash ( key, len ) % N;
  if ( table[h] != NULL ) {
    struct jsw_node *it = table[h];

    while ( it != NULL ) {
      if ( compare ( key, it->key ) == 0 )
        return 1;
      it = it->next;
    }
  }
  return 0;
}
```

❈ Thao tác chèn :

Chèn vào một danh sách liên kết đơn giản hơn tìm kiếm. Khi cần chèn thêm một mục dữ liệu vào bảng thì khoá sẽ được băm và cho kết quả là vị trí của danh sách cần chèn. Một nút sẽ được thêm vào danh sách (tạo phần header của nút nếu cần thiết) sau đó khoá và giá trị sẽ được đẩy vào phần trước của nút này.

Việc chèn một nút vào cuối danh sách liên kết khá đơn giản. Nếu một nút được chèn vào phía đầu của danh sách liên kết thì danh sách liên kết sẽ được sử dụng như một ngăn xếp với tính chất truy cập tuần tự. Trên thực tế chèn một nút vào trước danh sách là cách đơn giản và đạt hiệu quả tốt nhất.Ví dụ về cài thao tác này là :

```
int jsw_insert ( void *key, int len )
{
  unsigned h = hash ( key, len ) % N;
  struct jsw_node *it = malloc ( sizeof *it );
  if ( it == NULL )
    return 0;
  it->key = key;
  it->next = table[h];
  table[h] = it;
  return 1;
}
```

❋ Thao tác xoá :

Khi xoá một mục dữ liệu thì cần thực hiện thao tác tìm khoá cũ.

```
int jsw_remove ( void *key, int len )
{
  unsigned h = hash ( key, len ) % N;
  struct jsw_node *save;
  if ( table[h] == NULL )
    return 0;
  if ( compare ( key, table[h]->key ) == 0 ) {
    save = table[h]->next;
    table[h]->next = table[h]->next;
    free ( save );
  }
  else {
    struct jsw_node *it = table[h];
      while ( it->next != NULL ) {
     if ( compare ( key, it->next->key ) == 0 )
       break;
```

```
   it = it->next;
  }
  if ( it->next == NULL )
   return 0;
  save = it->next;
  it->next = it->next->next;
  free ( save );
 }
 return 1;
}
```

❋ Đánh giá :

Phương pháp móc nối trực tiếp là phương pháp xử lý xung đột phổ biến nhất vì nó khá linh hoạt. Nội dung của phương pháp này đơn giản và nó có một số ưu điểm sau :

- Giới hạn kích thước bảng không quá cứng nhắc.
- Việc thực thi tránh tối đa hiện tượng xung đột.
- Bảng dễ dàng xử lý các khoá trùng lặp.
- Thao tác xoá đơn giản và có thể thực hiện thường xuyên.

Tuy nhiên việc cài đặt phương pháp này phức tạp hơn các phương pháp khác. Nó cũng có một số nhược điểm :

- Phải xây dựng lại bảng khi việc thay đổi kích thước gặp khó khăn.
- Sử dụng nhiều không gian nhớ hơn cho các liên kết.
- Tốc độ xử lý có thể chậm hơn do sự tham chiếu qua lại của các liên kết.

3.3.3.Các phương pháp băm khác :

❖ *Băm lại :*

Chúng ta thấy rằng tất cả các phương pháp trên chỉ hiệu quả khi hệ số tải < 2/3, tức là bảng không quá đầy. Khi bảng băm bị lấp đầy các phương pháp này không còn hiệu quả nữa. Giải pháp đưa ra là tạo ra một bảng mới có kích thước lớn gấp đôi bảng ban đầu. Sau đó tất cả các khoá sẽ được băm lại.

Độ phức tạp của phương pháp này là $\boldsymbol{O}(n)$ do đó phương pháp này chỉ sử dụng khi kích thước dữ liệu lớn. Cá biệt trong phương pháp dò theo căn bậc hai chúng ta phải băm lại khi ½ bảng được sử dụng.

❖ *Băm mở rộng :*

Đây là một phương pháp hỗ trợ hữu hiệu cho việc sử dụng có hiệu quả không gian lưu trữ.

3.4. Đánh giá :

Khi xây dựng bảng băm nếu chúng ta biết trước số mục dữ liệu sẽ được chèn vào bảng thì mô hình băm theo địa chỉ mở với hệ số tải < ½ sẽ cho kết quả tốt nhất.

Mặt khác nếu chúng ta không biết trước có bao nhiêu mục dữ liệu được chèn vào bảng thì phương pháp móc nối trực tiếp có hiệu quả hơn.

Khi hệ số tải tăng sử dụng phương pháp băm theo địa chỉ mở sẽ gặp những bất lợi chủ yếu trong thực thi. Nhưng đối với phương pháp móc nối hiệu quả của nó chỉ bị giảm tuyến tính khi hệ số tải tăng. Do đó nếu không chắc chắn về hệ số tải thì phương pháp được lựa chọn là phương pháp móc nối.

4.Kết luận :

Bảng băm là một cấu trúc dữ liệu thích hợp cho việc lưu trữ và tìm kiếm dữ liệu không có trật tự. Tuy phải xử lý xung đột nhưng hầu hết các thao tác của bảng băm đều dễ thực hiện và các thao tác này có hiệu quả gần như không đổi trong hầu hết các trường hợp.

Hàm băm ánh xạ các khoá dữ liệu vào trong các ô có sẵn của bảng. Một bảng băm có hiệu quả khi chúng ta chọn được một hàm băm tốt. Vấn đề cốt yếu là :

- Sử dụng bảng băm có kích thước là một số nguyên tố.
- Sử dụng hàm băm có thể đảm bảo các khoá được phân bố đều trong bảng.Tuy nhiên ngay cả trong trường hợp chúng ta chọn được hàm băm tốt nhất thì hiện tượng xung đột vẫn có thể xảy ra.

Băm theo địa chỉ mở là một trong các phương pháp giải quyết xung đột. Trong đó :

- Phương pháp dò tuyến tính : sử dụng nếu các ô được duyệt toàn bộ và tìm vị trí thành công. Nhưng phương pháp này có thể gây ra hiện tượng bó cụm.
- Phương pháp dò căn bậc 2 : Phương pháp này tốt hơn do có thể giảm xác suất xảy ra hiện tượng bó cụm.
- Hệ số tải nên < ½ để tránh làm giảm hiệu quả thực hiện của phương pháp.

Móc nối cũng là một biện pháp xử lý xung đột hiệu quả. Xong phương pháp này đòi hỏi không gian lưu trữ lớn.

11.33.8. TÀI LIỆU THAM KHẢO.

1. Đinh Mạnh Tường, Cấu trúc dữ liệu và thuật toán, NXB Đại học Quốc gia Hà nội, 2002
2. Đỗ Xuân Lôi, Cấu trúc dữ liệu và giải thuật, NXB Khoa học và Kỹ thuật, 1997
3. Nguyễn Quốc Lượng, Hoàng Đức Hải ,Cấu trúc dữ liệu + giải thuật = chương trình, NXB Giáo dục, 1996
4. Hoare, C.A.R, Note on date Structuring in structured Programming Dahl, Dijkstra, and Hoare, pp 83-174.
5. Robert Sedgewick , Cẩm năng thuật toán, NXB KH Kỹ thuật, 2000