

TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT TP. HỒ CHÍ MINH
KHOA CÔNG NGHỆ THÔNG TIN

CẤU TRÚC DỮ LIỆU 1

Biên soạn: ThS. Lê Văn Vinh

Thành phố Hồ Chí Minh, 2009

LỜI NÓI ĐẦU

Cấu trúc dữ liệu và giải thuật là một trong những khối kiến thức cơ sở trong chương trình đào tạo cử nhân ngành Công nghệ Thông tin của Khoa Công nghệ Thông tin, trường Đại học Sư phạm Kỹ thuật Thành phố Hồ Chí Minh. Khối kiến thức này được giảng dạy trong hai môn học: **Cấu trúc dữ liệu 1** và **Cấu trúc dữ liệu 2**. Cấu trúc dữ liệu 1 cung cấp cho người học những kiến thức về các kiểu dữ liệu trừu tượng cơ bản thường được sử dụng khi xây dựng chương trình trên máy tính, cách hiện thực và áp dụng các kiểu dữ liệu đó trong thực tế.

Nội dung của bài giảng bao gồm 7 chương bao quát hầu hết các vấn đề cốt lõi của môn học. Nội dung trong mỗi chương được trình bày theo trình tự: trình bày các khái niệm, các kiến thức cơ bản trước, tiếp theo là nêu những ví dụ minh họa để người đọc dễ dàng tiếp cận lý thuyết mới và sau cùng là cài đặt giải thuật bằng ngôn ngữ lập trình cụ thể. Cuối mỗi chương đều có bài tập để người đọc tự kiểm tra và củng cố lại kiến thức của mình.

Rất mong nhận được những ý kiến đóng góp của các đồng nghiệp và các bạn sinh viên để bài giảng ngày càng hoàn thiện hơn. Mọi ý kiến đóng góp xin vui lòng gửi theo địa chỉ email: levinhcntt@gmail.com.

CHƯƠNG 1. TỔNG QUAN

Chương này trình bày những khái niệm cơ bản liên quan đến cấu trúc dữ liệu, vai trò và ý nghĩa của cấu trúc dữ liệu trong việc xây dựng một chương trình trên máy tính, và cách để đánh giá độ phức tạp của giải thuật.

I. Từ bài toán đến chương trình

Máy tính là công cụ giúp con người giải quyết một vấn đề, một bài toán trong thực tế. Khi viết một chương trình yêu cầu máy tính thực hiện công việc nào đó, chúng ta cần phải tiến hành theo các bước cơ bản sau:

1. Xác định bài toán

Ở bước này, ta xác định xem cần phải giải quyết vấn đề gì. Những giả thiết nào đã cho và ta cần phải đạt được những yêu cầu gì. Việc xác định đúng yêu cầu của bài toán là rất quan trọng vì nó ảnh hưởng đến cách thức giải quyết và tính hiệu quả của lời giải. Thông tin lấy từ một bài toán thực tế thường là mơ hồ và hình thức, ta cần phải phát biểu lại một cách rõ ràng và chính xác để có thể xây dựng thuật toán để giải quyết.

Ví dụ bài toán tô màu bản đồ như sau: “Hãy sử dụng số màu tối thiểu để tô màu cho bản đồ thế giới sao cho sao cho mỗi nước được tô một màu và hai nước láng giềng (cùng biên giới) của nhau thì phải được tô bằng hai màu khác nhau”.

Ta xem mỗi nước trên bản đồ là một đỉnh của đồ thị. Khi hai nước là láng giềng của nhau thì hai đỉnh đại diện được nối với nhau bằng một cạnh. Khi đó bài toán thực tế sẽ được chuyển thành bài toán về đồ thị như sau: “Cho một đồ thị có n đỉnh. Hãy sử dụng số màu tối thiểu để tô cho các đỉnh của đồ thị sao cho hai đỉnh kề nhau phải có màu khác nhau”.

Từ đây, ta có thể áp dụng các thuật toán của lý thuyết đồ thị để giải quyết bài toán một cách dễ dàng.

2. Xây dựng cấu trúc dữ liệu

Khi giải một bài toán, ta cần phải định nghĩa tập hợp dữ liệu để biểu diễn một cách đầy đủ những thông tin có trong thực tế của bài toán đó. Dữ liệu trong thực

tế luôn đa dạng, phong phú và thường chứa đựng những mối quan hệ nào đó với nhau. Việc lựa chọn *cách biểu diễn dữ liệu* hay *cấu trúc dữ liệu* phải tùy thuộc vào những yêu cầu của vấn đề ta cần giải quyết và những thao tác sẽ tiến hành trên dữ liệu đầu vào. Có những giải thuật chỉ phù hợp với một cách tổ chức dữ liệu nhất định, còn với cách tổ chức dữ liệu khác thì sẽ kém hiệu quả hoặc không thể thực hiện được. Vì vậy, khi lựa chọn cấu trúc dữ liệu cần phải đảm bảo những tiêu chuẩn sau:

- Phản ánh đúng và đầy đủ thông tin thực tế: Tiêu chuẩn này quyết định tính đúng đắn của toàn bộ chương trình.
- Phù hợp với các thao tác trên dữ liệu mà ta lựa chọn để giải quyết bài toán.
- Cấu trúc dữ liệu phải cài đặt được trên máy tính với các ngôn ngữ lập trình hiện có.

3. Thiết kế giải thuật

Giải thuật là một hệ thống chặt chẽ và rõ ràng các quy tắc nhằm xác định một dãy các thao tác trên cấu trúc dữ liệu sao cho: Với một bộ dữ liệu vào, sau một số hữu hạn bước thực hiện các thao tác đã chỉ ra, ta đạt được mục tiêu đã định. Tùy theo những yêu cầu thực tế mà người viết chương trình cần áp dụng các giải thuật phù hợp để giải quyết. Giải thuật sử dụng phải tương ứng với cấu trúc dữ liệu của bài toán. Hay nói một cách khác, giải thuật và cấu trúc dữ liệu có mối quan hệ chặt chẽ với nhau.

Donald Knuth đã trình bày một số tính chất quan trọng của giải thuật là:

- Tính hữu hạn (finiteness): Giải thuật phải luôn kết thúc sau một số hữu hạn bước.
- Tính xác định (definiteness): Mỗi bước của giải thuật phải được xác định rõ ràng và nhất quán.
- Tính hiệu quả (effectiveness): Giải thuật phải được thực hiện trong một khoảng thời gian chấp nhận được.

4. Lập trình

Khi đã xác định được thuật toán và cấu trúc dữ liệu sẽ sử dụng, ta sẽ tiến hành lập trình để hiện thực chúng. Thông thường, khi lập trình, ta không nên cụ thể hóa ngay toàn bộ chương trình mà nên tiến hành theo phương pháp tinh chế từng bước (stepwise refinement. Theo Niklaus Wirth):

- Ban đầu, chương trình nên được thể hiện bằng ngôn ngữ tự nhiên, hay bằng mô hình hóa.
- Những công việc nào đơn giản hoặc có thể cài đặt ngay thì tiến hành viết mã nguồn cho nó.
- Những công việc phức tạp thì chia thành những công việc nhỏ hơn để tiếp tục thực hiện với những công việc nhỏ đó.

5. Kiểm lỗi và sửa lỗi chương trình

Một chương trình viết xong chưa chắc đã có thể chạy được trên máy tính và cho ra kết quả mong muốn. Vì vậy, kỹ năng tìm lỗi, sửa lỗi cũng là một kỹ năng quan trọng của người lập trình. Thông thường, có ba loại lỗi phát sinh trong lập trình là:

- Lỗi cú pháp: Đây là loại lỗi đơn giản và dễ phát hiện nhất. Lỗi này thường là đã được các môi trường soạn thảo tự động tìm và báo cho người lập trình biết.
- Lỗi cài đặt: Đây là loại lỗi mà người lập trình không cài đặt theo đúng thuật toán đã xây dựng, thiết kế chương trình không đúng. Đối với loại lỗi này, cần phải xem xét lại bố cục chương trình, kết hợp với các chức năng sử lỗi của trình soạn thảo để tìm và sửa.
- Lỗi thuật toán: Đây là loại lỗi nghiêm trọng nhất, vì đã bị sai từ bước thiết kế giải thuật hoặc lựa chọn cấu trúc dữ liệu không phù hợp. Đối với loại lỗi này, thường là ta chỉ phát hiện khi chương trình chạy được nhưng cho ra kết quả không đúng.

❖ Kết luận

Trong phần này, chúng ta đã tìm hiểu các giai đoạn của việc xây dựng chương trình để giải quyết một bài toán, một vấn đề trong thực tế bằng máy tính. Rõ ràng, giai đoạn thiết kế giải thuật và lựa chọn cấu trúc dữ liệu đóng vai trò hết sức quan trọng. Bên cạnh đó hai giai đoạn này luôn đi song hành với nhau. Lựa chọn cấu trúc dữ liệu nào phải dựa vào giải thuật sẽ sử dụng, và ngược lại ta chỉ được phép sử dụng các giải thuật mà có thể áp dụng trên cấu trúc dữ liệu đã lựa chọn. Trong môn học này, chúng ta sẽ tìm hiểu các cấu trúc dữ liệu thông dụng và các giải thuật tương ứng với từng kiểu dữ liệu đó.

II. Kiểu dữ liệu

1. Định nghĩa kiểu dữ liệu

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 số nguyên trong ngôn ngữ C như int có giá trị trong khoảng -32768 đến 32767, và các phép toán trên nó như: + (cộng), - (Trừ), * (Nhân), / (Chia), % (Chia lấy dư).

+ Kiểu Boolean trong ngôn ngữ Visual C++ có tập giá trị là {TRUE, FALSE}, và các phép toán logic như: && (And), || (Or), ! (Not), ^ (Xor).

Kiểu dữ liệu có hai loại là kiểu dữ liệu cơ bản và kiểu dữ liệu có cấu trúc.

2. Các kiểu dữ liệu cơ bản

Sau đây ta tìm hiểu một số kiểu dữ liệu cơ bản chuẩn được cài đặt rộng rãi trên hầu hết các ngôn ngữ lập trình.

a. Kiểu số nguyên

Kiểu số nguyên bao gồm một tập các số mà miền giá trị của nó là khác nhau trên các hệ thống máy tính khác nhau. Nếu một máy tính sử dụng n bit để biểu diễn một số nguyên thì giá trị của một biến số nguyên x phải thỏa điều kiện $-2^{n-1} \leq x < 2^{n-1}$. Ví dụ, trong ngôn ngữ lập trình C, kiểu số nguyên được cài đặt là kiểu int (2 byte) có phạm vi là (-32768...32767), kiểu long (4 byte) có phạm vi là $(-2^{31} \dots 2^{31}-1)$. Các thao tác trên kiểu dữ liệu số nguyên trả về giá trị chính xác và

tuân theo luật của số học. Các thao tác chuẩn trên kiểu số nguyên là: Cộng, Trừ, Nhân, Chia, Phép lấy dư, lũy thừa, ...

b. Kiểu số thực

Trong ngôn ngữ C, kiểu số thực có thể được cài đặt là kiểu float, double hay long double. Trong khi các thao tác trên kiểu số nguyên luôn chắc chắn trả về giá trị chính xác thì đối với kiểu số thực, giá trị trả về có thể chỉ là giá trị gần đúng bằng cách làm tròn giá trị. Các thao tác chuẩn trên kiểu số thực là: Cộng, Trừ, Nhân, Chia.

c. Kiểu boolean

Hai giá trị chuẩn của kiểu logic là TRUE và FALSE. Các thao tác trên kiểu boolean là các phép toán logic như: AND, OR, XOR, NOT. Trong ngôn ngữ C chuẩn, không có kiểu boolean, nhưng người ta có thể dùng kiểu int với giá trị 0 và 1 để sử dụng trong các biểu thức điều kiện.

d. Kiểu ký tự

Kiểu ký tự là kiểu biểu diễn các ký tự có thể in ra được. Nó được xác định trên các bảng mã. Bảng mã chuẩn hiện nay thường dùng trên các hệ thống máy tính là bảng mã ASCII và bảng mã UNICODE.

3. Các kiểu dữ liệu có cấu trúc

Thực tế, các kiểu dữ liệu cơ bản không đủ để phản ánh một cách tự nhiên và đầy đủ thông tin trong thực tế. Vì vậy, người ta định nghĩa các kiểu dữ liệu có cấu trúc dựa trên các kiểu dữ liệu cơ bản như: Kiểu chuỗi ký tự (string), kiểu mảng (array), kiểu cấu trúc (struct), kiểu tập hợp (union).

4. 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. Có thể nói, kiểu dữ liệu trừu tượng là một kiểu dữ liệu ở mức khái niệm, nó chưa được cài đặt cụ thể bằng một ngôn ngữ lập trình.

Ví dụ: tập hợp số nguyên cùng với các phép toán hợp, giao, hiệu là một kiểu dữ liệu trừu tượng.

Trong phạm vi của môn học này, chúng ta sẽ nghiên cứu các kiểu dữ liệu trừu tượng như: Danh sách (list), ngăn xếp (stack), hàng đợi (queue) hay cây nhị phân (binary tree). Các kiểu dữ liệu trừu tượng này cùng với các phép toán trên nó sẽ được hiện thực bằng một ngôn ngữ lập trình C hoặc C++.

III. Đánh giá độ phức tạp của giải thuật

Trước khi bắt tay vào viết một chương trình để giải quyết một vấn đề nào đó, ta cần phải xác định giải thuật sẽ sử dụng. Một bài toán có nhiều giải thuật khác nhau để giải quyết, ta cần phải chọn giải thuật nào phù hợp nhất, đưa đến kết quả nhanh nhất. Vậy phải dựa vào tiêu chí nào để đánh giá một giải thuật là tốt hay xấu, là phù hợp hay không phù hợp?

Thời gian thực hiện một giải thuật bằng máy tính phụ thuộc vào rất nhiều yếu tố như: Phần cứng máy tính, ngôn ngữ lập trình sử dụng, trình biên dịch, ... Những yếu tố này ảnh hưởng đến tốc độ xử lý một cách không rõ ràng, và ta cũng không thể có một thước đo hợp lý dựa vào các yếu tố đó để đánh giá giải thuật. Vì vậy, người ta dựa vào số lần tính toán hay so sánh của giải thuật để đánh giá giải thuật nào tốt hơn. Sau đây là một số tính chất về độ phức tạp tính toán mà ta có thể tham khảo để áp dụng cho việc đánh giá thuật toán

- Khi thuật toán có độ phức tạp là hằng số, tức là độ phức tạp không phụ thuộc vào kích thước của dữ liệu. Khi đó ta ký hiệu là $O(1)$.
- Khi số phép tính toán của thuật toán có cấp là 2^n , $n!$, hay n^n ta gọi chung là độ phức tạp hàm mũ.
- Với $P(n)$ là một đa thức bậc k thì $O(P(n)) = O(n^k)$. Vì vậy, một thuật toán có độ phức tạp cấp đa thức, ta ký hiệu là $O(n^k)$.
- Với a và b là hai cơ số tùy ý, $f(n)$ là một hàm dương thì $\log_a f(n) = \log_a b \cdot \log_b f(n)$. Tức là $O(\log_a f(n)) = O(\log_b f(n))$. Khi thuật toán có độ phức tạp cấp logarit của $f(n)$, ta ký hiệu là $O(\log f(n))$ mà không cần quan tâm đến cơ số của nó.

IV. Bài tập chương 1

-
1. Phân biệt kiểu dữ liệu và kiểu dữ liệu trừu tượng. Kiểu dữ liệu cơ bản và kiểu dữ liệu có cấu trúc. Cho ví dụ minh họa.
 2. Liệt kê các kiểu dữ liệu cơ bản được cung cấp sẵn trong các ngôn ngữ lập trình Pascal, C++, Java, C#.
 3. Viết chương trình giải phương trình bậc hai: $ax^2+bx+c=0$.
 4. Viết chương trình nhập vào một mảng n số nguyên dương. Viết các hàm thực hiện các công việc sau:
 - + Tìm phần tử lớn nhất, nhỏ nhất trên mảng
 - + Tính tổng các phần tử của mảng
 - + Tìm phần tử âm đầu tiên trong mảngHãy xác định độ phức tạp của từng hàm đã cài đặt.

CHƯƠNG 2. ĐỆ QUY

Chương này giới thiệu giải thuật đệ quy, giải thuật mà một vấn đề được giải quyết bằng cách gọi đến chính giải thuật đó để thực hiện những công việc nhỏ hơn. Chúng ta sẽ tìm hiểu một số chương trình và ứng dụng mẫu. Chúng ta cũng phân tích tại sao sử dụng giải thuật đệ quy, và khi nào nên dùng, khi nào không nên dùng đệ quy.

I. Khái niệm đệ quy

Một đối tượng được gọi là đệ quy nếu nó được định nghĩa dựa vào chính nó hoặc một đối tượng khác cùng dạng với chính nó.

Ví dụ: Khi ta đặt hai chiếc gương đối diện nhau, chiếc gương thứ nhất chứa hình chiếc gương thứ hai. Chiếc gương thứ hai lại chứa hình chiếc gương thứ nhất, vì thế nó sẽ chứa lại chính hình ảnh của nó.

Trong toán học, ta cũng thường hay gặp các định nghĩa đệ quy như: Giai thừa của một số n (Ký hiệu: $n!$): Nếu $n=0$ thì $n!=1$, nếu $n>0$ thì $n!=n.(n-1)!$.

II. Xây dựng giải thuật đệ quy

1. Định nghĩa

Nếu lời giải của bài toán P được thực hiện bằng lời giải của bài toán P' có dạng giống như P thì đó là một lời giải đệ quy. Giải thuật tương ứng với lời giải như vậy gọi là giải thuật đệ quy.

Một giải thuật đệ quy gồm có hai phần:

- + Phần cơ sở: Phần này được thực hiện khi công việc đơn giản, có thể giải trực tiếp, không cần nhờ đến một bài toán con nào cả.
- + Phần đệ quy: Chưa thể giải trực tiếp, phải xác định và gọi các bài toán con.

2. Ví dụ

Sau đây ta sẽ phân tích bài toán tính giai thừa của một số tự nhiên, sau đó xây dựng giải thuật đệ quy cho bài toán. Trong toán học, giai thừa của một số nguyên không âm được định nghĩa như sau:

$$n! = n \times (n-1) \times \dots \times 1$$

Với cách định nghĩa này, ta không thể thấy được một cách rõ ràng cách tính giai thừa như thế nào. Vì thế, người ta thường chuyển nó về cách định nghĩa qui nạp như sau:

$$\begin{cases} n! = 1 & \text{Nếu } n=0 \\ n! = n \cdot (n-1)! & \text{Nếu } n > 0 \end{cases}$$

Trường hợp $n=4$ ta có được các kết quả tính toán từng bước như sau:

$$\begin{aligned} 4! &= 4 \cdot 3! \\ &= 4 \cdot (3 \cdot 2!) \\ &= 4 \cdot (3 \cdot (2 \cdot 1!)) \\ &= 4 \cdot (3 \cdot (2 \cdot (1 \cdot 0!))) \\ &= 4 \cdot (3 \cdot (2 \cdot (1 \cdot 1))) \\ &= 4 \cdot (3 \cdot (2 \cdot 1)) \\ &= 4 \cdot (3 \cdot 2) \\ &= 4 \cdot 6 \\ &= 24 \end{aligned}$$

Việc tính toán từng bước như trên cho chúng ta thấy rõ cách làm việc của giải thuật đệ quy. Như ta thấy, cách tính toán trong từng bước là hoàn toàn giống nhau, nhưng phạm vi hay giá trị tính toán được thu nhỏ dần cho đến khi nào gặp trường hợp cơ sở.

Sau đây ta cài đặt hàm tính giai thừa bằng ngôn ngữ C++

```
int GiaiThua(int n)
{
    if(n==0)
        return 1;
```

```
else
    return n*GiaiThua(n-1);
}
```

3. Thiết kế giải thuật đệ quy

Giải thuật đệ quy là một công cụ giúp người lập trình tập trung vào những khía cạnh chính của thuật toán. Chương trình sử dụng đệ quy được thường là nhỏ gọn và dễ hiểu so với chương trình sử dụng các phương pháp khác. Khi thiết kế một giải thuật đệ quy, chúng ta cần quan tâm đến những khía cạnh quan trọng sau:

- **Tìm ra phần đệ quy của giải thuật.** Trong ví dụ tính giai thừa, công thức $n! = n \cdot (n-1)!$ Nếu $n > 0$ là phần đệ quy.
- **Tìm ra phần cơ sở của giải thuật.** Trong ví dụ tính giai thừa, $n! = 1$ nếu $n = 0$ là phần cơ sở. Nó xác định, thuật toán sẽ dừng khi nào.
- **Kết hợp phần đệ quy và phần cơ sở.** Sử dụng cấu trúc điều khiển if để phân chia trường hợp tính toán.

III. Một số bài toán sử dụng đệ quy

1. Dãy số Fibonacci

Dãy số Fibonacci được định nghĩa như sau: Cho n là một số nguyên không âm, ta có:

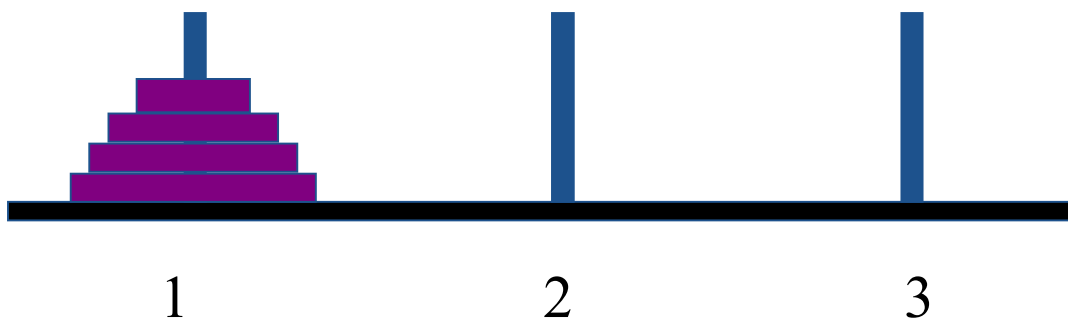
$$F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2} \text{ với } n \geq 2.$$

```
int Fibonacci(int n)
{
    if (n <= 0) return 0;
    else if (n == 1) return 1;
        else return Fibonacci(n-1) + Fibonacci(n-2);
}
```

2. Bài toán Tháp Hà Nội

Bài toán Tháp Hà Nội xuất phát từ trò chơi đồ Tháp Hà Nội như sau: "Người chơi được cho ba cái cọc và một số đĩa có kích thước khác nhau có thể cho vào các cọc này. Ban đầu sắp xếp các đĩa theo trật tự kích thước vào một cọc sao cho đĩa nhỏ nhất nằm trên cùng, tức là tạo ra một dạng hình nón. Người chơi phải di chuyển toàn bộ số đĩa sang một cọc khác, tuân theo các quy tắc sau:

- + Một lần chỉ được di chuyển một đĩa.
- + Một đĩa chỉ có thể được đặt lên một đĩa lớn hơn.



Trong trường hợp chỉ có 2 đĩa đặt ở vị trí 1, ta có thể thực hiện một cách đơn giản như sau: Chuyển đĩa nhỏ sang vị trí 3, đĩa lớn sang vị trí 2. Sau đó chuyển đĩa nhỏ từ vị trí 3 sang vị trí 2. Khi đó, cả 2 đĩa được chuyển từ vị trí 1 sang vị trí 2 theo đúng nguyên tắc. Đối với trường hợp có n đĩa, ta có thể phân tích bài toán theo tư duy quy nạp như sau:

- + Nếu $n=1$, ta di chuyển đĩa duy nhất từ vị trí 1 sang vị trí 2 là xong
- + Giả sử ta có phương pháp chuyển được $n-1$ đĩa từ vị trí x sang vị trí y thì công việc chuyển n đĩa từ vị trí 1 sang vị trí 2 có thể được lý giải như sau: Ta di chuyển $n-1$ phía trên đĩa từ vị trí 1 sang vị trí 3, chuyển đĩa nằm dưới cùng sang vị trí 2. Sau đó, chuyển $n-1$ đĩa từ vị trí 3 sang vị trí 2. Khi đó, n đĩa đã được chuyển từ vị trí 1 sang vị trí 2.

Phương pháp đó được hiện thực trong chương trình dưới đây:

```
#include<stdio.h>
```

```
#include<conio.h>
void Move(int n, int x, int y)//Chuyen n dia tu x sang y
{
    if(n==1)
        printf("\nDi chuyen 1 dia tu %d sang %d", x, y);
    else
    {
        Move(n-1, x, 6-x-y);//Chuyen n-1 dia tu x sang vi tri con lai (x+y+z=6)
        Move(1, x, y);//Chuyen 1 dia tu x sang y
        Move(n-1, 6-x-y, y);//Chuyen n-1 dia tu vi tri con lai sang y
    }
}
void main()
{
    int n;
    printf("\nNhap so dia: ");
    scanf("%d", &n);
    int x=1, y=2;
    Move(n,x,y);
}
```

Nhận xét:

Qua các ví dụ trên, ta có thể nhận thấy lợi ích của đệ quy trong việc giải quyết các bài toán. Một bài toán giải bằng đệ quy vẫn có thể giải được bằng các phương pháp khác, người ta gọi là dạng khử đệ quy. Tuy nhiên, đệ quy vẫn là phương pháp giúp việc thiết kế chương trình đơn giản và rõ ràng hơn. Việc chọn hay không chọn phương pháp đệ quy còn tùy thuộc vào yêu cầu cụ thể của từng bài toán. Đôi khi giải bằng các phương pháp khác lại hữu hiệu hơn so với phương pháp đệ quy, ví dụ bài toán tính giai thừa hay tính số Fibonacci. Phương pháp khử đệ quy sử dụng cấu trúc dữ liệu ngăn xếp (Stack) chúng ta sẽ được tìm hiểu ở chương 5.

V. Bài tập chương 2

1. Viết hàm đệ quy tính tổng tất cả các số từ 1 đến n. Với n là số nguyên dương.
2. Viết hàm đệ quy tìm và trả về phần tử nhỏ nhất trong một mảng kích thước n các số nguyên.
3. Viết hàm đệ quy xác định xem một mảng kiểu số nguyên gồm n phần tử có phải là mảng đối xứng hay không.
4. Viết hàm tính a^n theo hai cách: sử dụng đệ quy và không đệ quy. Với n là số nguyên không âm.
5. Viết hàm đệ quy tính C_n^k theo công thức sau

$$C_n^0 = C_n^n = 1$$

$$\text{Với } 0 < k < n: C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$$

6. Viết hàm đệ quy tính A(m, n) theo công thức sau

$$A(0, n) = n + 1 \text{ nếu } n \geq 0.$$

$$A(m, 0) = A(m-1, 1) \text{ nếu } m > 0.$$

$$A(m, n) = A(m-1, A(m, n-1)) \text{ nếu } m > 0 \text{ và } n > 0.$$

7. Sử dụng đệ quy để tìm cách đặt 8 hoàng hậu lên bàn cờ 8x8 sao cho các hoàng hậu không ăn nhau.

CHƯƠNG 3. TÌM KIẾM VÀ SẮP XẾP

Chương này giới thiệu những vấn đề liên quan đến việc tìm kiếm và sắp xếp các phần tử trên một danh sách. Chúng ta sẽ tìm hiểu hai thuật toán tìm kiếm thông dụng là: Tìm kiếm tuyến tính và tìm kiếm nhị phân. Đồng thời, chúng ta sẽ làm quen với các thuật toán sắp xếp và cách đánh giá mức độ hiệu quả của từng thuật toán.

I. Giới thiệu

Tra cứu thông tin là một công việc rất quan trọng mà máy tính mang lại. Ví dụ, khi chúng ta nhập họ tên để tìm số điện thoại của một người trong một danh bạ điện thoại, nhập số tài khoản để thực hiện các công việc như rút tiền, chuyển tiền, truy vấn tài khoản từ ngân hàng, hay nhập họ tên để tra cứu điểm thi. Tất cả những công việc đó đòi hỏi máy tính phải thực hiện thao tác tìm kiếm trong một danh sách đã được lưu trữ trước đó. Đứng ở vị trí người xây dựng phần mềm, chúng ta cần phải nắm rõ những phương pháp tìm kiếm, ưu và nhược điểm của từng phương pháp để áp dụng cho từng bài toán cụ thể.

Bên cạnh đó để hỗ trợ việc tìm kiếm sao cho nhanh chóng và hiệu quả, dữ liệu cần phải được tổ chức và sắp xếp sao cho hợp lý. Vì vậy, việc nắm vững các thuật toán sắp xếp là rất quan trọng đối với người xây dựng phần mềm.

Người ta chia bài toán các bài toán tìm kiếm và sắp xếp ra làm hai loại. Đối với các bài toán mà dữ liệu quá lớn, việc sắp xếp và tìm kiếm được thực hiện trực tiếp trên bộ nhớ phụ (dữ liệu được đặt trên các thiết bị lưu trữ như ổ cứng, đĩa mềm, USB, ...) gọi là **tìm kiếm và sắp xếp ngoại** hay tìm kiếm và sắp xếp trên tập tin. Loại thứ hai là tìm kiếm và sắp xếp dữ liệu trên bộ nhớ chính (lưu trên RAM) gọi là **tìm kiếm và sắp xếp nội**. Trong phạm vi môn học này, chúng ta chỉ nghiên cứu các thuật toán tìm kiếm và sắp xếp nội, các thuật toán tìm kiếm và sắp xếp ngoại sẽ được giới thiệu trong môn học Cấu trúc dữ liệu 2.

II. Các giải thuật tìm kiếm

Hai giải thuật tìm kiếm nội thông dụng là tìm kiếm tuyến tính và tìm kiếm nhị phân.

1. Tìm kiếm tuyến tính

a. Giải thuật

Đây là giải thuật đơn giản, nguyên tắc cơ bản là lần lượt so sánh khóa x cần tìm với các phần tử từ đầu đến cuối danh sách cho đến khi nào tìm thấy hoặc đã duyệt hết danh sách mà không tìm thấy x . Các bước thực hiện như sau:

Input: Mảng $a[N]$, khóa x cần tìm

Output: Kết quả tìm kiếm

Bước 1: $i=1$; //Bắt đầu từ phần tử đầu tiên của mảng

Bước 2: So sánh $a[i]$ với x .

+ Nếu $a[i] = x$: Tìm thấy. DỪNG

+ Ngược lại: Sang bước 3

Bước 3: $i = i+1$; //Xét phần tử kế tiếp trong mảng

+ Nếu $i > N$: Hết mảng. Không thấy. DỪNG

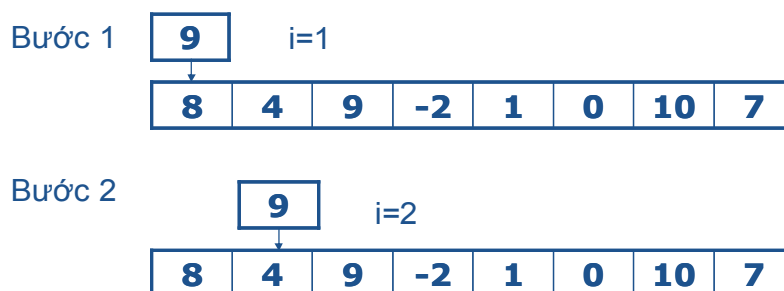
+ Ngược lại: Lặp lại bước 2.

❖ Ghi chú: Ở đây đề cập đến khái niệm "khóa". Dữ liệu tại mỗi phần tử có thể rất phức tạp (chứa trong một struct chẳng hạn). Trong trường hợp này, khóa của phần tử được tính dựa trên một hoặc nhiều trường nào đó gọi là trường khóa.

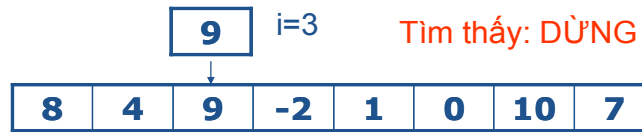
b. Ví dụ

Cho danh sách các số nguyên: 8, 4, 9, -2, 1, 0, 10, 7

Với $x=9$.



Bước 3



c. Cài đặt

Từ giải thuật ta xây dựng hàm tìm kiếm tuyến tính bằng ngôn ngữ lập trình C++.

```
int LinearSearch( int a[], int N, int x)
{
    int i=0;
    for(i=0;i<n;i++)
        if(a[i] == x)
            return 1; //tìm thấy
    return 0; //không tìm thấy
}
```

Ngoài việc cài đặt bằng cấu trúc điều khiển **for**, ta có thể cài đặt bằng vòng lặp **while**. Phần này dành cho người đọc xem như bài tập.

d. Đánh giá giải thuật

Trong thuật toán này, ta có thể tính toán được số lượng các phép toán so sánh. Trường hợp tốt nhất khi phần tử cần tìm là phần tử đầu tiên, và xấu nhất khi đó là phần tử cuối cùng hoặc không tìm thấy. Nếu chỉ xét đến số lần so sánh x với các phần tử (không tính số lần so sánh để thực hiện vòng lặp) thì trường hợp tốt nhất là thực hiện 1 phép so sánh, và xấu nhất là n phép so sánh. Như vậy số phép so sánh trung bình là: $\frac{n(n+1)}{2n} = \frac{(n+1)}{2}$. Từ đó ta có độ phức tạp tính toán của giải thuật là $O(n)$.

Giải thuật tìm kiếm tuyến tính không phụ thuộc vào thứ tự của các phần tử mảng. Đây là giải thuật tổng quát nhất. Tuy nhiên, đối với các mảng có thứ tự thì việc tìm kiếm bằng tuyến tính không phải là giải thuật tốt nhất. Ta sẽ tìm hiểu giải thuật tìm kiếm nhị phân giúp tận dụng tích chất có thứ tự để tìm kiếm hiệu quả hơn.

2. Tìm kiếm nhị phân

a. Giải thuật

Ý tưởng của giải thuật là ta tiến hành so sánh khóa x cần tìm với phần tử giữa của danh sách, với điều kiện danh sách có thứ tự tăng. Từ đó, ta xác định được x nằm ở nửa trước hay nửa sau của danh sách. Tiếp tục tìm theo phương pháp tương tự trên nửa có khả năng chứa x cho đến khi tìm thấy. Các bước thực hiện như sau:

Input: Mảng $a[N]$, khóa x cần tìm

Output: Kết quả tìm kiếm

Bước 1: $left=1$; $right=N$;

Bước 2: $mid = (left + right)/2$;

+ Nếu $a[mid] = x$; //Tìm thấy DỪNG

+ Nếu $a[mid] < x$ thì $right=mid - 1$;

+ Nếu $a[mid] > x$ thì $left = mid + 1$;

Bước 3: Nếu $left \leq right$ thì Lặp lại bước 2

Ngược lại: DỪNG

b. Ví dụ

Cho dãy số: 4, 6, 8, 9, 11, 22, 34, 40, 44

Khóa x cần tìm: 8

Lần 1

4	6	8	9	11	22	34	40	55
---	---	---	---	----	----	----	----	----

$Left=1$ $mid=5$ $Right=9$
 $a[mid]=11 \rightarrow a[mid] > x \rightarrow right = mid - 1 = 5 - 1 = 4$

Lần 2

4	6	8	9	11	22	34	40	55
---	---	---	---	----	----	----	----	----

$Left=1$ $mid=2$ $Right=4$
 $a[mid]=6 \rightarrow a[mid] < x \rightarrow left = mid + 1 = 2 + 1 = 3$

Lần 3

4	6	8	9	11	22	34	40	55
---	---	---	---	----	----	----	----	----

$mid=3$ $Left=3$ $Right=4$

$a[mid]=8 \rightarrow a[mid] = x \rightarrow DỪNG$

c. Cài đặt

```

int BinarySearch(int a[], int n, int x)
{
    int l=0, r=n-1;
    int mid;
    do{
        mid=(l+r)/2;
        if(x==a[mid])
            return mid;//Tìm thấy x tại vị trí mid
        else
            if(x<a[mid])

```

```
        r=mid-1;
    else
        l=mid+1;
    }while(l<=r);
    return -1;//Khong tim thay x
}
```

d. Đánh giá giải thuật

Trường hợp tốt nhất của giải thuật là khi x nằm ở vị trí giữa của mảng, khi đó số lần so sánh là 1. Trường hợp xấu nhất là khi x không có trong mảng, khi đó số lần so sánh là $\log_2 n$. Như vậy độ phức tạp tính toán của thuật giải là $O(\log_2 n)$.

Rõ ràng giải thuật tìm kiếm nhị phân tiết kiệm thời gian tính toán hơn rất nhiều so với giải thuật tuyến tính (có độ phức tạp $O(n)$). Tuy nhiên, đối với dữ liệu chưa được sắp xếp, muốn áp dụng thuật toán tìm kiếm nhị phân, ta phải thực hiện công việc sắp xếp trước. Điều này dẫn đến thời gian và chi phí tính toán của bài toán tăng lên. Vì thế, việc áp dụng giải thuật tìm kiếm tuyến tính hay nhị phân cần phải được cân nhắc kỹ dựa trên yêu cầu thực tế của bài toán.

III. Các giải thuật sắp xếp

1. Giới thiệu bài toán sắp xếp

Sắp xếp là quá trình xử lý trên một danh sách các phần tử để đặt chúng theo một thứ tự thỏa mãn một tiêu chuẩn nào đó dựa trên thông tin lưu giữ tại mỗi phần tử.

Khi tiến hành xây dựng thuật toán sắp xếp, ta cần chú ý giảm thiểu những phép so sánh và đổi chỗ không cần thiết. Vì sắp xếp dữ liệu trên bộ nhớ chính, nên nhu cầu tiết kiệm bộ nhớ được coi trọng. Do đó, phần lớn các giải thuật không quan tâm đến giải pháp dùng thêm bộ nhớ để lưu trữ những dữ liệu tạm thời mà tập trung vào giải pháp sắp xếp trực tiếp trên dãy số ban đầu (gọi là các giải thuật sắp xếp tại chỗ). Sau đây là các giải thuật sắp xếp thông dụng.

2. Phương pháp đổi chỗ trực tiếp (Interchange sort)

a. Giải thuật

Ý tưởng chính của giải thuật là xuất phát từ đầu danh sách, xét tất cả các cặp phần tử trong dãy, nếu phần tử sau nhỏ hơn (hoặc lớn hơn nếu sắp xếp giảm) phần tử trước thì tiến hành hoán vị. Các bước thực hiện như sau:

Bước 1: $i = 1$;

Bước 2: $j = i+1$;

Trong khi mà $j \leq N$ thực hiện

Nếu $a[j] < a[i]$ thì Hoán vị ($a[i], a[j]$);

$j = j+1$;

end;


Bước 3: $i = i+1$;

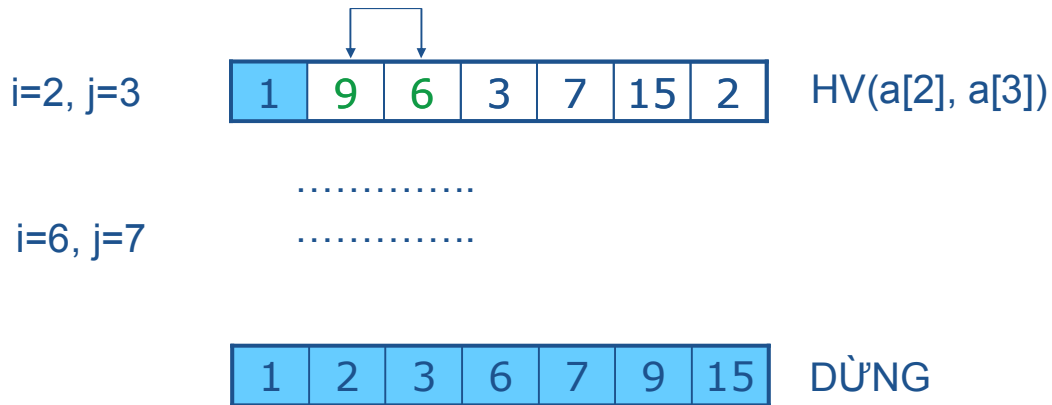
Nếu $i < N$: Lặp lại bước 2;

Ngược lại: DỪNG.

b. Ví dụ

Sắp xếp dãy số sau: 6, 9, 1, 3, 7, 15, 2.

$i=1, j=2$	<table border="1" style="display: inline-table; text-align: center;"><tr><td>6</td><td>9</td><td>1</td><td>3</td><td>7</td><td>15</td><td>2</td></tr></table>	6	9	1	3	7	15	2	Không hoán vị
6	9	1	3	7	15	2			
$i=1, j=3$	<table border="1" style="display: inline-table; text-align: center;"><tr><td>6</td><td>9</td><td>1</td><td>3</td><td>7</td><td>15</td><td>2</td></tr></table> 	6	9	1	3	7	15	2	HV($a[1], a[3]$)
6	9	1	3	7	15	2			
$i=1, j=4$	<table border="1" style="display: inline-table; text-align: center;"><tr><td>1</td><td>9</td><td>6</td><td>3</td><td>7</td><td>15</td><td>2</td></tr></table>	1	9	6	3	7	15	2	Không hoán vị
1	9	6	3	7	15	2			
$i=1, j=5$	<table border="1" style="display: inline-table; text-align: center;"><tr><td>1</td><td>9</td><td>6</td><td>3</td><td>7</td><td>15</td><td>2</td></tr></table>	1	9	6	3	7	15	2	Không hoán vị
1	9	6	3	7	15	2			
$i=1, j=6$	<table border="1" style="display: inline-table; text-align: center;"><tr><td>1</td><td>9</td><td>6</td><td>3</td><td>7</td><td>15</td><td>2</td></tr></table>	1	9	6	3	7	15	2	Không hoán vị
1	9	6	3	7	15	2			
$i=1, j=7$	<table border="1" style="display: inline-table; text-align: center;"><tr><td>1</td><td>9</td><td>6</td><td>3</td><td>7</td><td>15</td><td>2</td></tr></table>	1	9	6	3	7	15	2	Không hoán vị
1	9	6	3	7	15	2			



c. Cài đặt

```

void InterchangeSort(int a[], int n)
{
    for(int i=0;i<n-1;i++)
        for(int j=i+1;j<n;j++)
            if(a[i]>a[j])
            {
                int temp=a[i];
                a[i]=a[j];
                a[j]=temp;
            }
}

```

d. Đánh giá giải thuật

Số phép so sánh trong giải thuật đổi chỗ trực tiếp không phụ thuộc vào tình trạng của dãy số ban đầu, nhưng số phép hoán vị thì tùy thuộc vào kết quả so sánh. Tổng số lần so sánh là $n(n-1)/2$.

3. Phương pháp chọn trực tiếp (Selection sort)

a. Giải thuật

Ý tưởng của thuật toán như sau: Duyệt dãy n phần tử của danh sách, tìm phần tử nhỏ nhất, đưa phần tử này về vị trí đầu. Tiếp theo duyệt dãy $n-1$ phần tử, bỏ qua phần tử đầu tiên, tìm phần tử nhỏ nhất, đưa phần tử này về vị trí thứ hai. Lặp lại quá trình này cho đến khi dãy cần duyệt chỉ còn 1 phần tử. Các bước thực hiện như sau:

Bước 1: $i = 1$;

Bước 2: Tìm phần tử $a[\text{min}]$ nhỏ nhất trong dãy từ $a[i]$ đến $a[N]$

Bước 3: Hoán vị $a[\text{min}]$ và $a[i]$

Bước 4: Nếu $i < N-1$ thì

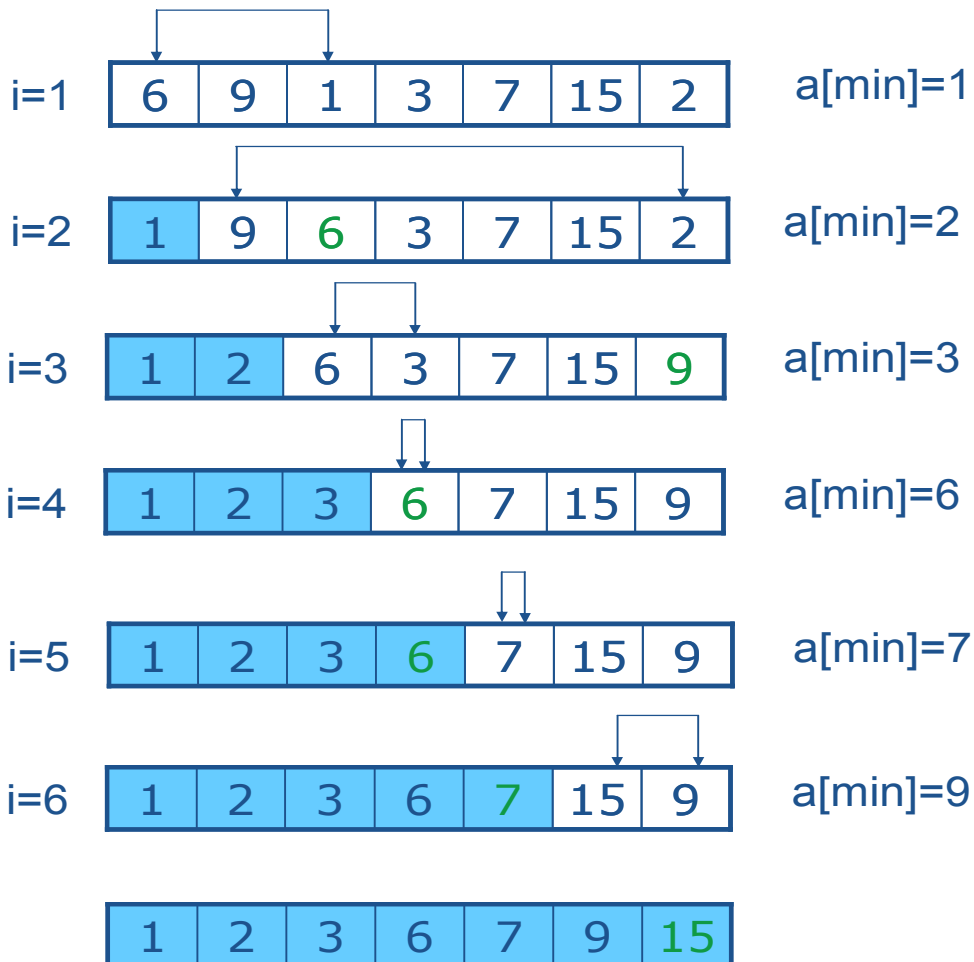
+ $i = i+1$;

+ Lặp lại bước 2

Ngược lại: DỪNG

b. Ví dụ

Sắp xếp dãy số sau: 6, 9, 1, 3, 7, 15, 2



c. Cài đặt

```

void SelectionSort(int a[], int n)
{
    int minIndex;
    for(int i=0;i<n-1;i++)
    {
        minIndex=i;
        for(int j=i+1;j<n;j++)
            if(a[j]<a[minIndex])
                minIndex=j;
        //Hoan vi a[i], a[minIndex]
        int temp=a[i];
        a[i]=a[minIndex];
        a[minIndex]=temp;
    }
}

```

d. Đánh giá giải thuật

Số phép so sánh của các phần tử trong dãy n phần tử lần lượt là: $n-1, n-2, \dots, 2, 1$. Như vậy tổng số số phép so sánh là: $n(n-1)/2$. Ta thấy rằng chi phí thực hiện của giải thuật không phụ thuộc vào tình trạng của dãy ban đầu.

4. Phương pháp chèn trực tiếp (Insertion sort)

a. Giải thuật

Ý tưởng của giải thuật là tiến hành chèn phần tử a_i vào dãy đã được sắp thứ tự để có dãy mới cũng có thứ tự. Các bước thực hiện như sau:

Bước 1: $i=2$;

Bước 2: $x=a[i]$;

Tìm vị trí pos thích hợp trong đoạn $a[1] \dots a[i-1]$ để chèn $a[i]$ vào.

Bước 3: Dời chỗ các phần tử từ $a[pos]$ đến $a[i-1]$ sang phải một vị trí.

Bước 4: $a[pos] = x$;

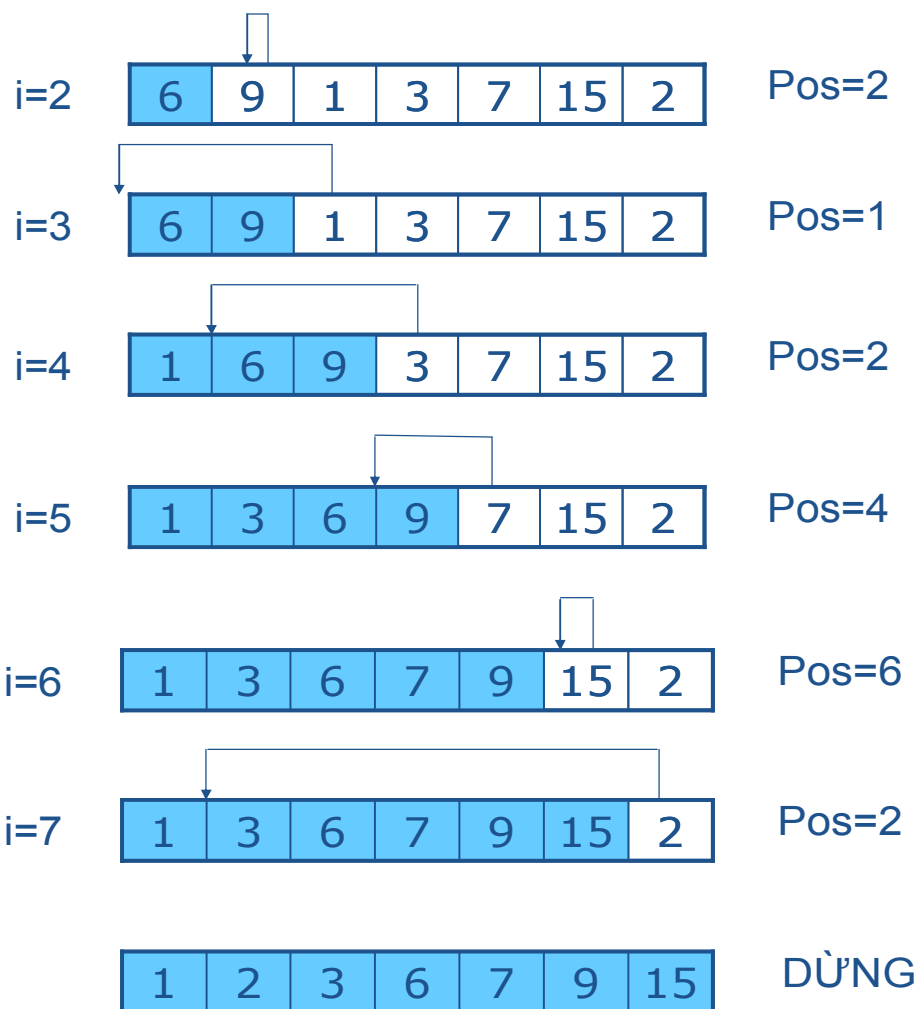
Bước 5: $i = i+1$;

Nếu $i \leq N$: Lặp lại bước 2

Ngược lại: DỪNG.

b. Ví dụ

Sắp xếp dãy số sau: 6, 9, 1, 3, 7, 15, 2



c. Cài đặt

```
void InsertionSort(int a[], int n)
{
    for(int i=1;i<n;i++)
    {
        int x=a[i];
        //Chen x vao vi tri thich hop
        for(int j=i-1;j>=0;j--)
            if(a[j]>x)
                a[j+1]=a[j];
            else
                break;
        a[j+1]=x;
    }
}
```

Chúng ta có thể cải tiến giải thuật bằng cách thay thế bước tìm kiếm tuyến tính vị trí cần chèn x bằng giải thuật tìm kiếm nhị phân, vì dãy cần chèn là dãy đã sắp thứ tự. Phần này dành cho người đọc xem như bài tập.

d. Đánh giá giải thuật

Giải thuật chèn trực tiếp là phương pháp sắp xếp dễ hiểu, dễ cài đặt nhưng không tối ưu vì thời gian thực hiện giải thuật chậm. Độ phức tạp của giải thuật là $O(n^2)$.

5. Phương pháp nổi bọt (Bubble sort)

a. Giải thuật

Ý tưởng của giải thuật là xuất phát từ cuối dãy, đổi chỗ các cặp phần tử kế cận để đưa phần tử các phần tử nhỏ (hoặc lớn hơn nếu sắp giảm dần) về phía đầu dãy. Các bước thực hiện như sau:

Bước 1: $i = 1$;

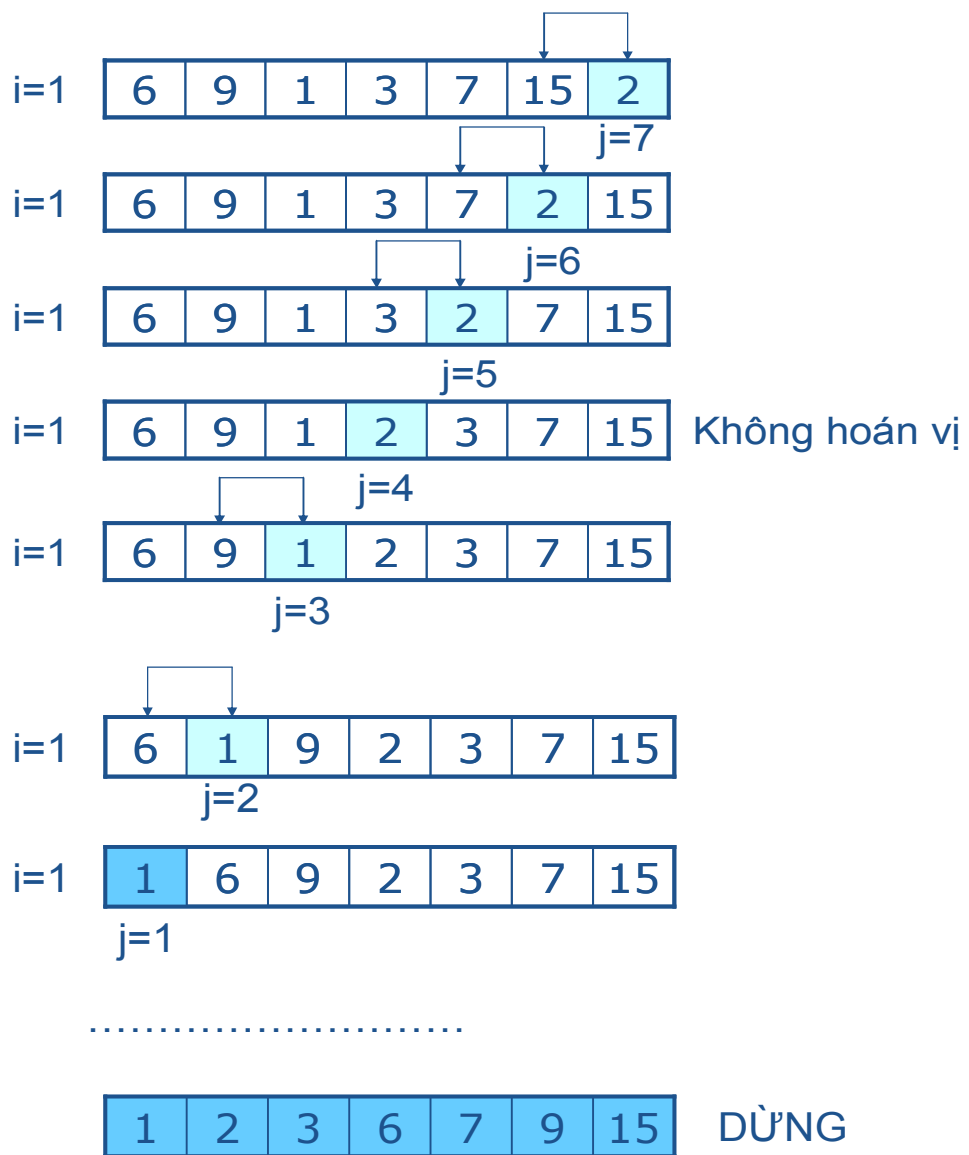
Bước 2: $j = N$;

Trong khi $j > i$ thực hiện:

```

        Nếu  $a[j] < a[j-1]$  thì HoánVị( $a[j], a[j-1]$ );
        j = j-1;
    end;
    Bước 3: i = i+1;
        Nếu  $i > N-1$  thì DỪNG
    Ngược lại: Lặp lại Bước 2;
    
```

b. Ví dụ



c. Cài đặt

```

void BubbleSort(int a[], int n)
{
    for(int i=0;i<n-1;i++)
        for(int j=n-1;j>i;j--)
            if(a[j]<a[j-1])
            {
                int temp=a[j];
                a[j]=a[j-1];
                a[j-1]=temp;
            }
}

```

d. Đánh giá giải thuật

Giải thuật BubbleSort không nhận diện được tình trạng dãy đã có thứ tự hay chưa. Độ phức tạp của giải thuật là $O(n^2)$. Có thể cải tiến thuật toán bằng cách thực hiện hai lượt đi về cùng lúc. Lượt đi sẽ đẩy các phần tử nhỏ về đầu dãy, lượt về sẽ đẩy các phần tử lớn về cuối dãy. Thuật toán cải tiến như vậy gọi là ShakerSort.

6. Sắp xếp dựa trên phân hoạch (Quicksort)

a. Giải thuật

Quick sort thuộc loại phương pháp “chia để trị” (divide and conquer). Chọn một số nguyên x nào đó, dãy cần sắp xếp sẽ được chia thành hai dãy con, một dãy bao gồm các phần tử không lớn hơn x , một dãy bao gồm các phần tử không nhỏ hơn x .



Tiếp tục thực hiện tương tự cho từng dãy con, cho đến khi dãy tất cả các dãy con đều có thứ tự, cũng có nghĩa là dãy cần sắp xếp đã có thứ tự. Việc lựa chọn phần tử nào là phần tử x trong dãy đang xét tùy thuộc vào người lập trình. Ở đây, phần tử x được chọn là phần tử nằm ở vị trí chính giữa của dãy.

Các bước thực hiện như sau:

Bước 1:

Đặt left là giá trị index đầu của dãy, right là giá trị index cuối của dãy.

Chọn phần tử $x=a[k]$, với $k=(left+right)/2$;

$i=left$;

$j=right$;

Bước 2:

Phát hiện và hiệu chỉnh cặp phần tử $a[i]$, $a[j]$ nằm sai vị trí:

Bước 2a: Trong khi $a[i] < x$ thì $i++$;

Bước 2b: Trong khi $a[j] > x$ thì $j--$;

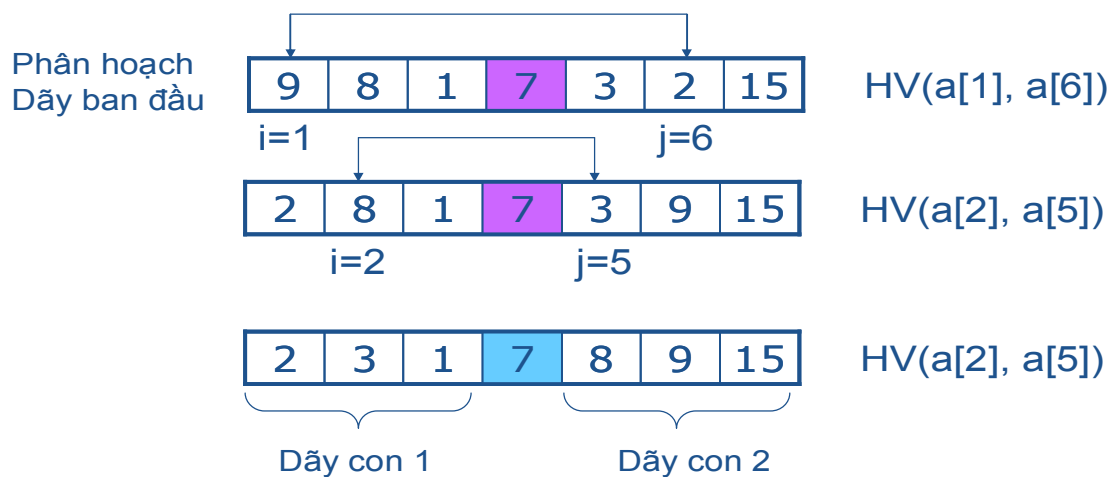
Bước 2c: Nếu $i < j$ thì Hoán vị $a[i]$ và $a[j]$;

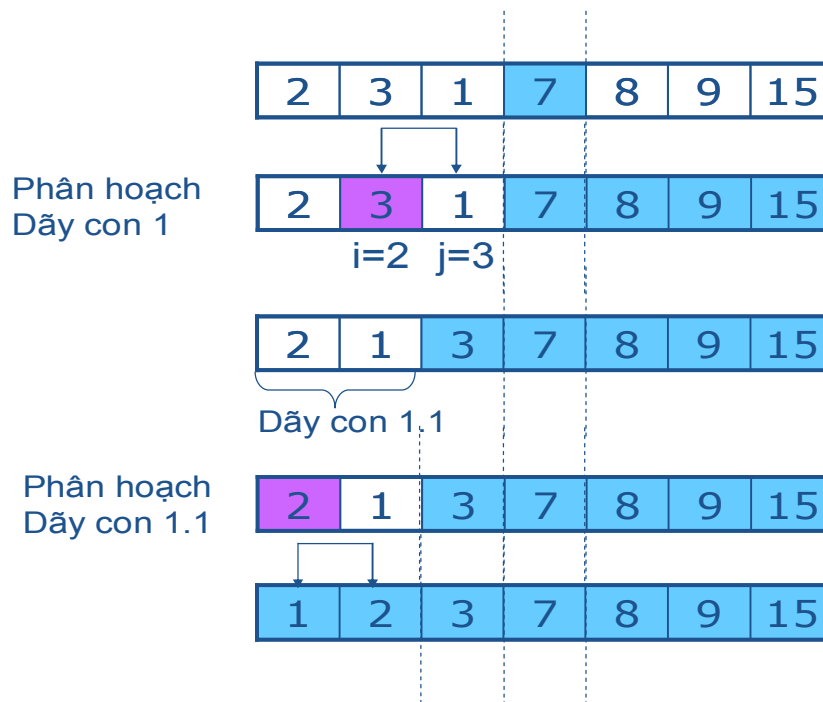
Bước 3:

Nếu $i < j$: Lặp lại bước 2

Nếu $i \geq j$: Dừng.

b. Ví dụ





c. Cài đặt

```

void QuickSort(int a[], int left, int right)
{
    int i, j;
    int x;
    x = a[(left + right)/2];
    i=left; j=right;
    do{
        while(a[i] < x) i++;
        while(a[j] > x) j--;
        if(i <= j)
        {
            HoanVi(a[i], a[j]);
            i++; j--;
        }
    }while(i < j)
    if(left < j) QuickSort(a, left, j);
    if(i < right) QuickSort(a, i, right);
}

```

d. Đánh giá giải thuật

Hiệu quả của phương pháp Quick sort phụ thuộc vào việc chọn giá trị mốc khi phân hoạch. Trường hợp tốt nhất nếu mỗi lần phân hoạch, ta đều chọn được phần tử làm mốc là phần tử tốt nhất, tức là có một nửa số phần tử của dãy lớn hơn, và một nửa còn lại nhỏ hơn phần tử làm mốc. Khi đó số lần phân hoạch là $\log_2 n$. Trường hợp xấu nhất là mỗi lần phân hoạch, ta chọn phần tử lớn nhất hoặc nhỏ nhất của dãy làm mốc. Khi đó số lần phân hoạch là n .

IV. Bài tập chương 3

1. Phân biệt hai loại giải thuật sắp xếp: Sắp xếp nội và sắp xếp ngoại
2. Cài đặt thuật toán tìm kiếm tuyến tính dùng vòng lặp **while**.
3. Cài đặt thuật toán Insertion sort bằng cách sử dụng phương pháp tìm kiếm nhị phân.
4. Phân tích và so sánh tính hiệu quả của giải thuật Quick sort với các giải thuật còn lại.
5. Viết chương trình minh họa các phương pháp tìm kiếm. Chương trình có các chức năng sau:
 - + Đọc danh sách các phần tử từ tập tin.
 - + Cho phép người lựa chọn các phương pháp sau để tìm kiếm một phần tử trên danh sách: Tìm kiếm tuyến tính, tìm kiếm nhị phân.
 - + Hiện thị thông tin số lần so sánh, số lần hoán vị của mỗi thuật toán.
 - + Xuất kết quả tìm kiếm ra tập tin.
6. Viết chương trình mô phỏng các phương pháp sắp xếp. Chương trình có các chức năng sau:
 - + Đọc danh sách các phần tử từ tập tin văn bản
 - + Cho phép người dùng thao tác trên danh sách như: Thêm, xóa, sửa các phần tử.
 - + Cho phép người dùng lựa chọn một trong các thuật toán sau để sắp xếp:

Interchange sort, Bubble sort, Selection sort, Insertion sort, Quick sort.

+ Hiển thị thông tin số lần so sánh, số lần hoán vị của mỗi thuật toán.

+ Xuất kết quả sắp xếp ra tập tin.

CHƯƠNG 4. DANH SÁCH (LIST)

Chương này trình bày một kiểu dữ liệu trừu tượng là danh sách (list) cùng với các tính chất của nó. Từ đó, chúng ta sẽ nghiên cứu các cách thức khác nhau để cài đặt danh sách trên máy tính.

I. Định nghĩa danh sách

Danh sách là một tập hợp hữu hạn các phần tử có cùng kiểu. Số phần tử của danh sách gọi là độ dài của danh sách. Nếu độ dài danh sách bằng không, ta nói đó là danh sách rỗng. Một tính chất quan trọng của danh sách là các phần tử của danh sách có thứ tự tuyến tính theo vị trí.

Trong thực tế, kiểu dữ liệu danh sách được dùng khi người ta cần lưu thông tin của tất cả sinh viên trong một lớp học, tất cả nhân viên trong một công ty, ... hay đơn giản là một dãy các số nguyên khi ta giải quyết một bài toán nào đó trên máy tính.

Trong phạm vi của giáo trình, ta ký hiệu danh sách là một dãy các phần tử: $a_1, a_2, \dots, a_{n-1}, a_n$, với $n \geq 0$. Một số phép toán cơ bản trên danh sách như:

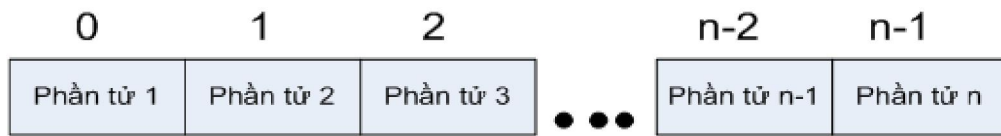
- + Chèn một phần tử vào danh sách
- + Hủy một phần tử khỏi danh sách
- + Tìm kiếm một phần tử trong danh sách
- + Liệt kê tất cả các phần tử của danh sách
- + Kiểm tra danh sách có rỗng hay không
- + Sắp xếp danh sách.
- +

II. Cài đặt danh sách

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

Kiểu dữ liệu mảng giúp lưu trữ các dữ liệu cùng kiểu trên những vùng nhớ liên tiếp nhau (vì vậy ta gọi là danh sách đặc). Khi cài đặt danh sách bằng một mảng, ta cần một biến nguyên n để lưu số phần tử hiện có trong danh sách. Nếu

mảng được đánh số bắt đầu từ 0 thì các phần tử trong danh sách được cất giữ trong mảng được đánh số từ 0 đến n-1.



Ta khai báo cấu trúc của kiểu dữ liệu list như sau:

```
#define MAX ...

#define ElementType <KieuDuLieu>

typedef struct tagNode
{
    ElementType a[MAX]; //Mảng các phần tử của danh sách
    int n; //Độ dài danh sách
} List;
```

Như vậy ta đã xây dựng được kiểu dữ liệu trừu tượng là List bằng cách sử dụng mảng. Từ đây ta sẽ cài đặt các phép toán cơ bản trên danh sách.

a. Kiểm tra danh sách rỗng

Nếu độ dài n của danh sách bằng 0 thì danh sách rỗng.

```
int IsEmptyList(List myList)
{
    if (myList.n == 0)
        return 0; //Danh sách rỗng
    else
        return 1; //Danh sách không rỗng
}
```

b. Chèn một phần tử vào danh sách

Khi chèn một phần tử vào danh sách sẽ xuất hiện hai khả năng:

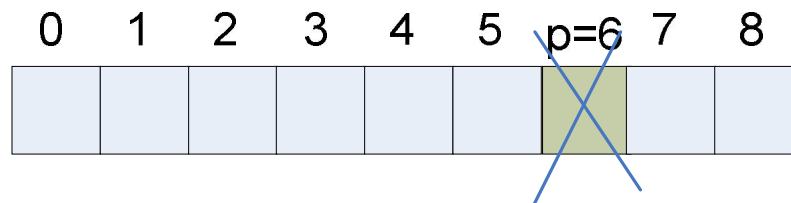
- Mảng đầy, tức là số phần tử n của danh sách bằng số ô nhớ tối đa được cấp MAX. Khi đó không thể thực hiện chèn thêm phần tử vào danh sách.
- Mảng chưa đầy. Khi đó ta phải kiểm tra vị trí p cần chèn có hợp lệ hay không. Nếu vị trí hợp lệ, ta tiến hành các bước sau:
 - + Dời các phần tử từ vị trí p đến cuối mảng ra sau 1 vị trí.
 - + Tăng độ dài của danh sách lên một đơn vị.
 - + Đặt phần tử cần chèn vào vị trí p .



c. Xóa một phần tử khỏi danh sách

Việc xóa một phần tử khỏi danh sách ta thực hiện ngược lại với giải thuật chèn phần tử vào danh sách. Trước hết, ta kiểm tra vị trí p cần xóa có hợp lệ hay không. Nếu vị trí hợp lệ, ta tiến hành các bước sau:

- + Dời các phần tử từ vị trí $p+1$ đến cuối mảng lên trước một vị trí
- + Giảm độ dài của danh sách xuống một đơn vị.



Phần cài đặt cho các giải thuật này và các phép toán cơ bản khác xin dành cho người đọc.

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

a. Giới thiệu

Việc sử dụng các kiểu dữ liệu tĩnh như kiểu số nguyên, kiểu số thực, kiểu ký tự, kiểu mảng, kiểu tập hợp, ... đều có thể giúp chúng ta giải quyết được hầu hết các bài toán. Nhưng việc sử dụng kiểu dữ liệu tĩnh trong nhiều trường hợp sẽ gây cho chúng ta nhiều khó khăn trong việc biểu diễn thông tin và kém hiệu quả. Sau đây là một vài ví dụ:

+ Khi biểu diễn thông tin số nhân viên trong một công ty, chúng ta cài đặt một danh sách bằng kiểu mảng, và quy định kích thước tối đa của mảng MAX=500. Nhưng thực tế, số nhân viên của công ty luôn luôn chỉ ở trong phạm vi từ 50 – 100 người. Như vậy, chúng ta đã cấp phát lãng phí một số lượng lớn vùng nhớ.

+ Biểu diễn thông tin của một lớp học ta khai báo cấu trúc SINHVIEN, và LOP như sau:

```
struct SINHVIEN
{
    char hoten[50];
    char quequan[50];
    char ngaysinh[8];
    .....
}
struct LOP
{
    char tenlop[20];
    int soSV;
    SINHVIEN dsSV[100]; //Danh sach sinh vien
}
```

Để biểu diễn thông tin danh sách các lớp học trong một trường, ta dùng một mảng kiểu LOP. Rõ ràng, kích thước mỗi phần tử của mảng là khá lớn vì thế nếu chúng ta thực hiện các thao tác trên mảng như sắp xếp, chèn thêm phần tử thì sẽ tốn nhiều chi phí.

+ Trong thực tế, một số đối tượng có thể được định nghĩa đệ quy như sau:

```
struct NGUOI
```

```
{
```

```
    char HoTen[50];
```

```
    char SoCMND[20];
```

```
    NGUOI cha, me;
```

```
}
```

Như chúng ta thấy, để biểu diễn thông tin về một người nào đó ta dùng cấu trúc NGUOI. Vì một người đều có thông tin về cha, mẹ nên bên trong cấu trúc NGUOI, ta phải dùng kiểu NGUOI để biểu diễn. Đó là các biểu diễn đệ quy. Nhưng làm sao để xác định được cấu trúc kiểu NGUOI? Đó chính là hạn chế của việc sử dụng kiểu dữ liệu tĩnh.

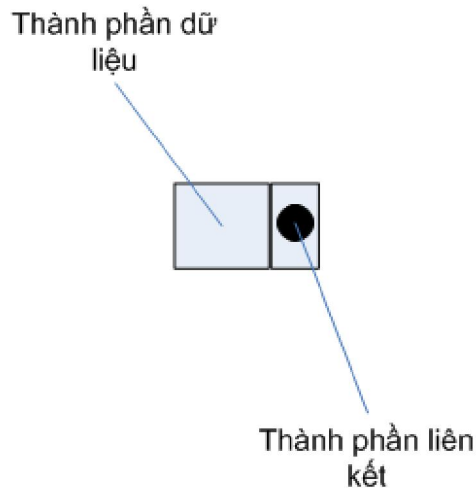
Do vậy, nhằm đáp ứng nhu cầu biểu diễn thông tin thực tế một cái chính xác và hiệu quả, người ta cần phải xây dựng các cấu trúc dữ liệu linh động hơn, có thể thay đổi kích thước, cấp phát và giải phóng bộ nhớ bất cứ khi nào cần thiết trong thời gian chương trình thực thi. Đó chính là các **cấu trúc dữ liệu động**. Phần tiếp theo ta sẽ nghiên cứu một cấu trúc dữ liệu động để cài đặt cho danh sách là danh sách liên kết (linked list).

b. Danh sách liên kết đơn

Danh sách liên kết đơn bao gồm một dãy các phần tử. Mỗi phần tử là một cấu trúc chứa hai thông tin sau:

+ Thành phần dữ liệu: Lưu trữ các thông tin về bản thân phần tử.

+ Thành phần liên kết: Lưu trữ địa chỉ của phần tử kế tiếp trong danh sách. Nếu là phần tử cuối của danh sách thì lưu trữ giá trị NULL.



Ta định nghĩa tổng quát như sau:

```
#define ElementType <KieuDuLieu>
typedef struct tagNode
{
    ElementType Info; //Thông tin của Node
    struct tagNode* pNext; //Con trỏ chỉ đến phần tử node tiếp theo
} Node;
```

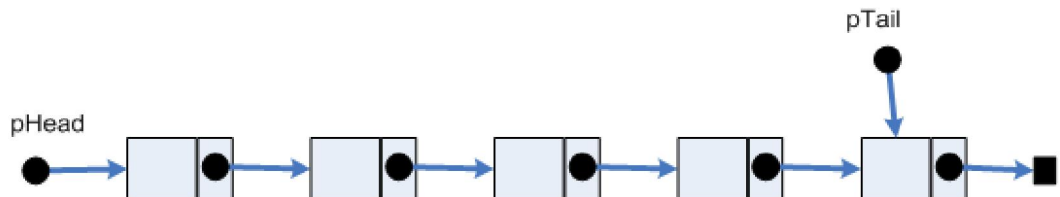
Ví dụ khai báo cấu trúc sinh viên như sau:

```
typedef struct tagSinhVien
{
    char HoTen[20]; } Phần thông tin
    int MSSV; } của Node
    struct tagSinhVien *pNext;
} SinhVien;
```

Như vậy danh sách liên kết đơn là một dãy các phần tử mà địa chỉ của bản thân mỗi phần tử được lưu ở phần tử liền trước nó. Vậy đối với phần tử đầu tiên của danh sách, địa chỉ của nó được lưu trữ ở đâu? Người ta đưa thêm vào một

biến kiểu con trỏ để lưu địa chỉ của phần tử đầu tiên trong danh sách, gọi là phần tử **pHead**. pHead là con trỏ cùng kiểu với các phần tử trong danh sách mà nó quản lý. Trong trường hợp giá trị pHead = NULL thì danh sách mà pHead quản lý là danh sách rỗng.

Về nguyên tắc ta chỉ cần phần tử pHead để quản lý danh sách. Khi có phần tử pHead, ta có thể truy suất đến tất cả các phần tử của danh sách thông qua giá trị pNext tại mỗi phần tử. Tuy nhiên, trong một số trường hợp chúng ta cần truy suất ngay đến phần tử cuối cùng của danh. Khi đó ta phải thực hiện việc truy suất từ đầu danh sách mới có thể lấy được địa chỉ của phần tử cuối danh sách. Điều này gây tốn kém chi phí. Vì vậy, người ta đưa thêm vào một biến lưu địa chỉ của phần tử cuối của danh sách, gọi là phần tử **pTail**.



Từ đây, ta xây dựng kiểu dữ liệu danh sách liên kết đơn (Singly linked list) như sau:

```
typedef struct
{
    Node*pHead;
    Node*pTail;
} List;
```

Sau đây, ta sẽ tìm hiểu và cài đặt các thao tác trên danh sách bằng danh sách liên kết đơn.

❖ Tạo một danh sách rỗng

```
void InitList(List &l)
{
```



```
l.pHead=NULL;

l.pTail=NULL;

}
```

❖ Tạo một phần tử mới

```
Node*MakeNode(ElementType x)
{
    Node*newNode;
    newNode=new Node;
    if(newNode == NULL)
    {
        cout<<"Khong the cap phat bo nho";
        return NULL;
    }
    p->Info=x;
    p->pNext=NULL;
    return p;
}
```

❖ Kiểm tra một danh sách rỗng

Danh sách là rỗng khi con trỏ pHead có giá trị NULL

```
int IsEmptyList(List l)
{
```

```

if(l.pHead == NULL)

    return 0;

    return 1;

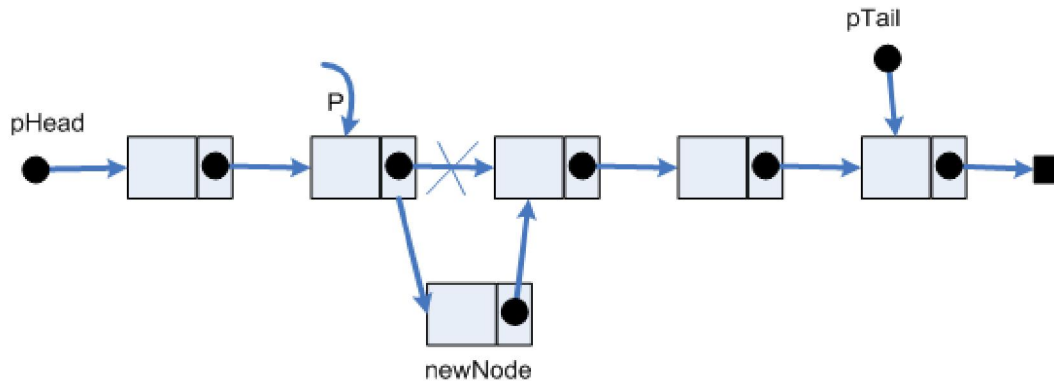
}

```

❖ Chèn một phần tử vào danh sách

Để chèn một phần tử vào danh sách liên kết đơn, ta cần phải xác định vị trí cần chèn. Vị trí đó có thể là đầu danh sách, giữa danh sách, hoặc cuối danh sách. Dưới đây, ta cài đặt hàm thực hiện chèn một phần tử vào sau một phần tử P xác định trong danh sách. Việc chèn ở đầu hoặc cuối danh sách sẽ làm thay đổi giá trị của con trỏ pHead và pTail. Phần này dành cho người đọc tự cài đặt.

Hàm sau đây thực hiện việc chèn một phần tử có giá trị X vào sau phần tử P xác định trong danh sách



Để chèn một phần tử có giá trị X vào trong danh sách liên kết đơn, ta cần cấp phát bộ nhớ cho phần tử đó, sau đó xác định các giá trị con trỏ nối kết giữa các phần tử.

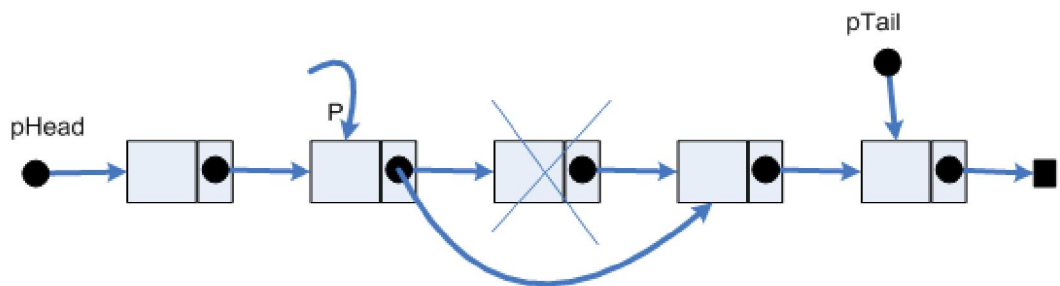
```

void InsertNode(ElementType x, Node* p, List& l)
{
    Node*newNode= MakeNode(x);//Tạo một phần tử mới
    newNode->pNext=p->pNext;
    p->pNext=newNode;
}

```

❖ Xóa một phần tử khỏi danh sách

Tương tự với chèn phần tử vào danh sách liên kết đơn, ta cần phải xác định của phần tử cần xóa trong danh sách. Sau khi thực hiện việc loại bỏ phần tử khỏi danh sách, ta cần giải phóng bộ nhớ cho danh sách đó. Nếu phần tử cần xóa nằm ở đầu hoặc cuối danh sách, thì sau khi xóa giá trị của các con trỏ pHead và pTail sẽ thay đổi. Sau đây ta cài đặt hàm xóa một phần tử nằm sau một phần tử P xác định trong danh sách.



```
void DeleleNode(Node*p, List& l)
{
    Node*delNode;
    if(p->pNext!=NULL)
    {
        delNode=p->pNext;//Phan tu can xoa
        p->pNext=delNode->pNext;
        delete delNode;
    }
}
```

❖ Tìm một phần tử trong danh sách

Để tìm kiếm một phần tử có giá trị X trong danh sách hay không, ta tiến hành tìm từ đầu danh sách (từ pHead) cho đến khi nào thấy phần tử đầu tiên có giá trị X, hoặc nếu không thấy thì trả về NULL.

```
Node* SearchNode(ElementType x, List l)
{
    Node*p;
    p=l.pHead;
```

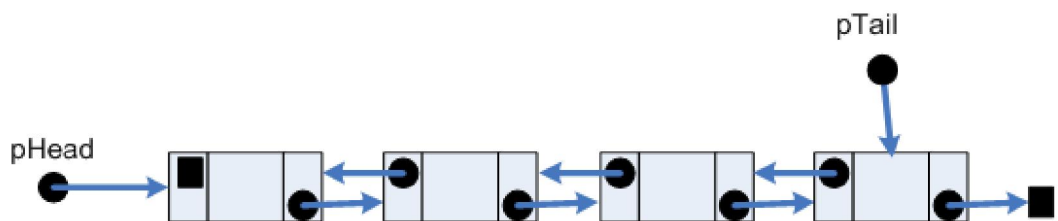
```

while(p!=NULL && p->Info!=x)
    p=p->pNext;
return p;
}

```

c. Danh sách liên kết kép

Danh sách liên kết kép là danh sách liên kết mà mỗi nút có hai thành phần liên kết: một thành phần lưu địa chỉ của phần tử đứng trước, một thành phần lưu địa chỉ của phần tử đứng sau.



Với danh sách liên kết kép, chúng ta có thể duyệt xuôi, hoặc duyệt ngược. Điều này giải quyết được hạn chế của danh sách liên kết đơn, khi mà chúng ta luôn phải duyệt từ đầu cho đến phần tử cần xét cho dù phần tử đó nằm gần cuối danh sách. Tuy nhiên, mỗi phần tử của danh sách liên kết kép cần tồn thêm một trường để lưu địa chỉ của phần tử đứng trước nó so với danh sách liên kết đơn. Sau đây ta định nghĩa một nút của danh sách liên kết kép, thành phần lưu địa chỉ của phần tử trước nó là pPrev, thành phần lưu địa chỉ của phần tử sau nó là pNext.

```
#define ElementType <Kiểu dữ liệu>
```

```
typedef struct tagNode
```

```
{
```

```
    ElementType Info;
```

```
    struct tagNode* pPrev;//Lưu địa chỉ của phần tử đứng trước
```

```
    struct tagNode* pNext;//Lưu địa chỉ của phần tử đứng sau
```

```
} DNode;
```

Ta định nghĩa kiểu dữ liệu danh sách liên kết kép như sau:

```
typedef struct tagDList
{
    DNode* pHead;
    DNode*pTail;
}DList;
```

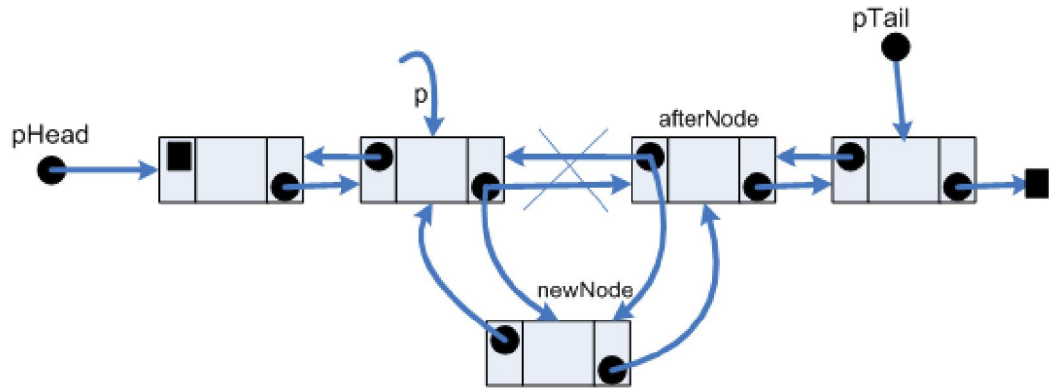
Sau đây ta cài đặt một số phép toán cơ bản trên danh sách liên kết kép

❖ Tạo một phần tử mới

```
DNode*MakeNode(ElementType x)
{
    DNode*p;
    p= new DNode;
    if(p==NULL)
    {
        cout<<"Khong the cap phat bo nho";
        return NULL;
    }
    p->Info=x;
    p->pPrev=NULL;
    p->pNext=NULL;
    return p;
}
```

❖ Chèn một phần tử vào danh sách

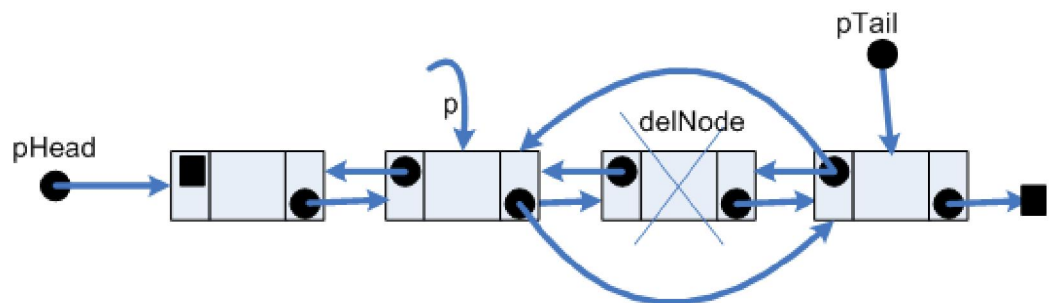
Phương pháp chèn phần tử vào danh sách liên kết kép cũng giống như thực hiện trên danh sách liên kết đơn, chỉ khác nhau ở việc xác định các mối liên kết giữa các phần tử mà thôi. Sau đây ta cài đặt hàm thực hiện chèn phần tử có giá trị X vào sau phần tử p xác định trong danh sách.



```
void InsertNode(ElementType x, DNode* p, DList& l)
{
    DNode* newNode= MakeNode(x); //Tạo một phần tử mới
    DNode* afterNode=p->pNext;
    newNode->pNext=afterNode;
    newNode->pPrev=p;
    p->pNext=newNode;
    if(afterNode!=NULL)
        afterNode->pPrev=newNode;
}
```

❖ Xóa một phần tử khỏi danh sách

+ Hàm xóa một phần tử đứng sau phần tử p xác định trong danh sách

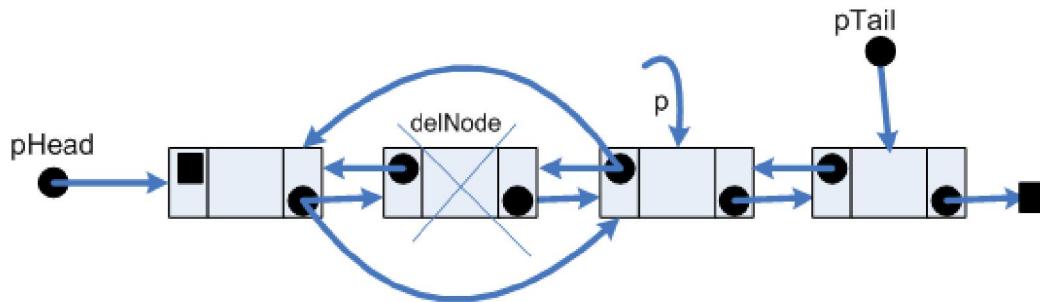


```

void DeleleAfterNode(DNode*p, DList& l)
{
    DNode*delNode;
    if(p->pNext!=NULL)
    {
        delNode=p->pNext;//Phan tu can xoa
        p->pNext=delNode->pNext;
        if(delNode->pNext != NULL)
            delNode->pNext->pPrev = p;
        delete delNode;
    }
}

```

+ Hàm xóa một phần tử đứng trước phần tử p xác định trong danh sách



```

void DeleleBeforNode(DNode*p, DList& l)
{
    DNode*delNode;
    if(p->pPrev!=NULL)
    {
        delNode=p->pPrev;//Phan tu can xoa
        p->pPrev=delNode->pPrev;
        if(delNode->pPrev != NULL)
            delNode->pPrev->pNext = p;
        delete delNode;
    }
}

```

Trong các hàm cài đặt cho danh sách liên kết trên, ta không xét đến việc con trỏ pHead, pTail có thể bị thay đổi do quá trình thêm hoặc xóa trên danh sách liên kết. Khi cài đặt cụ thể, chúng ta cần phải cập nhật lại thông tin này.

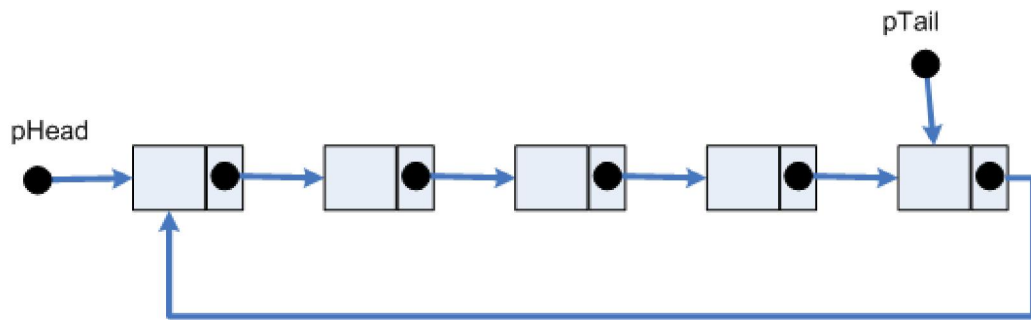
d. Một số danh sách liên kết khác

Ngoài hai dạng danh sách liên kết đã nghiên cứu, còn có các dạng danh sách liên kết khác như: danh sách liên kết vòng, danh sách có nhiều mỗi liên kết, danh sách tổng quát. Trong phạm vi của giáo trình, chúng ta không đi sâu vào nghiên cứu các loại danh sách liên kết này mà chỉ nêu định nghĩa, phân cài đặt các thuật toán dành cho người đọc.

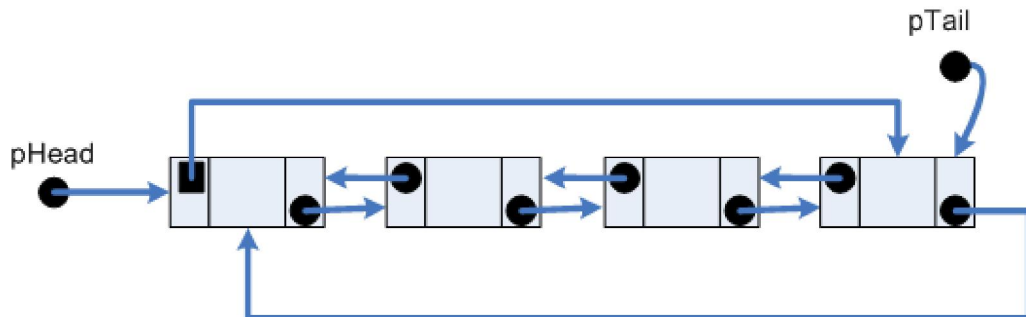
❖ Danh sách liên kết vòng

Danh sách liên kết vòng là một danh sách (đơn hoặc kép) mà phần tử cuối của danh sách trở tới phần tử đầu của nó.

+ Danh sách liên kết đơn vòng

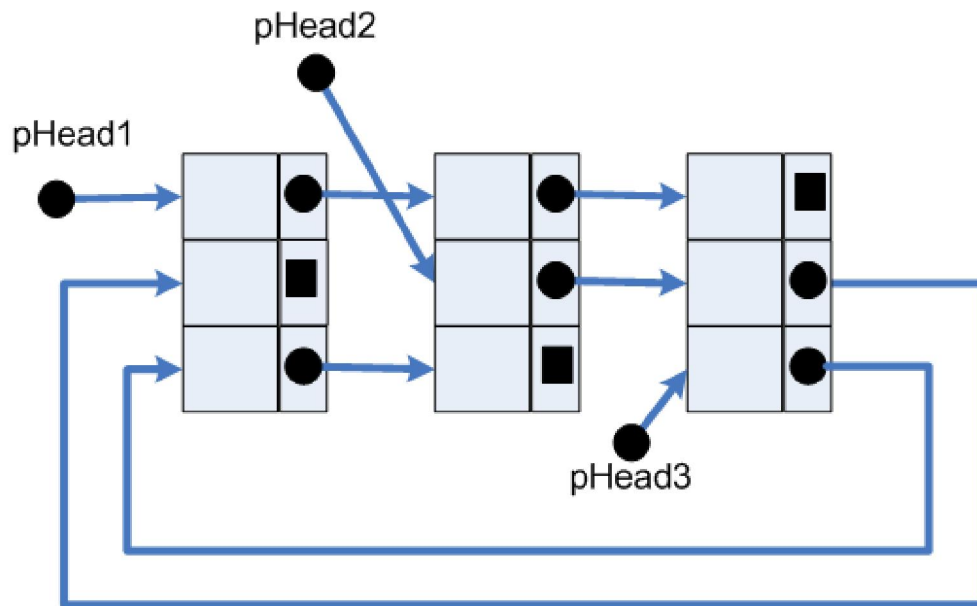


+ Danh sách liên kết kép vòng



❖ Danh sách có nhiều mỗi liên kết

Danh sách có nhiều mối liên kết là danh sách mà mỗi phần tử có nhiều mối liên kết với các phần tử khác theo từng loại thông tin mà nó lưu trữ.



III. Bài tập chương 4

1. Viết chương trình cho phép xây dựng một danh sách liên kết đơn các phần tử kiểu số nguyên. Cài đặt hoàn chỉnh hàm thực hiện thêm một phần tử có giá trị x vào danh sách.
2. Cài đặt hoàn chỉnh hàm thực hiện tìm kiếm và xóa một phần tử có giá trị x khỏi danh sách liên kết đơn.
3. Viết chương trình sử dụng danh sách liên kết đơn nhập vào danh sách sinh viên của một lớp. Thực hiện các thao tác thêm, xóa, sửa, tìm kiếm thông tin sinh viên thông qua khóa là MSSV.
4. Viết chương trình cho phép xây dựng một danh sách liên kết kép các phần tử kiểu số nguyên. Cài đặt các hàm thao tác trên danh sách.
5. Xây chương trình mô phỏng quá trình thao tác (thêm, xóa, sửa, sắp xếp) trên một danh sách liên kết đơn.

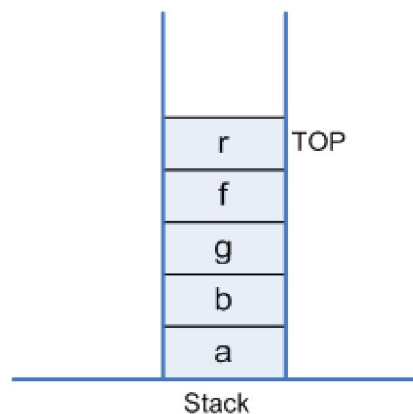
CHƯƠNG 5. NGĂN XẾP (STACK)

Chương này sẽ trình bày về ngăn xếp, một kiểu dữ liệu trừu tượng đơn giản nhưng lại hết sức quan trọng. Chúng ta sẽ tìm hiểu khái niệm cơ bản về ngăn xếp, sau đó hiện thực bằng các cách khác nhau.

I. Định nghĩa ngăn xếp

Ngăn xếp là một danh sách (list) được cài đặt nhằm sử dụng cho các ứng dụng cần xử lý đảo ngược. Trong cấu trúc dữ liệu ngăn xếp, tất cả các thao tác thêm, xóa một phần tử đều phải thực hiện ở một đầu danh sách, đầu này gọi là đỉnh (Top) của ngăn xếp.

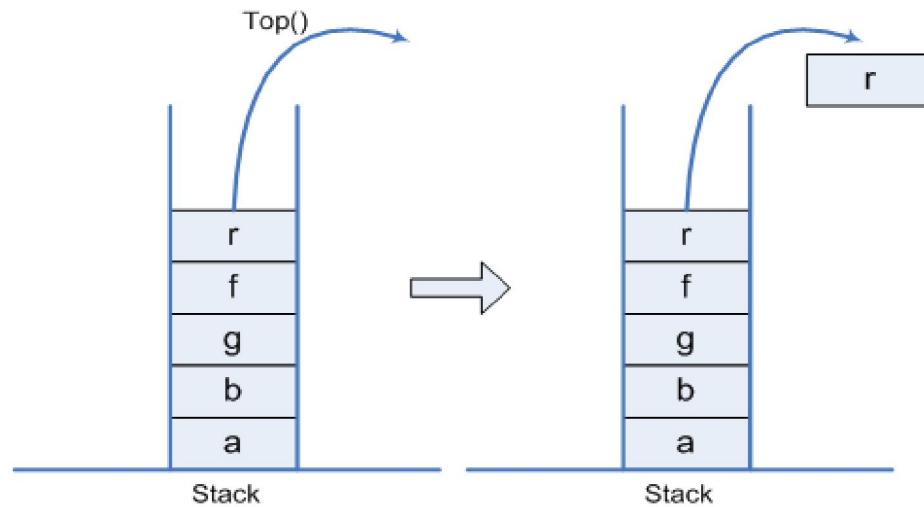
Có thể hình dung ngăn xếp thông qua hình ảnh một chồng đĩa đặt trên bàn. Nếu muốn thêm vào một đĩa, người ta phải đặt nó lên trên đỉnh. Nếu lấy ra một đĩa, người ta cũng phải lấy đĩa ở trên đỉnh chồng. Như vậy, ngăn xếp là một cấu trúc dữ liệu trừu tượng có tính chất “vào sau ra trước” (Last in first out – **LIFO**) hay “vào trước ra sau” (First in last out – **LILO**).



II. Một số phép toán trên ngăn xếp

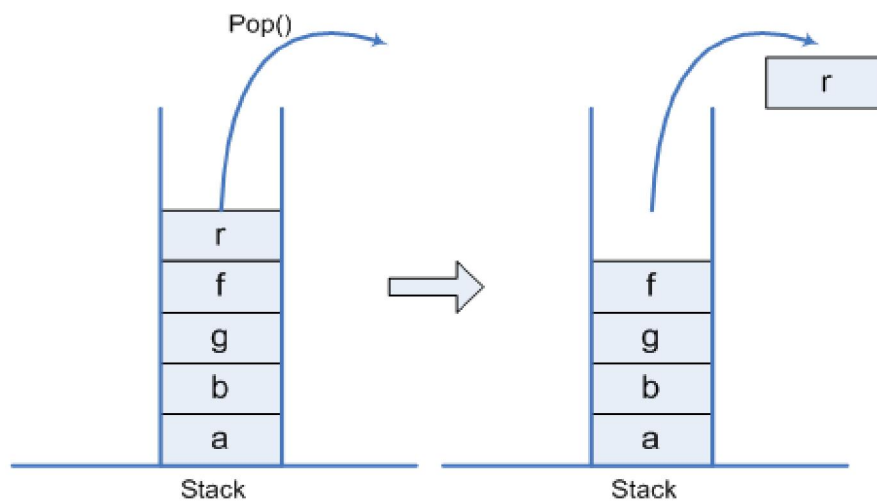
a. Lấy thông tin phần tử đầu của ngăn xếp (Top())

Là thao tác lấy giá trị của phần tử nằm ở đỉnh (Top) của ngăn xếp. Nếu ngăn xếp rỗng, trả về thông báo lỗi.



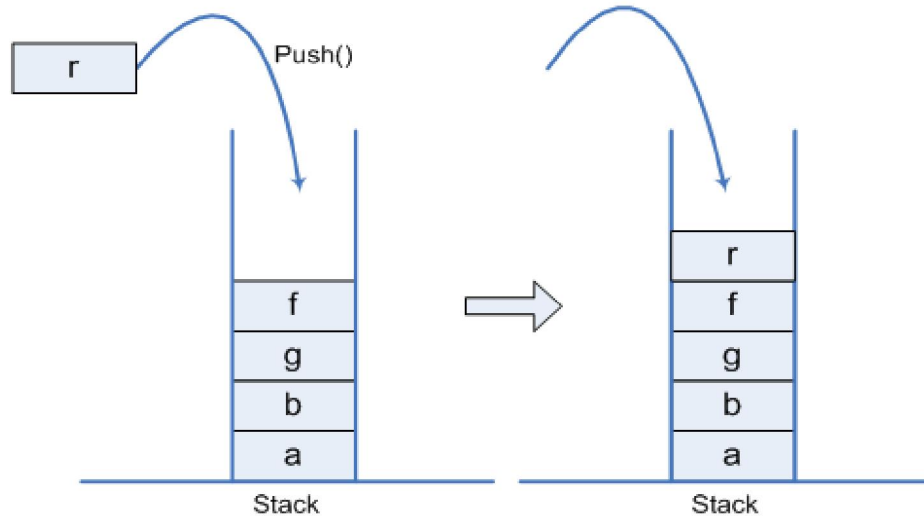
b. Trích hủy phần tử ra khỏi ngăn xếp (Pop())

Là thao tác lấy giá trị của phần tử nằm ở đỉnh ngăn xếp, đồng thời xóa phần tử này trong ngăn xếp.



c. Thêm một phần tử vào ngăn xếp (Push())

Là thao tác thêm một phần tử vào đầu ngăn xếp.



d. Kiểm tra ngăn xếp rỗng (IsEmptyStack())

Là thao tác kiểm tra xem hàng đợi có rỗng hay không.

III. Cài đặt ngăn xếp

Ngăn xếp là một dạng của danh sách, vì thế ta có thể cài đặt ngăn xếp theo các cách đã cài đặt danh sách là: Cài đặt bằng mảng, hoặc cài đặt bằng danh sách liên kết.

1. Cài đặt bằng mảng

Có thể tạo ngăn xếp bằng cách khai báo một mảng một chiều với kích thước tối đa là MAXSIZE nào đó. Như vậy, ngăn xếp có thể chứa tối đa MAXSIZE phần tử được đánh chỉ số từ 0 đến MAXSIZE – 1. Ta sử dụng một biến **nTop** để định vị phần tử nằm ở đỉnh ngăn xếp. Chúng ta khai báo cấu trúc dữ liệu ngăn xếp như sau:

```
#define MAXSIZE <Kích thước tối đa>
```

```
#define ElementType <Kiểu dữ liệu>
```

```
ElementType Stack[MAXSIZE];
```

```
int nTop;
```

Sau đây ta cài đặt các thao tác trên ngăn xếp

a. Tạo ngăn xếp rỗng

```
void InitStack()
{
    nTop=0;
}
```

b. Kiểm tra ngăn xếp rỗng hay không

```
int IsEmptyStack()
{
    if(nTop==0)
        return 1;
    return 0;
}
```

c. Kiểm tra ngăn xếp đầy hay không

```
int IsFullStack()
{
    if(nTop>=MAXSIZE)
        return 1;
    return 0;
}
```

d. Thêm một phần tử vào ngăn xếp

```
void Push(ElementType x)
{
    if(!IsFullStack())
    {
        Stack[nTop]=x;
        nTop++;
    }
    else
        cout<<"\nStack đầy";
}
```

e. Lấy thông tin phần tử ở đỉnh ngăn xếp

```
ElementType Top()
{
    ElementType x;
    if(!IsEmptyStack())
    {
        x=Stack[nTop-1];
        return x;
    }
    else
        cout<<"\nStack rong";
}
```

f. Lấy thông tin và hủy phần tử ở đỉnh ngăn xếp

```
ElementType Pop()
{
    ElementType x;
    if(!IsEmptyStack())
    {
        x=Stack[nTop-1];
        nTop--;
        return x;
    }
    else
        cout<<"\nStack rong";
}
```

❖ Nhận xét

Ưu điểm của việc cài đặt ngăn xếp bằng mảng là đơn giản và dễ hiểu. Tuy nhiên, phương pháp này có hạn chế lớn là giới hạn kích thước của ngăn xếp. Giá trị MAXSIZE được xác định ngay từ ban đầu có thể gây ra tình trạng hoặc không đủ bộ nhớ hoặc lãng phí bộ nhớ. Phần tiếp theo ta sẽ sử dụng danh sách liên kết đơn để cài đặt ngăn xếp.

2. Cài đặt bằng danh sách liên kết đơn

Danh sách liên kết đơn là kiểu dữ liệu rất phù hợp để cài đặt ngăn xếp vì ta có thể thao tác dễ dàng và nhanh chóng ở đầu danh sách. Chúng ta khai báo cấu trúc dữ liệu ngăn xếp như sau:

```
#define ElementType <Kiểu dữ liệu>
```

```
List Stack;
```

Sau đây ta cài đặt các thao tác trên ngăn xếp. Khi cài đặt ta sử dụng lại những hàm đã xây dựng cho kiểu dữ liệu danh sách liên kết.

a. Tạo ngăn xếp rỗng

```
void InitStack()
{
    InitList(Stack);
}
```

b. Kiểm tra ngăn xếp rỗng

```
int IsEmptyStack()
{
    return IsEmptyList(Stack);
}
```

c. Thêm một phần tử vào ngăn xếp

Để thêm một phần tử vào ngăn xếp, ta chỉ việc gọi hàm chèn một phần tử vào đầu của danh sách (Hàm này dành cho người đọc tự xây dựng).

```
void Push(ElementType x)
{
    InsertHead(Stack, x); //Hàm chèn một phần tử vào đầu danh sách
}
```


d. Lấy thông tin phần tử ở đỉnh ngăn xếp

```

ElementType Top()
{
    if(!IsEmptyStack())
        return Stack.pHead->Info;
    return NULLDATA; //NULLDATA là giá trị NULL của kiểu dữ liệu ElementType
}

```

e. Lấy thông tin và hủy phần tử ở đỉnh ngăn xếp

Để lấy thông tin và hủy phần tử ở đỉnh ngăn xếp, ta chỉ việc gọi hàm xóa phần tử đầu của danh sách (hàm này dành cho người đọc cài đặt).

```

ElementType Pop()
{
    if(!IsEmptyStack())
    {
        ElementType x=Stack.pHead;
        RemoveFirst(Stack);
        return x;
    }
    return NULLDATA; //NULLDATA là giá trị NULL của kiểu dữ liệu ElementType
}

```

IV. Ứng dụng ngăn xếp để loại bỏ đệ quy của chương trình

Xét một chương trình con đệ quy $Q(x)$. Khi $Q(x)$ được gọi từ chương trình chính, ta nói nó đang thực hiện ở mức 1. Khi thực hiện ở mức 1, nó gọi đến chính nó, ta nói $Q(x)$ đang thực hiện ở mức 2. Tiếp tục như vậy, cho đến một mức n nào đó thì chương trình sẽ không gọi đệ quy tiếp nữa. Rõ ràng, mức n phải được thực hiện xong thì chương trình mới quay về để thực hiện mức $n-1$. Mức $n-1$ sẽ sử dụng kết quả ở mức n để thực hiện.

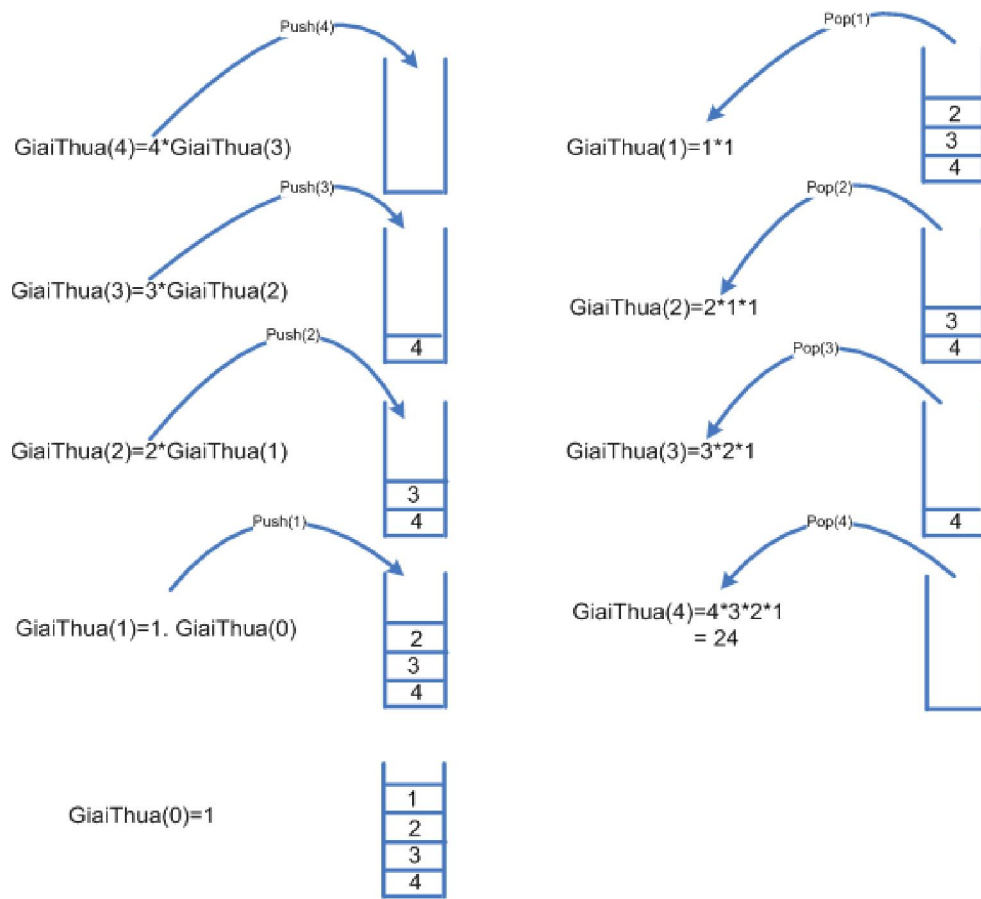
Khi chương trình con đi từ mức i vào mức $i+1$ thì các biến cục bộ ở mức i và địa chỉ mã lệnh đang giảng giờ phải được lưu trữ để khi chương trình quay trở lại sẽ sử dụng để thực hiện tiếp. Tính chất này gợi ý cho ta dùng một ngăn xếp để

lưu trữ các biến cục bộ của từng mức, sau đó sẽ truy suất ngược lại nhờ tính chất của ngăn xếp, khi đó ta sẽ loại bỏ được việc gọi đệ quy.

Sau đây ta sử dụng ngăn xếp để khử đệ quy của bài toán tính giai thừa và bài toán tháp Hà Nội.

1. Bài toán tính giai thừa

Tính giai thừa là bài toán có thể giải quyết bằng giải thuật đệ quy theo công thức: $n! = n * (n-1)!$. Sau đây ta mô phỏng việc lưu trữ các biến cục bộ vào ngăn xếp khi tính $4!$.



Để tính $4!$, vì $4! = 4 * 3!$ nên ta cần phải tính $3!$. Khi thực hiện $3!$, ta cần phải lưu trữ giá trị 4 vào ngăn xếp. Để tính $3!$, ta tính $2!$ và lưu giá trị 3 vào ngăn xếp. Công việc tiếp tục như vậy cho đến khi tính $0!$. Khi tính được $0! = 1$, ta chỉ việc nhân lần lượt với các giá trị được lấy ra từ hàng đợi.

Sau đây là hàm cài đặt tính giai thừa sử dụng hàng đợi

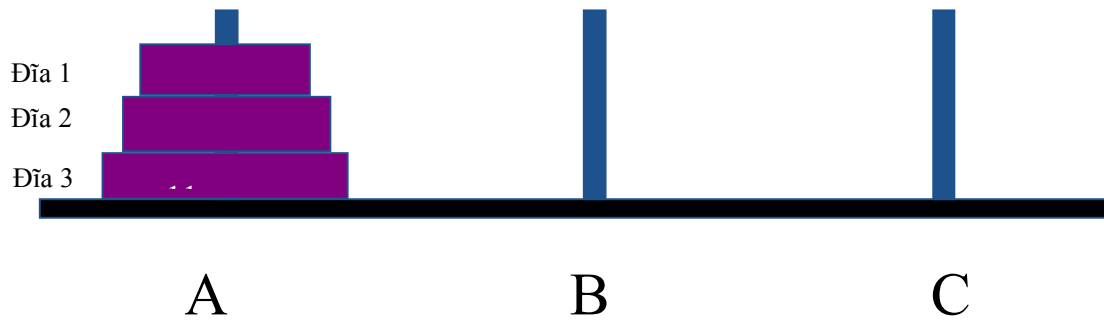
```
#define ElementType int

List*Stack;

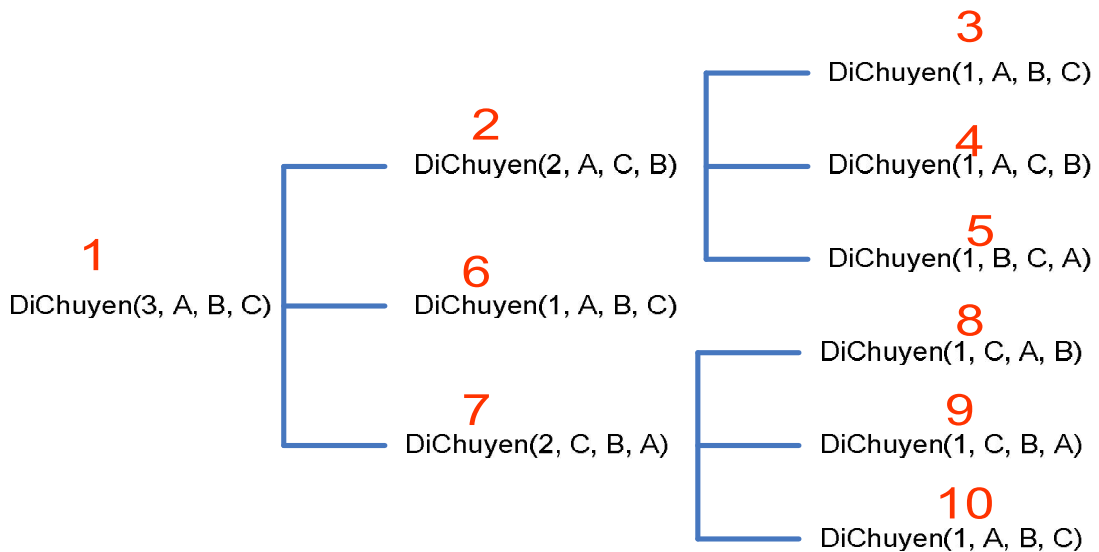
int GiaiThua(int a)
{
    int kq;
    if(a<0) return -1;
    InitStack(st);//Tạo ngăn xếp rỗng
    while(a != 0)
    {
        Push(a);
        a = a - 1;
    }
    kq=1;//0! = 1
    while(!IsEmptyStack())
    {
        int temp=Pop();
        kq=kq*temp;
    }
    return kq;
}
```

2. Bài toán Tháp Hà Nội

Bài toán Tháp Hà Nội đã được trình bày ở chương 2 và được cài đặt đệ quy.



Sau đây là thứ tự gọi các thủ tục **DiChuyen()** khi thực hiện chuyển 3 đĩa từ cột A sang cột B với cột C là cột trung gian.



Để di chuyển 3 đĩa từ cột A sang cột B với cột trung gian C (Thủ tục $DiChuyen(3,A,B,C)$), ta cần thực hiện lần lượt 3 công việc:

+ Công việc 2: Di chuyển 2 đĩa từ cột A sang cột C với cột trung gian B: $DiChuyen(2,A,C,B)$.

+ Công việc 6: Di chuyển 1 đĩa từ cột A sang cột B với cột trung gian C: $DiChuyen(1,A,B,C)$.

+ Công việc 7: Di chuyển 2 đĩa từ cột C sang cột B với cột trung gian A: $DiChuyen(2,C,B,A)$.

Tuy nhiên để thực hiện công việc 2, cần phải thực hiện 3 công việc khác (đánh số 3, 4, 5). Các công việc này sẽ được thực hiện trước công việc 2. Các số đánh phía trên các hàm trong hình trên thể hiện thứ tự gọi các thủ tục. Như vậy, khi gọi thực các công việc đánh số 3, 4, 5, ta cần phải lưu trữ lại thông tin của các công việc 6 và 7. Tương tự cho các nhánh khác của cây trong hình trên.

Sau đây ta cài đặt giải thuật không đệ quy để giải quyết bài toán Tháp Hà Nội, ta định nghĩa cấu trúc **ThuTuc** để lưu thông tin về một lần thực hiện di chuyển các đĩa.

```
typedef struct
{
    int N; //Số đĩa cần di chuyển
    char A; //Cột nguồn
    char B; //Cột đích
    char C; //Cột trung gian
} ThuTuc;

// Chương trình con MOVE không đệ qui
void Move(ThuTuc X){
    ThuTuc Temp, Temp1;
    MyStack St;
    InitStack(St);
    Push(X,St);
    do{
        Temp= Pop(St);
        if (Temp.N==1)
            printf("Chuyen 1 dia tu %c sang %c\n",Temp.A,Temp.B);
        else{
            // Luu cho loi gọi DiChuyen(n-1,C,B,A)
            Temp1.N=Temp.N-1;
```

```

Temp1.A=Temp.C;
Temp1.B=Temp.B;
Temp1.C=Temp.A;
Push(Temp1,St);
//Luu cho loi goi DiChuyen(1,A,B,C)
Temp1.N=1;
Temp1.A=Temp.A;
Temp1.B=Temp.B;
Temp1.C=Temp.C;
Push(Temp1,St);
//Luu cho loi goi DiChuyen(n-1,A,C,B)
Temp1.N=Temp.N-1;
Temp1.A=Temp.A;
Temp1.B=Temp.C;
Temp1.C=Temp.B;
Push(Temp1,St);
}
} while (!IsEmptyStack(St));
}

```

3. Một số ứng dụng khác của ngăn xếp

Trong thực tế, ngăn xếp được sử dụng rất nhiều. Chẳng hạn, trong trình biên dịch hay trình thông dịch, khi thực hiện các thủ tục, ngăn xếp được dùng để lưu môi trường (các biến, địa chỉ, ...) của các thủ tục. Ngăn xếp cũng được sử dụng trong việc giải quyết các bài toán cần lưu vết như các bài toán trong lý thuyết đồ thị, trí tuệ nhân tạo.

V. Bài tập chương 5

1. Hãy nêu hai ví dụ thực tế có sử dụng cấu trúc dữ liệu ngăn xếp.
2. Cài đặt các bài tập ở chương 2 bằng cách sử dụng ngăn xếp.
3. Viết chương trình mô phỏng việc lưu trữ và thao tác trên hàng đợi. Dữ liệu đưa vào là các số nguyên.
4. Viết chương trình mô phỏng bài toán tháp Hà Nội.

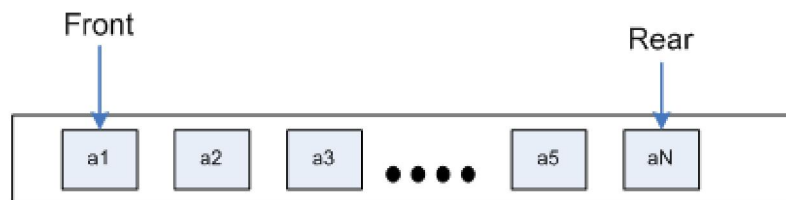
CHƯƠNG 6. HÀNG ĐỢI (QUEUE)

Hàng đợi là cấu trúc dữ liệu mô phỏng theo hình thức xếp hàng chờ đợi của con người. Cùng với ngăn xếp, hàng đợi là một trong những cấu trúc dữ liệu đơn giản và rất quan trọng. Chương này trình bày những tính chất của hàng đợi, nghiên cứu cách ứng dụng của nó và hiện thực theo các cách khác nhau.

I. Định nghĩa hàng đợi

Hàng đợi là một danh sách mà việc thêm một phần tử chỉ được phép thực hiện ở một đầu của danh sách gọi là cuối hàng đợi (REAR), và việc hủy chỉ được thực hiện ở đầu còn lại gọi là đầu hàng đợi (FRONT).

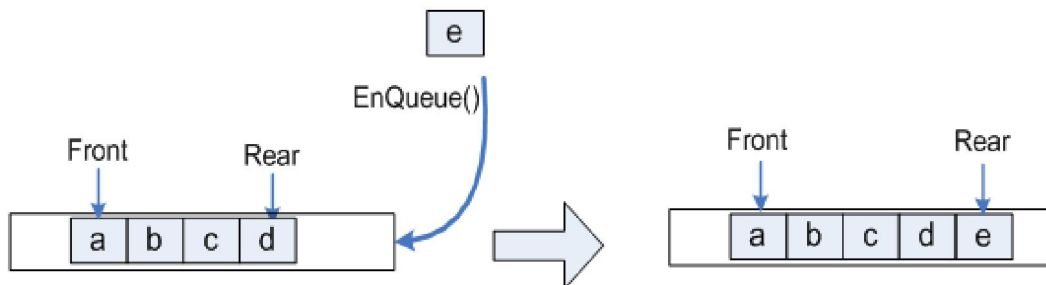
Ta có thể hình dung hình ảnh của hàng đợi thông qua việc xếp hàng chờ mua vé xem phim tại. Người nào vào hàng trước sẽ mua được vé trước và ra khỏi hàng trước. Vì vậy, hàng đợi có tính chất “vào trước ra trước” (First in first out – FIFO).



II. Một số phép toán trên hàng đợi

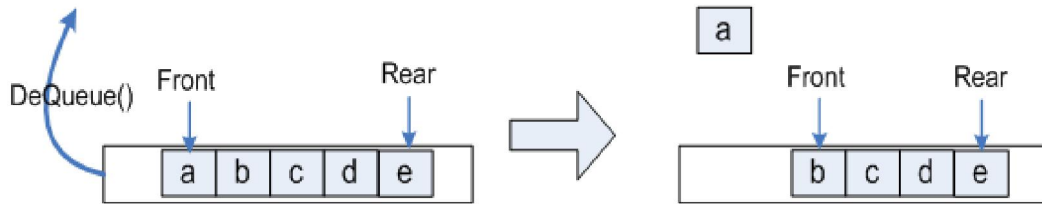
a. EnQueue

Là thao tác thêm một phần tử vào cuối hàng đợi.



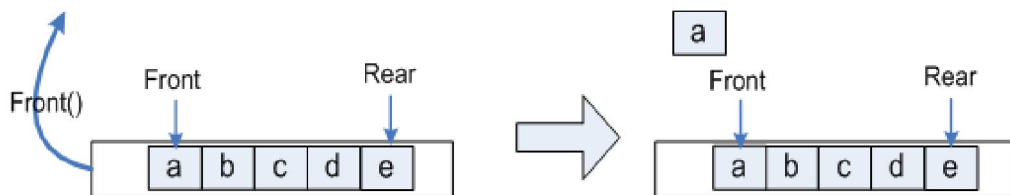
b. DeQueue

Là thao tác lấy thông tin và hủy phần tử ở đầu hàng đợi



c. Front

Là thao tác lấy thông tin phần tử ở đầu hàng đợi



d. IsEmptyQueue

Là thao tác kiểm tra hàng đợi có rỗng hay không.

III. Cài đặt hàng đợi

Giống kiểu dữ liệu ngăn xếp, ta có thể cài đặt hàng đợi bằng mảng hoặc danh sách liên kết.

1. Cài đặt bằng mảng

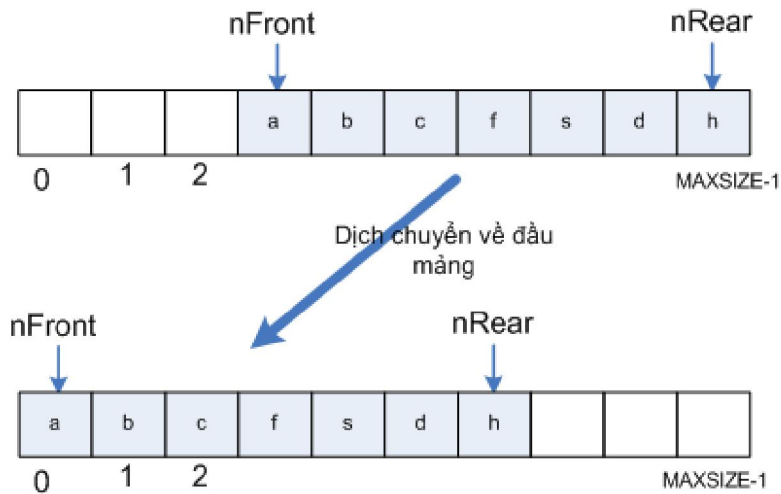
Ta có thể tạo một hàng đợi bằng cách khai báo một mảng một chiều với kích thước tối đa là $MAXSIZE$ nào đó. Như vậy, hàng đợi sẽ có thể chứa tối đa $MAXSIZE$ phần tử được đánh chỉ số từ 0 đến $MAXSIZE - 1$. Ta sử dụng biến **nFront** để định vị phần tử nằm ở đầu hàng đợi, biến **nRear** để định vị phần tử nằm ở cuối hàng đợi. Ngoài ra, ta cần dùng một giá trị đặt biệt để gán cho những ô còn trống trên hàng đợi. Ta ký hiệu giá trị này là **NULLDATA**.

Giả sử hàng đợi có n phần tử, ta có $nFront=0$, $nRear=n-1$. Khi thêm một phần tử $nRear$ tăng lên 1, khi xóa một phần tử $nFront$ tăng lên 1. Như vậy, dữ liệu lưu trên hàng đợi có khuynh hướng đi về cuối. Đến một lúc nào đó, mặc dù mảng còn nhiều ô trống phía trước $nFront$ nhưng ta không thể thêm được vào hàng đợi

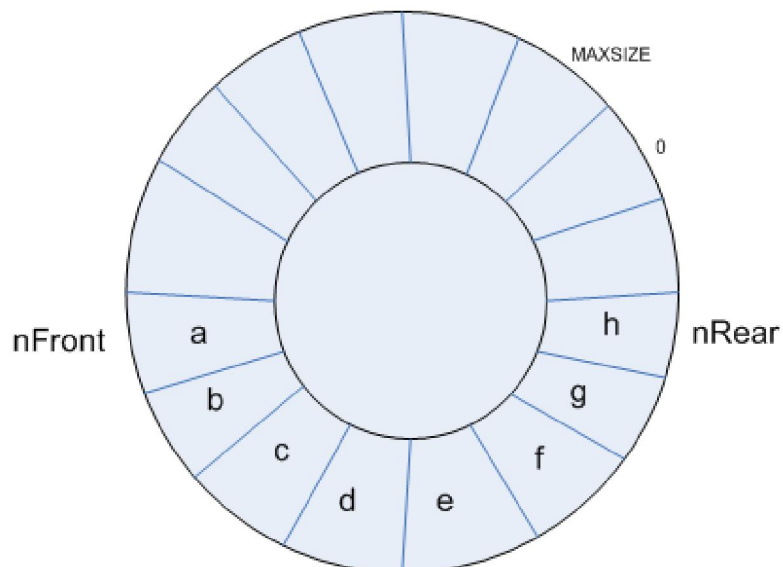
nữa vì $nRear = MAXSIZE - 1$. Trường hợp này ta gọi là hàng đợi bị tràn. Khi mà toàn bộ hàng đợi đã chứa các phần tử, ta gọi là hàng đợi bị đầy.

Trong trường hợp hàng đợi bị tràn, ta có thể giải quyết bằng một trong hai cách sau:

+ Tịnh tiến toàn bộ nội dung lên phía đầu mảng $nFront$ vị trí. Trong trường hợp này $nFront$ luôn luôn nhỏ hơn $nRear$.



+ Thực hiện quay vòng, xem như đầu mảng và cuối mảng là hai phần tử kề nhau. Khi vị trí cuối mảng $MAXSIZE - 1$ đã có nội dung lưu trữ, nếu thêm vào một phần tử nữa thì ta thêm vào vị trí 0, và tiếp tục như vậy cho đến khi mảng đầy. Trong trường hợp này $nFront$ có thể lớn hơn $nRear$.



Chúng ta khai báo cấu trúc dữ liệu hàng đợi như sau:

```
#define MAXSIZE <Kích thước tối đa>
```

```
#define ElementType <Kiểu dữ liệu>
```

```
ElementType Queue[MAXSIZE];
```

```
int nFront, nRear;
```

Sau đây ta cài đặt các thao tác trên hàng đợi theo phương pháp tịnh tiến. Việc cài đặt phương pháp quay vòng dành cho người đọc.

a. Khởi tạo hàng đợi rỗng

```
void InitQueue()
{
    nFront=nRear=0;
    for(int i=0;i<MAXSIZE;i++)
        Queue[i]=NULLDATA;
}
```

b. Kiểm tra hàng đợi rỗng

```
int IsEmptyQueue()
{
    if(Queue[nFront] ==NULLDATA)
        return 1;
    return 0;
}
```

c. Kiểm tra hàng đợi đầy

```
int IsFullQueue()
{
    if(nRear-nFront+1==MAXSIZE)
        return 1;
    return 0;
}
```

d. Thêm một phần tử vào cuối hàng đợi

```
void EnQueue(ElementType x)
{
    if(!IsFullQueue()) //Kiểm tra hàng đợi đầy
    {
        if(nRear==MAXSIZE - 1)//Kiểm tra hàng đợi tràn
        {
            //Tịnh tiến các phần tử
            for(int i=nFront;i<nRear;i++)
                Queue[i-nFront] = Queue[i];
            //Xác định vị trí nRear mới
            nRear=MAXSIZE - nFront;
            nFront=0;
        }
        //Lưu nội dung mới
        nRear=nRear+1;
        Queue[nRear]=x;
    }
    else
        cout<<"\nHang doi day!";
}
```

e. Lấy thông tin phần tử đầu hàng đợi

```
ElementType Front()
{
    if(!IsEmptyQueue())
        return Queue[nFront];
    return NULLDATA;
}
```

f. Lấy thông tin và hủy phần tử ở đầu hàng đợi

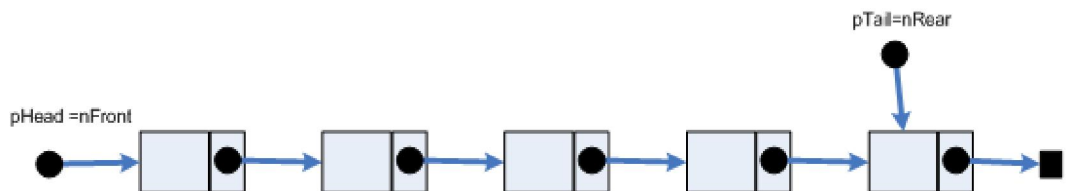
```

ElementType DeQueue()
{
    if(!IsEmptyQueue())
    {
        ElementType x;
        x=Queue[nFront];
        nFront=nFront + 1;
        if(nFront>nRear)
            InitQueue();//Đặt lại hàng đợi rỗng
        return x;
    }
    return NULLDATA;
}

```

2. Cài đặt bằng danh sách liên kết đơn

Danh sách liên kết đơn là kiểu dữ liệu rất phù hợp để cài đặt hàng đợi vì ta có thể thêm phần tử vào cuối và hủy phần tử đầu của danh sách một cách dễ dàng và nhanh chóng. pHead của danh sách là nFront và pTail của danh sách là nRear của hàng đợi.



Chúng ta khai báo cấu trúc dữ liệu hàng đợi như sau:

```
#define ElementType <Kiểu dữ liệu>
```

```
List Queue;
```

Sau đây là các thao tác trên hàng đợi. Trong đó ta sẽ không cài đặt lại những thao tác đã được xây dựng cho danh sách liên kết đơn.

a. Tạo hàng đợi rỗng

```
void InitQueue()
{
    InitList(Queue);
}
```

b. Kiểm tra hàng đợi rỗng

```
int IsEmptyQueue()
{
    return IsEmptyList(Queue);
}
```

c. Thêm một phần tử vào cuối hàng đợi

Thêm một phần tử vào cuối hàng đợi, chỉ cần gọi hàm chèn phần tử vào cuối danh sách `InsertTail()`, hàm này dành cho người đọc tự cài đặt.

```
void EnQueue(ElementType x)
{
    InsertTail(Queue, x);
}
```

d. Lấy thông tin phần tử ở đầu hàng đợi

```
ElementType Front()
{
    if(!IsEmptyQueue())
        return Queue.pHead->Info;
    return NULLDATA;
}
```

e. Lấy thông tin và hủy phần tử ở đầu hàng đợi

Để hủy phần tử ở đầu hàng đợi, chỉ việc gọi hàm hủy phần tử đầu danh sách `RemoveFirst()`, hàm này dành cho người đọc tự cài đặt.

```
ElementType DeQueue()
{
    if(!IsEmptyQueue())
    {
        ElementType x;
        x=Queue.pHead->Info;
        RemoveFirst(Queue);
        return x;
    }
    return NULLDATA;
}
```

IV. Các ứng dụng của hàng đợi

Hàng đợi là cấu trúc dữ liệu được sử dụng khá phổ biến trong thiết kế giải thuật. Những ứng dụng cần quản lý dữ liệu, phân phối hoạt động tiến trình,...theo tính chất vào trước ra trước đều có thể sử dụng hàng đợi. Chẳng hạn bài toán trong lý thuyết đồ thị như duyệt đồ thị theo chiều rộng, bài toán “sản xuất và tiêu thụ” ứng dụng trong các bài toán song song, xử lý các lệnh trong máy tính.

V. Bài tập chương 6

1. Hãy trình bày hai ví dụ thực tế có sử dụng hàng đợi.
2. Viết chương trình cài đặt hoàn chỉnh các thao tác trên hàng đợi sử dụng mảng.
3. Viết chương trình cài đặt hoàn chỉnh các thao tác trên hàng đợi sử dụng danh sách liên kết đơn.
4. Xây dựng hàng đợi hai đầu (gọi là Deque) có tính chất: Có thể thêm hoặc hủy các phần tử ở cả hai đầu của nó. Sử dụng hàng đợi hai đầu để cài đặt cùng lúc hai kiểu dữ liệu ngăn xếp và hàng đợi.

CHƯƠNG 7. CÂY NHỊ PHÂN

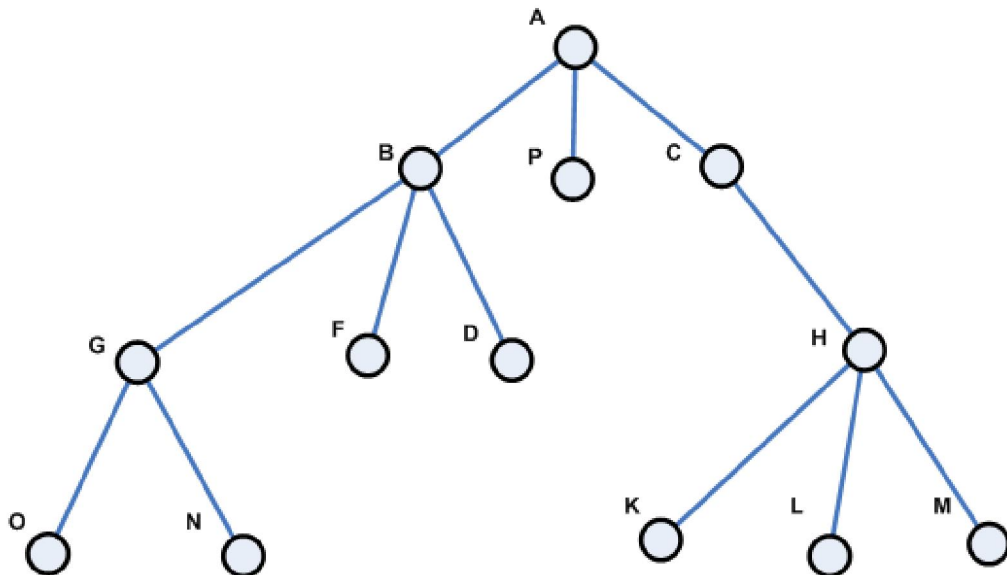
Chương này trình bày những khái niệm và tính chất cơ bản về cấu trúc dữ liệu dạng cây. Đây là loại cấu trúc dữ liệu hỗ trợ truy suất thông tin một cách nhanh chóng và hiệu quả. Nội dung sẽ tập trung vào kiểu dữ liệu cây nhị phân và cây nhị phân tìm kiếm.

I. Cấu trúc cây

1. Các thuật ngữ cơ bản trên cây

a. Định nghĩa

Cây là một tập hợp C gồm các phần tử gọi là **nút** (hay node) của cây. Trong đó có một nút đặc biệt gọi là **nút gốc** (root), các nút còn lại được chia thành các tập rời nhau C_1, C_2, \dots, C_n , trong đó C_i cũng được gọi là một cây. Mỗi nút ở cấp i sẽ quản lý một số nút ở cấp $i+1$. Mỗi quan hệ này gọi là mối quan hệ **cha – con** và các cây ở cấp $i+1$ đó gọi là **cây con** của cây ở cấp i . Sau đây là một số khái niệm cơ bản của cấu trúc dữ liệu cây.



b. Một số khái niệm cơ bản❖ **Bậc của nút**

Là số cây con của nút đó.

Ví dụ: Các nút A, B, H có bậc 3 vì có 3 cây con.

❖ **Bậc của một cây**

Là bậc lớn nhất của các nút trong cây. Một cây có bậc n gọi là cây n-phân.

Ví dụ: Trong cây gốc A trên, bậc của cây là 3.

❖ **Nút gốc**

Nút gốc (root) là nút không có cha.

Ví dụ: Nút A là nút gốc. Người ta gọi là cây con gốc A hay cây A.

❖ **Nút lá**

Là nút có bậc bằng 0, hay là nút không có con.

Ví dụ: Các nút O, N, F, D, P, K, L, M là các nút lá của cây.

❖ **Nút nhánh**

Nút nhánh (hay nút trung gian) là nút không phải nút lá và không phải nút gốc.

Ví dụ: Các nút G, B, C, H là các nút nhánh của cây.

❖ **Mức của một nút (level)**

Người ta quy định, nút gốc của cây có mức là 0. Sau đó mức của các nút còn lại được tính như sau:

Mức của nút = Mức của nút cha + 1.

Ví dụ: Mức của nút A, ký hiệu $\text{level}(A) = 0$.

$\text{level}(B) = \text{level}(P) = \text{level}(C) = \text{level}(A) + 1 = 1$.

❖ **Rừng cây**

Một tập hợp các cây riêng lẻ gọi là một rừng cây.

2. Các loại cấu trúc dữ liệu cây

Trước khi đi vào nghiên cứu cấu trúc dữ liệu cây nhị phân, chúng ta cần biết có những loại cấu trúc dữ liệu cây nào, và tên của từng loại thể hiện tính chất gì của cấu trúc cây đó.

Loại cấu trúc cây tổng quát là **cây nhiều nhánh (Multiway Trees)**. Đây là loại cấu trúc cây mà bậc của các nút trong cây có giá trị tối đa là một số n hữu hạn nào đó (còn gọi là cây n -phân).

Cây nhiều nhánh tìm kiếm (Multiway Search Tree) là một cây nhiều nhánh mà có thêm tính chất các giá trị khóa của các nút trong cây phải được sắp thứ tự tăng dần từ các cây con trái sang các cây con phải. Tính chất này sẽ giúp việc tìm kiếm trên cây được thực hiện một cách nhanh chóng.

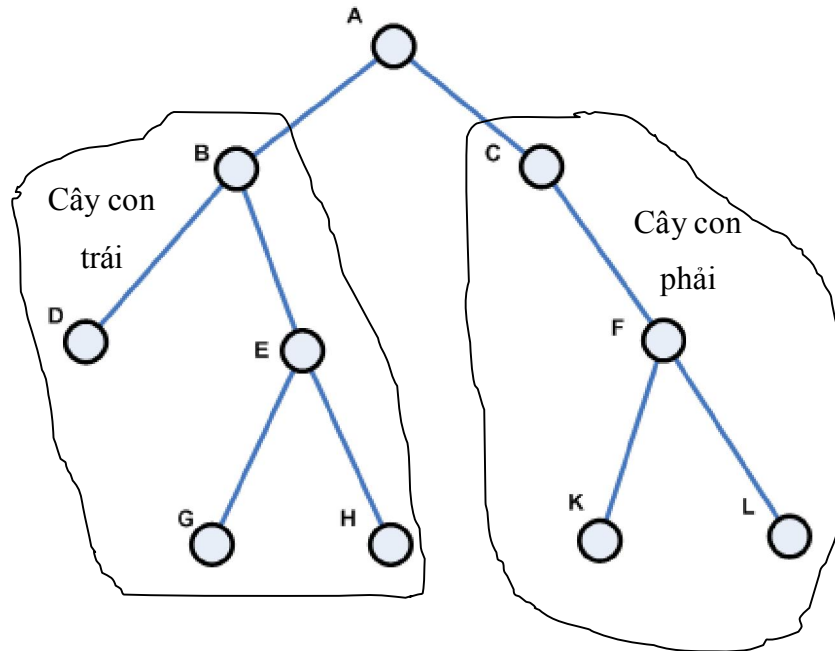
Cây nhiều nhánh cân bằng (Balanced Multiway Tree), gọi tắt B-Cây là một cây nhiều nhánh tìm kiếm thỏa thêm một số tính chất sao cho số nút ở cây con trái và cây con phải không quá chênh lệch nhau. Đây là dạng cây tối ưu, được sử dụng nhiều trong thực tế.

Trong những phần tiếp theo, chúng ta sẽ nghiên cứu cấu trúc **cây nhị phân** và **cây nhị phân tìm kiếm**. Đây là các dạng đặc biệt của cây nhiều nhánh, và cây nhiều nhánh tìm kiếm. Các loại cấu trúc cây khác sẽ được nghiên cứu sâu hơn trong môn học Cấu trúc dữ liệu 2.

II. Cây nhị phân

1. Định nghĩa

Cây nhị phân là cây rỗng hoặc là cây mà mỗi nút có tối đa 2 cây. Các nút con của cây được phân biệt thứ tự rõ ràng, một nút con gọi là nút con trái, nút còn lại là nút con phải.



2. Duyệt cây nhị phân

Có ba cách duyệt cây nhị phân thông dụng:

❖ Duyệt tiên tự (Node-Left-Right)

Trước tiên thăm nút gốc, sau đó thăm các nút của cây con trái, rồi đến cây con phải.

❖ Duyệt trung tự (Left-Node-Right)

Trước tiên thăm các nút của cây con trái, sau đó thăm nút gốc, rồi đến cây con phải.

❖ Duyệt hậu tự (Left-Right-Node)

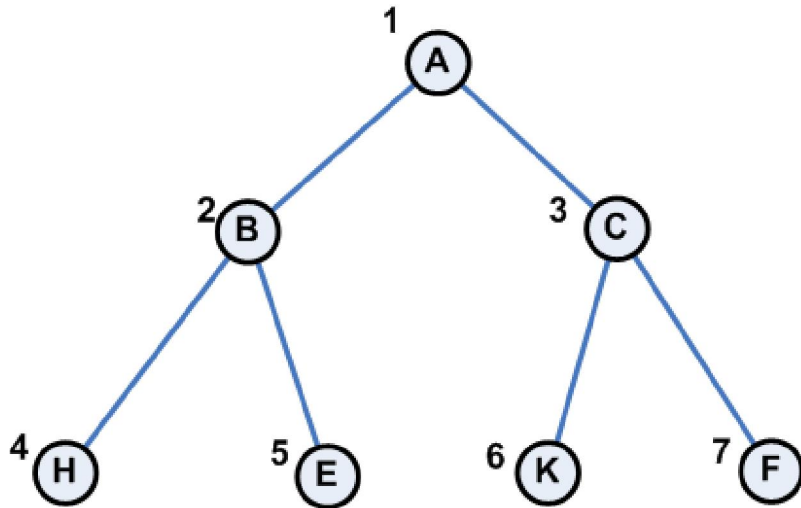
Trước tiên thăm các nút của cây con trái, sau đó thăm các nút của cây con phải, rồi cuối cùng thăm nút gốc.

3. Cài đặt cây nhị phân

Ta có thể cài đặt cấu trúc dữ liệu cây bằng mảng hoặc danh sách liên kết như sau:

a. Cài đặt bằng mảng

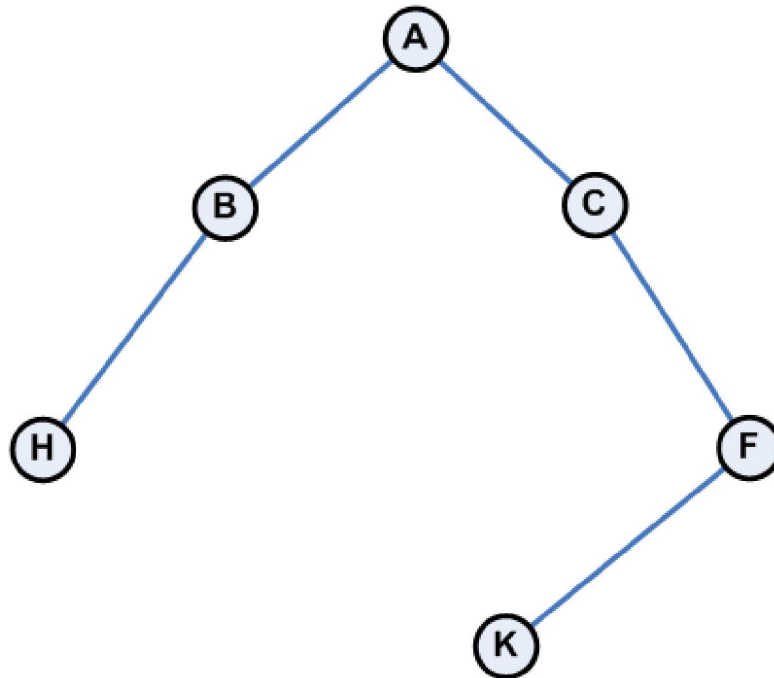
Xét trường hợp có một cây nhị phân đầy đủ, ta có thể đánh số các nút trên cây theo thứ tự từ mức 0 trở đi, hết mức này đến mức khác và từ trái qua phải đối với các nút ở mỗi mức như sau:



Theo cách đánh số này, nút thứ i có hai con là nút thứ $2*i$, và $2*i + 1$. Cha của nút thứ j là $j/2$ (Phép chia lấy phần nguyên). Dựa vào nguyên tắc này, ta có thể lưu trữ cây trên một mảng `Tree[]`, nút thứ i được lưu trữ trên phần tử `Tree[i]`. Đối với cây nhị phân đầy đủ trên, ta có mảng lưu trữ như hình dưới đây. Vì kiểu dữ liệu mảng trong ngôn ngữ lập trình C có chỉ số bắt đầu từ 0, nên ta không sử dụng phần tử đầu tiên của mảng.

0	1	2	3	4	5	6	7
	A	B	C	H	E	K	F

Đối với cây nhị phân không đầy đủ, ta có thể thêm vào một số nút giả để được cây nhị phân đầy đủ. Những nút giả này sẽ được gán một giá trị đặc biệt để ta có thể loại trừ ra khi xử lý trên cây. Chúng ta xem ví dụ dưới đây:



Với cây nhị phân không đầy đủ này, ta có thể lưu trữ trên mảng như sau:

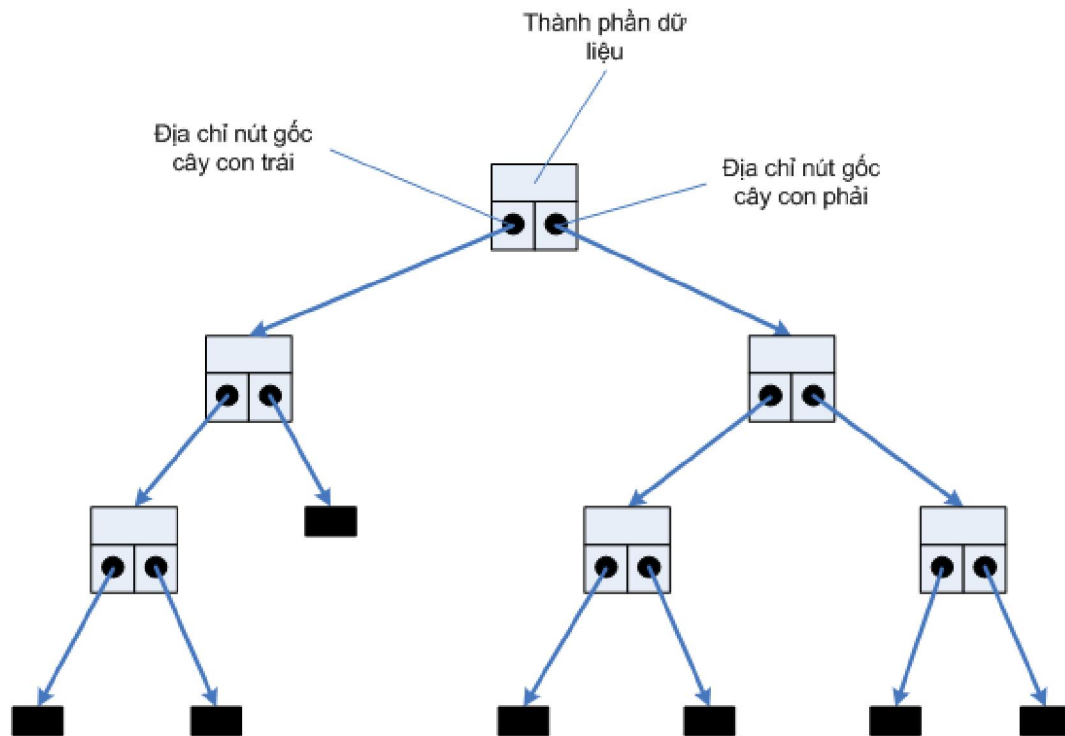
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	A	B	C	H										K	

Rõ ràng cách cài đặt này sẽ gây ra lãng phí bộ nhớ, đặc biệt là đối với cấu trúc cây lệch nhiều sang một phía. Ngoài ra, việc thực hiện các thao tác như loại bỏ hay thêm một nhánh của cây cũng sẽ tốn kém chi phí vì phải truy suất đến từng phần tử của nhánh đó để loại bỏ. Vì vậy, người ta thường cài đặt cây bằng danh sách liên kết. Khi đó sẽ giải quyết được những nhược điểm mà việc cài đặt bằng mảng gặp phải.

b. Cài đặt bằng cấu trúc liên kết

Cấu trúc cây nhị phân sẽ được cài đặt theo cấu trúc liên kết mà mỗi nút lưu trữ các thông tin sau:

- + Thông tin lưu trữ tại mỗi nút.
- + Địa chỉ nút gốc của cây con trái trong bộ nhớ.
- + Địa chỉ nút gốc của cây con phải trong bộ nhớ.



Cài đặt cụ thể như sau:

```
#define ElementType <Kiểu dữ liệu>
```

```
typedef struct tagTNode
```

```
{
```

```
    ElementType key;
```

```
    tagTNode* pLeft, *pRight;
```

```
} TNode;
```

Sau đây, ta cài đặt các phép toán cơ bản trên cây

❖ Tạo cây rỗng

```
void InitTree(TNode* root )
{
    root=NULL;
}
```

❖ Kiểm tra cây rỗng

```
int IsEmptyTree(TNode*root)
{
    if(root == NULL)
        return 0;
    return 1;
}
```

❖ Kiểm tra nút lá

Một nút là lá khi không có con nào, tức là giá trị pLeft và pRight là NULL.

```
int IsLeafNode(TNode*root)
{
    if(root->pLeft==NULL && root->pRight=NULL)
        return 1;
    return 0;
}
```

❖ Các thủ tục duyệt cây

Sử dụng phương pháp quy nạp để thực hiện các phép duyệt cây.

• Duyệt tiền tự

```
void PreOrder(TNode*root)
{
    if(root !=NULL)
    {
        //---Xử lý thông tin tại root---//
        PreOrder(root->pLeft);
        PreOrder(root->pRight);
    }
}
```

• Duyệt trung tự

```
void InOrder(TNode*root)
{
    if(root !=NULL)
    {
        PreOrder(root→pLeft);
        //---Xử lý thông tin tại root---//
        PreOrder(root→pRight);
    }
}
```

- **Duyệt hậu tự**

```
void PostOrder(TNode*root)
{
    if(root !=NULL)
    {
        PreOrder(root→pLeft);
        PreOrder(root→pRight);
        //---Xử lý thông tin tại root---//
    }
}
```

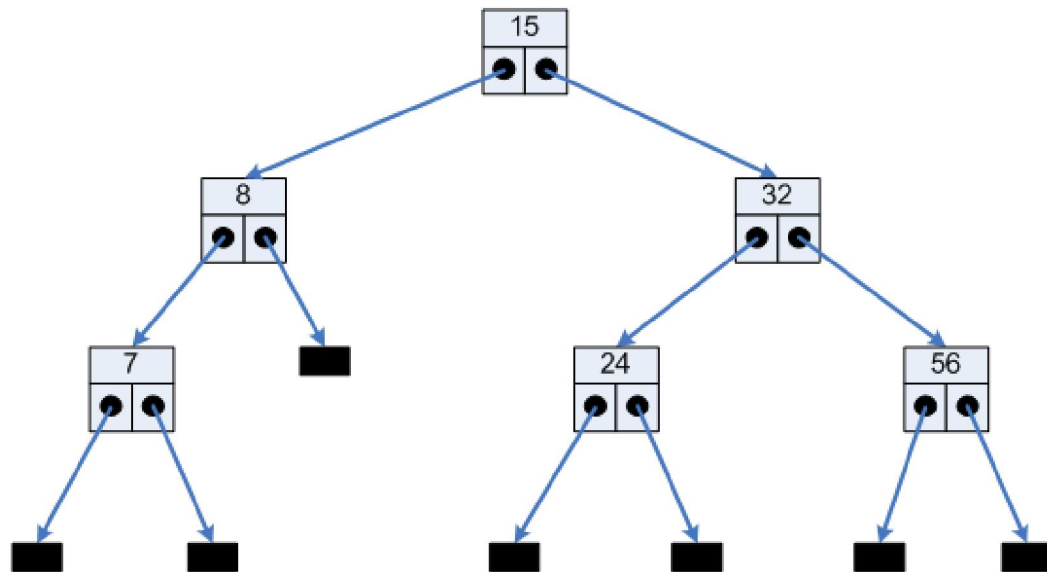
III. Cây nhị phân tìm kiếm

1. Định nghĩa

Cây nhị phân tìm kiếm là cây nhị phân mà khóa tại mỗi nút của cây lớn hơn khóa của tất cả các nút thuộc cây con trái và nhỏ hơn khóa của tất cả các nút thuộc cây con phải.

Một cây rỗng có thể coi là cây nhị phân tìm kiếm. Dựa vào định nghĩa, ta có một số nhận xét như sau:

- + Trên cây nhị phân tìm kiếm, không có các nút cùng khóa.
- + Các cây con trái, phải của một cây nhị phân tìm kiếm cũng là một cây nhị phân tìm kiếm.



2. Các thao tác trên cây nhị phân tìm kiếm (Cây NPTK)

a. Thêm một nút vào cây NPTK

```

int InsertNode(TNode* root, ElementType x)
{
    if(root != NULL)
    {
        if(root->key==x)
            return 0;//Đã tồn tại phần tử có khóa x
        if(root->key>x)
            return InsertNode(root->pLeft, x);//Thêm vào cây con trái
        else
            return InsertNode(root->pRight, x);//Thêm vào cây con phải
    }
    //Thêm tại root
    root=new TNode;
    if(root==NULL)
        return -1;//Không đủ bộ nhớ
    root->key=x;
    root->pLeft=root->pRight=NULL;
    return 1;//Thêm vào thành công
}
  
```

b. Tìm kiếm trên cây NPTK

Tìm kiếm nút trên cây có khóa bằng x.

```

TNode* SearchOnTree(TNode*root, ElementType x)
{
    if(root!=NULL)
    {
        if(root->key == x)
            return root;//Tìm thấy
        if(root->key > x)
            return SearchOnTree(root->pLeft, x);
        else
            return SearchOnTree(root->pRight, x);
    }
    return NULLDATA;
}

```

Có cài đặt hàm tìm kiếm mà không cần sử dụng đệ quy. Phần này dành cho người đọc.

c. Hủy một nút có khóa x

Khi thực hiện xóa một phần tử x khỏi một cây phải đảm bảo điều kiện ràng buộc của cây nhị phân tìm kiếm. Xảy ra 3 trường hợp như sau:

❖ Trường hợp x là nút lá

Trường hợp này ta chỉ việc giải phóng bộ nhớ cho phần tử này mà thôi.

❖ Trường hợp x có 1 nút con

Trường hợp này ta thực hiện hai việc là móc nối nút cha của x với con duy nhất của x, rồi hủy phần tử x.

❖ Trường hợp x có 2 nút con

Trường hợp này ta không thể hủy phần tử x được mà ta phải thay thế nó bằng nút lớn nhất trên cây con trái hoặc nút nhỏ nhất trên cây con phải. Khi đó nút được giải phóng bộ nhớ là một trong hai nút này. Trong thuật toán dưới đây, ta sẽ thay thế x bằng nút nhỏ nhất trên cây con phải (là phần tử cực trái của cây con phải).

Sau đây là hàm cài đặt hủy một nút có khóa x trên cây

```

int DeleteNode(TNode*root, ElementType x)
{
    if(root==NULL)
        return 0;//Không tồn tại nút có khóa x
    if(root->key>x)
        return DeleteNode(root->pLeft, x);
    if(root->key<x)
        return DeleteNode(root->pRight, x);
    //Xóa root
    TNode*temp;
    if(root->pLeft==NULL)
    {
        temp=root;
        root=root->pRight;
        delete temp;
    }
    else
        if(root->pRight==NULL)
        {
            temp=root;
            root=root->pLeft;
            delete temp;
        }
        else//Trường hợp root có đủ 2 con
        {
            TNode*p=root->pRight;//Truy xuất cây con phải
            MoveLeftMostNode(p, root);
        }
}
//Hàm tìm phần tử trái nhất trên cây con phải. Sau đó chuyển nội dung lên vị trí
của phần tử x và giải phóng bộ nhớ
void MoveLeftMostNode(TNode*p, TNode* root)
{
    if(p->pLeft != NULL)
        MoveLeftMostNode(p->pLeft, root);
    else
    {
        TNode*temp;
        temp=p;
        root->key=p->key; //Chuyển nội dung từ p sang root
        p=p->pRight;
        delete temp;
    }
}

```

d. Hủy toàn bộ cây NPTK

Hủy toàn bộ cây nhị phân tìm kiếm được thực hiện thông qua việc duyệt cây theo phương pháp hậu tự. Tức là ta xóa các cây con trái, rồi cây con phải trước khi xóa gốc.

```
void RemoveAllNodes(TNode*root)
{
    if(root != NULL)
    {
        RemoveAllNodes(root->pLeft);
        RemoveAllNodes(root->pRight);
        delete root;
    }
}
```

IV. Bài tập chương 7

1. Hãy vẽ tất cả các cây nhị phân có 3 nút, 4 nút.
2. Chứng minh một cây nhị phân có n nút lá thì có tất cả $2n-1$ nút.
3. Một cây nhị phân đầy đủ có n nút. Chứng minh chiều sâu của cây này là $\log_2(n+1)-1$.
4. Viết chương trình nhập vào một cây nhị phân. Hãy cài đặt các tác vụ trên cây như sau:
 - + Xác định số nút trên cây.
 - + Xác định số nút lá.
 - + Xác định số nút có một cây con
 - + Xác định số nút có hai cây con.
 - + Xác định chiều sâu của cây.
 - + Xác định số nút trên từng mức.
5. Viết chương trình mô phỏng các thao tác (thêm nút, xóa nút, tìm kiếm) trên cây.

TÀI LIỆU THAM KHẢO

1. Robert L. Kruse, Alexandr J. Ryba, *Data structures and Program Design in C++*. Prentice Hall, 2000.
2. Jim Keogh, Ken Davidson, *Data Structures Demystified*. McGraw-Hill, 2004.
3. Robert Lafore, *Data Structures and Algorithms in Java*. SAMS, 1998.
4. Nell Dale, *C++ Plus Data Structures*. Jones and Bartlett Publishers, 2003.
5. Donald Knuth, *The Art of Computer Programming, Volume 1, 2, 3*. Addison-Wesley, 1997.
6. ThS. Trần Hạnh Nhi, *Nhập môn cấu trúc dữ liệu và giải thuật*. Đại học KHTN TP. HCM, 2000.
7. ThS. Nguyễn Ngô Bảo Trân, *Giáo trình cấu trúc dữ liệu và giải thuật*. Đại học Bách Khoa TP. HCM, 2001.
8. Nguyễn Văn Linh, Trần Ngân Bình, *Giáo trình Cấu trúc dữ liệu*. Đại học Cần Thơ, 2003.
9. Lê Minh Hoàng, *Giải thuật và lập trình*. Đại học sư phạm Hà Nội, 2002.

MỤC LỤC

LỜI NÓI ĐẦU	2
CHƯƠNG 1. TỔNG QUAN.....	3
I. Từ bài toán đến chương trình.....	3
1. Xác định bài toán	3
2. Xây dựng cấu trúc dữ liệu	3
3. Thiết kế giải thuật	4
4. Lập trình	5
5. Kiểm lỗi và sửa lỗi chương trình	5
II. Kiểu dữ liệu.....	6
1. Định nghĩa kiểu dữ liệu.....	6
2. Các kiểu dữ liệu cơ bản.....	6
3. Các kiểu dữ liệu có cấu trúc	7
4. Kiểu dữ liệu trừu tượng.....	7
III. Đánh giá độ phức tạp của giải thuật.....	8
IV. Bài tập chương 1.....	8
CHƯƠNG 2. ĐỆ QUY.....	10
I. Khái niệm đệ quy	10
II. Xây dựng giải thuật đệ quy.....	10
1. Định nghĩa	10
2. Ví dụ.....	10
3. Thiết kế giải thuật đệ quy.....	12
III. Một số bài toán sử dụng đệ quy	12
1. Dãy số Fibonacci	12

2. Bài toán Tháp Hà Nội	13
V. Bài tập chương 2.....	15
CHƯƠNG 3. TÌM KIẾM VÀ SẮP XẾP.....	16
I. Giới thiệu	16
II. Các giải thuật tìm kiếm.....	16
1. Tìm kiếm tuyến tính.....	17
2. Tìm kiếm nhị phân.....	19
III. Các giải thuật sắp xếp.....	21
1. Giới thiệu bài toán sắp xếp.....	21
2. Phương pháp đổi chỗ trực tiếp (Interchange sort).....	21
3. Phương pháp chọn trực tiếp (Selection sort).....	23
4. Phương pháp chèn trực tiếp (Insertion sort).....	25
5. Phương pháp nổi bọt (Bubble sort).....	27
6. Sắp xếp dựa trên phân hoạch (Quicksort).....	29
IV. Bài tập chương 3.....	32
CHƯƠNG 4. DANH SÁCH (LIST)	34
I. Định nghĩa danh sách	34
II. Cài đặt danh sách.....	34
1. Cài đặt danh sách bằng mảng (danh sách đặc).....	34
2. Cài đặt danh sách bằng con trỏ (danh sách liên kết)	37
III. Bài tập chương 4.....	49
CHƯƠNG 5. NGĂN XẾP (STACK).....	51
I. Định nghĩa ngăn xếp.....	51
II. Một số phép toán trên ngăn xếp	51

III. Cài đặt ngăn xếp	53
1. Cài đặt bằng mảng	53
2. Cài đặt bằng danh sách liên kết đơn	56
IV. Ứng dụng ngăn xếp để loại bỏ đệ quy của chương trình.....	57
1. Bài toán tính giai thừa.....	58
2. Bài toán Tháp Hà Nội	59
3. Một số ứng dụng khác của ngăn xếp	62
V. Bài tập chương 5.....	62
CHƯƠNG 6. HÀNG ĐỢI (QUEUE).....	64
I. Định nghĩa hàng đợi	64
II. Một số phép toán trên hàng đợi.....	64
III. Cài đặt hàng đợi.....	65
1. Cài đặt bằng mảng	65
2. Cài đặt bằng danh sách liên kết đơn	69
IV. Các ứng dụng của hàng đợi.....	71
V. Bài tập chương 6.....	71
CHƯƠNG 7. CÂY NHỊ PHÂN.....	72
I. Cấu trúc cây.....	72
1. Các thuật ngữ cơ bản trên cây	72
2. Các loại cấu trúc dữ liệu cây	74
II. Cây nhị phân	74
1. Định nghĩa	74
2. Duyệt cây nhị phân	75
3. Cài đặt cây nhị phân.....	75

Cấu trúc dữ liệu 1	89
<hr/>	
III. Cây nhị phân tìm kiếm.....	80
1. Định nghĩa	80
2. Các thao tác trên cây nhị phân tìm kiếm (Cây NPTK).....	81
IV. Bài tập chương 7.....	84
TÀI LIỆU THAM KHẢO	85