BỘ GIÁO DỤC VÀ ĐÀO TẠO
TRƯỜNG ĐẠI HỌC SƯ PHẠM KỸ THUẬT
THÀNH PHỐ HỒ CHÍ MINH

****

TH.S LÊ VĂN VINH

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

NHÀ XUẤT BẢN ĐẠI HỌC QUỐC GIA
THÀNH PHỐ HỒ CHÍ MINH

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 giúp cho người học hiểu biết 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 giáo trình này bao gồm 6 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.

MỤC LỤC

CHƯƠNG 1: TỔNG QUAN

1.1. GIẢI QUYẾT BÀI TOÁN TRÊN MÁY TÍNH

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, một số bước cơ bản cần phải tiến hành như sau:

1.1.1. Xác định yêu cầu của bài toán

Ở bước này, chúng ta cần xác định xem phải giải quyết vấn đề gì. Những giả thiết nào đã cho và những yêu cầu nào cần phải đạt được. 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 mơ hồ và hình thức, chúng ta cần phát biểu lại một cách rõ ràng và chính xác để có thể xây dựng thuật toán phù hợp.

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 mỗi nước được tô bởi duy nhất một màu và hai nước láng giềng (cùng biên giới) của nhau thì phải được tô bằng hai màu khác nhau".

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

1.1.2. Xây dựng cấu trúc dữ liệu

Khi giải một bài toán, chúng ta cần định nghĩa tập hợp dữ liệu để biểu diễn một cách đầy đủ những thông tin có trong thực tế. 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 cần giải quyết và những thao tác sẽ tiến hành trên dữ liệu đầu vào. Một số giải thuật chỉ phù hợp với một cách tổ chức dữ liệu nhất định, còn với cách tổ chức dữ liệu khác thì có thể kém hiệu quả hoặc không thể thực hiện được. Vì vậy, khi lựa chọn cấu trúc dữ liệu, mốt số tiêu chuẩn sau cần được đảm bảo:

- 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 được 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.

1.1.3. 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 đầu vào, sau một số hữu hạn các bước thực hiện, chương trình có thể đạt được mục tiêu đã định. Tùy theo yêu cầu thực tế mà người viết chương trình cần áp dụng 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.

1.1.4. Cài đặt chương trình

Khi đã xác định được thuật toán và cấu trúc dữ liệu sẽ sử dụng, chúng ta tiến hành lập trình để hiện thực chúng. Thông thường, khi lập trình, chúng 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.

- Đối với những công việc đơn giản hoặc có thể cài đặt ngay thì tiến hành viết mã nguồn cho nó.
- Đối với những công việc phức tạp thì chia thành những công việc nhỏ hơn để tiếp tục thực hiện với những công việc nhỏ đó.

1.1.5. Kiểm lỗi và sửa lỗi chương trình

Tìm lỗi, sửa lỗi là công việc cần thực hiện trong quá trình cài đặt chương trinh. Thông thường, có ba loại lỗi phát sinh trong lập trình là:

- *Lỗi cú pháp*: Đây là nhóm lỗi do sử dụng sai từ khóa, sai quy tắc viết lệnh của ngôn ngữ lập trình. Nhóm lỗi này có thể được phát hiện bởi chương trình dịch (trình biên dịch, hay trình thông dịch).
- *Lỗi cài đặt*: Đây là nhóm lỗi mà người lập trình không cài đặt theo đúng thuật toán đã xây dựng. Đối với loại lỗi này, cần phải xem xét lại bố cục chương trình, kết hợp với các chức năng sửa lỗi của trình soạn thảo để tìm và sửa.
- *Lỗi thuật toán*: Nhóm lỗi này là lỗi nghiêm trọng nhất vì chương trình đã bị sai từ bước thiết kế giải thuật hoặc lựa chọn cấu trúc dữ liệu không phù hợp. Đối với nhóm lỗi này, lập trình viên có thể 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 trên máy tính. 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 chúng ta chỉ được phép sử dụng các giải thuật mà có thể áp dụng trên cấu trúc dữ liệu đã lựa chọn.

1.2. 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++.

1.3. ĐÁNH GIÁ ĐỘ PHỨC TẠP CỦA GIẢI THUẬT

Một bài toán có thể có nhiều giải thuật khác nhau để giải quyết. Chúng ta cần phải chọn giải thuật nào hiệu quả nhất. Thời gian tính toán là một tiêu chí quan trọng để đánh giá một giải thuật. 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. Vì vậy, người ta thường dựa vào số phép tính toán 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.

Để biểu diễn độ phức tạp của một giải thuật, một số ký hiệu thường được sử dụng như $O(f(n))$, $\Omega(f(n))$, $\Theta(f(n))$, $o(f(n))$, $\omega(f(n))$ (mỗi ký hiệu tương ứng với một cách đánh giá độ phức tạp cụ thể). Trong phạm vi giáo trình này, ký hiệu $O(f(n))$ được sử dụng để đánh giá giải thuật.

- 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).
- 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 $\log_a f(n)=\log_a b*\log_b f(n)$. 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.
- 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.

1.4. 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 mảng

Hãy xác định độ phức tạp của từng hàm đã cài đặt.

Chương 2
ĐỆ QUY

Chương này giới thiệu dạng giải thuật mà một vấn đề được giải quyết bằng cách gọi đến chính nó để thực hiện những công việc nhỏ hơn. Chúng ta sẽ tìm hiểu khái niệm cơ bản, các dạng đệ quy và một số ví dụ minh họa điển hình.

2.1. KHÁI NIỆM ĐỆ QUY

Khi định nghĩa hay mô tả về một vấn đề, sự vật, sự kiện nào đó, ta phải dùng các từ ngữ nhằm giải thích để làm rõ nghĩa của đối tượng đó. Người ta chia ra thành hai cách định nghĩa như sau:

- *Định nghĩa tường minh*: Nhằm giải thích một khái niệm mới bằng các khái niệm đã có. Ví dụ: "Người là động vật cấp cao". Trong cách định nghĩa này, chúng ta sử dụng khái niệm "động vật cấp cao" để giải thích cho khái niệm "người".
- *Định nghĩa theo kiểu đệ quy (hay định nghĩa không tường minh)*: Nhằm giải thích một khái niệm bằng chính khái niệm đó. Ví dụ "Người là con của hai người khác". Định nghĩa này đã sử dụng lại khái niệm "người" để giải thích cho khái niệm "người". Đây chính là cách diễn đạt bằng phương pháp đệ quy.

Như vậy, ***đệ quy (recursion)*** là phương pháp được sử dụng nhằm định nghĩa, mô tả một vấn đề nào đó bằng cách sử dụng lại chính khái niệm đang cần định nghĩa. Đệ quy có ý nghĩa là ***quay về***.

Trong thực tế, chúng ta có thể thấy tính chất đệ quy được thể hiện trong bất cứ lĩnh vực nào của cuộc sống. Ví dụ, khi đặt hai chiếc gương đối diện nhau, chiếc gương thứ nhất sẽ chứa hình chiếc gương thứ hai. Ngược lại, trong chiếc gương thứ hai lại chứa hình chiếc gương thứ nhất. Vì thế, trong hình ảnh của chiếc gương thứ nhất chứa hình ảnh của chính nó.

Trong toán học, chúng 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 $n!=n.(n-1)!$.

Khái niệm đệ quy được định nghĩa như sau:

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.

2.2. GIẢI THUẬT ĐỆ QUY

2.2.1. 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.

Ví dụ:

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

$$\begin{cases} n!=1 \text{ Nếu } n=0. \\ n!=n.(n-1)\ldots 2.1. \text{ Nếu } n>0. \end{cases}$$

Với cách định nghĩa này, chúng ta có thể thấy được rõ ràng cách tính giai thừa của một số. Tuy nhiên, cách diễn đạt này không cho thấy quy luật tính toán tổng quát của nó. Vì thế, người ta thường chuyển về dạng quy nạp như sau:

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

Như vậy, để đưa ra lời giải cho bài toán *tính giai thừa của n* (bài toán *P*), chúng ta đã sử dụng lời giải của bài toán *tính giai thừa của n-1* (bài toán *P'*). Lời giải này chính là lời giải đệ quy. Với trường hợp *n=4*, chúng ta có được các kết quả tính toán từng bước như sau:

$$\begin{aligned} 4! &= 4.3! \\ &= 4.(3.2!) \\ &= 4.(3.(2.1!)) \\ &= 4.(3.(2.(1.0!))) \\ &= 4.(3.(2.(1.1))) \end{aligned}$$

=4.(3.(2.1))

=4.(3.2)

=4.6

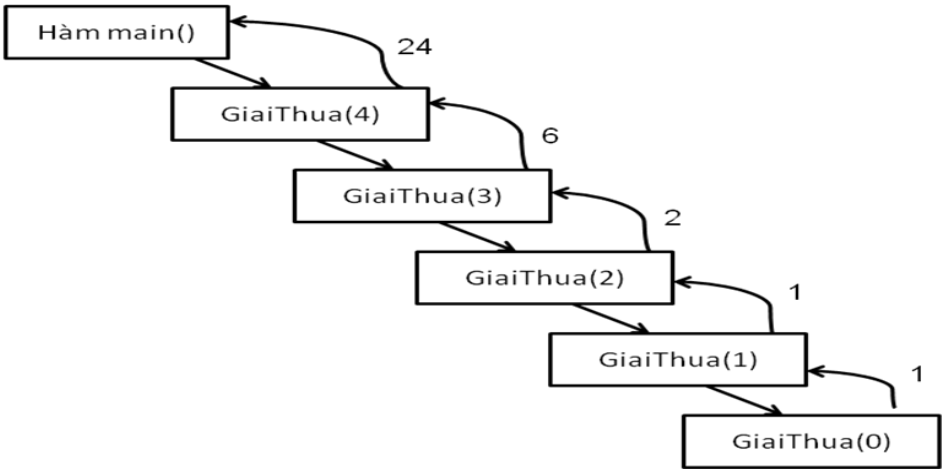
=24

Việc tính toán từng bước như trên cho chúng ta thấy cách làm việc của giải thuật đệ quy. 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 (trường hợp *n=0*).

Dưới đây là chương trình cài đặt cho giải thuật tính giai thừa bằng phương pháp đệ quy.

```
#include<<iostream.h>>
int GiaiThua(int n)//Hàm tính giai thừa của n
{
        if(n==0)
                return 1;
        return n*GiaiThua(n-1);//Gọi lại hàm tính giai thừa của n-1
}
void main()
{
        int n, kq;
        cin>>n;
        kq=GiaiThua(n);
        cout<<kq;
}
-----------------------------------------
Nhập vào: n=4
Kết quả: 24
```

Thứ tự gọi hàm trong thời điểm chương trình chạy được minh họa trong hình dưới đây (trong trường hợp *n=4*):



Hình 1. Thứ tự gọi hàm trong giải thuật tính giai thừa

2.2.2. Thiết kế giải thuật đệ quy

Để có thể áp dụng giải thuật đệ quy cho một bài toán cụ thể, việc đầu tiên là phải tìm ra tính chất đệ quy của nó. Chẳng hạn, với bài toán tìm dữ liệu trong một thư mục trên máy tính (tổ chức dưới dạng cây), một *thư mục* có thể bao gồm các *thư mục con* và các tập tin. Các thư mục con cũng có thể chứa các *thư mục con khác* và các tập tin. Tất cả khái niệm như: *thư mục, thư mục con, thư mục con khác* đều chỉ cùng một dạng đối tượng là *thư mục*.

Tiếp đến, vấn đề cần được quan tâm là phần cơ sở của giải thuật. Điều này cũng có nghĩa là chúng ta quan tâm đến việc thuật toán sẽ dừng lại khi nào. Nếu chúng ta không tìm ra (hoặc không tìm ra hết) các điều kiện cơ sở, sẽ dẫn đến khả năng chương trình chạy vô tận. Chẳng hạn, với bài toán tính giai thừa của *n*. Khi *n=0*, giải thuật không tiếp tục gọi chính nó nữa, mà trả về kết quả mặc định là 1. Tương tự như vậy, khi duyệt trên cây thư mục, thuật toán cần phải kiểm tra khi thư mục đang được xét không chứa bất kỳ một thư mục con nào khác (có thể chứa tập tin) thì không tiếp tục gọi đệ quy nữa. Đó là điều kiện dừng của bài toán.

Sau khi đã xác định được phần đệ quy và phần cơ sở của thuật toán, việc còn lại là kết hợp các trường hợp này sao cho hợp lý với bài toán cần giải quyết.

Tóm lại, việc thiết kế một giải thuật đệ quy bao gồm ba bước như sau:

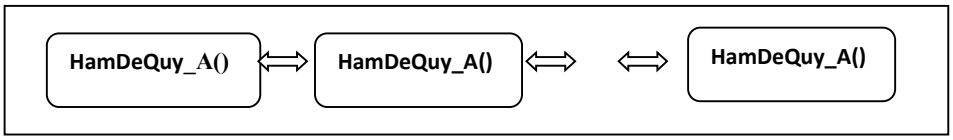
- Tìm ra phần đệ quy của giải thuật.
- Tìm ra phần cơ sở của giải thuật.
- Kết hợp phần đệ quy và phần cơ sở.

2.3. PHÂN LOẠI ĐỆ QUY

Một cách tổng quát, người ta chia các hàm đệ quy ra thành hai nhóm. Nhóm các hàm đệ quy trực tiếp (direct recursion) và nhóm các hàm đệ quy gián tiếp (indirect recursion).

2.3.1. Đệ quy trực tiếp

Hàm là đệ quy trực tiếp khi nó gọi trực tiếp đến chính nó (Hình 2).



Hình 2. Đệ quy trực tiếp

Sau đây là một số bài toán ví dụ có sử dụng đệ quy trực tiếp.

❖ ***Ví dụ 1****:* Tính số hạng thứ *n* trong dãy số Fibonacci.

Dãy số Fibonacci có tính chất như sau: $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$ với $n \geq 2$.

Mã nguồn hàm cài đặt:

```
#include<iostream.h>
int Fibonacci(int n)
{
        if (n==0)
                return 0;
        if (n==1)
                 return 1;
        return  Fibonacci(n-1) + Fibonacci(n-2);
}
void main()
{
        int n, kq;
        cin>>n;
        kq=Fibonacci(n);
        cout<<kq;
}
----------------------------------------
Nhập vào: n=10
```

Kết quả: 55

❖ ***Ví dụ 2**:* Tính giá trị của U_n với:

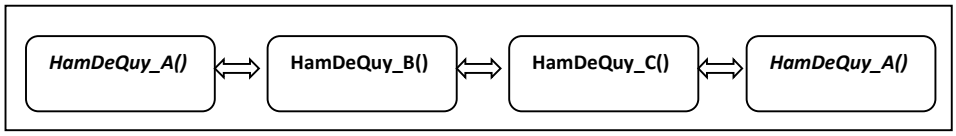
$$U_n = \begin{cases} n \text{ với } n<6. \\ U_{n-5} + U_{n-4} + U_{n-3} + U_{n-2} + U_{n-1} \text{ với } n \geq 6. \end{cases}$$

Mã nguồn hàm cài đặt:

```
int TongU(int n)
{
	if(n<6)
		return n;
	int tong = 0;
	for(int i=1;i<=5;i++)
		tong = tong + TongU(n-i);
	return tong;
}
void main()
{
	int n, kq;
	cin>>n;
	kq=TongU(n);
	cout<<kq;
}
----------------------------------------
Nhập vào: n=6
Kết quả: 15
```

2.3.2. Đệ quy gián tiếp

Hàm là đệ quy gián tiếp khi nó gọi đến chính nó thông qua một hoặc nhiều hàm đệ quy khác. Ví dụ, hàm đệ quy A gọi hàm đệ quy B, hàm đệ quy B gọi hàm C, hàm đệ quy C gọi lại hàm đệ quy A. Như vậy, hàm đệ quy A gọi lại chính nó một cách gián tiếp thông qua các hàm B, C (Hình 3).



Hình 3. Đệ quy gián tiếp

Sau đây là ví dụ về đệ quy gián tiếp

❖ ***Ví dụ 3****:* Tính giá trị của biểu thức: *U=P(n) + Q(n)*

+ Nếu *n=1, P(n)=Q(n)=1*

+ Nếu *n>1, P(n)=Q(n)*n, Q(n)=P(n)+n*

Sau đây là mã nguồn cài đặt:

```
#include<iostream.h>
int Tinh_Q(int n);
int Tinh_P(int n);
int Tinh_P(int n)
{
        if(n==1)
                return 1;
        return Tinh_Q(n-1)*n;
}
int Tinh_Q(int n)
{
        if(n==1)
                return 1;
        return Tinh_P(n-1)+n;
}
void main()
{
        int n, kq;
        cin>>n;
        kq=Tinh_P(n) + Tinh_Q(n);
        cout<<kq;
}
```

```
-----------------------------------------
Nhập vào: n=5
Kết quả: 90
```

Trong chương trình này, hàm Tinh_Q() gọi hàm Tinh_P(), hàm Tinh_P() gọi hàm Tinh_Q(). Chúng ta có thể thấy hàm Tinh_Q() đã gọi lại chính nó một cách gián tiếp thông qua hàm Tinh_P().

2.4. ĐỆ QUY ĐUÔI (TAIL-RECURSION)

Một trong những khía cạnh cần được quan tâm khi xây dựng giải thuật đệ quy là ***đệ quy đuôi*** vì nó liên quan đến vấn đề tối ưu chi phí lưu trữ trong bộ nhớ ở thời điểm thực thi.

Giải thuật thuật đệ quy đuôi là giải thuật có lời gọi đệ quy là lệnh thực thi cuối cùng của nó. Nói một cách khác, ở thời điểm thực thi, chương trình sẽ không phải thực hiện một công việc nào khác sau khi lời gọi đệ quy trả về.

Xem xét ví dụ tính giai thừa của *n* như trên, cách viết này có phải đệ quy đuôi hay không?

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

Nhận xét:

Trong hàm tính GiaiThua(n), sau khi hàm tính GiaiThua(n-1) thực thi xong và trả về kết quả. Lệnh tính toán cuối cùng là phép nhân (*): ***n*GiaiThua(n-1)***. Như vậy, cách viết này không được gọi là đệ quy đuôi.

Sau đây là cách viết lại hàm này theo cách đệ quy đuôi:

```
#include<iostream.h>
int GiaiThua_i(int n, int kq);
int GiaiThua(int n)
{
        if(n == 0)
                return 1;
        return  GiaiThua_i(n, 1);
```

Nhận xét:

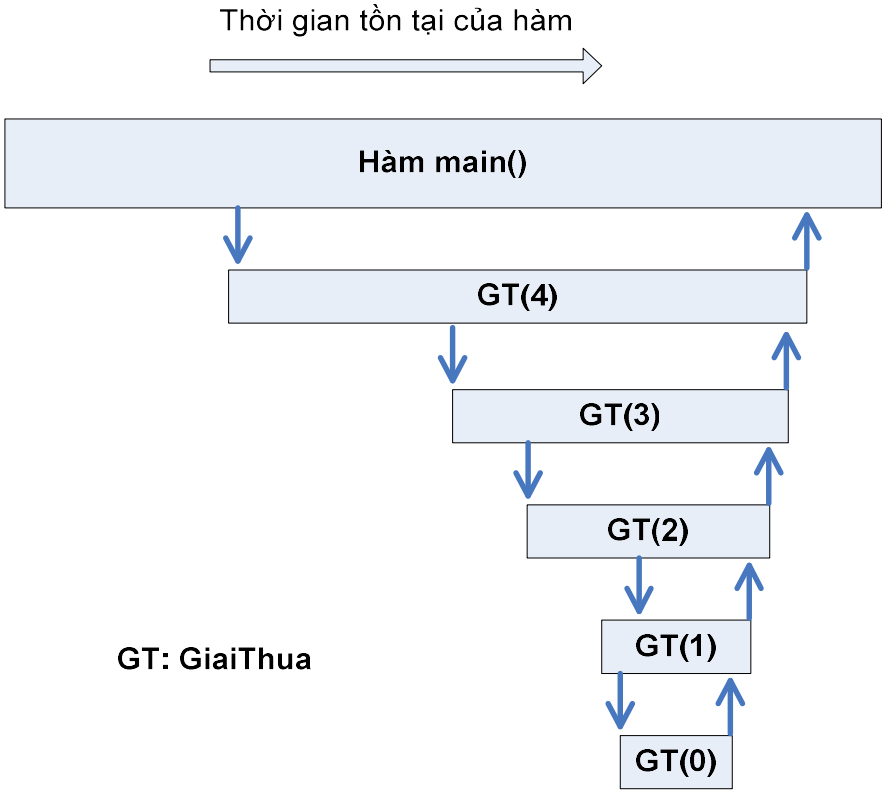
Theo cách viết này, hàm đệ quy là hàm ***GiaiThua_i***. Trong hàm này, sau khi kết quả từ hàm ***GiaiThua_i(n-1,kq*n)*** trả về, chương trình không phải thực hiện bất cứ một công việc nào khác. Vì vậy, hàm này được xem là đệ quy đuôi.

```
}
int GiaiThua_i(int n, int kq)
{
        if(n == 1)
                return kq;
        return GiaiThua_i(n - 1, kq*n);
}
void main()
{
        int n, kq;
        cin>>n;
        kq=GiaiThua(n);
        cout<<kq;
}
----------------------------------------
Nhập vào: n=4
Kết quả: 24
```

Như vậy, sự khác biệt giữa hàm đệ quy đuôi và hàm không phải đệ quy đuôi là gì? tại sao chúng ta lại phải quan tâm đến nó?

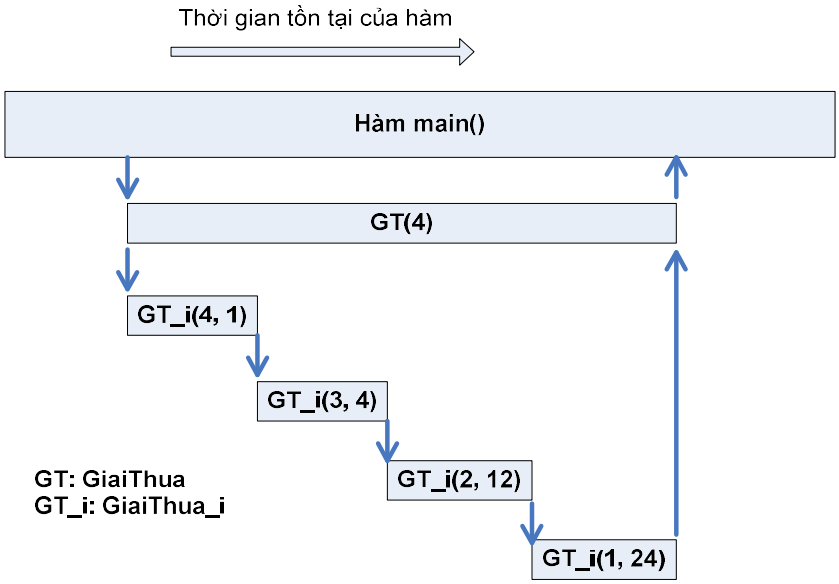
Trước hết, chúng ta cần biết rằng, ở thời điểm thực thi, khi xử lý cho một hàm đệ quy, trình biên dịch sử dụng kiểu dữ liệu ngăn xếp (stack) để lưu trữ thông tin và dữ liệu trung gian. Chẳng hạn, trong ví dụ tính giai thừa (Hình 4). Khi tính giai thừa của 4 (GiaiThua(4)), chương trình gọi đến hàm tính giai thừa của 3 (GiaiThua(3)). Dĩ nhiên, sau khi hàm GiaiThua(3) thực thi xong và trả về giá trị thì hàm GiaiThua(4) mới tiếp tục thực hiện. Theo cách viết không phải đệ quy đuôi, vẫn còn một lệnh thực hiện nữa sau khi có kết quả của hàm GiaiThua(3) (phép nhân n*GiaiThua(n-1)). Như vậy, giá trị của *n=4* cần phải được lưu trữ trong bộ nhớ. Cho đến khi hàm GiaiThua(3) thực thi xong thì giá trị này mới được lấy ra để tính toán. Giá trị *n* và thông tin về hàm GiaiThua(4) được trình biên dịch lưu trữ vào trong ngăn xếp (stack). Tương tự cho các hàm GiaiThua(3), GiaiThua(2), GiaiThua(1), GiaiThua(0). Tuy nhiên, vấn đề

thường xảy ra khi số lần gọi đệ quy quá nhiều là ngăn xếp bị tràn (stack overflow). Khi đó chương trình không thể thực thi đúng theo yêu cầu.



Hình 4. Tính giai thừa – Trường hợp không dùng đệ quy đuôi

Trong trường hợp hàm là đệ quy đuôi, hầu hết các trình biên dịch của các ngôn ngữ lập trình đều nhận biết ra điều này và chuyển nó về dạng vòng lặp để thực thi. Việc chuyển về dạng vòng lặp giúp tiết kiệm bộ nhớ lưu trữ dữ liệu trung gian và tránh các lỗi về tràn ngăn xếp. Hình 5 minh họa thời gian tồn tại trong bộ nhớ của chương trình tính giai thừa của 4 bằng đệ quy đuôi.



Hình 5. Tính giai thừa – Trường hợp dùng đệ quy đuôi

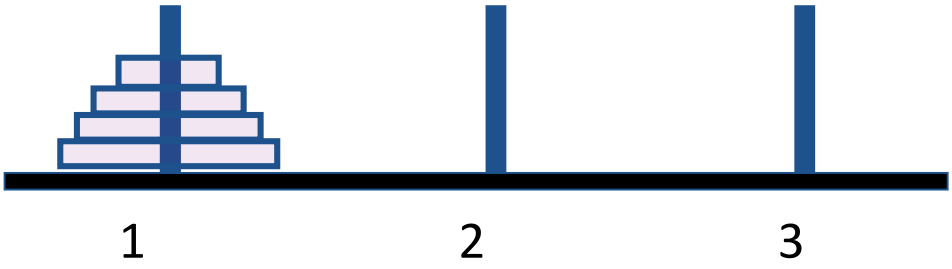
2.5. BÀI TOÁN THÁP HÀ NỘI (TOWER OF HANOI)

Bài toán tháp Hà Nội là một trong những bài toán kinh điển thường được dùng để minh họa cho trường hợp vận dụng giải thuật đệ quy. Bài toán này xuất phát từ trò chơi đố 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:

+ Mỗi lần chỉ được di chuyển một đĩa.

+ Một đĩa chỉ có thể được đặt lên một đĩa lớn hơn".



Hình 6. Trò chơi tháp Hà Nội

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, chúng ta có thể phân tích bài toán theo ý tưởng đệ quy như sau:

+ Nếu *n=1*, chúng ta di chuyển đĩa duy nhất từ vị trí 1 sang vị trí 2 là xong.

+ Giả sử, chúng 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: Chúng ta di chuyển *n-1* phía trên đĩa từ vị trí 1 sang vị trí 3. Tiếp theo, chuyển đĩa nằm dưới cùng sang vị trí 2. Cuối cùng, 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 trên được hiện thực trong chương trình dưới đây:

```
#include<stdio.h>
#include<conio.h>
void Move(int n, int x, int y)//Chuyển n đĩa từ cột x sang cột y
{
	if(n==1)
		printf("\nDi chuyen 1 dia tu %d sang %d", x, y);
	else
	{
		//Chuyển n-1 đĩa từ cột x sang cột trung gian (x+y+z=6)
		Move(n-1, x, 6-x-y);
		//Chuyển 1 đĩa từ cột x sang cột y
		Move(1, x, y);
		//Chuyển n-1 đĩa từ cột trung gian sang cột y
		Move(n-1, 6-x-y, y);
	}
}
void main()
{
```

```
        int n;
        printf("\nNhap so dia: ");
        scanf("%d", &n);
        int x=1, y=2;
        Move(n,x,y);
}
-------------------------------------
Nhập vào: n=3.
Kết quả:
        Di chuyen 1 dia tu 1 sang 2
        Di chuyen 1 dia tu 1 sang 3
        Di chuyen 1 dia tu 2 sang 3
        Di chuyen 1 dia tu 1 sang 2
        Di chuyen 1 dia tu 3 sang 1
        Di chuyen 1 dia tu 3 sang 2
        Di chuyen 1 dia tu 1 sang 2
```

❖ **Nhận xét**

Qua các ví dụ trên, chúng ta có thể nhận thấy lợi ích của đệ quy trong việc giải quyết các bài toán. Một bài toán được 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 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. Phương pháp khử đệ quy sử dụng cấu trúc dữ liệu ngăn xếp (Stack) sẽ được trình bày ở chương 5.

2.6. 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

   $C_n^0 = C_n^n = 1$

   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 nếu n≥0.

   A(m, 0)=A(m-1, 1) nếu m>0.

   A(m, n)= A(m-1, A(m, n-1)) nếu m>0 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 (Tìm kiếm tuyến tính, Tìm kiếm nhị phân), và năm thuật toán sắp xếp thông dụng. Mỗi thuật toán cũng sẽ được đánh giá mức độ hiệu quả dựa trên việc tính toán số phép so sánh.

Người ta chia giải thuật tìm kiếm và sắp xếp ra làm hai loại: Đối với những bài toán mà dữ liệu quá lớn, không thể đọc hết toàn bộ lên bộ nhớ chính, 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 giải thuật ***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 (dữ liệu đọc lên trên RAM) gọi là giải thuật ***tìm kiếm và sắp xếp nội***.

Trong phạm vi giáo trình này, chúng ta chỉ nghiên cứu các thuật toán tìm kiếm và sắp xếp nội. Việc minh họa giải thuật được thực hiện trên một mảng kiểu số nguyên.

3.1. GIẢI THUẬT TÌM KIẾM

Tra cứu thông tin là một trong những nhu cầu rất cần thiết đối với con người. Đặc biệt, việc phải tìm kiếm trên một lượng thông tin lớn sẽ tiêu tốn rất nhiều chi phí. Máy tính là công cụ hỗ trợ đắc lực của con người trong việc lưu trữ dữ liệu và tìm kiếm thông tin. Chẳng hạn, 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 từ kết quả thi.

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 phù hợp cho từng bài toán cụ thể. Sau đây là các giải thuật tìm kiếm trên một danh sách (sử dụng kiểu mảng).

3.1.1. Tìm kiếm tuyến tính

a. Ý tưởng của giải thuật

Đây là giải thuật đơn giản, nguyên tắc cơ bản là: Lần lượt so sánh giá trị cần tìm x 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 của giải thuật:

+ Đầu vào: Mảng các số nguyên A[*n*], *n* là kích thước của mảng. Giá trị cần tìm *x*.

+ Đầu ra: Kết quả tìm kiếm (Tìm thấy hay không thấy)

Bước 1: Gán i=1;//Bắt đầu từ tìm phần tử đầu tiên

Bước 2: So sánh A[i] với x.

- Nếu A[i] = x: Tìm thấy. DỪNG
- Ngược lại: Qua **bước 3**

Bước 3: Tăng i: i = i+1;//Xét phần tử kế tiếp

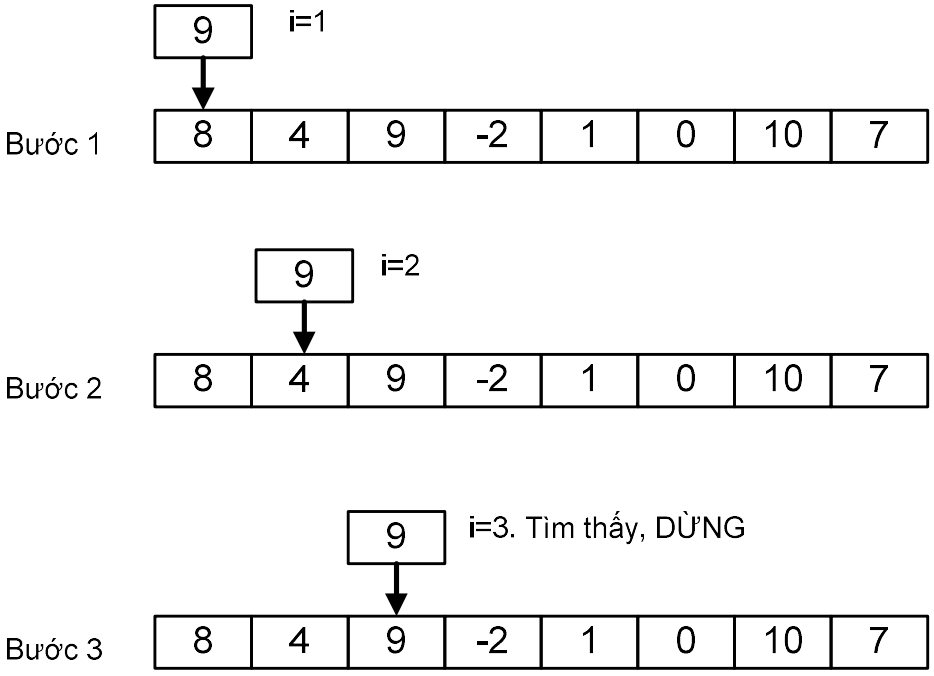
+ Nếu i>n: Hết mảng. Không thấy. DỪNG

+ Ngược lại: Lặp lại bước 2.

b. Ví dụ minh họa

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

Minh họa quá trình tìm kiếm phần tử có giá trị x=9.



c. Cài đặt

Dưới đây là chương trình cài đặt của giải thuật

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

❖ **Đánh giá giải thuật**

Nếu chỉ xét đến số lần so sánh phần tử cần tìm *x* với các phần tử của mảng (không xét đến các phép so sánh để thực hiện vòng lặp) thì trường hợp tốt nhất của giải thật là khi phần tử cần tìm nằm ở vị trí đầu tiên. Khi đó, chỉ cần duy nhất một phép so sánh.

Trường hợp xấu nhất xảy ra khi phần tử cần tìm nằm ở vị trí cuối cùng của mảng, hoặc không tồn tại phần tử này trong mảng. Khi đó, cần *n* phép so sánh.

Như vậy, tổng số phép so sánh của giải thuật (xét theo tất cả các trường hợp) là:

$$\sum_{i=1}^{n} i = 1 + 2 + 3 + \cdots + n = \frac{n(n-1)}{2}.$$

Số phép so sánh trung bình của giải thuật là: $\frac{n(n-1)}{2n} = \frac{n-1}{2}$. Vậy, độ phức tạp của giải thuật là O(*n*).

3.1.2. Tìm kiếm nhị phân

a. Ý tưởng giải thuật

Trong thực tế, chúng ta thấy rằng, thông tin ghi trong một cuốn danh bạ điện thoại, trong một cuốn từ điển, hay trong danh sách thí sinh trong một phòng thi luôn được sắp xếp theo một tiêu chí nào đó. Danh bạ điện thoại thì sắp xếp theo tên chủ nhân. Từ điển được sắp theo bảng chữ cái. Danh sách thí sinh được sắp xếp theo số báo danh. Điều này mang lại thuận lợi cho con người khi cần tra cứu thông tin. Chẳng hạn, khi chúng ta mở một cuốn từ điển tiếng anh để tìm nghĩa của từ "data", trang từ điển mở ra có nội dung là các từ có chữ cái đầu là "p". Chúng ta biết chắc rằng từ "data" sẽ được đặt ở các trang trước trang hiện tại chứ không phải

là các trang sau của trang này vì chữ "d" nằm trước chữ "p" trong bảng chữ cái tiếng anh. Như vậy, chúng ta không cần phải tìm kiếm ở các trang sau. Tiếp tục phương pháp như vậy, công việc tìm kiếm nghĩa của từ trở nên nhanh chóng vì không gian tìm kiếm được thu hẹp dần.

Phương pháp này được vận dụng trong giải thuật tìm kiếm nhị phân, cụ thể như sau:

Trong một danh sách đã sắp thứ tự tăng, tiến hành so sánh giá trị *x* cần tìm với phần tử giữa. Khi đó, xác định được *x* có thể nằm ở nửa trước hay nửa sau của danh sách. Tiếp tục tìm theo phương pháp này trên nửa có khả năng chứa *x* cho đến khi tìm thấy hoặc dãy con không còn phần tử nào.

Các bước thực hiện như sau:

+ Đầu vào: Mảng các số nguyên A[*n*]. Giá trị cần tìm *x*.

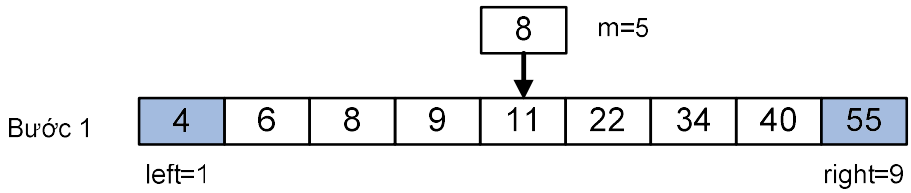
+ Đầu ra: Kết quả tìm kiếm (Tìm thấy hay không thấy)

Bước 1: Gán left=1; right=n;
Bước 2: Gán m = (left + right)/2;
+ Nếu A[m] = x;//Tìm thấy DỪNG
+ Nếu A[m] < x thì right=mid – 1;
+ Nếu A[m] > x thì left = mid + 1;
Bước 3: Nếu left ≤ right thì Lặp lại **bước 2**
Ngược lại: DỪNG

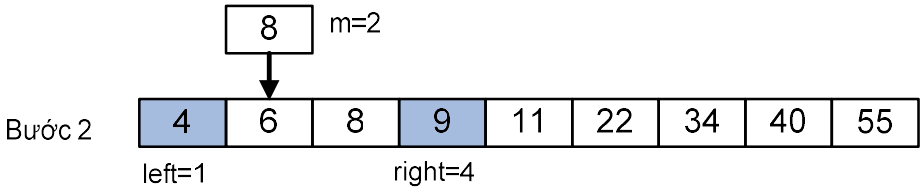
b. Ví dụ minh họa

Cho danh sách các số nguyên (đã sắp xếp tăng) sau: 4, 6, 8, 9, 11, 22, 34, 40, 44

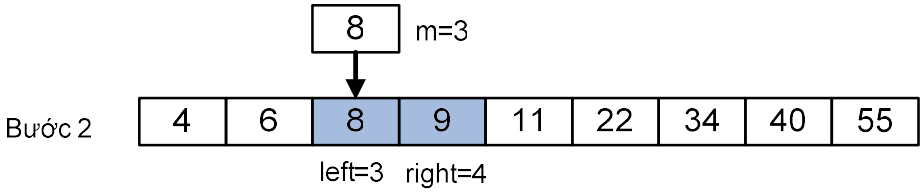
Minh họa quá trình tìm kiếm phần tử có giá trị x =8.



A[m]=11 > 8 => right=m-1=5-1=4.



A[m]=6 < 8 => left=m+1=2+1=3.



A[m]=8 => Tìm thấy, DỪNG.

c. Cài đặt

Dưới đây là chương trình cài đặt của giải thuật

```
int BinarySearch(int a[], int n, int x)
{
	int left=0, right=n-1;
	int m;
	do{
		m=(left+right)/2;
		if(a[m]==x)
			return m;//Tìm thấy x tại ví trí m
		if(x<a[m])
			right=m-1;
		else
			left=m+1;
	}while(left<=right);
	return -1;//Không tìm thấy x
}
```

❖ **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 n). Để tính được số lần so sánh trong trường hợp xấu nhất này, chúng ta xem xét một ví dụ nhỏ như sau.

Giả sử số phần tử của mảng là $n=10$.

+ Lần tìm thứ nhất, chia dãy thành 2 dãy con, số phần tử của mỗi dãy con là: 10/2=5.

+ Lần tìm thứ hai, tìm trong dãy gồm 5 phần tử, chia dãy thành 2 dãy con, số phần tử của mỗi dãy con là: 5/2=2.

+ Lần tìm thứ ba, tìm trong dãy gồm 2 phần tử, chia dãy thành 2 dãy con, số phần tử của mỗi dãy con là: 2/2=1. Dừng.

Như vậy, sau 3 lần lặp thì thuật toán dừng, ta có: $10 \approx 2*2*2=2^3$ ($3 \approx \log_2 10$)

Chúng ta xem như mỗi lần lặp thực hiện một phép so sánh với x (hoặc là một hằng số, tùy theo từng cách cài đặt). Suy ra, số phép so sánh tối đa là: $\log_2 10=3$.

Xét một cách tổng quát, với số phần tử của mảng là n, gọi k là số lần lặp của thuật toán, chúng ta có công thức: $n \approx \underbrace{2*2*\ldots*2}_{k \text{ lần}}=2^k$. Hay $k \approx \log_2 n$. Số phép so sánh tối đa là: $\log_2 n$

Suy ra độ phức tạp của thuật toán O(log n).

Rõ ràng, giải thuật tìm kiếm nhị phân tiết kiệm thời gian tính toán hơn 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, chúng ta cần thực hiện 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.

3.2. GIẢI THUẬT SẮP XẾP

Sắp xếp là quá trình thực hiện di chuyển hay hoán vị các phần tử để đặt chúng theo một thứ tự thỏa mãn một tiêu chuẩn nào đó dựa trên thông tin lữu giữ tại mỗi phần tử.

Khi tiến hành xây dựng thuật toán sắp xếp, chúng 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. Sau đây là các giải thuật sắp xếp thông dụng.

3.2.1. Phương pháp đổi chỗ trực tiếp (Interchange Sort)

a. Ý tưởng giải thuật

Đây được xem là giải thuật sắp xếp đơn giản nhất. Nguyên tắc thực hiện là 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: Gán i = 1;

Bước 2: Gán j = i+1;

Bắt đầu lặp: Trong khi mà j ≤ n:

- Nếu A[j] < A[i] thì Hoán vị (A[i], A[j]);
- Tăng j: j = j+1;

Hết vòng lặp;

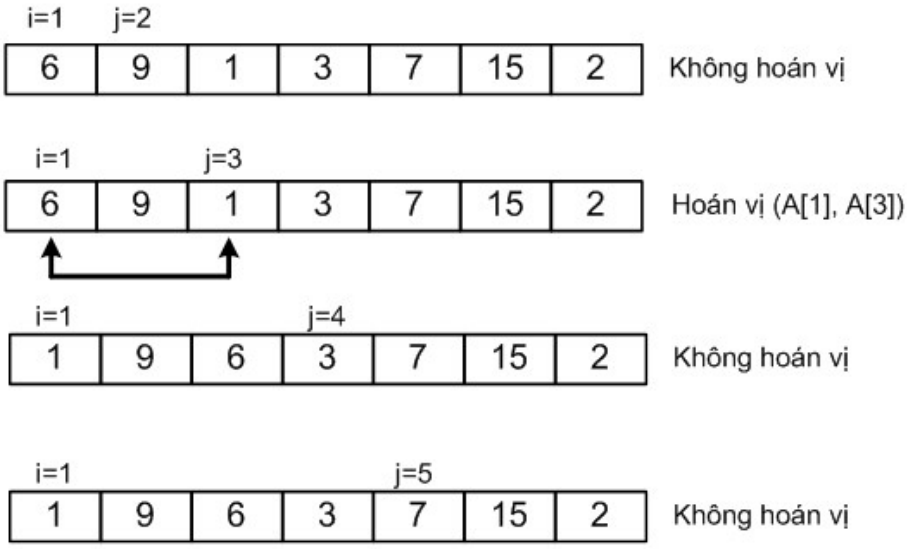
Bước 3: Tăng i: i = i+1;

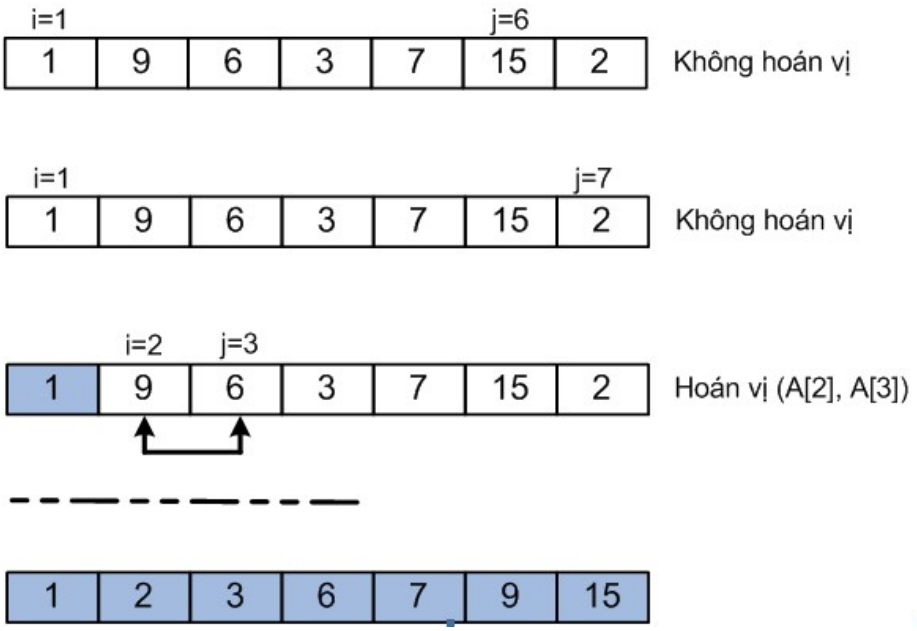
Nếu i < n: Lặp lại **bước 2**;

Ngược lại: DỪNG.

b. Ví dụ minh họa

Sắp xếp danh sách các số nguyên sau: 6, 9, 1, 3, 7, 15, 2.





c. Cài đặt

Dưới đây là hàm cài đặt của thuật toán

```
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;
                        }
}
```

❖ **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. Ta dễ dàng tính được số phép so sánh cần thực hiện như sau:

Tổng số phép so sánh=$\sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1}(1) = \sum_{i=0}^{n-2}(n-i-1) = \frac{n(n-1)}{2}$

Độ phức tạp của giải thuật là O(n^2).

3.2.2. Phương pháp chọn trực tiếp (Selection Sort)

a. Ý tưởng giải thuật

Trong số *n* phần tử của danh sách, chọn phần tử nhỏ nhất, và đặt ở vị trí đầu tiên. Tiếp theo, trong số *n-1* phần tử còn lại, chọn phần tử nhỏ nhất, và đặt ở vị trí thứ hai. Tương tự, lặp lại quá trình này cho đến khi dãy cần duyệt chỉ còn một phần tử.

Các bước thực hiện như sau:

Bước 1: Gán i = 1;

Bước 2: Tìm vị trí minPos của phần tử nhỏ nhất trong dãy từ A[i] đến A[n]

Gán minPos=i, j=i+1;

Bắt đầu lặp: Trong khi mà j<n

-Nếu A[j]<a[minPos] thì Gán minPos=j;

-Tăng j: j=j+1;

Hết vòng lặp.

Bước 3: Nếu minPos ≠ i thì: Hoán vị A[minPos] và a[i]

Bước 4: Nếu i<n-1 thì

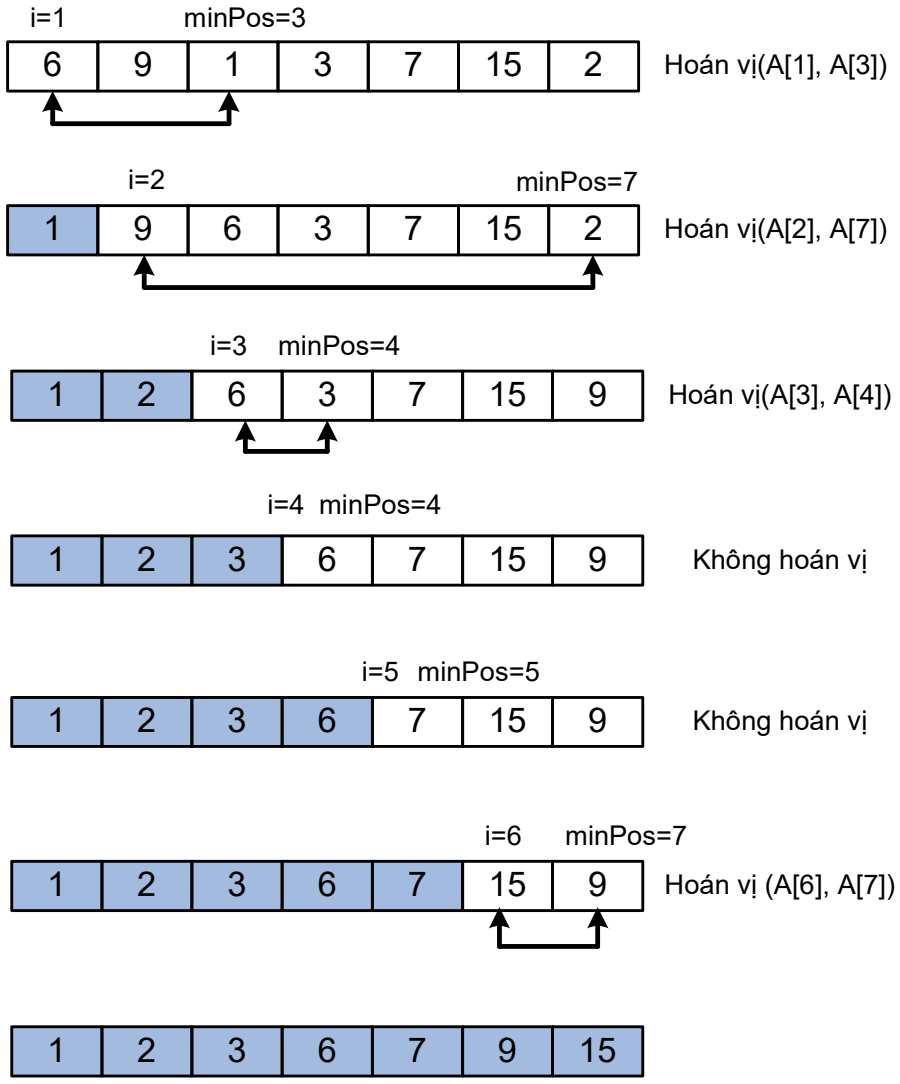
+ Tăng i: i = i+1;

+ Lặp lại **bước 2**

Ngược lại: DỪNG

b. Ví dụ minh họa

Sắp xếp danh sách các số nguyên sau: 6, 9, 1, 3, 7, 15, 2.



c. Cài đặt

Dưới đây là hàm cài đặt của thuật toán.

```
void SelectionSort(int A[], int n)
{
        int minPos;
        for(int i=0;i<n-1;i++)
        {
                //Tìm vị trí phần tử nhỏ nhất
```

```
                minPos=i;
                for(int j=i+1;j<n;j++)
                                if(A[j]<A[minPos])
                                                minPos=j;
                //Hoán vị A[i], A[minIndex]
                if(minPos!=i)
                {
                        int temp=a[i];
                        a[i]=a[minPos];
                        a[minPos]=temp;
                }
        }
}
```

❖ **Đánh giá giải thuật**

Sau mỗi bước lặp theo i, cần tối đa 1 phép hoán vị để đưa phần tử nhỏ nhất về đầu dãy đang xét. Như vậy, một danh sách *n* phần tử, cần nhiều nhất *n*-1 phép hoán vị. Điều này cho thấy thuật toán chọn trực tiếp về mặt lý thuyết có số phép hoán vị nhỏ hơn hoặc bằng thuật toán đổi chỗ trực tiếp. Bởi vì, trong thuật toán này, sau mỗi bước lặp theo i, có thể cần nhiều hơn 1 phép hoán vị để có thể đưa phần tử nhỏ nhất trong dãy đang xét về vị trí đầu (độc giả tự kiểm tra điều này, xem như là bài tập).

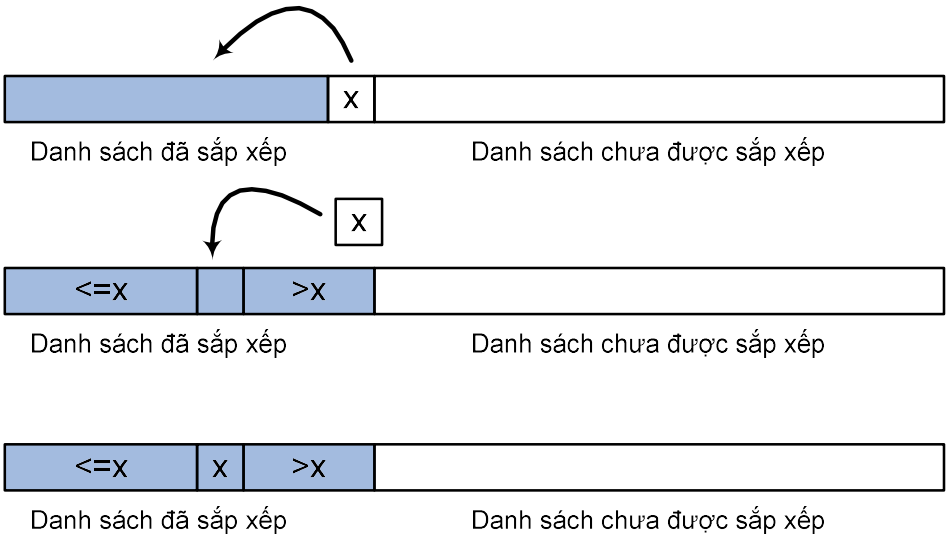
Xét số phép so sánh, hoàn toàn tương tự cách tính trong thuật toán đổi chỗ trực tiếp, ta có tổng số phép so sánh là $\frac{n(n-1)}{2}$. Độ phức tạp của giải thuật là O(n^2).

3.2.3. Phương pháp chèn trực tiếp (Insertion Sort)

a. Ý tưởng giải thuật

Xem danh sách ban đầu *n* phần tử bao gồm 2 danh sách con: Danh sách đã được sắp xếp gồm 1 phần tử đầu tiên, và danh sách các phần tử chưa được sắp xếp gồm *n-1* phần tử còn lại. Tiến hành chèn lần lượt các phần tử trong danh sách chưa được sắp xếp vào danh sách đã được sắp xếp (đảm bảo sau khi chèn, danh sách mới cũng đã được sắp xếp) cho đến khi nào danh sách chưa được sắp xếp không còn phần tử nào.

Ví dụ hình 7: Chèn phần tử x ở đầu danh sách chưa được sắp xếp vào danh sách đã được sắp xếp để được một danh sách đã được sắp xếp mới.



Hình 7. Phương pháp chèn trực tiếp

Các bước thực hiện của giải thuật như sau:

Bước 1: Gán i=2;

Bước 2: Gán x=a[i];

Tìm vị trí pos thích hợp trong đoạn đã sắp xếp từ A[1] đến A[i-1] để chèn x vào.

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

Bước 4: Gán A[pos] = x;

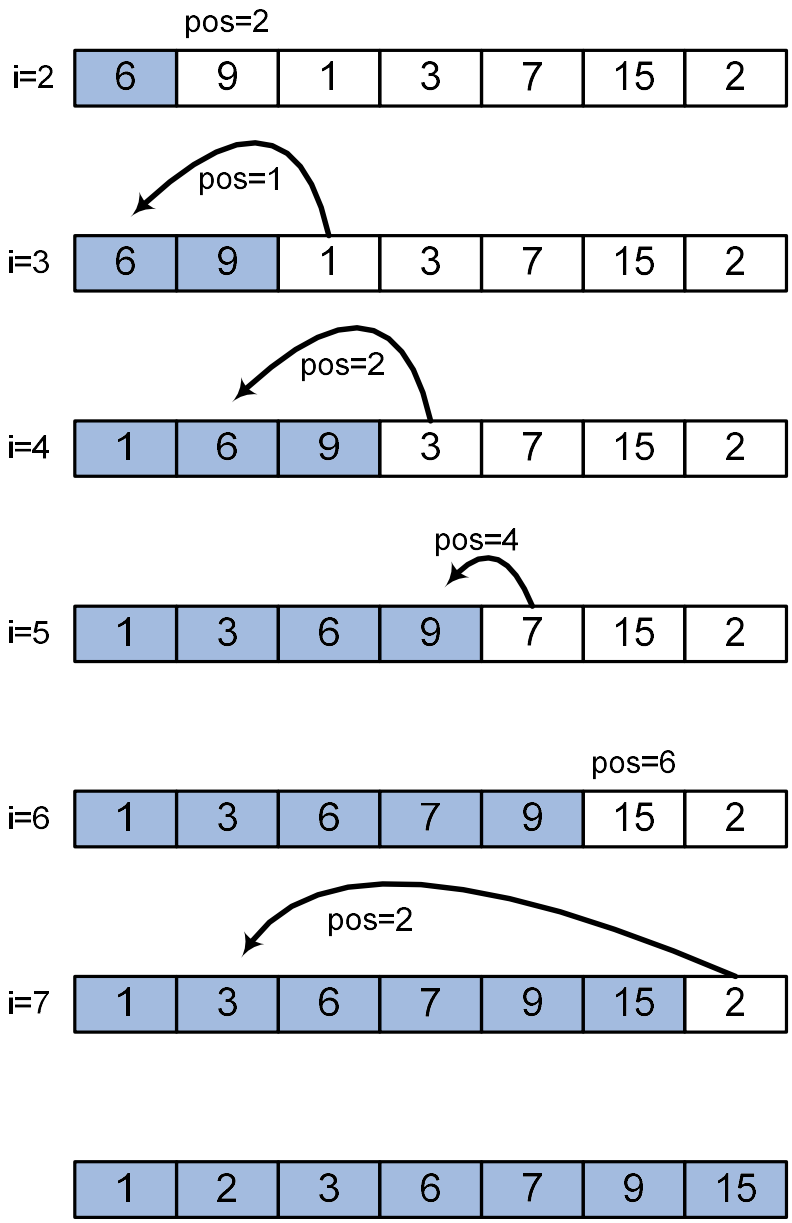
Bước 5: Tăng i: i = i+1;

Nếu i ≤ N: Lặp lại **bước 2**

Ngược lại: DỪNG.

b. Ví dụ minh họa

Sắp xếp danh sách các số nguyên sau: 6, 9, 1, 3, 7, 15, 2.



c. Cài đặt

Trong giải thuật, với mỗi vòng lặp theo i, có hai công việc cần thực hiện là: (1) Tìm vị trí thích hợp để chèn phần tử x, và (2) dịch chuyển các phần tử trong danh sách đã sắp xếp sang phải một vị trí.

Đối với việc tìm vị trí thích hợp để chèn (1), chúng ta có thể thực hiện tìm theo phương pháp *tìm kiếm tuyến tính* (tìm từ đầu dãy về cuối

dãy hoặc ngược lại). Ngoài ra, công việc này có thể được áp dụng phương pháp *tìm kiếm nhị phân* để giảm chi phí tìm kiếm vì dãy đã được sắp xếp.

Hai công việc này (1 và 2), có thể được thực hiện riêng rẽ (tìm xong thì mới tiến hành dịch chuyển các phần tử sang phải một vị trí) hoặc kết hợp (vừa tìm vừa dịch chuyển phần tử). Giải pháp kết hợp chỉ có thể thực hiện khi chúng ta tiến hành tìm kiếm theo phương pháp tìm tuyến tính, và tìm từ cuối dãy về đầu dãy. Dưới đây là mã nguồn cài đặt theo phương pháp này. Bạn đọc tự cài đặt theo các cách còn lại xem như là bài tập.

```
void InsertionSort(int A[], int n)
{
	for(int i=1;i<n;i++)
	{
		int x=A[i];
		/*Tìm vị trí cần chèn và dịch chuyển các phần tử sang
		phải*/
		int j=i-1;
		while(j>=0 && A[j]>x)
		{
			A[j+1]=A[j];
			j--;
		}
		A[j+1]=x;//Chèn x vào vị trí j+1
	}
}
```

❖ Đánh giá giải thuật

Giải thuật chèn trực tiếp được đánh giá là hiệu quả đối với trường hợp tập dữ liệu cần sắp xếp nhỏ, nhưng không hiệu quả đối với tập dữ liệu lớn. Ngoài ra, giải thuật này cũng có một số ưu điểm như đơn giản, dễ cài đặt, có khả năng tận dụng những đoạn dữ liệu đã có thứ tự sẵn giúp giảm chi phí thực thi.

Giải thuật này không đòi hỏi chi phí hoán vị, nhưng tốn nhiều chi phí dịch chuyển các phần tử. Đối với chi phí thực hiện so sánh (chỉ xét các so sánh giá trị các phần tử với nhau) còn phụ thuộc vào tính chất của dữ liệu. Rõ ràng, nếu danh sách đầu vào đã được sắp xếp, số phép so

sánh cần sử dụng là *n-1*. Đây chính là số phép so sánh tối thiểu của giải thuật. Số phép so sánh tối đa được tính như sau:

Tổng số phép so sánh tối đa $=\sum_{i=1}^{n-1}\sum_{j=0}^{i-1}(1)=\sum_{i=1}^{n-1}(i)=\frac{n(n-1)}{2}$

Độ phức tạp của thuật toán là O(n^2).

3.2.4. Phương pháp nổi bọt (Bubble Sort)

a. Ý tưởng giải thuật

So sánh mỗi cặp các phần tử liền kề từ đầu đến cuối danh sách (*n* phần tử), nếu phần tử sau nhỏ hơn phần tử đứng trước thì thực hiện hoán vị, giảm số phần tử còn cần xét còn *n-1* phần tử (bỏ qua phần tử cuối vì đã đúng vị trí). Lặp lại quá trình này cho đến khi nào bước thực hiện trước đó vẫn còn phải hoán vị.

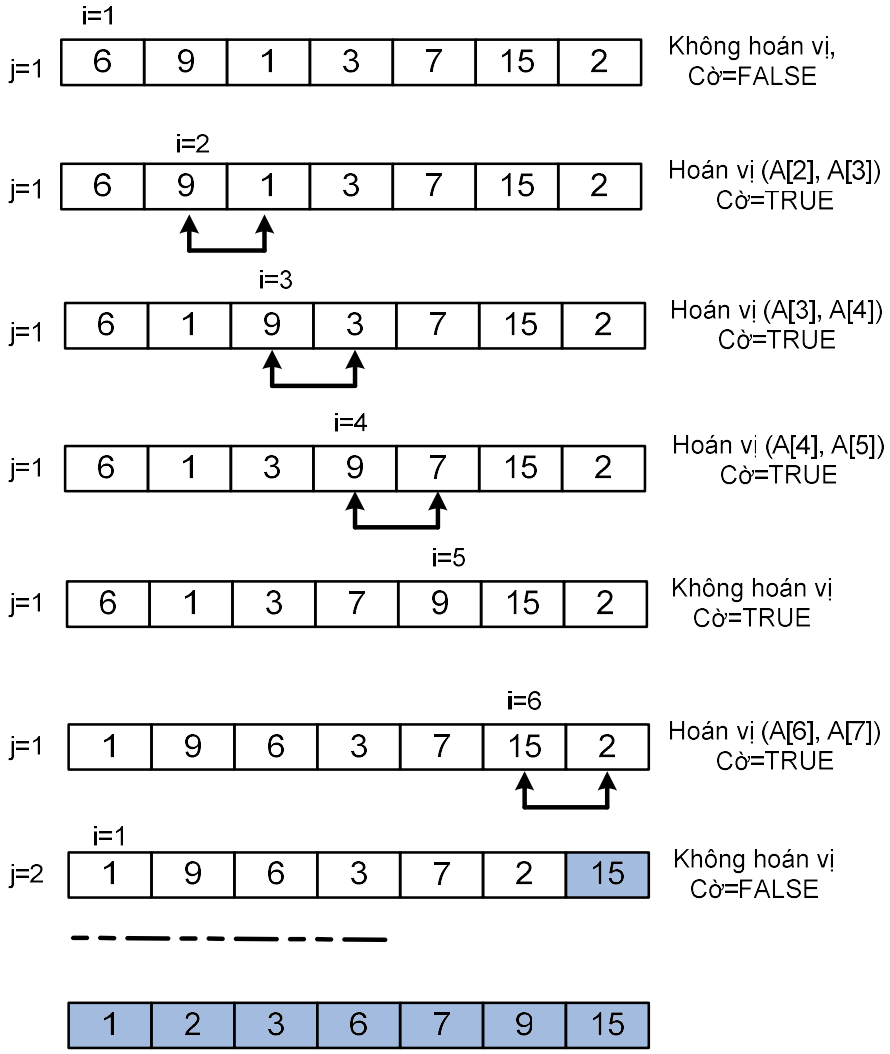
Quá trình này vừa giúp đưa phần tử nhỏ nhất về cuối dãy đang xét, vừa dịch chuyển dần các phần tử lớn về sau, và các phần tử nhỏ về trước.

Các bước thực hiện như sau:

Bước 1: Gán j = 1;.
Bước 2: Gán: Cờ=FALSE, i=1
    Trong khi i<=n-j thực hiện:
        Nếu a[i] > a[i+1] thì
            + HoánVị(a[i], a[i+1]);
            + Gán Cờ=TRUE
        Cuối nếu;
        Tăng i: i=i+1
    Hết vòng lặp;
Bước 3: Tăng j: j=j+1
    Nếu Cờ=FALSE thì DỪNG;
    Ngược lại: Lặp lại **bước 2**

b. Ví dụ minh họa

Sắp xếp danh sách các số nguyên sau: 6, 9, 1, 3, 7, 15, 2.



c. Cài đặt

```
void BubbleSort(int A[], int n)
{
        int flag = 1;
        int j = 0;
        while (flag)
        {
                flag = 0;
                j++;
                for (int i = 0; i < n - j; i++)
```

```
                {
                        if (A[i] > A[i + 1])
                        {
                                int temp = A[i];
                                A[i] = A[i + 1];
                                A[i + 1] = temp;
                                flag = 1;
                        }
                }
        }
}
```

❖ **Đánh giá giải thuật**

Giải thuật sắp xếp nổi bọt có ưu điểm là cài đặt dễ dàng, chi phí thực thi có phụ thuộc vào tính chất của dữ liệu. Tuy nhiên, đây là giải thuật được xem là kém hiệu quả so với các giải thuật khác có cùng độ phức tạp (ví dụ giải thuật chèn trực tiếp).

Số phép hoán vị của giải thuật phụ thuộc vào kết quả so sánh của các cặp phần tử liền kề. Còn số phép so sánh cũng tùy thuộc vào dữ liệu có được sắp xếp một cách tương đối hay chưa (các phần tử ban đầu nằm đúng hoặc gần vị trí so với sau khi đã sắp xếp). Nếu đầu vào của giải thuật là một danh sách đã được sắp xếp thì số phép so sánh cần thiết là *n-1* (số phép so sánh tối thiểu). Số phép so sánh tối đa có thể tính được như sau:

Tổng số phép so sánh tối đa $=\sum_{j=1}^{n-1}\sum_{i=0}^{n-j-1}(1)=\sum_{j=1}^{n-1}(n-j)=\frac{n(n-1)}{2}$.

Độ phức tạp của giải thuật là O(n^2).

3.2.5. Phương pháp dựa trên phân hoạch (Quick Sort)

a. Ý tưởng giải thuật

Giải thuật Quick sort áp dụng giải pháp chia để trị (devide-and-conquer), là một giải thuật được đánh giá chạy nhanh và hiệu quả, đặc biệt là đối với trường hợp tập dữ liệu lớn. Ý tưởng của giải thuật như sau:

+ Chọn một phần tử làm phần tử cầm canh (phần tử pivot), thể là phần tử đầu, cuối hoặc phần tử ngẫu nhiên trong dãy.

+ Chuyển các phần tử nhỏ hơn pivot về phần bên trái của dãy, các phần tử lớn hơn pivot về phần bên phải của dãy. Các phần tử bằng pivot có thể thuộc phần trái, phần phải hoặc nằm giữa hai phần tùy theo tình trạng của dữ liệu.

+ Cũng áp dụng phương pháp tương tự để sắp xếp cho từng phần con (dãy con) bên trái và bên phải. Khi hai các dãy con đã được sắp xếp đồng nghĩa với việc dãy dữ liệu ban đầu đã được sắp xếp.

Các bước thực hiện như sau:

Bước 1:

+ Đặt left là vị trí đầu của dãy, right là vị trí cuối của dãy.

+ Chọn phần tử cầm canh pivot=A[k];//k trong khoảng {1..n}

+ Gán i=left;

+ Gán j=right;

Bước 2:

Trong khi mà i<=j thì:

//Phát hiện và hoán vị cặp phần tử A[i], A[j] nằm sai vị trí:

+ Trong khi A[i] < pivot thì tăng i: i=i+1;

+ Trong khi A[j] > pivot thì giảm j: j=j-1;

+ Nếu i<=j thì Hoán vị A[i] và A[j];

Cuối vòng lặp.

Bước 3:

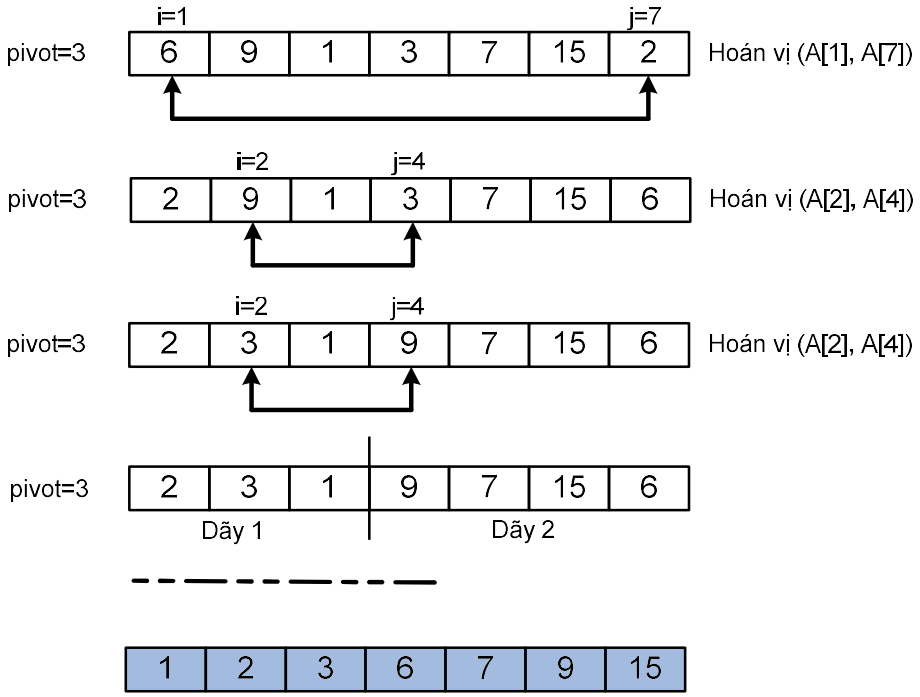
+ Nếu left<j thì: Gọi đệ quy sắp xếp cho dãy từ A[left] đến A[j].

+ Nếu i<right thì: Gọi đệ quy sắp xếp cho dãy từ A[i] đến A[right].

+ DỪNG.

b. Ví dụ minh họa

Sắp xếp danh sách các số nguyên sau: 6, 9, 1, 3, 7, 15, 2. Phần tử cầm canh (pivot) là phần tử giữa của dãy: pivot=A[(left+right)/2]=A[4]=3.



c. Cài đặt

Trong phần cài đặt này, phần tử cầm canh được chọn là phần tử giữa của dãy: pivot=A[(left+right)/2].

```
void QuickSort(int A[], int left, int right)
{
	int i = left, j = right;
	int pivot = A[(left + right) / 2];
	/* Phan hoach */
	while (i <= j)
	{
		while (A[i] < pivot)
			i++;
		while (A[j] > pivot)
			j--;
		if (i <= j)
		{
			int temp = A[i];
			A[i] = A[j];
```

```
                A[j] = temp;
                i++;
                j--;
            }
        };
        /* Goi de quy */
        if (left < j)
            QuickSort(A, left, j);
        if (i < right)
            QuickSort(A, i, right);
}
```

Quá trình phân hoạch được thực hiện như sau:

Hai biến i và j dùng để xác định vị trí của phần tử không nằm đúng phía trái hoặc phải theo ý tưởng phân hoạch. i chạy từ phần tử đầu tiên của dãy (vị trí left), tăng dần cho đến khi gặp phần tử lớn hơn hoặc bằng phần tử cầm canh (pivot). Tương tự, j chạy từ phần tử cuối của dãy (vị trí right), giảm dần cho đến khi gặp phần tử nhỏ hơn hoặc bằng phần tử cầm canh. Nếu sau đó, $i \leq j$ thì hoán vị hai phần tử ở hai vị trí i và j để đưa chúng về đúng phía. Sau đó tăng i đến vị phần tử phía sau và giảm j về vị trí phần tử phía trước. Quá trình thực hiện cho đến khi i lớn hơn j.

Sau khi phân hoạch, các phần tử ở vị trí phía trước phần tử thứ i nhỏ hơn hoặc bằng phần tử cầm canh (pivot). Các phần tử ở vị trí phía sau phần tử thứ j lớn hơn hoặc bằng phần tử cầm canh.

❖ Đánh giá giải thuật

Hiệu quả của giải thuật Quick sort còn phụ thuộc vào việc chọn phần tử cầm canh (pivot) khi phân hoạch. Trường hợp tốt nhất là nếu mỗi lần phân hoạch, chúng ta đều chọn được phần tử cầm canh là phần tử tốt nhất. Tức là, có chính xác một nửa các phần tử thuộc dãy lớn hơn, và một nữa còn lại thuộc dãy nhỏ hơn phần tử cầm canh. Độ phức tạp của giải thuật khi đó là O($nlog\ n$).

Trường hợp xấu nhất là trong mỗi lần phân hoạch, chúng ta luôn chọn phần tử lớn nhất hoặc nhỏ nhất làm phần tử cầm canh. Khi đó, độ phức tạp của giải thuật là O(n^2).

Để tránh xảy ra xảy ra trường hợp xấu nhất, giải pháp được đưa ra là mỗi bước phân hoạch ta thực hiện chọn ngẫu nhiên phần tử cầm canh

(dựa vào chọn ngẫu nhiên vị trí của các phần tử). Giải pháp này đã được chứng minh là hiệu quả trong thực tế.

3.3. 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.

   + Hiển 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

Chương này trình bày một trong những kiểu dữ liệu trừu tượng phổ biến là danh sách. Bên cạnh việc tìm hiểu các tính chất, chúng ta hiện thực danh sách theo các phương pháp khác nhau và đánh giá ưu, nhược điểm của từng phương pháp.

4.1. ĐỊNH NGHĨA

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 0, ta nói đó là danh sách rỗng.

Có thể lấy một vài ví dụ về công việc trong thực tế sử dụng kiểu dữ liệu danh sách như: Quản lý danh sách nhân viên trong một công ty; quản lý danh sách sản phẩm, hàng hóa; quản lý thông tin học sinh sinh viên trong một trường học, hay đơn giản là lưu trữ và tính toán trên dãy số khi giải quyết một bài toán trên máy tính, v.v… Trong phạm vi của giáo trình, chúng ta cài đặt minh họa trên danh sách kiểu số nguyên.

Một số phép toán cơ bản trên danh sách như:

+ Khởi tạo danh sách ban đầu.

+ Kiểm tra danh sách rỗng.

+ Kiểm tra danh sách đầy.

+ Thêm/chèn một phần tử vào danh sách (Thêm phần tử vào đầu, giữa, cuối danh sách).

+ Xóa/hủy một phần tử khỏi danh sách (Xóa phần tử đầu, giữa, cuối danh sách).

+ Tìm kiếm một phần tử trong danh sách

+ Liệt kê tất cả các phần tử của danh sách

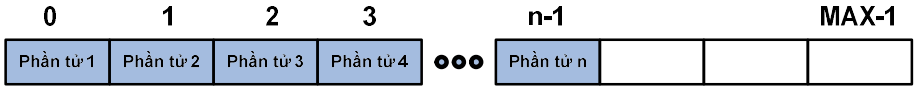
+ Sắp xếp danh sách.

Kiểu danh sách có thể được hiện thực trên máy tính bằng các phương pháp khác nhau. Trong chương này, chúng ta tìm hiểu hai cách cài đặt thông dụng là sử dụng kiểu mảng (khi đó danh sách được gọi là danh sách đặc), và kiểu con trỏ (khi đó danh sách được gọi là danh sách liên kết).

4.2. DANH SÁCH ĐẶC

Danh sách đặc là khái niệm để chỉ kiểu dữ liệu danh sách được cài đặt bằng cách sử dụng kiểu mảng. Ngoài tính chất cho phép lưu trữ các phần tử có cùng kiểu dữ liệu, một đặc điểm đặc trưng của kiểu mảng là các vùng nhớ lưu trữ các phần tử nằm liên tiếp nhau trong bộ nhớ.

Khi cài đặt danh sách bằng kiểu mảng, chúng ta cần một giá trị nguyên (giá trị MAX trong ví dụ bên dưới) cho biết số phần tử tối đa mà danh sách có thể lưu trữ, và một biến kiểu số 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*, chỉ số tối đa của mảng là MAX-1.



Danh sách đặc có thể được khai báo như sau:

#define MAX ***Số_phần tử_tối_đa***

struct **LIST**

{

Kiểu_dữ_liệu_của_phần_tử A[MAX]; //*Mảng các phần tử của danh sách*

int n; //*Độ dài danh sách*

};

Cụ thể, khai báo danh sách đặc kiểu số nguyên như sau:

```
#define MAX 100 //Danh sách tối đa 100 phần tử
struct LIST
{
        int A[MAX]; //Mảng các phần tử của danh sách
        int n; //Độ dài danh sách
};
```

Sau đây, chúng ta tìm hiểu và cài đặt các thao tác cơ bản trên danh sách đặc kiểu số nguyên.

4.2.1. Khởi tạo danh sách

Khi thực hiện khai báo danh sách (ví dụ LIST myList), chương trình mặc định cung cấp cho biến myList một vùng nhớ có kích thước MAX lần kích thước của một phần tử (MAX*kích_thước_mỗi_phần_tử). Thao tác khởi tạo danh sách như thế nào là tùy vào bài toán cụ thể mà chúng ta áp dụng. Ở đây, chúng ta chọn cách khởi tạo ban đầu là gán số phần tử ban đầu là 0 (danh sách rỗng).

```
void InitList(LIST& myList)
{
        myList.n=0;
}
```

4.2.2. Kiểm tra danh sách rỗng

Danh sách rỗng khi số phần tử là 0.

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

4.2.3. Kiểm tra danh sách đầy

Số phần tử tối đa trong một danh sách là MAX. Vì vậy, khi n=MAX, danh sách được xem là đầy. Khi đó, danh sách không thể lưu trữ thêm được phần tử nào khác.

```
int IsFullList(LIST myList)
{
        if(myList.n==MAX)
                return 1;//Danh sách đầy
        return 0;//Danh sách chưa đầy
}
```

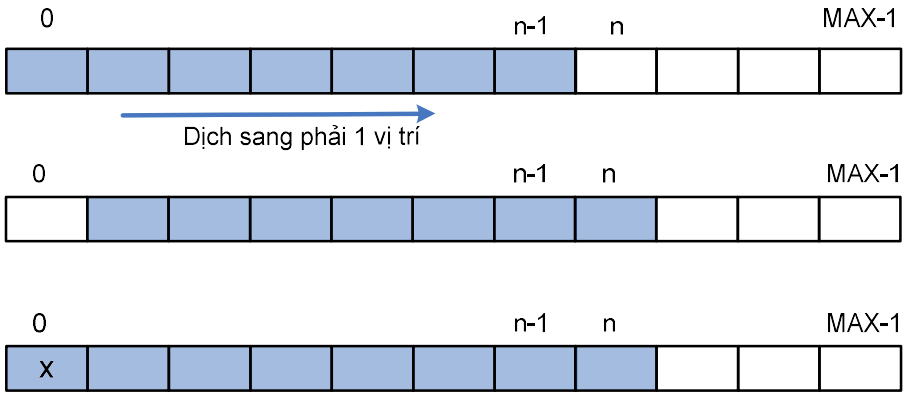
4.2.4. Thêm một phần tử vào danh sách

Một vấn đề cần phải quan tâm khi thực hiện thêm phần tử vào danh sách đặc là kiểm tra xem danh sách có đầy hay không. Vì vậy, trong tất

cả các hàm thêm phần tử, cần phải thực hiện kiểm tra danh sách đầy trước khi thực hiện.

a. Thêm một phần tử vào đầu

Để thêm một phần tử x vào đầu danh sách đặc, chúng ta cần thực hiện dịch chuyển tất cả các phần tử sang phải một vị trí, tăng kích thước của danh sách (tăng *n*), sau đó gán giá trị phần tử đầu bằng x.



Thêm phần tử x vào đầu

Hàm cài đặt như sau:

```
void AddFirst(LIST& myList, int x)
{
	if(IsFullList(myList))
		cout<<"\nDanh sach day!";
	else
	{
		myList.n++;//Tăng kích thước danh sách
		/*Dịch chuyển các phần tử sang phải*/
		for(int i=myList.n;i>0;i--)
			myList.A[i]=myList.A[i-1];
		myList.A[0]=x;//Gán giá trị phần tử đầu bằng x

	}
}
```

b. Thêm một phần tử vào cuối

Thao tác thêm một phần tử x vào cuối danh sách đặc đơn giản hơn so với thao tác thêm vào đầu. Chúng ta chỉ cần tăng kích thước danh sách và gán x vào vị trí cuối.

Hàm cài đặt như sau:

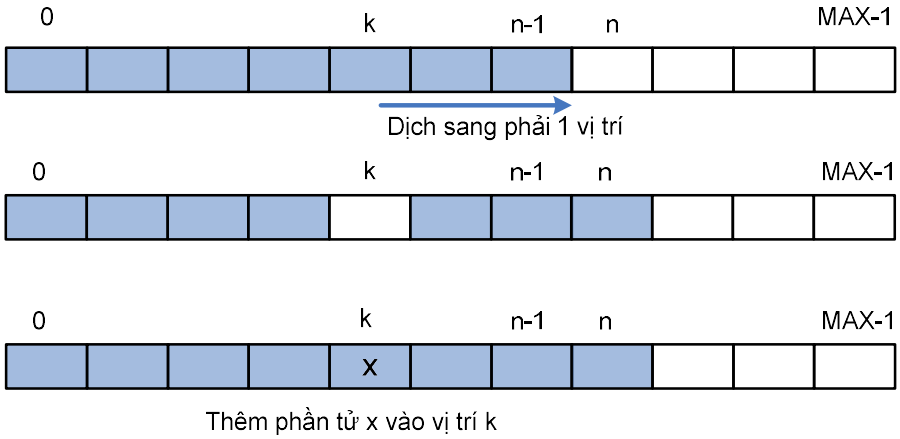
```
void AddLast(LIST& myList, int x)
{
	if(IsFullList(myList))
		cout<<"\nDanh sach day!";
	else
	{
		myList.n++;//Tăng kích thước danh sách
		myList.A[myList.n-1]=x;//Gán phần tử đầu bằng x

	}
}
```

c. Thêm một phần tử vào vị trí k

Theo tác thêm một phần tử x vào vị trí k là thao tác thêm tổng quát vào danh sách (vì nó có thể là thao tác thêm vào đầu, hay vào cuối danh sách tùy theo giá trị của k). Ngoài việc kiểm tra danh sách đầy hay chưa, chúng ta cần kiểm tra vị trí k có hợp lệ hay không. Vị trí hợp lệ là vị trí nằm trong khoảng từ 0 đến n.

Các bước cần thực hiện bao gồm: Dịch chuyển các phần tử từ vị trí k sang phải một vị trí (nếu $0 \le k < n$); tăng kích thước mảng; sau đó gán giá trị vị trí k bằng x. Khi k=0, thao tác này là thao tác thêm một phần tử vào đầu danh sách. Khi k=n, thao tác này là thao tác thêm một phần tử vào cuối danh sách.



Hàm cài đặt như sau:

```
void AddItem(LIST& myList, int x, int k)
{
	if(IsFullList(myList))
		cout<<"\nDanh sach day!";
	else
		if(k<0 || k>myList.n)
			cout<<"\nVi tri can them khong hop le!";
		else
		{
			myList.n++;//Tăng kích thước danh sách
			/*Dịch chuyển các phần tử từ k sang phải*/
			for(int i=myList.n;i>k;i--)
				myList.A[i]=myList.A[i-1];
			myList.A[k]=x;//Gán phần tử vị trí k bằng x
		}
}
```

4.2.5. Hủy một phần tử khỏi danh sách

Khi cài đặt các thao tác hủy một phần tử khỏi danh sách đặc, chúng ta cần kiểm tra xem danh sách có rỗng hay không. Sau đây là một số thao tác hủy cơ bản.

a. Hủy phần tử đầu

Thao tác này thực hiện ngược lại với thao tác thêm phần tử vào đầu danh sách đặc. Chúng ta cần dịch chuyển các phần tử từ vị trí 1 đến n-1 sang trái một vị trí, sau đó giảm kích thước danh sách.

Sau đây là hàm cài đặt:

```
void RemoveFirst(LIST& myList)
{
	if(IsEmptyList(myList))
		cout<<"\nDanh sach rong!";
	else
	{
		/*Dịch các phần tử sang trái*/
		for(int i=0;i<myList.n-1;i++)
			myList.A[i]=myList.A[i+1];
		myList.n--;//Giảm kích thước danh sách
	}
}
```

b. Hủy phần tử cuối

Chúng ta biết rằng, khi khai báo một biến kiểu LIST, chương trình sẽ được cấp phát một vùng nhớ có kích thước cố định cho MAX phần tử. Việc hủy phần tử chỉ có ý nghĩa là chương trình không quan tâm và không sử dụng đến vùng nhớ được cấp phát cho phần tử đó nữa. Tuy nhiên, vùng nhớ này vẫn được hệ thống (hệ điều hành chẳng hạn) xem là chương trình đang sử dụng nó. Vì vậy, với trường hợp hủy phần tử cuối, chúng ta chỉ cần giảm kích thước của danh sách. Có nghĩa là chúng ta không sử dụng đến vùng nhớ cấp phát cho phần tử cuối nữa.

Hàm cài đặt như sau:

```
void RemoveLast(LIST& myList)
{
	if(IsEmptyList(myList))
		cout<<"\nDanh sach rong!";
	else
		myList.n--;//Giảm kích thước danh sách
}
```

c. Hủy phần tử ở vị trí k

Tương tự thao tác thêm phần tử ở vị trí k, chúng ta cần kiểm tra vị trí k có hợp lệ hay không. k hợp lệ khi nằm trong khoảng từ 0 đến n-1. Quá trình thực hiện trải qua hai bước: Dịch chuyển các phần tử sau vị trí k sang trái một vị trí (nếu 0≤k<n-1) và giảm kích thước danh sách.

Sau đây là mã nguồn cài đặt:

```
void RemoveItem(LIST& myList, int k)
{
	if(IsEmptyList(myList))
		cout<<"\nDanh sach rong!";
	else
		if(k<0 || k>=myList.n)
			cout<<"\nVi tri can huy khong hop le!";
		else
		{
			for(int i=k;i<myList.n-1;i++)
				myList.A[i]=myList.A[i+1];
			myList.n--;
		}
}
```

❖ Trên đây là một số thao tác cơ bản trên kiểu dữ liệu danh sách đặc. Những hàm này có thể được sử dụng để cài đặt cho những thao tác phức tạp hơn. Người đọc tự xây dựng các thao tác khác như: tìm kiếm trên danh sách, sắp xếp nội dung danh sách bằng các phương pháp đã trình bày ở chương 3.

❖ **Nhận xét**

Danh sách đặc có ưu điểm là đơn giản, dễ hiểu, dễ cài đặt. Ngoài ra, việc truy suất đến từng phần tử cũng nhanh chóng bằng cách định vị theo chỉ số của mảng A[i]. Tuy nhiên, danh sách đặc có những nhược điểm như sau:

- *Cấp phát vùng nhớ thiếu linh hoạt*

Chúng ta thấy rằng, danh sách đặc yêu cầu phải được cấp phát vùng nhớ cố định ngay từ ban đầu. Vì vậy, có thể thường xuyên xảy ra khả năng là mặc dù chương trình khai báo một số lượng phần tử của danh sách rất lớn, nhưng ở thời điểm thực thi, chương trình chỉ dùng đến một số ít phần tử. Vùng nhớ còn lại không được dùng đến, cũng không được

giải phóng cho các chương trình khác trên máy tính sử dụng. Điều này là gây ***lãng phí bộ nhớ*** trên máy tính.

Ngoài ra, việc cấp phát vùng nhớ tĩnh của danh sách đặc còn có thể dẫn đến vấn đề ***cấp phát thiếu vùng nhớ***. Chẳng hạn, ban đầu chương trình cấp phát mặc định MAX phần tử, nhưng trong thời điểm thực thi, nhu cầu lưu trữ dữ liệu trên bộ nhớ chính vượt quá số lượng cấp phát ban đầu.

Hơn thế nữa, một trường hợp khác có thể xảy ra (nhưng rất hiếm) là khi tổng dung lượng vùng nhớ còn trống trong bộ nhớ có đủ nhưng ***không thể cấp phát*** cho một danh sách vì các vùng nhớ trống không liên tiếp nhau.

- *Chi phí thao tác trên danh sách tốn kém*

Đây là điểm hạn chế lớn của danh sách đặc. Rõ ràng, các thao tác thêm, xóa, sắp xếp nội dung trên danh sách đặc đòi hỏi tác vụ di chuyển, hay hoán vị nội dung phần tử. Trong các ứng dụng thực tế, dữ liệu thực có thể rất lớn. Việc di chuyển hay hoán vị nội dung không phải là một cách hiệu quả vì tốn nhiều thời gian và chi phí lưu trữ.

Trong phần tiếp theo, chúng ta tìm hiểu một cách hiện thực kiểu danh sách khác là danh sách liên kết. Kiểu dữ liệu này giúp hạn chế các nhược điểm của kiểu danh sách đặc.

4.3. DANH SÁCH LIÊN KẾT

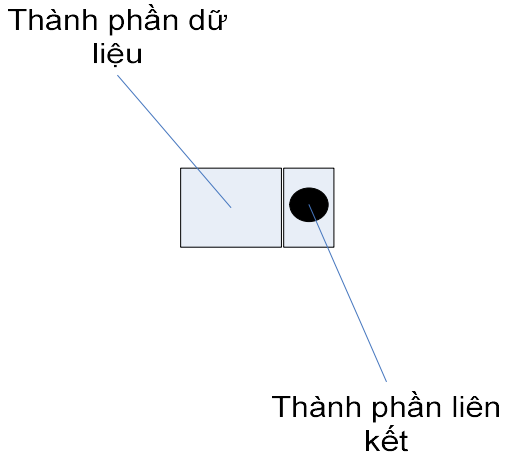
Danh sách liên kết (Linked list) là một kiểu danh sách được hiện thực bằng cách sử dụng kiểu dữ liệu con trỏ. Kiểu danh sách này cho phép thêm, hủy phần tử một cách linh động, đồng thời giảm chi phí thao tác so với danh sách đặc. Hai dạng danh sách liên kết thông dụng là danh sách liên kết đơn và danh sách liên kết kép.

4.3.1. Danh sách liên kết đơn

Trong danh sách liên kết đơn, mỗi phần tử (còn gọi là node) là một cấu trúc chứa hai thành phần thông tin:

+ 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 rỗng Trong phạm vi giáo trình này, chúng ta sử dụng ký hiệu NULL để biểu diễn cho giá trị rỗng.



Mỗi phần tử được định nghĩa tổng quát như sau:

```
struct NODE
{
        Kiểu_dữ_liệu_của_phần_tử Info; //Thông tin của Node
        struct NODE* next; //Con trỏ chỉ đến phần tử node tiếp theo
};
```

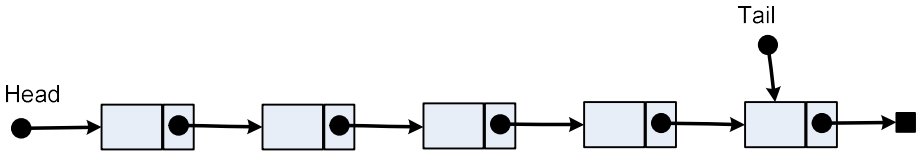
Cụ thể, định nghĩa cấu trúc một phần tử cho danh sách liên kết đơn kiểu số nguyên như sau:

```
struct NODE
{
        int info;
        struct NODE*next;
};
```

Nói một cách khác, *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*. Còn 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, chúng ta gọi là phần tử **Head**. Head là con trỏ cùng kiểu với các phần tử trong danh sách mà nó quản lý. Trong trường hợp giá trị Head = NULL thì danh sách mà Head quản lý là danh sách rỗng.

Về nguyên tắc, chúng ta chỉ cần phần tử Head để quản lý danh sách. Khi có phần tử Head, chúng ta có thể truy suất đến tất cả các phần

tử của danh sách thông qua giá trị **next** tại mỗi phần tử. Tuy nhiên, trong một số trường hợp, chúng ta cần truy suất ngay đến phần tử cuối cùng của danh sách. Khi đó, đòi hỏi phải thực hiện việc truy suất từ đầu danh sách mới có thể lấy được địa chỉ của phần tử cuối cùng. Điều này gây tốn kém chí phí. Vì vậy, người ta đưa thêm vào một biến lưu địa chỉ của phần tử cuối của danh sách, chúng ta gọi là phần tử **Tail**.



Kiểu dữ liệu danh sách liên kết đơn (Singly linked list) được định nghĩa như sau:

```
struct LINKEDLIST
{
        NODE*Head;
        NODE*Tail;
};
```

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

a. Khởi tạo danh sách

Danh sách liên kết đơn được quản lý bởi con trỏ Head và Tail. Ở bước khởi tạo ban đầu, danh sách chưa có phần tử nào, chúng ta gán giá trị hai con trỏ này là NULL.

```
void InitList(LINKEDLIST& myList)
{
        myList.Head=myList.Tail=NULL;
}
```

b. Kiểm tra danh sách rỗng

Danh sách rỗng khi không chứa bất cứ phần tử nào. Khi đó, giá trị hai con trỏ Head và Tail là NULL. Chúng ta chỉ cần kiểm tra giá trị của một trong hai con trỏ này để biết danh sách có rỗng hay không.

```
int IsEmptyList(LINKEDLIST myList)
{
        if(myList.Head==NULL)
```

```
            return 1;//Danh sách rỗng
        return 0;//Danh sách không rỗng
}
```

c. Thêm một phần tử vào danh sách

Danh sách liên kết là kiểu dữ liệu động. Tức là, vùng nhớ lưu trữ cho các phần tử sẽ chỉ được cấp phát khi nào cần, và sẽ được giải phóng khi không sử dụng đến nữa. Vì vậy, việc đầu tiên cần làm cho thao tác thêm một phần tử mới vào danh sách là cấp phát vùng nhớ và gán dữ liệu cho phần tử đó. Trường hợp cấp phát không được, có nghĩa là bộ nhớ máy tính đã đầy, không đủ để lưu trữ thêm dữ liệu. Dưới đây, chúng ta xây dựng hàm **CreateNode** để cấp phát vùng nhớ mới cho một phần tử, dữ liệu lưu trữ trong phần tử là một giá trị nguyên x.

```
NODE*CreateNode(int x)
{
        NODE*p=new NODE;
        if(p==NULL)
        {
                cout<<"\nKhong du bo nho";
                return NULL;
        }
        p->info=x;
        p->next=NULL;
        return p;
}
```

Để tạo ra phần tử mới (p) có giá trị là x, ta chỉ cần gọi hàm:

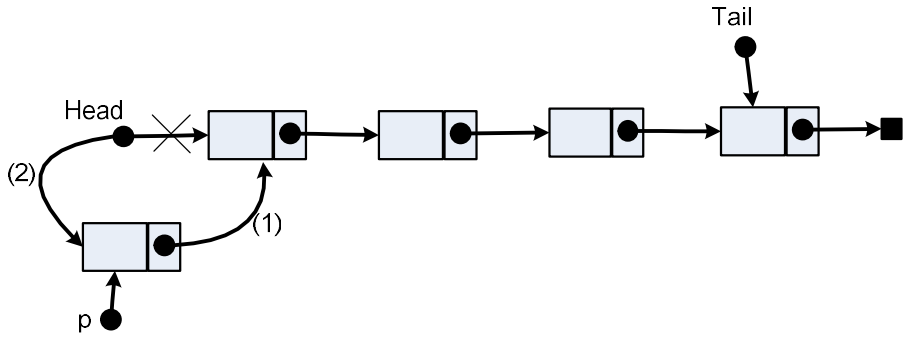
```
NODE*p=CreateNode(x);
```

Phần tử p sẽ là đầu vào cho các hàm thêm phần tử vào danh sách liên kết dưới đây.

Ưu điểm của thao tác thêm vào danh sách liên kết so với thêm vào danh sách đặc là *không di chuyển vị trí lưu trữ* của các phần tử trong bộ nhớ. Việc cần làm là *gán địa chỉ* vùng nhớ để tạo mối liên kết giữa các phần tử.

- **Thêm một phần tử vào đầu**

Để thêm phần tử vào đầu danh sách liên kết, chúng ta cần kiểm tra xem danh sách có rỗng hay không. Bởi vì, khi danh sách rỗng, sau khi thêm phần tử mới, giá trị con trỏ Tail thay đổi. Ngược lại, chúng ta chỉ cần quan tâm đến con trỏ Head mà thôi.



Hai thao tác cần thực hiện:

Thao tác (1): Gán địa chỉ của phần tử đầu vào con trỏ next của p:

p-> next=Head.

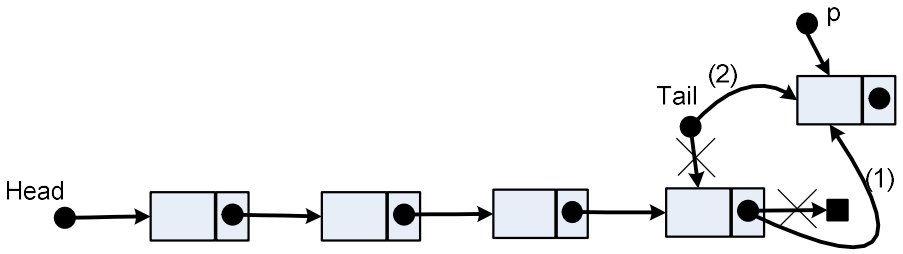
Thao tác (2): Cập nhật lại con trỏ Head, phần tử mới thêm vào là phần tử đầu: Head=p.

Sau đây là hàm cài đặt:

```
Void AddFirst(LINKEDLIST& myList, NODE*p)
{
        if(IsEmptyList(myList))//Danh sách rỗng
                myList.Head=myList.Tail=p;
        else//Danh sách không rỗng
        {
                p->next=myList.Head;
                myList.Head=p;
        }
}
```

- **Thêm một phần tử vào cuối**

Tương tự, để thêm một phần tử vào cuối danh sách liên kết, chúng ta cần kiểm tra xem danh sách có rỗng hay không. Bởi vì, khi đó, cần cập nhật lại giá trị của con trỏ đầu Head.



Hai thao tác cần thực hiện:

Thao tác (1): Gán địa chỉ p cho con trỏ next của phần tử cuối:

Tail->next=p.

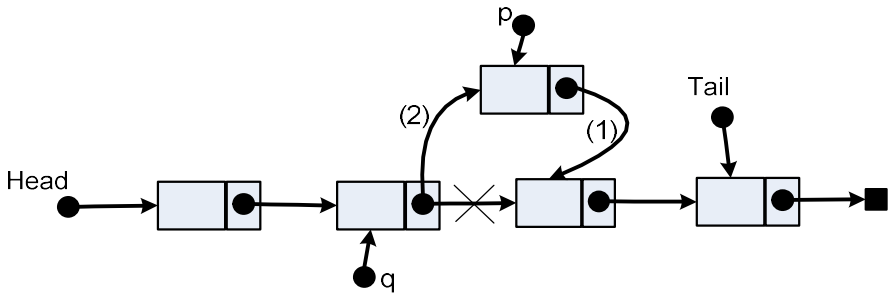
Thao tác (2): Cập nhật lại con trỏ Tail, phần tử mới thêm vào là phần tử cuối: Tail=p.

Sau đây là hàm cài đặt:

```
void AddLast(LINKEDLIST& myList, NODE*p)
{
        if(IsEmptyList(myList))//Danh sách rỗng
                myList.Head=myList.Tail=p;
        else//Danh sách không rỗng
        {
                myList.Tail->next=p;
                myList.Tail=p;
        }
}
```

- **Thêm một phần tử sau phần tử q**

Phần tử q là một phần tử tồn tại trong danh sách liên kết có sẵn. Khi thêm một phần tử mới p vào sau q, chúng ta cần kiểm tra xem q có phải là phần tử cuối của danh sách hay không. Nếu q là phần tử cuối, cần cập nhật lại con trỏ Tail.



Hai thao tác cần thực hiện:

Thao tác (1): Gán địa chỉ của phần tử phía sau q vào con trỏ next của p: p->next=q->next.

Thao tác (2): Gán địa chỉ p vào con trỏ next của q: q->next=p.

Sau đây là hàm cài đặt:

```
void AddNode(LINKEDLIST& myList, NODE*q, NODE*p)
{
        p->next=q->next;
        q->next=p;
        if(myList.Tail==q)
                myList.Tail=p;
}
```

d. Hủy một phần tử khỏi danh sách

Quá trình hủy một phần tử p khỏi danh sách liên kết bao gồm hai công việc chính: (1) Loại bỏ phần tử p ra khỏi mối liên kết của danh sách và (2) Giải phóng vùng nhớ của p.

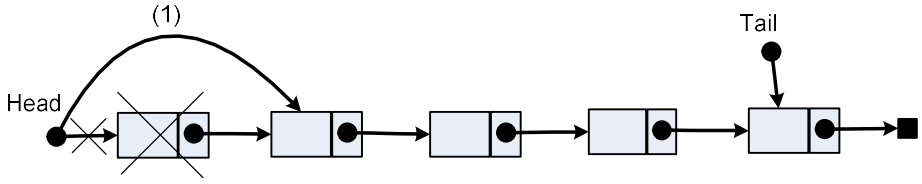
- **Hủy phần tử đầu**

Khi hủy phần tử đầu, những khả năng có thể xảy ra bao gồm:

+ Danh sách rỗng: Không cho phép hủy phần tử

+ Danh sách có nhiều hơn một phần tử: Con trỏ Head bị thay đổi.

+ Danh sách có một phần tử: Con trỏ Head và Tail cùng bị thay đổi (bằng NULL).



Để loại bỏ phần tử đầu ra khỏi danh sách liên kết, chúng ta cần thực hiện một thao tác: (1) Gán địa chỉ phần tử thứ hai vào con trỏ Head: Head=Head->next.

Sau đây là mã nguồn cài đặt:

```
void RemoveFirst(LINKEDLIST& myList)
{
	if(IsEmptyList(myList))//Danh sách rỗng
		cout<<"\nDanh sach rong";
	else
	{
		NODE*p=myList.Head;
		if(myList.Head==myList.Tail)//Danh sách có 1 phần tử
			myList.Head=myList.Tail=NULL;
		else//Danh sách có nhiều hơn 1 phần tử
			myList.Head=myList.Head->next;
		delete p;//Giải phóng vùng nhớ
	}
}
```

- **Hủy phần tử cuối**

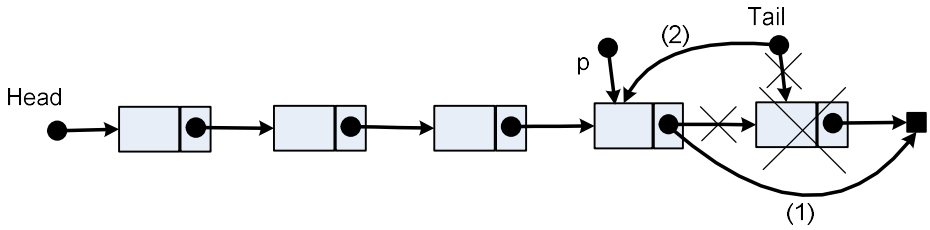
Khi hủy phần tử cuối, những khả năng có thể xảy ra bao gồm:

+ Danh sách rỗng: Không cho phép hủy phần tử

+ Danh sách có nhiều hơn một phần tử: Con trỏ Tail bị thay đổi.

+ Danh sách có một phần tử: Con trỏ Head và Tail cùng bị thay đổi (bằng NULL).

Để loại bỏ phần tử cuối ra khỏi danh sách liên kết, đòi hỏi chúng ta phải thực hiện duyệt danh sách từ đầu để có địa chỉ của phần tử p nằm trước phần tử cuối.



Hai thao tác cần thực hiện để loại bỏ phần tử cuối:

Thao tác (1): Gán con trỏ next của p bằng NULL: p=NULL.

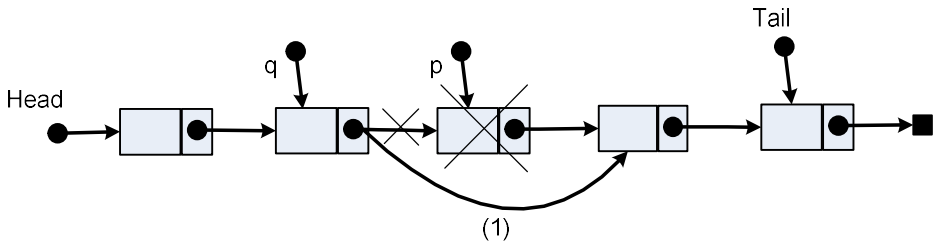
Thao tác (2): Cập nhật để phần tử kế cuối là phần tử cuối: Tail=p.

Sau đây là hàm cài đặt:

```
void RemoveLast(LINKEDLIST& myList)
{
	if(IsEmptyList(myList))//Danh sách rỗng
		cout<<"\nDanh sach rong";
	else
	{
		NODE*q=myList.Tail;
		if(myList.Head==myList.Tail)//Danh sách có 1 phần tử
			myList.Head=myList.Tail=NULL;
		else//Danh sách có nhiều hơn 1 một tử
		{
			NODE*p;
			for(p=myList.Head;p->next!=myList.Tail;p=p->next);
			p->next=NULL;
			myList.Tail=p;
		}
		delete q;//Giải phóng vùng nhớ
	}
}
```

- **Hủy phần tử sau phần tử q**

Chúng ta cần kiểm tra phần tử sau q có tồn tại hay không trước khi thực hiện hủy nó ra khỏi danh sách liên kết. Trong trường hợp phần tử bị hủy là phần tử cuối, con trỏ Tail cần được cập nhật lại giá trị.



Trong hình trên, p là phần tử sau phần tử q. Thao tác cần thực hiện: (1) Gán địa chỉ phần tử phía sau p vào con trỏ next của q: q->next=p->next.

Sau đây là hàm cài đặt:

```
void RemoveNode(LINKEDLIST& myList, NODE*q)
{
	NODE*p=q->next;
	if(p==NULL)
		cout<<"\nKhong ton tai phan tu sau q";
	else
	{
		q->next=p->next;
		if(p==myList.Tail)//Nếu p là phần tử cuối
			myList.Tail=q;
		delete p;
	}
}
```

e. Tìm kiếm một phần tử trên danh sách

Để tìm kiếm một phần tử trên danh sách liên kết, chúng ta có thể thực hiện tìm tuyến tính bằng cách duyệt từ phần tử đầu đến phần tử cuối của danh sách. Hàm cài đặt sau dùng để tìm một phần tử có giá trị x trong danh sách:

```
int SearchNode(LINKEDLIST myList, int x)
{
	NODE*p=myList.Head;
```

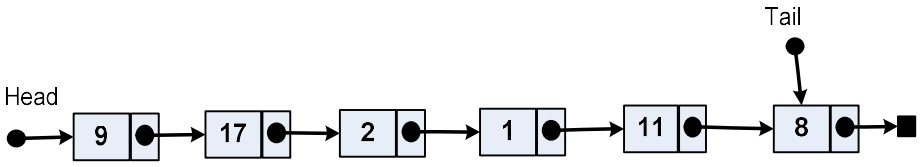
```
        while(p!=NULL)
        {
                if(p->info==x)
                        return 1;//Tìm thấy
                p=p->next;
        }
        return 0;//Không tìm thấy
}
```

f. Sắp xếp danh sách

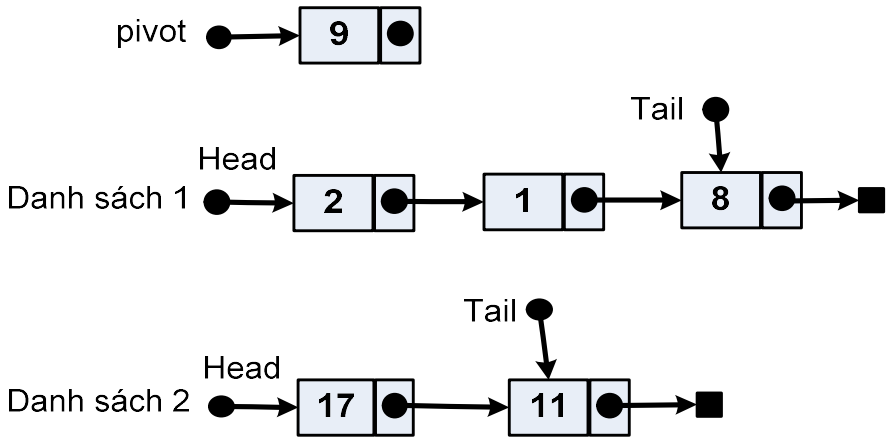
Trong chương 3, chúng ta đã tìm hiểu một số phương pháp sắp xếp trên kiểu dữ liệu mảng. Các thuật toán này vẫn có thể được áp dụng tương tự cho danh sách liên kết bằng cách thực hiện *hoán vị hay dịch chuyển nội dung* của các phần tử trong danh sách. Khi đ, chi phí thực thi của các giải thuật không tốt hơn so với cách sử dụng mảng, thậm chí còn gây khó khăn khi cài đặt hơn. Vì vậy, đối với danh sách liên kết, khi cài đặt các hàm sắp xếp, người ta xây dựng theo nguyên tắc *hoán vị hay gán địa chỉ* của các phần tử, nhằm giảm chi phí thực thi của thuật toán.

Trong phần này, chúng ta tìm hiểu phương pháp sắp xếp danh sách liên kết bằng thuật toán ***Quick sort***. Trong minh họa này, phần tử cầm canh (pivot) được chọn là phần đầu của danh sách. Dựa vào phần tử cầm canh, chia danh sách thành hai danh sách con. Danh sách 1 bao gồm các phần tử nhỏ hơn pivot, các phần tử còn lại thuộc danh sách 2. Sau đó, thực hiện sắp xếp trên từng danh sách con.

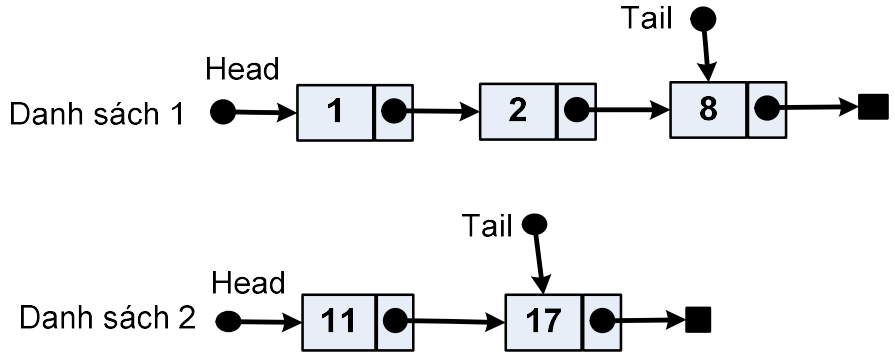
Danh sách ban đầu:



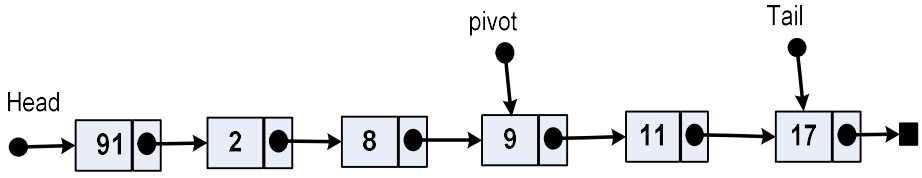
Phần tử pivot và hai danh sách con:



Các danh sách con sau khi đã sắp xếp:



Danh sách sau khi đã sắp xếp:



Sau đây là mã nguồn cài đặt của thuật toán. Trong thuật toán này, chúng ta sử dụng hàm thêm một phần tử vào cuối danh sách (***AddLast()***). Hàm này xem như có sẵn.

```
void QuickSort(LINKEDLIST& myList)
{
        LINKEDLIST myList1;
        LINKEDLIST myList2;
```

```
NODE *pivot, *p;
InitList(myList1);
InitList(myList2);
/*Trường hợp danh sách rỗng hoặc có 1 phần tử*/
if (myList.Head==myList.Tail)
	return;
/*Phân hoạch danh sách thành 2 danh sách con*/
pivot = myList.Head;//Phần tử cầm canh
p=myList.Head->next;
while (p!=NULL)
{
	NODE*q = p;
	p=p->next;
	q->next=NULL;
	if (q->info < pivot->info)
		AddLast(myList1, q);//Thêm vào cuối danh sách 1
	else
		AddLast(myList2, q);//Thêm vào cuối danh sách 2
};
//Gọi đệ quy sắp xếp cho các danh sách con
QuickSort(myList1);
QuickSort(myList2);
//Ghép nối danh sách 1 + pivot
if (!IsEmptyList(myList1))
{
	myList.Head=myList1.Head;
	myList1.Tail->next=pivot;
}
else
	myList.Head=pivot;
//Ghép nối pivot + danh sách 2
pivot->next=myList2.Head;
if (!IsEmptyList(myList2))
	myList.Tail=myList2.Tail;
```

```
        else
                myList.Tail=pivot;
}
```

Trong phần này, chúng ta đã tìm hiểu các thao tác cơ bản trên danh sách liên kết đơn. Người đọc có thể tham khảo mã nguồn của toàn bộ chương trình cài đặt ở phần phụ lục của giáo trình.

4.4. 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 mã số sinh viên (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 VÀ HÀNG ĐỢI

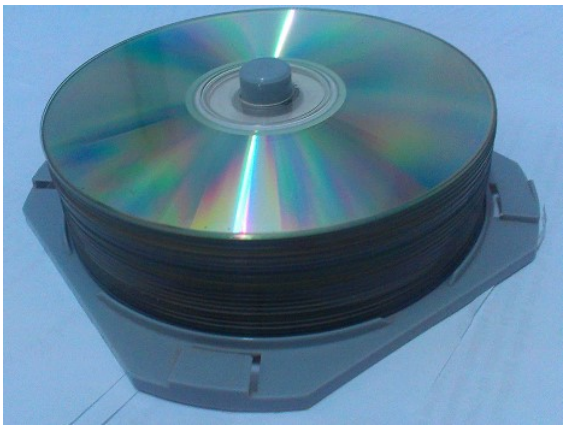
Chương này trình bày hai kiểu dữ liệu trừu tượng là ngăn xếp (Stack) và hàng đợi (Queue). Đây là hai kiểu dữ liệu cho phép người sử dụng chúng giải quyết tình huống đòi hỏi thứ tự thực hiện công việc (yêu cầu trước giải quyết sau, hay yêu cầu trước giải quyết trước) trong lập trình.

5.1. NGĂN XẾP

5.1.1. Định nghĩa ngăn xếp

Ngăn xếp (Stack) là một dạng danh sách được cài đặt nhằm sử dụng cho các ứng dụng cần xử lý theo thứ tự 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. 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).



Hình 8. Minh họa ngăn xếp

Ngăn xếp là một cấu trúc dữ liệu đơn giản nhưng rất hữu ích cho việc xây dựng giải thuật cài đặt trên máy tính. Nó được dùng trong những bài toán cần xử lý đảo ngược, khử đệ quy. Chẳng hạn, những bài toán cần

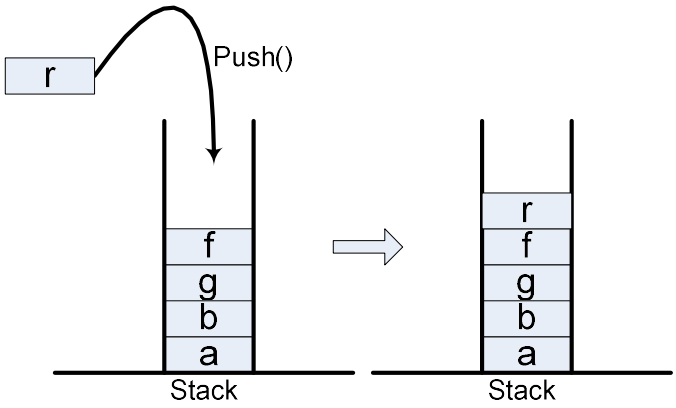
lưu vết trong lý thuyết đồ thị, trí tuệ nhân tạo, hay các chương trình dịch (trình biên dịch, trình thông dịch) trên máy tính, hệ điều hành rất thường sử dụng kiểu dữ liệu ngăn xếp.

5.1.2. Một số phép toán trên ngăn xếp

Ba thao tác chính trên ngăn xếp là: Thêm một phần tử vào ngăn xếp (Push()), lấy thông tin phần tử đầu (Top()), trích hủy phần tử ra khỏi ngăn xếp (Pop()).

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

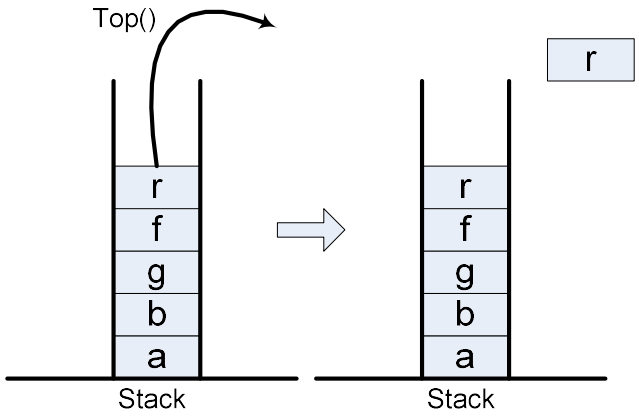
Phần tử mới được thêm vào ngăn xếp luôn luôn nằm ở đỉnh ngăn xếp (top). Trong ví dụ này, r là phần tử mới được thêm vào ngăn xếp.



Hình 9. Thao tác thêm phần tử vào ngăn xếp

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

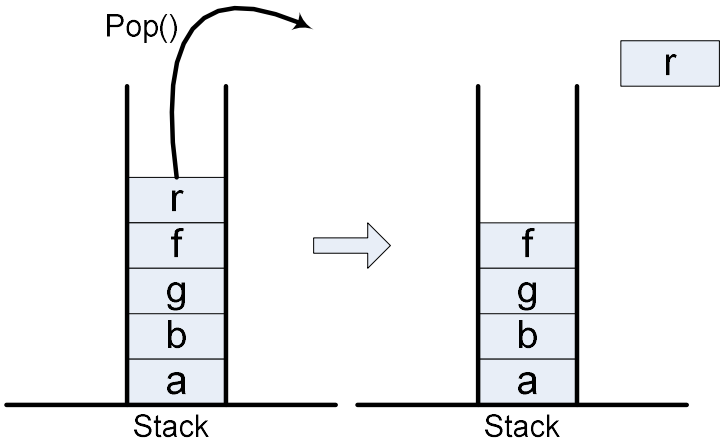
Thao tác này cho phép lấy thông tin phần tử đầu (top) của ngăn xếp nhưng không hủy hay loại bỏ phần tử đó khỏi ngăn xếp. Trong ví dụ, dùng hàm Top() để lấy thông tin phần tử r.



Hình 10. Thao tác lấy thông tin phần tử đầu của ngăn xếp

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

Phần tử bị loại bỏ khỏi ngăn xếp chính là phần tử được đưa vào ở lần gần nhất, và là phần tử ở định (top) của ngăn xếp. Trong ví dụ, phần tử r đã được đưa ra khỏi ngăn xếp sau khi dùng hàm Pop().



Hình 11. Thao tác trích hủy phần tử khỏi ngăn xếp

Ngoài ba thao tác chính ở trên, khi hiện thực kiểu dữ liệu ngăn xếp trên máy tính, chúng ta có thể cài đặt thêm một số phép toán hỗ trợ lập trình khác như: Khởi tạo ngăn xếp, kiểm tra ngăn xếp rỗng, kiểm tra ngăn xếp đầy.

5.1.3. Cài đặt ngăn xếp

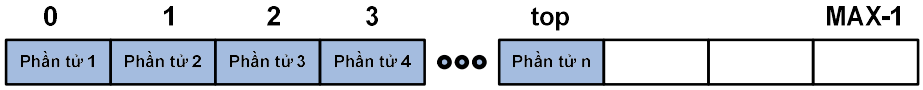
Có nhiều cách khác nhau để hiện thực ngăn xếp trên máy tính. Chúng ta có thể sử dụng kiểu dữ liệu danh sách đặc (LIST), hay danh sách liên kết (LINKEDLIST, DLINKEDLIST) ở chương 4 để cài đặt ngăn xếp. Khi đó, kiểu dữ liệu ngăn xếp chính là kiểu danh sách nhưng chỉ sử dụng một số phép toán cần thiết như: Thêm phần tử vào đầu (AddFirst), hủy phần tử đầu (RemoveFist), hoặc thêm phần tử vào cuối (AddLast), hủy phần tử cuối (RemoveLast).

Trong phần này, chúng ta tìm hiểu cách xây dựng ngăn xếp từ đầu (không sử dụng kiểu danh sách đặc hay danh sách liên kết) bằng kiểu mảng và kiểu con trỏ.

a. Cài đặt ngăn xếp bằng mảng

Để tạo kiểu dữ liệu ngăn xếp, chúng ta cần khai báo một mảng một chiều với kích thước đối đa cho trước (giá trị MAX trong minh họa bên dưới), và một biến **top** để cho biết chỉ số của phần tử ở đỉnh ngăn xếp.

Như chúng ta đã tìm hiểu ở chương 4, khi thao tác trên một mảng, chi phí thêm vào hủy phần tử cuối thấp hơn so với thêm và hủy phần tử đầu vì không phải thực hiện công việc dịch chuyển phần tử. Vì vậy, ở đây chúng ta chọn phần tử cuối mảng là đỉnh đầu của ngăn xếp.



Cấu trúc dữ liệu ngăn xếp được định nghĩa như sau:

#define MAX ***Số_phần tử_tối_đa***

struct **STACK**

{

Kiểu_dữ_liệu_của_phần_tử A[MAX]; //*Mảng các phần tử của ngăn xếp*

int top; //*Chỉ số phần tử đỉnh đầu ngăn xếp*

};

Cụ thể, khai báo ngăn xếp kiểu số nguyên như sau:

```
#define MAX 100 //Ngăn xếp chứa tối đa 100 phần tử
struct STACK
{
        int A[MAX]; //Mảng các phần tử của ngăn xếp
        int top; //Chỉ số phần tử đỉnh đầu ngăn xếp
};
```

Tiếp theo chúng ta xây dựng các thao tác cơ bản trên ngăn xếp.

- **Tạo ngăn xếp rỗng**

Ngăn xếp rỗng là ngăn xếp không có phần tử nào. Khi đó chỉ số top của phần tử đầu phải nhỏ hơn không.

```
void InitStack(STACK& myStack)
{
        myStack.top=-1;
}
```

- **Kiểm tra ngăn xếp rỗng**

Ngăn xếp rỗng khi chỉ số top của phần tử đầu bằng -1.

```
int IsEmptyStack(STACK& myStack)
{
        if(myStack.top==-1)//Ngăn xếp rỗng
                return 1;
        return 0;//Ngăn xếp không rỗng
}
```

- **Kiểm tra ngăn xếp đầy hay không**

Ngăn xếp đầy khi nó lưu trữ đủ số phần tử được cấp phát từ ban đầu. Chúng ta so sánh giá trị top và MAX để biết ngăn xếp đầy hay chưa.

```
int IsFullStack(STACK& myStack)
{
        if(myStack.top==MAX-1)//Ngăn xếp đầy
                return 1;
        return 0;//Ngăn xếp chưa đầy
}
```

- **Thêm một phần tử vào ngăn xếp**

Trước khi thêm một phần tử vào ngăn xếp, cần kiểm tra xem ngăn xếp có đầy hay không. Nếu ngăn xếp đầy thì thao tác thêm sẽ không được thực hiện. Quá trình thêm gồm hai công việc: (1) Gán giá trị thêm vào vị trí top+1 và (2) tăng giá trị top lên 1 đơn vị.

```
void Push(STACK& myStack, int x)
{
        if(IsFullStack(myStack))
                cout<<"\nNgan xep day!";
        else
        {
                myStack.A[myStack.top+1]=x;
                myStack.top++;
        }
}
```

- **Lấy thông tin phần tử ở đỉnh ngăn xếp**

Thao tác này chỉ lấy thông tin ở vị trí top, không loại bỏ phần tử này khỏi ngăn xếp.

```
int Top(STACK myStack)
{
        return myStack.A[myStack.top];
}
```

- **Trích hủy phần tử ở đỉnh ngăn xếp**

Thao tác này lấy thông tin phần tử đầu và loại bỏ nó ra khỏi ngăn xếp. Chúng ta chỉ cần giảm giá trị chỉ số top để loại bỏ phần tử đầu.

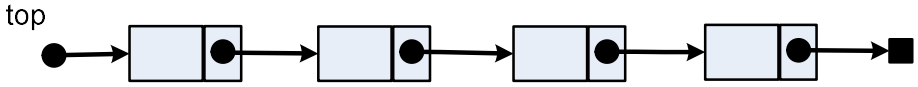
```
int Pop(STACK& myStack)
{
        int t=myStack.A[myStack.top];
        myStack.top--;
        return t;
}
```

❖ **Nhận xét**

Ưu điểm của việc cài đặt ngăn xếp bằng mảng là đơn giản, dễ hiểu, dễ cài đặt. Tuy nhiên, cũng nó cũng có những hạn chế giống như việc sử dụng mảng để cài đặt cho kiểu danh sách (danh sách đặc). Phần tiếp theo, chúng ta tìm hiểu cách xây dựng ngăn xếp bằng kiểu con trỏ.

b. Cài đặt ngăn xếp bằng kiểu con trỏ

Cách cài đặt là xây dựng một danh sách liên kết nhưng chỉ cho phép thao tác ở một đầu danh sách mà thôi. Chúng ta chỉ cần sử dụng một con trỏ **top** để quản lý toàn bộ danh sách (lưu địa chỉ phần tử đầu).



Tương tự phần cài đặt cho danh sách liên kết đơn, một phần tử NODE của ngăn xếp được định nghĩa như sau:

```
struct NODE
{
        Kiểu_dữ_liệu_của_phần_tử Info; //Thông tin của Node
```

```
        struct NODE* Next; //Con trỏ chỉ đến phần tử node tiếp theo
};
```

Cụ thể, định nghĩa kiểu NODE cho ngăn xếp lưu trữ các phần tử kiểu số nguyên.

```
struct NODE
{
        int info;
        struct NODE*next;
};
```

Định nghĩa kiểu ngăn xếp:

```
struct LINKEDSTACK
{
        NODE*top;
};
```

Hoàn toàn tương tự cách cài đặt trên danh sách liên kết đơn, chúng ta xây dựng các thao tác trên ngăn xếp một cách dễ dàng. Dưới đây là phần cài đặt các thao tác chính trên ngăn xếp.

```
//Khởi tạo ngăn xếp
void InitStack(LINKEDSTACK& myStack)
{
        myStack.top=NULL;
}
//Kiểm tra ngăn xếp rỗng
int IsEmptyStack(LINKEDSTACK myStack)
{
        if(myStack.top==NULL)
                return 1;//Ngăn xếp rỗng
        return 0;//Ngăn xếp không rỗng
}
//Tạo phần tử mới
NODE*CreateNode(int x)
{
        NODE*p=new NODE;
```

```
	if(p==NULL)
	{
		cout<<"\nKhong du bo nho";
		return NULL;
	}
	p->info=x;
	p->next=NULL;
	return p;
}
//Thêm 1 phần tử vào ngăn xếp
void Push(LINKEDSTACK& myStack, NODE*p)
{
	p->next=myStack.top;
	myStack.top=p;
}
//Lấy thông tin phần tử đầu ngăn xếp
int Top(LINKEDSTACK myStack)
{
	return myStack.top->info;
}
//Trích hủy phần tử đầu ngăn xếp
int Pop(LINKEDSTACK& myStack)
{
	NODE*p=myStack.top;
	int t=myStack.top->info;
	myStack.top=myStack.top->next;
	delete p;//Giải phóng vùng nhớ
	return t;
}
```

5.1.4. Một số ứng dụng của ngăn xếp

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*.

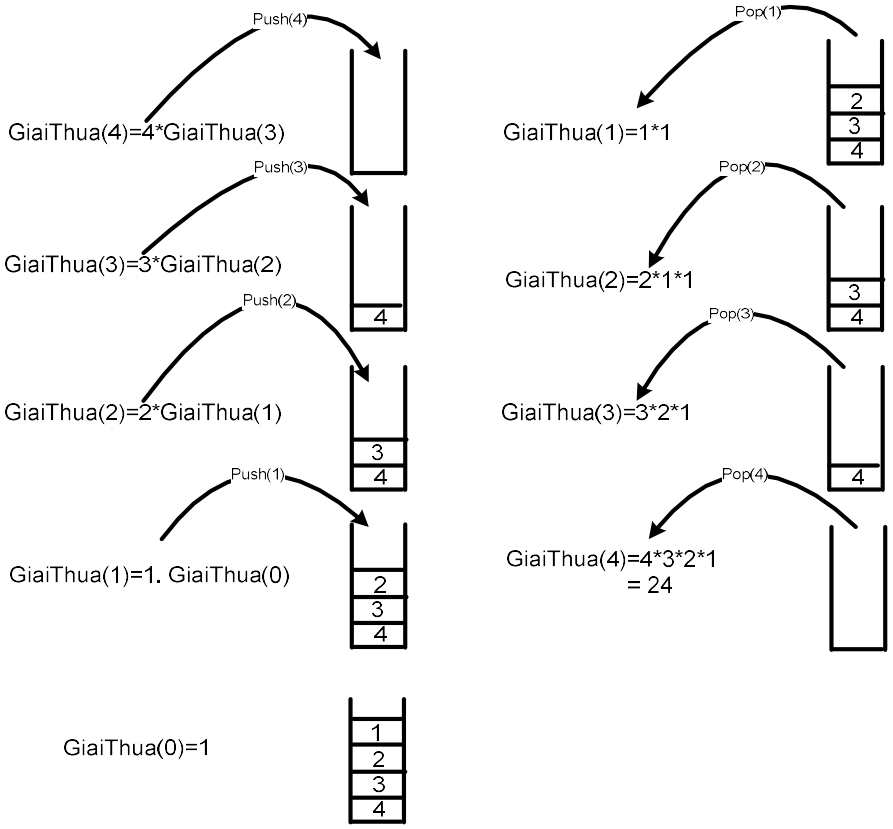
Khi chương trình con đi từ mức i vào mức i+1 thì các biến cục bộ ở mức i và địa chỉ mã lệnh đang dang dở phải được lưu trữ để khi chương trình quay trở lại sẽ sử dụng để thực hiện tiếp.

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. Đây là cách mà các chương trình dịch (trình biên dịch, trình thông dịch) sử dụng để khử để quy cho các chương trình.

Sau đây, chúng ta tìm hiểu nguyên tắc sử dụng ngăn xếp để khử để quy thông qua bài toán tính giai thừa và bài toán tháp Hà Nội.

- **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)!. Hình dưới đây mô phỏng việc lưu trữ các biến cục bộ vào ngăn xếp khi tính 4!.



Hình 12. Sử dụng ngăn xếp cho bài toán tính giai thừa

Để tính GiaiThua(4), vì 4!=4*3!, nên cần phải tính GiaiThua(3) trước. Khi thực hiện tính GiaiThua(3), cần phải lưu trữ giá trị 4 vào ngăn xếp.

Để tính GiaiThua(3), cần tính GiaiThua(2) và lưu giá trị 3 vào ngăn xếp. Công việc tiếp tục như vậy cho đến khi tính GiaiThua(0). Khi tính được GiaiThua(0)=1, tiến hành nhân lần lượt với các giá trị được lấy ra từ hàng đợi.

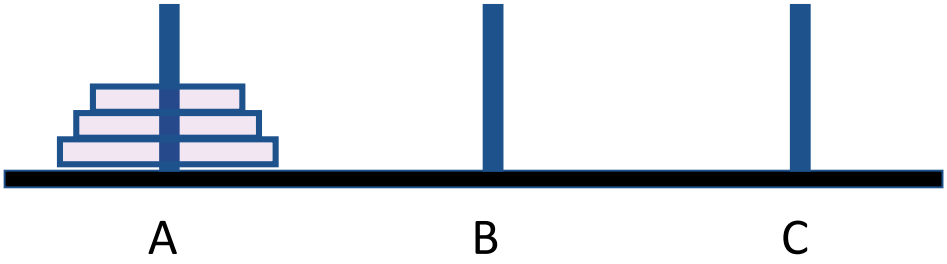
Hàm cài đặt dưới đây minh họa việc sử dụng ngăn xếp cho bài toán tính giai thừa. Trong minh họa này, chúng ta sử dụng kiểu ngăn xếp được cài đặt bằng con trỏ (LINKEDSTACK).

```
#include<iostream.h>
int GiaiThua(int n)
{
	LINKEDSTACK myStack;
	if(n<0)
		return -1;
	InitStack(myStack);//Khởi tạo ngăn xếp
	while(n != 0)
	{
		NODE*p=CreateNode(n);//Tạo 1 node mới
		Push(myStack,p);
		n=n-1;
	}
	int kq=1;//0! = 1
	while(!IsEmptyStack(myStack))
	{
		int temp=Pop(myStack);
		kq=kq*temp;
	}
	return kq;
}
void main()
{
	int n, kq;
	cout<<"Nhap n:";
```

```
        cin>>n;
        kq=GiaiThua(n);
        cout<<kq;
}
-----------------------------------------
Nhập vào: n=4.
Kết quả: 24
```

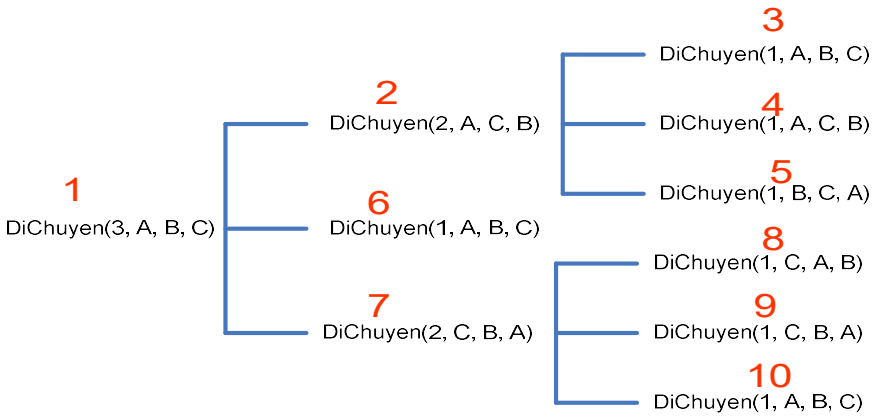
- **Bài toán tháp Hà Nội**

Bài toán tháp Hà Nội đã được trình bày ở chương 2 và được cài đặt đệ quy. Chúng ta xem xét ví dụ trong trường hợp số đĩa di chuyển là 3. Tên các cột được đặt lần lượt là A, B, C.



Hình 13. Bài toán tháp Hà Nội

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.



Hình 14. Thứ tự gọi hàm trong giải thuật đệ quy, bài toán tháp Hà Nội

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 14 thể hiện thứ tự gọi các thủ tục. Như vậy, khi gọi thực các công việc được 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 chúng ta cài đặt giải thuật không đệ quy để giải quyết bài toán Tháp Hà Nội. Chúng 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 DiChuyen không đệ qui
void DiChuyen(ThuTuc X)
{
        ThuTuc Temp, Temp1;
        STACK myStack;
        InitStack(myStack);
        Push(myStack, X);
        do{
                Temp= Pop(myStack);
                if (Temp.N==1)
                        printf("Chuyen 1 dia tu %c sang %c\n", Temp.A,
```

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

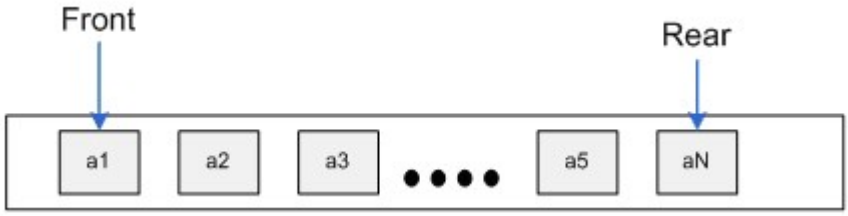
5.2. HÀNG ĐỢI

Hàng đợi là cấu trúc dữ liệu được 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, cách ứng dụng của nó và hiện thực theo các cách khác nhau.

5.3. 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 (gọi là REAR trong giáo trình này), và việc hủy chỉ được thực hiện ở đầu còn lại gọi là đầu hàng đợi (gọi là FRONT trong giáo trình này).

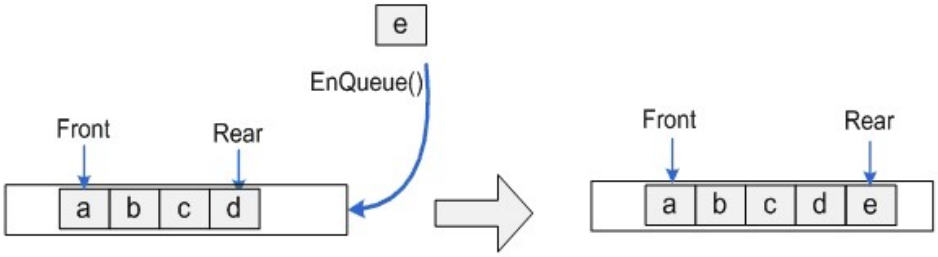
Chúng ta có thể hình dung hình ảnh của hàng đợi thông qua việc xếp hàng chờ mua vé xem phim tại rạp. Người nào vào hàng trước sẽ mua được vé trước và ra khỏi hàng trước. Vì vậy, hàng đợi có tính chất "*vào trước ra trước*" (First in first out – FIFO).



5.3.1. Một số phép toán trên hàng đợi

❖ **EnQueue**

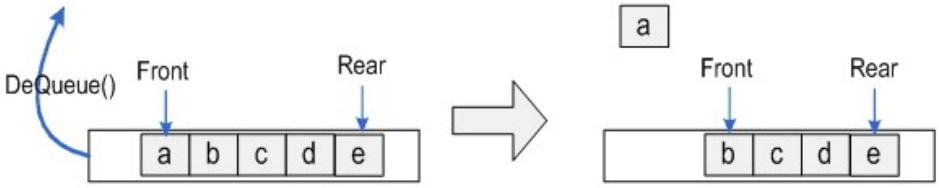
Là thao tác thêm một phần tử vào cuối hàng đợi.



Hình 15. Thêm một phần tử vào hàng đợi

❖ **DeQueue**

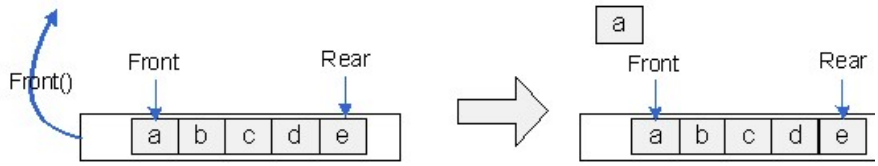
Là thao tác lấy thông tin và hủy phần tử ở đầu hàng đợi



Hình 16. Lấy thông tin và hủy phần tử đầu hàng đợi

❖ **Front**

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



Hình 17. Lấy thông tin phần tử đầu hàng đợi

❖ **IsEmptyQueue**

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

5.3.2. Cài đặt hàng đợi

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

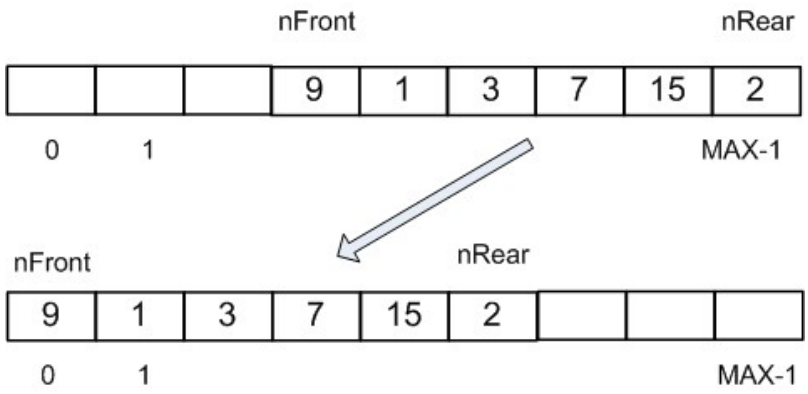
❖ **Cài đặt bằng mảng**

Chúng ta có thể tạo một hàng đợi bằng cách khai báo một mảng một chiều với kích thước tối đa là MAX nào đó. Như vậy, hàng đợi sẽ có thể chứa tối đa MAX phần tử được đánh chỉ số từ 0 đến MAX – 1. 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, chúng 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 NULLVALUE.

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 chúng ta không thể thêm được vào hàng đợi nữa vì nRear=MAX-1. Trường hợp này, người ta gọi là ***hàng đợi bị tràn***. Khi mà toàn bộ hàng đợi đã chứa các phần tử, ***hàng đợi được gọi là bị đầy***.

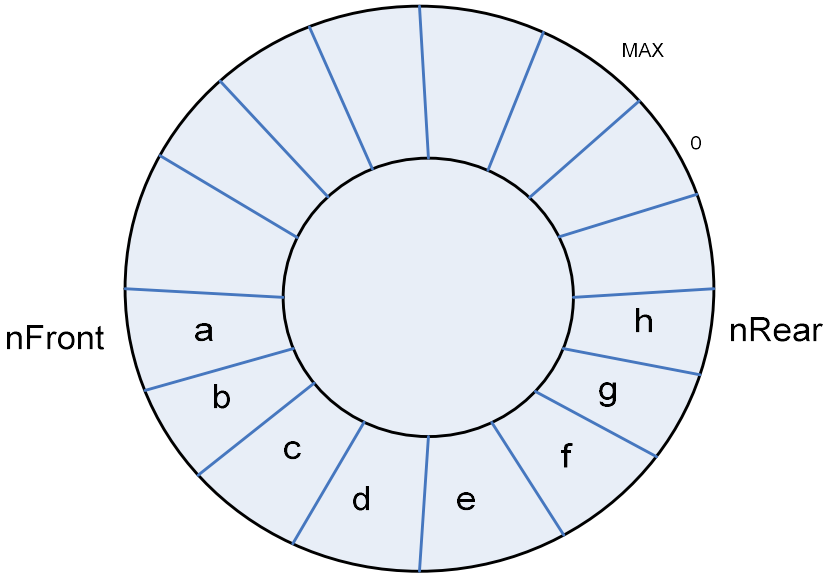
Trong trường hợp hàng đợi bị tràn, có thể giải quyết bằng một trong hai cách sau:

+ Cách 1: Tịnh tiến toàn bộ nội dung lên phía đầu mảng nFront vị trí. Trong trường hợp này nFront phải được quy định luôn luôn nhỏ hơn nRear.



Hình 18. Xử lý hàng đợi tràn bằng tịnh tiến phần tử của mảng

+ Cách 2: Thực hiện quay vòng. Xem như đầu mảng và cuối mảng là hai phần tử kề nhau. Khi vị trí cuối mảng MAX – 1 đã có nội dung lưu trữ, nếu thêm vào một phần tử nữa thì ta thêm vào vị trí 0. Tiếp tục như vậy cho đến khi mảng đầy. Trong trường hợp này nFront cần được quy định có thể lớn hơn nRear.



Hình 19. Hàng đợi vòng

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

#define MAX <**Kích thước tối đa**>

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

ElementType Queue[MAX];

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.

❖ **Khởi tạo hàng đợi rỗng**

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

}
```

❖ **Kiểm tra hàng đợi rỗng**

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

❖ **Kiểm tra hàng đợi đầy**

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

❖ **Thêm một phần tử vào cuối hàng đợi**

```
void EnQueue(ElementType x)
{
        if(!IsFullQueue()) //Kiểm tra hàng đợi đầy
        {
                if(nRear==MAX - 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=MAX – nFront;
                    nFront=0;
                }
                //Lưu nội dung mới
                nRear=nRear+1;
                Queue[nRear]=x;
        }
        else
                cout<<"\nHang doi day!";
}
```

❖ **Lấy thông tin phần tử đầu hàng đợi**

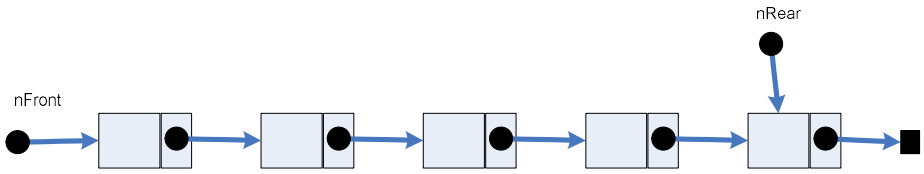
```
ElementType Front()
{
        if(!IsEmptyQueue())
                return Queue[nFront];
        return NULLDATA;
}
```

❖ **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 NULLVALUE;
}
```

5.3.3. 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 chúng ta có thể thêm phần tử vào cuối và hủy phần tử đầu của danh sách một cách dễ dàng và nhanh chóng. Con trỏ Head của danh sách là nFront và con trỏ Tail 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**>

LINKEDLIST 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.

❖ **Tạo hàng đợi rỗng**

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

❖ **Kiểm tra hàng đợi rỗng**

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

❖ **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);
}
```

❖ **Lấy thông tin phần tử ở đầu hàng đợi**

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

❖ **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 NULL;
}
```

5.4. BÀI TẬP CHƯƠNG 5

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 Dequeue) 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 6
CẤU TRÚC CÂY

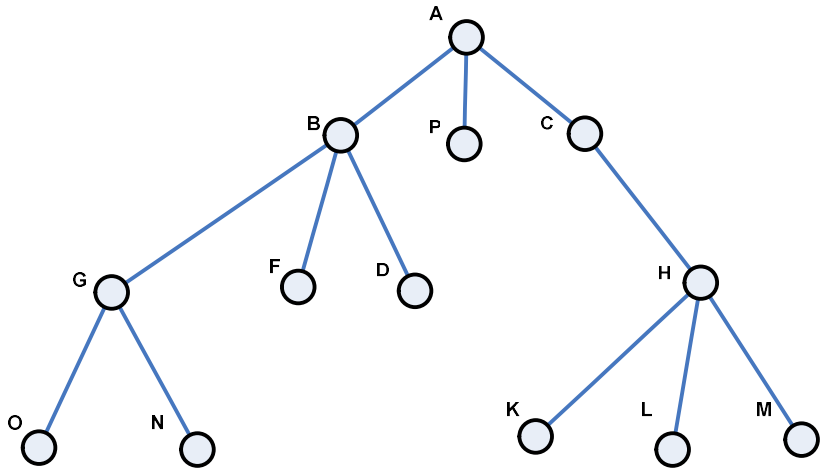
Chương này trình bày những khái niệm và tính chất cơ bản về cấu trúc dữ liệu dạng cây. Nội dung sẽ tập trung vào kiểu dữ liệu cây nhị phân và cây nhị phân tìm kiếm.

6.1. CẤU TRÚC CÂY

6.1.1. 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, $C_{2,}$, ..., 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. Mối quan hệ này gọi là mối quan hệ ***cha – con***. Ngược lại, 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.



Hình 20. Cấu trúc 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 level(A) = 0.

level(B) = level(P) = level(C) = level(A) + 1 = 1.

❖ **Rừng cây**

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

6.1.2. 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.

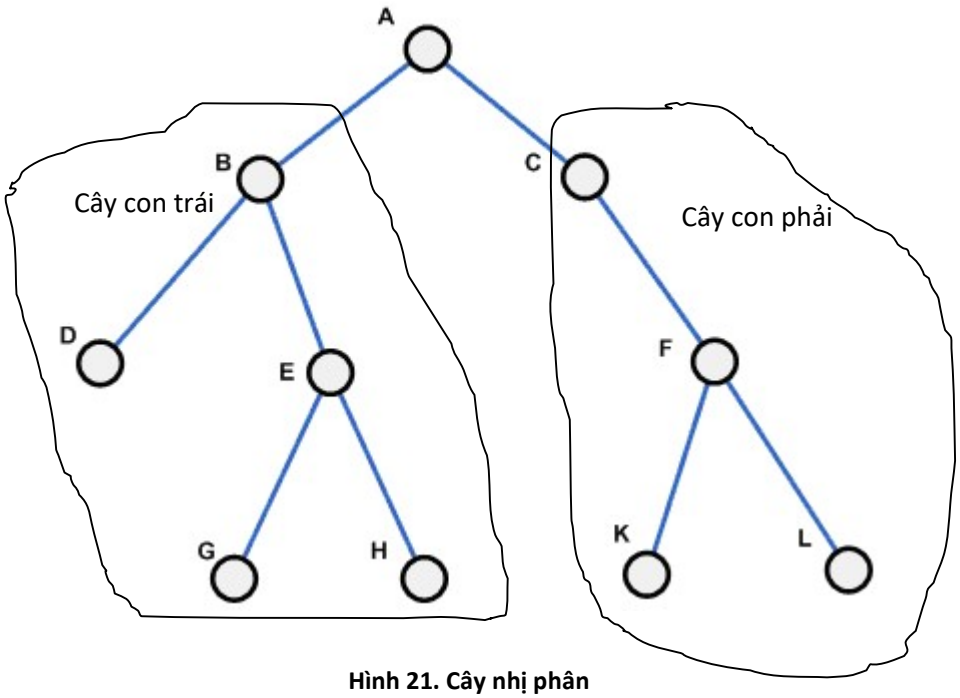
Cây nhiều nhánh cân bằng (Balanced Multiway Tree), gọi tắt B-Cây là một cây nhiều nhánh tìm kiếm thỏa thêm một số tính chất sao cho số nút ở cây con trái và cây con phải không quá chênh lệch nhau.

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.

6.2. CÂY NHỊ PHÂN

6.2.1. Định nghĩa

Cây nhị phân là cây rỗng hoặc là cây mà mỗi nút có tối đa 2 nút con. 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.



Hình 21. Cây nhị phân

a. Duyệt cây nhị phân

Có ba cách duyệt cây nhị phân thông dụng:

❖ **Duyệt tiền 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)**

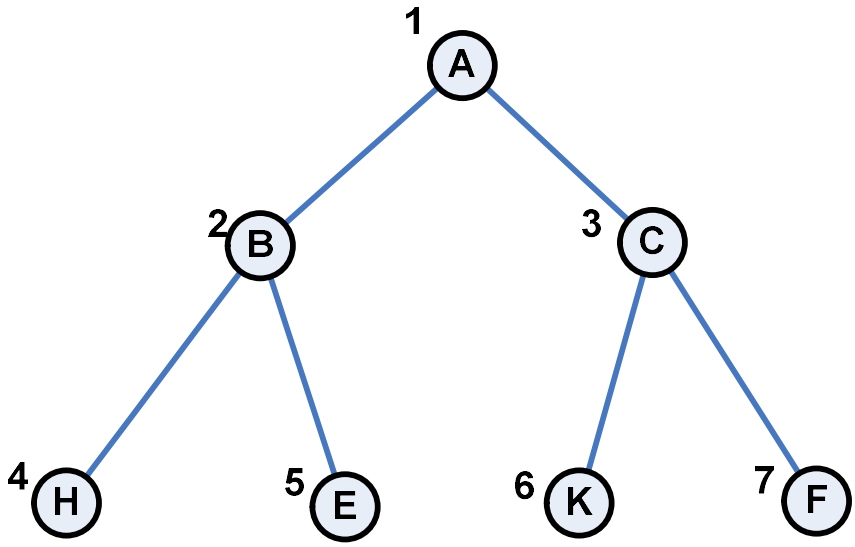
Trước tiên, thăm các nút của cây con trái. Sau đó, thăm các nút của cây con phải, rồi cuối cùng thăm nút gốc.

6.2.2. Cài đặt cây nhị phân

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

❖ **Cài đặt bằng mảng**

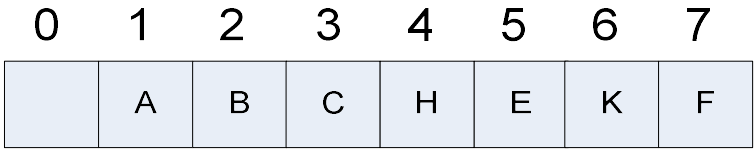
Xét trường hợp có một cây nhị phân đầy đủ, chúng ta có thể đánh số các nút trên cây theo thứ tự từ mức 0 trở đi, hết mức này đến mức khác và từ trái qua phải đối với các nút ở mỗi mức như sau:



Hình 22. Đánh số trên cây nhị phân

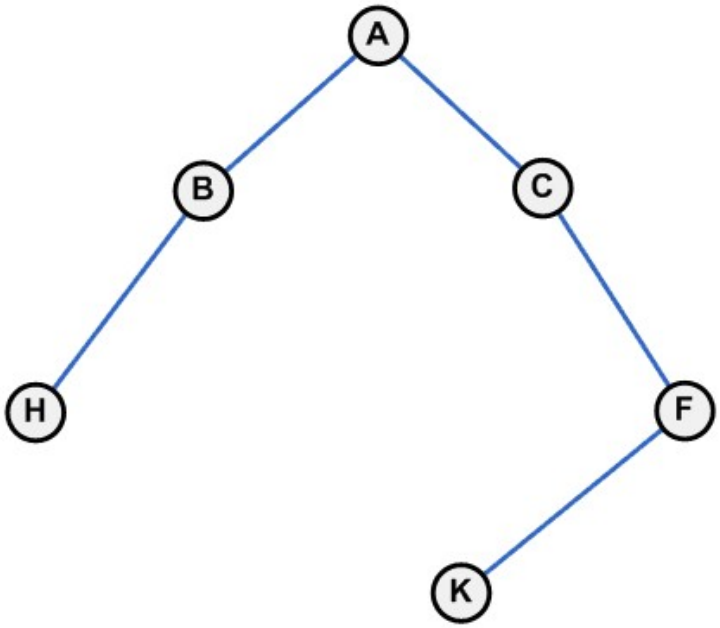
Theo cách đánh số này, nút thứ i có hai con là nút thứ 2*i, và 2*i + 1. Cha của nút thứ j là j/2 (Phép chia lấy phần nguyên). Dựa vào nguyên tắc này, chúng ta có thể lưu trữ cây trên một mảng Tree[]. Nút thứ i được lưu trữ trên phần tử Tree[i]. Đối với cây nhị phân đầy đủ trên, ta có mảng lưu trữ như hình 23. Vì kiểu dữ liệu mảng trong ngôn ngữ lập trình C có

chỉ số bắt đầu từ 0, nên chúng ta không sử dụng phần tử đầu tiên của mảng.



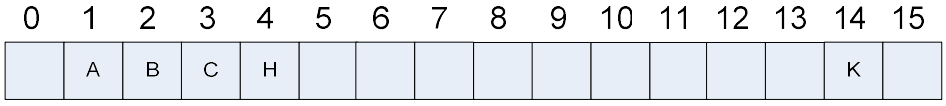
Hình 23. Lưu trữ cây nhị phân trên mảng

Đối với cây nhị phân không đầy đủ, 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 có thể loại trừ chúng ra khi xử lý trên cây. Chúng ta xem ví dụ dưới đây:



Hình 24. Cây nhị phân không đầy đủ

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



Hình 25. Lưu trữ cây nhị phân không đầy đủ trên mảng

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.

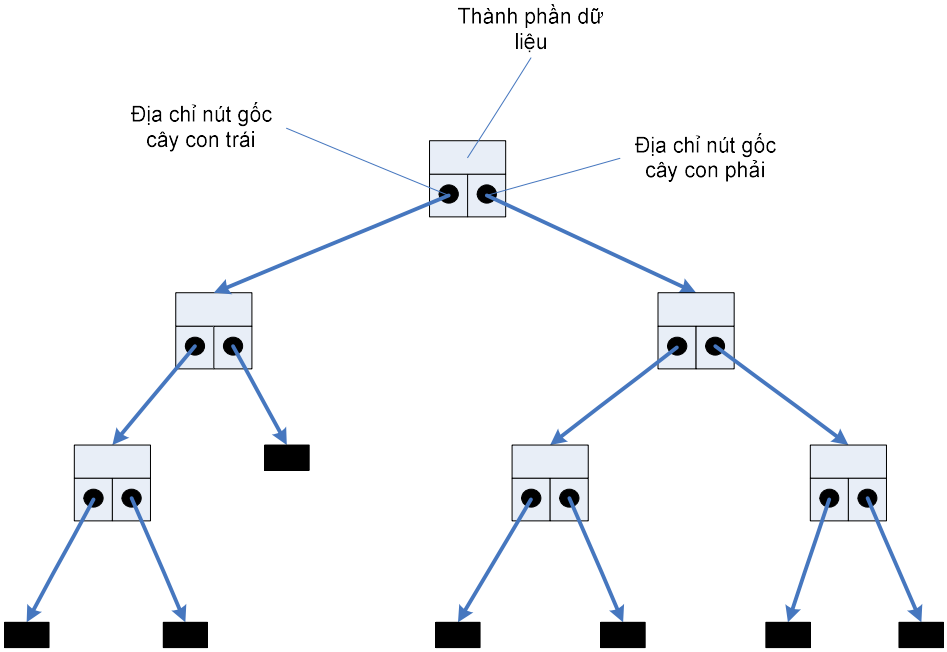
❖ Cài đặt bằng cấu trúc liên kết

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.



Hình 26. Cài đặt cây bằng kiểu con trỏ

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)
        {
                InOrder(root→pLeft);
                //---Xử lý thông tin tại root----//
                InOrder(root→pRight);
        }
}
```

- **Duyệt hậu tự**

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

6.3. CÂY NHỊ PHÂN TÌM KIẾM

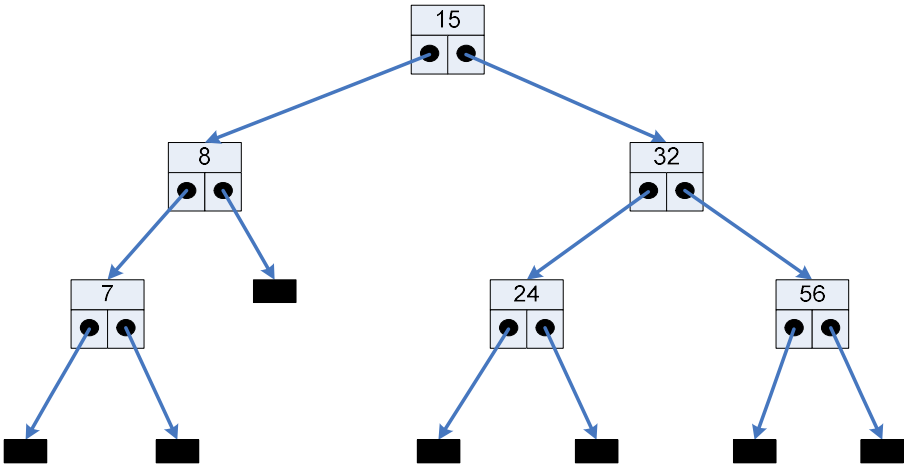
6.3.1. Định nghĩa

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, chúng 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.



Hình 27. Cây nhị phân tìm kiếm

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

a. Thêm một nút vào cây NPTK

```
int InsertNode(TNode* root, ElementType x)
{
        if(root != NULL)
        {
                if(root→key==x)
                        return 0;//Đã tôn 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 NULL;
}
```

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

c. Hủy một nút trên cây

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**

Với trường hợp này, chúng ta chỉ việc giải phóng bộ nhớ cho phần tử này mà thôi.

❖ **Trường hợp x có 1 nút con**

Khi đó, chúng ta thực hiện hai việc là: (1) móc nối nút cha của x với con duy nhất của x, rồi (2) hủy phần tử x.

❖ **Trường hợp x có 2 nút con**

Trường hợp này, chúng 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, chúng 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;
	}
}
```

❖ **Hủy toàn bộ cây NPTK**

Thao tác hủy toàn bộ cây nhị phân tìm kiếm dưới đây được thực hiện thông qua việc duyệt cây theo phương pháp hậu tự. Tức là, chúng 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;
	}
}
```

6.4. 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

Tiếng nước ngoài

[1]. Robert L.Kruse và Alexander J.Ryba. *Data Structure and Program Design in C++*. Prentice-Hall Inc, 2000.

[2]. Ashok N. Kamthane. *Introduction to Data Structures in C*. Pearson Education India, 2007.

[3]. Robert Lafore, *Data Strucutures and Algorithms in Java*. SAMS, 1998.

[4]. Donald Knuth, *The Art of Computer Programming, Volume 1, 2, 3*. Addison-Wesley, 1997.

Tiếng việt

[5]. Trần Hạnh Nhi, *Nhập môn cấu trúc dữ liệu và giải thuật*. Đại học KHTN TP. HCM, 2000.

[6]. Nguyễn Văn Linh, Trần Ngân Bình, *Giáo trình Cấu trúc dữ liệu*. Đại học Cần thơ, 2003.

PHỤ LỤC

Chương trình cài đặt các thao tác trên danh sách liên kết đơn:

```
#include<iostream.h>
//Khai báo cấu trúc một phần tử
struct NODE
{
	int info;
	struct NODE*next;
};
//Khai báo cấu trúc danh sách
struct LINKEDLIST
{
	NODE*Head;
	NODE*Tail;
};
//Khởi tạo danh sách
void InitList(LINKEDLIST& myList)
{
	myList.Head=myList.Tail=NULL;
}
//Kiểm tra danh sách rỗng
int IsEmptyList(LINKEDLIST myList)
{
	if(myList.Head==NULL)
		return 1;//Danh sách rỗng
	return 0;//Danh sách không rỗng
}
//Tạo phần tử mới
NODE*CreateNode(int x)
{
	NODE*p=new NODE;
	if(p==NULL)
```

```
	{
		cout<<"\nKhong du bo nho";
		return NULL;
	}
	p->info=x;
	p->next=NULL;
	return p;
}
//Thêm phần tử vào đầu
void AddFirst(LINKEDLIST& myList, NODE*p)
{
	if(IsEmptyList(myList))//Danh sách rỗng
		myList.Head=myList.Tail=p;
	else//Danh sách không rỗng
	{
		p->next=myList.Head;
		myList.Head=p;
	}
}
//Thêm phần tử vào cuối
void AddLast(LINKEDLIST& myList, NODE*p)
{
	if(IsEmptyList(myList))//Danh sách rỗng
		myList.Head=myList.Tail=p;
	else//Danh sách không rỗng
	{
		myList.Tail->next=p;
		myList.Tail=p;
	}
}
//Thêm phần tử vào sau phần tử q
void AddNode(LINKEDLIST& myList, NODE*q, NODE*p)
{
	p->next=q->next;
```

```
	q->next=p;
	if(myList.Tail==q)
		myList.Tail=p;
}
//Hủy phần tử đầu
void RemoveFirst(LINKEDLIST& myList)
{
	if(IsEmptyList(myList))//Danh sách rỗng
		cout<<"\nDanh sach rong";
	else
	{
		NODE*p=myList.Head;
		if(myList.Head==myList.Tail)//Danh sách có 1 phần tử
			myList.Head=myList.Tail=NULL;
		else//Danh sách có nhiều hơn 1 phần tử
			myList.Head=myList.Head->next;
		delete p;//Giải phóng vùng nhớ
	}
}
//Hủy phần tử cuối
void RemoveLast(LINKEDLIST& myList)
{
	if(IsEmptyList(myList))//Danh sách rỗng
		cout<<"\nDanh sach rong";
	else
	{
		NODE*q=myList.Tail;
		if(myList.Head==myList.Tail)//Danh sách có 1 phần tử
			myList.Head=myList.Tail=NULL;
		else//Danh sách có nhiều hơn 1 một tử
		{
			NODE*p;
			for(p=myList.Head;p->next!=myList.Tail;p=p-
>next);
```

```
                        p->next=NULL;
                        myList.Tail=p;
                }
                delete q;//Giải phóng vùng nhớ
        }
}
//Hủy phần tử sau phần tử q
void RemoveNode(LINKEDLIST& myList, NODE*q)
{
        NODE*p=q->next;
        if(p==NULL)
                cout<<"\nKhong ton tai phan tu sau q";
        else
        {
                q->next=p->next;
                if(p==myList.Tail)//Nếu p là phần tử cuối
                        myList.Tail=q;
                delete p;
        }
}
//Tìm một phần tử có giá trị x trong danh sách
int SearchNode(LINKEDLIST myList, int x)
{
        NODE*p=myList.Head;
        while(p!=NULL)
        {
                if(p->info==x)
                        return 1;//Tìm thấy
                p=p->next;
        }
        return 0;//Không tìm thấy
}
//Sắp xếp danh sách bằng thuật toán Quick sort
void QuickSort(LINKEDLIST& myList)
```

```
{
	LINKEDLIST myList1;
	LINKEDLIST myList2;
	NODE *pivot, *p;
	InitList(myList1);
	InitList(myList2);
	/*Trường hợp danh sách rỗng hoặc có 1 phần tử*/
	if (myList.Head==myList.Tail)
		return;
	/*Phân hoạch danh sách thành 2 danh sách con*/
	pivot = myList.Head;//Phần tử cầm canh
	p=myList.Head->next;
	while (p!=NULL)
	{
		NODE*q = p;
		p=p->next;
		q->next=NULL;
		if (q->info < pivot->info)
			AddLast(myList1, q);//Thêm vào cuối danh sách
1
		else
			AddLast(myList2, q);//Thêm vào cuối danh sách
2
	};
	//Gọi đệ quy sắp xếp cho các danh sách con
	QuickSort(myList1);
	QuickSort(myList2);
	//Ghép nối danh sách 1 + pivot
	if (!IsEmptyList(myList1))
	{
		myList.Head=myList1.Head;
		myList1.Tail->next=pivot;
	}
	else
```

```
                myList.Head=pivot;
        //Ghép nối pivot + danh sách 2
        pivot->next=myList2.Head;
        if (!IsEmptyList(myList2))
                myList.Tail=myList2.Tail;
        else
                myList.Tail=pivot;
}
//Liệt kê nội dung các phần tử
void PrintList(LINKEDLIST myList)
{
        for(NODE*p=myList.Head;p!=NULL;p=p->next)
                cout<<p->info<<"   ";
}
void main()
{
        LINKEDLIST myList;
        InitList(myList);
        NODE*p1=CreateNode(3);
        NODE*p2=CreateNode(4);
        NODE*p3=CreateNode(5);
        NODE*p4=CreateNode(25);
        NODE*p5=CreateNode(15);

        //Thêm các phần tử vào đầu
        AddFirst(myList,p1);
        AddFirst(myList,p2);
        AddFirst(myList,p4);
        cout<<"\nThem phan tu vao dau"<<endl;
        PrintList(myList);
        //Thêm các phần tử vào cuối
        AddLast(myList,p3);
        AddLast(myList,p5);
        cout<<"\nThem phan tu vao cuoi"<<endl;
```

```
        PrintList(myList);
        //Hủy phần tử đầu
        RemoveFirst(myList);
        cout<<"\nHuy phan tu dau"<<endl;
        PrintList(myList);
        //Hủy phần tử cuối
        RemoveLast(myList);
        cout<<"\nHuy phan tu cuoi"<<endl;
        PrintList(myList);
        //Sắp xếp danh sách
        QuickSort(myList);
        cout<<"\nSap xep danh sach"<<endl;
        PrintList(myList);
}
```