BỘ CÔNG THƯƠNG
TRƯỜNG CAO ĐẲNG CÔNG NGHIỆP HUẾ

GIÁO TRÌNH

CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

(Lưu hành nội bộ)

Huế, tháng 08 năm 2009

MỤC LỤC

Trang

LỜI NÓI ĐẦU 1
Chương 1. THUẬT TOÁN VÀ CẤU TRÚC DỮ LIỆU 2
1.1. Thuật toán và cấu trúc dữ liệu 2
1.2. Giải thuật đệ quy 4
1.3. Độ phức tạp của thuật toán 10
Bài tập cuối chương 16
Chương 2. DANH SÁCH 18
2.1. Kiểu dữ liệu trừu tượng danh sách (List Abstract Data Type) 18
2.2. Danh sách đặc (Condensed List) 18
2.3. Danh sách liên kết đơn (Single Linked List) 20
2.3. Danh sách liên kết đơn vòng (Circular Linked List) 28
2.4. Danh sách liên kết kép (Double Linked List) 32
Bài tập cuối chương 38
Chương 3. NGĂN XẾP VÀ HÀNG ĐỢI 40
3.1. Kiểu cấu trúc dữ liệu trừu tượng stack (The Stack Abstract Data Type) . 40
3.2. Kiểu cấu trúc dữ liệu trừu tượng queue (The Queue Abstract Data Type) 44
Bài tập cuối chương 49
Chương 4. CÂY 50
4.1. Một số khái niệm 50
4.2. Cách chứa cây trong bộ nhớ 54
4.3. Cây nhị phân (Binary Tree) 55
4.4. Cây nhị phân tìm kiếm (cây-BST: Binary Search Tree) 59
4.5. Cây cân bằng (Balanced Tree) 64
Bài tập cuối chương 70
Chương 5. ĐỒ THỊ 73
5.1. Các khái niệm cơ bản 73
5.2. Biểu diễn đồ thị trên máy tính 75
5.3. Tìm kiếm trên đồ thị 78
Bài tập cuối chương 80
Chương 6. SẮP XẾP VÀ TÌM KIẾM 81
6.1. Sắp xếp 81
6.2. Tìm kiếm 89
Bài tập cuối chương 90

LỜI NÓI ĐẦU

Cấu trúc dữ liệu và giải thuật luôn đi liền với nhau và là môn học cơ sở cho tất cả những ai nghiên cứu về tin học. Tất cả các chương trình khi chạy trên máy tính đều sử dụng các dạng cấu trúc dữ liệu và các giải thuật. Việc nắm rõ cấu trúc dữ liệu và giải thuật cho phép các lập trình viên, các nhà khoa học máy tính có nhiều lựa chọn trong việc đưa ra các giải pháp để giải quyết bài toán.

Giáo trình này được xây dựng dựa trên những kinh nghiệm giảng dạy và nghiên cứu của tác giả, cùng với sự tham khảo tài liệu của các bạn bè, đồng nghiệp, các tác giả trong và ngoài nước,… nhằm mục đích phục vụ cho việc nghiên cứu và học tập của các sinh viên Đại học, Cao đẳng, Trung cấp chuyên nghiệp ngành Công nghệ Thông tin. Giáo trình được chia làm 6 chương:

Chương 1: Thuật toán và Cấu trúc dữ liệu

Giới thiệu sơ lược về thuật toán và các dạng cấu trúc dữ liệu; trình bày giải thuật đệ quy và cách phân tích thuật toán.

Chương 2: Danh sách

Trình bày các loại danh sách đặc, danh sách liên kết đơn, danh sách liên kết vòng, danh sách liên kết kép; cách biểu diễn và các phép toán thực hiện trên danh sách.

Chương 3: Ngăn xếp & Hàng đợi

Giới thiệu ngăn xếp và hàng đợi; cách biểu diễn và các phép toán thường gặp trên ngăn xếp và hàng đợi.

Chương 4: Cây

Mô tả cây, cây nhị phân, cây BST, cây cân bằng; cách biểu diễn và các phép toán thường gặp trên cây nhị phân, cây BST, cây cân bằng.

Chương 5: Đồ thị

Giới thiệu đồ thị vô hướng, đồ thị có hướng; trình bày cách biểu đồ thị trên máy tính, các phép toán cơ bản trên đồ thị.

Chương 6: Sắp xếp và Tìm kiếm

Trình bày các thuật toán sắp xếp và tìm kiếm cơ bản và đánh giá độ phức tạp của từng thuật toán.

Dù cố gắng rất nhiều nhưng không thể tránh khỏi thiếu sót. Do vậy, rất mong nhận được sự góp ý từ phía các đồng nghiệp và bạn đọc. Các ý kiến xin gửi về: khoa Công nghệ Thông tin, trường Cao đẳng Công nghiệp Huế, 70 Nguyễn Huệ, TP.Huế.

Huỳnh Bảo Quốc Dũng

Chương 1. THUẬT TOÁN VÀ CẤU TRÚC DỮ LIỆU

1.1. Thuật toán và cấu trúc dữ liệu

1.1.1. Thuật toán (algorithm)

1.1.1.1. Định nghĩa thuật toán

Thuật toán là một dãy hữu hạn các bước, mỗi bước mô tả chính xác các phép toán hoặc hành động cần thực hiện để giải quyết vấn đề đặt ra.

1.1.1.2. Đặc trưng của thuật toán

Thuật toán có các đặc trưng sau:

i. Dữ liệu đầu vào (input data): Mỗi thuật toán cần có một số (có thể bằng 0) dữ liệu vào (input). Đó là các giá trị cần nhập vào khi thuật toán bắt đầu làm việc. Các dữ liệu cần được lấy từ các tập giá trị cụ thể nào đó.

ii. Dữ liệu đầu ra (output data): Mỗi thuật toán có thể có một hoặc nhiều dữ liệu ra (output). Đó là các giá trị có quan hệ hoàn toàn xác định với các dữ liệu đầu vào và là kết quả của việc thực hiện thuật toán.

iii. Tính xác định (defineteness): Mỗi bước của thuật toán cần phải được xác định rõ ràng và phải được thực hiện một cách chính xác và nhất quán. Để đảm bảo được tính xác định, thuật toán cần phải được mô tả thông qua một ngôn ngữ lập trình cụ thể. Trong ngôn ngữ này, các mệnh đề được tạo thành theo qui tắc, cú pháp nghiêm ngặt và chỉ có một ý nghĩa duy nhất.

iv. Tính hữu hạn (finiteness): Thuật toán phải kết thúc sau một số hữu hạn bước.

v. Tính hiệu quả (effectiveness): Các bước trong thuật toán phải được thực hiện trong một lượng thời gian hữu hạn.

1.1.2. Cấu trúc dữ liệu (data structure)

1.1.2.1. Khái niệm

Dữ liệu làm việc trong một chương trình thường là dữ liệu có cấu trúc hay là còn gọi là cấu trúc dữ liệu.

Việc chọn đúng loại dữ liệu giúp ta giải bài toán đơn giản và hiệu quả, ngược lại bài toán sẽ trở nên nặng nề và kém hiệu quả.

1.1.2.2. Các loại cấu trúc dữ liệu

Cấu trúc dữ liệu gồm các loại: mảng, bản ghi, tập hợp, con trỏ, ngăn xếp, hàng đợi, cây, đồ thị, file, …. Những loại dữ liệu có cấu trúc thường có sẵn trong các ngôn ngữ lập trình.

Khi làm việc cấu trúc dữ liệu, ta phải chú ý đến: từ khóa khai báo loại cấu trúc dữ liệu, các phép toán, các hàm phục vụ cho cấu trúc dữ liệu đó.

1.1.3. Ngôn ngữ diễn đạt thuật toán

Có nhiều phương pháp biểu diễn thuật toán, có thể biểu diễn thuật toán bằng danh sách các bước thông qua ngôn ngữ tự nhiên và các ký hiệu toán học, có thể biểu diễn bằng sơ đồ khối. Tuy nhiên, như đã trình bày, để đảm bảo tính xác định của thuật toán, thuật toán cần được viết bằng ngôn ngữ lập trình cụ thể.

1.1.3.1. Sử dụng ngôn ngữ tự nhiên

- ***Ví dụ***: Mô tả thuật toán tìm ước số chung lớn nhất của hai số nguyên.

  Đầu vào: Hai số nguyên a, b

  Đầu ra: Ước số chung lớn nhất của a và b

  Thuật toán:

  Bước 1: Nếu a=b thì USCLN(a,b)=a.

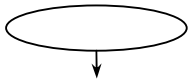
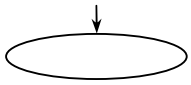
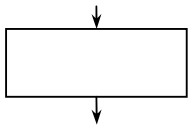
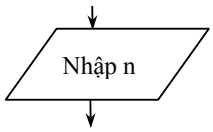
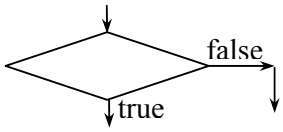
  Bước 2: Nếu a>b thì tìm USCLN của a-b và b, rồi quay lại bước 1.

  Bước 3: Nếu a<b thì tìm USCLN của b-a và a, rồi quay lại bước 1.

1.1.3.2. Sử dụng sơ đồ khối

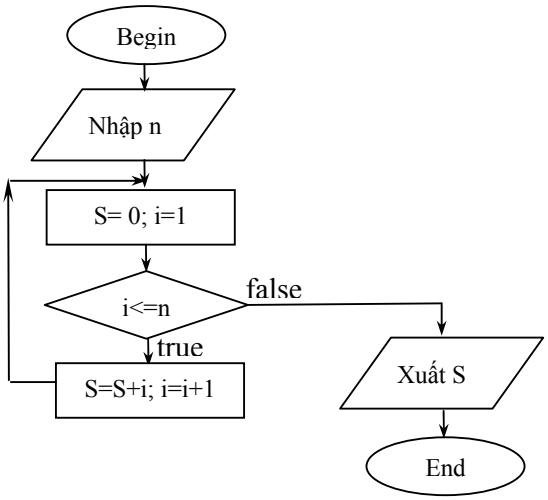
Sử dụng các kí hiệu hình khối cơ bản để tạo thành một mô tả mang tính hình thức (phương pháp này rõ ràng hơn với việc mô tả các bước thực hiện thuật toán)

Sau đây là một số hình khối thường được sử dụng

Hình khối	**Chức năng**
	Khối bắt đầu
	Khối kết thúc
	Khối tính toán
	Khối nhập/xuất dữ liệu
	Khối so sánh

Hình 1.1. *Hình khối*

- ***Ví dụ:*** Giải thuật toán tính S = 1 + 2 + … + n (với n nhập từ bàn phím) bằng sơ đồ khối



Hình 1.2. *Giải thuật tính S=1+2+…+n*

1.1.3.3. Sử dụng ngôn ngữ giả (pseudo code)

Ngôn ngữ giả hoặc mã giả là sự vay mượn các cú pháp của một ngôn ngữ lập trình nào đó để thể hiện thuật toán. Tuy nhiên, trong mã giả ta vẫn dùng một phần ngôn ngữ tự nhiên.

- ***Ví dụ:*** Đoạn mã giả thực hiện thuật toán giải phương trình bậc hai.

```
Nhập: a, b, c
Delta = a*a – 4*a*c
if(Delta>0)
 {
    x1 = (-b – sqrt(Delta))/(2*a)
    x2 = (-b + sqrt(Delta))/(2*a)
    xuất kết quả: Phương trình có hai nghiệm là x1 và x2
 }
else
 if (Delta == 0)
    Xuất kết quả: Phương trình có nghiệm kép là –b/(2*a)
 else {trường hợp Delta<0}
    Xuất kết quả: phương trình vô nghiệm
```

1.2. Giải thuật đệ quy

1.2.1. Khái niệm về đệ quy

Nếu lời giải của của một bài toán T được giải bằng lời giải của một bài toán T1, có dạng giống như T, thì lời giải đó được gọi là lời giải đệ quy. Giải thuật tương ứng với lời giải đệ quy gọi là giải thuật đệ quy.

Ở đây T1 có dạng giống T nhưng theo một nghĩa nào đó T1 phải "nhỏ" hơn T.

Chẳng hạn với bài toán tính n! thì n! là bài toán T còn (n-1)! là bài toán T1 ta thấy T1 cùng dạng với T nhưng nhỏ hơn (n-1 < n).

1.2.2. Giải thuật và thủ tục đệ quy

Giải thuật đệ quy thường được biểu diễn thông qua chương trình con trong ngôn ngữ lập trình, hay còn được gọi là thủ tục đệ quy. Thủ tục đệ quy có các đặc điểm cơ bản sau:

i. Trong thủ tục đệ quy có lời gọi đến chính thủ tục đó.

ii. Sau mỗi lần có lời gọi đệ quy thì kích thước của bài toán được thu nhỏ (hoặc phóng lớn) hơn trước.

iii. Thủ tục đệ quy phải có tính dừng. Nếu không thỏa mãn đặc điểm này thì bài toán đệ quy sẽ gây hiện tượng treo máy.

Một số ngôn ngữ cấp cao như: Pascal, C,... cho phép viết các thủ tục đệ quy. Nếu thủ tục đệ quy chứa lời gọi đến chính nó thì gọi là đệ quy trực tiếp. Cũng có trường hợp thủ chứa lời gọi đến thủ tục khác mà ở thủ tục này lại chứa lời gọi đến nó, trường hợp này gọi là đệ quy gián tiếp.

1.2.3. Thiết kế giải thuật đệ quy

Khi bài toán đang xét hoặc dữ liệu đang xử lý được định nghĩa dưới dạng đệ quy thì việc thiết kế các giải thuật đệ quy tỏ ra rất thuận lợi. Hầu như nó phản ánh rất sát nội dung của định nghĩa đó.

Ta xét một số bài toán sau:

i. Hàm tính n!

Hàm này được định nghĩa như sau:

$$Factorial(n) = \begin{cases} 1 & ; \text{if } n = 0 \\ n * \text{Factorial}(n-1); & \text{if } n > 0 \end{cases}$$

Giải thuật đệ quy được viết dưới dạng hàm dưới đây:

```
long Factorial(int n)
{
     if(n==0)Factorial:=1;
     else Factorial = n*Factorial(n-1);
}
```

Trong hàm trên lời gọi lại nó nằm ở câu lệnh gán sau else.

Mỗi lần gọi đệ quy đến Factorial, thì giá trị của n giảm đi 1. Ví dụ, Factorial(4) gọi đến Factorial(3), gọi đến Factorial(2), gọi đến Factorial(1), gọi đến Factorial(0) đây là trường hợp suy biến, nó được tính theo cách đặc biệt Factorial(0) = 1.

ii. Bài toán dãy số FIBONACCI

Dãy số Fibonacci bắt nguồn từ bài toán cổ về việc sinh sản của các cặp thỏ. Bài toán được đặt ra như sau:

- Các con thỏ không bao giờ chết.
- Hai tháng sau khi ra đời một cặp thỏ mới sẽ sinh ra một cặp thỏ con.
- Khi đã sinh con rồi thì cứ mỗi tháng tiếp theo chúng lại sinh được một cặp con mới.

Giả sử bắt đầu từ một cặp thỏ mới ra đời thì đến tháng thứ n sẽ có bao nhiêu cặp?

Ví dụ với n = 6, ta thấy.
Tháng thứ 1: 1 cặp (cặp ban đầu)
Tháng thứ 2: 1 cặp (cặp ban đầu vẫn chưa đẻ)
Tháng thứ 3: 2 cặp (đã có thêm 1 cặp con)
Tháng thứ 4: 3 cặp (cặp đầu vẫn đẻ thêm)
Tháng thứ 5: 5 cặp (cặp con bắt đầu đẻ)
Tháng thứ 6: 8 cặp (cặp con vẫn đẻ tiếp)

Đặt F(n) là số cặp thỏ ở tháng thứ n. Ta thấy chỉ những cặp thỏ đã có ở tháng thứ n-2 mới sinh con ở tháng thứ n do đó số cặp thỏ ở tháng thứ n là:

F(n) = f(n-2) + F(n-1) vì vậy F(n) có thể được tính như sau:

$$F(n) = \begin{cases} 1 & ; \text{if } n \le 2 \\ F(n-2) + F(n-1) & ; otherwise \end{cases}$$

Dãy số thể hiện F(n) ứng với các giá trị của n = 1, 2, 3, 4..., có dạng

1 1 2 3 5 8 13 21 34 55.... dãy này được gọi là dãy số Fibonacci. Đây là mô hình của rất nhiều hiện tượng tự nhiên và cũng được sử dụng nhiều trong tin học.

Sau đây là thủ tục đệ quy thể hiện giải thuật tính F(n).

```
long F(int n)
{
      if(n<=2) F=1;
      else F = F(n-2) + F(n-1);
}
```

Ở đây trường hợp suy biến ứng với 2 giá trị F(1) = 1 và F(2) = 1.

iii. Chú ý

Đối với hai bài toán nêu trên thì việc thiết kế các giải thuật đệ quy tương ứng khá thuận lợi vì cả hai đều thuộc dạng tính giá trị hàm mà định nghĩa của nó xác định được dễ dàng.

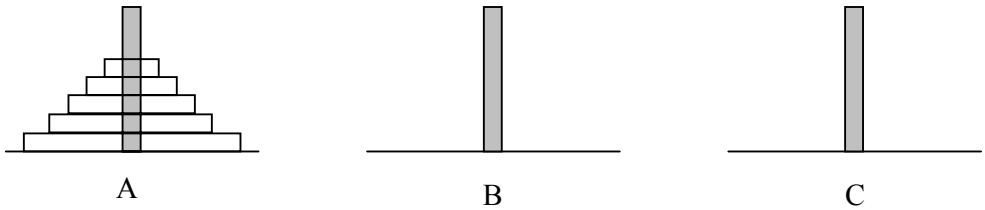
Nhưng không phải lúc nào tính đệ quy trong cách giải bài toán cũng thể hiện rõ nét và đơn giản như vậy. Việc thiết kế một giải thuật đệ quy đòi hỏi phải giải đáp được các câu hỏi sau:

- Có thể định nghĩa được bài toán dưới dạng một bài toán cùng loại, nhưng nhỏ hơn như thế nào?
- Như thế nào là kích thước của bài toán được giảm đi ở mỗi lần gọi đệ quy?
- Trường hợp đặc biệt nào của bài toán được gọi là trường hợp suy biến?

Sau đây ta xét thêm bài toán phức tạp hơn.

iv. Bài toán "Tháp Hà Nội"

Có n đĩa, kích thước nhỏ dần, mỗi đĩa có lỗ ở giữa. Có thể xếp chồng chúng lên nhau xuyên qua một cọc, đĩa to ở dưới, đĩa nhỏ ở trên để cuối cùng có một chồng đĩa dạng như hình tháp như hình 1.3. dưới đây.



Hình 1.3. *Bài toán tháp Hà Nội*

Yêu cầu đặt ra là:

Chuyển chồng đĩa từ cọc A sang cọc khác, chẳng hạn cọc C, theo những điều kiện:

- Mỗi lần chỉ được chuyển một đĩa.
- Không khi nào có tình huống đĩa to ở trên đĩa nhỏ (dù là tạm thời).
- Được phép sử dụng một cọc trung gian, chẳng hạn cọc B để đặt tạm đĩa (gọi là cọc trung gian).

Để đi tới cách giải tổng quát, trước hết ta xét vài trường hợp đơn giản.

* Trường hợp có 1 đĩa:

- Chuyển đĩa từ cọc A sang cọc C.

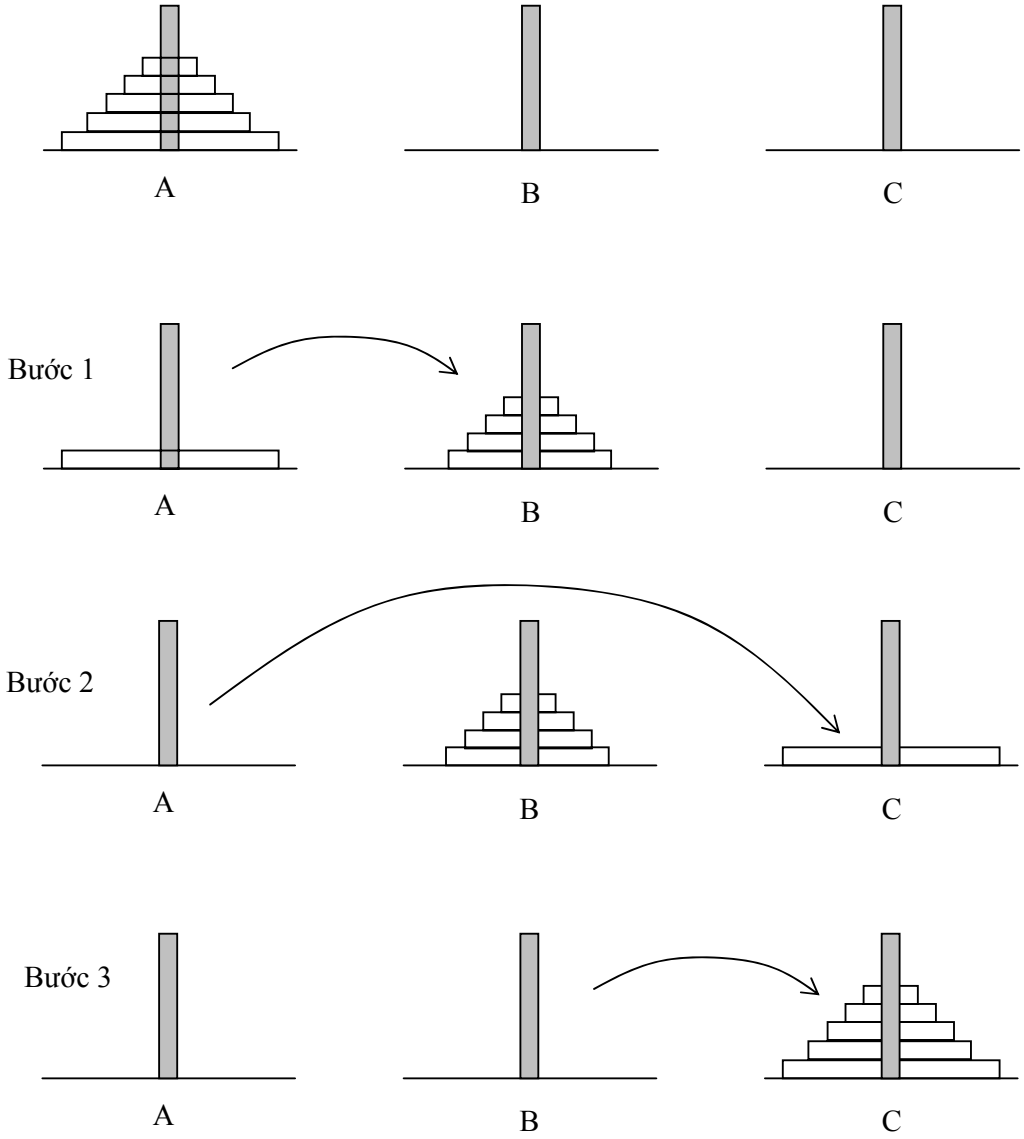
* Trường hợp 2 đĩa:

- Chuyển đĩa thứ nhất từ cọc A sang cọc B.
- Chuyển đĩa thứ hai từ cọc A sang cọc C.
- Chuyển đĩa thứ nhất từ cọc B sang cọc C.

Ta thấy với trường hợp n đĩa (n>2) nếu coi n-1 đĩa ở trên, đóng vai trò như đĩa thứ nhất thì có thể xử lý giống như trường hợp 2 đĩa được, nghĩa là:

- Chuyển n-1 đĩa trên từ A sang B.
- Chuyển đĩa thứ n từ A sang C.
- Chuyển n-1 đĩa từ B sang C.

Lược đồ thể hiện 3 bước này như sau:



Hình 1.4. *Các bước di chuyển đĩa trong bài toán tháp Hà Nội*

Như vậy, bài toán "Tháp Hà Nội" tổng quát với n đĩa đã được dẫn đến bài toán tương tự với kích thước nhỏ hơn, chẳng hạn từ chỗ chuyển n đĩa từ cọc A sang cọc C nay là chuyển n-1 đĩa từ cọc A sang cọc B và ở mức này thì giải thuật lại là:

- Chuyển n-2 đĩa từ cọc A sang cọc C.
- Chuyển 1 đĩa từ cọc A sang cọc B.
- Chuyển n-2 đĩa từ cọc C sang cọc B.

và cứ như thế cho tới khi trường hợp suy biến xảy ra, đó là trường hợp ứng với bài toán chuyển 1 đĩa.

Vậy thì các đặc điểm của đệ quy trong giải thuật đã được xác định và ta có thể viết giải thuật đệ quy của bài toán "Tháp Hà Nội" như sau:

```
void Chuyen(n, A, B, C)
{    if( n==1) chuyển đĩa từ A sang C
     else
     {
          call Chuyen(n-1, A, C, B);
          call Chuyen(1, A, B, C);
          call Chuyen(n-1, B, A, C)    ;
     }
}
```

1.2.4. Hiệu lực của đệ quy

Qua các ví dụ trên ta có thể thấy: đệ quy là một công cụ để giải quyết các bài toán. Bên cạnh giải thuật đệ quy vẫn có những giải thuật lặp khá đơn giản và hữu hiệu. Chẳng hạn giải thuật lặp tính n! có thể viết:

```
long Factorial(int n)
{
     long gt;
      if (n==0 || n==1) gt=1;
     else
     {    gt=1;
          for(int  i=2; i<=n; i++)
               gt = gt*i;
     }
     return gt;
}
```

Hoặc ta xét giải thuật lặp tính số Fibonacci thứ n:

```
long Fibonacci(int n)
{
     long Fibn;
     if(n<=2) Fibn = 1;
     else
     {
          long Fib1 = 1, Fib2 = 1;
          for(int i=3; i<=n; i++)
          {
               Fibn = Fib1 + Fib2;
               Fib1 = Fib2;
               Fib2 = Fibn;
          }
          return Fibn;
     }
}
```

Tuy vậy, đệ quy vẫn có vai trò xứng đáng của nó. Có những bài toán, việc nghĩ ra giải thuật đệ quy thuận lợi hơn nhiều so với giải thuật lặp và có những

giải thuật đệ quy thực sự có hiệu lực cao, chẳng hạn giải thuật sắp xếp kiểu phân đoạn (Quick Sort) hoặc các giải thuật duyệt cây nhị phân mà ta sẽ có dịp xét tới trong môn học này.

Một điều nữa cần nói thêm là: về mặt định nghĩa, công cụ đệ quy đã cho phép xác định một tập vô hạn các đối tượng bằng một phát biểu hữu hạn. Ta sẽ thấy vai trò của công cụ này trong định nghĩa văn phạm, định nghĩa cú pháp ngôn ngữ, định nghĩa một số cấu trúc dữ liệu v.v...

Chú thích: khi thay các giải thuật đệ quy bằng các giải thuật lặp tương ứng ta gọi là khử đệ quy. Tuy nhiên có những bài toán việc khử đệ quy tương đối đơn giản (ví dụ: giải thuật tính n!, tính số Fibonacci ...), nhưng có những bài toán việc khử đệ quy là rất phức tạp (ví dụ: bài toán tháp Hà Nội, giải thuật sắp xếp phân đoạn,...).

1.3. Độ phức tạp của thuật toán

1.3.1. Phân tích thuật toán

Với một bài toán, không chỉ có một thuật toán để giải. Câu hỏi đặt ra là, chúng ta cần chọn thuật toán nào trong số các thuật toán đã có để giải bài toán một cách hiệu quả nhất. Sau đây ta phân tích thuật toán và đánh giá độ phức tạp tính toán của nó.

1.3.1.1. Tính hiệu quả của thuật toán

Khi giải một bài toán, ta cần chọn một thuật toán được xem là tốt nhất trong tất cả các thuật toán đưa ra để giải quyết bài toán. Vậy lựa chọn thuật toán dựa trên cơ sở nào? Thông thường ta dựa trên hai tiêu chuẩn sau đây:

1. Thuật toán đơn giản, dễ hiểu, dễ cài đặt (dễ viết chương trình)

2. Thuật toán sử dụng tiết kiệm nhất nguồn tài nguyên của máy tính, và đặc biệt, chạy nhanh nhất có thể được.

Tiêu chuẩn (2) được xem là tính hiệu quả của thuật toán. Tính hiệu quả của thuật toán bao gồm hai nhân tố cơ bản:

i. Dung lượng không gian nhớ cần thiết để lưu giữ các dữ liệu vào, các kết quả tính toán trung gian và các kết quả của thuật toán.

ii. Thời gian cần thiết để thực hiện thuật toán (ta gọi là thời gian chạy chương trình, thời gian này không phụ thuộc vào các yếu tố vật lý của máy tính (tốc độ xử lý của máy tính, ngôn ngữ viết chương trình,...))

Ta sẽ chỉ quan tâm đến thời gian thực hiện thuật toán. Vì vậy khi nói đến đánh giá độ phức tạp của thuật toán, có nghĩa là ta nói đến đánh giá thời gian thực hiện. Một thuật toán có hiệu quả được xem là thuật toán có thời gian chạy ít hơn các thuật toán khác.

1.3.3.2. Đánh giá thời gian thực hiện của thuật toán

Có hai cách tiếp cận để đánh giá thời gian thực hiện của một thuật toán

Phương pháp thử nghiệm: Viết chương trình và cho chạy chương trình với các dữ liệu vào khác nhau trên một máy tính nào đó. Thời gian chạy chương trình phụ thuộc vào các nhân tố sau đây:

1. Các dữ liệu vào

2. Chương trình dịch để chuyển chương trình nguồn thành chương trình mã máy.

3. Tốc độ thực hiện các phép toán trên máy tính được sử dụng để chạy chương trình.

Vì thời gian chạy chương trình phụ thuộc vào nhiều nhân tố, nên ta không thể biểu diễn chính xác thời gian chạy của chương trình là bao nhiêu đơn vị thời gian (theo thời gian chuẩn), chẳng hạn là bao nhiêu giây?

Phương pháp lý thuyết: Xem thời gian thực hiện của thuật toán như là hàm số của kích thước dữ liệu đầu vào. Kích thước của dữ liệu đầu vào là một tham số đặc trưng, nó có ảnh hưởng quyết định đến thời gian thực hiện chương trình. Cái được chọn làm kích thước của dữ liệu đầu vào phụ thuộc vào thuật toán cụ thể. Chẳng hạn, đối với các thuật toán sắp xếp mảng, thì kích thước của dữ liệu đầu vào là số thành phần của mảng; đối với thuật toán giải hệ n phương trình tuyến tính với n ẩn, ta chọn n là kích thước. Thông thường dữ liệu đầu vào là một số nguyên dương n. Ta sẽ sử dụng hàm số T(n), trong đó n là kích thước dữ liệu đầu vào, để biểu diễn thời gian thực hiện của một thuật toán.

Thời gian thực hiện T(n) là số phép toán sơ cấp cần phải tiến hành khi thực hiện thuật toán. Các phép toán sơ cấp là các phép toán mà thời gian thực hiện bị chặn trên bởi một hằng số chỉ phụ thuộc vào cách cài đặt được sử dụng. Chẳng hạn các phép toán số học +, -, *, /, các phép toán so sánh =, >, <, >=, <=, ... là các phép toán sơ cấp.

1.3.2. Độ phức tạp tính toán của giải thuật

Khi đánh giá thời gian thực hiện bằng phương pháp toán học, ta sẽ bỏ qua nhân tố phụ thuộc vào cách cài đặt, mà chỉ tập trung vào xác định độ lớn của thời gian thực hiện T(n). Ký hiệu toán học O (đọc là ô lớn) được sử dụng để mô tả độ lớn của hàm T(n).

1.3.2.1. Định nghĩa

Giả sử n là số nguyên không âm, T(n) và f(n) là các hàm thực không âm. Ta viết T(n) = O(f(n)) (đọc : T(n) bằng ô lớn của f(n)), nếu và chỉ nếu tồn tại các hằng số dương c và n_o sao cho $T(n) \le c.f(n)$, với $\forall\ n > n_o$.

Nếu một thuật toán có thời gian thực hiện T(n) = O(f(n)), chúng ta sẽ nói rằng thuật toán có thời gian thực hiện cấp f(n).

- ***Ví dụ:*** Giả sử $T(n) = 10n^2 + 4n + 4$

  Ta có : $T(n) \le 10n^2 + 4n^2 + 4n^2 = 18n^2$, với $\forall n \ge 1$ (ở đây c=18, n_o=1)

Vậy $T(n) = O(n^2)$. Trong trường hợp này ta nói thuật toán có độ phức tạp (có thời gian thực hiện) cấp n^2.

Bảng 1.1 sau đây cho biết các cấp thời gian thực hiện thuật toán được sử dụng rộng rãi nhất và tên gọi thông thường của chúng.

Bảng 1.1. *Các cấp thời gian thực hiện thuật toán*

Ký hiệu ô lớn	Tên gọi thông thường
$O(1)$	Hằng
$O(\log_2 n)$	logarit
$O(n)$	Tuyến tính
$O(n\log_2 n)$	$n\log_2 n$
$O(n^2)$	Bình phương
$O(n^3)$	Lập phương
$O(2^n)$	Mũ

Danh sách trên sắp xếp theo thứ tự tăng dần của cấp thời gian thực hiện

Các hàm như $\log_2 n$, n, $n\log_2 n$, n^2, n^3 được gọi là các hàm đa thức. Giải thuật với thời gian thực hiện có cấp hàm đa thức thì thường chấp nhận được.

Các hàm như 2^n, $n!$, n^n được gọi là hàm loại mũ. Một giải thuật mà thời gian thực hiện của nó là các hàm loại mũ thì tốc độ rất chậm.

1.3.2.2. Xác định độ phức tạp tính toán

Xác định độ phức tạp tính toán của một giải thuật bất kỳ có thể dẫn đến những bài toán phức tạp. Tuy nhiên, trong thực tế, đối với một số giải thuật ta cũng có thể phân tích được bằng một số qui tắc đơn giản.

- **Qui tắc tổng**: Giả sử $T_1(n)$ và $T_2(n)$ là thời gian thực hiện của hai giai đoạn chương trình P_1 và P_2 mà $T_1(n) = O(f(n))$; $T_2(n) = O(g(n))$ thì thời gian thực hiện đoạn P_1 rồi P_2 tiếp theo sẽ là $T_1(n) + T_2(n) = O(\max(f(n),g(n)))$.

Chứng minh:

Theo định nghĩa:

$T_1(n) = O(f(n))$ nên $\exists c_1>0$ và $n_1>0$ sao cho $T_1(n) \le c_1 f(n)$ với $\forall n>n_1$

$T_2(n) = O(g(n))$ nên $\exists c_2>0$ và $n_2>0$ sao cho $T_2(n) \le c_2 g(n)$ với $\forall n>n_2$

Chọn $n_0=\max(n_1, n_2)$, $c=\max(c_1,c_2)$; ta có:

$\forall n>n_0$:

$T_1(n) + T_2(n) \le c_1 f(n) + c_2 g(n) \le c(f(n) + g(n))$.

Vậy $T_1(n) + T_2(n) = O(\max(f(n),g(n)))$.

Ví dụ: Trong một chương trình có 3 bước thực hiện mà thời gian thực hiện từng bước lần lượt là $O(n^2)$, $O(n^3)$ và $O(n\log_2 n)$ thì thời gian thực hiện 2 bước đầu là $O(\max(n^2, n^3)) = O(n^3)$. Khi đó thời gian thực hiện chương trình sẽ là $O(\max(n^3, n\log_2 n)) = O(n^3)$.

- **Qui tắc nhân**: Nếu tương ứng với P_1 và P_2 là $T_1(n) = O(f(n))$, $T_2(n) = O(g(n))$ thì thời gian thực hiện P_1 và P_2 lồng nhau sẽ là : $T_1(n)T_2(n) = O(f(n)g(n))$

Chứng minh:

Theo định nghĩa:

$T_1(n) = O(f(n))$ nên $\exists c_1>0$ và $n_1>0$ sao cho $T_1(n) \le c_1 f(n)$ với $\forall n>n_1$

$T_2(n) = O(g(n))$ nên $\exists c_2>0$ và $n_2>0$ sao cho $T_2(n) \le c_2 g(n)$ với $\forall n>n_2$

Chọn $n_0=\max(n_1, n_2)$, $c=c_1c_2$; ta có:

$\forall n>n_0$:

$T_1(n)T_2(n) \le c_1c_2(f(n)g(n))$

Vậy $T_1(n)T_2(n) = O(f(n)g(n))$

Để đánh giá thời gian thực hiện thuật toán, ta cần biết thời gian thực hiện của các lệnh như sau:

1. Thời gian thực hiện các lệnh đơn: gán, đọc, viết là O(1)

2. Lệnh hợp thành (hay lệnh ghép): thời gian thực hiện lệnh hợp thành được xác định bởi luật tổng.

3. Lệnh **if**:

if (<bthức logic>) S1; **else** S2;

Giả sử thời gian thực hiện các lệnh S_1, S_2 tương ứng là O(f(n)) và O(g(n)). Khi đó thời gian thực hiện lệnh **if** là O(max (f(n), g(n)))

4. Lệnh **chọn lựa**: Lệnh này được đánh giá như lệnh **if**

5. Lệnh **while**:

while (<bthức logic>**)** S;

Giả sử thời gian thực hiện lệnh S (thân của **while**) là O(f(n)) . Giả sử g(n) là số tối đa các lần thực hiện lệnh **while.** Lúc đó độ phức tạp của toàn vòng lặp này là O(f(n)g(n)).

6. Các lệnh lặp khác có tính chất tương tự.

1.3.2.3. Đánh giá độ phức tạp của thủ tục (hoặc hàm) đệ qui

Trước hết chúng ta xét một ví dụ cụ thể. Ta sẽ đánh giá thời gian thực hiện của hàm đệ qui sau

```
long Fact (int n)
{
```

```
        if( n <= 1) Fact =1;
        else Fact = n* Fact (n - 1);
    }
```

Trong hàm này cỡ của dữ liệu vào là n. Giả sử thời gian thực hiện hàm là T(n). Với n =1, chỉ cần thực hiện lệnh gán fact = 1, do đó T(1) = O(1).

Với n > 1, cần thực hiện lệnh gán fact = n*fact(n - 1). Do đó thời gian T(n) là O(1) (để thực hiện phép nhân và phép gán) cộng với T(n -1) (để thực hiện lời gọi đệ qui fact(n - 1)). Tóm lại, ta có quan hệ đệ qui sau:

$$T(1) = O(1)$$
$$T(n) = O(1) + T(n - 1)$$

Thay các O(1) bởi các hằng nào đó, ta nhận được quan hệ đệ qui sau

$$T(1) = C_1$$
$$T(n) = C_2 + T(n - 1)$$

Để giải phương trình đệ qui, tìm T(n), chúng ta áp dụng phương pháp thế lặp.

$$\begin{aligned} T(n) &= C_2 + T(n - 1) \\ &= C_2 + [C_2 + T(n-2)] = 2C_2 + T(n-2) \\ &= 2C_2 + [C_2 + T(n-3)] = 3C_2 + T(n-3) \\ &\ldots\ldots \\ &= (n - 1)\, C_2 + T(1) \end{aligned}$$

hay $T(n) = (n - 1)\, C_2 + C_1$, trong đó C_1 và C_2 là các hằng. Do đó, T(n) = O(n).

Từ ví dụ trên, ta suy ra phương pháp tổng quát sau đây để đánh giá thời gian thực hiện thủ tục (hàm) đệ qui. Để đơn giản, giả thiết rằng các thủ tục (hàm) là đệ qui trực tiếp. Điều đó có nghĩa là các thủ tục (hàm) chỉ chứa các lời gọi đệ qui đến chính nó. Giả sử thời gian thực hiện thủ tục là T(n), với n là cỡ dữ liệu đầu vào. Khi đó thời gian thực hiện các lời gọi đệ qui được đánh giá thông qua các bước đi sau.

1.3.2.4. Một số ví dụ

- ***Ví dụ 1**:* Giải thuật tính giá trị của e^x tính theo công thức gần đúng

$e^x = 1 + x/1! + x^2/2! + \ldots + x^n/ n!$, với x và n cho trước

```
float Exp1 (int n, int x)
/* Tính từng số hạng sau đó cộng dồn lại */
{  float s, p;
   s := 1;                                  (1)
   for(int i=1; i<=n; i++)                  (2)
   {
          p = 1;                            (3)
          for(int j=1; j<=i; j++)           (4)
               p = p*x/j;                   (5)
          s = s + p;                        (6)
   }
   return  s;                               (7)
```

```
}
```

Ta thấy câu lệnh (1) và (7) là các câu lệnh gán nên chúng có thời gian thực hiện là O(1). Do đó thời gian thực hiện của giải thuật phụ thuộc vào câu lệnh (2). Ta đánh giá thời gian thực hiện câu lệnh này. Trong thân của câu lệnh này bao gồm các lệnh (3), (4), (5) và (6). Hai câu lệnh (3) và (7) có thời gian thực hiện là O(n) vì mỗi câu lệnh được thực hiện n lần. Riêng câu lệnh (5) thì thời gian thực hiện nó còn phụ thuộc vào câu lệnh (4) nên ta còn phải đánh giá thời gian thực hiện câu lệnh (4).

Với i = 1 thì câu lệnh (5) được thực hiện 1 lần

Với i = 2 thì câu lệnh này được thực hiện 2 lần

.....

Với i = n thì câu lệnh này được thực hiện n lần

Suy ra tổng số lần thực hiện câu lệnh (5) là :

$1 + 2 + ... + n = n(n + 1)/2$ lần

Do đó thời gian thực hiện câu lệnh này là $O(n^2)$ và đây cũng là thời gian thực hiện của giải thuật.

- ***Ví dụ 2:*** Phân tích thuật toán Euclide (thuật toán tìm ước số chung lớn nhất của hai số nguyên)

```
int Euclide(int m, int n)
{
    int r = m%n;                    (1)
    while( r !=0 )                  (2)
    {
            m = n;                  (3)
            n = r;                  (4)
            r = m%n;                (5)
    }
    return n;                       (6)
}
```

Thời gian thực hiện thuật toán phụ thuộc vào số nhỏ nhất trong hai số m và n. Giả sử $m \geq n > 0$, khi đó cỡ của dữ liệu vào là n. Các lệnh (1) và (6) có thời gian thực hiện là O(1) vì chúng là các câu lệnh gán. Do đó thời gian thực hiện thuật toán là thời gian thực hiện các lệnh while, ta đánh giá thời gian thực hiện câu lệnh (2). Thân của lệnh này, là khối gồm ba lệnh (3), (4) và (5). Mỗi lệnh có thời gian thực hiện là O(1). Do đó khối có thời gian thực hiện là O(1). Ta còn phải đánh giá số lớn nhất các lần thực hiện lặp khối.

Ta có:

$m = n.q_1 + r_1$, $0 \leq r_1 < n$

$n = r_1.q_2 + r_2$, $0 \leq r_2 < r_1$

Nếu $r_1 \leq n/2$ thì $r_2 < r_1 \leq n/2$, do đó $r_2 < n/2$

Nếu r1 > n/2 thì q2 = 1, tức là n = r1 + r2, do đó r2 < n/2.

Tóm lại, ta luôn có r2 < n/2.

Như vậy cứ hai lần thực hiện khối lệnh thì phần dư r giảm đi còn một nửa của n. Gọi k là số nguyên lớn nhất sao cho $2^k \leq n$. Suy ra số lần lặp tối đa là $2k + 1 \leq 2\log_2 n + 1$. Do đó thời gian thực hiện lệnh while là $O(\log_2 n)$. Đó cũng là thời gian thực hiện của thuật toán.

Bài tập cuối chương

1. giả sử rằng $T_1(n) = O(f(n))$ và $T_2(n) = O(f(n))$. câu nào sau đây là đúng?
   a. $T_1(n) + T_2(n) = O(f(n))$
   b. $T_1(n) - T_2(n) = O(f(n))$
   c. $T_1(n)/T_2(n) = O(1)$
   d. $T_1(n) = O(T_2(n))$
2. Chứng minh rằng với bất kỳ hằng số k, ta có: $\log^k n = O(n)$.
3. Phân tích các thuật toán sau theo thời gian thực

a.
```
sum = 0;
for( i=0; i<n; i++ )sum++;
```
b.
```
sum = 0;
for( i=0; i<n; i++ )
    for( j=0; j<n; j++ )sum++;
```
c.
```
sum = 0;
for( i=0; i<n; i++ )
    for( j=0; j<n*n; j++ )sum++;
```
d.
```
sum = 0;
for( i=0; i<n; i++ )
    for( j=0; j<i; j++ )sum++;
```
e.
```
sum = 0;
for( i=0; i<n; i++ )
   for( j=0; j<i*i; j++ )
      for( k=0; k<j; k++) sum++;
```
f.
```
sum = 0;
for( i=1; i<n; i++ )
   for( j=1; j<i*i; j++ )
      if( j%1 == 0 )
       for( k=0; k<j; k++ ) sum++;
```
g.
```
sum = 0;
for(int i = 1; i<=n; i++){readln(x); sum := sum + 1;}
```
h.
```
for(int i = 1; i<=n; i++)
      for(int j = 1; j<=n; j++)
      {
          C[i][j] = 0;
          for(int k = 1; k<=n; k++)
          C[i][j]=C[i][j] + A[i][k] + B[k][j];
      }
```
i.
```
for(int i = 1; i <= n - 1; i++)
  for(int j =1; j <= n -1; j++)
       if(X[j] > X[j + 1])
       {tg=X[j]; X[j]=X[j+1]; X[j+1]= tg;}
```

4. Cho *i* và *j* là hai số nguyên và định nghĩa $q(i, j)$ bởi $q(i, j) = 0$ nếu $i < j$ và $q(i - j, j) + 1$ nếu $i \geq j$

a. Giá trị của $q(7,2)$ là bao nhiêu?

b. Ta có thể xác định $q(0,0)$ không?

c. Ta có thể xác định $q(-3, -4)$ không?

5. Cho a là mảng số thực và i, n là các số nguyên dương. Hàm m(a,i,n) được định nghĩa: m(a,i,n) = a[i] nếu i = n và max(m(a,i + 1,n) ,a[i]) nếu i < n

a. Tìm m(a,i,n) nếu i = 1, n = 6, và a là

0. 6.8
1. 3.2
2. -5.0
3. 14.6
4. 7.8
5. 9.6
6. 3.2
7. 4.0

b. Hàm m(a,i,n) nghĩa là gì?

6. Viết các hàm đệ qui thực hiện các công việc sau:

- Tính n!
- Tính S= 1 + 2 + 3 + … + n
- Tính S= 1 + 3 +5 + … + (2k+1) với $2k+1 \leq n$
- Đổi số nguyên n hệ 10 sang hệ 2
- Đảo ngược một số nguyên dương
- Dãy số Fibonaci
- Tìm ước số chung lớn nhất của 2 số nguyên A & B
- Tính 2^n
- Tính x^y

Chương 2. DANH SÁCH
(List)

2.1. Kiểu dữ liệu trừu tượng danh sách (List Abstract Data Type)

2.1.1. Định nghĩa danh sách

Danh sách là một dãy hữu hạn các phần tử cùng kiểu. Chẳng hạn, danh sách sinh viên của một lớp, danh sách các số nguyên,…

Giả sử L là một danh sách có n phần tử (n>=0).

$$L = (a_1, a_2, ..., a_n)$$

Ta gọi n là độ dài của danh sách. Nếu n>=1 thì a_1 được gọi là phần tử đầu tiên, a_n được gọi là phần tử cuối cùng của danh sách L; nếu n=0 thì L được gọi là danh sách rỗng (empty list).

Một tính chất quan trọng của danh sách là các phần tử của nó được sắp tuyến tính: nếu n>1 thì phần tử a_{i-1} "đi trước" phần tử a_i (với i=2, …, n). Ta gọi a_i (với i=1, 2, ..., n) là phần tử ở vị trí thứ i của danh sách.

2.1.2. Các phép toán trên danh sách

Khi mô tả một mô hình dữ liệu, ta cần xác định các phép toán có thể thực hiện trên mô hình toán học được dùng làm cở sở cho mô hình dữ liệu. Có rất nhiều phép toán trên danh sách, và trong các ứng dụng, thông thường ta chỉ sử dụng một nhóm các phép toán nào đó. Sau đây là một số phép toán chính trên danh sách.

1. Khởi tạo danh sách rỗng.
2. Xác định độ dài của danh sách.
3. Loại phần tử ở vị trí thứ p của danh sách.
4. Xen phần tử X vào danh sách sau vị trí thứ p.
5. Xen phần tử X vào danh sách trước vị trí thứ p.
6. Tìm phần X trong danh danh sách.
7. Kiểm tra xem danh sách có rỗng không?
8. Kiểm tra xem danh sách có đầy không?
9. Duyệt danh sách.
10. Các phép toán khác:Truy cập đến phần tử thứ i của danh sách (để tham khảo hoặc thay thế), kết hợp hai danh sách thành một danh sách, tách một danh sách thành nhiều danh sách v.v...

2.2. Danh sách đặc (Condensed List)

2.2.1. Định nghĩa danh sách đặc

Danh sách đặc là danh sách mà các phần tử được sắp xếp kế tiếp nhau trong bộ nhớ, đứng ngay sau phần tử thứ a_i là phần tử thứ a_{i+1}.

2.2.2. Cài đặt danh sách đặc bởi mảng

Để đơn giản, ta sử dụng một mảng nguyên gồm n phần tử a[0], a[1], …, a[n-1] để biểu diễn danh sách đặc.

➢ *Ta cài đặt danh sách đặc như sau:*

```
#define Max_size 100
int a[Max_size];
int n;//n biểu diễn số phần tử trong danh sách.
```

2.2.3. Các phép toán trên danh sách

2.2.3.1. Khởi tạo danh sách

Khi khởi tạo, danh sách là rỗng, ta cho số phần tử n bằng –1.

➢ *Giải thuật:*

```
void Initialize()
{ n = -1;}
```

2.2.3.2. Kiểm tra danh sách có rỗng không

Kiểm tra, nếu danh sách rỗng (nghĩa là n = -1) thì trả về kết quả TRUE, ngược lại trả về kết quả FALSE.

➢ *Giải thuật:*

```
int IsEmpty()
{ return (n == -1)?1:0;}
```

2.2.3.3. Kiểm tra danh sách có đầy không

Kiểm tra, nếu danh sách đầy (nghĩa là n = Max_size - 1) thì trả về kết quả TRUE, ngược lại trả về kết quả FALSE.

➢ *Giải thuật:*

```
int IsFull()
{ return (n == Max_size - 1)?1:0;}
```

2.2.3.4. Thêm một phần tử vào danh sách

Giả sử ta cần thêm phần tử có giá trị x tại vị trí thứ i, khi đó các phần tử từ a[i] đến a[n] được di chuyển ra sau một vị trí.

➢ *Giải thuật:*

```
void Add(int x, int i)
{if(!IsFull())
  {   n++;
      for(int j=n; j>i; j--) a[j]=a[j-1];
      a[i] = x;
  }
}
```

2.2.3.5. Loại bỏ một phần tử khỏi danh sách

Giả sử cần loại bỏ phần tử a[i] của danh sách, khi đó các phần tử a[i+1] đến a[n] được di chuyển đến trước một vị trí.

➢ *Giải thuật:*

```
void Delete(int i)
{
  if(!IsEmpty())
  {
       for(int j=i; j<n; j++) a[j]=a[j+1];
       n--;
  }
}
```

2.2.3.6. Tìm kiếm một phần tử trong danh sách

Giả sử ta cần tìm xem có phần tử nào có giá trị x trong danh sách không? Nếu tìm thấy thì trả về vị trí cần tìm, nếu không trả về -1 (không tìm thấy).

➢ *Giải thuật (Tìm kiếm tuần tự):*

```
int Search(int x)
{
     for(int i=0; i<=n; i++)
          if(a[i]==x)return i;
     return -1;
}
```

2.2.4. Đặc điểm của danh sách đặc

2.2.4.1. Ưu điểm

- Khi sử dụng danh sách với mật độ cao nhất 100% thì không lãng phí bộ nhớ.
- Dễ dàng truy xuất đến phần tử thứ i trong danh sách.
- Dễ dàng tìm kiếm một phần tử có nội dung là x.

2.2.4.2. Nhược điểm

- Khi không sử dụng danh sách với mật độ cao nhất thì gây lãng phí bộ nhớ.
- Không phù hợp cho các phép toán thêm vào và loại bỏ.

2.3. Danh sách liên kết đơn (Single Linked List)

2.3.1. Định nghĩa

Danh sách liên kết (DSLK) đơn là danh sách mà mỗi phần tử được nối kết với nhau thông qua một vùng liên kết của chúng.

2.3.2. Biểu diễn danh sách liên kết đơn

➢ Một phần tử trong danh sách liên kết đơn bao gồm hai vùng chính (Hình 2.1):

- Vùng chứa nội dung (info).
- Vùng liên kết (link)

info	link

Hình 2.1. *Biểu diễn nút trong DSLK đơn*

➢ Các phần tử trong DSLK được biểu diễn bằng kiểu con trỏ (pointer).

➢ Ngoài các phần tử trong danh sách liên kết đơn, ta còn sử dụng một biến chỉ điểm đầu First trỏ vào phần tử đầu tiên (hoặc chứa địa chỉ phần tử đầu tiên) của danh sách liên kết đơn.

☞ ***Khai báo danh sách liên kết đơn:***

```
struct Tro
{
   …..        //Khai báo các trường nội dung
   Tro *link;//Khai báo trường liên kết
};
Tro *First;
```

❑ ***Ví dụ 1:*** Khai báo danh sách liên kết đơn có chỉ điểm đầu First, các nút (phần tử trong danh sách liên kết) có trường nội dung kiểu nguyên.

```
struct Tro
{
  int nd;       //Khai báo trường nội dung
  Tro *link;    //Khai báo trường liên kết
};
Tro *First;
```

❑ ***Ví dụ 2:*** Khai báo danh sách liên kết đơn có chỉ điểm đầu First, với mỗi nút của danh sách có các trường nội dung là: Tên, Tuổi.

```
struct Tro
{
  char Ten[10];
  int Tuoi;
  Tro *link;         //Khai báo trường liên kết
};
Tro *First;
```

✍ ***Ghi chú:*** Để đơn giản, ta sử dụng cách khai báo trong ví dụ 1 để cài đặt các phép toán trên danh sách liên kết đơn.

2.3.3. Các phép toán trên danh sách liên kết đơn

2.3.3.1. Khởi tạo danh sách liên kết đơn

Khi khởi tạo, danh sách là rỗng, ta cho First trỏ đến NULL.

➢ *Giải thuật:*

```
void Initialize()
{First=NULL;}
```

2.3.3.2. Chèn một phần tử vào danh sách liên kết đơn

Để chèn một phần tử kiểu con trỏ, đầu tiên ta phải cấp phát bộ nhớ cho phần tử này bằng toán tử ***new*** theo cú pháp sau:

<biến trỏ> = new <Kiểu trỏ>;

- **Bài toán:** Hãy chèn phần tử có giá trị là x vào danh sách liên kết đơn có chỉ điểm đầu là First. Khi thực hiện phép chèn, ta có thể thực hiện theo một trong 3 cách sau:

i. Chèn vào đầu danh sách

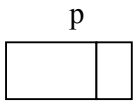
➢ *Giải thuật*

```
void InsertFirst(int x, Tro *&First)  //First là tham biến trỏ
{
 Tro *p=new Tro;      {1} //Khai báo và khởi tạo biến trỏ p
 p->nd=x;             {2}
 p->link=First;       {3}
 First=p;             {4}
}
```

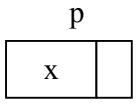
Trong giải thuật này ta chú ý đến 2 trường hợp:

✍ Trường hợp danh sách liên kết rỗng (First=NULL):

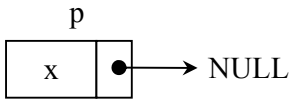
{1} trong bộ nhớ sẽ tạo ra biến trỏ ***p***



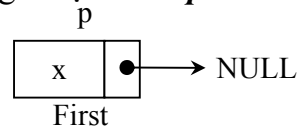
{2} đặt giá trị ***x*** vào trường nội dung của ***p***



{3} Trường liên kết của ***p*** trỏ đến ***First***. Nhưng do ***First=NULL***, cho nên trường liên kết của ***p*** sẽ trỏ đến ***NULL***



{4} Biến trỏ ***First*** nhận giá trị mới là ***p***



✍ Trường hợp danh sách liên kết không rỗng:

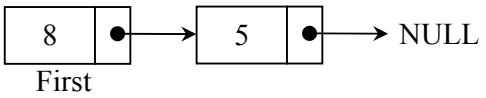
{1} và {2} tương tự như trường hợp danh sách liên kết rỗng.

{3} Trường liên kết của ***p*** trỏ đến ***First*** (chú ý First không rỗng).

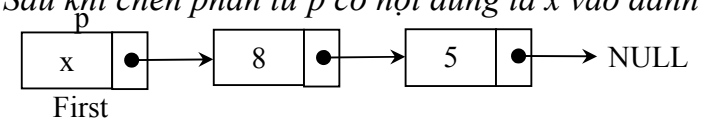
{4} Biến trỏ nhận giá trị mới là ***p***

❑ ***Ví dụ:***

Trước khi chèn phần tử có nội dung là x vào danh sách liên kết đơn



Sau khi chèn phần tử p có nội dung là x vào danh sách liên kết đơn



Hình 2.2. *DSLK trước và sau khi chèn phần tử có nội dung là x*

ii. Chèn vào cuối danh sách

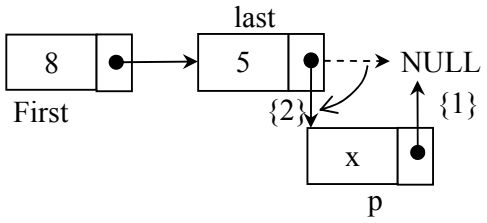
➢ *Giải thuật*

```
void InsertLast(int x, Tro *&First)
{
  Tro *p=new Tro;
  p->nd=x;
  if(!First)
  { p->link=First; First=p; }
  else
  {
      Tro *last=First;
      while(last->link)last=last->link;
      p->link=last->link;      {1} //hoặc p->link=NULL;
      last->link=p;            {2}
  }
}
```

Trong giải thuật này có hai trường hợp xảy ra:

✍ Trường hợp ***if***: tương tự như giải thuật chèn đầu vào danh sách rỗng.

✍ Trường hợp ***else***: ta phải tìm đến phần tử cuối cùng trong danh sách thông qua vòng lặp ***while***. Kết thúc vòng lặp ***while***, ta có kết quả:



Hình 2.3. *Chèn phần tử có giá trị x vào cuối DSLK đơn*

- Câu lệnh {1} nghĩa là: Do ***last->link*** trỏ đến ***NULL***, suy ra ***p->link*** trỏ đến ***NULL***.

- Câu lệnh {2} nghĩa là: ***last->link*** trỏ đến phần tử ***p***.

➢ *Giải thuật đệ quy*

```
void InsertLast1(int x, Tro *&First)
{
      if(!First)
      {First = new Tro; First -> nd = x; First->link=NULL; }
      else InsertLast1(x, First->link);
}
```

iii. Chèn vào vị trí bất kỳ trong danh sách

Giả sử ta cần chèn phần tử có giá trị x vào danh sách liên kết đơn có thứ tự tăng dần (theo trường nội dung của mỗi phần tử trong danh sách) và trỏ đầu bởi First. Yêu cầu sau khi chèn, ta phải thu được danh sách liên kết đơn có thứ tự tăng dần!

➢ *Giải thuật*

```
void InsertAnywhere(int x, Tro *&First)
{
 Tro *p = new Tro;      //Cấp phát bộ nhớ cho biến trỏ p
 p->nd = x;             //Gán trường nd của p bằng x
 if(First == NULL)  {1}  //Trường hợp DSLK rỗng
 {
      p->link = First; First=p;
 }
 else
    if(First->nd > x){2}//Trường hợp chèn p vào đầu DSLK
    {
          p->link=First; First=p;
     }
     else            {3}//Trường hợp chèn p vào giữa hoặc cuối
     {
       Tro *befo,*q=First;
       while(q && q->nd<x)
      {
            befo=q;q=q->link;
       }
       p->link=q;       {3.1}
       befo->link=p;    {3.2}
     }
}
```

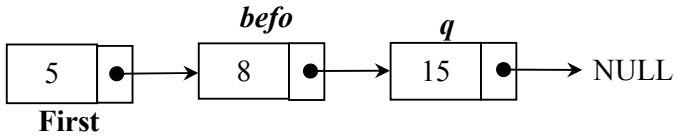
Trong giải thuật này, sau khi cấp phát bộ nhớ cho biến p (thông qua lệnh ***Tro *p = new Tro;***), gán x cho nội dung của p (thông qua lệnh ***p->nd = x***). Ta còn phải xét đến 3 trường hợp nhằm đưa phần tử p vào DSLK đơn:

✍ Trường hợp {1}: Trong trường hợp này DSLK rỗng, cho nên trường hợp này tương tự mục ***i.*** và ***ii.*** Nghĩa là chèn ***p*** trong trường hợp DSLK rỗng (***First = NULL***)

✍ Trường hợp {2}: Trong trường hợp này luôn chèn ***p*** vào đầu DSLK (trường hợp này tương tự mục ***i.***)

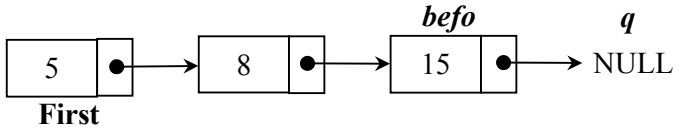
✍ Trường hợp {3}: Ta phải tìm ra hai phần tử ***befo*** và ***q*** trong DSLK thông qua vòng lặp ***while***. Kết thúc vòng lặp ***while***, ta luôn tìm được kết quả: phần tử ***befo*** luôn đi trước phần tử ***q***.

- ***Ví dụ:*** với x=10, kết thúc vòng lặp while, ta có kết quả:



Hình 2.4. *Vị trí của con trỏ befo và q, với x=10*

- ***Ví dụ:*** với x=20, kết thúc vòng lặp while, ta có kết quả:



Hình 2.5. *Vị trí của con trỏ befo và q, với x=20*

Tiếp theo:

- Câu lệnh {3.1} nghĩa là: trường liên kết của ***p*** trỏ đến ***q***.
- Câu lệnh {3.2} nghĩa là: ***befo->link*** trỏ đến phần tử ***p***.

➢ *Giải thuật đệ quy*

```
void InsertAnywhere1(int x, Tro *&First)
{
  if(!First)
  {First = new Tro; First -> nd = x;
   First->link=NULL; }
  else
    if(First->nd > x)
    {
      Tro *p=new Tro; p->nd=x;p->link=First;First=p;
    }
    else InsertAnywhere1(x, First->link);
}
```

2.3.3.3. Xóa một phần tử khỏi danh sách liên kết đơn

Để xóa một phần tử kiểu con trỏ khỏi bộ nhớ (thu hồi bộ nhớ), ta sử dụng toán tử ***delete*** theo cú pháp sau:

delete <biến trỏ>;

Khi thực hiện thao tác xóa, ta có thể rơi vào một trong 3 cách sau:

i. Xóa phần tử đầu tiên trong DSLK

➢ *Giải thuật*

```
void DeleteFirst(Tro *&First)
{
  if(First)
```

```
    {Tro *p=First; First=First->link; delete p;}
}
```

ii. Xóa phần tử cuối cùng trong DSLK

➢ *Giải thuật*

```
void DeleteLast(Tro *&First)
{
  if(First)
     if(First->link == NULL)
     { Tro *p=First; First = First ->link; delete p;}
     else DeleteLast(x, First->link);
}
```

iii. Xóa phần tử bất kỳ trong DSLK

Giả sử ta cần xóa phần tử có giá trị x khỏi DSLK đơn và trỏ đầu bởi First.

➢ *Giải thuật*

```
void DeleteAnywhere(int x, Tro *&First)
{
  if(First)
     if(First->nd == x)
     { Tro *p=First; First = First ->link; delete p;}
     else DeleteAnywhere(x, First->link);
}
```

2.3.3.4. Tìm kiếm trong danh sách liên kết đơn

- **Bài toán:** Hãy tìm phần tử có giá trị là x trong danh sách liên kết đơn có chỉ điểm đầu là First. Nếu tìm thấy thì trả về địa chỉ của phần tử chứa x trong DSLK đơn, nếu không thì trả về NULL

➢ *Giải thuật*

```
Tro *FindElement(int x, Tro *First)
{
  if(!First) return NULL;
  else
       if(First->nd == x) return First;
       else return FindElement(x, First->link);
}
```

2.3.3.5. In ra màn hình giá trị các phần tử trong danh sách liên kết đơn

➢ *Giải thuật*

```
void Print(Tro *First)
{
  if(First)
  { cout<<p->nd<<" ";  Print(First->link); }
}
```

2.3.3.6. Sắp xếp trong danh sách liên kết đơn

Hãy sắp xếp tăng dần theo trường nội dung của các nút trong DSLK đơn có chỉ điểm đầu là First.

➢ *Giải thuật*

```
void Sort(Tro *First)
{
  Tro *p=First;
  while(p->link->link)
  {
       Tro *q=p->link;
       while(q->link)
       {
         if(p->nd > q->nd)
         { int temp=p->nd; p->nd=q->nd; q->nd=temp;}
         q=q->link;
       }
       p=p->link;
  }
}
```

2.3.3.7. Ví dụ về danh sách liên kết đơn

- **Bài toán:** Khai báo danh sách liên kết đơn có chỉ điểm đầu First, các nút (phần tử trong danh sách liên kết) có trường nội dung kiểu nguyên. Sau đó hãy thực hiện các công việc:

  i. Chèn 5 giá trị bất kỳ cho danh sách này.

  ii. In ra màn hình các giá trị có trong danh sách.

  iii. Tìm xem x có trong danh sách không (với x nhập từ bàn phím)?

  iv. Hãy xóa phần tử có giá trị là x trong danh sách.

- **Thực hiện:**

```
#include<iostream.h>
#include<conio.h>
struct Tro
 {
  int nd;
  Tro *link;
 };
Tro *First;
void Initialize()
{First=NULL;}
void InsertFirst(int x,Tro *&First)
{ Tro *p=new Tro; p->nd=x; p->link=First; First=p; }
Tro *FindElement(int x, Tro *First)
{
  if(!First) return NULL;
  else
       if(First->nd == x) return First;
       else return FindElement(x, First->link);
}
```

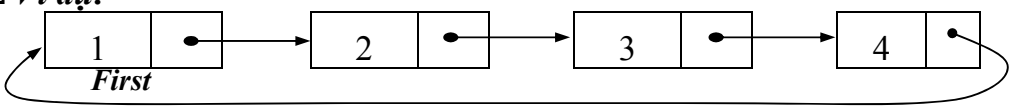
```
void DeleteAnywhere(int x, Tro *&First)
{
      if(First)
        if(First->nd == x)
        {Tro *p=First; First=First ->link; delete p;}
        else DeleteAnywhere(x, First->link);
}
void Print(Tro *First)
{ if(First){cout<<First->nd<<" ";Print(First->link);}}
void main()
{
 clrscr();
 Initialize(); int x;
 for(int i=1; i<=5; i++)
 {
  cout<<"Noi dung nut <<i<<" la:"; cin>>x;
  InsertFirst(x, First);
 }
 cout<<"Cac gia tri trong DSLK la:\n";
 Print(First);
 cout<<"Hay nhap gia tri can tim: "; cin>>x;
 if(FindElement(x,First))
   cout<<x<<" la nd cua mot phan tư trong DSLK\n";
 else
   cout<<" DSLK khong co phan tư có nd la "<<x<<"\n";
 if(FindElement(x,First))
 {
   DeleteAnywhere(int x, First);
   cout<<"Danh sach sau khi xoa:\n";
   Print(First);
 }
 getch();
}
```

2.3. Danh sách liên kết đơn vòng (Circular Linked List)

2.3.1. Định nghĩa

Danh sách liên kết đơn vòng là danh sách liên kết đơn mà trường liên kết của nút cuối cùng trong danh sách chứa địa chỉ của nút đầu tiên (trỏ đến phần tử đầu tiên).

- ***Ví dụ:***



Hình 2.6. *Danh sách liên kết vòng*

2.3.2. Biểu diễn danh sách liên kết vòng

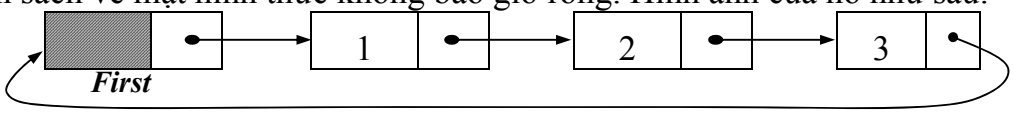
Cách biểu diễn danh sách liên kết vòng tương tự như cách biểu diễn danh sách liên kết đơn.

2.3.3. Các thao tác trên danh sách liên kết vòng

Để đơn giản các phép toán được thực hiện trên danh liên kết vòng với mỗi nút của danh sách có hai trường: ***nd*** (kiểu nguyên), ***link*** (kiểu Tro); được khai báo như sau:

```
struct Tro
{
  int nd;        //Khai báo các trường nội dung
  Tro *link;     //Khai báo trường liên kết
};
Tro *First;
```

Trong danh sách liên kết vòng, ta sử dụng một nút đặc biệt gọi là "nút đầu danh sách". Trường ***nd*** của nút này không chứa dữ liệu và con trỏ ***First*** bây giờ trỏ tới nút đầu danh sách này. Việc dùng thêm nút đầu danh sách đã khiến cho danh sách về mặt hình thức không bao giờ rỗng. Hình ảnh của nó như sau:



Hình 2.7. *Danh sách liên kết vòng với nút đầu đặc biệt*

2.3.3.1. Khởi tạo danh sách liên kết vòng

Khi khởi tạo, danh sách chỉ có một nút đặc biệt ***First***, và trường liên kết của nút này trỏ đến chính nó.

➢ *Giải thuật:*

```
void Initialize()
{ First = new Tro; First->link = First;}
```

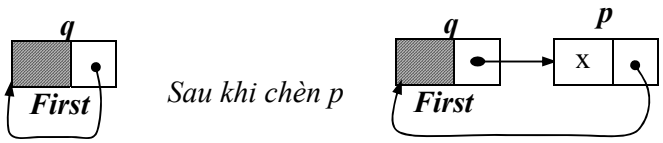
2.3.3.2. Chèn một phần tử vào danh sách liên kết vòng

➢ *Giải thuật:*

```
void Insert(int x)
{ Tro *q=First;
  while(q->link!=First)q=q->link;
  Tro *p=new Tro;p->nd=x;p->link=First;q->link=p;
}
```

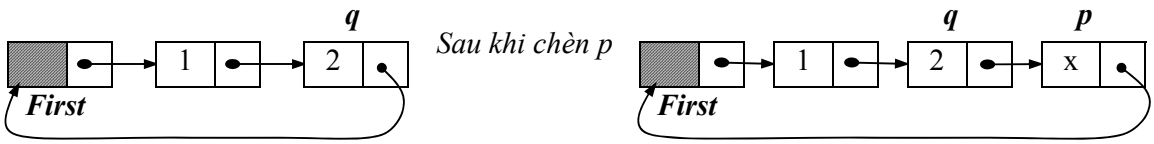
Trong giải thuật này, ta phải tìm đến phần tử cuối cùng q trong danh sách liên kết vòng, sau đó chèn p vào sau q. Có 2 trường hợp xảy ra đối với q:

i. Trường hợp DSLK vòng rỗng:



Hình 2.8. *DSLK vòng rỗng trước và sau khi chèn*

ii. Trường hợp DSLK vòng không rỗng:



Hình 2.9. *DSLK vòng khác rỗng trước và sau khi chèn*

2.3.3.3. Tìm kiếm phần tử trong DSLK vòng có nội dung là x

Viết hàm tìm kiếm phần tử trong DSLK vòng có nội dung là x, nếu tìm thấy thì trả kết quả là 1, ngược lại trả về 0.

➢ *Giải thuật:*

```
int Search(int x)
{
  if(First->link==First)return 0;
   else
   {  Tro *q=First->link;
       int i=1;
       while(q->nd!=x && q->link!=First)q=q->link;
  if(q->nd==x) return 1;
  else return 0;
   }
}
```

2.3.3.4. Xoá các phần tử trong DSLK vòng có nội dung là x

➢ *Giải thuật:*

```
void Delete(int x, Tro *&First)
{
 if(Search(x))
    if(First->nd == x)
    {  Tro *p=First; First=First->link;delete p;
       Delete(x,First);}
    else Delete(x, First->link);
  else
    cout<<"Danh sach rong, khong the xoa duoc!!!\n";
}
```

2.3.3.5. In các phần tử trong DSLK vòng ra màn hình

➢ *Giải thuật:*

```
void Print()
{
 if(First->link != First)
 {
       Tro *p=First->link;
       while(p!=First)
       {cout<<p->nd<<"  ";p=p->link;}
 }
 else cout<<"Danh sach rong.";
}
```

2.3.3.6. Ví dụ về danh sách liên kết vòng

❑ **Bài toán:** Khai báo danh sách liên kết vòng có chỉ điểm đầu First, các nút (phần tử trong danh sách liên kết) có trường nội dung kiểu nguyên. Sau đó hãy thực hiện các công việc:

i. Chèn n giá trị bất kỳ cho danh sách này.

ii. In ra màn hình các giá trị có trong danh sách.

iii. Tìm xem x có trong danh sách không (với x nhập từ bàn phím)?

iv. Hãy xóa các phần tử có giá trị x trong danh sách.

❑ **Thực hiện:**

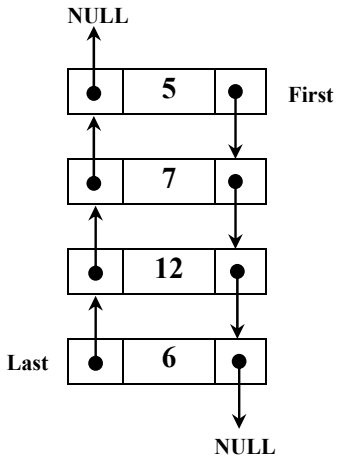
```
#include<iostream.h>
#include<conio.h>
struct Tro
{
  int nd; Tro *link;
};
Tro *First;
void Initialize()
{ First=new Tro; First->link=First;}
void Insert(int x)
{
  Tro *q=First;
  while(q->link!=First)q=q->link;
  Tro *p=new Tro;p->nd=x;p->link=First;q->link=p;
}
int Search(int x)
{
  if(First->link==First)return 0;
  else
  {
       Tro *q=First->link;
       int i=1;
       while(q->nd!=x && q->link!=First)q=q->link;
       if(q->nd==x) return 1;
       else return 0;
  }
}
void Delete(int x, Tro *&First)
{

 if(Search(x))
     if(First->nd==x)
      {
            Tro *p=First; First=First->link;delete p;
```

```
            Delete(x,First);
        }
       else Delete(x, First->link);
    else
      cout<<"Danh sach rong, khong the xoa duoc!!!\n";
}
void Print()
{
    if(First->link != First)
     {
         Tro *p=First->link;
         while(p!=First){cout<<p->nd<<"  ";p=p->link;}
     }
    else cout<<"Danh sach rong.";
}
void main()
{
     clrscr();
     Initialize();
     int n,x;
     cout<<"Can nhap vao ds bao nhieu nut:";cin>>n;
     for(int i=1;i<=n;i++)
     { cout<<"Nd nut "<<i<<" la: "; cin>>x; Insert(x);}
     Print();
     cout<<"\nMuon xoa nut co nd bang may: "; cin>>x;
     Delete(x,First);
     cout<<"Danh sach sau khi xoa: ";
     Print();
     getch();
}
```

2.4. Danh sách liên kết kép (Double Linked List)

2.4.1. Định nghĩa

Danh sách liên kết kép (còn gọi là danh sách liên kết đôi) là danh sách mà mỗi nút của nó có 2 trường liên kết (hình 2.10). Một vùng liên kết chỉ đến phần tử trước nó gọi là previous, một vùng liên kết chỉ đến phần tử sau nó gọi là next.



Hình 2.10. *DSLK kép*

2.4.2. Biểu diễn danh sách liên kết kép

➢ Mỗi nút có 3 vùng chính: vùng chứa các trường nội dung (Info), hai vùng chứa trường liên kết: pre (chỉ đến phần tử trước nó) và next (chỉ đến phần tử sau nó).

pre	Info	next

Hình 2.11. *Biểu diễn nút trong DSLK kép*

➢ Để biểu diễn danh sách liên kết kép, ta sử dụng kiểu dữ liệu con trỏ.

➢ Ngoài các phần tử trong danh sách liên kết kép, ta còn sử dụng 2 biến chỉ điểm, biến chỉ điểm đầu First trỏ vào phần tử đầu tiên (hoặc chứa địa chỉ phần tử đầu tiên), biến chỉ điểm cuối Last trỏ vào phần tử cuối cùng (hoặc chứa địa chỉ phần tử cuối cùng) của danh sách liên kết kép.

☞ ***Khai báo danh sách liên kết kép:***

```
struct Tro
{
   …..            //Khai báo các trường nội dung
   Tro *next;     //Khai báo trường liên kết đến nút sau
   Tro *pre;      //Khai báo trường liên kết đến nút trước
};
Tro *First, *Last;
```

❑ ***Ví dụ:*** Khai báo danh sách liên kết kép có chỉ điểm đầu First, chỉ điểm cuối Last với các nút (phần tử trong danh sách liên kết) có trường nội dung kiểu nguyên.

```
struct Tro
{
  int nd;                 //Khai báo các trường nội dung
  Tro *pre, *next;   //Khai báo trường liên kết
};
Tro *First, *Last;
```

✍ ***Ghi chú:*** Để đơn giản, ta sử dụng cách khai báo trong ví dụ này để cài đặt các phép toán trên danh sách liên kết kép.

2.4.3. Các thao tác trên danh sách liên kết kép

2.3.3.1. Khởi tạo danh sách liên kết kép

Khi khởi tạo, danh sách là rỗng, ta cho First và Last trỏ đến NULL.

➢ *Giải thuật:*

```
void Initialize()
{First=Last=NULL;}
```

2.3.3.2. Chèn một phần tử vào danh sách liên kết kép

Để chèn một phần tử kiểu con trỏ, đầu tiên ta phải cấp phát bộ nhớ cho phần tử này bằng toán tử ***new*** theo cú pháp sau:

<biến trỏ> = new <Kiểu trỏ>;

Để chèn phần tử có giá trị x vào DSLK kép có chỉ điểm đầu First, chỉ điểm cuối Last, ta sử dụng một trong 3 giải thuật sau:

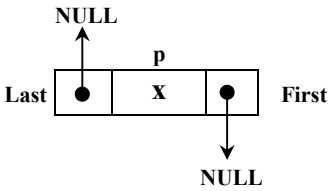
i. Chèn vào đầu danh sách

➢ *Giải thuật*

```
void InsertFirst(int x, Tro *&First, Tro *&Last)
{
    Tro *p=new Tro; p->nd=x;
    p->next=First; p->pre=NULL;
    if(First==NULL)
    {First=p;Last=p;}
    else
    {First->pre=p; First=p;}
}
```

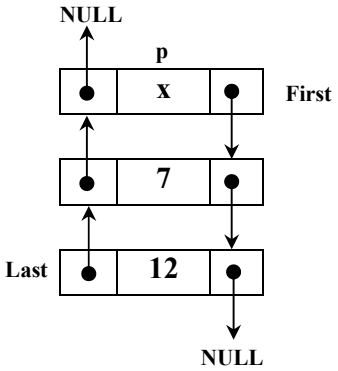
Trong khi chèn, có hai trường xảy ra:

- Trường hợp DSLK kép rỗng, sau khi chèn ta có kết quả:



Hình 2.12. *Chèn dữ liệu vào DSLK kép rỗng*

- Trường hợp DSLK kép khác rỗng, sau khi chèn ta có kết quả:



Hình 2.13. *Chèn dữ liệu vào DSLK kép khác rỗng*

ii. Chèn vào cuối danh sách: *tương tự trường hợp i*

iii. Chèn vào vị trí bất kỳ trong danh sách:

Giả sử danh sách liên kết kép đã có thứ tự tăng dần, giải thuật chèn giá trị x vào danh sách như sau:

➢ *Giải thuật*

```
void InsertAnywhere(int x, Tro *&First, Tro *&Last)
{
    if(!First)
    {
       First=new Tro; First->nd=x; First->next=NULL;
       First->pre=Last;Last=First;
    }
    else
      if(First->nd<x)
       InsertAnywhere(x,First->next,Last);
      else
       {
            Tro *p=new Tro; p->nd=x; p->next=First;
            p->pre=First->pre;First->pre=p;First=p;
       }
}
```

2.3.3.3. Xoá phần tử có giá trị x trong danh sách liên kết kép

➢ *Giải thuật*

```
void DeleteEle(int x,Tro *&First)
{
  if(First)
    if(First->nd==x)
    {  Tro *p=First;
       if(First==Last)
        if(First->pre){Last=First->pre;First=NULL;}
        else First=Last=NULL;
       else
        {First=First->next;First->pre=p->pre;}
       delete p;
    }
   else DeleteEle(x,First->next);
  else cout<<"\nKhong ton tai ptu "<<x<<" trong ds.\n";
}
```

2.3.3.4. In ấn các phần tử trong danh sách liên kết kép

➢ *Giải thuật*

```
void PrintFirst(Tro *First)
{ if(First)
  {cout<<First->nd<<"  ";PrintFirst(First->next);}
}
```

2.3.3.5. Ví dụ về danh sách liên kết kép

❑ **Bài toán:** Khai báo danh sách liên kết kép có chỉ điểm đầu First, chỉ điểm cuối Last, các nút (phần tử trong danh sách liên kết) có trường nội dung kiểu nguyên. Sau đó hãy thực hiện các công việc:

i. Chèn n giá trị bất kỳ cho danh sách này.
ii. In ra màn hình các giá trị có trong danh sách xuất phát từ First.
iii. In ra màn hình các giá trị có trong danh sách xuất phát từ Last.
iv. Hãy xóa phần tử có giá trị x trong danh sách.

- **Thực hiện:**

```
#include<iostream.h>
#include<conio.h>
struct Tro
{
 int nd;
 Tro *next,*pre;
};
Tro *First,*Last;
void Initialize()
{ First=Last=NULL;}
void InsertFirst(int x, Tro *&First, Tro *&Last)
{
  Tro *p=new Tro; p->nd=x;
   p->next=First; p->pre=NULL;
   if(First==NULL)
   { First=p;Last=p; }
   else
   { First->pre=p; First=p; }
}
void InsertAnywhere(int x, Tro *&First, Tro *&Last)
{
   if(!First)
   {
    First=new Tro; First->nd=x; First->next=NULL;
    First->pre=Last;Last=First;
   }
 else
    if(First->nd<x)InsertAnywhere(x,First->next,Last);
    else
     {
       Tro *p=new Tro; p->nd=x; p->next=First;
        p->pre=First->pre;First->pre=p; First=p;
     }
}
void PrintFirst(Tro *First)
{
 if(First)
 {cout<<First->nd<<"  ";PrintFirst(First->next);}
```

```
}
void PrintLast(Tro *Last)
{
 if(Last)
 {cout<<Last->nd<<"  ";PrintLast(Last->pre);}
}
void DeleteEle(int x,Tro *&First)
{
   if(First)
    if(First->nd==x)
    {
       Tro *p=First;
       if(First==Last)
            if(First->pre){Last=First->pre;First=NULL;}
            else First=Last=NULL;
       else
       if(First->pre)
            {First=First->next;First->pre=p->pre;}
       else {First=First->next;First->pre=NULL;}
       delete p;
    }
    else DeleteEle(x,First->next);
   else
    cout<<"\nKhong ton tai ptu "<<x<<" trong d s.\n";
}
void main()
{
 clrscr();
 Initialize();
 int n,x;
 cout<<"Nhap so phan tu: ";cin>>n;
 for(int i=1;i<=n;i++)
 {
   cout<<"Noi dung phan tu "<<i<<" la: ";cin>>x;
   InsertAnywhere(x,First,Last);
 }
 cout<<"\nDanh sach in tu First:";PrintFirst(First);
 cout<<"\nDanh sach in tu Last:";PrintLast(Last);
 cout<<"\nNhap vao gia tri can xoa trong danh sach:";
 cin>>x;
 DeleteEle(x,First);
 cout<<”\nGia tri con lai cua ds sau khi xoa: “;
 PrintFirst(First);
 getch();
}
```

Bài tập cuối chương

1. Khai báo và cài đặt các hàm sau bằng danh sách đặc:

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

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

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

2. Viết hàm loại bỏ các phần tử trùng nhau (giữ lại duy nhất 1 phần tử) trong một danh sách đặc có thứ tự không giảm.

3. Viết hàm xóa khỏi danh sách đặc lưu trữ các số nguyên các phần tử là số nguyên lẻ.

Các bài tập từ 4-12:

Giả sử rằng mỗi nút của DSLK đơn có 2 trường: nd kiểu nguyên và link (trường liên kết)

4. Viết hàm Length(First) nhằm đếm số nút của một danh sách liên kết đơn được trỏ đầu bởi First.

5. Cho $X = (x_1, x_2, ..., x_n)$ và $Y = (y_1, y_2, ..., y_m)$ là hai DSLK đơn. Viết giải thuật trộn hai danh sách này để thu được danh sách $Z = (x_1, y_1, x_2, y_2, ..., x_m, y_m, x_{m+1}, ... x_n)$ nếu $m \leq n$ và $Z = (x_1, y_1, x_2, y_2, ..., x_n, y_n, y_{n+1}, ..., y_m)$ nếu $m > n$. Yêu cầu không thêm bất kỳ một nút mới nào trong quá trình trộn.

6. Viết chương trình con trộn hai danh sách liên kết chứa các số nguyên theo thứ tự tăng để được một danh sách cũng có thứ tự tăng. Yêu cầu không thêm bất kỳ một nút mới nào trong quá trình trộn.

7. Hãy làm bài tập 1, 2 và 3 trong trường hợp DSLK đơn vòng.

8. Hãy nghĩ cách biểu diễn một danh sách, nơi các phép chèn và xóa được thực hiện ở một đầu danh sách. (Hướng dẫn: viết 2 hàm Chèn_đầu và Xóa_đầu).

9. Cho đa thức $F(x) = a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$. Để biểu diễn các hệ số $a_i \neq 0$ của đa thức, người ta sử dụng một DSLK đơn trỏ đầu bởi First, với mỗi nút trong danh sách có các trường: ***hệ số*** (lưu trữ hệ số khác 0 trong đa thức), ***số mũ***, ***link***.

a. Hãy định nghĩa DSLK này.

b. Hãy viết hàm nhập liệu cho danh sách trên.

c. Tính giá trị của đa thức $F(x_0)$, với x_0 là giá trị bất kỳ nhập từ bàn phím.

d. Hãy tính đạo hàm bậc một của đa thức thông qua DSLK đơn.

e. Viết hàm cộng hai đa thức.

f. Viết hàm nhân hai đa thức.

10. Viết giải thuật đảo chiều liên kết trong một DSLK đơn.

11. Làm bài tập 7 trong trường hợp DSLK đơn vòng.

12. Viết hàm tách các phần tử trong một danh sách liên kết đơn thành 2 danh sách, danh sách chứa các số nguyên lẻ, và danh sách chứa các số nguyên chẵn.

13. Minh họa các thao tác sau trên danh sách liên kết đơn, danh sách liên kết kép, danh sách liên kết vòng

a. Khởi tạo danh sách.

b. Thêm một nút có giá trị x vào danh sách.

c. Xóa nút có giá trị x ra khỏi danh sách.

d. Tìm kiếm trên danh sách theo các tiêu chí sau:

i. 1 số x cho trước

ii. số lớn nhất

iii. số bé nhất

iv. số nguyên tố đầu tiên

v. số chính phương đầu tiên

vi. số nguyên tố lớn nhất

vii. số nguyên tố bé nhất

viii. số chính phương lớn nhất

ix. số chính phương bé nhất

e. Sắp xếp danh sách.

14. Viết hàm loại bỏ các phần tử trùng nhau (giữ lại duy nhất 1 phần tử) trong một danh sách liên kết có thứ tự không giảm.

Yêu cầu: Hàm được viết bằng DSLK đơn, DSLK đơn vòng, DSLK kép.

15. Cho 2 danh sách nguyên có thứ tự tăng. Viết các hàm sau đây:

a. Tìm giao của hai danh sách.

b. Tìm hợp của hai danh sách.

c. Tìm hiệu của hai danh sách.

Yêu cầu: Cài đặt các hàm trên bằng DSLK đơn, DSLK đơn vòng, DSLK kép.

Chương 3. NGĂN XẾP VÀ HÀNG ĐỢI
(Stacks and Queues)

Trong chương này ta nghiên cứu hai kiểu dữ liệu thường được thấy trong lĩnh vực khoa học máy tính, đó là stack và queue. Các kiểu dữ liệu này là những trường hợp đặc biệt so với các kiểu dữ liệu thông thường mà ta đã học.

3.1. Kiểu cấu trúc dữ liệu trừu tượng stack (The Stack Abstract Data Type)

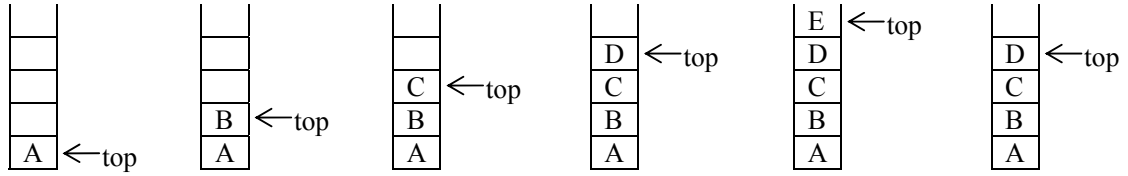
3.1.1. Định nghĩa stack

Stack là một danh sách có thứ tự mà phép chèn và xóa được thực hiện tại đầu cuối của danh sách và người ta gọi đầu cuối này là *đỉnh (top)* của stack.

☞ Dĩ nhiên một phần tử được chèn vào stack khi stack còn chỗ trống, và một phần tử được lấy ra khỏi stack với điều kiện tồn tại ít nhất một phần tử trong stack.

- ***Ví dụ:*** Với stack S = (a_0, a_1,…,a_{n-1}) được cho, ta nói rằng a_0 là phần tử đáy (bottom element) và a_{n-1} là phần tử đỉnh (top element), còn a_i nằm trên phần tử a_{i-1}, với $0<i<n$.

Hạn chế trên stack đó là khi ta thêm lần lượt các phần tử A, B, C, D, E vào stack, thì E sẽ là phần tử đầu tiên được xoá (lấy ra) khỏi stack. Hình 3.1 minh hoạ lần lượt các động tác này.



Hình 3.1. *Phép chèn và xoá phần tử khi thực hiện trên stack*

Do phần tử cuối cùng được chèn cũng là phần tử đầu tiên bị xoá ra khỏi stack. Do vậy stack còn được gọi là một danh sách ***vào sau ra trước***, hay còn gọi là danh sách ***LIFO*** (Last In First Out).

3.1.2. Biểu diễn stack

Cách thức đơn giản nhất để biểu diễn stack đó là sử dụng mảng một chiều.

Gọi stack[max_size] là mảng dùng để lưu các phần tử vào stack, trong đó max_size là số phần tử cực đại có thể lưu vào stack. Phần tử đầu tiên (hay phần tử đáy) của stack được lưu trữ tại stack[0]; phần tử thứ hai tại stack[1] và phần tử thứ i tại stack[i-1].

Kết hợp cùng với mảng stack, ta sử dụng biến *top* làm con trỏ trỏ đến phần tử đỉnh của stack. Bắt đầu, ta khởi gán *top* = -1 nhằm biểu thị stack rỗng.

3.1.3. Các phép toán trên stack

Với cách biểu diễn được cho, ta có thể định nghĩa các phép toán trên stack theo các hàm như sau - *với item là phần tử được thêm hay xoá, max_size nguyên dương, stack có kiểu Stack*:

1. *Stack* CreateS(*max_size*) ::= Tạo một stack rỗng có kích thước cực đại *max_size*.

2. *Boolean* IsFull(*stack, max_size*) ::=

    if (số phần tử trong stack ==max_size) **return** TRUE;

    else return FALSE;

3. *Stack* Add(*stack, item*) ::=

    if (IsFull(*stack*))*stack_full*

    else chèn *item* vào đỉnh của *stack*

4. *Boolean* IsEmpty(*stack*) ::=

    if(*stack* == CreateS(*max_size*)) **return** TRUE;

    else return FALSE;

5. *Element* Delete(*stack*) ::=

    if(IsEmpty(*stack*)) **return**

    else xoá và trả về *item* trên đỉnh stack.

Trong trường hợp này, ta chỉ định phần tử *element* là phần tử có cấu trúc với trường *key*. Thông thường, ta tạo phần tử cấu trúc với nhiều trường. Tuy vậy, ta sử dụng phần tử cấu trúc *element* làm mẫu trong chương này, và ta có thể thêm hoặc chỉnh sửa các trường bên trong cấu trúc này tuỳ theo các yêu cầu ứng dụng của bạn.

- *Giải thuật của hàm Stack CreateS(max_size)*

```
#define MAX_SIZE 100
struct element
{
    int key;
    /* khai báo các trường khác */
};
element stack[MAX_SIZE];
int top = -1;
```

- *Giải thuật của hàm Boolean IsEmpty(stack)*

```
int isemptys()
{return (top<0)?1:0;}
```

- *Giải thuật của hàm Boolean IsFull(stack, max_size)*

```
int isfulls()
```

```
  {return (top>=MAX_SIZE-1)?1:0;}
```

- *Giải thuật của hàm Stack Add(stack, item)*

```
void adds(int *top, element item)
{
   if(*top >= MAX_SIZE-1) cout<<"stack full\n";
   else stack[++*top] = item;
}
```

- *Giải thuật của hàm Element Delete(stack)*

```
void deletes(int *top, element &item)
{ if(*top == -1) cout<<"stack is empty.\n";
  else
  { item=stack[(*top)]; (*top)--;}
}
```

Cách gọi hàm *add* và hàm *delete* là: *adds(&top, item)* và *deletes(&top, item)*, trong đó *item* là biến có kiểu *element*.

3.1.4. Ví dụ

- ***Ví dụ 1:*** Xác định giá trị của số Fibonacci ở vị trí thứ n bằng cách không sử dụng đệ qui.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MAX_SIZE 100
struct element
{
  int key;
};
element stack[MAX_SIZE];
int top=-1;
int isemptys()
{return (top<0)?1:0;}
int isfulls()
{return (top>=MAX_SIZE-1)?1:0;}
void adds(int *top,element item)
{
  if(isfulls())cout<<"stack is full.\n";
  else stack[++(*top)]=item;
}
void deletes(int *top,element &item)
{
  if(isemptys())
   cout<<"stack is empty, so you can't delete.\n";
  else
  {item=stack[(*top)];(*top)--;}
}
void main()
{
  clrscr();
  element i,i1,i2;
  int n,S=0;
  cout<<"input Fib number which you do: ");cin>>n;
```

```
  i.key=n;
  adds(&top,i);
  while(!isemptys())
   {
    deletes(&top,i);
    if(i.key==1||i.key==2)S++;
    else
     {
      i1.key=i.key-1;adds(&top,i1);
      i2.key=i.key-2;adds(&top,i2);
     }
    }
   cout<<"\nFib number in”<<n<<”th position is "<<S;
   getch();
  }
```

- ***Ví dụ 2:*** Xây dựng bài toán tháp Hà nội bằng cách không sử dụng đệ qui.

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MAX_SIZE 100
struct element
 {
  int key;
  char c1,c2,c3;
 };
element stack[MAX_SIZE];
int top=-1;
int isempty()
 {return (top<0)?1:0;}
int isfull()
 {return (top>=MAX_SIZE-1)?1:0;}
void adds(int *top,element item)
 {
  if(isfull())cout<<"stack is full.\n";
  else stack[++(*top)]=item;
 }
void deletes(int *top,element &item)
 {
  if(isempty())cout<<"stack is empty, so you can't delete.\n";
  else
   {item=stack[(*top)];(*top)--;}
 }
void main()
 {
  clrscr();
  element i,i1,i2,i3;
  int n;
```

```
cout<<"HANOI TOWER PROGRAM WAS DESIGNED BY …\n\n";
cout<<"input the number of plate:";cin>>n;
i.key=n;i.c1='a';i.c2='b';i.c3='c';
adds(&top,i);
while(!isempty())
 {
  deletes(&top,i);
  if(i.key==1)cout<<i.c1<<"->"<<i.c3<<"\n";
  else
   {
    i1.key=i.key-1;
    i1.c1=i.c2;
    i1.c2=i.c1;
    i1.c3=i.c3;
    adds(&top,i1);
    i2.key=1;
    i2.c1=i.c1;
    i2.c2=i.c2;
    i2.c3=i.c3;
    adds(&top,i2);
    i3.key=i.key-1;
    i3.c1=i.c1;
    i3.c2=i.c3;
    i3.c3=i.c2;
    adds(&top,i3);
   }
 }
getch();
}
```

3.2. Kiểu cấu trúc dữ liệu trừu tượng queue (The Queue Abstract Data Type)

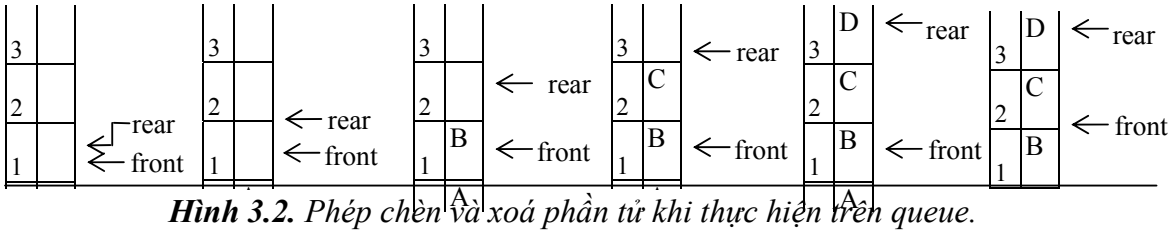
3.2.1. Định nghĩa queue

Queue là một danh sách có thứ tự mà tất cả phép toán thêm vào được thực hiện ở đầu cuối (rear) và phép toán loại bỏ được thực hiện ở đầu ngược lại (rear).

☞ Dĩ nhiên khi queue còn chỗ trống thì ta mới thêm phần tử vào được, còn muốn loại bỏ một phần tử ra khỏi queue thì queue phải tồn tại ít nhất một phần tử trong queue.

- ***Ví dụ:*** Với queue Q = (a_0, a_1,…,a_{n-1}) được cho, a_0 là phần tử đằng trước (front element), a_{n-1} là phần tử đằng sau (rear element), a_{i+1} là phần tử đằng sau a_i, $0 \le i < n-1$.

Hạn chế trên queue đó là khi ta thêm lần lượt các phần tử A, B, C, D vào queue thì A là phần tử đầu tiên bị xoá ra khỏi queue. Hình 3.2 minh hoạ lần lượt các động tác này.



Hình 3.2. *Phép chèn và xoá phần tử khi thực hiện trên queue.*

Do phần tử đầu tiên được chèn cũng là phần tử đầu tiên bị xoá ra khỏi queue. Do vậy queue còn được gọi là danh sách **vào trước ra trước**, hay còn gọi là danh sách **FIFO** (First In First Out).

3.2.2. Biểu diễn queue

Biểu diễn của queue khó hơn so với stack, bởi vì phép thêm vào và loại bỏ được thực hiện ở hai đầu khác nhau. Cách đơn giản nhất đó là sử dụng mảng một chiều cộng với hai biến *front* và *rear* tương ứng cho hai vị trí loại bỏ và thêm vào.

Để bắt đầu, ta khởi gán *front* = *rear* = -1.

3.2.3. Các phép toán trên queue

Với cách biểu diễn được cho, ta có thể định nghĩa các phép toán trên queue theo các hàm như sau - *với item có kiểu element, max_size nguyên dương, queue có kiểu Queue:*

1. *Queue* CreateQ(*max_size*) ::= Tạo một queue rỗng có kích thước cực đại *max_size.*

2. *Boolean* IsFullQ(*queue, item*) ::=

    if(số phần tử trong *queue* == *max_size*) **return** TRUE

    else return FALSE

3. *Queue* AddQ(*queue, item*) ::=

    if(IsFullQ(*queue*)) queue_full

    else chèn item tại vị trí rear của *queue* và **return** *queue*

4. *Boolean* IsEmpty(*queue*) ::=

    if(*queue* == CreateQ(*max_size*)) **return** TRUE

    else return FALSE

5. *Element* DeleteQ(*queue*) ::=

    if(IsEmpty(*queue*)) **return**

    else xoá và **return** (item tại vị trí front của queue)

➢ *Giải thuật của hàm Queue CreateQ(max_size)*

```
#define MAX_SIZE 100
struct element
{
       int key;
      /* khai báo các trường khác */
};
element queue[MAX_SIZE];
int rear = -1, front = -1;
```

➢ *Giải thuật của hàm Boolean IsEmpty(queue)*

```
int isemptyq()
{return (front == rear)?1:0;}
```

➢ *Thuật toán của hàm Boolean IsFullQ(queue, item)*

```
int isfullq()
{return (rear == MAX_SIZE - 1)?1:0;}
```

➢ *Giải thuật của hàm Queue AddQ(queue, item)*

```
void addq(int *rear, element item)
{
  if(*rear == MAX_SIZE - 1) queue_full();
  else queue[++*rear] = item;
}
```

➢ *Giải thuật của hàm Element DeleteQ(queue)*

```
void deleteq(int *front, int rear, element &item)
{ if(*front == rear)  cout<<"queue is empty\n";
  else  {item = queue[*front];++*front; } }
```

🕮 **Nhận xét:**

Các hàm *addq* và *deleteq* của quêu có cấu trúc tương tự các hàm *add* và *delete* của stack. Tuy thế, trong lúc stack sử dụng biến *top* trong cả hai hàm *add* và *delete*, thì queue lại sử dụng biến *rear* trong hàm *addq* và *front* trong hàm *deleteq*.

Cách gọi của hàm *addq* và *deleteq* là: *addq(&rear, item)* và *delete(&front, rear, item)*, trong đó *item* là biến có kiểu *element*.

- ***Ví dụ [sắp lịch công việc]:*** Queue thường được sử dụng trong các chương trình máy tính, và một ví dụ tiêu biểu đó là tạo một queue công việc bằng hệ điều hành. Nếu hệ điều hành không áp đặt quyền ưu tiên lên các công việc thì các công việc này sẽ được xử lý theo trình tự mà nó được nhập vào hệ thống. Hình 3.3. minh họa cách thức một hệ điều hành xử lý các công việc khi nó đóng vai trò là một queue tuần tự.

front	*rear*	Q[0]	Q[1]	Q[2]	Q[3]	Comments
-1	-1					Queue is empty
-1	0	J1				Job 1 is added
-1	1	J1	J2			Job 2 is added
-1	2	J1	J2	J3		Job 3 is added
0	2		J2	J3		Job 1 is deleted
1	2			J3		Job 2 is deleted

Hình 3.3. *Phép chèn và loại bỏ trên một queue tuần tự (sequential queue).*

Hiển nhiên khi ta nói rằng các công việc trong ví dụ trên lần lượt được nhập vào và rời khỏi hệ thống, và queue dần dần di chuyển về phía bên phải. Điều này có nghĩa là cuối cùng chỉ mục của *rear* bằng với MAX_SIZE – 1, và ta bảo queue bị đầy. Trong trường hợp này hàm *queue_full* sẽ di chuyển toàn bộ các phần tử trên queue về bên trái, và phần tử đầu tiên lại được bắt đầu từ *queue*[0] và *front* được bắt đầu tại –1; bên cạnh đó cũng phải tính lại giá trị của *rear* sao cho vị trí của nó được chính xác trong queue. Việc di chuyển mảng queue này tốn rất nhiều thời gian, bởi vì thông thường thì có rất nhiều phần tử trên mảng này. Thực ra, *queue_full* có độ phức tạp trong trường hợp xấu nhất là O(MAX_SIZE).

➢ *Giải thuật queue_full()*

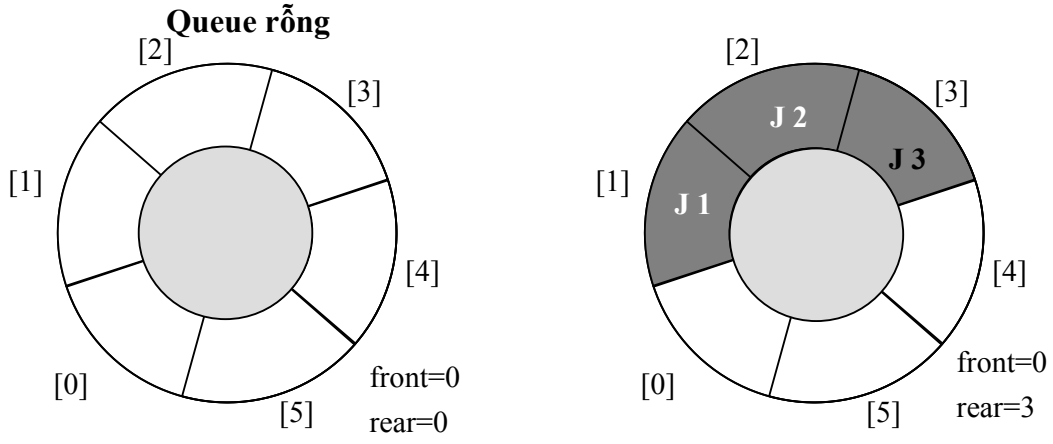
```
void queue_full()
{
  for(int i=front+1; i<=rear;i++)
       queue[i-front-1]=queue[i];
  rear = rear - front -1; front = -1;
}
```

3.2.4.Queue vòng (Circular Queue)

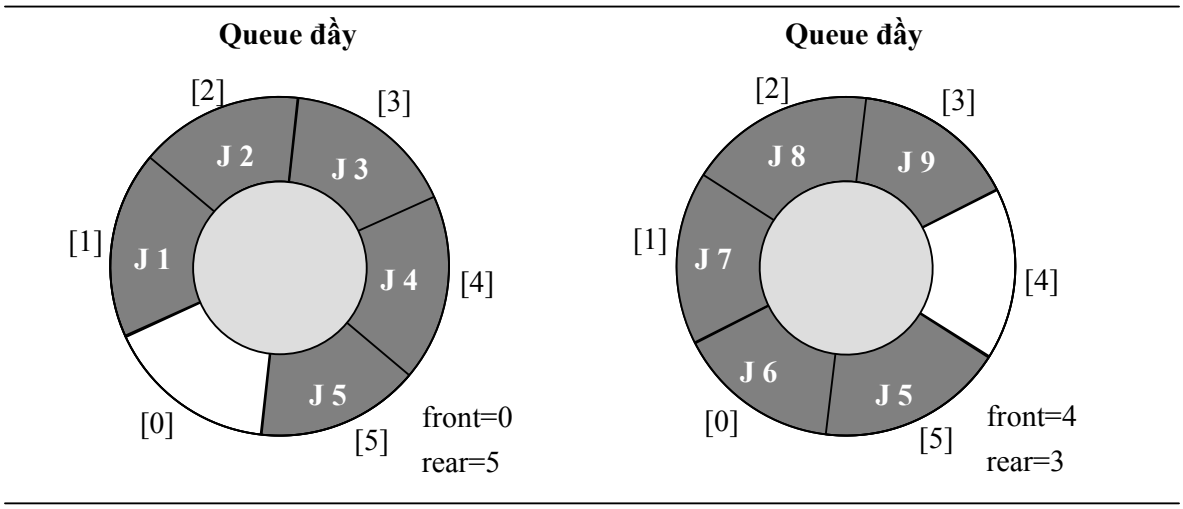
Mảng *queue*[MAX_SIZE] tuần tự tỏ ra hiệu quả hơn khi nó trở thành mảng queue vòng. Đối với dạng queue này, ta khởi tạo *front* và *rear* cùng bằng 0 (hoặc bằng -1).

Giá trị của *front* chính là vị trí của phần tử đầu tiên trên queue nhưng lệch một vị trí so với chiều ngược chiều kim đồng hồ. Giá trị *rear* chính là vị trí của phần tử cuối trong queue hiện hành. Một queue rỗng khi và chỉ khi *front = rear*.

Hình 3.4 trình bày queue vòng rỗng và không rỗng với MAX_SIZE = 6; hình 3.5 minh hoạ hai queue đầy (full queue) trong trường hợp MAX_SIZE = 6.



Hình 3.4. *Các queue vòng rỗng và không rỗng*



Hình 3.5. *Các queue vòng đầy.*

Giả sử trong các queue vòng này có không quá một ô trống, và phép thêm một phần tử vào queue này sẽ dẫn đến kết quả *front = rear* và như vậy ta sẽ không phân biệt được queue trống với queue đầy.

Vì vậy, ta qui ước rằng kích cỡ của queue vòng là MAX_SIZE và cho phép queue vòng này chứa tối đa MAX_SIZE –1 phần tử.

Việc thực hiện hàm addq và *deleteq* cho queue vòng có khó hơn một chút, bởi vì ta phải bảo đảm rằng luôn tồn tại sự thay đổi luân phiên trong vòng. Điều này dễ dàng có được bằng cách sử dụng phép toán chia lấy phần dư (modulus operator).

Sự thay đổi luân phiên của giá trị *rear* trong hàm *addq* của vòng quay được biểu diễn dưới dạng:

*rear = (*rear + 1) % MAX_SIZE;

Lưu ý rằng ta cần phải thay đổi luân phiên giá trị *rear* trước khi đặt *item* vào *queue*[*rear*]. Tương tự, trong hàm *deleteq* sự thay đổi luân phiên của giá trị *front* được biểu diễn dưới dạng:

*front = (*front + 1) % MAX_SIZE;

và nhờ vậy ta xoá được phần tử *item*.

➢ *Giải thuật thêm một phần tử vào queue vòng*

```
void addq(int front, int *rear, element item)
{
  *rear = (*rear + 1) % MAX_SIZE;
  if(front == *rear) cout<<"queue is full.\n";
  else queue[*rear] = item;
}
```

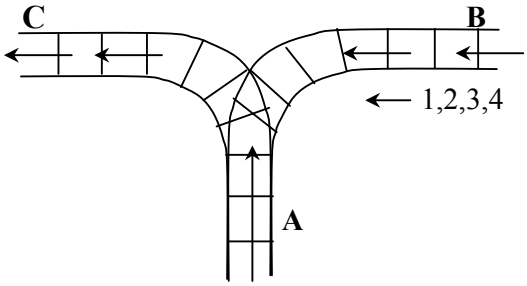
➢ *Giải thuật xoá một phần tử ra khỏi queue vòng*

```
element deleteq(int *front, int rear)
{
  if(*front == rear)
```

```
    {cout<<"queue is empty.\n"; exit(1); }
    *front = (*front + 1) % MAX_SIZE;
    return queue[*front];
}
```

Bài tập cuối chương

1. Cho mạng chuyển đổi đường sắt như sau:



Hình 3.6. *Sơ đường sắt*

Các toa tầu được đánh số thứ tự từ 1 đến 4 và nằm phía trên đường ray B. Người ta muốn chuyển các toa tầu từ đường ray B sang đường ray C nhờ đường ray phụ A.

a. Việc chuyển đổi các toa tầu từ đường ray B sang đường ray C qua trung gian đường ray A ứng với dạng cấu trúc dữ liệu nào mà anh/chị đã học.

b. Hãy liệt kê các trường hợp hoán vị của các toa tầu có thể thu được trên ray C nhờ dạng cấu trúc dữ liệu trong câu 3a.

c. Hãy nêu ý tưởng trình bày thuật toán này.

2. Cho một stack S. Hãy viết chương trình con thực hiện các công việc sau:

a. Đếm số phần tử của stack S

b. Xuất nội dung phần tử thứ n của stack S

c. Xuất nội dung của stack S

d. Loại phần tử thứ n của stack S

Trong các chương trình con trên yêu cầu bảo toàn thứ tự các phần tử của stack S

3. Cũng với nội dung câu 1 nhưng với kiểu dữ liệu Queue

4. Viết chương trình con đảo ngược một stack

5. Viết chương trình con đảo ngược một queue

6. Dùng stack và queue để kiểm tra một chuỗi ký tự có đối xứng không

7. Hãy cài đặt một ngăn xếp bằng cách dùng con trỏ.

8. Dùng ngăn xếp để viết chương trình con đổi một số thập phân sang số nhị phân.

Chương 4. **CÂY**

(Trees)

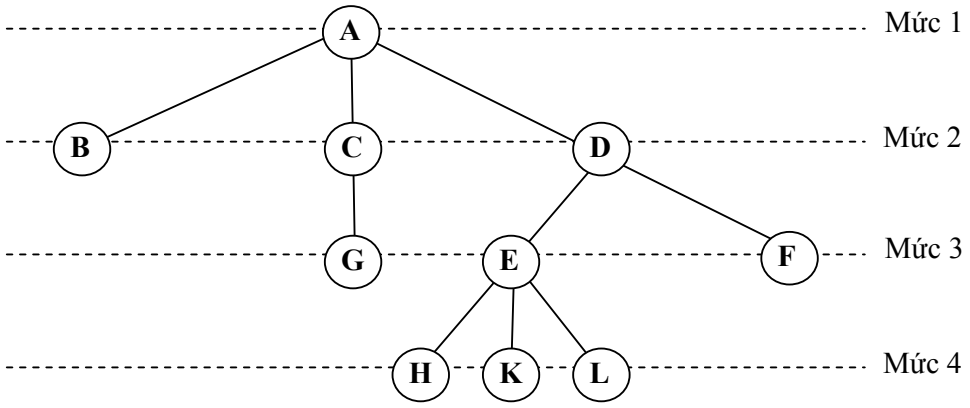
Cây là một cấu trúc rất quan trọng được sử dụng nhiều trong các giải thuật. Trong chương này, ta sẽ tìm hiểu các khái niệm cơ bản về cây, một số phép toán quan trọng trên cây, biểu diễn cây trong máy tính. Cây có ứng dụng nhiều trong đời sống hàng ngày, chẳng hạn cây gia phả, cây toán học,…

4.1. Một số khái niệm

4.1.1. Một số khái niệm

4.1.1.1. Định nghĩa cây

Cây là tập hợp rỗng hoặc gồm nhiều phần tử gọi là nút, trong đó:



Hình 4.1. *Cây và các nút*

4.1.1.2. Bậc của nút và bậc của cây

Bậc của nút là số cây con của nút đó.

Bậc của cây là bậc lớn nhất của các nút của cây. Nếu cây có bậc là n thì gọi là cây n-phân.

- ***Ví dụ:*** Bậc của nút A là 3, của nút B là 0, của nút D là 2. Bậc của cây là 3, ta có cây tam phân.

4.1.1.3. Nút kết thúc và nút trung gian

Nút kết thúc (hoặc nút lá – leaf) là nút có bậc bằng 0.

Nút trung gian là nút có bậc khác 0 và không phải là nút gốc.

- ***Ví dụ:*** Các nút lá: B, G, H, K, L, F. Các nút trung gian: C, D, E.

4.1.1.4. Mức (level) của nút và chiều cao của cây

Mức của nút gốc bằng 1, mức các nút khác gốc bằng mức của nút gốc của cây con nhỏ nhất chứa nó cộng 1.

Chiều cao của cây là mức lớn nhất của các nút lá.

- ***Ví dụ:*** Các nút B, C, D có mức là 2; G, E, F có mức là 3. Chiều cao của cây là 4.

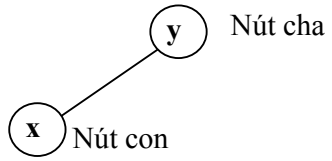
4.1.1.5. Nút trước và nút sau

Nút y được gọi là trước nút x nếu cây con có gốc là y chứa nút x, và khi đó nút x được gọi là nút sau của y.

- ***Ví dụ:*** D là nút trước của H, và H là nút sau của D.

4.1.1.6. Nút cha và nút con (ancestor and descendant)

Nếu y là nút trước của x, và mức của x bằng mức của y cộng 1, thì nút y gọi là cha của nút x, và x được gọi là con của y.

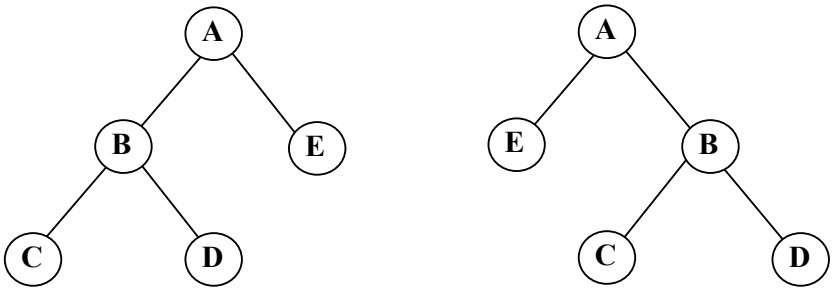


Hình 4.2. *Nút cha, nút con*

- ***Ví dụ:*** E là nút cha của H, và H là nút con của E; còn D không phải là cha nút H.

4.1.1.7. Cây có thứ tự (ordered tree)

Một cây được gọi là có thứ tự nếu ta thay đổi vị trí của các cây con thì ta sẽ có một cây mới. Như vậy khi ta đổi nút con bên trái với nút con bên phải thì ta sẽ được một cây mới.



Hình 4.3. *Cây có thứ tự*

4.1.1.8. Chiều dài đường đi (path length)

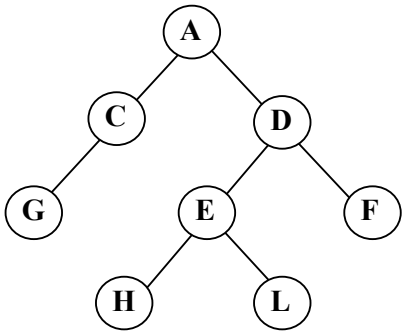
i. Chiều dài đường đi của nút:

Số các nhánh hay cạnh đi qua từ nút gốc đến một nút x gọi là chiều dài đường đi của nút x. Nút gốc có chiều dài đường đi là 1, các nút con trực tiếp của nó có chiều dài đường đi là 2, … Một cách tổng quát, một nút tại mức i có chiều dài đường đi là i.

ii. Chiều dài đường đi của cây:

Chiều dài đường đi của một cây được định nghĩa là tổng của các chiều dài đường đi của tất cả các nút của cây và gọi là chiều dài đường đi bên trong của cây.

- ***Ví dụ:*** Chiều dài đường đi bên trong của cây sau là 22.



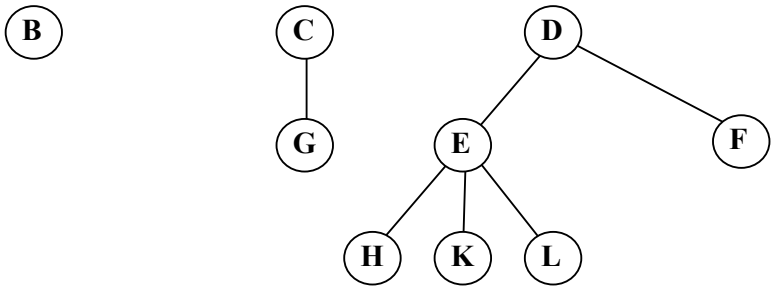
Hình 4.4. *Chiều dài đường đi trong cây là 22*

Chiều dài đường đi trung bình bên trong cây = (Chiều dài đường đi bên trong cây)/(Tổng số nút trên cây).

4.1.1.9. Rừng (Forest)

Rừng là tập hợp các cây.

- ***Ví dụ:***

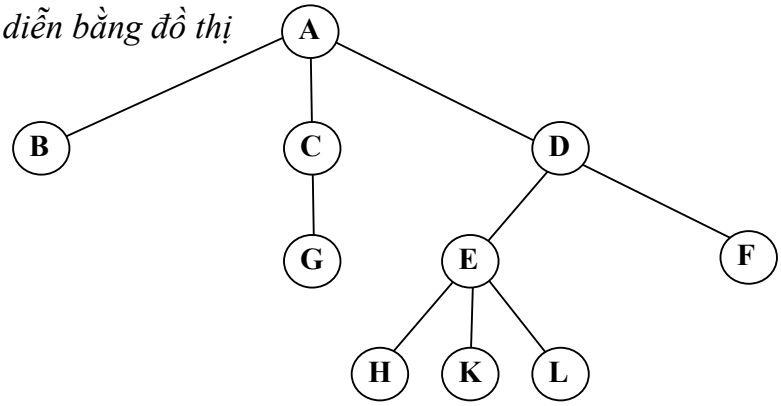


Hình 4.5. *Rừng và 3 cây con*

4.1.2. Các cách biểu diễn cây

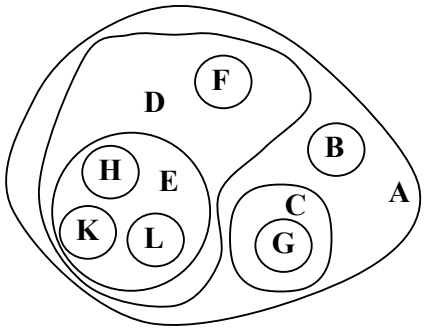
Để biểu diễn một cây, ta có nhiều cách khác nhau: biểu diễn bằng đồ thị, bằng giản đồ, bằng các dấu ngoặc lồng nhau, bằng chỉ số, …

4.1.2.1. Biểu diễn bằng đồ thị



Hình 4.6. *Biểu diễn cây bằng đồ thị*

4.1.2.2. Biểu diễn bằng giản đồ Ven



Hình 4.7. *Biểu diễn cây bằng giản đồ Ven*

4.1.2.3. Biểu diễn bằng các dấu ngoặc lồng nhau

(A(B)(C(G))(D(E(H)(K)(L))(F)))

4.1.2.4. Biểu diễn bằng phương pháp thụt dòng

--A

----B

----C

------G

----D

------E

--------H

--------K

--------L

------F

4.1.2.5. Biểu diễn bằng chỉ số

1.A

1.1.B

1.2.C

1.2.1.G

1.3.D

1.3.1.E

1.3.1.1.H

1.3.1.2.K

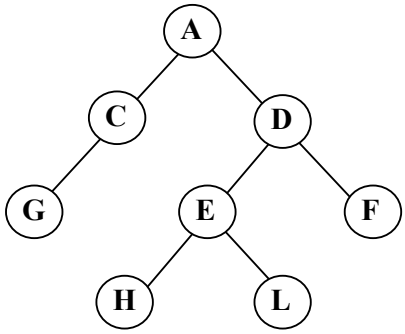
1.3.1.3.L

1.3.2.F

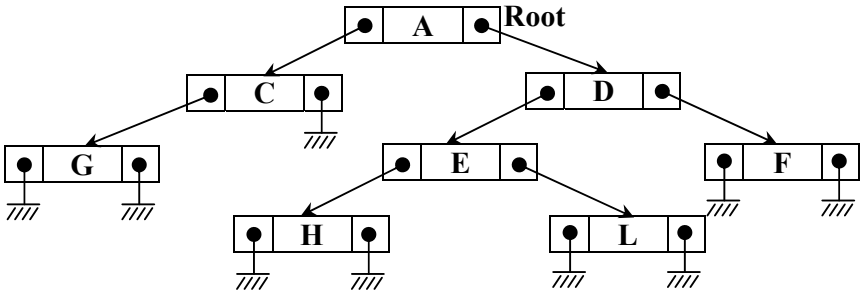
4.2. Cách chứa cây trong bộ nhớ

Ta có thể chứa cây trong bộ nhớ bằng cách sử dụng danh sách liên kết. Trong trường hợp tổng quát, ta sử dụng danh sách n-liên kết để chứa cây n-phân.

- ***Ví dụ:*** Cây nhị phân

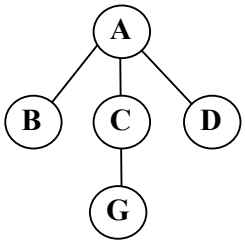


Được lưu trong bộ nhớ

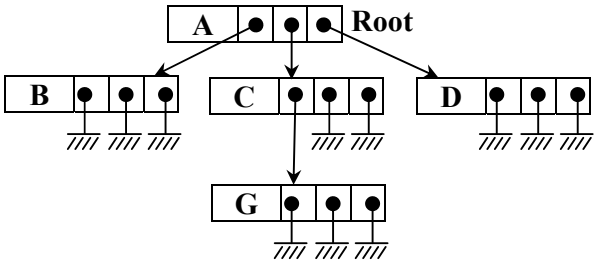


Hình 4.8. *Cây nhị phân lưu trữ trong bộ nhớ*

- ***Ví dụ:*** Cây tam phân



Được lưu trong bộ nhớ



Hình 4.9. *Cây tam phân lưu trữ trong bộ nhớ*

Cây nhị phân (Binary Tree) là cây cơ bản và được ứng dụng nhiều nhất nên trong chương này ta sẽ nghiên cứu các phép toán trên cây nhị phân.

4.3. Cây nhị phân (Binary Tree)

Cây nhị phân là cây bậc hai, tại mỗi nút của cây có nhiều nhất là hai cây con

Để biểu diễn cây nhị phân trong bộ nhớ, ta dùng nút danh sách có hai vùng liên kết: một vùng Left chỉ đến nút con bên trái, một vùng Right chỉ đến nút con bên phải, và vùng chứa nội dung (information). Một chỉ điểm Root chỉ vào nút gốc của cây.

Left	Info	Right

Hình 4.10. *Biểu diễn nút của cây nhị phân*

☞ ***Khai báo cây nhị phân:***

```
struct NodeTree
{
   …..                          //Khai báo các trường nội dung
   NodeTree *Left, *Right; //Khai báo trường liên kết
};
NodeTree *Root;
```

- ***Ví dụ:*** Khai báo cây nhị phân có chỉ điểm đầu Root, các nút (phần tử trong cây) có trường nội dung kiểu nguyên.

```
struct NodeTree
{
  int nd;            //Khai báo các trường nội dung
  NodeTree *Left, *Right; //Khai báo trường liên kết
};
NodeTree *Root;
```

✍ ***Ghi chú:*** Để đơn giản, ta sử dụng cách khai báo trong ví dụ này để cài đặt các phép toán trên cây nhị phân.

4.3.1. Khởi tạo cây

Khi khởi tạo, cây nhị phân là rỗng, ta cho Root trỏ đến NULL.

➢ *Giải thuật:*

```
void Initialize()
{ Root = NULL;}
```

4.3.2. Các phép duyệt cây nhị phân

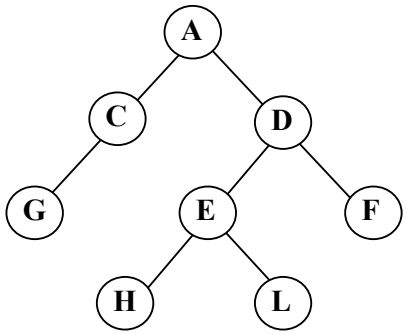
Phép duyệt cây là lần lượt đi qua tất cả các nút của cây và mỗi nút chỉ được duyệt một lần. Có 6 phép duyệt cây dựa vào thứ tự duyệt của các nút bên trái (L), nút gốc (N), các nút bên phải (R) là:

+ Duyệt theo thứ tự trước (Preorder): NLR, NRL

+ Duyệt theo thứ tự giữa (Inorder): LNR, RNL

+ Duyệt theo thứ tự sau (Postorder): LRN, RLN

- ***Ví dụ:*** Duyệt cây nhị phân sau.



Hình 4.11. *Cây nhị phân với các phép duyệt*

Thứ tự của 6 phép duyệt là

NLR: ACGDEHLF NRL:ADFELHCG

LRN: GCHLEFDA RLN: FLHEDGCA

LNR: GCAHELDF RNL: FDLEHACG

4.3.2.1. Duyệt theo thứ tự trước (NLR)

➢ *Giải thuật:*

```
void PrintNLR(NodeTree *p)
{
  if(p)
  {
   cout<<p->nd<<"  ";
   PrintNLR(p->Left);
   PrintNLR(p->Right);
  }
}
```

4.3.2.2. Duyệt theo thứ tự giữa (LNR)

➢ *Giải thuật:*

```
void PrintLNR(NodeTree *p)
{
  if(p)
  {
   PrintLNR(p->Left);
   cout<<p->nd<<"  ";
   PrintLNR(p->Right);
  }
}
```

4.3.2.3. Duyệt theo thứ tự sau (LRN)

➢ *Giải thuật:*

```
void PrintLRN(NodeTree *p)
{
  if(p)
  {
```

```
    PrintLRN(p->Left);
    PrintLRN(p->Right);
    cout<<p->nd<<"  ";
   }
}
```

4.3.3. Tạo cây nhị phân bất kỳ

➢ *Giải thuật:*

```
NodeTree *Init(int x, NodeTree *T, NodeTree *P)
{
  NodeTree *p=new NodeTree;
  p->nd=x; p->Left=T; p->Right=P;
  return p;
}
```

4.3.4. Tìm kiếm nút chứa giá trị x trên cây nhị phân

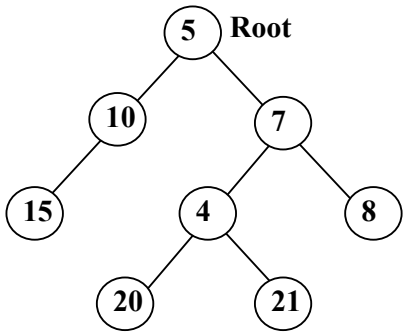
➢ *Giải thuật:*

```
NodeTree *FindX(int x, NodeTree *Root)
{
 if(Root)
  if(Root->nd==x)return Root;
  else
  {
   NodeTree *q;
   q=FindX(x,Root->Left);
   if(!q)q=FindX(x,Root->Right);
   return q;
  }
 else return NULL;
}
```

4.3.5. Ví dụ về cây nhị phân

❑ **Bài toán:** Khai báo cây nhị phân có chỉ điểm đầu Root, các nút (phần tử trong cây) có trường nội dung kiểu nguyên. Sau đó hãy thực hiện các công việc:

i. Tạo cây như hình sau.



Hình 4.12. *Cây nhị phân cần tạo*

ii. In ra màn hình các giá trị có trong cây theo thứ tự RLN.

iii. Tìm xem trong cây có nút nào chứa nội dung là x không?

❑ **Thực hiện:**

```
#include<iostream.h>
#include<conio.h>
struct NodeTree
{
 int nd;
 NodeTree *Left,*Right;
};
NodeTree *Root;
NodeTree *Init(int x,NodeTree *T, NodeTree *P)
{
 NodeTree *p=new NodeTree; p->nd=x;
 p->Left=T; p->Right=P;
 return p;
}
void PrintRLN(NodeTree *p)
{
  if(p)
  {
   PrintRLN(p->Right);
   PrintRLN(p->Left);
   cout<<p->nd<<"  ";
  }
}
NodeTree *FindX(int x, NodeTree *Root)
{
  if(Root)
       if(Root->nd==x)return Root;
       else
       {
            NodeTree *q;
            q=FindX(x,Root->Left);
            if(!q)q=FindX(x,Root->Right);
            return q;
       }
  else return NULL;
}
void main()
{
   clrscr();
   NodeTree *p, *q;
   p=Init(10,Init(15,NULL,NULL),NULL));
   q=Init(7,Init(4,Init(20,NULL,NULL),Init(21,NULL,NULL)),Init(
   8,NULL,NULL));
   Root=Init(5, p, q);
```

```
    cout<<"Duyet cay theo thu tu RLN:";
    PrintRLN(Root);
    int x;
    cout<<"\nNhap gia tri can tim: "; cin>>x;
    NodeTree *q=FindX(x,Root);
    if(q) cout<<"\nCay co ptu chua gia tri "<<x<<";
    else
       cout<<"\nCay khong co ptu chua gia tri "<<x<<";
    getch();
}
```

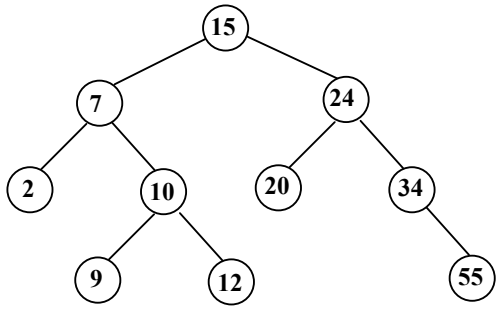
4.4. Cây nhị phân tìm kiếm (cây-BST: Binary Search Tree)

4.4.1. Định nghĩa

Cây nhị phân tìm kiếm (Cây-BST) là cây nhị phân hoặc rỗng hoặc thoả mãn đồng thời các điều kiện sau:

- Khoá của các đỉnh thuộc cây con trái nhỏ hơn khoá nút gốc
- Khoá của nút gốc nhỏ hơn khoá của các đỉnh thuộc cây con phải của của gốc
- Cây con trái và cây con phải của gốc cũng là cây nhị phân tìm kiếm

❑ ***Ví dụ:*** Hình 4.13 biểu diễn cây-BST, trong đó khoá của các đỉnh là các số nguyên.



Hình 4.13. *Cây BST*

4.4.2. Cài đặt cây nhị phân tìm kiếm

Mỗi nút trên cây nhị phân tìm kiếm có dạng như hình 4.14

Left	Info	Right

Hình 4.14. *Biểu diễn nút của cây BST*

Trong đó:

Left: trường liên kết chỉ đến cây con trái.

Right: trường liên kết chỉ đến cây con phải

Info : chứa thông tin của nút

☞ ***Khai báo cây-BST:***

```
struct NodeTree
{
   …..                          //Khai báo các trường nội dung
   int nd;                      //Khai báo khóa
   NodeTree *Left, *Right; //Khai báo trường liên kết
};
NodeTree *Root;
```

- ***Ví dụ:*** Khai báo cây-BST có chỉ điểm đầu Root, các nút (phần tử trong cây-BST) có trường nội dung kiểu nguyên.

```
struct NodeTree
{
 int nd;                //Khai báo trường nd làm khóa
 NodeTree *Left,*Right;//Khai báo trường liên kết
};
NodeTree *Root;
```

✍ ***Ghi chú:*** Để đơn giản, ta sử dụng cách khai báo trong ví dụ này để cài đặt các phép toán trên cây-BST.

4.4.3. Các thao tác cơ bản trên cây nhị phân tìm kiếm

4.4.3.1. Khởi tạo cây

Khi khởi tạo, cây-BST là rỗng, ta cho Root trỏ đến NULL.

➢ *Giải thuật:*

```
void Initialize()
{Root=NULL;}
```

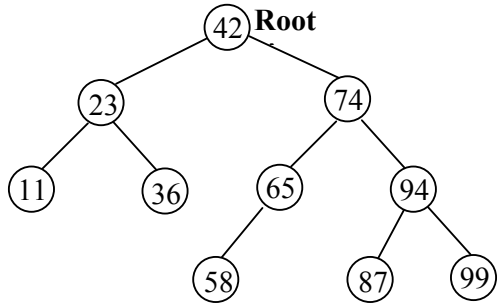
4.4.3.2. Chèn một nút vào cây BST

Việc thêm một một nút có nội dung bằng x vào cây-BST phải đảm bảo điều kiện ràng buộc của cây-BST. Ta có thể thêm vào nhiều chỗ khác nhau trên cây, nhưng trong giải thuật này ta chỉ thêm vào ở nút lá.

➢ *Giải thuật:*

```
void InsertBST(int x,NodeTree *&Root)
{
 if(!Root)
 {Root=new NodeTree;Root->nd=x;
  Root->Left=Root->Right=NULL;}
 else
  if(Root->nd<x)InsertBST(x,Root->Right);
  else
   if(Root->nd>x)InsertBST(x,Root->Left);
   else cout<<x<<" da ton tai trong cay.\n";
}
```

- ***Ví dụ:*** Khi thêm lần lượt các giá trị x sau: 42 23 74 11 65 58 94 36 99 87 vào cây-BST, ta được cây-BST như trong hình 4.15.



Hình 4.15. *Cây BST với các giá trị lần lượt thêm vào*

4.4.3.3. Loại bỏ nút trên cây nhị phân tìm kiếm

Ngược với phép toán chèn vào là phép toán loại bỏ. Chúng ta cần phải loại bỏ khỏi cây-BST một đỉnh có nội dung là x cho trước, sao cho việc huỷ một nút ra khỏi cây cũng phải bảo đảm điều kiện ràng buộc của cây-BST.

Có ba trường hợp khi huỷ một nút x có thể xảy ra:

- x là nút lá
- x là nút nửa lá (chỉ có một con trái hoặc con phải)
- x có đủ hai con (trường hợp tổng quát)

Trong phép toán này ta sử dụng hàm phụ:

- Hàm **DeleteLeft**: Hàm này tìm đến nút con phải nhất của cây-BST con, sau đó xóa nút con này và trả về giá trị của nút con vừa xóa cho hàm.

➢ *Giải thuật:*

```
int DeleteLeft(NodeTree *&Root)
{
 if(Root->Right==NULL)
 {
   NodeTree *p=Root; Root=Root->Left;
   int x=p->nd;delete p; return x;
 }
 else return DeleteLeft(Root->Right);
}
```

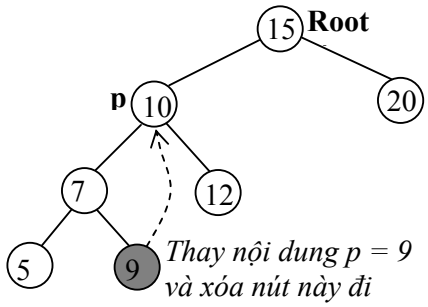
➢ *Giải thuật xóa nút có nội dung là x trên cây-BST:*

```
void DeleteNode(int x,NodeTree *&Root)
{
 if(Root)
   if(Root->nd==x)
     if(Root->Left)
       Root->nd = DeleteLeft(Root->Left);                {1}
     else //(Root->Right ||(!Root->Left && !Root->Right))  {2}
       {NodeTree *p=Root; Root=Root->Right; delete p;}
```

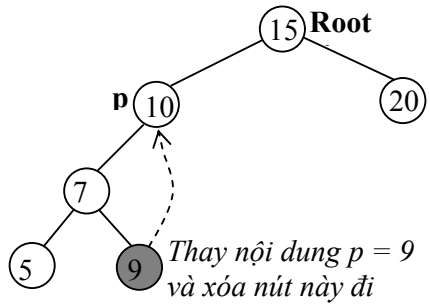
```
        else
          if(Root->nd<x)DeleteNode(x,Root->Right);
          else DeleteNode(x,Root->Left);
      else cout<<x<<" khong ton tai tren cay BST.\n";
    }
```

1 trong 4 trường hợp của cây-BST rơi vào điều kiện {1} của giải thuật là:

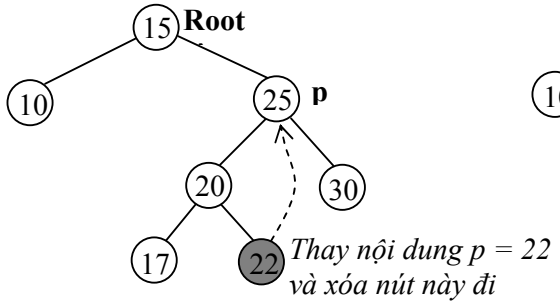
a. Xóa nút có nội dung x=10



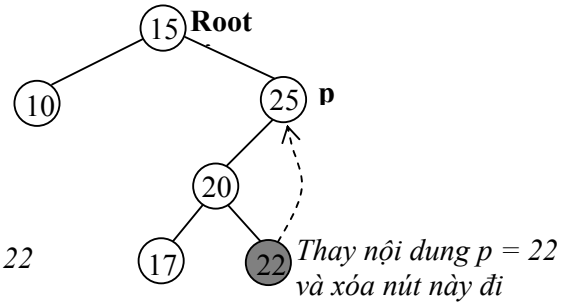
b. Xóa nút có nội dung x=10



c. Xóa nút có nội dung x=22

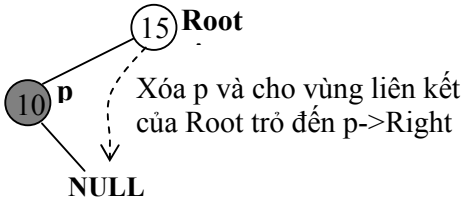


d. Xóa nút có nội dung x=22

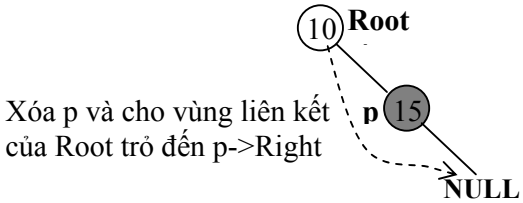


1 trong 4 trường hợp của cây-BST rơi vào điều kiện {2} của giải thuật là:

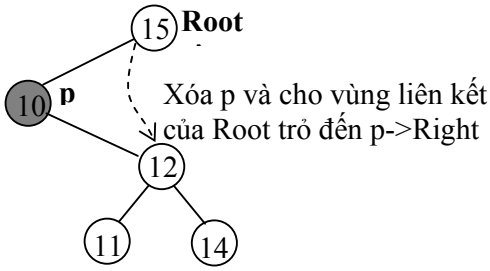
a. Xóa nút có nội dung x=10



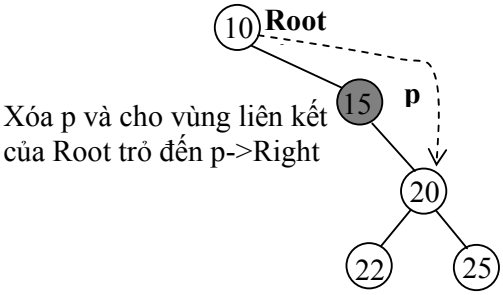
b. Xóa nút có nội dung x=15



c. Xóa nút có nội dung x=10



d. Xóa nút có nội dung x=15



Hình 4.16. *Các trường hợp xóa nút trên cây BST*

4.4.3.4. Ví dụ

❑ **Bài toán:** Khai báo cây-BST có chỉ điểm đầu Root, các nút (phần tử trong cây-BST) có trường nội dung kiểu nguyên. Sau đó hãy thực hiện các công việc:

i. Tạo cây-BST với n nút có giá trị bất kỳ.

ii. In ra màn hình các giá trị có trong cây theo thứ tự NLR.

iii. Xóa nút chứa nội dung là x?

❑ **Thực hiện:**

```
#include<iostream.h>
#include<conio.h>
struct NodeTree
{
 int nd;
 NodeTree *Left,*Right;
};
NodeTree *Root;
void Initialize()
{Root=NULL;}
void InsertBST(int x,NodeTree *&Root)
{
 if(!Root)
 {Root=new NodeTree;Root->nd=x;
  Root->Left=Root->Right=NULL;}
 else
  if(Root->nd<x)InsertBST(x,Root->Right);
  else
   if(Root->nd>x)InsertBST(x,Root->Left);
   else cout<<x<<" da ton tai trong cay.\n";
}
void PrintNLR(NodeTree *&Root)
{
 if(Root)
 {
  cout<<Root->nd<<"  ";
  PrintNLR(Root->Left);
  PrintNLR(Root->Right);
 }
}
int DeleteLeft(NodeTree *&Root)
{
 if(Root->Right==NULL)
 {
   NodeTree *p=Root; Root=Root->Left;
   int x=p->nd;delete p; return x;
```

```
 }
 else return DeleteLeft(Root->Right);
}
void DeleteNode(int x,NodeTree *&Root)
{
 if(Root)
   if(Root->nd==x)
      if(Root->Left)Root->nd = DeleteLeft(Root->Left);
     else
       //(Root->Right ||(!Root->Left && !Root->Right))
     {NodeTree *p=Root; Root=Root->Right; delete p;}
   else
     if(Root->nd<x)DeleteNode(x,Root->Right);
     else DeleteNode(x,Root->Left);
 else cout<<x<<" khong ton tai tren cay BST.\n";
}
void main()
{
 clrscr();
 Initialize();
 int x,n;
 cout<<"Nhap so nut cua cay: ";cin>>n;
 for(int i=1;i<=n;i++)
 {
  cout<<"Nut "<<i<<" co gia tri:";cin>>x;
  InsertBST(x,Root);
 }
 cout<<"Gia tri tren cay duyet theo thu tu NLR:\n";
 PrintLNR(Root);
 cout<<"\nNhap gia tri can xoa:";cin>>x;
 DeleteNode(x,Root);
 cout<<"\nGtri sau khi xoa duyet theo NLR:\n";
 PrintNLR(Root);
 getch();
}
```

4.5. Cây cân bằng (Balanced Tree)

4.5.1. Định nghĩa

Cây cân bằng là cây nhị phân mà tại bất kỳ nút nào trên cây ta luôn tìm được tổng số nút con trái và tổng số nút con phải lệch nhau không quá một đơn vị.

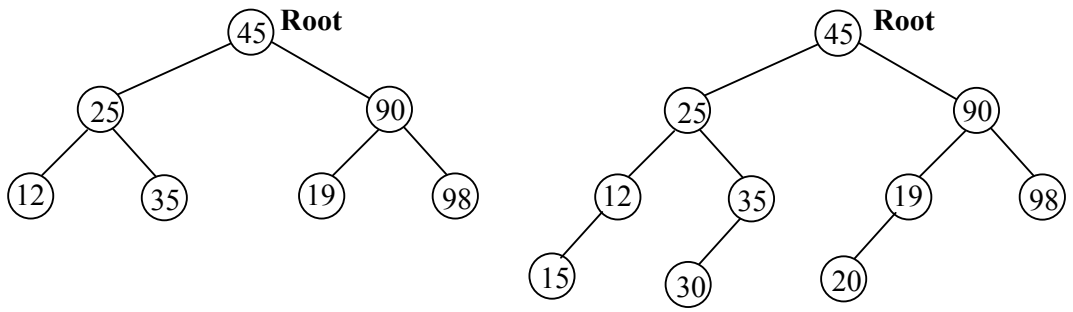
Như vậy, nếu gọi:

- N là nút đang xét.
- N_L là tổng số nút con trái của N.

- N_R là tổng số nút con phải của N.

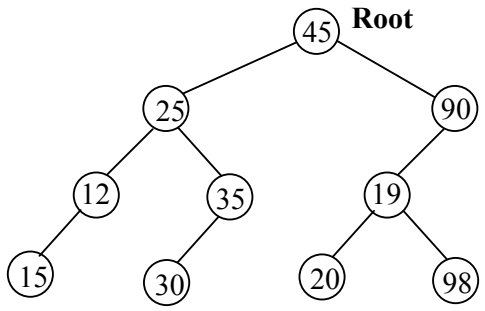
thì: $|N_L - N_R| \leq 1$

- ***Ví dụ:*** Các hình sau biểu diễn cây cân bằng, trong đó khoá của các đỉnh là các số nguyên.



Hình 4.17. *Các cây cân bằng*

- ***Ví dụ:*** Hình sau biểu diễn cây không cân bằng tại nút có khóa bằng 90, trong đó khoá của các đỉnh là các số nguyên.



Hình 4.18. *Cây không cân bằng*

4.5.2. Khai báo và biểu diễn cây cân bằng

Cách khai báo và biểu diễn cây cân bằng tương tự cách khai báo và biểu diễn của cây-BST.

- ***Ví dụ:*** Khai báo cây cân bằng có chỉ điểm đầu Root, các nút (phần tử trong cây) có trường nội dung kiểu nguyên.

```
struct NodeTree
{
  int nd;                  //Khai báo các trường nội dung
  NodeTree *Left, *Right; //Khai báo trường liên kết
};
NodeTree *Root;
```

✍ ***Ghi chú:*** Để đơn giản, ta sử dụng cách khai báo trong ví dụ này để cài đặt các phép toán trên cây cân bằng.

4.5.3. Các phép toán trên cây cân bằng

4.5.3.1. Khởi tạo cây cân bằng

Khi khởi tạo, cây cân bằng là rỗng, ta cho Root trỏ đến NULL.

➢ *Giải thuật:*

```
void Initialize()
{Root=NULL;}
```

4.5.3.2. Chèn một nút vào cây cân bằng

Việc thêm một một nút có nội dung bằng x vào cây cân bằng phải đảm bảo điều kiện ràng buộc của cây cân bằng (tổng số nút con trái và tổng số nút con phải lệch nhau không quá một đơn vị). Ta có thể thêm vào nhiều chỗ khác nhau trên cây, nhưng trong giải thuật này ta chỉ thêm vào ở nút lá.

Để thực hiện được phép toán này, ta phải sử dụng hàm phụ ***CountNode***; hàm này nhằm đếm số nút trên cây nhị phân.

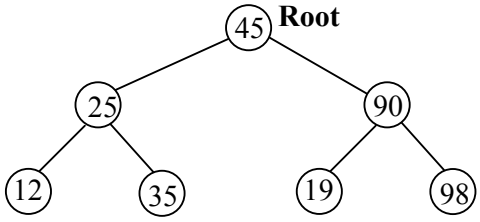
➢ *Giải thuật:*

```
int CountNode(NodeTree *p)
{
 if(p)
  return 1+CountNode(p->Left)+CountNode(p->Right);
 else return 0;
}
```

➢ *Giải thuật chèn nút có giá trị x vào cây cân bằng:*

```
void InsertEle(int x,NodeTree *&Root)
{
 if(!Root)
 {
  Root=new NodeTree;Root->nd=x;
  Root->Left=Root->Right=NULL;
 }
 else
  if(CountNode(Root->Left) > CountNode(Root->Right))
      InsertEle(x,Root->Right);
  else InsertEle(x,Root->Left);
}
```

❑ ***Ví dụ:*** Khi thêm lần lượt các giá trị x sau: 45 25 90 12 19 35 98, ta thu được cây cân bằng sau



Hình 4.19. *Khi thêm lần lượt các giá trị x vào cây cân bằng*

4.5.3.3. Tìm kiếm trên cây cân bằng

Tìm trên cây cân bằng xem có nút nào có nội dung bằng x không?

➢ *Giải thuật:*

```
NodeTree *FindX(int x, NodeTree *Root)
{
 if(Root)
  if(Root->nd==x)return Root;
  else
  {
   NodeTree *q;
   q=FindX(x,Root->Left);
   if(!q)q=FindX(x,Root->Right);
   return q;
  }
 else return NULL;
}
```

Trong giải thuật này, nếu tìm thấy trên cây nút có nội dung bằng x thì trả về địa chỉ nút đó, ngược lại trả về NULL.

4.5.3.4. Xóa nút trên cây cân bằng

Hãy tìm trên cây cân bằng xem có nút nào có nội dung bằng x không? Nếu có thì hãy xóa nút đó. Yêu cầu cây sau khi xóa vẫn cân bằng.

Để thực hiện được phép toán này, ta phải sử dụng hàm phụ ***DeleteNode***; hàm này nhằm xóa nút nằm trên nhánh có nhiều nút nhất của cây cân bằng, sau đó trả về nội dung của nút vừa xóa.

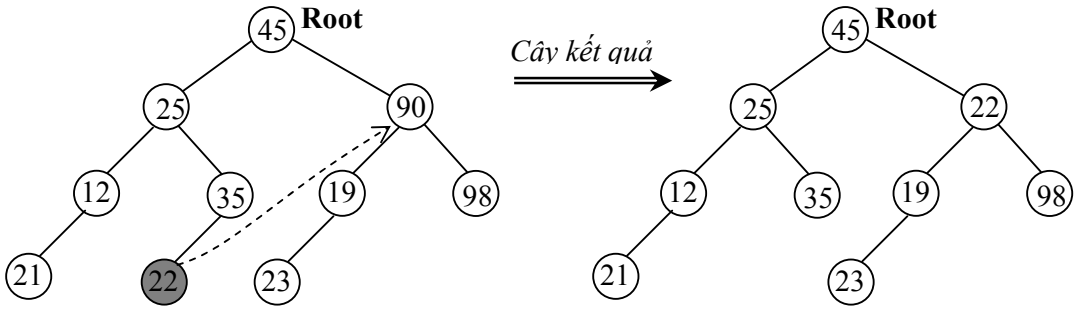
➢ *Giải thuật:*

```
int DeleteNode(NodeTree *&Root)
{
  if(!Root->Left && !Root->Right)
  {    NodeTree *p=Root;Root=NULL;
       int x=p->nd;delete p; return x;}
  else
   if(CountNode(Root->Left) > CountNode(Root->Right))
      return DeleteNode(Root->Left);
   else return DeleteNode(Root->Right);
}
```

➢ *Giải thuật xóa nút có giá trị x khỏi cây cân bằng:*

```
void DeleteX(int x,NodeTree *&Root)
{
 if(Root)
 {NodeTree *p=FindX(x,Root);p->nd=DeleteNode(Root);}
 else cout<<"\nCay khong chua ptu co gia tri "<<x;
}
```

❑ ***Ví dụ:*** Sau khi xóa nút có giá trị x=90 trong cây trong hình 4.20



Ghi chú: *nút bôi đen bị xóa và giá trị được chuyển đến cho nút có giá trị 90*

Hình 4.20. *Cây cân bằng trước và sau khi xóa nút*

4.5.3.5. Ví dụ

❑ **Bài toán:** Khai báo cây cân bằng có chỉ điểm đầu Root, các nút (phần tử trong cây cân bằng) có trường nội dung kiểu nguyên. Sau đó hãy thực hiện các công việc:

i. Tạo cây cân bằng với n nút có giá trị bất kỳ.

ii. In ra màn hình các giá trị có trong cây theo thứ tự NLR.

iii. Xóa nút chứa nội dung là x?

❑ **Thực hiện:**

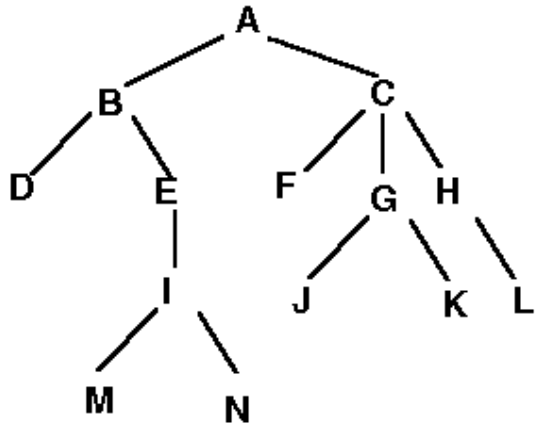
```
#include<iostream.h>
#include<conio.h>
struct NodeTree
{
 int nd;
 NodeTree *Left,*Right;
};
NodeTree *Root;
void Initialize()
{
 Root=NULL;
}
int CountNode(NodeTree *p)
{
 if(p)return 1+CountNode(p->Left)+CountNode(p->Right);
 else return 0;
}
void InsertEle(int x,NodeTree *&Root)
{
 if(!Root)
 {
  Root=new NodeTree;Root->nd=x;
  Root->Left=Root->Right=NULL;
 }
```

```
 else
  if(CountNode(Root->Left) > CountNode(Root->Right))
      InsertEle(x,Root->Right);
  else InsertEle(x,Root->Left);
}
NodeTree *FindX(int x, NodeTree *Root)
{
 if(Root)
  if(Root->nd==x)return Root;
  else
  {
   NodeTree *q;
   q=FindX(x,Root->Left);
   if(!q)q=FindX(x,Root->Right);
   return q;
  }
 else return NULL;
}
int DeleteNode(NodeTree *&Root)
{
  if(!Root->Left && !Root->Right)
  {NodeTree *p=Root;Root=NULL;
   int x=p->nd;delete p; return x;}
  else
   if(CountNode(Root->Left) > CountNode(Root->Right))
      return DeleteNode(Root->Left);
   else return DeleteNode(Root->Right);
}
void DeleteX(int x,NodeTree *&Root)
{
 if(Root)
 {NodeTree *p=FindX(x,Root);p->nd=DeleteNode(Root);}
 else cout<<"\nCay khong chua ptu co gia tri "<<x;
}
void PrintLNR(NodeTree *p)
{
  if(p)
  {
   PrintLNR(p->Left);
   cout<<p->nd<<"  ";
   PrintLNR(p->Right);
  }
}
void main()
{
```

```
    clrscr();
    Initialize();
    int n;
    cin>>"Nhap so nut cua cay can bang: ";cin>>n;
    for(int i=1;i<=n;i++)
    {
     cout<<"Gia tri nut thu "<<i<<" la: ";cin>>x;
     InsertEle(x,Root);
    }
    cout<<"\nGtri tren cay duyet theo thu tu LNR: ";
    PrintLNR(Root);
    cout<<"\n\nNhap gia tri muon tim:";cin>>x;
    NodeTree *q=FindX(x,Root);
    cout<<"\n\nNhap gia tri muon xoa trong cay:";
    cin>>x;
    DeleteX(x,Root);
    cout<<"\nCay sau khi xoa:\n";
    PrintLNR(Root);
    getch();
}
```

Bài tập cuối chương

1. Trình bày các biểu thức duyệt tiền tự, trung tự, hậu tự của cây trong hình 4.21



Hình 4.21. *Cây tam phân*

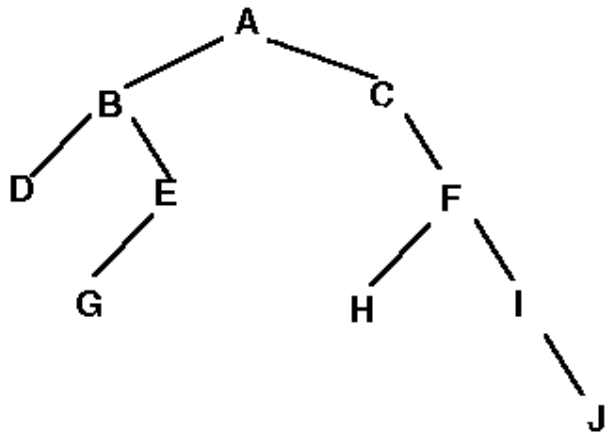
2. Duyệt cây theo mức là duyệt bắt đầu từ gốc, rồi duyệt các nút nằm trên mức 1 theo thứ tự từ trái sang phải, rồi đến các nút nằm trên mức 2 theo thứ tự từ trái sang phải...và cứ như vậy.

- Hãy liệt kê các nút theo thứ tự duyệt theo mức của cây trong câu 1.
- Viết chương trình con duyệt cây theo mức. (Gợi ý: dùng hàng đợi)

3. Vẽ cây biểu diễn cho biểu thức ((a+b)+c*(d+e)+f)*(g+h)

Trình bày biểu thức tiền tố và hậu tố của biểu thức đã cho.

4. Cho cây nhị phân trong hình 4.22



Hình 4.22. *Cây nhị phân*

- Hãy trình bày các phép duyệt: tiền tự, trung tự, hậu tự.
- Minh hoạ sự lưu trữ kế tiếp các nút cây này trong mảng.

5. Hãy viết các chương trình con sau thực hiện trên cây nhị phân:

- Kiểm tra cây rỗng.
- Kiểm tra nút n có phải là nút lá không.
- Kiểm tra nút n có phải là nút cha của nút m không.
- Tính chiều cao của cây.
- Tính số nút của cây.
- Duyệt tiền tự, trung tự, hậu tự.
- Đếm số nút lá của cây.
- Đếm số nút trung gian của cây.

6. Vẽ hình cây nhị phân tìm kiếm tạo ra từ cây rỗng bằng cách lần lượt thêm vào các khoá là các số nguyên: 54, 31, 43, 29, 65, 10, 20, 36, 78, 59.

7. Vẽ lại hình cây nhị phân tìm kiếm ở câu 6 sau khi lần lượt xen thêm các nút 15, 45, 55.

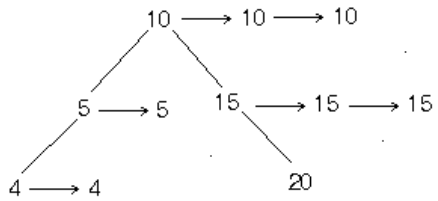
8. Vẽ lại hình cây nhị phân tìm kiếm ở câu 6 sau khi lần lượt xoá các nút 10, 20, 43, 65, 54.

9. Hãy dựng cây nhị phân tìm kiếm ứng với dãy khóa (thứ tự tính theo qui tắc so sánh chuỗi (string)): HAIPHONG, CANTHO, NHATRANG, DALAT, HANOI, ANGIANG, MINHHAI, HUE, SAIGON, VINHLONG. Đánh dấu đường đi trên cây khi tìm kiếm khóa DONGTHAP.

10. Cài đặt cây TKNP có khoá là chuỗi (String) với các phép toán thêm, xoá. Bổ sung thêm các chương trình con cần thiết đề có 1 chương trình hoàn chỉnh, cung cấp giao diện để người dùng có thể thêm, xoá 1 khoá và duyệt cây để kiểm tra kết quả.

11. Viết các chương trình con thêm, xoá một nút có khoá x trên cây nhị phân tìm kiếm bằng cách không đệ qui.

12. Để mở rộng khả năng xử lí các khoá trùng nhau trên cây nhị phân tìm kiếm, ta có thể tổ chức cây nhị phân tìm kiếm như sau: tại mỗi nút của cây ta tổ chức một danh sách liên kết chứa các khoá trùng nhau đó. Chẳng hạn cây được thiết lập từ dãy khoá số nguyên 10, 15, 5, 10, 20, 4, 5, 10, 15, 15, 4, 15 như hình 4.23.



Hình 4.23. *Cây BST chứa DSLK*

Trong đó các mũi tên nằm ngang chỉ các con trỏ của danh sách liên kết.

Hãy viết khai báo cấu trúc dữ liệu và các chương trình con/hàm để cài đặt cây TKNP mở rộng như vậy.

Chương 5. ĐỒ THỊ
(Graph)

Đồ thị được dùng để giải các bài toán trong nhiều lĩnh vực khác nhau. Thí dụ, dùng đồ thị để xác định xem có thực hiện một mạch điện trên một bảng điện phẳng được không. Ta cũng có thể phân biệt hai hợp chất hóa học có cùng công thức phân tử nhưng có cấu trúc khác nhau nhờ đồ thị. Hoặc cũng có thể xác định xem hai máy tính có được nối với nhau bằng một đường truyền hay không nếu dùng mô hình đồ thị mạng máy tính. Đồ thị với các trọng số được gán cho các cạnh của nó có thể dùng để giải các bài toán như bài toán tìm đường đi ngắn nhất giữa hai thành phố trong một mạng giao thông. Chúng ta cũng có thể dùng đồ thị để lập lịch thi và phân chia kênh cho các đài truyền hình…

5.1. Các khái niệm cơ bản

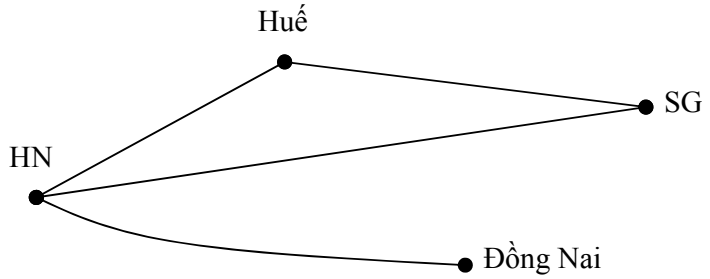
5.1.1. Các định nghĩa

Đồ thị là một cấu trúc rời rạc gồm các đỉnh và các cạnh (vô hướng hoặc có hướng) nối các đỉnh đó. Có hai loại đồ thị: vô hướng và có hướng.

Đồ thị vô hướng G = (V, E) gồm một tập khác rỗng V mà các phần tử của nó gọi là các đỉnh và một tập E mà các phần tử của nó gọi là các cạnh, trong đó mỗi cạnh của E được nối bởi hai phần tử khác nhau của V.

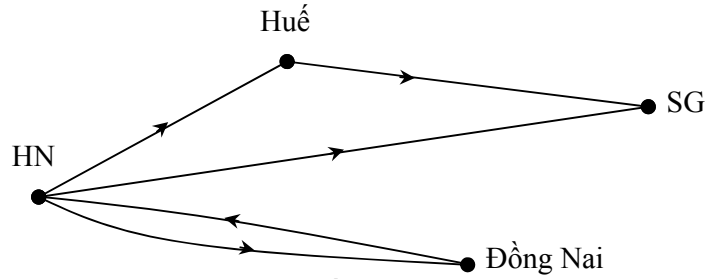
Đồ thị có hướng G = (V, E) gồm một tập khác rỗng V mà các phần tử của nó gọi là các đỉnh và một tập E mà các phần tử của nó gọi là các cung, trong đó mỗi cung của E được nối bởi hai phần tử khác nhau của V.

- ***Ví dụ:*** Đồ thị trong hình 5.1 là đồ thị vô hướng. Trong đó V = {HN, Huế, Đồng Nai, SG}, E = {(HN, Huế), (Huế, SG), (HN, SG), (HN, Đồng Nai)}



Hình 5.1. *Đồ thị vô hướng G*

- ***Ví dụ:*** Đồ thị trong hình 5.2 là đồ thị có hướng. Trong đó V = {HN, Huế, Đồng Nai, SG}, E = {(HN, Huế), (Huế, SG), (HN, SG), (HN, Đồng Nai), (Đồng Nai, HN)}



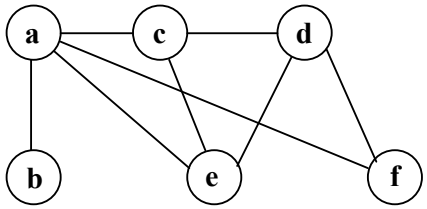
Hình 5.1. *Đồ thị có hướng G*

5.1.2. Các thuật ngữ

a. Hai đỉnh u, v của đồ thị vô hướng G được gọi là kề nhau nếu (u, v) là cạnh của đồ thị G.

b. Bậc của một đỉnh v trong đồ thị vô hướng là số cạnh liên thuộc với nó và kí hiệu deg(v).

- ***Ví dụ:***



deg(a) = 4, deg(b) = 1
deg(c) = 3, deg(d) = 3
deg(e) = 3, deg(f) = 2

Hình 5.3. *Đồ thị vô hướng G và bậc của các đỉnh*

5.1.3. Đường đi, chu trình, đồ thị liên thông

5.1.3.1. Đường đi

- Đường đi trong đồ thị là độ dài n từ đỉnh u đến đỉnh v. Trong đó: n-nguyên dương, u-đỉnh đầu của đồ thị, v-đỉnh cuối của đồ thị.

- Trên đồ thị G = (V, E) đường đi là dãy $x_0, x_1,\ldots,x_n$. Trong đó $u=x_0$ và $v=x_n$. Đường đi trong đồ thị này có thể biểu diễn bởi các cạnh: $(x_0, x_1), \ldots, (x_{n-1}, x_n)$

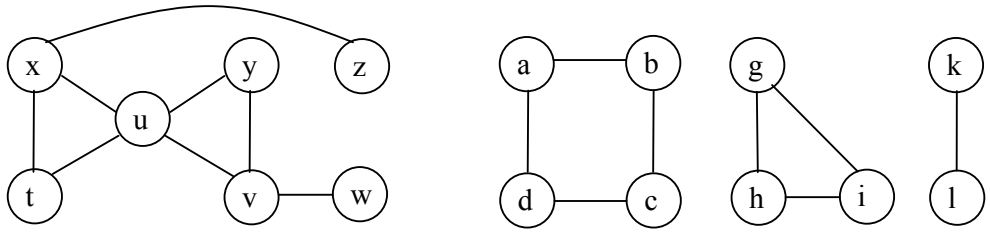
5.1.3.2. Chu trình

Trong đồ thị, nếu đường đi có đỉnh đầu trùng với đỉnh cuối, thì đường đi này gọi là chu trình.

5.1.3.3. Đồ thị liên thông

Định nghĩa: Đồ thị vô hướng G = (V, E) gọi là liên thông nếu luôn tìm được đường đi giữa hai đỉnh bất kỳ trong đồ thị.

- ***Ví dụ:*** Đồ thị G là liên thông, nhưng đồ thị G' không liên thông và có 3 thành phần liên thông.



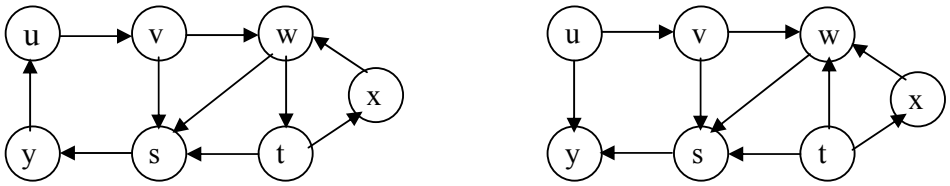
Đồ thị G **Đồ thị G'**

Hình 5.4. *Đồ thị liên thông G, và không liên thông G'*

Định nghĩa: Đồ thị có hướng G = (V, E) gọi là liên thông mạnh nếu luôn tìm được đường đi giữa hai đỉnh bất kỳ trong đồ thị.

Định nghĩa: Đồ thị có hướng G = (V, E) gọi là liên thông yếu nếu đồ thị vô hướng tương ứng với nó là liên thông.

- ***Ví dụ:*** Đồ thị G là liên thông mạnh nhưng đồ thị G' là liên thông yếu (không có đường đi từ u tới x cũng như từ x tới u).



Đồ thị G **Đồ thị G'**

Hình 5.5. *Đồ thị liên thông mạnh G, và liên thông yếu G'*

5.2. Biểu diễn đồ thị trên máy tính

5.2.1. Ma trận kề, ma trận trọng số

Xét đồ thị vô hướng G = (V, E); trong đó tập đỉnh V = {1, 2, …, n}, tập cạnh E = {e_1, …, e_n}. Ta gọi ma trận kề của đồ thị G là ma trận hai chiều chỉ chứa các phần tử (0, 1).

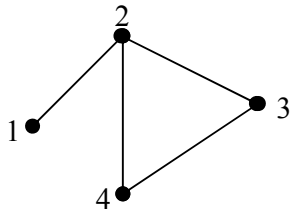
Nghĩa là:

$$A = \{a_{ij} : \ i, j = 1, 2, \ldots, n\}$$

với các phần tử trên A được xác định như sau:

$$a_{ij} = \begin{cases} 0 & if\ (i, j) \in E \\ 1 & otherwise \end{cases}$$

- ***Ví dụ:***



Ma trận kề tương ứng

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

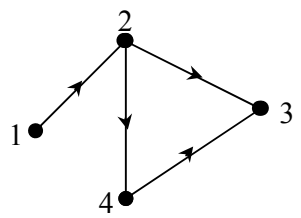
Hình 5.6. *Đồ thị vô hướng G và ma trận kề tương ứng*

➢ **Kết luận:**

- Ma trận kề của đồ thị vô hướng là ma trận đối xứng: $a_{ij} = a_{ji}$; i,j=1, 2, …, n
- Tổng các phần tử trên hàng i (cột j) của ma trận kề chính bằng bậc của đỉnh i (đỉnh j).

• **Ma trận kề của đồ thị có hướng:** Được định nghĩa tương tự với đồ thị vô hướng.

❑ ***Ví dụ:***



Ma trận kề tương ứng

$$A = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \end{bmatrix}$$

Hình 5.7. *Đồ thị có hướng G và ma trận kề tương ứng*

➢ **Kết luận:** Ma trận kề của đồ thị có hướng là ma trận không đối xứng.

✍ ***Ghi chú:*** Trong ứng dụng của đồ thị, mỗi cạnh của đồ thị thường được gán bởi một trọng số và được gọi là ma trận trọng số.

➢ *Giải thuật cài đặt ma trận kề*

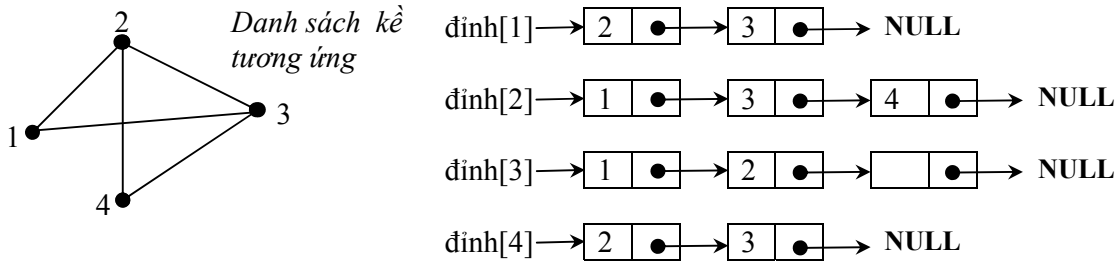
```
#include<iostream.h>
#include<conio.h>
#define MaxV 101     //Đồ thị có tối đa 100 đỉnh
int A[MaxV][MaxV];
void main()
{
 clrscr();
 int n,x,i,j;
 cout<<"Do thi co so dinh: ";cin>>n;
 for(i=1;i<=n;i++)
  for(j=1;j<=n;j++)A[i][j]=0;
 for(i=1;i<=n;i++)
 {
   cout<<"Nhap cac dinh ke voi "<<i<<;
   cout<<", bam so 0 de ket thuc.\n";
   cout<<"dinh "<<i<<" ke dinh:";
   do
   {
     cin>>x;
     if(x!=0)A[i][x]=1;
   }while(x!=0);
 }
 //Xuất kết quả
 for(i=1;i<=n;i++)
 {
  cout<<"\nke dinh "<<i<<" co cac dinh:";
  for(j=1;j<=n;j++)
   if(A[i][j])cout<<j<<"  ";
```

```
  cout<<"\n";
 }
 getch();
}
```

5.2.2. Danh sách kề

Trong cách biểu diễn này, n hàng của ma trận kề được biểu diễn bởi n danh sách liên kết. Mỗi danh sách tương ứng với mỗi đỉnh trong G. Các nút trong danh sách i (1≤ i ≤ n) biểu diễn đỉnh liền kề với đỉnh i. Mỗi nút trong danh sách có ít nhất hai trường: ***nd*** (chứa đỉnh liền kề) và trường ***link*** (trường liên kết).

❑ ***Ví dụ:***



Hình 5.8. *Đồ thị vô hướng G và danh sách kề tương ứng*

➢ *Giải thuật cài đặt danh sách kề*

```
#include<iostream.h>
#include<conio.h>
#define MaxV 101     //Đồ thị có tối đa 100 đỉnh
struct Tro
{
 int nd;Tro *link;
};
Tro *Dinh[MaxV];
void Initialize(int k)
{
  for(int i=1;i<=k;i++)Dinh[i]=NULL;
}
void Insert(int x,Tro *&p)
{
 Tro *q=new Tro;q->nd=x;q->link=p;p=q;
}
void Print(Tro *p)
{
 if(p)
 {cout<<p->nd<<"  ";Print(p->link);}
}
void main()
{
 clrscr();
 int n,x;
 cout<<"Do thi co so dinh: ";cin>>n;
 Initialize(k);
 for(int i=1;i<=n;i++)
 {
   cout<<"Nhap cac dinh ke voi "<<i<<;
```

```
    cout<<", bam so 0 de ket thuc.\n";
    cout<<"dinh "<<i<<" ke dinh:";
    do
    { cin>>x;
      if(x!=0)Insert(x,Dinh[i]);
    }while(x!=0);
  }
  for(i=1;i<=n;i++)
  {
   cout<<"\nke dinh "<<i<<" co cac dinh:";
   Print(Dinh[i]);
  }
  getch();
}
```

5.3. Tìm kiếm trên đồ thị

Với đồ thị vô hướng G = (V, E), và với đỉnh v bất kỳ trong tập V. Vấn đề ta quan tâm ở đây là xuất phát từ đỉnh v, ta có thể đi đến các đỉnh còn lại trên đồ thị G hay không?

Để làm được điều này, ta có hai cách: Sử dụng thuật toán tìm kiếm theo chiều rộng (Breadth First Search) hoặc sử dụng thuật toán tìm kiếm theo chiều sâu (Depth First Search).

5.3.1. Tìm kiếm theo chiều sâu trên đồ thị

Xuất phát từ đỉnh v của đồ thị, sau đó chọn đỉnh w là đỉnh tùy ý kề với v, và lặp lại quá trình tìm kiếm đối với đỉnh w. Ở bước tổng quát, giả sử ta đang xét đỉnh u, nếu như trong số các đỉnh kề với u tìm được đỉnh nào chưa xét thì ta sẽ xét đỉnh này và bắt đầu từ nó ta lặp lại quá trình tìm kiếm.

Giải thuật này kiểm tra xem xuất phát tại một đỉnh bất kỳ trong đồ thị có tồn tại đường đi đến các đỉnh khác không? Nếu có, thì tất cả các đỉnh này sẽ được thăm, ngược lại nếu đỉnh nào không được thăm thì không tồn tại đường đi đến đỉnh đó.

➢ *Giải thuật tìm kiếm theo chiều sâu*

```
procedure DFS(v)
/* Đồ thị vô hướng G=(V,E) có n đỉnh và mảng VlSlTED(n) được
khởi tạo bằng 0, giải thuật này thăm tất cả các đỉnh xuất phát
từ đỉnh v, với G và VISITED là biến toàn cục*/
VISITED (v) ← 1
for each đỉnh w kề với v do
if VISlTED(w) = 0 then call DFS(w)
end
end DFS
procedure COMP(G,n)
/*G là đồ thị có n đỉnh, VISITED là biến mảng địa phương.*/
for i ← 1 to n do
VISITED(i) ← 0 //Khởi tạo các đỉnh chưa được thăm
end
for i ←1 to n do
 if VISITED(i) = 0 then
   [call DFS(i); Xuất các đỉnh mới thăm cùng với các
```

```
   cạnh liên quan với chúng]
end
end COMP
```

➢ *Cài đặt giải thuật tìm kiếm theo chiều sâu bằng ngôn ngữ C/C++*

```
void DFS(int v)
{
  VISITED[v] = 1;
  Tro *w=Dinh[v];
  while(w)
  {
       if(VISITED[w->nd]==0)DFS(w->nd); w = w->link;
  }
}
void Tham(int n)
{
 for(int i=1;i<=n; i++) VISITED[i]=0;
}
void COMP(int n)
{
for(int i=1;i<=n; i++)
 if(VISITED[i] == 0)
 {
   DFS(i);
   for(int j=1;j<=n;j++)
    if(!VISITED[j])
      cout<<"\nKhong co ddi tu dinh "<<i<<" den "<<j;
   Tham(n);
  }
}
```

5.3.1. Tìm kiếm theo chiều rộng trên đồ thị

Thuật toán tìm kiếm theo chiều rộng tương tự tìm kiếm theo chiều sâu, nhưng trong giải thuật tìm kiếm theo chiều sâu ta sử dụng STACK (đệ quy), còn thuật toán tìm kiếm theo rộng sử dụng QUEUE.

➢ *Giải thuật tìm kiếm theo chiều rộng*

```
procedure BFS(v)
/*Xuất phát tại đỉnh v của đồ thị G, tất cả các đỉnh được thăm
được đánh dấu bởi VISlTED(i)= 1. Đồ thị G và mảng VISITED là
biến toàn cục và VISITED được khởi tạo bằng 0.*/
VISITED (v) ← 1
Khởi tạo Q rỗng          /*Q is a queue*/
loop
for các đỉnh w kề với v do
if VISITED(w) = 0           /*add w to queue*/
then [call ADDQ(w,Q); VISITED(w)←1]
/*mark w as VISITED*/
end
if Q rỗng then return
call DELETEQ(v,Q)
forever
end BFS
```

✍ ***Ghi chú:*** Ta có thể sử dụng thuật toán tìm kiếm theo chiều sâu (hoặc tìm kiếm theo chiều rộng) để kiểm tra một đồ thị có liên thông hay không.

Bài tập cuối chương

1. Hãy cài đặt thuật toán tìm đường đi giữa hai đỉnh bất kỳ trong đồ thị theo thuật toán:

   a. Tìm kiếm theo chiều rộng.

   b. Tìm kiếm theo chiều sâu.

2. Viết thuật toán kiểm tra đồ thị vô hướng G có liên thông không theo hai thuật toán:

   a. Tìm kiếm theo chiều rộng.

   b. Tìm kiếm theo chiều sâu.

3. Làm bài tập 1 và 2 theo phương pháp khử đệ quy đối với thuật toán tìm kiếm theo chiều sâu.

4. Hãy cài đặt thuật toán danh sách kề trong trường hợp đồ thị có trọng số.

Chương 6. SẮP XẾP VÀ TÌM KIẾM
(Sorting and Searching)

Trong chương này ta thảo luận bài toán sắp xếp, tìm kiếm trên một mảng có n phần tử. Để đơn giản, các ví dụ được thực hiện trên mảng chứa các số nguyên, tuy thế ta cũng có thể thực hiện trên mảng chứa các phần tử có cấu trúc phức tạp hơn.

6.1. Sắp xếp

Giả sử ta cần sắp xếp tăng dần một mảng nguyên A có n phần tử.

6.1.1. Các phương pháp sắp xếp đơn giản

6.1.1.1. Phương pháp sắp xếp chèn (Insertion Sort)

➢ *Giải thuật*

```
void Insertion_sort(int *a, unsigned int n)
{
     unsigned int j, p;
     int tmp;
     for( p=1; p < n; p++ )
     { tmp = a[p];
       for(j=p; tmp<a[j-1] && j>0; j--) a[j] = a[j-1];
       a[j] = tmp;
     }
}
```

❑ ***Ví dụ:*** Với mảng A có các giá trị ban đầu 34, 8, 64, 51, 32, 21; cách thực hiện của giải thuật Insertion_sort được thực hiện trong bảng 6.1.

Bảng 6.1. *Giải thuật sắp xếp chèn ứng với mỗi vị trí của p*

Vị trí	0	1	2	3	4	5	
Mảng gốc A	34	8	64	51	32	21	Số vị trí di chuyển
Khi *p* = 1	8	34	64	51	32	21	1
Khi *p* = 2	8	34	64	51	32	21	0
Khi *p* = 3	8	34	51	64	32	21	1
Khi *p* = 4	8	32	34	51	64	21	3
Khi *p* = 5	8	21	32	34	51	64	4

✍ ***Phân tích giải thuật sắp xếp chèn:***

Độ phức tạp của giải thuật là

T(n) = $\sum_{p=1}^{n-1}\sum_{j=1}^{p} C = C\sum_{p=1}^{n-1} p = C[1+2+...+(n-1)] = C.\frac{n(n-1)}{2} = O(n^2)$, với C là hằng số nào đó.

6.1.1.2. Phương pháp sắp xếp nổi bọt (Bubble Sort)

➢ *Giải thuật*

```
void Bubble_sort(int *a, unsigned int n)
{
      unsigned int i, j;
      int tmp;
      for(i=0; i<n-1; i++ )
         for(j=i+1; j<n; j++)
            if(a[i]>a[j])
                 {tmp=a[i]; a[i]=a[j]; a[j]=tmp; }
}
```

❑ ***Ví dụ:*** Với mảng A có các giá trị ban đầu 34, 8, 64, 51, 32, 21; cách thực hiện của giải thuật Bubble_sort là:

Bảng 6.2. *Giải thuật sắp xếp nổi bọt ứng với mỗi vị trí của i (các phần tử khoanh tròn trên cùng hàng đã đổi chỗ cho nhau)*

Vị trí	0	1	2	3	4	5	
Mảng gốc A	34	8	64	51	32	21	Số cặp hoán vị
Khi i = 0	8	34	64	51	32	21	1
Khi i = 1	8	32	64	51	34	21	2
	8	21	51	64	34	32	
Khi i = 2	8	21	34	64	51	32	2
	8	21	32	64	51	34	
Khi i = 3	8	21	32	51	64	34	2
	8	21	32	34	64	51	
Khi i = 4	8	21	32	34	51	64	1

✍ ***Phân tích giải thuật sắp xếp nổi bọt:***

Độ phức tạp của giải thuật là

$$T(n) = \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} C = C\sum_{i=0}^{n-2}(n-i-1) = C[(n-1)+...+2+1] = C.\frac{n(n-1)}{2} = O(n^2),$$

với C là hằng số nào đó.

6.1.1.3. Phương pháp sắp xếp chọn (Selection Sort)

➢ *Giải thuật*

```
void Selection_sort(int *a, unsigned int n)
{
      unsigned int i, j;  int tmp, k, x;
      for(i=0; i<n-1; i++ )
      {  x=a[i]; k=i;
            for(j=i+1; j<n; j++)
                 if(a[j]<tmp){x=a[j]; k=j;
            if(k!=i){tmp=a[i]; a[i]=a[k]; a[k]=tmp;}
      }
}
```

❑ ***Ví dụ:*** Với mảng A có các giá trị ban đầu 34, 8, 64, 51, 32, 21; cách thực hiện của giải thuật Selection_sort là:

Bảng 6.3. *Giải thuật sắp xếp chọn ứng với mỗi vị trí của i (các phần tử khoanh tròn trên cùng hàng đã đổi chỗ cho nhau)*

Vị trí	0	1	2	3	4	5	
Mảng gốc A	34	8	64	51	32	21	Số cặp hoán vị
Khi i = 0	8	34	64	51	32	21	1
Khi i = 1	8	21	64	51	32	34	1
Khi i = 2	8	21	32	51	64	34	1
Khi i = 3	8	21	32	34	64	51	1
Khi i = 4	8	21	32	34	51	64	1

✍ ***Phân tích giải thuật sắp xếp chọn:***

Độ phức tạp của giải thuật là

$$T(n) = \sum_{i=0}^{n-2}\sum_{j=i+1}^{n-1} C = C\sum_{i=0}^{n-2}(n-i-1) = C[(n-1)+...+2+1] = C.\frac{n(n-1)}{2} = O(n^2),$$

với C là hằng số nào đó.

6.1.2. Các phương pháp sắp xếp nhanh

6.1.2.1. Phương pháp sắp xếp Quick sort

Ý tưởng của phương pháp này là chọn x làm phần tử giữa của dãy, ta phân hoạch dãy này thành 3 dãy con liên tiếp:

- Dãy con thứ nhất chứa các giá trị nhỏ hơn x.
- Dãy con thứ hai chứa các giá trị bằng x.
- Dãy con thứ ba chứa các giá trị lớn hơn x.

Sau đó áp dụng giải thuật đệ quy cho dãy con thứ nhất và dãy con thứ 3 nếu dãy này có nhiều hơn một phần tử.

➢ *Giải thuật*

```
void Quick_sort(int *a,unsigned int L,unsigned  int R)
{
 unsigned int i, j;
 i=L;j=R; int x=a[(i+j)/2];
 while(1)
 {
  while(a[i]<x)i++;
  while(a[j]>x)j--;
  if(i<=j){int tmp=a[i]; a[i]=a[j];a[j]=tmp;i++;j--;}
  else break;
 }
 if(L<j)Quick_sort(a,L,j);
 if(R>i)Quick_sort(a,i,R);
}
```

✍ ***Ghi chú:*** Để sắp xếp mảng nguyên A có n phần tử, thì trong chương trình chính chỉ cần gọi Quick_sort(A, 0, n-1);

✍ ***Phân tích giải thuật Quick sort:***

❑ **Trường hợp xấu nhất:**

Dãy con thứ nhất có n-1 phần tử, dãy con thứ 2 có 1 phần tử. Lúc đó thời gian tiêu tốn cho trường hợp này là:

$$T(n) = \begin{cases} 1 & ,if \quad n = 1 \\ T(n-1) + n & , otherwise \end{cases}$$

Sử dụng phương pháp lặp, ta có

$$T(n) = \sum_{i=1}^{n} i = 1 + 2 + ... + n = \frac{n(n+1)}{2} = O(n^2)$$

❑ **Trường hợp tốt nhất:**

Dãy con thứ nhất có n/2 phần tử, dãy con thứ 2 có n/2 phần tử. Lúc đó thời gian tiêu tốn cho trường hợp này là:

$$T(n) = \begin{cases} 1 & ,if \quad n = 1 \\ 2T(\frac{n}{2}) + n & , otherwise \end{cases}$$

Sử dụng phương pháp lặp, ta có:

$$T(n) = xn + 2^x T(\frac{n}{2^x}); x \in R$$

Quá trình lặp này sẽ kết thúc khi $T(n/2^x) = 1$, nghĩa là $n/2^x = 1 \rightarrow x = \log n$. Vậy

$$T(n) = (n+1)\log n = O(n \log n)$$

6.1.2.2. Phương pháp sắp xếp Merge sort

Ý tưởng của phương pháp

- Ta chia đôi một dãy cần sắp thành 2 dãy con có chiều dài n/2, sau đó ta tiếp tục chia đôi 2 dãy con có chiều dài là n/2 thành các dãy con có chiều dài n/4,… cho đến khi mỗi dãy con chỉ còn lại chiều dài là 1.

- Ta tiến hành trộn (merge) các cặp dãy có chiều là 1 để tạo thành các dãy sắp xếp có chiều dài là 2, tiến hành trộn các cặp dãy có chiều dài là 2 để tạo thành các dãy sắp xếp có chiều là 4, …cho đến khi hai dãy có chiều dài n/2 được trộn để tạo thành dãy sắp xếp chung có chiều là n.

➢ *Giải thuật*

```
void Merge(int *a, int L, int mid, int R)
      {
        unsigned int i, j, k;
        int *b=new int[R-L+1];
        i=L;j=mid+1;k=0;
        while(i<=mid && j<=R)
```

```
    if(a[i]<=a[j]){b[k]=a[i];i++;k++;}
    else {b[k]=a[j];j++;k++;}
   while(i<=mid){b[k]=a[i];i++;k++;}
   while(j<=R){b[k]=a[j];j++;k++;}
   for(i=0;i<R-L+1;i++)a[i+L]=b[i];
   delete b;
}
void Merge_sort(int *a, int L, int R)
{
   if(L<R)
   {
    int mid=(L+R)/2;
    Merge_sort(a,L,mid);
    Merge_sort(a,mid+1,R);
    Merge(a,L,mid,R);
   }
}
```

✍ ***Phân tích giải thuật Merge sort:***

Độ phức tạp của giải thuật Merge sort là:

$$T(n) = \begin{cases} 1 & ,if \quad n = 1 \\ 2T(\frac{n}{2}) + n & ,otherwise \end{cases}$$

Sử dụng phương pháp lặp tương tự phương pháp Quick sort, ta có:

$$T(n) = (n+1)\log n = O(n\log n)$$

6.1.2.3. Phương pháp sắp xếp Heap sort

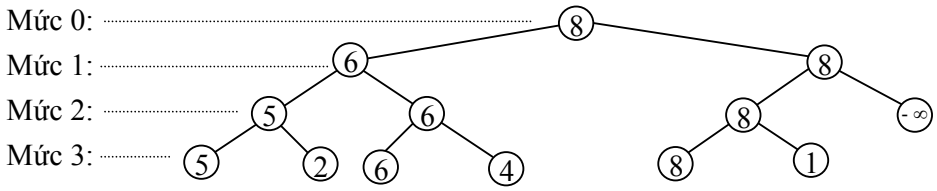
a. Ý tưởng:

Phương pháp chọn lựa trực tiếp dựa vào việc chọn phần tử nhỏ nhất từ n phần tử, từ n-1 phần tử, … Do vậy độ phức tạp của thuật toán này là $O(n^2)$.

Vì lý do trên người ta tìm cách xây dựng một thuật toán sắp xếp khác có thể khắc phục nhược điểm này.

Mấu chốt để giải quyết vấn đề vừa nêu là phải tìm ra được một cấu trúc dữ liệu cho phép tích lũy các thông tin về sự so sánh giá trị các phần tử trong qua trình sắp xếp.

Giả sử dữ liệu cần sắp xếp là dãy số: 5 2 6 4 8 1 được bố trí theo quan hệ so sánh và tạo thành sơ đồ dạng cây như trong hình 6.1.

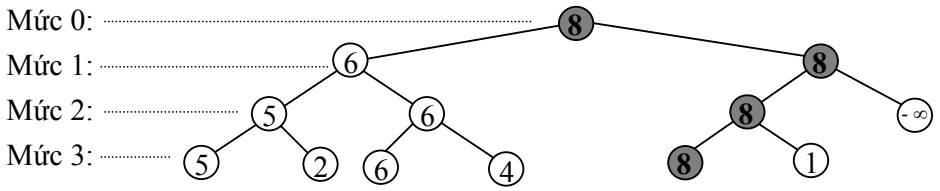


Hình 6.1. *Cây sắp xếp*

Trong đó một phần tử ở mức i chính là phần tử lớn trong cặp phần tử ở mức i+1, do đó phần tử ở mức 0 (nút gốc của cây) luôn là phần tử lớn nhất của dãy.

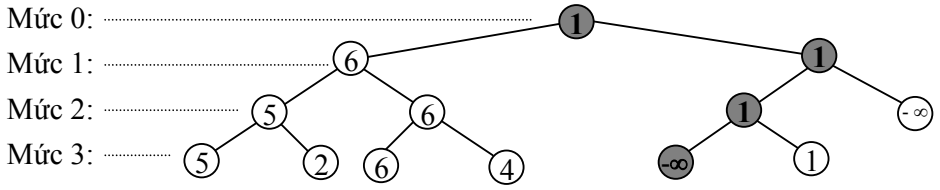
Nếu loại bỏ phần tử gốc ra khỏi cây (nghĩa là đưa phần tử lớn nhất về đúng vị trí), thì việc cập nhật cây chỉ xảy ra trên những nhánh liên quan đến phần tử mới loại bỏ, còn các nhánh khác được bảo toàn, nghĩa là bước kế tiếp có thể sử dụng lại các kết quả so sánh ở bước hiện tại.

Trong ví dụ trên ta có:



Hình 6.2. *Cây sắp xếp với nút gốc được chọn để loại bỏ*

Loại bỏ 8 ra khỏi cây và thế vào các chỗ trống giá trị -∞ để tiện việc cập nhật lại cây:



Hình 6.3. *Cây sắp xếp với nút gốc đã bị loại*

Tiến hành nhiều lần việc loại bỏ phần tử gốc của cây cho đến khi tất cả các phần tử của cây đều là -∞, khi đó xếp các phần tử theo thứ tự loại bỏ trên cây sẽ có dãy đã sắp xếp. Trên đây là ý tưởng của giải thuật sắp xếp cây.

b. Cấu trúc dữ liệu Heap

Tuy nhiên, để cài đặt thuật toán này một cách hiệu quả, cần phải tổ chức một cấu trúc lưu trữ dữ liệu có khả năng thể hiện được quan hệ của các phần tử trong cây với n ô nhớ thay vì 2n-1 như trong ví dụ.

Khái niệm heap và phương pháp sắp xếp Heapsort do J.Williams đề xuất đã giải quyết được các khó khăn trên.

Định nghĩa Heap:

Giả sử xét trường hợp sắp xếp tăng dần, khi đó dãy các phần tử $a_1, a_2, \ldots, a_n$ là một Heap nếu thoả: $a_k >= a_j$; với k=[j/2] và 1 <= k < j <=n

Heap có các tính chất sau :

Tính chất 1 : Nếu $a_1, a_2, \ldots, a_n$ là một heap thì dãy này chưa chắc có thứ tự, bởi vì:

$a_1 >= a_2$; $a_1 >= a_3$; nhưng có thể $a_2 < a_3$

Nhưng nếu dãy $a_1, a_2, ..., a_n$ có thứ tự thì dãy này là một heap.

Tính chất 2 : Nếu $a_1, a_2, ..., a_n$ là một heap thì phần tử a_1 (đầu heap) luôn là phần tử lớn nhất trong heap.

Tính chất 3 : Mọi dãy $a_1, a_2, ..., a_n$, dãy con $a_j, a_{j+1}, \ldots, a_n$ tạo thành một heap với j=(n/2 +1).

Nếu dãy $a_1, a_2, ..., a_n$ là một heap thì dãy này có thể mô tả thành cây nhị phân như sau:

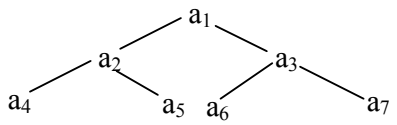
a_1 là nút gốc

a_1 có hai nút con là a_2 và a_3

a_2 có hai nút con là a_4 và a_5

…..

Ví dụ: Nếu n=7, ta có cây nhị phân biểu diễn heap như hình 6.4.



Hình 6.1. *Cây nhị phân biểu diễn heap*

Tạo heap ban đầu

Dãy $a_{q-1}, a_q, \ldots, a_r$ là một heap nếu ta đã có một heap $a_q, \ldots, a_r$

➢ *Giải thuật*

```
procedure Heaprify(a,q,r)
/* a là mảng, với q là vị trí phần tử đầu tiên, r là vị trí
phần tử cuối cùng trong mảng a*/
i ← q
j ← con trái của i
x ← a[i]
while j<=r do
   if j<r và a[j]<a[j+1] then tăng j lên 1
   if x>a[j] then thoát khỏi vòng lặp
   ngược lại
       chép giá trị nút con j vào nút i
       i ← j; j ← 2*i
end while
end procedure
procedure BuildHeap(a, n)
/* a là mảng, n là số phần tử trong mảng a*/
for q ← n div 2 downto 1 do Heaprify(a,q,n);
end procedure
```

➢ *Giải thuật trong C*

```
void Heaprify(int *a, int q, int r)
{
```

```
  int i,j,x;
  i=q; j=2*i+1; x=a[i];//i là vị trí phần tử đầu dãy
                  //j là vị trí phần tử con trái a[i]
  while(j<r)
  {
    if(j<r-1 && a[j]<a[j+1])j++;//Tìm nút con lớn nhất
    if(x>a[j])break;
    else
    {    //chép nút giá trị nút con j vào nút i
         a[i]=a[j]; i=j; j=2*i+1;
    }
  }
  a[i]=x;
}
void BuildHeap(int *a, int n)
{
    for(int q=n/2; q>=0; q--)Heaprify(a,q,n);
}
```

Lưu ý: Nếu dãy $a_0, a_1, \ldots, a_{n-1}$ là một heap thì dãy này có thể mô tả thành cây nhị phân trong C như sau:

a_0 là nút gốc

a_0 có hai nút con là a_1 và a_2

a_1 có hai nút con là a_3 và a_4

…..

Tổng quát :

a_i có hai nút con là a_{2i+1} và a_{2i+2} ; với $0 <= i <= (n-1)/2$

Ví dụ: Ta cần sắp thứ tự dãy số 3 2 6 4 8 1

Tạo heap ban đầu như bảng 6.4.

Bảng 6.4. *Khởi tạo heap*

	0	1	2	3	4	5
i=2	3	2	6:	4	8	1
i=1	3	2:	6	4	8	1
i=0	3:	8	6	4	2	1
	8	4	6	3	2	1

Giải thuật sắp xếp Heap sort

➢ *Giải thuật*

```
void HeapSort(int *a, int n)
{
    if(n>0)
    {
         BuildHeap(a,n);
```

```
            int tam=a[0]; a[0]=a[n-1]; a[n-1]=tam;
            HeapSort(a,n-1);
        }
    }
```

✍ ***Phân tích giải thuật Heap sort:***

Độ phức tạp của hàm Heaprify chính là vòng while(j<r); với j=0, 2, 4, 8,…, (n-1). Do vậy độ phức tạp của hàm này là O(logn)

Độ phức tạp của hàm HeapSort chính là độ phức tạp của hàm BuildHeap. Do trong hàm BuildHeap có vòng lặp for lặp lại n/2 lần, mỗi lần là O(logn), nên độ phức tạp của hàm này là O(nlogn).

Vậy độ phức tạp của hàm HeapSort là O(nlogn)

6.2. Tìm kiếm

6.2.1. Tìm kiến tuần tự (Sequential searching)

Giả sử ta cần tìm trên một mảng nguyên A có n phần tử xem có phần tử nào có giá trị bằng x không.

Nội dung của phương pháp này đó là ta bắt đầu tìm kiếm từ phần tử đầu tiên. Nếu phần tử này không có giá trị bằng x thì ta tìm đến phần tử kế tiếp, và cứ như thế cho đến khi tìm thấy phần tử cần tìm hoặc đến khi hết danh sách.

Kết quả tìm kiếm sẽ trả về vị trí phần tử tìm thấy hoặc -1 (nếu không tìm thấy).

➢ *Giải thuật*

```
int Sequential_search(int x, int *a, unsigned int n)
{
  for(int i=0; i<n; i++)
       if(a[i]==x) return i;
  return -1;
}
```

✍ ***Phân tích giải thuật tìm kiếm tuần tự:***

Độ phức tạp của giải thuật tìm kiếm tuần tự là: T(n) = $\sum_{i=0}^{n-1} C = nC = O(n)$, với C là hằng số nào đó.

6.2.2. Tìm kiến nhị phân (Binary searching)

Ý tưởng của phương pháp này là tìm kiếm giá trị x trên danh sách đã có thứ tự (giả sử danh sách đã có thứ tự tăng dần). Phương pháp này được thực hiện như sau:

Lấy giá trị của phần tử giữa trong danh sách rồi so sánh với x. Lúc này sẽ xảy ra 1 trong 3 trường hợp:

- Nếu bằng nhau thì trả về vị trí phần tử tìm thấy.
- Nếu phần tử giữa có giá trị nhỏ hơn x thì danh sách mới cần tìm sẽ nằm bên phải phần tử giữa.

- Nếu phần tử giữa có giá trị lớn hơn x thì danh sách mới cần tìm sẽ nằm bên trái phần tử giữa.

Quá trình trên được tiếp tục cho đến khi tìm thấy phần tử trong danh sách, hoặc danh sách cần tìm không còn phần tử nào.

➢ *Giải thuật*

```
int Binary_search(int x,int *a, int L, int R)
{
 if(L<=R)
 {
   int mid=(L+R)/2;
   if(a[mid]==x)return mid;
   else
    if(a[mid]<x)return Binary_search(x,a, mid+1,R);
    else return Binary_search(x,a, L,mid-1);
  }
  else return -1;
}
```

Bài tập cuối chương

1. Sắp xếp dãy số 3, 1, 4, 1, 5, 9, 2, 6, 5 bằng thuật toán chèn.

2. Thời gian thực hiện của thuật toán chèn như thế nào nếu tất cả các giá trị trong mảng đều bằng nhau.

3. Sắp xếp dãy số 3, 1, 4, 1, 5, 9, 2, 6 bằng thuật toán Merge sort.

4. Viết giải thuật Merge sort không sử dụng đệ quy.

5. Sắp xếp dãy số 3, 1, 4, 10, 5, 9, 2, 6, 5, 3, 5 bằng thuật toán Quick sort.

6. Cho dãy số 13, 1, 4, 10, 5, 9, 2, 6, 5, 3, 15. Hãy:

   a. Tạo hai heap ban đầu bằng thuật toán BuildHeap.

   b. Cài đặt thuật toán cho câu a

7. Viết hàm tìm kiếm giá trị x trên mảng nguyên n phần tử bằng thuật toán:

   a. Tìm kiếm tuần tự.

   b. Nhị phân tìm kiếm.

Yêu cầu: Kết quả tìm thấy trả về kiểu logic.

TÀI LIỆU THAM KHẢO

[1] Đỗ Xuân Lôi [1995], *Cấu trúc dữ liệu và giải thuật*, Nhà xuất bản Khoa học và Kỹ thuật, 1995.

[2] Alfred V.Aho, J. E. Hopcroft, J. D. Ullman, *Data Structure and Algorihtms*, ISBN0201000237, Addison Wesley, 1983

[3] R. Sedgewick, *Algorithms*, Addison Wesley,1987.

[4] N. Wirth, *Chương trình = cấu trúc dữ liệu + giải thuật*, 1983.

[5] Nguyễn Trung Trực, *Cấu trúc dữ liệu*, ĐHBK TP.HCM, 1990.

[6] Đinh Mạnh Tường, *Cấu trúc dữ liệu và giải thuật,* Nhà xuất bản Khoa học và Kỹ thuật, 2001.

[7] Larry N. Hoff, Sanford Leestma, *Lập trình nâng cao bằng Pascal với các cấu trúc dữ liệu – bản dịch của Lê Minh Trung,* Công ty scitec, 1991.