GIỚI THIỆU MÔN HỌC

Trong ngôn ngữ lập trình, dữ liệu bao gồm hai kiểu chính là :

- Kiểu dữ liệu đơn giản : char, int, long, float, enumeration, subrange.

- Kiểu dữ liệu có cấu trúc : struct, array, file (kiểu dữ liệu có kích thước không đổi)...

Giáo trình này tập trung vào việc nghiên cứu các kiểu dữ liệu có cấu trúc có kích thước không đổi hoặc thay đổi trong ngôn ngữ lập trình, mô tả thông qua ngôn ngữ C. Ngoài ra còn giới thiệu các giải thuật chung quanh các cấu trúc dữ liệu này như cách tổ chức, thực hiện các phép toán tìm kiếm, sắp thứ tự nội, sắp thứ tự ngoại...

Điều kiện để có thể tìm hiểu rõ ràng về môn học này là học viên đã biết các khái niệm về kỹ thuật lập trình trên ngôn ngữ C. Trong phần mở đầu, bài giảng này sẽ giới thiệu cách thức phân tích & thiết kế một giải thuật trước khi tìm hiểu về các cấu trúc dữ liệu cụ thể.

Vào cuối khóa học, sinh viên có thể:

- Phân tích độ phức tạp của các chương trình có kích thước nhỏ và trung bình.

- Nhận thức được sự cần thiết của việc thiết kế cấu trúc dữ liệu.

- Làm quen với các khái niệm stacks, queues, danh sách đặc, danh sách liên kết, cây nhị phân, cây nhị phân tìm kiếm, ...

- Hiểu được nguyên lý của việc xây dựng một chương trình máy tính.

- Có thể chọn lựa việc tổ chức dữ liệu phù hợp và các giải thuật xử lý dữ liệu có hiệu quả trong khi xây dựng chương trình. Sinh viên cần lưu ý rằng, tùy vào công việc cụ thể mà ta nên chọn cấu trúc dữ liệu nào là thích hợp theo hướng tối ưu về thời gian thực hiện hay tối ưu về bộ nhớ.

Chương I

PHÂN TÍCH & THIẾT KẾ GIẢI THUẬT

I. MỞ ĐẦU

Hầu hết các bài toán đều có nhiều giải thuật khác nhau để giải quyết chúng. Vậy làm thế nào chọn được một giải thuật tốt nhất ?

Việc chọn lựa phụ thuộc vào nhiều yếu tố như : Độ phức tạp tính toán của giải thuật, chiếm dung lượng bộ nhớ, tần suất sử dụng, tính đơn giản, tốc độ thực hiện...

Thông thường mục tiêu chọn lựa là :

1. Giải thuật rõ ràng, dễ hiểu, dễ mã hóa và hiệu chỉnh.
2. Giải thuật sử dụng có hiệu quả tài nguyên của máy tính và đặc biệt chạy càng nhanh càng tốt.

Do đó khi viết chương trình để chạy một lần hoặc ít chạy thì mục tiêu 1 là quan trọng hơn cả.

Ngược lại khi viết chương trình để chạy nhiều lần thì phí tổn chạy chương trình có thể vượt quá phí tổn lập chương trình, nhất là khi phải nhập nhiều số liệu. Nói chung, người lập trình phải biết chọn lựa, viết, đánh giá các giải thuật để có được giải thuật tối ưu cho bài toán của mình.

II. ĐÁNH GIÁ THỜI GIAN CHẠY CỦA CHƯƠNG TRÌNH

Thời gian chạy của chương trình phụ thuộc vào :

1. Input cho chương trình
2. Chất lượng mã sinh ra của chương trình dịch.
3. Trạng thái và tốc độ của các lệnh chạy trên máy.
4. Độ phức tạp thời gian của giải thuật.

Điều 1 là chức năng nhập. Kích thước của input (ví dụ là n) và ta thường ký hiệu T(n) là đại lượng thời gian cần thiết để giải bài toán kích thước n.

Điều 2, 3 thường đánh giá khó khăn vì phụ thuộc vào phần mềm chương trình dịch và phần cứng của máy.

Điều 4 là điều mà người lập trình cần khảo sát để làm tăng tốc độ của chương trình.

III. Ký hiệu o(n) vÀ Ω(n) :

Ta đánh giá tỷ lệ phát triển các hàm T(n) qua ký hiệu O(n).

Ta nói thời gian chạy T(n) của chương trình là $O(n^2)$ có nghĩa là :

$\exists\ c > 0$ và n_0 sao cho $\forall\ n \geq n_0$ ta có $T(n) \leq c.n^2$.

Ví dụ : Giả sử $T(0) = 1$, $T(1) = 4$, v v...

Tổng quát $T(n) = (n+1)^2$ thì ta nói $T(n)$ là $O(n^2)$ vì có thể đặt $c_1 = 4$, $n_0 = 1$, thì khi $n \geq 1$ ta có $(n+1)^2 \leq 4n^2$.

Nhưng không thể lấy $n_0 = 0$ vì $T(0) = 1$ không nhỏ hơn $c.0^2 = 0, \forall c$; giả thiết rằng $n \geq 0$ và $T(n) \geq 0$.

Ta nói $T(n)$ là $O(f(n))$ nếu $\exists$ const c và n_0 sao cho $T(n) \leq c.f(n), \ \forall n \geq n_0$.

Chương trình chạy với thời gian $O(f(n))$ ta nói nó phát triển tỷ lệ với $f(n)$. Khi nói $T(n)$ là $O(f(n))$ thì $f(n)$ là chặn trên của $T(n)$.

Để nói chặn dưới của $T(n)$ ta dùng ký hiệu Ω.

Ta nói $T(n)$ là $\Omega(g(n))$ nếu $\exists$ const c, n_0 sao cho $T(n) \geq c.g(n), \forall n \geq n_0$.

Ví dụ : Để kiểm tra $T(n) = n^3 + 2n^2$ là $\Omega(n^3)$ ta đặt $c = 1$ thì $T(n) \geq c.n^3, \forall n = 0, 1, ...$ $(n_o = 0)$.

* Sự trái ngược của tỷ lệ phát triển :

Ta giả sử các chương trình có thể đánh giá bằng cách so sánh các hàm thời gian của chúng với các hằng tỷ lệ không đáng kể. Khi đó ta nói chương trình có thời gian chạy $O(n^2)$. Nếu chương trình 1 chạy mất $100.n^2$ thời gian (mili giây) thì chương trình 2 chạy mất $5.n^3$ thời gian, thì ta có tỷ số thời gian của 2 chương trình là $5.n^3/100.n^2 = n/20$, nghĩa là khi $n = 20$ thì thời gian chạy 2 chương trình là bằng nhau, khi $n < 20$ thì chương trình 2 chạy nhanh hơn chương trình 1. Do đó khi $n > 20$ thì nên dùng chương trình 1.

Ví dụ : Có 4 chương trình có 4 độ phức tạp khác nhau được biểu diễn trong bảng dưới đây.

Thời gian chạy T(n)	Kích thước bài toán tối đa cho 10^3s	Kích thước bài toán tối đa cho 10^4s	Tỷ lệ tăng về kích thước
$100.n$	10	100	10.0 lần
$5.n^2$	14	45	3.2 lần
$n^{3/2}$	12	27	2.3 lần
2^n	10	13	1.3 lần

Giả sử trong 10^3s thì 4 chương trình giải các bài toán có kích thước tối đa trong cột 2. Nếu có máy tốt tốc độ tăng lên 10 lần thì kích thước tối đa tương ứng của 4 chương trình trình bày ở cột 3. Tỉ lệ hai cột 1,2 ghi ở cột 4. Như vậy nếu đầu tư về tốc độ 10 lần thì chỉ thu lợi có 30% về kích thước bài toán nếu dùng chương trình có độ phức tạp $O(2^n)$.

IV. CÁCH TÍNH THỜI GIAN CHẠY CHƯƠNG TRÌNH :

1. Qui tắc tổng:

Giả sử $T_1(n)$ và $T_2(n)$ là thời gian chạy chương trình P_1 và P_2 tương ứng được đánh giá là $O(f(n))$ và $O(g(n))$. Khi đó $T_1(n) + T_2(n)$ sẽ là $O(\max(f(n),g(n)))$ (chạy xong chương trình P_1 thì chạy P_2).

Chứng minh:

Theo định nghĩa O(f(n)) và O(g(n)) thì $\exists\ c_1, n_1, c_2, n_2$ sao cho

$T_1(n) \le c_1.f(n)\ \ \forall\ n \ge n_1$; $T_2(n) \le c_2.g(n)\ \ \forall\ n \ge n_2$.

Đặt $n_0 = \max(n_1, n_2)$

Nếu $n \ge n_o$ thì $T_1(n) + T_2(n) \le (c_1 + c_2).\max(f(n),g(n))$.

2. Qui tắc tích:

$T_1(n).\ T_2(n)$ là O(f(n).g(n)).

Chứng minh : tương tự như tổng.

Ví dụ : Có 3 chương trình có thời gian chạy tương ứng là $O(n^2)$, $O(n^3)$, O(n.logn). Thế thì thời gian chạy 3 chương trình đồng thời là $O(\max(n^2, n^3, n\log n))$ sẽ là $O(n^3)$.

Nói chung thời gian chạy một dãy cố định các bước là thời gian chạy lớn nhất của một bước nào đó trong dãy. Cũng có trường hợp có 2 hay nhiều bước có thời gian chạy không tương xứng (không lớn hơn mà cũng không nhỏ hơn). Khi đó qui tắc tính tổng phải được tính trong từng trường hợp.

Ví dụ : $f(n) = \begin{cases} n^4 & \text{nếu n chẵn} \\ n^2 & \text{nếu n lẻ} \end{cases}$

$g(n) = \begin{cases} n^2 & \text{nếu n chẵn} \\ n^3 & \text{nếu n lẽ} \end{cases}$

Thời gian chạy là O(max(f(n),g(n))) là n^4 nếu n chẵn và n^3 nếu n lẻ.

Nếu $g(n) \le f(n)$, $\forall\ n \ge n_o$, n_o là const nào đó thì O(f(n)+g(n)) sẽ là O(f(n)).

Ví dụ : $O(n^2 + n) = O(n^2)$

Trước khi đưa ra qui tắc chung để phân tích thời gian chạy của chương trình thì ta xét ví dụ đơn giản sau.

Ví dụ : Xét chương trình Bubble dùng sắp dãy số nguyên theo chiều tăng.

```
Procedure Bubble (var A: array [1..n] of integer);
Var i, j, temp : integer ;
Begin
        For i := 2 to n do
          For j :=  n downto  i do
          If A[j-1] > A[j] then
         Begin
             temp := A[j-1] ;
             A[j-1] := A[j] ;
```

6 A[j] := temp ;

End ;

End ;

<u>Phân tích</u> :

- N là số phần tử - kích thước của bài toán. Mỗi lệnh gán từ dòng 4 - > dòng 6 mất 3 đơn vị thời gian, theo qui tắc tính tổng sẽ là O(max(1,1,1) = O(1).

- Vòng If và For lồng nhau, ta phải xét từ trong ra ngoài. Đối với điều kiện sau If phải kiểm tra O(1) thời gian. Ta không chắc thân lệnh If từ 4 - 6 có thực hiện hay không. Vì xét trong trường hợp xấu nhất nên ta giả thuyết là các lệnh từ 4 - 6 đều có thực hiện. Vậy nhóm If từ các lệnh 3 -6 làm mất O(1) thời gian.

- Ta xét vòng lặp ngoài từ 2 - 6. Nguyên tắc chung của vòng lặp: thời gian vòng lặp là tổng thời gian mỗi lần lặp trong thân vòng lặp. ít nhất là O(1) cho mỗi lần lặp khi chỉ số tăng. Số lần lặp từ 2 - 6 là n - i +1

Vậy theo qui tắc tích : O((n - i +1), 1) là O(n -i +1).

- Ta xét vòng ngoài cùng chứa các lệnh của chương trình. Lệnh 1 làm n-1 lần, tốn n-1 đơn vị thời gian. Vậy tổng thời gian chạy của chương trình bị chặn dưới bởi 1 thời gian cố định là :

$$\sum_{i=2}^{n}(n-i+1) = n*(n-1)/2 \text{ tức là } O(n^2)$$

Tuy nhiên không có qui tắc đầy đủ để phân tích chương trình.

Nói chung thời gian chạy của 1 lệnh hoặc 1 nhóm lệnh có thể là 1 hàm của kích thước các input hoặc 1 hay nhiều biến. Nhưng chỉ có n - kích thước của bài toán là thông số cho phép đối với thời gian chạy của chương trình.

3. <u>Qui tắc tính thời gian chạy</u>

a) Thời gian chạy của mỗi lệnh gán, read, write có giả thiết là O(1).

b) Thời gian chạy của 1 dãy lệnh xác định theo qui tắc tổng; nghĩa là thời gian chạy của dãy là thời gian lớn nhất của 1 lệnh nào đó trong dãy lệnh.

c) Thời gian chạy lệnh If là thời gian thực hiện lệnh điều kiện cộng với thời gian kiểm tra điều kiện.

Thời gian thực hiện lệnh If có cấu trúc If then else là thời gian kiểm tra điều kiện cộng với thời gian lớn nhất của 1 trong 2 lệnh rẽ nhánh true và false.

d) Thời gian thực hiện vòng lặp là tổng thời gian thực hiện thân vòng lặp và thời gian kiểm tra kết thúc vòng lặp.

e) Gọi thủ tục:Nếu chương trình có các thủ tục và không có thủ tục nào là đệ qui thì ta có thể tính thời gian chạy cùng một lúc, bắt đầu từ các thủ tục không gọi đến các thủ tục khác. Tất nhiên phải có ít nhất 1 thủ tục như vậy trong trường hợp này, nếu không thì phải có thủ tục đệ qui. Sau đó ta có thể đánh giá thời gian chạy của các thủ tục có gọi, đến các thủ tục không chứa lời gọi đã được đánh giá. Cứ như thế ta lại đánh giá thời gian chạy của các thủ tục có lời gọi đến các thủ tục đã đánh giá, nghĩa là mỗi thủ tục được đánh giá sau khi đánh giá hết các thủ tục mà được nó gọi.

Nếu có thủ tục đệ qui thì không thể tìm được thứ tự của tất cả các thủ tục sao cho mỗi thủ tục chỉ gọi đến các thủ tục đã đánh giá. Khi đó ta phải lập 1 liên hệ giữa mỗi thủ tục đệ qui với 1 hàm thời gian chưa biết T(n) trong đó n là kích thước của đối số của thủ tục. Lúc đó ta có thể nhận được sự truy hồi đối với T(n), nghĩa là 1 phương trình diễn tả T(n) qua các T(k) với các giá trị k khác nhau.

Ví dụ : Xét chương trình đệ qui tính n giai thừa (n!), trong đó n là kích thước của hàm nêu trên.

```
Function Fact (n:integer) : LongInt ;
Begin
   1    If n <= 1 then
   2       Fact := 1
        Else
   3       Fact := n*fact (n-1)
End ;
```

Phân tích:

Ta ký hiệu T(n) là thời gian chạy để tính hàm Fact(n).

Thời gian chạy đối với các dòng 1, 2 là O(1) và đối với dòng 3 là O(1) + T(n-1). Vậy với các hằng c, d nào đó ta có phương trình:

$$T(n) = \begin{cases} c + T(n-1) & \text{nếu } n > 1 \\ d & \text{nếu } n \le 1 \end{cases}$$

Giải phương trình :

Giả sử n > 2, ta có thể khai triển T(n-1) trong công thức :

$$T(n) = 2.c + T(n-2) \qquad \text{nếu } n > 2$$

Sau đó ta lại thay T(n-2) = c + T(n-3) ta được.

$$T(n) = 3.c + T(n-3) \qquad \text{nếu } n > 3$$

.....

$$T(n) = i.c + T(n-i) \qquad \text{nếu } n > i$$

Cuối cùng ta thay i = n - 1, ta được

$$T(n) = c(n-1) + T(1) = c(n-1) + d$$

Kết luận T(n) là O(n).

V. SỰ PHÂN LỚP CÁC THUẬT TOÁN :

Như đã được chú ý ở trên, hầu hết các thuật toán đều có một tham số chính là N, Thông thường đó là số lượng các phần tử dữ liệu được xử lý mà ảnh hưởng rất nhiều tới thời gian chạy. Tham số N có thể là bậc của 1 đa thức, kích thước của 1 tập tin được sắp xếp hay tìm kiếm, số nút trong 1 đồ thị...Hầu hết tất cả thuật toán trong bài giảng này có thời gian chạy tiệm cận tới 1 trong các hàm sau :

1. Hầu hết tất cả các chỉ thị của các chương trình đều được thực hiện một lần hay nhiều nhất chỉ một vài lần. Nếu tất cả các chỉ thị của cùng 1 chương trình có tính chất này thì chúng ta sẽ nói rằng thời gian chạy của nó là hằng số. Điều này hiển nhiên là mục tiêu phấn đấu để đạt được trong việc thiết kế thuật toán.

2. logN

Khi thời gian chạy của chương trình là *logarit*, tức là thời gian chạy chương trình tiến chậm khi N lớn dần. Thời gian chạy loại này xuất hiện trong các chương trình mà giải 1 bài toán lớn bằng cách chuyển nó thành bài toán nhỏ hơn, bằng cách cắt bỏ kích thước bớt 1 hằng số nào đó. Với mục đích của chúng ta, thời gian chạy có được xem như nhỏ hơn 1 hằng số "lớn". Cơ số của *logarit* làm thay đổi hằng số đó nhưng không nhiều: Khi n là 1000 thì logN là 3 nếu cơ số là 10; là 10 nếu cơ số là 2 ; khi N là 1000000, logN được nhân gấp đôi. Bất cứ khi nào N được nhân gấp đôi, logN được tăng lên thêm một hằng số, nhưng logN không được nhân gấp đôi tới khi N tăng tới N^2.

3. N

Khi thời gian chạy của chương trình là *tuyến tính*, nói chung đây là trường hợp mà một số lượng nhỏ các xử lý được làm cho mỗi phần tử dữ liệu nhập.

Khi N là 1.000.000 thì thời gian chạy cũng cỡ như vậy.

Khi N được nhân gấp đôi thì thời gian chạy cũng được nhân gấp đôi. Đây là tình huống tối ưu cho 1 thuật toán mà phải xử lý N dữ liệu nhập (hay sản sinh ra N dữ liệu xuất).

4. NlogN

Đây là thời gian chạy tăng dần lên cho các thuật toán mà giải 1 bài toán bằng cách tách nó thành các bài toán con nhỏ hơn, kế đến giải quyết chúng 1 cách độc lập và sau đó tổ hợp các lời giải. Bởi vì thiếu 1 tính từ tốt hơn (có lẽ là "tuyến tính logarit" ?), chúng ta nói rằng thời gian chạy của thuật toán như thế là "NlogN".

Khi N là 1000000, NlogN có lẽ khoảng 6 triệu.

Khi N được nhân gấp đôi, thời gian chạy bị nhân lên nhiều hơn gấp đôi (nhưng không nhiều lắm).

5. N^2

Khi thời gian chạy của 1 thuật toán là *bậc hai*, trường hợp này chỉ có ý nghĩa thực tế cho các bài toán tương đối nhỏ. Thời gian bình phương thường tăng lên trong các thuật toán mà xử lý tất cả các cặp phần tử dữ liệu (có thể là 2 vòng lặp lồng nhau).

Khi N là 1000 thì thời gian chạy là 1000000.

Khi N được nhân đôi thì thời gian chạy tăng lên gấp 4 lần.

6. N^3

Tương tự, một thuật toán mà xử lý một bộ 3 của các phần tử dữ liệu (có lẽ 3 vòng lặp lồng nhau) có thời gian chạy bậc 3 và cũng chỉ có ý nghĩa thực tế trong các bài toán nhỏ.

Khi N là 100 thì thời gian chạy là 1.000.000.

Khi N được nhân đôi thì thời gian chạy tăng lên gấp 8 lần.

7. 2^n

Một số ít thuật toán có thời gian chạy lũy thừa lại thích hợp trong 1 số trường hợp thực tế, mặc dù các thuật toán như thế là "sự ép buộc thô bạo" để giải bài toán.

Khi N là 20 thì thời gian chạy xấp xỉ là 1.000.000

Khi N là gấp 2 thì thời gian chạy được nâng lên lũy thừa 2.

Thời gian chạy của 1 chương trình cụ thể đôi khi là một hằng số nhân với các số hạng nói trên cộng thêm một số hạng nhỏ hơn. Các giá trị của hằng số và các số hạng phụ thuộc vào các kết quả của sự phân tích và các chi tiết cài đặt. Hệ số của hằng số liên quan tới số chỉ thị bên trong vòng lặp : ở 1 tầng tùy ý của thiết kế thuật toán thì phải cẩn thận giới hạn số chỉ thị như thế. Với N lớn thì các hằng số đóng vai trò chủ chốt, với N nhỏ thì các số hạng cùng đóng góp vào và sự so sánh thuật toán sẽ khó khăn hơn. Ngoài những hàm vừa nói trên cũng còn có 1 số hàm khác, ví dụ như 1 thuật toán với N^2 phần tử dữ liệu nhập mà có thời gian chạy là bậc 3 theo N thì sẽ được phân lớp như 1 thuật toán $N^{3/2}$. Một số thuật toán có 2 giai đoạn phân tách thành các bài toán con và có thời gian chạy xấp xỉ với $N\log^2 N$.

VI. cÁc công thức truy hồi cơ sở :

Phần lớn các thuật toán đều dựa trên việc phân rã đệ qui một bài toán lớn thành các bài toán nhỏ hơn, rồi dùng các lời giải của các bài toán nhỏ để giải bài toán ban đầu. Thời gian chạy của các thuật toán như thế được xác định bởi kích thước và số lượng các bài toán con và giá phải trả của sự phân rã. Trong phần này ta quan sát các phương pháp cơ sở để phân tích các thuật toán như thế và trình bày một vài công thức chuẩn thường được áp dụng trong việc phân tích nhiều thuật toán.

Tính chất rất tự nhiên của 1 chương trình đệ qui là thời gian chạy cho dữ liệu nhập có kích thước N sẽ phụ thuộc vào thời gian chạy cho các dữ liệu nhập có kích thước nhỏ hơn : điều này được diễn dịch thành 1 công thức toán học gọi là quan hệ truy hồi. Các công thức như thế mô tả chính xác tính năng của các thuật toán tương ứng, do đó để có được thời gian chạy chúng ta phải giải các bài toán truy hồi. Bây giờ chúng ta chú ý vào các công thức chứ không phải các thuật toán.

Công thức 1 :

Công thức này thường dùng cho các chương trình đệ qui mà có vòng lặp duyệt qua dữ liệu nhập để bỏ bớt 1 phần tử.

$C_n = C_{n-1} + n$, với n >= 2 và $C_1 = 1$

Chứng minh :

C_n khoảng $n^2/2$. Để giải 1 công thức truy hồi như trên, chúng ta lần lượt áp dụng chính công thức đó như sau :

$$\begin{aligned} C_n &= C_{n-1} + n \\ &= C_{n-2} + (n-1) + n \\ &= \ldots \\ &= C_1 + 2 + \ldots + (n-2) + (n-1) + n \\ &= 1 + 2 + \ldots + n \\ &= n(n+1)/2 \end{aligned}$$

Công thức 2 :

Công thức này dùng cho chương trình đệ qui mà chia dữ liệu nhập thành 2 phần trong mỗi bước.

$C_n = C_{n/2} + 1$, với n >= 2 và $C_1 = 0$

Chứng minh :

C_n khoảng logn. Phương trình này vô nghĩa trừ phi n chẵn hay chúng ta giả sử rằng n/2 là phép chia nguyên : bây giờ chúng ta giả sử rằng $n = 2^m$ để cho công thức luôn luôn có nghĩa. Chúng ta viết như sau :

$$\begin{aligned} C_{2^m} &= C_{2^{m-1}} + 1 \\ &= C_{2^{m-2}} + 2 \\ &= C_{2^{m-3}} + 3 \\ &= \ldots\ldots \\ &= C_{2^{m-m}} + m \\ &= m = \log n \end{aligned}$$

Công thức chính xác cho n tổng quát thì phụ thuộc vào biểu diễn nhị phân của n, nói chung C_n khoảng logn với mọi n.

Công thức 3 :

Công thức này dùng cho chương trình đệ qui mà chia đôi dữ liệu nhập nhưng có thể kiểm tra mỗi phần tử của dữ liệu nhập.

$C_n = C_{n/2} + n$, với $n >= 2$ và $C_1 = 0$

Chứng minh :

C_n khoảng 2n. Tương tự trên, công thức này chính là tổng n + n/2 + n/4 + ... (dĩ nhiên điều này chỉ chính xác khi n là lũy thừa của 2).

Nếu dãy là vô hạn, thì đây là 1 chuỗi hình học đơn giản mà được ước lượng chính xác là 2n. Trong trường hợp tổng quát lời giải chính xác phụ thuộc vào biểu diễn nhị phân của n.

Công thức 4 :

Công thức này dùng cho chương trình đệ qui mà duyệt tuyến tính xuyên qua dữ liệu nhập, trước, trong, hay sau khi dữ liệu nhập được chia đôi.

$C_n = 2C_{n/2} + n$, với $n >= 2$ và $C_1 = 0$

Chứng minh :

C_n khoảng nlogn. Công thức này áp dụng cho nhiều thuật toán theo phương pháp "chia để trị".

$$
\begin{aligned}
C_{2^m} &= C_{2^{m-1}} + 2^m \\
\frac{C_{2^m}}{2^m} &= \frac{C_{2^{m-1}}}{2^{m-1}} + 1 \\
&= \frac{C_{2^{m-2}}}{2^{m-2}} + 1 + 1 \\
&= \frac{C_{2^{m-m}}}{2^{m-m}} + m \\
&= C_{2^0} + m = m \\
&\Rightarrow C_n = m2^m = n\log n
\end{aligned}
$$

Lời giải cho công thức này rất giống như trong công thức 2, nhưng phải chia 2 vế của công thức cho 2^n trong bước thứ hai.

Công thức 5 :

Công thức này dùng cho chương trình đệ qui mà tách dữ liệu thành 2 phần.

$C_n = 2C_{n/2} + 1$, với $n >= 2$ và $C_1 = 0$

Chứng minh :

C_n khoảng 2n. Chứng minh giống như công thức 4.

Các biến dạng của những công thức này chẳng hạn như điều kiện khác nhau hay các số hạng thêm vào khác nhau một ít, có thể ước lượng bằng cách dùng cũng một kỹ thuật như trên. Mặc dù vậy, chúng ta cũng nên chú ý 1 quan hệ truy hồi dường như tương tự với một quan hệ đã biết thì đôi khi lại khó giải hơn rất nhiều.

VII. giải phương trÌnh truy hồi :

Để giải phương trình truy hồi có nhiều cách giải khác nhau, ở đây chúng tôi trình bày cách giải phương trình truy hồi bằng cách truy về phương trình đặc trưng. Chúng tôi dùng cách giải này để viết chương trình giải tự động phương trình truy hồi.

a) Ta xét phương trình truy hồi thuần nhất tuyến tính với các hệ số không đổi sau đây :

$a_0t_n + a_1t_{n-1} + ... + a_kt_{n-k} = 0$ (VII.1)

trong đó t_i(i=n, n-1,..., n-k) là các ẩn số cần tìm.

Tuyến tính vì các t_i chỉ có bậc nhất, thuần nhất vì vế phải bằng không và các hệ số $a_0, a_1,..., a_k$ là không đổi vì không phụ thuộc vào n.

Sau khi giả thiết $t_n = x^n$ ta đưa (VII.1) về dạng:

$a_0x^n + a_1x^{n-1} +...+ a_kx^{n-k} = 0$

hay $x^{n-k}(a_0x^k + a_1x^{k-1} +...+ a_k) = 0$

Rõ ràng x = 0 là nghiệm hiển nhiên, nhưng ta quan tâm nghiệm phương trình $a_0x^k + a_1x^{k-1} +...+ a_k = 0$ (VII.2) và đây chính là phương trình đặc trưng bậc k của phương trình truy hồi (VII.1)

Giả sử $r_1, r_2,..., r_k$ là k nghiệm của phương trình (VII.2) và chúng khác nhau (có thể phức). Dễ dàng kiểm tra:

$$t_n = \sum_{i=1}^{k} c_i r_i^n$$

Với $c_1, c_2,..., c_k$ là các hằng xác định từ k điều kiện ban đầu.

Ví dụ 1 :

Xét phương trình truy hồi:

$$t_n - 3t_{n-1} - 4t_{n-2} = 0$$

Điều kiện ban đầu : $t_0 = 1$; $t_1 =1$

Phương trình đặc trưng tương ứng của nó là:

$x^2 - 3x - 4 = 0$

có nghiệm bằng -1 và 4. Vậy nghiệm tổng quát là :

$$t_n = c_1(-1)^n + c_2(4)^n$$

Theo điều kiện ban đầu (khi n =0 và n = 1) ta có :

$c_1 + c_2 = 1 = t_0$

$- c_1 + 4c_2 =1$

Vậy $c_1 = 3/5$, $c_2 = 2/5$. Ta được $t_n = - [4^n - (-1)^n] /5$

Ví dụ 2 : (phương trình Fibonacci)

$t_n = t_{n-1} + t_{n-2} \qquad n \geq 2$

Điều kiện : $t_0 = 0, t_1 = 1$

Viết lại phương trình trên :

$t_n - t_{n-1} - t_{n-2} = 0$

Phương trình đặc trưng tương ứng :

$x^2 - x - 1 = 0$

Nghiệm : $r_1 = (1+\sqrt{5})/2,\ r_2 = (1-\sqrt{5})/2$

Nghiệm tổng quát : $t_n = c_1 r_1^n + c_2 r_2^n$

Từ điều kiện ban đầu :

$c_1 + c_2 = 0 \qquad (n = 0)$
$r_1 c_1 + r_2 c_2 = 1 \qquad (n = 1)$

Ta có $c_1 = 1/\sqrt{5}$,

$c_2 = -1/\sqrt{5}$

Vậy: $t_n = (r_1 n - r_2 n)/\sqrt{5}$

Giả sử các nghiệm phương trình đặc trưng là không phân biệt, P(x) là 1 đa thức.

$P(x) = a_0 x^k + a_1 x^{k-1} + ... + a_k$

và r là nghiệm kép.

Với mỗi r > k, ta xét đa thức bậc n được xác định như sau :

$h(x) = x\,[x^{n-k} P(x)] = a_0 n x^n + a_1 (n-1) x^{n-1} + ... + a_k (n-k) x^{n-k}$

Đặt q(x) là đa thức thỏa điều kiện

$P(x) = (x-r)^2\, q(x)$

Ta có :

$h(x) = x[(x-r)^2\, x^{n-k} q(x)] = x[2(x-r)x^{n-k} q(x) + (x-r)^2 [x^{n-k} q(x)]]$

Rõ ràng h(r) = 0, do đó

$a_0 n r^n + a_1 (n-1) x^{n-1} + ... + a_k (n-k)\, r^{n-k} = 0$

Nghĩa là $t_n = n r_n$ cũng là nghiệm của (5.13). Tổng quát hơn, nếu nghiệm r trùng nhau m lần (r bội m) thì

$t_n = r^n,\ t_n = n r^n,\ t_n = n^2 r^n, ...,\ t_n = n^{m-1} r^n$

cũng là các nghiệm của (5.13). Nghiệm tổng quát (nghiệm chung) là tổ hợp tuyến tính của các nghiệm riêng này và nghiệm riêng khác của phương trình đặc trưng. K hằng được xác định từ các điều kiện ban đầu.

Ví dụ 3 : Xét phương trình

$t_n = 5t_{n-1} - 8t_{n-2} + 4t_{n-3} \quad n \geq 3$

với các điều kiện $t_0 = 0$, $t_1 = 1$, $t_2 = 2$

Ta viết lại phương trình:

$t_n - 5t_{n-1} + 8t_{n-2} - 4t_{n-3} = 0$

và phương trình đặc trưng tương ứng là:

$x^3 - 5x^2 + 8x - 4 \quad = 0$

hay $(x-1)(x-2)^2 \quad = 0$

Ta có các nghiệm 1 (có bội 1), 2 (có bội 2). Vậy nghiệm tổng quát là :

$t_n = c_1 1^n + c_2 2^n + c_3 n 2^n$

Các điều kiện ban đầu cho trước là:

$c_1 + c_2 = 0$	khi	$n = 0$
$c_1 + 2c_2 + 2c_3 = 1$	khi	$n = 1$
$c_1 + 4c_2 + 8c_3 = 2$	khi	$n = 2$

Từ đây ta tìm ra $c_1 = -2$, $c_2 = 2$, $c_3 = -1/2$. Vậy $t_n = 2^{n+1} - n2^{n-1} - 2$

b) Phương trình truy hồi không thuần nhất:

Ta xét phương trình có dạng sau :

$a_0t_n + a_1t_{n-1} + \ldots + a_kt_{n-k} = b^nP(n) \qquad \text{(VII.3)}$

Trong đó vế trái như trong (VII.1), vế phải là $b^nP(n)$ với b là hằng và P(n) là đa thức.

Ví dụ 4 :

$t_n - 2t_{n-1} = 3^n$

thì b = 3 và P(n) = 1 là đa thức bậc 0.

Bằng phép biến đổi đơn giản ta có thể chuyển ví dụ này về dạng (VII.1) là phương trình truy hồi thuần nhất. Trước hết ta nhân 2 vế cho 3 ta được :

$3t_n - 6t_{n-1} = 3^{n+1} \qquad (1)$

Sau đó ta thay n ra n + 1 trong phương trình ban đầu:

$t_{n+1} - 2t_n = 3^{n+1} \qquad (2)$

Cuối cùng, lấy (2) - (1) , ta thu được (có cùng vế phải 3^{n+1}), ta được:

$t_{n+1} - 5t_n + 6t_{n-1} = 0$

Đến đây ta có thể giải phương trình đã trình bày ở mục a.

Phương trình đặc trưng của nó là :

$x^2 - 5x + 6 = 0$

hay $(x-2)(x-3) = 0$

Trực giác cho ta thấy rằng thành phần (x-2) tương ứng với vế trái phương trình ban đầu; còn (x-3) là biểu hiện kết quả của việc biến đổi và trừ vế phải.

Ví dụ 5 :

$$t_n - 2t_{n-1} = (n+5)3^n$$

Sự biến đổi có phức tạp như sau :

- Nhân 2 vế cho 9
- Thay n bởi n+2
- Thay n bởi n+1,sau đó nhân cho -6
- Ta được kết quả :

$$9t_n - 18t_{n-1} = (n + 5)\ 3^{n+2}$$
$$t_{n+2} - 2t_{n+1} = (n + 7)\ 3^{n+2}$$
$$-6t_{n+1} + 12t_n = -6(n + 6)\ 3^{n+1}$$

Cộng 3 phương trình lại ta được :

$$t_{n+2} - 8t_{n+1} + 21t_n - 18t_{n-1} = 0$$

Phương trình đặc trưng.

$$x^2 - 8x^2 + 21x - 18 = 0 \text{ hay } (x-2)\ (x-3)^2 = 0$$

Ta lại thấy (x-2) tương ứng vế trái phương trình ban đầu và $(x-3)^2$ là kết quả của sự biến đổi.

Như vậy chúng ta có thể nói rằng để giải phương trình truy hồi không thuần nhất có dạng (VII.3) ta chỉ cần giải phương trình đặc trưng sau.

$$(a_0x^k + a_1x^{k-1} + ... + a_k)\ (x-b)^{d+1} = 0 \qquad \text{(VII.4)}$$

Ví dụ 6 : (bài toán tháp Hà Nội)

Phương trình truy hồi cho bài toán chuyển n đĩa này có dạng:

$$t_n = \begin{cases} 1 & \text{nếu } n = 1 \\ 2t_{n-1} + 1 & \text{nếu } n > 1 \end{cases}$$

hay $t_n = 2t_{n-1} + 1$, $n > 1$ với $t_0 = 0$

Ta viết lại phương trình :

$$t_n - 2t_{n-1} = 1$$

Và thấy nó có dạng (VII.3) với b = 1, P(n) = 1 bậc 0

Phương trình đặc trưng là (x-2) (x-1) = 0

Vậy : $t_n = c_11^n + c_22^n$. Từ $t_0 = 0$ để tìm t_1 ta viết :

$$t_1 = 2t_o + 1 = 1$$

Vậy $c_1 + c_2 = 0, n = 0$

$c_1 + 2c_2 = 1, n = 1$

Suy ra $c_1 = -1$, $c_2 = 1$. Vậy $t_n = 2^n-1$

Ta nhận thấy từ $t_n = c_1 1^n + c_2 2^n$ cũng có thể kết luận t_n có $O(2^n)$.

Ví dụ 7 :

$$t_n = 2t_{n-1} + n$$

hay $\quad t_n - 2t_{n-1} = n$

ở đây $\quad b = 1$, $P(n) = n$ bậc 1

Ta có phương trình đặc trưng:

$$(x-2)(x-1)^2 = 0$$

Vậy nghiệm tổng quát là :

$$t_n = c_1 2^n + c_2 1^n + c_3 n 1^n$$

Nếu $t_n > 0$ với mỗi n thì ta có thể kết luận $T(n)= O(2n)$,

Bây giờ ta xét phương trình truy hồi không thuần nhất tổng quát hơn.

$a_o t_n + a_1 t_{n-1} + ... + a_k t_{n-k} = b_1^n p_1(n) + b_2^n p_2(n) + ...$ (VII.5)

Trong đó b_i là các hằng khác nhau và $p_i(n)$ là các đa thức bậc d_i của n.

Bằng cách biến đổi gần như tương tự với dạng phương trình (VII.1), ta có thể viết được phương trình đặc trưng cho dạng (VII.1)5) như sau :

$(a_o x^k + a_1 x^{k-1} + ... + a_k)(x-b_1)^{d1+1}(x-b_2)^{d2+1}= 0$ (VII.6)

Ví dụ 8 : Giải phương trình

$$t_n = 2t_{n-1} + n + 2^n \qquad n \geq 1$$

với $t_o = 0$, ta có

$$t_n - 2t_{n-1} = n + 2^n$$

có dạng (VII.1)5) với $b_1 = 1$, $p_1(n) = n$, $b_2 = 2$, $p_2(n) = 1$. Bậc của $p_1(n)$ là 1, bậc của $p_2(n)$ là 0. Vậy phương trình đặc trưng là :

$$(x-2)(x-1)^2 (x-2) = 0$$

Cả hai nghiệm 1 và 2 đều có bội là 2. Vậy nghiệm tổng quát của phương trình truy hồi là :

$$t_n = c_1 1^n + c_2 n 1^n + c_3 2^n + c_4 n 2^n$$

Sử dụng dạng truy hồi $t_n = 2t_{n-1} + n + 2^n$ với $t_o = 0$ ta có thể tính được t_1, t_2 và t_3 và từ đó xác định được các hệ số c_1, c_2, c_3 và c_4 qua hệ sau:

$c_1 + c_3 = 0$ khi $n = 0$

$c_1 + c_2 + 2c_3 + 2c_4 = 3$ $\quad n = 1$

$c_1 + 2c_2 + 4c_3 + 8c_4 = 12$ $\quad n = 2$

$c_1 + 3c_2 + 8c_3 + 24c_4 = 35$ $\quad n = 3$

Kết quả cuối cùng là :

$t_n = -2 - n + 2^{n+1} + n2^n$

Dễ dàng nhận thấy t_n có $O(n2^n)$

Ví dụ 9 :

Giả sử n là lũy thừa của 2. Tìm T(n) từ phương trình:

$T(n) = 4T(n/2) + n, \qquad n > 1$

Thay n bằng 2k (với k = logn), ta có T(2k) = 4T(2k-1) + 2k. Khi đó ta có thể viết:

$t_k = 4t_{k-1} + 2^k$

Nếu $t_k = T(2k) = T(n)$. Ta đã biết cách giải phương trình truy hồi mới này.

Phương trình đặc trưng của nó là :

$(x-4)(x-2) = 0$

và từ đó ta có $t_k = c_1 4^k + c_2 2^k$

Đặt n ngược lại thay cho k, ta tìm được :

$T(n) = c_1 n^2 + c_2 n$

Do đó T(n) có $0(n^2)$ khi n là lũy thừa của 2.

Ví dụ 10 :

$T(n) = 4t(n/2) + n^2$, n > 1, lũy thừa của 2. Đặt $n = 2^k$, ta có.

$T(2^k) = 4T(2^{k-1}) + 4^k$

Phương trình đặc trưng (x-4)2 = 0 và ta có $t_k = c_1 4^k + c_2 k 4^k$

Vậy $T(n) = c_1 n2 + c_2 n^2 \log n$ và T(n) là $0(n^2 \log n)$ với n lũy thừa 2.

Ví dụ 11 :

$T(n) = 2T(n/2) + n\log n, \qquad n > 1$

Sau khi thay n = 2k, ta có $T(2^k) = 2T(2^{k-1}) + k2^k$ và

$t_k = 2t_{k-1} + k2^k$

Phương trình đặc trưng $(x-2)^3 = 0$ và $t_k = c_1 2^k + c_2 k 2^k + c_3 k^2 2^k$.

Vậy $T(n) = c_1 n + c_2 n\log n + c_3 n\log^2 n$ có $0(n\log^2 n)$, n lũy thừa 2.

Ví dụ 12 :

$T(n) = 3T(n/2) + cn$ (c là const, $n = 2^k > 1$)

Ta sẽ nhận được :

$T(2^k) = 3T(2^{k-1}) + c2^k$; $t_k = 3t_{k-1} + c2^k$

Phương trình đặc trưng là (x-3) (x-2) = 0, và do đó.

$Tk = c_1 3^k + c_2 2^k$; $T(n) = c_1 3^{\log n} + c_2 n$

Do alogb = bloga nên $T(n) = c_1 n^{\log 3} + c_2 n$ có $0(n^{\log 3})$,n lũy thừa 2.

c) <u>Phương trình truy hồi có hệ số biến đổi</u>

Ta xét ví dụ cụ thể :

$T(1) = 6$

$T(n) = nT^2(n/2)$, n > 1,n lũy thừa của 2

(hệ số ở vế phải là biến n)

Trước hết ta đặt $t_k = T(2^k)$ và từ đấy ta có :

$t_k = 2^k t^2{}_{K-1}$ $\quad$ k > 0

to = 6

Để lập phương trình truy hồi mới không có hệ số biến đổi ta đặt $V_k = lgt_k$, ta được :

$V_k = K + 2V_{k-1}$ $\quad$ k >0

$V_o = lg6$

Nghĩa là ta có hệ phương trình dạng (VI.3)

Phương trình đặc trưng sẽ là

(x-2) (x-1)2 = 0

và do đó :

$V_k = c_1 2^k + c_2 1^k + c_3 k 1^k$

Từ $V_o = 1 + lg3$, $V_1 = 3+2lg3$ và $V_2 = 8 + 4lg3$ ta tìm ra

$c_1 = 3 + lg3$, $c_2 = -2$ và $c_3 = -1$.

Vậy $\quad V_3 = (3 + lg3)\ 2^k - K -2$

Cuối cùng, sử dụng $tk = 2^{vk}$ và T(n) = tlgn, ta được :

$$T(n) = \frac{2^{3n-2}3^n}{n}$$

chương Ii

ĐỆ QUI

I. <u>KhÁi niệm</u> :

Đệ qui là 1 công cụ rất thường dùng trong khoa học máy tính và trong toán học để giải quyết các vấn đề. Trước hết, chúng ta hãy khảo sát thế nào là một vấn đề có đệ qui qua ví dụ sau:

Tính S(n) = 1 +2 +3 +4+ ... +n-1+n =S(n-1) + n

Ta nhận thấy rằng, công thức trên có thể diễn đạt lại như sau:

S(n) = S(n-1) + n, và

S(n-1) = S(n-2) + (n-1)

.....

S(2) = S(1) + 2

S(1) = 1

Như vậy, *một vấn đề có đệ qui là vấn đề được định nghĩa lại bằng chính nó.*

Một cách tổng quát, một chương trình đệ qui có thể được biểu diễn như bộ P gồm các mệnh đề cơ sở S (không chứa P) và bản thân P:

$P \equiv P(S_i, P)$

<u>Để tính S(n)</u>: ta có kết quả của S(1), thay nó vào S(2), có S(2) ta thay nó vào S(3)...., cứ như vậy có S(n-1) ta sẽ tính được S(n)

Cũng như các lệnh lặp, các thủ tục đệ qui cũng có thể thực hiện các tính toán không kết thúc, vì vậy ta phải xét đến vấn đề kết thúc các tính toán trong giải thuật đệ qui. Rõ ràng 1 thủ tục P được gọi đệ qui chỉ khi nào thỏa 1 điều kiện B, và dĩ nhiên điều kiện B này phải không được thỏa mãn tại 1 thời điểm nào đó. Như vậy mô hình về các giải thuật đệ qui là:

$P \equiv$ if (B) $P(S_i, P)$

hay $P \equiv P(S_i,$ if (B) $P)$.

Thông thường trong các vòng lặp while, để đảm bảo cho vòng lặp kết thúc ta phải định nghĩa một hàm f(x) (x là 1 biến trong chương trình) sao cho nó phải trả về trị bằng 0 tại một thời điểm nào đó. Tương tự như vậy, chương trình đệ qui cũng có thể được chứng minh là sẽ dừng bằng cách chứng minh rằng hàm f(x) sẽ giảm sau mỗi lần thực hiện. Một cách thường làm là kết hợp một giá trị n với P và gọi P một cách đệ qui với giá trị tham số là n-1. Điều kiện B bây giờ là n > 0 thì sẽ đảm bảo được sự kết thúc của giải thuật đệ qui. Như vậy, ta có mô hình đệ qui mới:

$P(n) \equiv$ if (n >0) $P(S_i, P(n-1))$

Hay $P \equiv P(S_i,$ if (n>0) $P(n-1))$

II. HÀm đệ qui vÀ Stack:

Một chương trình C thường gồm có hàm main() và các hàm khác. Khi chạy chương trình C thì hàm main() sẽ được gọi chạy trước, sau đó hàm main() gọi các hàm khác, các hàm này trong khi chạy có thể gọi các hàm khác nữa. Khi một hàm được gọi, thì một khung kích hoạt của nó được tạo ra trong bộ nhớ stack. Khung kích hoạt này chứa các biến cục bộ của hàm và mẩu tin hoạt động của hàm. Mẩu tin hoạt động chứa địa chỉ trở về của hàm gọi nó và các tham số khác.

	Biến cục bộ
Mẩu tin hoạt động	Địa chỉ trở về
	Thông số khác

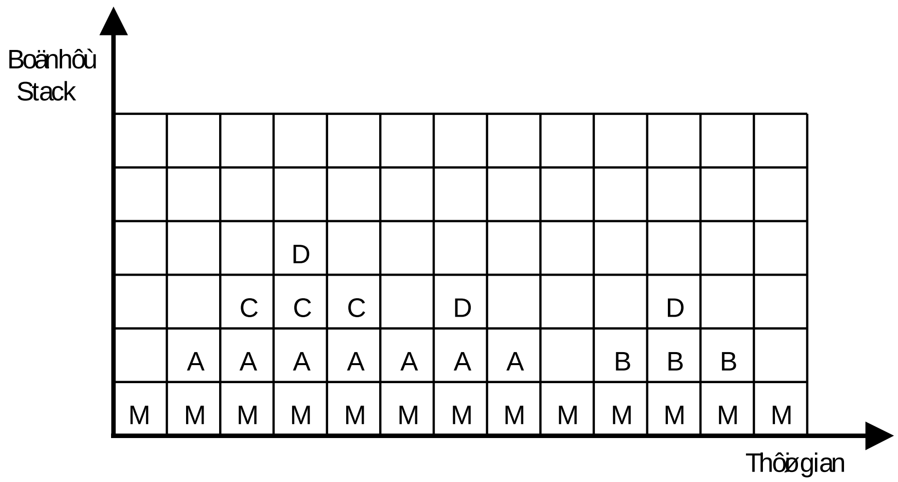
Khung kích hoạt

Sau khi hàm được gọi đã thi hành xong thì chương trình sẽ thực hiện tiếp dòng lệnh ở địa chỉ trở về của hàm gọi nó, đồng thời xóa khung kích hoạt của hàm đó khỏi bộ nhớ.

Giả sử ta có cơ chế gọi hàm trong một chương trình C như sau:

main() { ...... A(); .....; B(); ....; }	A() {.....; C(); ....; D(); }	B() {.....; D(); }	C() {......; D(); .....; }	D() {........; ....; }

Hình sau đây cho ta thấy sự chiếm dụng bộ nhớ stack khi chạy chương trình C như mô tả ở trên.



Tương tự với trường hợp hàm đệ qui, khi gọi đệ qui lẫn nhau thì một loạt các khung kích hoạt sẽ được tạo ra và nạp vào bộ nhớ Stack. Cấp đệ qui càng cao thì số khung kích hoạt trong Stack càng nhiều, do đó, có khả năng dẫn đến tràn Stack (Stack overflow). Trong nhiều trường hợp khi lập trình, nếu có thể được, ta nên gỡ đệ qui cho các bài toán.

III. VÍ DỤ

Ví dụ 1: Hàm giai thừa:

$$n! = \begin{cases} 1*2*3*......*(n-1)*n & , n>0 \\ 1 & , n=0 \end{cases}$$

$$n! = \begin{cases} n*(n-1)! & , n>0 \\ 1 & , n= 0 \end{cases}$$

Nhận xét:

- Theo công thức trên, ta nhận thấy trong định nghĩa của n giai thừa (n!) có định nghĩa lại chính nó nên hàm giai thừa có đệ qui.

- Điều kiện dừng tính hàm giai thừa là n=0, khi đó n! = 1

- Hàm đệ qui:

```
long giaithua(int n)
{
  if (n == 0||n==1)
    return(1);
  else
    return(n * giaithua(n-1));
}
```

hay:

```
long giaithua(int n)
{  return ((n==0) ? 1 : n*giaithua(n-1));
}
```

- Hàm không đệ qui:

```
long giaithua (int n)
{  long gt=1;
   for (int i=1; i<=n; i++)
        gt= gt * i ;
  return (gt);
}
```

Ví dụ 2: Hàm FIBONACCI:

$$F_n = \begin{cases} 1 & ; n =0,1 \\ F_{n-1} + F_{n-2} & ; n>1 \end{cases}$$

Nhận xét:

- Theo định nghĩa trên, hàm Fibonacci có lời gọi đệ qui.

- Quá trình tính dừng lại khi n= 1

- <u>Hàm đệ qui</u>:

```
long fib(int n)
{  if (n==0 || n==1)
     return 1 ;
   else return(fib(n-1) + fib(n-2));
}
```

- <u>Hàm không đệ qui</u>:

```
long fib(int n)
{  long kq, Fn_1, Fn_2;
   kq = 1;
   if (n > 1)
   {
      Fn_1 = 1;
      Fn_2 = 1;
      for (int i=2; i<=n; i++)
      {
          kq = Fn_1 + Fn_2 ;
          Fn_2 = Fn_1;
          Fn_1 = kq;
      }
   }
      return (kq);
}
```

<u>Ví dụ 3</u>: Bài toán Tháp Hà nội

Có 3 cột A, B, C. Cột A hiện đang có n dĩa kích thước khác nhau, dĩa nhỏ ở trên dĩa lớn ở dưới. Hãy dời n dĩa từ cột A sang cột C (xem cột B là cột trung gian) với điều kiện mỗi lần chỉ được dời 1 dĩa và dĩa đặt trên bao giờ cũng nhỏ hơn dĩa đặt dưới.

- <u>Giải thuật đệ qui</u>: Để dời n dĩa từ cột A sang cột C (với cột B là cột trung gian), ta có thể xem như :

+ Dời (n-1) dĩa từ cột A sang cột B (với cột C là cột trung gian)

+ Dời dĩa thứ n từ cột A sang cột C

+ Dời (n-1) dĩa từ cột B sang cột C (với cột A là cột trung gian)

- <u>Chương trình</u>:

void hanoi (int n, char cotA, char cotC, char cotB)

```
{
  if(n == 1)
      printf("\n%s%c%s%c", "   chuyen dia 1 tu cot ", cotA, " den cot ", cotC);
  else
 {
    hanoi(n-1, cotA, cotB, cotC);
    printf("\n%s%d%s%c%s%c", "   chuyen dia ", n, " tu cot ", cotA,
    " den cot ", cotC);
    hanoi(n-1, cotB, cotC, cotA);
 }
}
```

IV. CÁC THUẬT TOÁN LẦN NGƯỢC:

Trong lập trình, đôi khi ta phải xác định các thuật giải để tìm lời giải cho các bài toán nhất định nhưng không phải theo một luật tính toán cố định, mà bằng cách thử-và-sai. Cách chung là phân tích thử-và-sai thành những công việc cục bộ. Thông thường công việc này được thực hiện trong dạng đệ qui và bao gồm việc thăm dò một số hữu hạn các nhiệm vụ nhỏ. Trong bài giảng này ta không tìm hiểu các qui tắc tìm kiếm tổng quát, mà chỉ tìm những nguyên lý chung để chia việc giải bài toán thành những việc nhỏ và ứng dụng của sự đệ qui là chủ đề chính. Trước hết, ta minh họa kỹ thuật căn bản bằng cách xét bài toán mã đi tuần.

Ví dụ 1. Bài toán mã đi tuần.

Cho bàn cờ có n x n ô. Một con mã được phép đi theo luật cờ vua, đầu tiên nó được đặt ở ô có toạ độ x_0, y_0. Câu hỏi là, nếu có thì hãy tìm cách sao cho con mã đi qua được tất cả các ô của bàn cờ, mỗi ô đi qua đúng 1 lần.

* Luật đi của con mã trên bàn cờ: Tại một ô có tọa độ cột x_0, hàng y_0 (x_0,y_0) trên bàn cờ, con mã có 1 trong 8 nước đi như sau:

	3		2	
4				1
		Con Mã		
5				8
	6		7	

Hình 2.1 8 nước đi có thể của con mã xuất phát từ cột x_0, hàng y_0.

Với tọa độ bắt đầu (x_0,y_0), có tất cả 8 ô (u,v) mà con mã có thể đi đến được. Chúng được đánh số từ 1 đến 8 trong hình 2.1

Phương pháp đơn giản để có được u,v từ x,y là cộng các chênh lệch cột, dòng về tọa độ được lưu trong 2 mảng a và b. Các giá trị trong 2 mảng a, b đã được khởi động thích ứng như sau:

Ta xem như có 1 hệ trục tọa độ (Oxy) ngay tại vị trí (x_0,y_0) của con mã, thì :

+ Vị trí 1 mà con mã có thể đi được là :

$u= x_0 + 2, v = y_0 + 1$

+ Vị trí 2 mà con mã có thể đi được là :

$u= x_0 + 1, v = y_0 + 2$

+ Vị trí 3 mà con mã có thể đi được là :

$u= x_0 + (-1), v = y_0 + 2$ ....

Như vậy, mảng a và b có giá trị sau:

int a[8] = {2, 1, -1,-2, -2, -1, 1, 2};
int b[8] = {1, 2, 2, 1, -1, -2,-2, -1};

* Cách biểu diễn dữ liệu: Để mô tả được bàn cờ, ta dùng ma trận BanCo theo khai báo sau:

#define KICHTHUOC 5 // Kích thước của bàn cờ

int BanCo[KICHTHUOC][KICHTHUOC]; // Tổ chức bàn cờ là mãng hai chiều

Ta thể hiện mỗi ô cờ bằng 1 số nguyên để đánh đấu ô đó đã được đi qua chưa, vì ta muốn lần dò theo quá trình di chuyển của con mã. Và ta qui ước như sau:

BanCo [x][y]=0 ; ô (x,y) chưa đi qua
BanCo [x][y]=i ; ô (x,y) đã được đi qua ở nước thứ i ($1 \le i \le n^2$)

* **Thuật giải**:

Cách giải quyết là ta phải xét xem có thể thực hiện một nước đi kế nữa hay không từ vị trí x_0, y_0. Thuật giải để thử thực hiện nước đi kế.

```
void thử nước đi kế
{  khởi động các chọn lựa có thể đi
 do
{      chọn một nước đi;
       if chấp nhận được
       {  ghi nhận nước đi;
          if bàn cờ chưa đầy
          {      thử nước đi kế tại vị trí vừa ghi nhận được;
                 if không được
                   xóa nước đi trước
```

```
            }
        }
    } while (không đi được && còn nước đi)
}
```

* **Nhận xét**:

- Để xác định tọa độ (u,v) của nước đi kế (0 ≤ i ≤ 7), ta thực hiện như sau:

u = x + a[i] ; v = y + b[i]

- Điều kiện để nước đi kế chấp nhận được là (u,v) phải thuộc bàn cờ và con mã chưa đi qua ô đó, nghĩa là ta phải thỏa các điều kiện sau:

(0 ≤ u <KICHTHUOC && 0 ≤ v< KICHTHUOC && BanCo[u][v]==0)

- Ghi nhận nước đi thứ n, nghĩa là BanCo [u][v] = n; còn bỏ việc ghi nhận nước đi là BanCo [u][v] = 0

- Bàn cờ đầy khi ta đã đi qua tất cả các ô trong BanCo, lúc đó :

n = KICHTHUOC ^{2}.

Qua nhận xét trên, ta có thuật giải chi tiết hơn như sau:

```
void thu_nuoc_di(int n, int x, int y, int &q) // thử 8 cách đi của con mã tại
                                              // nước thứ n xuất phát từ ô (x,y)
{  int u,v, q1;
   khởi động các chọn lựa có thể đi
  do
  {      u = x + a[i] ;
         v = y + b[i];
         if  (0 ≤ u <  KICHTHUOC   && 0 ≤ v <  KICHTHUOC
                                   && BanCo [u][v] == 0 )
         {       BanCo [u][v] = n;
                 if n < KICHTHUOC*KICHTHUOC
               { thu_nuoc_di (n+1, u, v, q1)
                 if !q1
                        BanCo [u][v] = 0;
               }
               else q1=TRUE;
         }
  } while (q1==FALSE && còn nước đi)
}
```

* **Chương trình mã đi tuần**:

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

```
#include <stdlib.h>
#include <conio.h>
#define KICHTHUOC 5     // Kích thước của bàn cờ
#define TRUE 1
#define FALSE 0               // Tổ chức bàn cờ là mãng hai chiều
int BanCo[KICHTHUOC][KICHTHUOC];
                              // 8 cách đi của con mã
int a[8] = {2,  1, -1,-2, -2, -1, 1,  2};
int b[8] = {1,  2,  2, 1, -1,  -2,-2, -1};
void innuocdi(int BanCo[][KICHTHUOC])
{
  int i, j;
  char c;
  randomize();
  textmode(C80);
  textbackground(BLACK);
  textcolor(1);
  for(i = 0; i < KICHTHUOC; i++)
    for(j = 0; j < KICHTHUOC; j++)
    {
      gotoxy(23+5*j, 8+2*i);
      textcolor(1 + random(15));
      if(BanCo[i][j] == 0 ? printf("  ") : cprintf("%2d", BanCo[i][j]));
    }
}
// Hàm thu_nuoc_di giúp đi nước thứ n xuất phát từ ô(x, y)
void try(int n, int x, int y, int &q)
{
  int k=-1,u,v, q1;
  do
  {  k++; q1=FALSE;
    u = x+a[k];
    v = y+b[k];
    if (u >= 0 && u < KICHTHUOC  && v >= 0 && v < KICHTHUOC)
     if (BanCo[u][v] ==0)
     {
      // Đi nước thứ n
       BanCo[u][v] = n;
```

```
        if (n < KICHTHUOC*KICHTHUOC)  // D/kien dung, di duoc nuoc cuoi
        { try(n+1, u, v, q1);                 // Buoc de qui, goi di nuoc n+1
            if (q1 == FALSE)
               BanCo[u][v]=0;
        }
        else q1=TRUE;
       }
   } while (q1==FALSE && k <7);
   q=q1;
}
void vebanco()
{
printf("\n\t\t\t     CHUONG TRINH MA DI TUAN\n");
printf("\n\t\tKich thuoc ban co %dx%d", KICHTHUOC, KICHTHUOC);
printf("\n\n\t\t     0      1      2      3     4       5     6      7");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 0 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 1 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 2 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 3 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 4 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 5 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 6 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 7 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
}
void main()
{
   int i, j,q;
   clrscr();
   vebanco();
   for(i = 0; i < KICHTHUOC; i++)
      for(j = 0; j < KICHTHUOC; j++)
```

```
            BanCo[i][j] = 0;
    // Chon nuoc di dau tien va goi ham de qui de di nuoc thu hai
    BanCo[0][0] = 1;
    try(2, 0, 0,q);
    if (q==FALSE)
        printf ("\n Khong co loi giai");
    else innuocdi(BanCo);
}
```

* Bảng sau đây cho ta một số lời giải tương ứng với các vị trí đầu và kích thước của bàn cờ:

- Kích thước bàn cờ = 5

+ Vị trí bắt đầu (1,1)

1	6	15	10	21
14	9	20	5	16
19	2	7	22	11
8	13	24	17	4
25	18	3	12	23

+ Vị trí bắt đầu (3,3)

23	10	15	4	25
16	5	24	9	14
11	22	**1**	18	3
6	17	20	13	8
21	12	7	2	19

- Kích thước bàn cờ = 8

+ Vị trí bắt đầu (1,1)

1	60	39	34	31	18	9	64
38	35	32	61	10	63	30	17
59	2	37	40	33	28	19	8
36	49	42	27	62	11	16	29
43	58	3	50	41	24	7	20
48	51	46	55	26	21	12	15
57	44	53	4	23	14	25	6
52	47	56	45	54	5	22	13

<u>Ví dụ 2</u>: Bài toán tám hoàng hậu.

Bài toán tám hàng hậu được mô tả như sau: tám hoàng hậu được đặt lên bàn cờ vua sao cho không bà nào có thể chiếm lấy các bà khác.

* Theo luật của cờ vua, một hoàng hậu có thể chiếm lấy các quân khác nằm ở cùng dòng, hay cùng cột, hay cùng các đường chéo. Do đó, ta suy ra rằng mỗi cột chỉ có thể chứa một hoàng hậu và chỉ 1 mà thôi. Ta qui ước hoàng hậu thứ 0 sẽ đặt ở cột 0, hoàng hậu thứ 1 sẽ đặt ở cột 1,.., hoàng hậu thứ 7 sẽ đặt ở cột 7. Như vậy, việc chọn chỗ cho hoàng hậu thứ i là tìm vị trí dòng j có thể có trên cột i.

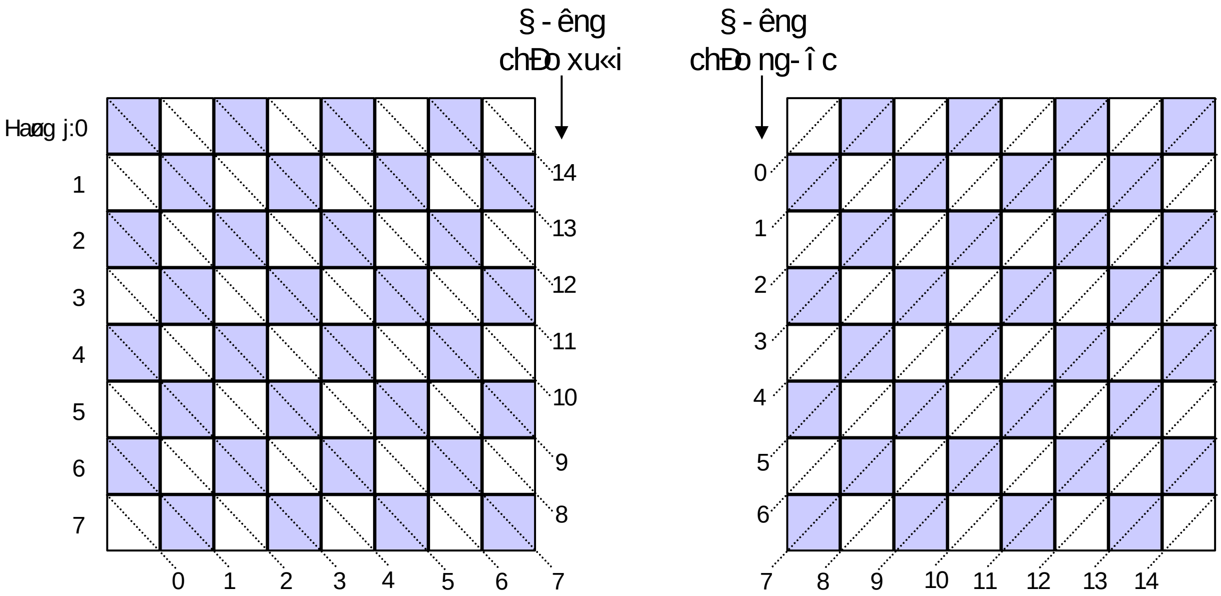
Sau đây là hình minh họa cho một lời giải của bài toán: (0 6 4 7 1 3 5 2)

0	Q							
1					Q			
2								Q
3						Q		
4			Q					
5							Q	
6		Q						
7				Q				

* <u>Cách biểu diễn dữ liệu</u>:

Bàn cờ 8x8 có 8 hàng, 8 cột, 15 đường chéo xuôi, 15 đường chéo ngược, ta qui ước 1 con hậu chỉ có thể đặt trên 1 cột i và hàng j nào đó của bàn cờ. Khi đó, các vị trí trên đường chéo xuôi và đường chéo ngược của bàn cờ không thể dùng để đặt các con hậu khác. Ta lưu ý rằng các phần tử cùng nằm trên 1 đường chéo xuôi của bàn cờ thỏa mãn biểu thức sau (cột i - hàng j + 7 = hằng số), và các phần tử cùng nằm trên 1 đường chéo ngược của bàn cờ thỏa mãn biểu thức sau (cột i + hàng j = hằng số) như hình sau:

đường chéo xuôi **đường chéo ngược**



Như vậy, ta sẽ xây dựng cấu trúc dữ liệu sau để lưu trữ dữ liệu:

int hang_trong[8] ; // hàng trống còn có thể đặt hoàng hậu

int cheo_xuoi[15];

// duong cheo xuoi co the dat hoang hau. Cac phan tu tren duong cheo xuoi //thoa (cot i -hang j +7 = hangso)

int cheo_nguoc[15];

// duong cheo nguoc co the dat hoang hau. Cac phan tu tren duong cheo // nguoc thoa (cot i +hang j = hangso)

int loi_giai[8] ; // loi giai chua ket qua.

với:

- hang_trong[j] = TRUE nghĩa là chưa có con hậu nào trên hàng thứ j
- cheo_xuoi[k] = TRUE nghĩa là chưa có con hậu nào trên đường chéo xuôi thứ k
- cheo_nguoc[k] = TRUE nghĩa là chưa có con hậu nào trên đường chéo ngược thứ k

Ví dụ:

Các vị trí trên đường chéo xuôi thứ 12 : 5 - 0 +7 = 6 -1 +7 = 7 -2 + 7 = 12

Các vị trí trên đường chéo ngược thứ 12: 7 + 5 = 6 + 6 = 5 + 7 = 12

-loi_giai[i] chỉ vị trí của hoàng hậu ở cột thứ i

*** Thuật giải:**

void chon_vi_tri (int i) // tìm vị trí thích hợp cho hoàng hậu thứ i
{ khởi động các vị trí có thể chọn cho hoàng hậu thứ i
do
{ chọn vị trí kế;
if an toàn

```
                  { đặt hoàng hậu
                    if chưa đặt hết 8 hoàng hậu
                     { chon_vi_tri (i+1);
                       if không thành công
                           dời hoàng hậu đi
                     }
                   }
             } while  (không thành công  && chưa đặt hết các vị trí)
   }
```

*** <u>Nhận xét</u>**: Với các dữ liệu đã cho, thì:

- Điều kiện an toàn là điều kiện sao cho hoàng hậu thứ i (cột i) nằm trên hàng j sao cho hàng j và các đường chéo đi qua ô (j,i) chưa bị chiếm giữ bởi các hoàng hậu khác; nghĩa là nó phải thỏa biểu thức logic:

    hang_trong[j] && cheo_xuoi [i-j+7] && cheo_nguoc[i+j]

- Đặt hoàng hậu sẽ được thể hiện bởi:

  loi_giai[i] = j ;

  hang_trong[j] = FALSE ; cheo_xuoi [i-j+7] =FALSE; cheo_nguoc[i+j] = FALSE;

- Dời hoàng hậu đi sẽ được thể hiện bởi:

  hang_trong[j]=TRUE ; cheo_xuoi [i-j+7] =TRUE;

  cheo_nguoc[i+j] = TRUE;

- Điều kiện chưa đặt hết các hoàng hậu : i < 7

- Để biết được đặt hoàng hậu thứ i có thành công hay không, ta dùng thêm 1 tham số hình thức biến q. Nếu đặt thành công thì q = TRUE, ngược lại q=FALSE.

* <u>Chương trình</u>: Qua nhận xét trên, ta có chương trình của bài toán 8 hoàng hậu như sau:

```
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#define TRUE 1
#define FALSE 0
// tim 1 loi giai cho bai toan 8 hoang hau
int  hang_trong[8] ;  // hàng trong con co the dat hoang hau
int  cheo_xuoi[15];   // duong cheo xuoi co the dat hoang hau. Cac phan tu
                      // tren duong cheo xuoi thoa (cot i -hang j +7 = hangso)
int  cheo_nguoc[15];         // duong cheo nguoc co the dat hoang hau. Cac
phan tu
```

```
                        // tren duong cheo nguoc thoa (cot i +hang j  = hangso)
int  loi_giai[8] ;        // loi giai chua ket qua.
int  i, q;
void in_loigiai(int *loigiai)
{
   int i, j;
   char c;
   randomize();
   textmode(C80);
   textbackground(BLACK);
   clrscr();
   textcolor(1 + random(15));
printf("\n\t\t\t     CHUONG TRINH 8 HOANG HAU\n");
printf("\n\n\t\t     0      1      2      3     4      5     6      7     ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 0 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 1 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 2 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 3 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 4 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 5 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 6 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 7 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
    for(i = 0; i < 8; i++)
   {
     gotoxy(24+5*i,8+2*loigiai[i] );
     textcolor(1 + random(15));
     cprintf("Q");
   }
   gotoxy(13, 25);
   printf("Nhan phim bat ky de thoat ...");
   getche();
```

```
}
void chon_vi_tri ( int i, int &q)
{ int j= -1;
  do
  {
    j++;
   q=FALSE;
   if (hang_trong[j] && cheo_xuoi[i-j+7] && cheo_nguoc[i+j])
   { loi_giai[i]= j;
       hang_trong[j]=FALSE; cheo_xuoi[i-j+7] = FALSE;
       cheo_nguoc[i+j]=FALSE;
       if (i < 7)
     {
         chon_vi_tri (i+1,q);
         if (q==FALSE)
         {
             hang_trong[j]=TRUE; cheo_xuoi[i-j+7] = TRUE;
             cheo_nguoc[i+j]=TRUE;
         }
     }
     else q=TRUE;
   }
  } while ((q==FALSE) && (j<7)); //Chưa thành công và chưa hết vị trí
                                            // thì tiếp tục
}
void main (void)
{
  /*Khoi dong tat ca cac hang, duong cheo xuoi, duong cheo nguoc deu co
    the dat hoang hau */
  for(i = 0; i < 8; i++)
    hang_trong[i] = TRUE;
  for(i = 0; i < 15; i++)
  {
    cheo_xuoi [i] = TRUE;
    cheo_nguoc[i] = TRUE;
  }
  // Goi ham de qui de bat dau dat HoangHau0 (hoang hau o cot 0)
  chon_vi_tri (0,q);
```

```
  in_loigiai(loi_giai);
}
```

<u>Lưu ý</u>:

- Trên đây là thuật giải tìm một lời giải cho bài toán 8 hoàng hậu. Tuy nhiên, ta có thể mở rộng để có thể tìm mọi lời giải cho bài toán. Sơ đồ tổng quát cho giải thuật back-tracking để tìm mọi lời giải cho bài toán:

```
void chon_vi_tri (int  i)
{ int j;
  for (j=0; j < m; j++)
  {       chọn bước thứ j;
          if được
          { ghi nhận
            if i < n
                  chon_vi_tri (i+1) ;
             else in lời giải;
             bỏ việc ghi nhận;
          }
  }
}
```

- <u>Chương trình tìm mọi lời giải cho bài toán tám hoàng hậu</u>:

```
#include <conio.h>
#include <stdio.h>
#include <stdlib.h>
#define TRUE 1
#define FALSE 0          // tim tat ca loi giai cho bai toan 8 hoang hauint
hang_trong[8] ;          // cot trong con co the dat hoang hauint
cheo_xuoi[15];           // duong cheo xuoi co the dat hoang hau
int  cheo_nguoc[15];     // duong cheo nguoc co the dat hoang hau
int  loi_giai[8] ;       // loi giai chua ket qua
int  i, q;
int SoLoiGiai =0;
void in_loigiai(int *loigiai)
{
int i, j;
char c;
randomize();
textmode(C80);
textbackground(BLACK);
```

```
clrscr();
textcolor(1 + random(15));
cprintf("\n          CHUONG TRINH 8 HOANG HAU\n   ");
printf("\n                     Loi giai thu %d", ++SoLoiGiai);
printf("\n\n\t\t     0      1      2      3     4      5     6     7     ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 0 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 1 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 2 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 3 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 4 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 5 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 6 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
printf("\n\t\t 7 │     │     │     │     │     │     │     │     │ ");
printf("\n\t\t    +-----+-----+-----+-----+-----+-----+-----+-----+ ");
      for(i = 0; i < 8; i++)
        {
          gotoxy(24+5*i,8+2*loigiai[i] );
          textcolor(1 + random(15));
          cprintf("Q");
        }
      gotoxy(13, 25);
      printf("Nhan phim <ESC> de thoat, nhan phim bat ky de tiep tuc ...");
      c = getche();
      if(c == 27)
          exit(1);
      }
```

void chon_vi_tri(int i)

```
{ int j;
  for (j=0; j<8; j++)
  {
    if (hang_trong[j] && cheo_xuoi[i-j+7] && cheo_nguoc[i+j])
    { loi_giai[i]= j;
```

```
            hang_trong[j]=FALSE; cheo_xuoi[i-j+7] = FALSE;
            cheo_nguoc[i+j]=FALSE;
            if (i < 7)
                chon_vi_tri (i+1);
            else in_loigiai(loi_giai);
            hang_trong[j]=TRUE; cheo_xuoi[i-j+7] = TRUE;
            cheo_nguoc[i+j]=TRUE;
          }
        }  // for
      } // chon_vi_tri
void main(void)
{
  /* Khoi dong tat ca cac cot, duong cheo xuoi, duong cheo nguoc deu co
      the dat hoang hau */
  for(i = 0; i < 8; i++)
    hang_trong[i] = TRUE;
  for(i = 0; i < 15; i++)
  {
    cheo_xuoi [i] = TRUE;
    cheo_nguoc[i] = TRUE;
  }
  // Goi ham de qui de bat dau dat HoangHau0 (hoang hau o hang 0)
 chon_vi_tri (0);
}
```

BÀi tập

1. Viết một hàm đệ quy và không đệ quy để tính giá trị của hàm

$$P_n(x) = \begin{cases} 1 & , n=0 \\ x & , n=1 \\ P_{n-1}(x) - P_{n-2}(x) & , n >= 2 \end{cases}$$

2. Viết chương trình Tháp Hà Nội.

   Mô tả chương trình Tháp Hà Nội : Ta có 3 cọc A, B, C và n dĩa được xếp trên cọc A sao cho dĩa nhỏ trên dĩa lớn.

   Hãy viết chương trình di chuyển n dĩa từ cọc A sang cọc C với cọc B làm trung gian, theo điều kiện :

   - Mỗi lần chỉ di chuyển một dĩa
   - Bao giờ dĩa nhỏ cũng nằm trên dĩa lớn

3. Viết chương trình tìm mọi lời giải cho bài toán mã đi tuần.

CHƯƠNG III

DANH SÁCH TUYẾN TÍNH

KHÁI NIỆM :

Danh sách là một tập hợp n phần tử a_0, a_1, a_2,....., a_{n-1}, mỗi phần tử có kiểu đơn giản hoặc kiểu dữ liệu có cấu trúc.

Tính tuyến tính của danh sách thể hiện ở sự ràng buộc giữa các phần tử trong danh sách với nhau, ví dụ như từ vị trí của phần tử a_i ta sẽ tìm được giá trị của phần tử a_{i+1}.

I. Định nghĩa:

Danh sách tuyến tính là 1 dãy các phần tử có cùng kiểu dữ liệu được sắp xếp liên tiếp nhau trong bộ nhớ.

0100		
0102	0	int
0104	1	danh sách n phần tử
	n-1	

Bộ nhớ

Đặc điểm của danh sách tuyến tính:

- Kích thước của danh sách sẽ được cấp phát theo khai báo.

- Các phần tử của danh sách nằm liên tục nhau trong bộ nhớ, giữa các phần tử này không có khoảng trống.

- Tiêu biểu cho danh sách đặc là dãy (array). Để cài đặt danh sách tuyến tính, ta dùng mảng 1 chiều.

➬ **Khai báo:** Ta khai báo cấu trúc list là một mẩu tin (struct) có 2 field:

- n : cho biết số phần tử hiện có trong danh sách. Nếu n ==0 thì có nghĩa là danh sách rỗng.

- nodes : là mảng 1 chiều, mỗi phần tử của mảng là 1 phần tử trong danh sách.

```
#define MAXLIST 100
typedef struct list
{int n;                        // số nút của danh sách
  int nodes[MAXLIST];          // nodes là mãng 1 chiều
};
list ds;  // biến ds thuộc kiểu struct list
```

Ví dụ: Khai báo 1 danh sách họ tên học viên của 1 lớp học, có tối đa 50 học viên.

```
#define MAXLIST  50
typedef struct list
{int n;
  char dshv[MAXLIST][31];       // họ tên học viên có tối đa 30 ký tự
  };
  list ds;
```

II. CÁC PHÉP TOÁN TRÊN DANH SÁCH TUYẾN TÍNH:

Để đơn giản, các phép toán sau đây sẽ được thao tác trên danh sách các số nguyên với khai báo như sau:

```
#define MAXLIST 100
typedef struct list
{int n;                          // số nút của danh sách
  int nodes[MAXLIST];      // danh sách là mãng 1 chiều
};
list ds;  // biến ds thuộc kiểu struct list
```

1. Phép toán Empty: kiểm tra xem danh sách có rỗng hay không?

```
int Empty(list plist)
{
        return plist.n==0 ;
}
```

2. Phép toán Full: kiểm tra xem danh sách đã đầy chưa?

```
int Full(list plist)
{
        return (plist.n==MAXLIST ? 1 : 0);
}
```

3. Phép thêm vào : Thêm một phần tử có nội dung là info vào vị trí thứ i của danh sách.

Nếu i ==0 : thêm phần tử vào đầu danh sách

Nếu i ==ds.n+1 : thêm phần tử vào cuối danh sách.

Lưu ý:

Khi thêm một phần tử vào danh sách, ta phải kiểm tra xem danh sách đã đầy hay chưa?

```
int Insert_item(struct list &plist, int i, int info)
{
  int j;
  if(i < 0 || i > plist.n+1 || Full(plist))
```

```
        return 0;
    if (i==0) i=1;
    for(j = plist.n -1; j >= i-1; j--)
            plist.nodes[j+1] = plist.nodes[j];
    plist.nodes[i-1] = info;
    plist.n ++;
    return 1;
}
```

4. <u>Phép loại bỏ</u> : Loại bỏ phần tử thứ i khỏi danh sách tuyến tính. Khi loại bỏ 1 phần tử thì danh sách phải có ít nhất một phần tử.

```
int Delete_item (struct list &plist, int i)
{
    int j;
    if(i < 0 || i > plist.n || plist.n==0) return 0;
    if (i==0) i=1;
    for(j = i;  j< plist.n ; j++)
        plist.nodes[j-1] = plist.nodes[j];
    plist.n--;
    return 1;
}
```

* Muốn loại bỏ tất cả các phần tử trong danh sách, ta chỉ cần cho ds.n=0

5. <u>Duyệt danh sách</u>: duyệt từ đầu cho đến cuối danh sách, mỗi phần tử được duyệt qua 1 lần. Giả sử ta duyệt danh sách để in giá trị các phần tử.

```
void Traverse(struct list plist)
{
    int i;
    if(plist.n == 0)
    {
        printf("\n     Danh sach khong co phan tu");
        return;
    }
    for(i = 0; i < plist.n; i++)
        printf("%8d",  plist.nodes[i]);
}
```

6. <u>Tìm kiếm</u>: tìm vị trí đầu tiên của phần tử có giá trị info trong danh sách plist. Nếu không có info trong plist thì hàm Search_info sẽ trả về giá trị -1.

```
int Search_info(struct list plist, int info)
{
    for ( int vitri =0 ; vitri <plist.n ; vitri++)
```

```
            if (plist.nodes[vitri] == info) return vitri;
        return -1;
    }
```

Lưu ý: Để nhập danh sách, ta có thể dùng giải thuật sau:

```
void Create_list(struct list &plist)
{ int i;
  printf("\nSo phan tu cua danh sach :");
  scanf("%d", &plist.n);
  for (i=0; i< plist.n; i++)
  { printf("List[%d] =", i+1);
    scanf("%d",&plist.nodes[i]);
  }
}
```

Nhận xét : Danh sách đặc dùng phương pháp truy xuất trực tiếp nên thời gian truy xuất nhanh, nhưng hiệu quả sử dụng bộ nhớ thấp. Danh sách đặc không phù hợp với phép thêm vào và loại bỏ vì mỗi lần thêm vào và loại bỏ thì chúng ta phải đổi chỗ nhiều lần. Đặc biệt trường hợp xấu nhất là khi thêm vào và loại bỏ ở đầu danh sách

Kết luận : Danh sách đặc không nên sử dụng cho các danh sách hay bị biến động. Còn đối với những danh sách thường bị biến động thì người ta chọn cấu trúc là danh sách liên kết.

Ví dụ: Tạo một menu cho phép ta thực hiện các phép toán sau trên danh sách các số nguyên:

1. Tạo danh sách
2. Liệt kê danh sách trên màn hình
3. Thêm một phần tử có giá trị info tại vị trí thứ i
   - Nếu i ==0 : thêm phần tử vào đầu danh sách
   - Nếu i ==ds.n+1 : thêm phần tử vào cuối danh sách.
4. Xóa phần tử đầu tiên có giá trị info trong danh sách
5. Xóa toàn bộ danh sách. Trước khi xóa hỏi lại người sử dụng có muốn xóa hay không? Nếu người sử dụng đồng ý "C" thì mới xóa.

*** Chương trình :**

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#define MAXLIST 100   // so phan tu toi da trong danh sach
#define TRUE 1
#define FALSE 0
```

```
typedef struct list
{
    int n;
    int nodes[MAXLIST];
};
// Phep toan empty: kiem tra danh sach co bi rong khong
int empty(struct list plist)
{
    return(plist.n == 0 ? TRUE : FALSE);
}
// Phep toan  full: kiem tra danh sach bi day khong
int full(struct list plist)
{
    return(plist.n == MAXLIST ? TRUE : FALSE);
}
// Tao danh sach
void create_list(struct list &plist)
{ int i;
  printf("\nSo phan tu cua danh sach :");
  scanf("%d", &plist.n);
  for (i=0; i < plist.n; i++)
  { printf("List[%d] =", i+1);
    scanf("%d",&plist.nodes[i]);
  }
}
// Tac vu insert_item: chen nut co noi dung info vao vi tri i
// i==0 : them vao dau danh sach
// i==plist->n +1 : them vao cuoi danh sach
void insert_item(struct list &plist, int i, int info)
{
    int j;
    if(i < 0 || i > plist.n+1)
            printf("Vi tri chen khong phu hop.");
    else
            if(full(plist))
                    printf("Danh sach bi day.");
            else
            {  if (i==0) i=1;
               for(j = plist.n -1; j >= i-1; j--)
                    plist.nodes[j+1] = plist.nodes[j];
```

```
                plist.nodes[i-1] = info;
                plist.n ++;
            }
    }
    // Tac vu delete_item: xoa nut tai vi tri i trong danh sach
    void delete_item (struct list &plist, int i)
    {
        int j;
        int temp;
        if(i <= 0 || i > plist.n)
            printf("Vi tri xoa khong phu hop.");
        else
            if(empty(plist))
                printf("Danh sach khong co phan tu.");
            else
            {
                for(j = i;  j< plist.n ; j++)
                    plist.nodes[j-1] = plist.nodes[j];
                plist.n--;
            }
    }
    // Tac vu clearlist: xoa tat ca cac nut trong danh sach
    void clearlist(struct list &plist)
    {
        plist.n = 0;
    }
    // Tac vu traverse: duyet danh sach cac so nguyen
    void traverse(struct list plist)
    {
        int i;
        if(plist.n == 0)
        {
            printf("\n     Danh sach khong co phan tu");
            return;
        }
        for(i = 0; i < plist.n; i++)
            printf("%7d",  plist.nodes[i]);
    }
/* Phep toan search: tim kiem tuyen tinh, neu khong tim thay ham nay tra
```

```
ve -1, neu tim thay ham nay tra ve vi tri tim thay */
int search_info(struct list plist, int info)
{
        int vitri = 0;
        while( vitri < plist.n && plist.nodes[vitri] != info  )
                vitri++;
        return(vitri==plist.n ? -1 : vitri+1);
}

int menu()
{ int chucnang;
clrscr();
  // menu chinh cua chuong trinh
printf("\n\n CHUONG TRINH QUAN LY DANH SACH CAC SO \n");
printf(" Cac chuc nang cua chuong trinh:\n");
printf("      1: Nhap danh sach\n");
printf("      2: Xem danh sach \n");
printf("      3: Them mot so vao vi tri thu i\n");
printf("      4: Xoa phan tu dau tien co tri info\n");
printf("      5: Xoa toan bo danh sach\n");
printf("      0: Ket thuc chuong trinh\n");
printf("  Chuc nang ban chon: ");
do
   scanf("%d", &chucnang);
while (chucnang<0 || chucnang >5);
return chucnang;
}
void main()
{
        struct list ds;
        int chucnang, vitri, info;
        char c;
        ds.n=0;
  do
  {
        clrscr();
        chucnang=menu();
                switch(chucnang)
                {
```

```
case 1:
{
        printf("\nNhap danh sach: ");
        create_list(ds);
        break;
}
case 2:
{
        printf("\nDanh sach so: ");
        traverse(ds);
        getche();
        break;
}
case 3:
{
        printf("\nVi tri them (1, 2, ...): ");
        scanf("%d", &vitri);
        printf("Gia tri: ");
        scanf("%d", &info);
        insert_item(ds, vitri, info);
        getche();
        break;
}
case 4:
{
        printf("\nGia tri so can xoa: ");
        scanf("%d", &info);
        vitri = search_info(ds, info);
        if(vitri == -1)
           printf("Khong co so %d trong danh sach",  info);
        else
          delete_item(ds, vitri);
        getche();
        break;
}
case 5:
{
        printf("\nBan co chac muon xoa hay khong (c/k):");
        c = toupper(getche());
```

```
                if( c == 'C')
                clearlist(ds);
                break;
            }
        }
    } while(chucnang != 0);
}
```

III. <u>STACK (CHỒNG)</u>:

III.1. <u>Khái niệm</u>:

Stack là một danh sách mà việc thêm vào và loại bỏ chỉ diễn ra cùng một đầu của danh sách, tức là theo cơ chế LIFO (Last In First Out). Stack gồm nhiều phần tử có cùng kiểu dữ liệu, phần tử trên cùng Stack luôn luôn có một con trỏ chỉ tới ta gọi là Stack Pointer (ký hiệu: sp).

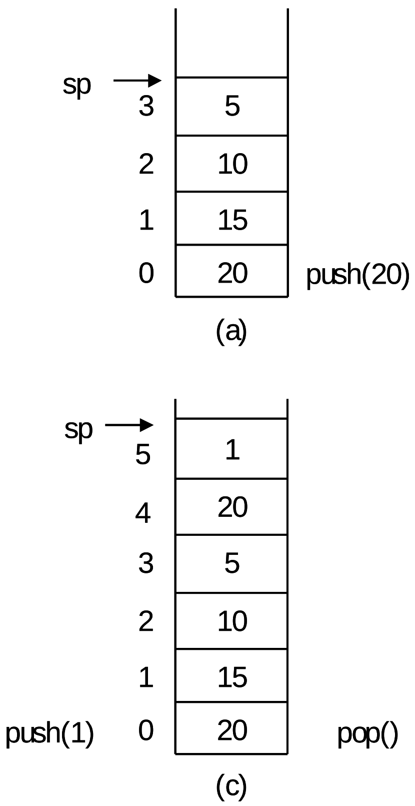
Để tạo Stack ta có hai cách : danh sách tuyến tính (mảng) hoặc danh sách liên kết (con trỏ). Trong chương này, ta chỉ quan tâm đến việc tạo Stack bằng danh sách tuyến tính.

- Stack có 2 phép toán chính :

* Push : thêm một phần tử vào đầu Stack

* Pop : xóa một phần tử khỏi Stack, trả cho chương trình gọi giá trị của phần tử vừa xóa.

Dưới đây là hình minh họa cho các phép toán push, pop trên Stack.



*** Lưu ý:** Ta khai báo biến st có kiểu cấu trúc stack như sau:

```
#define STACKSIZE 100
#define TRUE 1
#define FALSE 0
typedef struct stack
{
  int sp;
  int nodes[STACKSIZE];
} st;
```

III.2. Các phép toán trên stack:

a. Phép toán push : thêm một phần tử có giá trị x vào đầu stack

```
int push (stack &st, int x)
{
  if(st.sp == STACKSIZE-1)  return 0;
  st.nodes[++(st.sp)] = x;
  return 1;
}
```

b. Phép toán pop : loại bỏ phần tử khỏi Stack và trả về giá trị của phần tử vừa xóa; trước khi xóa, ta phải kiểm tra Stack có khác rỗng hay không.

```
int pop(stack &st, int &x)
{
  if(st.sp==-1)  return 0;
  x=st.nodes[(st.sp)--];
  return 1;
}
```

Ứng dụng của Stack:

- Stack thường được dùng trong các bài toán có cơ chế LIFO (vào sau ra trước)

- Stack cũng được dùng trong các bài toán gỡ đệ qui (chuyển một giải thuật đệ qui thành giải thuật không đệ qui)

III.3. Ví dụ:

a. Viết chương trình đổi số nguyên không âm ở hệ thập phân sang số ở hệ nhị phân.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define STACKSIZE 100
```

```
#define TRUE 1
#define FALSE 0
typedef struct stack
{
  int sp;
  int nodes[STACKSIZE];
};
int empty(struct stack st)
{
  if(st.sp == -1)
    return(TRUE);
  else
    return(FALSE);
}
int push(struct stack &st, int x)
{
  if(st.sp == STACKSIZE-1) return 0;
  st.nodes[++(st.sp)] = x;
  return 1;
}
int pop(struct stack &st, int &x)
{
  if(empty(st)) return 0;
  x=st.nodes[(st.sp)--];
  return 1;
}
void main()
{
  struct stack st;
  int  sodu;
  long  so;
  char c;
  clrscr();
   do
  {
    st.sp =- 1;  // khoi dong stack
    printf("\n\nNhap vao mot so thap phan: ");
    scanf("%ld", &so);
    do
    {
```

```
            sodu = so % 2;
            push(st, sodu);  // push so du vao stack
            so = so / 2;
        } while (so != 0);
        printf("So da doi la: ");
        while(!empty(st))
         {     pop(st,x);
               printf("%d", x);  // pop so du ra khoi stack
          }
        printf("\n\nBan co muon tiep tuc khong? (c/k): ");
        c = getche();
      } while(c == 'c' || c == 'C');
    }
```

b. Viết chương trình tính trị một biểu thức dạng hậu tố (PostFix), biết rằng mỗi số hạng là 1 ký số và các toán tử trong biểu thức gồm có: cộng(+), trừ (-), nhân (*), chia (/), lũy thừa (^).

Dạng hậu tố của biểu thức có dạng như sau:

```
82-       =  6
84-21+    =  64
23+3^     =  125
```

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <math.h>
#define TOIDA 80
#define TRUE 1
#define FALSE 0
// Khai bao stack chua cac toan hang
struct stack
{
  int sp;
  double nodes[TOIDA];
} st;
int empty(struct stack st)
{
  if (st.sp == -1)
    return(TRUE);
  else
    return(FALSE);
```

```
}
int push(struct stack &st, double x)
{
  if(st.sp == TOIDA-1) return 0
   st.nodes[++(st.sp)] = x;
   return 1;
}
int pop(struct stack &st, double &x)
{
  if(empty(st)) return 0;
   x=st.nodes[(st.sp)--];
   return 1;
}
/* Ham tinh: tính trị của hai toán hạng toanhang1 va toanhang2 qua
  phép toán toantu */
double tinh(int toantu, double toanhang1, double toanhang2)
{
  switch(toantu)
  {
    case '+':
          return(toanhang1 + toanhang2); break;
    case '-':
          return(toanhang1 - toanhang2); break;
    case '*':
          return(toanhang1 * toanhang2); break;
    case '/':
          return(toanhang1 / toanhang2); break;
    case '^':
          return(pow(toanhang1, toanhang2)); break;
    default:
      printf("%s", "toan tu khong hop le");
      exit(1);
  }
}
// Hàm dinhtri: tính một biểu thức postfix
double dinhtri(char bieuthuc[])
{
  int c, i;
  double toanhang1, toanhang2, tri;
   st.sp = -1;  // khoi dong stack
```

```
    for(i = 0; (c = bieuthuc[i]) != '\0'; i++)
      if(c>='0' && c<='9')  // c la toan hang
        push(st, (double)(c-'0'));
      else          // c la toan tu
      {
        pop(st,toanhang2);
        pop(st,toanhang1);
        tri = tinh(c, toanhang1, toanhang2);  // tinh ket qua trung gian
        push(st, tri);
      }
     pop(st,toanhang1);
     return(toanhang1) ;
}
void main()
{
   char c, bieuthuc[TOIDA];
    clrscr();
  do
   {
      printf("\n\nNhap bieu thuc postfix can dinh tri: ");
      gets(bieuthuc);
          double ketqua = dinhtri(bieuthuc) ;
      if (empty(st))
            printf("Bieu thuc %s co tri la %5.2f",  bieuthuc, ketqua);
      else
  printf("Bieu thuc sai nen khong the tinh");
      printf("\n\nTiep tuc khong ? (c/k): ");
      c = getche();
   } while(c == 'c' || c == 'C');
}
```

IV. <u>QUEUE (HÀNG ĐỢI)</u>:

IV.1. <u>Khái niệm</u>:

Queue là một danh sách hạn chế mà việc thêm vào được thực hiện ở đầu danh sách, và việc loại bỏ được thực hiện ở đầu còn lại (FIFO - First In First Out). Queue chứa các phần tử có cùng kiểu dữ liệu. Queue cũng có thể được tổ chức theo danh sách tuyến tính hoặc danh sách liên kết. Ở đây, ta chỉ tổ chức queue theo danh sách tuyến tính.

- Vị trí để loại bỏ phần tử được gọi là Front
- Vị trí để thêm vào được gọi là Rear

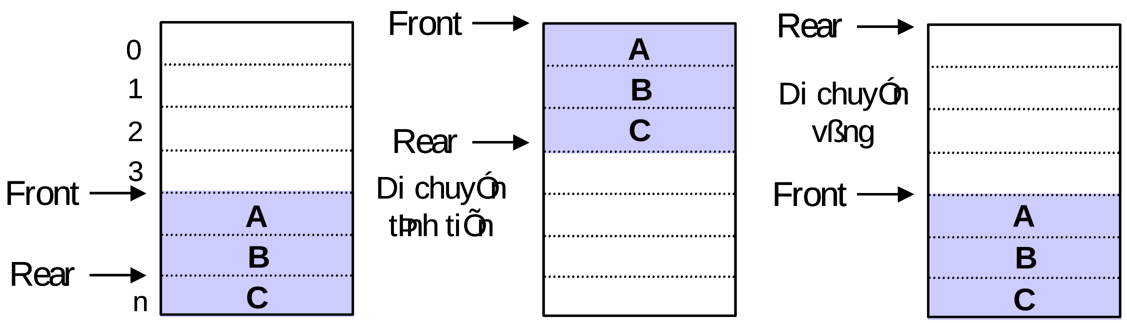
Queue có hai phép toán chính:

- Insert_queue : thêm một phần tử vào hàng đợi; khi thêm ta phải lưu ý xem hàng đợi bị tràn hay bị đầy để xử lý cho thích hợp.

+ Hàng đợi bị tràn : là trường hợp khi Rear = QUEUESIZE-1 (=n) và Front > 0, tức là không thể thêm phần tử mới vào cuối hàng. Để khắc phục trường hợp này, ta có thể dùng một trong 2 cách:

- Di chuyển tịnh tiến từng phần tử lên để có Front = 0 và Rear < QUEUESIZE-1 (trường hợp này Front luôn nhỏ hơn Rear)
- Di chuyển vòng : cho Rear = 0, Front giữ nguyên

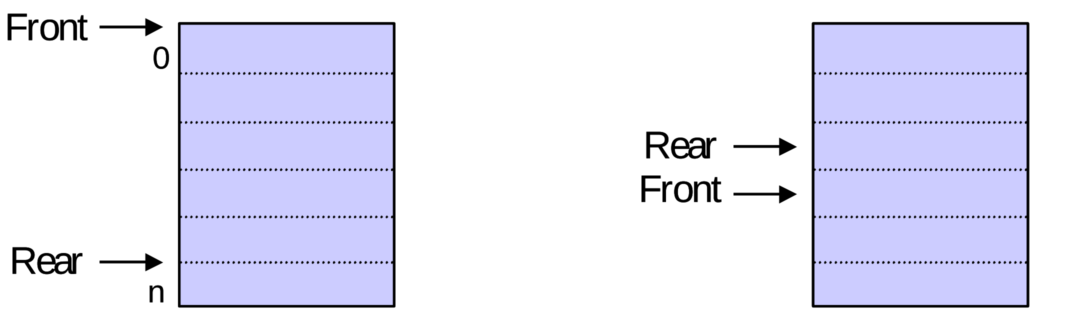
(trường hợp này Front có lúc nhỏ hơn, có lúc lớn hơn Rear)



Hàng đợi bị tràn　　　Rear ≥ Front　　　Rear >< Front

+ Hàng đợi bị đầy: hàng đợi không có phần tử nào trống: Front = 0 và Rear = n hoặc Rear đứng ngay trước Front; do đó nếu tiếp tục thêm vào sẽ bị mất dữ liệu.

Rear = QUEUESIZE-1
Front = 0
} Rear - Front + 1 = QUEUESIZE　**hoặc**　Rear đứng ngay trước Front:
Rear - Front + 1 = 0



Rear - Front + 1 = QUEUESIZE　　　Rear - Front + 1 = 0

- Delete_queue: loại bỏ phần tử khỏi Queue và trả về giá trị của phần tử vừa xóa; trước khi xóa, ta phải kiểm tra Queue có khác rỗng hay không.

- Khởi tạo hàng đợi:

Front := -1;
Rear := -1;

<u>Lưu ý:</u>

- Phép toán Insert_queue thực hiện ở cuối hàng đợi, còn phép toán Delete_queue thực hiện ở đầu hàng đợi.

Ta khai báo biến q có kiểu cấu trúc Queue gồm 3 thành phần:

- front, rear : số nguyên chỉ đầu và cuối hàng đợi
- nodes: mảng 1 chiều, mỗi phần tử của mảng là 1 phần tử trong queue.

```
#define QUEUESIZE 100
#define TRUE 1
#define FALSE 0
struct queue
{
  int front, rear;
  int nodes[QUEUESIZE];
} q;
```

IV.2. Các phép toán trên hàng: Đối với hàng đợi, ta có hai phép toán chủ yếu là thêm vào (Insert_queue) và loại bỏ (Delete_queue).

a. Phép thêm vào : thêm một phần tử x vào hàng đợi; khi thêm ta phải lưu ý xem hàng đợi bị tràn hay bị đầy để xử lý cho thích hợp.

```
int Insert_queue(struct queue &q, int x)
{
  if (q.rear - q.front + 1== 0 || q.rear -q.front+1== QUEUESIZE)
    return 0 ;
   if(q.front==-1)
    { q.front=0;
      q.rear =-1;
    }
    if (q.rear==QUEUESIZE-1) q.rear=-1;
    ++q.rear;
    q.nodes[q.rear]=x;
      return 1;
}
```

b. Phép loại bỏ : loại bỏ phần tử khỏi Queue và trả về giá trị của phần tử vừa xóa; trước khi xóa, ta phải kiểm tra Queue có khác rỗng hay không.

```
int Delete_queue(struct queue &q, int &x)
{
  if(q.front==-1)      return 0;
    x= q.nodes[q.front];
    if(q.front == q.rear)  // Hang chi co 1 phan tu
    {
```

```
        q.front = -1;
        q.rear  = -1;
    }
    else
    {
        (q.front)++;
        if (q.front ==QUEUESIZE)
          q.front=0;
    }
    return 1;
}
```

* <u>ứng dụng của Queue:</u>

- Queue thường được dùng trong các bài toán có cơ chế FIFO (vào trước ra trước).

- Queue thường được dùng trong các công việc đang đợi phục vụ trong các hệ điều hành đa nhiệm, để quản lý các hàng đợi in trên máy in mạng (print server)...

IV.3. <u>Ví dụ</u>:

Viết chương trình đổi phần thập phân của số không âm ở hệ thập phân sang số ở hệ nhị phân, tối đa ta chỉ lấy 8 số lẽ trong hệ nhị phân

* Giải thuật:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <math.h>
#define QUEUESIZE 8
#define TRUE 1
#define FALSE 0
struct queue
{
  int front, rear;
  int nodes[QUEUESIZE];
};
struct queue q;
void Initialize(struct queue &q)
{
  q.front = q.rear = -1;
}
int empty(struct queue q)
```

```
{
  return((q.front == -1 || q.rear== -1) ? TRUE : FALSE);
}
void Insert_queue(struct queue &q, int x)
{
  if (q.rear - q.front + 1== 0 || q.rear -q.front+1== QUEUESIZE)
  {
    printf("\nHang bi day");
  }
  else
  {
    if(q.front==-1)
    { q.front=0;
      q.rear =-1;
    }
    if (q.rear==QUEUESIZE-1) q.rear=-1;
    ++q.rear;
    q.nodes[q.rear]=x;
  }
}
int Delete_queue(struct queue &q)
{
  int x;
  if(empty(q))
    printf("Hang doi rong");
  else
  {
    x= q.nodes[q.front];
    if(q.front == q.rear)  // Hang chi co 1 phan tu
    {
       q.front = -1;
       q.rear  = -1;
    }
    else
    {
       (q.front)++;
       if (q.front ==QUEUESIZE)
         q.front=0;
    }
    return x;
```

```
    }
  }
  void dec_bin2(double n)
  {  double positive;
    double r, le;
    le = modf(n,&positive);   // Hàm modf tách số double n ra thành 2 phần
  //   phần nguyên chứa trong positive (double) và phần thập phân chứa
  // trong le (double)
    int i=0;
  do
    { r =le*2;
      le = modf(r,&positive);
      Insert_queue(q, positive);
      i++ ;
      } while (i <8 && r!=1);
      printf("\n So nhi phan cua phan le : 0.");
      while (!empty(q))
      {
        printf("%d", Delete_queue(q));
      }
      getch();
  }
  // chuong trinh chinh
  void main(void)
  {
  int chucnang, so;
     float n;
     clrscr();
  // khoi tao queue
     Initialize(q);
     printf("\nNhap phan thap phan cua so thuc :");
     scanf("%e", &n);
     dec_bin2(n);
  }
```

BÀi tập

1. Viết chương trình cho phép :
   - Nhập một văn bản có tối đa 100 câu, mỗi câu có tối đa 80 ký tự, và mỗi từ trong câu có ít nhất một khoảng trắng. Ta kết thúc việc nhập bằng câu rỗng.
   - Xử lý câu :

+ In ra màn hình các câu bất kỳ trong văn bản

+ Loại bỏ một đoạn gồm một số câu nào đó trong văn bản

+ Xen vào một đoạn mới tại một vị trí bất kỳ trong văn bản

- Xử lý từ :

+ Cho biết số lần xuất hiện của một từ trong văn bản

+ Thay thế một từ bằng một từ khác.

2. Tạo menu thực hiện các công việc sau:

a. Nhập danh sách học viên, mỗi học viên gồm : maso (số nguyên), ho (chuỗi), ten (chuỗi). Danh sách học viên được lưu trữ trong 1 danh sách tuyến tính có tối đa 100 phần tử.

b. Liệt kê danh sách trên màn hình

c. Sắp xếp lại danh sách theo thứ tự tăng dần của tên, trùng tên thì sắp qua họ.

d. Thêm một học viên vào danh sách sao cho sau khi thêm thì vẫn đảm bảo tính thứ tự của danh sách.

e. In ra màn hình thông tin của học viên có mã số do ta nhập

f. Xóa học viên có mã số do ta nhập

g. Save danh sách học viên vào file DSHV.TXT.

h. Load danh sách học viên từ file DSHV.TXT vào danh sách tuyến tính.

3. Tạo menu thực hiện các công việc sau:

a. Đổi số thực dương ở hệ thập phân sang hệ nhị phân

b. Đổi số thực dương ở hệ thập phân sang hệ thập lục

c. Đổi số nhị phân sang số thập phân

d. Đổi số thập lục sang số thập phân

4. Hãy đếm có bao nhiêu bit 1 trong 1000 từ, mỗi từ 4 byte

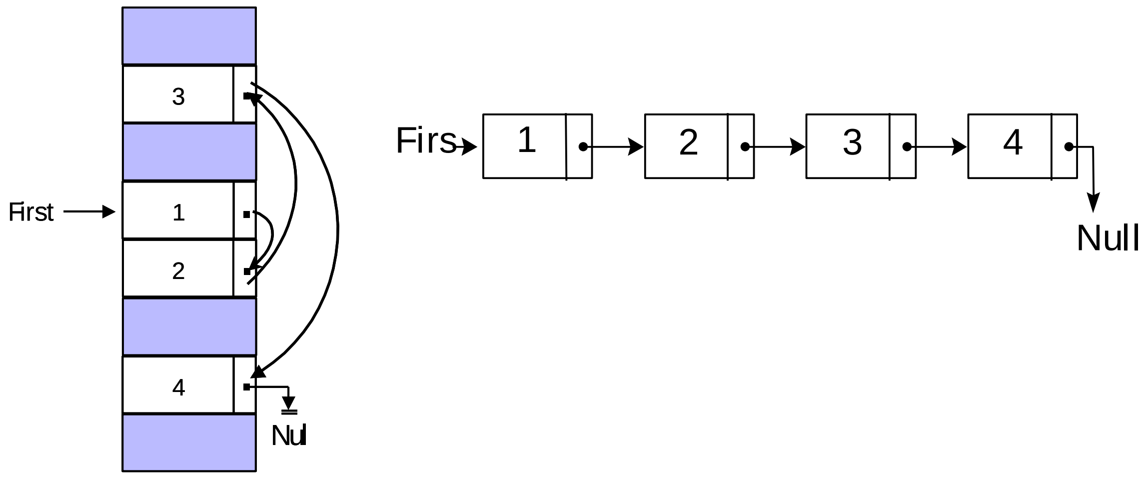
CHƯƠNG IV

DANH SÁCH LIÊN KẾT
(LINKED LIST)

I. KHÁI NIỆM:

Cấu trúc danh sách liên kết là cấu trúc động, việc cấp phát nút và giải phóng nút trên danh sách xảy ra khi chương trình đang chạy. Ta thường cấp phát nút cho danh sách liên kết bằng biến động. Danh sách liên kết có nhiều loại như danh sách liên kết đơn, danh sách liên kết kép, danh sách liên kết đa và danh sách liên kết vòng. Trong chương này ra sẽ khảo sát một số loại danh sách liên kết như đơn, kép và vòng.

Các phần tử trong danh sách liên kết sẽ được cấp phát vùng nhớ trong quá trình thực thi chương trình, do đó chúng có thể nằm rải rác ở nhiều nơi khác nhau trong bộ nhớ (không liên tục).



Hình 4.1 Minh họa danh sách liên kết trong bộ nhớ

Các phần tử trong danh sách được kết nối với nhau theo chùm liên kết như hình trên:

- First là con trỏ chỉ đến phần tử đầu của danh sách liên kết

- Phần tử cuối của danh sách liên kết với vùng liên kết có giá trị NULL

- Mỗi nút của danh sách có trường **info** chứa nội dung của nút và trường **next** là con trỏ chỉ đến nút kế tiếp trong danh sách.

* ***Lưu ý***:

- Cấu trúc danh sách liên kết là cấu trúc động, các nút được cấp phát hoặc bị giải phóng khi chương trình đang chạy.

- Danh sách liên kết rất thích hợp khi thực hiện các phép toán trên danh sách thường bị biến động. Trong trường hợp xóa hay thêm phần tử trong danh sách liên kết thì ta không dời các phần tử đi như trong danh sách tuyến tính (mảng) mà chỉ việc hiệu chỉnh lại trường next tại các nút đang thao tác. Thời gian thực hiện các phép toán thêm vào và loại bỏ không phụ thuộc vào số phần tử của danh sách liên kết.

- Tuy nhiên, danh sách liên kết cũng có các điểm hạn chế sau:

+ Vì mỗi nút của danh sách liên kết phải chứa thêm trường next nên danh sách liên kết phải tốn thêm bộ nhớ.

+ Tìm kiếm trên danh sách liên kết không nhanh vì ta chỉ được truy xuất tuần tự từ đầu danh sách.

▪ Khai báo : Một phần tử của danh sách liên kết ít nhất phải có hai thành phần : nội dung của phần tử (info) và thành phần next để liên kết phần tử này với phần tử khác.

Giả sử ta khai báo kiểu PTR là kiểu con trỏ chỉ đến nút trong 1 danh sách liên kết, mỗi phần tử có 2 thành phần : info (số nguyên) và next.

```
typedef struct node
{  int info ;
   struct node *next ;
};
typedef struct node * PTR;
```

- Để khai báo biến First quản lý danh sách liên kết ta viết như sau:

```
PTR First;
```

- Khởi tạo danh sách liên kết : First = NULL;

- Ghi chú :

· Thành phần chứa nội dung có thể gồm nhiều vùng với các kiểu dữ liệu khác nhau.

Ví dụ: Khai báo biến First để quản lý một danh sách sinh viên với cấu trúc dữ liệu là danh sách liên kết đơn, mỗi sinh viên gồm có 2 thành phần là: mssv (số nguyên) và họ tên.

```
typedef struct sinhvien
  int mssv;
  char hoten[30];
};
typedef struct node
{  sinhvien sv ;
   struct node *next ;
};
typedef struct node *NODEPTR;
NODEPTR First;
```

· Thành phần liên kết cũng có thể nhiều hơn một nếu là danh sách đa liên kết hoặc danh sách liên kết kép.

· First là con trỏ trỏ đến phần tử đầu tiên của danh sách liên kết, nó có thể là kiểu con trỏ (như khai báo trên), và cũng có thể là một struct có hai thành phần: First trỏ đến phần tử đầu tiên của danh sách liên kết, và Last trỏ đến phần tử cuối của danh sách liên kết.

```
struct Linked_List;
{  NODEPTR First;
   NODEPTR Last;
};
```

II. <u>CÁC PHÉP TOÁN TRÊN DANH SÁCH LIÊN KẾT</u>:

Để đơn giản, các phép toán sau đây sẽ được thao tác trên danh sách các số nguyên với khai báo như sau:

```
typedef struct node
{  int info ;
   struct node *next ;
};
typedef struct node *NODEPTR;
```

- Để khai báo biến First quản lý danh sách liên kết ta viết như sau:

```
NODEPTR First;
```

II.1. ***<u>Tạo danh sách</u>***:

a. <u>Khởi tạo danh sách</u> (Initialize): dùng để khởi động một danh sách liên kết, cho chương trình hiểu là hiện tại danh sách liên kết chưa có phần tử.

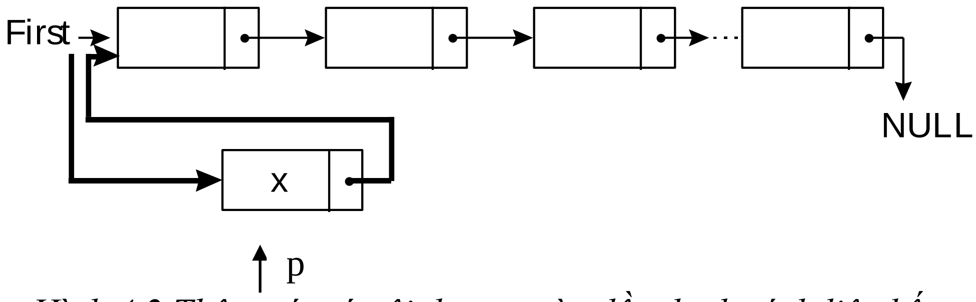
```
void Initialize(NODEPTR &First)
{
  First = NULL;
}
```

b. <u>Cấp phát vùng nhớ</u> (New_Node): cấp phát một nút cho danh sách liên kết. Hàm New_Node này trả về địa chỉ của nút vừa cấp phát.

Trong chương trình có sử dụng hàm malloc (trong <alloc.h>), hàm này cấp phát một khối nhớ tính theo byte từ bộ nhớ heap. Nếu cấp phát thành công, hàm malloc trả về địa chỉ của vùng nhớ vừa cấp phát, ngược lại nó sẽ trả về NULL.

```
NODEPTR Newnode(void)
{
     NODEPTR p= new node;
     return p;
}
PTR p = Newnode();  // PTR p= new node;
```

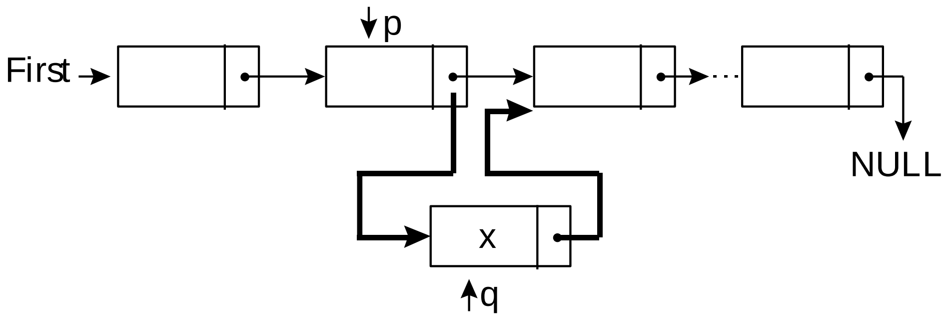
c. <u>Thêm vào đầu danh sách</u> (Insert_First): thêm một nút có nội dung x vào đầu danh sách liên kết.



Hình 4.2 Thêm nút có nội dung x vào đầu danh sách liên kết

```
void Insert_First(NODEPTR &First, int x)
{
  NODEPTR p;
  p = new node;
  p->info = x;
  p->next = First;
  First = p;
}
```

d. <u>Thêm nút mới vào sau nút có địa chỉ p</u> (Insert_After): thêm một nút có nội dung x vào sau nút có địa chỉ p trong danh sách liên kết First.



Hình 4.3 Thêm nút có nội dung x vào sau nút có địa chỉ p

```
void Insert_after(NODEPTR p, int x)
{
  NODEPTR q;
  if(p == NULL)
        printf("khong them phan tu vao danh sach duoc");
  else
  {
    q =new node;
    q->info = x;
    q->next = p->next;
    p->next = q;
  }
```

}

II.2. Tìm kiếm (Search_info):

Tìm nút đầu tiên trong danh sách có info bằng với x. Do đây là danh sách liên kết nên ta phải tìm từ đầu danh sách.

Hàm Search_info nếu tìm thấy x trong danh sách thì trả về địa chỉ của nút có trị bằng x trong danh sách, nếu không có thì trả về trị NULL.

```
PTR Search_info(PTR First, int x)
{
  for (PTR p = First; p != NULL ; p=p->next )
     if ( p->info == x  ) return p;
  return NULL;
}
```

II.3. Cập nhật danh sách:

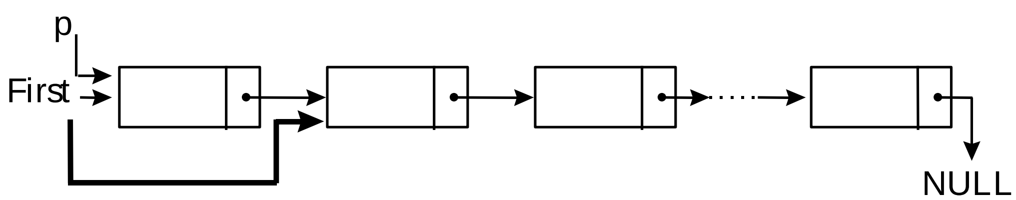
a. Giải phóng vùng nhớ (free): Hàm này dùng để hủy nút đã cấp phát, và trả vùng nhớ về lại cho memory heap.

```
delete p;
```

b. Kiểm tra danh sách liên kết rỗng hay không (Empty): hàm Empty trả về TRUE nếu danh sách liên kết rỗng, và ngược lại.

```
int Empty(NODEPTR First)
{
  return(First == NULL);
}
```

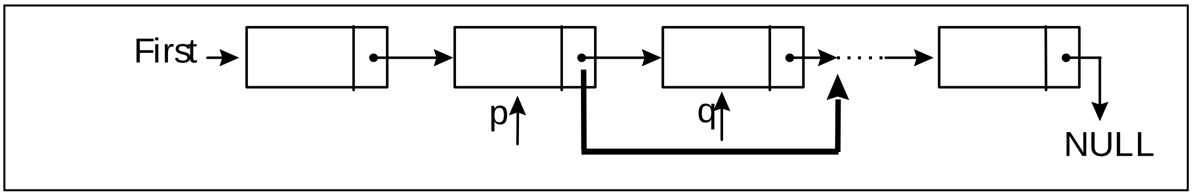
c. Xóa phần tử đầu của danh sách (Delete_First): muốn xóa 1 phần tử khỏi danh sách liên kết thì ta phải kiểm tra xem danh sách có rỗng hay không. Nếu danh sách có phần tử thì mới xóa được.



Hình 4.4 Xóa nút đầu tiên trong danh sách liên kết

```
int Delete_First (NODEPTR &First)
{ NODEPTR p;
  if (Empty(First))
    return 0;
   p = First;    // nut can xoa la nut dau
   First = p->next;
   delete p;
```

```
        return 1;
      }
```



d. <u>Xóa phần tử đứng sau nút có địa chỉ p (Delete_after)</u>:

Hình 4.5 Xóa nút sau nút có địa chỉ p

```
int  Delete_after(NODEPTR p)
{   NODEPTR q;
  // nếu p là NULL hoặc sau p không có nút
   if((p == NULL) || (p->next == NULL))
     return 0;
     q = p->next;  // q chi nut can xoa
    p->next = q->next;
    delete q;
    return 1;
}
```

e. <u>Xóa phần tử theo nội dung (Delete_info):</u>

- Tìm địa chỉ của phần tử có nội dung là x trong danh sách liên kết.
- Loại bỏ phần tử này theo địa chỉ.

```
int Delete_info(NODEPTR &First,int x)
{ NODEPTR q,p;
  p= search_info(First,x);
  if (p==NULL) return 0;
  if (p==First)
       Delete_first(First);
    else
    {
       q=First;
       while (q->next != p)
         q=q->next;
       Delete_after(q);
    }
    return 1;
}
```

Lưu ý : Giải thuật này chỉ loại bỏ phần tử đầu tiên trong danh sách có giá trị info = x.

f. Xóa toàn bộ danh sách (Clearlist): ta có thể sử dụng lệnh First = NULL để xóa toàn bộ danh sách, nhưng trong bộ nhớ, các vùng nhớ đã cấp phát cho các nút không giải phóng về lại cho memory heap, nên sẽ lãng phí vùng nhớ. Do đó, ta sử dụng giải thuật sau:

```
void Clearlist(NODEPTR &First)
{
  NODEPTR p;
 while(First != NULL)
  {
    p = First;
    First = First->next;
    delete p;
  }
}
```

II.4. Duyệt danh sách:

Thông thường ta hay duyệt danh sách liên kết để thực hiện một công việc gì đó, như liệt kê dữ liệu trong danh sách hay đếm số nút trong danh sách...

```
void Traverse(NODEPTR First)
{ NODEPTR p;
  int stt = 0;
  p = First;
  if(p == NULL)
    printf("\n   (Khong co phan tu trong danh sach)");
  while(p != NULL)
  {
    printf("\n   %5d%8d", stt++, p->info);
    p = p->next;
  }
}
```

II.5. Sắp xếp (Selection_Sort): sắp xếp danh sách liên kết theo thứ tự info tăng dần.

- Nội dung: Ta so sánh tất cả các phần tử của danh sách để chọn ra một phần tử nhỏ nhất đưa về đầu danh sách; sau đó, tiếp tục chọn phần tử nhỏ nhất trong các phần tử còn lại để đưa về phần tử thứ hai trong danh sách. Quá trình này lặp lại cho đến khi chọn ra được phần tử nhỏ thứ (n-1).

- Giải thuật:

```
void Selection_Sort(NODEPTR &First)
```

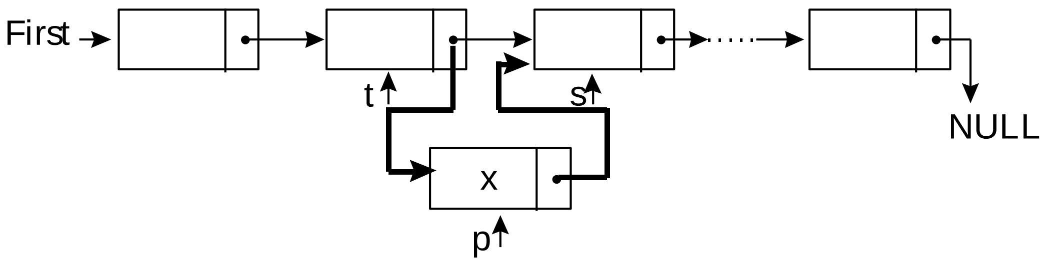
```
{ NODEPTR p, q, pmin;
      int min;
      for(p = First; p->next != NULL; p = p->next)
      {     min = p->info;
            pmin = p;
            for(q = p->next; q != NULL; q = q->next)
                  if(min > q->info)
                  {
                        min = q->info;
                        pmin = q;
                  }
            // hoan doi truong info cua hai nut p va pmin
            pmin->info = p->info;
            p->info = min;
      }
}
```

III. <u>CÁC PHÉP TOÁN TRÊN DANH SÁCH LIÊN KẾT CÓ THỨ TỰ</u>:

Danh sách liên kết có thứ tự là một danh sách liên kết đã được sắp xếp theo một thứ tự nhất định (tăng hay giảm) trên một thành phần nào đó của nội dung. ở đây, ta giả sử danh sách liên kết First có thứ tự tăng theo thành phần info.

III.1. <u>Phép thêm vào</u> :

Thêm vào danh sách liên kết có thứ tự một phần tử có nội dung là x sao cho sau khi thêm vào vẫn đảm bảo tính có thứ tự của danh sách.



Hình 4.6 Thêm nút có nội dung x vào danh sách liên kết có thứ tự

* <u>Giải thuật:</u>

- Tạo phần tử mới, gán giá trị x cho nó

```
p=New_Node  () ;
p->info = x ;
```

- Tìm vị trí thích hợp để đưa phần tử mới vào, nghĩa là tìm hai vị trí t và s sao cho: t->info <= x <= s->info

```
for(s = First; s != NULL && s->info < x; t=s, s = s->next);
```

- Gán liên kết thích hợp sao cho p nằm giữa hai phần tử có địa chỉ t và s:

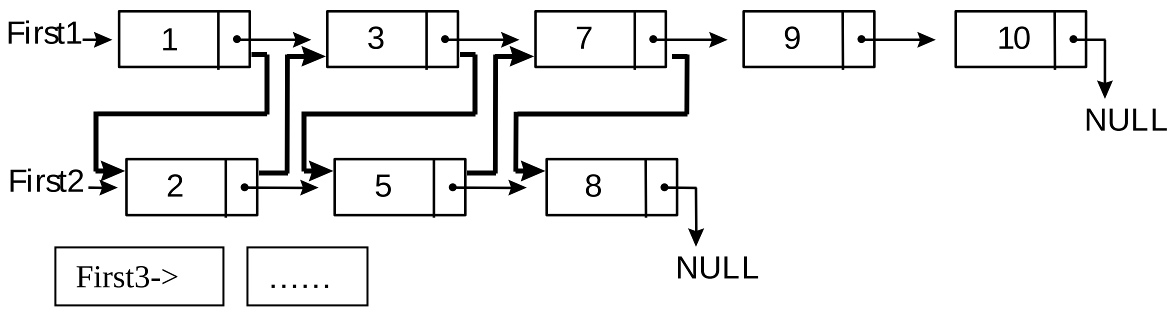
```
 if(s == First)  // them nut vao dau danh sach lien ket
{
   p->next = First;
   First = p;
}
else          // them nut p vao truoc nut s
{
   p->next= s;
   t->next= p;
}
```

*** <u>Chương trình</u>:**

```
void Insert_Order(NODEPTR &First, int x)
{
   NODEPTR p, t, s;  // q la nut truoc, p la nut sau
   p=new node;
   p->info=x;
   for(s = First; s != NULL && s->info < x ; t=s, s = s->next);
  if(s == First)  // them nut vao dau danh sach lien ket
    {
      p->next = First;
      First = p;
    }
    else          // them nut p vao truoc nut s
    {
      p->next= s;
      t->next= p;
    }
 }
```

III.2. <u>Phép trộn</u> :

Cho hai danh sách liên kết First1, First2 đã có thứ tự. Hãy trộn hai danh sách này lại thành một danh sách liên kết mới First3 sao cho nó cũng có thứ tự.



Hình 4.7 Trộn hai danh sách liên kết có thứ tự

a. <u>Giải thuật</u>: Gọi p1, p2, p3 là 3 biến con trỏ để duyệt 3 danh sách First1, First2, First3

- Tạo giả nút đầu tiên trong danh sách liên kết First3 để hình thành một phần tử cho p3 chỉ đến.

- Duyệt First1 và First2:

Nếu p1->info < p2->info :

Đưa phần tử p1 vào sau phần tử p3

Cho p1 chỉ đến phần tử kế trong danh sách First1

Nếu p2->info < p1->info :

Đưa phần tử p2 vào sau phần tử p3

Cho p2 chỉ đến phần tử kế trong danh sách First2

Quá trình duyệt sẽ dừng lại khi 1 trong 2 danh sách đã duyệt xong

- Đưa nốt phần còn lại của danh sách chưa duyệt xong vào danh sách liên kết First3.

- Xóa phần tử giả đầu danh sách First3 đã tạo ở trên.

```
NODEPTR Merge(NODEPTR &First1, NODEPTR &First2)
{ NODEPTR p1, p2, p3;
  NODEPTR First3= new node ;
  p1=First1; p2 = First2; p3=First3;
  while (p1 !=NULL && p2 !=NULL)
    if (p1->info < p2->info)
    { p3->next = p1;
      p3=p1;
      p1=p1->next ;
    }
    else
    { p3->next = p2;
      p3=p2;
      p2=p2->next ;
    }
    if (p1==NULL)
        p3->next=p2;
    else
        p3->next=p1;
    p3 = First3;
    First3=p3->next;
    delete p3;
```

```
  First1=First2=NULL;
  return First3;
}
```

Ví dụ:

Viết chương trình tạo một menu để quản lý danh sách sinh viên gồm các công việc sau:

1. Tạo danh sách sinh viên: Quá trình nhập danh sách sẽ dừng lại khi ta nhập mã số là 0.
2. Thêm sinh viên vào danh sách: Thêm 1 sinh viên vào danh sách, vị trí sinh viên thêm vào do ta chọn
3. Xem danh sách sinh viên: Liệt kê danh sách sinh viên trên màn hình
4. Hiệu chỉnh sinh viên: nhập vào vị trí sinh viên cần hiệu chỉnh, sau đó chương trình cho phép ta hiệu chỉnh lại mã số, họ, tên của sinh viên.
5. Xóa sinh viên trong danh sách: xóa sinh viên theo vị trí.
6. Tìm kiếm sinh viên theo mã số: nhập vào mã số sinh viên, sau đó in ra vị trí của sinh viên trong danh sách.
7. Sắp xếp danh sách sinh viên theo mã số tăng dần
8. Thêm sinh viên vào danh sách đã có thứ tự tăng dần theo mã số sao cho sau khi thêm thì danh sách vẫn còn tăng dần theo mã số.
9. Xóa toàn bộ danh sách sinh viên.

   Biết rằng:

   - Mỗi sinh viên gồm các thông tin: mã số (int), họ, tên

   - Danh sách sinh viên được tổ chức theo danh sách liên kết đơn.

Chương trình:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <alloc.h>
#include <ctype.h>
#define TRUE 1
#define FALSE 0
struct sinhvien
{
  int mssv;
  char ho[30];
  char ten[10];
};
struct node
```

```
{
   sinhvien sv;
   struct node *next;
};
typedef node *NODEPTR;
  NODEPTR First;
   sinhvien sv;
   NODEPTR p;
// Phep toan New_node: cap phat mot nut cho danh sach lien ket
NODEPTR New_node(void)
{
   NODEPTR p;
   p = (NODEPTR)malloc(sizeof(struct node));
   return(p);
}
/* Tac vu nodepointer: xac dinh con tro cua nut i trong danh sach lien ket
   (i = 2, ...) */
NODEPTR nodepointer(NODEPTR First, int i)
{
      NODEPTR p;
      int vitri=1;
      p = First;
      while(p != NULL && vitri < i)
      {
            p = p->next;
            vitri++;
      }
      return(p);
}
// Tac vu position: xac dinh vi tri cua nut p trong danh sach lien ket
int position(NODEPTR First, NODEPTR p)
{
      int vitri;
      NODEPTR q;
      q = First;
      vitri = 1;
      while(q != NULL && q != p)
      {
            q = q->next;
            vitri++;
```

```
    }
  if(q == NULL)
    return(-1);
  return(vitri);
}
// Phep toan initialize: khoi dong danh sach lien ket
void initialize(NODEPTR &First)
{
  First = NULL;
}
// Tac vu Empty: kiem tra danh sach lien ket co bi rong khong
int Empty(NODEPTR First)
{
  return(First == NULL ? TRUE : FALSE);
}
// Phep toan Insert_first: them nut moi vao dau danh sach lien ket
void Insert_first(NODEPTR &First, sinhvien x)
{
  NODEPTR p;
  p = New_node();
  p->sv = x;
  p->next = First;
  First = p;
}
// Phep toan Insert_after: them nut moi sau nut co dia chi p
void Insert_after(NODEPTR p, sinhvien x)
{
  NODEPTR q;
  if(p == NULL)
           printf("khong them sinh vien vao danh sach duoc");
  else
  {
    q = New_node();
    q->sv = x;
    q->next = p->next;
    p->next = q;
  }
}
// Phep toan Delete_first: xoa nut o dau danh sach lien ket
void Delete_first(NODEPTR &First)
```

```
{
  NODEPTR p;
  if(Empty(First))
    printf("Khong co sinh vien trong danh sach");
  else
  {
    p = First;    // nut can xoa la nut dau
    First = p->next;
    free(p);
  }
}
// Tac vu Delete_after: xoa nut sau nut p
void Delete_after(NODEPTR p)
{
  NODEPTR q;
  // neu p la NULL hoac p chi nut cuoi
  if((p == NULL) || (p->next == NULL))
    printf("khong xoa sinh vien nay duoc");
  else
  {
    q = p->next;  // q chi nut can xoa
    p->next = q->next;
    free(q);
  }
}
/* Phep toan Insert_Order: Phep toan nay chi su dung khi them nut vao
danh sach lien ket da co thu tu */
void Insert_Order(NODEPTR &First, sinhvien x)
{
  NODEPTR p, q;  // q la nut truoc, p la nut sau
  q = NULL;
  for(p = First; p != NULL && p->sv.mssv< x.mssv; p = p->next)
    q = p;
  if(q == NULL)  // them nut vao dau danh sach lien ket
    Insert_first(First, x);
  else           // them nut vao sau nut q
    Insert_after(q, x);
}
// Phep toan clearlist: xoa tat ca cac nut trong danh sach lien ket
void clearlist(NODEPTR &First)
```

```
{
  NODEPTR p, q;  // q la nut truoc, p la nut sau
  p = First;
  while(First != NULL)
  {
    p = First;
    First = First->next;
    free(p);
  }
}
// Phep toan traverse: duyet danh sach lien ket
void traverse(NODEPTR First)
{
  NODEPTR p;
  int stt = 0;
  p = First;
  if(p == NULL)
    printf("\n   (Khong co sinh vien trong danh sach)");
  while(p != NULL)
  {
    printf("\n %5d %8d %-30s %-10s", ++stt, p->sv.mssv, p->sv.ho,p->sv.ten);
    p = p->next;
  }
}

/* Tac vu search_info: tim kiem theo phuong phap tim kiem tuyen tinh,
neu khong tim thay ham nay tra ve NULL, neu tim thay ham nay tra ve
con tro chi  nut tim thay */
NODEPTR search_info(NODEPTR First, int x)
{
     NODEPTR p;
     p = First;
     while(p != NULL && p->sv.mssv != x  )
            p = p->next;
     return(p);
}
// Tac vu selectionsort: sap xep danh sach lien ket theo MSSV tang dan
void selectionsort(NODEPTR &First)
{
  NODEPTR p, q, pmin;
```

```
	sinhvien min;
	for(p = First; p->next != NULL; p = p->next)
	{
		min = p->sv;
		pmin = p;
		for(q = p->next; q != NULL; q = q->next)
			if(min.mssv > q->sv.mssv)
			{
				min = q->sv;
				pmin = q;
			}
		// hoan doi truong info cua hai nut p va pmin
		pmin->sv  = p->sv;
		p->sv   = min;
	}
}
```
char menu ()

```
{  char chucnang;
   do
   {  clrscr();
      printf("\n\n\t\tCHUONG TRINH QUAN LY DANH SACH SINH
VIEN");
      printf("\n\nCac chuc nang cua chuong trinh:\n");
      printf("   1: Tao danh sach sinh vien\n");
      printf("   2: Them sinh vien vao danh sach\n");
      printf("   3: Xem danh sach sinh vien\n");
      printf("   4: Hieu chinh sinh vien\n");
      printf("   5: Xoa sinh vien trong danh sach\n");
      printf("   6: Tim kiem sinh vien theo MSSV\n");
      printf("   7: Sap xep danh sach theo MSSV\n");
      printf("   8: Them sinh vien vao danh sach da co thu tu\n");
      printf("   9: Xoa toan bo danh sach\n");
      printf("   0: Ket thuc chuong trinh\n");
      printf("Chuc nang ban chon: ");
      chucnang = getche();
      } while(chucnang < '0' || chucnang >'9') ;
      return chucnang;
}
```
void Create_list(NODEPTR &First)

```
{ NODEPTR Last,p ;
```

```
  sinhvien sv;

  char maso [5],c;
     clearlist(First);
     printf("Ma so sinh vien: ");
     gets(maso);
     sv.mssv = atoi(maso);
  while (sv.mssv !=0)
  {
           printf("Ho sinh vien: ");
           gets(sv.ho);
           printf("Ten sinh vien: ");
           gets(sv.ten);
           p=New_node();
           p->sv=sv;
           if (First==NULL)
             First=p;
           else
             Last->next = p;
           Last=p;
           p->next=NULL;
           printf("Ma so sinh vien moi: ");
           gets(maso);
           sv.mssv = atoi(maso);
  }
}
// chuong trinh chinh
void main()
{
   int vitri;
   char chucnang, c, maso [5], c_vitri[5];
   // khoi dong danh sach lien ket
   initialize(First);
   do
   {
     chucnang = menu();
     flushall();
     switch(chucnang)
     {
     case '1':
```

```
{
   Create_list(First);
   break;
}
case '2':
{
        printf("\nVi tri them (1, 2, ...): ");
        gets(c_vitri);
        vitri = atoi(c_vitri);
        p = nodepointer(First, vitri-1);//p chi nut truoc nut can them
        if (vitri <=0 || p==NULL)
        {
                printf("Vi tri khong hop le");
                getche();
        }
        else
        {
                printf("Ma so sinh vien: ");
                gets(maso);
                sv.mssv = atoi(maso);
                printf("Ho sinh vien: ");
                gets(sv.ho);
                printf("Ten sinh vien: ");
                gets(sv.ten);
                if (vitri == 1)
                    Insert_first(First, sv);
                else
                    Insert_after(p, sv);
        }
        break;
}
case '3':
{
        printf("\nDanh sach sinh vien: ");
        printf("\n     STT   MSSV          HO TEN");
        traverse(First);
        getche();
        break;
}
```

```
case '4':
{
        printf("\nVi tri hieu chinh (1, 2, ...): ");
        gets(c_vitri);
        vitri = atoi(c_vitri);
        p = nodepointer(First, vitri); // p chi nut can hieu chinh
        if(p == NULL)
        {
           printf("Vi tri khong phu hop");
          getche();
        }
        else
        {
          printf("\nSTT:%d  MSSV:%d  HO:%s          TEN:%s",
                vitri,p->sv.mssv, p->sv.ho, p->sv.ten);
          printf("\nMa so sv moi: ");
          gets(maso);
          sv.mssv = atoi(maso);
          printf("Ho sv moi: ");
          gets(sv.ho);
          printf("Ten sv moi: ");
          gets(sv.ten);
          p->sv = sv;
        }
break;
}
case '5':
{
        printf("\nVi tri xoa ( 1, 2, ...): ");
        gets(c_vitri);
        vitri = atoi(c_vitri);
        p = nodepointer(First, vitri-1);//p chi nut truoc nut can xoa
        if (vitri <=0 || p==NULL)
        {
                printf("Vi tri khong hop le");
                getche();
        }
        else
            if(vitri == 1)
```

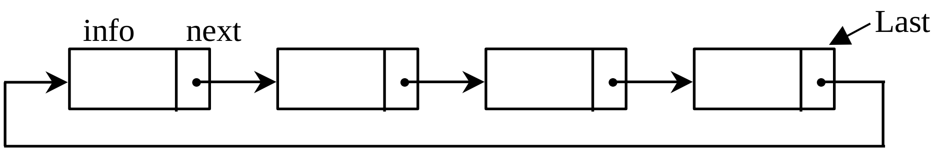
```
                    Delete_first(First);
                else
                    Delete_after(p);
            break;
    }
    case '6':
    {
            printf("\nMa so sinh vien can tim: ");
            gets(maso);
            sv.mssv = atoi(maso);
            p = search_info(First, sv.mssv);
            if(p == NULL)
           printf("Khong co sinh vien co MSSV %d trong danh sach",
                              sv.mssv);
        else
        printf("Tim thay o vi tri %d trong danh sach", position(First,
p));
        getche();
        break;
    }
    case '7':
    {
        printf("\n   Ban co chac khong? (c/k): ");
        c = toupper(getche());
        if( c == 'C')
            selectionsort(First);
        break;
    }
    case '8':
    {
       printf("\n   Ban nho sap xep danh sach truoc. Nhan phim bat ky
...");
       getche();
       printf("\nMa so sinh vien: ");
       gets(maso);
       sv.mssv = atoi(maso);
       printf("Ho sinh vien: ");
       gets(sv.ho);
       printf("Ten sinh vien: ");
       gets(sv.ten);
```

```
            Insert_Order(First, sv);
            break;
         }
         case '9':
         {
            printf("\n   Ban co chac khong (c/k): ");
            c = getche();
            if(c == 'c' || c == 'C')
                  clearlist(First);
            break;
         }
         }
      } while(chucnang != '0');

      // xoa tat ca cac nut tren danh sach lien ket
         clearlist(First);
   }
```

Iv. <u>Danh sÁch liÊn kết vòng</u>:

IV.1. <u>Khái niệm</u> :

Danh sách liên kết vòng là danh sách liên kết mà trường next của phần tử cuối sẽ chỉ tới phần tử đầu của danh sách.



Hình 4.8 Danh sách liên kết vòng

<u>Qui ước</u>: Để đơn giản giải thuật, ta qui ước dùng con trỏ Last để quản lý danh sách liên kết vòng, con trỏ này sẽ chỉ tới phần tử cuối trong danh sách liên kết vòng.

Như vậy:

+ Nếu danh sách liên kết vòng rỗng ⇔ Last = NULL

+ Nếu danh sách liên kết vòng chỉ có một phần tử ⇔ (Last ==Last->next)

- Khai báo: Ta khai báo biến Last quản lý danh sách liên kết vòng với thành phần nội dung là số nguyên như sau:

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

```
typedef struct node *NODEPTR;
NODEPTR Last;
```

IV.2. <u>Các phép toán trên danh sách liên kết vòng</u>:

IV.2.1.*<u>Tạo danh sách</u>*:

a. <u>Khởi tạo danh sách (Initialize)</u>: dùng để khởi động một danh sách liên kết, cho chương trình hiểu là hiện tại danh sách liên kết chưa có phần tử.

```
void Initialize(NODEPTR &Last)
{
  Last = NULL;
}
```

b. <u>Cấp phát vùng nhớ (New_Node)</u>: cấp phát một nút cho danh sách liên kết vòng. Hàm New_Node này trả về địa chỉ của nút vừa cấp phát.

```
NODEPTR New_node(void)
{
  NODEPTR p;
  p = (NODEPTR)malloc(sizeof(struct node));
  return (p);
}
```

c. <u>Thêm vào đầu danh sách (Ins_first)</u>: thêm một nút có nội dung x vào đầu danh sách liên kết vòng.

```
void Ins_first(NODEPTR &Last, int x)
{
  NODEPTR p;
  p = New_node();
  p->info = x;
  if (Empty(Last))
      Last=p;
  else
      p->next = Last->next;
  Last->next = p;
}
```

d. <u>Thêm vào cuối danh sách (Ins_last)</u>: thêm một nút có nội dung x vào cuối danh sách liên kết vòng.

```
void Ins_last(NODEPTR &Last, int x)
{
  NODEPTR p;
  p = New_node();
```

```
      p->info = x;
      if (Empty(Last))
          p->next=p;
      else
      {
          p->next = Last->next;
          Last->next = p;
      }
      Last = p;
        }
```

e. <u>Thêm nút mới vào sau nút có địa chỉ p (Ins_after)</u>: thêm một nút có nội dung x vào sau nút có địa chỉ p trong danh sách liên kết vòng.

```
    void Ins_after(NODEPTR Last, NODEPTR p, int x)
    {
      NODEPTR q;
      if(p == NULL)
          printf("Nut hien tai khong co, nen khong the them");
      else
      {
        if (p==Last)
          Ins_last(Last,x);
        else
        {  q = New_node();
           q->info = x;
           q->next = p->next;
           p->next = q;
        }
      }
    }
```

IV.2.2. *<u>Duyệt danh sách</u>*:

Thông thường ta hay duyệt danh sách liên kết để thực hiện một công việc gì đó, như liệt kê dữ liệu trong danh sách hay đếm số nút trong danh sách...

```
void Traverse(NODEPTR Last)
{
  NODEPTR p;
  p = Last->next;  // p chi toi phan tu dau trong dslk vong
  if(Last == NULL)
    printf("\n   Danh sach rong ");
  else
```

```
  { printf("\n");
    while(p != Last)
    {
        printf("%8d", p->info);
        p = p->next;
    }
    printf("%8d",  p->info);
  }
}
```

IV.2.3. *Phép loại bỏ*:

a. Giải phóng vùng nhớ (free): Hàm này dùng để hủy nút đã cấp phát, và trả vùng nhớ về lại cho memory heap.

free(p) ; với p là biến con trỏ

b. Kiểm tra danh sách liên kết rỗng hay không (Empty): hàm Empty trả về TRUE nếu danh sách liên kết vòng rỗng, và ngược lại.

```
int Empty(NODEPTR Last)
{
  return(Last == NULL ? TRUE : FALSE);
}
```

c. Xóa phần tử đầu của danh sách (Del_first): muốn xóa 1 phần tử khỏi danh sách liên kết thì ta phải kiểm tra xem danh sách có rỗng hay không. Nếu danh sách có phần tử thì mới xóa được.

```
void Del_first(NODEPTR &Last)
{
  NODEPTR p;
  if(Empty(Last))
    printf("Khong co nut trong danh sach lien ket vong, nen khong the
xoa");
  else
  {
    p = Last->next;    // nut can xoa la nut dau
    if (p==Last)           // danh sach chi co 1 nut
       Last=NULL;
    else
       Last->next = p->next;
    free(p);
  }
}
```

d. Xóa phần tử cuối của danh sách (Del_last): muốn xóa 1 phần tử khỏi danh sách liên kết thì ta phải kiểm tra xem danh sách có rỗng hay không. Nếu danh sách có phần tử thì mới xóa được.

```
void Del_last(NODEPTR &Last)
{
  NODEPTR p;
  if(Empty(Last))
    printf("Khong co nut trong danh sach lien ket vong, nen khong the
xoa");
  else
  {
    p = Last;    // nut can xoa la nut cuoi
    if (Last->next==Last)        // danh sach chi co 1 nut
      Last=NULL;
    else
    {
      for (NODEPTR q=Last->next;q->next !=Last; q=q->next);
     // q dung ngay truoc Last
      q->next = Last->next;
      Last=q;
    }
    free(p);
  }
}
```

e. <u>Xóa phần tử đứng sau nút có địa chỉ p (Del_after)</u>: xóa nút sau nút p. Phép toán này không xóa được khi danh sách đã rỗng hoặc danh sách chỉ có 1 nút

```
void Del_after(NODEPTR &Last, NODEPTR p)
{
  NODEPTR q;
  if(Empty(Last))
    printf("Khong co nut trong danh sach lien ket vong, nen khong the
xoa");
  else
  { // neu p la NULL hoac danh sach chi co 1 nut
   if((p == NULL) || (Last->next == Last))
      printf("khong the xoa trong danh sach lien ket vong duoc");
   else
   {
    q=p->next;
    if (p->next == Last)
```

```
        {
          p->next=Last->next;
          Last=p;
        }
        else
          p->next=q->next;
        free(q);
      }
    }
  }
```

f. <u>Xóa toàn bộ danh sách (Clearlist)</u>: ta có thể sử dụng lệnh Last=NULL để xóa toàn bộ danh sách, nhưng trong bộ nhớ, các vùng nhớ đã cấp phát cho các nút không giải phóng về lại cho memory heap, nên sẽ lãng phí vùng nhớ. Do đó, ta sử dụng giải thuật sau:

```
void clearlist(NODEPTR &Last)
{
  while(Last != NULL)
    Del_first(Last);
}
```

IV.2.4. *<u>Tìm kiếm (Srch_info)</u>* :

Tìm nút đầu tiên trong danh sách liên kết vòng có info bằng với x.

Hàm Srch_info nếu tìm thấy x trong danh sách thì trả về địa chỉ của nút đó trong danh sách, nếu không có thì trả về trị NULL.

```
NODEPTR Srch_info(NODEPTR Last, int x)
{
  NODEPTR p;
  if (Empty(Last))
    return (NULL);
  else
  {
    p = Last->next;  // p chi toi phan tu dau cua dslk vong
    if (p->info==x)
      return (p);
    else
    {  p=p->next;
       while(p != Last->next && p->info != x  )
           p = p->next;
       return (p->info==x ? p : NULL);
    }
```

```
 }
}
```

IV.2.5. ***<u>Sắp xếp (Selection_Sort)</u>***:

Sắp xếp danh sách liên kết vòng theo thứ tự info tăng dần theo phương pháp Selection sort.

- <u>Nội dung</u>: Ta so sánh tất cả các phần tử của danh sách để chọn ra một phần tử nhỏ nhất đưa về đầu danh sách; sau đó, tiếp tục chọn phần tử nhỏ nhất trong các phần tử còn lại để đưa về phần tử thứ hai trong danh sách. Quá trình này lặp lại cho đến khi chọn ra được phần tử nhỏ thứ (n-1).

- <u>Giải thuật:</u>

```
void selectionsort(NODEPTR &Last)
{
  NODEPTR p, q, pmin;
  int min;
  for(p = Last->next; p->next != Last->next; p = p->next)
  {
        min = p->info;
        pmin = p;
        for(q = p->next; q != Last->next; q = q->next)
              if(min > q->info)
              {
                    min = q->info;
                    pmin = q;
              }
        // hoan doi truong info cua hai nut p va pmin
        pmin->info  = p->info;
        p->info     = min;
  }
}
```

<u>Ví dụ</u>: Viết chương trình thực hiện các công việc sau trên một danh sách các số nguyên với cấu trúc dữ liệu là danh sách liên kết vòng :

1. Tạo danh sách số
2. Thêm phần tử vào đầu danh sách
3. Thêm phần tử vào cuối danh sách
4. Thêm phần tử vào sau phần tử có giá trị x
5. Xóa phần tử đầu trong danh sách
6. Xóa phần tử cuối trong danh sách
7. Liệt kê danh sách
8. Sắp xếp danh sách theo thứ tự tăng

9. Xóa toàn bộ danh sách

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <alloc.h>
#include <ctype.h>
#define TRUE 1
#define FALSE 0
struct node
{
  int info;
  struct node *next;
};
typedef struct node *NODEPTR;
 NODEPTR Last;
// Phep toan New_node: cap phat mot nut cho danh sach lien ket
NODEPTR New_node(void)
{
  NODEPTR p;
  p = (NODEPTR)malloc(sizeof(struct node));
  return(p);
}
// Phep toan initialize: khoi dong danh sach lien ket
void Initialize(NODEPTR &Last)
{
  Last = NULL;
}
// Tac vu Empty: kiem tra danh sach lien ket co bi rong khong
int Empty(NODEPTR Last)
{
  return(Last == NULL ? TRUE : FALSE);
}
// Phep toan Ins_first: them nut moi vao dau danh sach lien ket vong
void Ins_first(NODEPTR &Last, int x)
{
  NODEPTR p;
  p = New_node();
  p->info = x;
  if (Empty(Last))
      Last=p;
```

```
  else
       p->next = Last->next;
  Last->next = p;
}
// Phep toan Ins_last: them nut moi vao cuoi danh sach lien ket vong
void Ins_last(NODEPTR &Last, int x)
{
  NODEPTR p;
  p = New_node();
  p->info = x;
  if (Empty(Last))
       p->next=p;
  else
  {
       p->next = Last->next;
       Last->next = p;
  }
  Last = p;
}
// Phep toan Ins_after: them nut moi sau nut co dia chi p
void Ins_after(NODEPTR Last, NODEPTR p, int x)
{
  NODEPTR q;
  if(p == NULL)
       printf("Nut hien tai khong co, nen khong the them");
  else
  {
    if (p==Last)
       Ins_last(Last,x);
    else
    {  q = New_node();
        q->info = x;
        q->next = p->next;
        p->next = q;
    }
  }
}
// Phep toan Del_first: xoa nut o dau danh sach lien ket
void Del_first(NODEPTR &Last)
{
```

```
  NODEPTR p;
  if(Empty(Last))
    printf("Khong co nut trong danh sach lien ket vong, nen khong the xoa");
  else
  {
    p = Last->next;    // nut can xoa la nut dau
    if (p==Last)   // danh sach chi co 1 nut
      Last=NULL;
    else
      Last->next = p->next;
    free(p);
  }
}
// Phep toan Del_last: xoa nut o cuoi danh sach lien ket
void Del_last(NODEPTR &Last)
{
  NODEPTR p;
  if(Empty(Last))
    printf("Khong co nut trong danh sach lien ket vong, nen khong the xoa");
  else
  {
    p = Last;    // nut can xoa la nut cuoi
    if (Last->next==Last)        // danh sach chi co 1 nut
      Last=NULL;
    else
    {
       for (NODEPTR q=Last->next;q->next !=Last; q=q->next);
        // q dung ngay truoc Last
       q->next = Last->next;
       Last=q;
    }
    free(p);
  }
}
// Tac vu Del_after: xoa nut sau nut p.  Phep toan nay khong xoa duoc
// khi da rong hoac ds chi co 1 nut
void Del_after(NODEPTR &Last, NODEPTR p)
{
  NODEPTR q;
  if(Empty(Last))
```

```
    printf("Khong co nut trong danh sach lien ket vong, nen khong the
xoa");
  else
  { // neu p la NULL hoac danh sach chi co 1 nut
    if((p == NULL) || (Last->next == Last))
       printf("khong the xoa trong danh sach lien ket vong duoc");
    else
    {
      q=p->next;
      if (p->next == Last)
      {
        p->next=Last->next;
        Last=p;
      }
      else
        p->next=q->next;
      free(q);
    }
  }
}
// Phep toan clearlist: xoa tat ca cac nut trong danh sach lien ket vong
void clearlist(NODEPTR &Last)
{
  while(Last != NULL)
    Del_first(Last);
}
// Phep toan traverse: duyet danh sach lien ket vong
void traverse (NODEPTR Last)
{
  NODEPTR p;
  p = Last->next;  // p chi toi phan tu dau trong dslk vong
  if(Last == NULL)
    printf("\n   Danh sach rong ");
  else
  {  printf("\n");
    while(p != Last)
    {
      printf("%8d", p->info);
      p = p->next;
    }
```

```
    printf("%8d",  p->info);
  }
}
/* Phép toán Srch_info: tim kiem theo phuong phap tim kiem tuyen tinh,
neu khong  tim thay ham nay tra ve NULL, neu tim thay ham nay tra ve
con tro chi   nut tim thay */
NODEPTR Srch_info(NODEPTR Last, int x)
{
 NODEPTR p;
 if (Empty(Last))
   return (NULL);
 else
 {
   p = Last->next;  // p chi toi phan tu dau cua dslk vong
   if (p->info==x)
     return (p);
   else
   {  p=p->next;
      while(p != Last->next && p->info != x  )
            p = p->next;
      return (p->info==x ? p : NULL);
   }
 }
}
// Tac vu selectionsort: sap xep danh sach lien ket vong theo info tang dan
void selectionsort(NODEPTR &Last)
{
  NODEPTR p, q, pmin;
      int min;
      for(p = Last->next; p->next != Last->next; p = p->next)
      {
            min = p->info;
            pmin = p;
            for(q = p->next; q != Last->next; q = q->next)
                  if(min > q->info)
                  {
                        min = q->info;
                        pmin = q;
                  }
            // hoan doi truong info cua hai nut p va pmin
```

```
            pmin->info  = p->info;
            p->info     = min;
      }
}
void Create_list(NODEPTR &Last)
{
 int nd;
      clearlist(Last);
      printf("Nhap so (ket thuc bang 0: ");
      scanf("%d", &nd);
 while (nd !=0)
 {
      Ins_last(Last, nd);
      printf("Nhap so ke : ");
      scanf("%d", &nd);
 }
}
char menu ()
{  char chucnang;
   do
   {  clrscr();
     printf("\n\n\t\tCHUONG TRINH QUAN LY DANH SACH LIEN KET VONG");
     printf("\n\nCac chuc nang cua chuong trinh:\n");
     printf("   1: Tao danh sach \n");
     printf("   2: Them phan tu vao dau danh sach\n");
     printf("   3: Them phan tu vao cuoi danh sach\n");
     printf("   4: Them phan tu vao sau phan tu co gia tri x\n");
     printf("   5: Xoa phan tu dau trong danh sach \n");
     printf("   6: Xoa phan tu cuoi trong danh sach \n");
     printf("   7: Liet ke danh sach \n");
     printf("   8: Sap xep danh sach theo thu tu tang\n");
     printf("   9: Xoa toan bo danh sach\n");
     printf("   0: Ket thuc chuong trinh\n");
     printf("Chuc nang ban chon: ");
     chucnang = getche();
     } while(chucnang < '0' || chucnang >'9') ;
     printf("\n");
     return chucnang;
}
// chuong trinh chinh
```

```
void main()
{
  int  x, info;
  char chucnang, c;
  NODEPTR p;
  // khoi dong danh sach lien ket
  Initialize(Last);
  do
  {
    chucnang = menu();
    switch(chucnang)
    {
          case '1':
          {
             Create_list(Last);
             break;
          }
          case '2':
          {
             printf("\nNoi dung muon them: ");
             scanf("%d",&x);
             Ins_first(Last,x);
             break;
          }
          case '3':
          {
             printf("\nNoi dung muon them: ");
             scanf("%d",&x);
             Ins_last(Last,x);
             break;
          }
          case '4':
          {
             printf("\nNoi dung phan tu muon them: ");
             scanf("%d",&x);
             printf("\nBan muon them no vao sau phan tu co info = ");
             scanf("%d",&info);
             p=Srch_info(Last,info);
             if (p==NULL)
             {
```

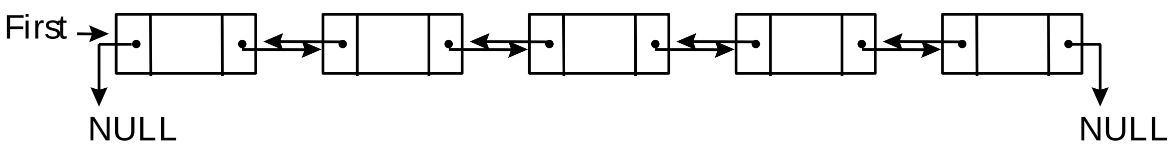
```
        printf("Khong co phan tu voi x=%d", info);
        getch();
       }
       else
        if (p==Last)
         Ins_last(Last,x);
        else
         Ins_after(Last,p,x);
       break;
     }
     case '5':
         Del_first(Last);
         break;
     case '6':
         Del_last(Last);
         break;
     case '7':
     {
         Traverse(Last);
         getche();
         break;
      }
      case '8':
      {
         printf("\n   Ban co chac khong? (c/k): ");
         c = toupper(getche());
         if( c == 'C')
           selectionsort(Last);
         break;
     }
     case '9':
     {
         printf("\n   Ban co chac khong (c/k): ");
         c = getche();
         if(c == 'c' || c == 'C')
           clearlist(Last);
         break;
     }
  }
} while(chucnang != '0');
```

```
    // xoa tat ca cac nut tren danh sach lien ket
        clearlist(Last);
}
```

V. Danh sÁch liÊn kết kép (Doubly Linked List) :

V.1. Khái niệm :

Danh sách liên kết kép là một danh sách liên kết mà mỗi phần tử của nó có 2 vùng liên kết, liên kết thuận dùng để chỉ đến phần tử đứng ngay sau nó (right), liên kết nghịch dùng để chỉ đến phần tử đứng ngay trước nó (left).



Hình 4.9 Danh sách liên kết kép

Lưu ý:

- Nút cuối của danh sách liên kết kép có trường right là NULL, và nút đầu của danh sách liên kết kép có trường left là NULL.

- Chúng ta có thể duyệt danh sách liên kết kép theo hai chiều duyệt xuôi và duyệt ngược.

* Khai báo: Ta khai báo biến First quản lý danh sách liên kết kép với thành phần nội dung là số nguyên như sau:

```
struct node
{  int info ;
   struct node *left, *right ;
};
typedef struct node *NODEPTR;
NODEPTR First. Last;
```

* Khởi tạo :

First = NULL ;

- Biến First : con trỏ chỉ đến phần tử đầu danh sách liên kết kép

V.2. Các phép toán trên danh sách liên kết kép:

V.2.1.*Tạo danh sách*:

a. Khởi tạo danh sách (Initialize): dùng để khởi động một danh sách liên kết, cho chương trình hiểu là hiện tại danh sách liên kết chưa có phần tử.

```
void Initialize(NODEPTR &First)
{
  First = NULL;
```

```
}
```

b. Cấp phát vùng nhớ (New_Node): cấp phát một nút cho danh sách liên kết kép. Hàm New_Node này trả về địa chỉ của nút vừa cấp phát.

```
NODEPTR New_node(void)
{
  NODEPTR p;
  p = (NODEPTR)malloc(sizeof(struct node));
  return (p);
}
```

c. Thêm vào đầu danh sách (Insert_first): thêm một nút có nội dung x vào đầu danh sách liên kết kép.

```
void Insert_first(NODEPTR &First, int x)
{
  NODEPTR p;
  p = New_node();
  p->info = x;
  if(First == NULL)  // truong hop danh sach rong
    p->right = NULL;
  else
  {
    // tao lien ket giua p va First
    p->right = First;
    First->left = p;
  }
   First = p;
   p->left = NULL;
}
```

d. Thêm nút mới vào sau nút có địa chỉ p (Insert_right): thêm một nút có nội dung x vào sau nút có địa chỉ p trong danh sách liên kết kép. Phép toán này cũng được dùng để thêm một nút vào cuối danh sách.

```
void Insert_right(NODEPTR p, int x)
{
  NODEPTR q, r;  // q la nut can them vao, p la nut truoc, r la nut sau
  if(p == NULL)
    printf("Nut p khong hien huu, khong them nut duoc\n");
  else
  {
    q = New_node();
    q->info = x;
```

```
            r = p->right;
            // tao hai lien ket giua r va q
            r->left = q;
            q->right = r;
            // tao hai lien ket giua p va q
            q->left = p;
            p->right = q;
          }
        }
```

e. <u>Thêm nút mới vào trước nút có địa chỉ p (Insert_left)</u>: thêm một nút có nội dung x vào trước nút có địa chỉ p trong danh sách liên kết kép.

```
void Insert_left(NODEPTR &First, NODEPTR p, int x)
        {
          NODEPTR q, r;  // q la nut can them vao, p la nut sau, r la nut truoc
          if(p == NULL)
            printf("Nut p khong hien huu, khong them nut duoc\n");
          else
          {
            if(p == First)  // them nut vao dau danh sach
              Insert_first(First, x);
            else
            {
              q = New_node();
              q->info = x;
              r = p->left;
              // tao hai lien ket giua r va q
              r->right = q;
              q->left = r;
              // tao hai lien ket giua p va q
              q->right = p;
              p->left = q;
            }
          }
        }
```

V.2.2. *<u>Duyệt danh sách</u>*:

Thông thường ta hay duyệt danh sách liên kết để thực hiện một công việc gì đó, như liệt kê dữ liệu trong danh sách hay đếm số nút trong danh sách...

a. <u>Duyệt xuôi</u>: Duyệt danh sách liên kết kép từ nút đầu cho tới nút cuối danh sách.

```
void Right_traverse(NODEPTR First)
{
  NODEPTR p;
   if(empty(First))
    printf("\n  (khong co doan nao)");
  else
  {
    p = First;  // p chi nut dau
    while(p != NULL)
    {
        printf("\n%8d, p->info);
        p = p->right;
    }
  }
}
```

b. <u>Duyệt ngược</u>: Duyệt danh sách liên kết kép từ nút cuối cho tới nút đầu danh sách.

```
void Left_traverse(NODEPTR First)
{
  NODEPTR p;
  if(empty(First))
    printf("\n  (khong co doan nao)");
  else
  {
    for (p=First; p->right!=NULL; p=p->right); // p chi toi nut cuoi
    while(p != NULL)
    {
      printf("\n%8d, p->info
      p = p->left;
    }
  }
}
```

V.2.3. *<u>Phép loại bỏ</u>*:

a. <u>Giải phóng vùng nhớ(free)</u>: Hàm này dùng để hủy nút đã cấp phát, và trả vùng nhớ về lại cho memory heap.

free(p) ; với p là biến con trỏ

b. <u>Kiểm tra danh sách liên kết rỗng hay không (Empty)</u>: hàm Empty trả về TRUE nếu danh sách liên kết vòng rỗng, và ngược lại.

```
int Empty(NODEPTR First)
```

```
{
  return(First == NULL ? TRUE : FALSE);
}
```

c. <u>Xóa phần tử đầu của danh sách (Delete_first)</u>: muốn xóa 1 phần tử khỏi danh sách liên kết thì ta phải kiểm tra xem danh sách có rỗng hay không; nếu danh sách có phần tử thì mới xóa được.

```
void Delete_first(NODEPTR &First)
{
  NODEPTR p;
  if(empty(First))  // truong hop danh sach rong
    printf("Danh sach rong, khong xoa nut duoc");
  else
  {
    p = First;  // p la nut can xoa
    if(First->right == NULL)  // truong hop danh sach co mot nut
      First = NULL;
    else
    {
      First = p->right;
      First->left = NULL;
    }
    free(p);
  }
 }
```

d. <u>Xóa phần tử có địa chỉ p (Delete_node)</u>:

```
void Delete_node(NODEPTR &First, NODEPTR p)
{
  NODEPTR q, r;
  if(p == NULL)
    printf("Nut p khong hien huu, khong xoa nut duoc\n");
  else
  {
    if(First == NULL)  // truong hop danh sach rong
      printf("Danh sach rong, khong xoa nut duoc");
    else
    {
      if(p == First)  // truong hop xoa nut dau
        Delete_first(First);
      else
      {
```

```
              q = p->left;   // q la nut truoc
              r = p->right;  // r la nut sau
              // tao hai lien ket giua q va r
              r->left = q;
              q->right = r;
              free(p);
            }
          }
        }
      }
```

e. **<u>Xóa toàn bộ danh sách (Clearlist)</u>**:

```
  void clearlist(NODEPTR &First)
  {
    while(First != NULL)
      Delete_first(Last);
  }
```

V.2.4. <u>*Tìm kiếm*</u> (Search_info):

Tìm nút đầu tiên trong danh sách liên kết kép có info bằng với x.

Hàm Search_info nếu tìm thấy x trong danh sách thì trả về địa chỉ của nút đó trong danh sách, nếu không có thì trả về trị NULL.

```
NODEPTR Search_info(NODEPTR First, int x)
{
  NODEPTR p;
  p = First;
  while(p->info != x && p != NULL)
    p = p->right;
  return(p);
}
```

<u>Ví dụ</u>:

Viết chương trình quản lý và điều hành tuyến xe lửa TP HCM - HA NOI bằng danh sách liên kết kép; mỗi nút của danh sách là một đoạn đường có ga trước, ga sau, chiều dài và thời gian xe lửa chạy trên đoạn đường đó.

Chương trình có các chức năng sau:

1. Thêm một đoạn đường
2. Xóa một đoạn đường
3. Xem toàn tuyến đường theo liên kết xuôi
4. Xem toàn tuyến đường theo liên kết ngược
5. Xem thông tin của đoạn đường thứ i

6. Hiệu chỉnh thông tin của đoạn đường thứ i
7. Báo lộ trình: nhập nơi đi và nơi đến, chương trình sẽ cho biết các ga trung gian phải đi qua, tổng chiều dài và tổng thời gian của lộ trình.

- Chương trình:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
#include <alloc.h>
#include <dos.h>
#include <ctype.h>
#define TRUE 1
#define FALSE 0
// Khai bao cau truc cua mot doan duong tren tuyen duong
typedef struct doan
{
  char gatruoc[12];
  char gasau[12];
  int chieudai;   // km
  int thoigian;   // thoi gian xe lua chay tren doan, tinh theo phut
};
// Khai bao cau truc cua mot nut
struct node
{
  doan info;
  struct node *left, *right;
};
typedef struct node *NODEPTR;
// Tac vu New_node: cap phat mot nut cho danh sach lien ket kep
NODEPTR New_node(void)
{
  NODEPTR p;
  p = (NODEPTR)malloc(sizeof(struct node));
  return(p);
}
// Tac vu initialize: khoi dong danh sach lien ket kep
void Initialize(NODEPTR &First)
{
  First = NULL;
}
```

```
// Tac vu empty: kiem tra danh sach lien ket kep co bi rong khong
int empty(NODEPTR First)
{
  return((First == NULL) ? TRUE : FALSE);
}
// Tac vu Insert_first: them nut vao dau danh sach lien ket
void Insert_first(NODEPTR &First, doan x)
{
  NODEPTR p;
  p = New_node();
  p->info = x;
  if(First == NULL)  // truong hop danh sach rong
    p->right = NULL;
  else
  {
    // tao lien ket giua p va First
    p->right = First;
    First->left = p;
}
   First = p;
   p->left = NULL;
}
/ Tac vu Insert_right: them nut moi sau nut p
void Insert_right(NODEPTR p, doan x)
{
  NODEPTR q, r;  // q la nut can them vao, p la nut truoc, r la nut sau
  if(p == NULL)
    printf("Nut p khong hien huu, khong them nut duoc\n");
  else
  {
    q = New_node();
    q->info = x;
    r = p->right;
    // tao hai lien ket giua r va q
    r->left = q;
    q->right = r;
    // tao hai lien ket giua p va q
    q->left = p;
    p->right = q;
  }
```

```
}
// Tac vu Insert_left: them nut moi truoc nut p
void Insert_left(NODEPTR &First, NODEPTR p, doan x)
{
  NODEPTR q, r;  // q la nut can them vao, p la nut sau, r la nut truoc
  if(p == NULL)
    printf("Nut p khong hien huu, khong them nut duoc\n");
  else
  {
    if(p == First)  // them nut vao dau danh sach
       Insert_first(First, x);
    else
    {
       q = New_node();
       q->info = x;
       r = p->left;
       // tao hai lien ket giua r va q
       r->right = q;
       q->left = r;
       // tao hai lien ket giua p va q
       q->right = p;
       p->left = q;
    }
  }
}
// Tac vu Delete_first: xoa nut o dau danh sach lien ket
void Delete_first(NODEPTR &First)
{
  NODEPTR p;
  if(empty(First))  // truong hop danh sach rong
    printf("Danh sach rong, khong xoa nut duoc");
  else
  {
    p = First;  // p la nut can xoa
    if(First->right == NULL)  // truong hop danh sach co mot nut
       First = NULL;
    else
    {
       First = p->right;
       First->left = NULL;
```

```
    }
    free(p);
  }
}
// Tac vu Delete_node: xoa nut co con tro la p
void Delete_node(NODEPTR &First, NODEPTR p)
{
  NODEPTR q, r;
  if(p == NULL)
    printf("Nut p khong hien huu, khong xoa nut duoc\n");
  else
  {
    if(First == NULL)  // truong hop danh sach rong
      printf("Danh sach rong, khong xoa nut duoc");
    else
    {
      if(p == First)  // truong hop xoa nut dau
        Delete_first(First);
      else
      {
        q = p->left;   // q la nut truoc
        r = p->right;  // r la nut sau
        // tao hai lien ket giua q va r
        r->left = q;
        q->right = r;
        free(p);
      }
    }
  }
}
// Tac vu Right_traverse: duyet danh sach tu trai sang phai (duyet xuoi)
void Right_traverse(NODEPTR First)
{
  NODEPTR p;
  int stt;
  if(empty(First))
    printf("\n  (khong co doan nao)");
  else
  {
    p = First;  // p chi nut dau
```

```
    stt = 1;
    while(p != NULL)
    {
              printf("\n%5d%12s%12s%7d%7d", stt++, p->info.gatruoc,
          p->info.gasau, p->info.chieudai, p->info.thoigian);
       p = p->right;
    }
  }
}
// Tac vu Left_traverse: duyet danh sach tu phai sang trai (duyet nguoc)
void Left_traverse(NODEPTR First)
{
  NODEPTR p;
  int stt;
  if(empty(First))
    printf("\n  (khong co doan nao)");
  else
  {
    for (p=First; p->right!=NULL; p=p->right); // p chi toi nut cuoi
    stt = 1;
    while(p != NULL)
    {
      printf("\n%5d%12s%12s%7d%7d", stt++, p->info.gasau,
       p->info.gatruoc, p->info.chieudai, p->info.thoigian);
      p = p->left;
    }
  }
}
// Tac vu Search_info1: tim ga truoc cua mot doan
NODEPTR Search_info1(NODEPTR First, char x[])
{
  NODEPTR p;
  p = First;
  while(strcmp(p->info.gatruoc, x) != 0 && p != NULL)
    p = p->right;
  return(p);
}
// Tac vu Search_info2: tim ga sau cua mot doan
NODEPTR Search_info2(NODEPTR First, char x[])
{
```

```
  NODEPTR p;
  p = First;
  while(strcmp(p->info.gasau, x) != 0 && p != NULL)
    p = p->right;
  return(p);
}
// Tac vu clearlist: xoa toan bo danh sach lien ket kep
void clearlist(NODEPTR &First)
{
  while(First != NULL)
    Delete_first(First);
}
int position(NODEPTR First, NODEPTR p)
{
      int vitri;
  NODEPTR q;
      q = First;
      vitri = 0;
      while(q != NULL && q != p)
      {
              q = q->right;
              vitri++;
      }
  if(q == NULL)
    return(-1);
      return(vitri);
}
void baolotrinh(NODEPTR &First, char noidi[], char noiden[], char c)
{
  NODEPTR p1, p2;
  int kc, tg;
  if(c == 'X')
  {
        p1 = Search_info1(First, noidi);
        if(p1 == NULL)
        {
          printf("Khong co noi di");
          return;
        }
       if(strcmp(noidi, noiden) == 0)
```

```
    {
      printf("Noi di trung noi den");
      return;
    }
    p2 = Search_info2(First, noiden);
    if(p2 == NULL)
    {
       printf("Khong co noi den");
       return;
    }
   if(position(First, p1) <= position(First, p2))
    {
     kc = tg = 0;
     while(p1 != p2)
    {
       kc = kc + p1->info.chieudai;
       tg = tg + p1->info.thoigian;
       printf("\n%s -> %s :  %d km  %d phut", p1->info.gatruoc,
              p1->info.gasau, p1->info.chieudai, p1->info.thoigian);
       p1 = p1->right;
    }
   kc = kc + p1->info.chieudai;
   tg = tg + p1->info.thoigian;
   printf("\n%s -> %s :  %d km  %d phut", p1->info.gatruoc,
          p1->info.gasau, p1->info.chieudai, p1->info.thoigian);
    printf("\nTong chieu dai lo trinh: %d km. Tong thoi gian van
           chuyen %d phut", kc, tg);
  }
  else
    printf("Khong di xuoi duoc");
  return;
}

if(c == 'N')
{
      p1 = Search_info2(First, noidi);
      if(p1 == NULL)
      {
        printf("Khong co noi di");
        return;
```

```
        }
        if(strcmp(noidi, noiden) == 0)
        {
          printf("Noi di trung noi den");
          return;
        }
        p2 = Search_info1(First, noiden);
        if(p2 == NULL)
        {
              printf("Khong co noi den");
              return;
        }
       if(position(First, p1) >= position(First, p2))
       {
         kc = tg = 0;
         while(p1 != p2)
         {
             kc = kc + p1->info.chieudai;
             tg = tg + p1->info.thoigian;
             printf("\n%s -> %s :  %d km  %d phut", p1->info.gasau,
                   p1->info.gatruoc, p1->info.chieudai, p1->info.thoigian);
             p1 = p1->left;
           }
           kc = kc + p1->info.chieudai;
           tg = tg + p1->info.thoigian;
           printf("\n%s -> %s :  %d km  %d phut", p1->info.gasau,
                p1->info.gatruoc, p1->info.chieudai, p1->info.thoigian);
           printf("\nTong chieu dai lo trinh: %d km. Tong thoi gian van
                    chuyen %d phut", kc, tg);
    }
    else
       printf("Khong di nguoc duoc");
    return;
  }
}
/* Tac vu nodepointer: xac dinh con tro chi nut thu i (i=0,1,2,...) trong
  danh sach lien ket kep */
NODEPTR nodepointer(NODEPTR First, int i)
{
  NODEPTR p;
```

```
  int vitri;
  p = First;      // p chi nut dau dslk vong
  vitri = 1;
  while(p != NULL && vitri < i)
  {
    p = p->right;
    vitri++;
  }
  return(p);
}
char menu ()
{  char chucnang;
  do
  {  clrscr();
    // menu chinh cua chuong trinh
    printf("\n\nCHUONG TRINH QUAN LY VA DIEU HANH
               TUYEN XE LUA TPHCM - HANOI\n");
    printf("   1: Them mot doan\n");
    printf("   2: Xoa mot doan\n");
    printf("   3: Xem lo trinh 1 (duyet xuoi)\n");
    printf("   4: Xem lo trinh 2 (duyet nguoc)\n");
    printf("   5: Xem thong tin cua doan thu i\n");
    printf("   6: Hieu chinh thong tin ve doan thu i\n");
    printf("   7: Bao lo trinh\n");
    printf("   0: Ket thuc chuong trinh\n");
    printf("Chuc nang ban chon: ");
    chucnang = getche();
    } while(chucnang < '0' || chucnang >'7') ;
    return chucnang;
}
// chuong trinh chinh
void main()
{
  NODEPTR First, p, p1;
  doan ga;
  int vitri;
  char c, chucnang;
  char noidi[12], noiden[12];
  char c_vitri[3], c_chieudai[10], c_thoigian[10];
  clrscr();
```

```
// khoi dong danh sach lien ket kep
Initialize(First);
do
{
   chucnang=menu();
   switch(chucnang)
  {
     case '1':
         {
            printf("\nVi tri them (1, 2, ...): ");
            gets(c_vitri);
            vitri = atoi(c_vitri);
            p = nodepointer(First, vitri-1);//p chi nut truoc nut can them
            if (vitri <=0 || (p==NULL && First !=NULL))
            {
                    printf("Vi tri khong hop le");
                    getche();
            }
            else
            {
             printf("Ten ga truoc: ");
             gets(ga.gatruoc);
             printf("Ten ga sau: ");
             gets(ga.gasau);
             printf("Chieu dai (km): ");
             gets(c_chieudai);
             ga.chieudai = atoi(c_chieudai);
             printf("Thoi gian (phut): ");
             gets(c_thoigian);
             ga.thoigian = atoi(c_thoigian);
             if(vitri == 1 || First ==NULL)
               Insert_first(First, ga);
             else
               Insert_right(p, ga);
         }
             break;
         }
         case '2':
         {
             printf("\nVi tri muon xoa(1,2,...): ");
```

```
        gets(c_vitri);
        vitri = atoi(c_vitri);
        p = nodepointer(First, vitri);
        if(p == NULL)
          printf("Vi tri khong hop le");
        else
        {
          if(vitri == 1)
            Delete_first(First);
          else
            Delete_node(First, p);
          printf("Da xoa xong ");
        }
        delay(2000);
        break;
    }
    case '3':
    {
        printf("\nXem lo trinh 1 (duyet xuoi): ");
        printf("\n  STT        TU        DEN     CD     TG");
        Right_traverse(First);
        getche();
        break;
    }
    case '4':
    {
        printf("\nXem lo trinh 2 (duyet nguoc): ");
        printf("\n  STT        TU        DEN     CD     TG");
        Left_traverse(First);
        getche();
        break;
    }
    case '5':
    {
        printf("\nVi tri doan muon xem thong tin(1,2,...): ");
        gets(c_vitri);
        vitri = atoi(c_vitri);
        p = nodepointer(First, vitri);
        if(p == NULL)
         printf("Vi tri khong hop le");
```

```
                    else
                      printf("\nDoan:%d  Tu:%s  Den:%s  Chieu dai:%d km
                                 Thoi gian:%d phut", vitri, p->info.gatruoc,
                                 p->info.gasau, p->info.chieudai, p-
>info.thoigian);
                    getche();
                    break;
                }
                case '6':
                {
                    printf("\nVi tri doan muon hieu chinh(1,2,...): ");
                    gets(c_vitri);
                    vitri = atoi(c_vitri);
                    p = nodepointer(First, vitri);
                    if(p == NULL)
                      printf("Vi tri khong hop le");
                    else
                    {
                      printf("\nDoan:%d  Tu:%s  Den:%s  Chieu dai:%d km
                            Thoi gian:%d phut\n", vitri, p->info.gatruoc,
                            p->info.gasau, p->info.chieudai, p->info.thoigian);
                      printf("Ten ga truoc: ");
                      gets(ga.gatruoc);
                      printf("Ten ga sau: ");
                      gets(ga.gasau);
                      printf("Chieu dai (km): ");
                      gets(c_chieudai);
                      ga.chieudai = atoi(c_chieudai);
                      printf("Thoi gian (phut): ");
                      gets(c_thoigian);
                      ga.thoigian = atoi(c_thoigian);
                      p->info = ga;
                    }
                    break;
                }
                case '7':
                {
                   printf("\nBan di xuoi hay nguoc (x/n): ");
                   c = toupper(getch());
```

```
                printf("\nCho biet noi di: ");
                gets(noidi);
                printf("\nCho biet noi den: ");
                gets(noiden);
                baolotrinh(First, noidi, noiden, c);
                getch();
                break;
            }
        }
    } while(chucnang != '0');
    // Xoa toan bo cac nut tren danh sach lien ket kep
    clearlist(First);
}
```

VI. stack & queue trÊn danh sÁch liÊn kết:

VI.1 Stack:

VI.1.1. *Khái niệm*:

Như ta đã biết, Stack là một danh sách mà việc thêm vào và loại bỏ một phần tử chỉ diễn ra cùng một đầu của danh sách, tức là theo cơ chế LIFO (Last In First Out). Trong chương 3, ta đã khảo sát Stack với cấu trúc dữ liệu là danh sách tuyến tính, việc thêm vào và loại bỏ diễn ra ở cuối danh sách. Với danh sách tuyến tính làm Stack thì Stack có điểm hạn chế về số lượng phần tử phải khai báo trước. Để khắc phục nhược điểm này, ta sẽ xây dựng Stack với cấu trúc dữ liệu là danh sách liên kết đơn. Việc thêm vào và loại bỏ sẽ diễn ra ở đầu danh sách.

- Khai báo: Ta khai báo biến sp (Stack Pointer) là con trỏ chỉ đến một danh sách là Stack, mỗi phần tử trong Stack là 1 số nguyên như sau:

```
struct node
{  int info ;
   struct node *next ;
};
typedef struct node *Stack;
Stack sp;
```

- Khởi tạo Stack : sp = NULL;

Lưu ý:

- Với Stack là danh sách liên kết, ta chỉ thực hiện các phép toán ở đầu danh sách liên kết.

- Không có trường hợp Stack đầy ; Stack rỗng khi Sp = NULL ;

VI.1.2. *Các phép toán trên Stack*:

Đối với Stack, có hai phép toán chủ yếu là thêm vào (Push) và loại bỏ

(Pop)

a. <u>Phép thêm vào (push)</u> : Thêm một phần tử có giá trị x vào đầu Stack.

```
void push(Stack &sp, int x)
{
  Stack p;
  p = new node;
  p->info = x;
  p->next = sp;
  sp = p;
}
```

b. <u>Phép loại bỏ (pop)</u>: Xóa phần tử khỏi Stack và trả cho chương trình gọi giá trị của phần tử vừa xóa.

```
int pop(Stack &sp, int &x)
{
  Stack p;
  if(sp==NULL) return 0;
   p = sp;    // nut can xoa la nut dau
   x =sp->info;
   sp = p->next;
   delete p;
   return 1;
}
```

VI.2. <u>Queue:</u>

VI.2.1.*<u>Khái niệm</u>*:

Queue là một danh sách hạn chế mà việc thêm vào được thực hiện ở đầu danh sách, và việc loại bỏ được thực hiện ở đầu còn lại của danh sách (FIFO - First In First Out). Queue chứa các phần tử có cùng kiểu dữ liệu. ở chương này, ta chỉ tổ chức queue theo danh sách liên kết.

- Vị trí để loại bỏ phần tử được gọi là Front

- Vị trí để thêm vào được gọi là Rear

Queue có hai phép toán chính:

- Insert_queue : thêm một phần tử vào hàng đợi; Trong trường hợp này ta không cần quan tâm hàng đợi bị tràn hay bị đầy.

- Delete_queue: loại bỏ phần tử khỏi Queue và trả về giá trị của phần tử vừa xóa; trước khi xóa, ta phải kiểm tra Queue có khác rỗng hay không.

<u>Lưu ý:</u>

- Phép toán Insert_queue thực hiện ở cuối hàng đợi, còn phép toán Delete_queue thực hiện ở đầu hàng đợi.

- Khai báo: Ta khai báo biến q có kiểu cấu trúc Queue gồm 2 thành phần front, rear là con trỏ chỉ đầu và cuối hàng đợi. Mỗi phần tử của hàng đợi là một nút chứa một số nguyên.

```
struct node
{
  int info;
  struct node *next;
};
typedef struct node *Hangdoi;
struct Queue
{
   Hangdoi Front, Rear;
};
struct Queue q;
```

- Khởi tạo hàng đợi: q.Front = NULL;

VI.2.2. *<u>Các phép toán trên Queue</u>:*

a. <u>Phép thêm vào</u> : Thêm vào cuối danh sách liên kết nên sẽ thay đổi giá trị của Rear

```
void Insert_queue(Queue &q, int x)
{
  Hangdoi p;
  p = new node;
  p->info = x;    p->next=NULL;
  if (q.Front==NULL)
    q.Front=p;
  else q.Rear->next=p;
  q.Rear=p;
}
```

b. <u>Phép loại bỏ</u> : Loại bỏ phần tử đầu danh sách liên kết, do đó thay đổi giá trị của Front

```
int Delete_queue(Queue &q,int &x)
{
  Hangdoi p;
  if(q.Front==NULL) return 0;
  p = q.Front;    // nut can xoa la nut dau
  x = p->info;
  q.Front = p->next;
  delete p;
     return 1;
}
```

Ví dụ: Viết chương trình đổi số không âm hệ decimal ra số hệ nhị phân, với Stack và Queue là danh sách liên kết đơn.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <math.h>
#define TRUE 1
#define FALSE 0
struct node
{
  int info;
  struct node *next;
};
typedef node *Stack;
Stack sp;
struct node_
{
  int info;
  struct node_ *next;
};
typedef node_ *Hangdoi;
struct Queue
{
   Hangdoi Front, Rear;
};
struct Queue q;
void push(Stack &sp, int x)
{
  Stack p;
  p = (Stack)malloc(sizeof(struct node));
  p->info = x;
  p->next = sp;
  sp = p;
}
int pop(Stack &sp)
{
  Stack p;
  int x;
  if(sp==NULL)
```

```
    {
      printf("\nStack rong");
      getche();
      exit(1);
    }
    else
    {
      p = sp;    // nut can xoa la nut dau
      x =sp->info;
      sp = p->next;
      free(p);
      return x;
    }
}
```

void Insert_queue(Queue &q, int x)

```
{
  Hangdoi p;
  p = (Hangdoi)malloc(sizeof(struct node));
  p->info = x;
  if (q.Front==NULL)
    q.Front=p;
  else q.Rear->next=p;
  q.Rear=p;
  p->next=NULL;
}
```

int Delete_queue(Queue &q)

```
{
  Hangdoi p;
  int x;
  if(q.Front==NULL)
  {
    printf("\nHang doi rong");
    getche();
    exit(1);
  }
  else
  {
    p = q.Front;    // nut can xoa la nut dau
    x = p->info;
    q.Front = p->next;
```

```
      free(p);
      return x;
    }
}
void main()
{
   int  sodu;
   float so;
   char c;
   clrscr();
   do
   {
      sp =NULL;  // khoi dong stack
      q.Front =NULL ; // khoi dong hang doi
      printf("\n\nNhap vao mot so thuc khong am: ");
      scanf("%e", &so);
      double positive;
      double r, le, nguyen;
      le = modf(so,&nguyen);
      do
      {
        sodu = (int) nguyen % 2;
        push(sp, sodu);  // push so du vao stack
        nguyen = int(nguyen / 2);
      } while (nguyen != 0);
    printf("So da doi la: ");
    while(sp!=NULL)
       printf("%d", pop(sp));  // pop so du ra khoi stack
  // Doi phan le ra so nhi phan
  if (le!=0)
  {
    printf(".");
    int i=0;
    do
    {      r =le*2;
           le = modf(r,&positive);
           Insert_queue(q, positive);
           i++ ;
    } while (i <8 && r!=1);
    while (q.Front!=NULL)
```

```
    {
      printf("%d", Delete_queue(q));
    }
  }
     printf("\n\nBan co muon tiep tuc khong? (c/k): ");
     c = getche();
   } while(c == 'c' || c == 'C');
}
```

BÀi tập:

1. Viết chương trình tạo một menu thực hiện các công việc sau:
   a. Nhập danh sách liên kết theo giải thuật thêm về cuối danh sách, mỗi phần tử gồm có các thông tin sau: mssv (int), và hoten (char hoten[30]).
   b. Liệt kê danh sách ra màn hình
   c. Cho biết tổng số nút trong danh sách liên kết, đặt tên hàm là Reccount (int Reccount (NODEPTR First))
   d. Thêm 1 phần tử có nội dung info (mssv, hoten) vào sau phần tử có thứ tự thứ i trong danh sách.

      Ghi chú: - Thứ tự theo qui ước bắt đầu là 1
      - Nếu (i = 0) thêm vào đầu danh sách
      - Nếu i > Reccount(First) thì thêm vào cuối danh sách.
   e. In ra họ tên của sinh viên có mã do ta nhập vào.
   f. Loại bỏ nút có mã do ta nhập vào, trước khi xóa hỏi lại "Bạn thật sự muốn xóa (Y/N) ? "
   g. Sắp xếp lại danh sách theo thứ tự mã số tang dần.
   h. Ghi toàn bộ danh sách vào file tên 'DSSV.DAT'
   i. Nạp danh sách từ file 'DSSV.DAT' vào danh sách liên kết. Nếu trong danh sách liên kết đã có nút thì xóa tất cả dữ liệu hiện có trong danh sách liên kết trước khi đưa dữ liệu từ file vào.
2. Viết chương trình tạo một danh sách liên kết theo giải thuật thêm vào đầu danh sách, mỗi nút chứa một số nguyên.
3. a) Viết hàm tên Delete_Node để xóa nút có địa chỉ p.

   b) Viết một hàm loại bỏ tất cả các nút có nội dung x trong danh sách liên kết First.
4. Viết hàm Copy_List trên danh sách liên kết để tạo ra một danh sách liên kết mới giống danh sách liên kết cũ.
5. Ghép một danh sách liên kết có địa chỉ đầu là First2 vào một danh sách liên kết có địa chỉ đầu là First1 ngay sau phần tử thứ i trong danh sách liên kết First1.
6. Viết hàm lọc danh sách liên kết để tránh trường hợp các nút trong danh sách liên kết bị trùng info.
7. Đảo ngược vùng liên kết của một danh sách liên kết sao cho:
   - First sẽ chỉ đến phần tử cuối
   - Phần tử đầu có liên kết là NULL.
8. Viết hàm Left_Traverse (NODEPTR First) để duyệt ngược danh sách liên kết.

9. Viết giải thuật tách một danh sách liên kết thành hai danh sách liên kết, trong đó một danh sách liên kết chứa các phần tử có số thứ tự lẽ và một danh sách liên kết chứa các phần tử có số thứ tự chẵn trong danh sách liên kết cũ.

10. Tạo một danh sách liên kết chứa tên học viên, điểm trung bình, hạng của học viên (với điều kiện chỉ nhập tên và điểm trung bình). Quá trình nhập sẽ dừng lại khi tên nhập vào là rỗng.

    Xếp hạng cho các học viên. In ra danh sách học viên thứ tự hạng tăng dần (Ghi chú : Cùng điểm trung bình thì cùng hạng).

11. Nhập hai đa thức theo danh sách liên kết. In ra tích của hai đa thức này.

    Ví dụ: Đa thức First1 : $2x^5+4x^2-1$

    Đa thức First2 : $10x^7-3x^4+x^2$

    $\Rightarrow$ Kết quả in ra : $20x^{12} + 34x^9 - 8x^7 - 12x^6 + 7x^4 - x^2$

    (Ghi chú : Không nhập và in ra các số hạng có hệ số bằng 0)

12. Viết giải thuật thêm phần tử có nội dung x vào danh sách liên kết có thứ tự tăng dần sao cho sau khi thêm danh sách liên kết vẫn có thứ tự tăng.

13. Loại bỏ phần tử có nội dung là x trong danh sách liên kết có thứ tự tăng dần.

14. Cho 2 danh sách liên kết First1, First2 có thứ tự tăng dần theo info. Viết giải thuật Merge để trộn 2 danh sách liên kết này lại sao cho danh sách liên kết sau khi trộn cũng có thứ tự tăng dần.

CHƯƠNG V

CÂY (TREE)

Cây là một cấu trúc dữ liệu rất thông dụng và quan trọng trong nhiều phạm vi khác nhau của kỹ thuật máy tính.

Ví dụ : Tổ chức các quan hệ họ hàng trong một gia phả, mục lục của một cuốn sách, xây dựng cấu trúc về cú pháp trong các trình biên dịch.

Trong chương trình này, chúng ta khảo sát các khái niệm cơ bản về cây, các phép toán trên cây nhị phân, cũng như các phép toán trên cây nhị phân cân bằng (AVL tree) và ứng dụng của hai loại cây nhị phân tìm kiếm (BST), cây nhị phân cân bằng (AVL tree).

I. ĐỊNH NGHĨA VÀ KHÁI NIỆM:

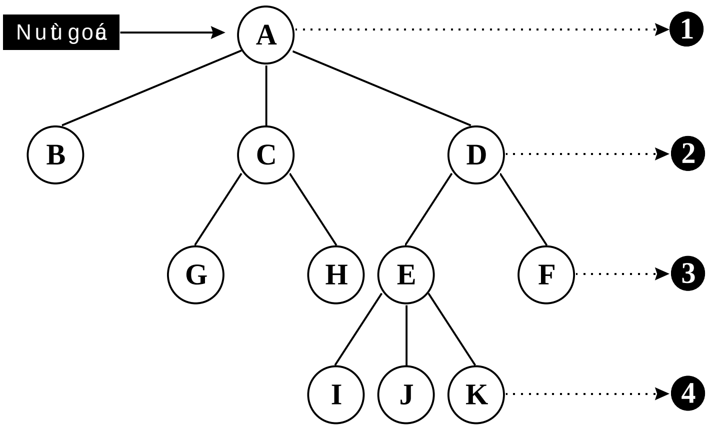
I.1. Một số khái niệm cơ bản:

1. Cây: Cây là tập hợp các phần tử gọi là nút, một nút (tương tự như một phần tử của danh sách) có thể có kiểu bất kỳ. Các nút được biểu diễn bởi 1 ký tự chữ, một chuỗi, một số ghi trong một vòng tròn.

Một số định nghĩa theo đệ quy

- Một nút đơn cũng chính là một cây.
- Các nút được gọi là ở cùng một cây khi có đường đi giữa các nút này.
- Một cây sẽ bao gồm một nút gốc (Root) và **m** cây con, trong mỗi cây con lại có một nút gốc và **m1** cây con nhỏ hơn v.v.
- Một cây không có một nút nào cả gọi là cây rỗng.

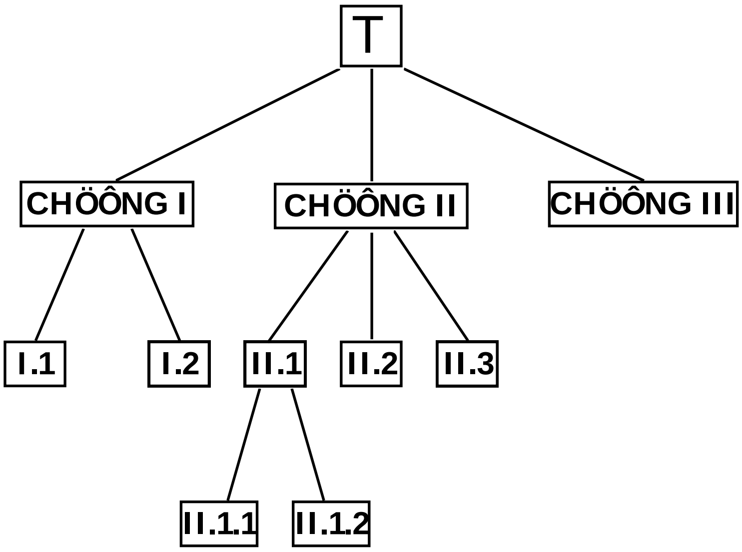
Ví dụ 1 :



- A là nút gốc với 3 cây con lần lượt có 3 nút gốc riêng là B, C, D
- Nút cha *(ancestor)*
  Nút con *(descendent)*
  A là nút cha của B, C, D
  G, H là nút con của C
  G, H không quan hệ cha con với A

Hình 5.1. Cây với nút gốc là A

Ví dụ 2 : Với đề cương một môn học T, ta có thể biểu diễn dạng cây như sau :



CHươNG I
I.1
I.2
CHươNG II
II.1
II.1.1
II.1.2
II.2
II.3
CHươNG III

Hình 5.2 Cây biểu diễn đề cương môn học

2. **<u>Nút cha (Ancestor)</u> :** Nút đứng trên của một nút được gọi là nút cha

   C là nút cha của G, H

   <u>Nút con (descendent)</u> : Nút đứng sau một nút khác được gọi là nút con của nút đó.

   Nút I, J, K là nút con của nút E

3. **<u>Bậc (degree)</u> :**

   - Bậc của nút là số cây con của nút đó.

     C có bậc là 2, E có bậc là 3 (Hình 5.1)

   - Bậc của cây là bậc lớn nhất của các nút trong cây.

     Cây trong hình 5.1 có bậc là 3.

     Cây bậc n được gọi là cây n phân như cây nhị phân, cây tam phân....

4. **<u>Nút lá và nút trung gian</u> :**

   - Nút lá là nút có bậc bằng 0 (tức là không có cây con nào) :

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

   <u>Ví dụ</u>: Trong hình 5.1, B, G, H, I, J, K, F là nút lá

   C, D, E là nút trung gian.

5. **<u>Mức của nút (level)</u>** : Nút gốc có mức là 1

   Mức của nút con = mức của nút cha + 1

   <u>Ví dụ</u>: trong hình 5.1,

A	có mức là 1
B, C, D	có mức là 2
G, H, E, F	có mức là 3

I, J, K   có mức là 4

6. **<u>Chiều cao của cây (height)</u>** : là mức lớn nhất của các nút lá trong cây.

<u>Ví dụ</u>: Cây trong hình 5.1 có chiều cao là 4

7. **<u>Thứ tự của các nút (order of nodes)</u>** : Nếu cây được gọi là có thứ tự thì phải đảm bảo vị trí của các nút con từ trái qua phải, tức là nếu thay đổi vị trí của một nút con bất kỳ thì ta đã có một cây mới.

<u>Ví dụ</u> :

Hình 5.3: Sau khi đổi vị trí của 2 nút B, C ta đã có cây mới.

8. **<u>Chiều dài đường đi</u> (Path length) :**

- Chiều dài đường đi của nút x: là số các nút đi từ nút gốc đến nút x.

<u>Ví dụ</u>: Trong hình 5.1:

Nút gốc A có chiều dài đường đi là 1

Nút B, C, D có chiều dài đường đi là 2

<u>Tổng quát</u>: một nút tại mức i có chiều dài đường đi là i

- Chiều dài đường đi của cây: là tổng của các chiều dài đường đi của tất cả các nút trong cây.

<u>Ví dụ</u>: Chiều dài đường đi của cây trong hình 5.1 là 31.

Chiều dài đường đi trung bình của cây:

$$P_i = (\sum_i n_i . i) / n$$

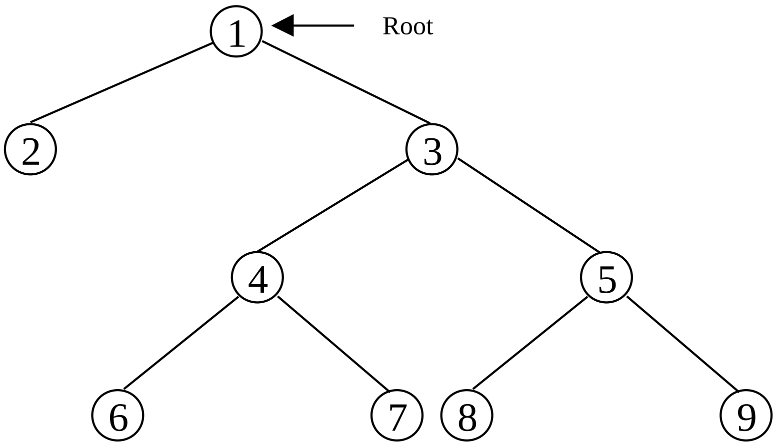
trong đó n_i là số các nút ở mức i và n là tổng số các nút trong cây.

I.2. <u>Cách biểu diễn cây</u>:

Để biểu diễn 1 cây, ta có nhiều cách như biểu diễn bằng đồ thị, bằng giản đồ, bằng chỉ số.. Nhưng thông thường, ta hay dùng dạng đồ thị để biểu diễn 1 cây như hình 5.1

I.3. <u>Biểu diễn thứ tự các nút trong cây</u> :

Một cây thường tổ chức các nút theo một thứ tự nhất định căn cứ vào một nội dung gọi là khóa của các nút. Có thể tổ chức cây có khóa tăng dần theo mức từ trái qua phải như ví dụ sau :



ROOT →1→2→3→4→5→6→7→8→9

Như vậy khi duyệt lại cây theo mức tăng dần và từ trái qua phải ta sẽ lại có được thứ tự các nút như trên.

Hình 5.4. Cây có thứ tự tăng dần theo mức từ trái qua phải

II. Cây nhị phân (Binary tree)

II.1. Định nghĩa :

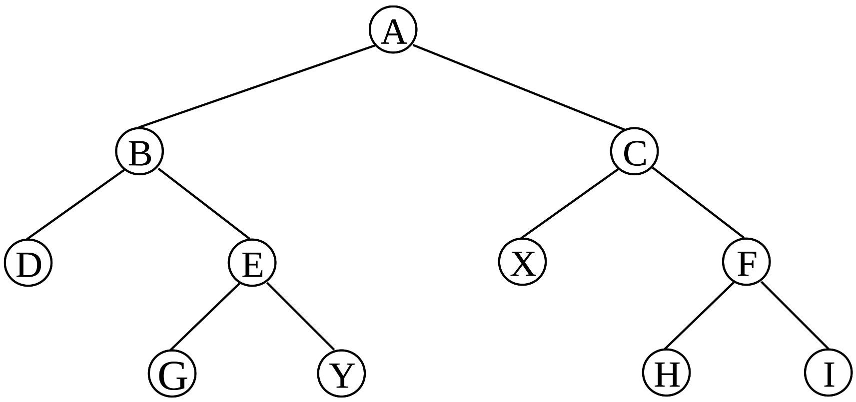
1. Cây nhị phân : Là cây có bậc bằng 2, tức là số nút con tối đa của một nút bất kỳ trong cây là 2.

Cây nhị phân có thể là một cây rỗng (không có nút nào) hoặc cây chỉ có một nút, hoặc cây chỉ có các nút con bên trái (Left Child) hoặc nút con bên phải (Right Child) hoặc cả hai.

Ví dụ: Hình 5.4 là cây nhị phân.

2. Các cây nhị phân đặc biệt:

- Cây nhị phân đúng: Một cây nhị phân được gọi là cây nhị phân đúng nếu nút gốc và tất cả các nút trung gian đều có 2 nút con.

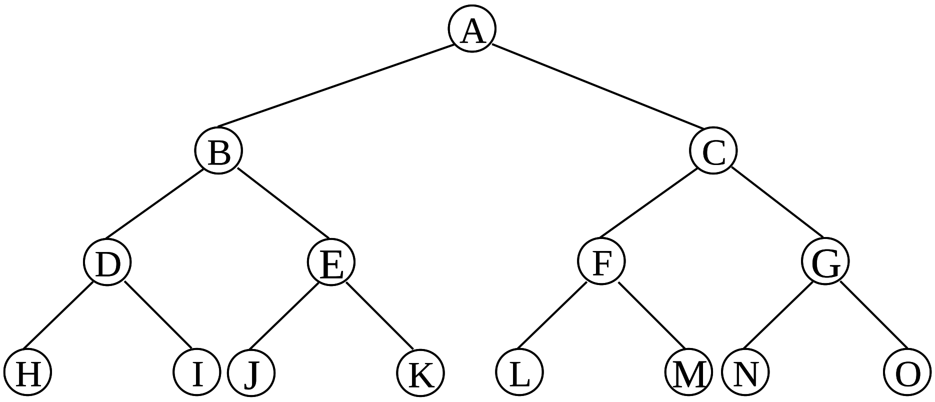


Hình 5.5. Cây nhị phân đúng

Ghi chú: nếu cây nhị phân đúng có n nút lá thì cây này sẽ có tất cả m= 2n-1 nút.

- Cây nhị phân đầy: Một cây nhị phân gọi là cây nhị phân đầy với chiều cao d thì:
  - Nó phải là cây nhị phân đúng và
  - Tất cả các nút lá đều có mức là d.

Hình 5.5 không phải là cây nhị phân đầy.

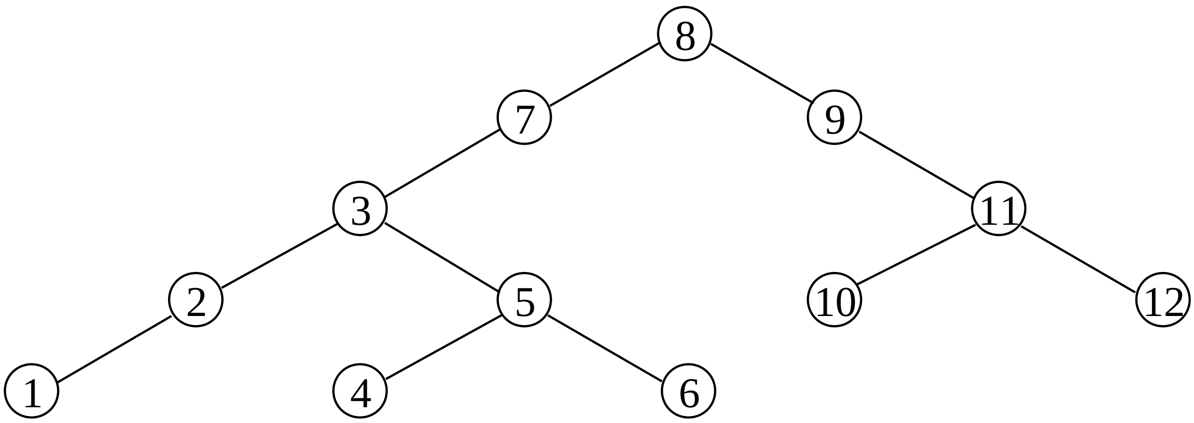


Hình 5.6. Cây nhị phân đầy.

Ghi chú: Cây nhị phân đầy là cây nhị phân có số nút tối đa ở mỗi mức.

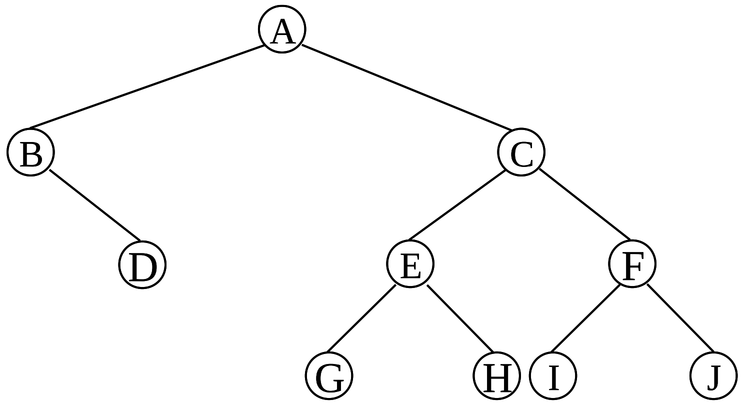
- Cây nhị phân tìm kiếm (Binary Search Tree - BST): Một cây nhị phân gọi là cây nhị phân tìm kiếm nếu và chỉ nếu đối với mọi nút của cây thì khóa của một nút bất kỳ phải lớn hơn khóa của tất cả các nút trong cây con bên trái của nó và phải nhỏ hơn khóa của tất cả các nút trong cây con bên phải của nó.

Ví dụ:



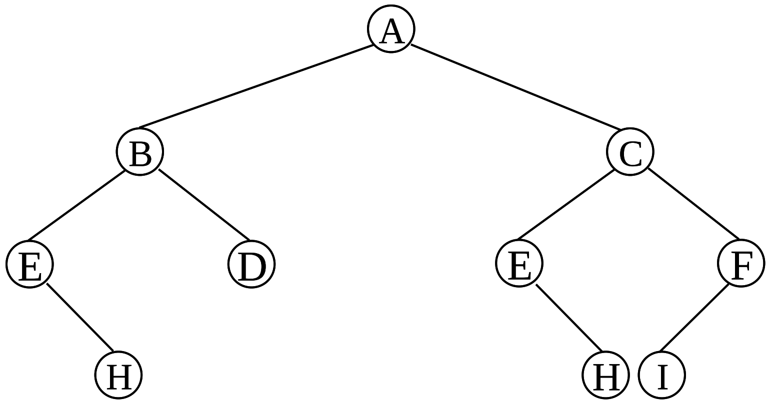
Hình 5.7 Cây nhị phân tìm kiếm (BST)

- Cây nhị phân cân bằng (AVL): Một cây nhị phân được gọi là cây nhị phân cân bằng nếu và chỉ nếu đối với mọi nút của cây thì chiều cao của cây con bên trái và chiều cao của cây con bên phải hơn kém nhau nhiều nhất là 1. (Theo Adelson - Velski và Landis).



Hình 5.8. Cây nhị phân cân bằng

- <u>Cây nhị phân cân bằng hoàn toàn</u>: Một cây nhị phân được gọi là cây nhị phân cân bằng hoàn toàn nếu và chỉ nếu đối với mọi nút của cây thì số nút của cây con bên trái và số nút của cây con bên phải hơn kém nhau nhiều nhất là 1.



Hình 5.9. Cây nhị phân cân bằng hoàn toàn

3. <u>Các phép duyệt cây (Traverse)</u> : Là quá trình đi qua các nút đúng một lần. Khi duyệt cây, ta thường dùng các cách duyệt cơ bản sau :

- ***Preorder*** - Tiền tự (NLR) duyệt qua nút gốc trước, sau đó đi qua cây con bên trái lại áp dụng Preorder cho cây con bên trái. Cuối cùng qua cây con bên phải, áp dụng Preorder cho cây con bên phải.

<u>Ví dụ</u>: Theo cây nhị phân 5.4, ta có:

ROOT → 1 → 2 → 3 → 4 → 6 → 7 → 5 → 8 → 9

- ***Inorder*** - Trung tự (LNR) : qua cây con bên trái duyệt trước (theo thứ tự LNR), sau đó thăm nút gốc, cuối cùng qua cây con bên phải (theo thứ tự LNR)

<u>Ví dụ</u>: Theo cây nhị phân 5.4, ta có:

ROOT → 2 → 1 → 6 → 4 → 7 → 3 → 8 → 5 → 9

- ***Postorder*** - Hậu tự (LRN) : qua cây con bên trái duyệt trước (theo thứ tự LRN), sau đó qua cây con bên phải (theo thứ tự LRN), cuối cùng thăm nút gốc.

<u>Ví dụ</u>: Theo cây nhị phân 5.4, ta có:

ROOT → 2 → 6 → 7 → 4 → 8 → 9 → 5 → 3 → 1

Ghi chú : Đối với cây ta có thể tổ chức thứ tự theo khóa là một nội dung của nút hoặc ta đặt thêm 1 field gọi là khóa của nút.

II.2. Các phép toán trên cây nhị phân:

- **Khai báo:** Để tổ chức dữ liệu theo cây nhị phân, ta có thể dùng một nội dung của dữ liệu để làm khóa sắp xếp và tổ chức cây theo nhiều cách khác nhau. Nhưng thông thường để thuận tiện cho việc tìm kiếm và thực hiện các phép toán khác trên cây, người ta tạo thêm một khóa riêng trong các phần tử và tạo ra cây nhị phân tìm kiếm.

Để khai báo biến **tree** quản lý một cây nhị phân, với nội dung info chứa số nguyên, ta khai báo như sau:

```
struct node
{
  int key;
  int info;
  struct node *left;
  struct node *right;
};
typedef struct node *NODEPTR;
NODEPTR tree;
```

II.2.1. *Tạo cây :*

a. Khởi tạo cây (Initialize): dùng để khởi tạo cây nhị phân, cho chương trình hiểu là hiện tại cây nhị phân rỗng.

```
void Initialize(NODEPTR &root)
{
  root = NULL;
}
```

Lời gọi hàm: Initialize(tree);

b. Cấp phát vùng nhớ (New_Node): cấp phát một nút cho cây nhị phân. Hàm New_Node này trả về địa chỉ của nút vừa cấp phát.

```
NODEPTR New_Node(void)
{
  NODEPTR p;
  p = new node;
  return(p);
}
```

Lời gọi hàm: p= New_Node();

c. Tạo cây BST (Create_Tree): Trong giải thuật tạo cây BST, ta có dùng hàm Insert_node.

Hàm Insert_node: Dùng phương pháp đệ qui thêm nút có khóa x, nội dung a vào cây có nút gốc root. Cây nhị phân tạo được qua giải thuật Create_Tree là cây nhị phân tìm kiếm (BST).

```
void Insert_node(NODEPTR &p, int x, int a)
{
  if(p == NULL)    // nút p hiện tại sẽ là nút lá
  {
    p = new node;
    p->key  = x;
    p->info = a;
    p->left = NULL;
    p->right = NULL;
  }
  else
   if(x < p->key )
     Insert_node(p->left, x, a);
   else if(x > p->key )
          Insert_node(p->right, x, a);
}
void Create_Tree(NODEPTR &root)
{
  int khoa, noidung;
  do
  { printf("Nhap khoa :");
    cin >> khoa;
    if (khoa !=0)
{
    printf("Nhap noi dung :");
    cin >> noidung;
    Insert_node(root,khoa,noidung);
 }
} while (khoa!=0); // khóa bằng 0 thì dừng nhập
}
```

Ghi chú : Để tạo cây nhị phân do biến tree quản lý, ta gọi:

```
Create_Tree(tree);
```

II.2.2. *Cập nhật cây* :

a. Giải phóng vùng nhớ (Free_Node): giải phóng vùng nhớ mà p đang trỏ đến.

```
delete p;
```

b. <u>Kiểm tra cây nhị phân rỗng hay không (Empty)</u>: hàm Empty trả về TRUE nếu cây nhị phân rỗng, và ngược lại.

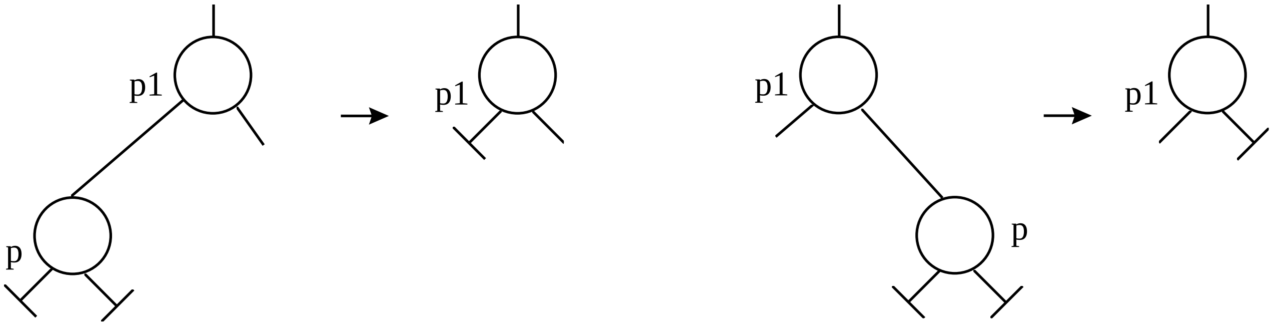
```
int Empty(NODEPTR root)
  return(root == NULL ? TRUE : FALSE);
}
```

<u>Lời gọi hàm</u> : Empty(tree)

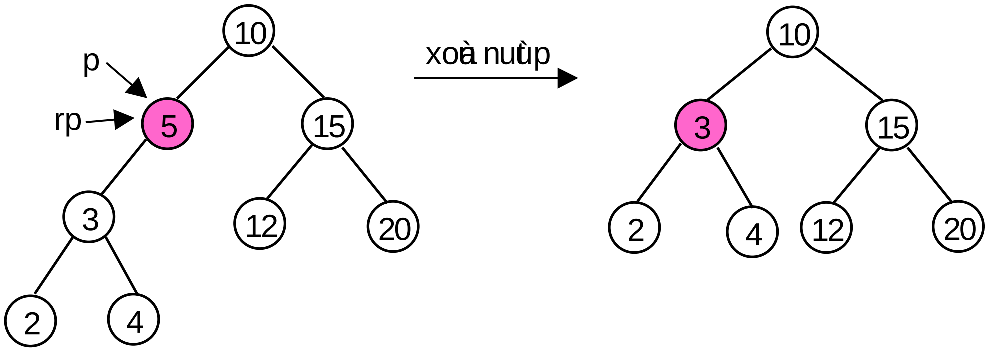
c. <u>Hủy bỏ một nút trong cây nhị phân BST (Remove)</u>:

Xóa nút có khóa là x trong cây nhị phân tìm kiếm sao cho sau khi xóa thì cây nhị phân vẫn là cây nhị phân tìm kiếm. Ta có 3 trường hợp :

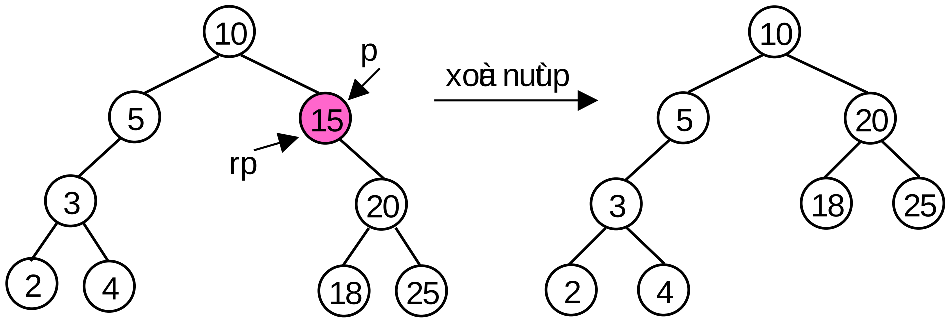
- *<u>Trường hợp 1</u>*: nút p cần xóa là nút lá. Việc xóa nút p chỉ đơn giản là hủy nút p



- *<u>Trường hợp 2</u>*: Nút p cần xóa có 1 cây con, thì ta cho rp chỉ tới nút p. Sau đó, ta tạo liên kết từ nút cha của p tới nút con của rp, cuối cùng hủy nút p.

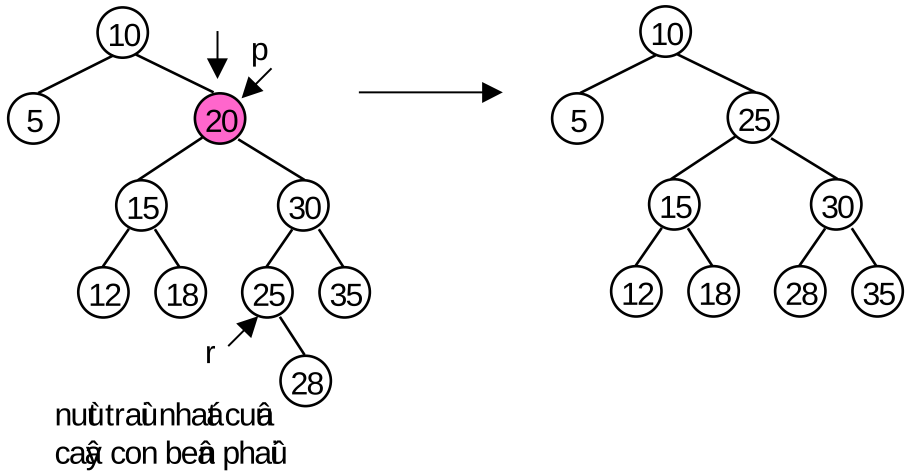


Hình 5.10. Xóa nút p trong trường hợp nút này có 1 cây con bên trái.



Hình 5.11. Xóa nút p trong trường hợp nút này có 1 cây con bên phải.

- *<u>Trường hợp 3</u>*: Nút p cần xóa có 2 cây con. Ta cho rp chỉ tới nút p. Do tính chất nút cực trái của cây con bên phải của p có khóa vừa lớn hơn khóa của p, nên để loại p thì ta sẽ cho r chỉ tới nút cực trái đó. Sau đó, ta sao chép nội dung và khóa của nút r vào nút mà rp đang chỉ tới. Ta tạo liên kết thích hợp để bứt nút rp ra khỏi cây nhị phân và cuối cùng xóa nút rp.



Hình 5.12. Xóa nút p trong trường hợp nút này có 2 cây con.

Hàm **Remove** xóa nút có khóa là x:

```
NODEPTR rp;
void  remove_case_3 ( NODEPTR  &r )
{
  if (r->left != NULL)
     remove_case_3 (r->left);
   //den day r la nut cuc trai cua cay con ben phai co nut goc la rp}
    else
    {
        rp->key = r->key;                //Chep noi dung cua r sang rp ";
        rp->info =r->info;               //  de lat nua free(rp)
        rp = r;
        r = r->right;
    }
  }
void  remove (int x ,  NODEPTR &p )
{
  if (p == NULL)  printf ("Khong tm thay");
  else
    if (x < p->key)   remove (x, p->left);
    else if (x > p->key)
              remove (x, p->right);
         else       // p->key = x
         {
            rp = p;
```

```
        if (rp->right == NULL)  p = rp->left;
            // p là nút lá hoac la nut chi co cay con ben trai
        else   if (rp->left == NULL)
                    p = rp->right;  // p là nut co cay con ben phai
                else remove_case_3 (rp->right);
        delete rp;
    }
}
```

Lời gọi hàm: Remove(x, tree); // x là khóa của nút muốn xóa

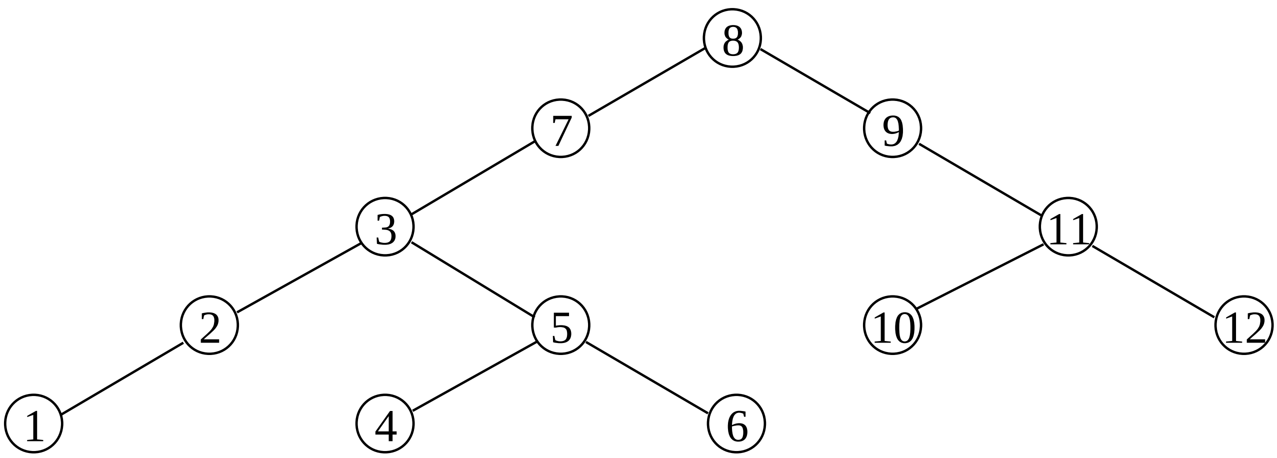
d. Tìm kiếm (Search): Tìm nút có khóa bằng x trên cây nhị phân BST có gốc là root. Nếu tìm thấy x trong cây thì trả về địa chỉ của nút có trị bằng x trong cây, nếu không có thì trả về trị NULL.

Do cây nhị phân là BST nên ta có thể tìm kiếm nhanh bằng phương pháp tìm kiếm nhị phân.

```
NODEPTR Search(NODEPTR root, int x)
{
  NODEPTR p;
  p = root;
  while(p != NULL && p->key !=x)
    if(x < p->key)
        p = p->left;
     else
        p = p->right;
return(p);
}
```

Lời gọi hàm: p=Search(tree, x);

II.2.3. *Các phép duyệt cây*: Có 3 cách duyệt cơ bản là NLR, LNR, LRN và một cách đặc biệt là duyệt cây theo mức. Xét cây sau :



Hình 5.13 Cây nhị phân minh họa cho phần duyệt cây.

a. Duyệt cây theo thứ tự NLR (Preorder):

◆ Giải thuật đệ qui:

```
void Preorder (NODEPTR p)
{ if(p != NULL)
  { printf("%d ", p->info);
    Preorder(p->left);
    Preorder (p->right);
  }
}
```

◆ Giải thuật không đệ qui: Đi từ gốc qua hết nhánh bên trái, mỗi lần qua một nút giữ lại địa chỉ nút con bên phải của nó trong Stack.

Sau khi duyệt đến nút lá tận cùng bên trái, ta lần lượt quay về duyệt các nút bên phải bằng cách lấy địa chỉ của nút này từ Stack. Tại mỗi nút lấy ra, ta lại duyệt sang nhánh trái của nó và cất địa chỉ nút con bên phải vào Stack. Quá trình này sẽ dừng lại khi ta đã duyệt hết các nút của cây nhị phân.

```
void Pretrav (NODEPTR root)
{
  const STACKSIZE = 500;
  NODEPTR Stack[STACKSIZE];
  int sp=  -1;  // Khoi tao Stack rong
  NODEPTR p=root;
  while (p!=NULL )
  {
    cout << p->info << " "; // “xu ly p”

    if (p->right != NULL)
         Stack[++sp]= p->right;

    if (p->left != NULL)
         p=p->left;
    else  if (sp==-1)
        break;
    else
        p=Stack[sp--];
  }

}
```

b. Duyệt cây theo thứ tự LNR (Inorder):

◆ Giải thuật đệ qui:

```
void Inorder(NODEPTR p)
{ if(p != NULL)
  {  Inorder(p->left);
     printf("%d ", p->info);
     Inorder(p->right);
  }
}
```

◆ Giải thuật không đệ qui:

(i) Đi từ nút gốc qua hết nhánh bên trái, mỗi lần qua một nút ta giữ lại địa chỉ của nút đó trong Stack.

(ii) Lấy địa chỉ một nút trong Stack ra, sau đó duyệt sang nhánh phải của nút vừa lấy trong Stack.

(iii) Quay về (i) cho tới khi Stack rỗng thì dừng.

```
void Intrav(NODEPTR root)
{
  const STACKSIZE = 500;
  NODEPTR Stack[STACKSIZE];
  NODEPTR p=root;
  int sp=-1;// Khoi tao Stack rong
  do
  {
    while (p != NULL)
    {
      Stack[++sp]= p;
      p= p->left;
    }
    if (sp != -1)
    {
      p=Stack[sp--];
      cout << p->info << " ";
      p=p->right;
    }
    else break;
  } while (1);
}
```

c. Duyệt cây theo thứ tự LRN (Posorder):

◆ Giải thuật đệ qui:

```
void Posorder(NODEPTR p)
{  if(p != NULL)
   {  Posorder(p->left);
      Posorder(p->right);
      printf("%d ", p->info);
   }
}
```

◆ Giải thuật không đệ qui: So với hai cách duyệt trước, cách duyệt LRN không đệ quy sẽ khó hơn vì ta phải giữ cả nút cha và nút con bên phải vào Stack. Nút cha được đưa vào Stack trước, nút con bên phải đưa vào sau để lấy ra trước.

Cách thực hiện :

(1) Nếu nút có nút con bên trái và nút con bên phải, ta cất nút này và nút con bên phải của nó vào Stack và đi qua nút con bên trái.

(2) Tiếp tục bước trên cho đến khi tới nút lá tận cùng bên trái, duyệt nút này

(3) Lấy nút trong Stack ra, nếu nút này có nút con ta lặp lại bước 1; nếu không có và là nút phải thì duyệt nút này và nút cha của nó (trong Stack)

(4) Lặp lại các bước (1), (2) và (3) cho đến khi Stack rỗng.

Stack sẽ gồm hai thành phần và được khởi tạo bằng một phần tử có địa chỉ = NULL.

```
void Postrav(NODEPTR root)
{
  const STACKSIZE = 500;
  struct phantu
  {
    NODEPTR diachi;
    int kieu;         //de danh dau la nut cha hay nut con ben phai
  };                      // kieu = TRUE -> nut con ben phai
                   // kieu= FALSE -> nut cha
  phantu Stack[STACKSIZE];
  int typ, sp;
  NODEPTR p=root;
  sp=0;
  typ=TRUE;
  Stack[0].diachi=NULL; // Khoi tao Stack
  do
  {
    while (p != NULL && typ)
    {
      Stack[++sp].diachi= p;
```

```
            Stack[sp].kieu= FALSE;
            if (p->right != NULL)
            {
             Stack[++sp].diachi= p->right;
             Stack[sp].kieu= TRUE;
            }
            p= p->left;
        }
        if (p != NULL)
               cout << p->info << " ";
        p=Stack[sp].diachi;
        typ=Stack[sp--].kieu;
      } while (p!=NULL);
   }
```

d. <u>Duyệt cây theo mức</u>: Ngoài 3 cách duyệt cơ bản trên, ta có thể duyệt cây theo mức từ mức thấp đến mức cao, trong từng mức thì duyệt từ trái qua phải.

Duyệt cây theo mức :

Root → 8 → 7 → 9 → 3 → 11 → 2 → 5 → 10 → 12 → 1 → 4 → 6

Để thực hiện phép duyệt, ta dùng một hàng để giữ địa chỉ các nút, với tổ chức hàng là một danh sách liên kết.

```
struct node
{
     NODEPTR diachi ;
     struct node *next;
};
typedef node *Node_queue;
struct Queue
{
     Node_queue Front, Rear;
} q;
void Insert_queue(Queue &q, NODEPTR x)
{
   Node_queue p;
   p = new node;
   p->diachi = x;  p->next=NULL;
   if (q.Front==NULL)
      q.Front=p;
   else q.Rear->next=p;
```

```
        q.Rear=p;
      }
   NODEPTR Delete_queue(Queue &q)
   {
    Node_queue p;
    NODEPTR x;
    if(q.Front==NULL)
    {
      printf("\nHang doi rong");
      getche();
      exit(1);
    }
    else
    {
      p = q.Front;    // nut can xoa la nut dau
      x = p->diachi;
      q.Front = p->next;
      delete p;
      return x;
    }
  }
```

Giải thuật duyệt như sau:

- Đưa nút gốc vào hàng

- while (hàng != rỗng)

  + Lấy 1 nút ra khỏi hàng

  + Xử lý nút này

  + Đưa các nút con của nút này vào hàng nếu có

Do hàng được tổ chức theo kiểu FIFO, nên ta sẽ luôn luôn xử lý nút gốc ở mức thấp trước, và do đưa cả 2 nút con vào hàng theo thứ tự nút con bên trái trước nên khi lấy ra khỏi hàng ta sẽ xử lý được tất cả các nút trong cùng một mức theo thứ tự từ trái qua phải.

* Cài đặt:

```
 void leveltrav (NODEPTR root)
 {
  NODEPTR p;
  q.Front=NULL;
  q.Rear = NULL;
  if (root!=NULL)
    Insert_queue(q, root);
```

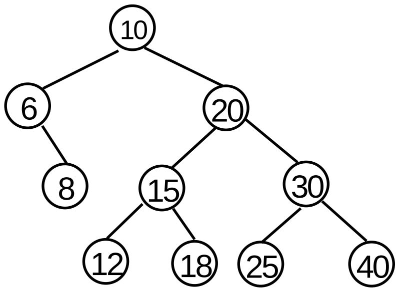
```
while (q.Front !=NULL)
{
  p=Delete_queue(q);
  cout << p->info << " ";
  if (p->left !=NULL)
    Insert_queue(q,p->left);
  if (p->right !=NULL)
    Insert_queue(q,p->right);
 }
}
```

III. cây nhị phân TÌM KIếM cân bằng (AVL):

Chúng ta tạo cây nhị phân tìm kiếm mục đích là để tìm khóa cho nhanh, tuy nhiên để tăng tốc độ tìm kiếm thì cây cần phải cân đối về 2 nhánh theo từng nút trong cây. Do vậy, ta sẽ tìm cách tổ chức lại cây BST sao cho nó cân bằng.

III.1. Định nghĩa:

- Cây nhị phân tìm kiếm cân bằng (AVL) là cây nhị phân tìm kiếm mà tại tất cả các nút của nó chiều cao của cây con bên trái của nó và chiều cao của cây con bên phải chênh lệch nhau không quá một.



Hình 5.14. Cây nhị phân tìm kiếm cân bằng

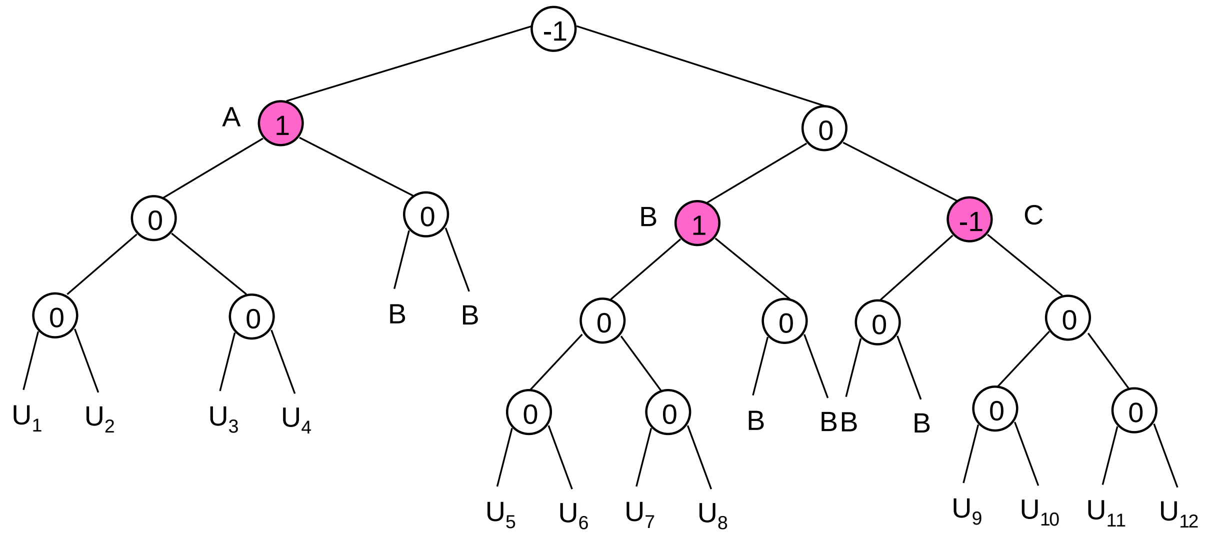
Lưu ý: Với cây AVL, việc thêm vào hay loại bỏ 1 nút trên cây có thể làm cây mất cân bằng, khi đó ta phải cân bằng lại cây. Tuy nhiên việc cân bằng lại trên cây AVL chỉ xảy ra ở phạm vi cục bộ bằng cách xoay trái hoặc xoay phải ở một vài nhánh cây con nên giảm thiểu chi phí cân bằng.

- **Chỉ số cân bằng** (balance factor) của một nút p trên cây AVL= **lh(p) - rh(p)**

Trong đó: lh (p) là chiều cao của cây con bên trái của p
rh(p) là chiều cao của cây con bên phải của p

Ta có các trường hợp sau:

bf(p) = 0 nếu lh(p) = rh(p) — nút p cân bằng
bf(p) = 1 nếu lh(p) = rh(p) +1 — nút p bị lệch về trái
bf(p) = -1 nếu lh(p) = rh(p) -1 — nút p bị lệch về phải



Hình 5.15. Minh họa các vị trí có thể thêm nút lá vào cây AVL, khi thêm nút lá vào 1 trong các vị trí B thì cây vẫn cân bằng, khi thêm nút lá vào 1 trong các vị trí U thì cây sẽ mất cân bằng. Các số trên cây là chỉ số cân bằng của các nút trước khi thêm nút

III.2. Các phép toán trên cây AVL:

*** Khai báo:** Ta khai báo cây AVL với mỗi nút có thêm trường bf cho biết chỉ số cân bằng của nút đó.

```
struct nodetype
{
  int key;
  int info;
  int bf;
  struct nodetype *left, *right;
};
typedef struct nodetype *NODEPTR;
```

III.2.1. *Thêm nút*:

- Nội dung**:** Thêm 1 nút có khóa x, nội dung a vào cây nhị phân tìm kiếm cân bằng sao cho sau khi thêm thì cây nhị phân vẫn là cây nhị phân tìm kiếm cân bằng.

- Giải thuật:
  - Thêm nút vào cây như bình thường, nghĩa là nút vừa thêm sẽ là nút lá.
  - Tính lại chỉ số cân bằng của các nút có bị ảnh hưởng
  - Kiểm tra xem cây có bị mất cân bằng hay không? Nếu cây bị mất cân bằng thì ta cân bằng lại cây.

* ***Trước hết, ta hãy xét xem các trường hợp nào khi thêm nút làm cây bị mất cân bằng***.

Xem cây hình 5.15, ta nhận thấy:

- Nếu thêm nút vào 1 trong 6 vị trí B trên cây thì cây vẫn cân bằng.

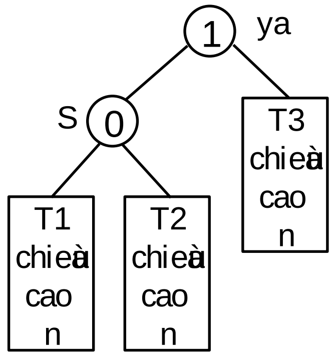
- Nếu thêm nút vào 1 trong các vị trí U1→U12 trên cây thì cây sẽ mất cân bằng.

+ Thêm các nút vào sau bên trái của nút A (bf_A = 1) thì cây sẽ bị mất cân bằng vì nút A đang bị lệch trái. Đó là các vị trí U1, U2, U3, U4.

+ Thêm các nút vào sau bên phải của nút C (bf_C = -1) thì cây sẽ bị mất cân bằng vì nút C đang bị lệch phải. Đó là các vị trí U9, U10, U11, U12.

Tóm lại: Ta có 2 trường hợp khi thêm nút x vào cây AVL làm cây mất cân bằng, đó là thêm các nút vào sau bên trái của nút có bf = 1, và thêm các nút vào sau bên phải của nút có bf = -1.

* ***Cân bằng lại cây***: Gọi ya là nút trước gần nhất bị mất cân bằng khi thêm nút x vào cây AVL. Do cả 2 trường hợp bị mất cân bằng khi thêm nút x là tương tự nhau nên ta chỉ xét trường hợp bf_{ya}=1 và nút lá thêm vào là nút sau bên trái của nút ya.



Hình 5.16. Nhánh cây con nút gốc ya trước khi thêm nút.

Nhận xét:

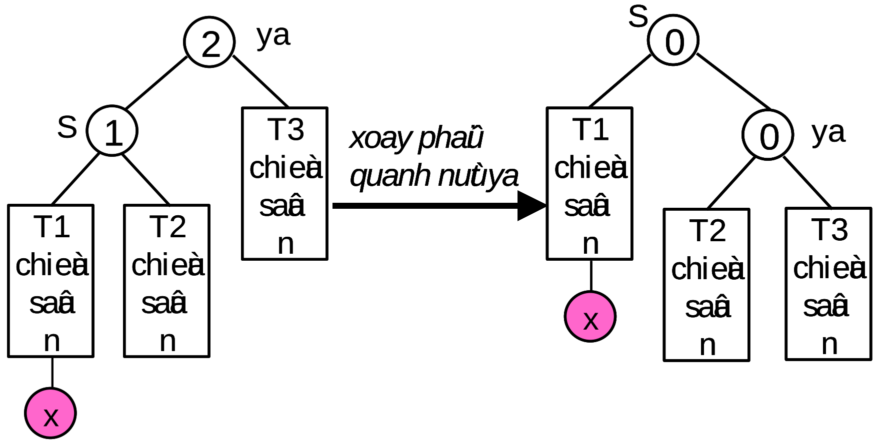
- Vì nút ya có bf_{ya} = 1 nên nút ya chắc chắn có nút con bên trái s với bf_s = 0

- Vì ya là nút gần nhất có bf là 1 nên nút s và các nút trước khác của nút x (sẽ thêm vào) có bf là 0.

- Độ cao (T1) = Độ cao(T2) = Độ cao(T3)

Trường hợp 1a: Nếu thêm nút mới x vào vị trí nút sau bên trái của s (thuộc nhánh T1) ta xoay phải quanh nút ya

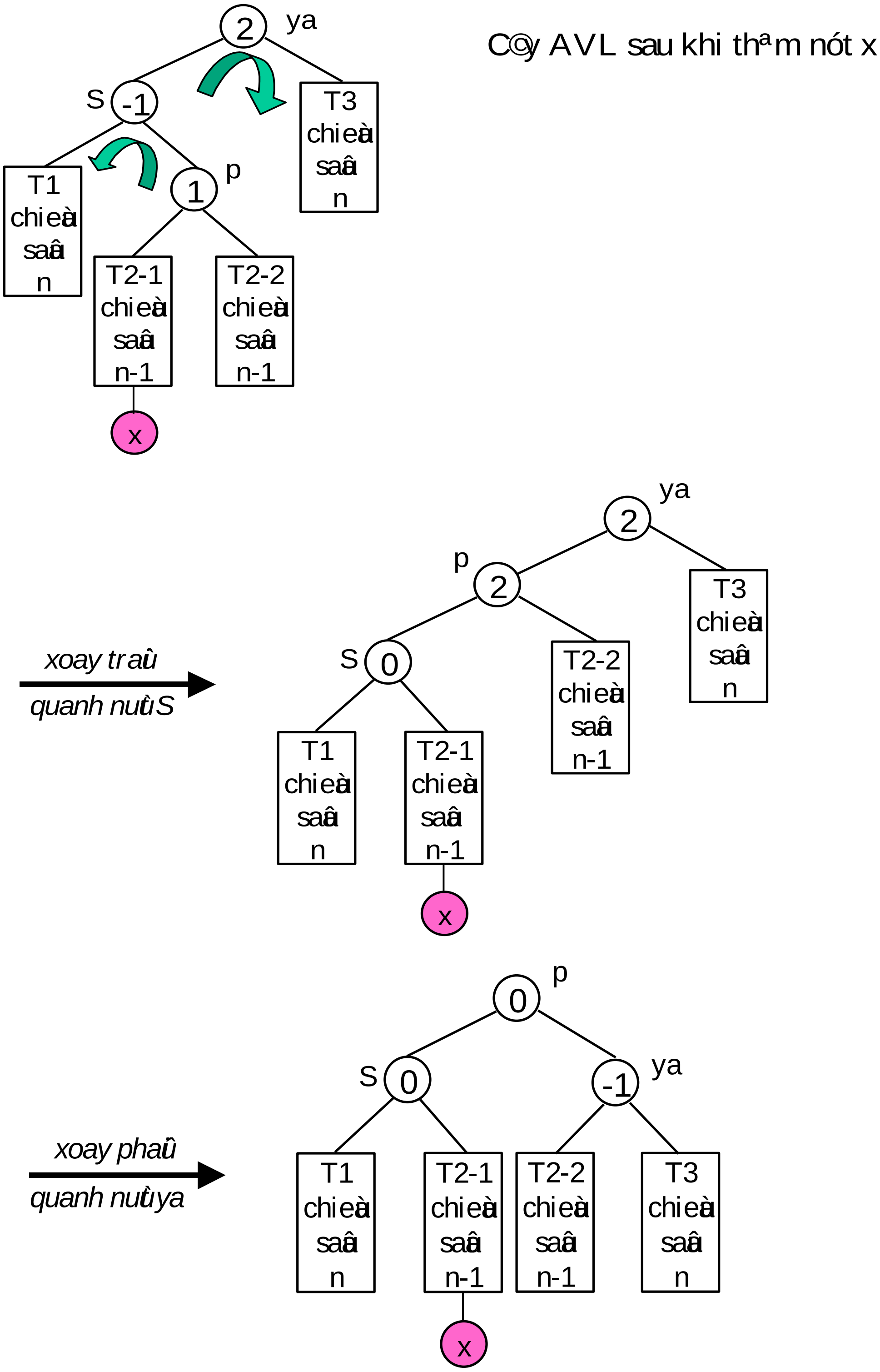
- Nút s sẽ là nút gốc mới của nhánh cây này với bf_s = 0
- Nút ya là nút con bên phải của s với bf_{ya} = 0.



Hình 5.17. Xoay phải quanh nút ya để cân bằng lại cây.

Trường hợp 1b: Nếu thêm nút mới x vào vị trí nút sau bên phải của s (thuộc nhánh T2) ta xoay 2 lần (xoay kép): xoay trái quanh nút s và xoay phải quanh nút ya

- Nút p sẽ là nút gốc mới của nhánh cây này với $bf_p = 0$
- Nút ya là nút con bên phải của p với $bf_{ya} = -1$
- Nút s là nút con bên trái của p với $bf_s = 0$



Hình 5.18. Xoay kép (xoay trái quanh nút s, xoay phải quanh nút ya)
để cân bằng lại cây.

Bảng sau đây phân biệt các trường hợp cây bị mất cân bằng khi thêm nút và các phép xoay cây tương ứng để cân bằng lại cây:

Trường hợp	**Trước khi thêm nút x**	**Sau khi thêm nút x**	**Các phép xoay cây và chỉ số cân bằng mới**
1.a	$bf_{ya} = 1$ $bf_s = 0$	$bf_{ya} = 2$ $bf_s = 1$	*Xoay phải quanh nút ya* bf_s=0, bf_{ya} = 0
1.b	$bf_{ya} = 1$ $bf_s = 0$	$bf_{ya} = 2$ $bf_s = -1$	*Xoay kép* 1. Xoay trái quanh nút s 2. Xoay phải quanh nút ya bf_s=0, bf_{ya} = -1
2.a	$bf_{ya} = -1$ $bf_s = 0$	$bf_{ya} = -2$ $bf_s = -1$	*Xoay trái quanh nút ya* bf_s=0, bf_{ya} = 0
2.b	$bf_{ya} = -1$ $bf_s = 0$	$bf_{ya} = -2$ $bf_s = 1$	*Xoay kép* 1. Xoay phải quanh nút s 2. Xoay trái quanh nút ya bf_s=0, bf_{ya} = 1

- Giải thuật:

➪ **Phép xoay trái (Rotate_Left)**: xoay trái cây nhị phân tìm kiếm có nút gốc là root, yêu cầu root phải có nút con bên phải (gọi là nút p). Sau khi xoay trái thì nút p trở thành nút gốc, nút gốc cũ trở thành nút con bên trái của nút gốc mới.

Phép xoay trái trả về con trỏ chỉ nút gốc mới.

```
NODEPTR Rotate_Left(NODEPTR root)
{
  NODEPTR p;
  if(root == NULL)
    printf("Khong the xoay trai vi cay bi rong.");
  else
    if(root->right == NULL)
            printf("Khong the xoay trai vi khong co nut con ben phai.");
    else
    {
            p = root->right;
            root->right = p->left;
            p->left = root;
    }
  return p;
```

```
}
```

➪ **Phép xoay phải (Rotate_Right)**: xoay phải cây nhị phân tìm kiếm có nút gốc là root, yêu cầu root phải có nút con bên trái (gọi là nút p). Sau khi xoay phải thì nút p trở thành nút gốc, nút gốc cũ trở thành nút con bên phải của nút gốc mới.

Phép xoay phải trả về con trỏ chỉ nút gốc mới.

```
NODEPTR Rotate_Right(NODEPTR root)
{
  NODEPTR p;
  if(root == NULL)
     printf("Khong the xoay phai vi cay bi rong.");
  else
     if(root->left == NULL)
          printf("Khong the xoay phai vi khong co nut con ben trai.");
     else
     {
          p = root->left;
          root->left = p->right;
          p->right = root;
     }
  return p;
}
```

➪ **Thêm nút (Insert)**: thêm nút có khóa x, nội dung a vào cây AVL:

- Thêm nút theo giải thuật thêm nút vào cây nhị phân tìm kiếm .
- Cân bằng lại cây bằng cách xoay đơn hay xoay kép

```
void Insert(NODEPTR &pavltree, int x, int a)
{
  NODEPTR fp, p, q,      // fp là nút cha của p, q là con của p
        fya, ya,         /* ya là nút trước gần nhất có thể mất cân bằng
                            fya là nút cha của ya */
        s;               // s là nút con của ya theo hướng mất cân bằng
  int imbal;             /* imbal =  1 nếu bị lệch về nhánh trái
                                  = -1 nếu bị lệch về nhánh phải */
  // Khởi động các giá trị
  fp = NULL;   p = pavltree;
  fya = NULL;   ya = p;
// tim nut fp, ya va fya, nut moi them vao la nut la con cua nut fp
  while(p != NULL)
  {
```

```
    if (x == p->info)  // bi trung noi dung
      return;
    if (x < p->info)
        q = p->left;
    else
        q = p->right;
    if(q != NULL)
      if(q->bf != 0) // truong hop chi so can bang cua q la 1 hay -1
      { fya = p;
        ya = q;
      }
    fp = p;
    p = q;
  }
// Thêm nút mới (nut la) la con cua nut fp
   q = New_Node();              // cấp phát vùng nhớ
   q->key =x;  q->info = a;  q->bf = 0;
   q->left = NULL;  q->right = NULL;
   if(x < fp->info)
    fp->left = q;
  else
    fp->right = q;
  /* Hieu chinh chi so can bang cua tat ca cac nut giua ya va q, neu bi lech
    ve phia trai thi chi so can bang cua tat ca cac nut giua ya va q deu la
    1, neu bi lech ve phia phai thi chi so can bang cua tat ca cac nut giua
    ya va q deu la -1 */
  if(x < ya->info)
    p = ya->left;
  else
    p = ya->right;
  s = p;     // s la con nut ya
  while(p != q)
  { if(x < p->info)
    { p->bf = 1;
      p = p->left;
    }
    else
    {   p->bf = -1;
        p = p->right;
    }
  }
```

```
// xac dinh huong lech
if(x < ya->info)
  imbal = 1;
else
  imbal = -1;
if(ya->bf == 0)
{ ya->bf = imbal;
  return;
}
if(ya->bf != imbal)
{ ya->bf = 0;
  return;
}
if(s->bf == imbal)   // Truong hop xoay don
{ if(imbal == 1)    // xoay phai
    p = Rotate_Right(ya);
  else              // xoay trai
    p = Rotate_Left(ya);
  ya->bf = 0;
  s->bf = 0;
}
else                // Truong hop xoay kep
{ if(imbal == 1)    // xoay kep trai-phai
  { ya->left = Rotate_Left(s);
    p = Rotate_Right(ya);
  }
  else              // xoay kep phai-trai -
  {  ya->right = Rotate_Right(s);
     p = Rotate_Left(ya);
  }
  if(p->bf == 0)    // truong hop p la nut moi them vao
  { ya->bf = 0;
    s->bf = 0;
  }
  else
    if(p->bf == imbal)
    { ya->bf = -imbal;
      s->bf = 0;
    }
    else
    { ya->bf = 0;
```

```
            s->bf = imbal;
          }
        p->bf = 0;
      }
      if(fya == NULL)
        pavltree = p;
      else
        if(ya == fya->right)
          fya->right = p;
        else
          fya->left = p;
   }
```

* Để tạo cây nhị phân tìm kiếm cân bằng, ta sử dụng giải thuật sau:

```
void Create_AVLTree(NODEPTR &root)
{ int khoa, noidung;
  char so[10];
  NODEPTR p;
  do
  { printf("Nhap khoa :");    gets(so) ;
    khoa = atoi(so);
    if (khoa !=0)
    { printf("Nhap noi dung :");
      gets(so) ;
      noidung = atoi(so);
      if (root==NULL)
      {     p = New_Node();
            p->key = khoa;      p->info = noidung;   p->bf   = 0 ;
            p->left = NULL;   p->right = NULL;
            root =p;
       }
      else Insert(root,khoa,noidung);
      }
    } while (khoa!=0);              // khóa =0 thì dừng nhập
}
```

<u>Ghi chú</u>: Để tạo cây nhị phân do biến tree quản lý, ta gọi:

Create_AVLTree(tree);

III.2.2. *<u>Cập nhật cây</u>*:

1. <u>Tìm kiếm (Search)</u>: Tìm nút có khóa bằng x trên cây nhị phân AVL có gốc là root. Nếu tìm thấy x trong cây thì trả về địa chỉ của nút có trị bằng x trongcây, nếu không có thì trả về trị NULL.

Do AVL là cây nhị phân BST nên ta có thể tìm kiếm nhanh bằng phương pháp tìm kiếm nhị phân, và do tính chất lúc này cây cân bằng nên thời gian tìm kiếm sẽ nhanh hơn rất nhiều.

```
NODEPTR search(NODEPTR root, int x)
{
  NODEPTR p;
  p = root;
  while(p != NULL && x!=p->key)
   if(x < p->key)
      p = p->left;
    else
      p = p->right;
return(p);
}
```

2. <u>Xóa nút : Remove (root, x)</u>:

- <u>Nội dung</u>: xóa nút có khóa x trên cây AVL với địa chỉ sao đầu root sao cho sau khi xóa thì cây vẫn là AVL.

- <u>Giải thuật</u>:

Nếu root == NULL thì Thôngbáo ("Không thể xóa được nút x trên cây")
Nếu root != NULL thì
  Nếu x< root->info thì:
   + Gọi đệ qui để xóa nút x ở nhánh bên trái của root :
     Remove(root->left, x)
   + Gọi balance_left để cân bằng lại cây nút gốc root nếu nhánh cây con bên trái bị giảm độ cao.
     Nếu x > root->info thì:
   + Gọi đệ qui để xóa nút x ở nhánh bên phải của root :
     Remove(root->right, x)
   + Gọi balance_right để cân bằng lại cây nút gốc root nếu nhánh cây con bên phải bị giảm độ cao.
   Nếu x==root->info thì:
   Xóa nút root như phép toán xóa trên cây nhị phân BST.

- <u>Chương trình</u> : tự cài đặt.

III.2.3. ***<u>Các phép duyệt cây</u>*:**

Do cây AVL cũng là cây nhị phân nên ta sẽ áp dụng lại các phương pháp duyệt Preorder, Inorder và Postorder vào cây AVL.

a. <u>Duyệt cây theo thứ tự NLR (Preorder):</u>

```
void Preorder (NODEPTR root)
{
```

```
  if (root != NULL)
  {
    printf("%d ", root->info);
    Preorder(root->left);
    Preorder (root->right);
  }
}
```

b. Duyệt cây theo thứ tự LNR (Inorder):

```
void Inorder(NODEPTR root)
{
  if(root != NULL)
{
    Inorder(root->left);
    printf("%d ", root->info);
    Inorder(root->right);
  }
}
```

c. *Duyệt cây theo thứ tự LRN (Posorder):*

```
void Posorder(NODEPTR root)
{
  if(root != NULL)
  {
    Posorder(root->left);
    Posorder(root->right);
    printf("%d ", root->info);
  }
}
```

BÀi tập:

1. Viết chương trình tạo một menu thực hiện các mục sau:
   a. Tạo cây nhị phân tìm kiếm với nội dung là số nguyên (không trùng nhau).
   b. Liệt kê cây nhị phân ra màn hình theo thứ tự NLR
   c. Đếm tổng số nút, số nút lá, và số nút trung gian của cây.
   d. Tính độ cao của cây.
   e. Loại bỏ nút có nội dung là x trong cây nhị phân BST.
   f. Thêm nút có nội dung x vào cây nhị phân BST sao cho sau khi thêm thì cây vẫn là BST.
   g. Vẽ cây nhị phân ra màn hình.
2. Cho một cây nhị phân tree, hãy viết chương trình để sao chép nó thành một cây mới tree2, với khóa, nội dung, và liên kết giống như cây tree.
3. Viết các hàm kiểm tra xem cây nhị phân:
   a. Có phải là cây nhị phân đúng không.
   b. Có phải là cây nhị phân đầy không.
4. Viết hàm kiểm tra nút x và y có trên cây hay không, nếu có cả x lẫn y trên cây thì xác định nút gốc của cây con nhỏ nhất có chứa x và y.
5. Hãy trình bày cách chuyển một biểu thức số học sang cây biểu thức. Vẽ cây biểu thức của biểu thức số học sau:

   (10 + 5) ^ 2 / (52 * 4 - 3)
6. Cho một cây biểu thức, hãy viết hàm Calculate (NODEPTR root) để tính giá trị của cây biểu thức đó, biết rằng các toán tử được dùng trong biểu thức là:

   \+ - * / ^ % !

CHƯƠNG VI

SẮP XẾP VÀ TÌM KIẾM

Trong thực tiễn cuộc sống cũng như trong lĩnh vực lập trình việc quản lý dữ liệu thường đòi hỏi sự tìm kiếm các dữ liệu cần thiết; để thuận tiện cho việc tìm kiếm, dữ liệu thường được sắp xếp theo một thứ tự nào đó.

Có rất nhiều phương pháp sắp thứ tự, trong giáo trình này chỉ khảo sát một số phương pháp có đặc điểm là :

- Phương pháp tốt nhất trong một số trường hợp cụ thể.

- Mang được nét tiêu biểu cho tất cả các phương pháp khác.

- Giải thuật dễ hiểu và dễ viết.

Có thể phân các phương pháp sắp thứ tự chính như sau :

- Sắp thứ tự nội (Internal Sorting) : Toàn bộ dữ liệu sẽ được đưa vào bộ nhớ trong để thực hiện việc sắp xếp. Do toàn bộ dữ liệu được đưa vào bộ nhớ trong nên kích thước dữ liệu cần sắp không lớn, tuy nhiên ưu điểm của phương pháp này là giải thuật dễ hiểu và thời gian sắp xếp nhanh.

- Sắp thứ tự ngoại (External Sorting) : Dữ liệu nhiều phải chứa trong tập tin hoặc dĩa, mỗi lần sắp thứ tự ta chỉ đưa một phần của dữ liệu vào bộ nhớ để thực hiện. Do đó, thời gian sắp xếp chậm.

<u>Trong chương này ta chỉ phân tích các phương pháp sắp thứ tự nội trên danh sách kề</u>. Khi sắp xếp ta phải chú ý đến một số khái niệm sau:

- Khóa (Key) : Là dữ liệu của mỗi phần tử mà ta căn cứ vào nó để sắp thứ tự; khóa này có thể là chữ hoặc số.

- Thời gian thực hiện : Các giải thuật sắp xếp nội thường được thực hiện bằng cách so sánh và đổi chỗ hai phần tử với nhau. Do đó thời gian thực hiện của 1 giải thuật sắp xếp phụ thuộc vào hai yếu tố :

+ Số lần đổi chỗ các phần tử (đối với danh sách đặc), hoặc số lần thay đổi con trỏ (đối với danh sách liên kết). Đây là yếu tố chiếm nhiều thời gian nhất. Ký hiệu : M(n)

+ Số lần so sánh khóa. Ký hiệu : C(n), với n là số phần tử trong danh sách.

I. <u>MỘT SỐ PHƯƠNG PHÁP SẮP XẾP ĐƠN GIẢN</u> :

I.1. <u>*Sắp xếp theo phương pháp Bubble_Sort*</u> (phương pháp nổi bọt)

- <u>Nội dung</u> : Cho dãy A có n phần tử. Ta cho i duyệt dãy a[0],...,a[n-1]; nếu a[i-1] lớn hơn a[i] thì ta hoán đổi (a[i-1],a[i]). Lặp lại quá trình duyệt dãy này cho đến khi không có xảy ra việc đổi chỗ của hai phần tử.

Ví dụ: Ta sắp thứ tự dãy số sau : 26 33 35 29 19 12 32

Bước 0	**1**	**2**	**3**	**4**	**5**	**6**
26	12	12	12	12	12	12
33	26	19	19	19	19	19
35	33	26	26	26	26	26
29	35	33	29	29	29	29
19	29	35	33	32	32	32
12	19	29	35	33	33	33
32	32	32	32	35	35	35

Hình 8.1: Minh họa quá trình sắp xếp qua phương pháp Buble Sort

- Chương trình:

```
void Bubble_Sort(int A[], int n)
{ int i,j,temp;
  for (i=1; i<n; i++)
    for (j=n-1;j>=i; j--)
      if (A[j-1] > A[j])
       {      temp = A[j-1];
              A[j-1] = A[j];
              A[j] = temp;
       }
}
```

Ta nhận thấy phương pháp này có thể được cải tiến dễ dàng. Nếu ở lần duyệt dãy nào đó mà không có có sự đổi chỗ giữa hai phần tử thì dãy đã có thứ tự và giải thuật kết thúc. Trong trường hợp này, ta dùng một cờ hiệu flag để ghi nhận điều này, và giải thuật Bubble Sort được cải tiến như sau:

```
#define FALSE 0
#define TRUE  1
void Bubble_Sort_Ad(int A[], int n)
{ int i,temp;
  unsigned char flag=TRUE;
  while (flag)
  { flag = FALSE ;
    for (i=0; i<n-1; i++)
      if (A[i] > A[i+1])
        {   temp = A[i];
            A[i] = A[i+1];
            A[i+1] = temp;
```

```
                flag=TRUE;
            }
    }
}
```

* Phân tích: Giả sử dãy có n phần tử.

- Phân tích thời gian thực hiện của giải thuật Bubble Sort chưa cải tiến :

+ Số lần so sánh C(n):

Vòng lặp đầu có n-1 lần duyệt

Với mỗi i, ta lại có lần duyệt phụ thuộc vào i, cụ thể như sau:

i	Số lần duyệt
1	n-1
2	n-2
3	n-3
...	....
n-2	2

⇨ C(n) = 1 + 2 + 3 + n-2 + n-1-1 = (n-1)*n/2 -1

Bậc của C(n) là $O(n^2)$

+ Số lần đổi chỗ M(n): ta không thể xác định chính xác số lần đổi chỗ 2 phần tử trong dãy vì nó phụ thuộc vào trật tự hiện có trong dãy. Tuy nhiên, ta biết chắc rằng số lần đổi chỗ tối đa sẽ bằng C(n).

Do đó, $M(n) \le C(n)$

Bậc của M(n) là $O(n^2)$

Vậy thời gian thực hiện của giải thuật Bubble Sort chưa cải tiến có bậc $O(n^2)$.

- Phân tích thời gian thực hiện của giải thuật Bubble Sort cải tiến :

+ Số lần so sánh C(n):

Vòng lặp while có k lần duyệt (k<= n-1)

Có n-i lần so sánh trong lần duyệt thứ i

⇨ C(n) = (n-1) + (n-2) + ... + (n-k) = (n-k)*(n-k+1)/2

Bậc của C(n) là $O(n^2)$

+ Số lần đổi chỗ M(n): Tương tự như trên, chúng ta có $M(n) \le C(n)$

Bậc của M(n) là $O(n^2)$

Vậy thời gian thực hiện của giải thuật Bubble Sort cải tiến cũng có bậc $O(n^2)$.

* **Nhận xét**: Giải thuật Bubble Sort dễ hiểu nhưng không tối ưu vì thời gian thực hiện giải thuật chậm (có bậc $O(n^2)$)

Giải thuật Bubble Sort có cải tiến không làm thay đổi bậc nhưng có làm giảm hệ số nên chỉ nhanh hơn không đáng kể.

I.2. <u>Insertion Sort (Phương pháp xen vào)</u>

a. <u>Nội dung</u> :

Xét một danh sách có n phần tử a_0, a_1, a_2,..........,a_{n-1}, nội dung tổng quát của phương pháp xen vào là : nếu trong danh sách đã có i-1 phần tử trước đã có thứ tự, tiếp tục so sánh phần tử a_i với i-1 phần tử này để tìm vị trí thích hợp cho a_i xen vào. Vì danh sách có một phần tử thì tự nó đã có thứ tự, do đó ta chỉ cần so sánh từ phần tử thứ 2 cho đến phần tử thứ n.

<u>Ví dụ</u> : Cho mãng A có 5 phần tử :

a_0	a_1	a_2	a_3	a_4
44	55	12	42	94

i=	a_0	a_1	a_2	a_3	a_4
1	44	<u>55</u>	12	42	94
2	44	55	<u>12</u>	42	94
3	12	44	55	<u>42</u>	94
4	12	42	44	55	<u>94</u>

Hình 8.2: Minh họa quá trình sắp xếp qua phương pháp Insertion Sort

b. <u>Giải thuật</u> :

```
void Insertion_sort(int A[], int n)
{
  int x;
  int i,j;
  for (i=1; i< n; i++)
  {
    x = A[i];
    for (j=i-1; j>=0 && x<A[j]; j--)
       A[j+1] = A[j];
    A[j+1]=x;
  }
}
```

c. <u>Phân tích</u>

- Số lần so sánh C(n):

+ Trường hợp tốt nhất : Khi danh sách đã có thứ tự ban đầu, mỗi phần tử chỉ cần so sánh một lần với phần tử đứng trước nó

C_{min} = n-1

+ Trường hợp xấu nhất: Khi danh sách có thứ tự ngược, một phần tử A[i] sẽ phải so sánh i-1 lần

$$C_{max} = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \approx \frac{n^2}{2}$$

+ Trường hợp trung bình : Mỗi phần tử A[i] sẽ có số lần so sánh trung bình là $\frac{i}{2}$ lần

$$C_{average} = \frac{1+2+3+...+(n-1)}{2} = \frac{n(n-1)}{4} \approx \frac{n^2}{4}$$

Bậc của C(n) là O(n^2)

- Số lần đổi chỗ M(n)

Gán x = A[i] ;

+ Trường hợp tốt nhất : Sau lần so sánh duy nhất ta lại gán x trở lại cho A[i]; do đó với n phần tử sẽ có n-1 lần gán này, nên :

$M_{min} = 2*(n-1)$

+ Trường hợp xấu nhất : Mỗi bước sẽ có i-1 lần so sánh tương ứng với i-1 lần đổi chỗ, cộng thêm 2 lần để gán trị như trên; do đó ta có :

$M_{max} = C_{max} + 2*(n-1)$

$$= \frac{n(n-1)}{2} + 2*(n-1) \approx \frac{n^2}{2} + 2*(n-1)$$

+ Trường hợp trung bình : Ta cũng có số lần đổi chỗ trung bình là :

$$M_{average} = C_{average} + 2*(n-1) \approx \frac{n^2}{4} + 2*(n-1)$$

Bậc của M(n) cũng là O(n^2)

I.3. Selection Sort (Phương pháp lựa chọn) :

a. Nội dung : Xét một danh sách có n phần tử $a_0, a_1, a_2, ...,a_{n-1}$; để sắp thứ tự một danh sách, ta so sánh tất cả các phần tử của danh sách để chọn ra một phần tử nhỏ nhất đưa về đầu danh sách; sau đó tiếp tục chọn phần tử nhỏ nhất trong các phần tử còn lại để tạo thành phần tử thứ 2 trong danh sách. Quá trình này được lặp đi lặp lại cho đến khi chọn ra được phần tử nhỏ thứ (n-1)

b. Giải thuật (đối với danh sách đặc)

```
void Selection_Sort(int A[], int n)
{
 int min, vitrimin;
 int i,j;
 for (i=0; i< n-1; i++)
 {
  min = A[i];
  vitrimin=i;
```

```
      for (j=i+1; j<n; j++)
       if (A[j] < min)
       { min = A[j];
         vitrimin=j;
       }
      // Doi cho 2 phan tu A[i] va A[vitrimin]
      // min = A[vitrimin]
      A[vitrimin] = A[i];
      A[i] = min;
    }
  }
```

c. <u>Phân tích</u> :

- Số lần so sánh C(n) : không phụ thuộc vào thứ tự ban đầu của danh sách, mà phụ thuộc vào số lần thực hiện của hai vòng For lồng nhau.

Vòng For ngoài sẽ lặp n-1 lần, với mỗi lần lặp thì nó sẽ thực hiện vòng For trong. Số lần so sánh của vòng For trong tùy thuộc vào chỉ số i, tức là vị trí đang xét :

i= 0 → so sánh (n-1) lần (j= 1 ÷ n-1)

i= 1 → so sánh (n-2) lần (j= 2 ÷ n-1)

....

i= n-2 → so sánh 1 lần (j:= n-1 ÷ n-1)

Suy ra số lần so sánh là:

$$C = (n-1) + (n-2) + (n-3) + ... + 1$$

$$= \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

$\Rightarrow$ C(n) $\approx \frac{n^2}{2}$ số lần đổi chỗ

Bậc của C(n) cũng là O(n^2)

Như vậy số lần so sánh ở phương pháp chọn lựa luôn luôn tương đương với số lần so sánh trong trường hợp xấu nhất của phương pháp xen vào

- Số lần đổi chỗ M(n) : Tùy thuộc thứ tự ban đầu của danh sách, nó sẽ nhỏ nhất khi các khóa ban đầu đã có thứ tự và lớn nhất khi khóa có thứ tự ngược.

II. <u>Quick_Sort</u> : (Sắp xếp theo phương pháp phân đoạn)

1. *<u>Nội dung</u>*: Chọn một phần tử bất kỳ trong danh sách làm điểm chốt x, so sánh và đổi chỗ những phần tử trong danh sách này để tạo ra 3 phần: phần có giá trị nhỏ hơn x, phần có giá trị bằng x, và phần có giá trị lớn hơn x. Lại tiếp tục chia 2 phần có giá trị nhỏ hơn và lớn hơn x theo nguyên tắc như trên; quá trình chia phần sẽ kết thúc khi mỗi phần chỉ còn lại một phần tử, lúc này ta đã có một danh sách có thứ tự.

Ví dụ: Xét dãy 26 33 35 29 19 12 32

• Lần chia phần thứ nhất : Chọn phần tử chốt có khóa là 29, đặt là x

26	33	35	29	19	12	32
i →						← j

Dùng hai biến chỉ số i và j để duyệt từ hai đầu danh sách đến x. Nếu i gặp phần tử lớn hơn hay bằng x sẽ dừng lại, j gặp phần tử nhỏ hơn hay bằng x sẽ dừng lại, rồi đổi chỗ hai phần tử này; sau đó tiếp tục duyệt cho đến khi i>j thì ngừng lại.

Lúc này dãy sẽ có 3 phần khác nhau như hình vẽ sau :

26	33	35	29	19	12	32
	i				**j**	
26	12	35	29	19	33	32
		i		**j**		
26	12	19	29	35	33	32
			ij			
26	12	19	29	35	33	32
q		**j**		**I**		**r**

• Lần chia phần thứ hai cho dãy con 26 12 19, chọn chốt x=12

26	12	19	→	12	26	19
i	**j**			**j**	**i**	

Kết thúc ta sẽ có hai phần : 12 ; 26 19

• Lần chia phần thứ 3 cho dãy con 26 19, chọn chốt x=26

26	19	→	19	26	Kết thúc quá trình chia nhỏ dãy con 26 12 19
i	**j**		**j**	**i**	

- Lần chia phần thứ 4 cho dãy con 35 33 32, chọn chốt x= 33

35	33	32	→	32	33	35	→	32	33	35
i		**j**			**ij**			**j**		**i**

Kết thúc ta sẽ có ba phần : 32 ; 33 ; 35

Đến đây quá trình chia phần kết thúc vì tất cả các phần chỉ có một phần tử, lúc này ta sẽ có một danh sách có thứ tự là :

12 19 26 29 32 33 35

2. Giải thuật:

a. Giải thuật không đệ quy:

- Ta tạo một Stack, mỗi phần tử của Stack có 2 thành phần là q, r chứa chỉ số đầu và chỉ số cuối của dãy cần sắp. Ban đầu, Stack[0].q = 0 và Stack[0].r =n-1

- Tiến hành phân hoạch dãy số gồm các số bắt đầu từ chỉ số q đến chỉ số r

- Sau mỗi lần chia phần, ta kiểm tra xem phần có giá trị nhỏ hơn chốt và phần có giá trị lớn hơn chốt nếu có từ 2 phần tử trở lên thì đưa vào Stack. Sau mỗi lần phân hoạch, ta lại lấy dãy số mới từ Stack ra phân hoạch tiếp.

- Quá trình cứ như thế cho tới khi Stack rỗng thì kết thúc.

* Chương trình:

```
void Quick_Sort(int A[], int n)
{ struct Element_Stack       // kiểu phần tử trong Stack
  {
     int q, r;
  } ;
  Element_Stack Stack[50]; // Stack có tối đa 50 phần tử
  int sp=0;                // con trỏ Stack, khởi tạo sp=0
  int i,j,x,q,r,temp;
  Stack[0].q =0 ;          // chỉ số đầu của mảng cần sắp
  Stack[0].r =n-1;         // chỉ số cuối của mảng cần sắp
  do
  { // Lấy một phân hoạch ra từ Stack
   q = Stack[sp].q ; r =Stack[sp].r ;
   sp--;                   // Xóa 1 phần tử khỏi Stack
   do
   {  // Phân đoạn dãy con a[q],..., a[r]
      x = A[(q+r) / 2] ;  // Lấy phần tử giữa của dãy cần sắp thứ tự làm chốt
    i = q; j =r;
    do
    {  while (A[i] < x)  i++; //Tìm phần tử đầu tiên có trị lớn hơn hay bằng x
       while (A[j] > x) j--; //Tìm phần tử đầu tiên có trị nhỏ hơn hay bằng x
       if (i<=j)          // Đổi chỗ A[i] với A[j]
       { temp = A[i];
         A[i] =A[j];
         A[j] = temp;
         i++ ;  j--;
       }
    } while (i<=j);
    if (i<r)              // phần thứ ba có từ 2 phần tử trở lên
    { // Đưa vào Stack chỉ số đầu và chỉ số cuối của phần thứ ba
         sp++;
         Stack[sp].q=i;
         Stack[sp].r=r;
    }
```

```
        r = j ;    // Chuẩn bị vị trí để phân hoạch phần có giá trị nhỏ hơn chốt
      } while (q< r);
    } while (sp!=-1);  // Ket thuc khi Stack rong
  }
```

b. Giải thuật Quick Sort đệ qui: về cơ chế thực hiện thì cũng giống như giải thuật không đệ qui, nhưng ta không kiểm soát Stack mà để cho quá trình gọi đệ qui tự tạo ra Stack.

* Chương trình:

```
void Sort(int A[], int q,int r)
  { int temp;
    int i=q;
    int j=r;
    int x = A[(q+r) / 2] ; //Lấy phần tử giữa của dãy cần sắp thứ tự làm chốt
    do
    {  // Phân đoạn dãy con a[q],..., a[r]
      while (A[i] < x)  i++; //Tìm phần tử đầu tiên có trị lớn hơn hay bằng x
      while (A[j] > x)  j--; //Tìm phần tử đầu tiên có trị nhỏ hơn hay bằng x
      if (i<=j)   // Doi cho A[i] voi A[j]
     { temp = A[i];
       A[i] =A[j];
       A[j] = temp;
           i++ ;  j--;
    }
  } while (i<=j);
   if (q<j)                  // phần thứ nhất có từ 2 phần tử trở lên
        Sort(A,q,j);
   if (i<r)                  // phần thứ ba có từ 2 phần tử trở lên
        Sort (A,i,r);
  }

void Quick_Sort(int A[], int n)
{ Sort( A,0,n-1);          // Gọi hàm Sort với phần tử đầu có chỉ số 0 đến
                           // phần tử cuối cùng có chỉ số n-1
  }
```

3. Phân tích :

Ta nhận thấy nhờ chia thành những dãy con trong mỗi lần chia phần sẽ có khả năng giảm được số lần so sánh. Tuy nhiên còn tùy thuộc vào việc chọn phần tử chốt và thứ tự ban đầu của dãy.

Khi chọn vị trí phần tử chốt là ở giữa thì nếu chia ta sẽ được các dãy con có số phần tử gần bằng nhau và do đó sẽ giảm được số lần so sánh.

- Trường hợp tốt nhất:

Để dễ tính C(n) và M(n), ta giả sử số nút của danh sách cần sắp xếp là $n = 2^m$

Với 1 danh sách ban đầu : n lần so sánh (trong cả 2 hướng quét)

Với 2 danh sách con tiếp theo : nhỏ hơn hoặc bằng n/2 lần so sánh trong mỗi danh sách con

Với 4 danh sách con tiếp theo : nhỏ hơn hoặc bằng n/4 lần so sánh trong mỗi danh sách con

.....

Với n danh sách con cuối cùng: nhỏ hơn hoặc bằng n/n lần so sánh trong mỗi danh sách.

+ Tổng số lần so sánh C(n):

$C(n) \le (1*n) + (2* n/2) + (4* n/4) + ... + (n* n/n)$

$\le n + n + n + ... + n$

Vì ta sẽ có nhiều nhất là m lần chia nên:

$C(n) \le m * n = n \lg n$

Vậy bậc của C(n) là O(nlgn)

+ Tổng số lần đổi chổ M(n):

Số lần đổi chổ phải ít hơn hoặc bằng số lần so sánh:

$M(n) \le C(n)$

Bậc của M(n) cũng là O(nlgn)

- Trường hợp xấu nhất: Với giải thuật Quick Sort, trường hợp xấu nhất khi nút làm chốt lúc nào cũng ở đầu hay cuối danh sách sau mỗi lần chia phần. Hai danh sách con suy biến thành một danh sách con có số nút bớt đi 1. Trong trường hợp này:

Với danh sách ban đầu : n lần so sánh

Với danh sách con tiếp theo : n-1 lần so sánh

Với danh sách con tiếp theo : n-2 lần so sánh

....

Với danh sách con cuối cùng : 1 lần so sánh

+ Tổng số lần so sánh C(n):

$C(n) = n + (n-1) + (n-2) + ... + 1 = n(n+1) /2$

Bậc của C(n) là $O(n^2)$

+ Tổng số lần đổi chổ M(n):

Số lần đổi chổ phải ít hơn hoặc bằng số lần so sánh:

M(n) ≤ C(n)

Bậc của M(n) cũng là $O(n^2)$

- Trường hợp trung bình : Trường hợp trung bình thì C(n) và M(n) có bậc ở khoảng giữa O(nlgn) và $O(n^2)$

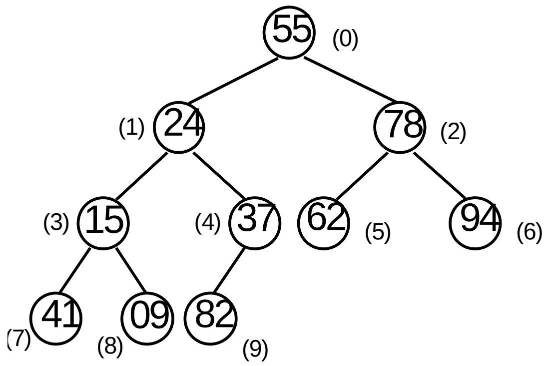
III. Heap Sort (Sắp xếp kiểu vun đống)

III.1. Định nghĩa Heap: Heap là cây nhị phân gần đầy được cài đặt bằng mảng một chiều với các nút trên Heap có nội dung lớn hơn hay bằng nội dung của các nút con của nó.

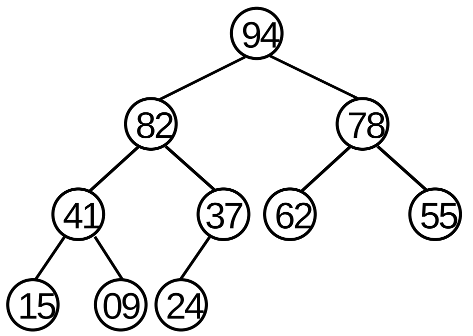
Ví dụ: Ta có dãy số sau:

55	24	78	15	37	62	94	41	09	82
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)

thì cây nhị phân ban đầu của dãy là:



và heap của dãy sau khi vun đống sẽ là:



III.2. Thuật toán Heap Sort:

a. Nội dung:

- Trước tiên ta cài đặt hàm Adjust (A, r,n) để điều chỉnh cây có gốc ở vị trí r với n nút thành đống với giả thiết hai cây con đã là đống.

Lưu ý: Do giả thiết là ta đã có hai cây con là đống nên ta phải vun đống từ dưới lên. Trong cây chỉ có các nút với vị trí 0,1,..., n/2-1 mới là các nút gốc có cây con. Do đó, ta chỉ cần vun đống các cây ở vị trí 0,1,..., n/2-1.

- Do sau khi vun đống thì nút ở vị trí 0 sẽ có nội dung lớn nhất nên ta sẽ hoán đổi nội dung của A_0 với A_{n-1}.

- Ta tiếp tục Adjust(A,0, n-1) để vun đống cây với gốc ở nút có vị trí 0 và số nút là n-1 nút còn lại (A_0, A_1,..., A_{n-2}). Sau đó, sẽ hoán đổi nội dung của A_0 với A_{n-2}.

- Quá trình trên tiếp tục cho đến khi cây được vun đống chỉ còn 1 nút thì dừng lại. Cuối cùng, ta đã có dãy đã được sắp theo thứ tự tăng dần.

* **Thuật toán Adjust** (A,r,n)

- Ta so sánh nội dung của 2 nút con tìm ra nội dung của nút con lớn hơn

- So sánh tiếp nội dung của nút con lớn hơn với nội dung của nút gốc :

+ Nếu nội dung của nút con đó mà nhỏ hơn nội dung của nút gốc thì cây con đó là đống.

+ Nếu nội dung của nút con đó mà lớn hơn nội dung của nút gốc thì di chuyển nội dung của nút con lên vị trí của nút gốc r. Sau đó, điều chỉnh tiếp cây có gốc ở vị trí nút là nút con lớn hơn đó.

b. Cài đặt:

```
void Adjust(int A[], int r, int n)
{
int j=2*r+1;  // vi tri nut con ben trai
  int x=A[r];
  int cont=TRUE;
  while (j<=n-1 && cont)
  {
    if (j<n-1)   // luc nay r moi co nut con ben phai. Neu j=n-1 thi r khong
                 // co nut con ben phai thi khong can so sanh
      if (A[j] < A[j+1])  // tim vi tri nut con lon hon
        j++;
    if (A[j] <=x)
      cont=FALSE;
    else
    {  // di chuyen nut con j len r
       A[r]= A[j];
       r=j ;      // xem lai nut con co phai la dong khong
       j=2*r+1;
    }
  }
  A[r]=x;
}
```

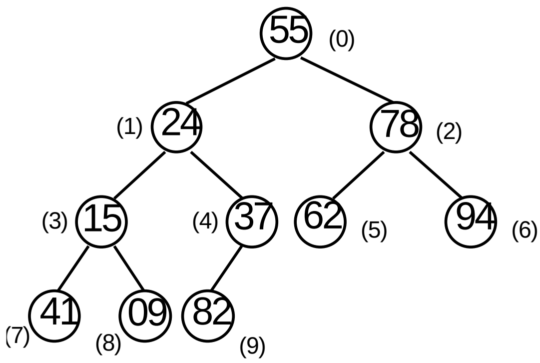
```
void Heap_sort(int A[], int n)
{
  int i, temp;
  for (i=n/2-1; i>=0; i--)  // Tao heap ban dau
    Adjust(A, i,n);

  for (i=n-2; i>=0; i--)
  {
     temp= A[0];          // Cho A[0] ve cuoi heap
     A[0] = A[i+1];
     A[i+1] = temp;
     Adjust(A,0,i+1);       // Dieu chinh lai heap tai vi tri 0
                          // Luc nay, 2 cay con o vi tri 1 va 2 da la heap
   }
}
```

c. <u>Ví dụ</u>: Dùng thuật toán heap sort để sắp xếp dãy số sau theo chiều tăng dần:

55	24	78	15	37	62	94	41	09	82
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)

n = 10;

☞ Cây nhị phân ban đầu của dãy là:

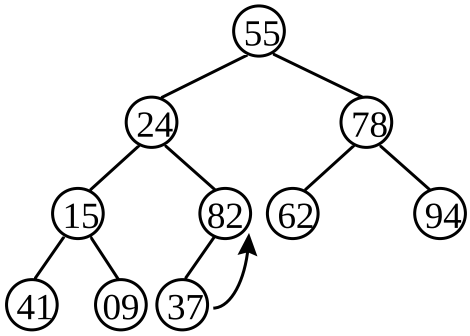


Vun cây ban đầu thành đống bằng giải thuật Adjust

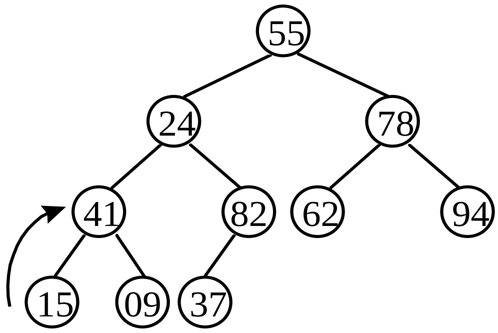
☞ Vun đống : Vun từ dưới lên, Trong cây chỉ có các nút 0, 1,2,... n/2-1=4 mới là các nút gốc

```
for (i=n/2-1; i>=0; i--)  // Tao heap ban dau
Adjust(A, i,n);
```

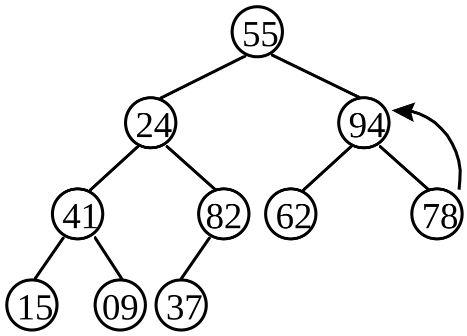
+ Vun cây có gốc là n/2 -1 = ④



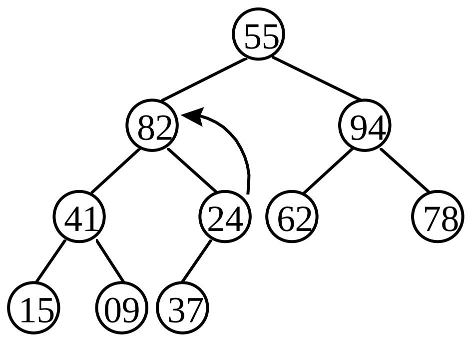
+ Vun cây có gốc là n/2 -2 = ③



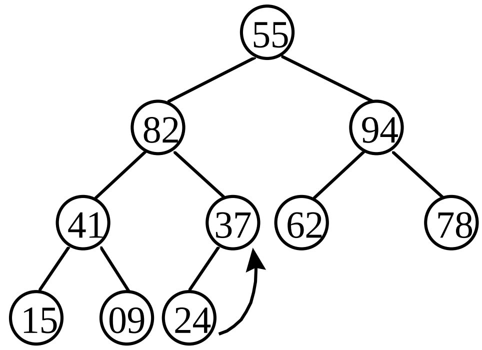
+ Vun cây có gốc là n/2 -2 = ②



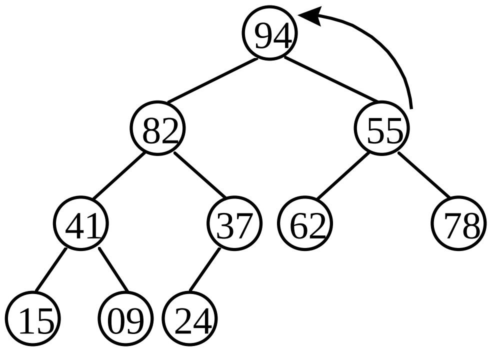
+ Vun cây có gốc là n/2 -2 = ①



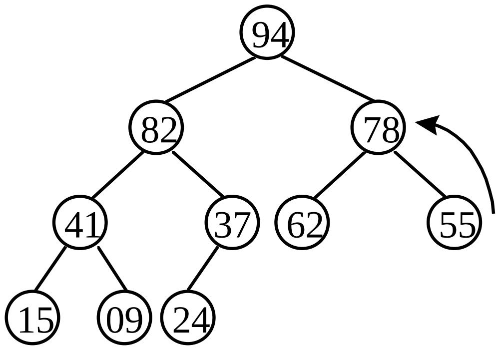
Do sau khi hoán đổi thì cây con không phải là đống nên ta Adjust lại:



+ Vun cây có gốc là n/2 -2 = 0



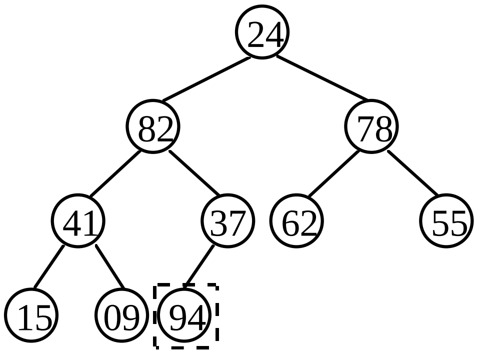
Do sau khi hoán đổi thì cây con không phải là đống nên ta Adjust lại:



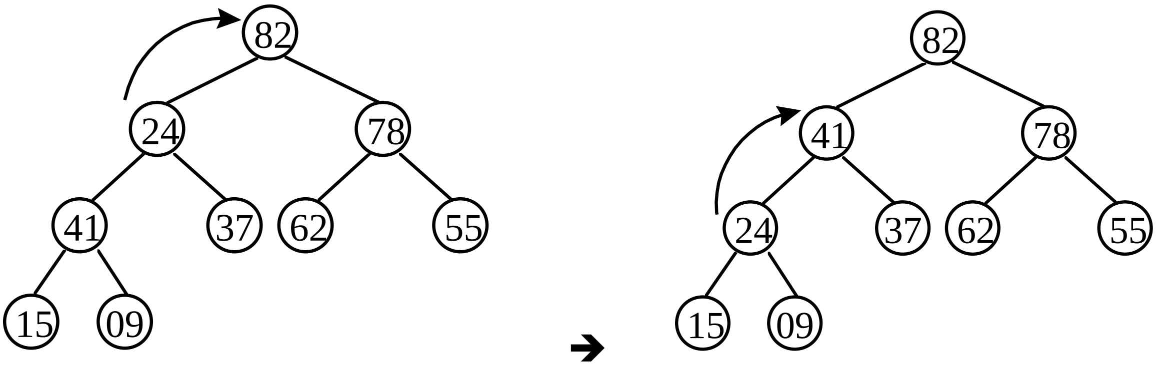
☞ Ta đổi chỗ A[0] với A[n-1] và sau đó Adjust lại cây ở vị trí 0 với số nút giảm đi 1, và quá trình trên sẽ dừng lại khi số nút bằng 0.

```
for (i=n-2; i>=0; i--)
  {
    temp= A[0];            // Cho A[0] ve cuoi heap
    A[0] = A[i+1];
    A[i+1] = temp;
    Adjust(A,0,i+1);       // Dieu chinh lai heap tai vi tri 0
                           // Luc nay, 2 cay con o vi tri 1 va 2 da la heap
  }
```

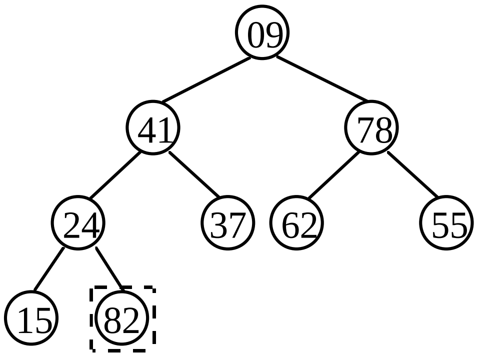
+ i = n-2 = 8: Đổi chỗ A[0] với A[n-1] thì dãy {A[n-1]} đã sắp xếp.



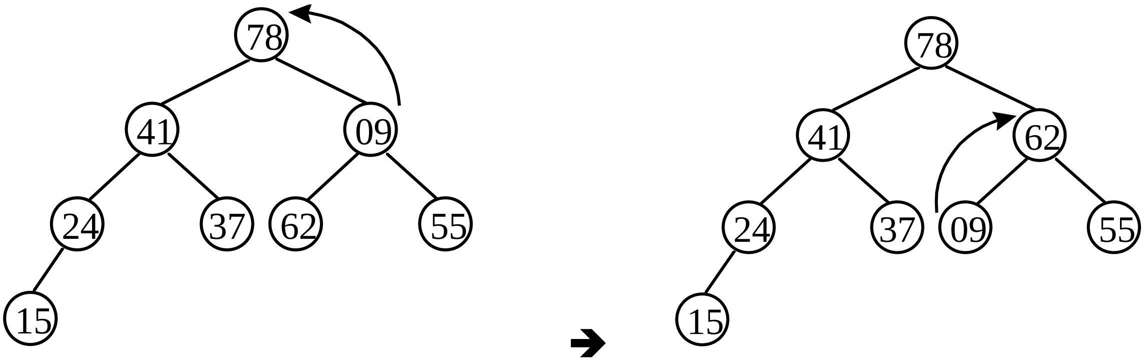
Sau đó, ta lại vun cây có gốc là 0 và số nút là n-1 =9 thành đống.



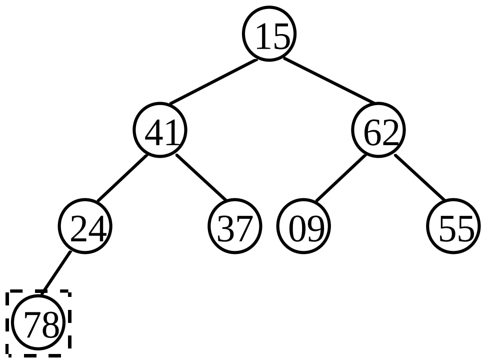
+ i = 7: Đổi chổ A[0] với A[n-2] thì dãy {A[n-2], A[n-1]} đã sắp xếp.



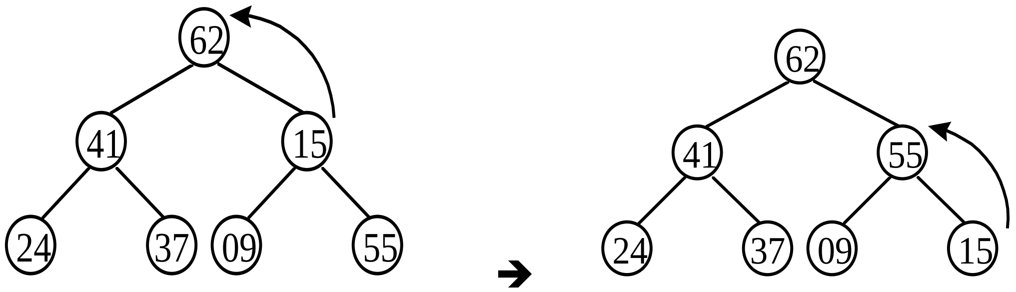
Sau đó, ta lại vun cây có gốc là 0 và số nút là n-2=8 thành đống.



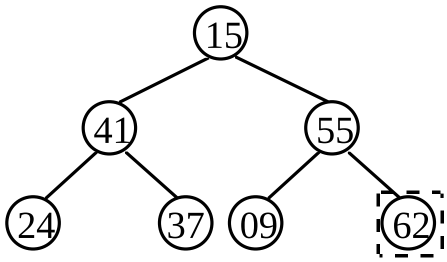
+ i = 6: Đổi chổ A[0] với A[n-3] thì dãy {A[n-3], A[n-2], A[n-1]} đã sắp xếp.



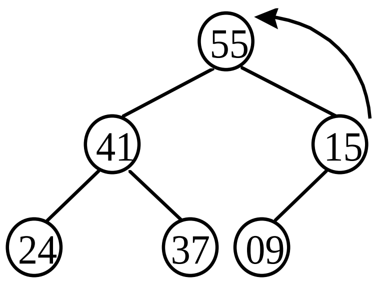
Sau đó, ta lại vun cây có gốc là 0 và số nút là n-3=7 thành đống.



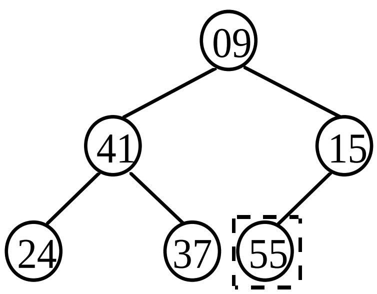
+ i = 5: Đổi chỗ A[0] với A[n-4] thì dãy {A[n-4], A[n-3], A[n-2], A[n-1]} đã sắp xếp.



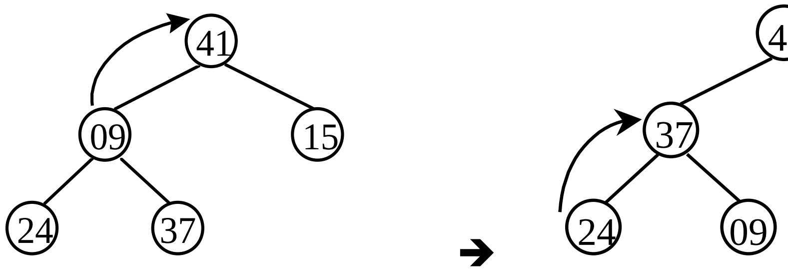
Sau đó, ta lại vun cây có gốc là 0 và số nút là n-4=6 thành đống.



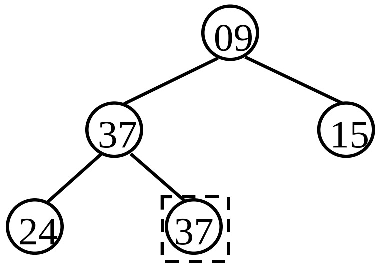
+ i = 4: Đổi chỗ A[0] với A[n-5] thì dãy {A[n-5],A[n-4], A[n-3], A[n-2], A[n-1]} đã sắp xếp.



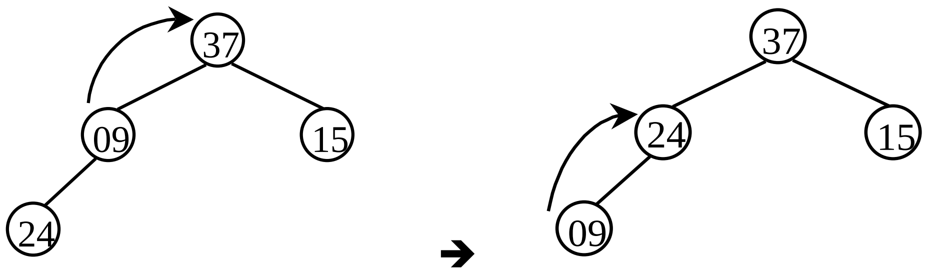
Sau đó, ta lại vun cây có gốc là 0 và số nút là n-5=5 thành đống.



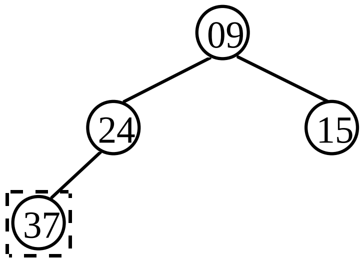
+ i = 3: Đổi chỗ A[0] với A[n-6] thì dãy {A[n-6], A[n-5],A[n-4], A[n-3], A[n-2], A[n-1]} đã sắp xếp.



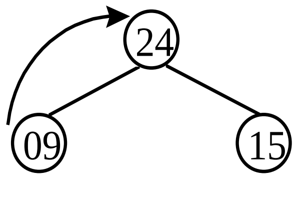
Sau đó, ta lại vun cây có gốc là 0 và số nút là n-6=4 thành đống.



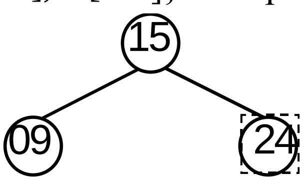
+ i = 2: Đổi chỗ A[0] với A[n-7] thì dãy {A[n-7], A[n-6], A[n-5],A[n-4], A[n-3], A[n-2], A[n-1]} đã sắp xếp.



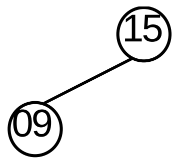
Sau đó, ta lại vun cây có gốc là 0 và số nút là n-7=3 thành đống.



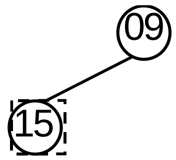
+ i = 1: Đổi chỗ A[0] với A[n-8] thì dãy {A[n-8], A[n-7], A[n-6], A[n-5],A[n-4], A[n-3], A[n-2], A[n-1]} đã sắp xếp.



Sau đó, ta lại vun cây có gốc là 0 và số nút là n-7=3 thành đống.



+ i = 0: Đổi chỗ A[0] với A[n-9] thì dãy {A[n-9], A[n-8], A[n-7], A[n-6], A[n-5],A[n-4], A[n-3], A[n-2], A[n-1]} đã sắp xếp.



Cuối cùng, dãy đã được sắp theo thứ tự tăng dần.

IV. <u>Merge Sort (Sắp xếp kiểu trộn)</u>

a. <u>Nội dung</u>: Merge Sort là phương pháp sắp xếp bằng cách trộn hai danh sách đã có thứ tự thành một danh sách có thứ tự. Phương pháp này gồm nhiều bước như sau:

a.1. Xem danh sách cần sắp xếp như n danh sách con đã có thứ tự, mỗi danh sách con có 1 phần tử.

Trộn từng cặp hai danh sách con kế cận, ta sẽ được n/2 danh sách con đã có thứ tự, mỗi danh sách con có 2 nút.

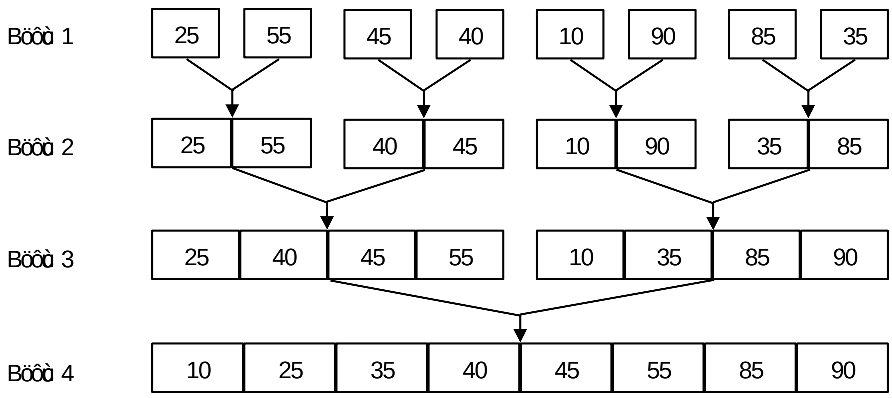
a.2. Ta tiếp tục xem danh sách cần sắp xếp như n/2 danh sách con đã có thứ tự, mỗi danh sách con có 2 phần tử.

Trộn từng cặp hai danh sách con kế cận, ta sẽ được n/4 danh sách con đã có thứ tự, mỗi danh sách con có 4 nút.

a.3. Quá trình trên cứ tiếp tục cho đến khi được 1 danh sách con có n phần tử.

<u>Ví dụ</u>: Ta dùng phương pháp Merge Sort để sắp xếp dãy số sau:

25 55 45 40 10 90 85 35



Hình 8.3: Các bước trộn của giải thuật Merge Sort.

b. <u>Giải thuật</u>:

<u>Các biến sử dụng</u>:

- A là dãy số cần sắp có n phần tử

- low1, up1, low2, up2 là cận dưới và cận trên của 2 danh sách con đang trộn

- size là kích thước của danh sách con, ở bước trộn 1 thì size=1, ở bước trộn 2 thì size=2, ở bước trộn 3 thì size=4, ở bước trộn 4 thì size=8,...

```
#define MAXLIST 100
int A[MAXLIST];
void mergesort(int A[], int n)
{
  int i, j, k, low1, up1, low2, up2, size;
  int dstam[MAXLIST];
  size = 1;
  while(size < n)
  {
    low1 = 0;
    k = 0;
    while(low1+size < n)
    {
      up1 = low1+ size-1;
      low2 = up1 + 1;

      up2 = (low2+size-1 < n) ? low2+size-1 : n-1;

      for(i = low1, j = low2; i <= up1 && j <= up2; k++)
              if(A[i] <= A[j])
                dstam[k] = A[i++];
              else
                dstam[k] = A[j++];
          for(; i <= up1; k++)
              dstam[k] = A[i++];
          for(; j <= up2; k++)
              dstam[k] = A[j++];
          low1 = up2+1;
    }
    for(i = low1; k < n; i++)
          dstam[k++] = A[i];
    for(i = 0; i < n; i++)  // gan nguoc tra lai cho A
          A[i] = dstam[i];
    size *= 2;
  }
}
```

c. <u>Phân tích</u>

Giải thuật Merge Sort không hiệu quả về mặt bộ nhớ vì có dùng thêm danh sách tạm trong quá trình sắp xếp.

- Số lần so sánh C(n) :

Giải thuật Merge Sort có $\log_2 n$ bước trộn, có ít hơn n lần so sánh trong từng bước

Suy ra $C(n) < n\log_2 n$

Bậc của C(n) là O(nlgn)

- Số lần đổi chỗ M(n) :

Giải thuật Merge Sort có $\log_2 n$ bước trộn, trong từng bước trộn có chép n nút từ danh sách A[] sanh dstam[] và chép n nút từ danh sách dstam[] về danh sách A[]

Vậy M(n) có bậc là O(nlgn)

V. TÌM KIẾM:

V.1. Khái niệm: Cho danh sách A có n phần tử. Tìm x trong danh sách A, nếu có thì trả về vị trí của phần tử đó trong danh sách, ngược lại nếu tìm không thấy thì trả về -1. Thông thường danh sách A chưa có thứ tự hoặc đã được sắp theo 1 trật tự nào đó.

V.2. Tìm kiếm tuần tự:

- Nội dung: Ta tìm từ đầu danh sách cho đến khi nào gặp phần tử đầu tiên có trị bằng với x hoặc đã tìm hết danh sách thì dừng lại. Giải thuật này được dùng trong danh sách chưa có thứ tự.

- Giải thuật:

```
int  Search(int A[], int n, int x)
{ int i=0;
  while (i<n && A[i] <> x)
     i++;
  return (i<n ? i : -1) ;
}
```

V.3. Tìm kiếm nhị phân: chỉ dùng được đối với danh sách đã có thứ tự. Ta giả sử danh sách có thứ tự tăng dần.

- Nội dung:

➴ Bước 1: Phạm vi tìm kiếm ban đầu là toàn bộ danh sách.

➴ Bước 2: Lấy phần tử chính giữa của phạm vi tìm kiếm (gọi là y) so sánh với x.

- Nếu x=y thì ta đã tìm thấy, trả về chỉ số. Giải thuật kết thúc
- Nếu x < y thì phạm vi tìm kiếm mới là các phần tử nằm phía trước của y.

- Nếu x > y thì phạm vi tìm kiếm mới là các phần tử nằm phía sau của y.

➢ Bước 3: Nếu còn tồn tại phạm vi tìm kiếm thì lặp lại bước 2, ngược lại giải thuật kết thúc với kết quả là không có x trong dãy số.

- Giải thuật:

```
int  Binary_Search(int A[], int n, int x)
{ // Phạm vi ban đầu tìm kiếm là từ left=0 →right =n-1
  int left=0;
  int right=n-1;
  int j;
  while (left <= right)
  { j=(left + right) /2;                //chỉ số phần tử giữa
    if (A[j]==x)  return j;
    if (x>A[j])  left=j+1;              // Phạm vi tìm mới là (j+1, right)
        else right=j-1;                 // Phạm vi tìm mới là (left, j-1)
  }
  return  -1 ;
}
```

V.4. Phép tìm kiếm nhị phân đệ qui:

- Nội dung: tương tự như trên

➢ Bước 1: Phạm vi tìm kiếm ban đầu là toàn bộ danh sách (left=0→right=n-1).

➢ Bước 2: Lấy phần tử chính giữa của phạm vi tìm kiếm (gọi là y) so sánh với x.

- Nếu x=y thì ta đã tìm thấy, trả về chỉ số. Giải thuật kết thúc
- Nếu x < y thì phạm vi tìm kiếm mới là các phần tử nằm phía trước của y, nên ta gọi đệ qui với phạm vi mới là (left, j-1)
- Nếu x > y thì phạm vi tìm kiếm mới là các phần tử nằm phía sau của y, nên ta gọi đệ qui với phạm vi mới là (j+1,right)

➢ Điều kiện dừng: x=y hoặc left > right.

- Giải thuật:

```
int Binary_Search2(int A[], int left,int right, int x)
{ int j=(left+right) /2;
  if (left > right)  return -1 ;
  else  if (A[j]==x)   return j ;
    else Binary_Search2(A, (A[j]<x ? j+1:left), (A[j] > x ?j-1:right),x);
}
```

BÀi tập :

1. Viết lại hàm QuickSort trong trường hợp chọn nút chốt là nút giữa của danh sách cần sắp.
2. Viết giải thuật tìm k nút lớn nhất trong danh sách có n nút, yêu cầu giải thuật có dùng cấu trúc heap.
3. Cài đặt giải thuật Seletion Sort trên danh sách liên kết.
4. Viết chương trình minh họa các phương pháp sắp xếp. Chương trình có các chức năng sau:
   a. Nhập ngẫu nhiên n số vào danh sách với n khá lớn
   b. Chọn phương pháp sắp xếp, có báo thời gian thực hiện quá trình sắp xếp: Bubble Sort, Insertion Sort, Selection Sort, Quick Sort, Heap Sort, Merge Sort.
   c. Xem danh sách
   d. Kết thúc chương trình

CHƯƠNG VII

Đồ thị (Graph)

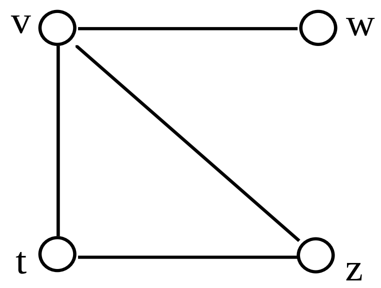
Một cấu trúc dữ liệu được áp dụng rất nhiều trong kỹ thuật lập trình là đồ thị. Cấu trúc dữ liệu đồ thị được sử dụng trong các bài toán của rất nhiều lĩnh vực như đường đi, sơ đồ mạng máy tính, sơ đồ đường xe lửa - đường xe điện ngầm trong thành phố....

Chương này sẽ mô tả các cách tổ chức và cấu trúc dữ liệu khác nhau cho 2 loại đồ thị: đồ thị vô hướng và đồ thị có hướng. Chúng ta sẽ nghiên cứu 2 cách cài đặt đồ thị như ma trận kề và danh sách kề, hai phương pháp duyệt đồ thị (theo chiều sâu và theo chiều rộng). Ngoài ra ta sẽ tham khảo một giải thuật tìm đường đi ngắn nhất (Shortest paths algorithm) trên một đồ thị có trọng số.

I. CẤU TRÚC DỮ LIỆU CHO ĐỒ THỊ:

I.1. Định nghĩa đồ thị :

- Một đồ thị G (Graph) bao gồm một tập V (vertices) chứa các nút của đồ thị và một tập E chứa các cặp nút (v,w) sao cho giữa v và w có một cạnh.



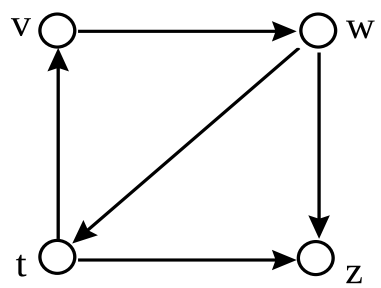
Hình 6.1 Đồ thị vô hướng

$$V = \{v, w, z, t\}$$

$$E = \{ (v,w); (v,t); (t,z); (v,z) \}$$

- Đồ thị vô hướng (Undirected graph): là đồ thị mà các cung không có chiều nhất định.

- Đồ thị hữu hướng (Directed graph): là đồ thị trong đó mỗi cung có một chiều nhất định.

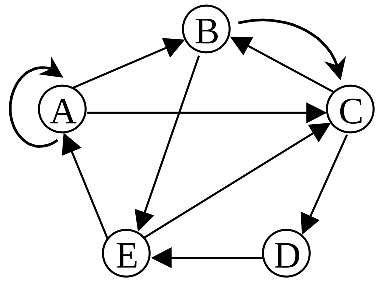


Hình 6.2 Đồ thị hữu hướng

$$G = \begin{cases} V = \{v, w, z, t\} \\ E = \{\langle v,w\rangle; \langle w,t\rangle; \langle w,z\rangle; \langle t,v\rangle; \langle t,z\rangle\} \end{cases}$$

I.2. <u>Các khái niệm trên đồ thị</u>:

Trong bài học, chúng ta sẽ dùng đồ thị hình 6.3 để minh họa các khái niệm trên đồ thị.



Hình 6.3 Đồ thị hữu hướng minh họa cho các khái niệm của đồ thị.

- Cạnh nối hai đỉnh a và b trên đồ thị vô hướng được ký hiệu (a,b)

- Cạnh nối hai đỉnh a và b trên đồ thị hữu hướng được ký hiệu <a,b>.

- Nút: Đồ thị hình 6.3 có 5 nút là A, B, C, D, E

- Cung : Mỗi cung trên đồ thị được xác định bởi 2 nút: nút đỉnh (nút trước của cung) và nút ngọn (nút sau của cung)

+ Khuyên: Cạnh (hay cung) nối từ 1 đỉnh đến chính đỉnh đó gọi là khuyên. Khuyên là chu trình có chiều dài là 1.

- Bậc của nút: số cung liên kết với nút

+ Bậc vào: số nút ngọn liên kết với nút

+ Bậc ra: số nút đỉnh liên kết với nút.

Bậc của nút = bậc vào + bậc ra

- Nút kề: Nút y được gọi là nút kề với nút x nếu có 1 cung đi từ nút x đến nút y.

- Đường đi: ta nói từ nút x đến nút y có 1 đường đi với chiều dài là k khi đi từ nút x đến nút y ta qua 1 chuỗi k-1 nút $x \rightarrow n_1 \rightarrow n_2 \rightarrow ... \rightarrow n_{k-1} \rightarrow y$ với nút n_i là nút kề với nút n_{i-1}.

<u>Ví dụ:</u> Đường đi từ nút A đến nút D qua các nút A, C, D có chiều dài là 2.

- Chu trình : Ta nói qua nút x có 1 chu trình chiều dài k nếu xuất phát từ nút x chúng ta qua k-1 nút trung gian và về lại nút x ($x \rightarrow n_1 \rightarrow n_2 \rightarrow ... \rightarrow n_{k-1} \rightarrow x$)

<u>Ví dụ:</u>

Khuyên A -> A (chiều dài chu trình này bằng 1)

Chu trình A -> C -> D -> E -> A có chiều dài bằng 4

- Đồ thị có trọng số: là đồ thị mà mỗi cung có liên kết với 1 trọng số; thông thường trọng số này sẽ có một ý nghĩa nào đó chẳng hạn như chiều dài của đoạn đường, chi phí vận chuyển trên một quãng đường, thời gian vận chuyển...

- Đồ thị liên thông: đồ thị được gọi là liên thông nếu với mọi cặp nút phân biệt bao giờ cũng có 1 đường đi từ nút này đến nút kia.

Ví dụ: Đồ thị hình 6.3 là đồ thị liên thông.

I.3 Tổ chức dữ liệu cho đồ thị :

Một đồ thị G bao gồm một tập các nút v, mỗi nút $v \in V$ sẽ có một tập A_v chứa các nút $w \in V$ sao cho có một cung từ $v \rightarrow w \in E$

w được gọi là nút kề của v.

Chúng ta có các phương pháp để cài đặt cấu trúc dữ liệu cho tập các nút của một đồ thị : ma trận kề và danh sách kề.

a. Ma trận kề (mảng 2 chiều):

- Trong trường hợp G là đồ thị vô hướng thì ta quy ước :

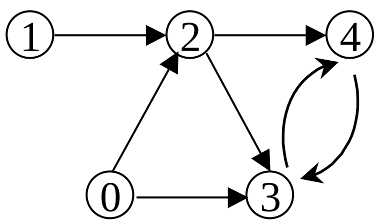
$G[v][w] = G[w][v] = 1$ nếu $(v,w) \in E$

- Trong trường hợp G là đồ thị hữu hướng thì ta quy ước :

$G[v][w] = 1$ nếu <v,w> $\in E$

Ví dụ : Ma trận kề của đồ thị hình 6.4 có dạng:

$$\begin{vmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{vmatrix}$$



Hình 6.4

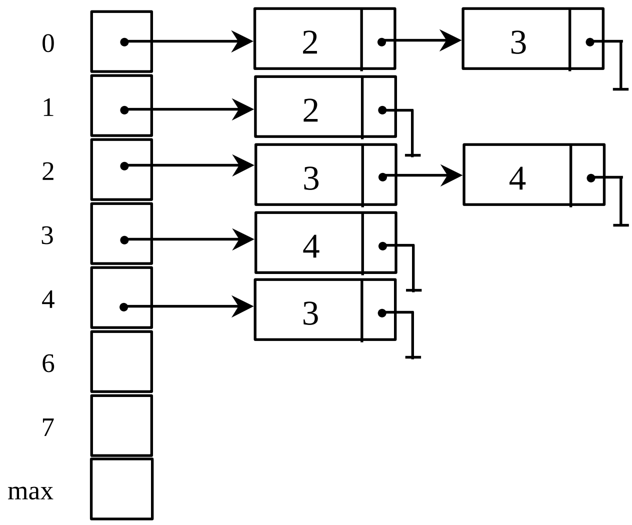
- Khai báo :

```
const MAX = 50;                          // Số đỉnh tối đa của đồ thị
typedef int Dothi[MAX][MAX] ;
  Dothi G ;                              // G là ma trận kề biểu diễn đồ thị
```

b. Danh sách kề (mảng 1 chiều kết hợp với danh sách liên kết):

Một đồ thị được xem là bao gồm danh sách các nút, mỗi nút có một danh sách các nút kề tương ứng.

Ví dụ : Danh sách kề của đồ thị hình 6.4 có dạng:



Hình 6.5 Danh sách kề của đồ thị 6.4

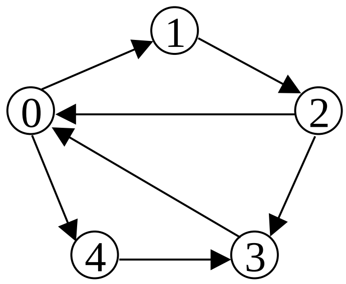
- Khai báo :

```
typedef int BYTE;
  const MAX = 6;
struct node
 {
    int dinh_ke;
    struct node *next;
 };

 typedef struct node *NODEPTR;
 struct   phantu
 {
   NODEPTR pF;
   NODEPTR pL  ;
 };
 typedef struct phantu  Dothi[MAX];
 Dothi G;
```

II. <u>duyệt đồ thị</u>:

Trong đa số các bài toán trên đồ thị, việc lần lượt đi qua tất cả các nút của một đồ thị là rất cần thiết; việc này gọi là duyệt một đồ thị. Ta có nhiều phương pháp để duyệt một đồ thị: duyệt theo chiều sâu và duyệt theo độ rộng.

Để minh họa cho các giải thuật duyệt đồ thị, ta sử dụng đồ thị hình 6.6 sau:



Hình 6.6 Đồ thị minh họa cho các giải thuật duyệt

II.1. <u>Duyệt theo chiều sâu (Depth-First Travelsal)</u>

<u>Nguyên tắc</u> *:*

Giả sử ta đến một nút v có các nút kề lần lượt là w_1, w_2,...w_k. Sau khi duyệt nút v, ta sẽ đi qua w_1 và giữ lại các nút w_2,...,w_k vào Stack. Tiếp tục duyệt nút w_1 và tất cả các nút kề của w_1, rồi mới trở lại duyệt w_2. Lặp lại cho đến khi duyệt hết nút w_k và các nút kề của nó.

<u>Lưu ý</u> :

- Không duyệt một nút hai lần.

- Để tránh trường hợp duyệt sót một nút k' trong đồ thị, ta phải tạo một vòng lặp để có thể đảm bảo duyệt hết các nút của đồ thị.

<u>Giải thuật:</u>

```
void Depth_traverse(int i0)        // nút bắt đầu để từ đó duyệt
{
  int C [MAX] ;                    // de danh dau cac dinh da di qua
  int Stack[MAX] ;                 // Stack de chua cac dinh trong khi duyet
  int sp ;  // sp : con tro dau stack
  int i,x ;

  for (i=0; i<MAX; C[i++] =0 );   // C = {0};
  sp=0;                            // khoi tao Stack
  Stack[sp]=i0;
  C[i0]=1 ;                        // da duyet qua dinh i0
  while (sp >-1)                   // Khi Stack khac rong
  {
    x=Stack[sp];
    sp-- ; // xoa dinh vua tham ra khoi Stack
   cout << x << "    ";            // Tham dinh vua lay ra
   for (i=0; i<MAX; i++)      // dua tat ca cac nút ngon chua duyet tu x vao C
  if (G[x][i] && C[i]==0)
  {
    Stack[++sp]=i;
```

```
    C[i]=1;
   }
   }
}
```

Ví dụ: Áp dụng giải thuật trên cho đồ thị hình 6.6 ta sẽ nhận được kết quả tương ứng với các đỉnh bắt đầu :

Đỉnh bắt đầu	Trình tự duyệt
0	$0 \rightarrow 4 \rightarrow 3 \rightarrow 1 \rightarrow 2$
1	$1 \rightarrow 2 \rightarrow 3 \rightarrow 0 \rightarrow 4$
2	$2 \rightarrow 3 \rightarrow 0 \rightarrow 4 \rightarrow 1$
3	$3 \rightarrow 0 \rightarrow 4 \rightarrow 1 \rightarrow 2$
4	$4 \rightarrow 3 \rightarrow 0 \rightarrow 1 \rightarrow 2$

II.2. Duyệt theo độ rộng (Breadth First Travelsal)

Nguyên tắc :

Giả sử ta đến một nút v có các nút kề lần lượt là w_1, w_2,...w_k. Sau khi duyệt nút v, ta duyệt hết các nút w_i của v, rồi mới tiếp tục xem các nút kề của từng w_i. Duyệt các nút kề chưa được duyệt của các nút w_i. Cứ tiếp tục như vậy cho đến khi hết các nút của đồ thị.

**** Giải thuật:***

```
// Hang doi phuc vu cho cong viec duyet Width_traverse
 struct node
 {
     int diachi ;
     struct node *next;
 };
 typedef node *Node_queue;
 struct Queue
 {
     Node_queue Front, Rear;
 } Q;
 void Insert_queue(Queue &Q, int x)
 {
   Node_queue p;
   p = (Node_queue)malloc(sizeof(struct node));
   p->diachi = x;
   if (Q.Front==NULL)
     Q.Front=p;
```

```
   else Q.Rear->next=p;
      Q.Rear=p;
   p->next=NULL;
 }
 int Delete_queue(Queue &Q)
 {
  Node_queue p;
  int x;
  if(Q.Front==NULL)
  {
    cout <<"\nHang doi rong";
    getche();
    exit(1);
  }
  else
  {
    p = Q.Front;          // nut can xoa la nut dau
    x = p->diachi;
    Q.Front = p->next;
    free(p);
    return x;
  }
}
void Width_traverse(int i0)   // dinh bat dau de tu do duyet
{
 BYTE C [MAX] ;        // de danh dau cac dinh da di qua
 int i,x ;
 cout << "Cac dinh cua do thi theo giai thuat duyet rong \n";
 for (i=0 ; i< MAX; C[i++]=0);
 Q.Front= NULL;        // khoi tao hang doi
 Insert_queue(Q,i0);
 C[i0]=1 ;          // da duyet qua dinh i0
 while (Q.Front !=NULL)
 {
   x=Delete_queue(Q);  // xoa dinh vua tham ra khoi hang doi
   cout << x << "    ";   // Tham dinh Q[l]
   for (i=0; i<MAX; i++) // dua tat ca cac dinh ngon chua duyet tu x vao Q
    if (G[x][i] && C[i]==0)
    {
      Insert_queue(Q,i);
```

```
        C[i]=1;
      }
    }
  }
```

Ví dụ: Áp dụng giải thuật trên cho đồ thị hình 6.6 ta sẽ nhận được kết quả tương ứng với các đỉnh bắt đầu :

Đỉnh bắt đầu	**Trình tự duyệt**
0	$0 \to 1 \to 4 \to 2 \to 3$
1	$1 \to 2 \to 0 \to 3 \to 4$
2	$2 \to 0 \to 3 \to 1 \to 4$
3	$3 \to 0 \to 1 \to 4 \to 2$
4	$4 \to 3 \to 0 \to 1 \to 2$

III. bÀi toÁn bao đóng truyền ứng:

III.1. Khái niệm:

Bao đóng truyền ứng của một đồ thị G có n nút là một ma trận cho chúng ta biết giữa 2 nút x và y bất kỳ trên đồ thị có tồn tại một đường đi với chiều dài nhỏ hơn hay bằng n hay không.

Ma trận hình 6.7 là bao đóng truyền ứng của đồ thị hình 6.4. Các số 1 trong ma trận cho ta biết từ nút x đến nút y tồn tại đường đi với chiều dài nhỏ hơn hay bằng 5.

	0	**1**	**2**	**3**	**4**
0	0	0	1	1	1
1	0	0	1	1	1
2	0	0	0	1	1
3	0	0	0	1	1
4	0	0	0	1	1

Hình 6.7: Bao đóng truyền ứng của đồ thị hình 6.4

III.2. Thuật toán WarShall :

1- Cho P^1 là ma trận kề của đồ thị G cho chúng ta biết giữa 2 nút x và y bất kỳ trên đồ thị có tồn tại một đường đi với chiều dài bằng 1.

2. Tính $P^2 = P^1 \times P^1$: cho chúng ta biết giữa 2 nút x và y bất kỳ trên đồ thị có tồn tại một đường đi với chiều dài bằng 2.

$P^1 \times P^1$ chính là phép nhân 2 ma trận với phép nhân là and và phép cộng là or.

$$P_{ij}^{(2)} = \bigcup_{k=1}^{n} \left(P_{ik}^{1} * P_{kj}^{1} \right)$$

3. Tương tự, ta tính P^3, P^4, ..., P^n.

4. Bao đóng truyền ứng = $P^1 \cup P^2 \cup P^3 ... \cup P^n$.

Ví dụ: Với đồ thị G hình 6.4, thì lần lượt các P^i là:

$P^1 =$	0	0	1	1	0
	0	0	1	0	0
	0	0	0	1	1
	0	0	0	0	1
	0	0	0	1	0

$P^2 =$	0	0	0	1	1
	0	0	0	1	1
	0	0	0	1	1
	0	0	0	1	0
	0	0	0	0	1

$P^3 =$	0	0	0	1	1
	0	0	0	1	1
	0	0	0	1	1
	0	0	0	0	1
	0	0	0	1	0

$P^4 =$	0	0	0	1	1
	0	0	0	1	1
	0	0	0	1	1
	0	0	0	1	0
	0	0	0	0	1

$P^5 =$	0	0	0	1	1
	0	0	0	1	1
	0	0	0	1	1
	0	0	0	0	1
	0	0	0	1	0

Bao đóng truyền ứng của đồ thị G :

	0	**1**	**2**	**3**	**4**

0	0	0	1	1	1
1	0	0	1	1	1
2	0	0	0	1	1
3	0	0	0	1	1
4	0	0	0	1	1

* **Chương trình**:

```
#include <stdio.h>
#include <conio.h>
const MAX = 4;
int G[MAX][MAX]= { {0,0,1,0},
                   {0,0,1,0},
                   {0,0,1,1},
                   {0,1,0,0}
          } ;
void Xuat(int P[][MAX])
{
  int i,j;
  for( i=0; i<MAX;i++)
  {
    for( j=0; j< MAX; j++)
      printf ("%4d",  P[i][j] ) ;
    printf ( "\n");
  }
 getch();
 printf("\n");
}
void WarShall( int G[][MAX])
{
   int i,j,k, dem;
   int C[MAX][MAX], P[MAX][MAX];
   int BD[MAX][MAX];          // bao đóng truyền ứng của ma trận G
   for( i=0; i<MAX;i++)  // P1
    for (j=0;j<MAX; j++)
      BD[i][j]= P[i][j]=G[i][j];
   for ( dem=2; dem<=MAX; dem++)
   {
    for ( i=0; i<MAX; i++)
      for (j=0; j< MAX;j++)
      {
```

```
        C[i][j]=0;
        for (k=0; k<MAX; k++)
          C[i][j]= C[i][j] || (P[i][k] && G[k][j]);
      }
    for (i=0;i<MAX;i++)
     for (j=0 ; j<MAX; j++)
     {
       P[i][j]=C[i][j];
       BD[i][j]=BD[i][j] || P[i][j]; // OR don ma tran P vua tinh vao bao dong
     }
     Xuat(P);       // Kiem tra tung Pi
   }
 Xuat(BD);
}
```

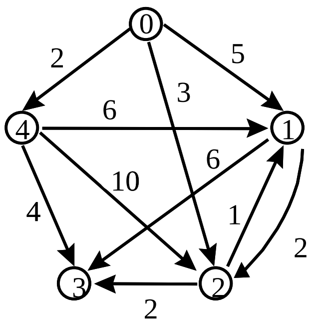
void main()

```
{
 int P[MAX][MAX] ;
 int i,j ;
 clrscr();
 WarShall(G);
 getch();
}
```

IV. <u>GIảI THUậT TÌM ĐườNG ĐI NGẮN NHấT</u>:

Đối với một đồ thị có trọng số, mỗi cạnh sẽ có một giá trị trọng số tương ứng, tìm đường đi ngắn nhất trên đồ thị G từ một nút v đến một nút w là bài toán tìm đường đi có trọng lượng nhỏ nhất từ v đến w.

Trọng số của một cạnh có thể là thời gian để đi qua một cạnh, phí tổn, khoảng cách hoặc lưu lượng.



Hình 6.8 Đồ thị hữu hướng có trọng số

* <u>Thuật toán Dijkstra</u>: Tìm các đường đi ngắn nhất từ nút v đến các nút còn lại của đồ thị.

<u>Input</u> : - Đồ thị G là ma trận kề hữu hướng có trọng số với qui ước sau:

+ Nếu u kề với v thì độ dài cung > 0

+ Nếu u không kề với v thì độ dài cung = -1

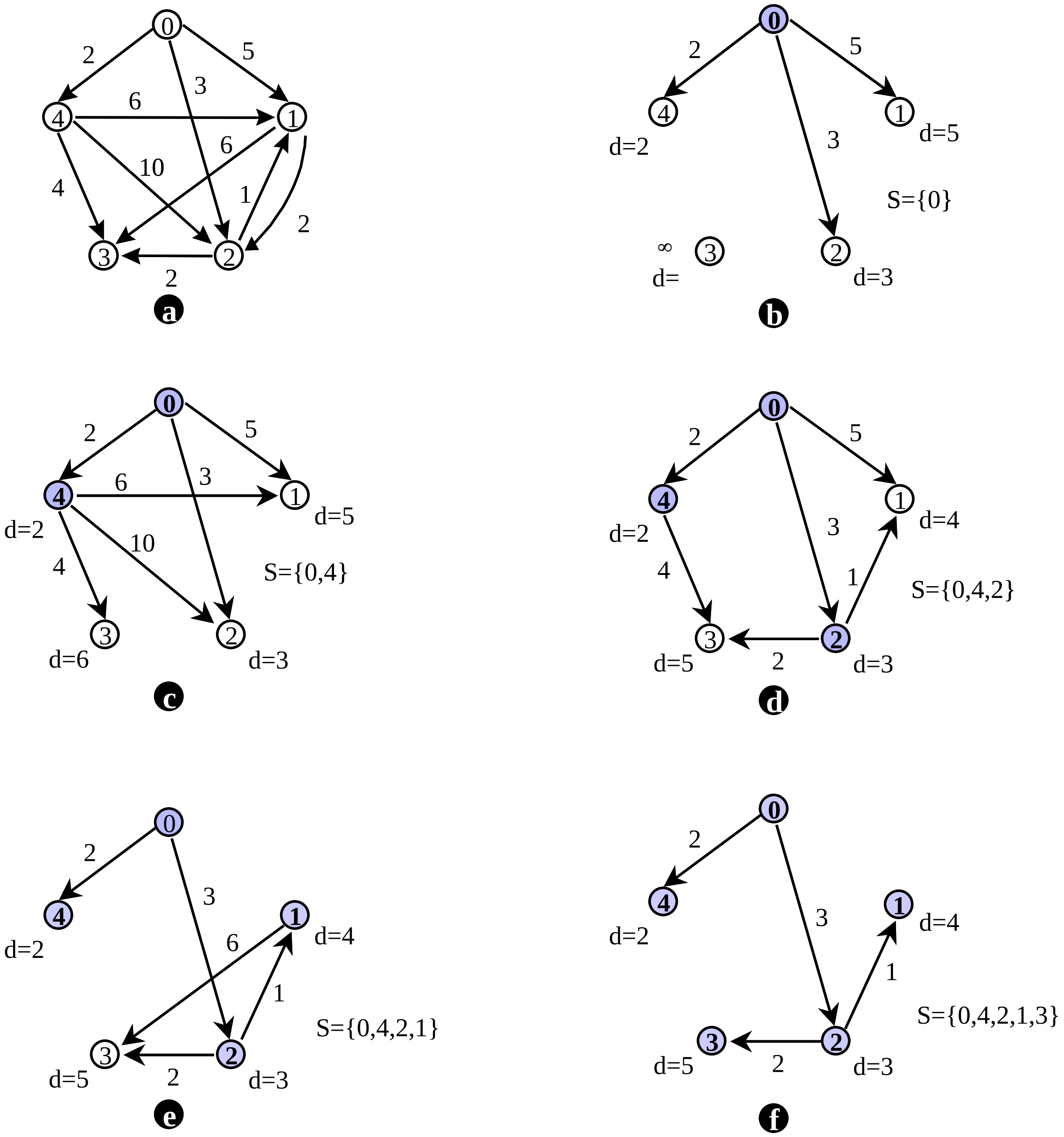
- Nút v là nút ta bắt đầu tìm các đường đi ngắn nhất từ v đến tất cả các nút còn lại trong đồ thị.

Output: Độ dài ngắn nhất từ v đến tất cả các nút còn lại trong đồ thị.

* Giải thuật:

Dùng một tập S để chứa các nút đã xác định được khoảng cách ngắn nhất từ nút v đến các nút đó.

Dùng một mảng Dist để chứa giá trị các khoảng cách ngắn nhất này. Nếu nút u ở trong S thì Dist [u] là giá trị khoảng cách ngắn nhất từ v cho đến u. Nếu u chưa có trong S thì Dist [u] chứa độ dài từ v đến một nút w nào đó trong S cộng với khoảng cách từ w đến u. Mảng Dist sẽ được khởi tạo bằng giá trị trọng lượng từ nút v đến các nút còn lại nếu có cạnh trực tiếp, và bằng vô cùng (MAXINT) nếu không có cạnh trực tiếp.



Hình 6.9 Minh họa các bước áp dụng giải thuật Dijkstra tìm đường đi ngắn nhất từ nút 0 cho đến tất cả các nút còn lại trong đồ thị hình 6.8

* <u>Giải thuật</u>

```
void Shortest_path(BYTE v, const long G[][MAX])
{
/* Cost chua do dai cac cung cua do thi G, voi qui uoc
  neu G[i][j]=-1 thi Cost[i][j] = MAXINT */
long Cost[MAX][MAX] ;
/* Dist[j] : the hien do dai cua duong di ngan nhat tu nut v den nut j
      trong do thi dinh huong G co MAX nut; G duoc bieu dien boi
      ma tran ke huu huong co trong so kich thuoc MAX x MAX */
long int Dist  [MAX] ;
```

```
int duongdi[MAX] ; // chua lo trinh duong di ngan nhat tu v den cac dinh
duongdi= {v};
/* S : Tap cac dinh (ke ca v) theo do cac duong di ngan nhat da xac lap */
        int S[MAX] ;
        int w,i,j,u,k ;
  for (i=0; i<MAX; i++)
         for (j=0;j<MAX;j++)
         Cost[i][j]= (G[i][j]==-1? MAXINT : G[i][j]);
  // Khoi tao S va Dist
  for (i=0; i< MAX; S[i]=0,  Dist[i]=Cost[v][i],i++ );
  S[v]=1; Dist[v]=0 ; k=1; //dua v vao S
  while (k < MAX) // xac dinh n-1 duong di tu dinh v
  {
          // chon u sao cho: Dist[u] = min (Dist[j]), S[j]=0
          j=0;
          while (j<MAX && S[j]!=0) j++; // Tim S[j] = 0 dau tien
          u=j;
          for (j=u; j<MAX; j++)
                  if (S[j] == 0 && Dist[u] > Dist[j])       u=j;

          //Dua u vao tap S *)
          S[u]=1 ; k++;
          for (w=0; w< MAX; w++)
                  if (S[w] == 0)
                          if (Dist[u]+Cost[u][w] < Dist [w])
                          {       Dist[w]= Dist[u]+Cost[u][w];
                                  duongdi[w] = u; // ie  : u → w
                          }
  }
  for (w=0; w<MAX; w++)
    if (Dist[w] < MAXINT)
       cout << "\n" << v << "->" <<w <<": " << Dist[w];
    else
       cout << "\n" << v << "->" <<w << ": Khong co duong di";
}
```

Muốn in lộ trình ngắn nhất từ đỉnh s đến đỉnh t :

```
printf("\nLo trinh tu %d->%d la: ", s, t);
i = t;
while(i != s)
```

```
        { printf("%d <- ", i);  i = duongdi[i];  }
    printf("%d", s);
```

V. SẮp thứ tự Topo:

V.1. Khái niệm:

Sắp thứ tự Topo là một quá trình sắp thứ tự các phần tử mà trong đó có định nghĩa một thứ tự bộ phận, nghĩa là một thứ tự cho trước trên một vài cặp các phần tử mà không phải trên tất cả các phần tử.

Một thứ tự bộ phận của một tập hợp S là một quan hệ giữa các phần tử của S. Nó được ký hiệu bởi <, đọc là "đứng trước", và thỏa mãn ba tính chất sau đây đối với mọi phần tử phân biệt x, y, z của S:

(1) Nếu x < y và y < z thì x < z (tính bắc cầu)

(2) Nếu x < y thì không có thể có y < x (tính phản xứng)

(3) Không thể có x < x (tính không phản xạ)

Thông thường, bài toán Topo nhằm để sắp xếp các phần việc trong một công việc nào đó cho logic, nghĩa là khi ta thực hiện đến phần việc thứ i thì phải đảm bảo đã thực hiện các phần việc chuẩn bị cho nó trước rồi. Chẳng hạn như sắp xếp các tín chỉ môn học sao cho khi đăng ký đến môn học i thì ta phải học qua các môn chuẩn bị trước cho nó.

V.2. Thuật toán: Để đơn giản, ta lấy ví dụ sau để minh họa:

Giả sử khoa công nghệ thông tin có giảng dạy các môn học sau: đại số tuyến tính (ĐSTT), Tin học cơ bản (THCB), Lập trình căn bản (LTCB), Kỹ thuật lập trình (KTLT), Cấu trúc dữ liệu (CTDL), Cấu trúc máy tính (CTMT), Cơ sở dữ liệu (CSDL), Quản trị giao tác (QTGT), Phân tích & thiết kế hệ thống thông tin (PTTK), Hệ quản trị cơ sở dữ liệu (HQT).

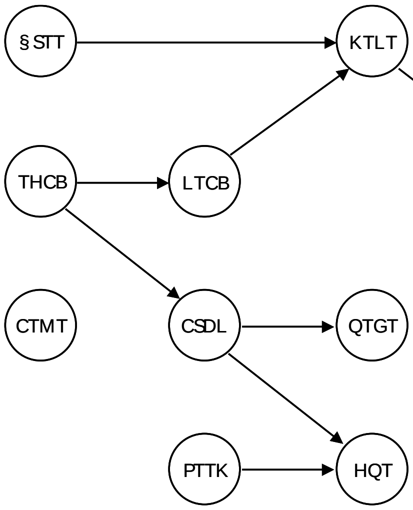
Yêu cầu: Hãy sắp xếp các môn học trên sao cho khi sinh viên đăng ký tín chỉ môn học thì phải đảm bảo các điều kiện sau:

Môn học	Các môn phải học trước
Đại số tuyến tính	
Tin học cơ bản	
Lập trình căn bản	Tin học cơ bản
Kỹ thuật lập trình	Lập trình căn bản, Đại số tuyến tính
Cấu trúc dữ liệu	Kỹ thuật lập trình
Cấu trúc máy tính	
Cơ sở dữ liệu	Tin học cơ bản
Quản trị giao tác	Cơ sở dữ liệu
Phân tích & thiết kế hệ thống thông tin	

Hệ quản trị cơ sở dữ liệu	Cơ sở dữ liệu, Phân tích & thiết kế hệ thống thông tin

Ta có đồ thị minh họa bài toán trên với qui ước:

Cung <u,v> với u là môn phải học trước môn v



Hình 6.10 Đồ thị minh họa bài toán sắp thứ tự các môn học thỏa ràng buộc đã cho

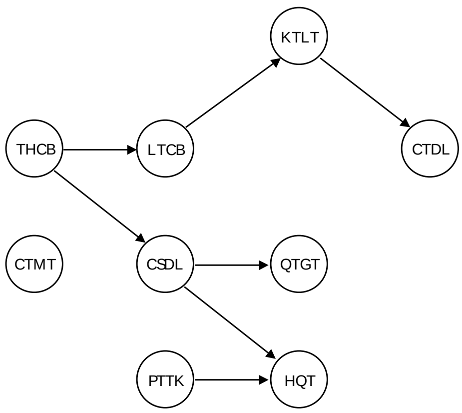
<u>Giải thuật:</u>

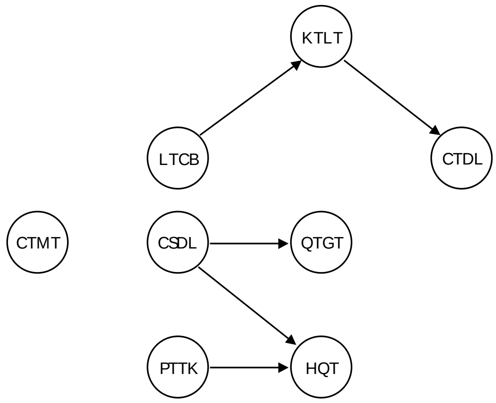
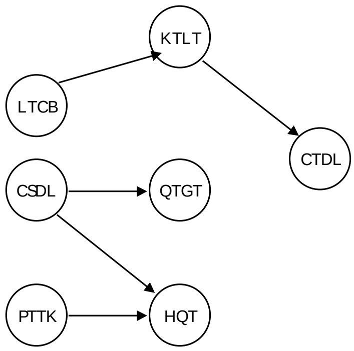
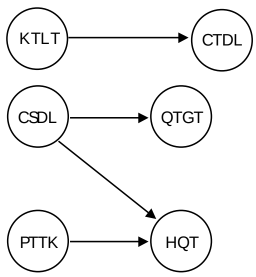
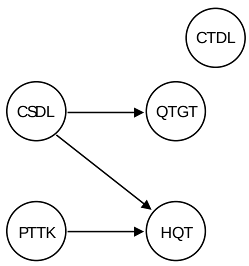
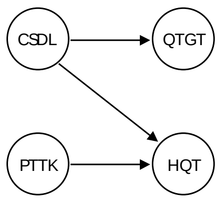
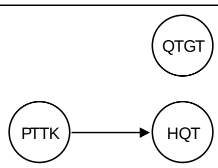
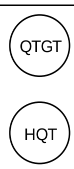
(i) Ta tìm nút nào không có cung đến nó thì chọn, sau đó hủy tất cả các cung từ nút đó đi ra.

(ii) Lặp lại bước i cho đến khi không còn nút nào trên đồ thị

<u>Lưu ý</u>: Nếu trong quá trình chọn mà không tìm được 1 nút không có cung tới nó thì có nghĩa là đồ thị có chu trình. Do đó, không thể thực hiện sắp Topo được.

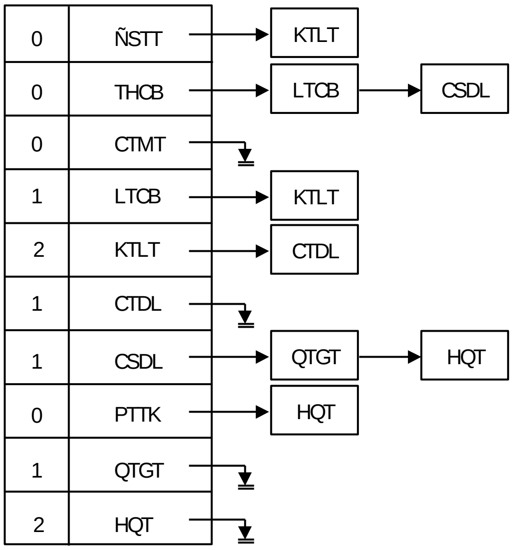
Áp dụng giải thuật trên với đồ thị hình 6.10

Nút chọn	**Đồ thị còn lại**
ĐSTT	

THCB	
CTMT	
LTCB	
KTLT	
CTDL	
CSDL	
PTTK	

QTGT	HQT
HQT	

V.3. Cài đặt: Do số nút trên đồ thị thường nhiều và số cung trên đồ thị tương đối ít nên để tiết kiệm bộ nhớ, ta chọn cấu trúc dữ liệu để lưu trữ là danh sách kề; trong đó mảng 1 chiều chứa danh sách các nút của đồ thị, còn danh sách liên kết sẽ chứa các cung trên đồ thị. Chẳng hạn như danh sách kề của đồ thị hình 6.10 như sau:



Hình 6.11 Danh sách kề của đồ thị hình 6.10

Để biết được có bao nhiêu cung đi tới nút i, ta thêm trường count vào mảng chứa danh sách các nút.

Dưới đây là chương trình sắp Topo với giả thiết của bài toán được chứa trong 1 file văn bản. File văn bản có dạng sau:

Số n : số nút của đồ thị

Ma trận số biểu diễn đồ thị

<u>Ví dụ</u>: File văn bản biểu diễn đồ thị hình 6.10 có dạng:

10

0	0	0	0	**1**	0	0	0	0	0
0	0	0	**1**	0	0	**1**	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	**1**	0	0	0	0	0
0	0	0	0	0	**1**	0	0	0	0
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	**1**	**1**
0	0	0	0	0	0	0	0	0	**1**
0	0	0	0	0	0	0	0	0	0

0 0 0 0 0 0 0 0 0 0

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <alloc.h>
#include <stdlib.h>
struct node
 {
    int dinh_ke;
    struct node *next;
 };
typedef struct node *NODEPTR;
 struct   phantu_ke
 {
   int count;
   NODEPTR pF;
   NODEPTR pL  ;
 };
 typedef struct phantu_ke  Dothi[100];
 Dothi G;
 int MAX;
void Init_graph(Dothi G)
{
  for(int i=0; i< MAX; G[i++].pF=NULL);
}
void Create_graph()
{
 int i,j ;
 NODEPTR p;
 unsigned B[100][100];
 FILE *fptr;
 if ( (fptr = fopen ("dt.txt", "rt")) == NULL )
 {  printf("\nKhong the mo file dt.txt");
    getch();
    exit(0);
 }
 fscanf(fptr,"%d", &MAX);
 for (i=0; i< MAX ; i++)
   for (j=0; j<MAX; j++)
```

```
    fscanf(fptr,"%d", &B[i][j]);
  /// Khoi tao rong do thi
  Init_graph(G);
  //Tao count : so cung toi dinh j
  for (j=0; j<MAX; j++)
  {
    G[j].count=0;
    for (i=0; i< MAX; i++)
      if (B[i][j] ==1)  G[j].count++;
  }
  for (i=0; i< MAX; i++)
    for (j=0;j<MAX; j++)
      if (B[i][j] == 1)
      {
            p = (NODEPTR)malloc(sizeof(struct node));
            p->next=NULL;
            p->dinh_ke=j;
            if( G[i].pF == NULL)  G[i].pF=p;
            else G[i].pL->next=p;
            G[i].pL=p;
      }
}
```

void Topo_Sort(Dothi G)

```
{
  int Stack[100];
  int i,j,k, Sp=-1 ;
  NODEPTR  p;
  for(i=0;i<MAX; i++)  // Dua vao Stack tat cac cac nut khong co cung di
                       // toi no
    if (G[i].count==0)  {  // day la cac task co the lam doc lap
                        Stack[++ Sp]=i;
                     }
  for( i=0; i<MAX; i++)
  {
    if (Sp ==-1)
    {
       printf("\nCo chu trinh trong do thi!!!");
       exit(0);
    }
    j=Stack[Sp--];  printf("%5d",j);  // Lay 1 nut trong Stack ra
```

```
    p=G[j].pF;
    while (p !=NULL)
    {
      k=p->dinh_ke; // k la ngon cua cung j --> k
      G[k].count --;
      if (G[k].count == 0)  // khong co dinh nao toi nut k
      {
              Stack[++Sp]=k;
      }
      p=p->next;
    }
  }
}
```

void main()

```
{  clrscr();
   Create_graph();
   Topo_Sort(G);
   getch();
}
```

BÀi tập :

1. Viết thủ tục ReadGraph để nhập vào các đỉnh và các cạnh của đồ thị G từ 1 file văn bản, biết rằng:

   - Nội dung của file văn bản là như sau:

     n

     u v trọngsố

     .....

     với : n : số nút của đồ thị G

     u v trọngsố : chiều dài đường đi từ nút u đến nút v

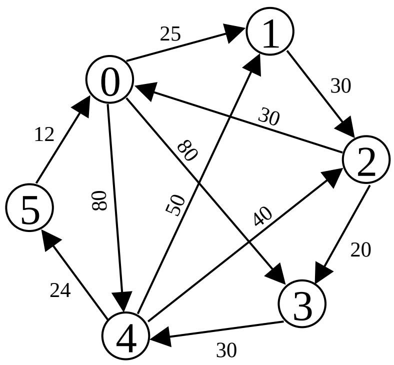
   - Cấu trúc dữ liệu của đồ thị G được sử dụng là :

   a. Bảng kề

   b. Danh sách kề

2. Cho một đồ thị G, viết thủ tục WriteGraph để in các đỉnh của đồ thị, và các cạnh của đồ thị ra màn hình.
3. Cho một đồ thị G. Hãy xác định xem giữa 2 nút u và v có đường đi hay không? Nếu có, hãy xác định lộ trình từ nút u đến nút v.
4. Cho một đồ thị G. Hãy xác định xem đồ thị G có liên thông hay không ?
5. Cài đặt và kiểm tra thủ tục tìm đường đi ngắn nhất từ nút u cho đến nút v trong một đồ thị có hướng. Hãy xác định rõ lộ trình đó và cho biết chiều dài đường đi ngắn nhất là bao nhiêu?

Minh họa các bước của giải thuật Dijkstra tìm đường đi ngắn nhất từ nút 0 đến nút 5 trên đồ thị sau:



TÀI LIỆU THAM KHẢO

[1]	Cấu trúc dữ liệu - ứng dụng và cài đặt bằng C	Nguyễn Hồng Chương	1999
[2]	Cấu trúc dữ liệu + Giải thuật = Chương trình	Niklaus Wirth - Người dịch Nguyễn Quốc Cường	
[3]	Cấu trúc dữ liệu	Đỗ Xuân Lôi	
[4]	Cấu trúc dữ liệu	Nguyễn Trung Trực	1992
[5]	Phân tích và thiết kế giải thuật ĐH BK Tp. Hồ Chí Minh	Đinh Nghiệp	1992
[6]	Course 12.2AB2 Data Structures and Algorithms II - http://www.cee.hw.ac.uk/~alison/alg/lectures.html	Alison Cousey	1999