



CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT

TS. Phan Thị Hà
Học Viện CNBCVT



BT kiểm tra kiến thức LT

- Viết chương trình tính tổng các số nguyên tố của 1 ma trận các số nguyên. Biết rằng ct gồm có ít nhất các hàm sau:
- Void Nhập(int **a, int n); nhập ma trận
 - Voi nhap(int a[][10], int n)
- int ngt(int so): Xác định so có phải ng tố k?
- Int Tong(int ** a, int n): Tính tổng các số nguyên tố theo yêu cầu đầu bài
 - Int Tong(int a[][10], int n)
- Int Main()



Danh sách nhóm trưởng

- Nhóm 4: Bùi Thành Lộc 01663177081 mrkasa1201@gmail.com
 - Phan Văn Quang 0988806405 buinhulac110i@gmail.com
- Nhóm 5: Trần Quang Hoàn 0919963616 Tranhoanhq@gmail.com
 - Đặng thị Ngọc Trâm 0915365653 ngoctramnd2013@gmail.com
- Nhóm 7: Phan Văn Trung 0943241608 trungphanptit@gmail.com
 - Phạm Quang Sơn 0964535086 quágon137@gmail.com
- Nhóm 8: Bùi Văn Công 0978776096 conghcl2010@gmail.com
 - Nguyễn Mạnh Toàn 01677535717
nguyenmanhtoan1501@gmail.com
- Nhóm 9: Nguyễn Thanh Tuấn 01685155229
thanh.tuan.1995.0411@gmail.com.vn
 - Đào Bá Huỳnh 0974021385 boy.sl.pro.1994@gmail.com
- Nhóm 10: Nguyễn Văn Tiến 0966626212 nvtien94@gmail.com
 - Trần Lương Bằng 0975453108



Nội dung

- Chương 1: Phân tích – Thiết kế giải thuật
- Chương 2: Đệ Quy
- Chương 3: Mảng và Danh sách liên kết
- Chương 4: Ngăn xếp và Hàng đợi
- Chương 5: Cấu trúc dữ liệu kiểu cây
- Chương 6: Đồ thị
- Chương 7: Sắp xếp và tìm kiếm



CHƯƠNG 1:

PHÂN TÍCH – THIẾT KẾ GIẢI THUẬT



Giải thuật là gì?

- Giải thuật – hay thuật toán

- *Thuật toán là một chuỗi hữu hạn các lệnh. Mỗi lệnh có một ngữ nghĩa rõ ràng và có thể được thực hiện với một lượng hữu hạn tài nguyên trong một khoảng hữu hạn thời gian.*

- *Tính chất của thuật toán:*

- *Xác định rõ đầu vào, đầu ra*
- *Hữu hạn*
- *Chính xác*
- *Tổng quát*



Ngôn ngữ diễn đạt giải thuật

- Ngôn ngữ tự nhiên
- Sơ đồ khối
- Giả mã
- Vd



Phân tích thuật toán

- Nhiệm vụ: Phân tích – tính toán thời gian thực hiện của chương trình



Phân tích thuật toán

- **Định nghĩa.** Cho 2 hàm ***f và g là các hàm thực không âm*** có miền xác định trong tập số tự nhiên. Ta viết $f(n)=O(g(n))$ (đọc là $f(n)$ là O lớn của $g(n)$) nếu tồn tại hằng số $C>0$ và số tự nhiên n_0 sao cho $f(n)<Cg(n)$, với mọi $n>n_0$.
- Nếu một thuật toán có thời gian thực hiện $T(n)=O(g(n))$ ta nói rằng thuật toán có thời gian thực hiện cấp cao nhất là $g(n)$

Đôi khi ta cũng nói đơn giản là thuật toán có thời gian thực hiện cấp $g(n)$ hay độ phức tạp tính toán của thuật toán là $g(n)$.

Người ta cũng thường nói đến độ phức tạp tính toán của thuật toán trong trường hợp xấu nhất, trường hợp tốt nhất, độ phức tạp trung bình.



Các quy tắc để đánh giá thời gian thực hiện thuật toán

- Nếu $T_1(n) = O(f(n))$, $T_2(n) = O(g(n))$, khi đó
- Quy tắc tổng: $T_1(n) + T_2(n) = O(\max(f(n), g(n)))$
- Quy tắc nhân: $T_1(n)T_2(n) = O(f(n)g(n))$



Một số quy tắc chung:

- Thời gian thực hiện các lệnh gán, đọc, ghi .v.v, luôn là $O(1)$.
- Thời gian thực hiện chuỗi tuần tự các lệnh được xác định theo quy tắc cộng cấp độ tăng.
 - Có nghĩa là thời gian thực hiện của cả nhóm lệnh tuần tự được tính là thời gian thực hiện của lệnh lớn nhất.
- Thời gian thực hiện lệnh rẽ nhánh (If) được tính bằng thời gian thực hiện các lệnh khi điều kiện kiểm tra được thỏa mãn và thời gian thực hiện việc kiểm tra điều kiện.
 - Trong đó thời gian thực hiện việc kiểm tra điều kiện luôn là $O(1)$.
- Thời gian thực hiện 1 vòng lặp được tính là tổng thời gian thực hiện các lệnh ở thân vòng lặp qua tất cả các bước lặp và thời gian để kiểm tra điều kiện dừng.
 - Thời gian thực hiện này thường được tính theo quy tắc nhân cấp độ tăng số lần thực hiện bước lặp và thời gian thực hiện các lệnh ở thân vòng lặp.



Ví dụ 1

- Giả sử ta cần đánh giá độ phức tạp tính toán của các câu lệnh sau đây:
- (a) `for(i=0;i< n;i++)`
- `{x=0;`
- `for(j=0;j<n;j++) x++;`
- `};`
- (b) `for(i=0;i< n;i++) y++;`
- Câu lệnh `x++` được đánh giá là $O(1)$, do đó câu lệnh (a) có thời gian thực hiện là $n(n+1)$ do đó được đánh giá $O(n^2+n) = n^2$, câu lệnh (b) được đánh giá là $O(n)$. Do đó thời gian thực hiện cả 2 thuật toán trên được đánh giá là:
- $O(n(n+1)+n) = O(\max(n(n+1),n)=O(n(n+1))) = n^2$.

Thuật toán nổi bọt như sau :

```
void bubble (int a[n]){
```

```
    int i, j, temp;
```

```
    1. for (i = 0; i < n-1; i++)
```

```
        2. for (j = n-1; j >= i+1; j--)
```

```
            3. if (a[j-1] > a[j]){
```

```
                // Đổi chỗ cho a[j] và a[j-1]
```

```
                4. temp = a[j-1];
```

```
                5. a[j-1] = a[j];
```

```
                6. a[j] = temp;
```

```
            }
```

```
    }
```

Kích thước dữ liệu vào chính là số phần tử được sắp, n . Mỗi lệnh gán sẽ có thời gian thực hiện cố định, không phụ thuộc vào n , do vậy, các lệnh 4, 5, 6 sẽ có thời gian thực hiện là $O(1)$, tức thời gian thực hiện là hằng số. Theo quy tắc cộng cấp độ tăng thì tổng thời gian thực hiện cả 3 lệnh là $O(\max(1, 1, 1)) = O(1)$.

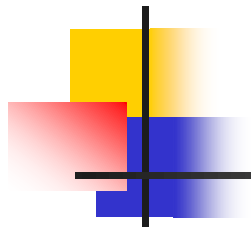
If sẽ có thời gian thực hiện là $O(1)$.

Vòng lặp j . mỗi bước lặp có thời gian thực hiện là $O(1)$. Số bước lặp là $n-i$, do đó theo quy tắc nhân cấp độ tăng thì tổng thời gian thực hiện của vòng lặp này là $O((n-i) \times 1) = O(n-i)$.

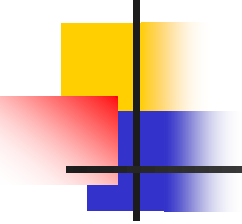
Vòng lặp i , được thực hiện $(n-1)$ lần: $O(n-1)$

Do đó, tổng thời gian thực hiện của chương trình là:

$$\sum (n-i) = n(n-1)/2 = n^2/2 - n/2 = O(n^2)$$



CHƯƠNG 2: ĐỆ QUY

- 
-
- Định nghĩa đệ qui
 - Hàm trình đệ qui
 - Các ưu điểm của hàm đệ qui
 - Một số thuật toán đệ qui



Đệ quy là gì (PP định nghĩa bằng ĐQ)

■ Khái niệm

- Một đối tượng được gọi là đệ quy nếu nó hoặc một phần của nó được định nghĩa thông qua khái niệm về chính nó.

■ Ví dụ

- Định nghĩa số tự nhiên:
 - 0 là số tự nhiên.
 - Nếu k là số tự nhiên thì $k+1$ cũng là số tự nhiên.
- Định nghĩa hàm giai thừa, $n!$.
 - Khi $n=0$, định nghĩa $0!=1$
 - Khi $n>0$, định nghĩa $n!=(n-1)! \times n$

Như vậy, khi $n=1$, ta có $1!=0! \times 1 = 1 \times 1 = 1$. Khi $n=2$, ta có $2!=1! \times 2 = 1 \times 2 = 2$, v.v.



Giải thuật đệ qui

- Nếu lời giải của bài toán P được thực hiện bằng lời giải của bài toán P' giống như P thì lời giải này được gọi là lời giải đệ qui. Giải thuật tương ứng với lời giải bài toán P được gọi là giải thuật đệ qui.



Hàm đệ qui

- Hàm này có thể gọi chính nó nhưng nhỏ hơn

Khi hàm gọi chính nó, mục đích là để giải quyết 1 vấn đề tương tự, nhưng nhỏ hơn. Vấn đề nhỏ hơn này, cho tới 1 lúc nào đó, sẽ đơn giản tới mức hàm có thể tự giải quyết được mà không cần gọi tới chính nó nữa.

Quá trình hoạt động của giải thuật ĐQ

- Bước phân tích
- Bước thay thế ngược lại

VD: tính $f(n) = n!$

+ $f(n) = n * f(n-1)$

.....

$f(1) = 1 * f(0)$

$f(0) = 0! = 1$

+ $f(0) = 1$

$f(1) = 1 * f(0)$

.....

$f(n) = n * f(n-1)$



Đặc điểm của hàm đệ quy

- ⇒ Chỉ ra trường hợp dừng đệ quy
- ⇒ Chỉ ra việc đệ quy



Khi nào sử dụng đệ quy

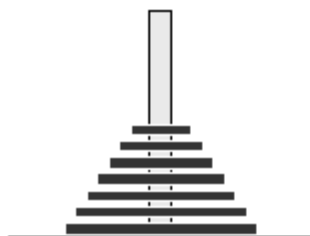
- Vấn đề cần xử lý phải được giải quyết có đặc điểm đệ quy
- Ngôn ngữ dùng để viết chương trình phải hỗ trợ đệ quy. Để có thể viết chương trình đệ quy chỉ cần sử dụng ngôn ngữ lập trình có hỗ trợ hàm hoặc thủ tục, nhờ đó một thủ tục hoặc hàm có thể có lời gọi đến chính thủ tục hoặc hàm đó



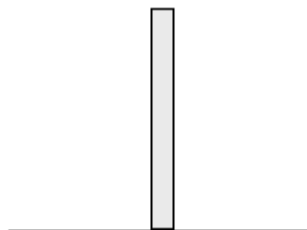
Một số thuật toán đệ qui

- Thuật toán cột cờ (sách)
- Thuật toán quay lui
 - Thuật toán sinh xâu (hoặc sinh hoán vị)
 - Mã đi tuần
 - Bài toán 8 quân hậu (sách)

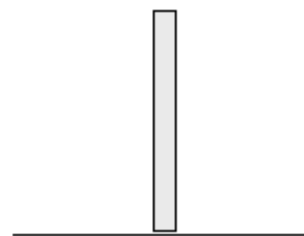
Bài toán cột cò



Cọc A

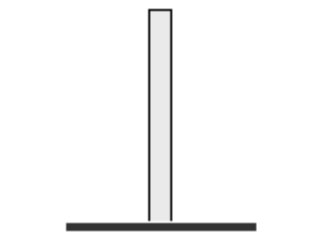


Cọc B

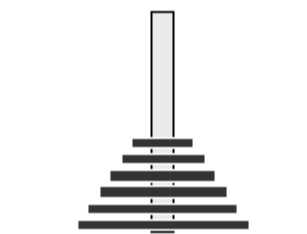


Cọc C

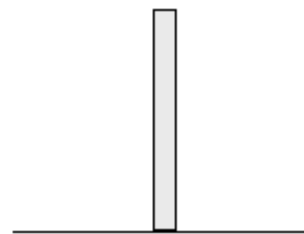
Bước 1: Chuyển $n-1$ đĩa bên trên từ cọc A sang cọc B, sử dụng cọc C làm cọc trung gian.



Cọc A



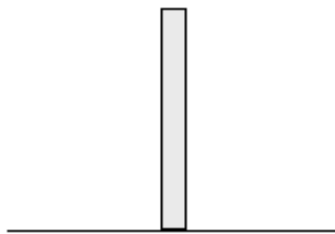
Cọc B



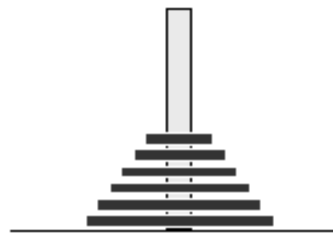
Cọc C

Bài toán cột cò

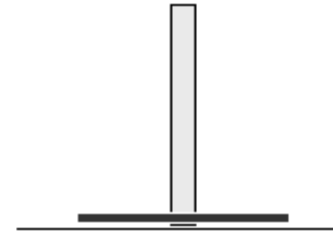
Bước 2: Chuyển đĩa dưới cùng từ cọc A thẳng sang cọc C.



Cọc A

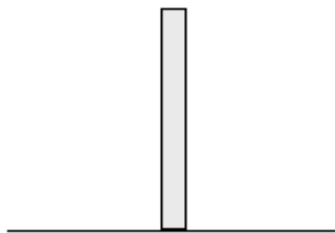


Cọc B

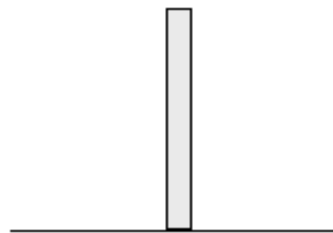


Cọc C

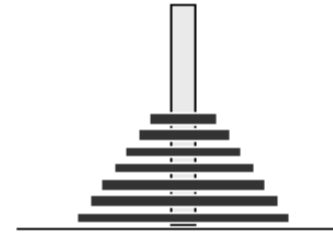
Bước 3: Chuyển $n-1$ đĩa từ cọc B sang cọc C sử dụng cọc A làm cọc trung gian.



Cọc A



Cọc B



Cọc C

Như vậy, ta thấy toàn bộ n đĩa đã được chuyển từ cọc A sang cọc C và không vi phạm bất cứ điều kiện nào của bài toán.



Phân tích bài toán CC

- T/C Chia để trị: Chuyển hàm với dl lớn (bài toán lớn) thành hàm với dl nhỏ hơn (bài toán nhỏ hơn),..., cứ tiếp tục như thế cho đến khi gặp ĐK dừng
 - Chỉ ra việc đệ quy: Bài toán chuyển n cọc đã được chuyển về bài toán đơn giản hơn là chuyển $n-1$ cọc.
 - Điểm dừng của thuật toán đệ quy là khi $n=1$ và ta chuyển thẳng cọc này từ cọc ban đầu sang cọc đích.



Thiết kế một số giải thuật đệ quy

- Giải thuật đệ quy CHIA ĐỂ TRỊ - Bài toán tháp Hà Nội

```
void chuyen(int n, char a, char c){  
    printf("Chuyen dia thu %d tu coc %c sang coc %c \n",n,a,c);  
    return;  
}  
void thaphanoi(int n, char a, char c, char b){  
    if (n==1) chuyen(1, a, c);  
    else{  
        thaphanoi(n-1, a, b, c);  
        chuyen(n, a, c);  
        thaphanoi(n-1, b, c,a);  
    }  
    return;  
}
```

Cho biet so dia can chuyen: 4

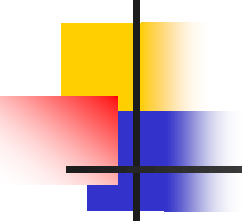
Cac buoc chuyen nhu sau:

```
Chuyen dia thu 1 tu coc a sang coc b
Chuyen dia thu 2 tu coc a sang coc c
Chuyen dia thu 1 tu coc b sang coc c
Chuyen dia thu 3 tu coc a sang coc b
Chuyen dia thu 1 tu coc c sang coc a
Chuyen dia thu 2 tu coc c sang coc b
Chuyen dia thu 1 tu coc a sang coc b
Chuyen dia thu 4 tu coc a sang coc c
Chuyen dia thu 1 tu coc b sang coc c
Chuyen dia thu 2 tu coc b sang coc a
Chuyen dia thu 1 tu coc c sang coc a
Chuyen dia thu 3 tu coc b sang coc c
Chuyen dia thu 1 tu coc a sang coc b
Chuyen dia thu 2 tu coc a sang coc c
Chuyen dia thu 1 tu coc b sang coc c
```



Thuật toán quay lui

- Xây dựng dần các thành phần của cấu hình bằng cách thử tất cả các khả năng. Giả sử cần phải tìm một cấu hình của bài toán $x = (x_1, x_2, \dots, x_n)$ mà $i-1$ thành phần x_1, x_2, \dots, x_{i-1} đã được xác định.
- Bây giờ ta xác định thành phần thứ i của cấu hình bằng cách duyệt tất cả các khả năng có thể có và đánh số các khả năng từ $1 \dots n_i$.
- Với mỗi khả năng j , kiểm tra xem j có chấp nhận được hay không. Khi đó có thể xảy ra hai trường hợp:

- 
- Nếu chấp nhận j thì xác định x_i theo j , nếu $i=n$ thì ta được một cấu hình cần tìm, ngược lại xác định tiếp thành phần x_{i+1} .
 - Nếu thử tất cả các khả năng mà không có khả năng nào được chấp nhận thì quay lại bước trước đó để xác định lại x_{i-1} .

Phân tích đặc điểm của bài toán quay lui

- Toàn bộ vấn đề được thực hiện dần từng bước. Tại mỗi bước có ghi lại kết quả để sau này có thể quay lại và hủy kết quả đó nếu phát hiện ra rằng hướng giải quyết theo bước đó đi vào ngõ cụt và không đem lại giải pháp tổng thể cho vấn đề.

=> Do đó, thuật toán được gọi là *thuật toán quay lui*

- Độ quy: Bước thứ sau làm giống hết bước trước, nhưng dl khác
- Điều kiện dừng: Sau khi duyệt tất cả các khả năng có thể có của dữ liệu ở mỗi bước
- Quay lui, nếu hướng giải quyết vào ngõ cụt (xóa vết hoặc không xóa vết)



Thuật toán

```
■ void Try( int i ) {  
■   int   j;  
■   for ( j = <Khả năng 1>; j < ni; j ++ ) {  
■     if ( <Chấp nhận j > ) {  
■       <Xác định xi theo j>  
■       if (i==n)  
■         <Ghi nhận cấu hình>;  
■       else      Try(i+1);  
■       <Quay lại cấu hình phía trước>}  
  
■ }
```



Thuật toán sinh xâu nhị phân(sử dụng quay lui)

```
■ Void Try ( int i ) {  
■     for (int j =0; j<=1; j++){  
■         X[i] = j;  
■         if ( i ==n) Result();  
■         else Try (i+1);  
■     }  
■ }
```





Hoán vị

```
■ Void Try ( int i ) {  
■     for (int j =1; j<=N; j++){  
■         if (chuaxet[j] ) {  
■             X[i] = j;chuaxet[j] = 0;  
■             if ( i ==N) Result();  
■             else Try (i+1);  
■             Chuaxet[j] = 1;  
■         }  
■     }  
■ }  
■ }
```

Danh sách 8 vị trí bước đi kế tiếp : $(x+2, y-1)$; $(x+1, y-2)$; $(x-1, y-2)$; $(x-2, y-1)$; $(x-2, y+1)$; $(x-1, y+2)$; $(x+1, y+2)$; $(x+2, y+1)$

Mã đi tuần

	3		2	
4				1
				
5				8
	6		7	

$\text{Banco}[x][y] = 0$: ô (x,y) chưa được quân mã đi qua

$\text{Banco}[x][y] = i$: ô (x,y) đã được quân mã đi qua tại nước thứ i .

Bước đi kế tiếp (u,v)

$$u = x + dx[i]$$

$$v = y + dy[i]$$

(u,v) chấp nhận được nếu:

+ $\text{Banco}[u][v] = 0$ - ĐK 1 để chấp nhận.

+ $0 \leq u, v < n$ - đk 2 để chấp nhận

ĐK dừng : Kiểm tra xem bàn cờ còn ô trống không bằng cách kiểm tra xem i đã bằng n^2 chưa và đã đi hết các nước đi chưa (nhiều nhất là 8)

Bài toán Mã đi tuần-Thuật toán quay lui

```
■ void ThuNuocTiepTheo;  
  ■ {  
  ■     Khởi tạo danh sách các nước đi kế tiếp;  
  ■     do{  
  ■         Lựa chọn 1 nước đi kế tiếp từ danh sách;  
  ■         if Chấp nhận được  
  ■         {  
  ■             Ghi lại nước đi;  
  ■             if Bàn cờ còn ô trống  
  ■             {  
  ■                 ThuNuocTiepTheo;  
  ■                 if Nước đi không thành công  
  ■                     Hủy bỏ nước đi đã lưu ở bước  
  ■                 trước  
  ■             }  
  ■         }  
  ■     }  
  ■ }while (nước đi không thành công) && (vẫn còn nước đi)  
  ■ }
```

■ *void ThuNuocTiepTheo(int i, int x, int y, int *q)*

■ {

■ *int k, u, v, *q1;*

■ *k=0;*

■ *do{*

■ **q1=0;*

■ *u=x+dx[k];*

■ *v=y+dy[k];*

■ *if ((0 <= u) && (u<n) && (0 <= v) && (v<n) &&*

■ *(Banco[u][v]==0))*

■ {

■ *Banco[u][v]=i;*

■ *if (i<n*n) {*

■ *ThuNuocTiepTheo(i+1, u, v, q1)*

■ *if (*q1==0) Banco[u][v]=0; }*

■ *else *q1=1;*

■ }

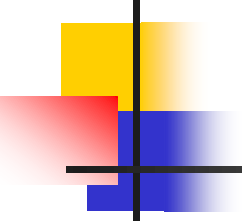
■ *k=k+1;*

■ *}while ((*q1==0) && (k<8));*

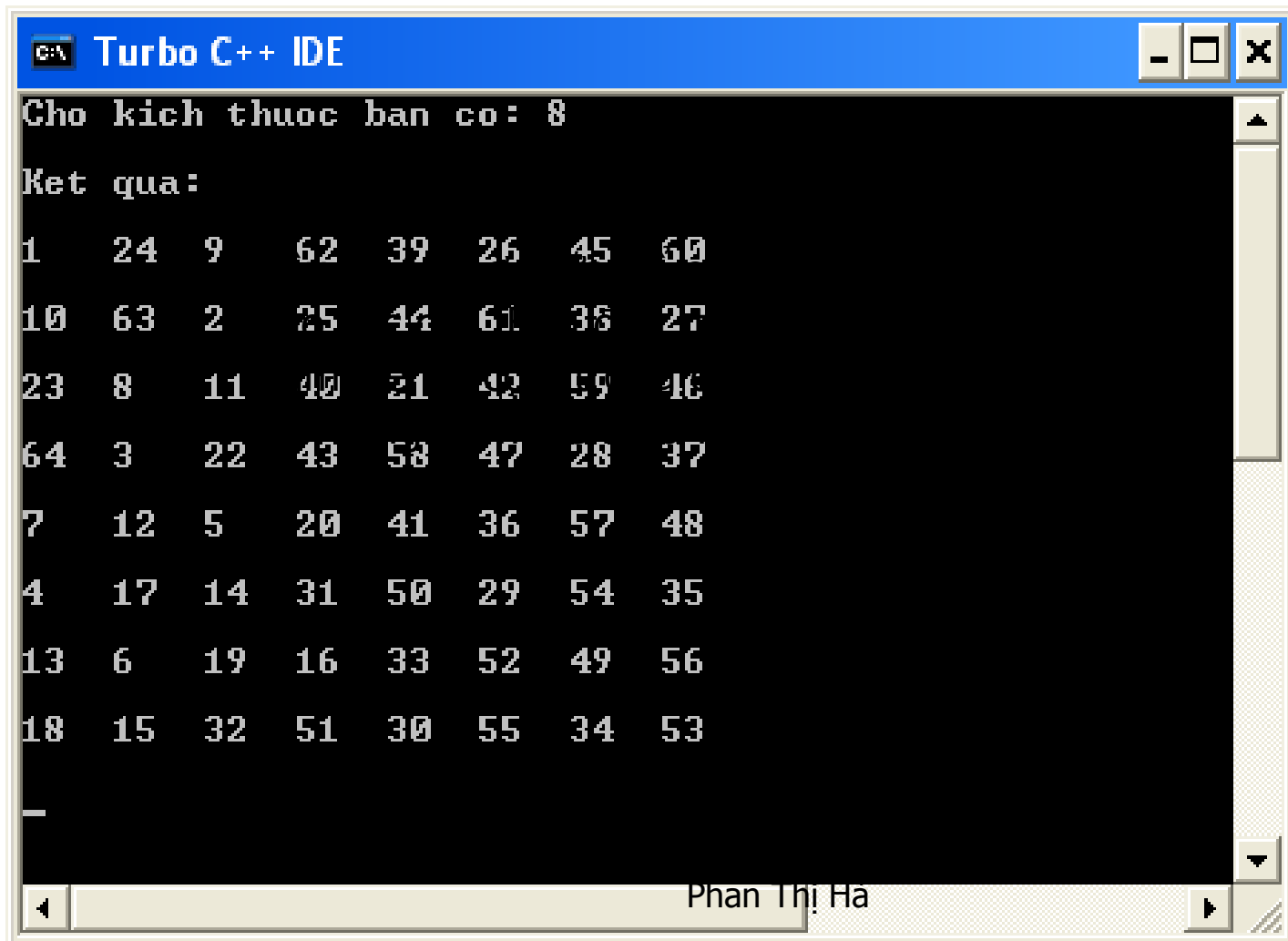
■ **q=*q1;*

■ Phan Thị Hà

■ }



```
int dx[8]={2,1,-1,-2,-2,-1,1,2};  
int dy[8]={-1,-2,-2,-1,1,2,2,1};  
int n=8;
```



Turbo C++ IDE

Cho kích thước bàn cờ: 8

Kết quả:

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

Phan Thị Hà

Thiết kế một số giải thuật đệ quy

Bài toán 8 quân hậu

```
void DatHau(int i)
{
    Khởi tạo danh sách các vị trí có thể đặt quân hậu tiếp theo;
    do{
        Lựa chọn vị trí đặt quân hậu tiếp theo;
        if Vị trí đặt là an toàn
        {
            Đặt hậu;
            if i<8
            {
                DatHau(i+1);
                if Không thành công
                    Bỏ hậu đã đặt ra khỏi vị trí
            }
        }
    }while (Không thành công) && (Vẫn còn lựa chọn)
}
```



CHƯƠNG 3: **MẢNG VÀ DANH SÁCH LIÊN KẾT**

Mảng

- Một mảng là 1 tập hợp cố định các thành phần có cùng 1 kiểu dữ liệu, được lưu trữ kế tiếp nhau và có thể được truy cập thông qua một chỉ số.
- Ví dụ, để truy cập tới phần tử thứ i của mảng a , ta viết $a[i]$.

a_1	a_2	...	a_i	a_{i+1}	...	a_n
-------	-------	-----	-------	-----------	-----	-------

- Mảng có thể có nhiều hơn 1 chiều. Khi đó, số các chỉ số của mảng sẽ tương ứng với số chiều.
- Chẳng hạn, trong mảng 2 chiều a , thành phần thuộc cột i , hàng j được viết là $a[i][j]$. Mảng 2 chiều còn được gọi là ma trận (matrix).

a_{11}	a_{21}	...	a_{i1}	a_{i+11}	...	a_{m1}
a_{12}	a_{22}	...	a_{i2}	a_{i+12}	...	a_{m2}
...
a_{1j}	a_{2j}	...	a_{ij}	a_{i+1j}	...	a_{mj}
a_{1j+1}	a_{2j+1}	...	a_{ij+1}	a_{i+1j+1}	...	a_{mj+1}
...
a_{1n}	a_{2n}	...	a_{in}	a_{i+1n}	...	a_{mn}



Nhược:

- Kích thước (số phần tử) của mảng là cố định



Ôn lại thao tác với mảng

- Khai báo mảng
 - Mảng tĩnh
 - Mảng động
- Truy nhập 1 phần tử của mảng

Ôn lại CẤP PHÁT BỘ NHỚ ĐỘNG

- Cấp phát bộ nhớ động
<tên con trỏ> = new <kiểu
con trỏ>;

- VD:
int *pa;
pa = new int;

- Hoặc
int *pa;
pa = new int(12);

- Giải phóng bộ nhớ động
delete <tên con trỏ>;

VD:

```
int *pa = new int(12);  
delete pa;
```

VD:

- int *pa = new int(12);
- int *pb = pa;
- *pb += 5;
- delete pa;

- Cấp phát bộ nhớ cho mảng động một chiều
<Tên con trỏ> = new <Kiểu con trỏ>[<Độ dài mảng>];

VD: `int *A = new int[5];`

- Giải phóng bộ nhớ của mảng động một chiều
`delete [] <tên con trỏ>;`

VD:

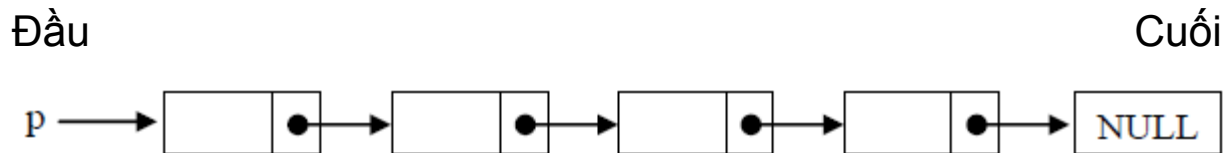
```
int *A = new int[5];  
delete [] A;
```

Danh sách liên kết

■ Khái niệm

- Danh sách liên kết là 1 cấu trúc dữ liệu bao gồm 1 tập các phần tử, trong đó mỗi phần tử là 1 phần của 1 nút có chứa một liên kết tới nút kế tiếp.

■ Biểu diễn



■ Điểm khác biệt giữa DSLK và Mảng

- Mảng có thể được truy cập ngẫu nhiên thông qua chỉ số, còn danh sách chỉ có thể truy cập tuần tự.
- Việc bố trí, sắp đặt lại các phần tử trong 1 danh sách liên kết đơn giản hơn nhiều so với mảng.
- Do bản chất động của danh sách liên kết, kích thước của danh sách liên kết có thể linh hoạt hơn nhiều so với mảng.



Khai báo danh sách trong C,C++

- Đơn giản

```
struct node {  
    int item;  
    struct node *next;  
};  
typedef struct node *listnode;
```

- Phức tạp

```
struct node {  
    itemstruct item;  
    struct node *next;  
};  
typedef struct node *listnode;
```



Các thao tác trên DSLK

- Tập các thao tác:
 - *Tạo, cấp phát, và giải phóng bộ nhớ cho 1 nút*
 - *Chèn một nút vào đầu danh sách*
 - *Chèn một nút vào cuối danh sách*
 - *Chèn một nút vào trước nút r trong danh sách*
 - *Xóa một nút ở đầu danh sách*
 - *Xóa một nút ở cuối danh sách*
 - *Xóa một nút ở trước nút r trong danh sách*
 - *Duyệt toàn bộ danh sách*
- *Cài đặt các thao tác: SGK*



lạo, cấp phát, và giải phóng bộ nhớ cho 1 nút

- `listnode p; // Khai báo biến p`
- `p = new (node); // cấp phát bộ nhớ cho p`
- `delete(p); // giải phóng bộ nhớ đã cấp phát cho nút p;`

Chèn thêm 1 nút vào đầu danh sách

Giả sử ta có 1 danh sách mà đầu của danh sách được trỏ tới bởi con trỏ p.



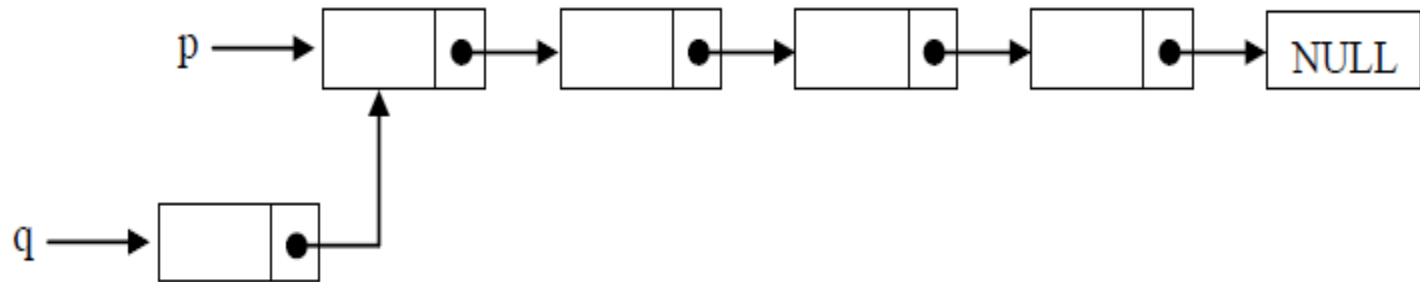
Các bước để chèn 1 nút mới vào đầu danh sách như sau:

- Tạo và cấp phát bộ nhớ cho 1 nút mới. Nút này được trỏ tới bởi q.

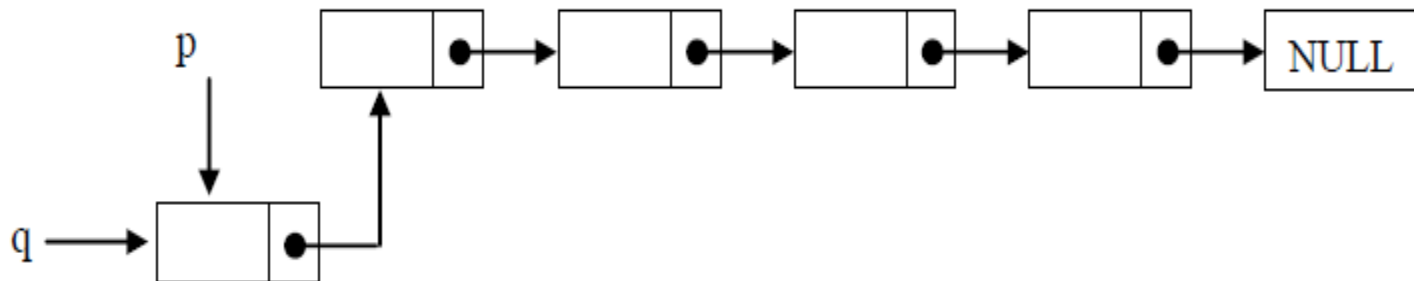


- Sau khi gán các giá trị thích hợp cho phần tử của nút mới, cho con trỏ tiếp của nút mới trỏ đến phần tử đầu tiên của nút.

- Sau khi gán các giá trị thích hợp cho phần tử của nút mới, cho con trỏ tiếp của nút mới trỏ đến phần tử đầu tiên của nút.



- Cuối cùng, để p vẫn trỏ đến nút đầu danh sách, ta cần cho p trỏ đến nút mới tạo.



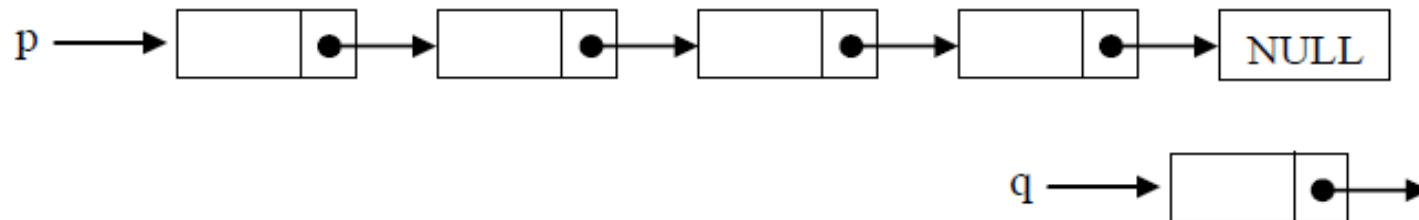
Chen them 1 nut vao dau danh sách

```
void Insert_Begin(listnode *p, int x){  
    listnode q;  
    q = new node  
    q->item = x;  
    q->next = *p;  
    *p = q;  
}
```

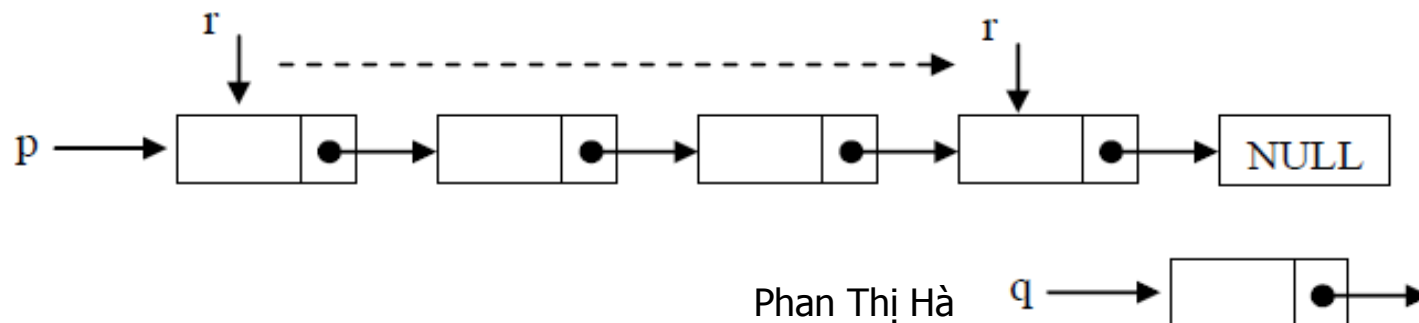
Chèn thêm 1 nút vào cuối ds

Các bước để chèn 1 nút mới vào cuối danh sách như sau (thực hiện đúng theo trình tự):

- Tạo và cấp phát bộ nhớ cho 1 nút mới. Nút này được trỏ tới bởi q.



- Dịch chuyển con trỏ tới nút cuối của danh sách. Để làm được việc này, ta phải khai báo 1 biến con trỏ mới r. Ban đầu, biến này, cũng với p, trỏ đến đầu danh sách. Lần lượt dịch chuyển r theo các nút kế tiếp cho tới khi đến cuối danh sách.





Chèn một nút vào cuối danh sách

```
void Insert_End(listnode *p, int x){
    listnode q, r;
    q = new node;
    q->item = x;
    q->next = NULL;
    if (*p==NULL) *p = q;
    else{
        r = *p;
        while (r->next != NULL) r = r->next;
        r->next = q;
    }
}
```

Chen một nút vào trước nút r trong danh sách

```
void insert_Middle(listnode *p, int position, int x){
```

```
int count=1, found=0;
```

```
listnode q, r;
```

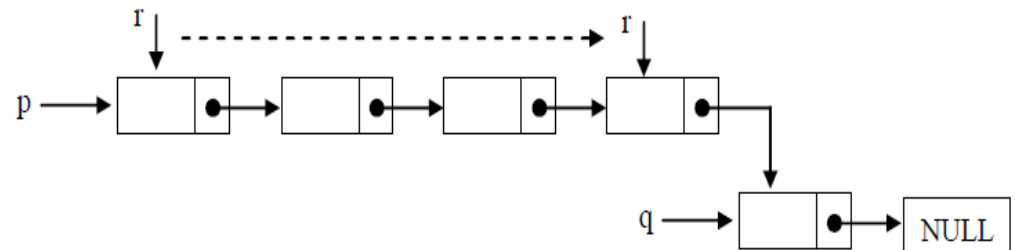
```
r = *p;
```

```
while ((r != NULL)&&(found==0))
```

```
{
```

```
    if (count == position){
```

- Cho con trỏ tiếp của nút cuối (được trỏ tới bởi r) trỏ đến nút mới tạo là q, và cho con trỏ tiếp của q trỏ tới null.



```
        q-> next = r-> next;
```

```
        r-> next = q;
```

```
        found = 1;
```

```
    }
```

```
    count ++;
```

```
    r = r-> next;
```

```
}
```

```
if (found==0)
```

```
    cout("Khong tim thay vi tri can chen !");
```

```
}
```

Phan Thị Hà

Xóa một nút ở đầu danh sách

```
void Remove_Begin(listnode *p){
```

```
listnode q;
```

```
if (*p == NULL) return;
```

```
q = *p;
```

```
*p = (*p)->next;
```

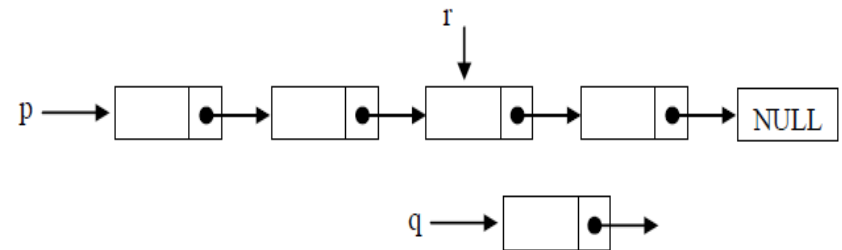
```
q->next = NULL;
```

```
free(q);
```

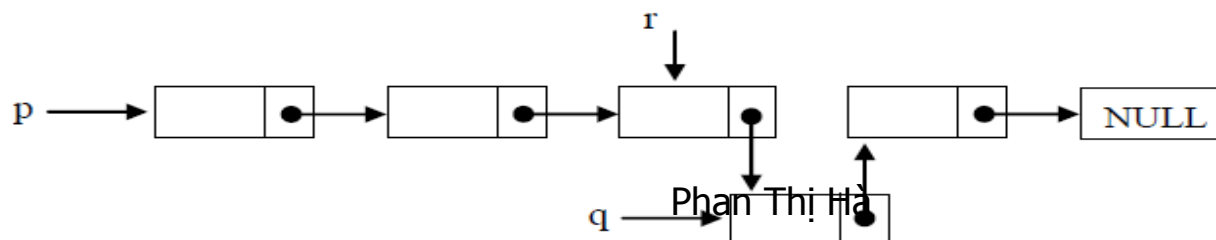
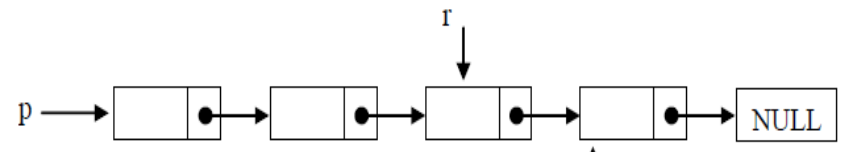
```
}
```

- Cho con trỏ tiếp của nút r trỏ đến q.

- Tạo và cấp phát bộ nhớ cho 1 nút mới. Nút này được trỏ tới bởi q.



- Cho con trỏ tiếp của nút mới trỏ đến nút kế tiếp của nút r.



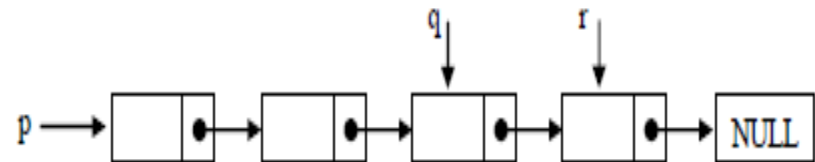
Xóa nút ở cuối danh sách

```
void Remove_End(listnode *p){
    listnode q, r;
    if (*p == NULL) return;
    if ((*p)-> next == NULL){
        Remove_Begin(*p);
        return;
    }
    r = *p;
    while (r-> next != NULL){
        q = r;
        r = r-> next;
    }
    q-> next = NULL;
    free(r);
}
```

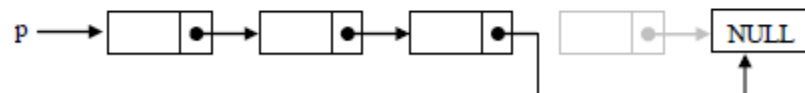


Các bước để xóa 1 nút ở cuối danh sách như sau:

- Dịch chuyển con trỏ tới nút gần nút cuối của danh sách. Để làm được việc này, ta phải dùng 2 biến tạm là q và r. Lần lượt dịch chuyển q và r từ đầu danh sách tới cuối danh sách, trong đó q luôn dịch chuyển sau r 1 nút. Khi r tới nút cuối cùng thì q là nút gần nút cuối cùng nhất.

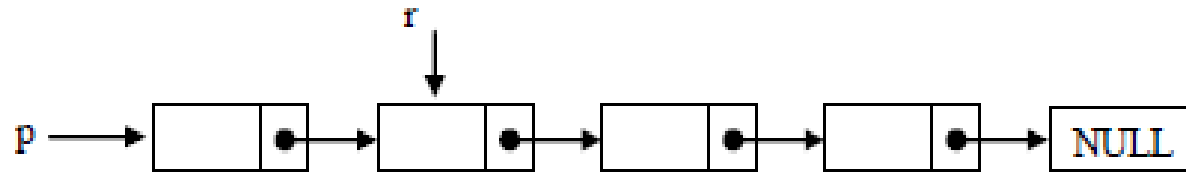


- Cho con trỏ tiếp của nút gần nút cuối cùng nhất (đang được trỏ bởi q) trỏ tới null. Giải phóng bộ nhớ cho nút cuối cùng (đang được trỏ bởi r).



Xóa đi 1 nút trước nút r trong ds

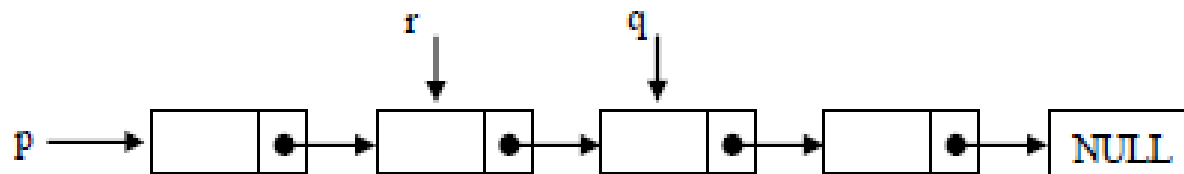
Giả sử ta có 1 danh sách mà đầu của danh sách được trỏ tới bởi con trỏ p, và 1 nút r trong danh sách.



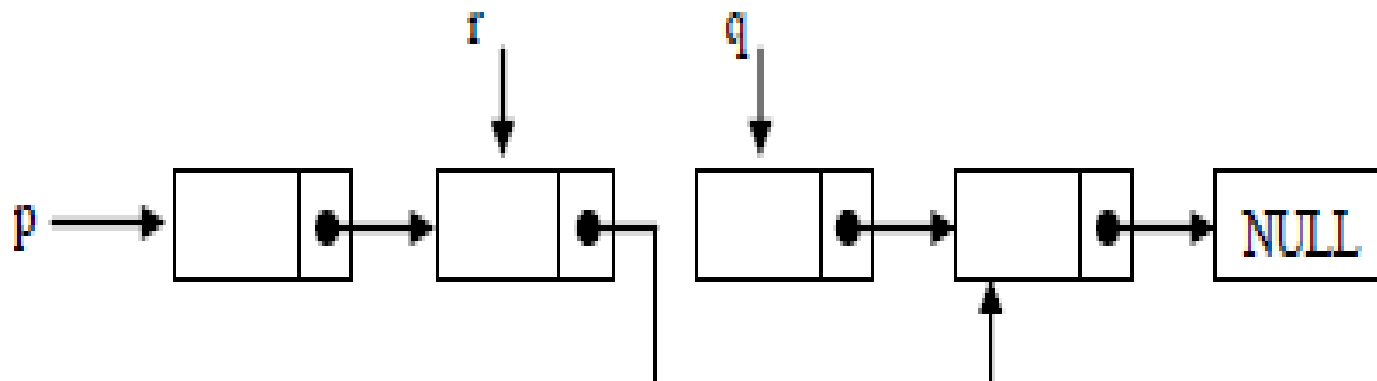
Ta giả thiết rằng nút r không phải là nút cuối cùng của danh sách, vì nếu như vậy thì sẽ không có nút đứng trước nút r.

Các bước để xóa 1 nút ở trước nút r trong danh sách như sau:

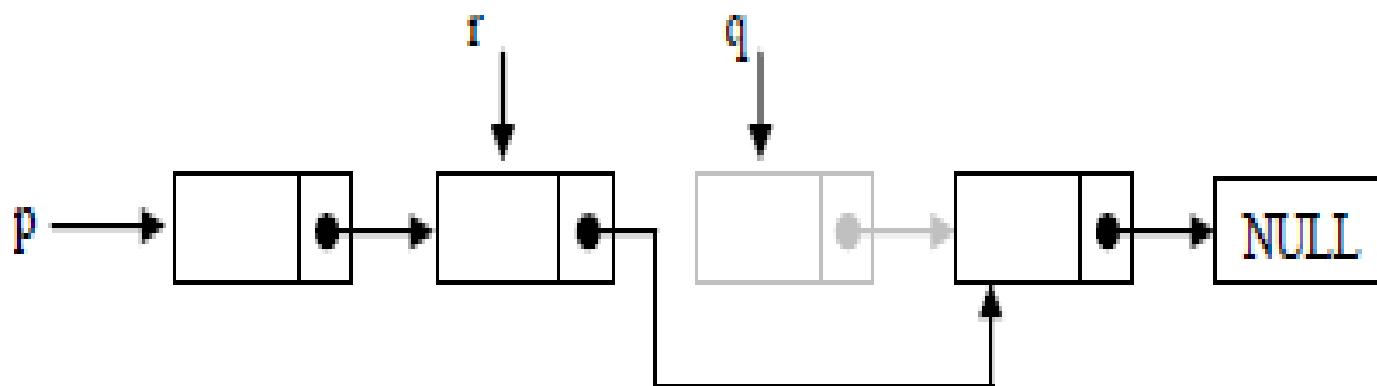
- Sử dụng 1 biến tạm q trỏ đến nút đứng trước nút r.



- Cho con trỏ tiếp của nút r trỏ tới nút đứng sau nút q.



- Ngắt liên kết của nút q và giải phóng bộ nhớ cho q .



Xoá một nút ở trước nút r trong danh sách

```
void Remove_Middle(listnode *p, int position){
    int count=1, found=0;
    listnode q, r;
    r = *p;
    while ((r != NULL)&&(found==0)){
        if (count == position){
            q = r-> next;
            r-> next = q-> next;
            q-> next = NULL;
            free (q);
            found = 1;
        }
        count ++;
        r = r-> next;
    }
    if (found==0)
        cout << "Khong tim thay vi tri can xoa !";
}
```

Duyệt toàn bộ danh sách

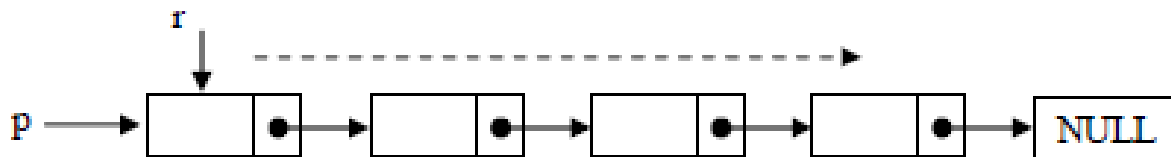
```
r = p;
```

```
while (r->next != null){
```

```
    //thực hiện các thao tác cần thiết
```

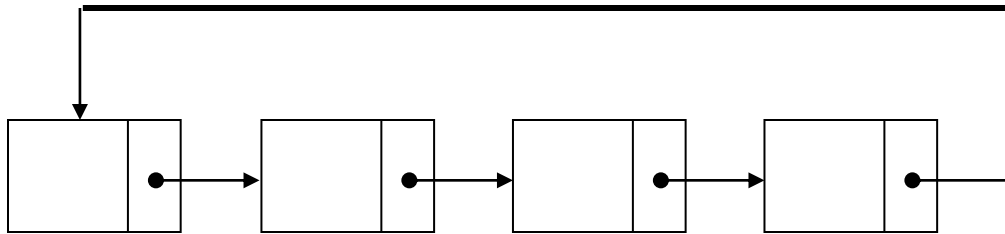
```
    r = r->next;
```

```
}
```

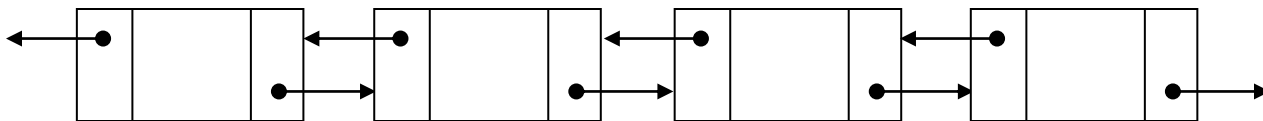


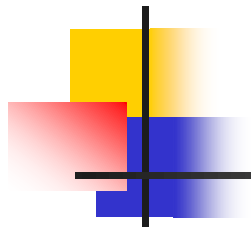
Một số dạng khác của DSLK

- Danh sách liên kết vòng



- Danh sách liên kết kép





CHƯƠNG 4

NGĂN XẾP VÀ HÀNG ĐỢI



Ngăn xếp (Stack)

■ Khái niệm

- Ngăn xếp là một dạng đặc biệt của danh sách mà việc bổ sung hay loại bỏ một phần tử đều được thực hiện ở 1 đầu của danh sách gọi là đỉnh.
- Nói cách khác, ngăn xếp là 1 cấu trúc dữ liệu có 2 thao tác cơ bản:
 - bổ sung (push) và
 - loại bỏ phần tử (pop),
- Trong đó việc loại bỏ sẽ tiến hành loại phần tử mới nhất được đưa vào danh sách.
- Chính vì tính chất này mà ngăn xếp còn được gọi là kiểu dữ liệu có nguyên tắc LIFO (Last In First Out - Vào sau ra trước).

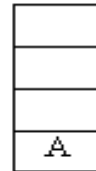
Stack

■ Ví dụ

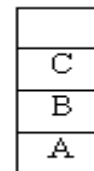
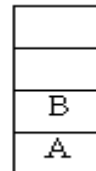
Giả sử ta có một stack S lưu trữ các kí tự. Ban đầu, ngăn xếp ở trạng thái rỗng:



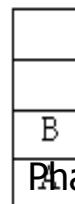
Khi thực hiện lệnh bổ xung phần tử A , $\text{push}(S, A)$, ngăn xếp có dạng:



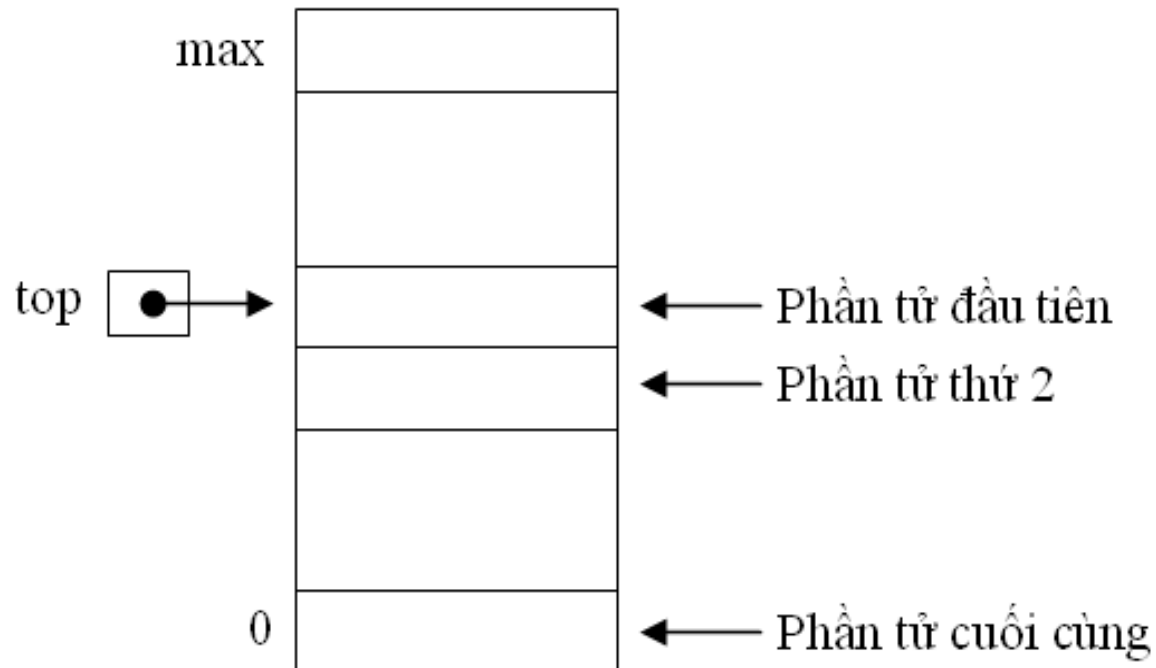
Tiếp theo là các lệnh $\text{push}(S, B)$, $\text{push}(S, C)$:



Lệnh $\text{pop}(S)$ sẽ loại bỏ phần tử mới nhất được đưa vào ra khỏi ngăn xếp, đó là C :



Cài Stack xếp bằng mảng





Khai báo bằng mảng cho 1 ngăn xếp chứa các số nguyên với tối đa 100 phần tử

- `#define MAX 100`
- `typedef struct {`
- `int top;`
- `int nut[MAX];`
- `} stack;`



Cài đặt Stack bằng mảng

Thao tác khởi tạo ngăn xếp

```
void StackInitialize(stack *s){  
    s-> top = -1;  
    return;  
}
```

Thao tác kiểm tra ngăn xếp rỗng

```
int StackEmpty(stack s){  
    return (s.top == -1);  
}
```

Thao tác kiểm tra ngăn xếp đầy

```
int StackFull(stack s){  
    return (s.top == MAX-1);  
}
```



Cài đặt Stack bằng mảng

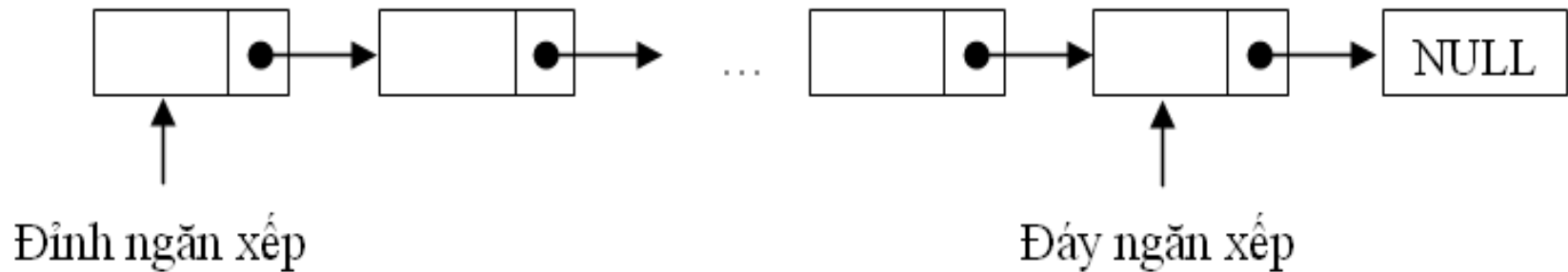
Thao tác bổ sung 1 phần tử vào ngăn xếp

```
void Push(stack *s, int x){
    if (StackFull(*s)){
        printf("Ngan xep day !");
        return;
    }else{
        s-> top ++;
        s-> nut[s-> top] = x;
        return;
    }
}
```

Thao tác lấy 1 phần tử ra khỏi ngăn xếp

```
int Pop(stack *s){
    if (StackEmpty(*s)){
        printf("Ngan xep rong !");
    }else{
        return s-> nut[s-> top--];
    }
}
```

Cài đặt Stack bằng DSLK



Khai báo 1 ngăn xếp bằng danh sách liên kết như sau:

```
struct node {  
    int item;  
    struct node *next;  
};  
typedef struct node *stacknode;  
typedef struct {  
    stacknode top;  
}stack;
```



Cài đặt Stack bằng DSLK

- **Thao tác khởi tạo ngăn xếp**

```
void StackInitialize(stack *s){  
    s-> top = NULL;  
    return;  
}
```

- **Thao tác kiểm tra ngăn xếp rỗng**

```
int StackEmpty(stack s){  
    return (s.top == NULL);  
}
```



Cài đặt Stack bằng DSLK

- **Thao tác bổ sung 1 phần tử vào ngăn xếp**

```
void Push(stack *s, int x){  
    stacknode p;  
    p = new node;  
    p->item = x;  
    p->next = s->top;  
    s->top = p;  
    return;  
}
```



Cài đặt Stack bằng DSLK

■ Thao tác lấy 1 phần tử ra khỏi ngăn xếp

```
int Pop(stack *s){  
    stacknode p;  
    if (StackEmpty(*s)){  
        cout<<"Ngan xep rong !";  
    }else{  
        p = s-> top;  
        s-> top = s-> top-> next;  
        return p->item;  
    }  
}
```




Một số ứng dụng của Stack

- Một số ví dụ:
 - Đảo ngược chuỗi ký tự.
 - Tính giá trị một biểu thức dạng hậu tố (postfix).
 - Chuyển một biểu thức dạng trung tố sang hậu tố (infix to postfix).
- Cài đặt: SGK

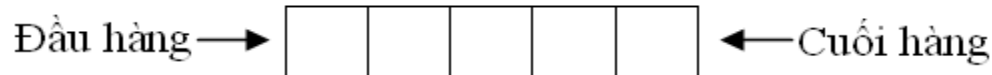


Hàng đợi

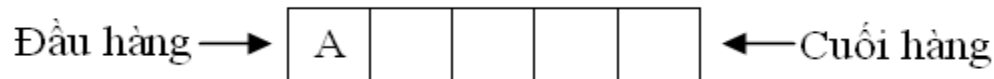
■ Khái niệm

- Hàng đợi là một cấu trúc dữ liệu gần giống với ngăn xếp, nhưng khác với ngăn xếp ở nguyên tắc chọn phần tử cần lấy ra khỏi tập phần tử. Trái ngược với ngăn xếp, phần tử được lấy ra khỏi hàng đợi không phải là phần tử mới nhất được đưa vào mà là phần tử đã được lưu trong hàng đợi lâu nhất.
- Quy luật này của hàng đợi còn được gọi là Vào trước ra trước (FIFO - First In First Out).

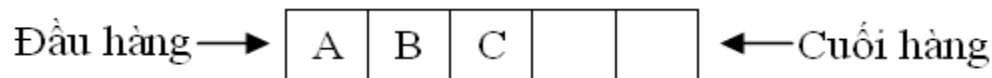
Hàng đợi



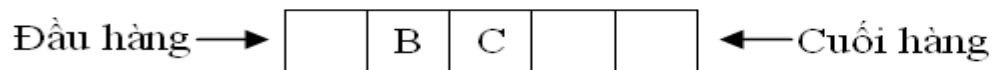
Khi thực hiện lệnh bổ sung phần tử A, $\text{put}(Q, A)$, hàng đợi có dạng:



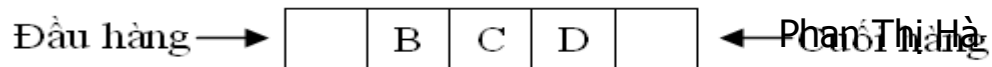
Tiếp theo là các lệnh $\text{put}(Q, B)$, $\text{put}(Q, C)$:



Khi thực hiện lệnh get để lấy ra 1 phần tử từ hàng đợi thì phần tử được lưu trữ lâu nhất trong hàng sẽ được lấy ra. Đó là phần tử đầu tiên ở đầu hàng.



Tiếp theo, thực hiện lệnh $\text{put}(Q, D)$ để bổ sung phần tử D. Phần tử này sẽ được bổ sung ở phía cuối của hàng.



Cài đặt hàng đợi bằng mảng



- Khai báo bằng mảng cho 1 hàng đợi chứa các số nguyên với tối đa 100 phần tử như sau:
 - `#define MAX 100`
 - `typedef struct {`
 - `int head, tail, count;`
 - `int node[MAX];`
 - `} queue;`



Cài đặt hàng đợi bằng mảng

- **Thao tác khởi tạo hàng đợi**
- Thao tác này thực hiện việc gán giá trị 0 cho biến head, giá trị MAX -1 cho biến tail, và giá trị 0 cho biến count, cho biết hàng đợi đang ở trạng thái rỗng.

```
void QueueInitialize(queue *q){  
    q-> head = 0;  
    q-> tail = MAX-1;  
    q-> count = 0;  
    return;  
}
```

- **Thao tác kiểm tra hàng đợi rỗng**
- Hàng đợi rỗng nếu có số phần tử nhỏ hơn hoặc bằng 0.

```
int QueueEmpty(queue q){  
    return (q.count <= 0);  
}
```



Cài đặt hàng đợi bằng mảng

- **Thao tác thêm 1 phần tử vào hàng đợi**

```
void Put(queue *q, int x){  
    if (q->count == MAX)  
        cout<<"Hang doi day !";  
    else{  
        if (q->tail == MAX-1 )  
            q->tail=0;  
        else  
            (q->tail)++;  
        q->node[q->tail]=x;  
        q->count++;  
    }  
    return;  
}
```



Cài đặt hàng đợi bằng mảng

■ Lấy phần tử ra khỏi hàng đợi

```
int Get(queue *q){  
    int x;  
    if (QueueEmpty(*q))  
        cout<<"Hang doi rong !";  
    else{  
        x = q-> node[q-> head];  
        if (q->head == MAX-1 )  
            q->head=0;  
        else  
            (q->head)++;  
        q-> count--;  
    }  
    return x;  
}
```

Cài đặt hàng đợi bằng DSLK



- Khai báo 1 hàng đợi bằng danh sách liên kết như sau:

```
struct node {  
    int item;  
    struct node *next;  
};  
typedef struct node *queuenode;  
typedef struct {  
    queuenode head;  
    queuenode tail;  
}queue;
```




Cài đặt hàng đợi bằng DSLK

- **Thao tác khởi tạo hàng đợi**
- Thao tác này thực hiện việc gán giá trị null cho nút đầu và cuối của hàng đợi, cho biết hàng đợi đang ở trạng thái rỗng.

```
void QueueInitialize(queue *q){  
    q-> head = q-> tail = NULL;  
    return;  
}
```

- **Thao tác kiểm tra hàng đợi rỗng**
- Hàng đợi rỗng nếu nút đầu trỏ đến NULL.

```
int QueueEmpty(queue q){  
    return (q.head == NULL);  
}
```



Cài đặt hàng đợi bằng DSLK

■ Thao tác thêm 1 phần tử vào hàng đợi

```
void Put(queue *q, int x){  
    queuenode p;  
    p = new node;  
    p-> item = x;  
    p-> next = NULL;  
    q-> tail-> next = p;  
    q-> tail = q-> tail-> next;  
    if (q-> head == NULL) q-> head = q-> tail;  
    return;  
}
```



Cài đặt hàng đợi bằng DSLK

■ Lấy phần tử ra khỏi hàng đợi

```
int Get(queue *q){
    queuenode p;
    if (QueueEmpty(*q)){
        printf("Ngan xep rong !");
        return 0;
    }else{
        p = q-> head;
        q-> head = q-> head-> next;
        return p->item;
    }
}
```



CHƯƠNG 5:

CẤU TRÚC DỮ LIỆU KIỂU CÂY



Cây là gì

■ Khái niệm cây

- Cây là một tập hợp các nút (các đỉnh) và các cạnh, thỏa mãn một số yêu cầu nào đó. Mỗi nút của cây đều có 1 định danh và có thể mang thông tin nào đó. Các cạnh dùng để liên kết các nút với nhau. Một đường đi trong cây là một danh sách các đỉnh phân biệt mà đỉnh trước có liên kết với đỉnh sau.
- Một tính chất rất quan trọng hình thành nên cây, đó là có đúng một đường đi nối 2 nút bất kỳ trong cây.

■ Định nghĩa cây dưới dạng đệ quy

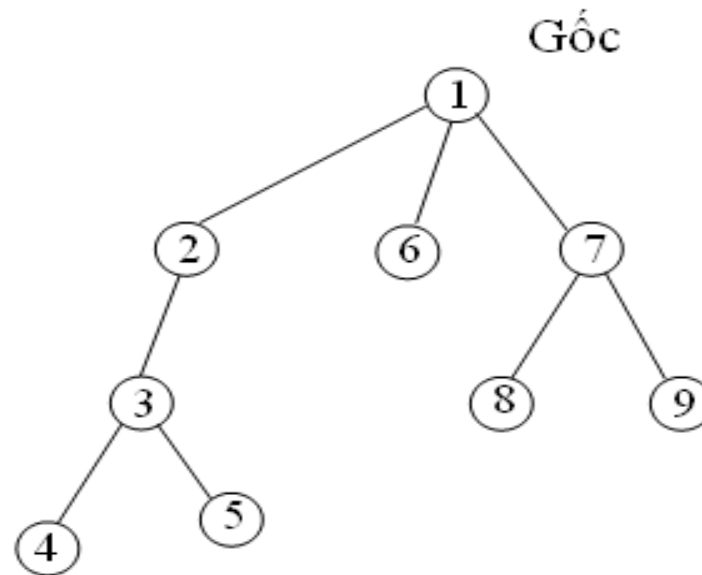
- Một nút đứng riêng lẻ (và nó chính là gốc của cây này).
- Hoặc một nút kết hợp với một số cây con bên dưới.

■ Một số khái niệm khác

- Mỗi nút trong cây (trừ nút gốc) có đúng 1 nút nằm trên nó, gọi là **nút cha (parent)**. Các nút nằm ngay dưới nút đó được gọi là các **nút con (subnode)**. Các nút nằm cùng cấp được gọi là các **nút anh em (sibling)**. Nút không có nút con nào được gọi là **nút lá (leaf)** hoặc **nút tận cùng**.
- **Chiều cao của nút** là đường đi dài nhất từ nút tới một lá. **Chiều cao của cây** chính là chiều cao của nút gốc. **Độ sâu của 1 nút** là độ dài đường đi duy nhất giữa nút gốc và nút đó.

Cài đặt cây

- Cài đặt cây bằng mảng các nút cha

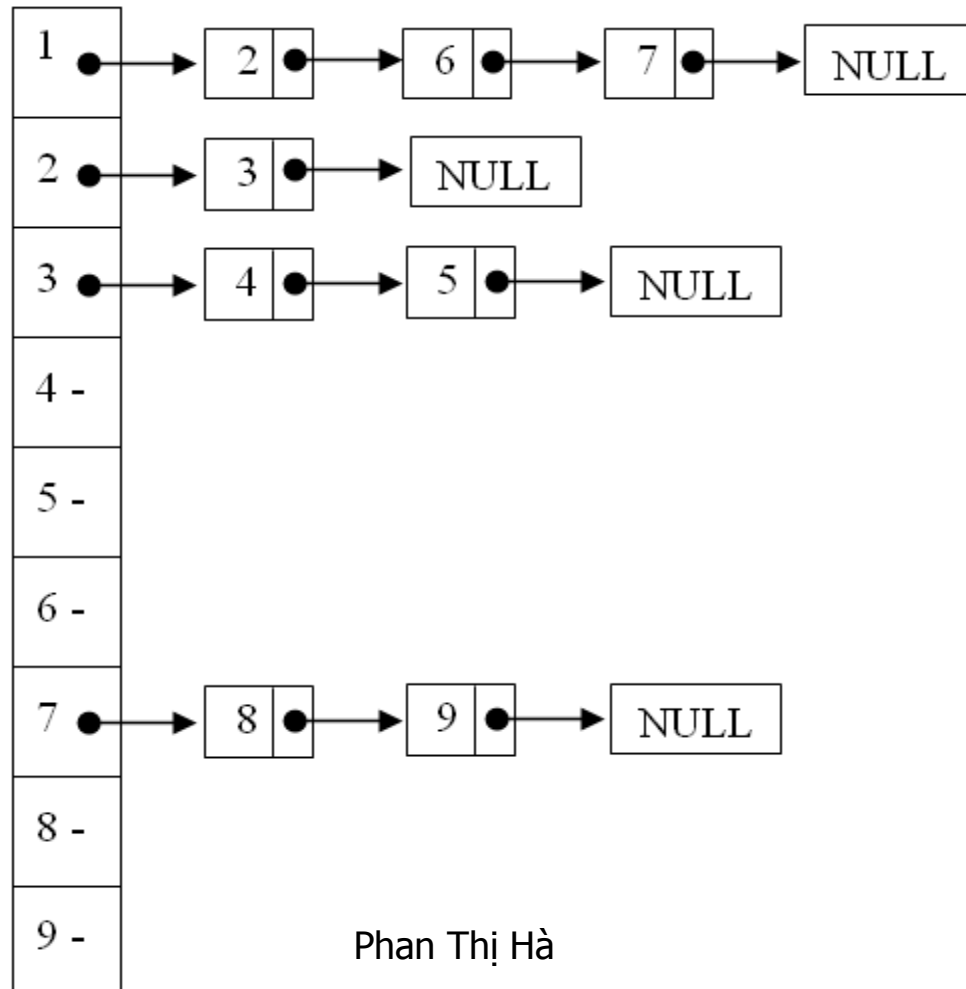


A

1	2	3	4	5	6	7	8	9
0	1	2	3	3	1	1	7	7

Cài đặt cây

- Cài đặt cây thông qua danh sách các nút con





Cài đặt cây

- Mã nguồn C cho cài đặt cây (bằng danh sách node con):

```
#define max = 100;
struct node {
    int item;
    struct node *next;
};
typedef struct node *listnode;
typedef struct {
    int root;
    listnode subnode[max];
} tree;
```




Duyệt cây

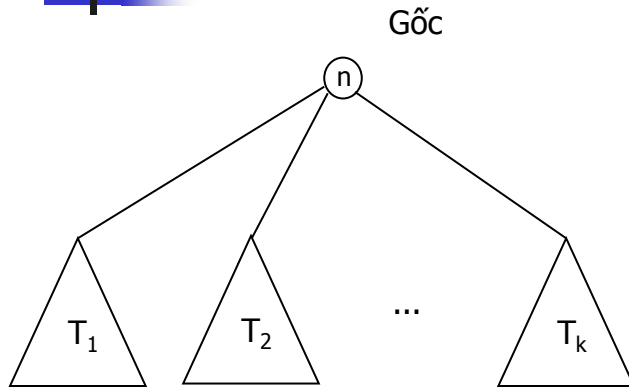
■ Duyệt là gì:

- Duyệt cây là hành động duyệt qua tất cả các nút của một cây theo một trình tự nào đó. Trong quá trình duyệt, tại mỗi nút ta có thể tiến hành một thao tác xử lý nào đó. Đối với các danh sách liên kết, việc duyệt qua danh sách đơn giản là đi từ nút đầu, qua các liên kết và tới nút cuối cùng.

■ Ba trình tự duyệt cây phổ biến:

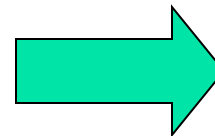
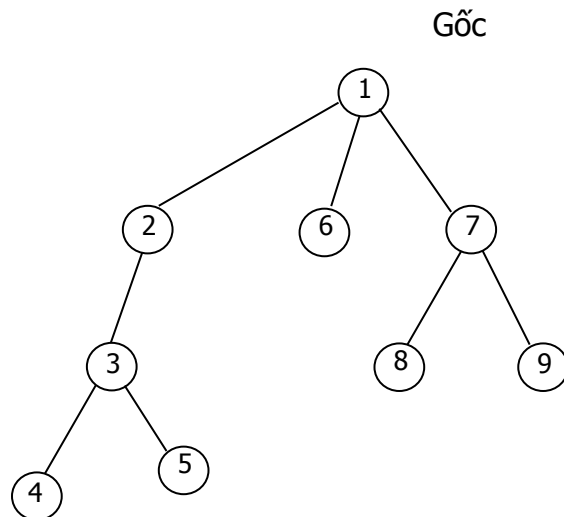
- Duyệt cây theo thứ tự trước.
- Duyệt cây theo thứ tự giữa.
- Duyệt cây theo thứ tự sau.

Duyệt cây theo thứ tự trước



Quá trình duyệt:

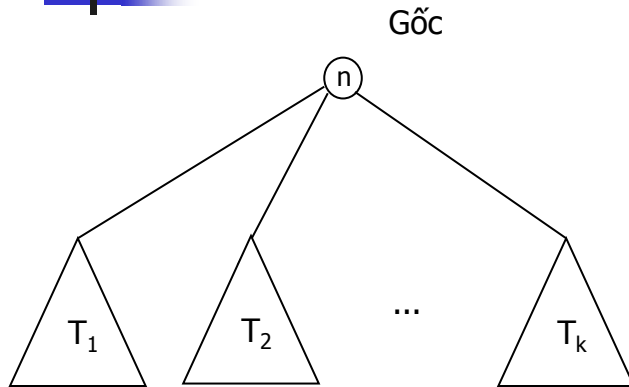
1. Thăm nút gốc n.
2. Thăm cây con T1 theo phương pháp thứ tự trước.
3. Thăm cây con T2 theo phương pháp thứ tự trước.
4. ...
5. Thăm cây con Tk theo phương pháp thứ tự trước.



1 -> 2 -> 3 -> 4 -> 5 -> 6 ->
7 -> 8 -> 9

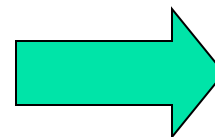
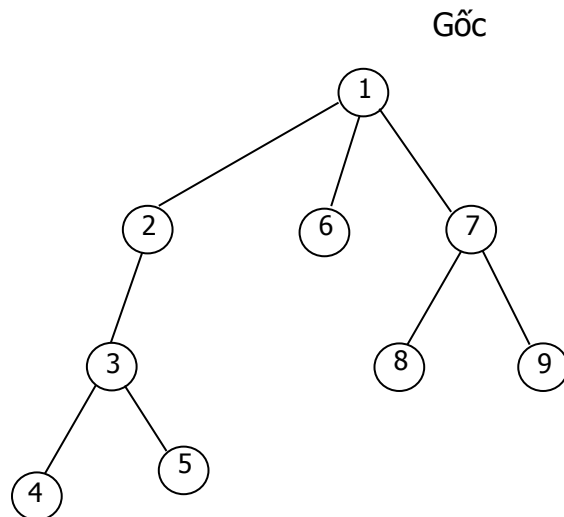
Phan Thị Hà

Duyệt cây theo thứ tự giữa



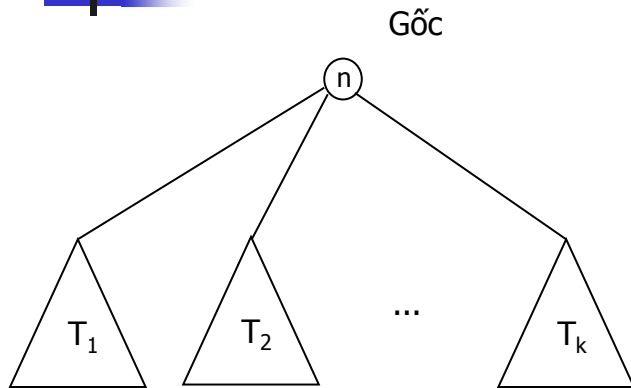
Quá trình duyệt:

1. Thăm cây con T1 theo phương pháp thứ tự giữa.
2. Thăm nút gốc n.
3. Thăm cây con T2 theo phương pháp thứ tự giữa.
4. ...
5. Thăm cây con Tk theo phương pháp thứ tự giữa.



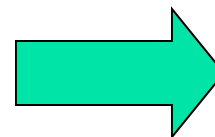
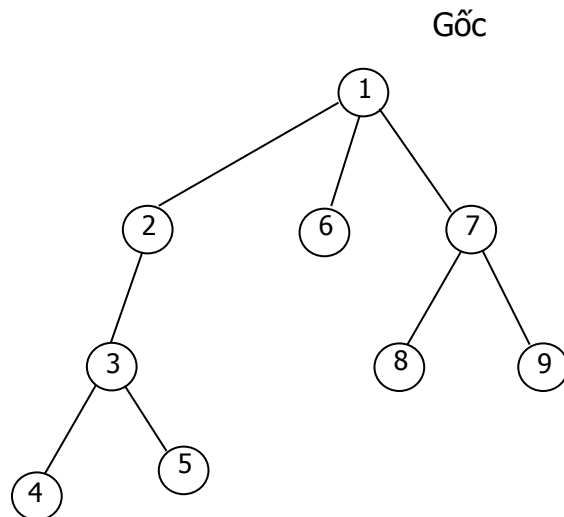
4 -> 3 -> 5 -> 2 -> 1 -> 6
-> 8 -> 7 -> 9

Duyệt cây theo thứ tự sau



Quá trình duyệt:

- Thăm cây con T1 theo phương pháp thứ tự sau.
- Thăm cây con T2 theo phương pháp thứ tự sau.
- ...
- Thăm cây con Tk theo phương pháp thứ tự sau.
- Thăm nút gốc n.



4 -> 5 -> 3 -> 2 -> 6 -> 8
-> 9 -> 7 -> 1

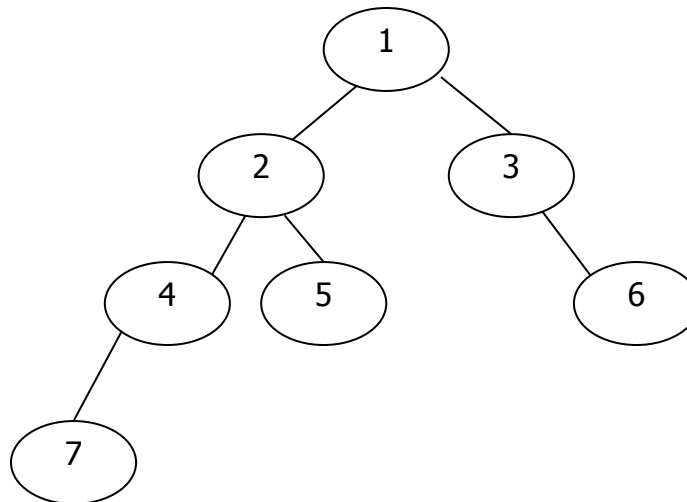
Phan Thị Hà

Cây nhị phân

■ Khái niệm

- Cây nhị phân là một loại cây đặc biệt mà mỗi nút của nó chỉ có nhiều nhất là 2 nút con. Khi đó, 2 cây con của mỗi nút được gọi là cây con trái và cây con phải.

■ Ví dụ

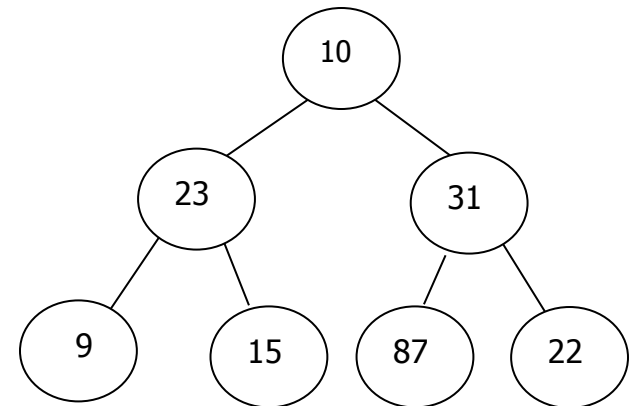


Cây nhị phân

- Một số dạng cây nhị phân đặc biệt:

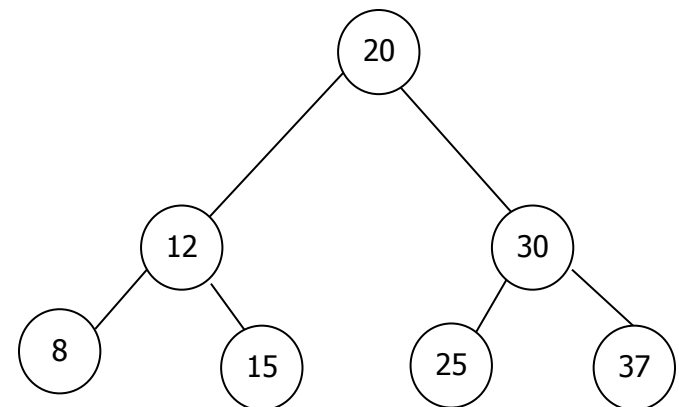
- Cây nhị phân đầy đủ:

- Là cây nhị phân mà mỗi nút không phải lá đều có đúng 2 nút con và các nút lá phải có cùng độ sâu.

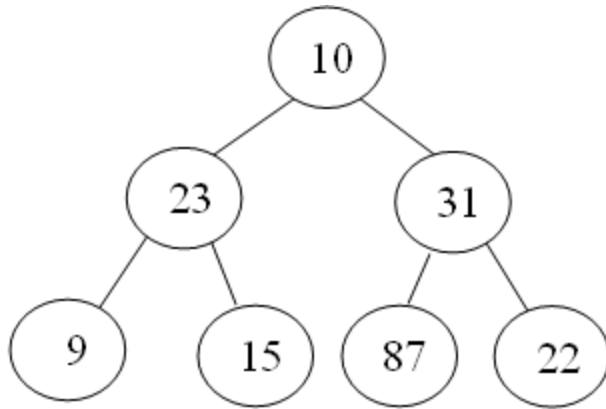


- Cây nhị phân tìm kiếm:

- Là cây nhị phân có tính chất khóa của nút con bên trái bao giờ cũng nhỏ hơn khóa của nút cha, còn khóa của cây con bên phải bao giờ cũng lớn hơn hoặc bằng khóa của nút



Cài đặt cây nhị phân bằng mảng



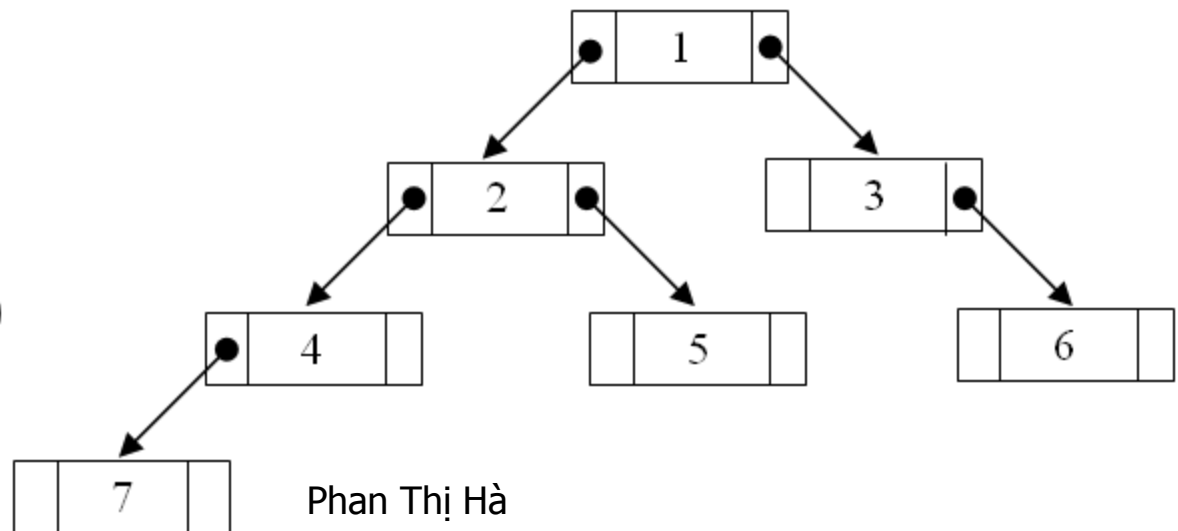
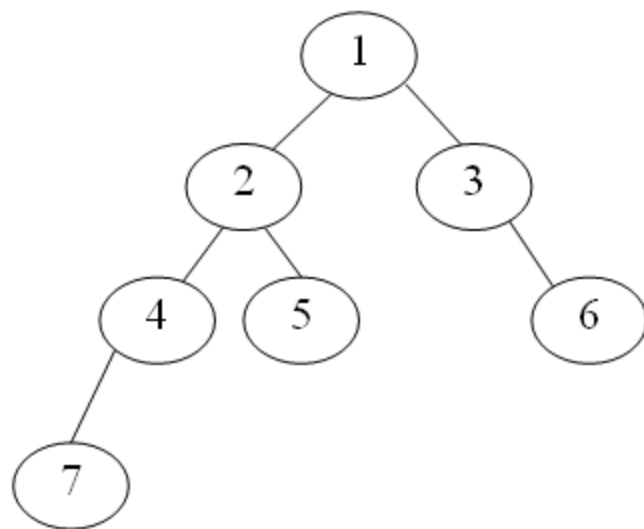
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]
10	23	31	9	15	87	22

- Đối với cây nhị phân không cân bằng, cần sử dụng cấu trúc sau:

```
typedef struct {  
    int item;  
    int leftchild;  
    int rightchild;  
} node;  
node tree[max];
```

Cài đặt cây nhị phân sử dụng DSLK

- Mỗi nút của cây nhị phân khi cài đặt bằng DSLK sẽ có 3 thành phần:
 - Thành phần item chứa thông tin về nút.
 - Con trỏ left trỏ đến nút con bên trái.
 - Con trỏ right trỏ đến nút con bên phải.





Cài đặt cây nhị phân sử dụng DSLK

- Mã nguồn:

```
struct    node {  
    int    item;  
    struct node *left;  
    struct node *right;  
}  
typedef struct node *treenode;  
treenode root;
```



Cài đặt các phương pháp duyệt cây nhị phân

■ Duyệt thứ tự trước

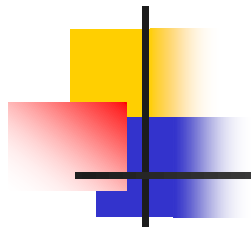
```
void PreOrder (treenode root ) {  
    if (root !=NULL) {  
        printf("%d", root.item);  
        PreOrder(root.left);  
        PreOrder(root.right);  
    }  
}
```

■ Duyệt thứ tự giữa

```
void InOrder (treenode root ) {  
    if (root !=NULL) {  
        PreOrder(root.left);  
        printf("%d", root.item);  
        PreOrder(root.right);  
    }  
}
```

■ Duyệt thứ tự sau

```
void PostOrder (treenode root ) {  
    if (root !=NULL) {  
        PreOrder(root.left);  
        PreOrder(root.right);  
        printf("%d", root.item);  
    }  
}
```



CHƯƠNG 6: ĐỒ THỊ

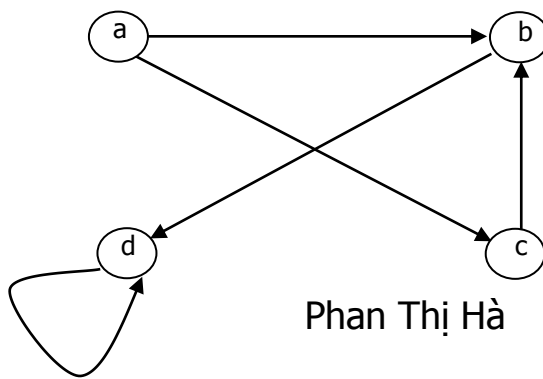
Các khái niệm cơ bản

■ Đồ thị có hướng

- Đồ thị có hướng $G = \langle V, E \rangle$ bao gồm:
 - V là một tập hữu hạn các đỉnh.
 - E là một tập hữu hạn, có thứ tự các cặp đỉnh của V , gọi là các cạnh.

■ Ví dụ:

- Đồ thị có hướng $G1 = \langle V1, E1 \rangle$, với $V1$ và $E1$ được xác định như sau:
 - $V1 = \{a, b, c, d\}$
 - $E1 = \{(a, b); (a, c); (b, d); (c, b), (d, d)\}$
- Khi đó, biểu diễn hình học của đồ thị này như sau:



Phan Thị Hà

Các khái niệm cơ bản (tiếp)

- Một **cạnh** (u, v) của đồ thị có hướng có thể được biểu thị dạng $u \rightarrow v$. Đỉnh u khi đó được gọi là đỉnh **kề** của v . **Cạnh** (u, v) được gọi là cạnh xuất phát từ u . Ta ký hiệu $A(u)$ là tập các cạnh xuất phát từ u .
- **Bậc ngoài** của 1 đỉnh là số các cạnh xuất phát từ đỉnh đó. Do đó, bậc ngoài của $u = |A(u)|$.
- **Bậc trong** của 1 đỉnh là số các cạnh đi tới đỉnh đó. Do đó, bậc trong của $v = |I(v)|$.
- Một **đường đi** trong đồ thị có hướng $G(V, E)$ là một chuỗi các đỉnh

$$P = \{v_1, v_2, \dots, v_k\}$$

Trong đó, $v_i \in V$ ($i = 1..k$), và $(v_i, v_{i+1}) \in E$ ($i = 1..k-1$).

- **Độ dài của đường đi** trong trường hợp này là $k - 1$.
- **Chu trình** là một đường đi mà đỉnh đầu và đỉnh cuối trùng nhau. **Đồ thị liên thông** là một đồ thị mà luôn tồn tại đường đi giữa 2 đỉnh bất kì.

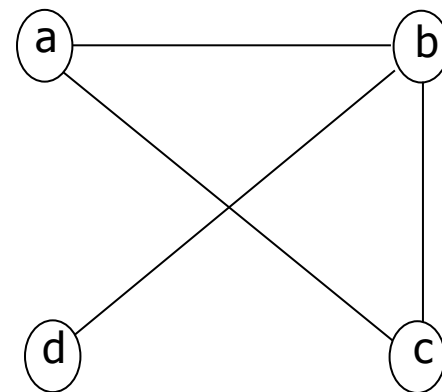
Các khái niệm cơ bản (tiếp)

■ Đồ thị vô hướng

- Đồ thị vô hướng là đồ thị có các cạnh không có hướng. Hai nút ở hai đầu của cạnh có vai trò như nhau. Định nghĩa về đồ thị vô hướng như sau:
- Đồ thị vô hướng $G = \langle V, E \rangle$ bao gồm:
 - V là một tập hữu hạn các đỉnh.
 - E là một tập hữu hạn các cặp đỉnh phân biệt của V , gọi là các cạnh.

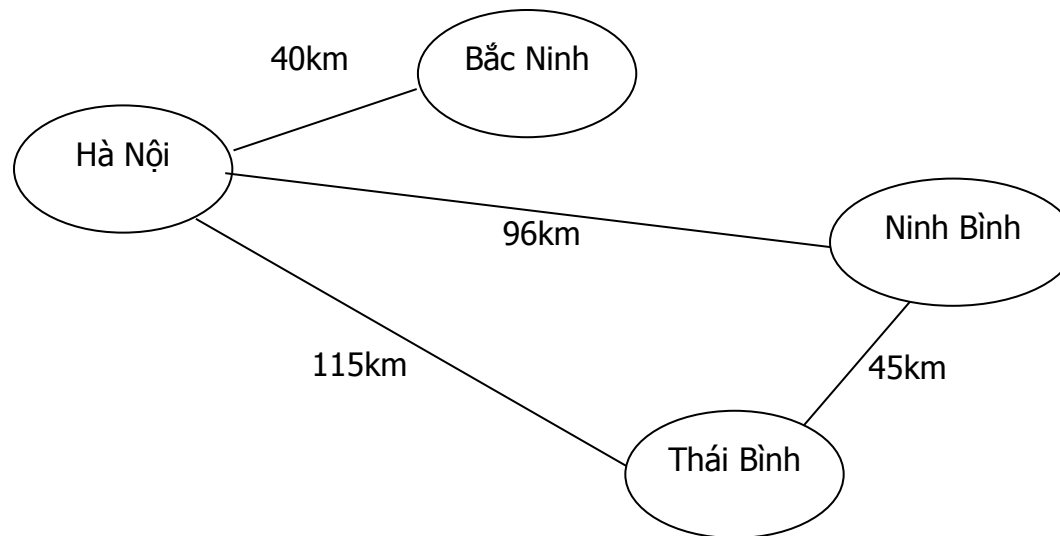
■ Ví dụ, đồ thị có hướng $G2 = \langle V2, E2 \rangle$, với $V2$ và $E2$ được xác định như sau:

- $V2 = \{a, b, c, d\}$
- $E2 = \{(a, b); (a, c); (b, d); (c, b)\}$



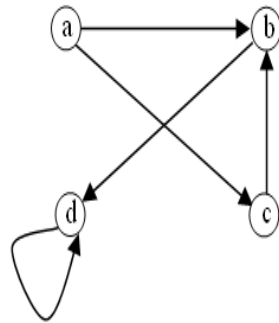
Các khái niệm cơ bản (tiếp)

- Đồ thị có trọng số



Biểu diễn đồ thị - Ma trận kề

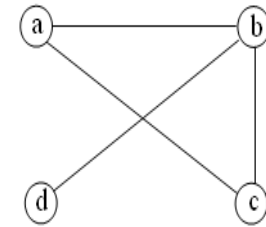
$$A_1 = \begin{vmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{vmatrix}$$



$$v_0 = a, v_1 = b$$

$$v_2 = c, v_3 = d$$

$$A_2 = \begin{vmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \end{vmatrix}$$

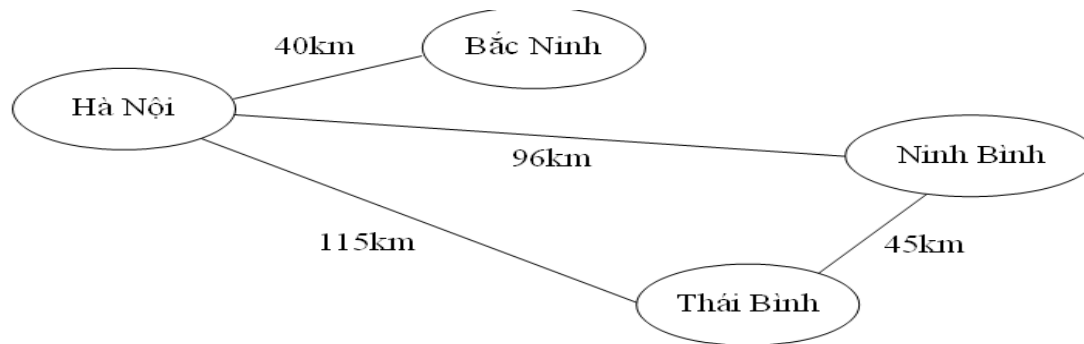


$$v_0 = a, v_1 = b$$

$$v_2 = c, v_3 = d$$

Đồ thị có hướng

Đồ thị vô hướng



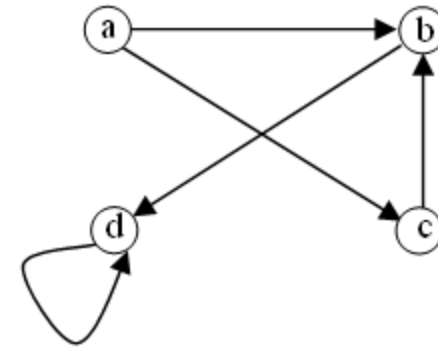
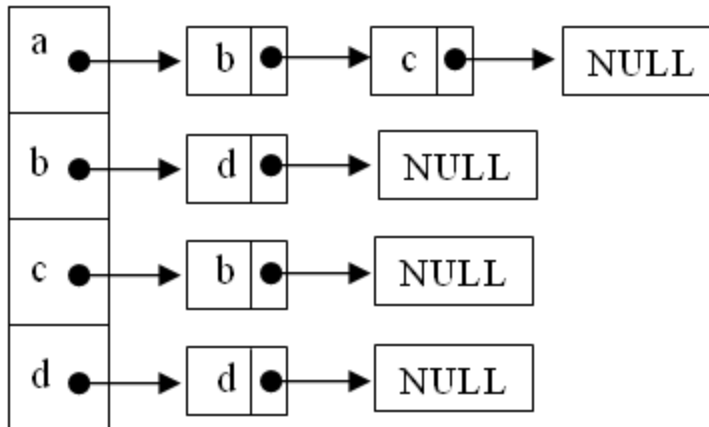
Đồ thị có trọng số

$$A_3 = \begin{vmatrix} \infty & 40 & 115 & 96 \\ 40 & \infty & \infty & \infty \\ 115 & \infty & \infty & 45 \\ 96 & \infty & 45 & \infty \end{vmatrix}$$

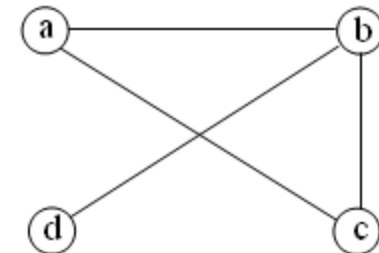
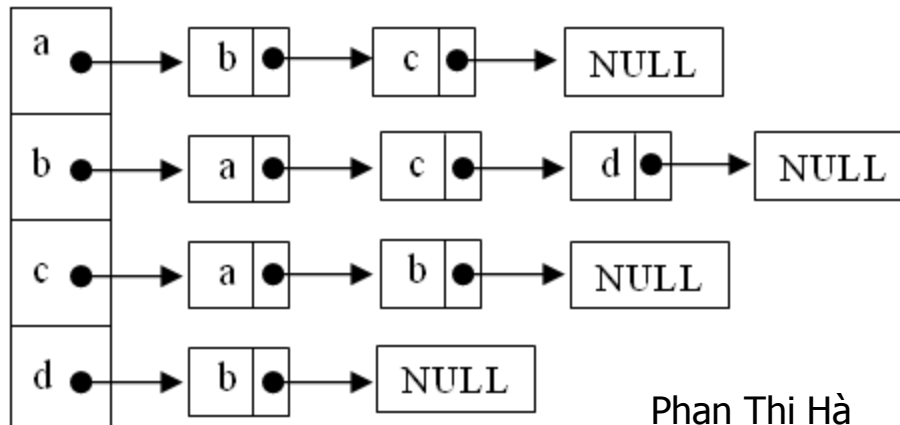
$$v_0 = \text{Hà Nội}, v_1 = \text{Bắc Ninh}, v_2 = \text{Thái Bình}, v_3 = \text{Ninh Bình}$$

Biểu diễn đồ thị - Danh sách kề

■ Đồ thị có hướng



■ Đồ thị vô hướng





Duyệt đồ thị

- Duyệt theo chiều sâu:

```
void DeepFirstSearch(int v){  
    Thăm đỉnh v;  
    daxet[v] = 1;  
    for mỗi đỉnh u kề với v {  
        if (daxet[u]=0 )  
            DeepFirstSearch(v);  
    }  
}
```

- Nếu có nhiều thành phần liên thông

```
for( i=1; i≤n; i++)  
    daxet[i] = 0;  
for( i:=1;i≤ n; i++)  
    if (daxet[i]=0)  
        DeepFirstSearch(i);
```



Duyệt đồ thị

- Duyệt theo chiều rộng

```
void BreadthFirstSearch(int v){
```

```
    queue =  $\phi$ ;
```

```
    Đưa v vào hàng đợi;
```

```
    daxet[v] = 1;
```

```
    while (queue  $\neq \phi$  ){
```

```
        Lấy phần tử ra khỏi hàng đợi, đưa vào biến u;
```

```
        Thăm đỉnh u;
```

```
        for mỗi đỉnh w kề với u {
```

```
            if (daxet[w]=0 ) {
```

```
                Đưa w vào hàng đợi;
```

```
                daxet[w] = 1;
```

```
            }
```

```
        }
```

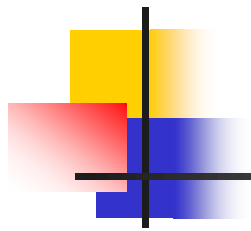
```
    }
```



Ứng dụng duyệt đồ thị để kiểm tra tính liên thông

- Ví dụ sử dụng duyệt theo chiều rộng:

```
int lt=0;
for( v=1; v≤n; v++)
    daxet[v] = 0;
for(v=1; v≤n; v++)
    if (daxet[v]=0){
        BreadthFirstSearch(u);
        lt++;
    }
if (lt ==1) printf("Do thi lien thong!");
else printf("Do thi khong lien thong, so thanh phan lien thong la %d",
    lt);
```



CHƯƠNG 7

SẮP XẾP VÀ TÌM KIẾM



Bài toán sắp xếp

■ Khái niệm

- Sắp xếp là quá trình bố trí lại các phần tử của 1 tập hợp theo thứ tự nào đó. Mục đích chính của sắp xếp là làm cho thao tác tìm kiếm phần tử trên tập đó được dễ dàng hơn.
- Ví dụ về tập các đối tượng được sắp phổ biến trong thực tế là: danh bạ điện thoại được sắp theo tên, các từ trong từ điển được sắp theo vần, sách trong thư viện được sắp theo mã số, theo tên, .v.v.

■ Các giải thuật sắp xếp được chia làm 2 loại.

- Loại thứ nhất là các giải thuật được cài đặt đơn giản, nhưng không hiệu quả (phải sử dụng nhiều thao tác).
- Loại thứ hai là các giải thuật được cài đặt phức tạp, nhưng hiệu quả hơn về mặt tốc độ (dùng ít thao tác hơn).
- Đối với các tập dữ liệu ít phần tử, tốt nhất là nên lựa chọn loại thứ nhất. Đối với tập có nhiều phần tử, loại thứ hai sẽ mang lại hiệu quả hơn.

Sắp xếp lựa chọn

Lựa chọn phần tử có giá trị nhỏ nhất, đổi chỗ cho phần tử đầu tiên. Tiếp theo, lựa chọn phần tử có giá trị nhỏ thứ nhì, đổi chỗ cho phần tử thứ 2. Quá trình tiếp tục cho tới khi toàn bộ dãy được sắp.

32	17	49	98	06	25	53	61
----	----	----	----	----	----	----	----

Bước 1: Chọn được phần tử nhỏ nhất là 06, đổi chỗ cho 32.

06	17	49	98	32	25	53	61
----	----	----	----	----	----	----	----

Bước 2: Chọn được phần tử nhỏ thứ nhì là 17, đó chính là phần tử thứ 2 nên giữ nguyên.

06	17	49	98	32	25	53	61
----	----	----	----	----	----	----	----

Bước 3: Chọn được phần tử nhỏ thứ ba là 25, đổi chỗ cho 49.

06	17	25	98	32	49	53	61
----	----	----	----	----	----	----	----



Sắp xếp lựa chọn

```
void selection_sort(){
    int i, j, k, temp;
    for (i = 0; i < N; i++){
        k = i;
        for (j = i+1; j < N; j++){
            if (a[j] < a[k]) k = j;
        }
        temp = a[i]; a[i] = a[k]; a[k] = temp;
    }
}
```

- Độ phức tạp trung bình của thuật toán là
 - $O(N * N/2) = O(N^2/2) = O(N^2)$.



Vd: 8,1,2,5,4,3

- $i=0; k=0 \Rightarrow k=1$: 1,8,2,5,4,3
- $i=1; k=1 \Rightarrow k=2$: 1,2,8,5,4,3
- $i=2; k=2 \Rightarrow k=3$: 1,2,5,8,4,3
 $k=4$: 1,2,4,8,5,3
 $k=5$: 1,2,3,8,5,4
- $i=3; k=3 \Rightarrow k=4$: 1,2,3,5,8,4
 $k=5$: 1,2,3,4,8,5
- $i=4; k=4 \Rightarrow k=5$: 1,2,3,4,5,8
- $i=5; k=5$:



Sắp xếp chèn

- Phần đầu là các phần tử đã được sắp. Từ phần tử tiếp theo, chèn nó vào vị trí thích hợp tại nửa đã sắp sao cho nó vẫn được sắp.
- Để chèn phần tử vào nửa đã sắp, chỉ cần dịch chuyển các phần tử lớn hơn nó sang phải 1 vị trí và đưa phần tử này vào vị trí trống trong dãy.

32	17	49	98	06	25	53	61
----	----	----	----	----	----	----	----

Bước 1: Chèn phần tử đầu của nửa chưa sắp là 32 vào nửa đã sắp. Do nửa đã sắp là trống nên có thể chèn vào vị trí bất kỳ.

32	17	49	98	06	25	53	61
Đã sắp		Chưa sắp					

Bước 2: Chèn phần tử 17 vào nửa đã sắp. Dịch chuyển 32 sang phải 1 vị trí và đưa 17 vào vị trí trống.

17	32	49	98	06	25	53	61
----	----	----	----	----	----	----	----

Bước 3, 4: Lần lượt chèn phần tử 49, 98 vào nửa đã sắp.

17	32	49	98	06	25	53	61
Đã sắp				Chưa sắp			

Bước 5: Chèn phần tử 06 vào nửa đã sắp. Dịch chuyển các phần tử 17, 32, 49, 98 sang phải 1 vị trí và đưa 06 vào vị trí trống.

06	17	32	49	98	25	53	61
Đã sắp					Chưa sắp		

....{



Sắp xếp chèn

```
void insertion_sort(){
    int i, j, k, temp;
    for (i = 1; i < N; i++){
        temp = a[i];
        j=i-1;
        while ((a[j] > temp)&&(j>=0)) {
            a[j+1]=a[j];
            j--;
        }
        a[j+1]=temp;
    }
}
```

- Độ phức tạp trung bình của thuật toán là
 - $O(N^2/4) = O(N^2)$.



Vd: 8,1,2,3,4,5

i=1;j=0; 1,8,2,3,4,5

i=2; temp=2;j=1; 1,2,8,3,4,5

i=3; temp=3;j=2; 1,2,3,8,4,5

i=4; temp=4;j=3; 1,2,3,4,8,5

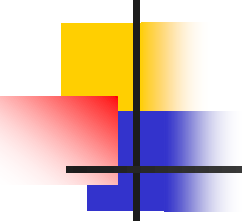
i=5; temp=5;j=4; 1,2,3,4,5,8



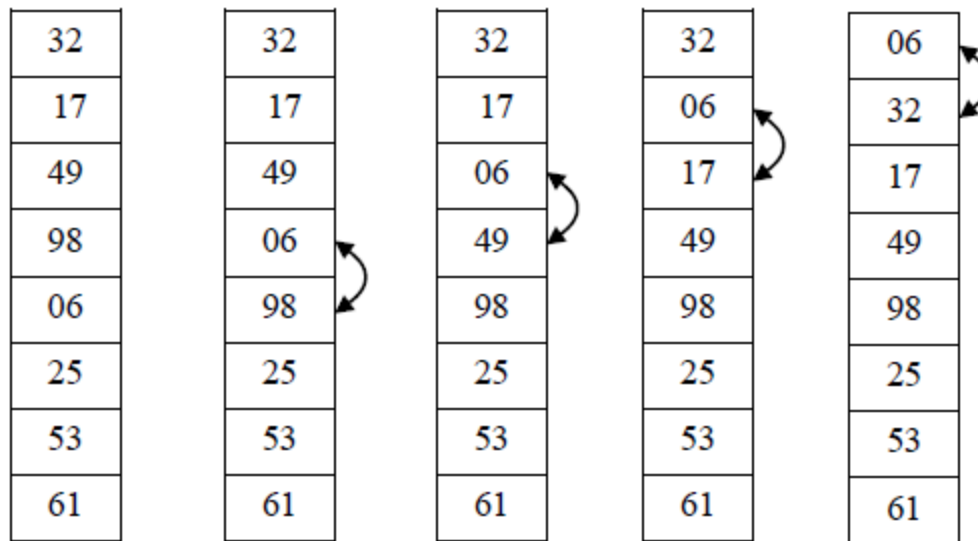
Sắp xếp nổi bọt

- Duyệt nhiều lần từ cuối lên đầu dãy, tiến hành đổi chỗ 2 phần tử liên tiếp nếu chúng ngược thứ tự. Đến một bước nào đó, khi không có phép đổi chỗ nào xảy ra thì toàn bộ dãy đã được sắp.

Như vậy, sau lần duyệt đầu tiên, phần tử nhỏ nhất của dãy sẽ lần lượt được đổi chỗ cho các phần tử lớn hơn và “nổi” lên đầu dãy. Lần duyệt thứ 2, phần tử nhỏ thứ 2 sẽ nổi lên vị trí thứ nhì dãy .v.v. Chú ý rằng, không nhất thiết phải tiến hành tất cả N lần duyệt, mà tới một lần duyệt nào đó, nếu không còn phép đổi chỗ nào xảy ra tức là tất cả các phần tử đã nằm đúng thứ tự và toàn bộ dãy đã được sắp.



Bước 1: Tại bước này, khi duyệt từ cuối dãy lên, lần lượt xuất hiện các cặp ngược thứ tự là (06, 98), (06, 49), (06, 17), (06, 32). Phần tử 06 “nổi” lên đầu dãy.



06
32
17
49
98
25
53
61

06
32
17
49
25
98
53
61

06
32
17
25
49
98
53
61

06
17
32
25
49
98
53
61

Bước 3: Duyệt từ cuối dãy lên, lần lượt xuất hiện các cặp ngược thứ tự là (53, 98), (25, 32). Phần tử 25 nổi lên vị trí thứ 3.

06
17
32
25
49
98
53
61

06
17
32
25
49
53
98
61

06
17
25
32
49
53
98
61



Sắp xếp nổi bọt

- Thủ tục thực hiện sắp xếp nổi bọt trong C như sau:

```
void bubble_sort(){
    int i, j, temp, no_exchange;
    i = 1;
    do{
        no_exchange = 1;
        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;
                no_exchange = 0;
            }
        }
        i++;
    } until (no_exchange || (i == n-1));
}
```

- Tổng cộng, số phép so sánh cần thực hiện là: $(N-1) + (N-2) + \dots + 2 + 1 = N(N-1)/2$. Như vậy, độ phức tạp cũng giải thuật sắp xếp nổi bọt cũng là $O(N^2)$.



QUICK SORT

- Ý tưởng dựa trên phương pháp chia để trị .
- Giải thuật chia dãy cần sắp thành 2 phần, sau đó thực hiện việc sắp xếp cho mỗi phần độc lập với nhau.
- Để thực hiện điều này, đầu tiên chọn ngẫu nhiên 1 phần tử nào đó của dãy làm khoá. Trong bước tiếp theo, các phần tử nhỏ hơn khoá phải được xếp vào phía trước khoá và các phần tử lớn hơn được xếp vào phía sau khoá. Để có được sự phân loại này, các phần tử sẽ được so sánh với khoá và hoán đổi vị trí cho nhau hoặc cho khoá nếu nó lớn hơn khoá mà lại nằm trước hoặc nhỏ hơn khoá mà lại nằm sau. Khi lượt hoán đổi đầu tiên thực hiện xong thì dãy được chia thành 2 đoạn: 1 đoạn bao gồm các phần tử nhỏ hơn khoá, đoạn còn lại bao gồm các phần tử lớn hơn khoá.

Ta hãy xem xét quá trình phân đôi dãy số đã cho ở phần trước.

32	17	49	98	06	25	53	61
----	----	----	----	----	----	----	----

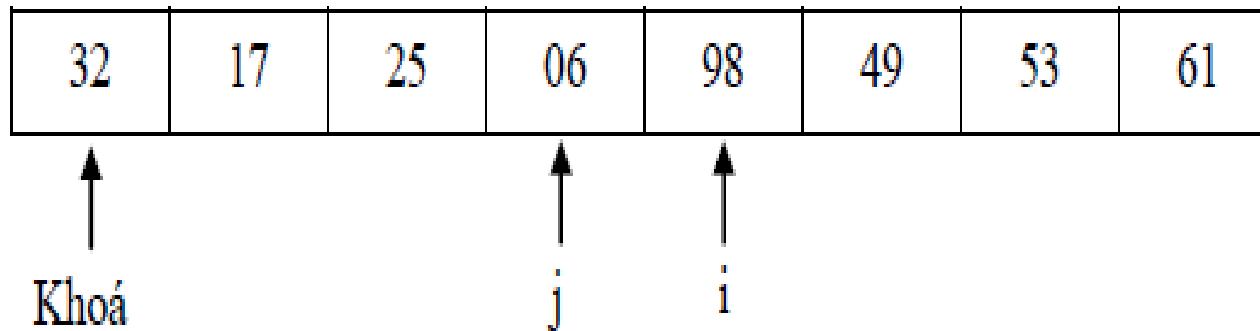
Chọn phần tử đầu tiên của dãy, phần tử 32, làm khoá. Quá trình duyệt từ bên trái với biến duyệt i sẽ dừng lại ở 49, vì đây là phần tử lớn hơn khoá. Quá trình duyệt từ bên phải với biến duyệt j sẽ dừng lại ở 25 vì đây là phần tử nhỏ hơn khoá. Tiến hành đổi chỗ 2 phần tử cho nhau.

32	17	25	98	06	49	53	61
↑		↑			↑		
Khoá		i			j		

Quá trình duyệt tiếp tục. Biến duyệt i dừng lại ở 98, còn biến duyệt j dừng lại ở 06. Lại tiến hành đổi vị trí 2 phần tử 98 và 06.

32	17	25	06	98	49	53	61
↑			↑	↑			
Khoá			i	j			

Tiếp tục quá trình duyệt. Các biến duyệt i và j gặp nhau và quá trình duyệt dừng lại.



Như vậy, dãy đã được chia làm 2 nửa. Nửa đầu từ phần tử đầu tiên đến phần tử thứ j , bao gồm các phần tử nhỏ hơn hoặc bằng khoá. Nửa sau từ phần tử thứ i đến phần tử cuối, bao gồm các phần tử lớn hơn hoặc bằng khoá.

Quá trình duyệt và đổi chỗ được lặp lại với 2 nửa dãy vừa được tạo ra, và cứ tiếp tục như vậy cho tới khi dãy được sắp hoàn toàn.



QUICK SORT

- Mã nguồn giải thuật:

```
void quick(int left, int right) {  
    int i,j;  
    int x,y;  
    i=left; j=right;  
    x= a[left];  
    do {  
        while(a[i]<x && i<right) i++;  
        while(a[j]>x && j>left) j--;  
        if(i<=j){  
            y=a[i];a[i]=a[j];a[j]=y;  
            i++;j--;  
        }  
    }while (i<=j);  
    if (left<j) quick(left,j);  
    if (i<right) quick(i,right);  
}
```

- Hàm chính:

```
void quick_sort(){  
    quick(0, n-1);  
}
```



QUICK SORT

- Độ phức tạp tính toán:

- Thời gian thực hiện thuật toán trong trường hợp xấu nhất này là khoảng $N^2/2$, có nghĩa là $O(N^2)$.
- Trong trường hợp tốt nhất, mỗi lần phân chia sẽ được 2 nửa dãy bằng nhau, khi đó thời gian thực hiện thuật toán $T(N)$ sẽ được tính là:
 - $T(N) = 2T(N/2) + N$
 - Khi đó, $T(N) \approx N \log N$.
- Trong trường hợp trung bình, thuật toán cũng có độ phức tạp khoảng $2N \log N = O(N \log N)$.



HEAP SORT

- Heap sort là một giải thuật đảm bảo kể cả trong trường hợp xấu nhất thì thời gian thực hiện thuật toán cũng chỉ là $O(N\log N)$.
- Ý tưởng cơ bản của giải thuật này là thực hiện sắp xếp thông qua việc tạo các heap, trong đó heap là 1 cây nhị phân hoàn chỉnh có tính chất là khóa ở nút cha bao giờ cũng lớn hơn khóa ở các nút con.

Việc thực hiện giải thuật này được chia làm 2 giai đoạn.

- GĐ1: Tạo heap từ dãy ban đầu. Theo định nghĩa của heap thì nút cha bao giờ cũng lớn hơn các nút con \Rightarrow nút gốc của heap bao giờ cũng là phần tử lớn nhất.
- GĐ2 : Sắp dãy dựa trên heap tạo được. Do nút gốc là nút lớn nhất nên nó sẽ được chuyển về vị trí cuối cùng của dãy và phần tử cuối cùng sẽ được thay vào gốc của heap. Khi đó ta có 1 cây mới, không phải heap, với số nút được bớt đi 1. Lại chuyển cây này về heap và lặp lại quá trình cho tới khi heap chỉ còn 1 nút. Đó chính là phần tử bé nhất của dãy và được đặt lên đầu.

- Với heap ban đầu chỉ có 1 phần tử là phần tử đầu tiên của dãy, ta lần lượt lấy các phần tử tiếp theo của dãy chèn vào heap sẽ tạo được 1 heap gồm toàn bộ n phần tử.

32	17	49	98	06	25	53	61
----	----	----	----	----	----	----	----

Đầu tiên, tạo 1 heap với chỉ 1 phần tử là 32:

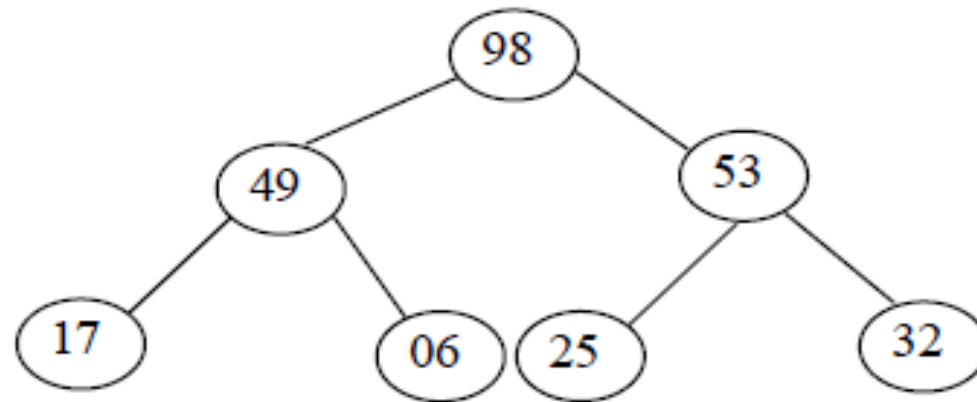


Bước 1: Tiến hành chèn 17 vào heap.

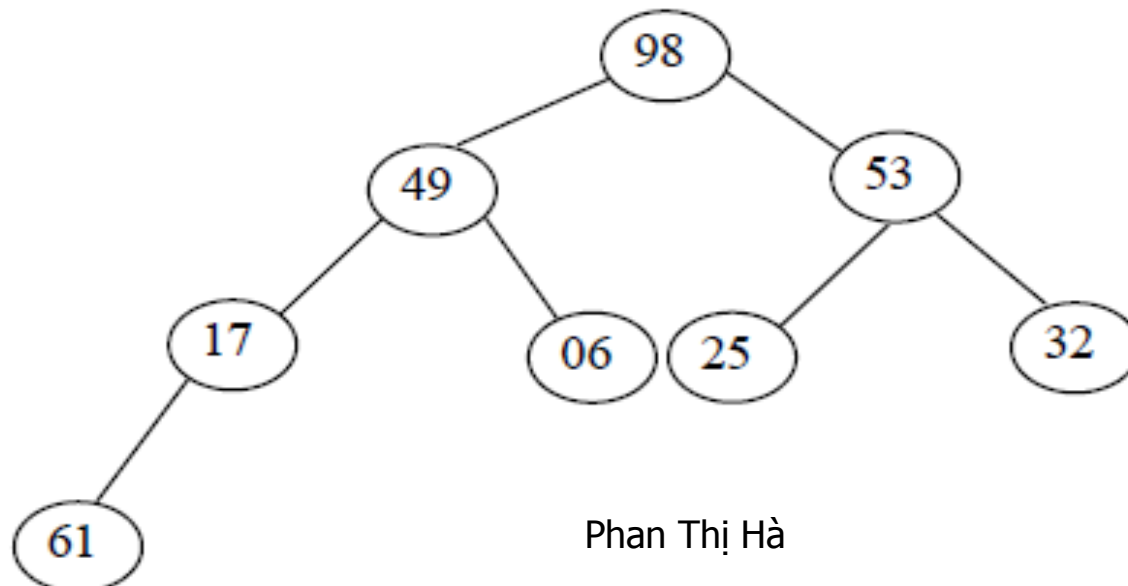


Do không vi phạm định nghĩa heap nên không thay đổi gì.

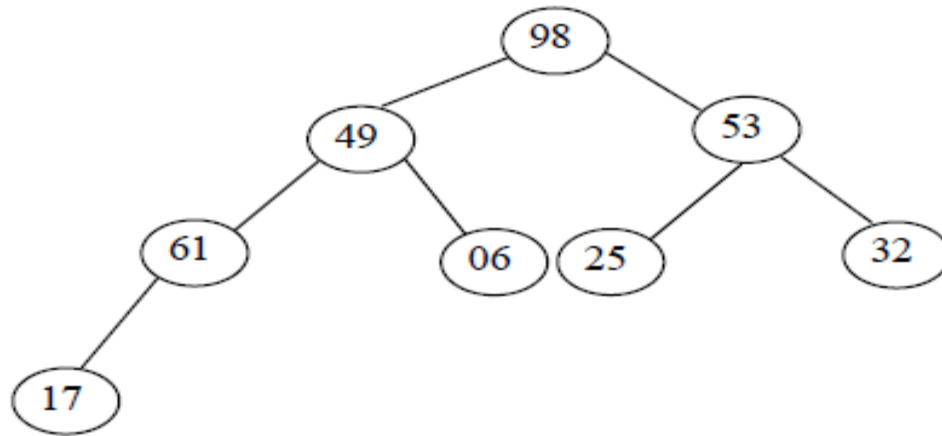
Giả sử ta đã có 1 heap như sau:



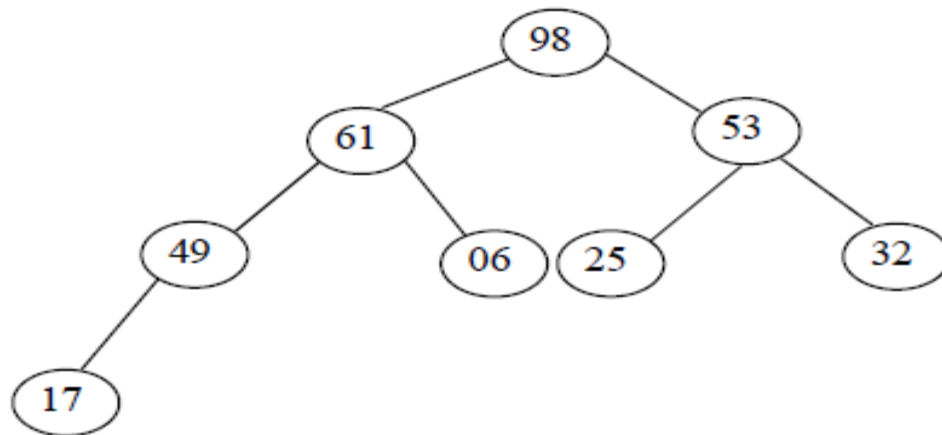
Để chèn phần tử 61 vào heap, đầu tiên, ta đặt nó vào vị trí cuối cùng trong cây.

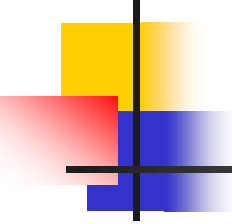


Rõ ràng cây mới vi phạm định nghĩa heap vì nút con 61 lớn hơn nút cha 17. Tiên hành đổi vị trí 2 nút này cho nhau:



Cây này vẫn tiếp tục vi phạm định nghĩa heap do nút con 61 lớn hơn nút cha 49. Lại đổi vị trí 61 cho 49.





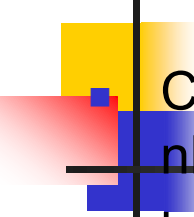
Chèn một phần tử x vào 1 heap đã có k phần tử, ta gán phần tử thứ $k + 1$, $a[k]$, bằng x , rồi gọi thủ tục $\text{upheap}(k)$.

```
■ void upheap(int m){  
■     int x;  
■     x=a[m];  
■     while ((a[(m-1)/2]<=x) && (m>0)){  
■         a[m]=a[(m-1)/2];  
■         m=(m-1)/2;  
■     }  
■     a[m]=x;  
■ }
```

```
■  
■ void insert_heap(int x){  
■     a[m]=x;  
■     upheap(m);  
■     m++;  
■ }
```

- Ta có thủ tục downheap để chỉnh lại heap khi nút k không thoả mãn định nghĩa heap :
-

```
■ void downheap(int k){  
■     int j, x;  
■     x=a[k];  
■     while (k<=(m-2)/2){  
■         j=2*k+1;  
■         if (j<m-1) if (a[j]<a[j+1]) j++;  
■         if (x>=a[j]) break;  
■         a[k]=a[j]; k=j;  
■     }  
■     a[k]=x;  
■ }
```



Cuối cùng, thủ tục heap sort thực hiện việc sắp xếp trên heap đã tạo như sau:

```
■ int remove_node(){  
■     int temp;  
■     temp=a[0];  
■     a[0]=a[m];  
■     m--;  
■     downheap(0);  
■     return temp;  
■ }  
■ void heap_sort(){  
■     int i;  
■     m=0;  
■     for (i=0; i<=n-1; i++) insert_heap(a[i]);  
■     m=n-1;  
■     for (i=n-1; i>=0; i--) a[i]=remove_node();  
■ }
```



MERGE SORT

- Thủ tục tiến hành trộn 2 dãy đã sắp như sau:

```
void merge(int *a, int al, int am, int ar){  
    int i=al, j=am+1, k;  
    for (k=al; k<=ar; k++){  
        if (i>am){  
            c[k]=a[j++];  
            continue;  
        }  
        if (j>ar){  
            c[k]=a[i++];  
            continue;  
        }  
        if (a[i]<a[j]) c[k]=a[i++];  
        else c[k]=a[j++];  
    }  
    for (k=al; k<=ar; k++) a[k]=c[k];  
}
```



MERGE SORT

- Cài đặt cho thuật toán merge_sort bằng đệ qui như sau :

```
void merge_sort(int *a, int left, int right){  
    int middle;  
    if (right<=left) return;  
    middle=(right+left)/2;  
    merge_sort(a, left, middle);  
    merge_sort(a, middle+1, right);  
    merge(a, left, middle ,right);  
}
```



Bài toán tìm kiếm

■ Khái niệm

- Tìm kiếm có thể định nghĩa là việc thu thập một số thông tin nào đó từ một khối thông tin lớn đã được lưu trữ trước đó. Thông tin khi lưu trữ thường được chia thành các bản ghi, mỗi bản ghi có một giá trị khoá để phục vụ cho mục đích tìm kiếm.
- Mục tiêu của việc tìm kiếm là tìm tất cả các bản ghi có giá trị khoá trùng với một giá trị cho trước. Khi tìm được bản ghi này, các thông tin đi kèm trong bản ghi sẽ được thu thập và xử lý.

■ Ví dụ:

- Từ điển máy tính



Tìm kiếm tuần tự

- Thủ tục tìm kiếm tuần tự trên một mảng các số nguyên như sau:

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

- Thuật toán tìm kiếm tuần tự có thời gian thực hiện là $O(n)$. Trong trường hợp xấu nhất, thuật toán mất n lần thực hiện so sánh và mất khoảng $n/2$ lần so sánh trong trường hợp trung bình.



Tìm kiếm nhị phân

- Hàm tìm kiếm nhị phân được cài đặt như sau (giả sử dãy a đã được sắp):

```
int binary_search(int *a, int x){  
    int k, left =0, right=n-1;  
    do{  
        k=(left+right)/2;  
        if (x<a[k]) right=k-1;  
        else l=k+1;  
    }while ((x!=a[k]) && (left<=right))  
    if (x=a[k]) return k;  
    else return -1;  
}
```

- Thuật toán tìm kiếm nhị phân có thời gian thực hiện là $\lg N$.



Tìm kiếm trên cây nhị phân tìm kiếm

- Mã nguồn:

```
struct      node {
    int      item;
    struct node *left;
    struct node *right;
}
typedef struct node *treenode;
treenode tree_search(int x, treenode root){
    int found=0;
    treenode temp=root;
    while (temp!=NULL){
        if (x < temp.item) temp=temp.left;
        elseif (x > temp.item)temp=temp.right;
        else break;
    }
    return temp;
}
```

- 
-
- Các nhóm tự tìm hiểu về cây AVL và làm báo cáo