Chương 1.
GIẢI THUẬT

LOGO

Nội dung

- Giải thuật và cấu trúc dữ liệu
  - Giải thuật và các đặc trưng của giải thuật
  - Diễn đạt giải thuật
  - Kiểu dữ liệu, ADT, Cấu trúc dữ liệu
- Phân tích và thiết kế giải thuật
  - Thiết kế giải thuật
  - Phân tích giải thuật
- Một số lớp các giải thuật

Mục tiêu

- Tìm hiểu các nội dung:
  - Thiết kế và phân tích được giải thuật
  - Hiểu rõ về Kiểu dữ liệu, Kiểu dữ liệu trừu tượng, Cấu trúc dữ liệu.
  - Đánh giá độ phức tạp của giải thuật cơ bản

Giải bài toán bằng máy tính

- Giải quyết một bài toán:
  - Làm gì ?
  - Làm như thế nào ?
- Giải quyết Bài toán Tin học $\Rightarrow$ phải:
  - Tổ chức biểu diễn các đối tượng thực tế
  - Xây dựng trình tự các thao tác xử lý trên các đối tượng dữ liệu đó

Giải bài toán bằng máy tính

- Hai yếu tố tạo nên một chương trình máy tính
  - Cấu trúc dữ liệu
  - Giải thuật

Cấu trúc dữ liệu + Giải thuật = Chương trình

Giải thuật

- **Định nghĩa**: là dãy các câu lệnh chặt chẽ và rõ ràng xác định một trình tự các thao tác trên một số đối tượng nào đó, sao cho sau một số hữu hạn bước thực hiện ta đạt được kết quả mong muốn
- Mỗi thuật toán có một **dữ liệu vào (Input)** và một **dữ liệu ra (Output)**;

Giải thuật

- Lý thuyết giải thuật quan tâm đến những vấn đề sau :
  - 1. Giải được bằng giải thuật :
  - 2. Tối ưu hóa giải thuật :
  - 3. Triển khai giải thuật:

Đặc trưng của giải thuật

- Tính xác định :
- Tính dừng (hữu hạn):
- Tính đúng đắn:
- Tính phổ dụng:
- Tính khả thi:

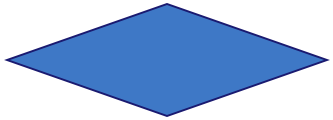
Diễn đạt giải thuật

- Dạng lưu đồ (sơ đồ khối)
- Dạng ngôn ngữ tự nhiên (Ngôn ngữ liệt kê từng bước)
- Dạng mã giả
- Ngôn ngữ lập trình

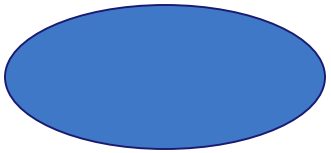
Diễn đạt giải thuật

Các nút biểu diễn giải thuật bằng sơ đồ khối

Nút thao tác:



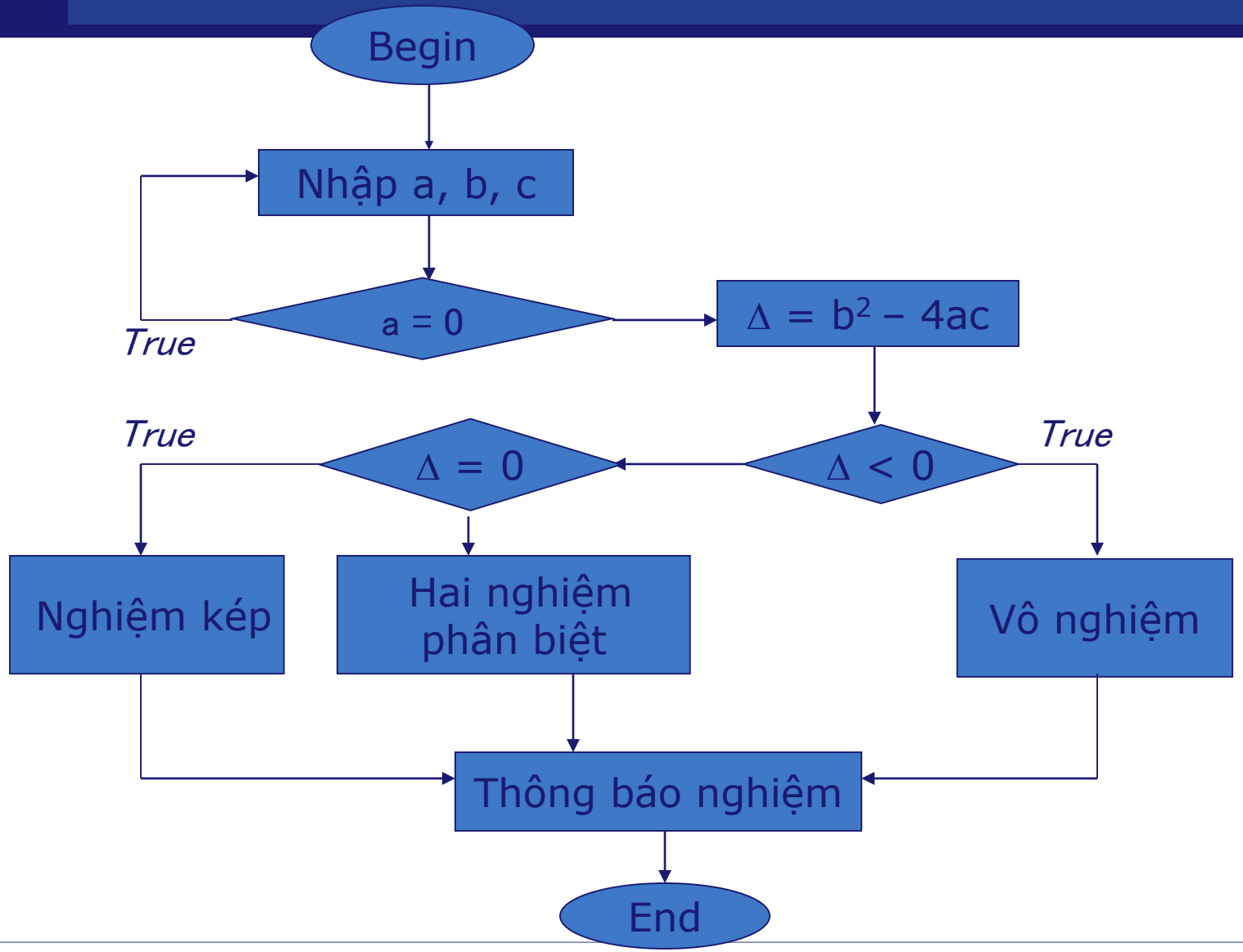
*Nút điều khiển:*trong đó ghi điều kiện cần kiểm tra trong quá trình tính toán.



Nút khởi đầu ,kết thúc:

Cung :

Ví dụ : Giải PT: $ax^2 + bx + c = 0$, giải thuật mô tả bằng sơ đồ khối



Diễn đạt giải thuật

- Ví dụ 1: Giải thuật xác định n là số nguyên tố
  - Bước 1: Ghi nhận n
  - Bước 2: Nếu n ≤ 1 → n ko nguyên tố → dừng
  - Bước 3: Nếu n > 2, gán i ← 2
  - Bước 4: Nếu i ≥ √n hay n chia hết cho i → bước 6
  - Bước 5: Gán i ← i+1, trở lại bước 4
  - Bước 6:
    - Nếu i > √n → n nguyên tố → dừng
    - Ngược lại, n không là nguyên tố → dừng

Diễn đạt giải thuật (tt)

- Ví dụ 2: Giải thuật tìm phần tử thứ n của dãy số Fibonacci
  - Bước 1: Ghi nhận n
  - Bước 2: Nếu n=1 hay n=2 → u_n=1 → dừng
  - Bước 3: Nếu n > 2, gán a←1, b←1, i←1
  - Bước 4: Gán c←a+b, a←b, b←c
  - Bước 5:
    - Nếu i = n - 2 → u_n=c → dừng
    - Ngược lại i ← i+1, quay lại bước 4

Diễn đạt giải thuật (tt)

- Ví dụ 3: tìm phần tử lớn nhất trong mảng A
  - Giải thuật timMax(*A*, *n*)

    Input: Mảng *A*, *gồm n* số nguyên

    Output: Giá trị lớn nhất của *A*

```
Max ← A[0]
for i ← 1 to n – 1 do
        if A[i] > Max then
                Max ← A[i]
return Max
```

Kiểu dữ liệu, Kiểu dữ liệu trừu tượng

- Kiểu dữ liệu (Data type)
- Kiểu dữ liệu trừu tượng (ADT - abstract data type):
  - Một kiểu dữ liệu trừu tượng là một mô hình toán học cùng với một tập hợp các phép toán (operation) được định nghĩa trên mô hình đó.

Cấu trúc dữ liệu

- Cấu trúc dữ liệu (Data structure)
- Trong ngôn ngữ lập trình, có một số cấu trúc dữ liệu riêng của nó được gọi là CTDL tiền định.

Cấu trúc lưu trữ (trong/ngoài)

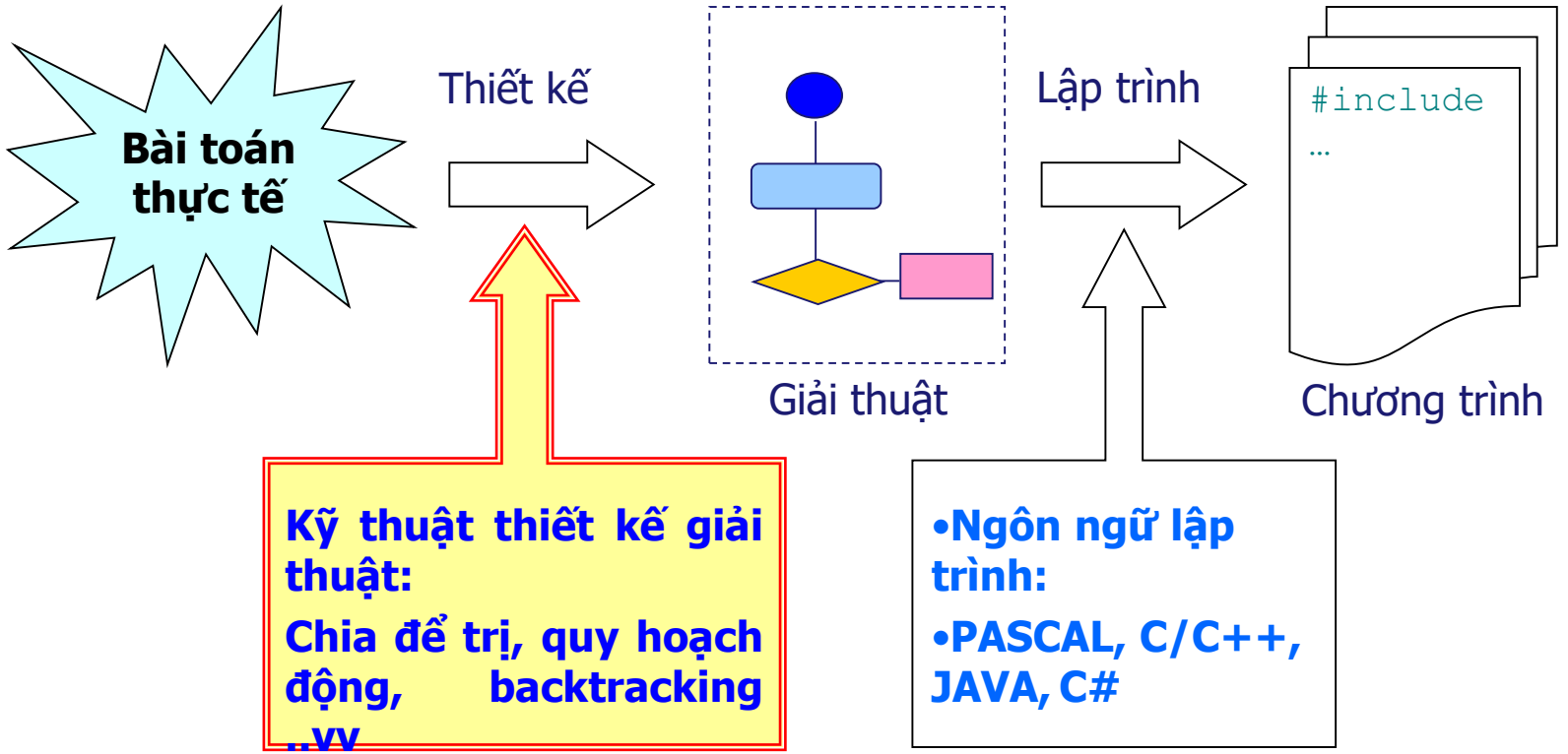
- Là các biểu diễn cấu trúc dữ liệu trên bộ nhớ (trong/ngoài) của máy tính
- Có nhiều cấu trúc lưu trữ khác nhau cho cùng một cấu trúc dữ liệu

Mối quan hệ giữa Giải thuật và Cấu trúc dữ liệu

- Đối tượng xử lý của giải thuật chính là dữ liệu
- Với một cấu trúc dữ liệu, sẽ có những giải thuật tương ứng.
- Khi cấu trúc dữ liệu thay đổi thường giải thuật cũng phải thay đổi theo.

Thiết kế giải thuật

- Từ bài toán đến chương trình



Thiết kế giải thuật (tt)

- Với một vấn đề đặt ra, làm thế nào để đưa ra thuật toán giải quyết nó?
- Chiến lược thiết kế:
  - Chia-để-trị (divide-and-conquer)
  - Quy hoạch động (dynamic programming)
  - Quay lui (backtracking)
  - Tham lam (greedy method)

Thiết kế giải thuật (tt)

- Module hoá và việc giải quyết bài toán
  - Chiến thuật chia để trị (divide-conquer):
  - Để thực hiện chiến thuật này, thường có hai cách thiết kế:

1. Từ trên xuống (Top-Down Design).
2. Tinh chỉnh từng bước

Thiết kế giải thuật (tt)

- **Sau đây là lược đồ của kỹ thuật chia-để-trị:**

```
DivideConquer (A,x)    // tìm nghiệm x của bài toán A.
{
   if (A đủ nhỏ)
         Solve (A);
    else {
         Chia bài toán A thành các bài toán con
                        A1, A2,…, Am;
         for (i = 1; i <= m ; i ++)
             DivideConquer (Ai , xi);
    Kết hợp các nghiệm xi của các bài toán con Ai (i=1, …, m)
    để nhận được nghiệm x của bài toán A;
    }
}
```

Thiết kế giải thuật (tt)

- Tinh chỉnh từng bước:
  - Biểu diễn ý tưởng bằng ngôn ngữ tự nhiên
  - Cụ thể từng phần, thay đổi bằng ngôn ngữ chương trình
  - Cuối cùng ta có chương trình

Thiết kế giải thuật (tt)

- Ví dụ: Bài toán sắp xếp một dãy n số, theo thư tự tăng dần
  - Chọn số bé nhất trong n số để vào vị trí thứ 1
  - Chọn số bé nhất trong n-1 số còn lại để vào vị trí thứ 2
  - ……
  - Chọn số bé nhất trong 2 số còn lại để vào vị trí thứ n-1

Thiết kế giải thuật (tt)

- ➔ *For (i= 1, i <= n-1, i++)*

  {- Chọn số bé nhất trong các số $x_i, x_{i+1}, \ldots, x_n$

  - Đổi chỗ cho xi

  }

- ➔ *For (i= 1, i <= n-1, i++)*

  {- tg = x[i]

  -So sánh tg với các số từ x_{i+1} *->* x_n *. Nếu x[i] > các số đó thì lại lấy số đó làm số tg*

  - đổi chổ x[i] và tg

}

Thiết kế giải thuật (tt)

```
for (i= 1; i <= n-1; i++)
 for(j = i+1; j <=n; j++) {
  If  (x[j]  <  x[i])
  {
     Tg = x[i] ;
      X[i] = x[j];
      X[j] := tg ;
   }
}
```

Thiết kế giải thuật (tt)

- Ví dụ 3: Tìm tất cả các số tự nhiên có hai chữ số, khi đảo trật tự của hai số đó sẽ tạo được một số nguyên tố cùng nhau với số đã cho
  - Phân tích giả thiết
    - Gọi x=ab là số có hai chữ số cần tìm
    - a,b = 0…9
    - a > 0
    - (ab,ba)=1

Thiết kế giải thuật (tt)

- Ví dụ 3 (tt)
  - Tinh chỉnh 1
    - x = 10…99
    - x’ = 10*donvi(x)+chuc(x)
    - (x,x’)=1 ⇔ USCLN(x,x’)=1
  - Tinh chỉnh 2
    - x chạy từ 10 đến 99
    - y=10*donvi(x)+chuc(x)
    - nếu USCLN(x,y)=1 thì Xuất(x)

Phân tích Giải thuật (tt)

- Khi một giải thuật được xây dựng, hàng loạt yêu cầu đặt ra
  - Yêu cầu về tính đúng đắn của giải thuật
  - Tính đơn giản của giải thuật.
  - Yêu cầu về không gian :
  - Yêu cầu về thời gian :

Phân tích Giải thuật (tt)

- Giải quyết bài toán
  - Phải đứng trước việc lựa chọn giải thuật nào ?
  - Dựa trên cơ sở nào để lựa chọn ?
- Có hai mục tiêu trái ngược
  - Thuật toán dễ hiểu, cài đặt và gỡ lỗi (1).
  - Thuật toán sử dụng hiệu quả tài nguyên máy tính, đặc biệt chạy càng nhanh càng tốt (2).

Phân tích Giải thuật (tt)

- Độ phức tạp không gian (Space complexity)
  - Dung lượng bộ nhớ mà thuật toán đòi hỏi
- Độ phức tạp thời gian (Time complexity)
  - Thời gian thực hiện thuật toán

Phân tích thời gian thực hiện giải thuật

- Thời gian thực hiện giải thuật phụ thuộc vào các yếu tố sau:
  - Dữ liệu vào
  - Tốc độ thực hiện các phép toán của máy tính (phần cứng máy tính)
  - Trình biên dịch

Phân tích thời gian thực hiện giải thuật

- Sử dụng các công cụ toán học để đánh giá thời gian chạy của giải thuật:
- Gọi n là kích thước của dữ liệu vào, thời gian thực hiện của giải thuật có thể biểu diễn là một như hàm của n: hàm T(n)

Tiến trình phân tích thời gian thực hiện giải thuật

- Bước 1: Phân tích kích thước dữ liệu vào
- Bước 2: Phân tích (toán học) tìm ra giá trị trung bình, và giá trị xấu nhất cho mỗi đại lượng cơ bản.

Độ phức tạp tính toán của giải thuật

- Ví dụ 4:
  - Giải thuật A, độ phức tạp thời gian $T_a(n)$
  - Giải thuật B, độ phức tạp thời gian $T_b(n)$
  - Khi n lớn, $T_a(n) >> T_b(n)$. Có thể kết luận giải thuật A chậm hơn giải thuật B.

Ký pháp để đánh giá độ phức tạp tính toán của giải thuật

- Ký hiệu O (big-Oh): hàm f(n) và g(n), ta nói:

  f(n) = O(g(n)), nếu tồn tại các hằng số dương c và n_0 sao cho $f(n) \leq cg(n)$ khi $n \geq n_0$.

- Ký hiệu này dùng để chỉ chặn trên của một hàm
- Ý nghĩa: Tốc độ tăng của hàm f(n) không lớn hơn hàm g(n)

Ký pháp để đánh giá độ phức tạp tính toán của giải thuật

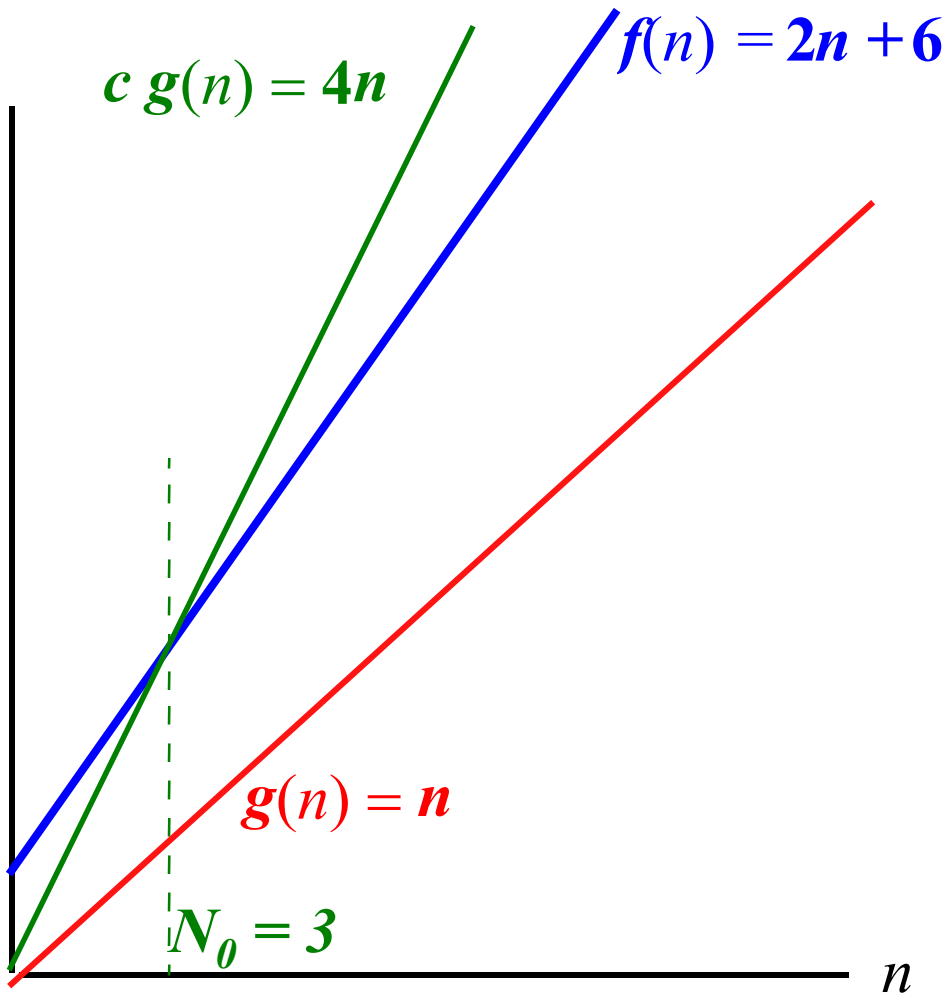
- Ví dụ :

f(n) = 2n+6,

g(n) = n và c = 4 ,

n_0=3

- f(n)= O(n)



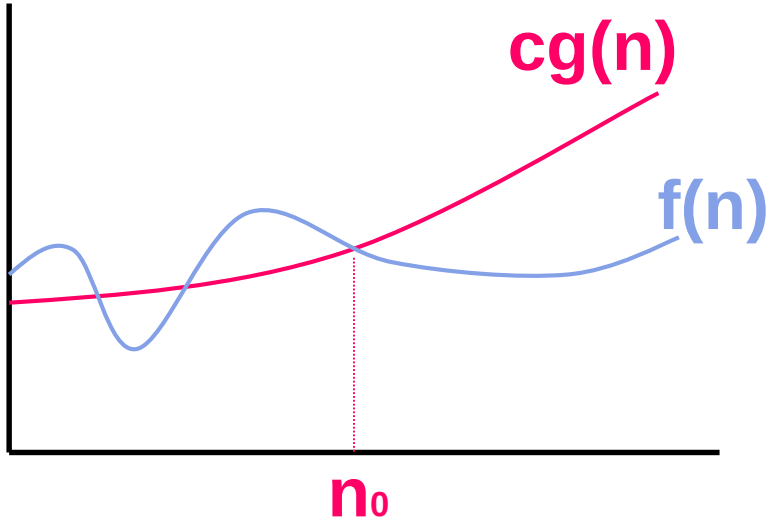
Ký pháp để đánh giá độ phức tạp tính toán của giải thuật

- Định nghĩa Ω:
  - f(n) = Ω(g(n)), nếu tồn tại các *hằng số dương c* và n_o sao cho $f(n) \geq cg(n)$ khi $n \geq n_o$
  - $f(n) = \Omega(g(n)) \Leftrightarrow g(n) = O(f(n))$
- Định nghĩa Θ

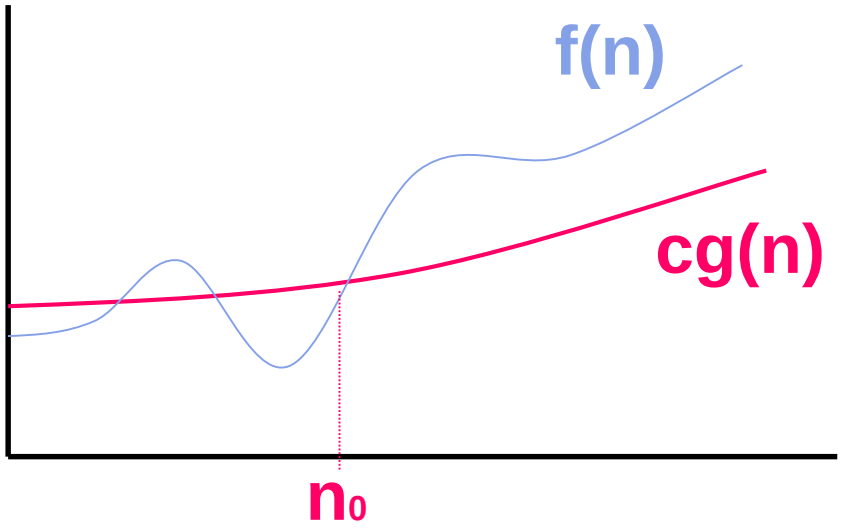
  f(n) = Θ(g(n)), nếu tồn tại các hằng số dương c_1, c_2 và n_0 sao cho $c_1.g(n) \leq f(n) \leq c_2.g(n)$ với mọi $n > n_0$

Ký pháp để đánh giá độ phức tạp tính toán của giải thuật

$$f(n)=O(g(n))$$

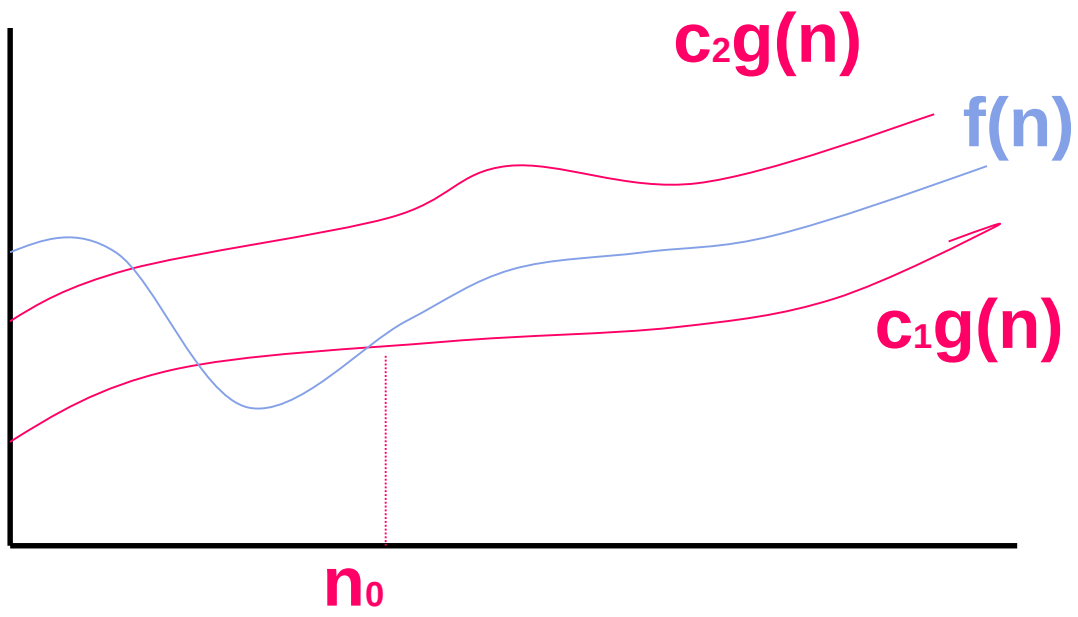


$$f(n)=\Omega(g(n))$$



Ký pháp để đánh giá độ phức tạp tính toán của giải thuật

$$f(n)=\Theta(g(n))$$



Ký pháp để đánh giá độ phức tạp tính toán của giải thuật

- Ta nói độ phức tạp tính toán của giải thuật có cấp f(n) nếu thỏa :
  - T(n) = O(f(n)), và
  - Nếu ∃ g(n), mà T(n) = O(g(n)) thì f(n) = O(g(n)).

Một số qui tắc về ký hiệu O lớn

- Nếu $f_1(n)=O(g_1(n))$ và $f_2(n)=O(g_2(n))$
  - $f_1(n)+f_2(n)=O(g_1(n)+g_2(n))=max(O(g_1(n),g_2(n))$
  - $f_1(n)*f_2(n)=O(g_1(n)*g_2(n))$
  - $\log_k N=O(N)$ với mọi hằng số k
- Nếu f(n) là một đa thức bậc k,
  - thì $\boldsymbol{f(n)}$ là $\boldsymbol{O(n^k)}$,

Một số qui tắc về ký hiệu O lớn

- f = O(f)
- f = O(g) và g = O(h) thì f = O(h)
- f = O(g) và h=O(r) thì fh = O(gr)
- f =O(g) và h=O(r) thì f+h = O(g+r)
- f =O(g) thì af = O(g) với mọi a>0.
- O(c)=O(1) , c là hằng số

Một số qui tắc về ký hiệu O lớn

- Ví dụ:
  - 2n là O(n)
  - $3n + 5$ là $O(n)$ thay vì $3n + 5$ là $O(3n)$
  - $4n^2 + 5n + 7$ la O(?)

Xác định độ phức tạp tính toán

- T1(n) và T2(n) là thời gian thực hiện của hai giai đoạn chương trình P1 và P2 mà T1(n) = O(f(n)); T2(n) = O(g(n))
- **Qui tắc tổng**:
  - Thời gian thực hiện đoạn P1 rồi P2 tiếp theo sẽ là T1(n) + T2(n) = O(max(f(n),g(n))).
- **Qui tắc nhân**:
  - Thời gian thực hiện P1 và P2 lồng nhau sẽ là : T1(n)T2(n) = O(f(n)*g(n))

Các qui tắc tổng quát

- Các phép gán, đọc, viết, goto là các phép toán sơ cấp:
  - Thời gian thực hiện là: O(1)
- **Lệnh lựa chọn: if-else** có dạng

  if (<điều kiện>)

  lệnh 1

  else

  lệnh 2

Các qui tắc tổng quát

- Câu lệnh switch được đánh giá tương tự như lệnh if-else.
- Các lệnh lặp: for, while, do-while
  - Cần đánh giá số tối đa các lần lặp, giả sử đó là L(n)
  - Tiếp theo đánh giá thời gian chạy của mỗi lần lặp là $T_i(n)$, (i=1,2,..., L(n))
  - Mỗi lần lặp, chi phí kiểm tra điều kiện lặp,là $T_0(n)$.
  - Chí phí lệnh lặp là: $\sum_{i=1}^{L(n)} (T_0(n) + T_i(n))$

Một số ví dụ

- **Ví dụ 1**. Mảng A các số thực, cỡ n, cần tìm xem mảng có chứa số thực x không.

```
(1)  i = 0;
(2)  while (i < n &&  x != A[i])
(3)   i++;
```

Một số ví dụ

Case1:
```
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        k++;
```
$O(n^2)$

Case 2:
```
for (i=0; i<n; i++)
    k++;
for (i=0; i<n; i++)
    for (j=0; j<n; j++)
        k++;
```
$O(n^2)$

Case 3:
```
for (int i=0; i<n-1; i++)
    for (int j=0; j<i; j++)
        int k+=1;
```
$O(n^2)$

Một số ví dụ

```
int MaxSubSum1(const int a[], int n) {
    int maxSum=0;

    for (int i=0; i<n; i++)
        for (int j=i; j<n; j++) {
            int thisSum=0;

            for (int k=i; k<=j; k++)
                thisSum+=a[k];

            if (thisSum>maxSum)
                maxSum=thisSum;
        }
    return maxSum;
}
```

$O(n^3)$

Một số ví dụ

```
int MaxSubSum2(const int  a[], int n) {
    int maxSum=0;

    for (int i=0; i<n; i++) {
        thisSum=0;
        for (int j=i; j<n; j++) {
            thisSum+=a[j];

            if (thisSum>maxSum)
                maxSum=thisSum;
        }
    }
    return maxSum;
}
```

$O(n^2)$

Một số ví dụ

```
int MaxSubSum4(const int a[], int n) {
    int maxSum=0, thisSum=0;

    for (int j=0; j<n; j++) {
        thisSum+=a[j];

        if (thisSum>maxSum)
            maxSum=thisSum;
        else if (thisSum<0)
            thisSum=0;
    }
    return maxSum;
}
```

O(n)

Một số ví dụ

```
Sum=0
for (j=0;j<N;j++)
  for (k=0;k<N*N;k++)
    Sum++;
```

$O(N^3)$

Một số ví dụ

Ví dụ: sắp xếp dãy

```
void BubbleSort(int a[], int n)
{      int i,j,temp;
(1)     for(i= 0; i<=n-2; i++)
(2)       for(j=n-1; j>=i+1;j--)
(3)         if (a[j] < a[j-1]) {
(4)            temp=a[j-1];
(5)         a[j-1] = a[j];
(6)         a[j]    = temp;
             }
 }
```

Một số ví dụ

- Lệnh (3), (4), (5) và (6) đều tốn O(1)
- Vòng lặp (2) thực hiện (n-i) lần, mỗi lần O(1) do đó vòng lặp (2) tốn T1(n) = O((n-i)*1) = O(n-i)= O(n)
- Vòng lặp (1), số lần lặp là n lần lặp, do đó thời gian thực hiện là T2(n) =O(n)
- 2 vòng lặp (1), (2) lồng nhau nên áp dụng nguyên lý nhân T(n) = T1(n).T2(n)= O(n.n) = $O(n^2)$
- Độ phức tạp của giải thuật là $O(n^2)$

PHÂN TÍCH CÁC HÀM ĐỆ QUY

- **Ví dụ:** Hàm tính giai thừa

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

- Với n <= 1, ta có T(1) = O(1).
- Với n > 1, ta có T(n) = 0(1) + T(n-1)

PHÂN TÍCH CÁC HÀM ĐỆ QUY

- Ta có quan hệ đệ quy sau:
  - T(1) = O(1)
  - T(n) = T(n-1) + O(1) với n > 1
- Thay các ký hiệu O(1) bởi các hằng số dương a và b tương ứng, ta có
  - T(1) = a
  - T(n) = T(n-1) + b với n > 1

PHÂN TÍCH CÁC HÀM ĐỆ QUY

- Sử dụng các phép thế T(n-1) = T(n-2) + b, T(n-2) = T(n-3) + b,..., ta có

$$
\begin{aligned}
T(n) &= T(n-1) + b \\
&= T(n-2) + 2b \\
&= T(n-3) + 3b \\
&\dots \\
&= T(1) + (n-1)b \\
&= a + (n-1)b
\end{aligned}
$$

Từ đó, ta suy ra T(n) = O(n).

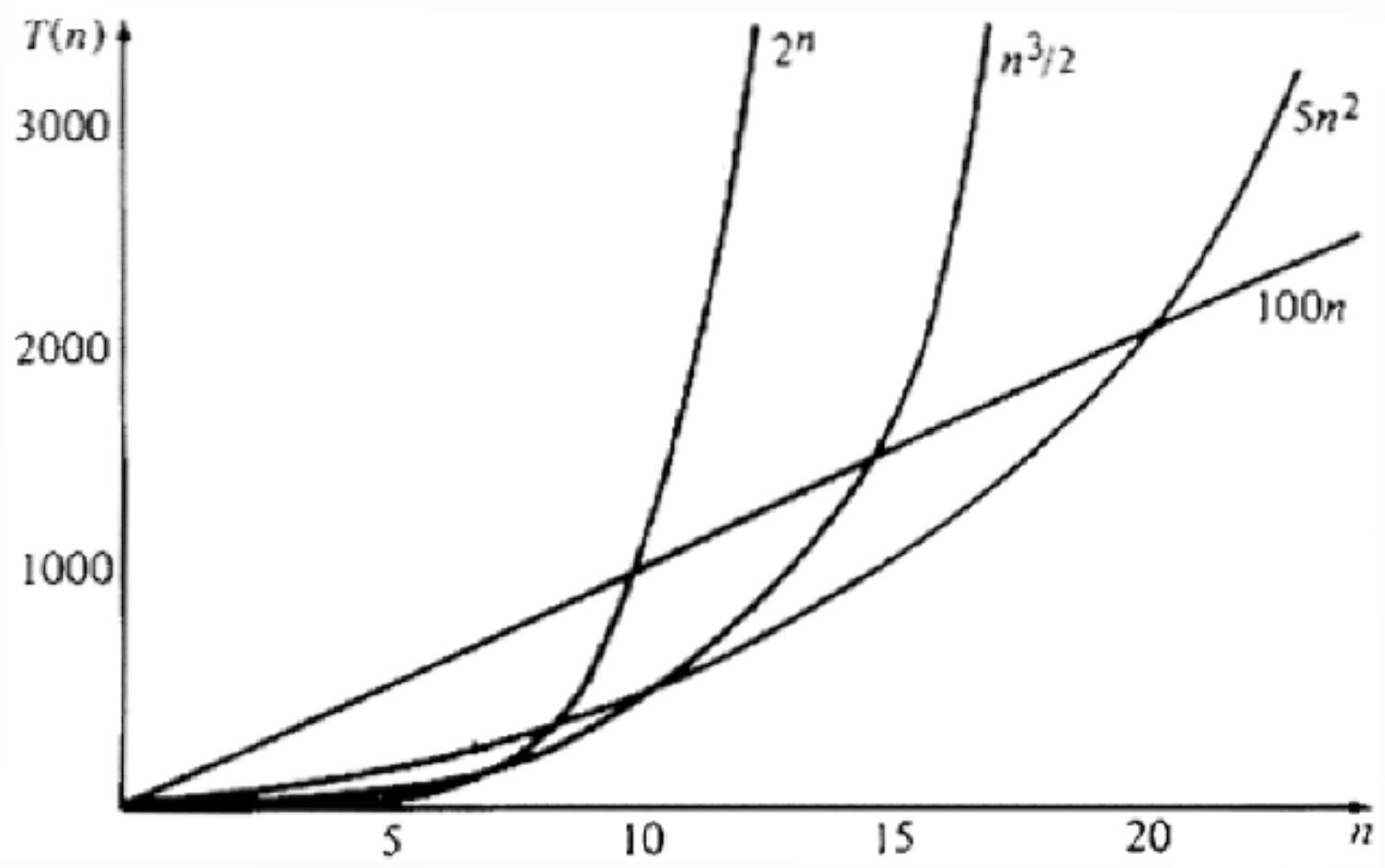
PHÂN TÍCH CÁC HÀM ĐỆ QUY

- Gọi T(n) là thời gian chạy của hàm đệ quy F
- Khi đó, thời gian chạy của các lời gọi hàm ở trong hàm F sẽ là T(m) (với m < n)
- Trước hết, phải đánh giá thời gian chạy của hàm F trên dữ liệu nhỏ nhất n = 1, giả sử T(1) = a (điều kiện dừng)
- Sau đó, đánh giá thời gian chạy của các câu lệnh trong thân của hàm F
- Tìm ra quan hệ đệ quy biểu diễn thời gian chạy của hàm F thông qua lời gọi hàm

Sự phân lớp của giải thuật

- Độ phức tạp
  - O(1) độ phức tạp hằng số
  - O(logn) độ phức tạp logarit
  - O(n) độ phức tạp tuyến tính
  - O(nlogn) độ phức tạp nlogn
  - $O(n^b)$ độ phức tạp đa thức
  - $O(b^n)$ độ phức tạp mũ
  - O(n!) độ phức tạp giai thừa

Sự phân lớp của giải thuật



Đánh giá độ phức tạp trong ba trường hợp

- Độ phức tạp tính toán của giải thuật trong các trường hợp
  - Xấu nhất
  - Tốt nhất
  - Trung bình

Đánh giá độ phức tạp trong ba trường hợp

- Ví dụ 8: Thuật toán tìm kiếm tuần tự

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

Đánh giá độ phức tạp trong ba trường hợp

- Tốt nhất: phần tử đầu tiên là phần tử cần tìm, số lượng phép so sánh là 2 → T(n) ~ O(2) = O(1)
- Xấu nhất: so sánh đến phần tử cuối cùng, số lượng phép so sánh là 2n → T(n) ~ O(n)
- Trung bình: so sánh đến phần tử thứ i, cần 2i phép so sánh, vậy trung bình cần

$(2+4+6+\ldots+2n)/n=2(1+2+\ldots+n)/n=n+1$

→ T(n) ~ O(n)

Kiến thức Toán học bổ trợ về Tổng các chuỗi

$$S(N) = 1 + 2 + \ldots + N = \sum_{i=1}^{N} i = N(1+N)/2$$

- Tổng các BP: $\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3}$ for large N

- Logarithms:
  - $x^a = b \Leftrightarrow \log_x b = a$

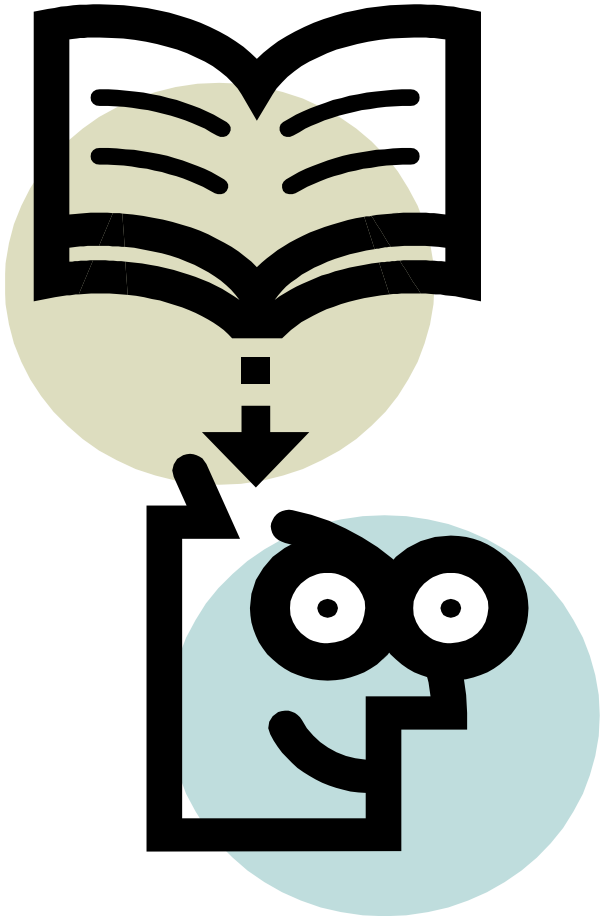
Kiến thức Toán học bổ trợ về Tổng các chuỗi

$$\sum_{i=1}^{N} i^k \approx \frac{N^{k+1}}{|k+1|} \text{ for large N and k} \neq -1$$

$$\sum_{i=0}^{N} A^i = \frac{A^{N+1}-1}{A-1}$$

- Đặc biệt khi A = 2
  - $2^0 + 2^1 + 2^2 + \ldots + 2^N = 2^{N+1} - 1$

Q&A



Đệ quy
và giải thuật đệ quy

LOGO

Nội dung

- Khái niệm đệ quy
- Giải thuật và chương trình đệ quy
- Thiết kế giải thuật đệ quy
- Ưu nhược điểm của đệ quy
- Một số dạng giải thuật đệ quy thường gặp
- Giải thuật đệ qui quay lui (backtracking)
- Một số bài toán giải bằng giải thuật đệ quy điển hình
- Đệ quy và quy nạp toán học

Mục tiêu

- Trang bị cho sinh viên các khái niệm và cách thiết kế giải thuật đệ qui, giải thuật đệ qui quay lui.
- Giới thiệu một số bài toán điển hình được giải bằng giải thuật đệ qui.
- Phân tích ưu và nhược điểm khi sử dụng giải thuật đệ qui

Khái niệm về đệ qui

- Đệ quy: Đưa ra 1 định nghĩa có sử dụng chính khái niệm đang cần định nghĩa(quay về).
- Ví dụ
  - Số tự nhiên: 0 là số tự nhiên, n là số tự nhiên nếu n- 1 là số tự nhiên
  - Hàm n!

Giải thuật và hàm đệ quy

- **Giải thuật đệ quy**
  - Nếu bài toán T được thực hiện bằng lời giải của bài toán T ’ có dạng giống T là lời giải đệ quy
  - Giải thuật tương ứng với lời giải như vậy gọi là giải thuật đệ quy.
- **Hàm đệ quy:** Một hàm được gọi là đệ quy, nếu trong quá trình thực hiện có phần gọi lại chính nó

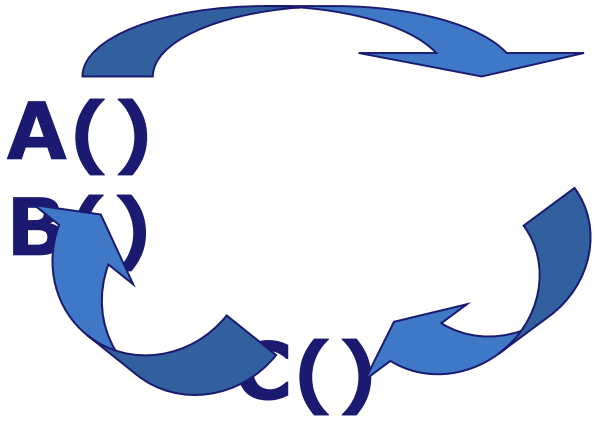
Giải thuật đệ quy

- Ví dụ: Xét bài toán tìm một từ trong quyển từ điển:

If (từ điển là một trang)
tìm từ trong trang này
else {
Mở từ điển vào trang "giữa"
Xác định xem nửa nào của từ điển chứa từ cần tìm;
if (từ đó nằm ở nửa trước)
tìm từ đó ở nửa trước
else tìm từ đó ở nửa sau.
}

Phân loại giải thuật đệ qui

- Đệ quy phân thành 2 loại :
  - **Đệ quy trực tiếp:** Hàm chứa lời gọi đến chính nó
  - **Đệ quy gián tiếp** (Tương hỗ): Hàm chứa lời gọi đến hàm khác, mà ở hàm này lại chứa lời gọi đến chính nó



Cài đặt hàm đệ quy

- Hàm đệ quy về cơ bản gồm hai phần:
  - **Phần cơ sở (Phần neo):** chứa các tác động của hàm với một số giá trị suy biến (neo) của tham số.
  - Phần đệ quy:Chứa các lời gọi đệ quy giải quyết các vấn đề con với cỡ nhỏ hơn.
  - Lời gọi đệ qui dần tiến tới trường hợp neo

Cài đặt hàm đệ quy (tt)

- Cấu trúc hàm đệ qui như sau

```
If (suy biến)
    <Giải quyết trường hợp suy biến>;
Else
    { <tiền xử lý đệ qui>;
     <Gọi đệ qui> ;
     <Xử lý hậu đệ qui>;
     }
```

Một số dạng giải thuật đệ quy đơn giản thường gặp

- **Đệ quy tuyến tính.** Hàm đệ qui tuyến tính dạng:

```
P (<tham số>)
{
        if (điều kiện dừng)
        {
          <Xử lý trường hợp neo>
        }
        Else
        {
        <Thực hiện một số công việc (nếu có)>
          P(<tham số>);
         <Thực hiện một số công việc (nếu có)>
        }
 }
```

Một số dạng giải thuật đệ quy đơn giản thường gặp (tt)

- Ví dụ 1 : Hàm Fact(n) tính số hạng n của dãy n!, định nghĩa như sau:
  - $fact_0 = 1$;
  - $f_n = n*fact_{n-1;}$ (n>=1)

```
longint Fact(int n)
{
    if (n==0)
                return 1;
    else
                return n*Fact(n-1);
}
```

Một số dạng giải thuật đệ quy đơn giản thường gặp (tt)

- **Đệ quy nhị phân.**

```
P (<tham số>)
{
        if (điều kiện dừng)
        {
          <Xử lý trường hợp neo>
        }
        Else
        {
         <Thực hiện một số công việc (nếu có)>
          P(<tham số>);
         <Thực hiện một số công việc (nếu có)>
          P(<tham số>);
          <Thực hiện một số công việc (nếu có)>
        }
 }
```

Một số dạng giải thuật đệ quy đơn giản thường gặp (tt)

- Ví dụ 1: Tính số hạng thứ n của dãy Fibonaci được định nghĩa như sau:
  - $f_1 = f_0 = 1$;
  - $f_n = f_{n-1} + f_{n-2}$; (n>1)

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

Một số dạng giải thuật đệ quy đơn giản thường gặp (tt)

- **Đệ quy phi tuyến.**

```
P (<danh sách tham số>) {
   for (int i = 1; i<=n; i++)
      {
       <Thực hiện một số công việc (nếu có)>
      if (điều kiện dừng)
            {
            <Xử lý trường hợp neo>
            }
      else
            {
                  <Thực hiện một số công việc (nếu có)>
                  P (<danh sách tham số>);
            }
      }
}
```

Một số dạng giải thuật đệ quy đơn giản thường gặp (tt)

- Ví dụ : Cho dãy $\{X_n\}$ xác định theo công thức truy hồi :
  - $X_0 = 1 ; X_n = n^2X_O + (n-1)^2X_1 + \ldots + 2^2X_{n-2} + 1^2X_{n-1}$

```
int X(int n ) ;
{ if ( n == 0 ) return 1 ;
else
{ int tg = 0 ;
for (int i = 0 ; i<n ; i++ ) tg = tg + sqr(n-i)*X(i);
return ( tg ) ;
}
```

Một số dạng giải thuật đệ quy đơn giản thường gặp (tt)

- **Đệ qui tương hỗ:**

```
P2(<danh sách tham số>);// khai báo nguyên mẫu
P1(<danh sách tham số>)
{
    <Thực hiện một số công việc (nếu có)>
    …P2 (<danh sách tham số>);
    <Thực hiện một số công việc (nếu có)>
}

P2 (<danh sách tham số>)
    {
     <Thực hiện một số công việc (nếu có)>
    P1 (<danh sách tham số>);
    <Thực hiện một số công việc (nếu có)>
}
```

Một số dạng giải thuật đệ quy đơn giản thường gặp (tt)

- Ví dụ: Tính số hạng thứ n của hai dãy $\{X_n\}$, $\{Y_n\}$ được định nghĩa như sau:
  - $X_0 = Y_0 = 1$; $X_n = X_{n-1} + Y_{n-1}$; (n>0) ; $Y_n = n_2X_{n-1} + Y_{n-1}$; (n>0)

```
long TinhYn(int n);
long TinhXn (int n)
    {
    if(n==0)
    return 1;
    return TinhXn(n-1) + TinhYn(n-1);
    }
```

```
long TinhYn (int n)
        {
         if(n==0)
          return 1;
          return
      n*n*TinhXn(n-1) +
          TinhYn(n-1);
        }
```

Thiết kế giải thuật đệ qui

- Để xây dựng giải thuật đệ quy, ta cần thực hiện tuần tự 3 nội dung sau :
  - Thông số hóa bài toán .
  - Tìm các trường hợp neo cùng giải thuật giải tương ứng .
  - Tìm giải thuật giải trong trường hợp tổng quát bằng phân rã bài toán theo kiểu đệ quy.

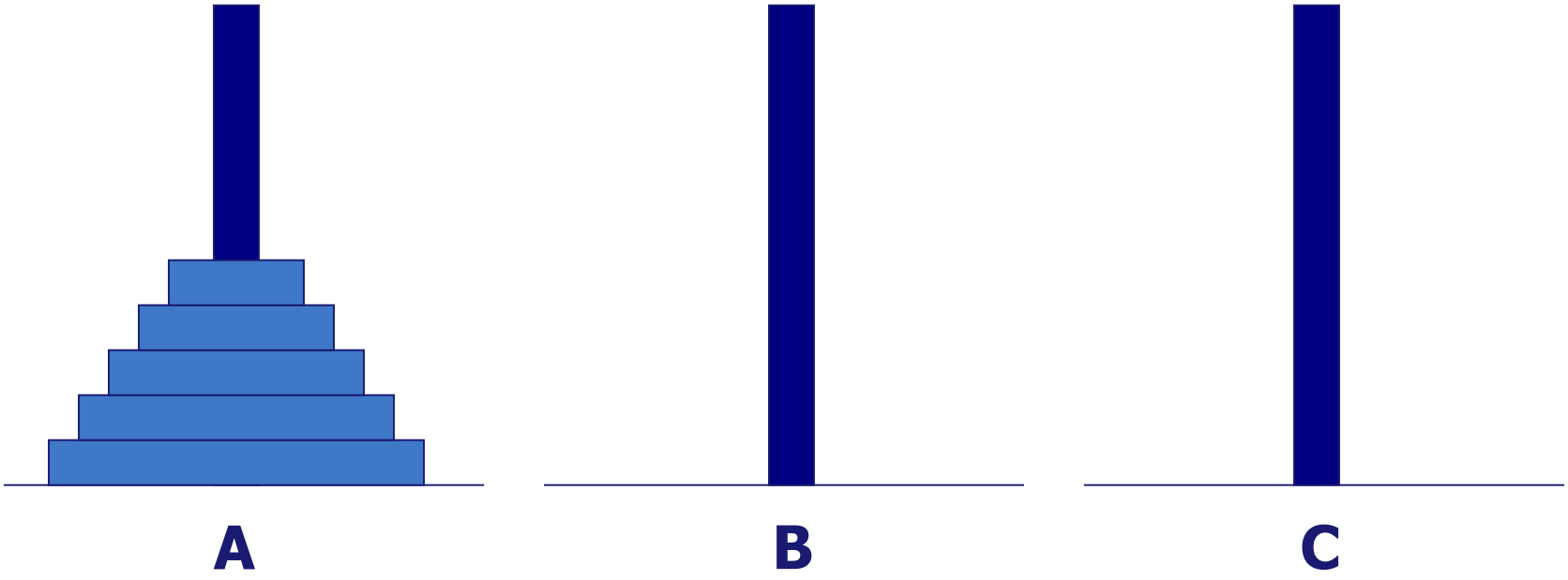
Ưu và nhược điểm của đệ qui

- Ưu điểm của đệ quy
  - Sáng sủa, dễ hiểu, nêu rõ bản chất vấn đề
  - Tiết kiệm thời gian hiện thực mã nguồn
- Nhược điểm của đệ quy
  - Tốn nhiều bộ nhớ, thời gian thực thi lâu
  - Một số bài toán không có lời giải đệ quy

Một số bài toán giải bằng giải thuật đệ qui điển hình

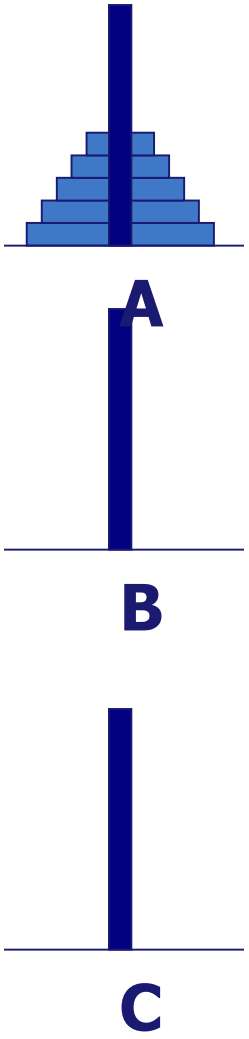
- Bài toán Tháp Hà Nội
- Bài toán chia thưởng

Bài toán tháp Hà Nội



Bài toán tháp Hà Nội

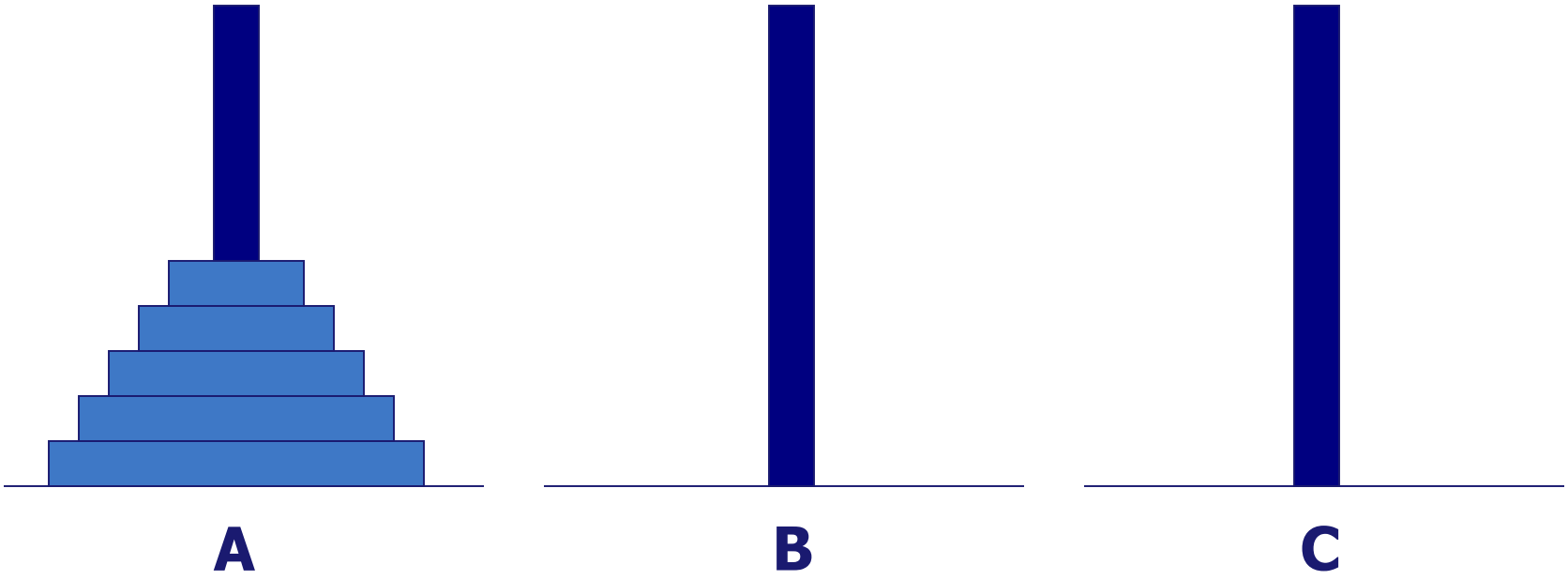
- Bài toán tháp Hà nội : n đĩa
  - Mỗi lần chỉ di chuyển một đĩa
  - Đĩa lớn luôn nằm dưới đĩa nhỏ
  - Được phép sử dụng một cọc trung gian
  - Ký hiệu
    - A: cọc nguồn
    - B: cọc trung gian
    - C: cọc đích



Bài toán tháp Hà Nội

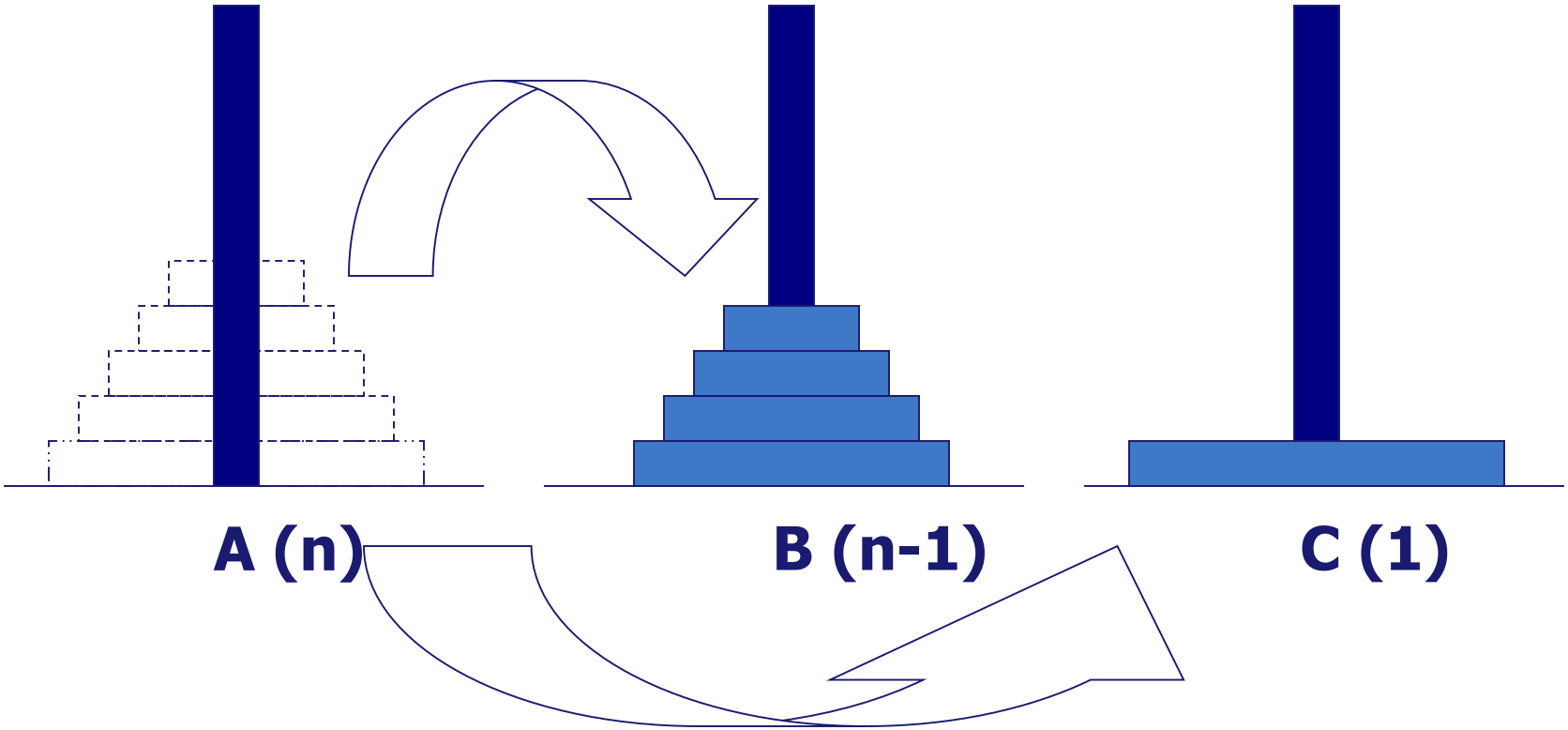
- Trường hợp n = 1
  - Chuyển từ A sang C
- Trường hợp n > 1
  - Chuyển (n-1) đĩa từ A sang B, C trung gian
  - Chuyển đĩa n từ A sang C
  - Chuyển (n-1) đĩa từ B sang C, A làm trung gian

Bài toán tháp Hà Nội



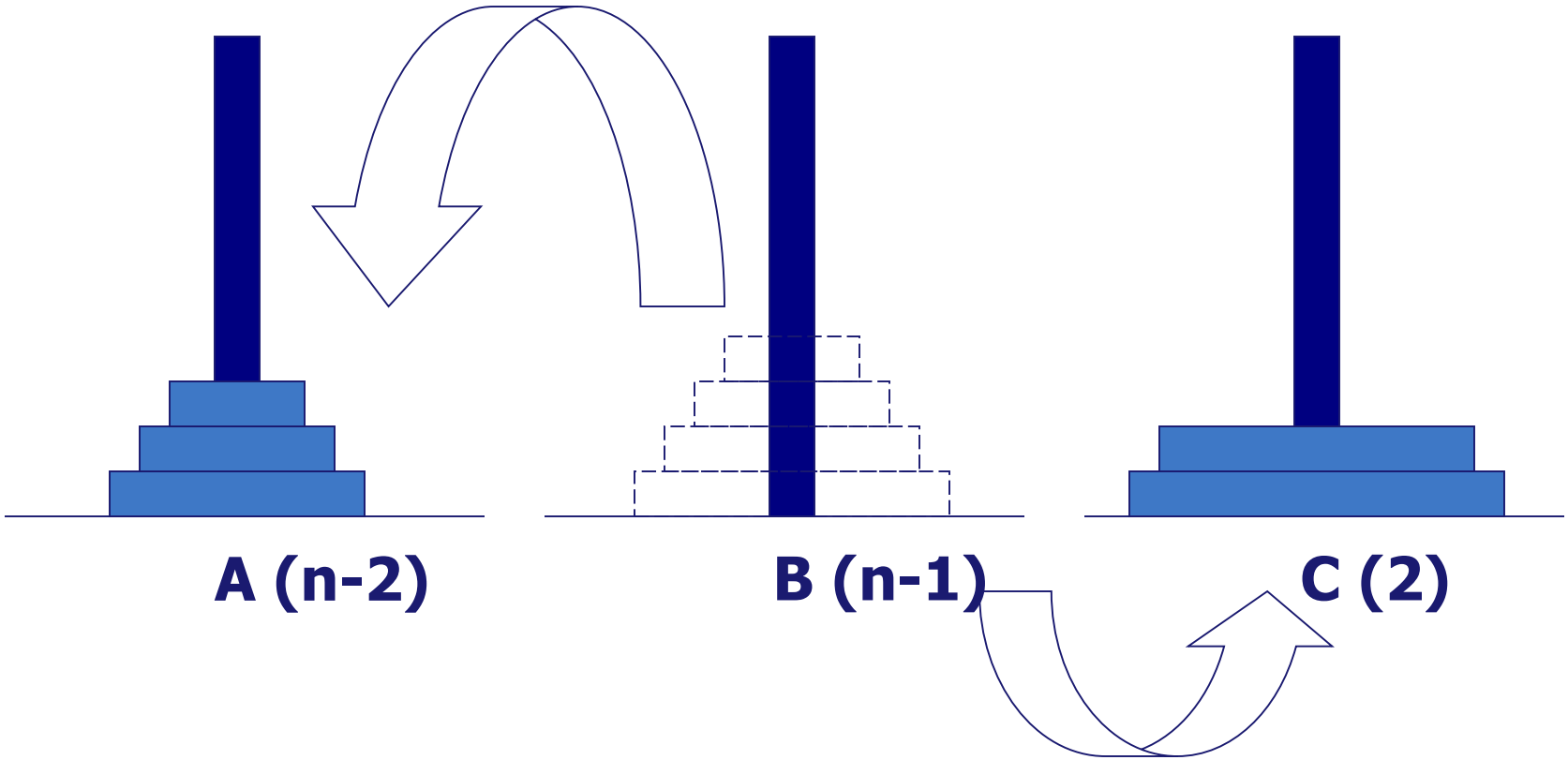
Bài toán tháp Hà Nội

- A → C, B trung gian



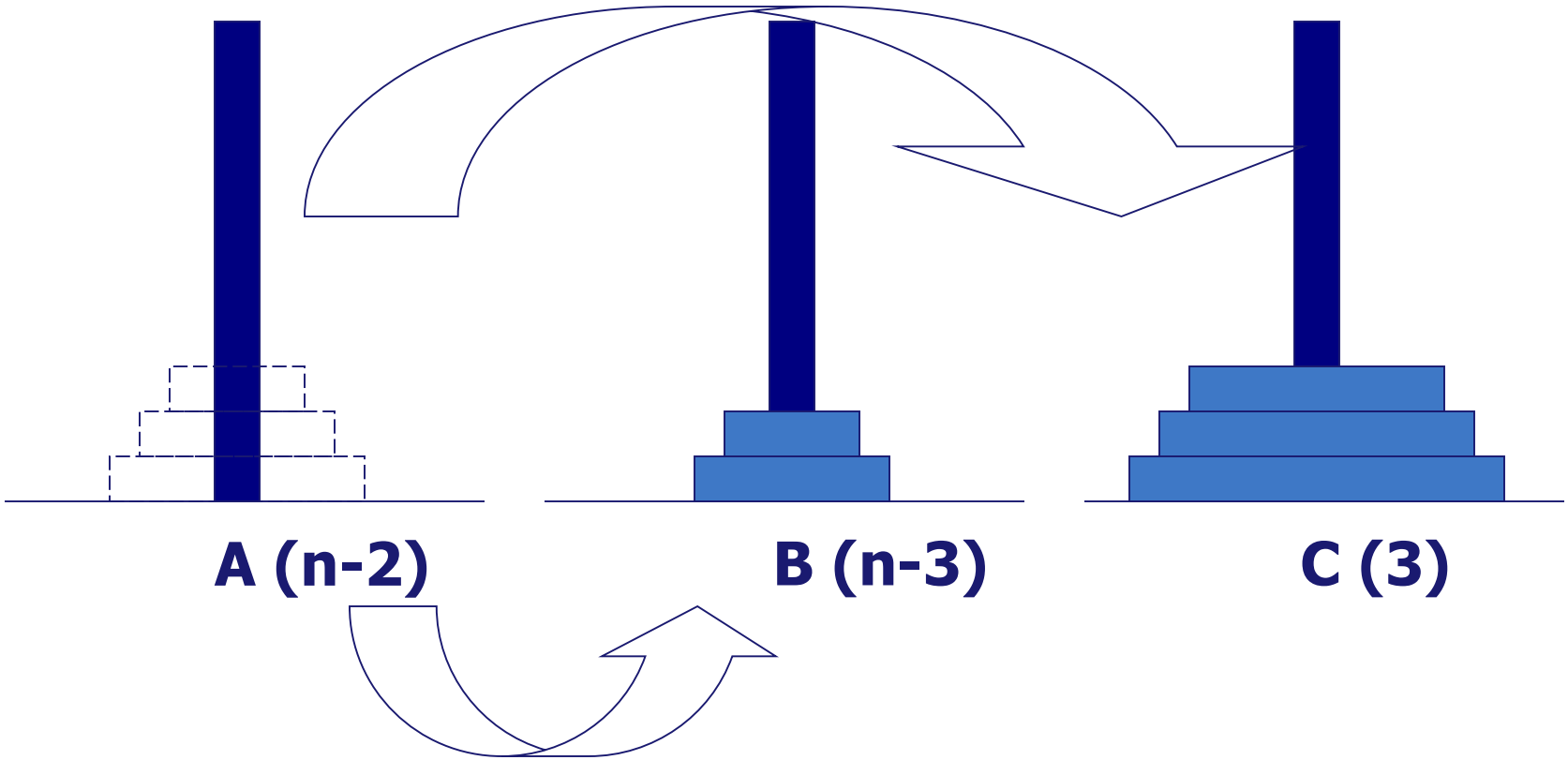
Bài toán tháp Hà Nội

- B → C (A trung gian)



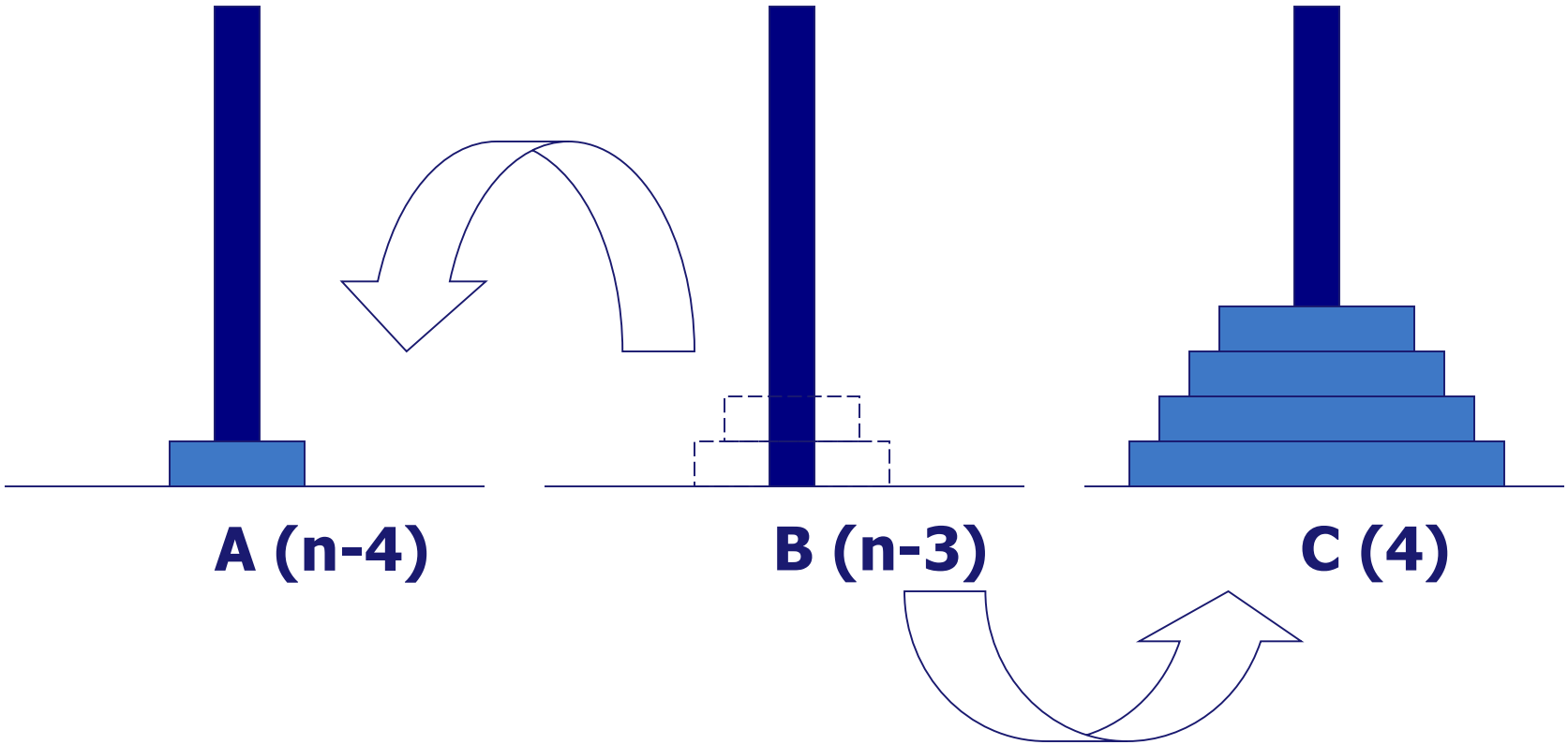
Bài toán tháp Hà Nội

- A → C (B trung gian)



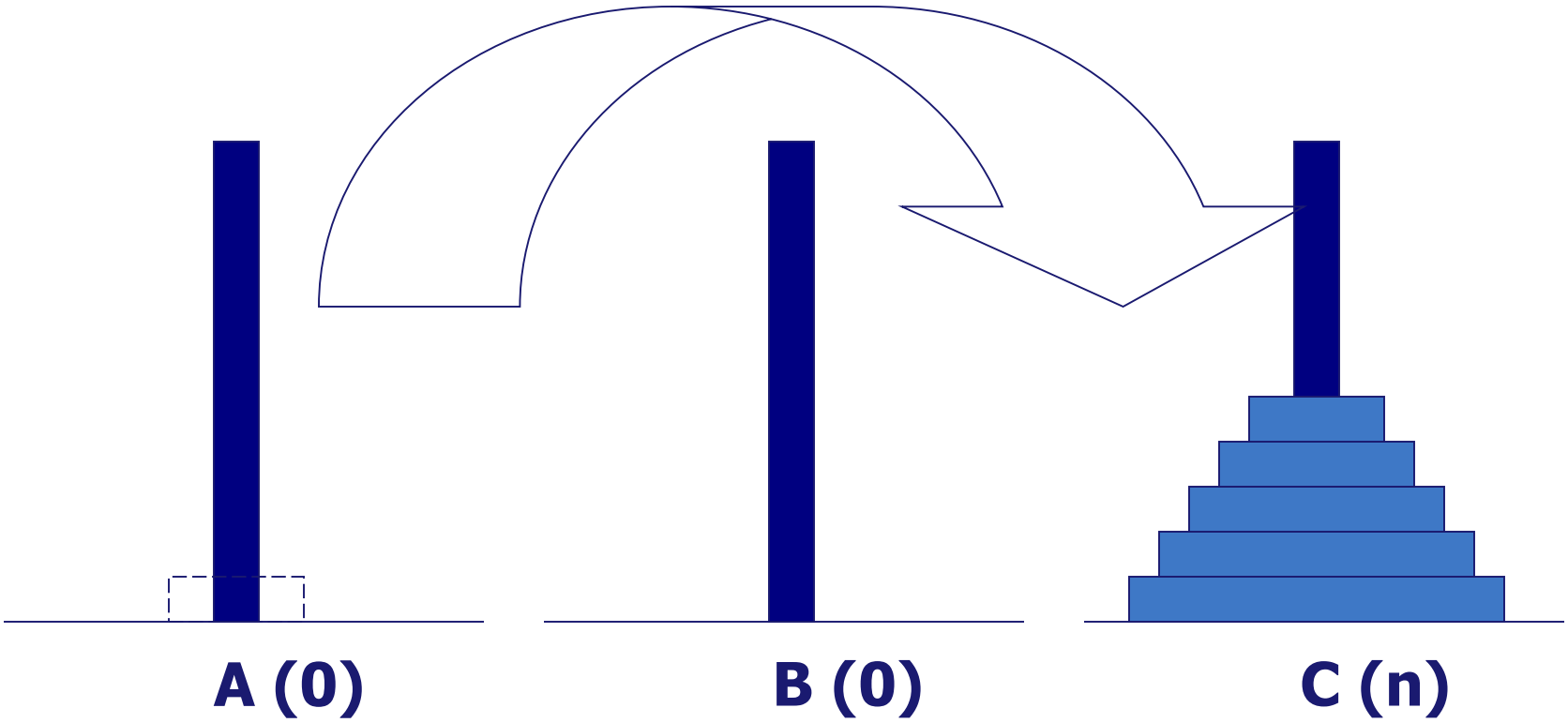
Bài toán tháp Hà Nội

- B → C (A trung gian)



Bài toán tháp Hà Nội

- A → C



Bài toán tháp Hà Nội

```
Void HANOI(int n, char A,B,C){
   if (n==1)
    cout << “chuyển đĩa từ” << A <<“sang”<< C
   else {
        HANOI(n-1,A,C,B);
        HANOI(1,A,B,C);
        HANOI(n-1,B,A,C);
   }
}
```

Bài toán chia thưởng

- Tìm số cách chia m phần thưởng cho n đối tượng học sinh giỏi có thứ tự 1, 2, ..,n. thỏa nguyên tắc
  - Học sinh A giỏi hơn học sinh B, thì số phần thưởng của A sẽ lớn hơn hoặc bằng B
  - Tất cả m phần thưởng đều chia hết cho học sinh
  - Hàm Part(int m,n) là số cách chia
    - Nếu m = 0, có 1 cách chia, tất cả học sinh đều có 0 phần thưởng
    - Nếu n = 0, không có cách chia nào cả

Bài toán chia thưởng

- Khi m < n, thì có n-m học sinh cuối không có phần thưởng, Part(m,n) = Part(m,m)
- Khi m>n, ta xét hai trường hợp
  - Khi học sinh cuối cùng không nhận được phần thưởng nào, dó đó Part(m,n) = Part(m, n-1)
  - Khi học sinh cuối cùng nhận được ít nhất 1 phần thưởng, do đó số cách chia là Part(m-n, n)
  - Tóm lại m > n, có Part(m,n) = Part(m, n-1) + Part(m-n, n)

Bài toán chia thưởng

```
int PART( int m , int n )
{
    if (m == 0 ) return 1 ;
    else
    if (n == 0 ) return 0 ;
    else
    if(m < n ) return ( PART(m , m )) ;
    else
    return ( PART(m , n -1 ) + PART(m -n , n ) ) ;
}
```

Phương pháp quay lui (back tracking)

- Đặc trưng : là các bước hướng tới lời giải cuối cùng của bài toán hoàn toàn được làm thử.
- Tại mỗi bước
  - Nếu có một lựa chọn được chấp nhận thì ghi nhận lại lựa chọn này và tiến hành các bước thử tiếp theo.
  - Ngược lại, không có lựa chọn nào thích hợp thì làm lại bước trước, xóa bỏ sự ghi nhận và quay về chu trình thử các lựa chọn còn lại

Phương pháp quay lui (back tracking)

- Mô hình bài toán:
  - Tìm $X=(x_1, x_2, ..,x_n)$ thỏa B.
  - Để chỉ ra lời giải X, ta phải dựng dần các thành phần lời giải x_i

Phương pháp quay lui (back tracking)

- Phương pháp: Ta xây dựng $x_1, x_2, ..,x_n$ theo cách sau
  - Đầu tiên, Tập T_1 các ứng cử viên có thể là thành phần đầu tiên của vectơ nghiệm chính là x_1
  - Chọn $x_1 \in T_1$,
  - Giả sử, đã xác định được k-1 phần tử đầu tiên của dãy đó là $x_1, x_2, ..,x_{k-1}$. Cần xác định phần tử kế tiếp x_k
  - Xác định T_k là tập tất cả các ứng viên mà x_k có thể nhận được, có hai khả năng

Phương pháp quay lui (back tracking)

- Nếu T_k không rỗng, ta chọn $x_i \in T_k$ và ta có được nghiệm bộ $(x_1,x_2,\ldots,x_{k-1},x_k)$, đồng thời loại x_k đã chọn khỏi T_k. Sau đó ta lại tiếp tục mở rộng bộ $(x_1,x_2,\ldots,x_k)$ bằng cách áp dụng đệ quy thủ tục mở rộng nghiệm.
- Nếu T_k rỗng, tức là không thể mở rộng bộ $(x_1,x_2,\ldots,x_{k-2},x_{k-1})$, thì ta quay lại chọn phần tử mới x'_{k-1} trong T_{k-1} làm thành phần thứ k-1 của vectơ nghiệm

- **Chú ý**: phải có thêm thao tác "Trả lại trạng thái cũ cho bài toán" để hỗ trợ cho bước quay lui

Phương pháp quay lui (back tracking)

```
Thu(k) {
if (k==n) <ghi nhận bộ giá trị mới>
else
for ( j = 1 → n_k) // Mỗi j thuộc tập T_k
    if ( j chấp nhận được){
            <Xác định x_k theo khả năng j>
            <Ghi nhận trạng thái mới>
             Thu(k+1);
            <Trả lại trạng thái cũ cho bài toán>;
     }
```

Phương pháp quay lui (back tracking)

- Quan tâm:
  - Làm thế nào để xác định được tập T_k, tức là tập tất cả các khả năng mà phàn tử thứ k của dãy $x_1, x_2, .., x_n$ có thể nhận
  - Khi đã có tập $T_{k,}$ để xác định x_k, thấy rằng x_k phụ thuộc vào chỉ số j mà còn phụ thuộc vào $x_1, x_2, .., x_{k-1}$

Bài toán: Liệt kê tất cả các hoán vị của n số tự nhiên đầu tiên

- Đặt N= {1, 2, ..,n}. Hoán vị của n số tự nhiên đầu tiên là một bộ x[0], x[1],..,x[n-1]. Trong đó $x[i] \neq x[j]$, $\forall i,j$ và $x[i] \in \{0..n-1\}$.
- T_1 = N, giả sử đã xác định được x[0], x[1], .., x[k-1]. khi đó, T_k = {1..n}- {x[0], x[1], .., x[k-1]}.
- Ghi nhớ tập T_k , k = 0..n-1, ta cần sử dụng một mảng b[0..n-1] là các giá trị 0, 1 sao cho b[i] = 1 khi và chỉ khi i thuộc T_k

Bài toán: Liệt kê tất cả các hoán vị của n số tự nhiên đầu tiên

```
Thu(int k){
   if (k== n) inkq();
   else
      for (int j=1; j<=n; j++)
            if (b[j])
            {
               x[k] = j;
               b[j] = 0;
               Thu(k+1);
               b[j] = 1;
            }
   }
```

```
int main()
{
  b[j] = 1; // j=1..n-1
  Thu (0);
   return 0;
}
```

Liệt kê dãy nhị phân độ dài n

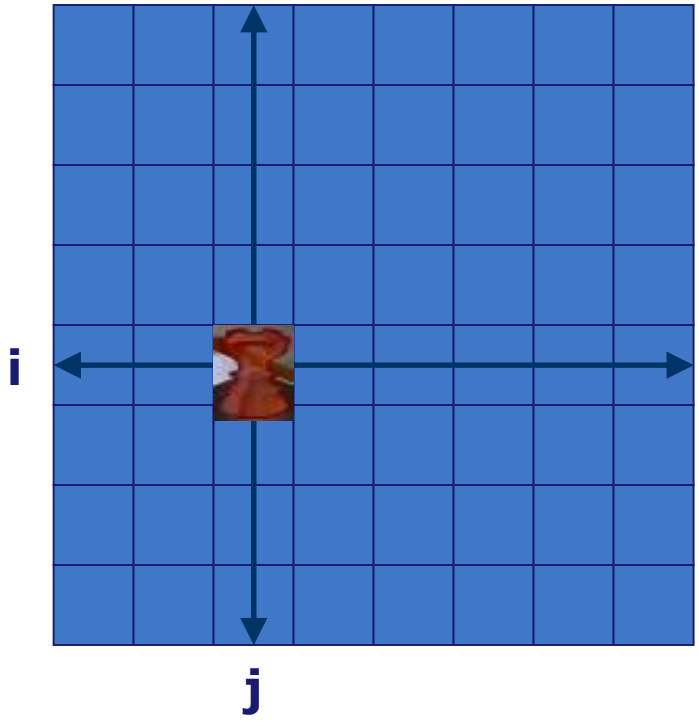
- Chuỗi nhị phân độ dài n có dạng x[0], x[1],..,x[n-1], Đặt B={0,1}
- T_1=B, Giả sử đã xác định được x[0], x[1], .., x[k-1]. Thấy rằng T_k = B

Liệt kê dãy nhị phân độ dài n

```
int x[20] ;
int n, d;
void Thu(int k)
    {
     if (k==n) inkq();
      else
                for (int j = 0; j <=1; j++)
                {
                x[k] = j;
                Thu(k+1);
                }
       }
```

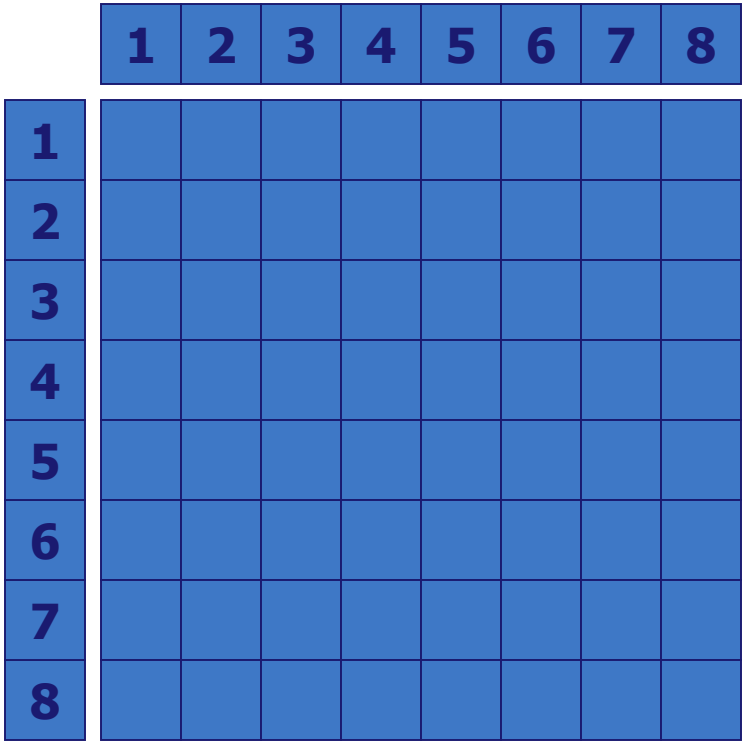
Bài toán 8 quân xe

- Sắp xếp 8 quân xe trên bàn cờ 8x8 sao cho chúng không 'ăn' lẫn nhau
  (mỗi hàng, mỗi cột, có đúng một quân)



Bài toán 8 quân xe

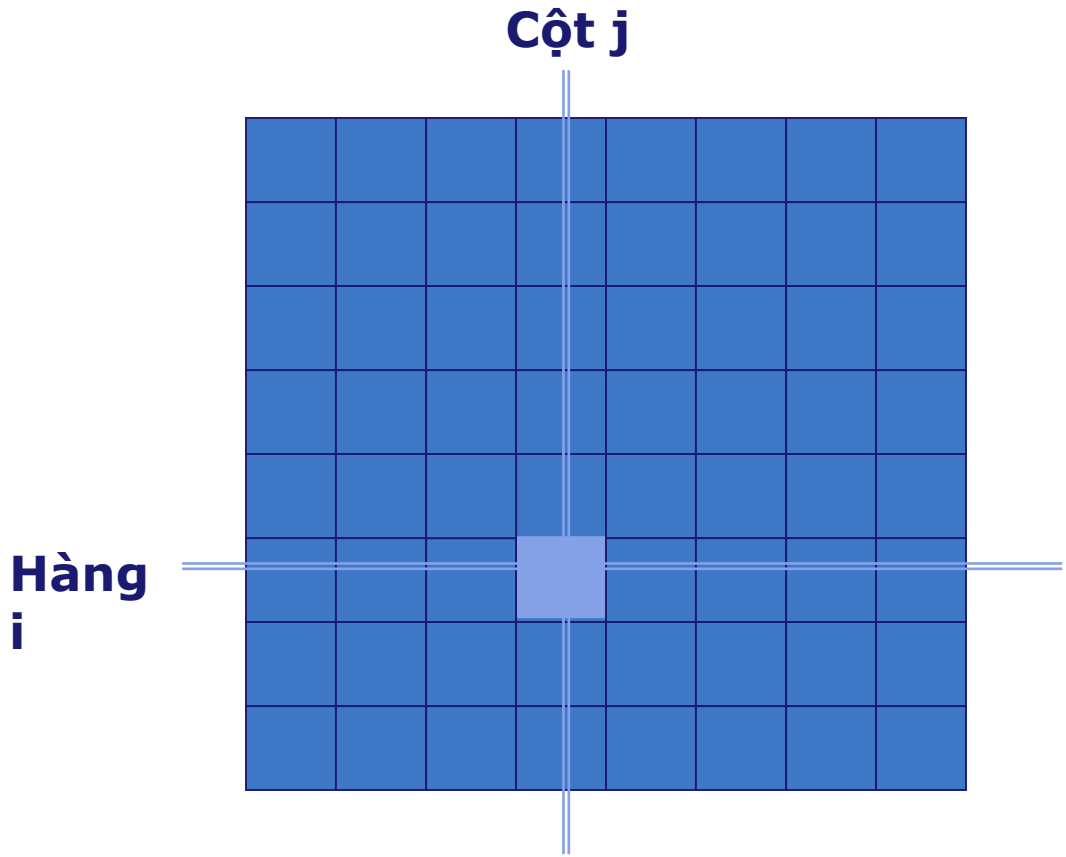
- Đặt quân xe thứ i vào cột thứ j sao cho nó không bị 'ăn' bởi i-1 quân xe hiện có trên bàn cờ
- Mỗi hàng chỉ có 1 quân xe, Nên việc chọn vị trí quân xe thứ i, chỉ nằm trên hàng i



Bài toán 8 quân xe

- Qui ước x[i]: chỉ quân xe thứ i năm ở hàng i
- X[i] = j, quân xe thứ i đặt ở cột j
- Để quân xe i (hàng i) chấp nhận cột j, thì cột j phải tự do.

Bài toán 8 quân xe



Bài toán 8 quân xe

- Do đó ta sẽ chọn các mảng Boole 1 chiều để biểu diễn các trạng thái này
  - a[j] = 1 : Có nghĩa là không có quân xe nào ở cột j.
  - 1<= i, j <=8

Bài toán 8 quân xe

int x[8], a[8],

- Với các dữ liệu đã cho, thì lệnh đặt quân xe sẽ thể hiện bởi :
  - x[i] = j: đặt quân xe thứ i trên cột j.
  - a[j] = 0: Khi đặt xe tại cột j

Bài toán 8 quân xe

- Lệnh dời quân xe là
  - a[j] = 1 ; // cột j tự do
- Còn điều kiện an toàn là ô có tọa độ (i,j) nằm ở cột chưa bị chiếm:
  - (a[j] == 1)

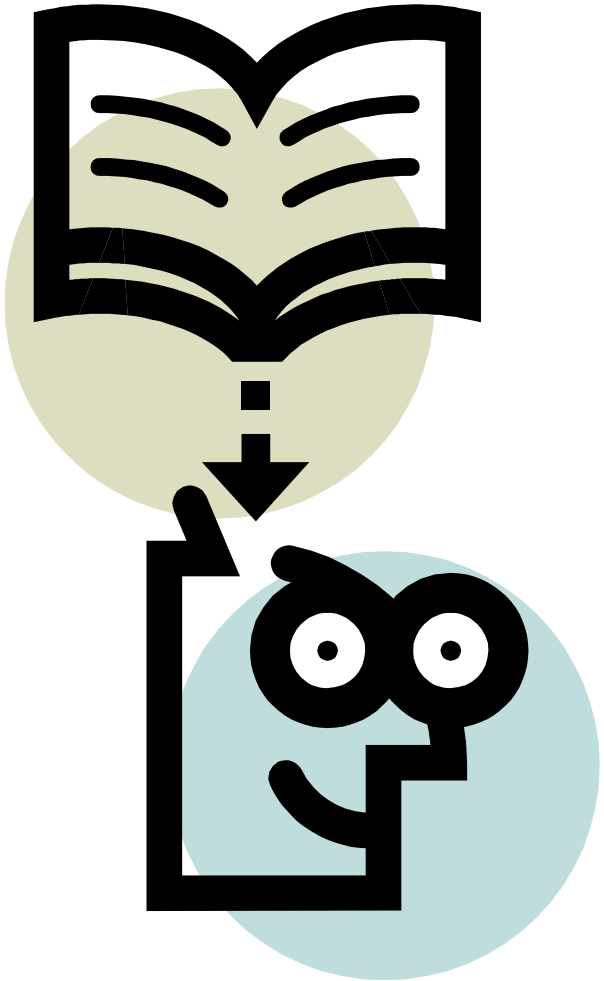
Bài toán 8 quân xe

```
Thu (i){
    If (i >8)
       Xuat (X)
    else
   for (j = 1; j <= 8; j++)
       if (a[j]) {
            x[i] = j;
             a[j] = 0;
           Thu (i+1);
           a[ j ] = 1
       }
   }
```

Đệ quy và quy nạp toán học

- Dùng đệ quy để giải các bài toán truy hồi
- Dùng quy nạp toán học để chứng minh tính đúng đắn, xác định độ phức tạp của giải thuật đệ quy

Q&A



Chương 2
CẤU TRÚC DỮ LIỆU

LOGO

Nội dung trình bày

- Danh sách và các phép toán trên danh sách
- Danh sách đặc
  - Định nghĩa, Cách biểu diễn và các phép toán
  - Ưu và nhược điểm của danh sách đặc
  - Tổ chức Stack và Queue theo kiểu danh sách đặc
- Danh sách liên kết
  - Khái niệm , Biểu diễn, Các phép toán
  - Ưu và nhược điểm
  - Tổ chức Stack và Queue theo kiểu danh sách liên kết
- Danh sách liên kết kép

Danh sách

- Định nghĩa danh sách
  - Danh sách là dãy hữu hạn có thứ tự bao gồm một số biến động các phần tử thuộc cùng một lớp đối tượng nào đó.
  - Mô tả danh sách : $L = (a_1, a_2, \ldots, a_n)$
  - Danh sách tuyến tính: là danh sách mà quan hệ lân cận giữa các phần tử được hiển thị

Lưu trữ danh sách

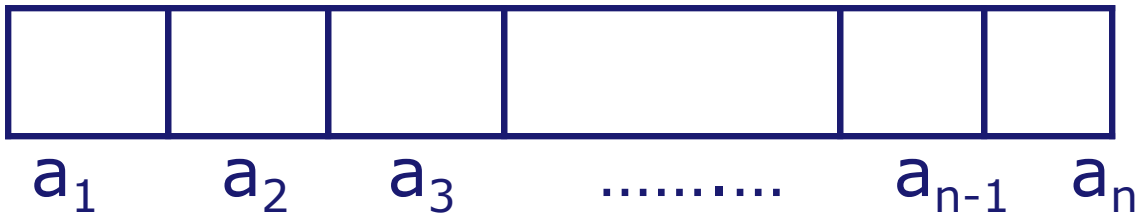
- Tổ chức lưu trữ danh sách trong bộ nhớ
  - Sử dụng mảng - Danh sách đặc
  - Đối tượng lớp - danh sách liên kết
    - Mỗi node là một đối tượng lớp

Các phép toán trên danh sách

- Thêm
- Loại bỏ
- Sắp xếp:
- Tìm kiếm
- Tách
- Ghép
- Duyệt:

Danh sách đặc (condensed list)

- Định nghĩa
  - Là danh sách có các phần tử được xếp kế tiếp nhau trong bộ nhớ



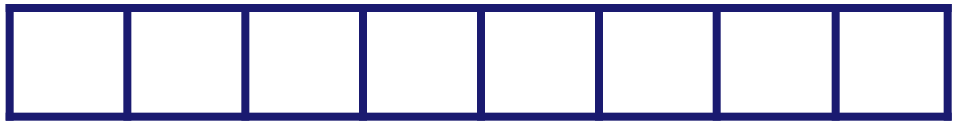
- Đặc điểm
  - d: chiều dài mỗi phần tử trong danh sách
  - l_0: địa chỉ của phần tử đầu tiên
    - ⇨ địa chỉ của phần tử thứ i là: $l_i=l_0+(i-1)d$

Danh sách đặc (condensed list)

- Ưu điểm
  - Mật độ sử dụng 100%
  - Dễ dàng truy xuất đến phần tử thứ i, thông qua chỉ mục
- Nhược điểm
  - Độ phức tạp thuật toán thêm/bớt một phần tử vào/ra danh sách là khá cao $T(n)_{max}=O(n)$
  - Lãng phí khi trong danh sách có nhiều phần tử cùng giá trị

Mảng danh sách đặc phổ biến

- Mảng một chiều a[ ]



Hình ảnh mảng

Hình ảnh một biến

- Khai báo:
  - Cách 1: <Kiểu dữ liệu> [] tên_mảng;
  - Tên_mảng = new <Kiểu dữ liệu>[size];
  - Ví dụ:
    - int[] myIntArray; myIntArray = new int[5];
    - int[] numbers; numbers = new int[] {0,1,2,3,4};

Mảng 2 chiều

- Mảng hai chiều a[,]
  - Khai báo mảng 2 chiều:
    int[,] grades = new int[2,3]; // 2 hàng, 3 cột

  - Truy cập phần tử của mảng <Tên mảng>[dòng, cột]

Khởi tạo mảng 2 chiều

```
int[,] grades = new int[,] {{1, 82, 74, 89, 100},
                            {2, 93, 96, 85, 86},
                            {3, 83, 72, 95, 89},
                            {4, 91, 98, 79, 88}}
```

Ví dụ cài đặt danh sách

```
class CArray {
   private int [] arr;
   private int upper;
   private int numElements;
   public CArray(int size) {
      arr = new int[size];
      upper = size-1;
      numElements = 0;
    }
```

Mảng danh sách đặc phổ biến

```
public void Insert(int item) {
   arr[numElements] = item;
   numElements++;
   }
public void DisplayElements() {
   for(inti=0;i<= upper; i++)
   Console.Write(arr[i]+"");
   }
```

Mảng danh sách đặc phổ biến

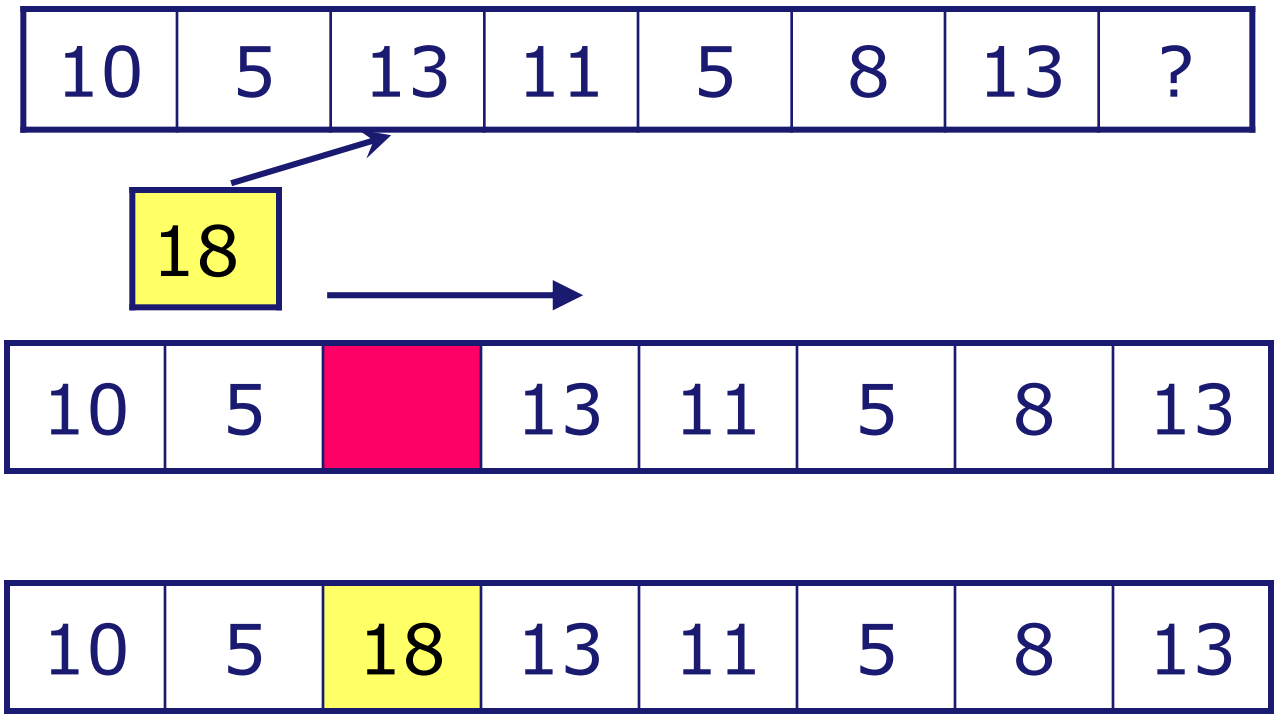
```
static void Main() {
   CArray nums = new CArray();
   for(inti=0;i<=49; i++)
   nums.Insert(i);
   nums.DisplayElements();
}
```

Mảng danh sách đặc phổ biến

```
static void Main() {
   CArray nums = new CArray();
   Random rnd = new Random(100);
   for(inti=0;i<10; i++)
   nums.Insert((int)(rnd.NextDouble() *
   100));
   nums.DisplayElements();
   }
```

Cài đặt danh sách bằng mảng

- Thêm một phần tử vào mảng



Thêm phần tử vào mảng

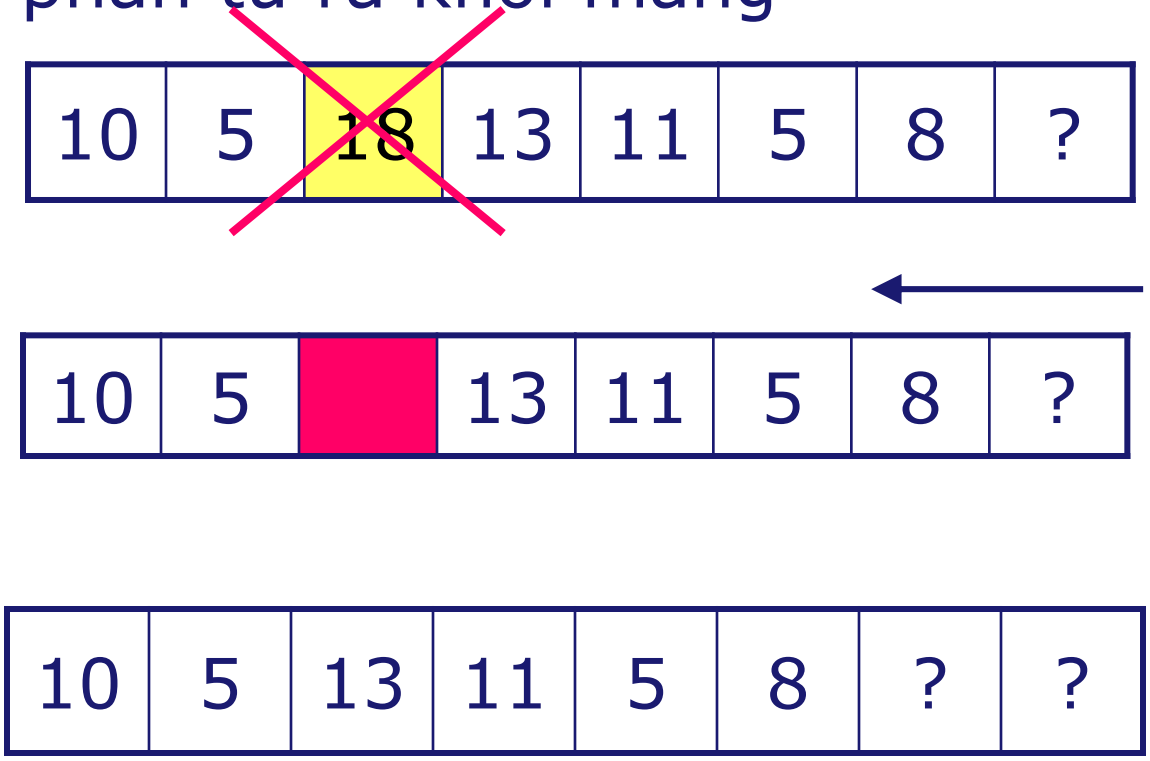
- Hàm Thêm vào mảng một giá trị x tại vị trí vt (kiểm tra tính hợp lệ của vt)
- Kiểm tra nếu k ∈[0, n] thì:
  - Dời các phần tử từ vị trí n-1 đến k lùi lại 1 vị trí.
  - Thêm x vào vị trí thứ k của mảng, tăng n thêm 1.

Thêm phần tử vào mảng

```
void ThemPhanTu(int a[], int &n, int x, int vt)
{
if(vt>=0 && vt<=n)
{
for(int i=n; i>vt; i--)
a[i] = a[i-1]; //Dịch các phần tử sang phải 1 vị trí
a[vt]=x; //Thêm x vào vị trí vt
n++; //Tăng số phần tử lên 1
}
else
Cout<<"Vi tri them khong hop le";
}
```

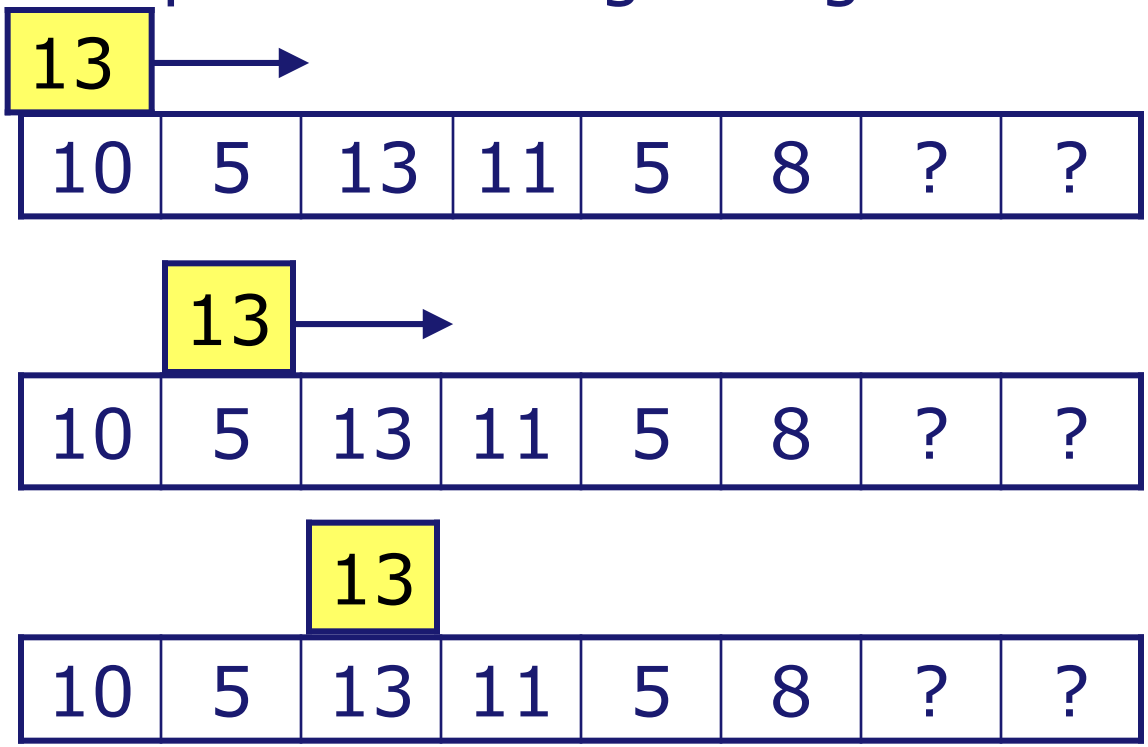
Cài đặt danh sách bằng mảng

- Xóa phần tử ra khỏi mảng



Cài đặt danh sách bằng mảng

- Tìm kiếm phần tử trong mảng



Xóa phần tử trong mảng

Để xóa một phần tử trong mảng ta phải:

Kiểm tra nếu x có tồn tại trong mảng thì:

Dời các phần tử sau x tới 1 vị trí.

Giảm n bớt 1.

Xóa phần tử trong mảng

Bước 1: tìm vị trí phần tử cần xóa trong mảng

```
int TimPhanTu(int a[], int n, int x)
{
for(int i=0; i<n; i++)
if(a[i] == x)
return i; //Tìm thấy x tại vị trí thứ i
return -1; //Không tìm thấy x trong mảng
}
```

Xóa phần tử trong mảng

Bước 2: xóa phần tử x trong mảng

```
void XoaPhanTu(int a[], int &n, int x)
{
int vt=TimPhanTu(a, n, x); //Tìm vị trí x trong mảng
if(vt==-1)
Cout<<"Khong tim thay phan tu<<x<<" muon xoa.";
else
{
for(int i=vt; i<=n-2; i++)
a[i] = a[i+1]; //Dịch các phần tử sang trái 1 vị trí
n--; //Giảm số phần tử bớt 1
}
}
```

Bài tập

- Nhập một dãy số nguyên từ bàn phím, và sắp xếp chúng theo thứ tự tăng dần

Input: 5 2 4 18 9 1

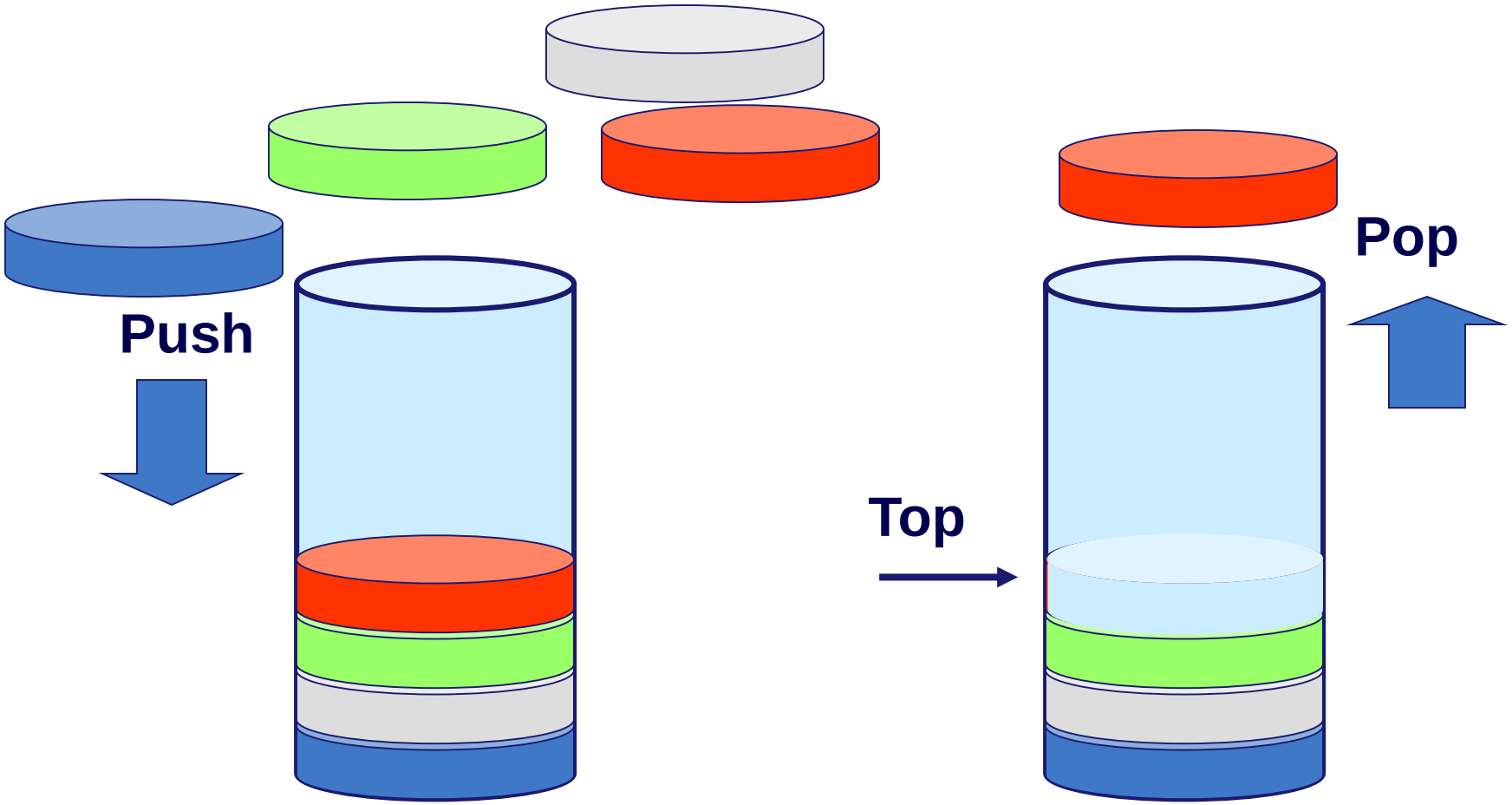
Output: 1 2 4 5 9 18

- Nhập một dãy số nguyên từ bàn phím, và cho biết số lần xuất hiện của từng số trong dãy số

Input: 1 3 2 9 4 3 2 9 8 1 1 3 2 9 1

Ouput: (1,4) (2,3) (3,3) (4,1) (8,1) (9,3)

Tổ chức Stack theo kiểu danh sách đặc

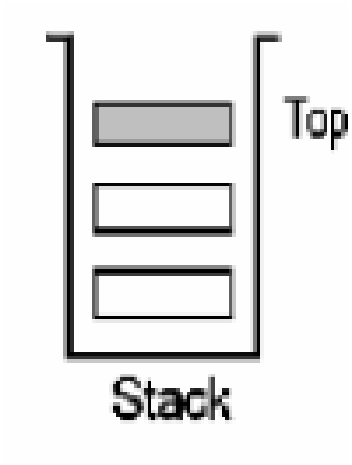


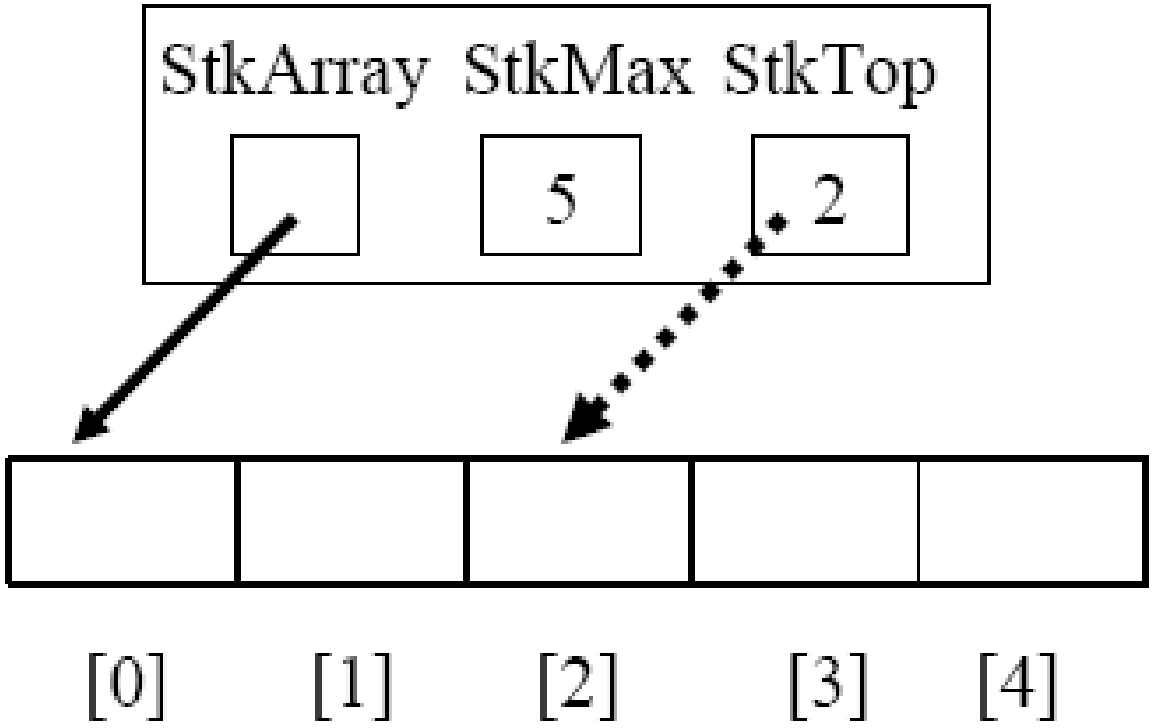
NGĂN XẾP
VÀ HÀNG ĐỢI

Tổ chức Stack theo kiểu danh sách đặc

- Cấu trúc của STACK
  - Dùng 1 mảng (StkArray) để chứa các phần tử
  - Dùng 1 số nguyên (StkMax) để lưu số phần tử tối đa trong Stack
  - Dùng 1 số nguyên (StkTop) để lưu chỉ số đỉnh của STACK

Tổ chức Stack theo kiểu danh sách đặc

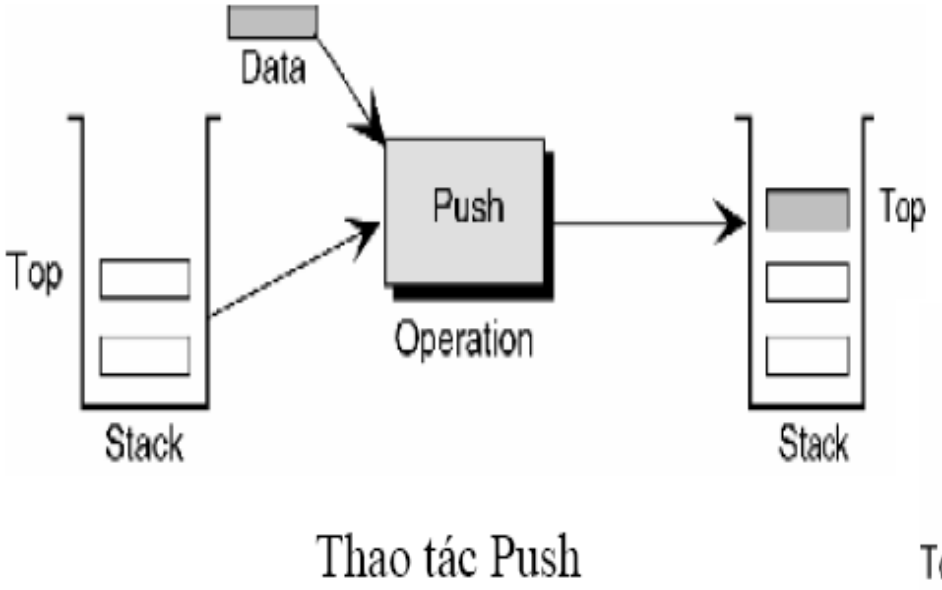




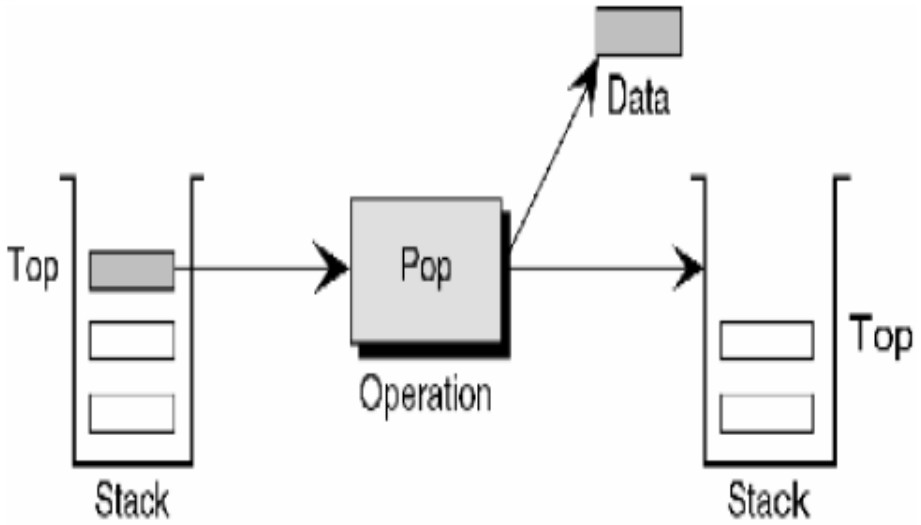
Tổ chức Stack theo kiểu danh sách đặc

- Các thao tác cơ bản trên Stack:
  - Stack: khởi tạo Stack rỗng
  - IsEmpty: kiểm tra Stack rỗng ?
  - IsFull: kiểm tra Stack đầy ?
  - Push: thêm 1 phần tử vào đỉnh Stack, có thể làm Stack đầy
  - Pop: lấy ra 1 phần tử từ đỉnh Stack, có thể làm Stack rỗng

Tổ chức Stack theo kiểu danh sách đặc



Thao tác Push



Thao tác Pop

Khao báo lớp Stack

- **Thao tác "Khởi tạo Stack rỗng"**

```
class Cstack{
   private int [] StkArr;
   private int StkTop;
   private int StkMax;
   public Cstack(int size) {
      StkArr = new int[size];
       StkMax = size;
      StkTop = -1; // Stack rỗng
    }
```

Kiểm tra Stack rỗng

```
boolean IsEmpty()
{
   if (StkTop == -1) return true; // Stack rỗng
   return false; // Stack không rỗng
}
```

Kiểm tra Stack đầy

```
boolean IsFull()
{
   if (StkTop == StkMax-1)
   return true; // Stack đầy
   return false; // Stack chưa đầy
}
```

Thêm một phần tử vào Stack

```
boolean Push(int newitem)
    {
    if (IsFull())
    return false; // Stack đầy, không thêm vào
      được
    StkTop++;
    StkArr[StkTop] = newitem;
    return  true; // Thêm thành công
    }
```

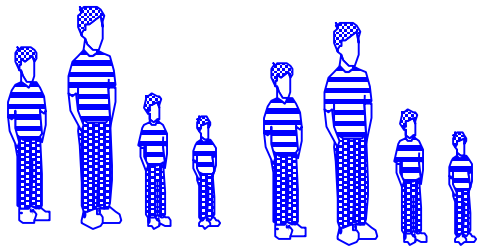
Lấy một phần tử ra khỏi Stack

- Thao tác "Pop": lấy ra 1 phần tử từ đỉnh Stack

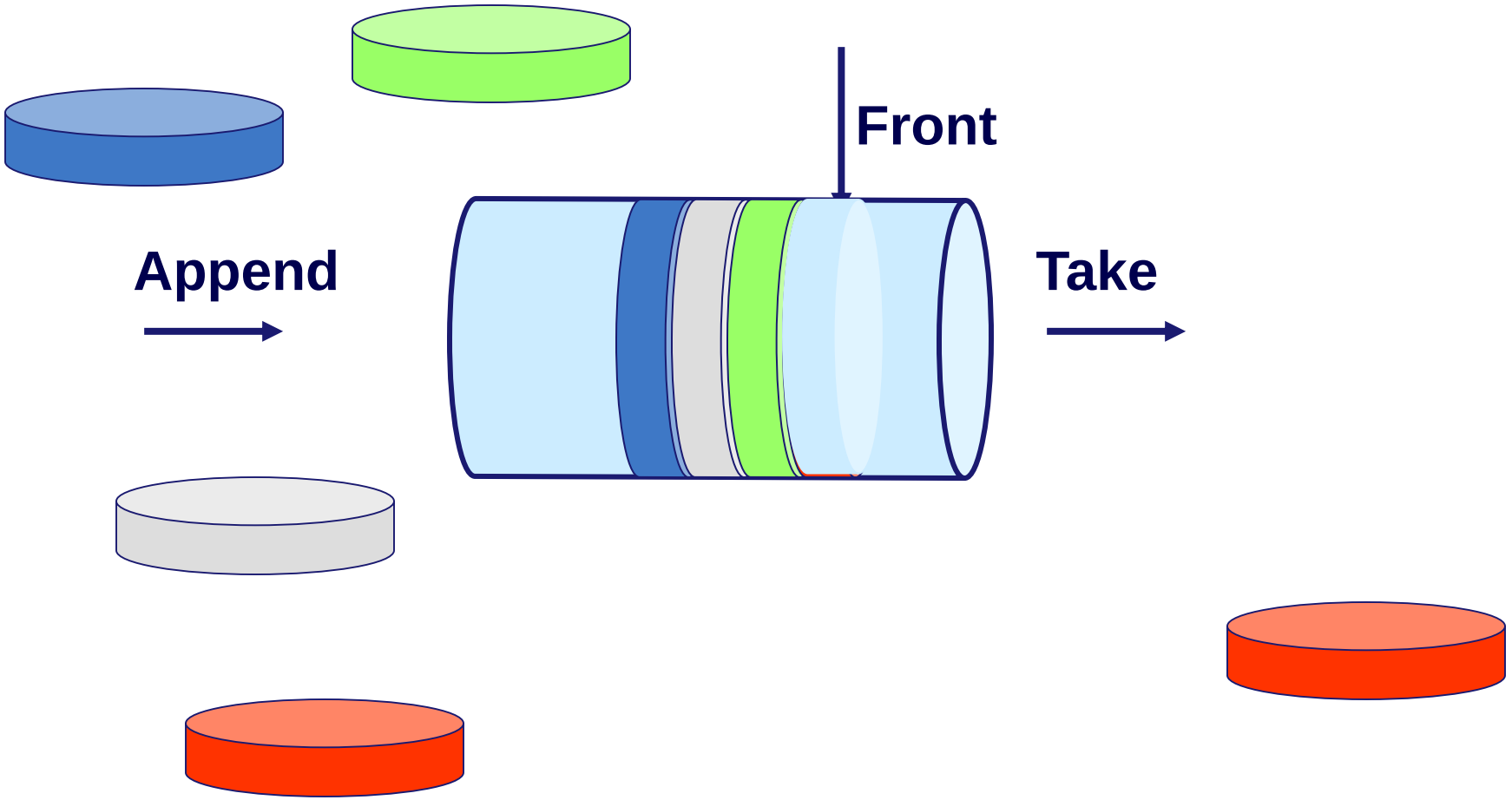
```
boolean Pop(int outitem)
    {
    if (IsEmpty())
    Return false; // Stack rỗng, không lấy ra được
    outitem = StkArr[StkTop];
    StkTop--;
    return true; // Lấy ra thành công
    }
```

Tổ chức Queue theo kiểu danh sách đặc

- Queue là 1 cấu trúc dữ liệu:
  - Gồm nhiều phần tử có thứ tự
  - Hoạt động theo cơ chế "Vào trước – Ra trước" (FIFO – First In, First Out)



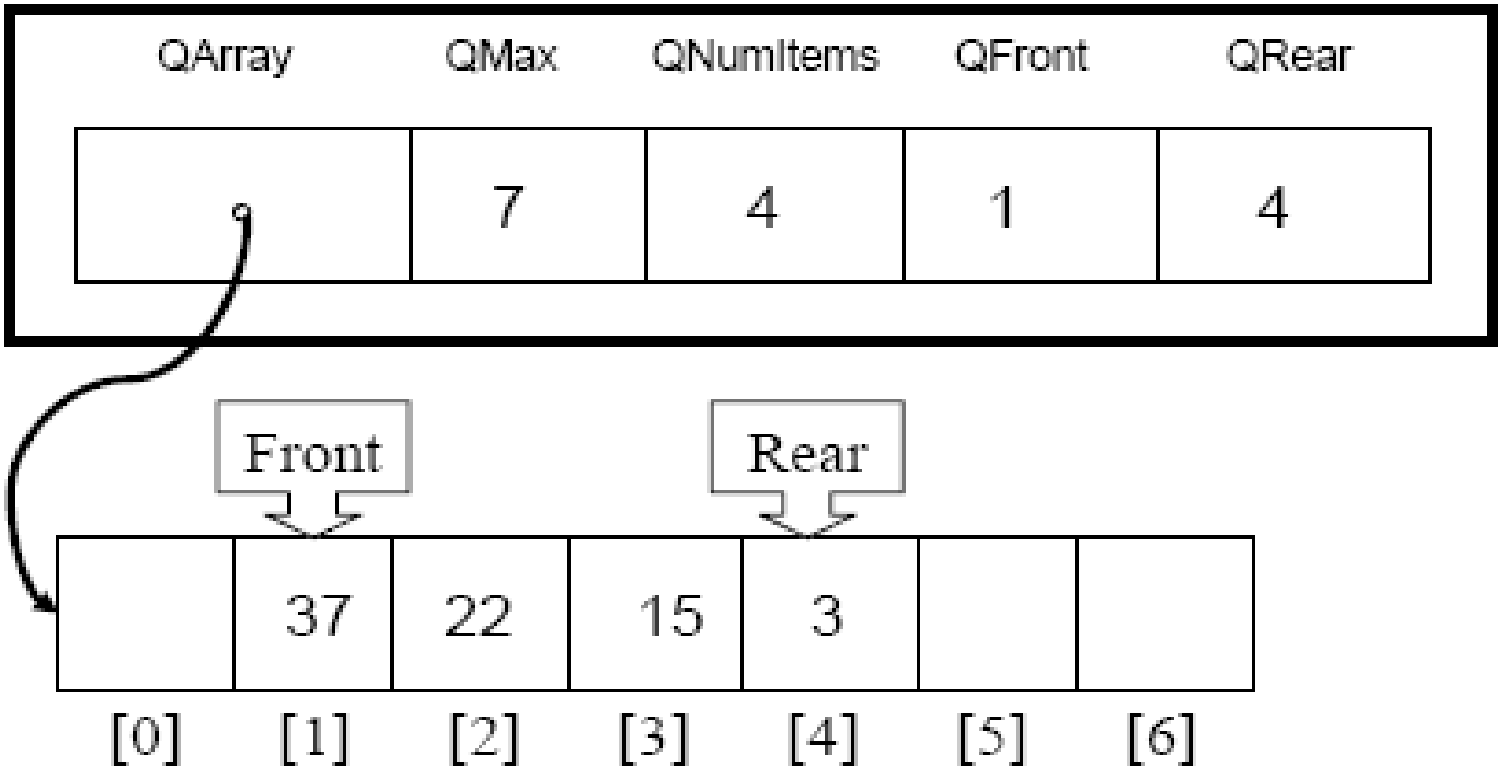
Tổ chức Queue theo kiểu danh sách đặc



Tổ chức Queue theo kiểu danh sách đặc

- Cấu trúc của Queue
  - Dùng 1 mảng (QArray) để chứa các phần tử
  - Dùng 1 số nguyên (QMax) để lưu số phần tử tối đa trong hàng đợi
  - Dùng 2 số nguyên (QFront, QRear) để xác định vị trí Đầu, Cuối hàng đợi
  - Dùng 1 số nguyên (QNumItems) để lưu số phần tử hiện có trong hàng đợi

Tổ chức Queue theo kiểu danh sách đặc



Tổ chức Queue theo kiểu danh sách đặc

- Các thao tác trên Queue
  - Queue: khởi tạo Queue rỗng
  - IsEmpty: kiểm tra Queue rỗng ?
  - IsFull: kiểm tra Queue đầy ?
  - Append: thêm 1 phần tử vào cuối Queue, có thể làm Queue đầy
  - Take: lấy ra 1 phần tử ở đầu Queue, có thể làm Queue rỗng

Khai báo lớp Queue

```
// Giả sử Queue chứa các phần tử kiểu nguyên (int)
Class Queue {
   private int [] QArray;
   private int QMax;
   private int QNumItems;
   private int QFront;
   private int QRear;
```

Khai báo lớp Queue

```
// Khởi tạo Queue chứa các phần tử kiểu nguyên (int)
public Queue(int size) {
    QArray = new int[size];
     QMax = size;
     QFront = Qrear=  -1; // Queue rỗng
    QNumItems = 0; // chưa có phần tử nào trong Queue

  }
```

Kiểm tra Queue rỗng, đầy

- Thao tác "Kiểm tra Queue rỗng"

```
boolean IsEmpty()
{
if (QNumItems==0)
return true; // Queue rỗng
return false; // Queue không rỗng
}
```

- Thao tác "Kiểm tra Queue đầy"

```
boolean IsFull()
{
if (QNumItems == QMax)
return true; // Queue đầy
return false; // Queue không đầy
}
```

Thêm 1 phần tử vào Queue

- Thao tác: thêm 1 phần tử vào cuối Queue

```
boolean Append(int newitem)
    {
    if (IsFull()) return false; //Queue đầy, không thêm vào được
    QRear++;
    QArray[q.QRear] = newitem; // thêm phần tử vào cuối Queue
    QNumItems++;
    return true; // Thêm thành công
    }
```

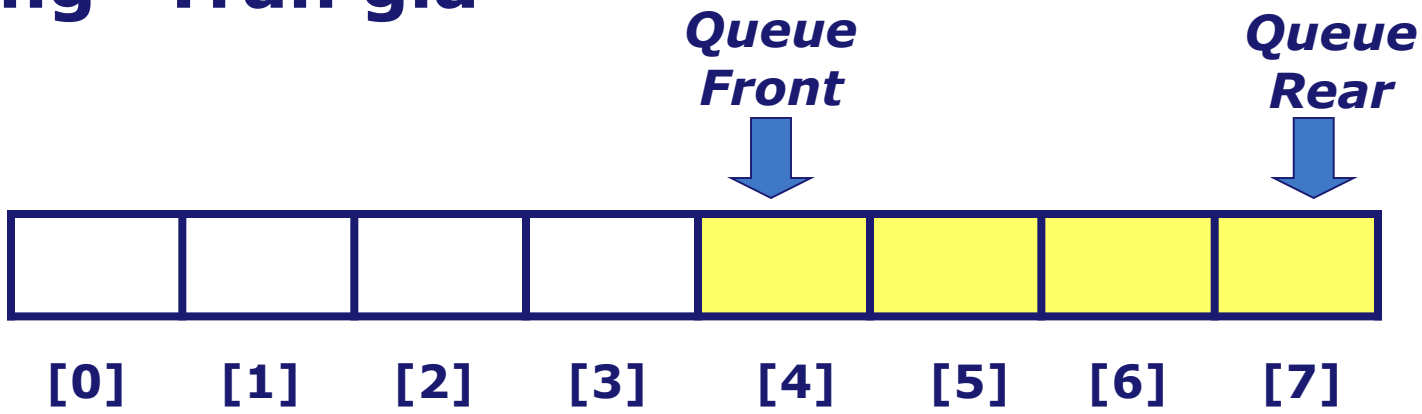
Lấy một phần tử ra khỏi Queue

- Thao tác Take: lấy ra 1 phần tử ở đầu Queue

```
boolean Take(int  itemout)
 {
 if (IsEmpty()) return  false; // Queue rỗng, không lấy ra được
 itemout = QArray[QFront]; // lấy phần tử đầu ra
 QFront++;
 QNumItems--;
 if (QFront==QMax) // nếu đi hết mảng …
 QFront = QRear = -1 ; // … quay trở về đầu mảng
 return true; // lấy thành công
 }
```

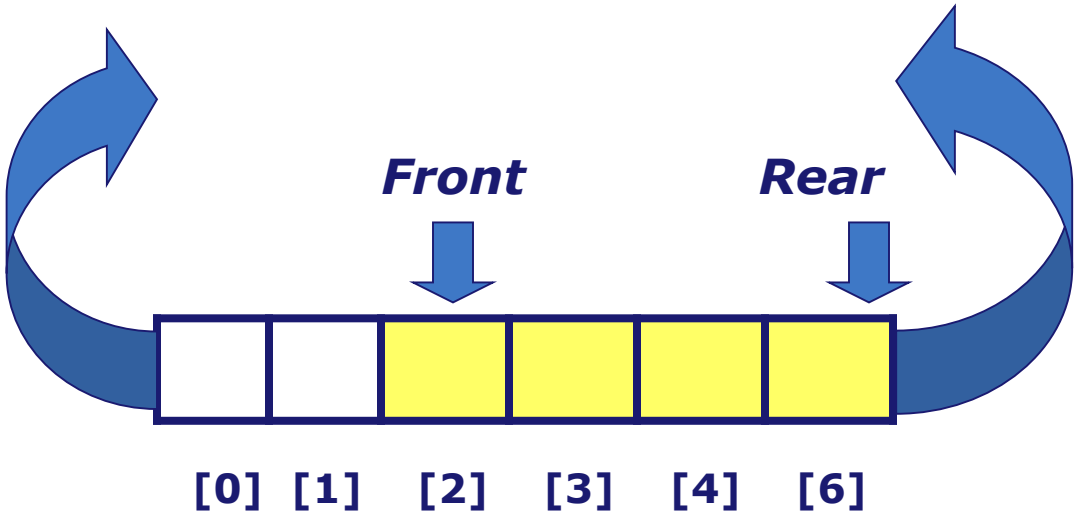
Hạn chế của Queue theo kiểu danh sách đặc

- Khi thêm nhiều phần tử, sẽ làm "tràn" mảng "Tràn giả"

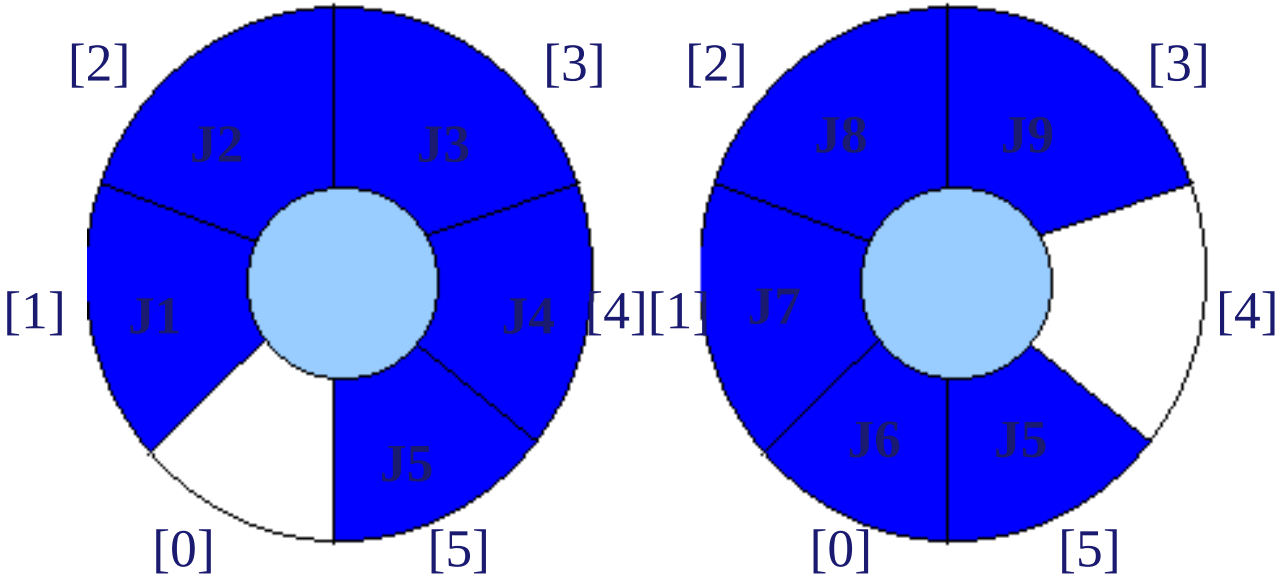


Giải pháp

- Giải pháp cho tình huống "tràn giả": xử lý mảng như là 1 mảng vòng tròn



Tổ chức Queue theo kiểu danh sách đặc



front =0
rear = 5

front =4
rear =3

Bài tập

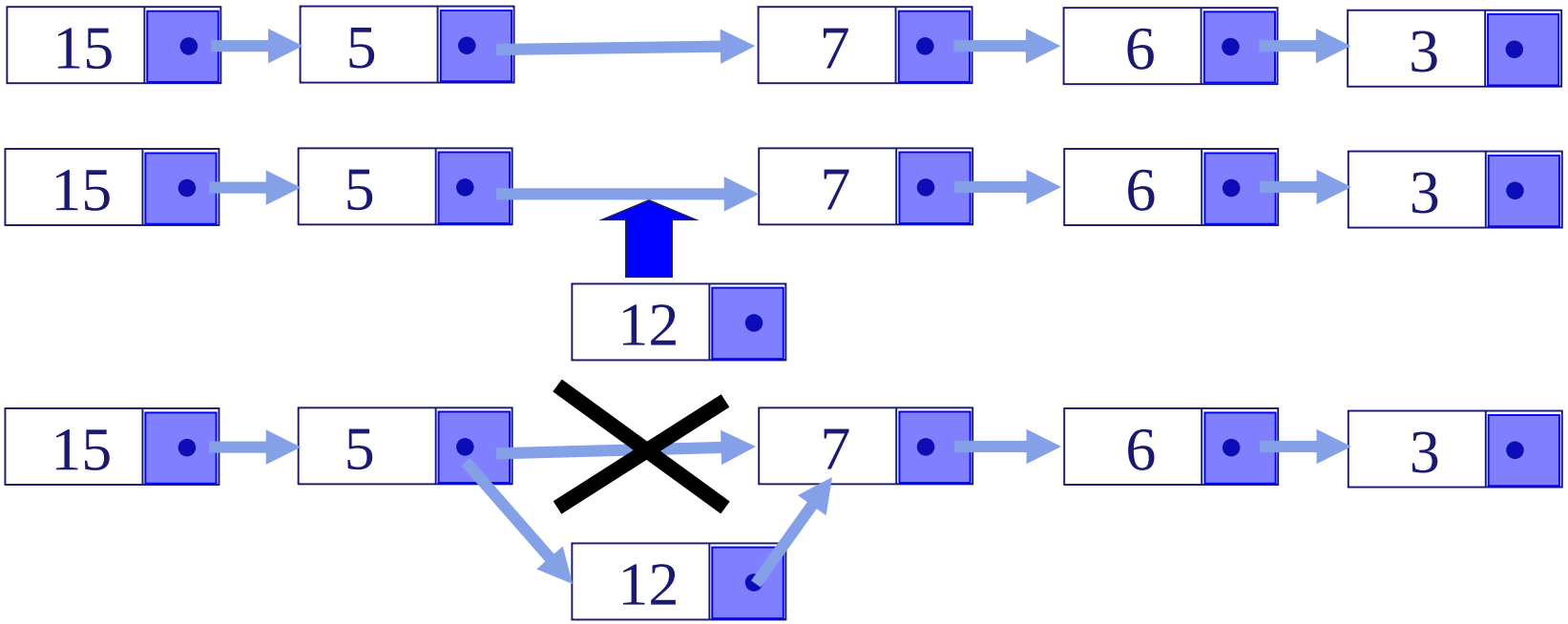
- Viết lại các giải thuật, bổ sung, lấy phần tử cho QUEUE nối vòng

DANH SÁCH LIÊN KẾT

- Đặt vấn đề:
  - Nếu muốn chèn vào 1 phần tử vào mảng ?
    - Chi phí là O(n)
  - Muốn xóa một phần tử trong mảng ?
    - Chi phí O(n)

DANH SÁCH LIÊN KẾT

- Ta tách rời các phần tử của mảng, và kết nối chúng lại với nhau bằng một "móc xích"



DANH SÁCH LIÊN KẾT

- Một dãy tuần tự các nút (Node)
- Giữa hai nút có một tham chiếu
- Các nút không cần phải lưu trữ liên tiếp nhau trong bộ nhớ
- Có thể mở rộng tuỳ ý (chỉ giới hạn bởi dung lượng bộ nhớ)
- Thao tác Chèn/Xóa không cần phải dịch chuyển phần tử
- Quản lý danh sách bởi con trỏ đầu pHead
- Có thể truy xuất đến các phần tử khác thông qua con trỏ liên kết

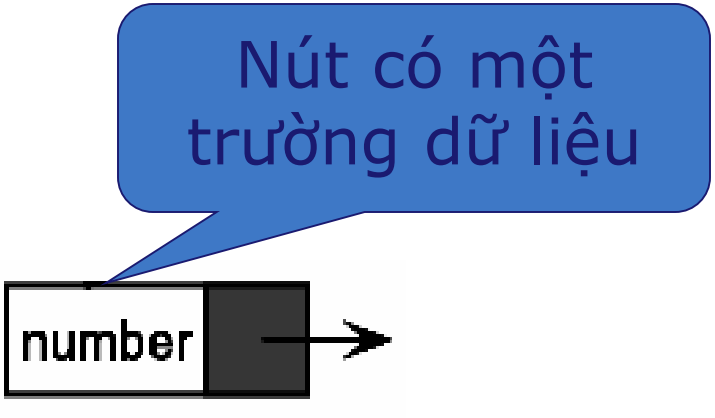
DANH SÁCH LIÊN KẾT

- Cấu tạo nút
  - Tạo lập bằng cách cấp phát bộ nhớ động
  - Mỗi nút có 2 thông tin:
    - Dữ liệu (data)
    - Tham chiếu liên kết đến phần tử kế tiếp trong danh sách (Next pointer link)

DANH SÁCH LIÊN KẾT

- Cấu tạo nút : Gồm 2 thành phần

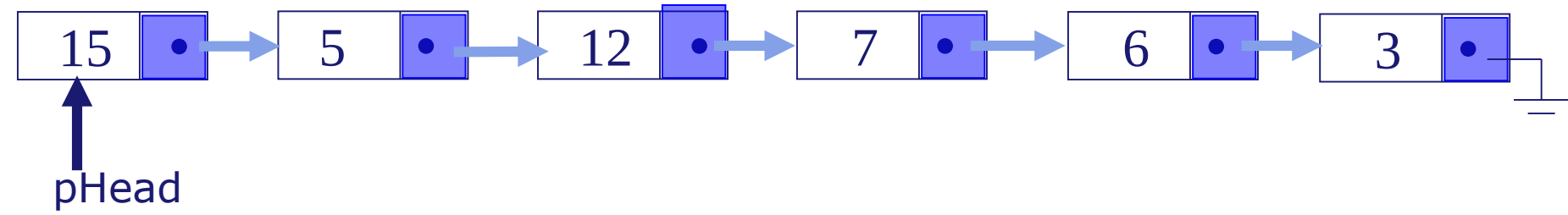
```
public class Node {
    public Object Element;
    public Node Link;
    public Node() {
        Element = null;
        Link = null;
                }
    public Node(Object theElement) {
        Element = theElement;
        Link = null;
                }
    }
```

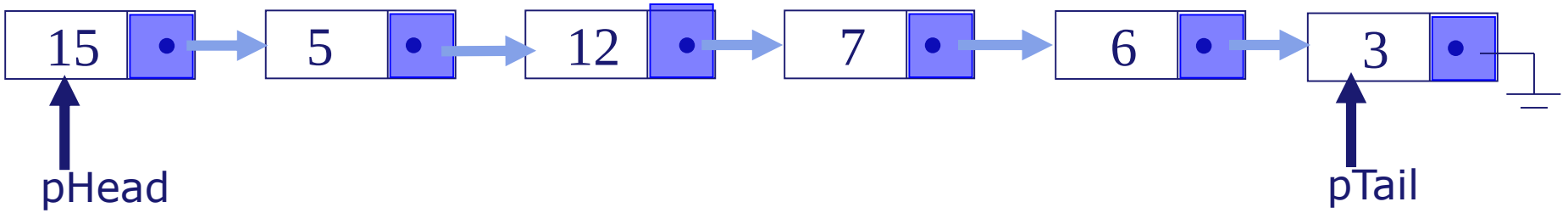


Cấu tạo của danh sách liên kết

- Quản lý danh sách qua con trỏ đầu pHead, có thể thêm con tro cuối pTail
- Có hai cách tổ chức danh sách
  - pHead, pTail là một nút của danh sách
  - pHead, pTail không phải là nút mà chỉ là con trỏ trỏ đến nút đầu và nút cuối của danh sách

Cấu tạo của danh sách liên kết





ƯU VÀ NHƯỢC CỦA DANH SÁCH LIÊN KẾT

- Ưu điểm
  - Tận dụng được không gian nhớ để lưu trữ
    - Các nút không cần lưu trữ kế tiếp nhau
    - Có thể mở rộng kích thước tùy ý (phụ thuộc bộ nhớ)
  - Việc thêm vào hay loại bỏ được tiến hành dễ dàng O(1)
  - Dễ dàng kết nối hay phân rã danh sách
- Nhược điểm
  - Truy xuất tuần tự từng phần tử

Các loại danh sách liên kết

- Danh sách liên kết đơn (Single-Linked list)
  - Mỗi nút chỉ có 1 con trỏ liên kết (pNext)
- Danh sách liên kết đôi (Double-Linked list)
  - Mỗi nút có 2 con trỏ liên kết (pPrev, pNext)
- Danh sách đa liên kết (Multi-Linked list)
  - Mỗi nút có nhiều hơn 2 con trỏ liên kết
- Danh sách liên kết vòng (Circular-Linked list)
  - Liên kết ở nút cuối cùng của danh sách chỉ đến nút đầu tiên trong danh sách

Danh sách liên kết đơn

- Mỗi nút, bao gồm hai phần,
  - Phần Data: chứa dữ liệu, có thể nhiều hơn 1 trường
  - Phần next: chỉ có duy nhất một liên kết đến nút kế tiếp
  - Phần tử cuối cùng có liên kết NULL

MẢNG & DANH SÁCH LIÊN KẾT

- Mảng
  - Phải biết trước số phần tử
  - Lưu trữ tuần tự
  - Khi chèn và xóa phải dịch chuyển các phần tử
  - Truy xuất qua chỉ mục
- Danh sách liên kết
  - Số phân tử tùy biến
  - Sử dụng con trỏ
  - Khi chèn/xóa chỉ cần thay đổi con trỏ liên kết
  - Truy xuất tuần tự

Các thao tác trên danh sách liên kết đơn

- Tạo lập danh sách rỗng
- Kiểm tra danh sách rỗng
- Đếm số phần tử trong danh sách
- Thêm 1 nút vào danh sách
- Xóa 1 nút khỏi danh sách
- Duyệt danh sách
- Tìm 1 phần tử trong danh sách

Tạo lập danh sách rỗng

```
public class LinkedList {
   protected Node header;
   public LinkedList() {
   header = new Node("header");
    }
...
}
```

Tạo lập danh sách rỗng

- Tạo lập danh sách rỗng

```
void Init_List( Node pHead)
    {
      pHead = NULL;
    }
```

Các thao tác trên danh sách liên kết đơn

- Kiểm tra danh sách rỗng

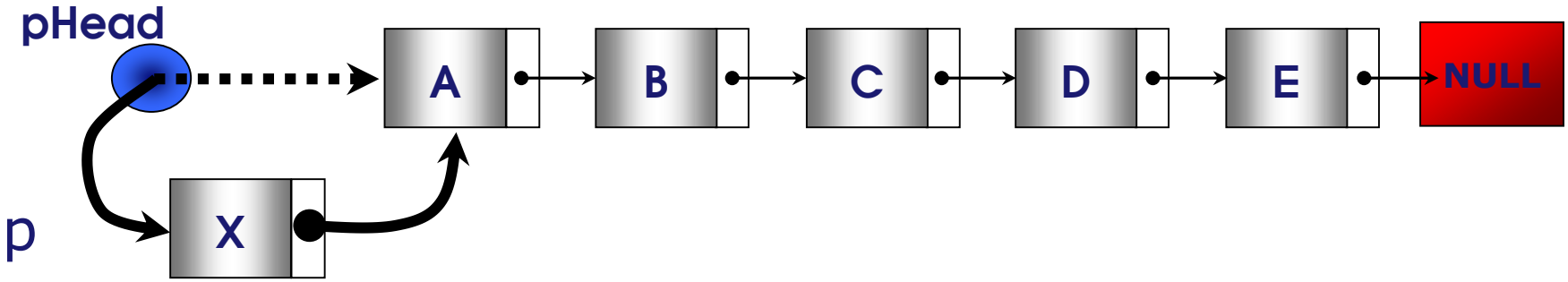
```
boolean IsEmptyList( pHead)
   { return (pHead ==NULL); }
```

- Đếm số phần tử trong danh sách

```
int CountNode(Node pHead){
      int count = 0;
      Node p = pHead;
    while (p != NULL)
         { cout++ ; p = p.Link; }
       return cout;
      }
```

Các thao tác trên danh sách liên kết đơn

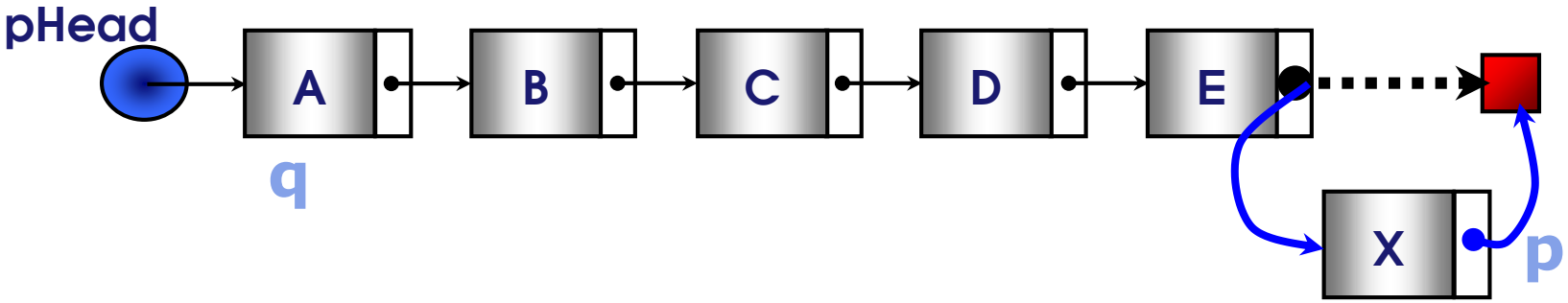
- Thêm một nút p vào đầu danh sách
  - Nếu Danh sách rỗng Thì
    - Gán: pHead = p;
  - Ngược lại
    - p.Link = pHead;
    - pHead = p;



Chèn nút P vào cuối Danh sách

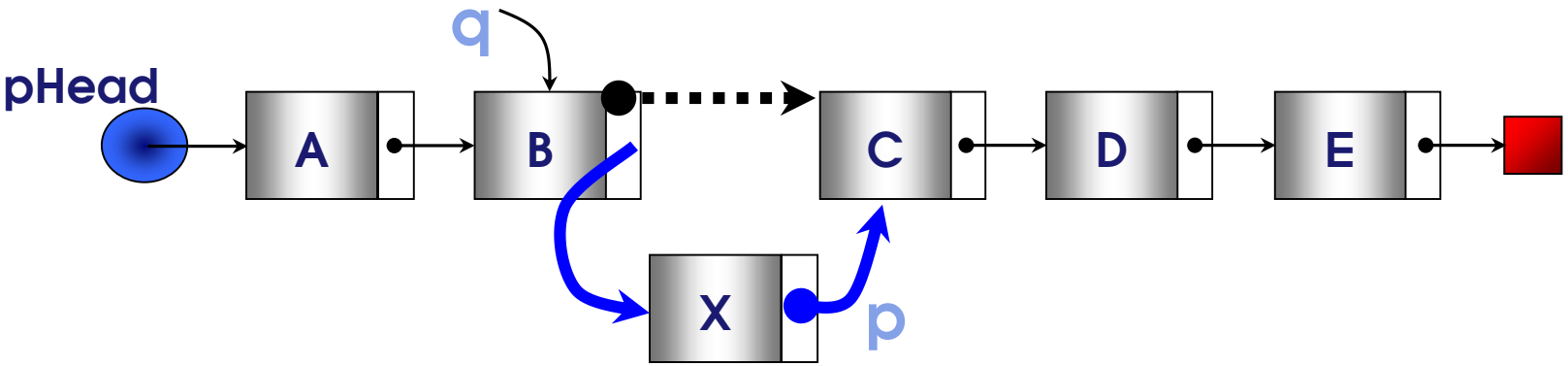
- Chèn nút p vào cuối
  - Nếu Danh sách rỗng Thì
    - Gán: pHead = p;
  - Ngược lại
    - Gán q = pHead;
    - Đi về nút cuối của danh sách

      (while (q.Link != NULL) q = q.Link;)
    - q.Link = p;



Thêm một nút p vào vào sau nút q

- Nếu (q != NULL)
  - P.Link = q.Link;
  - Q.Link = p ;
- Ngược lại: q = p;

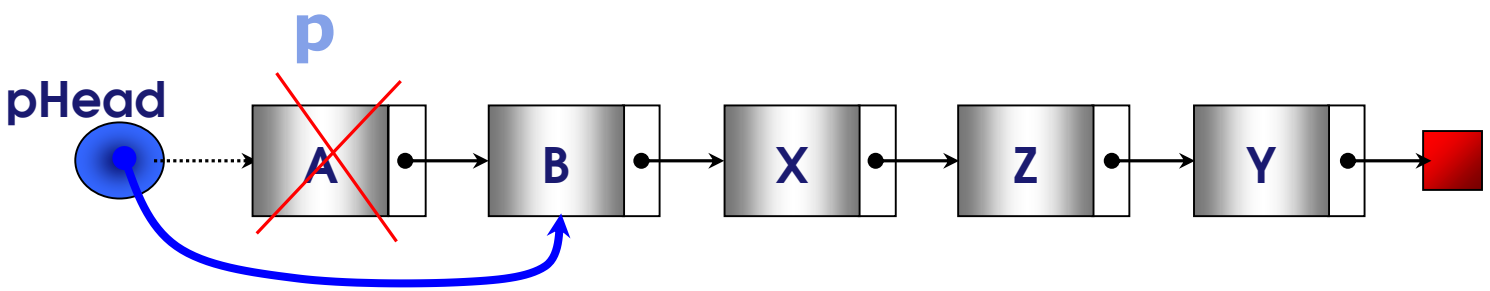


Các thao tác trên danh sách liên kết đơn

- Xóa một nút khỏi danh sách
  - Xóa nút đầu danh sách
  - Xóa một nút đứng sau nút q
  - Xóa một nút có khóa k

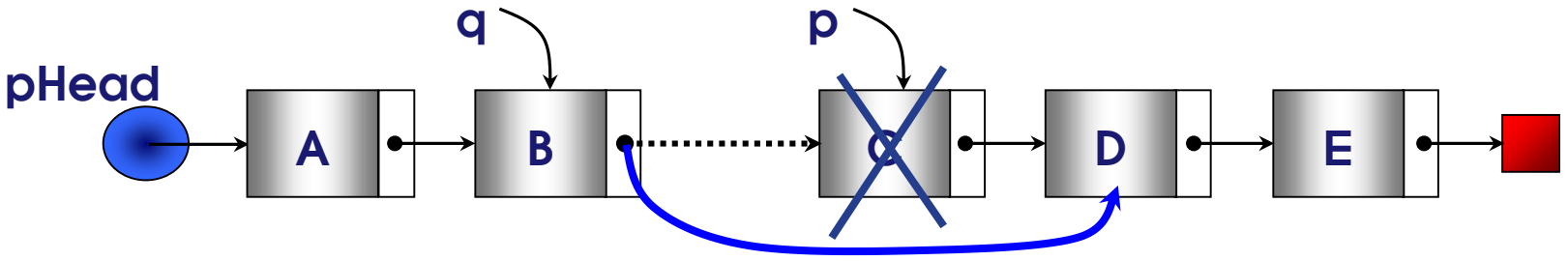
Các thao tác trên danh sách liên kết đơn

- Xóa nút đầu danh sách
  - Nếu (pHead != NULL) thì
    - p = pHead;*// p là nút cần xóa*
    - pHead = pHead.Link;



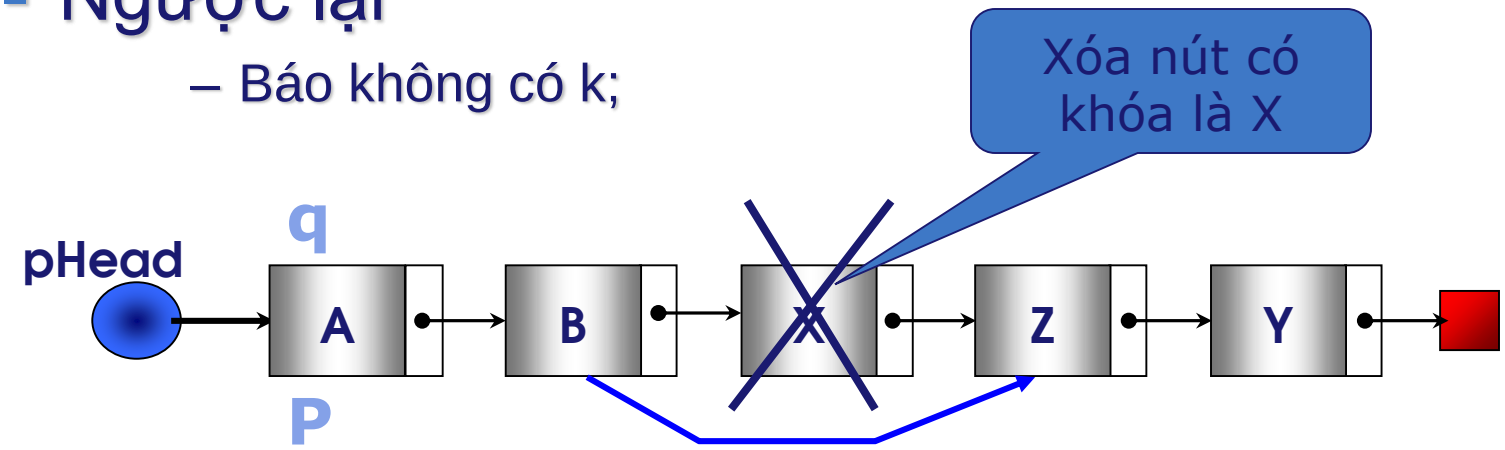
Các thao tác trên danh sách liên kết đơn

- Xóa một nút p, đứng sau q
  - Nếu (q!= NULL) thì
    - p = q.Next; *// p là nút cần xóa*
    - Q.Next = p.Next; *// tách p ra khỏi ds*
    - delete p; *// Giải phóng p*
  - Ngược lại:
    - Xóa nút đầu danh sách



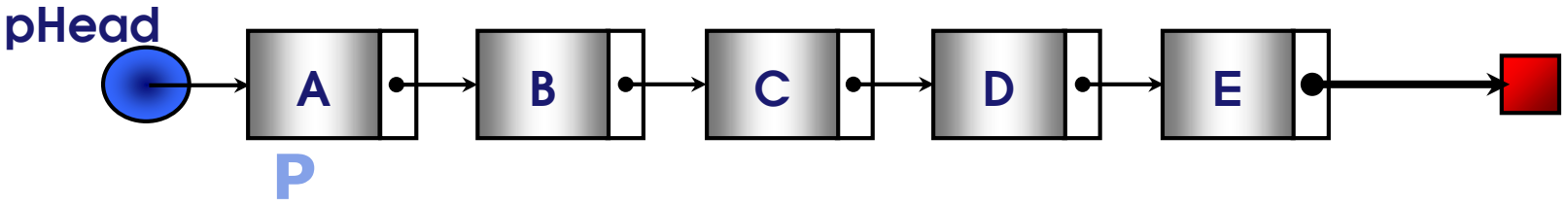
Các thao tác trên danh sách liên kết đơn

- Xóa một nút có khóa k
  - Tìm nút p có khóa k và phần tử q đứng trước nó
  - Nếu (p != NULL) // tìm thấy k
    - xóa p đứng sau nút q;
  - Ngược lại
    - Báo không có k;



Các thao tác trên danh sách liên kết đơn

- Duyệt danh sách
  - p = pHead;
  - Trong khi chưa hết danh sách
    - Xử lý nút p ;
    - p= p.pNext;



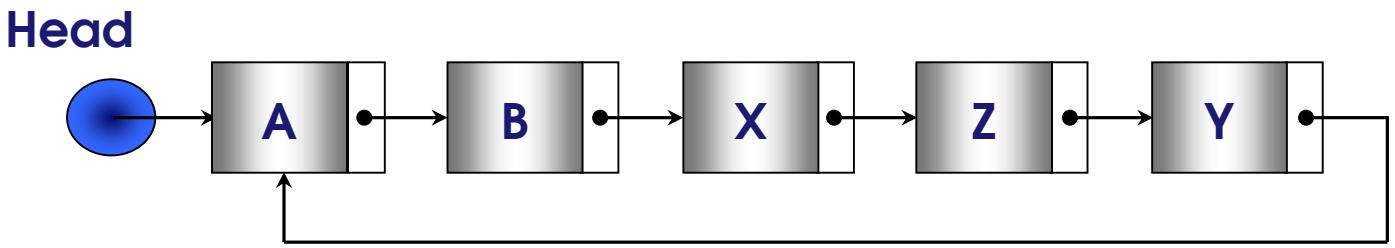
Duyệt danh sách

```
void TraverseList(Node pHead)
 {
  Node p = pHead;
  while (p !=NULL) {
            <Xử lý nút p>;
            p = p.Link; // chuyển sang nút kế
            }
 }
```

Bài tập

- Dùng danh sách liên kết quản lý sinh viên trong lớp (mssv, họ, tên, ngày sinh, điểm tổng kết học kỳ).
- Thực hiện các thao tác: thêm, bớt, sắp xếp sinh viên (theo điểm tổng kết) trong danh sách

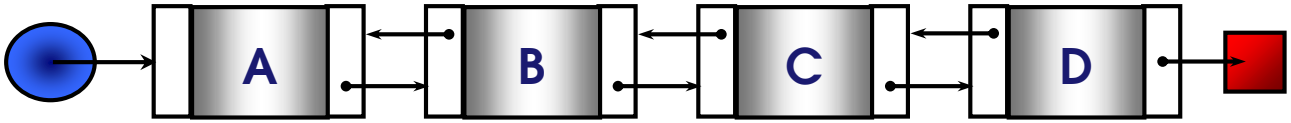
Danh sách liên kết vòng



- Các thao tác trên danh sách liên kết vòng
  - Giải thuật thêm một nút vào đầu danh sách
  - Giải thuật thêm một nút vào danh sách
  - Giải thuật loại một nút ra khỏi danh sách

Danh sách liên kết kép

- Mỗi nút có 2 con trỏ liên kết:



- Khai báo:

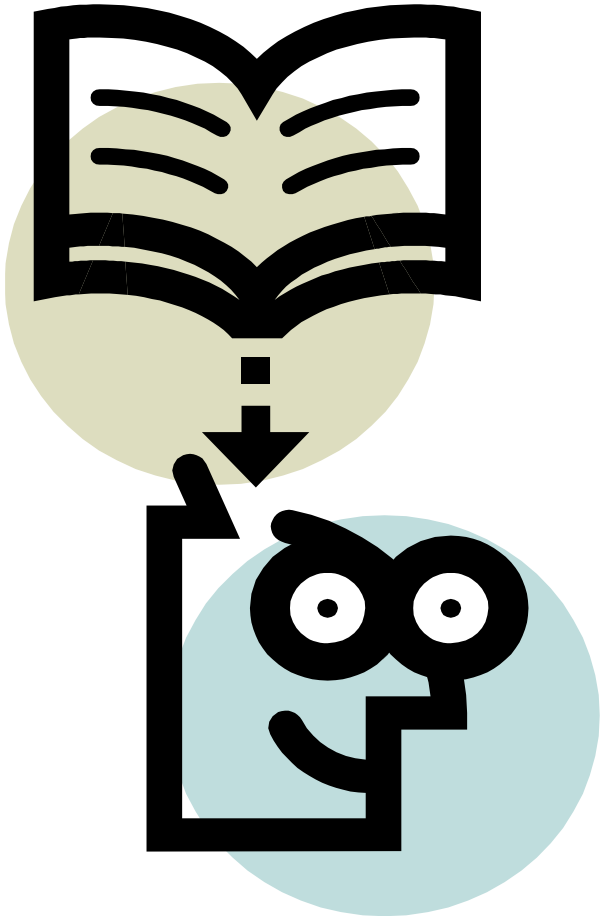
Danh sách liên kết kép

- Khi xóa 1 nút, không cần phải duyệt danh sách để tìm phần tử đứng trước
- Được sử dụng đối với các dữ liệu mà ta cần truy xuất theo cả 2 chiều:
- Bài tập: Viết các giải thuật, khởi tạo, bổ sung, tìm kiếm, duyệt, xóa trên danh sách liên kết kép.

Tổ chức STACK và QUEUE bằng danh sách liên kết

- Sinh viên tự cài đặt.
  - Tổ chức ngăn xếp (Stack)
  - Tổ chức hàng đợi (Queue)

Q&A



Cây

LOGO

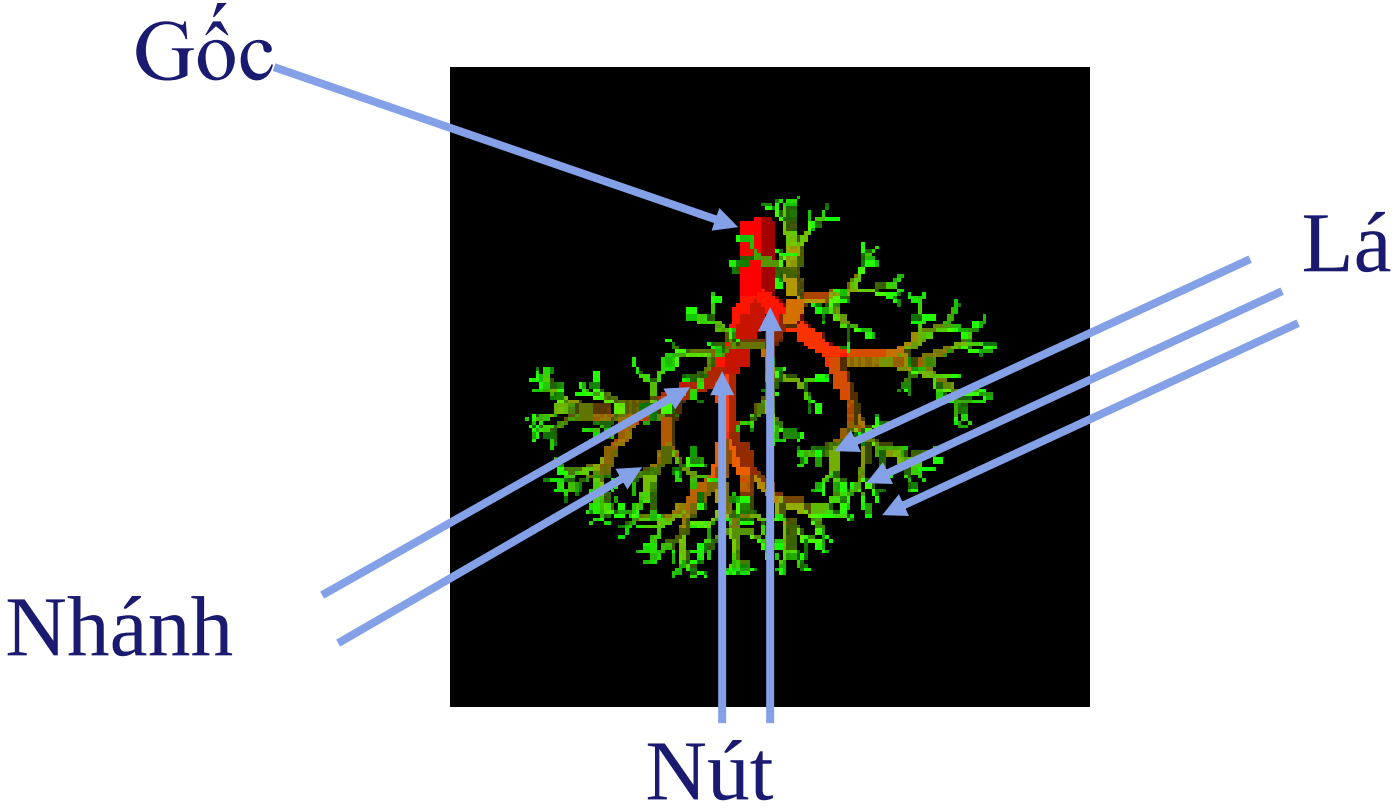
Mục tiêu

- Trang bị cho sinh viên các khái niệm và ứng dụng cây
- Cài đặt và thực hiện các phép toán trên cây, đặc biệt là các phép toán trên cây nhị phân nhị phân tìm kiếm.

Nội dung

- Định nghĩa và các khái niệm
- Cây nhị phân
- Cây nhị phân tìm kiếm (BST)
- Cây tổng quát

Cây (trong máy tính)



Khái niệm về cây (tree)

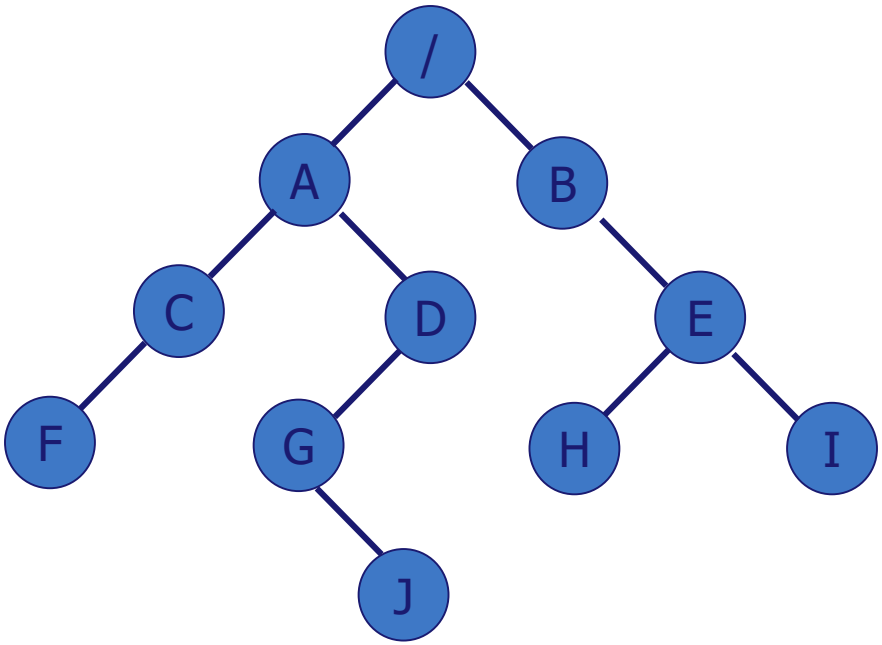
- Là tập hữu hạn các nút (tree node), sao cho
  - Có một nút gọi là nút gốc (root)
  - Các nút còn lại được phân hoạch thành n tập riêng biệt T1, T2 , ... , Tn, mỗi tập Ti là một cây
  - Giữa các nút có quan hệ phân cấp (hierarchical relationship) gọi là “quan hệ cha con”
- Cây không có nút gọi là cây rỗng (null tree)

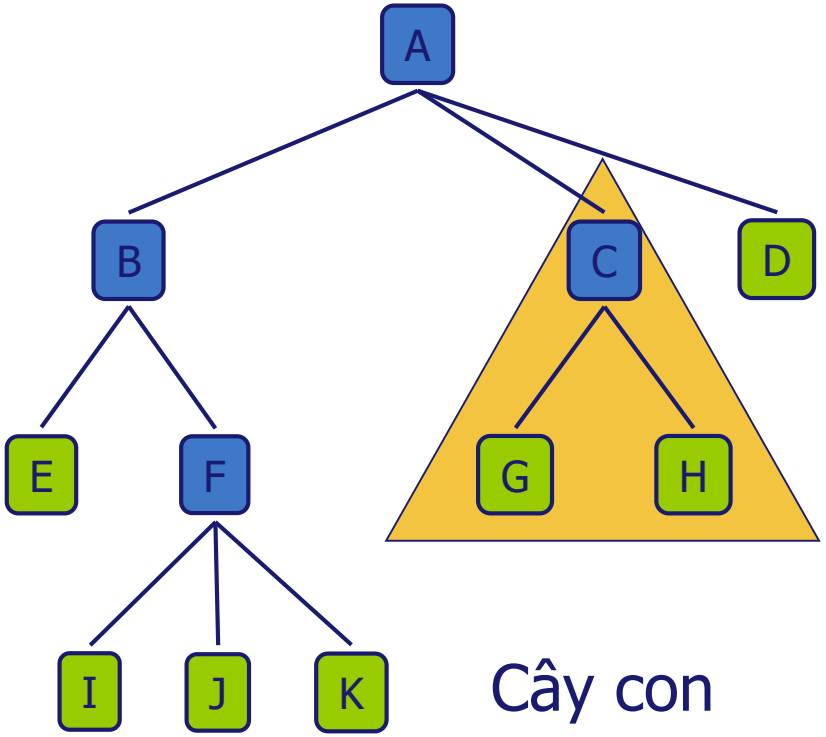
Biểu diễn cây

- Bằng đồ thị
- Bằng giản đồ
- Bằng danh sách (các dấu ngoặc lồng nhau)
- Bằng phương pháp Indentatio

Biểu diễn cây

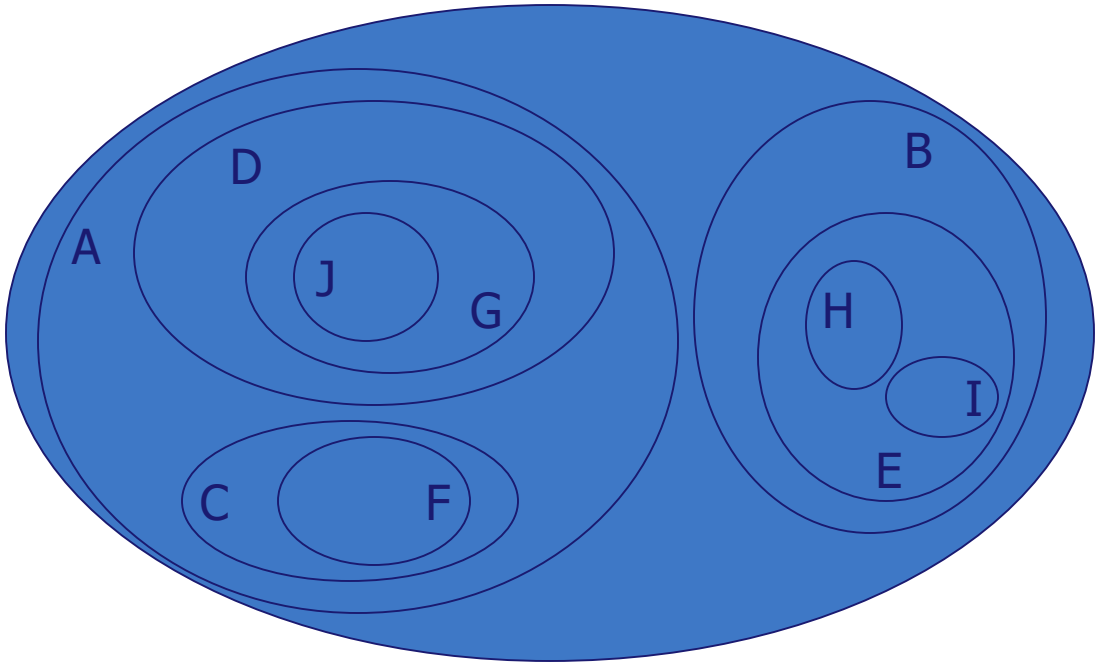
- Bằng đồ thị





Biểu diễn cây

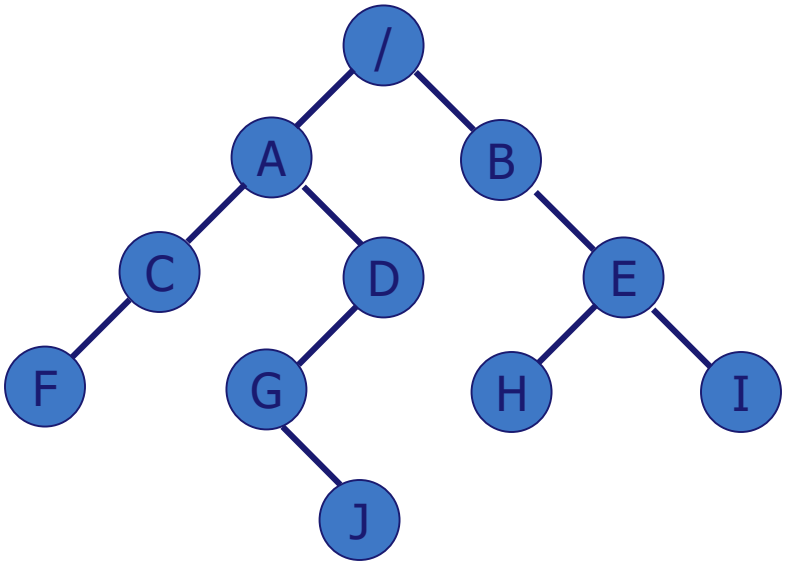
- Bằng giản đồ

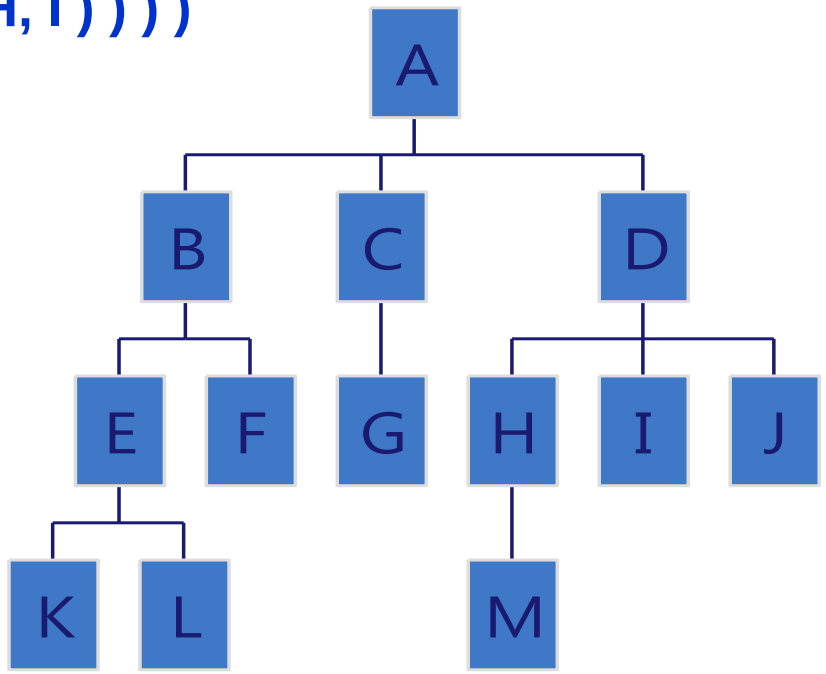


Biểu diễn cây

- Bằng danh sách (các dấu ngoặc lồng nhau)

(/(A (C (F), D (G (J)))), (B (E (H, I))))





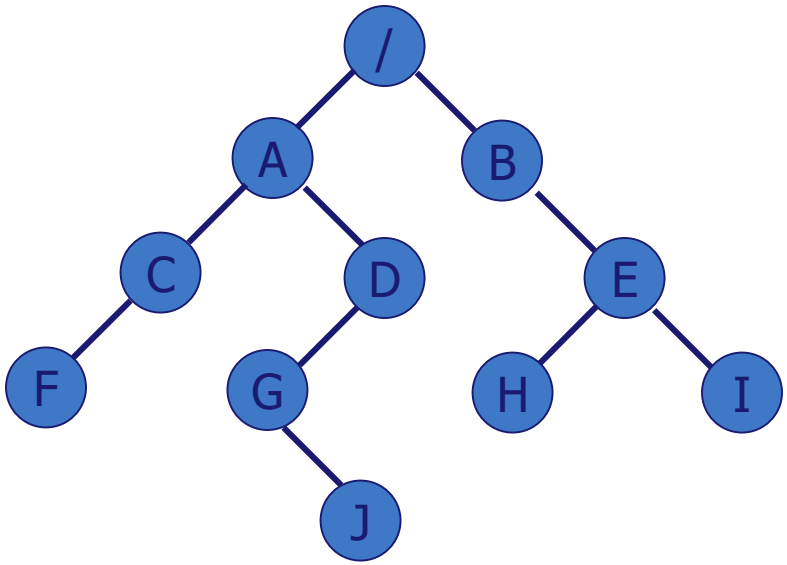
(A (B (E (K, L), F), C (G), D (H (M), I, J)))

Biểu diễn cây

- Bằng phương pháp Indentatio

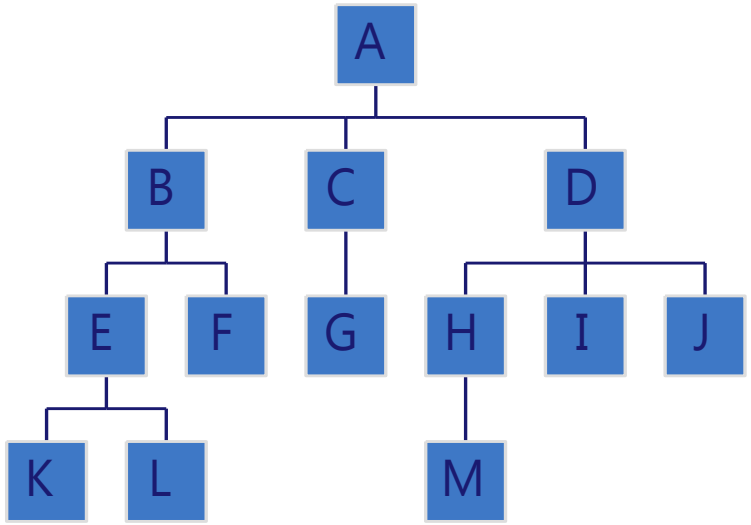
A
- C
  - F
- D
  - G
    - J

B
- E
  - H
  - I



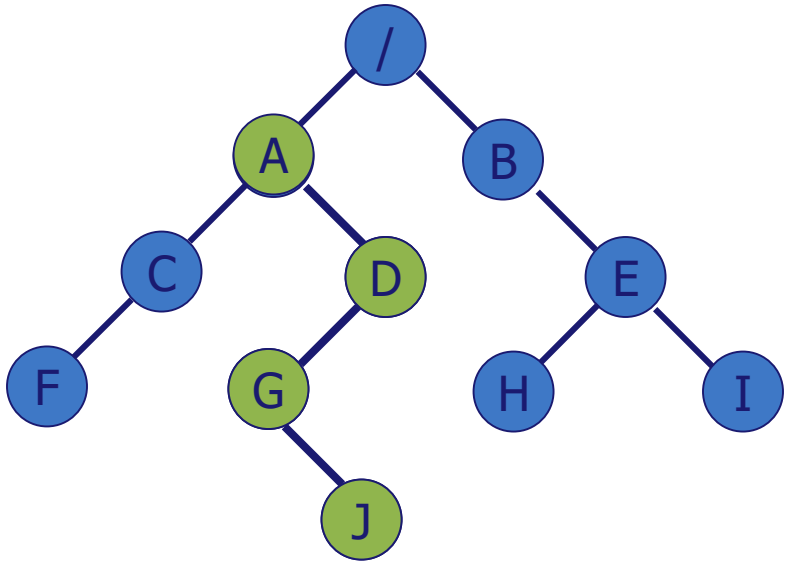
Các thuật ngữ

- Bậc của nút và bậc của cây
  - Nút A: bậc 3, nút C bậc 1
  - Bậc của cây: 3
- Nút gốc, Nút lá và nút nhánh
- Nút cha (Parent), nút con (children)



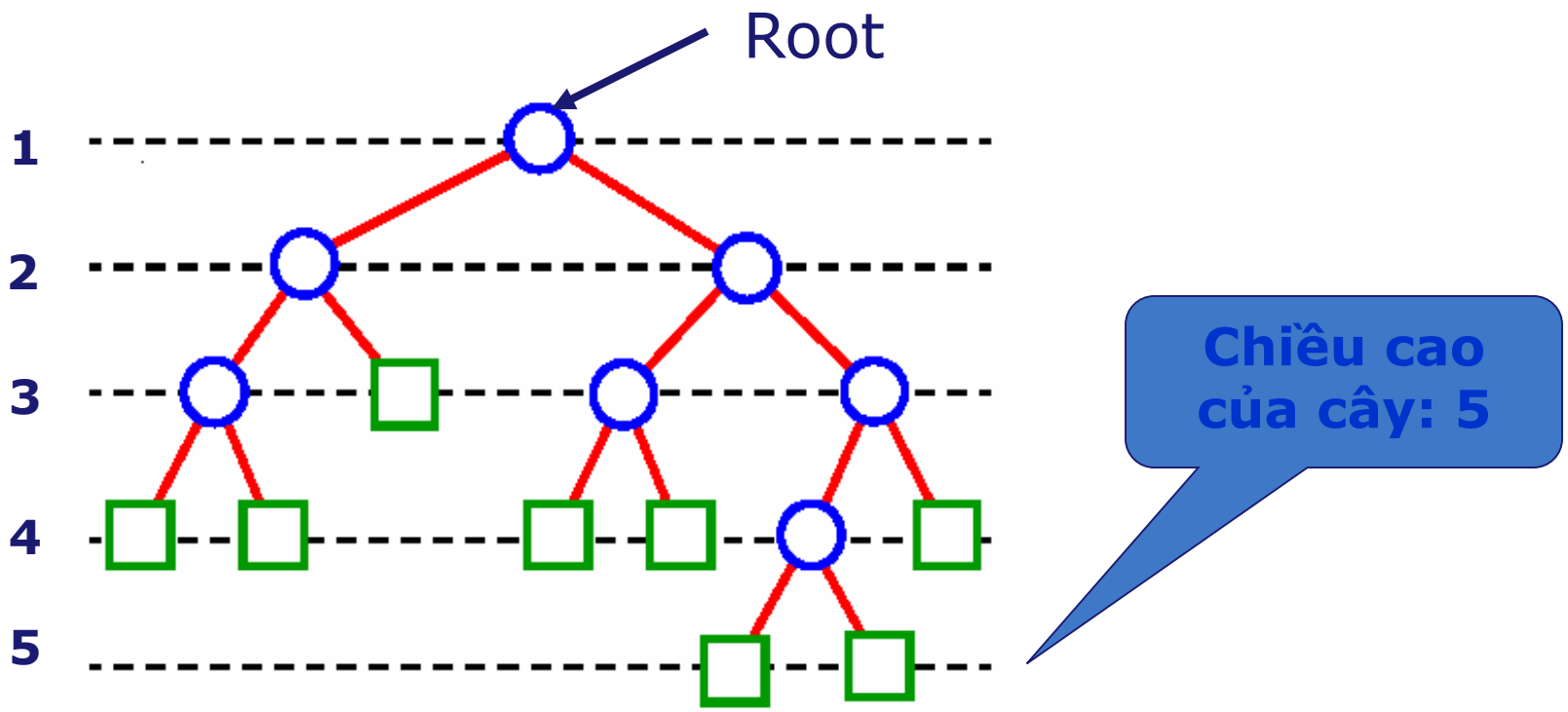
Các thuật ngữ

- Đường đi (path)



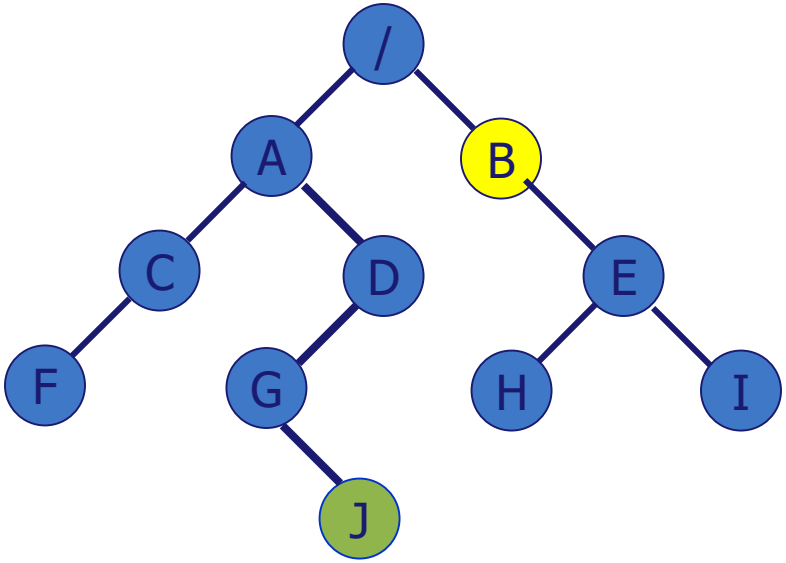
Các thuật ngữ

- Mức của nút và chiều cao của cây



Các thuật ngữ

- Tổ tiên (ancestors) của một nút
  - Tổ tiên của nút J
- Con cháu (Descendant) của một nút:
  - Con cháu của B
- Các con của cùng một cha gọi là anh em ruột (siblings)

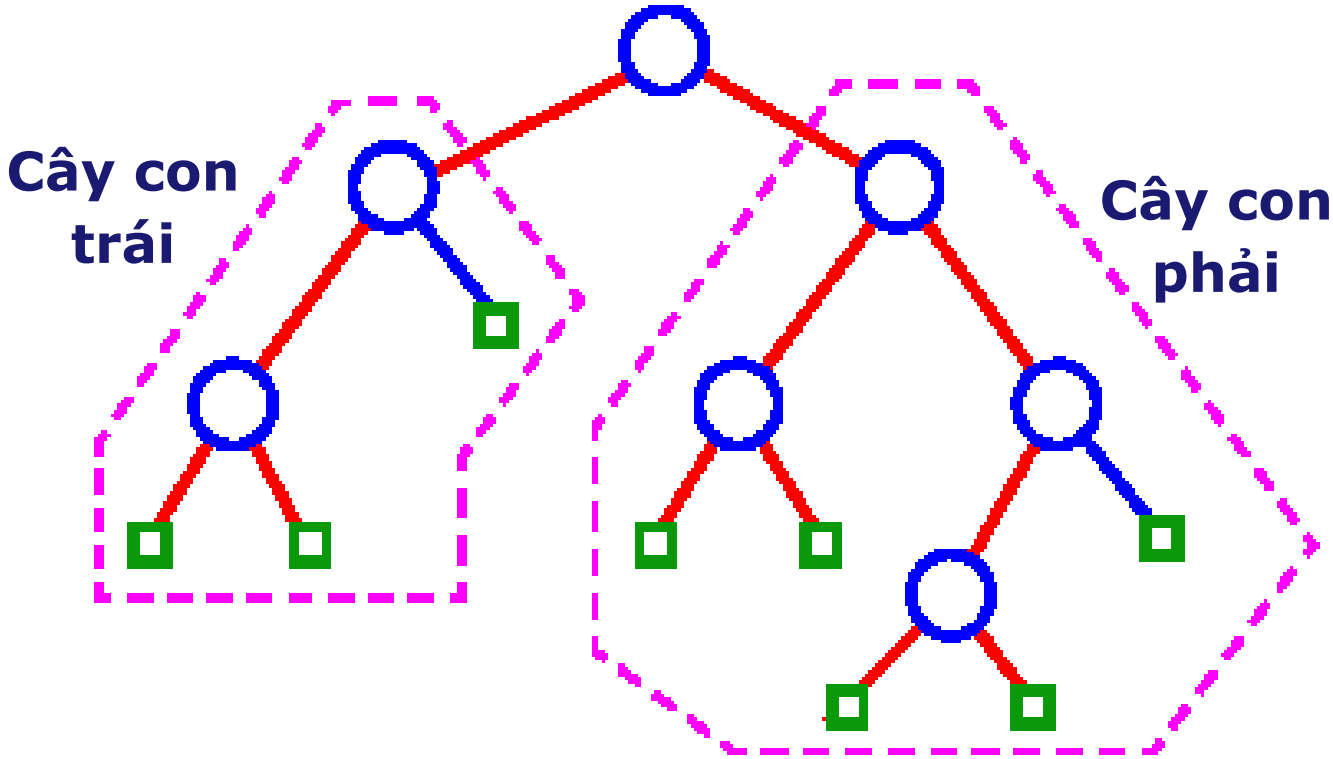


Cây có thứ tự và Rừng

- Cây có thứ tự (ordered tree)
  - Một cây gọi là có thứ tự khi ta thay đổi vị trí của các cây con, ta nhận được một cây mới
- Rừng (forest)
  - Tập hợp hữu hạn các cây phân biệt
  - Nếu bỏ đi nút gốc của một cây, ta sẽ thu được một rừng gồm nhiều cây phân biệt

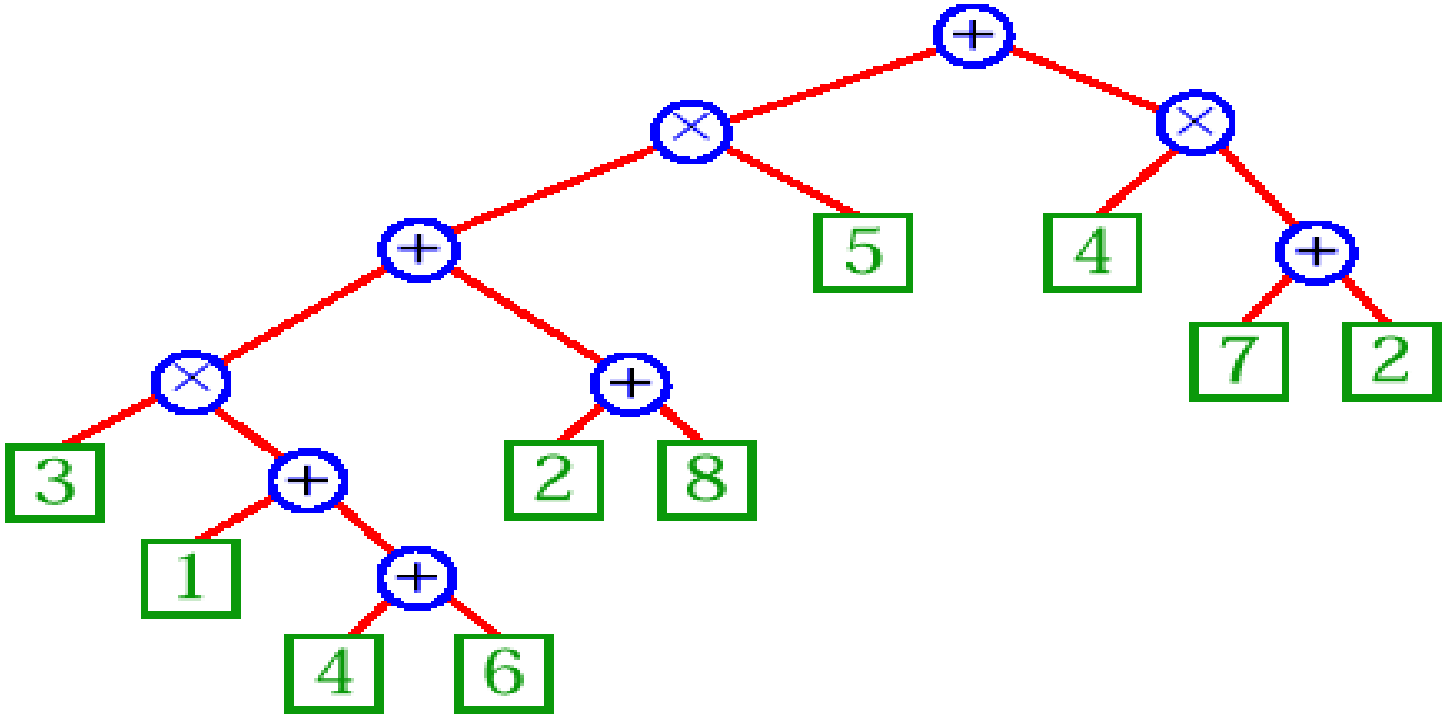
Cây nhị phân

- Định nghĩa



Cây nhị phân

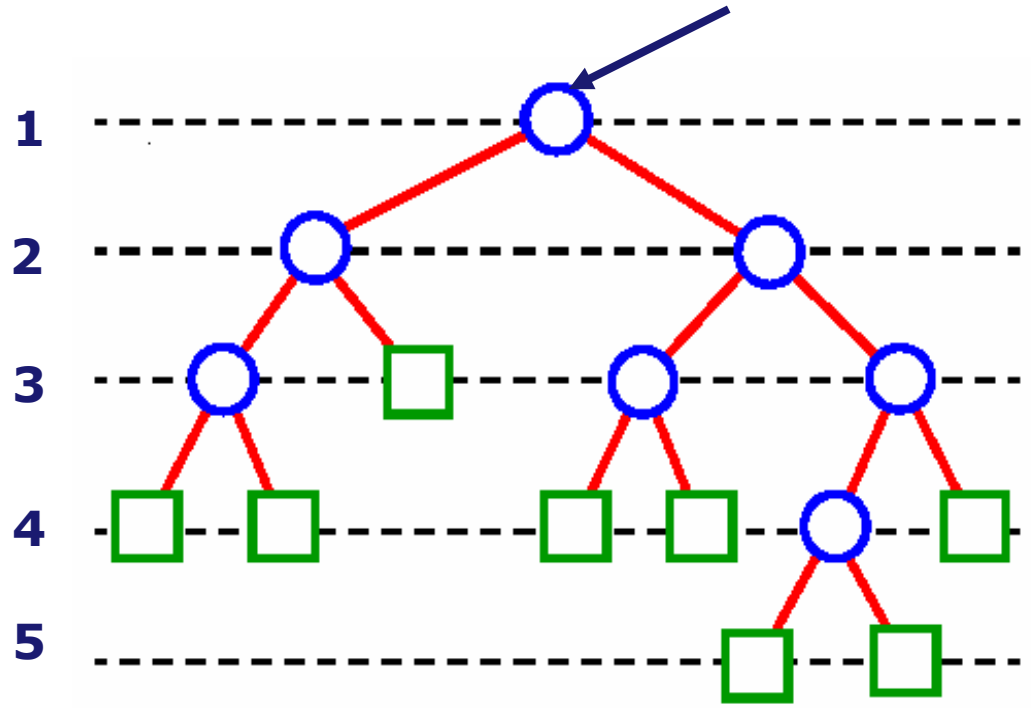
- Cây nhị phân biểu diễn biểu thức toán học



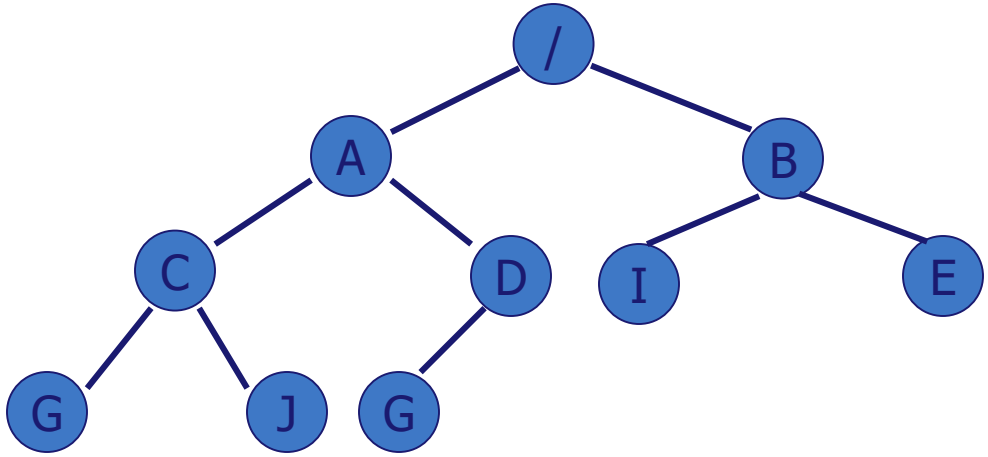
((((3 × (1 + (4 + 6))) + (2 + 8)) × 5) + (4 × (7 + 2)))

Tính chất của cây nhị phân

- Số nút tối đa mức i trong cây 2^{i-1}
- Số nút tối đa trong cây là 2^h-1 (h chiều cao của cây)
  - Chiều cao của cây $h \geq \log_2 N$ *(N là số nút trong cây).*

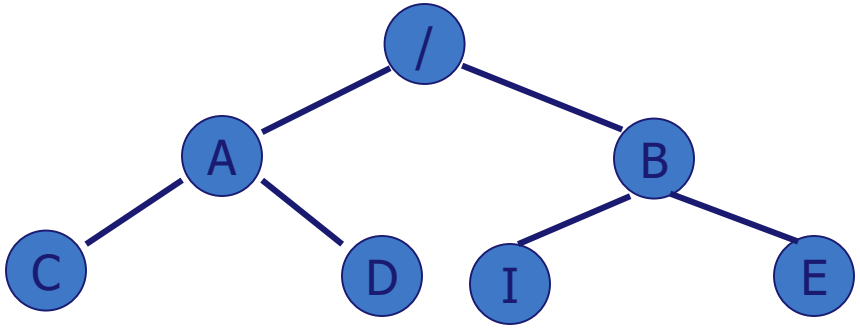


Cây nhị phân hoàn chỉnh



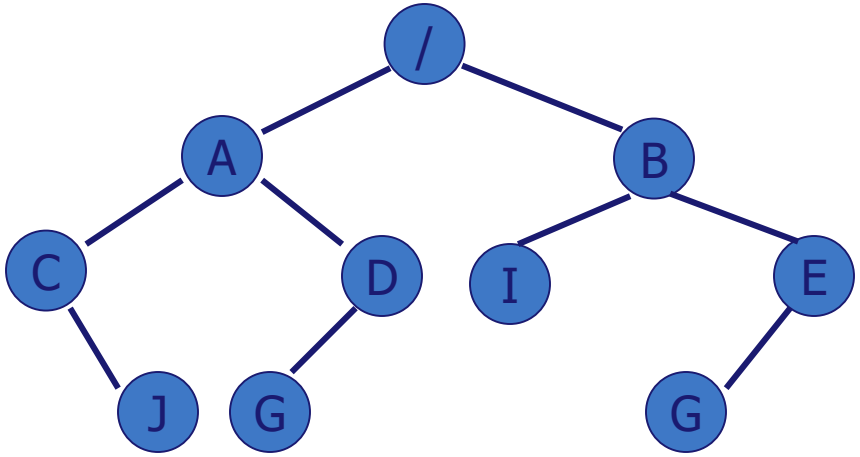
Các nút ứng với các mức trừ mức cuối đều đạt tối đa,
ở mức cuối, các nút đều đạt về phía trái

Cây nhị phân đầy đủ



Các nút đạt tối đa ở cả mọi mức

Cây nhị phân gần đầy



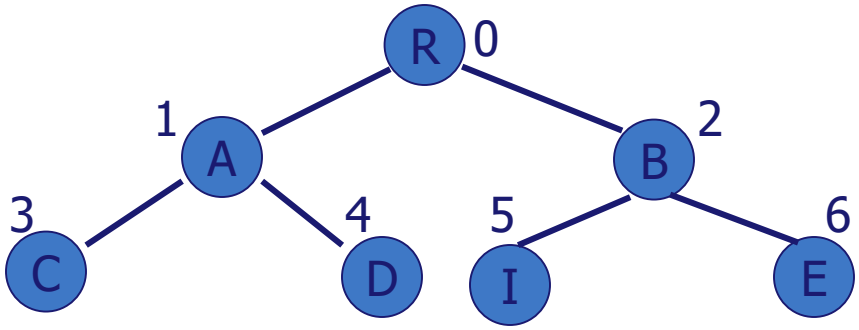
Các nút ứng với các mức trừ mức cuối đều đạt tối đa,
ở mức cuối, các nút không dạt đều về phía trái

Tổ chức lưu trữ cây nhị phân

- Sử dụng mảng một chiều (lưu trữ kế tiếp)
  - Đánh số thứ tự từ gốc, tại mỗi mức, đánh số các nút từ trái sang phải, từ mức thấp đến mức cao
- Sử dụng liên kết(Lưu trữ liên kết)
  - Quản lý cây thông qua nút gốc (root)
  - Mỗi nút cấp phát động, bao gồm dữ liệu và hai liên kết pLeft, pRight, liên kết tới cây con trái và cây con phải
  - Nút lá có hai liên kết trái phải đều rỗng

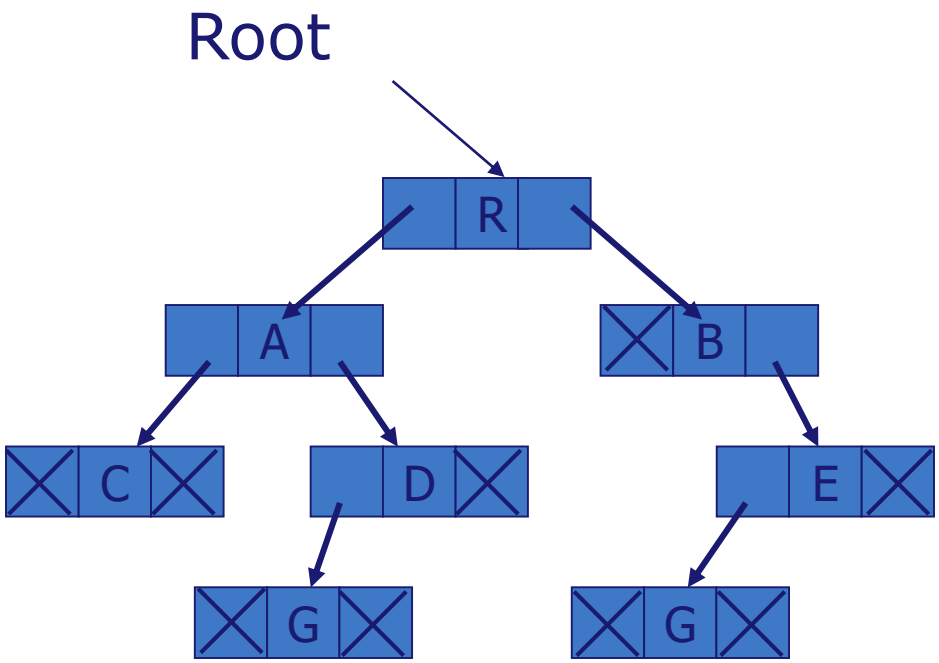
Lưu trữ kế tiếp cây nhị phân

- Con của nút thứ *i* là nút thứ *2i*+ 1và *2i*+2
- Cha của nút thứ *j* là nút *[(j-1)/2]*



R	A	B	C	D	I	E
V[0]	V[1]	V[2]	V[3]	V[4]	V[5]	V[6]

Sử dụng Liên kết

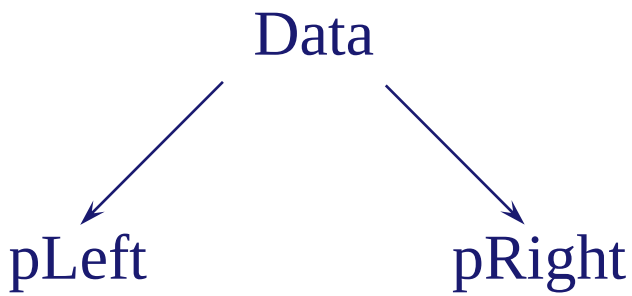


Sử dụng Liên kết

- Cấu tạo của nút
  - Tạo lập bằng cách cấp phát bộ nhớ động
  - Mỗi nút gồm có các thông tin:
    - Dữ liệu (data)
    - 2 liên kết pLeft, pRight liên kết đến nút con trái và nút con phải

Cấu trúc của nút

```
Class Node {
   int Data;
   Node pLeft; // liên kết đến nút con trái
   Node pRight; // liên kết đến nút con phải
};
Node root  = NULL; //gốc của cây
```

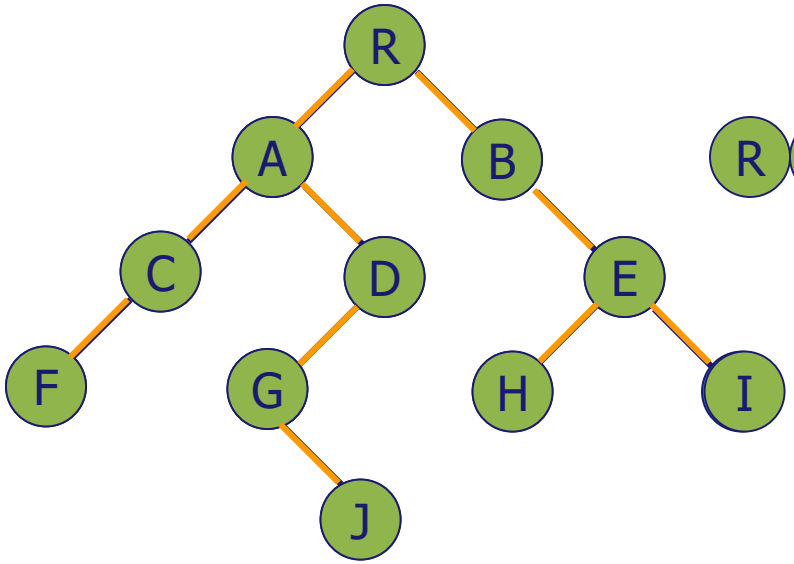


Phép duyệt cây nhị phân

- Định nghĩa
  - là phép xử lý các nút trên cây, mỗi nút một lần
- Duyệt cây theo thứ tự trước (preorder)
- Duyệt cây theo thứ tự giữa (inorder)
- Duyệt cây theo thứ tự sau (postorder)

Duyệt cây theo thứ tự trước

- Duyệt cây theo thứ tự trước (NLR)- Đệ qui
  - Thăm gốc
  - Duyệt cây con trái theo thứ tự trước
  - Duyệt cây con phải theo thứ tự trước



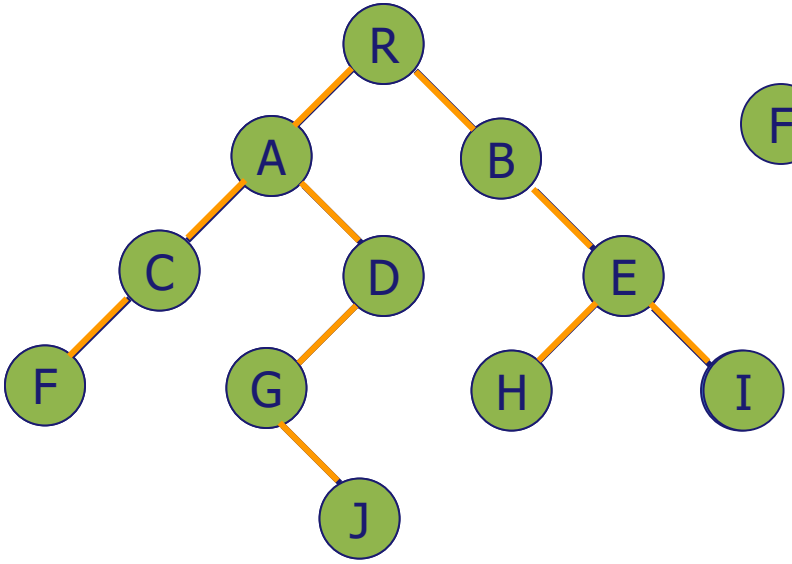
R A C F D G J B E H I

Duyệt theo thứ tự trước

```
void preorder(Node root)
{
    if (root != NULL) {
        - In ra : root.data;
        preorder(root.pLeft);
        predorder(root.pRight);
    }
}
```

Duyệt cây theo thứ tự giữa

- Duyệt cây theo thứ tự giữa (LNR)
  - Duyệt cây con trái theo thứ tự giữa
  - Thăm gốc
  - Duyệt cây con phải theo thứ tự giữa



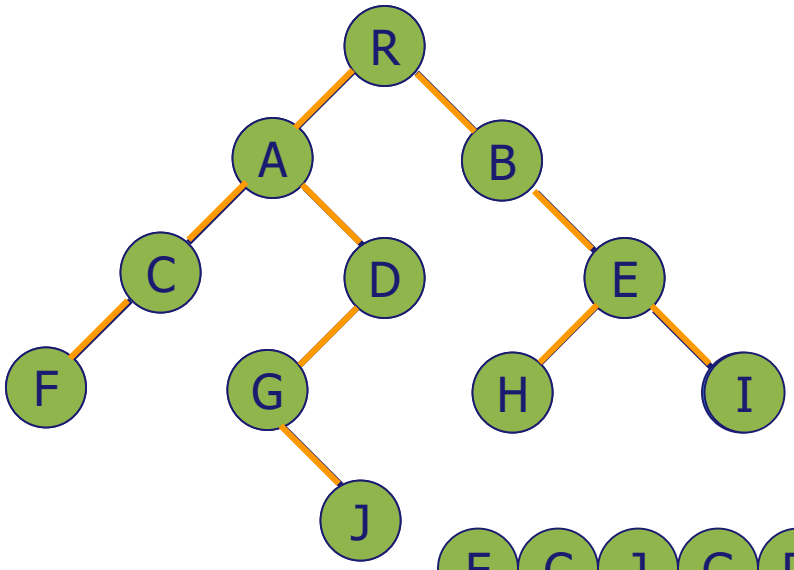
F C A G J D R B H E I

Duyệt cây theo thứ tự giữa

```
void inorder(Node root)
{
    if (root !=  NULL) {
        inorder(root.pLeft);
       In ra:  root.data;
        indorder(root.pRight);
    }
}
```

Duyệt cây theo thứ tự sau

- Duyệt cây theo thứ tự sau (LRN)
  - Duyệt cây con trái theo thứ tự sau
  - Duyệt cây con phải theo thứ tự sau
  - Thăm gốc



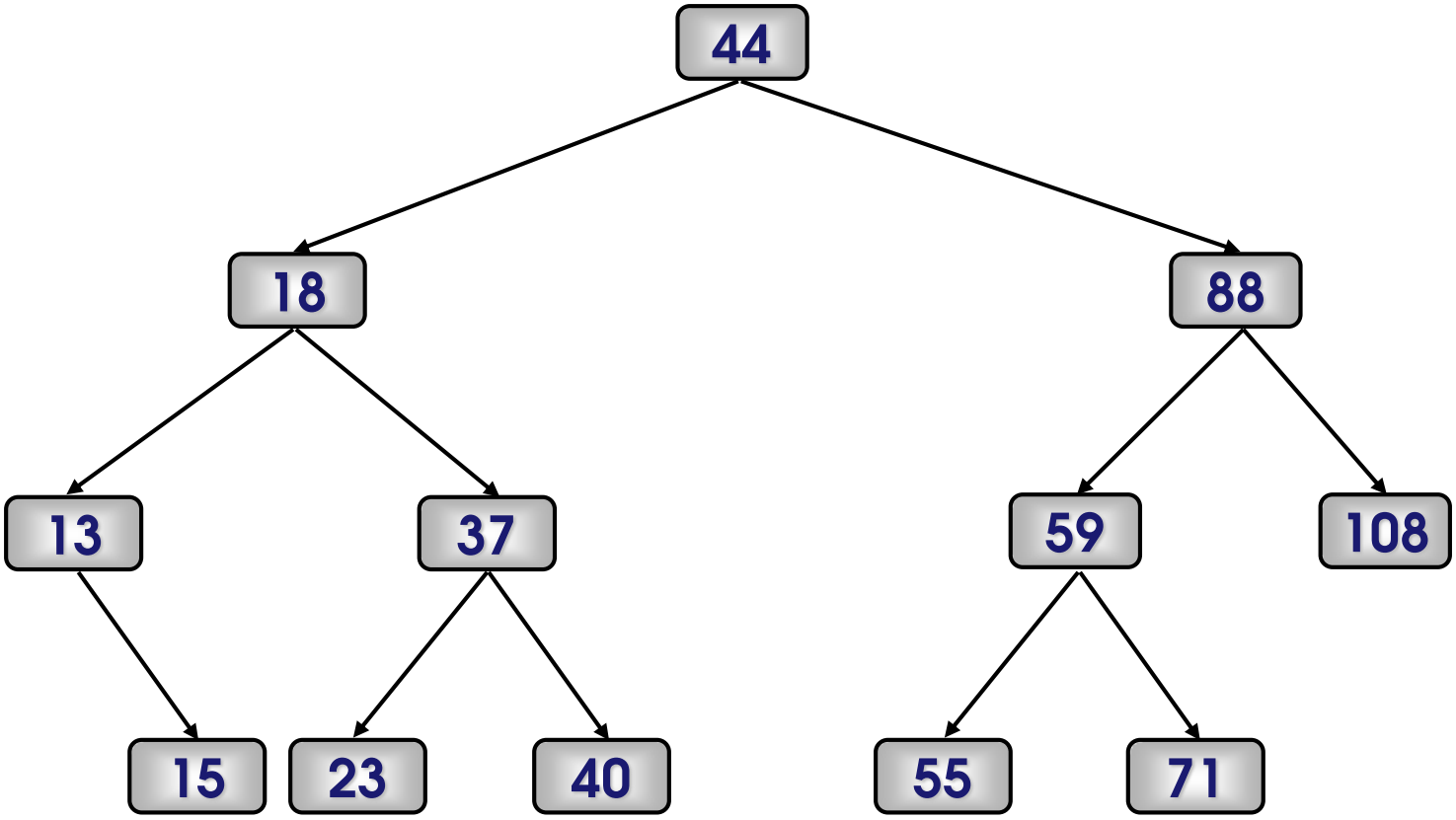
F C J G D A H I E B R

Duyệt cây theo thứ tự sau

```
void postorder(Node root)
{
    if (root !=null) {
        postorder(root.pleft);
        postdorder(root.pright);
        In ra: root.data;
    }
}
```

Cây nhị phân tìm kiếm

- Định nghĩa: (Binary Search Tree – BST)



Cây nhị phân tìm kiếm

- Khai báo cây

```
Class BSTNode {
  int Data;
    BSTNode pLeft; //con trỏ đến nút con trái
    BSTNode pRight; //con trỏ đến nút con phải
};
BSTNode root = NULL; //gốc của cây
```

Cây nhị phân tìm kiếm

- Các thao tác trên cây BST
  - Tìm nút có khóa x
  - Xóa một nút có khóa là x
  - Tìm nút có khóa nhỏ nhất
  - Tìm nút có khóa lớn nhất
  - Giải phóng cây

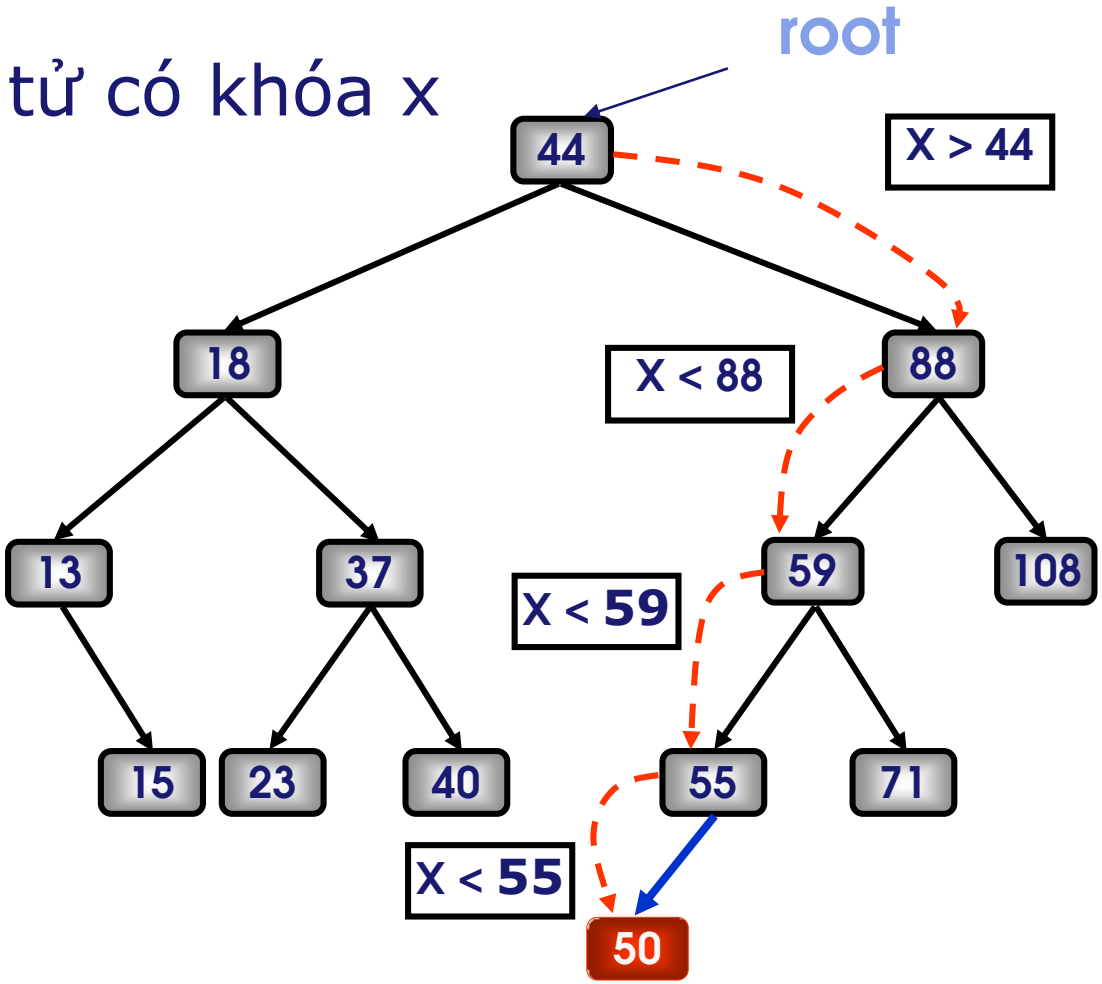
Cây nhị phân tìm kiếm

- int Insert(int X, BSTNode root);
- int Delete(int X, BSTNode root);
- BST_Node Find(int X, BSTNode root);
- BST_Node FindMin(BSTNode root);
- BST_Node FindMax(BSTNode root);
- void MakeEmpty(BSTNode root);

Thêm một phần tử vào cây nhị phân tìm kiếm

- Thêm vào phần tử có khóa x

Tìm X= 50



Thêm một phần tử vào cây nhị phân tìm kiếm

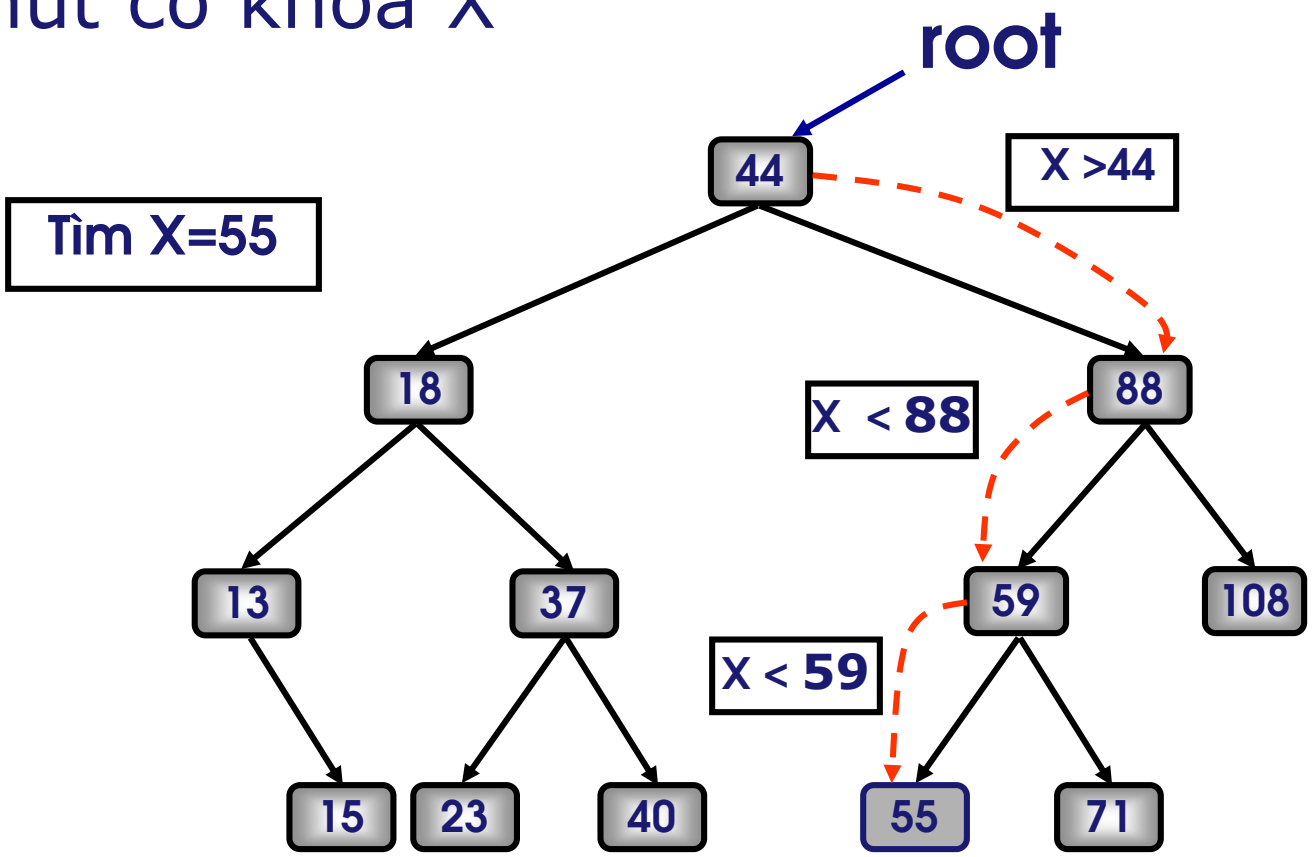
```
int  Insert(int X, BST_Node root)
{  if  (root == NULL)
     {   root =new BSTreeNode ;
         if  ( root == NULL )
              return -1; // Không thể cấp phát bộ nhớ
         else
          {
             root.Data = X;
             root.pLeft = root.pRight = NULL;
             return 1; // Thêm vào thành công
           }
       }
```

Thêm một phần tử vào cây nhị phân tìm kiếm

```
else
 if (root.Data ==X)
                return 0 ; // Đã tồn tại trong cây
 else
       if ( X< root.Data )
        return Insert( X, root.pLeft );
          else
         return Insert( X, root.pRight );
}
```

Tìm một nút có khóa X

- Tìm nút có khóa X



Tìm một nút có khóa X

```
BSTNode Find( int X, BSTNode root)
  {          if( root == NULL )
                return NULL;
             if ( X < root.data)
                return Find( X, root.pLeft );
             else if ( X > root.data)
                return Find( X, root.pRight );
             else
                return root;
   }
```

Tìm một nút có khóa X

- Tìm nút có khóa X, không dùng đệ qui

```
BTSNode Find2(int X, BTSNode root)
{
    BTSNode p = root;
    while (p != NULL)
    {
        if(X == p.data)   return p;
          else
                if(x < p.data) p = p.pLeft;
                else
             p = p.pRight;
     }
    return NULL;
}
```

Tìm nút có khóa nhỏ nhất

```
BSTNode FindMin( BSTNode root)
  {         if ( root == NULL )
                return NULL;
            else if( root.pLeft == NULL )
                return root;
            else
                return FindMin( root.pLeft );
  }
```

Tìm nút có khóa lớn nhất

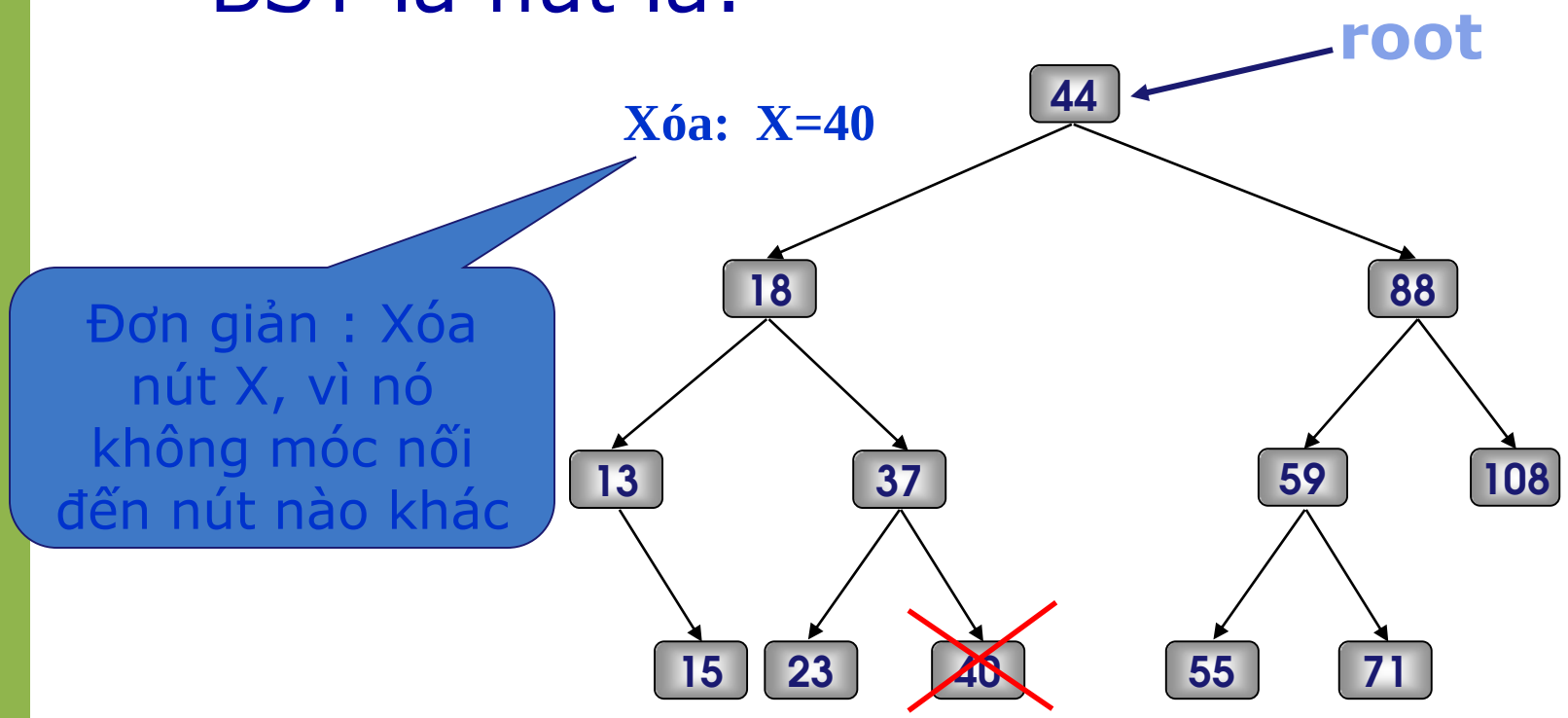
```
BST_Node FindMax(BSTNode root)
 {
      if (root != NULL )
            while (root.pRight != NULL )
                root = root.pRight;
        return root;
  }
```

Xóa một nút có khóa X trên cây BST

- Xóa một nút có khóa X trên cây BST, có ba trường hợp:
  - Nút có khóa X là nút lá.
  - Nút có khóa X chỉ có 1 con (trái hoặc phải).
  - Nút có khóa X có đủ cả 2 con

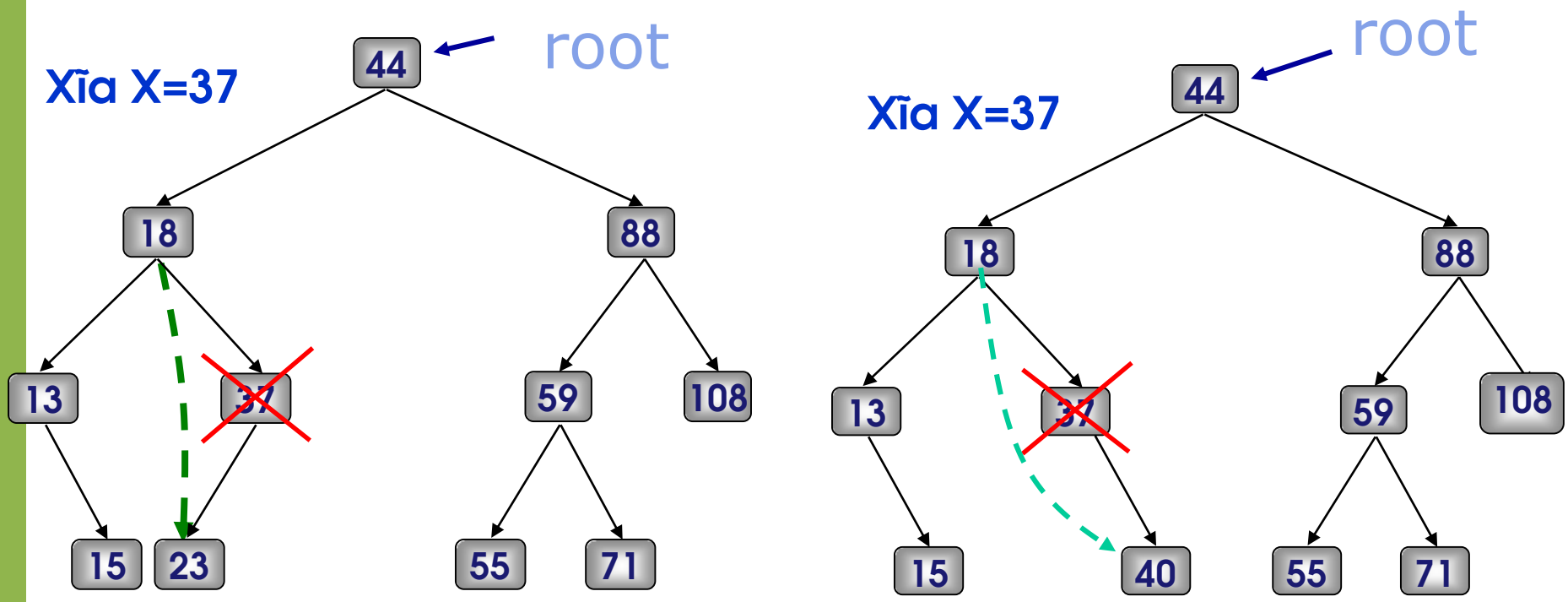
Xóa một nút có khóa X

- Trường hợp 1 :Nút có khóa X trên cây BST là nút lá:



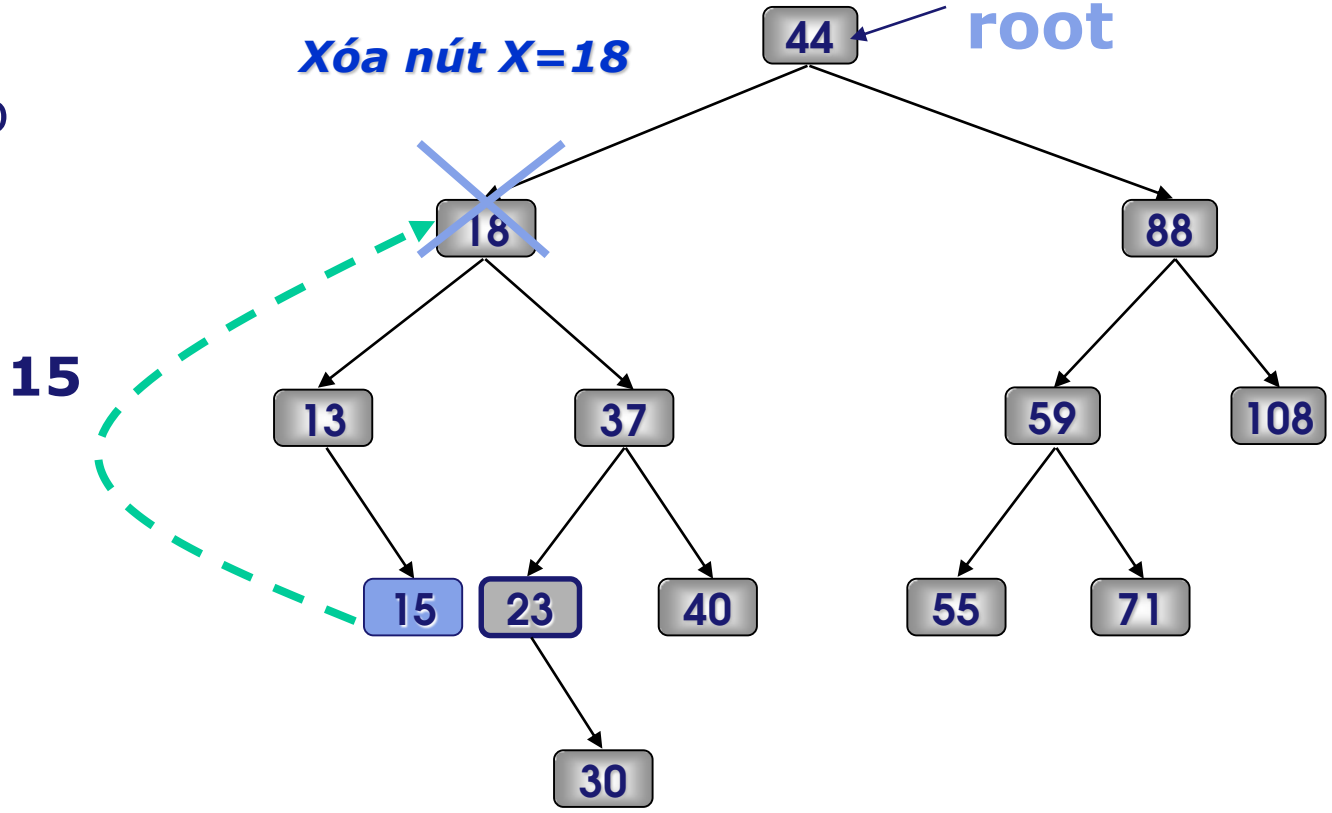
Xóa một nút có khóa X trên cây BST

- Trường hợp 2 : Nút X có một cây con trái hoặc phải



Xóa một nút có khóa X trên cây BST

- Trường hợp 3 : Nút X có một cây hai con trái và phải
  - Hủy gián tiếp



Cây nhị phân tìm kiếm (Binary Search Tree – BST)

```
int Delete( int X, BSTNode root)
 { BSTNode p;
    if ( root == NULL )
       return 0 ; // cây rỗng, không tim thấy
   else
        if ( X < root.Data)  // xóa trên cây con trái
            return Delete( X, root.pLeft );
         else
          if ( X > root.Data ) // xóa trên cây con phải
              retrurn Delete( X, root.pRight );
         else  // tìm ra nút cần xóa
```

Cây nhị phân tìm kiếm (Binary Search Tree – BST)

```
if ( root.pLeft && root.pRight )  // Có hai con
   {          p = FindMax(root.pLeft); //tìm nút có khóa lớn nhất trên con trái
              root.Data = p.Data;
              return Delete(root.Data, root.pLeft);
   }
   else  // có một con hoặc không có con
   {      p = root;
          if ( root.pLeft == NULL ) // xử lý như không có con
             root = root.pRight;
          else if ( root.pRight == NULL )
               root = root.pLeft;
          p = null;
    }
    return 1;
}
```

Giải phóng cây BST

```
void MakeEmpty( BST_Node root);
{
   if (root)
    {
        MakeEmpty(root.pLeft);
              MakeEmpty(root.pRight);
              delete root ;
        }
 }
```

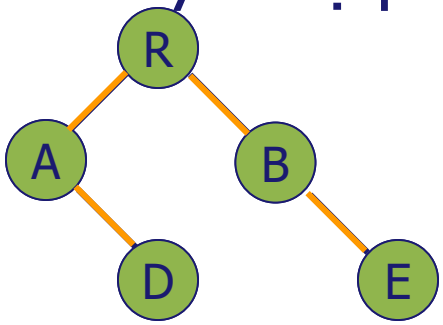
Cây nhị phân liên kết vòng

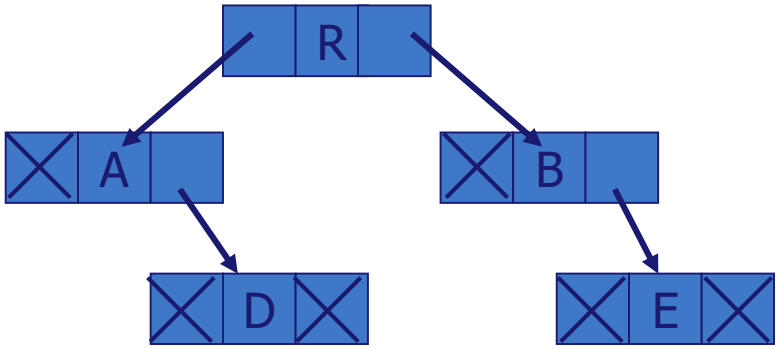
Định nghĩa

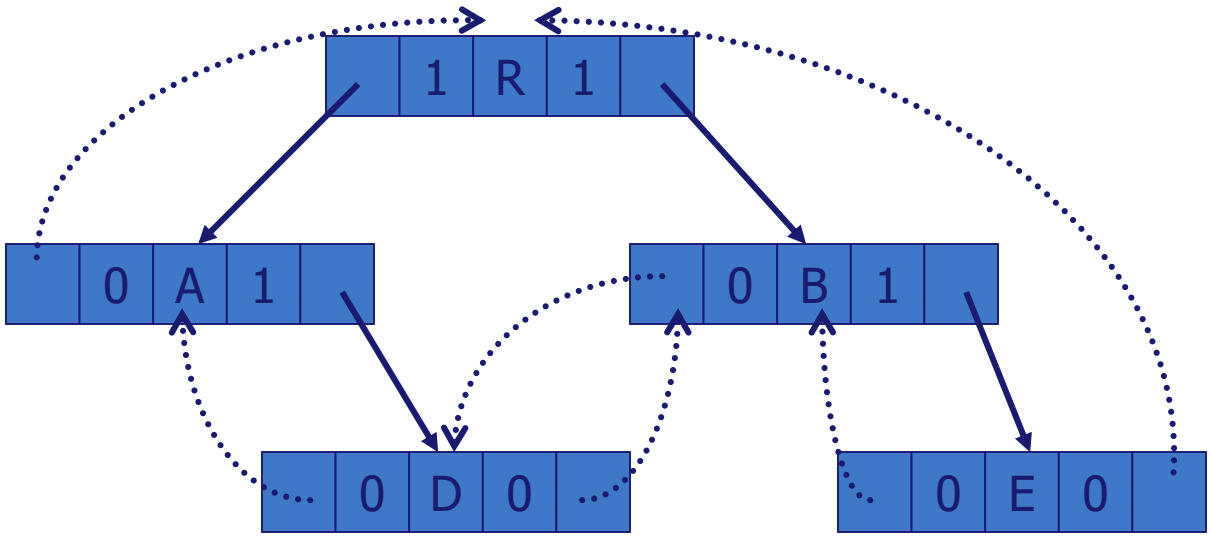
- Sử dụng liên kết NULL để lưu trữ liên kết tới nút kế tiếp trong phép duyệt cây nhị phân -> phép duyệt được thực hiện dễ dàng
- Sử dụng giá trị kiểm tra liên kết thật (đến nút trong cây) hay liên kết giả (nút trong phép duyệt)
- Ltype = true, nếu liên kết trái là liên kết thật
- Rtype = true, nếu liên kết phải là liên kết thật

Cây nhị phân liên kết vòng

- Cây nhị phân liên kết vòng (NLR)







Cây tổng quát

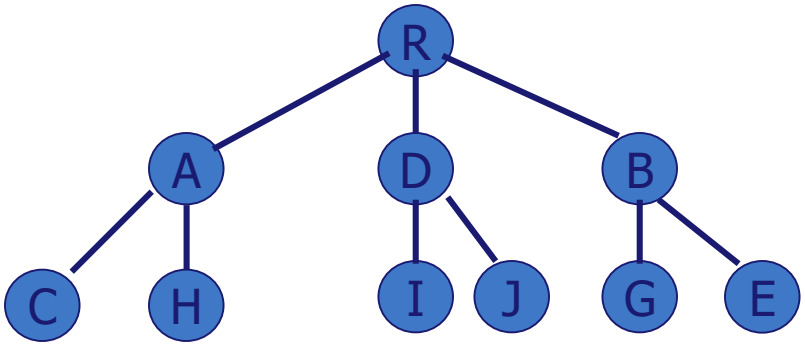
- Định nghĩa
  - Cây m phân là cây mà mỗi nút có tối đa m nút con (cây con)
  - Biểu diễn cây m phân bằng liên kết động
    - Mỗi nút có m+1 trường, với m mối nối
    - Với cây m phân đầy đủ, có n(m-1)+1 mối liên kết NULL

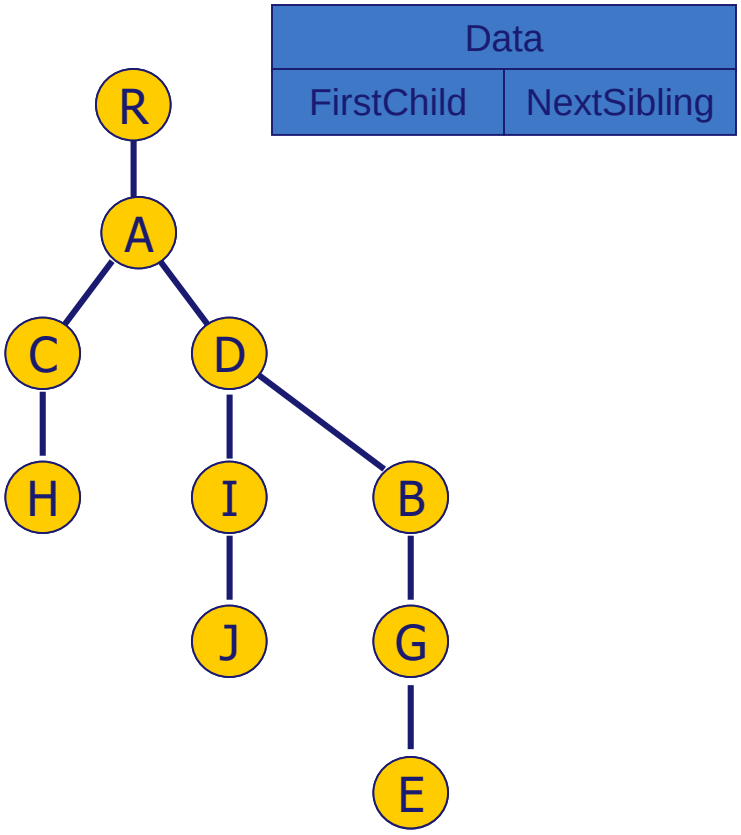
Cây tổng quát

- Biểu diễn cây tổng quát
  - Biểu diễn cây tổng quát bằng cây nhị phân
  - Đối với một nút trên cây tổng quát
    - Một nút con nằm ở vị trí trái nhất (con cả 1)
    - Một nút kế cận với nút đang xét kể từ trái sang (em kế - 2)

Cây tổng quát

- Biểu diễn cây tổng quát






```
class TreeNode
        {int info;
         TreeNode FirstChild; // con cả
         TreeNode NextSibling; //em kê
        };
```

Cây tổng quát

- Phép duyệt cây tổng quát NLR(T)
  - Nếu T rỗng, dừng
  - Ngược lại, $T_1,\ldots,T_n$ là cây con gốc T
    - Thăm gốc của T
    - NLR(T_1), T_1 cây con thứ nhất của gốc T
    - Duyệt cây con $T_2,\ldots,T_n$ của T theo thứ tự trước

Cây tổng quát

- Phép duyệt cây tổng quát LNR(T)
  - Nếu T rỗng, dừng
  - Ngược lại, $T_1,\ldots,T_n$ là cây con gốc T
    - LNR(T_1), T_1 cây con thứ nhất của gốc T
    - Thăm gốc của T
    - Duyệt cây con $T_2,\ldots,T_n$ của T theo thứ tự giữa

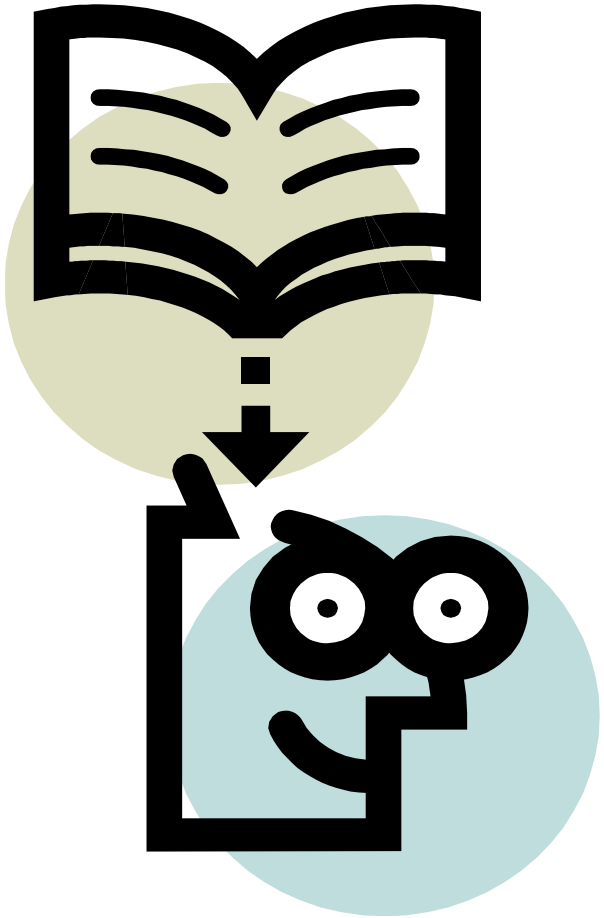
Cây tổng quát

- Phép duyệt cây tổng quát LRN(T)
  - Nếu T rỗng, dừng
  - Ngược lại, $T_1,\ldots,T_n$ là cây con gốc T
    - LNR(T_1), T_1 cây con thứ nhất của gốc T
    - Duyệt cây con $T_2,\ldots,T_n$ của T theo thứ tự giữa
    - Thăm gốc của T

Cây tổng quát

- Duyệt cây theo mức
  - Duyệt cây theo chiều rộng
  - Ý tưởng
    - Tổ chức thành một hàng đợi
    - Đưa nút gốc vào hàng đợi
    - Lặp
      - Lấy một nút ra khỏi hàng đợi
      - Duyệt nút T
      - Đưa các nút con của T (nếu có) vào hàng đợi

Q&A



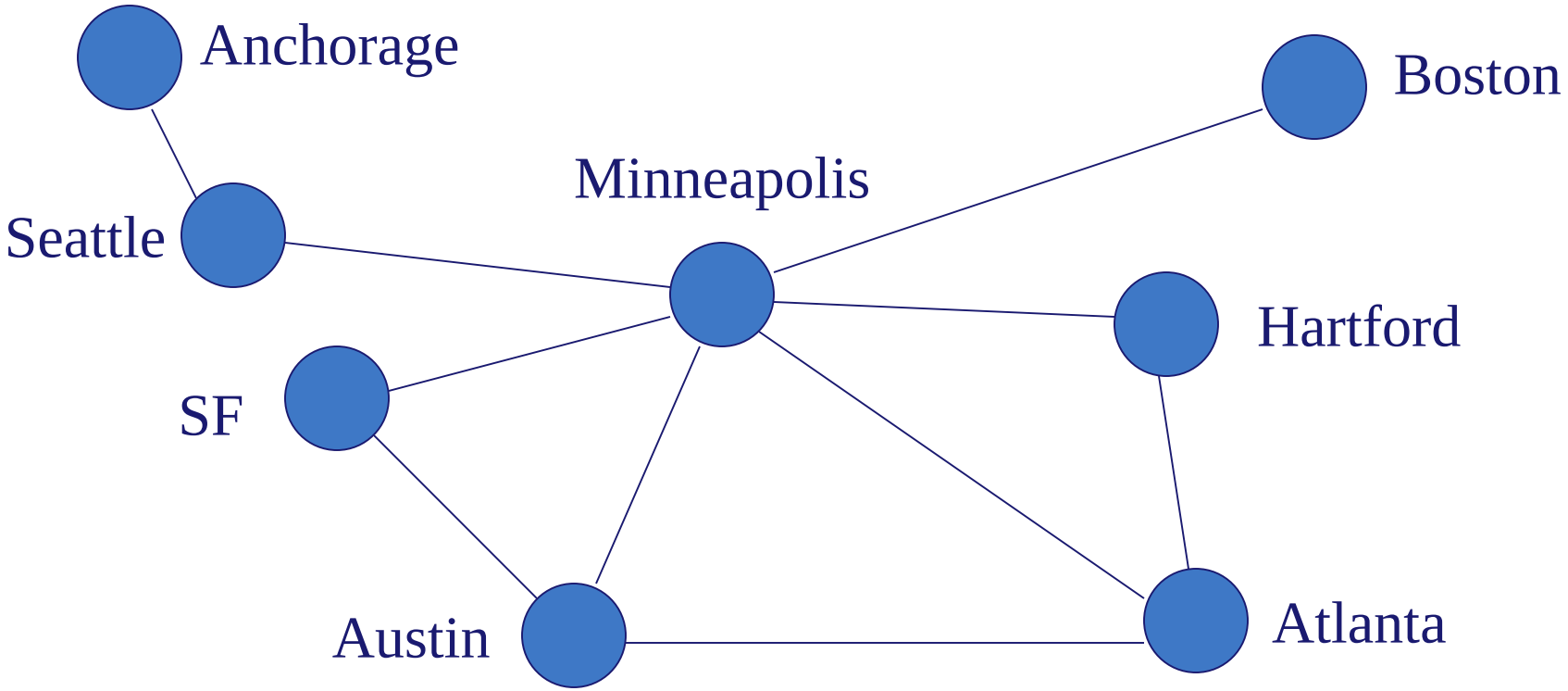
ĐỒ THỊ

LOGO

Mục tiêu của chương

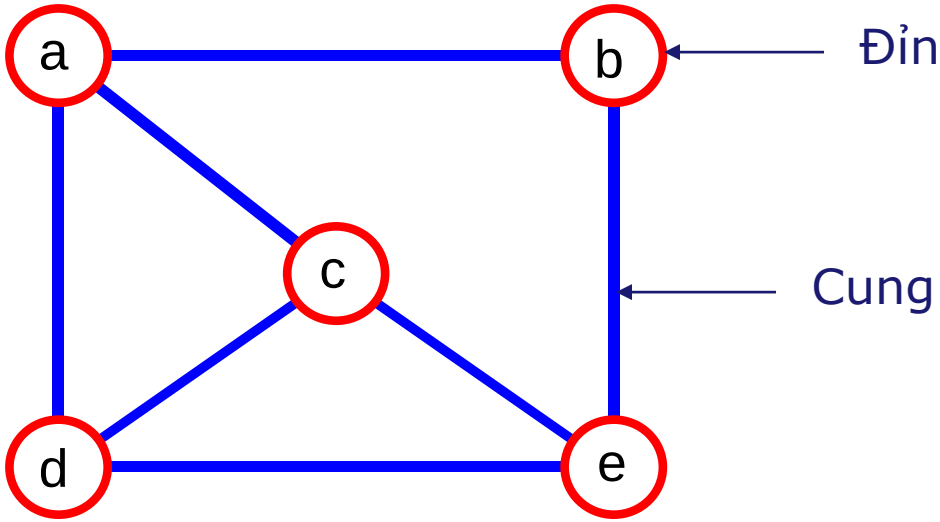
- Trình bày những kiến thức căn bản về lý thuyết đồ thị, cách biểu diễn, một số thuật toán trên đồ thị
- Đánh giá thuật toán
- Một số ứng dụng của đồ thị

Định nghĩa


Anchorage
Boston
Minneapolis
Seattle
Hartford
SF
Austin
Atlanta

Định nghĩa

- Đồ thị G = (V,E)
  - V = tập hợp hữu hạn các phần tử (đỉnh hay nút)
  - E $\subseteq$ V × V, tập hữu hạn các cạnh (cung)



Các khái niệm

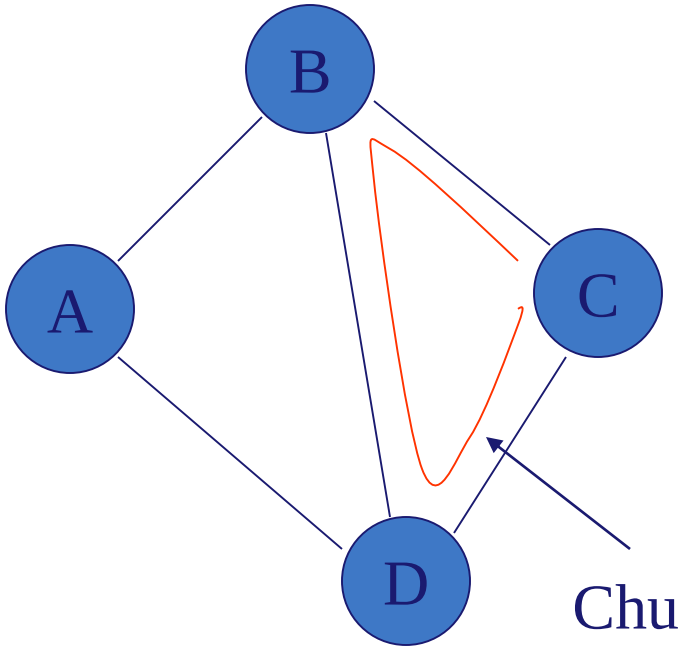
- Nếu $(x,y) \in E$
  - x gọi là đỉnh gốc, y là ngọn
  - Nếu $x \equiv y$, (x,y) gọi là khuyên
- Một dãy $u_1,u_2,\ldots,u_n$, $\forall u_i \in V$ $(i=1,n)$ gọi là một đường, nếu $(u_{i-1},u_i) \in E$
- Độ dài đường: $length(u_1,u_2,\ldots,u_n)=n$
- $(u_1,u_2,\ldots,u_n)$ đường đi đơn, nếu $u_i \neq u_j$, $1<\forall i \neq j<n$ (là đường đi, mà các đỉnh phân biệt, ngoại trừ đình đầu và đỉnh cuối)

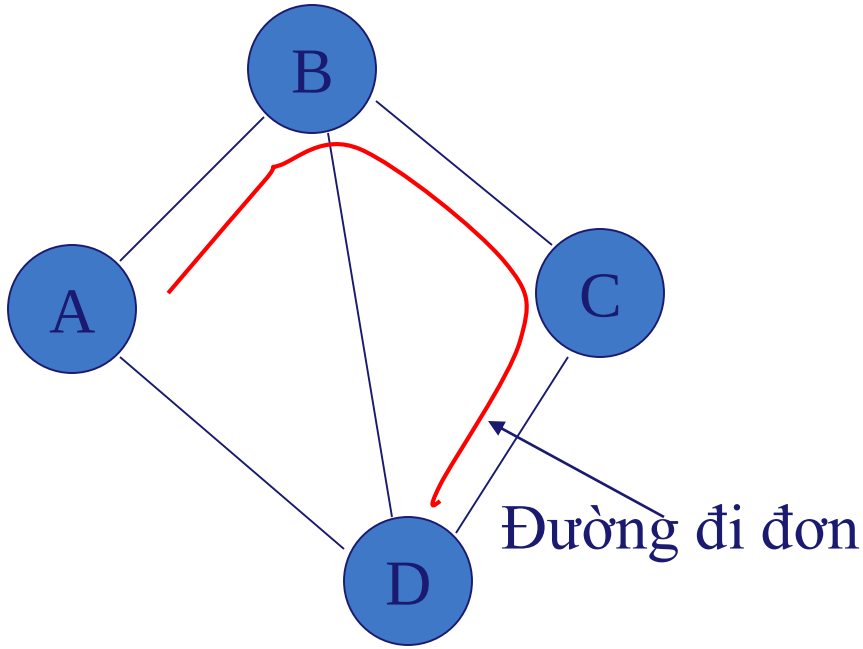
Các khái niệm (tt)

- Chu trình (cycle) = $(u_1,u_2,\ldots,u_n)$, $u_1 \equiv u_n$
- Đồ thị định hướng (directed graph)
  - $\forall (x,y) \in E : (x,y) \neq (y,x)$
- Đồ thị vô hướng
  - $\forall (x,y) \in E : (y,x) \in E$
  - $(x,y) \equiv (y,x)$

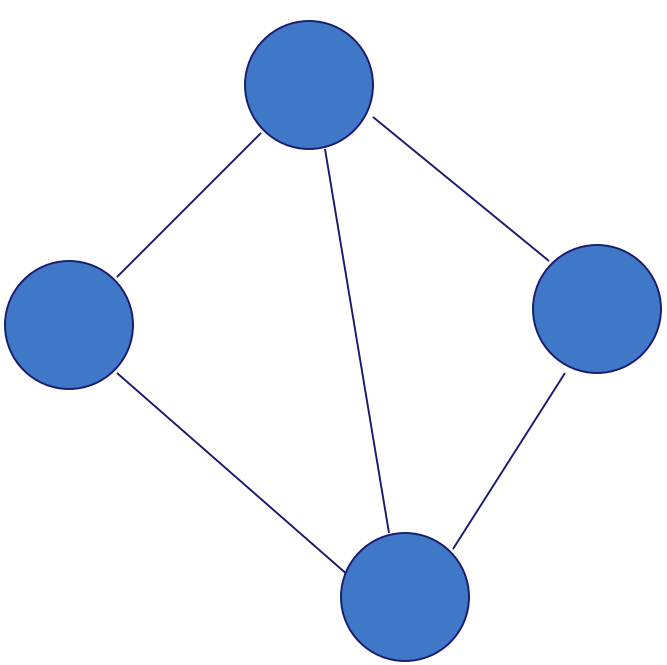
Các khái niệm (tt)

- *CBDC là một chu trình*

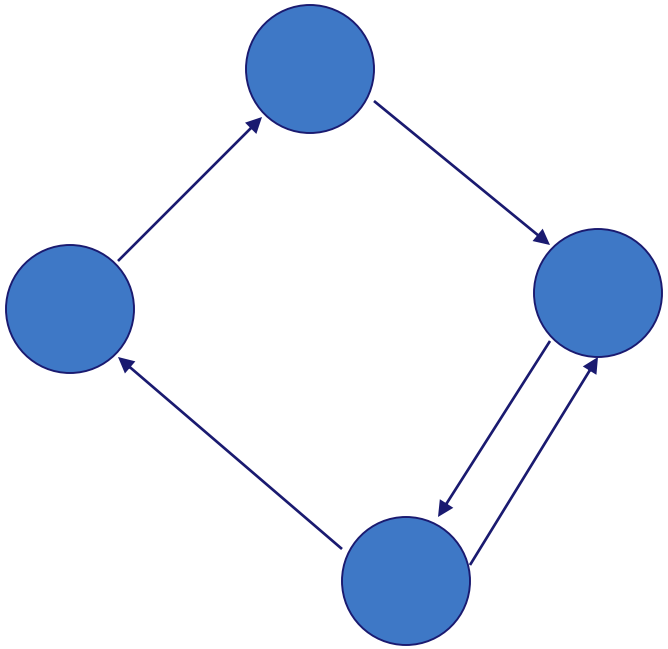




Các khái niệm (tt)



Đồ thị vô hướng

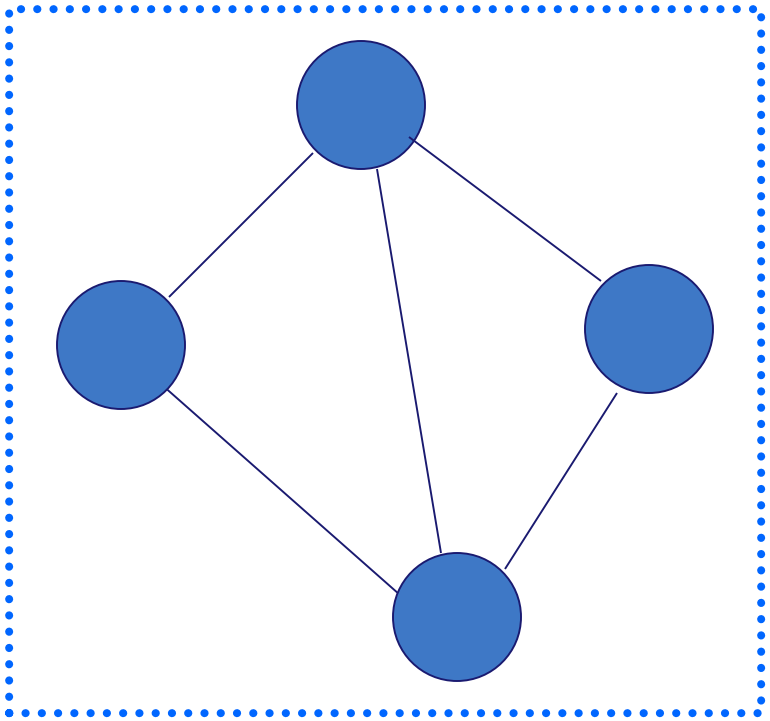


Đồ thị định hướng

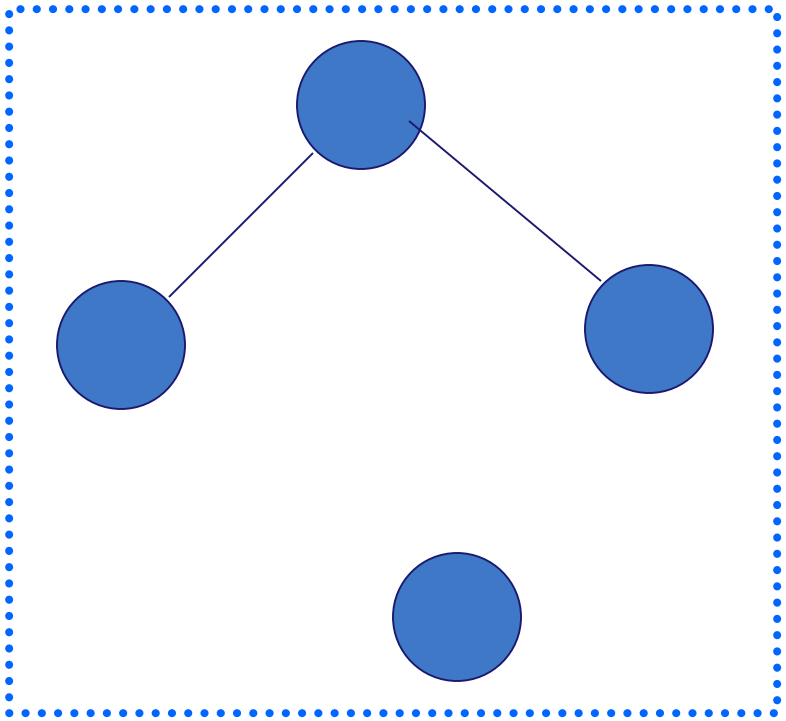
Các khái niệm (tt)

- Tính liên thông (connectivity)
  - Trong đồ thị G, hai đỉnh x,y gọi là liên thông (connected), nếu có một đường từ x đến y
  - Đồ thị G liên thông, nếu $\forall(x,y) \in E$, $\exists$ đường đi từ x đến y
- Đồ thị G gọi là có trọng số, nếu mỗi cung được gán một giá trị số đặc trưng

Các khái niệm (tt)



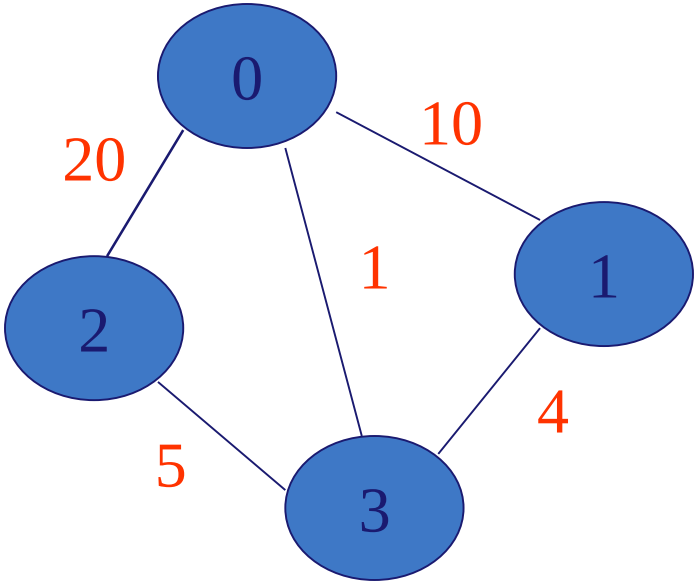
Đồ thị liên thông



Đồ thị không liên thông

Các khái niệm (tt)

- Đồ thị có trọng số



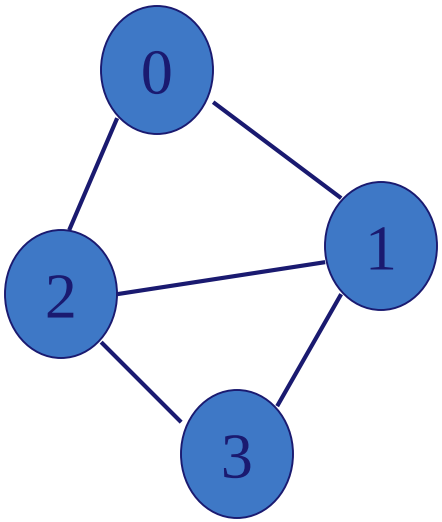
Biểu diễn đồ thị

- Biểu diễn bằng ma trận kề
  - Adjacency matrice
- Biểu diễn bằng danh sách kề
  - Adjacency list

Biểu diễn bằng ma trận kề

- Xét G=(V,E) với V={x_1,…,x_n}
- Biểu diễn G bằng ma trận A=(a_{ij}), i,j=1..n
  - a_{ij}=1, nếu ∃ (x_i,x_j) ∈ E
  - a_{ij}=0, nếu !∃ (x_i,x_j) ∈ E

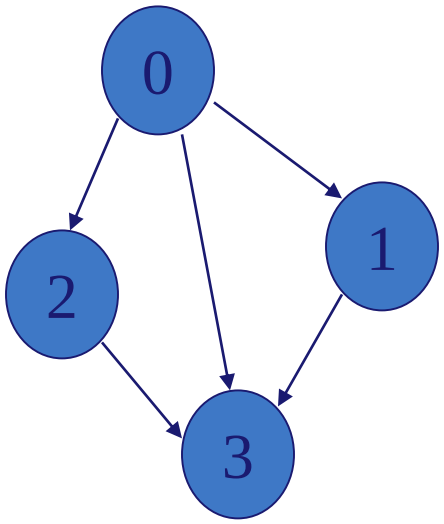
Biểu diễn bằng ma trận kề(tt)



A[i][j]	0	1	2	3
0	0	1	1	0
1	1	0	1	1
2	1	1	0	1
3	0	1	1	0

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

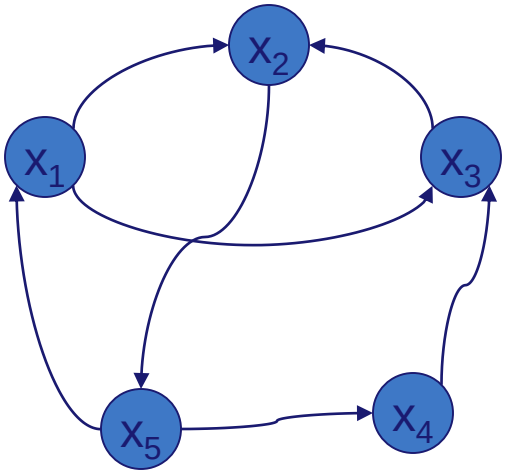
Biểu diễn bằng ma trận kề(tt)



A[i][j]	0	1	2	3
0	0	1	1	1
1	0	0	0	1
2	0	0	0	1
3	0	0	0	0

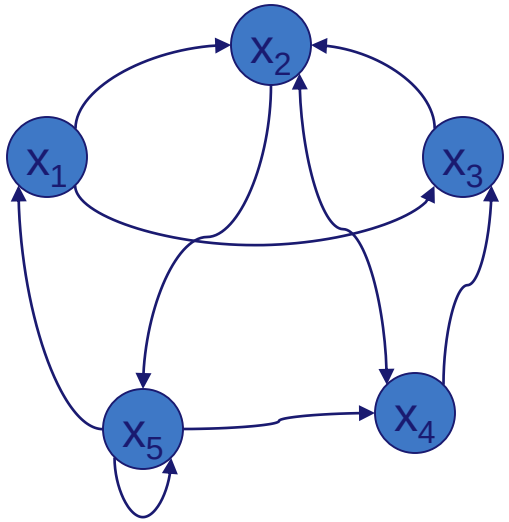
$$A = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

Biểu diễn bằng ma trận kề(tt)



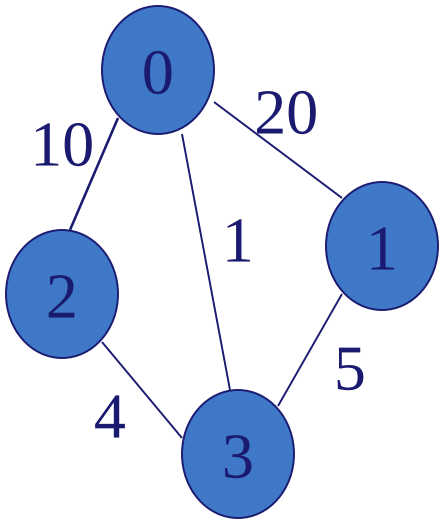
$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Biểu diễn bằng ma trận kề (tt)



$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \end{pmatrix}$$

Biểu diễn bằng ma trận kề (tt)



A[i][j]	0	1	2	3
0	0	20	10	1
1	20	0	0	5
2	10	0	0	4
3	1	5	4	0

$$A = \begin{pmatrix} 0 & 20 & 10 & 1 \\ 20 & 0 & 0 & 5 \\ 10 & 0 & 0 & 4 \\ 1 & 5 & 4 & 0 \end{pmatrix}$$

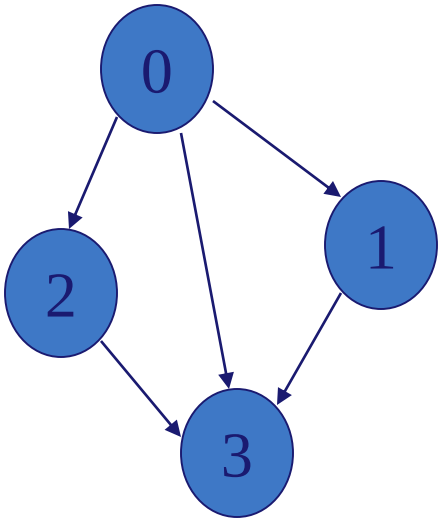
Biểu diễn bằng ma trận kề (tt)

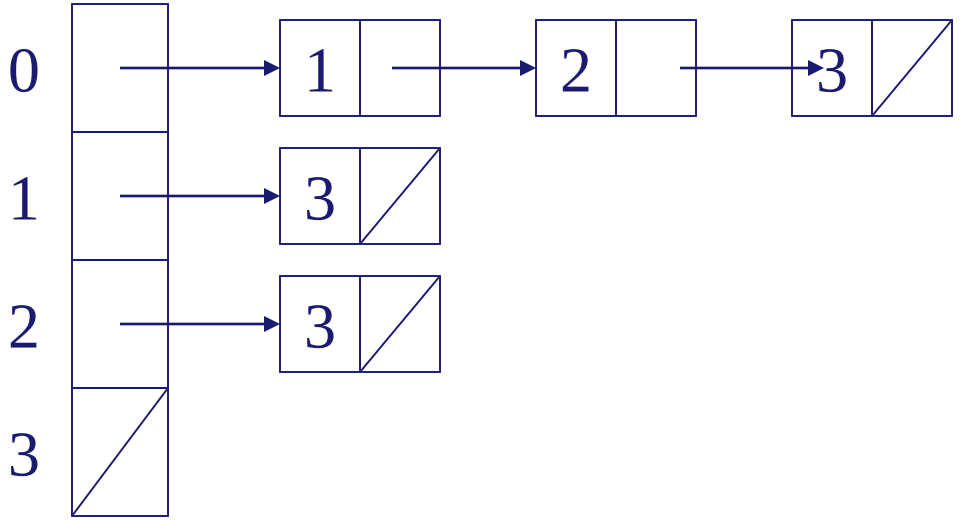
- Chú ý
  - Đối với đồ thị không định hướng, ma trận kề là ma trận đối xứng
  - Đối với đồ thị định hướng, số lượng phần tử 0 khá lớn
  - Đối với đồ thị có trọng số, thay thế giá trị 1 bằng giá trị trọng số

Biểu diễn đồ thị bằng danh sách kề

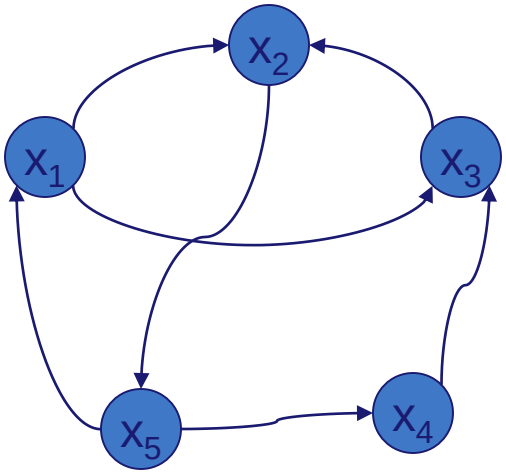
- Là một mảng các danh sách
- Ở đây, n hàng của ma trận kề thay thế bằng n danh sách liên kết động
- Mỗi đỉnh của G có một danh sách, mỗi nút trong danh sách thể hiện các đỉnh lân cận của nút này
  - Cấu trúc mỗi nút
    - id: tên đỉnh (chỉ số, danh hiệu)
    - next: con trỏ đến nút kế tiếp

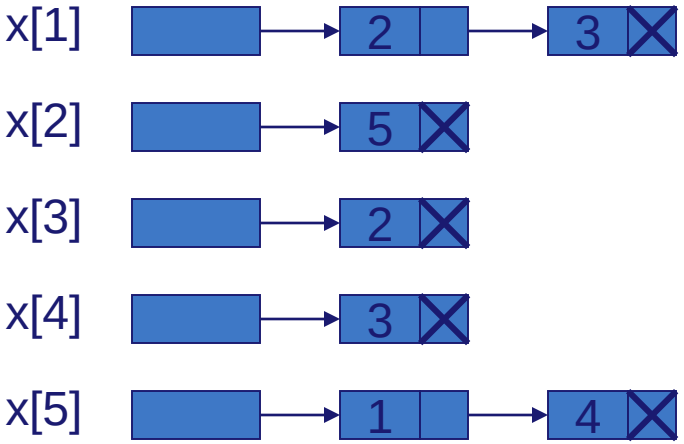
Biểu diễn đồ thị bằng danh sách kề (tt)



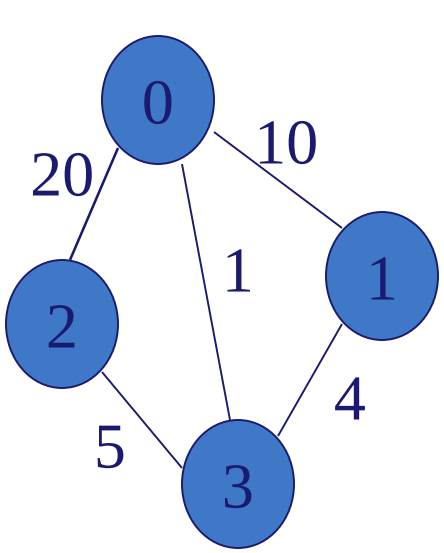


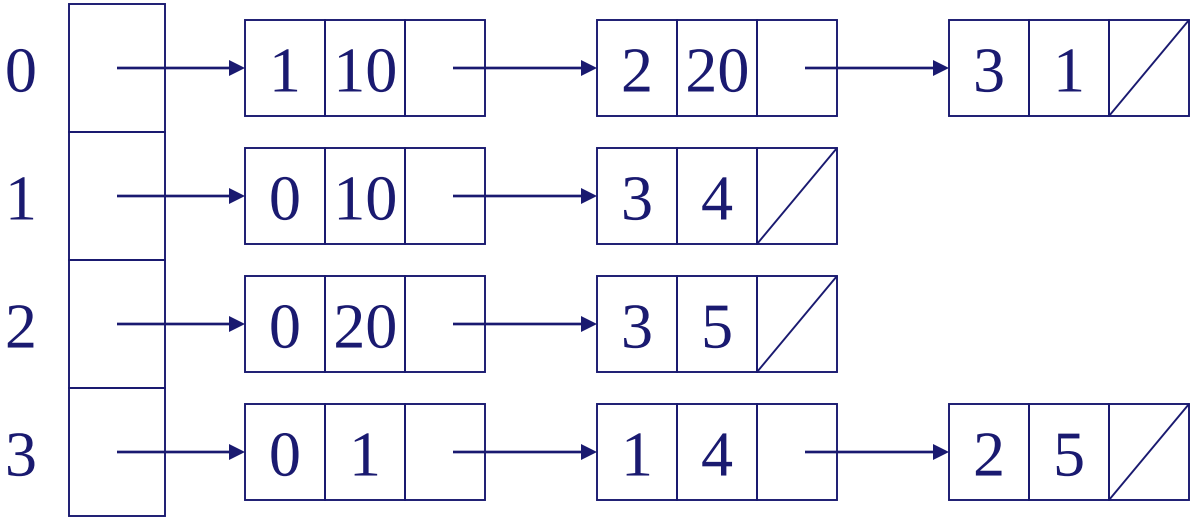
Biểu diễn đồ thị bằng danh sách kề (tt)





Biểu diễn đồ thị bằng danh sách kề (tt)





Biểu diễn đồ thị bằng danh sách kề (tt)

- Chú ý
  - Các nút đầu danh sách được lưu vào một mảng (truy cập nhanh)
  - Với đồ thị không định hướng có n đỉnh và e cạnh, thì cần n nút đầu và 2e nút ‘trong’ danh sách
  - Với đồ thị định hướng có n đỉnh và e cạnh, thì chỉ cần e nút ‘trong’ danh sách
  - Thứ tự các nút không quan trọng

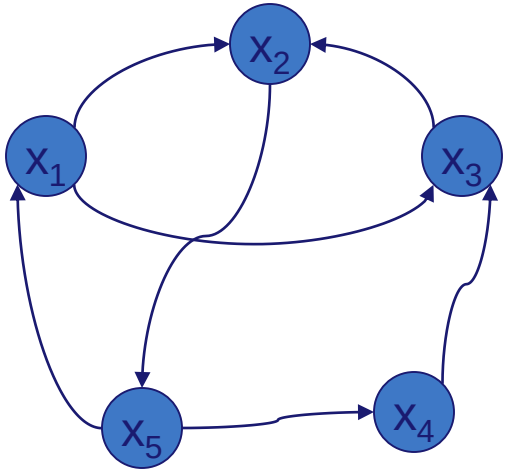
Phép duyệt đồ thị

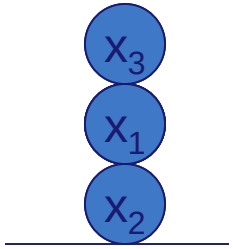
- Từ một đỉnh, liệt kê tất cả các đỉnh của đồ thị
  - Phép tìm kiếm theo chiều sâu
    - Depth first search
  - Phép tìm kiếm theo chiều rộng
    - Breadth first search

Phép tìm kiếm theo chiều sâu

- Ý tưởng
  - Tại đỉnh v bất kỳ, duyệt đỉnh v, và xét tập các đỉnh đến được từ đỉnh v
  - Lập lại thao tác trên đối với đỉnh w bất kỳ trong tập các đỉnh từ v nói trên

Phép tìm kiếm theo chiều sâu (tt)





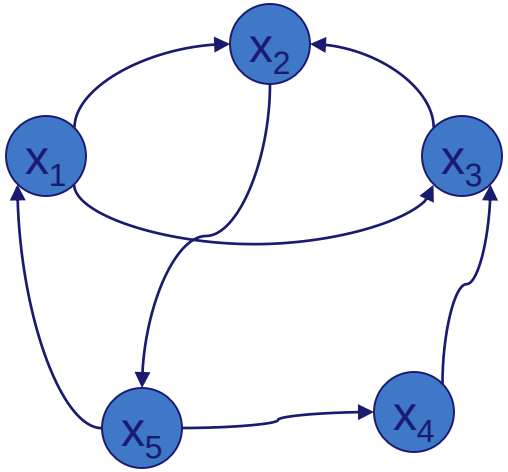
Phép tìm kiếm theo chiều sâu (tt)

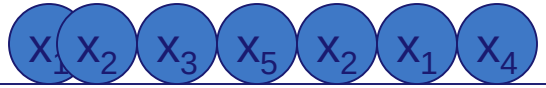
- Nhận xét
  - Thời gian thực hiện giải thuật ~ (n+e), nếu G được biểu diễn bằng danh sách kề
  - Thời gian thực hiện giải thuật ~ n^2, nếu G được biểu diễn bằng ma trận kề
  - Giải thuật này sử dụng để chứng minh một đồ thị có liên thông hay không

Phép tìm kiếm theo chiều rộng

- Tại điểm v bất kỳ, duyệt đỉnh v, thu được tập hợp W gồm các đỉnh w xuất phát từ v
- Lặp lại thao tác trên đối với tất cả các đỉnh w trong W, thu được tập hợp đỉnh Z
- Lặp lại thao tác trên đối với tất cả các đỉnh z trong Z
- Lặp lại cho đến khi tất cả mọi đỉnh đều được duyệt qua ít nhất một lần

Phép tìm kiếm theo chiều rộng(tt)



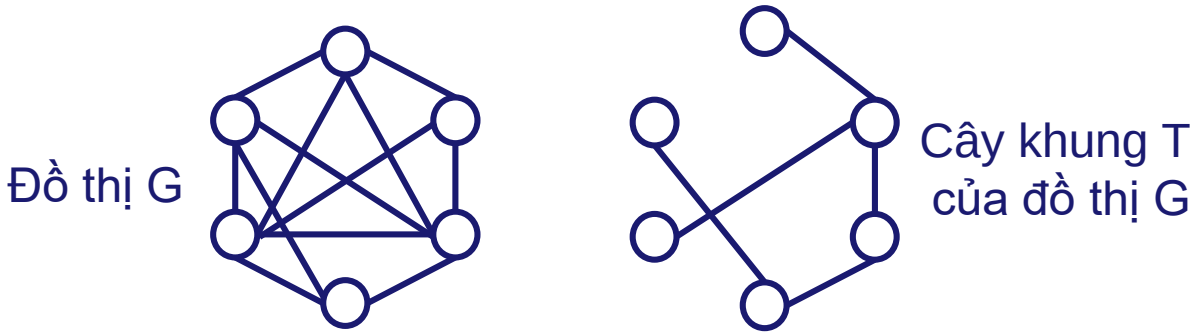


Phép tìm kiếm theo chiều rộng(tt)

- Nhận xét
  - Thời gian thực hiện giải thuật ~ (n+e), nếu G được biểu diễn bằng danh sách kề
  - Thời gian thực hiện giải thuật ~ n^2, nếu G được biểu diễn bằng ma trận kề

Cây khung (Spanning tree)

- T=(V,E') $\subseteq$ G=(V,E), với E $\supseteq$ E' bao gồm các cung thuộc một phép duyệt từ một đỉnh đến các đỉnh còn lại trong V
- Giá của cây khung T = tổng trọng số của các cung thuộc E'

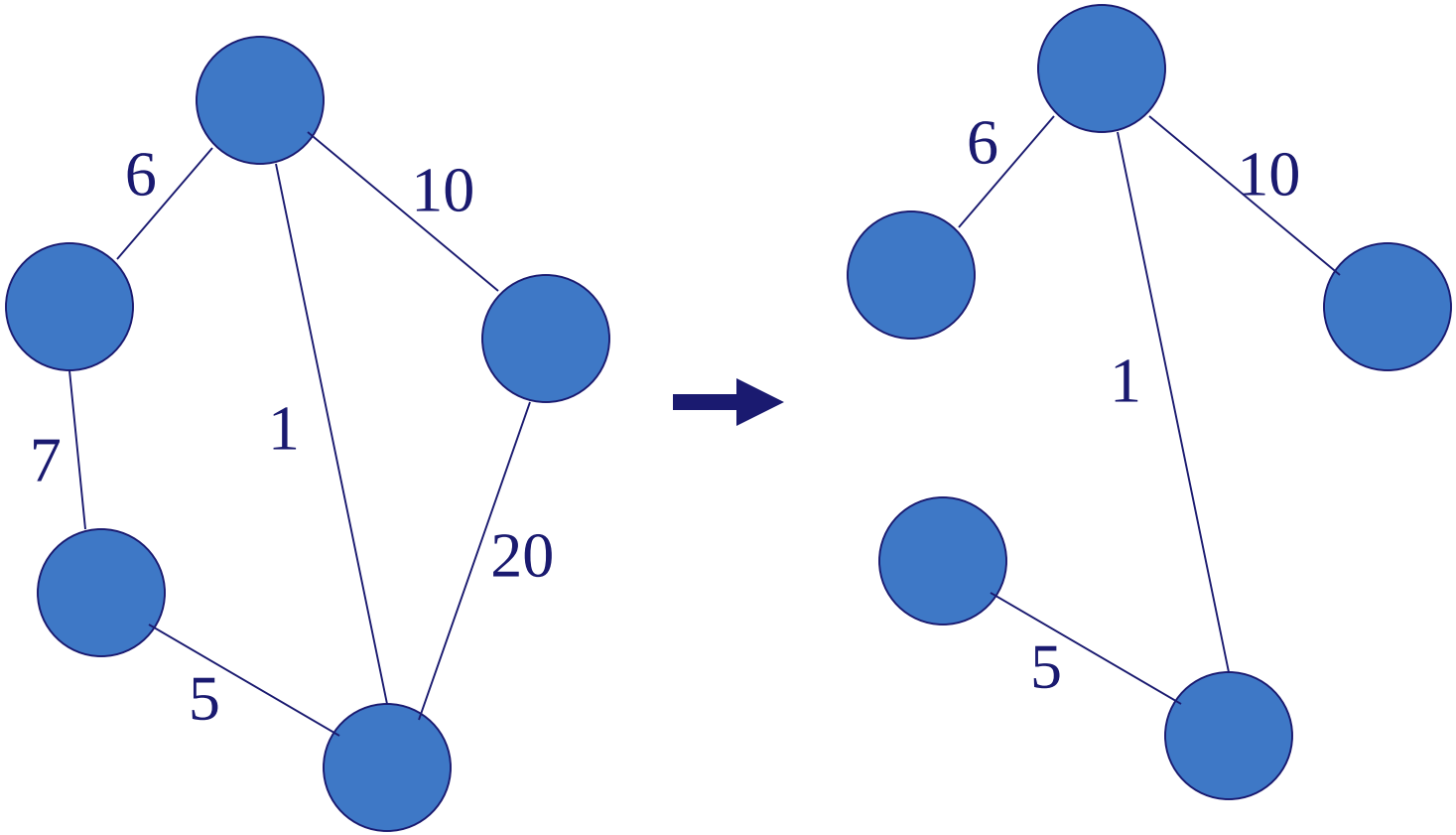


Cây khung (Spanning tree)

- Chú ý
  - Một đồ thị G có thể có nhiều cây khung
  - Cây khung theo chiều rộng, theo chiều sâu
  - Các cung trong cây khung không tạo nên chu trình
    - Giữa hai đỉnh trong một cây khung chỉ tồn tại duy nhất một đường đi từ đỉnh này đến đỉnh kia
  - Nếu đồ thị có n đỉnh, thì cây khung có n-1 cạnh

Cây khung cực tiêu

- Là cây khung với tổng các trọng số là cực tiểu

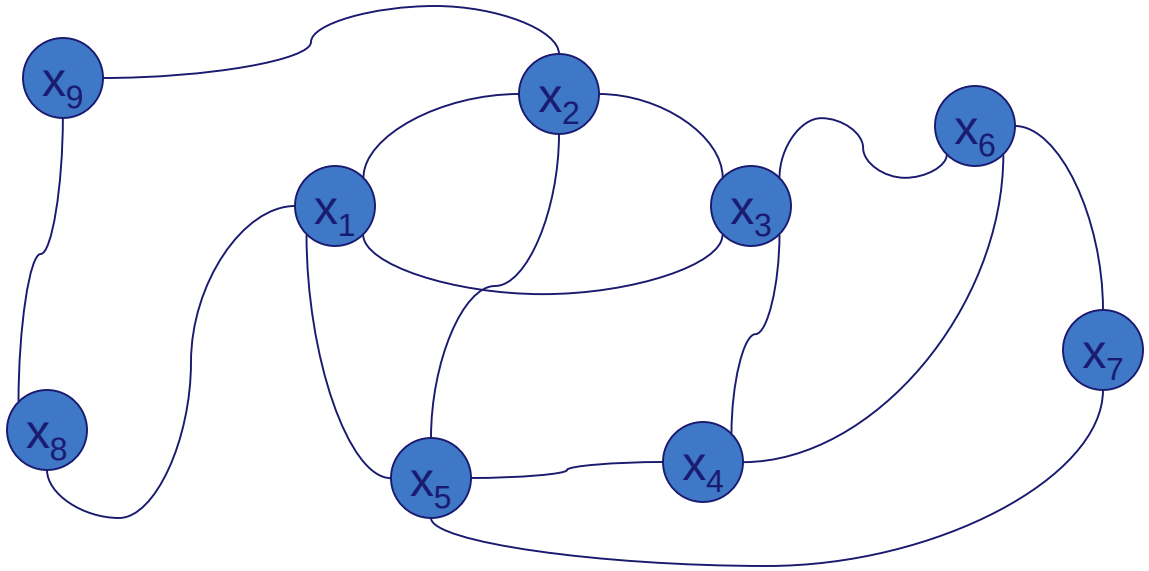


Thuật toán Kruskal

- Sắp xếp các cung theo thứ tự không giảm đối với trọng số
- Bắt đầu từ T=Ø
- Lặp cho đến khi ko còn đỉnh nào (|E'| = n-1)
  - lấy ra cung w có trọng số nhỏ nhất
  - thêm cung w vào T với điều kiện không tạo chu trình trong T

Cây khung (Spanning tree)

- Ví dụ: Cho một đồ thị G vô hướng, liên thông, có trọng số. Hãy tìm cây khung cực tiểu của G



Thuật toán Kruskal

- Để kiểm tra xem có tạo ra chu trình trong T hay không, chúng ta xem hai đỉnh của cung được thêm có thuộc tập các đỉnh hiện có trong T không, nếu có, nghĩa là sẽ tạo nên chu trình

Bài toán bao đóng truyền ứng

- Cho đồ thị G = (V,E)
  - Có tồn tại đường đi giữa hai nút x và y trong đồ thị G hay không?
  - Bài toán này có thể giải được dễ dàng bằng cách sử dụng ma trận kề của đồ thị

Bài toán bao đóng truyền ứng

- Ma trận kề A của đồ thị G=(V,E)
  - a_{ij}=true, nếu $\exists (x_i,x_j) \in E$
  - a_{ij}=false, nếu ngược lại
- Phép cộng $A=(a_{ij})$, $B=(b_{ij})$
  - $A \vee B = C$, với $c_{ij}=a_{ij} \vee b_{ij}$
- Phép nhân $A=(a_{ij})$, $B=(b_{ij})$
  - $D = A \wedge B$, với $d_{ij}=\bigvee_{k=1}^{n}(a_{ik} \wedge b_{kj})$

  Với 1<=i, j, <=n

Bài toán bao đóng truyền ứng

- Với ma trận A, nếu $a_{ij} = 1$, có nghĩa là có một cung từ i tới j.
- Xét $A^{(2)} = A \wedge A$. Rõ ràng nếu phần tử ở hàng i, cột j của $A^{(2)}$ bằng 1, thì có ít nhất có một đường đi có độ dài 2, từ đỉnh i đến đỉnh j, vì

  $$a^{(2)}_{ij} = \bigvee_{k=1}^{n} (a_{ik} \wedge a_{kj})$$

  - $a_{ik} \wedge a_{kj} = 1$, khi $a_{ik} = 1$ và $a_{kj} = 1$, => tức là có đường đi độ dài 1 từ i tới k và có đường đi đô dài 1 từ k tới j

Bài toán bao đóng truyền ứng

- Từ đó suy ra $A^{(r)} = A \wedge A^{(r-1)}$, R=2, 3,.. Nghĩa là $a^{(r)}_{ij} = 1$, thì có ít nhất 1 đường đi đô dài r, từ i tới j.
- Ta lập ma trân P = A V $A^{(2)}$ V.. V $A^{(n)}$
- Thì P sẽ cho biết có hay không một đường đi có độ dài lớn nhất là n, từ đỉnh i tới j. P được gọi là ma trận đường đi.

Bài toán bao đóng truyền ứng

- Thuật toán WARSHALL

```
Void WARSHALL(A, P, n){
  For (int k=0;k<n;k++)
    For (int i=0;i<n;i++)
      For (int j=0;j<n;j++)
        P[i,j]=P[i,j] V (P[i,k] Λ P[k,j])
  }
```

Bài toán đường đi ngắn nhất

- Vấn đề
  - Cho một đồ thị định hướng, liên thông, có trọng số G
  - Hãy tìm đường đi ngắn nhất từ một đỉnh đến tất cả các đỉnh khác trong đồ thị

Thuật toán Dijkstra

- Xét đồ thị có hướng G=(V,E), với |V|=n
- Ma trận trọng số d[u,v]≥0, ∀(u,v)∈E
- s∈V là điểm xuất phát
- H[v]=chiều dài cực tiểu từ s đến v (v∈V)

Thuật toán Dijkstra

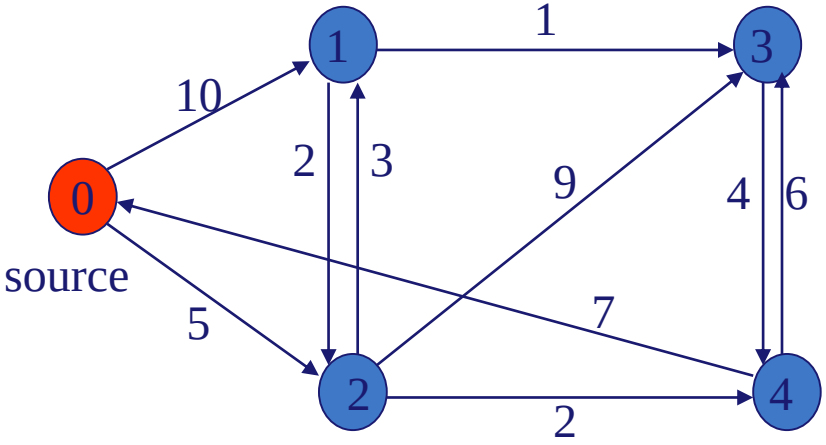
- Bắt đầu duyệt từ đỉnh s
- Gán giá trị cho H[v]
  - H[v]=d(s,v), nếu (s,v)$\in$E
  - H[v]=∞, nếu ngược lại
- Lặp lại cho đến khi duyệt hết các đỉnh
  - Chọn đỉnh w chưa duyệt có H[w] nhỏ nhất
  - Duyệt đỉnh w này
  - Với các đỉnh t chưa duyệt khác
    - H[t] = min(H[t],H[w]+d(w,t))

Thuật toán Dijkstra

- Hoạt động tốt trên đồ thị trọng số dương
- Độ phức tạp giải thuật là $O(n^2)$

Thuật toán Dijkstra

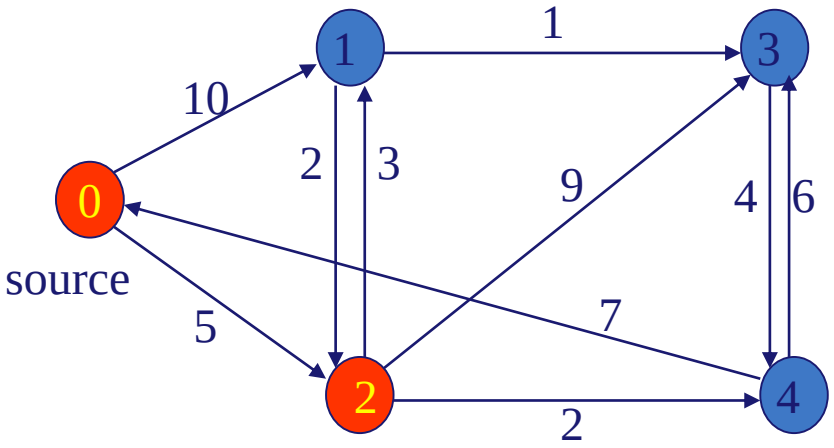
- Tìm đường đi từ v_0 tới các đỉnh còn lại



node	from node V_0 to other nodes			
V_1	10			
V_2	5			
V_3	∞			
V_4	∞			
best				

Thuật toán Dijkstra

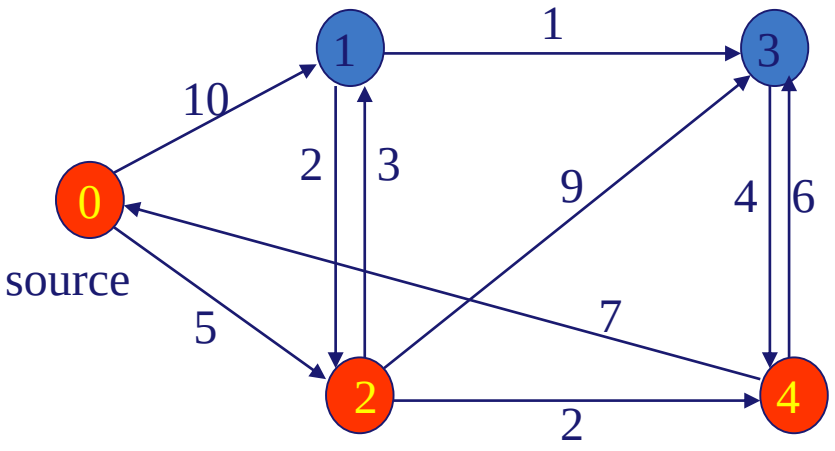
- step 1: tìm đường đi ngắn nhất từ 0
  - node 2 được chọn



node	from node V_0 to other nodes			
V_1	10			
V_2	5			
V_3	∞			
V_4	∞			
best	V_2			

Thuật toán Dijkstra

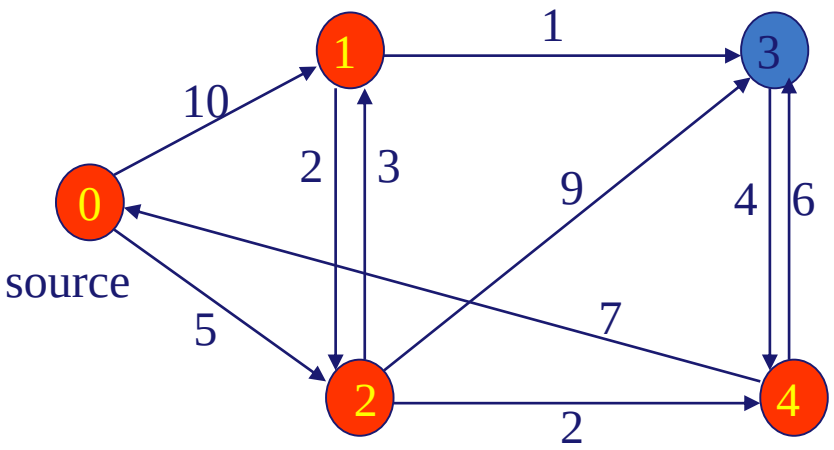
- step 2: Tính toán lại các đường đi đến tất cả các đỉnh
  - Tìm đường đi ngắn nhất đến node 0. Node 4 được chọn



node	from node V_0 to other nodes			
V_1	10	8		
V_2	5	5		
V_3	∞	14		
V_4	∞	7		
best	V_2	V_4		

Thuật toán Dijkstra

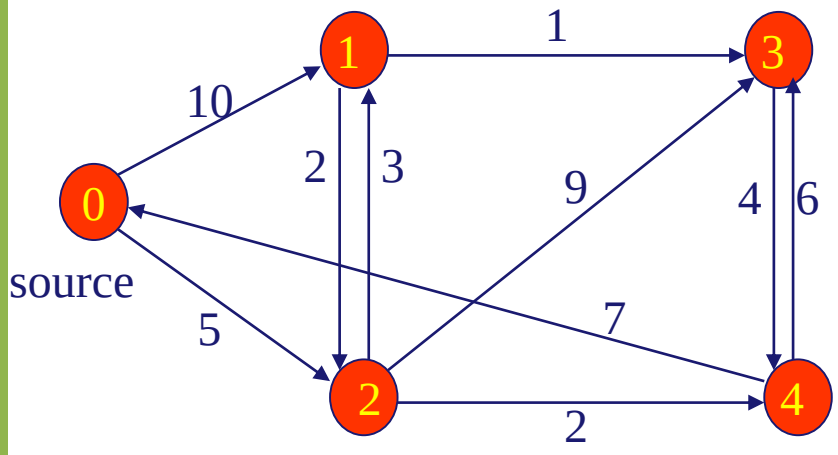
- step 2: Tính toán lại các đường đi đến tất cả các đỉnh
  - Tìm đường đi ngắn nhất từ node 0. node 1 được chọn



node	from node V_0 to other nodes			
V_1	10	8	8	
V_2	5	5	5	
V_3	∞	14	13	
V_4	∞	7	7	
best	V_2	V_4	V_1	

Thuật toán Dijkstra

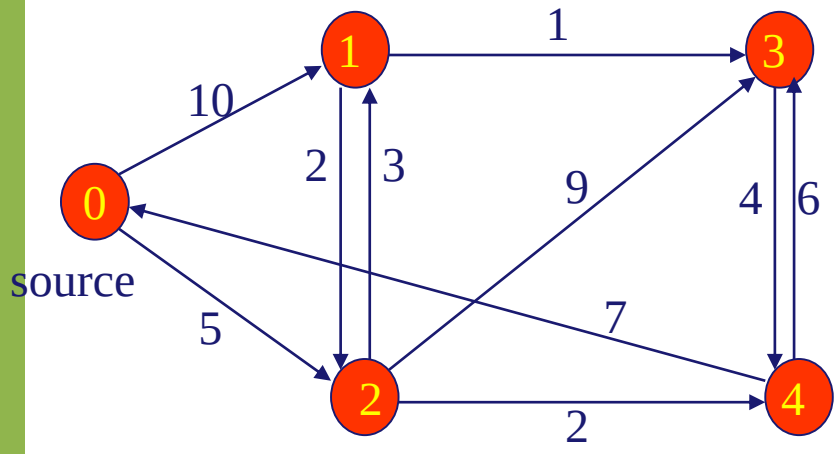
- step 2: Tính toán lại các đường đi đến tất cả các đỉnh
  - Tìm đường đi ngắn nhất từ node 0. node 3 được chọn



node	from node V_0 to other nodes			
V_1	10	8	8	8
V_2	5	5	5	5
V_3	∞	14	13	9
V_4	∞	7	7	7
best	V_2	V_4	V_1	V_3

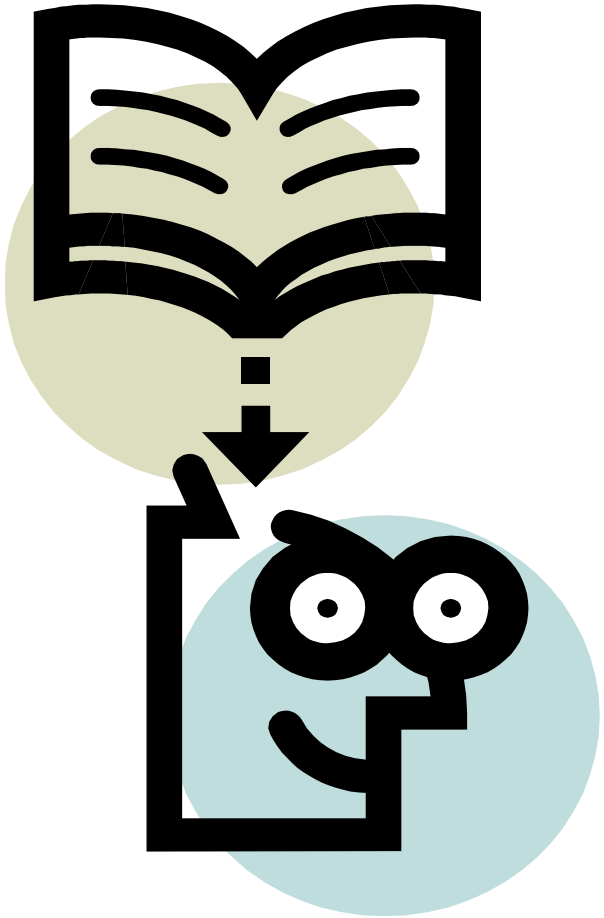
Thuật toán Dijkstra

- Chúng ta có tất cả các đường đi từ v_0



node	from node V_0 to other nodes			
V_1	10	8 (0,2)	8 (0,2)	8
V_2	5 (0,2)	5	5	5
V_3	∞	14 (0,2,3)	13 (0,2,4,3)	9 (0,2,1,3)
V_4	∞	7 (0,2,4)	7	7
best	V_2	V_4	V_1	V_3

Q&A



CẤU TRÚC DỮ LIỆU

CHƯƠNG 3

TÌM KIẾM & SẮP XẾP

Nội dung trình bày

- Tìm kiếm
- Sắp xếp

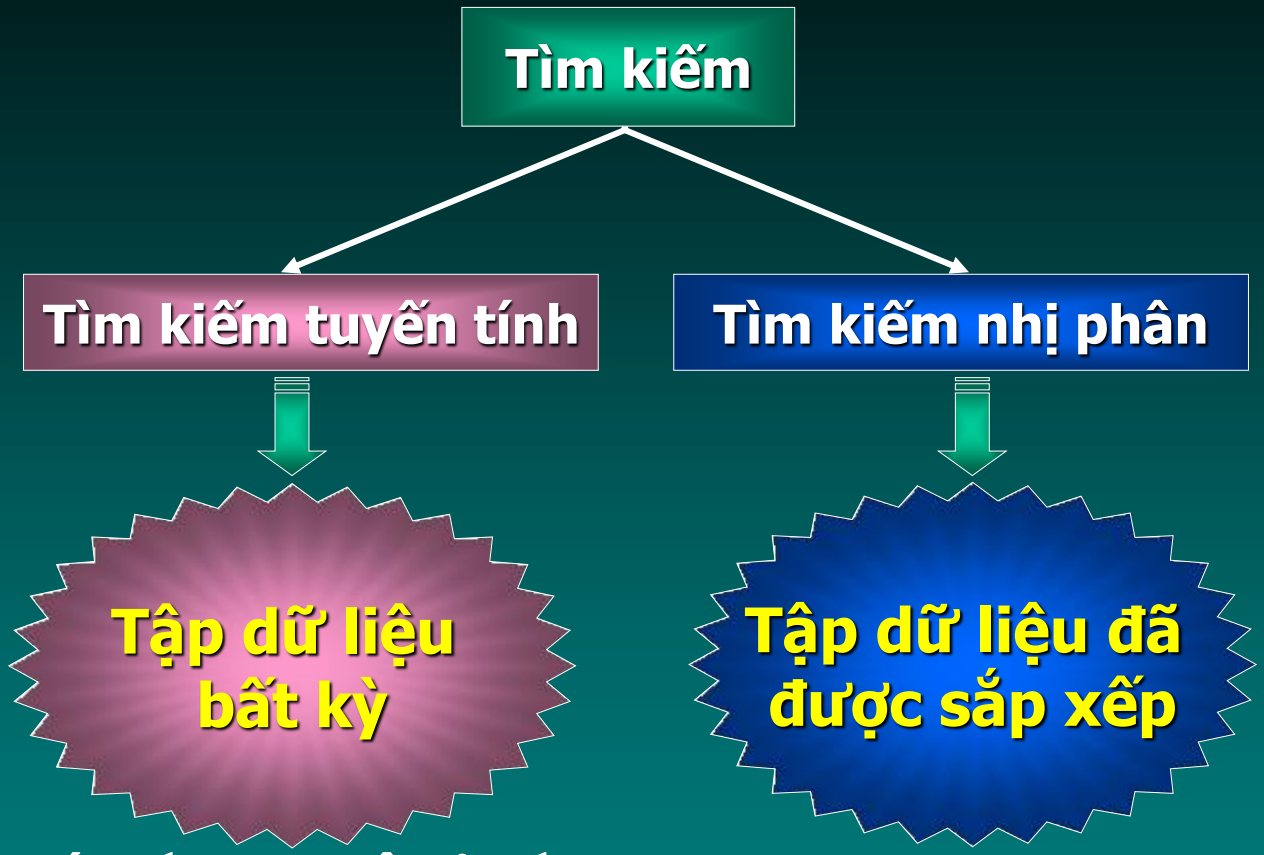
2.1 Tìm kiếm

- Tìm kiếm là thao tác quan trọng & thường xuyên trong tin học.
  - Tìm kiếm một nhân viên trong danh sách nhân viên.
  - Tìm một sinh viên trong danh sách sinh viên của một lớp…
  - Tìm kiếm một tên sách trong thư viện.

2.1 Tìm kiếm (2)

- Tìm kiếm là quá trình xác định một đối tượng nào đó trong một tập các đối tượng. Kết quả trả về là đối tượng tìm được (nếu có) hoặc một chỉ số (nếu có) xác định vị trí của đối tượng trong tập đó.
- Việc tìm kiếm dựa theo một trường nào đó của đối tượng, trường này là khóa (key) của việc tìm kiếm.
- VD: đối tượng sinh viên có các dữ liệu {MaSV, HoTen, DiaChi,…}. Khi đó tìm kiếm trên danh sách sinh viên thì khóa thường chọn là MaSV hoặc HoTen.

2.1 Tìm kiếm (3)



- Bài toán được mô tả như sau:
  - Tập dữ liệu được lưu trữ là dãy $a_1, a_2,..,a_n$. Giả sử chọn cấu trúc dữ liệu mảng để lưu trữ dãy số này trong bộ nhớ chính, có khai báo: int a[n];
  - Khóa cần tìm là x, có kiểu nguyên: int x;

2.1.1 Tìm kiếm tuyến tính (4)

- Ý tưởng chính: duyệt tuần tự từ phần tử đầu tiên, lần lượt so sánh khóa tìm kiếm với khoá tương ứng của các phần tử trong danh sách. Cho đến khi gặp phần tử cần tìm hoặc đến khi duyệt hết danh sách.
- Các bước tiến hành như sau:
  - Bước 1: i = 1;
  - Bước 2: So sánh a[i] với x, có hai khả năng
    - A[i] = x: Tìm thấy. ⇒ Dừng
    - A[i] ≠ x: Sang bước 3
  - Bước 3: i = i + 1 // xét phần tử kế tiếp trong mảng
    - Nếu i > N: Hết mảng, không tìm thấy. ⇒ Dừng
    - Nếu i ≤ N: Quay lại bước 2

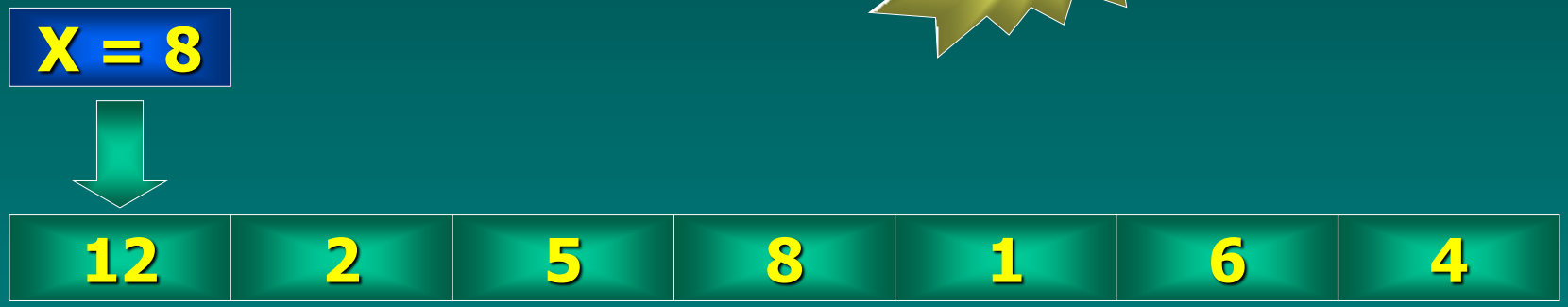
2.1.1 Tìm kiếm tuyến tính (5)

Minh họa tìm kiếm tuyến tính

Ví dụ

Cho dãy số a, giá trị tìm x = 8:

12 2 5 8 1 6 4



2.1.1 Tìm kiếm tuyến tính (6)

Thuật toán tìm kiếm tuyến tính

```
int Search(int a[], int n, int key)
{
    int i =0;
    while ((i<n) && (key != a[i]))
        i++;
    if (i >= n)
        return -1;              // tìm không thấy
    else
        return i;               // tìm thấy tại vị trí i
}
```

2.1.1 Tìm kiếm tuyến tính (7)

Thuật toán tìm kiếm tuyến tính cải tiến

```
int  Search(int a[], int n, int key)
{
    int i =0;
    a[n] =key;          // thêm phần tử thứ n+1
    while (key != a[i])
        i++;
    if (i == n)
        return -1;      // tìm hết mảng nhưng không có x
    else
        return i;       // tìm thấy x tại vị trí i
}
```

5.1.1 Tìm kiếm tuyến tính (8)

Nhận xét

- Giải thuật tìm kiếm tuyến tính không phụ thuộc vào thứ tự của các phần tử trong mảng, do vậy đây là phương pháp tổng quát nhất để tìm kiếm trên một dãy bất kỳ
- Một thuật toán có thể được cài đặt theo nhiều cách khác nhau, kỹ thuật cài đặt ảnh hưởng nhiều đến tốc độ thực hiện. Ví dụ như thuật toán Search cải tiến sẽ chạy nhanh hơn thuật toán trước do vòng lặp while chỉ so sánh một điều kiện...

5.1.2 Tìm kiếm nhị phân

KN

Phép tìm kiếm nhị phân được áp dụng trên dãy khóa đã có thứ tự: $k[1] \le k[2] \le ... \le k[n]$.

- Phương pháp này dựa trên ý tưởng sau:
  - Giả sử ta cần tìm trong đoạn a[left...right] với khoá tìm kiếm là x, trước hết ta xét phần tử giữa a[mid], với mid = (left + right)/2.
    - Nếu a[mid] < x thì có nghĩa là đoạn a[left] đến a[right] chỉ chứa khóa < x, ta tiến hành tìm kiếm từ a[mid+1] đến a[right].
    - Nếu a[mid] > x thì có nghĩa là đoạn a[m] đến a[right] chỉ chứa khoá > x, ta tiến hành tìm kiếm từ a[left] đến a[mid-1].
    - Nếu a[mid] = x thì việc tìm kiếm thành công.
    - Quá trình tìm kiếm thất bại nếu left > right.

2.1.2 Tìm kiếm nhị phân (2)

Các bước tiến hành

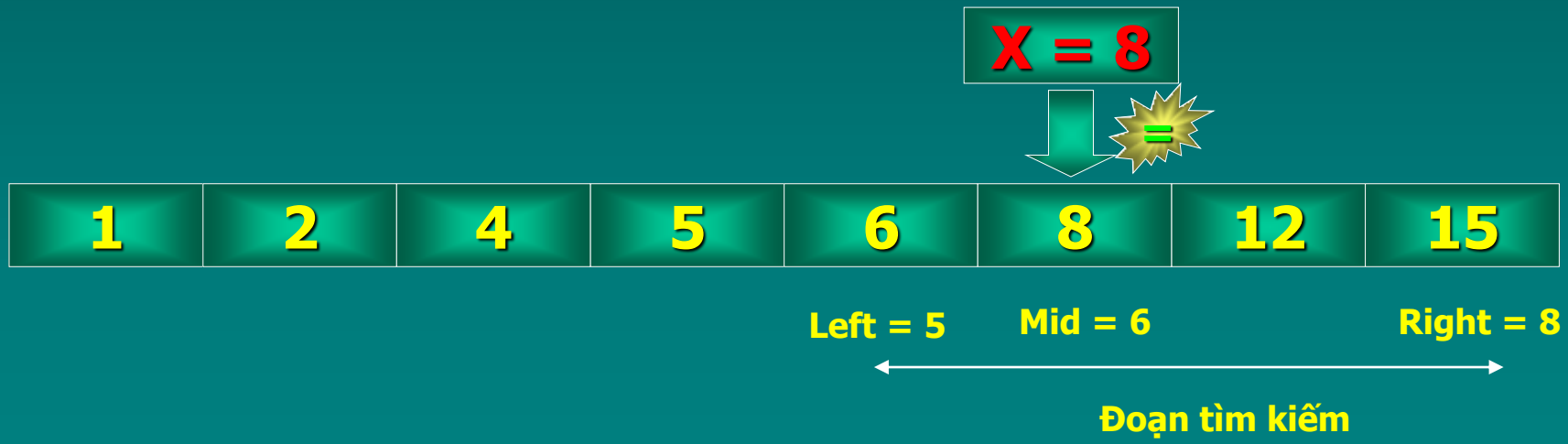
- B1: left =1, right = n // tìm kiếm trên tất cả phần tử
- B2: mid = (left + right)/2 // lấy mốc so sánh
  - So sánh a[mid] với x, có 3 khả năng
    - a[mid] = x: Tìm thấy ⇒ Dừng
    - a[mid] > x: // tìm tiếp trong dãy a[left]...a[mid-1]
      right = mid -1;
    - A[mid] < x: // tìm tiếp trong dãy a[mid+1]...a[right]
      left = mid +1
- B3:
  - Nếu left ≤ right // còn phần tử ⇒ tìm tiếp ⇒ Lặp B2
  - Ngược lại: Dừng // đã xét hết các phần tử

2.1.2 Tìm kiếm nhị phân (3)

Ví dụ

cho dãy số gồm 8 phần tử bên dưới và x = 8:

1 2 4 5 6 8 12 15



2.1.2 Tìm kiếm nhị phân (4)

Thuật toán tìm kiếm NP BinarySearch

```
int BinarySearch(int key)
{
    int left = 0, right = n-1, mid;
    while (left <= right)
    {
        mid = (left + right)/ 2;          // lấy điểm giữa
        if (a[mid] == key)                // nếu tìm được
            return mid;
        if (a[mid] < key)                 // tìm đoạn bên phải mid
            left = mid+1;
        else
            right = mid-1;                // tìm đoạn bên trái mid
    }
    return -1;                            // không tìm được
}
```

2.1.2 Tìm kiếm nhị phân (5)

Nhận xét

- Thuật giải nhị phân dựa vào quan hệ giá trị của các phần tử trong mảng để định hướng trong quá trình tìm kiếm, do vậy chỉ áp dụng được với dãy đã có thứ tự.
- Thuật giải nhị phân tìm kiếm nhanh hơn tìm kiếm tuyến tính.
- Tuy nhiên khi áp dụng thuật giải nhị phân thì cần phải quan tâm đến chi phí cho việc sắp xếp mảng. Vì khi mảng được sắp thứ tự rồi thì mới tìm kiếm nhị phân.

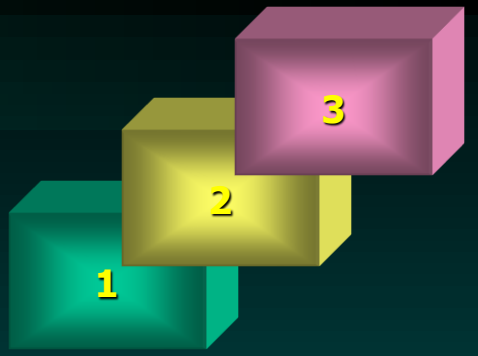
2.2 Sắp xếp

- Sắp xếp là quá trình bố trí lại các phần tử của một tập đối tượng theo một thứ tự nhất định.
- Ví dụ:
  - {1, 2, 5, 7, 9, 12}, {14, 12, 7, 5, 2, 1}
  - {“An” “Binh” “Dương” “Hương”}
- Việc sắp xếp là một bài toán phổ biến trong tin học.
  - Do các yêu cầu tìm kiếm thuận lợi, sắp xếp kết xuất cho các bảng biểu...

2.2 Sắp xếp

- Dữ liệu thường được tổ chức thành mảng các mẫu tin dữ liệu
- Mỗi mẫu tin thường có một số các trường dữ liệu khác nhau.
- Trường tham gia quá trình tìm kiếm gọi là khoá (key).
- Việc sắp xếp sẽ được tiến hành dựa vào giá trị khoá này.

2.2 Sắp xếp (2)



Các phương pháp sắp xếp

1. Selection Sort (*)
2. Insertion Sort (*)
3. Bubble Sort (*)
4. Interchange Sort
5. Shell sort
6. Quick sort
7. Radix...

2.2 Sắp xếp (3)

Mô tả bài toán

- Để tiện cho việc minh họa các thuật toán sắp xếp ta mô tả bài toán như sau:
  - Cho một mảng các phần tử e, mỗi phần tử trong mảng có một thuộc tính khóa. Hãy sắp xếp tăng hoặc giảm các phần tử trong mảng theo giá trị khóa này
    - Do mỗi phần tử có giá trị khoá nên ta gọi k[1..n] là mảng các khóa của các phần tử trong e.
    - Yêu cầu: sắp xếp các giá trị này sao cho mảng k có thứ tự tăng hoặc giảm.

2.2.1 Selection Sort

Ý tưởng chính

- Lượt thứ nhất, chọn trong dãy khoá k[1..n] ra khoá nhỏ nhất và đổi giá trị với k[1], khi đó k[1] sẽ trở thành khoá nhỏ nhất.
- Lượt thứ hai, chọn trong dãy khoá k[2..n] ra khóa nhỏ nhất và đổi giá trị với k[2].
- ...
- Lượt n-1, chọn giá trị nhỏ nhất trong k[n-1] và k[n] ra khoá nhỏ nhất và đổi giá trị với k[n-1].

2.2.1 Selection Sort (2)

Các bước thực hiện

- B1: i = 1
- B2: Tìm phần tử a[min] nhỏ nhất trong dãy hiện hành từ a[i] đến a[n]
- B3: Hoán vị a[i] và a[min]
- B4: Nếu i < n -1 thì i= i+1 ⇒ Lặp B2

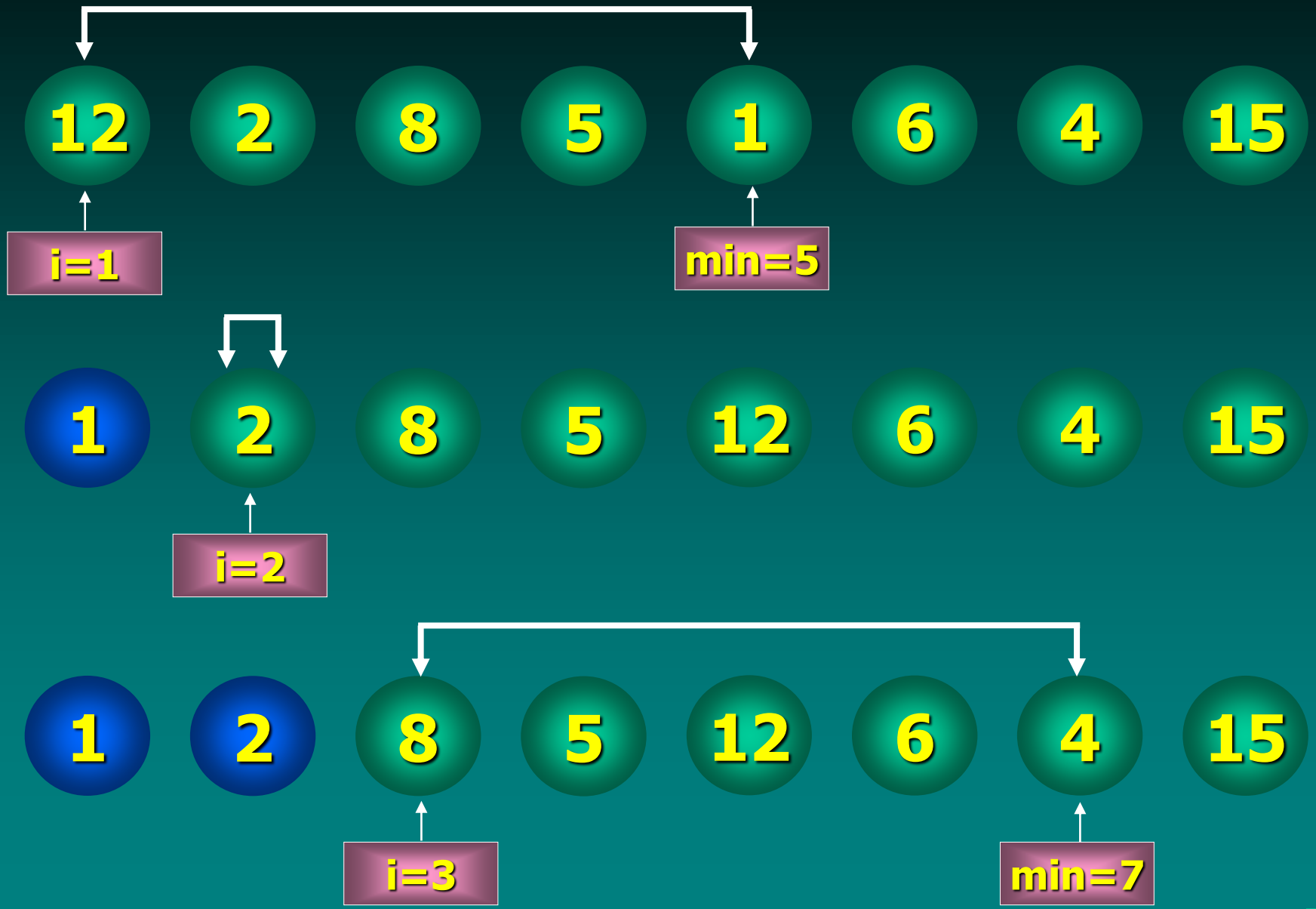
  Ngược lại ⇒ Dừng

VD cho dãy số như sau:

12 2 8 5 1 6 4 15

Minh họa phương pháp chọn như sau

2.2.1 Selection Sort (3)



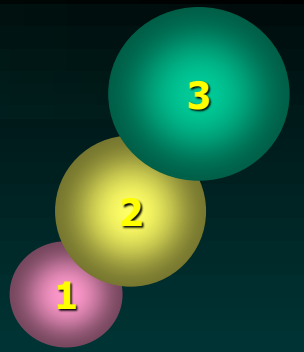
2.2.1 Selection Sort (4)

2.2.1 Selection Sort (5)

Cài đặt SelectionSort

```
void SelectionSort(int a[], int n)
{
    int min;    // lưu chỉ số phần tử nhỏ nhất
    for(int i = 0; i < n-1; i++) // duyệt qua n-1 phần tử
    {
        min = i;
        for(int j = i+1; j < n; j++)
            if (a[j] < a[min])
                min = j;
        Swap(a[min], a[i]);
    }
}
```

2.2.2 Bubble Sort



Ý tưởng chính

- Xuất phát từ cuối dãy, đổi chỗ các cặp phần tử kế cận để đưa phần tử nhỏ hơn về đầu.
- Sau đó ở bước tiếp theo không xét phần tử đó nữa. Do vậy lần xử lý thứ i sẽ có vị trí đầu dãy là i.
- Lặp lại xử lý trên cho đến khi không còn cặp phần tử nào được xét.

2.2.2 Bubble Sort (2)

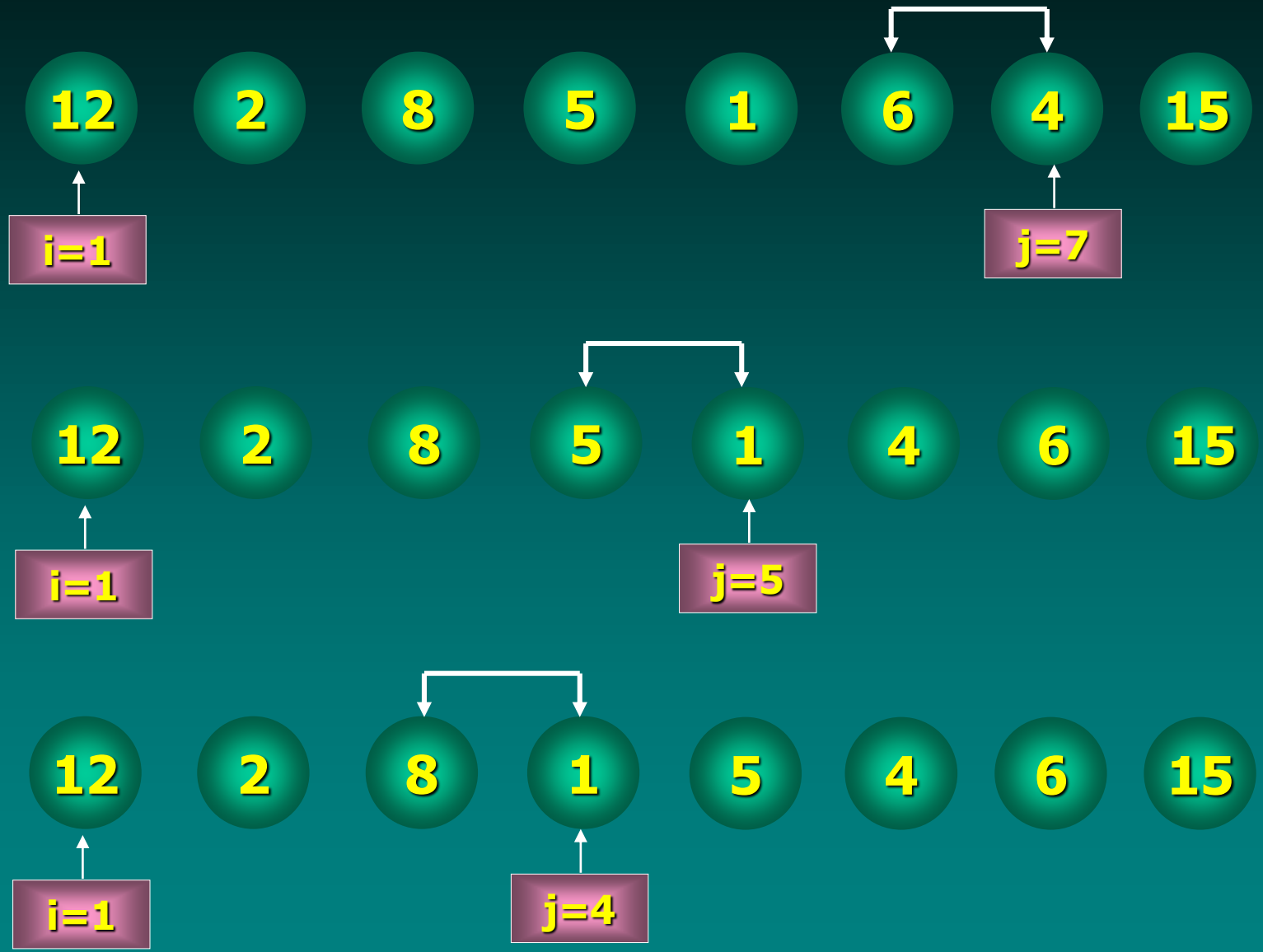
Các bước tiến hành

- B1: i=1; // lần xử lý đầu tiên
- B2: j=n; // duyệt từ cuối dãy ngược về vị trí i
  - Trong khi (j>i) thực hiện:
    - Nếu a[j] < a[j-1]: Hoán đổi a[j] và a[j-1]
    - j = j -1;
- B3: i = i+1; // lần xử lý kế tiếp
  - Nếu i > n-1: Hết dãy ⇒ Dừng
  - Ngược lại: quay lại B2

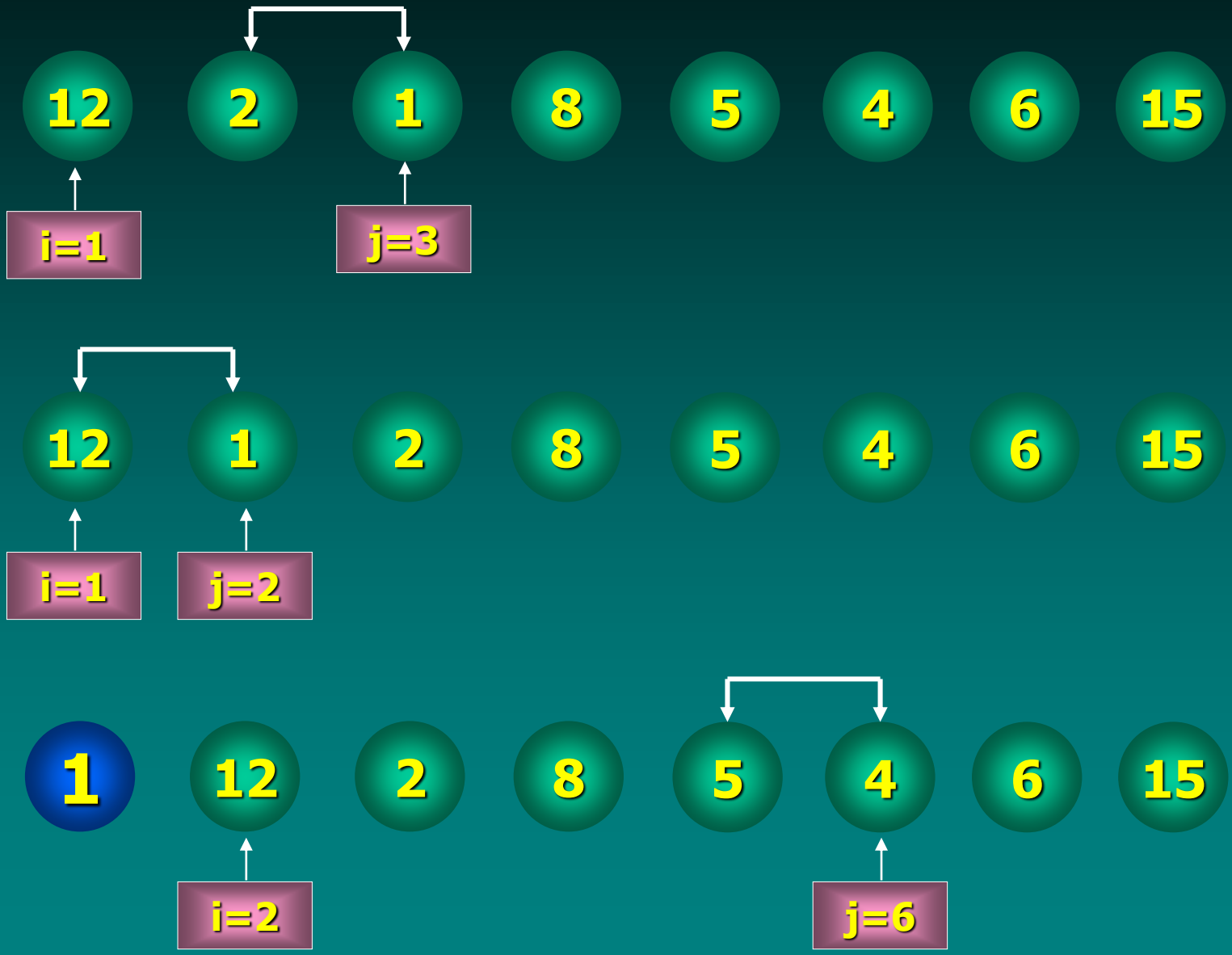
VD Minh họa sắp xếp dãy số sau:

12 2 8 5 1 6 4 15

2.2.2 Bubble Sort (3)



2.2.2 Bubble Sort (4)



2.2.2 Bubble Sort (5)

2.2.2 Bubble Sort (6)

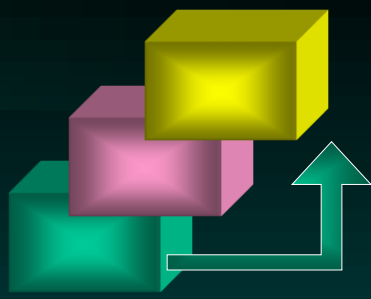
2.2.2 Bubble Sort (7)

2.2.2 Bubble Sort (8)

Cài đặt BubbleSort

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

2.2.3 Insertion Sort



Ý tưởng chính

- Cho dãy ban đầu a[1], a[2],.., a[n], ta có thể xem dãy con gồm một phần tử a[1] đã được sắp.
- Sau đó thêm a[2] vào đoạn a[1] sao cho a[1] a[2] được sắp.
- Tiếp tục thêm a[3] vào để có a[1] a[2] a[3] được sắp....
- Cho đến khi thêm xong a[n] vào đoạn a[1] a[2]...a[n-1] $\Rightarrow$ đoạn a[1] a[2]...a[n-1] a[n] được sắp.

2.2.3 Insertion Sort (2)

Các bước tiến hành

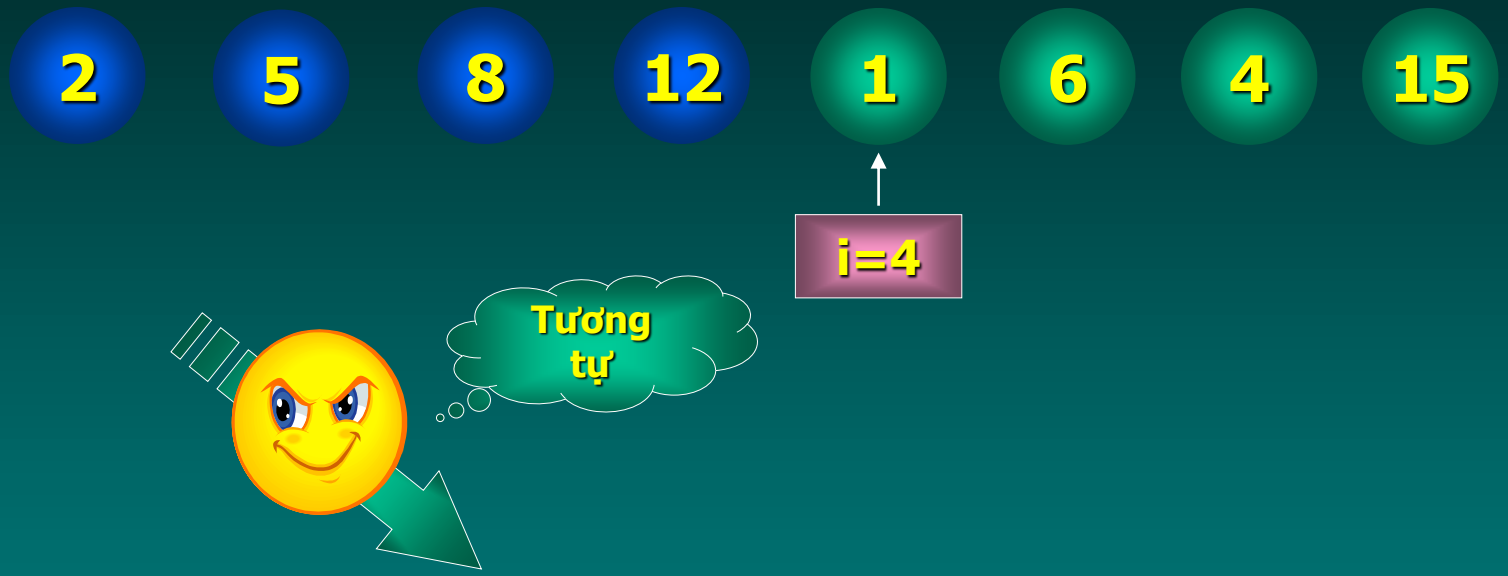
- B1: i = 2; //giả sử có đoạn a[1] đã được sắp
- B2: x= a[i];
  - Tìm được vị trí cần chèn x vào là **pos**
- B3: Dời chỗ các phần tử từ a[pos] ⇒ a[i-1] sang phải một vị trí để dành chỗ cho a[i].
- B4: a[pos] = x; // có đoạn a[1]...a[i] được sắp.
- B5: i = i +1;
  - Nếu i ≤ n: Lặp lại B2
  - Ngược lại: Dừng ⇒ Dãy đã được sắp

- Ví dụ: minh họa phương pháp chèn với dãy:

12 2 8 5 1 6 4 15

2.2.3 Insertion Sort

2.2.3 Insertion Sort



2.2.3 Insertion Sort (6)

Cài đặt InsertionSort

```
void InsertionSort(int a[], int n)
{
    int pos, i, x;            // x lưu phần tử a[i]
    for(i=1; i < n; i++)
    {
        x = a[i];  pos = i-1;
        while ((pos ≥ 0) && (a[pos] > x))
        {// kết hợp dời chỗ các phần tử đứng sau x trong dãy mới
            a[pos+1] = a[pos];
            pos--;
        }
        a[pos+1] = x;  // chèn x vào dãy mới
    }
}
```

2.2.4 Interchange Sort

Ý tưởng

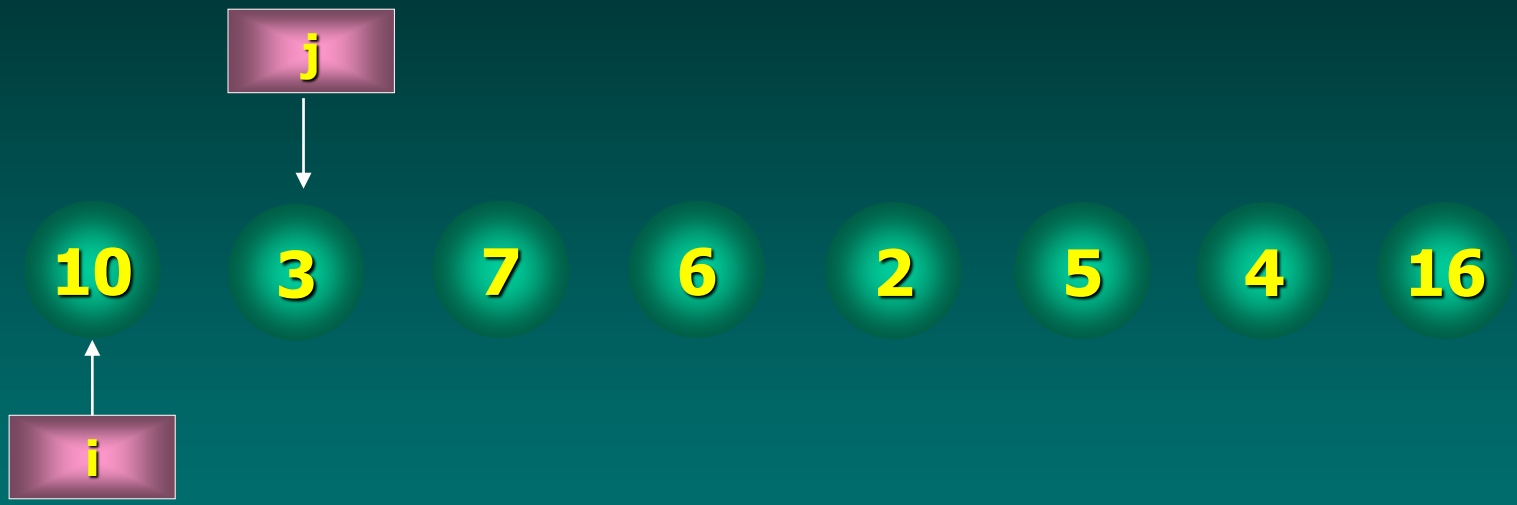
- Xuất phát từ đầu dãy, lần lượt tìm những phần tử còn lại ko thoả thứ tự với phần tử đang xét. Với mỗi phần tử tìm được mà ko thoả thứ tự
  - Thực hiện hoán vị để thoả thứ tự
- Lặp lại tương tự với các phần tử tiếp theo

2.2.4 Interchange Sort

Các bước tiến hành

- B1: i = 1; // đầu dãy
- B2: j = i +1; // duyệt qua các phần tử sau
- B3:
  - while j ≤ n do
    - if a[j] < a[i] then Swap(a[i], a[j]);
    - j = j +1;
- B4: i = i +1;
  - if i < n then B2;
  - else ⇒ Kết thúc!

2.2.4 Interchange Sort



2.2.5 PP ShellSort

Ý tưởng chính

- Cải tiến insertion sort
  - Hạn chế PP chèn: khi luôn chèn 1 phần tử vào đầu dãy!
- ShellSort cải tiến bằng cách chia làm nhiều dãy con và thực hiện pp chèn trên từng dãy con

2.2.5 PP ShellSort

- Xét một dãy a[1]...a[n], cho một số nguyên h ($1 \le h \le n$), chia dãy thành h dãy con như sau:
  - Dãy con 1: a[1], a[1+h], a[1+2h]...
  - Dãy con 2: a[2], a[2+h], a[2+2h]...
  - Dãy con 3: a[3], a[3+h], a[3+2h]...
  - ...
  - Dãy con h: a[h], a[2h], a[3h]...

2.2.5 PP ShellSort

- VD: cho dãy n = 8, h = 3

10 3 7 6 2 5 4 16

Dãy chính	**10**	**3**	**7**	**6**	**2**	**5**	**4**	**16**
Dãy con 1	**10**			**6**			**4**	
Dãy con 2		**3**			**2**			**16**
Dãy con 3			**7**			**5**		

2.2.5 PP ShellSort

- Với mỗi bước h, áp dụng Insertion Sort trên từng dãy con độc lập để làm mịn dần các phần tử trong dãy chính.
- Tiếp tục làm tương tự đối với bước h div 2... cho đến h = 1.
- Khi h =1 thực hiện Insertion Sort trên 1 dãy duy nhất là dãy chính
- Kết quả được dãy phần tử được sắp.

2.2.5 PP ShellSort

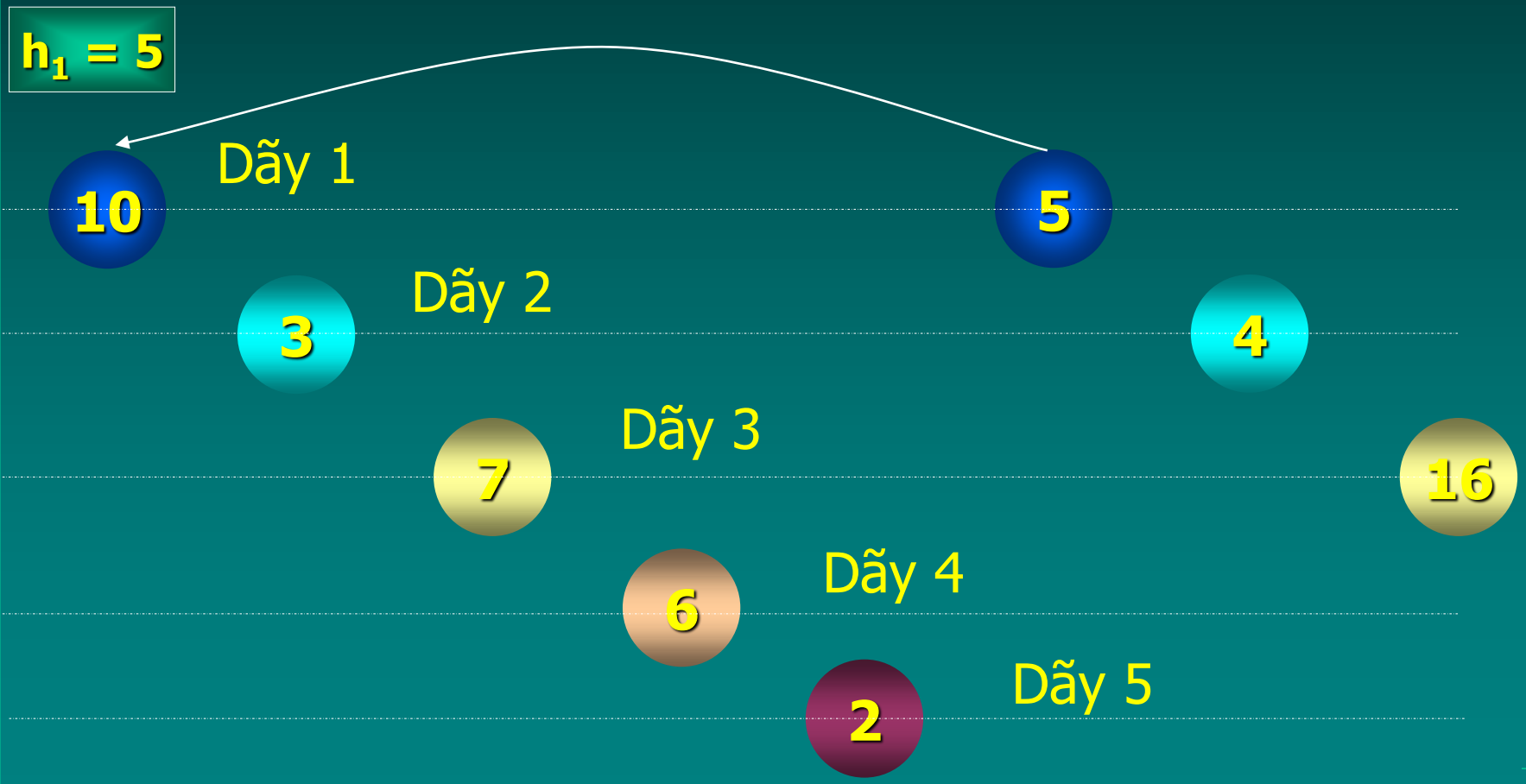
Các bước tiến hành

- B1: chọn k khoảng cách h[1], h[2],..,h[k], và i = 1;
- B2: Chia dãy ban đầu thành các dãy con có bước nhảy là h[i].
  - Thực hiện sắp xếp từng dãy con bằng Insertion sort.
- B3: i = i+1
  - Nếu i > k: $\Rightarrow$ Dừng
  - Ngược lại: $\Rightarrow$ Bước 2.

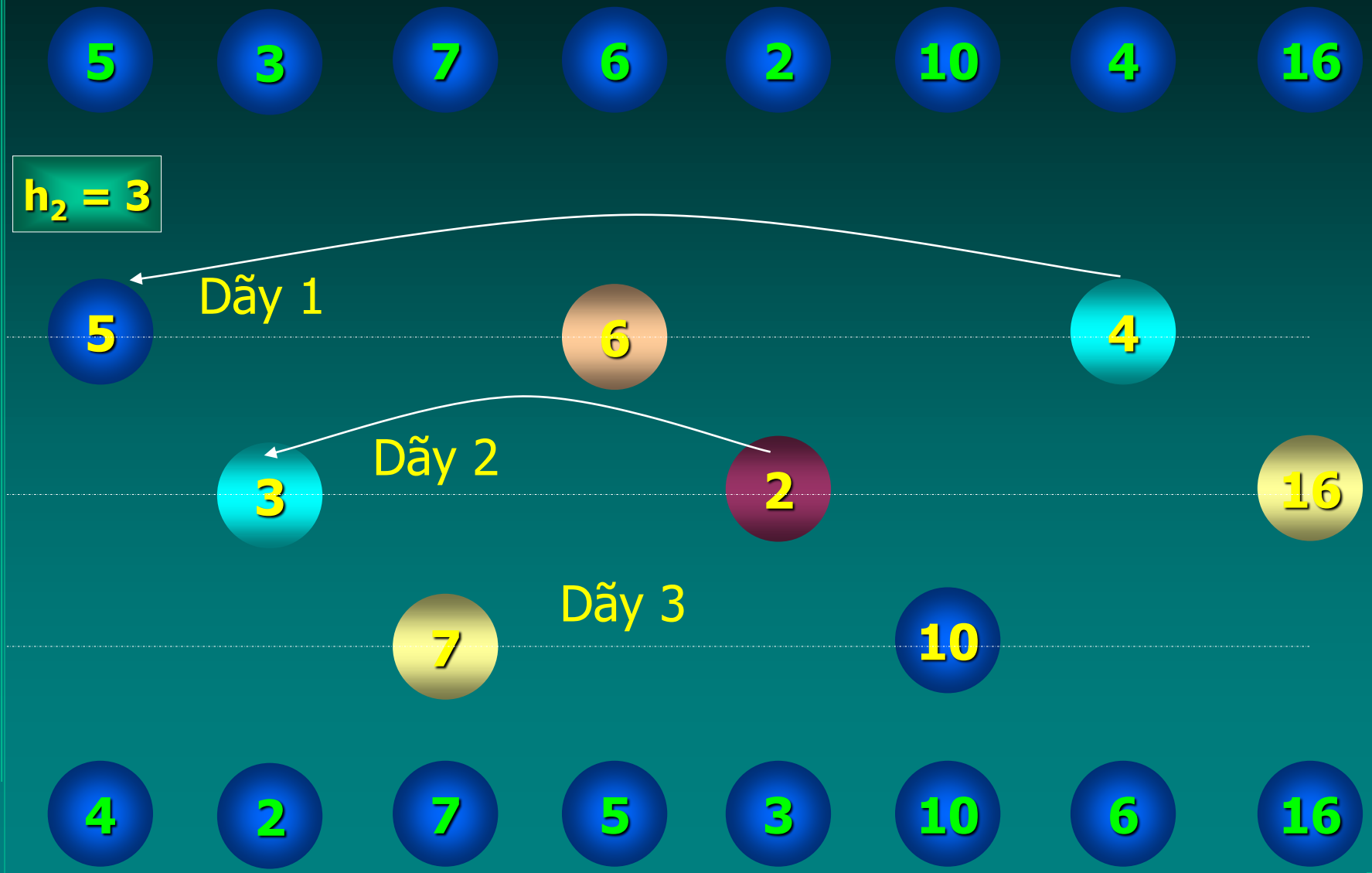
2.2.5 PP ShellSort

- Cho dãy bên dưới với n = 8, h = {5, 3, 1}.

10 3 7 6 2 5 4 16



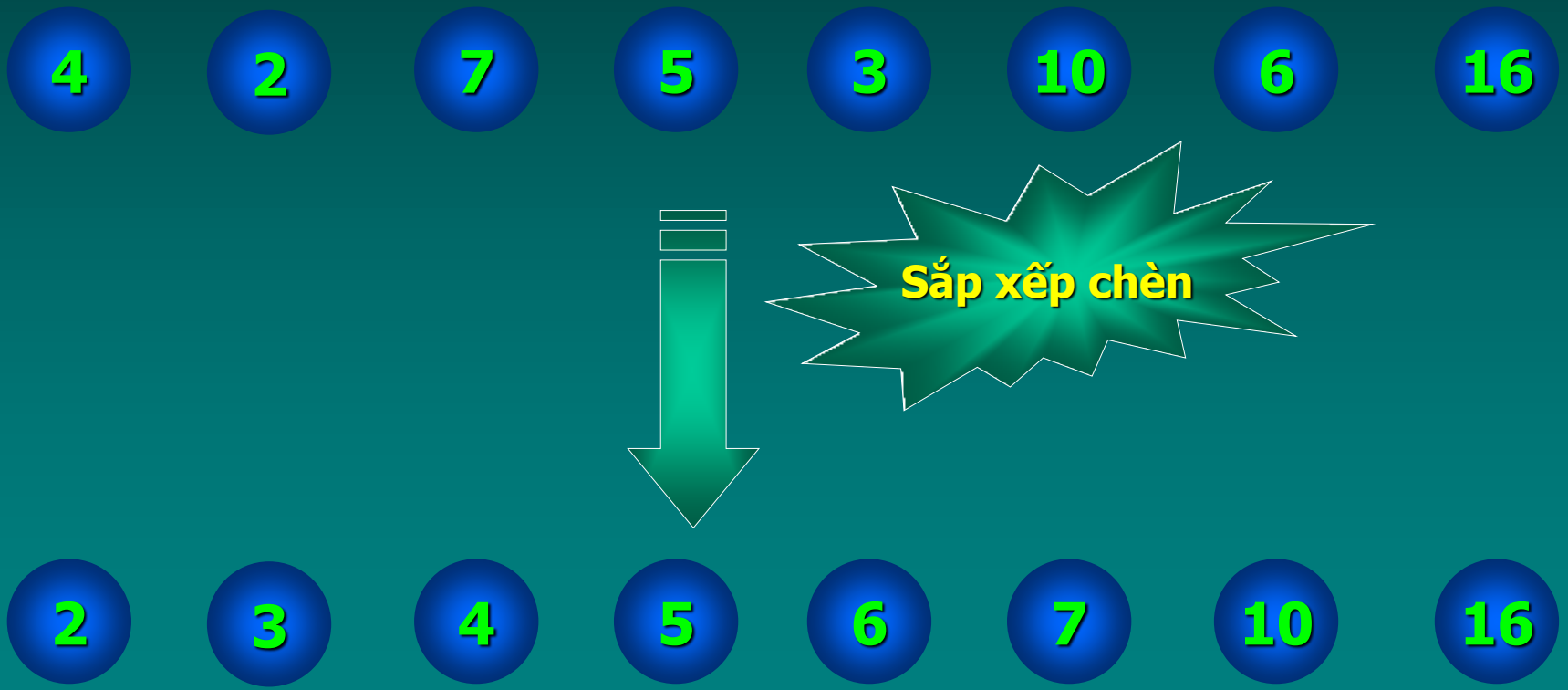
2.2.5 PP ShellSort



2.2.5 PP ShellSort

$h_3 = 1$

Dãy 1



2.2.5 PP ShellSort

- h[] chứa các bước nhảy, số phần tử h là k
- void ShellSort(int a[], int n, int h[], int k)
- {

```
1.          int step, i, j;
2.          int x, len;
3.          for(step = 0; step < k; step++) { // duyệt qua từng bước nhảy
4.              len = h[step];    // chiều dài của bước nhảy
5.              for(i = len; i < n; i++) {   // duyệt các dãy con
6.                  x = a[i];        // lưu phần tử cuối để tìm vị trí thích hợp trong dãy con
7.                  j = i – len;     // a[j] đứng trước a[i] trong cùng dãy con
8.                  while ((x < a[j]) && (j ≥ 0)) { // sắp xếp dãy con chứa x dùng pp chèn
9.                      a[j+len] = a[j]; // dời về sau theo dãy con
10.                     j = j – len;                  // qua phần tử trước trong dãy con
11.                 }
12.                 a[j+len] = x;          // đưa x vào vị trí thích hợp trong dãy con
13.             }// end for i
14.         }// end for step
15. }
```

2.2.6 PP QuickSort

- Thuật toán do Hoare đề xuất
  - Tốc độ trung bình nhanh hơn thuật toán khác
  - Do đó Hoare dùng "*quick*" để đặt tên
- Ý tưởng chính
  - QS phân hoạch dãy ban đầu thành hai phần dựa vào một giá trị x
    - Dãy 1: gồm các phần tử a[i] ko lớn hơn x
    - Dãy 2: gồm các phần tử a[i] ko nhỏ hơn x

2.2.6 PP QuickSort

- Sau khi phân hoạch thì dãy ban đầu được phân thành ba phần:
- a[k] < x, với k = 1...i
- a[k] = x, với k = i..j
- a[k] > x, với k = j..n

a[k] < x	a[k] = x	a[k] > x

2.2.6 PP QuickSort

- GT phân hoạch dãy a[left], a[left+1],...,a[right] thành hai dãy con:
- B1: Chọn tùy ý một phần tử a[k] trong dãy là giá trị mốc, left ≤ k ≤ right,
  - Cho x = a[k], i = left, j = right.
- B2: Tìm và hoán vị cặp phần tử a[i] và a[j] không đúng thứ tự đang sắp.
  - B2-1: Trong khi a[i] < x ⇒ i++;
  - B2-2: Trong khi a[j] > x ⇒ j--;
  - B2-3: Nếu i < j ⇒ Swap(a[i], a[j]) // a[i], a[j] sai thứ tự
- B3:
  - Nếu i < j: ⇒ Bước 2;
  - Nếu i ≥ j: ⇒ Dừng.

2.2.6 PP QuickSort

- GT để sắp xếp dãy a[left], a[left+1],...,a[right]: được phát biểu theo cách đệ quy như sau:
- B1: Phân hoạch dãy a[left]...a[right] thành các dãy con:
  - Dãy con 1: a[left]...a[j] < x
  - Dãy con 2: a[j+1]...a[i-1] = x
  - Dãy con 3: a[i]...a[right] > x
- B2:
  - Nếu (left < j) // dãy con 1 có nhiều hơn 1 phần tử
    - Phân hoạch dãy a[left]...a[j]
  - Nếu (i < right) // dãy con 3 có nhiều hơn 1 phần tử
    - Phân hoạch dãy a[i]...a[right]

2.2.6 PP QuickSort

```
void QuickSort(int a[], int left, int right) {
1.      int     i, j, x;
2.      x = a[(left+right)/2];              // chọn phần tử giữa làm gốc
3.      i = left;   j = right;
4.      do {
5.              while (a[i] < x) i++;       // lặp đến khi a[i] >= x
6.              while (a[j] > x) j--;       // lặp đến khi a[i] <= x
7.              if ( i <= j) {
8.                      Swap(a[i], a[j]);
9.                      i++;                // qua phần tử kế tiếp
10.                     j--;                // qua phần tử đứng trước
11.             }
12.     } while (i<j);
13.    if (left < j)                        // ph đoạn bên trái
14.         QuickSort(a, left, j);
15.    if (right > i)                       // ph đoạn bên phải
16.         QuickSort(a, i, right);
}
```

2.2.7 PP RadixSort

- Không quan tâm đến việc so sánh giá trị các phần tử
- Sử dụng cách thức phân loại các con số và thứ tự phân loại các con số này để tạo ra thứ tự
- Còn gọi là phương pháp phân lô

2.2.7 PP RadixSort

493 812 715 710 195 437 582 340 385

Số hàng đv	Dãy con
0	**710 340**
1	
2	**812 582**
3	**493**
4	
5	**715 195 385**
6	
7	**437**
8	
9	

Sau khi phân lô theo hàng đơn vị

710 340 812 582 493 715 195 385 437

2.2.7 PP RadixSort

710 340 812 582 493 715 195 385 437

Số hàng chục	Dãy con
0	
1	**710 812 715**
2	
3	**437**
4	**340**
5	
6	
7	
8	**582 385**
9	**493 195**

Sau khi phân lô theo hàng chục

710 812 715 437 340 582 385 493 195

2.2.7 PP RadixSort

710 812 715 437 340 582 385 493 195

Số hàng trăm	Dãy con
0	
1	**195**
2	
3	**340 385**
4	**437 493**
5	**582**
6	
7	**710 715**
8	**812**
9	

Sau khi phân lô theo hàng trăm

195 340 385 437 493 582 710 715 812

2.2.7 PP RadixSort

- GT RadixSort thực hiện như sau:
- Xem mỗi phần tử a[i] trong dãy a[1]...a[n] là một số nguyên có tối đa m chữ số
- Lần lượt phân loại các chữ số theo hàng đơn vị, hàng chục, hàng trăm...
  - Tại mỗi bước phân loại ta sẽ nối các dãy con từ danh sách đã phân loại theo thứ tự $0 \rightarrow 9$.
  - Sau khi phân loại xong ở hàng thứ m cao nhất ta sẽ thu được danh sách các phần tử được sắp.

2.2.7 PP RadixSort

- B1: k = 0; // k thể hiện chữ số phân loại, k =0 hàng đơn vị, k=1 hàng chục...
- B2: // Tạo các dãy chứa phần tử phân loại B[0]...B[9]
- Khởi tạo B[0]...B[9] rỗng, B[i] sẽ chứa các phần tử có chữ số thứ k là i.
- B3:
  - For i=1 to n do
    - Đặt a[i] vào **dãy B[j]** với j là chữ số thứ k của a[i].
  - Nối B[0], B[1],..., B[9] lại theo đúng trình tự thành a.
- B4:
  - k = k +1
  - Nếu k < m: ⇒ Bước 2. // m là số lượng chữ số tối đa của các số
  - Ngược lại: ⇒ Dừng.

2.2.7 PP RadixSort

```
void RadixSort(long a[], int n){
1.      int i, j, d, digit, num;
2.      int h = 10;                         // biến để lấy các con số, bắt đầu từ hàng đơn vị
3.      long B[10][MAX];  // mảng hai chiều chứa các phần tử phân lô
4.      int  Len[10];                       // kích thước của từng mảng B[i]
5.      for(d = 0; d < MAXDIGIT; d++) {
6.          for( i = 0; i < 10; i++) // khởi tạo kích thước các dãy B[i] là 0
7.              Len[i] = 0;
8.          for(i = 0; i < n; i++) { // duyệt qua tất cả các phần tử của mảng
9.              digit = (a[i] % h) / (h / 10);     // lấy con số theo hàng h
10.             B[digit][Len[digit]++] = a[i];
11.         }// end for i
12.         num = 0;  // chỉ số bắt đầu cho mảng a[]
13.         for(i = 0; i < 10; i++) // duyệt qua các dãy từ B[0] – đến B[9]
14.             for(j =0; j < Len[i]; j++)
15.                 a[num++] = B[i][j];
16.         h *= 10;   // qua hàng kế tiếp.
        }// end for d
}// end RadixSort
```

Tài liệu tham khảo

[1]. Cấu trúc dữ liệu & thuật toán, Dương Anh Đức, Trần Hạnh Nhi, ĐHKHTN, 2000.

[2]. Kỹ thuật lập trình, Học viện BCVT, 2002.

[3]. Cấu trúc dữ liệu, Nguyễn Trung Trực, ĐHBK, 1992.

[4]. Giải thuật & lập trình, Lê Minh Hoàng, ĐHSPHN, 1999-2002.