Cấu trúc dữ liệu và giải thuật

Người thực hiện: Đỗ Tuấn Anh
Email: anhdt@it-hut.edu.vn
ĐT: 0989095167

Tài liệu tham khảo

- Sách giáo trình: Đỗ Xuân Lôi, *Cấu trúc dữ liệu và Giải Thuật,* NXB ĐHQGHN
- R. Sedgewick, *Algorithm in C*, Addison Wesley

Nội dung

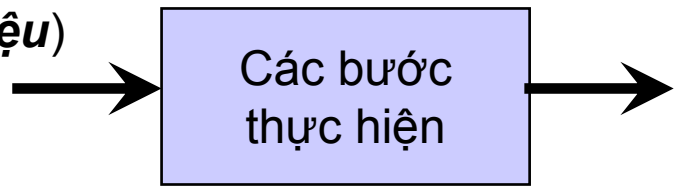
- Chương 1 – Thiết kế và phân tích
- Chương 2 – Giải thuật đệ quy
- Chương 3 – Mảng và danh sách
- Chương 4 – Ngăn xếp và hàng đợi
- Chương 5 – Cấu trúc cây
- Chương 6 – Đồ thị
- Chương 7 – Sắp xếp
- Chương 8 – Tìm kiếm

Chương 1 – Thiết kế và phân tích

1. Mở đầu
2. Từ bài toán đến chương trình
   2.1 Modul hóa bài toán
   2.2 Phương pháp tinh chỉnh từng bước
3. Phân tích giải thuật
   3.1 Độ phức tạp về thời gian thực hiện GT
   3.2 O-lớn, Omega-lớn, Theta-lớn
   3.3 Xác định độ phức tạp về thời gian

1. Mở đầu

- Giải thuật:
  - Các bước giải quyết bài toán
  - Một dãy câu lệnh 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.
    - Đầu vào(Input): tập các đối tượng (***dữ liệu***)
    - Đầu ra(Output): một tập các giá trị



- Cấu trúc dữ liệu:
  - Tập hợp dữ liệu
  - Có mối quan hệ với nhau trong bài toán xác định
- Lựa chọn cấu trúc dữ liệu và giải thuật thích hợp: rất quan trọng
  - Ví dụ: viết chương trình tìm kiếm số điện thoại theo tên đơn vị

Chương trình = Giải thuật + Dữ liệu

1. Mở đầu (tiếp)

- Biểu diễn cấu trúc dữ liệu trong bộ nhớ:
  - Lưu trữ trong
  - Lưu trữ ngoài
- Diễn đạt giải thuật:
  - Ngôn ngữ tự nhiên
  - Giả ngôn ngữ
  - Lưu đồ
  - Ngôn ngữ lập trình
- Cài đặt giải thuật: ngôn ngữ C/C++

Giả ngôn ngữ

1. Chú thích: /*…*/ hoặc //…
2. Đánh số thứ tự các đoạn của chương trình
3. Ký tự và biểu thức
   - 26 chữ cái Latin + 10 chữ số
   - Phép toán số học: +, -, *, /, ^(lũy thừa), %
   - Phép toán quan hệ: <, >, ==, <=, >=, !=
   - Phép toán logic: &&, ||, !
   - Giá trị logic: true, false
   - Biến chỉ số: a[i], a[i][j]
   - Thứ tự ưu tiên của các phép toán: như C và các ngôn ngữ chuẩn khác

Giả ngôn ngữ (tiếp)

- Lệnh gán: a = b; c = d = 2;
- Khối lệnh: { S1; S2; S3; }
- Lệnh điều kiện:

```
if (B)
   S;
```

```
if (B)
{s1;s2;s3;}
```

```
if (B)
   S1;
else
   S2;
```

Giả ngôn ngữ

- Lệnh lặp

```
for (i = 0 ; i<n; i++)
   S;
for ( i = n; i>= 0; i--)
   S;

do S  while (B);

while (B) do S;
```

Giả ngôn ngữ (tiếp)

- Lệnh vào/ra:

  read (<danh sách biến>)

  write (<danh sách biến hoặc dòng ký tự>)

- Chương trình con:

  function <tên hàm> (<danh sách tham số>)

  {

  S1; S2; …Sn;

  return; // nếu chương trình con trả lại một giá trị

  }

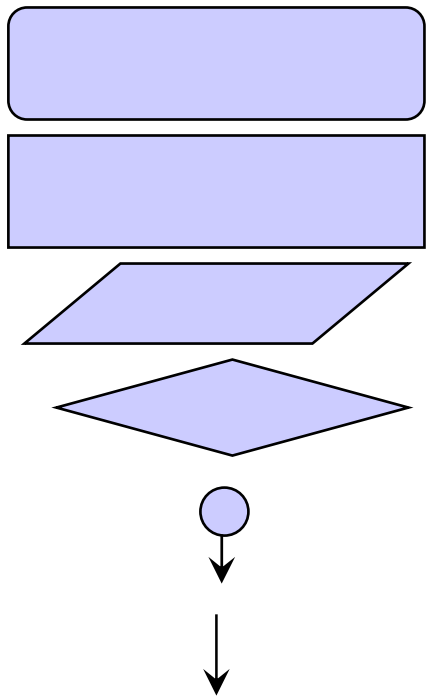
- Gọi chương trình con:

  <tên hàm> (<danh sách tham số thực sự>)

Sơ đồ

Lệnh điều khiển có thể là:

- Khối lệnh
- Lệnh điều kiện
- Lệnh lặp



Bắt đầu hoặc ***kết thúc***

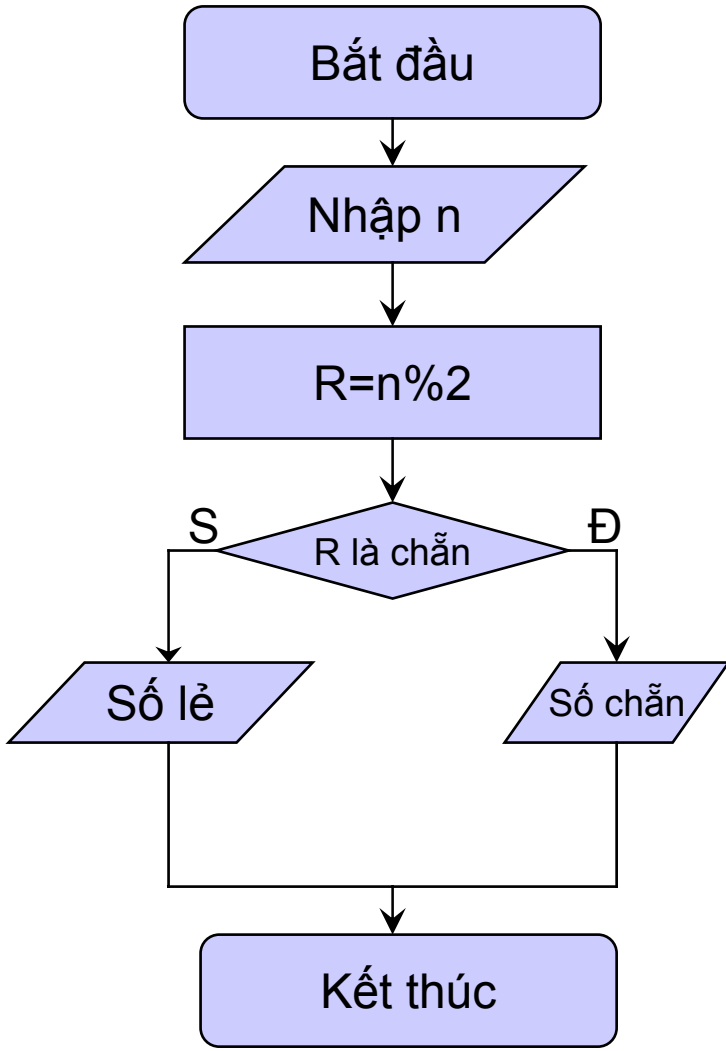
Lệnh gán

Lệnh vào, lệnh ra

Điều kiện

Nối tiếp đoạn lệnh

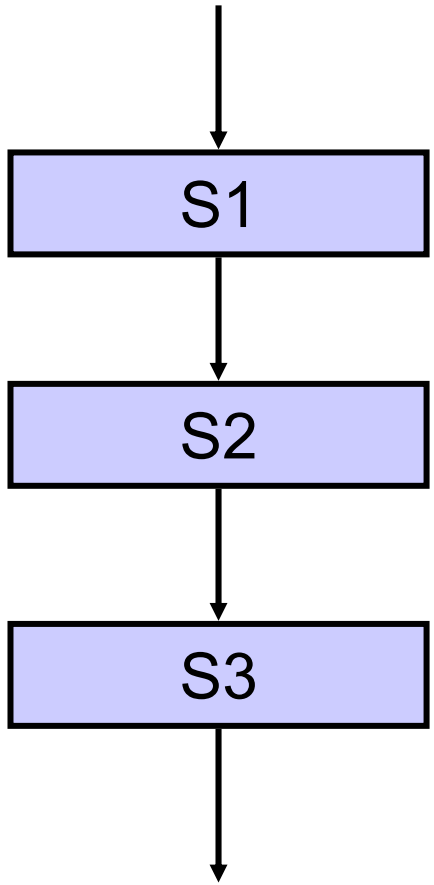
Luồng thực hiện



Khối lệnh

Cú pháp:

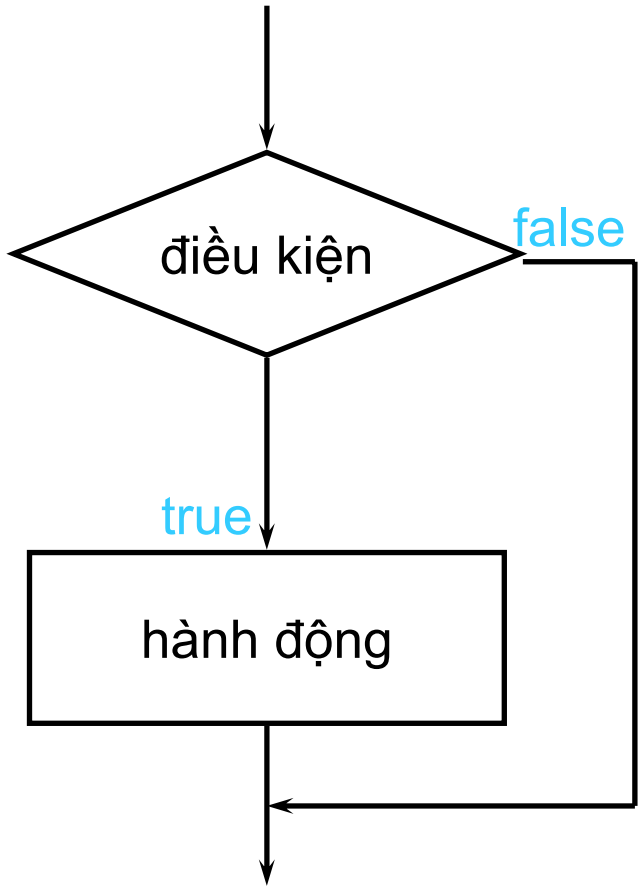
```
{
  S1;
  S2;
  S3;
}
```



Lệnh điều kiện

- Cú pháp

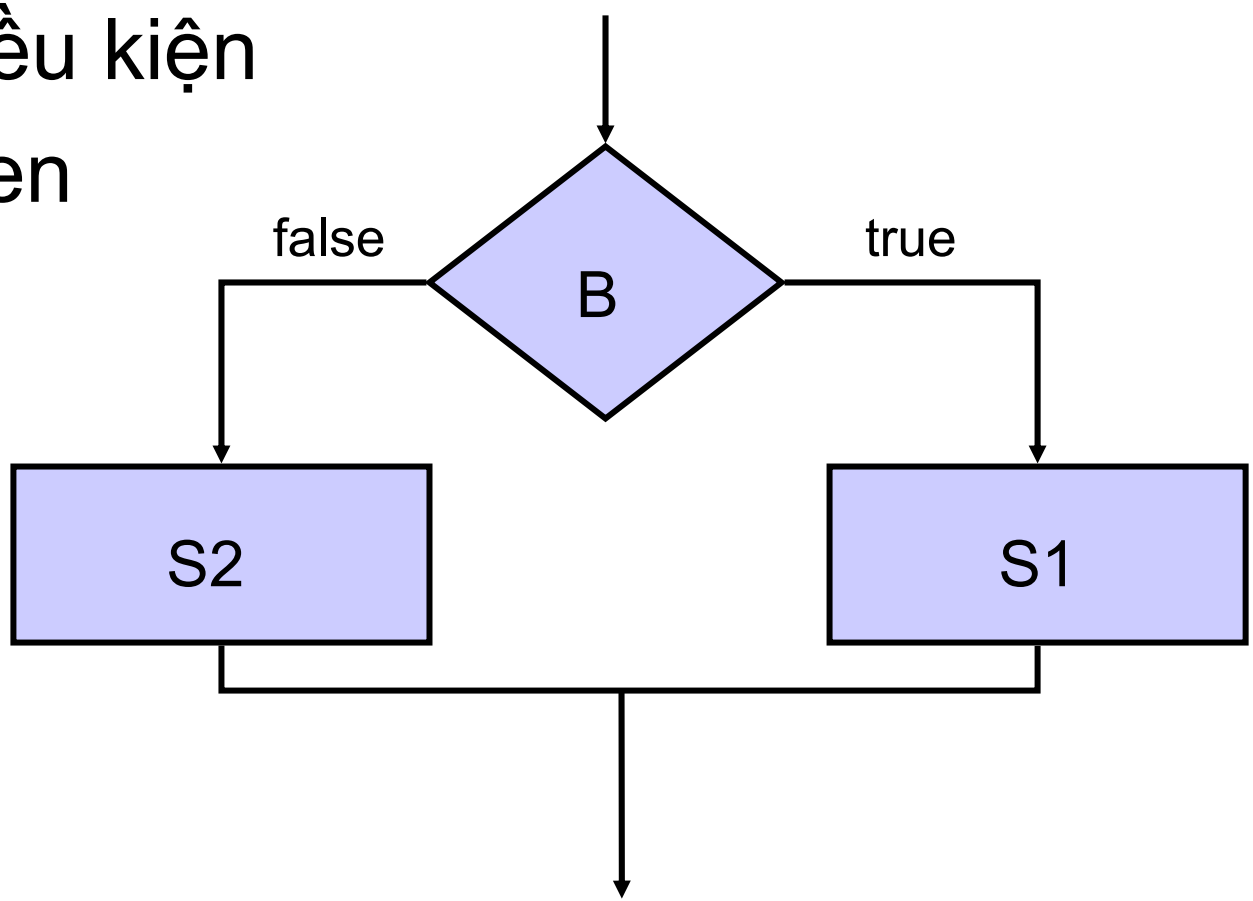
```
if(điều_kiện)
    hành_động
```



Lệnh điều kiện

- Lệnh điều kiện

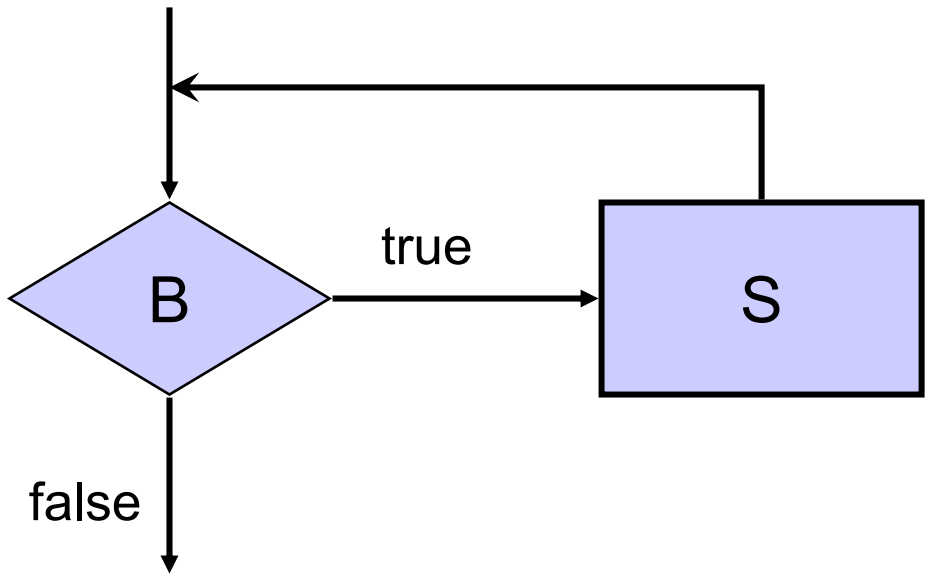
```
if (B) then
  S1;
else
  S2;
```



Lệnh lặp:

- Cú pháp:

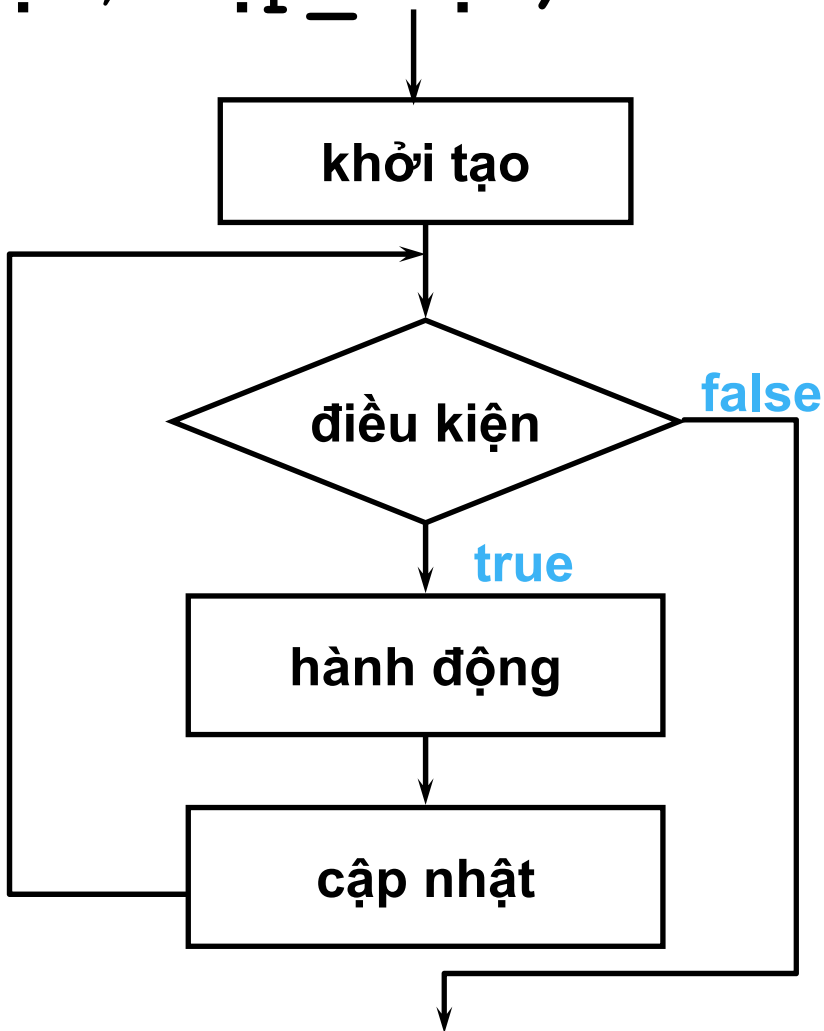
```
while (B) do
    S;
```



Lệnh lặp for

- Cú pháp

```
for (khởi_tạo; điều_kiện; cập_nhật)
    hành_động
```

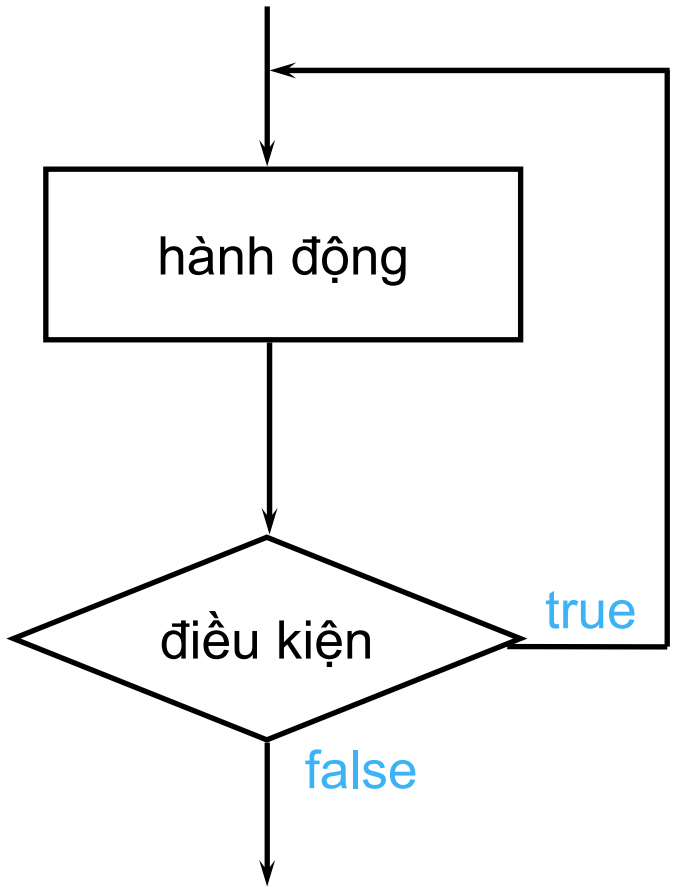


Lệnh lặp do-while

- Cú pháp

  `do` hành_động

  `while` (điều_kiện)



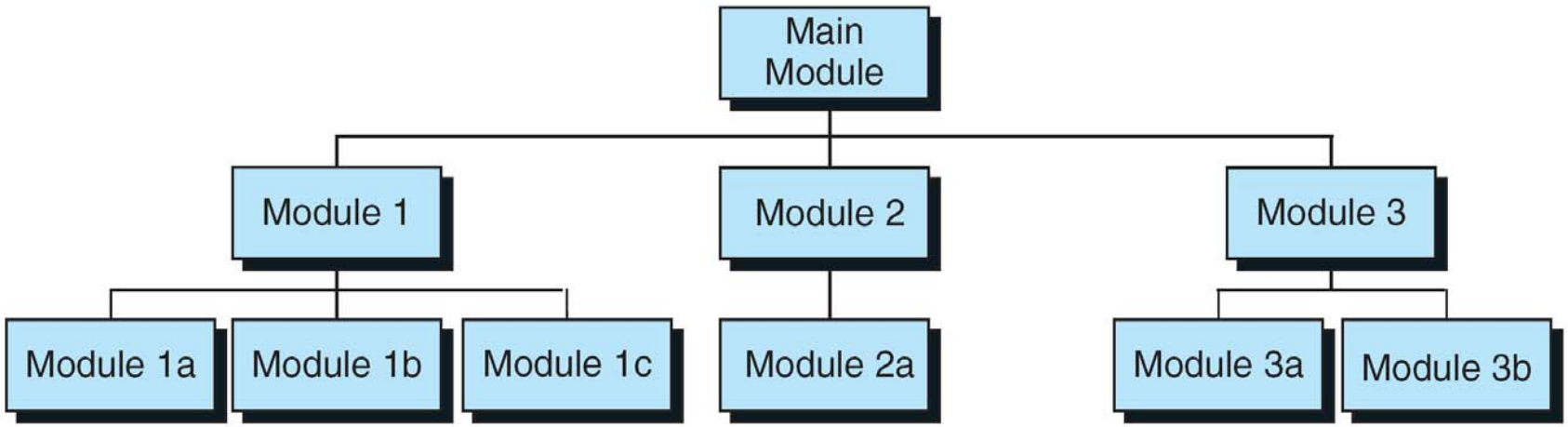
2. Từ bài toán đến chương trình

Mô đun hóa và việc giải quyết bài toán

- Chia bài toán lớn (module chính) thành các bài toán (module) nhỏ hơn
- Mỗi module thực hiện công việc cụ thể nào đó
- Lặp đi lặp lại cho đến khi các module là cô đọng và biết cách giải quyết.

=> chiến thuật “Chia để trị”

2.1 Module hóa bài toán



Module hóa bài toán

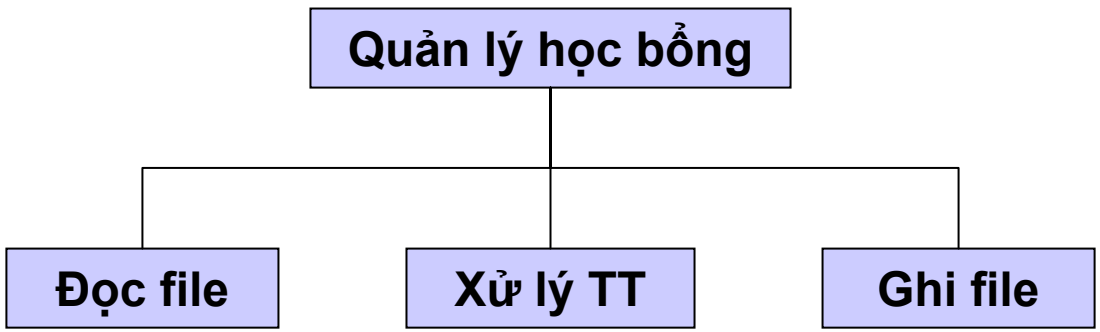
- Thiết kế Topdown – từ đỉnh xuống, hay từ khái quát đến chi tiết.
  - *Bước 1*: Xác định dữ kiện đầu vào, yêu cầu đặt ra
  - *Bước 2*: Xác định các công việc chủ yếu (mỗi công việc tương đương với 1 module)
  - *Bước 3*: Giải quyết từng công việc một cách chi tiết bằng cách lặp đi lặp lại bước 1 + 2
- Ví dụ Bài toán: “Quản lý và bảo trì các hồ sơ về học bổng của sinh viên, thường kỳ lập báo cáo tổng kết”.

Thiết kế Topdown – Bước 1

- Bước 1: Xác định dữ kiện đầu vào và các yêu cầu đặt ra
  - Đầu vào: Tập các file bao gồm các thông tin về học bổng của sinh viên: Mã SV, ĐiểmTB, Mức HB
  - Yêu cầu:
    - Tìm kiếm và hiển thị thông tin của bất kỳ sinh viên nào
    - Cập nhật thông tin của một sinh viên cho trước
    - In bản tổng kết

Thiết kế Topdown – Bước 2

- Bước 2: Xác định các công việc chủ yếu
  1. Đọc các thông tin của sinh viên từ file vào bộ nhớ trong (Đọc file)
  2. Xử lý các thông tin (Xử lý thông tin)
  3. Lưu thông tin đã cập nhật vào file (Ghi file)



Thiết kế Topdown – Bước 3

- Bước 3: Lặp lại bước 1 + 2
  - Đọc file:
    - Đầu vào: File thông tin trên đĩa
    - Yêu cầu: Đọc file và lưu vào mảng: mỗi phần tử mảng lưu thông tin của một sinh viên

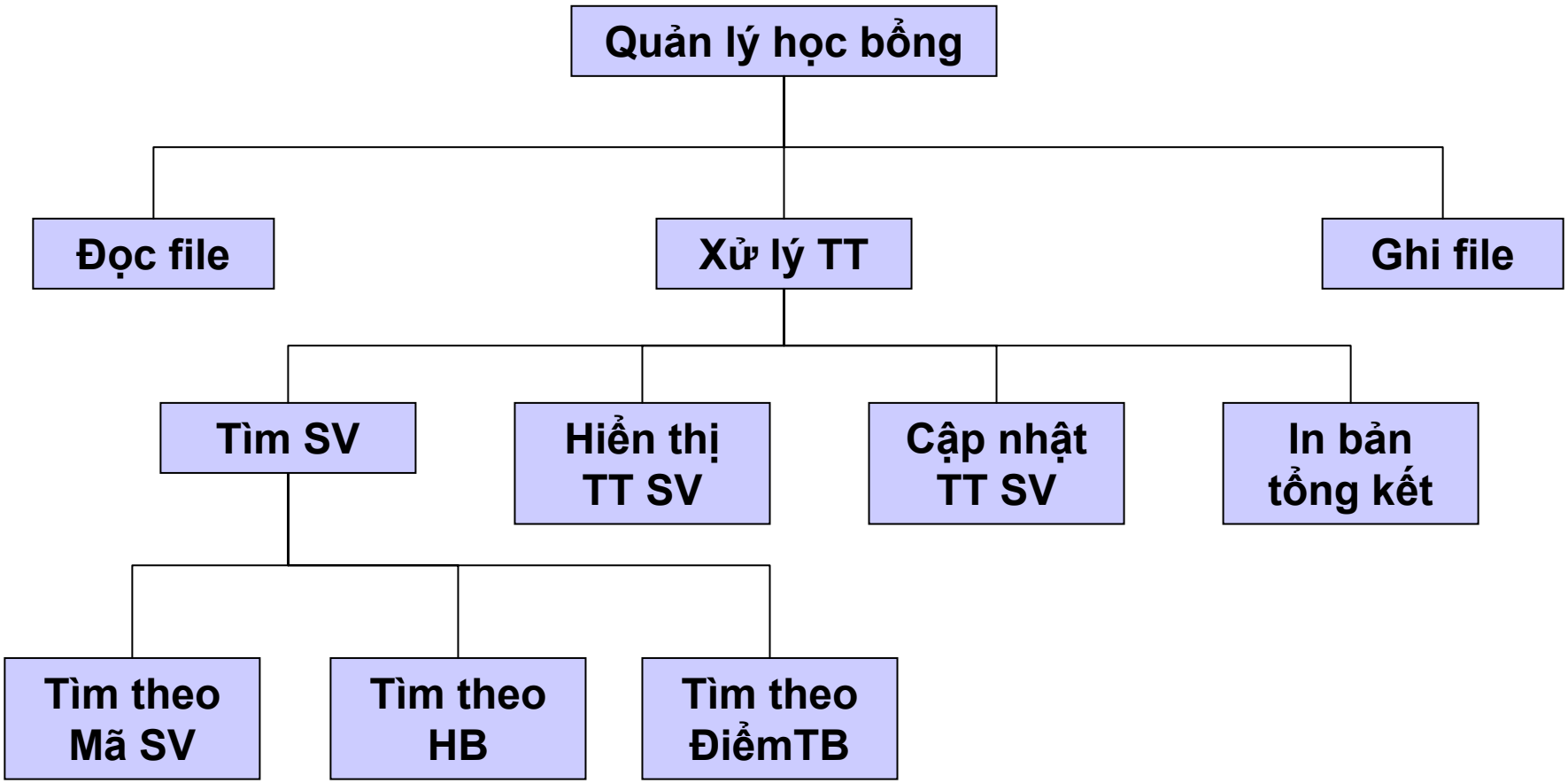
    ⇒ Đã cô đọng
  - Ghi file:
    - Đầu vào: Mảng lưu thông tin của các sinh viên
    - Yêu cầu: Lưu trở lại file

    ⇒Đã cô đọng

Thiết kế Topdown – Bước 3

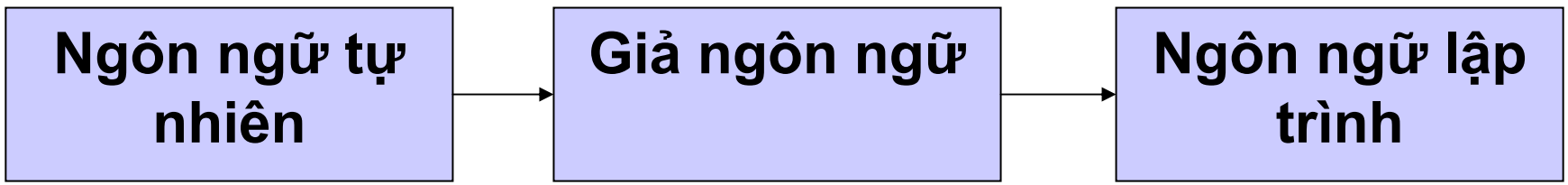
- Xử lý TT
  - Đầu vào: Mảng lưu thông tin của các sinh viên
  - Yêu cầu:
    - Tìm một sinh viên cho trước
    - Hiển thị thông tin của sinh viên
    - Cập nhật thông tin của sinh viên
    - In bản tổng kết

Thiết kế Topdown



2.2 Phương pháp tinh chỉnh từng bước (Stepwise Refinement)

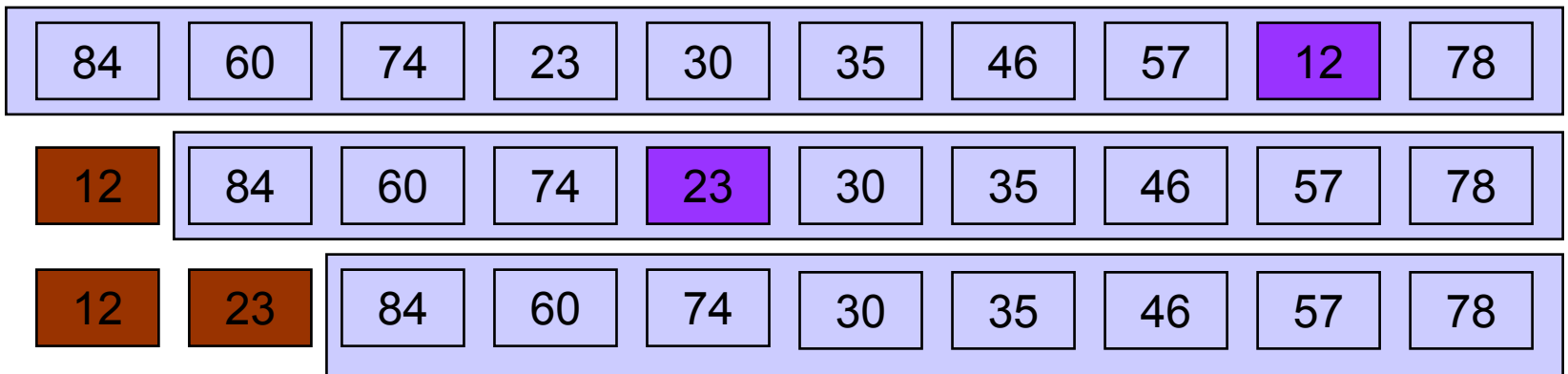
- Ban đầu giải thuật được trình bày ở dạng ngôn ngữ tự nhiên
- Chi tiết hóa dần – tinh chỉnh hướng về phía ngôn ngữ lập trình
- Giai đoạn trung gian – giả ngôn ngữ



Tinh chỉnh từng bước – Ví dụ

- Bài toán: “Sắp xếp một dãy n số nguyên theo thứ tự tăng dần”
- Giải thuật:
  - Từ dãy số nguyên chưa được sắp xếp chọn ra số nhỏ nhất và đặt vào đầu dãy đã được sắp xếp
  - Loại số nguyên đó ra khỏi dãy chưa được sắp xếp
  - Lặp lại cho đến khi dãy chưa được sắp xếp là rỗng

Ngôn ngữ tự nhiên



Tinh chỉnh từng bước – Ví dụ

- Cấu trúc dữ liệu:
  - Dãy số ban đầu được lưu trữ trong một mảng một chiều
  - Dãy đã sắp xếp sẽ được lưu trùng với dãy chưa sắp xếp

  => Giải thuật: Đặt số nhỏ nhất của lượt thứ i vào dãy đã sắp xếp = đổi chỗ với số thứ i trong dãy

Tinh chỉnh từng bước – Ví dụ

- Tinh chỉnh bước 1

for(i=0; i<n; i++)

{

1. Xét từ a_i đến a_{n-1} để tìm số nhỏ nhất a_j
2. Đổi chỗ a_i và a_j

}

Giả ngôn ngữ

Tinh chỉnh từng bước – Ví dụ

- Giải thuật 1: Xét từ a_i đến a_{n-1} để tìm số nhỏ nhất a_j
  - Coi a_i là "số nhỏ nhất" (j = i)
  - So sánh "số nhỏ nhất" và a_{i+1}, số nào nhỏ hơn thì coi là "số nhỏ nhất" (nếu $a_{i+1} < a_j$ thì j = i+1)
  - Tiếp tục so sánh "số nhỏ nhất" với a_{i+2}, a_{i+3}, … a_{n-1}, a_n
  - Xác định "số nhỏ nhất" bằng cách nắm được chỉ số của nó
- Tinh chỉnh bước 1

```
j = i;
for (k = i+1; k<n; k++)
   if(a_k < a_j) j = k;
```

Tinh chỉnh từng bước – Ví dụ

- Giải thuật 2: Đổi chỗ a_i và a_j
  - Sử dụng một biến trung chuyển
- Tinh chỉnh bước 2.2

  tmp = a_i;

  a_i = a_j;

  a_j = tmp;

Tinh chỉnh từng bước

```
function SapXep(a, n)
/* a là mảng các số nguyên
   n là số phần tử mảng  */
{
    for(i = 0; i<n; i++)
    {
        /* 1. Tìm số nhỏ nhất */
        j = i;
        for (k = i+1; k<n; k++)
            if(a_k < a_j) j = k+1;
        /* 2. Đổi chỗ */
         tmp = a_i; a_i = a_j; a_j = tmp;
    }
}
```

3. Phân tích giải thuật

- Tại sao cần phân tích giải thuật ?
  - Viết một chương trình chạy thông là chưa đủ
  - Chương trình có thể thực hiện chưa hiệu quả!
  - Nếu chương trình chạy trên một tập dữ liệu lớn, thì thời gian chạy sẽ là một vấn đề cần lưu ý

Ví dụ: Bài toán lựa chọn

- Cho một dãy gồm N số, hãy tìm phần tử lớn thứ k, với $k \leq N$.
- Thuật toán 1:
  (1) Đọc N số vào một mảng
  (2) Sắp xếp mảng theo thứ tự giảm dần
  (3) Trả lại phần tử ở vị trí thứ k

Ví dụ: Bài toán lựa chọn…

- Thuật toán 2:
  (1) Đọc k phần tử đầu tiên vào mảng và sắp xếp chúng theo thứ tự giảm dần
  (2) Mỗi phần tử còn lại chỉ đọc một lần
    - Nếu phần tử đó là nhỏ hơn phần tử thứ k, bỏ qua
    - Ngược lại, đặt nó vào vị trí phù hợp của mảng, đẩy phần tử hiện tại ra khỏi mảng.
  (3) Phần tử tại vị trí thứ k là phần tử cần tìm.

84	60	74	23	30	35	46	57	12	78

Ví dụ: Bài toán lựa chọn…

- Thuật toán nào là tốt hơn khi
  - N =100 và k = 100?
  - N =100 và k = 1?
- Điều gì sẽ xảy ra khi N = 1,000,000 và k = 500,000?
- Còn có những thuật toán tốt hơn

Phân tích thuật toán

- Chúng ta chỉ phân tích những thuật toán *đúng*
- Một thuật toán là đúng?
  - Nếu, với một dữ liệu đầu vào, thuật toán dừng và đưa ra kết quả đúng
- Thuật toán không đúng
  - Có thể không dừng với một số dữ liệu đầu vào
  - Dừng nhưng đưa ra kết quả sai
- Phân tích thuật toán
  - Dự đoán lượng tài nguyên mà thuật toán yêu cầu
  - Tài nguyên gồm
    - Bộ nhớ
    - Băng thông giao tiếp
    - Thời gian tính – Thời gian thực hiện GT (thường là quan trọng nhất)

Thời gian thực hiện giải thuật

- Các nhân tố ảnh hưởng đến thời gian tính
  - Máy tính
  - Chương trình dịch
  - Thuật toán được sử dụng
  - Dữ liệu đầu vào của thuật toán
    - Giá trị của dữ liệu ảnh hưởng đến thời gian tính
    - Thông thường, *kích thước của dữ liệu đầu vào* là nhân tố chính quyết định thời gian tính
      - VD với bài toán sắp xếp $\Rightarrow$ số phần tử sắp xếp
      - VD bài toán nhân ma trận $\Rightarrow$ tổng số phần tử của 2 ma trận

Độ phức tạp về thời gian

Thuật toán ***A*** mất 2 phút để chạy với dữ liệu đầu vào X.
Thuật toán ***B*** mất 1 phút 45 giây để chạy với cùng dữ liệu X.

Liệu ***B*** có phải là thuật toán "tốt hơn" *A*? **Không hẳn là như vậy**

- Chỉ kiểm tra với một bộ dữ liệu X.

  Có thể với dữ liệu X này ***B*** chạy nhanh hơn ***A***,
  nhưng với phần lớn các dữ liệu khác ***B*** chạy chậm hơn ***A***.
- Thuật toán ***A*** bị ngắt bởi các tiến trình khác.
- Thuật toán ***B*** được chạy trên máy tính có cấu hình cao hơn.

...

Phép đo cần phải không phụ thuộc vào máy.

Đo bằng cách đếm ***số các phép tính cơ sở***
(như phép gán, so sánh, các phép tính số học, vv.)

Ví dụ

Bài toán Tính tổng các số nguyên từ 1 đến n.

Thuật toán 1

```
int sum = 0;
for (int i = 1; i <= n; i++)
    sum = sum + i;
```

Thuật toán 2

```
int sum = ((n+1)*n) / 2;
```

Trường hợp tồi nhất / trung bình / tốt nhất

- *Thêi gian tÝnh tèt nhÊt:* Thêi gian *tèi thiÓu* cÇn thiÕt ®Ó thùc hiÖn thuËt to¸n víi mäi bé dữ liÖu ®Çu vµo kÝch th-íc *n*.
- *Thêi gian tÝnh tåi nhÊt:* Thêi gian nhiÒu nhÊt cÇn thiÕt ®Ó thùc hiÖn thuËt to¸n víi mäi bé d÷ liÖu ®Çu vµo kÝch th-íc n.
- *Thêi gian trung bình:* cÇn thiÕt ®Ó thùc hiÖn thuËt to¸n trªn tËp hữu h¹n c¸c ®Çu vµo kÝch th-íc *n*.

Thời gian tính phụ thuộc vào kích thước dữ liệu đầu vào

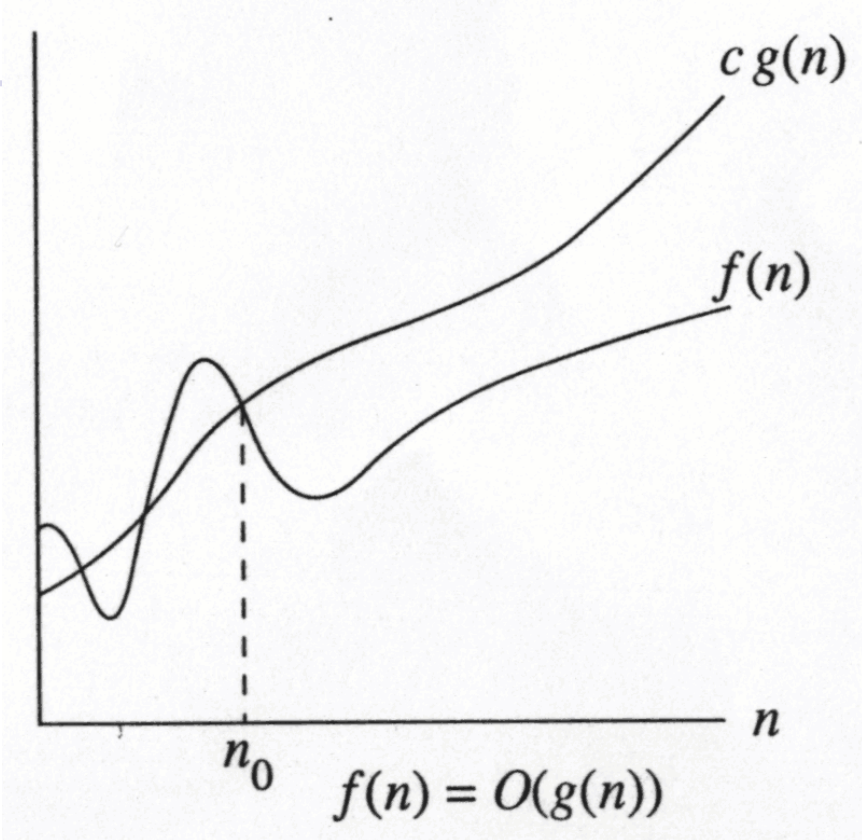
Điều quan trọng đối với giải thuật là

~~mất bao nhiêu giây để chạy với dữ liệu đầu vào có kích thước *n*.~~

tốc độ thay đổi của thời gian tính khi *n* tăng.

- *Thuật toán có thời gian hằng số* nếu thời gian chạy của nó là không đổi khi kích thước dữ liệu thay đổi.
- *Thuật toán có thời gian tuyến tính* nếu thời gian chạy của nó tỷ lệ thuận với *n*.
- *Thuật toán có thời gian tính hàm số mũ* nếu thời gian chạy tăng theo một hàm số mũ của *n*.

Tỉ suất tăng trưởng



- Thiết lập một thứ tự tương đối cho các hàm với đầu vào n lớn
- $\exists$ c , $n_0 > 0$ sao cho $f(n) \leq c\, g(n)$ khi $n \geq n_0$
- f(n) tăng không nhanh bằng g(n) khi n "lớn"

Khái niệm O-lớn

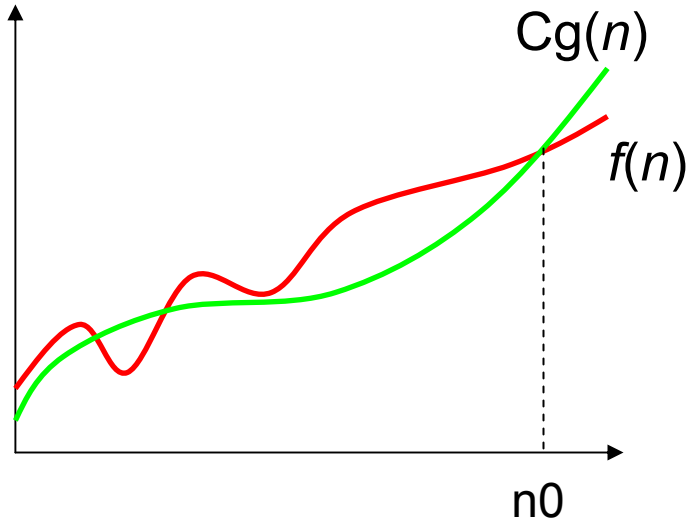
Một thuật toán là $O(f(n))=g(n)$ nếu

tồn tại một hằng số $C > 0$ và số nguyên n_0 sao cho thuật toán yêu cầu không vượt quá $C \bullet g(n)$ phép tính có tất cả các dữ liệu đầu vào có kích thước $n \geq n_0$.

Thuật toán 1 cần $4n + 3$ phép tính.

```
int sum = 0;
for (int i = 1; i <= n; i++)
    sum = sum + i;
```

Khái niệm O-lớn



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

★ O-lớn quan tâm đến tỉ suất tăng trưởng của thời gian tính khi $n \to \infty$.

Nó không quan tâm khi dữ liệu đầu vào có kích thước nhỏ

★ Hàm $g(n)$ trong $O(g(n))$ là hàm để so sánh với các thuật toán khác

Nhắc lại một số hàm logarit

$$x^a = b \quad \Leftrightarrow \quad \log_x b = a$$

$$\log ab = \log a + \log b$$

$$\log_a b = \frac{\log_m b}{\log_m a}$$

$$\log a^b = b \log a$$

$$a^{\log n} = n^{\log a}$$

$$\log^b a = (\log a)^b \neq \log a^b$$

$$\frac{d \ln x}{dx} = \frac{1}{x}$$

Một số quy tắc của O-lớn

- O-lớn bỏ qua các giá trị có bậc thấp hơn.

  Các bậc thấp hơn thường được tính bởi

  - các bước khởi tạo
  - phép tính đơn giản
  - …

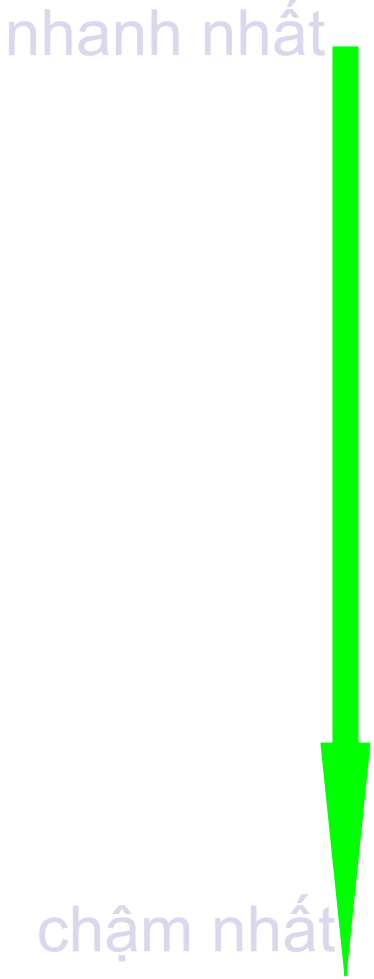
- O-lớn không tính đến hệ số nhân

  - Đây thường là khái niệm phụ thuộc vào máy tính

- Không cần chỉ ra cơ số của hàm logarit

  - Hoàn toàn có thể thay đổi cơ số của hàm logarit bằng cách nhân với một hằng số

Thứ hạng của O-lớn

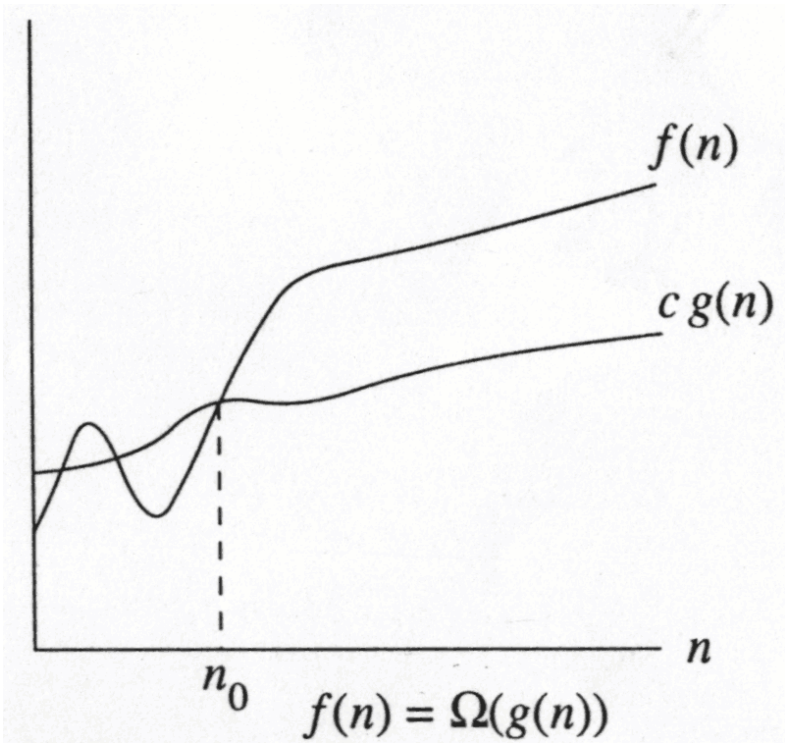


$O(1)$	thời gian hằng số
$O(\log n)$	thời gian logarit
$O(n)$	thời gian tuyến tính
$O(n \log n)$	
$O(n^2)$	bình phương
$O(n^2 \log n)$	
$O(n^3)$	mũ 3
$O(2^n)$	hàm số mũ n
$O(n!)$	giai thừa

Sự tăng trưởng của hàm?

Thuật toán	1	2	3	4	5
Thời gian (ms.)	$33n$	$46\ n \log n$	$13n^2$	$3.4n^3$	2^n
KT đầu vào (n)	Thời gian thực tế				
10	.00033 sec.	.0015s	.0013s	.0034s	.001s
100	.003s	.03s	.13s	3.4s	$4 \cdot 10$ yr.
1,000	.033s	.45s	13s	.94hr	
10,000	.33s	6.1s	22 min	39 days	
100,000	3.3s	1.3 min	1.5 days	108 yr.	
T/g cho phép	Kích thước dữ liệu tối đa				
1 sec.	30,000	2,000	280	67	20
1 min.	1,800,000	82,000	2,200	260	26

Khái niệm Omega-lớn



- $\exists$ c , $n_0 > 0$ sao cho $f(n) \geq c\, g(n)$ khi $n \geq n_0$
- f(n) tăng không chậm hơn g(n) với N "lớn"

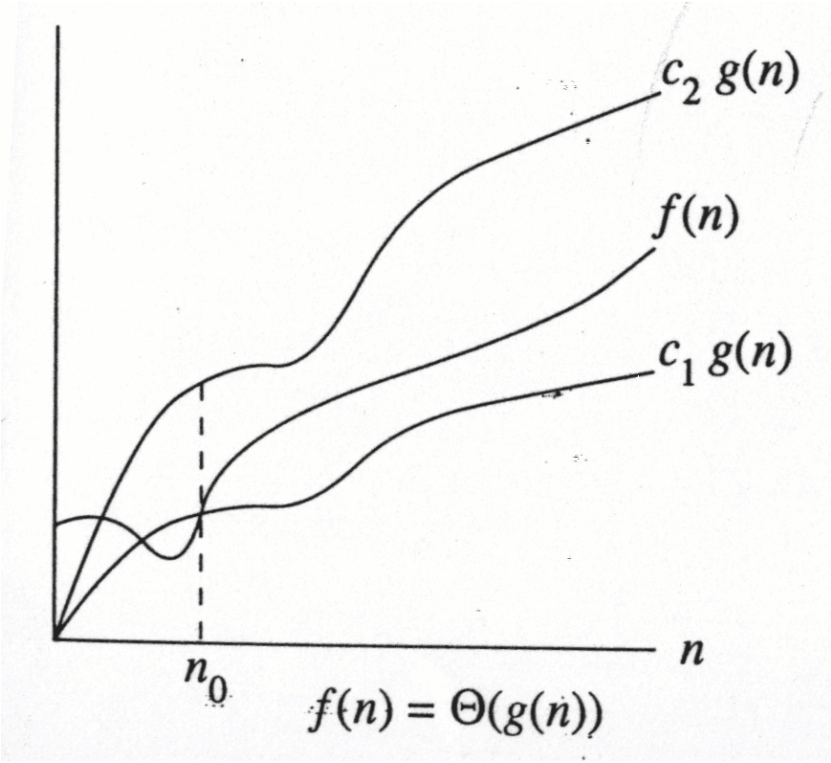
Khái niệm Omega-lớn

- $f(n) = \Omega(g(n))$
- Tồn tại một hằng số dương c và n_0 sao cho

$$f(n) \geq c\, g(n) \text{ khi } n \geq n_0$$

- Tỉ suất tăng của f(n) thì lớn hơn hoặc bằng tỉ suất tăng của g(n).

Khái niệm Theta-lớn



- Tỉ suất tăng của f(n) *bằng* tỉ suất tăng của g(n)

Theta-lớn

- $f(n) = \Theta(g(n))$ nếu và chỉ nếu
  $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
- Tỉ suất tăng của f(n) *bằng* tỉ suất tăng của g(n)
- Ví dụ: $f(N)=N^2$,

- Theta-lớn là cận chặt nhất có thể.

Một số quy tắc

- Nếu T(N) là một đa thức bậc k, thì
  $T(N) = \Theta(N^k)$.

- Với các hàm logarit,
  $T(\log_m N) = \Theta(\log N)$.

Ví dụ:

- $t(n) = 90\ n2 + 9n + 9$
  - Do
    $60n2 + 9n + 9 \leq 60\ n2 + 9n2 + n2 = 70\ n2$ với mọi $n \geq 1$,
    Chọn C1 = 70
    60n2 + 9n + 9 = O(n2).
  - Do
    60n2 + 9n + 9 ≥ 60 n2 với mọi n ≥ 1,
    Chọn C2=60
    60n2 + 9n + 9 = Ω (n2).
  - Do
    60n2 + 9n + 9 = O(n2) và 60n2 + 9n + 9 = Ω (n2)
    nên
    60n2 + 9n + 9 = Θ(n2).

Quy tắc L' Hopital

- Quy tắc L' Hopital
  - Nếu $\lim_{n\to\infty} f(N) = \infty$ và $\lim_{n\to\infty} g(N) = \infty$

    thì $\lim_{n\to\infty} \frac{f(N)}{g(N)} = \lim_{n\to\infty} \frac{f'(N)}{g'(N)}$

- Quyết định tỉ suất tăng tương đối (sử dụng quy tắc L' Hopital khi cần thiết)
  - Tính $\lim_{n\to\infty} \frac{f(N)}{g(N)}$
  - 0: f(N) = O(g(N)) và f(N) k phải là Θ(g(N))
  - Hằng số ≠ 0: f(N) = Θ(g(N))
  - ∞: f(N) = Ω(f(N)) và f(N) k phải là Θ(g(N))
  - không xác định: không có mối quan hệ gì

Xác định độ phức tạp về thời gian

Nếu $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$ thì

- Quy tắc tổng:

  $T_1(n) + T_2(n) = O(f(n)+g(n))$

  $= \max(O(f(n)),O(g(n)))$

  $= O(\max(f(n), g(n)))$

- Quy tắc nhân:

  $T_1(n) * T_2(n) = O(f(n) * g(n))$

Xác định độ phức tạp thời gian

- Với vòng lặp
  - là thời gian chạy của các câu lệnh bên trong vòng lặp (kể cả lệnh kiểm tra điều kiện) nhân với số lần lặp.
- Các vòng lặp lồng nhau

```
for (i=0;i<n;i++)
    for (j=0;j<n;j++)
        k++;
```

  - là thời gian chạy của câu lệnh nhân với tích của các kích thước của tất cả vòng lặp.

Xác định độ phức tạp thời gian

- Các câu lệnh kế tiếp

```
for (i=0;i<n;i++)
      a[i]=0;
for (i=0;i<n;i++)
      for (j=0;j<n;j++)
            a[i] += a[j]+i+j;
```

  - Thực hiện tính tổng
  - $O(N) + O(N^2) = O(N^2)$

- `If S1`

  `Else S2`

  - thời gian của lệnh kiểm tra điều kiện + thời gian tính lớn hơn trong S1 và S2.

Cấu trúc dữ liệu và giải thuật

Người thực hiện: Đỗ Tuấn Anh
Email: anhdt@it-hut.edu.vn
ĐT: 0989095167

Nội dung

- Chương 1 – Thiết kế và phân tích (5 tiết)
- Chương 2 – Giải thuật đệ quy (10 tiết)
- Chương 3 – Mảng và danh sách (5 tiết)
- Chương 4 – Ngăn xếp và hàng đợi (10 tiết)
- Chương 5 – Cấu trúc cây (10 tiết)
- Chương 8 – Tìm kiếm (5 tiết)
- Chương 7 – Sắp xếp (10 tiết)
- Chương 6 – Đồ thị (5 tiết)

Chương 2 – Giải thuật đệ quy

1. Khái niệm
2. Thiết kế giải thuật đệ quy
3. Hiệu lực của đệ quy
4. Đệ quy và quy nạp toán học
5. Đệ quy quay lui

1. Khái niệm

- Là một kỹ thuật giải quyết bài toán quan trọng trong đó:
  - Phân tích đối tượng thành các thành phần nhỏ hơn mang tính chất của chính đối tượng đó.
- Ví dụ:
  - Định nghĩa số tự nhiên:
    - 1 là một số tự nhiên
    - x là một số tự nhiên nếu x-1 là một số tự nhiên

Ví dụ 1 - Tính giai thừa

- Hàm tính giai thừa cho một số nguyên:

$$n! = n * (n - 1) * \ldots * 1$$

- Định nghĩa đệ quy:

$$n! = \begin{cases} 1 & \text{if } n = 0 \quad \text{// điều kiện dừng} \\ n * (n - 1)! & \text{if } n > 0 \quad \text{// bước đệ quy} \end{cases}$$

Tính giai thừa

- Định nghĩa đệ quy chỉ ra một cách chính xác cách tính giai thừa

```
4! =  4 * 3!                          // Bước đệ quy (n = 4)
   =  4 * ( 3 * 2! )                  // Bước đệ quy (n = 3)
   =  4 * ( 3 * ( 2 * 1!) )           // Bước đệ quy (n = 2)
   =  4 * ( 3 * ( 2 * ( 1 * 0! ) ) )  // Bước đệ quy (n = 1)
   =  4 * ( 3 * ( 2 * ( 1 * 1 ) ) )   // Điều kiện dừng ( n = 0)
   =  4 * ( 3 * ( 2 * 1 ) )
   =  4 * ( 3 * 2 )
   =  4 * 6
   =  24
```

1. Khái niệm (tiếp)

- Giải thuật đệ quy: T được thực hiện bằng T' có dạng giống như T
- Giải thuật đệ quy phải thỏa mãn 2 điều kiện:
  - Phải có điểm dừng: là trường hợp cơ sở (suy biến) nhỏ nhất, được thực hiện không cần đệ quy
  - Phải làm cho kích thước bài toán thu nhỏ hơn: do đó làm cho bài toán giảm dần đến trường hợp cơ sở
- Thủ tục đệ quy:
  - Có lời gọi đến chính nó (đệ quy trực tiếp) hoặc chứa lời gọi đến thủ tục khác và thủ tục này chứa lời gọi đến nó (đệ quy gián tiếp)
  - Sau mỗi lần gọi, kích thước bài toán thu nhỏ hơn
  - Phải kiểm tra điểm dừng

Giải thuật đệ quy – ví dụ

- Tìm file trong thư mục trên máy tính
- Tra từ trong từ điển Anh-Anh

2. Thiết kế giải thuật đệ quy

- 3 bước:
  - Thông số hóa bài toán
  - Tìm điều kiện dừng
  - Phân rã bài toán
- Ví dụ bài toán: Tính N!

Bước 1: Thông số hóa bài toán

- Tìm các thông số biểu thị kích thước của bài toán
- Quyết định độ phức tạp của bài toán
- Ví dụ: Tính N!
  - N trong hàm tính giai thừa của N

Bước 2: Tìm điều kiện dừng

- Là trường hợp giải không đệ quy
- Là trường hợp kích thước bài toán nhỏ nhất
- Ví dụ: Tính N!
  - 0! = 1

Bước 3: Phân rã bài toán

- Phân rã bài toán thành các thành phần:
  - Hoặc không đệ quy
  - Hoặc là bài toán trên nhưng kích thước nhỏ hơn
- Bài toán viết được dưới dạng công thức đệ quy => đơn giản
- Ví dụ: Tính N!
  - N! = N * (N-1)!

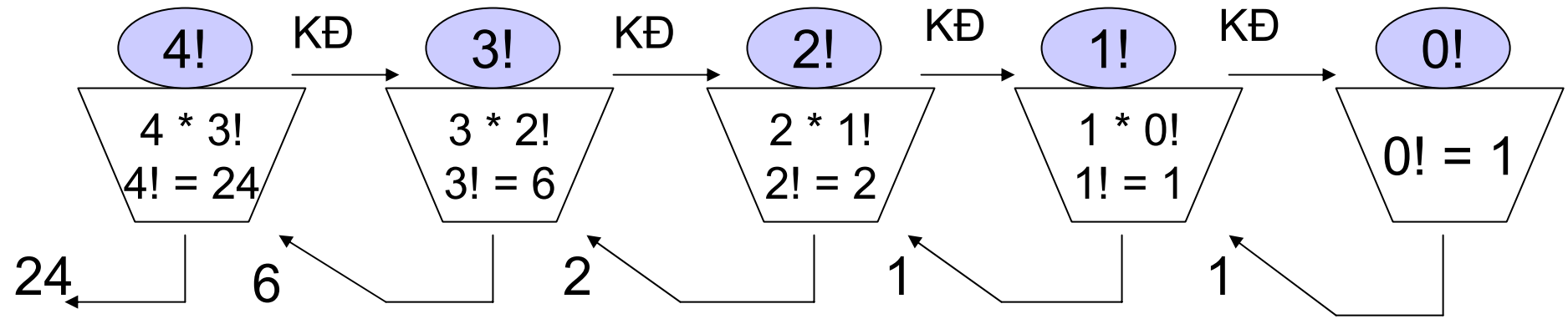
Chương trình tính giai thừa

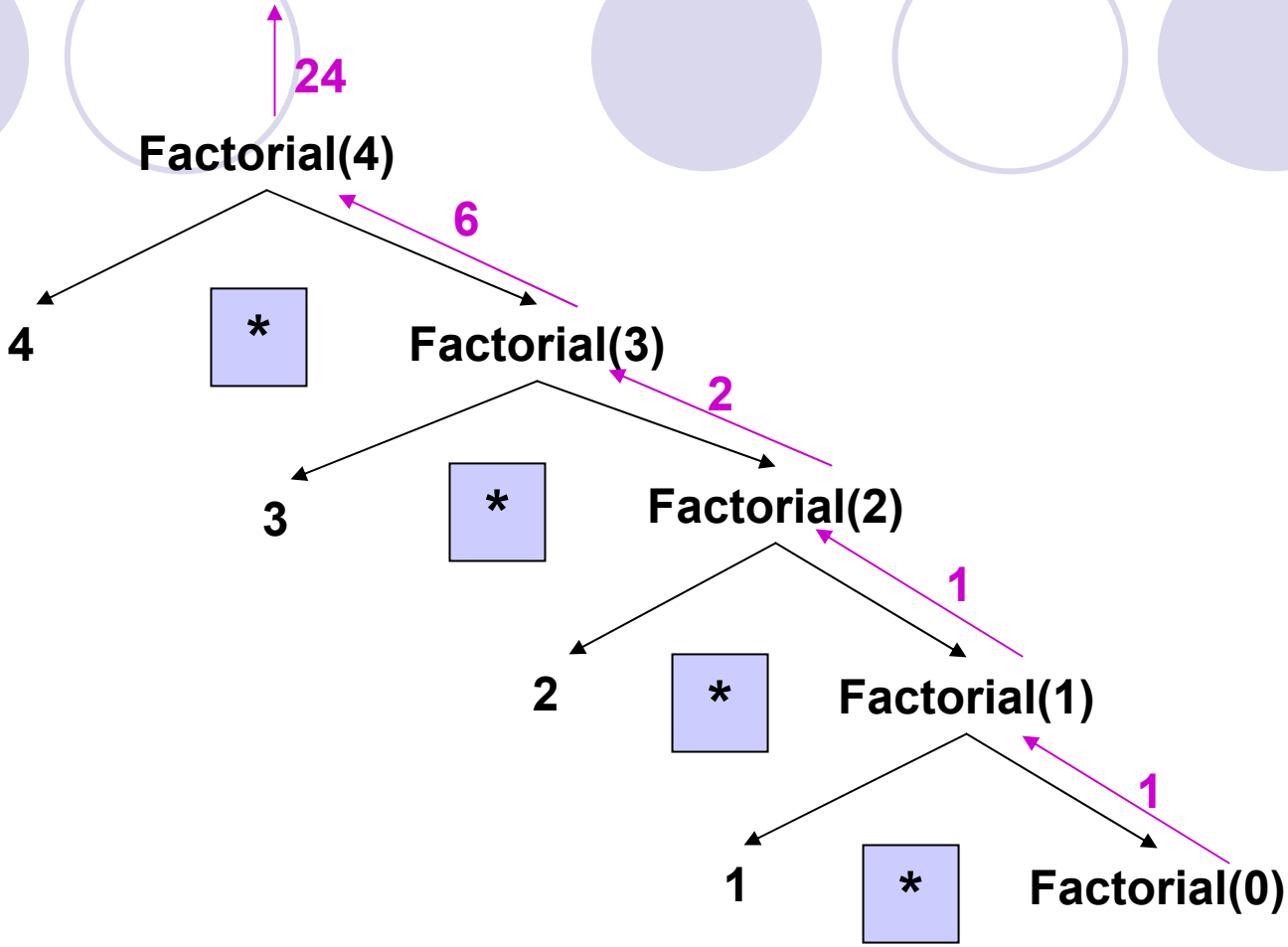
```
// Sử dụng đệ quy
long Factorial (long n)
{
    // điều kiện dừng n == 0
    if  (n == 0)
        return 1;
    else   // bước đệ quy
        return n * Factorial (n-1);
}
```

Quan điểm N-máy

Hàm tính giai thừa (n) có thể được xem như được thực hiện bởi n-máy:

Máy 4 (4 * 3!) khởi động máy 3
Máy 3 (3 * 2!) khởi động máy 2
Máy 2 (2 * 1!) khởi động máy 1
Máy 1 (1 * 0!) khởi động máy 0




24
Factorial(4)
6
4
*
Factorial(3)
2
3
*
Factorial(2)
1
2
*
Factorial(1)
1
1
*
Factorial(0)

Điều kiện đệ quy

Phải có điểm dừng: nếu không sẽ tạo thành một chuỗi vô hạn các lời gọi hàm

```
long Factorial(long n){
    return n * Factorial(n-1);
}
```

Oops!
Không có điểm dừng

Phải làm cho bài toán đơn giản hơn:

```
long Factorial(long n){
    if (n==0)
        return 1;
    else
        return n * Factorial(n+1);
}
```

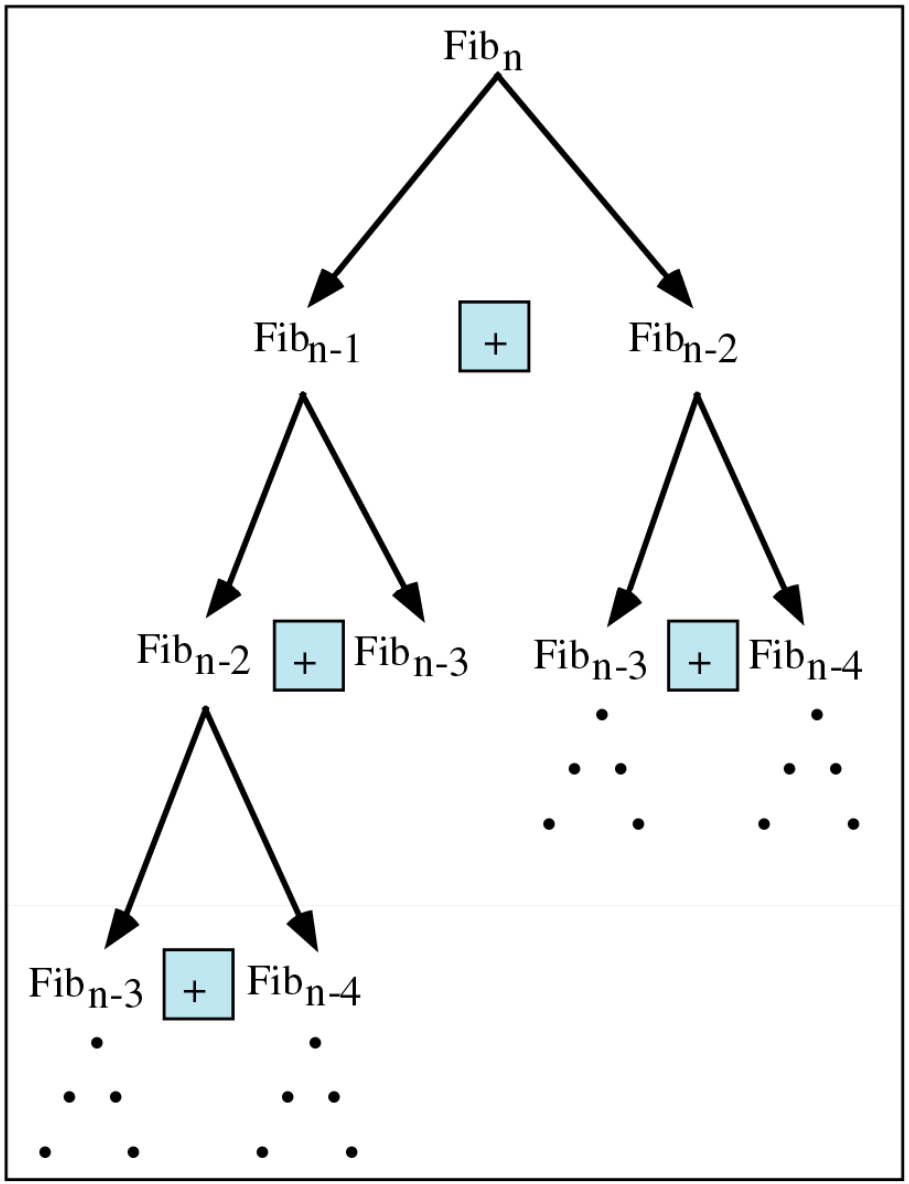
Oops!

Dãy số Fibonacci

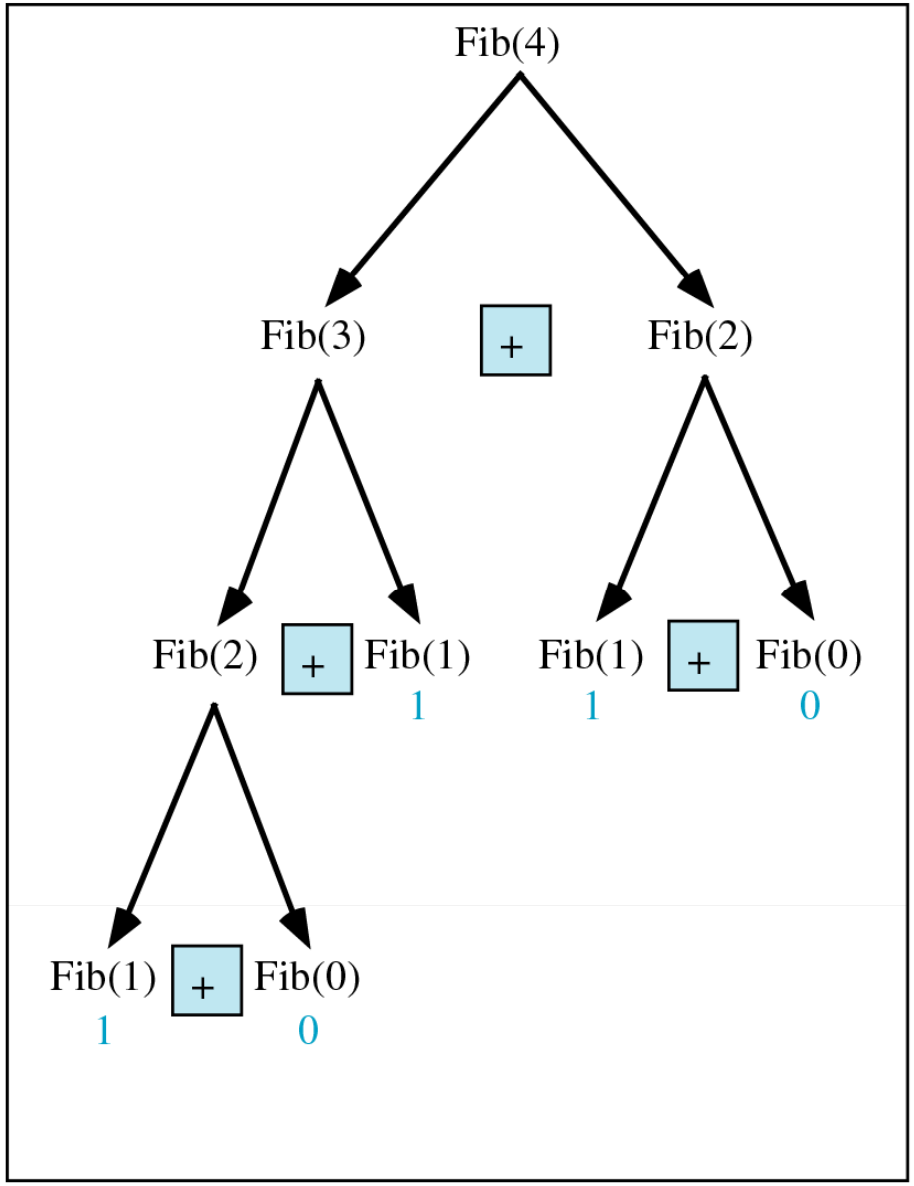
- Dãy số Fibonacci:

  `0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...`

  trong đó mỗi số là tổng của 2 số đứng trước nó.
- Định nghĩa theo đệ quy:
  - `F(0) = 0;`
  - `F(1) = 1;`
  - `F(n) = F(n-1)+ F(n-2);`


Fib$_n$
Fib$_{n-1}$
+
Fib$_{n-2}$
Fib$_{n-2}$
+
Fib$_{n-3}$
Fib$_{n-3}$
+
Fib$_{n-4}$
Fib$_{n-3}$
+
Fib$_{n-4}$

(a) Fib(n)


Fib(4)
Fib(3)
+
Fib(2)
Fib(2)
+
Fib(1)
1
Fib(1)
1
+
Fib(0)
0
Fib(1)
1
+
Fib(0)
0

(b) Fib(4)

Dãy số Fibonacci – Thủ tục đệ quy

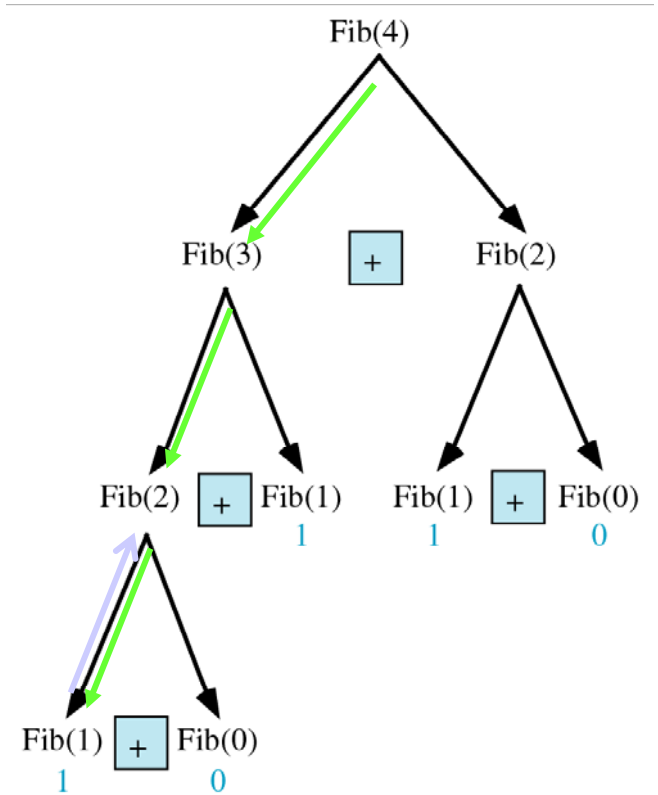
```
//Tính số Fibonacci sử dụng hàm đệ quy.
int fib(int number)
{
   if (number == 0) return 0;
   if (number == 1) return 1;
   return (fib(number-1) + fib(number-2));
}
int main(){
   int inp_number;
   printf ("Please enter an integer: “);
   scanf (“%d”, &inp_number);
   int intfib = fib(inp_number);
   printf("The Fibonacci number for %d is %d\n“,inp_number,intfib);
  return 0;
}
```

Cơ chế thực hiện

- Tính fibonacci của 4, num=4:

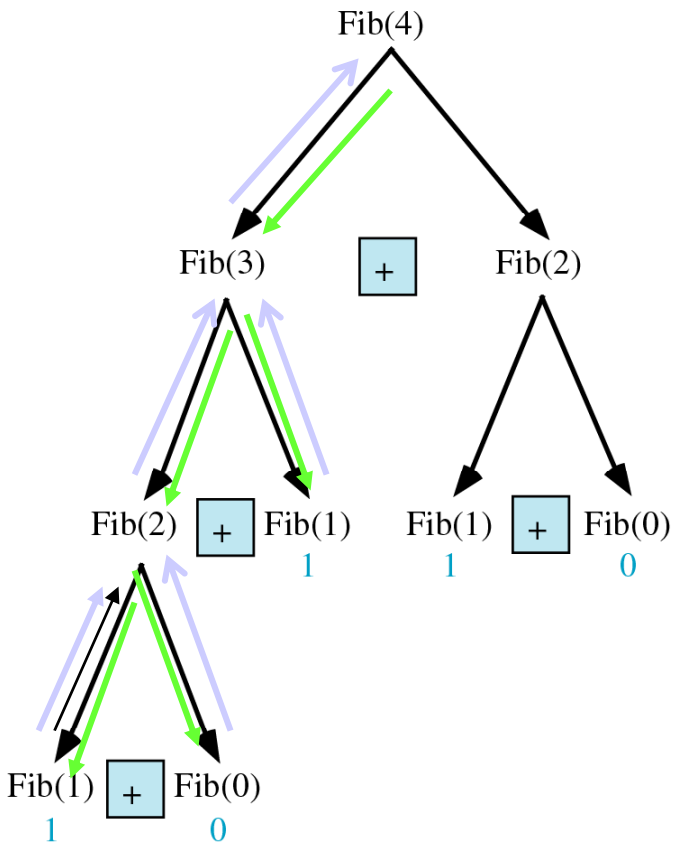
```
fib(4):
    4 == 0 ? Sai;    4 == 1?     Sai.
    fib(4) = fib(3) + fib(2)
    fib(3):
        3 == 0 ? Sai; 3 == 1? Sai.
        fib(3) = fib(2) + fib(1)
        fib(2):
            2 == 0? Sai; 2==1? Sai.
            fib(2) = fib(1)+fib(0)
            fib(1):
                1== 0 ? Sai;
                1 == 1? Đúng.
                fib(1) = 1;
                return fib(1);
```

```
int fib(int num)
{
    if (num == 0) return 0;
    if (num == 1) return 1;
    return
      (fib(num-1)+fib(num-2));
}
```



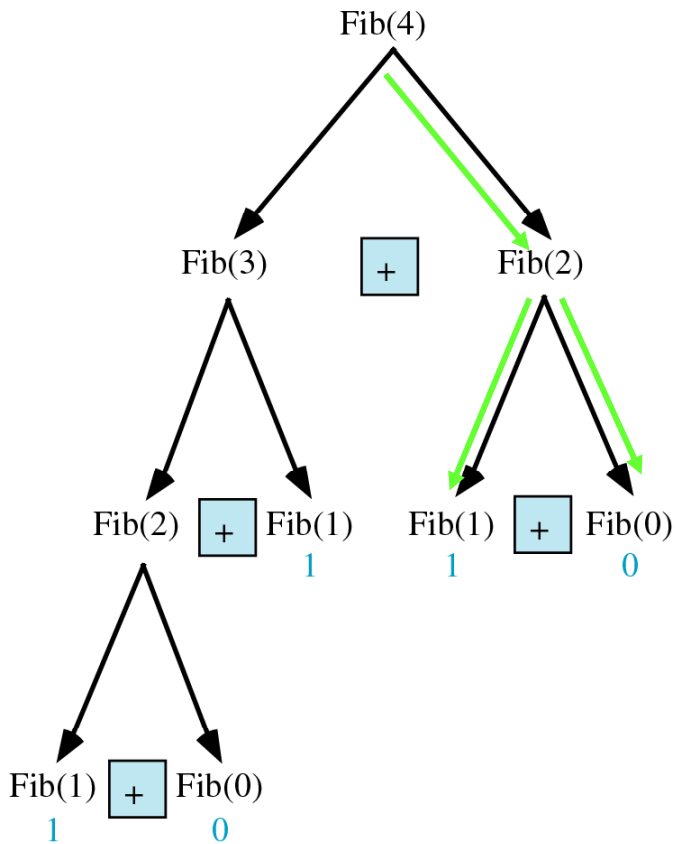
Cơ chế thực hiện

```
fib(0):
    0 == 0 ?  Đúng.
    fib(0) = 0;
    return fib(0);
fib(2) = 1 + 0 = 1;
return fib(2);
fib(3) = 1 + fib(1)
  fib(1):
   1 == 0 ? Sai;
   1 == 1? Đúng
   fib(1) = 1;
   return fib(1);
fib(3) = 1 + 1 = 2;
return fib(3)
```



Cơ chế thực hiện

```
fib(2):
    2 == 0 ? Sai; 2 == 1?  Sai.
    fib(2) = fib(1) + fib(0)
    fib(1):
       1== 0 ? Sai; 1 == 1? Đúng.
        fib(1) = 1;
        return fib(1);
    fib(0):
        0 == 0 ?   Đúng.
        fib(0) = 0;
        return fib(0);
    fib(2) = 1 + 0 = 1;
    return fib(2);
fib(4) = fib(3) + fib(2)
       = 2 + 1 = 3;
return fib(4);
```

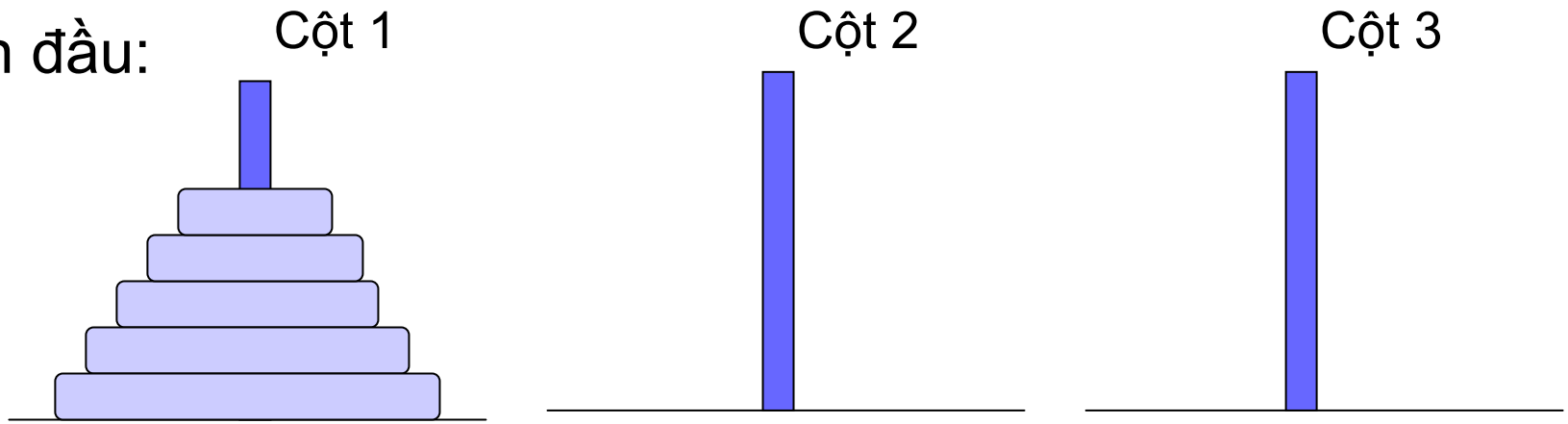


Thủ tục đệ quy tổng quát

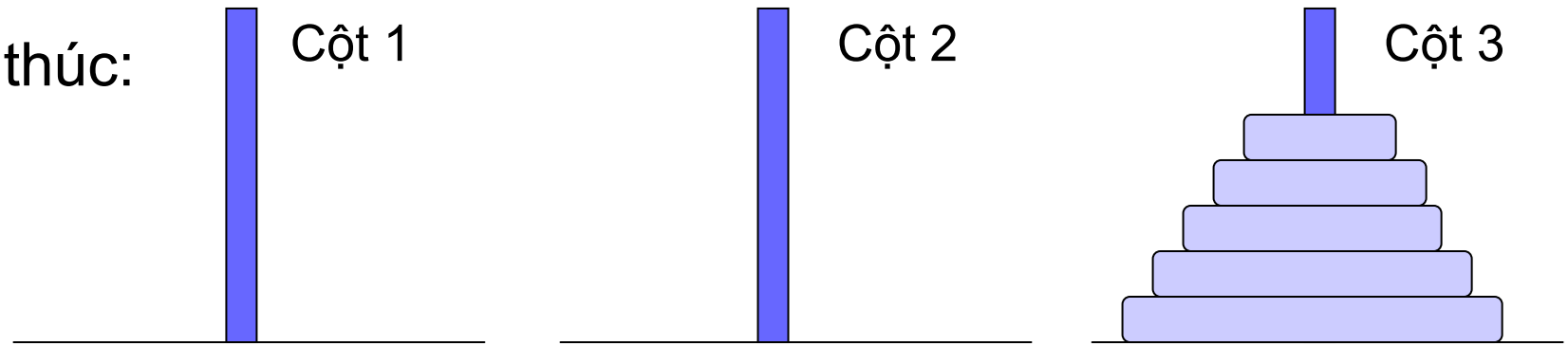
```
int Hàm_đệ_quy(DS tham số){
  if(thỏa mãn điều kiện dừng)
     return giá_trị_dừng_tương_ứng;
  // other stopping conditions if needed
  return hàm_đệ_quy(tham số suy giảm)
}
```

Bài toán tháp Hà Nội

Ban đầu:



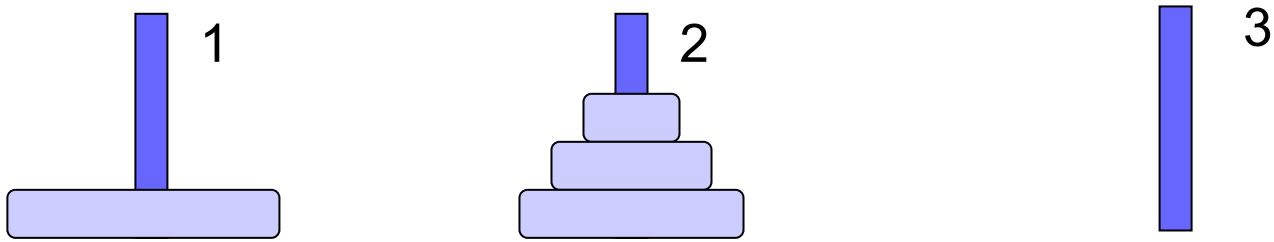
Kết thúc:



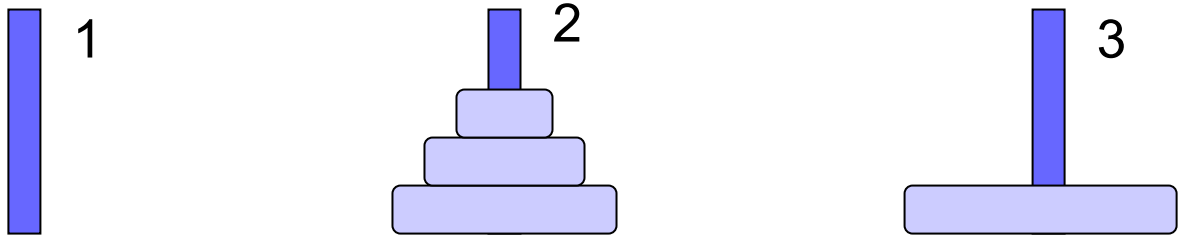
Quy tắc: đĩa lớn hơn không được đặt trên đĩa nhỏ hơn trong quá trình chuyển đĩa

Giải thuật đệ quy

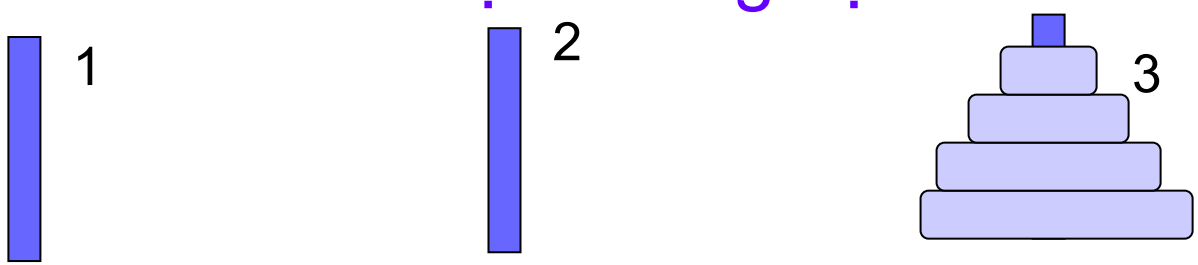
1. Chuyển *n* – 1 đĩa từ cột 1 sang cột 2



2. Chuyển đĩa dưới cùng từ cột 1 sang 3



3. Chuyển n-1 đĩa từ cột 2 sang cột 3



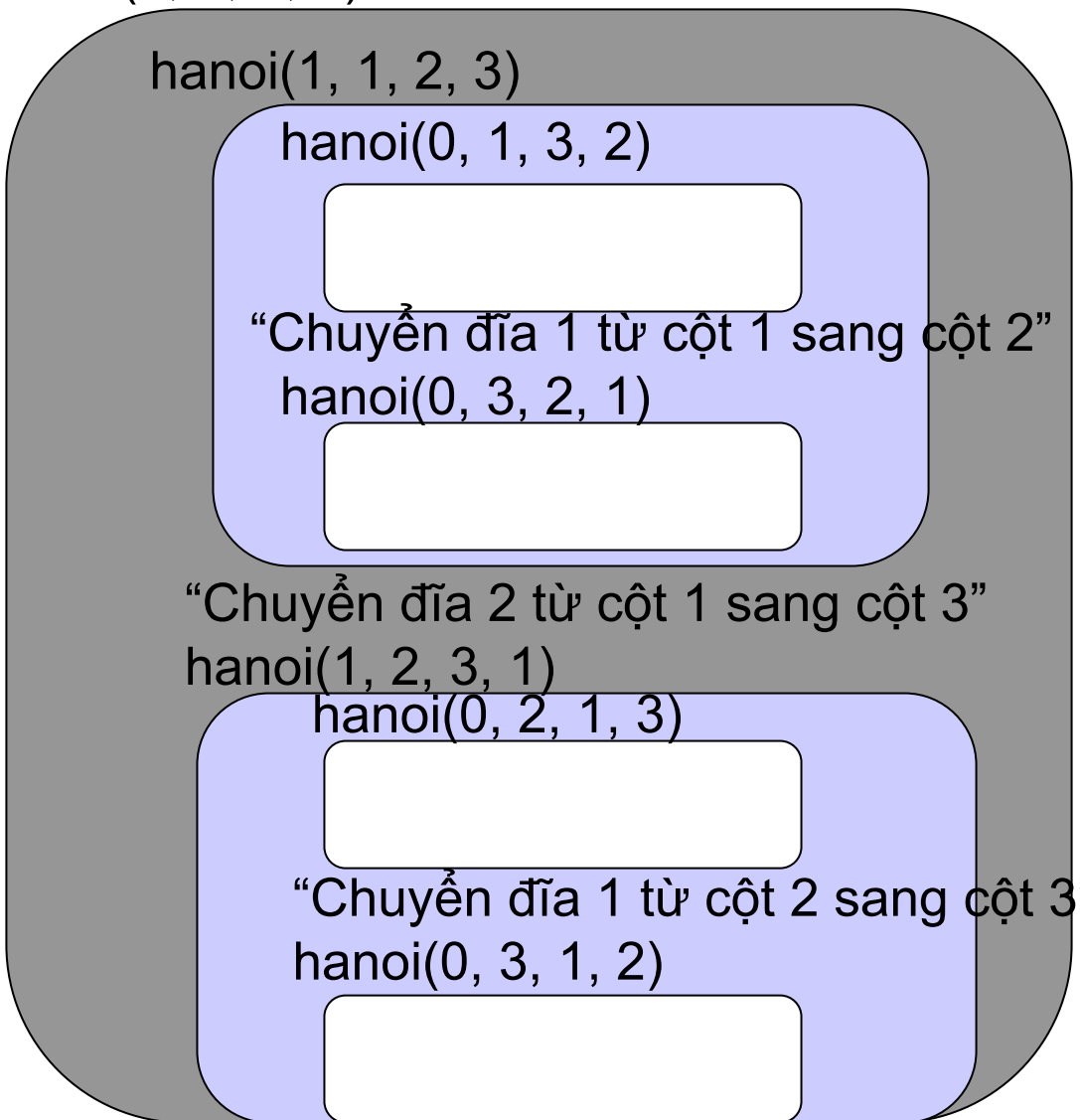
Thủ tục đệ quy

```
// chuyển n đĩa từ cột nguồn sang cột đích
// sử dụng một đĩa trung gian
void hanoi (int n, int cot1, int cot3, int cot2)
{
     if (n > 0)
     {
          hanoi(n-1, cot1, cot2, cot3);
          Chuyen_dia(n, cot1, cot3);
          hanoi(n-1, cot2, cot3, cot1);
      }
}
```

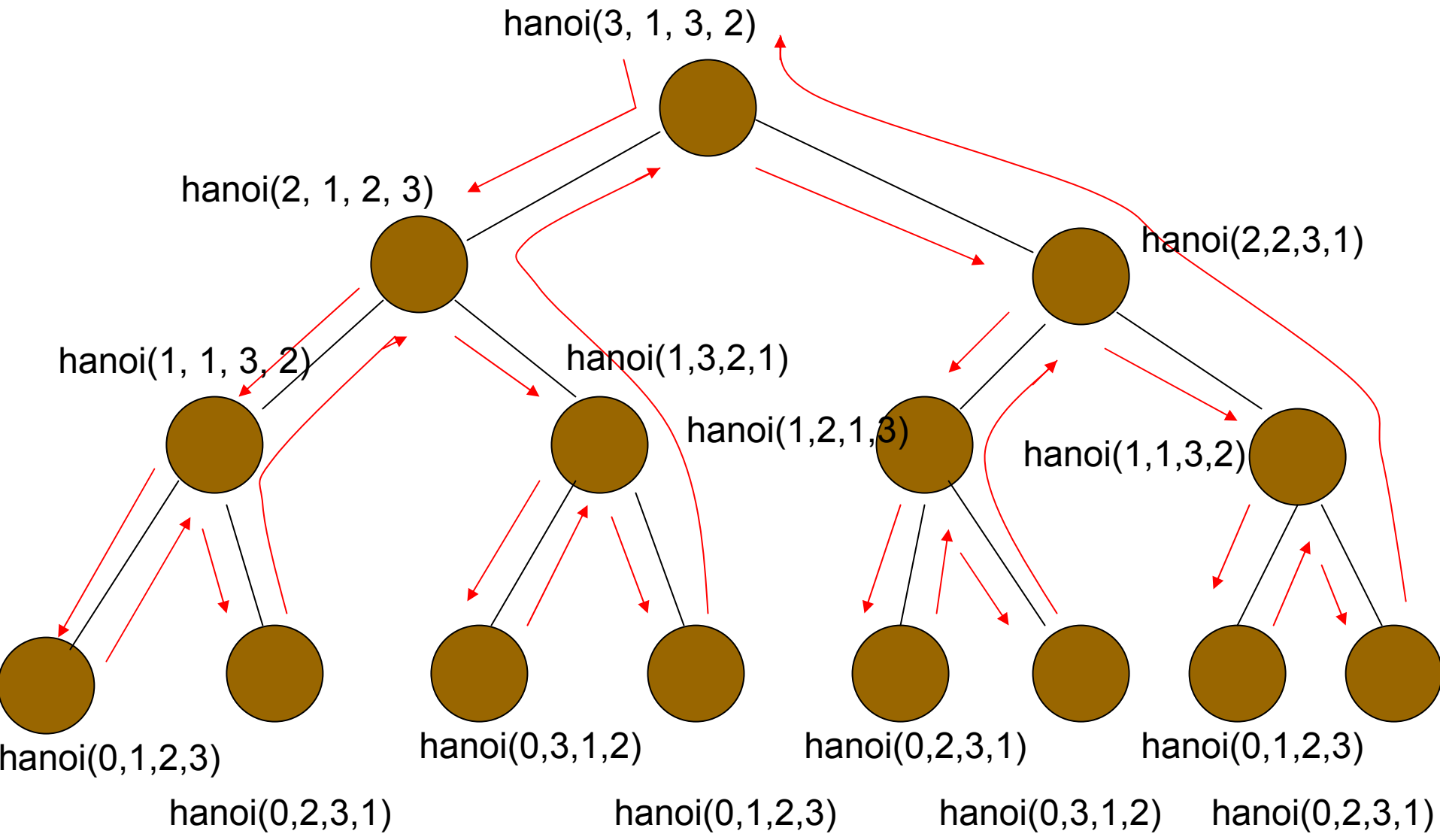
Cơ chế thực hiện

hanoi(n, cot1, cot3, cot2)

hanoi(2, 1, 3, 2)



Cây đệ quy trong trường hợp chuyển 3 đĩa

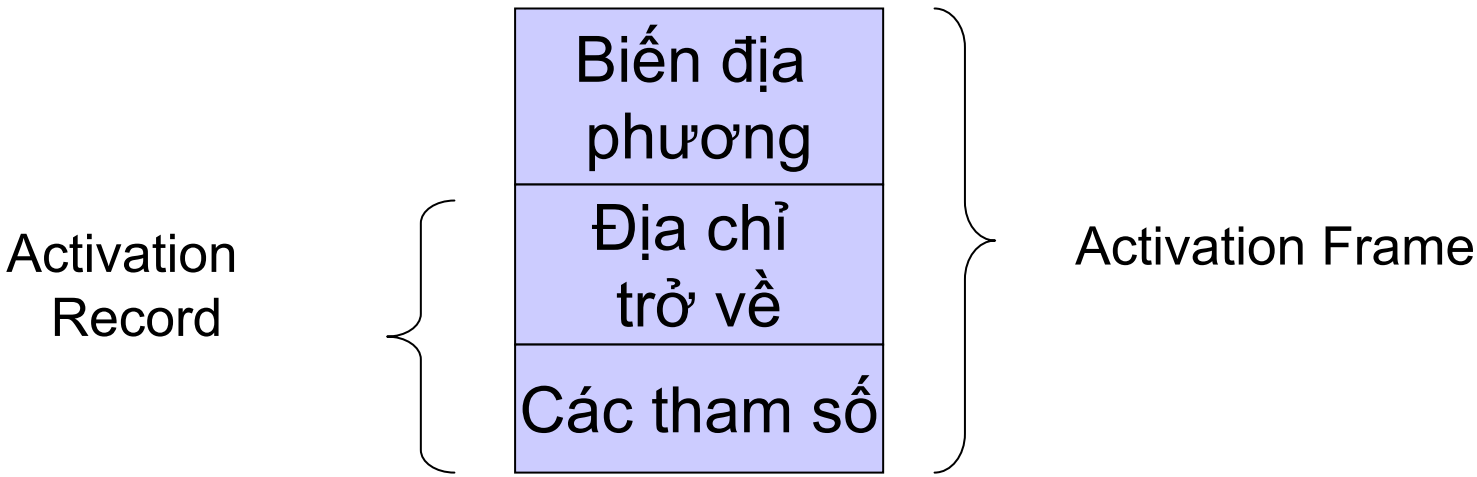


4. Hiệu quả của giải thuật đệ quy

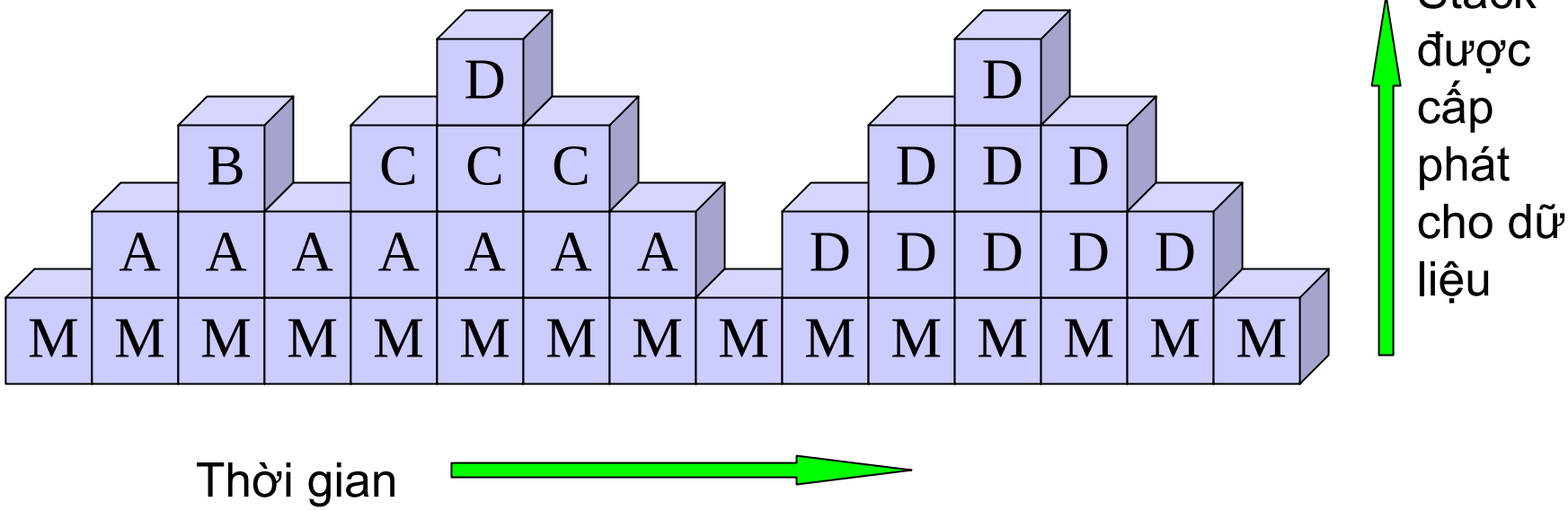
- Nhược điểm:
  - Tốn không gian nhớ
  - Tốc độ chậm
- Ưu điểm: đơn giản, ngắn gọn, dễ viết code
  - Một số giải thuật đệ quy cũng có hiệu lực cao, ví dụ như Quick sort
- Mọi giải thuật đệ quy đều có thể thay thế bằng một giải thuật không đệ quy (sử dụng vòng lặp)

Gọi hàm và Bộ nhớ Stack

Runtime stack: khi hàm được gọi, một vùng nhớ trên stack được sử dụng để lưu trữ: các tham số, địa chỉ trở về của hàm

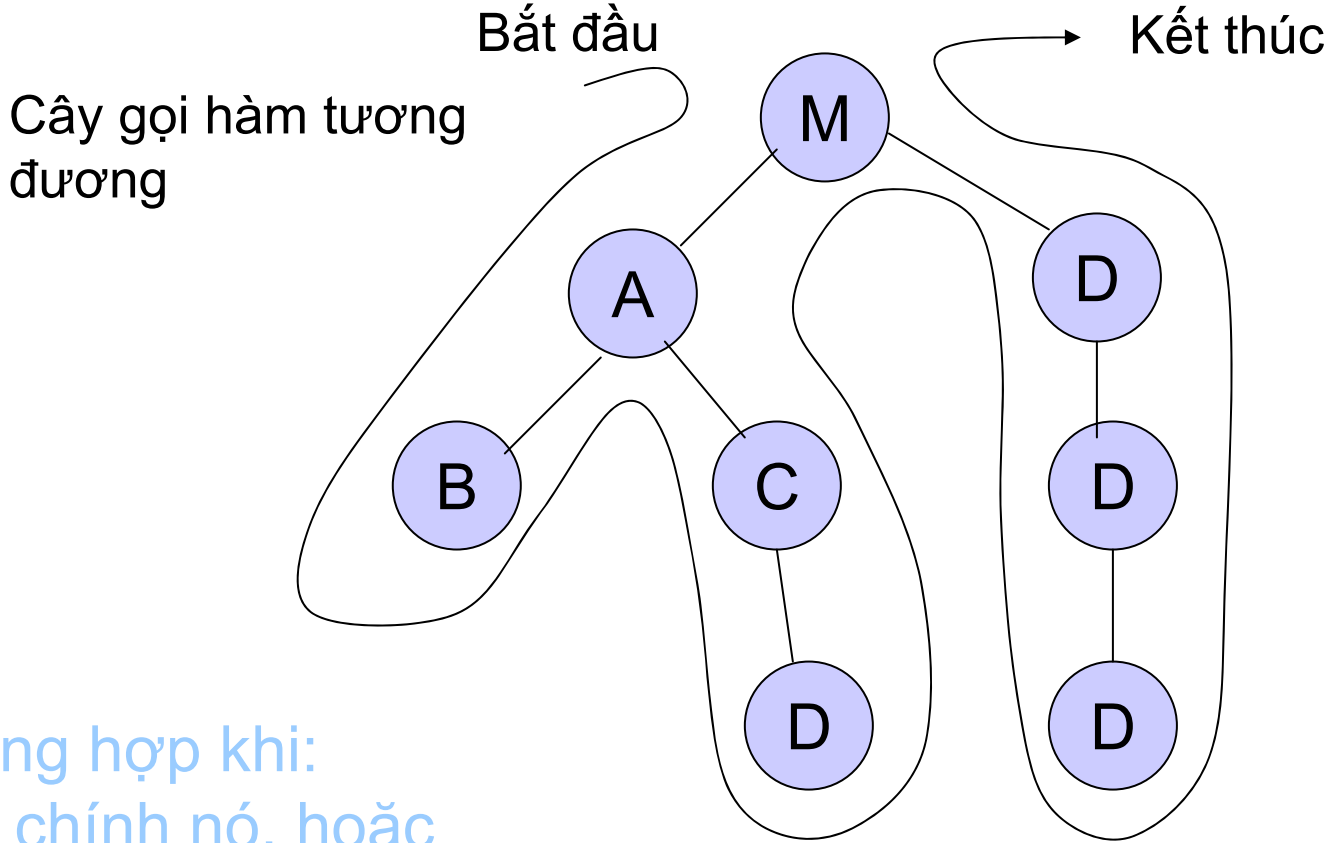


Đệ quy và Stack



Các cột theo chiều dọc chỉ ra nội dung của stack tại một thời điểm, và sự thay đổi của stack khi gọi hàm và thoát khỏi hàm

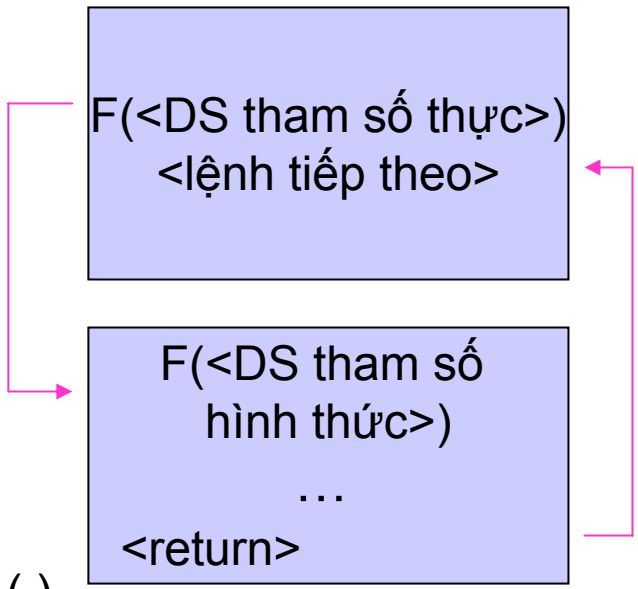
Cây lời gọi hàm



Đệ quy là trường hợp khi:

- Một hàm gọi chính nó, hoặc
- Một hàm gọi một chuỗi các hàm khác trong đó một/ một vài trong số chúng gọi lại hàm đầu tiên

Gọi hàm và địa chỉ trở về




```
void main ( )
{
    int n;
    // đẩy activation record của Factorial(4)
    // vào Stack
    n = Factorial(4);

}
```

RecLoc1

```
long Factorial (long n)
{
    int temp;
    if (n == 0)
        return 1;// giải phóng activation recor
    else
    {   // đẩy activation record của
        // lời gọi Factorial(n-1)
        temp = n * Factorial (n-1);
    }
    return temp;   // giải phóng activation
                   // record
}
```

RecLoc2

Factorial(4) và Stack

	tham_số	địa_chỉ_trả_về
Factorial(0)	0	RecLoc2
Factorial(1)	1	RecLoc2
Factorial(2)	2	RecLoc2
Factorial(3)	3	RecLoc2
Factorial(4)	4	RecLoc1

Lệnh trước khi trả về

```
temp = 1 * 1;  // 1 từ Factorial (0)
return temp;
```

```
temp = 2 * 1;  // 1 từ Factorial(1)
return temp;
```

```
temp = 3 * 2;  // 2 từ Factorial(2)
return temp;
```

```
temp = 4 * 6;  // 6 từ Factorial(3)
return temp;
```

```
N = Factorial(4);    // quay lại main
```

Khử đệ quy

- Hàm tính giai thừa không đệ quy

```
// Sử dụng vòng lặp
long Factorial (long n)
{
    long prod = 1;   // 0! = 1
    // tính tích = 1*2* … * n
    for (i = 1;  i < n+1;  i++)
        prod * = i;
    return prod;
}
```

Hàm tính Fibonacci không đệ quy

```
//Tính số Fibonacci sử dụng vòng lặp
//hiệu quả hơn nhiều so với dùng đệ quy
int fib(int n)
{
  int f[n+1];
  f[0] = 0; f[1] = 1;
   for (int i=2; i<= n; i++)
        f[i] = f[i-1] + f[i-2];
  return f[n];
}
```

4. Đệ quy và Quy nạp toán học

- Chứng minh tính đúng đắn của giải thuật Factorial

Đánh giá giải thuật Tháp Hà nội

Gọi f(n) là số lần chuyển đĩa cần thiết để chuyển n đĩa từ cột 1 sang cột 3.

$f(1) = 1;$

$f(n) = 2 * f(n - 1) + 1, \quad \text{if } n > 1$

Dự đoán: $f(n) = 2 * f(n - 1) + 1$

$= 2^2 * f(n - 2) + 2 + 1$

$= \ldots$

$= 2^{n-1} * f(1) + \ldots + 2 + 1$

$= 2^{n-1} + 2^{n-2} + \ldots + 2 + 1$

$= 2^n - 1$

Chứng minh?

- Chứng minh bằng quy nạp

  $f(1) = 2^1 - 1 = 1$

  Giả sử đúng với n = k

  $f(k) = 2^k - 1$

  $f(k+1) = 2*f(k) + 1$

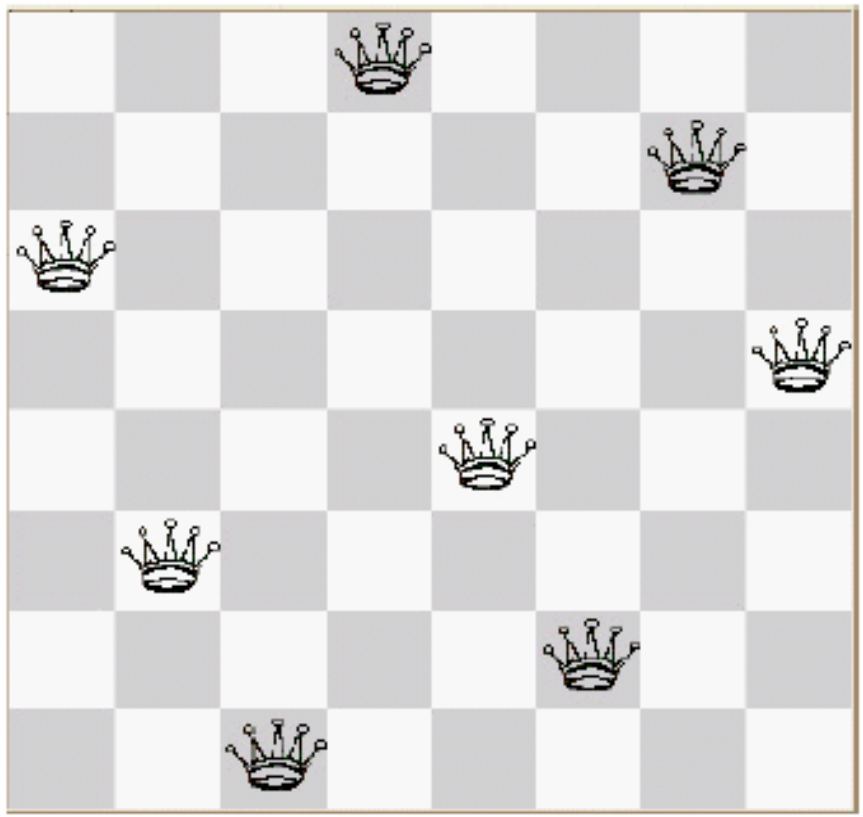
  $= 2*(2^k - 1) + 1$

  $= 2^{k+1} - 1$ => Công thức đúng

Các nhà sư phải chuyển 64 đĩa. Giả sử mỗi lần chuyển mất 1 giây, các nhà sư sẽ phải mất $5 * 10^{11}$ năm = 25 lần tuổi của vũ trụ. Khi chuyển xong chồng đĩa thì đã đến ngày tận thế!

5. Đệ quy quay lui (back tracking)

- Bài toán 8 con hậu: “Hãy xếp 8 con hậu trên bàn cờ 8x8 sao cho không có con hậu nào có thể ăn con hậu nào”



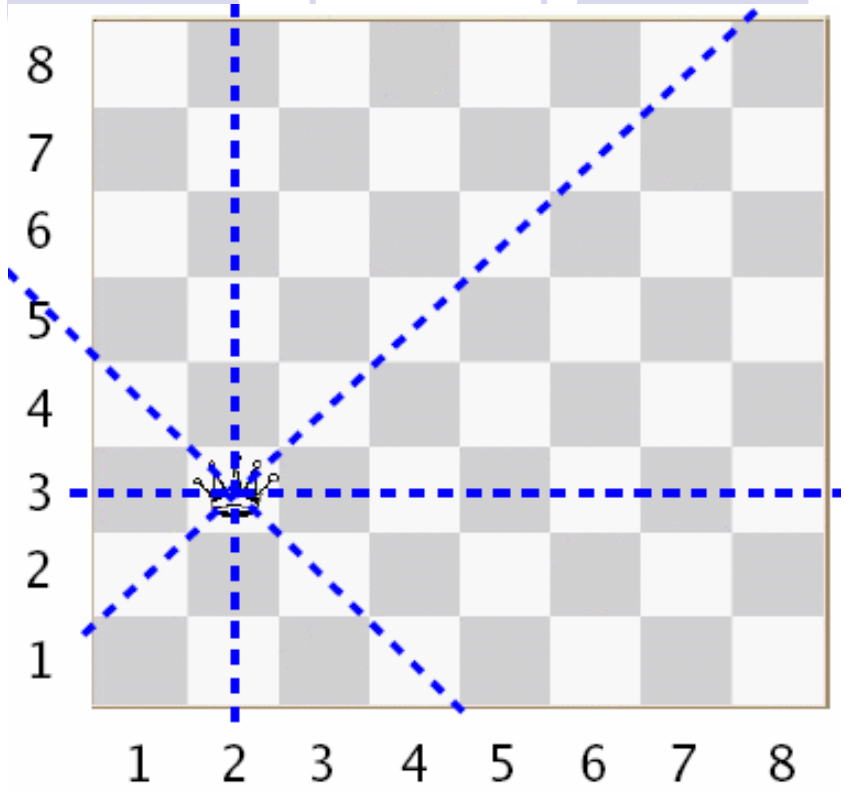
Đệ quy quay lui

- Phương pháp “thử từng bước”
  - Thử dự đoán
  - Nếu dự đoán là sai, quay trở lại và thử dự đoán khác => quay lui
- Dễ dang thể hiện phương pháp quay lui bằng đệ quy
  - Các biến trạng thái của hàm đệ quy được lưu trữ trên Stack
  - Quay lui lại trạng thái ban đầu ⇔ Quay trở lại hàm trước đó (hàm gọi hàm hiện tại)

Bài toán 8 con hậu

- Giải thuật 1:
  - Thử lần lượt tất cả các trường hợp ứng với mọi vị trí của 8 con hậu
  - Số phép thử = 64*63*…*58*57
    = 178,462,987,637,760

Bài toán 8 con hậu



- Nhận xét:
  - Mỗi cột phải có 1 con hậu
    - Con hậu 1 nằm trên cột 1
    - …
    - Con hậu j nằm trên cột j
    - …
    - Con hậu 8 nằm trên cột 8
  - Các con hậu phải không cùng hàng
  - Các con hậu phải không nằm trên đường chéo của nhau

Bài toán 8 con hậu

- Bài toán: Con hậu thứ j nằm trên cột j
  - [1/Y1, 2/Y2, 3/Y3, 4/Y4, 5/Y5, 6/Y6, 7/Y7, 8/Y8]
  - Lựa chọn hàng cho từng con hậu để mỗi con hậu không ăn nhau
- Giải thuật:
  - Thử lần lượt từng vị trí hàng của con hậu 1 (1-8)
  - Với từng vị trí của con hậu 1
    - Thử lần lượt từng vị trí hàng của con hậu 2
    - Với từng vị trí của con hậu 2
      - Thử lần lượt từng vị trí hàng của con hậu 3

Giải thuật

```
function Try (column) {
    for (row = 1; row <= 8; row++) {
        if ( [row, column] là an toàn) {
            Đặt con hậu vào vị trí [row, column];
            if (column == 8)
                In kết quả;
            else
                Try (column + 1);
            Xóa con hậu khỏi vị trí [row, column];
        }
    }
}
```

Thử lần lượt từng vị trí hàng

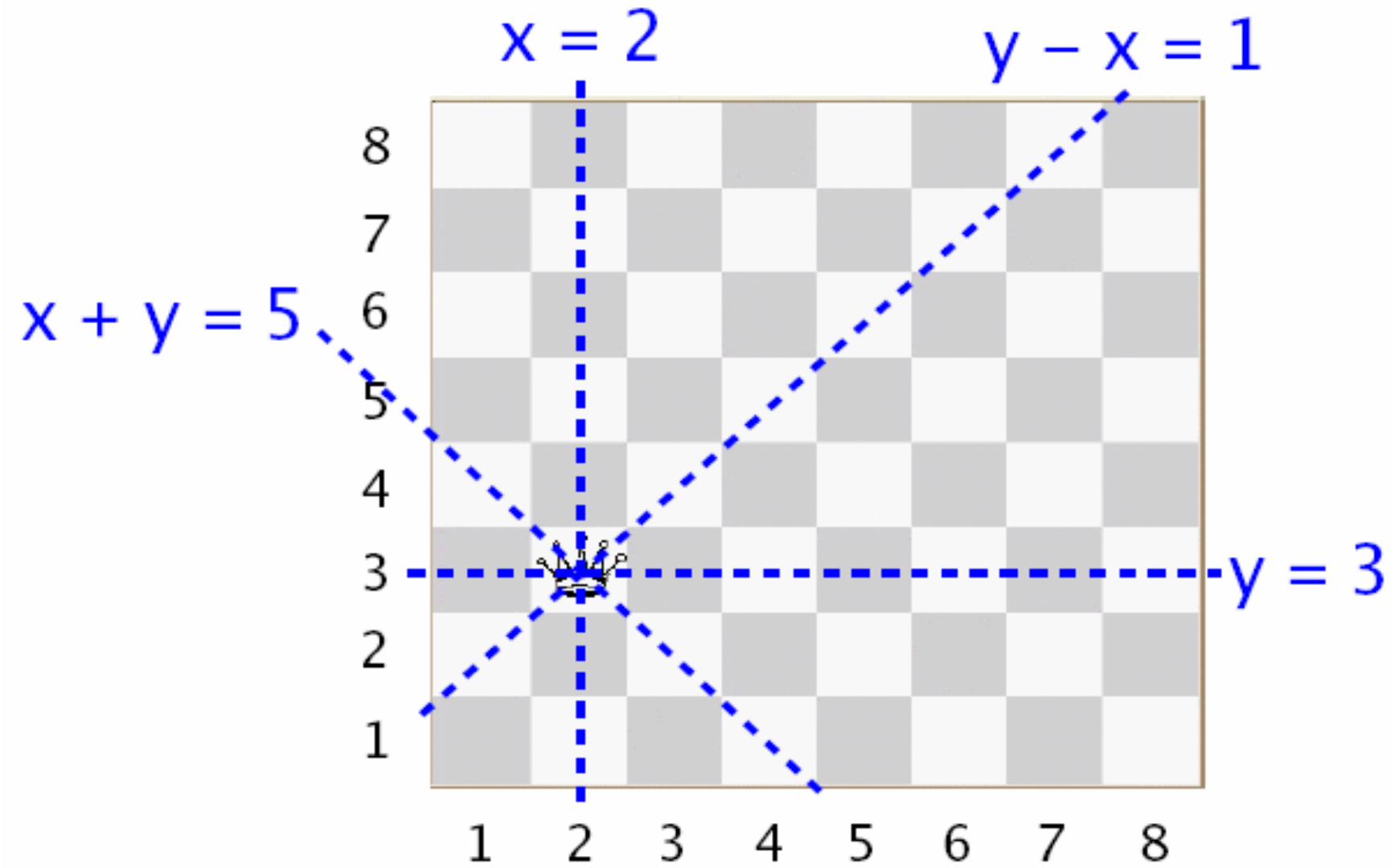
Nếu vị trí thử không bị con hậu nào tấn công

Con hậu thứ 8 là an toàn

Đệ quy để với con hậu tiếp

Xóa để tiếp tục thử vị trí [row+1, column]

Kiểm tra An toàn



Thiết kế dữ liệu

- int pos[] : lưu vị trí của con hậu
  - pos[column] = row ⇔ có con hậu tại vị trí (row, column)
- bool rowFlag[] : lưu trạng thái của các hàng
  - rowFlag[i] = false ⇔ không có con hậu nào chiếm hàng i
- bool rowPlusCol[] : lưu trạng thái của các đường chéo x+y ($2 \le x+y \le 16$)
  - rowPlusCol[x+y] = false ⇔ không có quân hậu nào chiếm đường chéo x+y
- bool rowMinusCol[] : lưu trạng thái của các đường chéo y-x ($-7 \le y-x \le 7$)
  - rowMinusCol[y-x] = false ⇔ không có quân hậu nào chiếm đường chéo y-x

Kiểm tra an toàn của vị trí [row, column]

- Hàng row chưa bị chiếm
  - rowFlag [row] == false ?
- Đường chéo row+column chưa bị chiếm
  - rowPlusCol [row+column] == false ?
- Đường chéo row-column chưa bị chiếm
  - rowMinusCol [row-column] == false ?

Đặt con hậu vào vị trí [row, column]

- Lưu vị trí của con hậu
  - pos[column] = row
- Đánh dấu hàng row đã bị chiếm
  - rowFlag[row] = true
- Đánh dấu đường chéo row+column đã bị chiếm
  - rowPlusCol [row+column] = true
- Đánh dấu đường chéo row-column đã bị chiếm
  - rowMinusCol [row-column] = true

Xóa con hậu khỏi vị trí [row, column]

- Xóa vị trí của con hậu
  - pos[column] = -1
- Đánh dấu lại hàng row chưa bị chiếm
  - rowFlag[row] = false
- Đánh dấu lại đường chéo row+column chưa bị chiếm
  - rowPlusCol [row+column] = false
- Đánh dấu lại đường chéo row-column chưa bị chiếm
  - rowMinusCol [row-column] = false

In kết quả

```
function PrintSolution(int pos[])
{
   for (int col=1; col<=8; col++)
        printf("Con hau thu %d nam tai hang
                                %d", col, pos[col] );
}
```

```
function Try (int column) {
    for (row = 1; row <= 8; row++) {
        if (!rowFlag [row] && !rowPlusCol [row+column] &&
            !rowMinusCol [row-column] ) {
                //Đặt con hậu vào vị trí [row, column]
                pos[column] = row;
                rowFlag[row] = true;
                rowPlusCol [row+column] = true;
                rowMinusCol [row-column] = true;

                if (column == 8)  // con hậu thứ 8 an toàn
                        PrintSolution(pos);
                else
                        Try (column + 1);

                // Xóa con hậu khỏi vị trí [row, column]
                pos[column] = -1;
                rowFlag[row] = false;
                rowPlusCol [row+column] = false;
                rowMinusCol [row-column] = false;
        }
    }
```

Cấu trúc dữ liệu và giải thuật

Đỗ Tuấn Anh

Email: anhdt@it-hut.edu.vn

Nội dung

- Chương 1 – Thiết kế và phân tích (5 tiết)
- Chương 2 – Giải thuật đệ quy (10 tiết)
- Chương 3 – Mảng và danh sách (5 tiết)
- Chương 4 – Ngăn xếp và hàng đợi (10 tiết)
- Chương 5 – Cấu trúc cây (10 tiết)
- Chương 8 – Tìm kiếm (5 tiết)
- Chương 7 – Sắp xếp (10 tiết)
- Chương 6 – Đồ thị (5 tiết)

Chương 3 – Mảng và Danh sách

1. Mảng
2. Danh sách
3. Một số phép toán trên danh sách nối đơn
4. Các dạng khác của danh sách móc nối
5. Sử dụng danh sách móc nối – Ví dụ bài toán cộng đa thức

1. Mảng

- Mảng:
  - Số phần tử cố đinh
  - Kích thước một phần tử cố định
  - Các phần tử mảng phải cùng kiểu
  - Truy cập ngẫu nhiên (theo chỉ số)

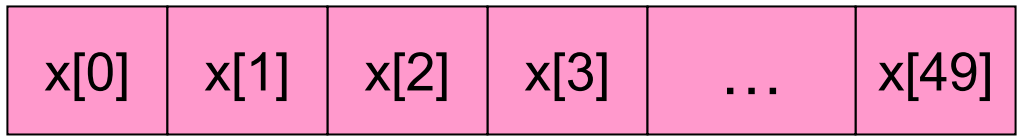
Mảng: Số phần tử cố định

- Kích thước mảng sau khi khai báo là cố định
- Ví dụ:

```
void notAllowed ();
{
  int size;
  int arr[size]; /* không được phép, kích
    thước                    mảng phải là hằng số
    xác định*/
  printf("Enter the size of the array: ");
  scanf("%d", &size);
}
```

Cấu trúc lưu trữ của mảng

double x[50];



addr

addr + 49 * sizeof(double)

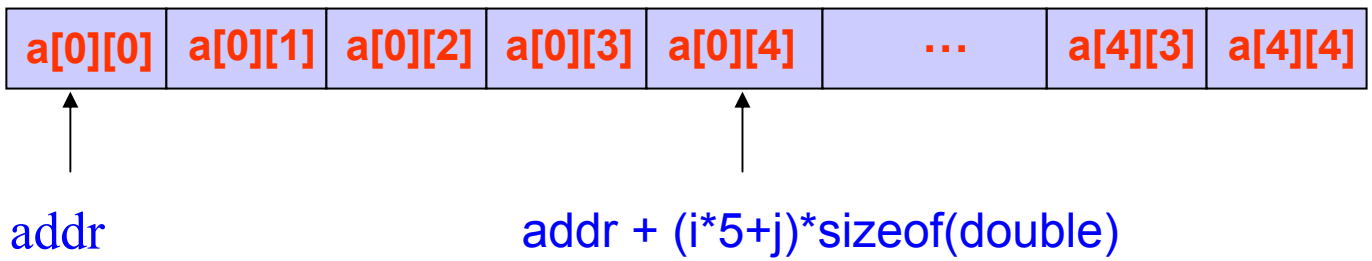
Mảng được lưu trữ kế tiếp => truy cập ngẫu nhiên sử dụng chỉ số => tốc độ truy cập tất cả các phần tử là như nhau

Mảng nhiều chiều

double a[5][5];

a[0]	a[0][0]	a[0][1]	a[0][2]		a[0][4]
a[1]	a[1][0]	a[1][1]			
a[4]	a[4][0]				a[4][4]

- Ma trận (mảng 2 chiều) là một mảng mà mỗi phần tử là một mảng một chiều
- C lưu trữ mảng nhiều chiều theo *thứ tự ưu tiên hàng* – mỗi phần tử là một hàng
- Mảng nhiều chiều vẫn được lưu trữ kế tiếp như mảng một chiều



2. Danh sách

- Danh sách những người đến khám bệnh
  - Ban đầu chưa có ai
  - Có người mới đến
  - Có người khám xong đi về
- (Tạo hình ảnh động tại đây)

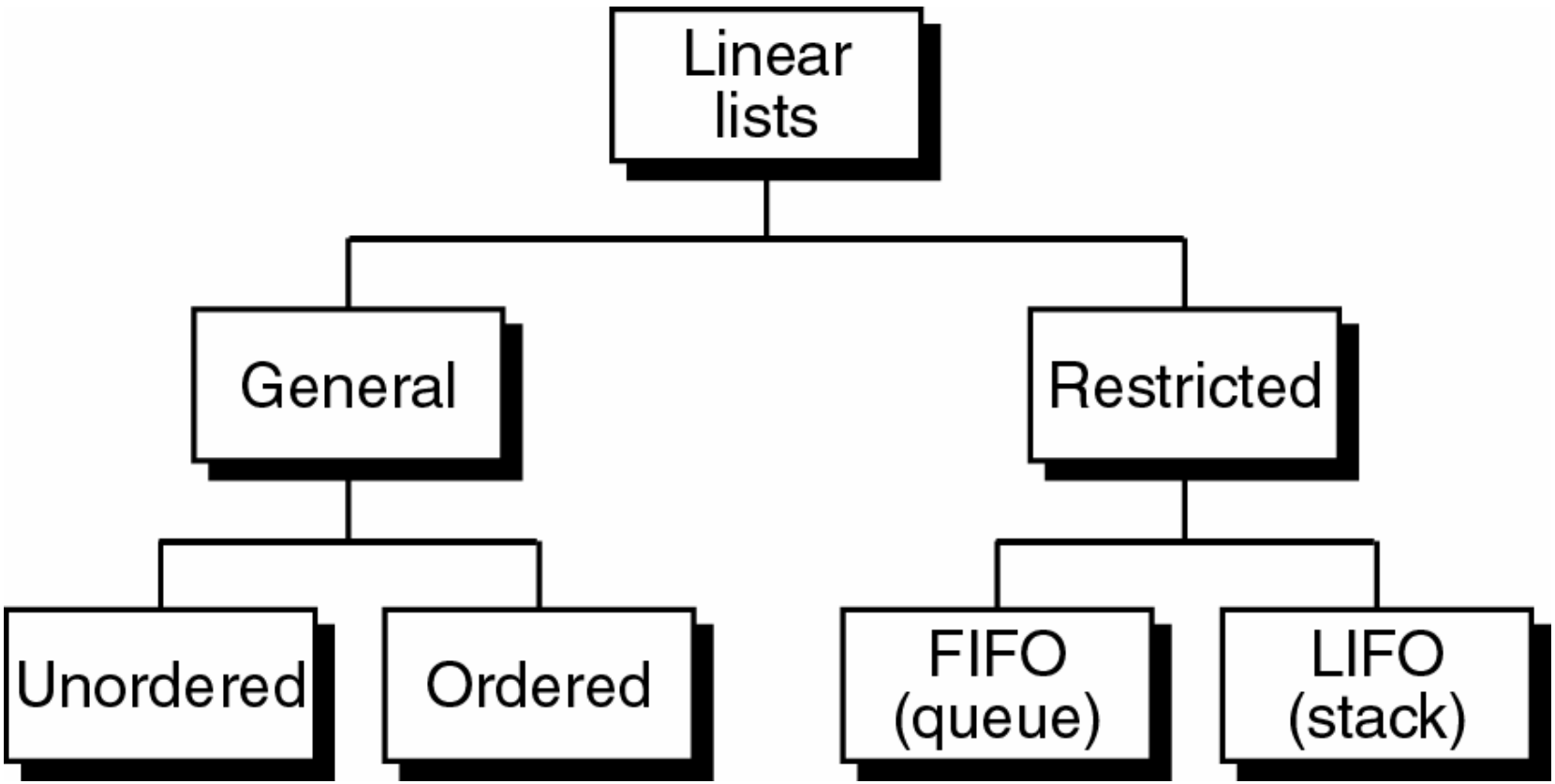
Danh sách tuyến tính

- Một chuỗi các phần tử
- Tồn tại phần tử đầu và phần tử cuối
- Mỗi phần tử có phần tử trước và phần tử sau

Danh sách tuyến tính

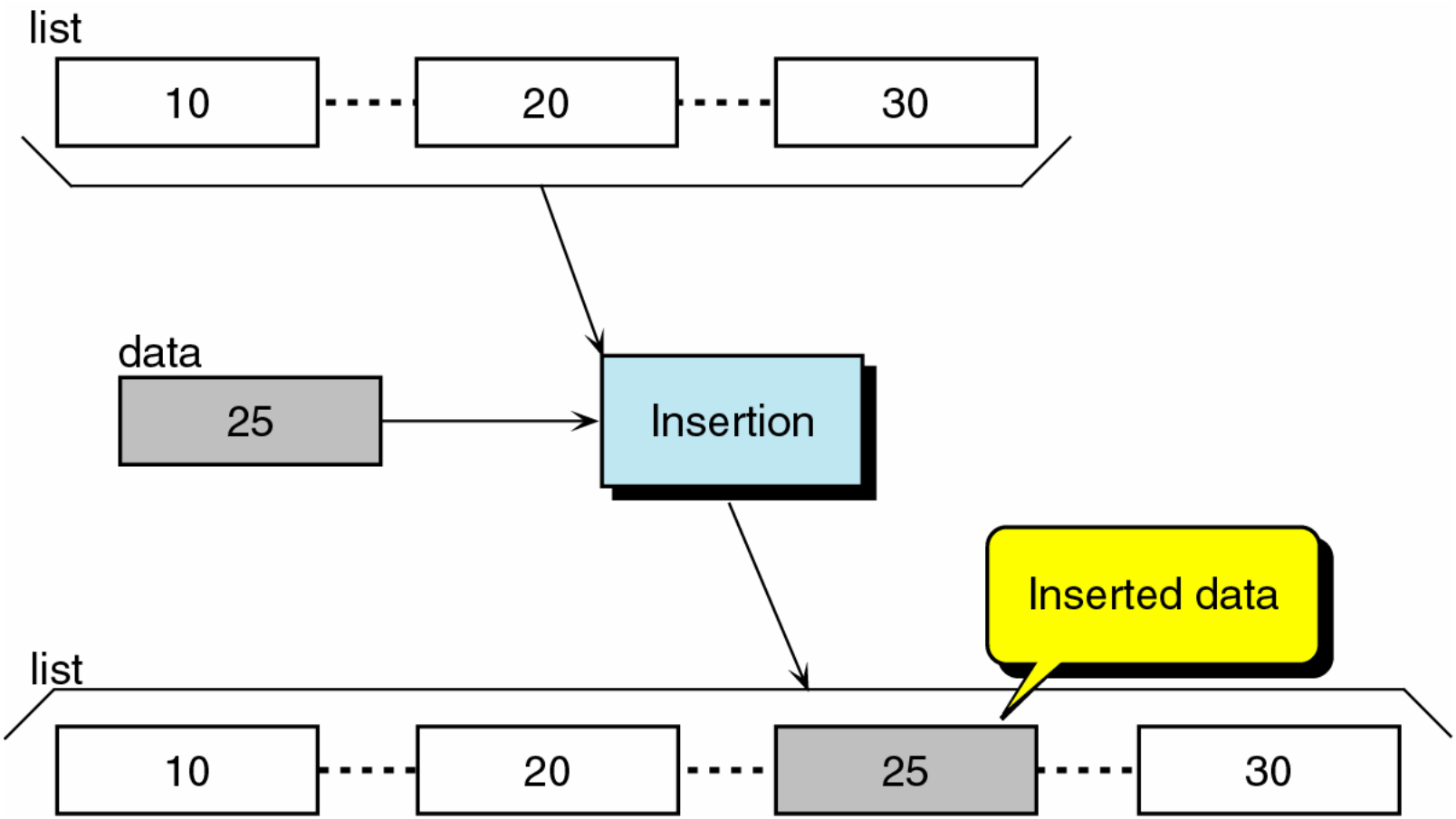
- Số phần tử biến đổi
- Một phần tử thường là một cấu trúc (struct)
- Thao tác thường xuyên nhất
  - Thêm phần tử
  - Xóa phần tử
- Các thao tác khác:
  - Tìm kiếm
  - Ghép 2 danh sách
  - Tách 1 danh sách thành nhiều danh sách
  - Sao chép danh sách
  - Cập nhật

Phân loại danh sách tuyến tính

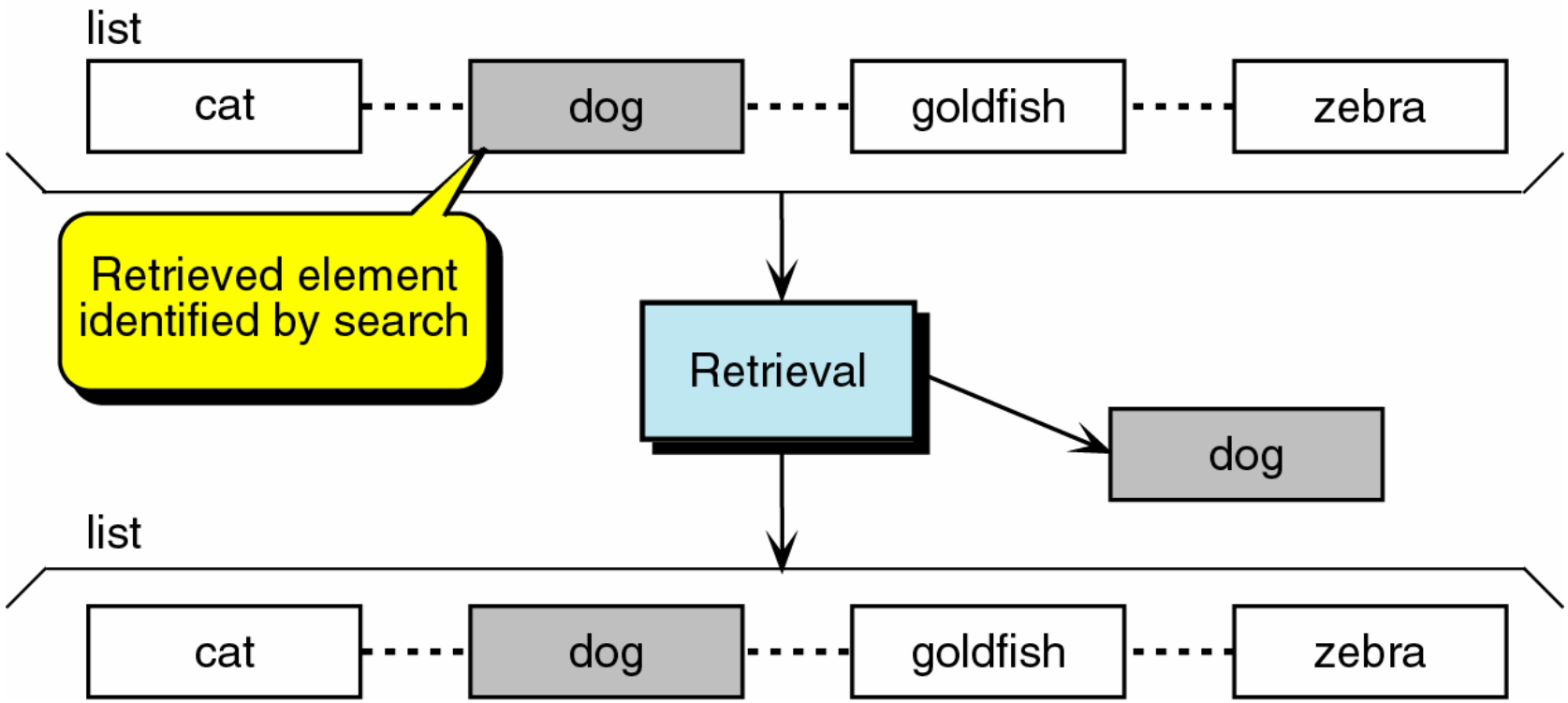


Nguồn: ***Data Structures : A Pseudocode Approach With C***
by Richard F. Gilberg, Behrouz A. Forouzan

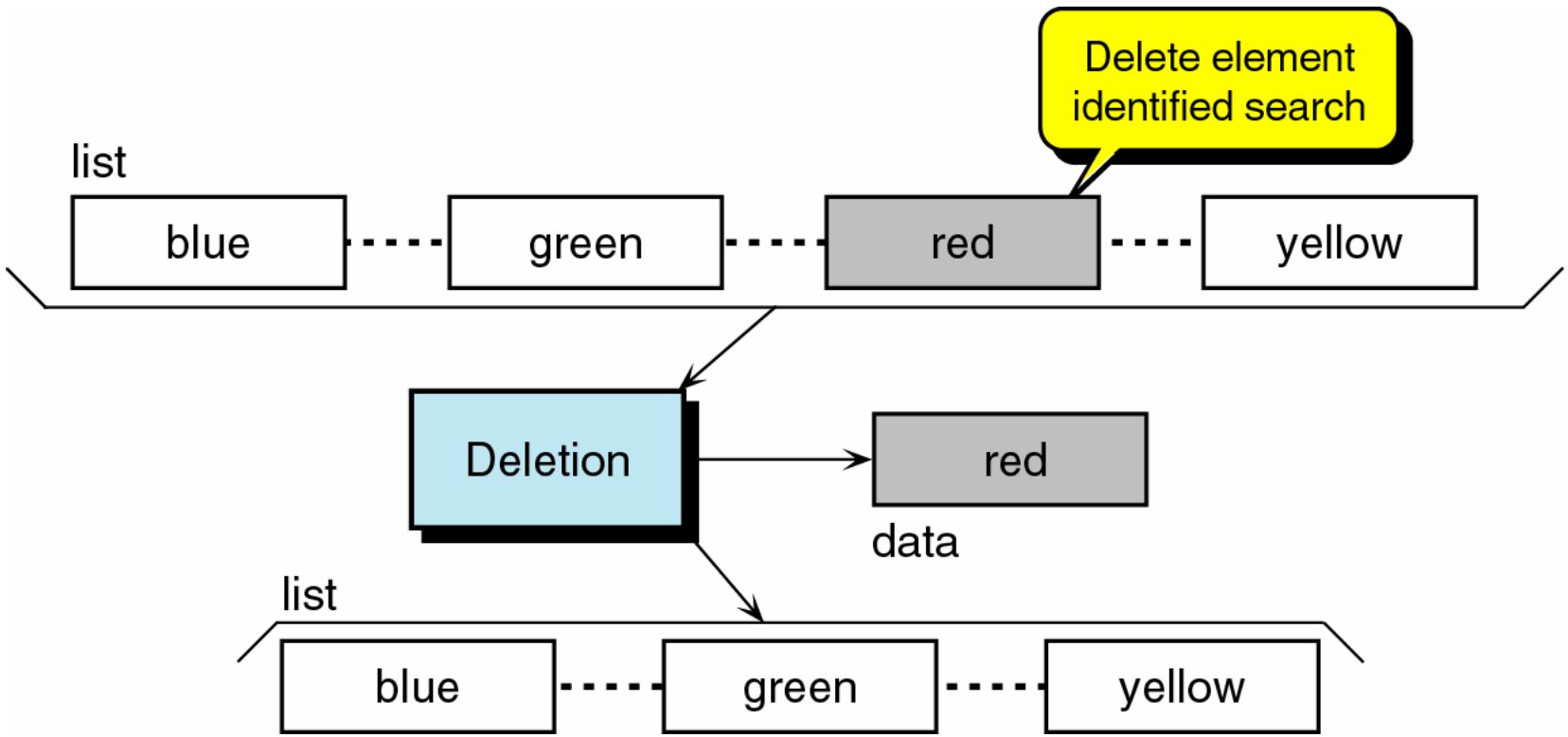
Thêm một phần tử mới



Tìm một phần tử



Xóa một phần tử khỏi danh sách



Lưu trữ danh sách liên kết

1. Lưu trữ kế tiếp sử dụng mảng
2. Lưu trữ móc nối

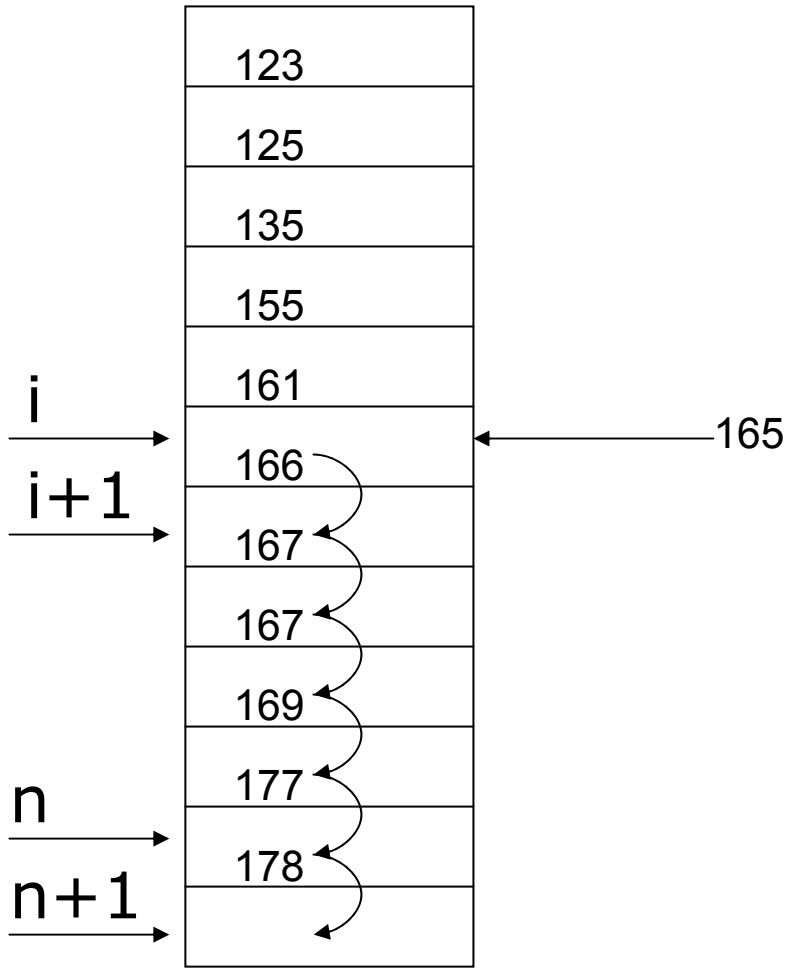
2.1 Danh sách - Lưu trữ kế tiếp

- Sử dụng mảng 1 chiều
- Tìm kiếm dễ dàng (tuần tự hoặc tìm kiếm nhị phân)
- Duyệt các phần tử dễ dàng sử dụng chỉ số:

```
for(i = 0; i <= N; ++i)
    if(a[i]) …
```

- Thêm và xóa KHÔNG dễ dàng
- Danh sách thường xuyên thêm bớt phần tử => Không biết trước số phần tử

Lưu trữ kế tiếp - Thêm một phần tử

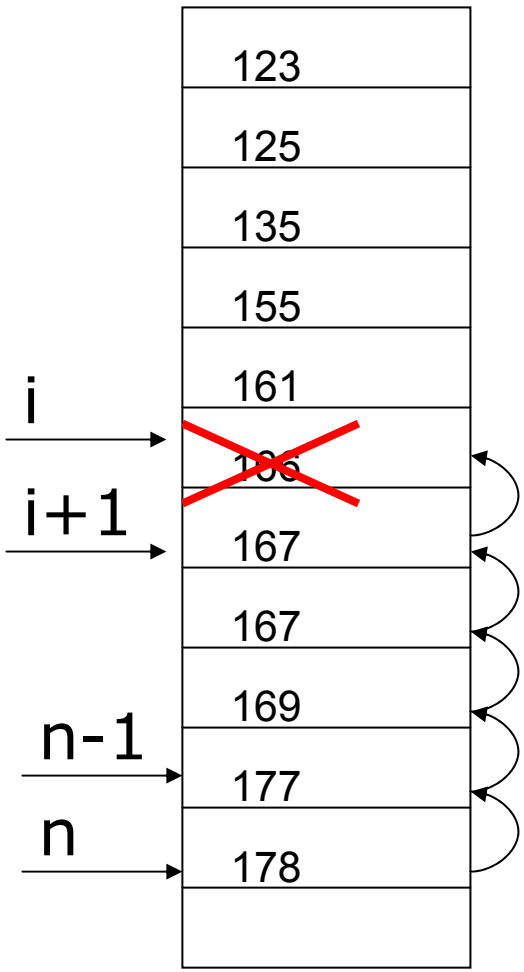


Thêm một phần tử thứ i vào mảng

- Chuyển các phần tử i->n xuống các vị trí i+1 ->n+1

- Thêm phần tử cần thêm vào vị trí thứ i

Lưu trữ kế tiếp - Xóa một phần tử

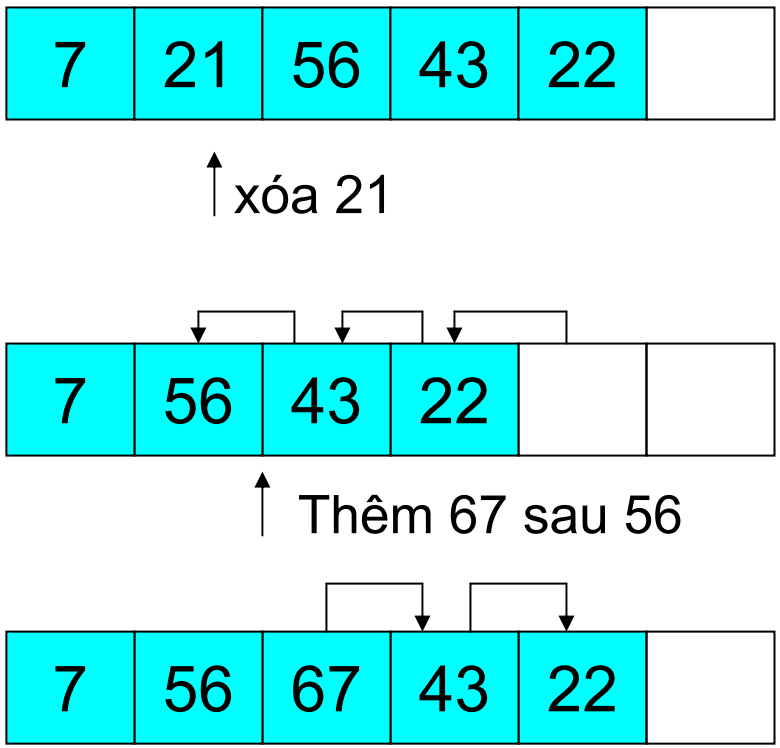


Xóa một phần tử thứ i khỏi mảng

- Chuyển các phần tử i+1->n vào các vị trí i ->n-1

Không hiệu quả

Việc lưu trữ liên tiếp ⇒ thao tác thêm và xóa không hiệu quả (dịch chuyển phần tử).



n/2 lần dịch chuyển (trung bình)

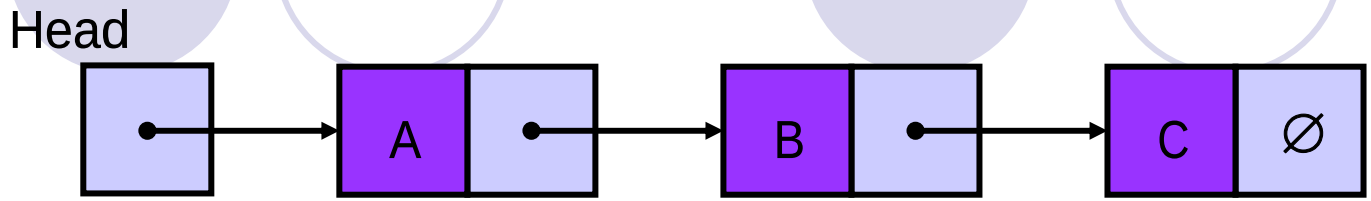
Thời gian tính: O(*n*)

Các thao tác **thêm** và **xóa** có thời gian chạy là *O*(*n*).

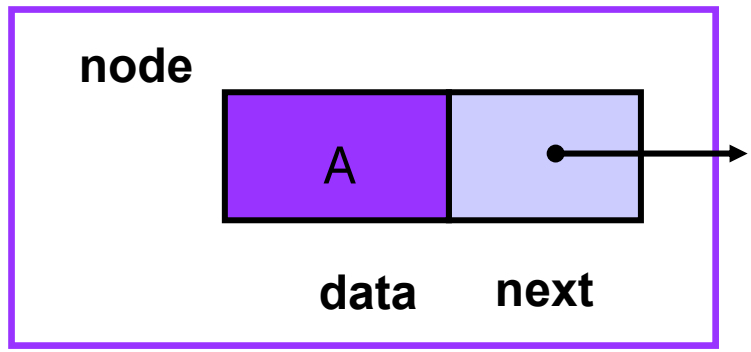
Lưu trữ kế tiếp

- Ưu điểm: truy cập nhanh (ngẫu nhiên, thời gian truy cập mọi phần tử là như nhau)
- Nhược điểm:
  - Việc thêm, bớt phần tử rất khó khăn (phải dịch chuyển nhiều phần tử khác)
  - Tốn bộ nhớ, cấp phát nhiều hơn cần thiết để giữ chỗ

2.2 Danh sách móc nối



- Một *danh sách móc nối* là một chuỗi các phần tử, gọi là *nút,* được móc nối với nhau
- Mỗi nút phải bao gồm
  - Dữ liệu
  - Móc nối (địa chỉ) tới nút tiếp theo trong danh sách

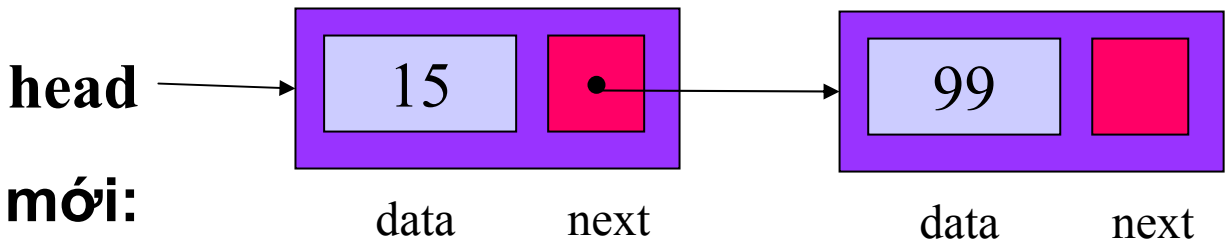


- *Head*: con trỏ trỏ đến nút đầu tiên
- Nút cuối cùng trỏ đến NULL

Tổ chức danh sách móc nối

- **Nút = dữ liệu + móc nối**
  - **Định nghĩa:**

```
typedef struct node {
        int data;
        struct node *next;
}Node;
```



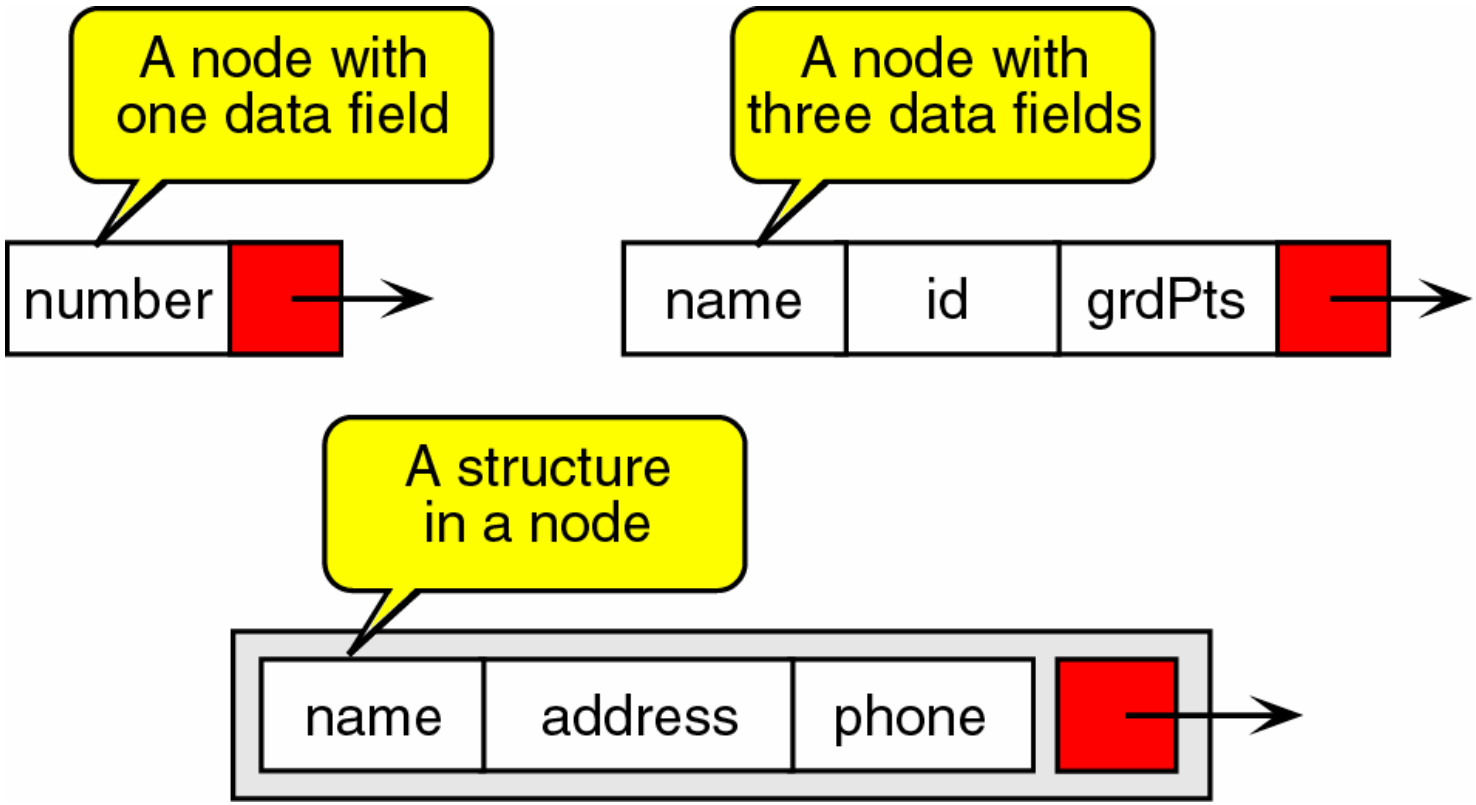
  - **Tạo nút mới:**

```
Node* p =
malloc(sizeof(Node));
```

  - **Giải phóng nút:**

```
free(p);
```

Nút – Phần tử của danh sách



Nguồn: ***Data Structures : A Pseudocode Approach With C***
by Richard F. Gilberg, Behrouz A. Forouzan

Khởi tạo và truy cập danh sách móc nối

- Khai báo một con trỏ

```
Node* Head;
```

Head là con trỏ trỏ đến nút đầu của danh sách.
Khi danh sách rỗng thì Head = NULL.

3. Một số thao tác với danh sách nối đơn

1. Thêm một nút mới tại vị trí cụ thể
2. Tìm nút có giá trị cho trước
3. Xóa một nút có giá trị cho trước
4. Ghép 2 danh sách nối đơn
5. Hủy danh sách nối đơn

Truyền danh sách móc nối vào hàm

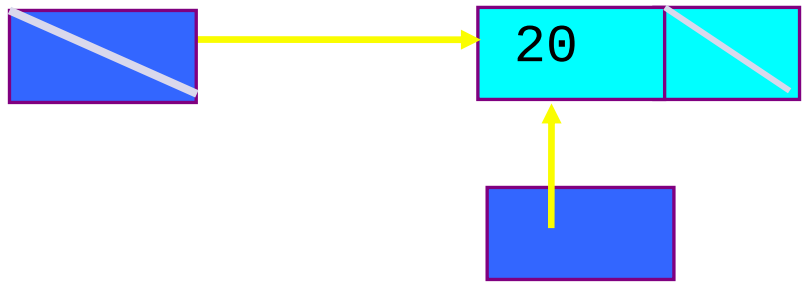
- Khi truyền danh sách móc nối vào hàm, chỉ cần truyền Head.
- Sử dụng Head để truy cập toàn bộ danh sách
- Note: nếu hàm thay đổi vị trí nút đầu của danh sách (thêm hoặc xóa nút đầu) thì Head sẽ không còn trỏ đến đầu danh sách
- Do đó nên truyền Head theo tham biến (hoặc trả lại một con trỏ mới)

Thêm một nút mới

- Các trường hợp của thêm nút
  1. Thêm vào danh sách rỗng
  2. Thêm vào đầu danh sách
  3. Thêm vào cuối danh sách
  4. Thêm vào giữa danh sách
- Thực tế chỉ cần xét 2 trường hợp
  - Thêm vào đầu danh sách (TH1 và TH2)
  - Thêm vào giữa hoặc cuối danh sách (TH3 và TH4

Thêm vào danh sách rỗng

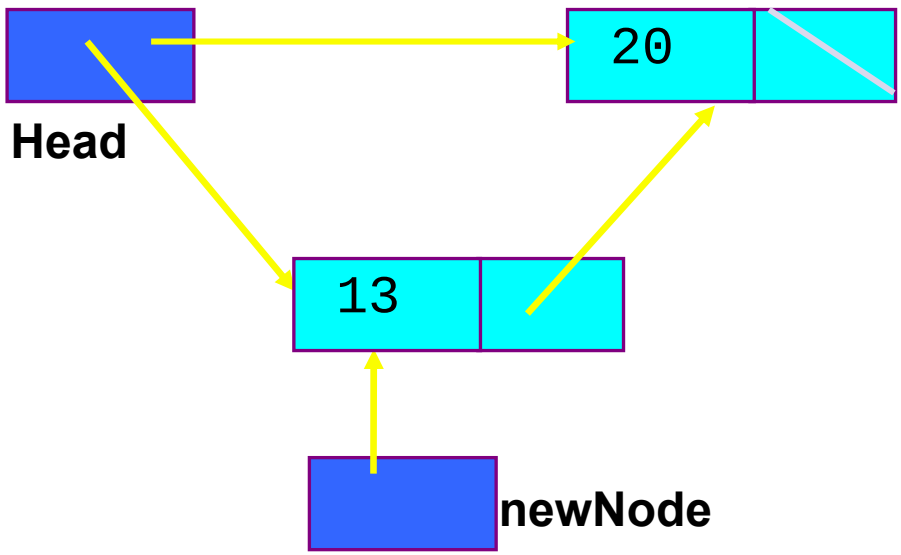
Head = NULL




```
Node* newNode;
newNode =
  malloc(sizeof(Node));
newNode->data = 20;
newNode->next = NULL;
Head = newNode;
```

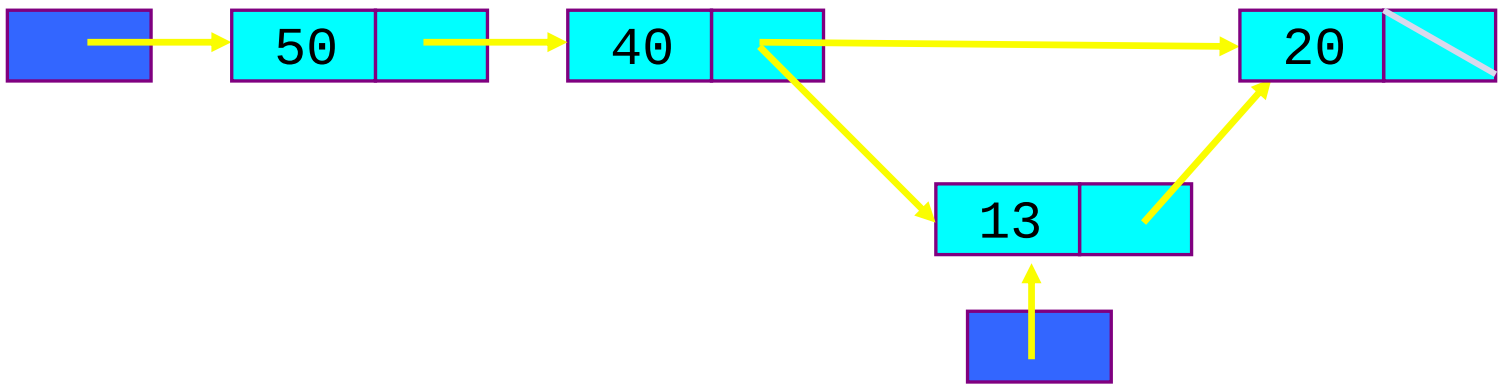
Thêm một nút vào đầu danh sách

```
newNode = malloc(sizeof(Node));
newNode->data = 13;
newNode->next = Head;
Head = newNode;
```



Thêm một nút vào giữa/cuối danh sách

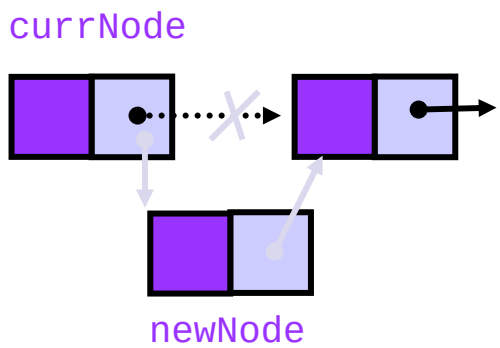
```
newNode = malloc(sizeof(Node));
newNode->data = 13;
newNode->next = currNode->next;
currNode->next = newNode;
```



Thêm một nút mới

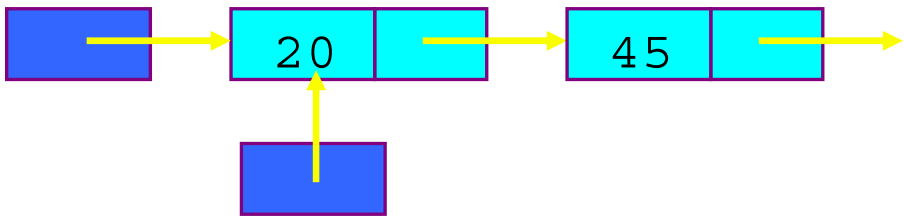
- **`Node* InsertNode(Node* head, int index, int x)`**
  - Thêm một nút mới với dữ liệu là x vào sau nút thứ `index`.
    (ví dụ, khi `index = 0`, nút được thêm là phần tử đầu danh sách; khi `index = 1`, chèn nút mới vào sau nút đầu tiên, v.v)
  - Nếu thao tác thêm thành công, trả lại nút được thêm. Ngược lại, trả lại `NULL`.
    (Nếu `index` < 0 hoặc > độ dài của danh sách, không thêm được.)
- Giải thuật
  1. Tìm nút thứ `index - currNode`
  2. Tạo nút mới
  3. Móc nối nút mới vào danh sách

```
newNode->next = currNode->next;
currNode->next = newNode;
```

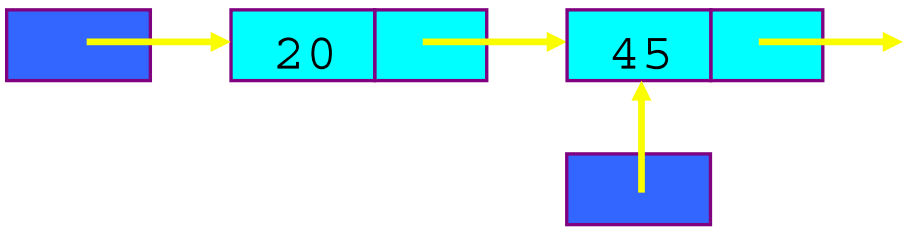


Duyệt danh sách móc nối

```
currNode = Head;
```




```
currNode = currNode->next;
```



Tìm currNode

Thêm một nút mới vào sau nút thứ `index`.

```
int currIndex   =    1;
Node* currNode  =    head;
while (currNode && index > currIndex)
{
   currNode =    currNode->next;
   currIndex++;
}
```

Thêm một nút mới

```
Node* InsertNode(Node* head, int index, int
    if (index < 0) return NULL;

    int currIndex  =       1;
    Node* currNode =       head;
    while (currNode && index > currIndex) {
           currNode        =       currNode->next;
           currIndex++;
    }
    if (index > 0 && currNode == NULL) return NULL;

    Node* newNode  =       (Node*) malloc(sizeof(Node));
    newNode->data  =       x;
    if (index == 0) {
           newNode->next   =       head;
           head            =       newNode;
    }
    else {
           newNode->next   =       currNode->next;
           currNode->next  =       newNode;
    }
    return newNode;
```

Tìm nút thứ `index`. Nếu không tìm thấy, trả lại `NULL`.

Thêm một nút mới

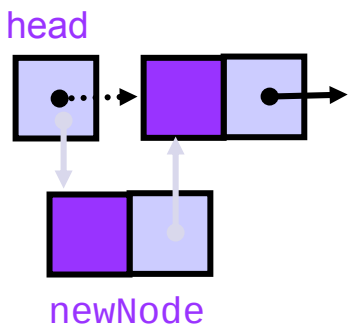
```
Node* InsertNode(Node* head, int index, int x) {
        if (index < 0) return NULL;

        int currIndex  =       1;
        Node* currNode =       head;
        while (currNode && index > currIndex) {
                currNode       =       currNode->next;
                currIndex++;
        }
        if (index > 0 && currNode == NULL) return NULL;

        Node* newNode  =       (Node*)malloc(sizeof(Node));
        newNode->data  =       x;
        if (index == 0) {
                newNode->next  =       head;
                head           =       newNode;
        }
        else {
                newNode->next  =       currNode->next;
                currNode->next =       newNode;
        }
        return newNode;
}
```

Tạo nút mới

Thêm một nút mới

```
Node* InsertNode(Node* head, int index, int x) {
        if (index < 0) return NULL;

        int currIndex  =        1;
        Node* currNode =        head;
        while (currNode && index > currIndex) {
                currNode        =       currNode->next;
                currIndex++;
        }
        if (index > 0 && currNode == NULL) return NULL;

        Node* newNode  =        (Node*)malloc
        newNode->data  =        x;
        if (index == 0) {
                newNode->next   =       head;
                head            =       newNode;
        }
        else {
                newNode->next   =       currNode->next;
                currNode->next =        newNode;
        }
        return newNode;
```

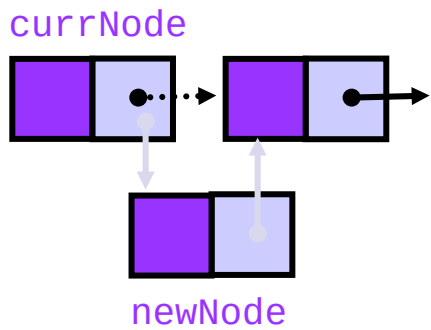
Thêm vào đầu danh sách



Thêm một nút mới

```
Node* InsertNode(Node* head, int index, int x) {
        if (index < 0) return NULL;

        int currIndex  =       1;
        Node* currNode =       head;
        while (currNode && index > currIndex) {
                currNode       =       currNode->next;
                currIndex++;
        }
        if (index > 0 && currNode == NULL) return NULL;

        Node* newNode  = (Node*)malloc(sizeof(Node));
        newNode->data  =       x;
        if (index == 0) {
                newNode->next  =       head;
                head           =       newNode;
        }
        else {
                newNode->next  =       currNode->next;
                currNode->next =       newNode;
        }
        return newNode;
}
```

Thêm vào sau currNode



Tìm nút

- `int FindNode(int x)`
  - Tìm nút có giá trị x trong danh sách.
  - Nếu tìm được trả lại vị trí của nút. Nếu không, trả lại 0.

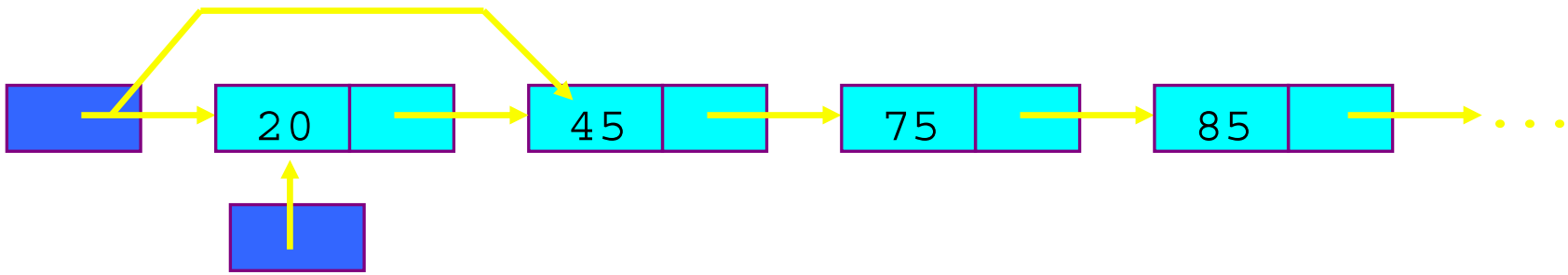
```
int FindNode(Node* head, int x) {
      Node* currNode    =     head;
      int currIndex     =     1;
      while (currNode && currNode->data != x) {
            currNode    =     currNode->next;
            currIndex++;
      }
      if (currNode) return currIndex;
      return 0;
}
```

Xóa nút

- `int DeleteNode(int x)`
  - Xóa nút có giá trị bằng x trong danh sách.
  - Nếu tìm thấy nút, trả lại vị trí của nó. Nếu không, trả lại 0.
- Giải thuật
  - Tìm nút có giá trị x (tương tự như `FindNode`)
  - Thiết lập nút trước của nút cần xóa nối đến nút sau của nút cần xóa
  - Giải phóng bộ nhớ cấp phát cho nút cần xóa
- Giống như `InsertNode`, có 2 trường hợp
  - Nút cần xóa là nút đầu tiên của danh sách
  - Nút cần xóa nằm ở giữa hoặc cuối danh sách

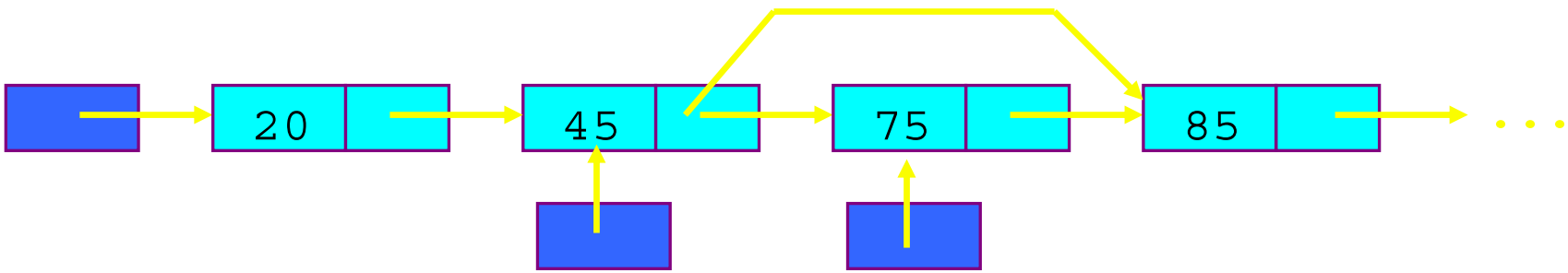
Xóa nút đầu danh sách

```
head = currNode->next;
free(currNode);
```



Xóa nút giữa/cuối danh sách

```
prevNode->next = currNode->next;
free(currNode);
```



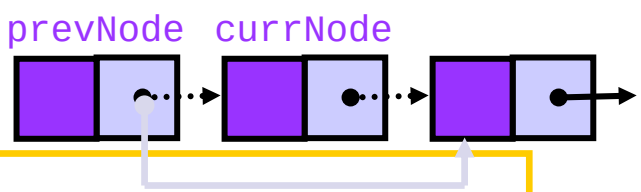
Xóa một nút

Tìm nút có giá trị bằng x

```
int DeleteNode(Node*& head, int x) {
        Node* prevNode =        NULL;
        Node* currNode =        head;
        int currIndex  =        1;
        while (currNode && currNode->data != x) {
                prevNode        =       currNode;
                currNode        =       currNode->next;
                currIndex++;
        }
        if (currNode) {
                if (prevNode) {
                        prevNode->next =        currNode->next;
                        free (currNode);
                }
                else {
                        head            =       currNode->next;
                        free (currNode);
                }
                return currIndex;
        }
        return 0;
```

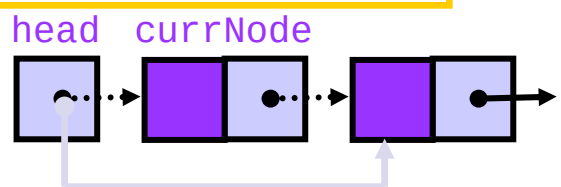
Xóa một nút

```
int DeleteNode(Node* head, int x) {
        Node* prevNode =        NULL;
        Node* currNode =        head;
        int currIndex  =        1;
        while (currNode && currNode->data != x) {
                prevNode        =       currNode;
                currNode        =       currNode->next;
                currIndex++;
        }
        if (currNode) {
                if (prevNode) {
                        prevNode->next =        currNode->next;
                        free (currNode);
                }
                else {
                        head            =       currNode->next;
                        free (currNode);
                }
                return currIndex;
        }
        return 0;
```



Xóa một nút

```
int DeleteNode(Node* head, int x) {
        Node* prevNode =        NULL;
        Node* currNode =        head;
        int currIndex  =        1;
        while (currNode && currNode->data != x) {
                prevNode        =       currNode;
                currNode        =       currNode->next;
                currIndex++;
        }
        if (currNode) {
                if (prevNode) {
                        prevNode->next =        currNode->next;
                        free (currNode);
                }
                else {
                        head            =       currNode->next;
                        free (currNode);
                }
                return currIndex;
        }
        return 0;
```



Hủy danh sách

- `void DestroyList(Node* head)`
  - Sử dụng hàm hủy để giải phóng bộ nhớ được cấp phát cho danh sách.
  - Duyệt toàn bộ danh sách và xóa lần lượt từng nút.

```
void DestroyList(Node* head)
{
   Node* currNode = head, *nextNode = NULL;
   while (currNode != NULL)
   {
      nextNode    =     currNode->next;
      // giải phóng nút vừa duyệt
      free (currNode);
      currNode    =     nextNode;
   }
}
```

In toàn bộ danh sách

- **`void DisplayList(Node* head)`**
  - In dữ liệu của tất cả các phần tử

```
void DisplayList(Node* head)
{
    int num         =     0;
    Node* currNode =      head;
    while (currNode != NULL){
       printf("%d \n", currNode->data);
       currNode    =      currNode->next;
       num++;
    }
}
```

Sử dụng danh sách

kết quả

6
7
5

6
5

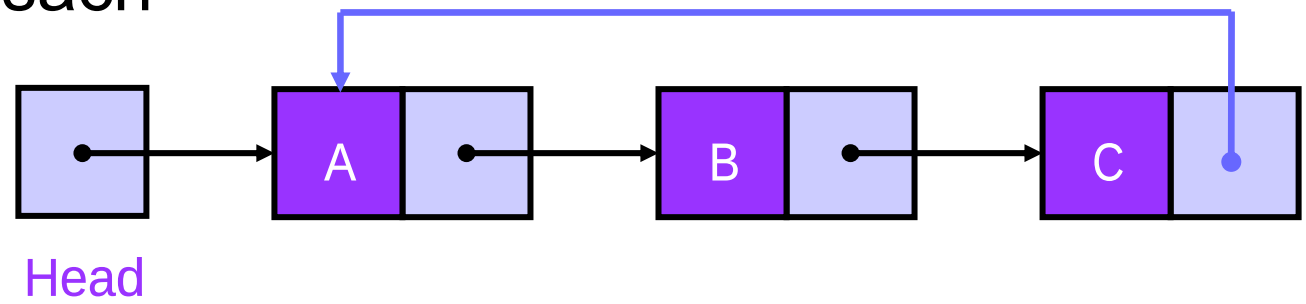
```
int main(void)
{
        Node* head = NULL;
        InsertNode(head, 0, 7);     // thêm vào đầu danh sách
        InsertNode(head, 1, 5);     // thêm vào sau phần tử đầu
        InsertNode(head, -1, 5);    // không thêm được
        InsertNode(head, 0, 6);     // thêm vào đầu danh sách
        InsertNode(head, 8, 4);     // không thêm được

        DisplayList(head);          // in danh sách
        DeleteNode(head, 7);        // xóa nút có giá trị = 7
        DisplayList(head);          // in danh sách
        DestroyList(head);          // hủy toàn bộ danh sách
        return 0;
}
```

So sánh mảng và danh sách liên kết

- Việc lập trình và quản lý danh sách liên kết khó hơn mảng, nhưng nó có những ưu điểm:
  - **Linh động**: danh sách liên kết có kích thước tăng hoặc giảm rất linh động.
    - Không cần biết trước có bao nhiêu nút trong danh sách. Tạo nút mới khi cần.
    - Ngược lại, kích thước của mảng là cố định tại thời gian biên dịch chương trình.
  - **Thao tác thêm và xóa dễ dàng**
    - Để thêm và xóa một phần tử mảng, cần phải copy dịch chuyển phần tử.
    - Với danh sách móc nối, không cần dịch chuyển mà chỉ cần thay đổi các móc nối

Các dạng khác của DSLK

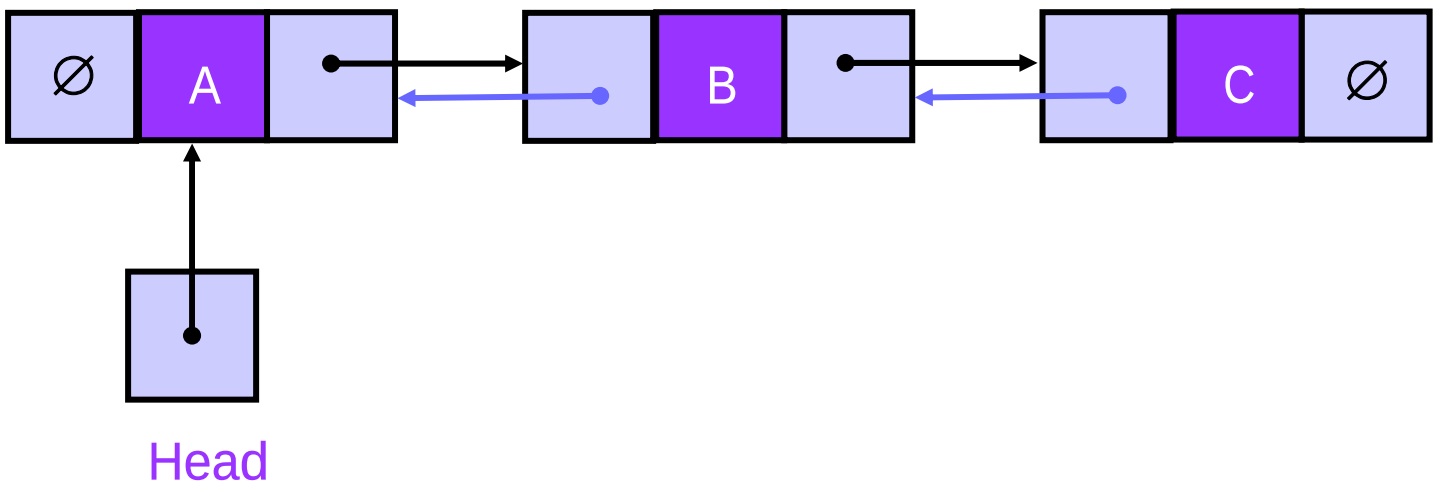
- *Danh sách nối vòng*
  - Nút cuối cùng nối đến nút đầu tiên của danh sách



  - Khi nào thì kết thúc duyệt? (kiểm tra currNode == head?)

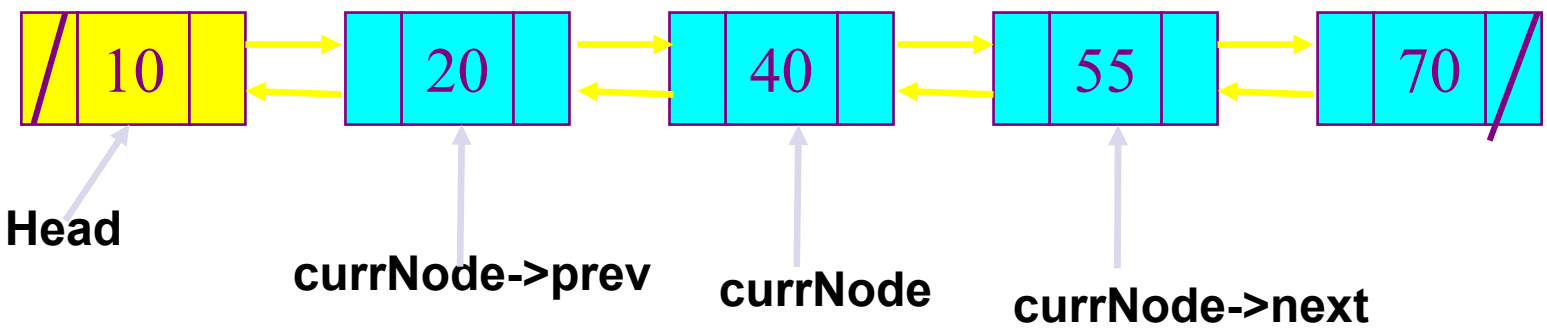
Các dạng khác của DSLK

- *Danh sách nối kép*
  - Mỗi nút không chỉ nối đến nút tiếp theo mà còn nối đến nút trước nó
  - Có 2 mối nối NULL: tại nút đầu và nút cuối của danh sách
  - Ưu điểm: tại một nút có thể thăm nút trước nó một cách dễ dàng. Cho phép duyệt danh sách theo chiều ngược lại



Danh sách nối kép

- Mỗi nút có 2 mối nối
  - prev nối đến phần tử trước
  - next nối đến phần tử sau



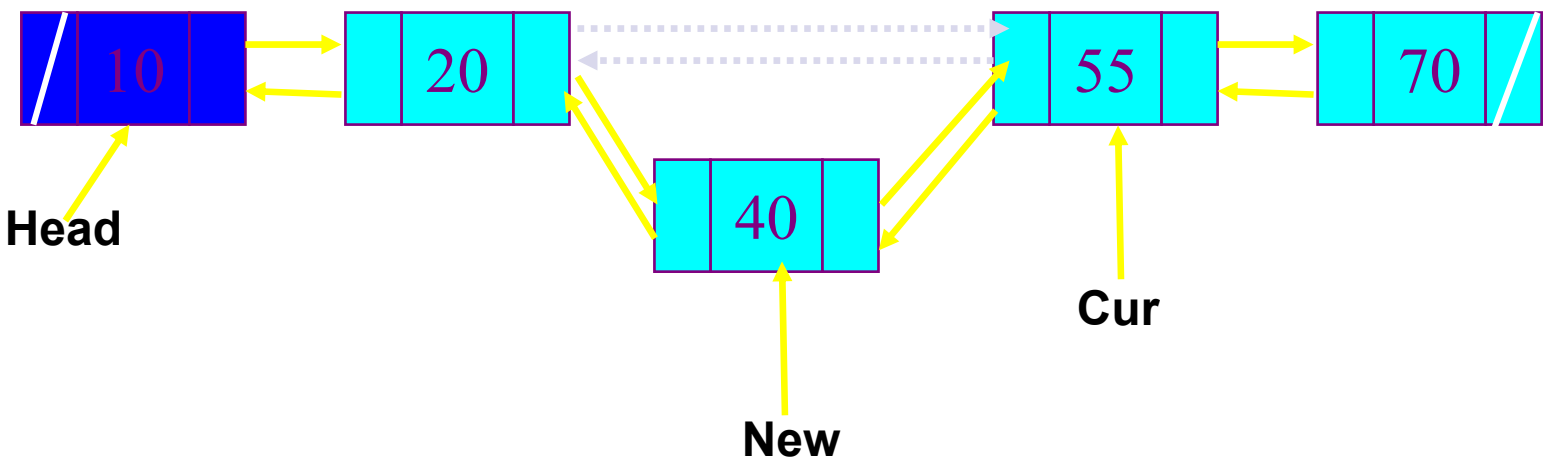
Định nghĩa danh sách nối kép

```
typedef struct Node{
        int data;
        struct Node* next;
        struct Node* prev;
}Node;
```

Thêm nút

- Thêm nút New nằm ngay trước Cur (không phải nút đầu hoặc cuối danh sách)

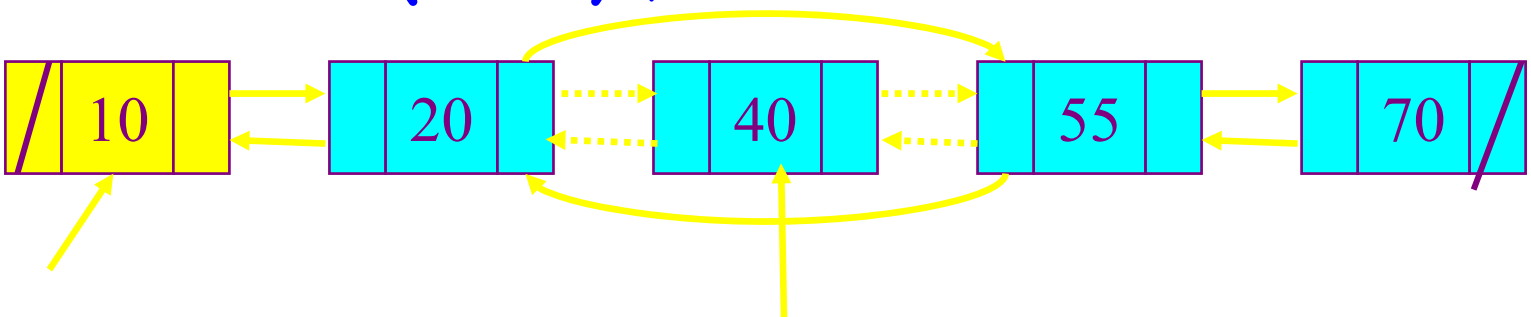
```
New->next = Cur;

New->prev = Cur->prev;

Cur->prev = New;

(New->prev)->next = New;
```



Xóa nút

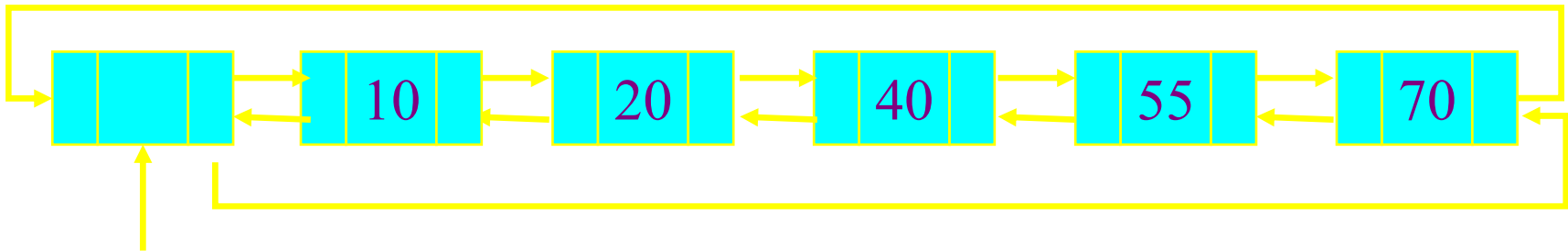
- Xóa nút `Cur` (không phải nút đầu hoặc cuối danh sách)

```
(Cur->prev)->next = Cur->next;
(Cur->next)->prev = Cur->prev;
    free (Cur);
```

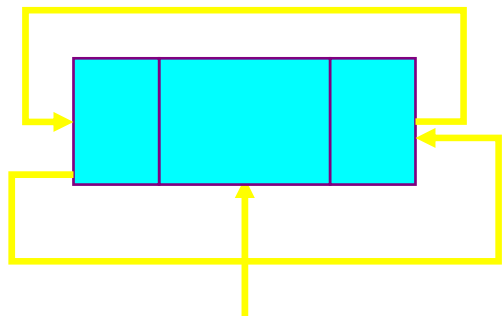


Danh sách nối kép với nút đầu giả

- Danh sách không rỗng

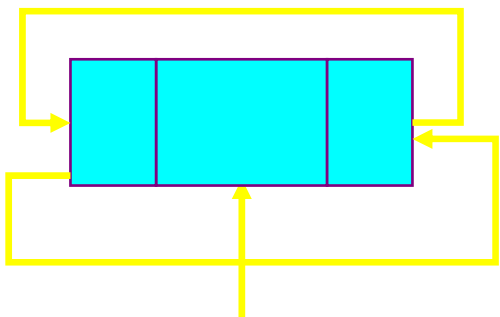


- Danh sách rỗng



Tạo danh sách nối kép rỗng

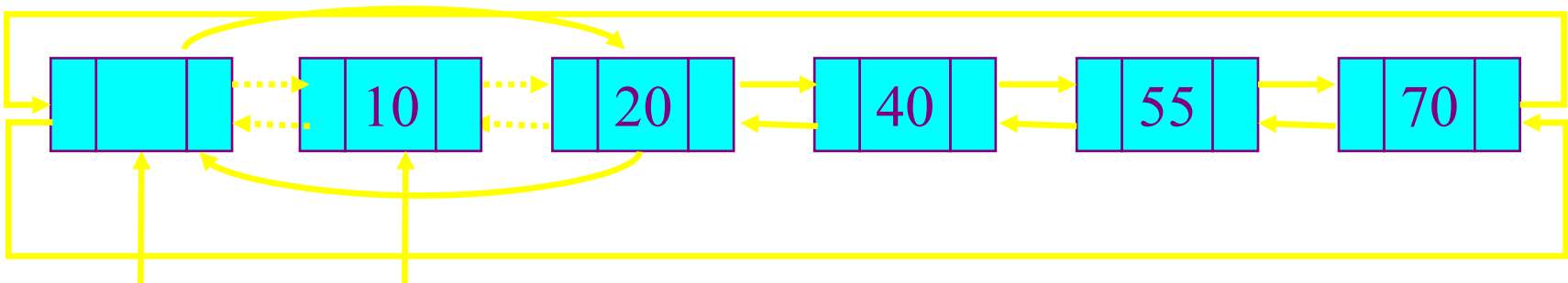
```
Node* Head = malloc
(sizeof(Node));
Head->next = Head;
Head->prev = Head;
```



Xóa nút

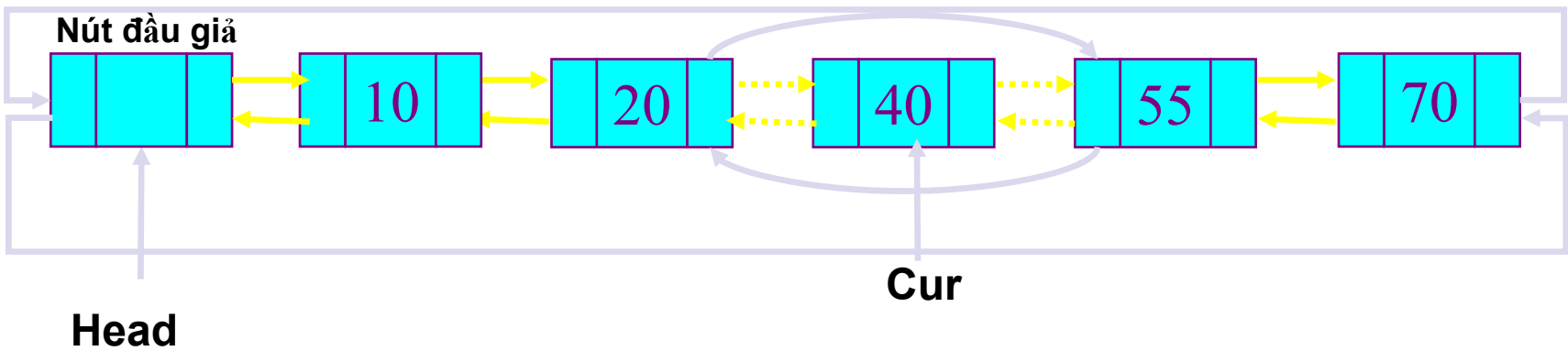
- Nút Cur cần xóa nằm tại đầu danh sách

```
(Cur->prev)->next = Cur->next;

(Cur->next)->prev = Cur->prev;

free (Cur);
```



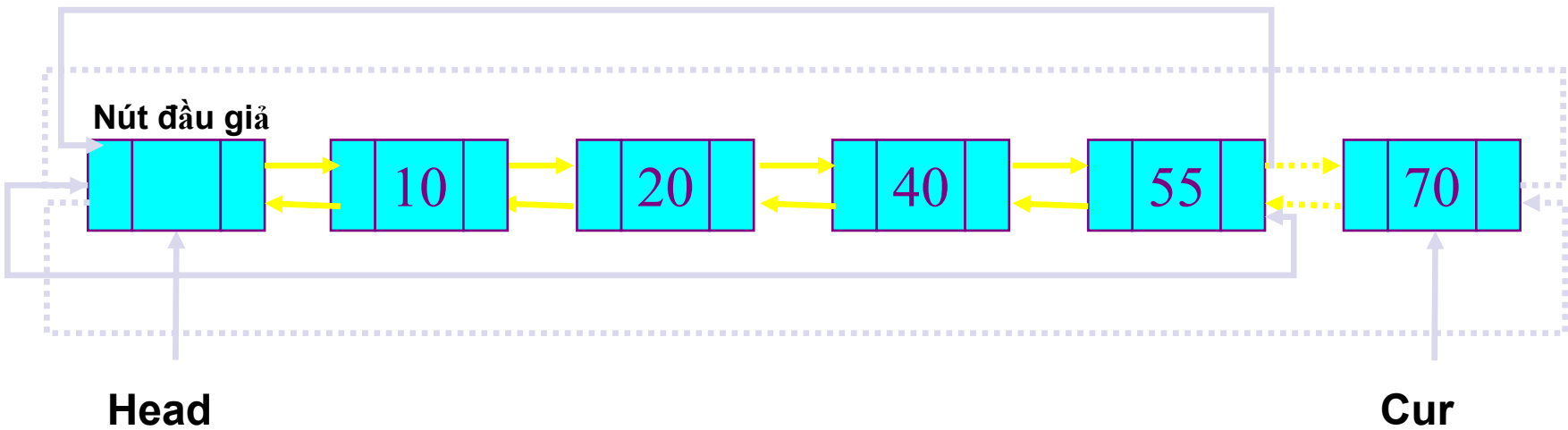
- Nút cần xóa nằm ở giữa danh sách

```
(Cur->prev)->next = Cur->next;

(Cur->next)->prev = Cur->prev;

free (Cur); // giống như xóa ở đầu DS!
```



- Nút cần xóa nằm tại cuối danh sách

```
(Cur->prev)->next = Cur->next;

(Cur->next)->prev = Cur->prev;

 free (Cur); // tương tự như xóa ở giữa DS!
```

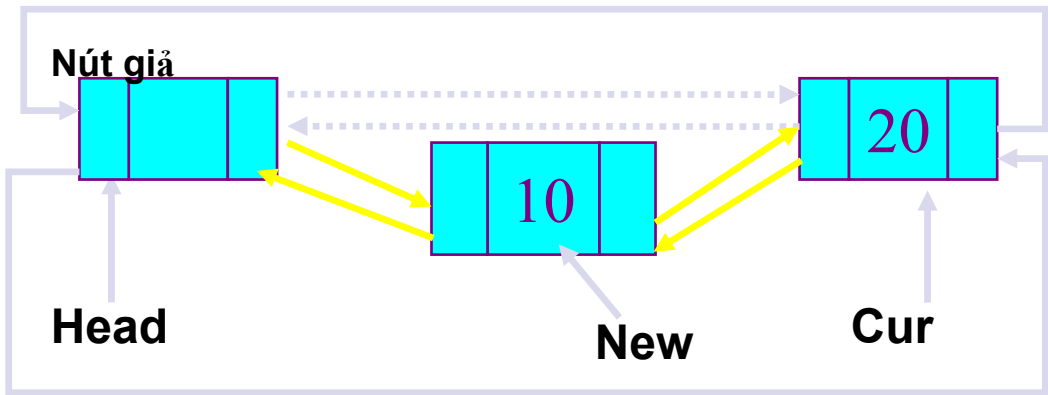


```
void deleteNode(Node* Head, int
  x){
  Node* Cur;
  Cur = FindNode(Head, x);
  if (Cur != NULL){
     Cur->prev->next = Cur->next;
     Cur->next->prev = Cur->prev;
     free (Cur);
  }
}
```

Thêm nút

- Thêm nút New vào sau nút giả (đầu danh sách) và trước nút Cur

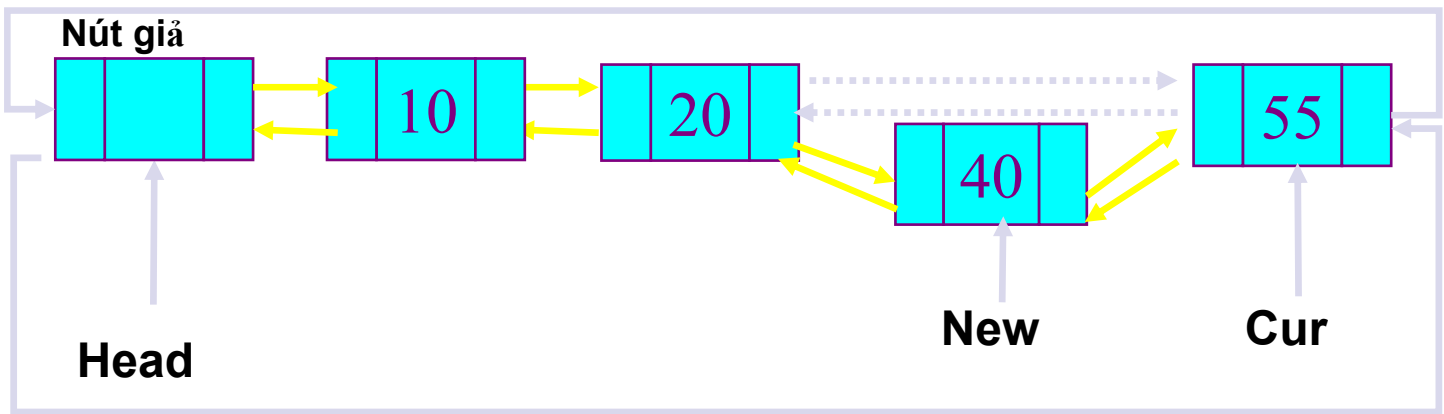
```
New->next = Cur;

New->prev = Cur->prev;

Cur->prev = New;

(New->prev)->next = New;
```



Thêm vào giữa DS

- Thêm nút New vào trước nút Cur

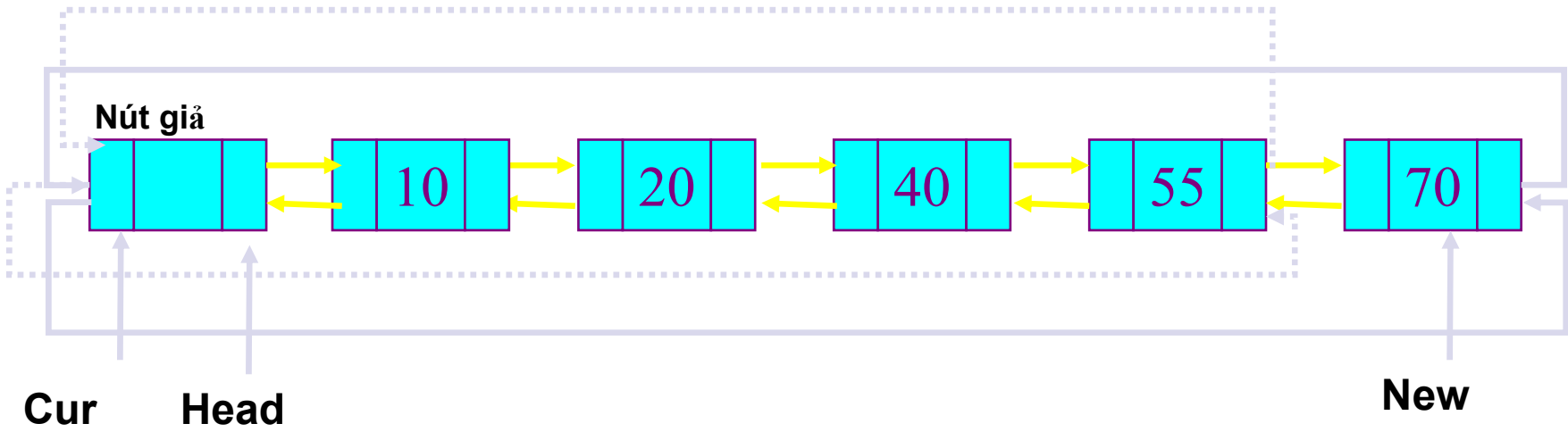
```
New->next = Cur;

New->prev = Cur->prev;

Cur->prev = New;

(New->prev)->next = New; // giống như thêm vào đầu!
```



Thêm vào cuối DS

- Thêm nút New vào cuối DS (lúc này Cur trỏ vào nút giả)

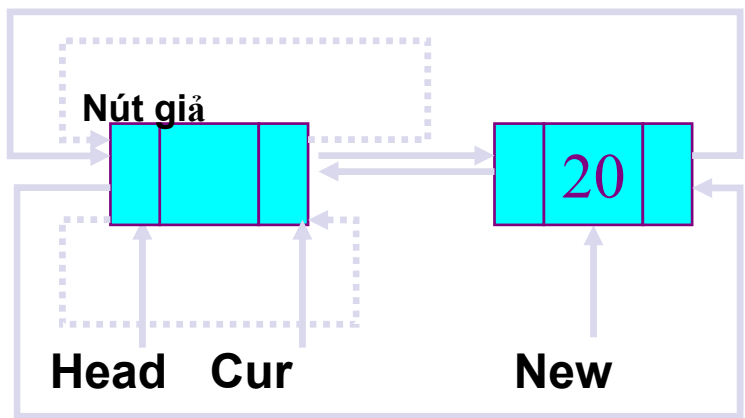
```
New->next = Cur;

New->prev = Cur->prev;

Cur->prev = New;

(New->prev)->next = New; // giống như thêm vào đầu
```



Thêm vào DS rỗng

- Thêm New vào danh sách rỗng (Cur trỏ vào nút giả)

```
New->next = Cur;

New->prev = Cur->prev;

Cur->prev = New;

(New->prev)->next = New;
```



```
void insertNode(Node* Head, int item){
   Node *New, *Cur;
   New = malloc (sizeof(Node));
   New->data = item;
   Cur = Head->next;
   while (Cur != Head){
       if (Cur->data < item)
             Cur = Cur->next;
       else
             break;
   }
   New->next = Cur;
   New->prev = Cur->prev;
   Cur->prev = New;
   (New->prev)->next = New;
}
```

5. Sử dụng danh sách móc nối

- Bài toán cộng 2 đa thức:

$$5x^4 + 6x^3 + 7$$
$$+ \quad 2x^3 - 7x^2 + 3x$$
$$= \quad 5x^4 + 8x^3 - 7x^2 + 3x + 7$$

- Mỗi nút của danh sách:

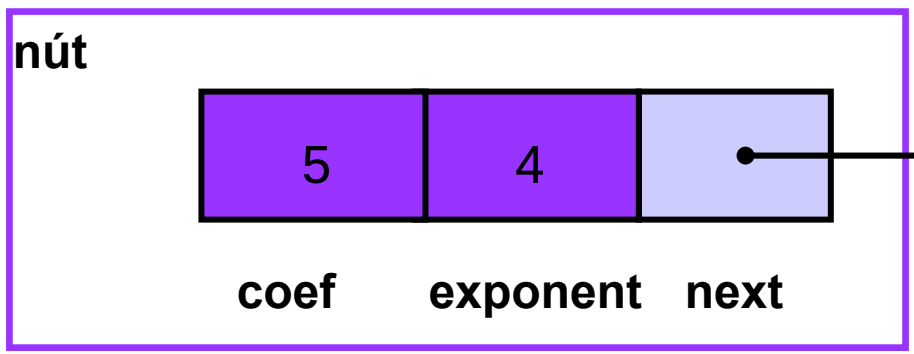
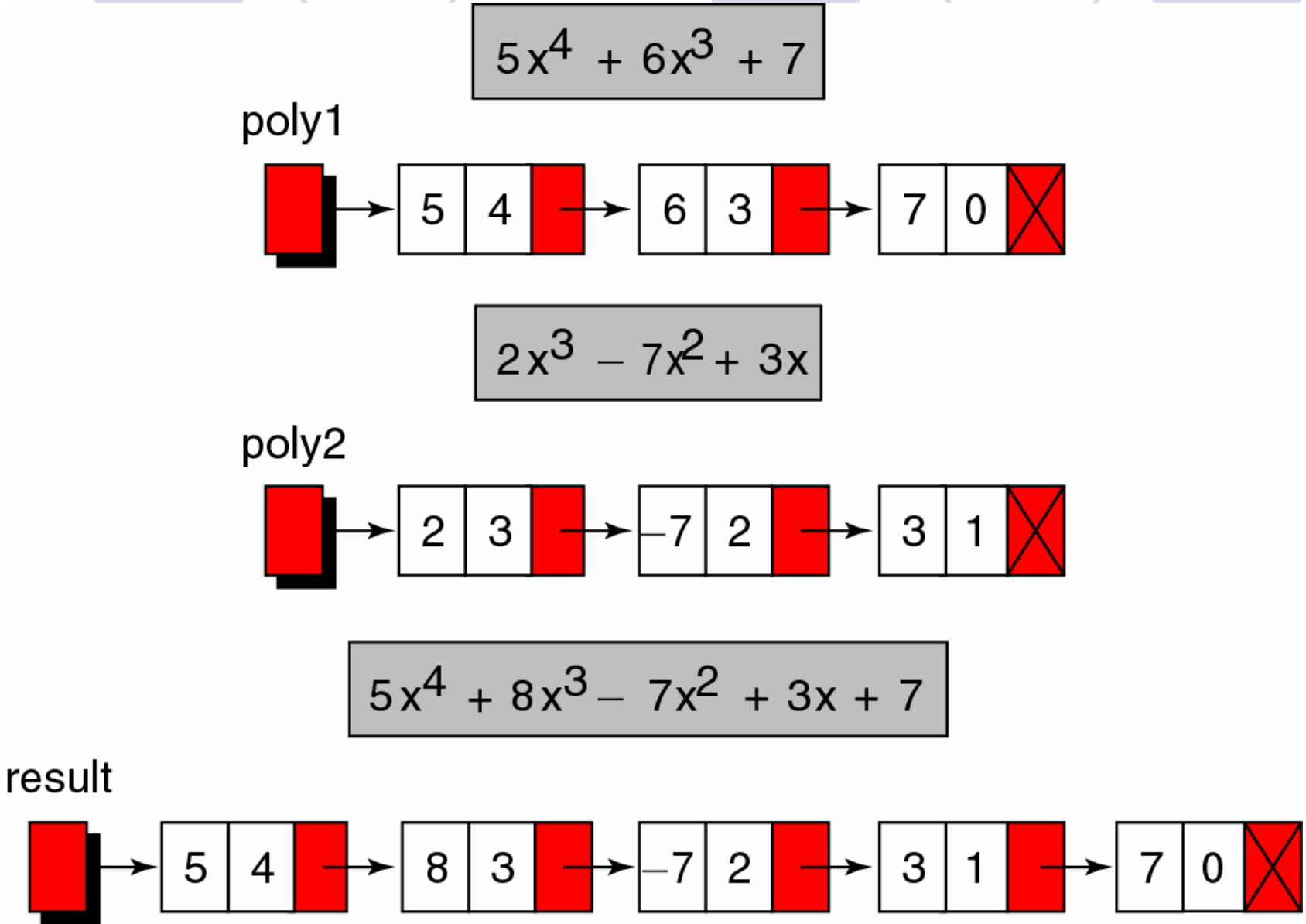


Figure 3-38

Biểu diễn đa thức



```
typedef struct poly{
        float hs;
        float sm;
        struct poly *nextNode;
}
```

Cấu trúc dữ liệu và giải thuật

Đỗ Tuấn Anh

anhdt@it-hut.edu.vn

Nội dung

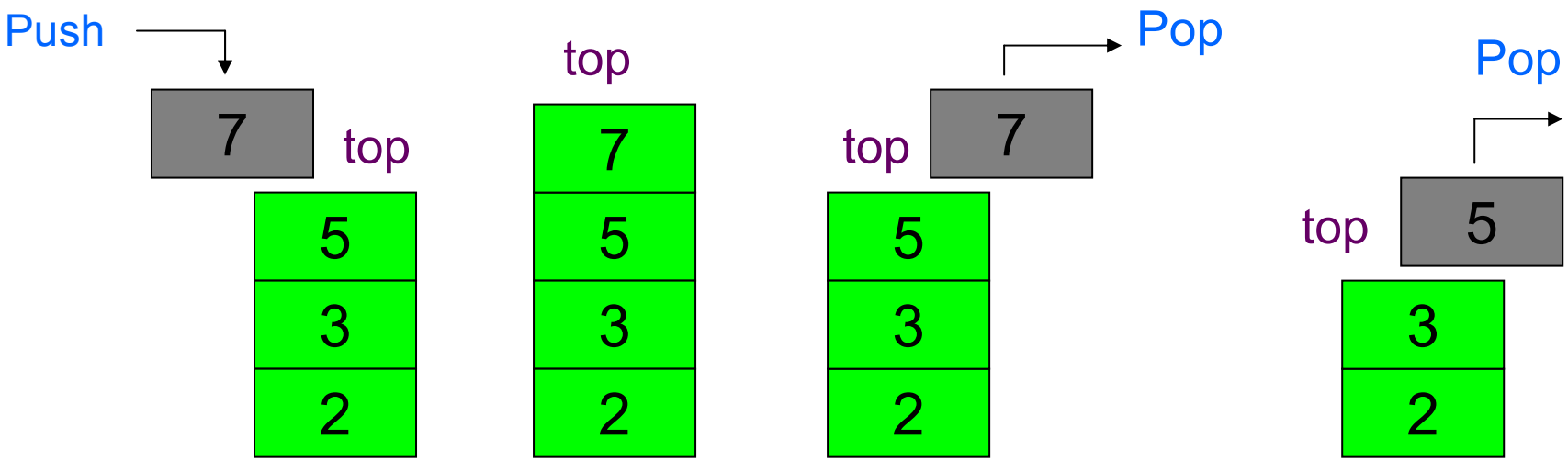
- Chương 1 – Thiết kế và phân tích (5 tiết)
- Chương 2 – Giải thuật đệ quy (10 tiết)
- Chương 3 – Mảng và danh sách (5 tiết)
- Chương 4 – Ngăn xếp và hàng đợi (10 tiết)
- Chương 5 – Cấu trúc cây (10 tiết)
- Chương 8 – Tìm kiếm (5 tiết)
- Chương 7 – Sắp xếp (10 tiết)
- Chương 6 – Đồ thị (5 tiết)

Chương 4 – Ngăn xếp và hàng đợi

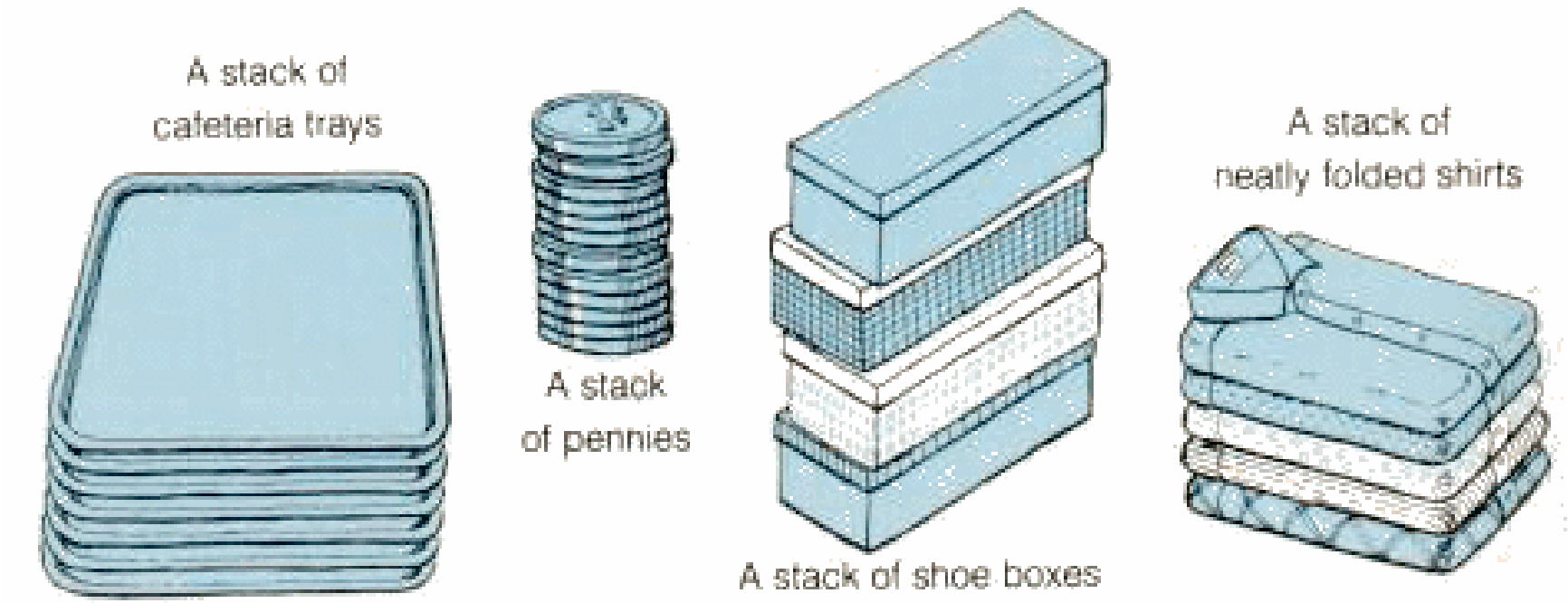
1. Định nghĩa Stack
2. Lưu trữ kế tiếp với Stack (sử dụng mảng)
3. Ứng dụng của Stack
4. Định nghĩa Queue
5. Lưu trữ kế tiếp với Queue (sử dụng mảng)
6. Ứng dụng của Queue (not yet)
7. Lưu trữ móc nối với Stack
8. Lưu trữ móc nối với Queue (bài tập)
9. Stack và cài đặt đệ quy (not neccesary)

1. Định nghĩa Stack

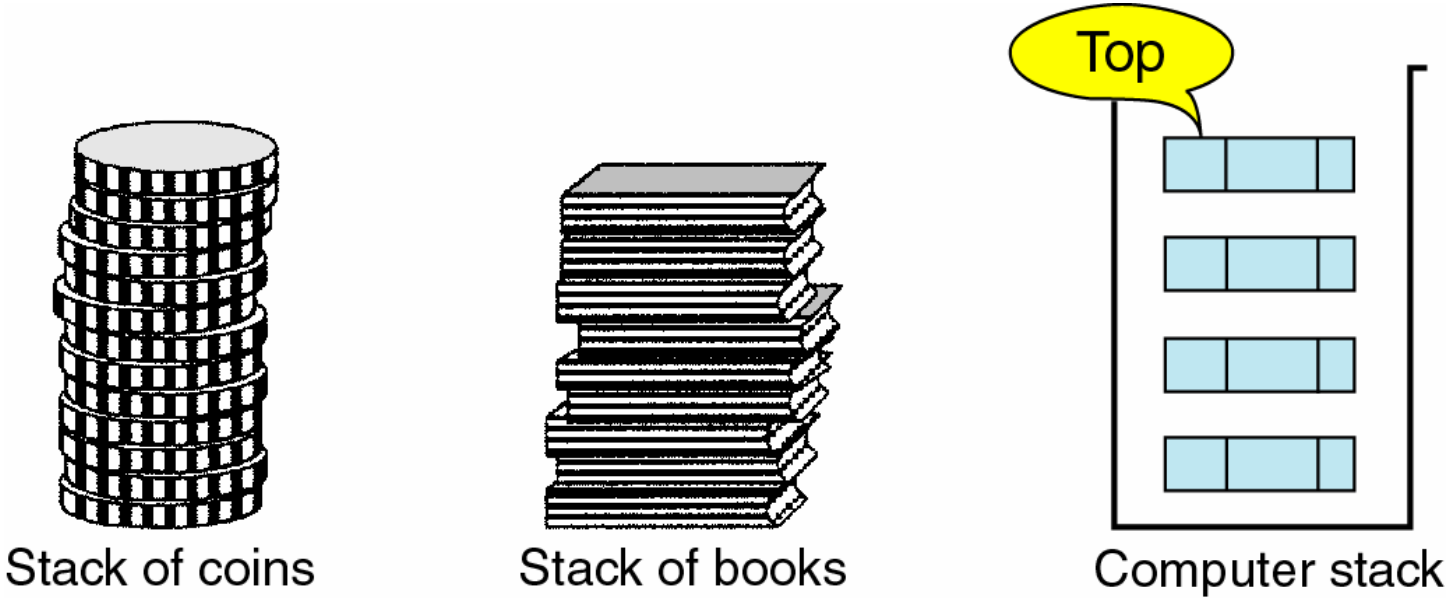
- Hai danh sách tuyến tính đặc biệt:
  - Ngăn xếp – Stack
  - Hàng đợi – Queue
- Stack: là danh sách mà xóa và thêm phần tử bắt buộc phải cùng được thực hiện tại một đầu duy nhất (đỉnh)



Ví dụ của Stack trong thực tế



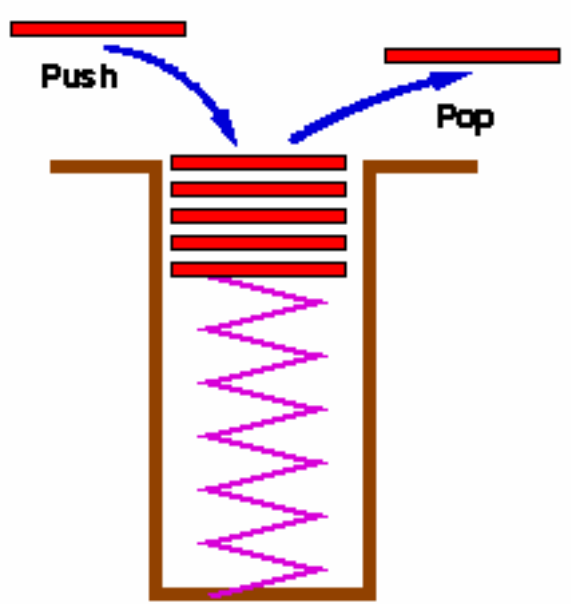
Ví dụ của Stack trong thực tế



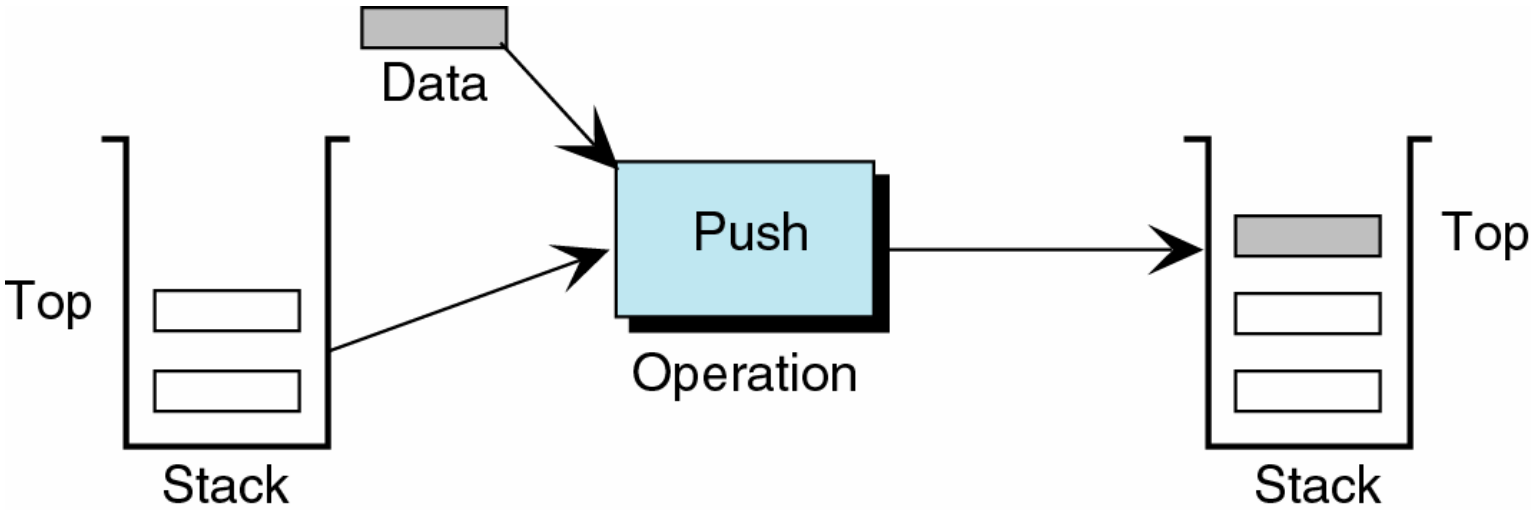
- Stack là một cấu trúc LIFO: Last In First Out

Các thao tác cơ bản trên Stack

- Push
  - Thêm một phần tử
    - Tràn (overflow)
- Pop
  - Xóa một phần tử
    - Underflow
- Top
  - Phần tử đỉnh
    - stack rỗng
- Kiểm tra rỗng/đầy

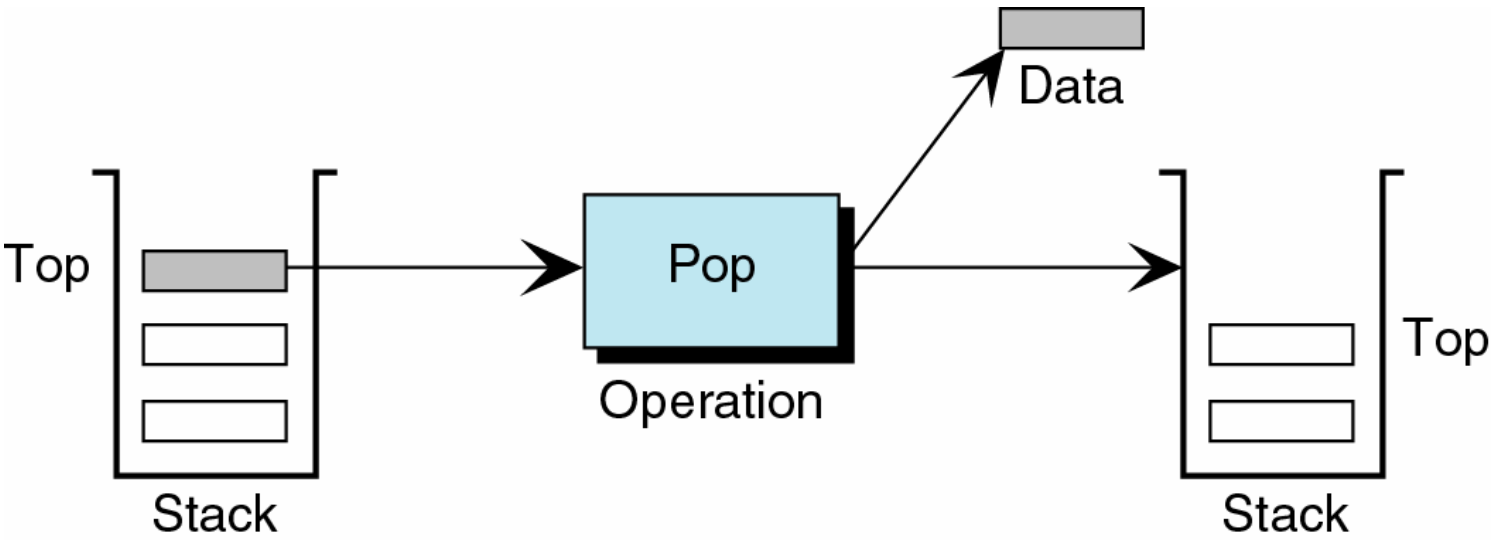


Push



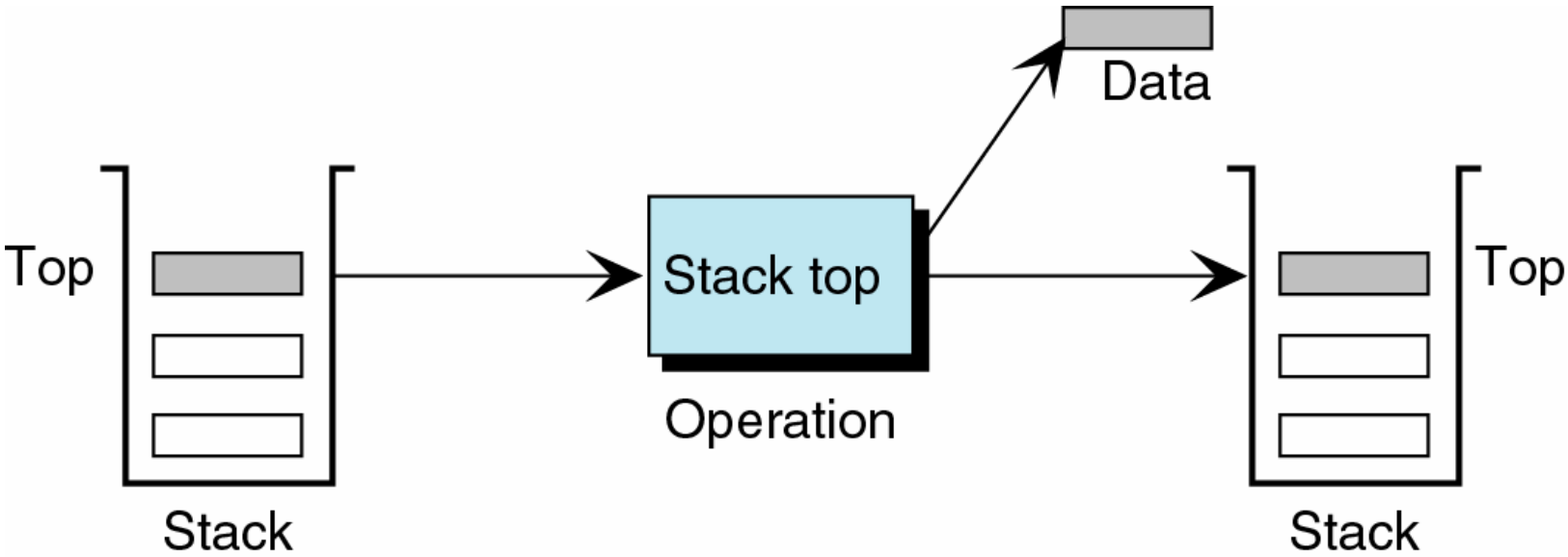
- Thêm phần tử mới vào đỉnh stack

Pop



- Rút một phần tử ra khỏi đỉnh stack

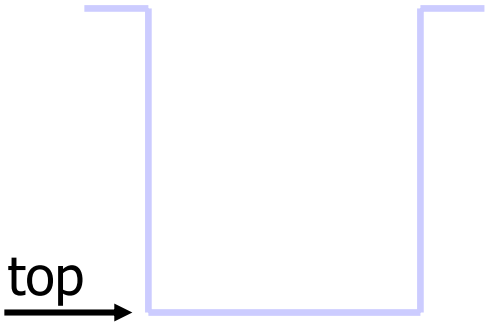
Top



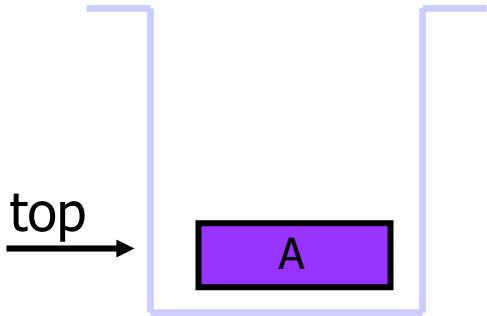
- Kiểm tra phần tử đỉnh. Stack không thay đổi

Push/Pop Stack

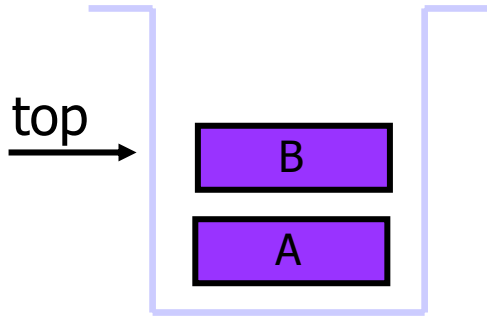
Stack rỗng



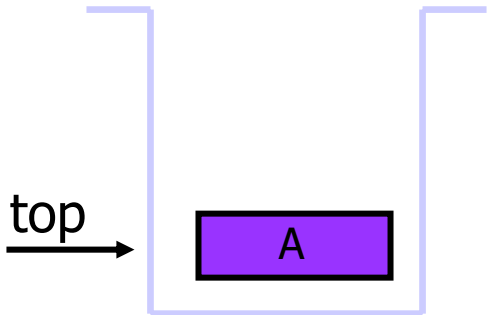
thêm một phần tử



Thêm một phần tử khác



Lấy một phần tử ra khỏi Stack



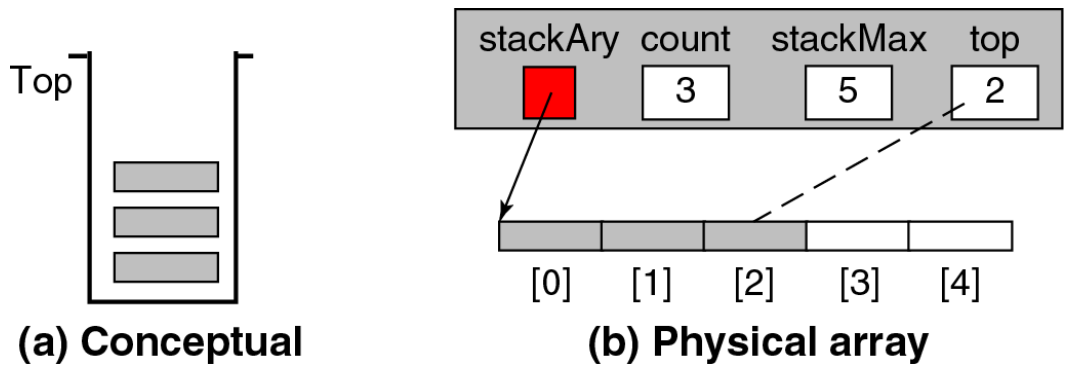
Lưu trữ Stack

- 2 cách lưu trữ:
  - Lưu trữ kế tiếp: sử dụng mảng
  - Lưu trữ móc nối: sử dụng danh sách móc nối

Lưu trữ Stack bằng Mảng

- Stack được lưu trữ như một mảng
  - Số các phần tử giới hạn

Figure 4-20

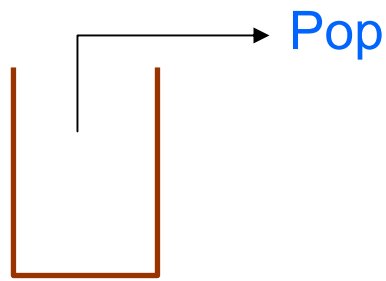


Cấu trúc dữ liệu

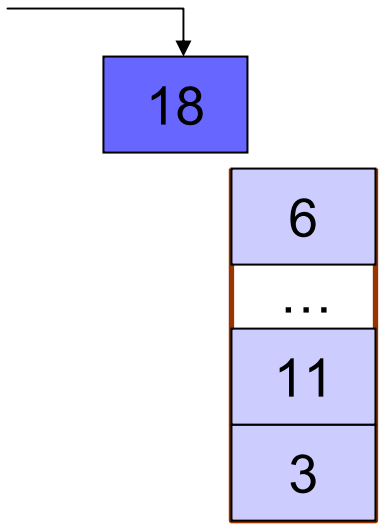
```
/* Stack của các số nguyên: intstack */

typedef struct intstack {
  int *stackAry;/* mảng lưu trữ các phần tử */
  int count;    /* số ptử hiện có của stack */
  int stackMax; /* giới hạn Max của số ptử */
  int top;      /* chỉ số của phần tử đỉnh */
}IntStack;
```

Tràn và Cạn

- Cạn (underflow) xảy ra khi cố gắng rút phần tử từ stack rỗng



- Tràn (overflow) xảy ra khi đẩy thêm phần tử vào stack đang đầy



Push

```
int PushStack(IntStack *stack,
              int dataIn) {
  /* Kiểm tra tràn */
  if (stack->count == stack->stackMax)
     return 0;

  /* Thêm phần tử vào stack */
  (stack->count)++;
  (stack->top)++; /* Tăng đỉnh */
  stack->stackAry[stack->top] =dataIn;
  return 1;
} /* pushStack */
```

Pop

```
int PopStack (IntStack *stack,
              int *dataOut) {
  /* Kiểm tra stack rỗng */
  if (stack->count == 0)
     return 0;
  /* Lấy giá trị phần tử bị loại*/
  *dataOut=stack->stackAry[stack->top];
  (stack->count)--;
  (stack->top)--; /* Giảm đỉnh */
  return 1;
} /* popStack */
```

Top

```
/* Lấy phần tử đỉnh của stack
   Trả lại 1 nếu thành công;
           0 nếu stack rỗng
           dataOut chứa kết quả */
int TopStack (IntStack *stack,
              int* dataOut) {
  if (stack->count == 0) // Stack rỗng
     return 0;
  *dataOut = stack->stackAry[stack->top];
  return 1;
} /* stackTop */
```

Kiểm tra rỗng?

```
/* Kiểm tra stack rỗng
   Trả lại 1 nếu là rỗng
           0 nếu không rỗng */
int IsEmptyStack (IntStack *stack)
{
  return (stack->count == 0);
} /* emptyStack */
```

Kiểm tra đầy?

```
/* Kiểm tra stack đầy
   Trả lại 1 nếu là đầy
           0 nếu không đầy */
int IsFullStack (IntStack *stack)
{
  return(stack->count==stack->stackMax);

} /* fullStack */
```

Tạo Stack

```
IntStack *CreateStack (int max) {
  IntStack *stack;
  stack=(IntStack*)malloc(sizeof(IntStack))
  if (stack == NULL)
     return NULL ;
  /* Khởi tạo stack rỗng */
  stack->top = -1;
  stack->count = 0;
  stack->stackMax = max;
  stack->stackAry =malloc(max*sizeof(int));
  return stack ;
} /* createStack */
```

3. Ứng dụng của Stack

- Bài toán đổi cơ số: Chuyển một số từ hệ thập phân sang hệ cơ số bất kỳ

(base 8) $28_{10} = 3 \bullet 8^1 + 4 \bullet 8^0 = 34_8$

(base 4) $72_{10} = 1 \bullet 4^3 + 0 \bullet 4^2 + 2 \bullet 4^1 + 0 \bullet 4^0 = 1020_4$

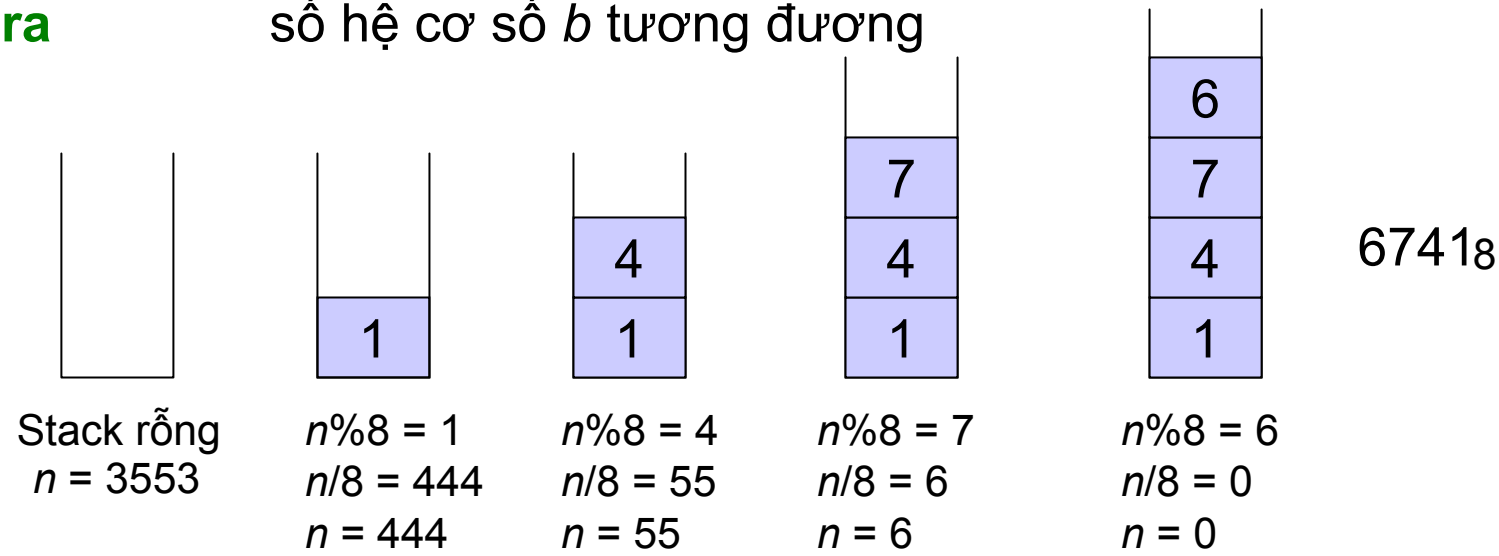
(base 2) $53_{10} = 1 \bullet 2^5 + 1 \bullet 2^4 + 0 \bullet 2^3 + 1 \bullet 2^2 + 0 \bullet 2^1 + 1 \bullet 2^0 = 110101_2$

3. Ứng dụng Stack

Đầu vào số thập phân *n*

Đầu ra số hệ cơ số *b* tương đương

Ex.



1. Chữ số bên phải nhất của kết quả = *n* % *b*. Đẩy vào Stack.
2. Thay *n* = *n* / *b* (để tìm các số tiếp theo).
3. Lặp lại bước 1-2 cho đến khi n = 0.
4. Rút lần lượt các chữ số lưu trong Stack, chuyển sang dạng ký tự tương ứng với hệ cơ số trước khi in ra kết quả

Chuyển sang dạng ký tự tương ứng:

```
char* digitChar = “0123456789ABCDEF”;
char d = digitChar[13]; // 13_10 = D_16
char f = digitChar[15]; // 13_10 = F_16
```

Đổi cơ số

```
void DoiCoSo(int n, int b) {
   char* digitChar = "0123456789ABCDEF";
   // Tạo một stack lưu trữ kết quả
   IntStack *stack = CreateStack(MAX);
   do {
      // Tính chữ số bên phải nhất,đẩy vào stack
      PushStack (stack, n % b);
      n /= b;       // Thay n = n/b để tính tiếp
   } while (n != 0); // Lặp đến khi n = 0

   while ( !IsEmptyStack(stack) ){
      // Rút lần lượt từng phần tử của stack
      PopStack(stack, &n);
      // chuyển sang dạng ký tự và in kết quả
      printf("%c", digitChar[n]);
   }
}
```

3. Ứng dụng của Stack (tiếp)

Ký pháp trung tố:

- Với phép toán 2 ngôi: Mỗi toán tử được đặt giữa hai toán hạng
- Với phép toán một ngôi: Toán tử được đặt trước toán hạng

-2 + 3 * 5 ⟺ (-2) + (3 * 5)

✦ Việc đánh giá biểu thức trung tố khá phức tạp

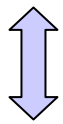
Sắp xếp giảm dần của thứ tự ưu tiên của toán tử:

() > ^ > * = % = / > + = –

Ký pháp hậu tố

Toán hạng đặt *trước* toán tử.

a b * c + Không cần dấu ngoặc



a * b + c (Biểu thức trung tố tương đương)

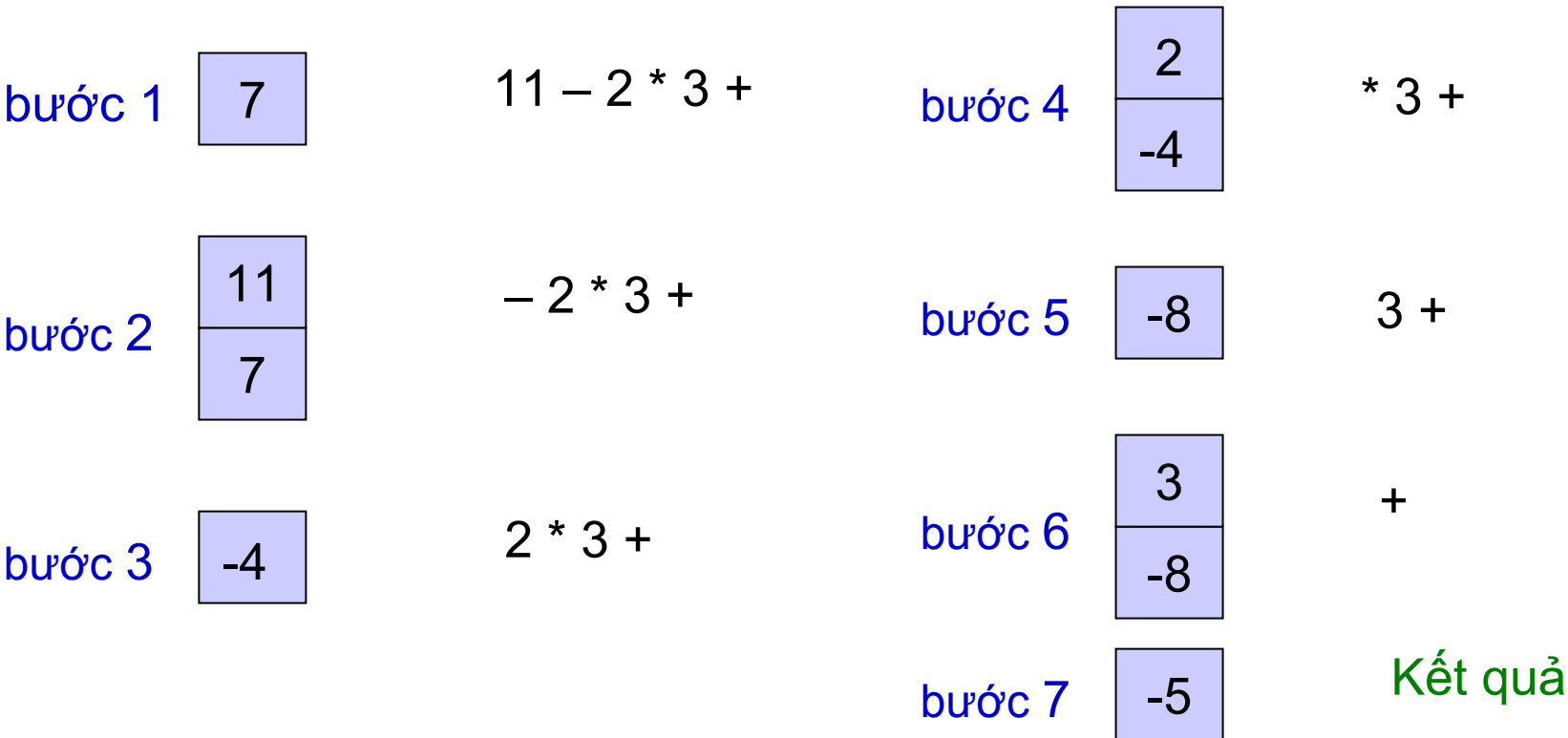
Ví dụ.

Trung tố	Hậu tố
a*b*c*d*e*f	ab*c*d*e*f*
1 + (-5) / (6 * (7+8))	1 5 - 6 7 8 + * / +
(x/y – a*b) * ((b+x) – y^y)	x y / a b * – b x + y y ^ – *
(x*y*z – x^2 / (y*2 – z^3) + 1/z) * (x – y)	xy*z*x2^y2*z3^ – / – 1z/+xy – *

Tính giá trị biểu thức hậu tố

Biểu thức trung tố: (7 – 11) * 2 + 3
Biểu thức hậu tố: 7 11 – 2 * 3 +

Sử dụng một stack lưu trữ toán hạng



postfixEval
Tính giá trị của biểu thức hậu tố

Tính giá trị của một một biểu thức hậu tố được lưu trong một xâu ký tự và trả về giá trị kết quả.

Toán hạng:

Các số nguyên không âm một chữ số (cho đơn giản ☺)

Toán tử:

+, -, *, /, %, ^ (lũy thừa)

Định nghĩa một số hàm

- int compute(int left, int right, char op);

  /* Thực hiện tính: “left op right” */

- bool isOperator(char op);

  /* Kiểm tra op có phải là toán tử không?

  op phải là một trong số '+','-','*','/','%','^‘

  */

Hàm isOperator()

Hàm isOperator() kiểm tra ký tự có phải là toán tử?

```
bool isOperator(char op)
{
    return op == '+' || op == '-' ||
           op == '*' || op == '%' ||
           op == '/' || op == '^';
}
```

```
int compute(int left, int right, char op) {
    int value;
    // Tính "left op right"
    switch(op){
        case '+': value = left + right;
            break;
        case '-': value = left - right;
            break;
        case '*': value = left * right;
            break;
        case '%': value = left % right;
            break;
        case '/': value = left / right;
            break;
        case '^': value = pow(left, right);
            break;
    }
    return value;
}
```

Hàm postfixEval()

```
int postfixEval(string expression)
{
      // expValue lưu kết quả của biểu thức
      int left, right, expValue;
      char ch;
      // Tạo một stack lưu trữ toán hạng
      IntStack* stack = CreateStack(MAX);

      // Duyệt từng ký tự cho đến khi hết xâu
      for (int i=0; i < expression.length(); i++)
      {
            // đọc một ký tự
            ch = expression[i];
            // nếu ch là toán hạng
            if (isdigit(ch))
                  // đẩy toán hạng vào stack
                  PushStack(stack, ch - '0');
```

```
        // nếu ch là toán tử
        else if (isOperator(ch)){
            // rút stack 2 lần để lấy 2
            // toán hạng left và right
            PopStack(stack, &right);
            PopStack(stack, &left);
            // Tính "left op right"
            result = compute(left, right, ch);
            // Đẩy result vào stack
            PushStack(stack, temp);
        }else //không phải toán hạng hoặc toán
            printf("Bieu thuc loi");
    }
    // Kết thúc tính toán, giá trị biểu thức
    // nằm trên đỉnh stack, đưa vào expValue
    PopStack(stack, expValue);
    return expValue;
}
```

Chuyển đổi trung tố→hậu tố

Trong khi quét biểu thức số học:

- Toán hạng sẽ được ghi ngay vào xâu kết quả
- Không cần sử dụng stack cho toán hạng.

stack toán tử

- Khi gặp toán tử hoặc dấu ngoặc, đẩy vào stack.
- Quản lý thứ tự ưu tiên giữa các toán tử
- Xử lý các biểu thức con.

Hạng

Chỉ xét các toán tử hai ngôi.

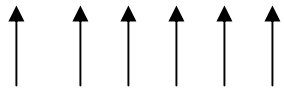
Hạng.

1	nếu là toán hạng
-1	nếu là +, -, *, /, %, ^
0	nếu là (,)

Ví dụ 1

stack dùng để lưu trữ một cách tạm thời các toán tử trong khi chờ toán hạng 2 (phải) tương ứng.

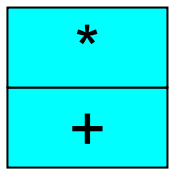
a + b * c



* có mức ưu tiên cao hơn +

$\Rightarrow$ Thêm vào stack

Stack toán tử:



Xâu hậu tố:

a	b	c	*	+

Ví dụ 2

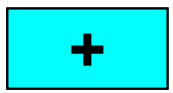
Sử dụng stack để xử lý các toán tử có cùng thứ tự ưu tiên hoặc thấp hơn.

a * b / c + d

* có cùng mức ưu tiên với /
⇒ rút * và ghi nó vào xâu hậu tố trước khi thêm / vào stack.

/ có mức ưu tiên cao hơn +

Stack toán tử:



Xâu hậu tố:

a	b	*	c	/	d	+

Ví dụ 3

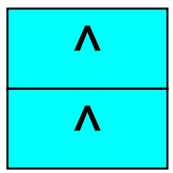
Sử dụng giá trị mức ưu tiên để xử lý ^ (tính lũy thừa).

Mức ưu tiên đầu vào: 4 khi ^ là đầu vào.

Mức ưu tiên tại stack: 3 khi ^ nằm trong stack.

a ^ b ^ c

Stack toán tử:



^ thứ 2 có mức ưu tiên là 4 nhưng

^ thứ 1 có mức ưu tiên là 3

$\Rightarrow$ ^ thứ 2 được đẩy tiếp vào stack

(do đó nó sẽ được rút ra trước ^ thứ 1)

Xâu hậu tố:

a	b	c	^	^

Ví dụ 4

Hai mức ưu tiên cho dấu ngoặc trái (

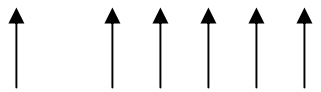
Mức ưu tiên đầu vào: 5 cao hơn bất kỳ toán tử nào.

(tất cả các toán tử trong stac phải giữ nguyên vì có một biểu thức con mới.)

Mức ưu tiên tại stack: -1 thấp hơn của bất kỳ toán tử nào.

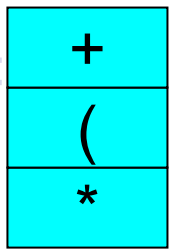
(không toán tử nào trong biểu thức con được xóa cho đến khi gặp dấu ngoặc mở)

a * (b + c)



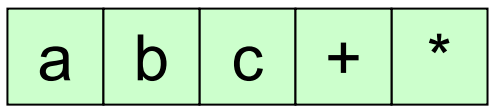
(có mức ưu tiên là 5 ⇒ đưa vào stack.

Stack toán tử:



(hiện có mức ưu tiên là -1 ⇒ tiếp tục ở trong stack.

Xâu hậu tố:



Mức ưu tiên đầu vào và tại Stack

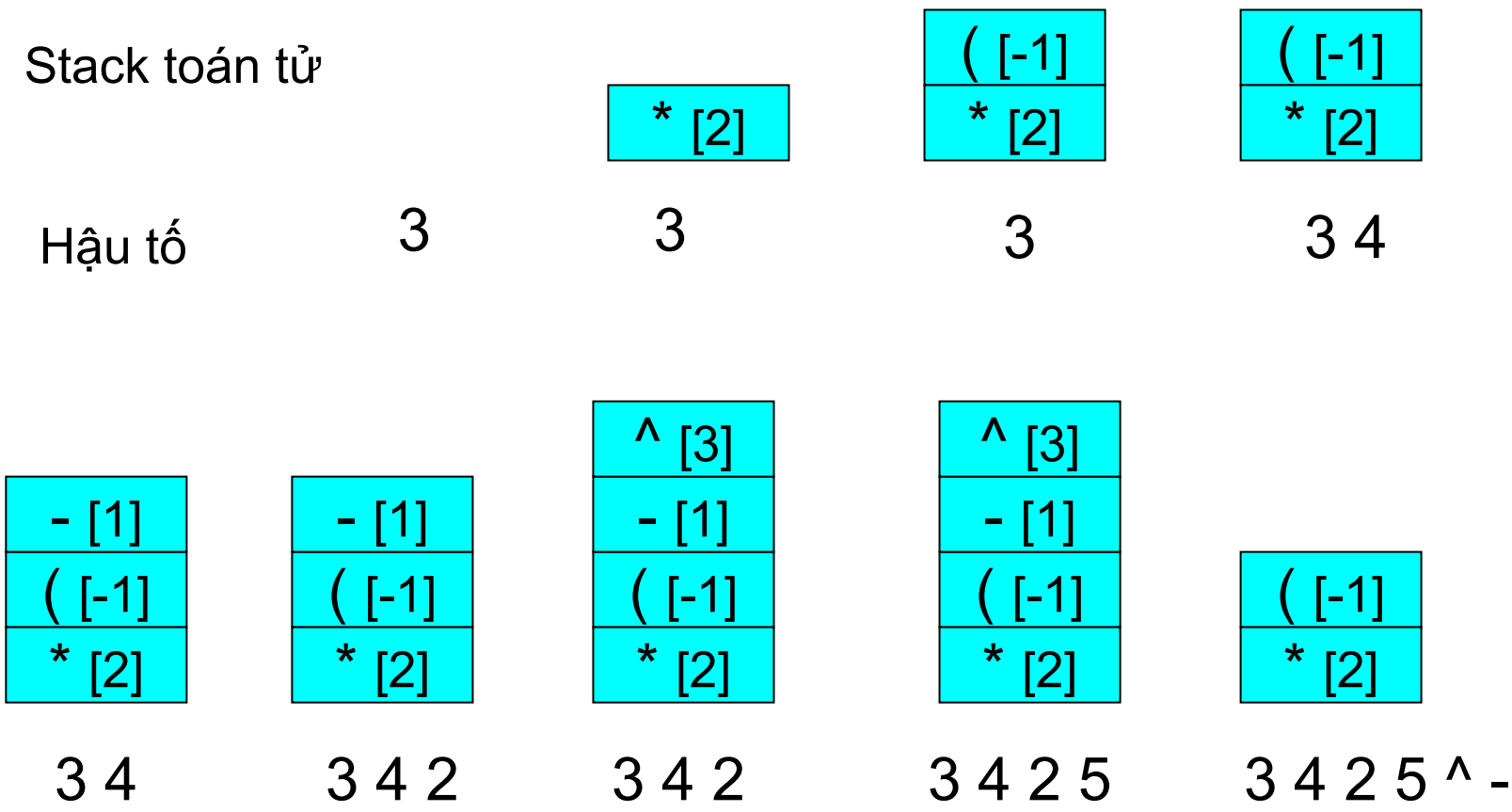
Toán tử	Mức ưu tiên đầu vào	Mức ưu tiên tại stack	Hạng
+ -	1	1	-1
* / %	2	2	-1
^	4	3	-1
(	5	-1	0
)	0	0	0

Các quy luật đánh giá

- Ghi ký tự vào xâu hậu tố nếu nó là toán hạng.
- Nếu ký tự là một toán tử hoặc **(**, so sánh mức ưu tiên của nó với mức ưu tiên của toán tử tại đỉnh stack.
  Rút phần tử đỉnh stack nếu mức ưu tiên của phần tử tại stack là cao hơn hoặc bằng và ghi tiếp nó vào xâu hậu tố.
  Lặp cho đến khi toán tử tại đỉnh stack có hạng thấp hơn, đẩy ký tự vào stack.
- Nếu ký tự là **)**, rút tất cả các toán tử ra khỏi stack cho đến khi gặp **(** và ghi các toán tử vào xâu hậu tố. Rút **(** ra khỏi stack.
- Khi kết thúc biểu thức trung tố, rút tất cả các toán tử ra khỏi stack và ghi vào xâu hậu tố.

Ví dụ

3 * (4 – 2 ^ 5) + 6


Stack toán tử
Hậu tố
* [2]
([-1]
* [2]
([-1]
* [2]
3
3
3
3 4
- [1]
([-1]
* [2]
- [1]
([-1]
* [2]
^ [3]
- [1]
([-1]
* [2]
^ [3]
- [1]
([-1]
* [2]
([-1]
* [2]
3 4
3 4 2
3 4 2
3 4 2 5
3 4 2 5 ^ -

cont'd

Pop (3 * (4 – 2 ^ 5) + 6

* [2]	+ [1]	+ [1]
3 4 2 5 ^ -	3 4 2 5 ^ - *	3 4 2 5 ^ - * 6

3 4 2 5 ^ - * 6 +

Stack

Xây dựng một stack cho phép lưu trữ các toán tử và mức ưu tiên của nó.

```
typedef struct Operator {
      char symbol;  // toán tử
      // mức ưu tiên đầu vào của toán tử op
      int inputPrecedence;
      // mức ưu tiên trong stack của toán tử op
      int stackPrecedence;
}Operator;

typedef struct OpStack {
      Operator * stackAry;
      ….
} OpStack ;
```

Output Stack Symbols

Rút các toán tử trong stack có *stack precedence* ≥ *input precendence* của ký tự đang đọc.

```
void PopHigherOrEqualOp(OpStack* stack, Operator& op
                        string& postfix)
{
      Operator op2;

      while(!IsEmpty(stack) &&
      (op2 = Top(stack)).stackPrecedence >=
                                    op.inputPrecedence)
      {
            Pop(stack);
            postfix += op2.symbol;
      }
}
```

Hàm chuyển đổi trung tố - hậu tố

Infix2Postfix() thực hiện những công việc sau:

- Ghi toán hạng ra xâu hậu tố.
- Gọi outputHigherOrEqual() nếu gặp toán tử.
- Gọi outputHigherOrEqual() nếu gặp).
- Kết thúc khi đọc hết biểu thức

```
string Infix2Postfix ( string infix) {
    string postfix; // lưu xâu biểu thức hậu tố
    OpStack* stack = CreateStack( MAX); // tạo stack
    // Duyệt từng ký tự của biểu thức
    for (i=0; i < infix.length(); i++) {
       ch = infix [i];
       //******  Trường hợp toán hạng ***********
       if (isdigit(ch))
          // ghi toán hạng vào biểu thức hậu tố
          postfix += ch;
       //*******  Trường hợp toán tử hoặc '('  *****
       else if (isOperator(ch) || ch == '(')
       {
          // rút các toán tử có mức ưu tiên cao hơn
          // ra khỏi stack
          Operator op = createOperator(ch);
          PopHigherOrEqualOp(stack, op, postfix);
          // đẩy toán tử hiện tại vào stack
          Push(stack, op);
       }
```

```
        //********  Trường hợp  ')'  ************
            else if (ch == ')')
        {
                // tạo một biến Operator cho ')'
                Operator op = CreateOperator(ch);
                // Rút tất cả toán tử của biểu thức con
                // cho đến khi gặp '('
                PopHigherOrEqualOp(stack, op, postfix);
                // Rút '('  ra khỏi stack
                Pop(stack);
        }
     } // end for
    // Rút các toán tử còn lại trg stack, ghi vào xâu
    while (!IsEmpty(stack)) {
        op = Pop(stack);
        postfix += op.symbol;
    }
    return postfix;
}
```

4. Định nghĩa Queue

- Queue: là danh sách mà thêm phải được thực hiện tại một đầu còn xóa phải thực hiện tại đầu kia.

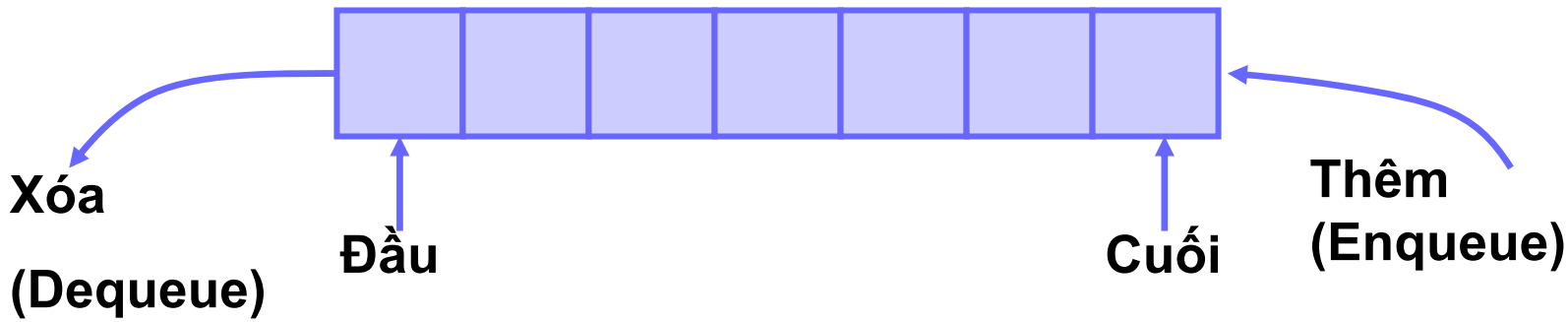
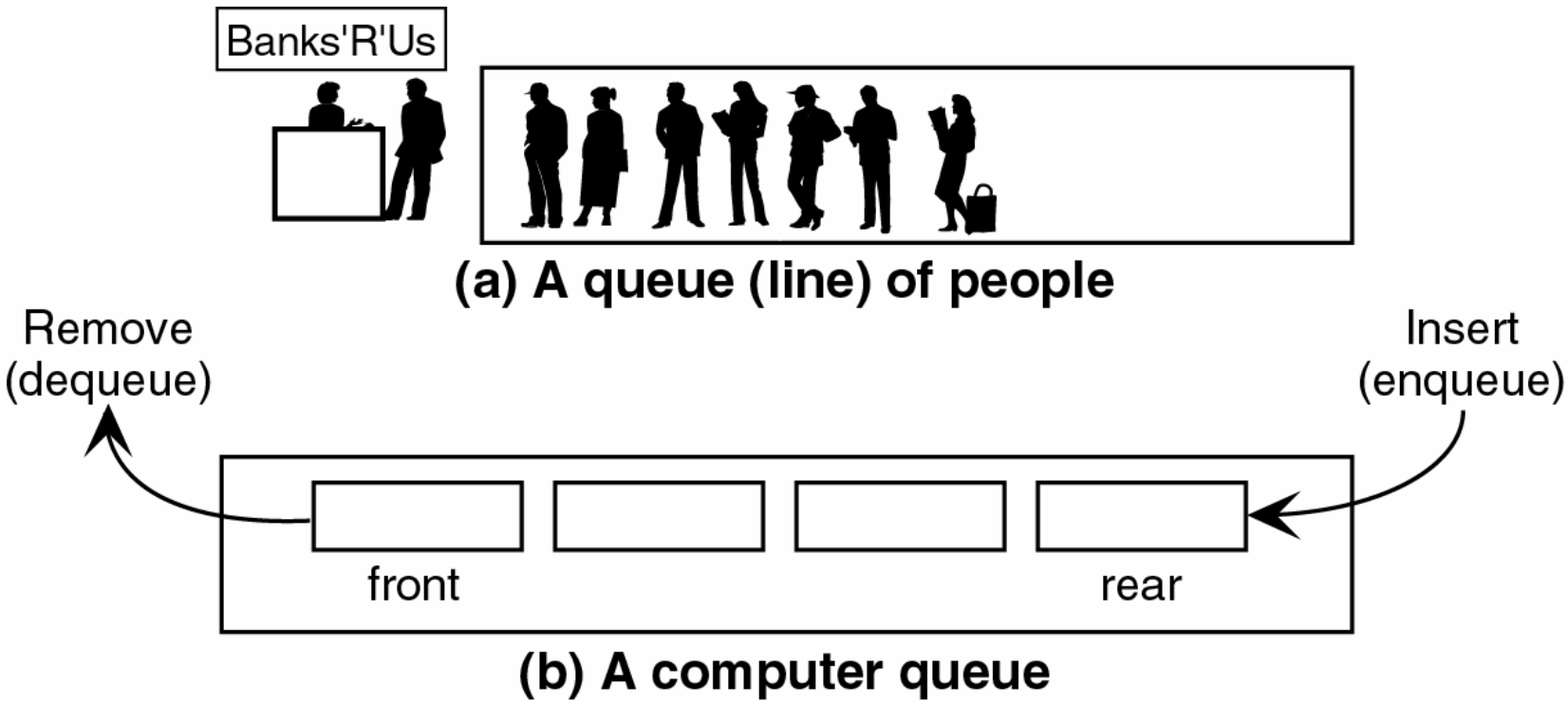


Figure 5-1

Ví dụ của Queue trong thực tế



- Queue là một kiểu cấu trúc FIFO: First In First Out

Các thao tác cơ bản với Queue

- Enqueue – Thêm một phần tử vào cuối queue
  - Tràn Overflow
- Dequeue – Xóa một phần tử tại đầu queue
  - Queue rỗng?
- Front – Trả lại phần tử tại đầu queue
  - Queue rỗng?
- End – Trả lại phần tử tại cuối queue
  - Queue rỗng

Figure 5-2

Enqueue

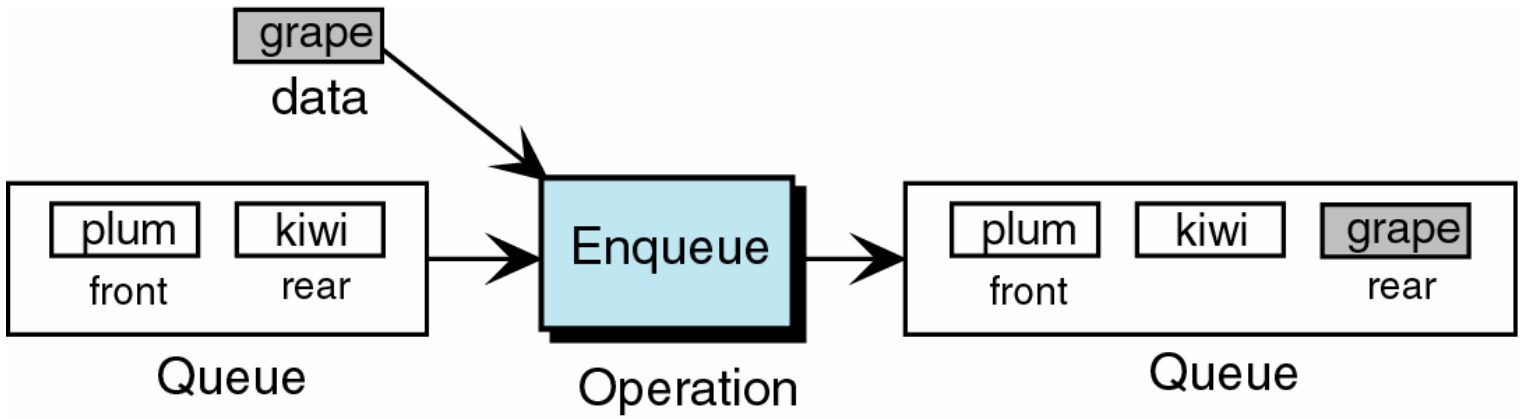


Figure 5-3

Dequeue

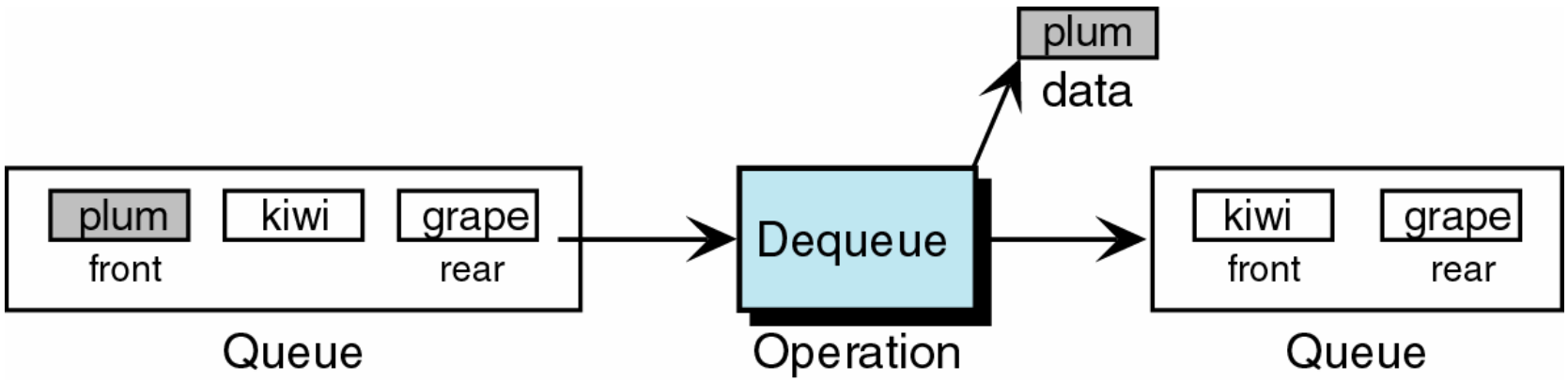


Figure 5-4

Queue Front

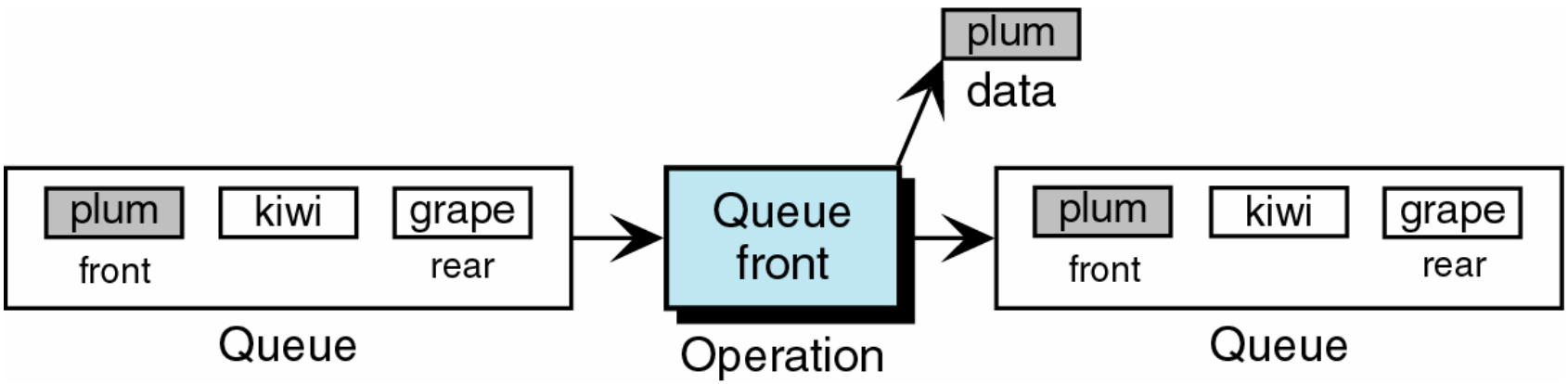
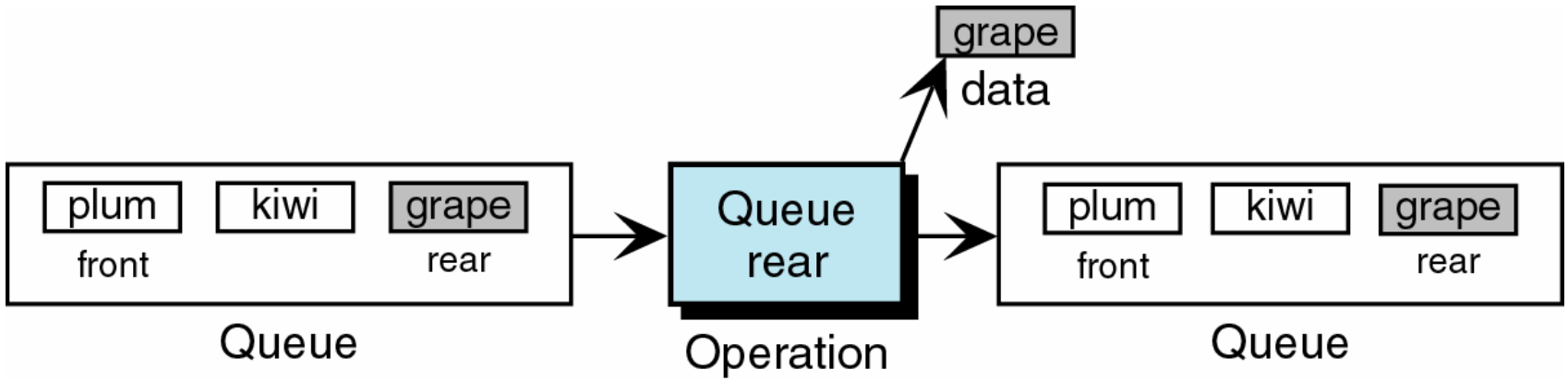


Figure 5-5

Queue Rear



Lưu trữ Queue

- Tương tự như Stack, có 2 cách lưu trữ:
  - Lưu trữ kế tiếp: sử dụng mảng
  - Lưu trữ móc nối: sử dụng danh sách móc nối

Figure 5-15

5. Lưu trữ kế tiếp với Queue

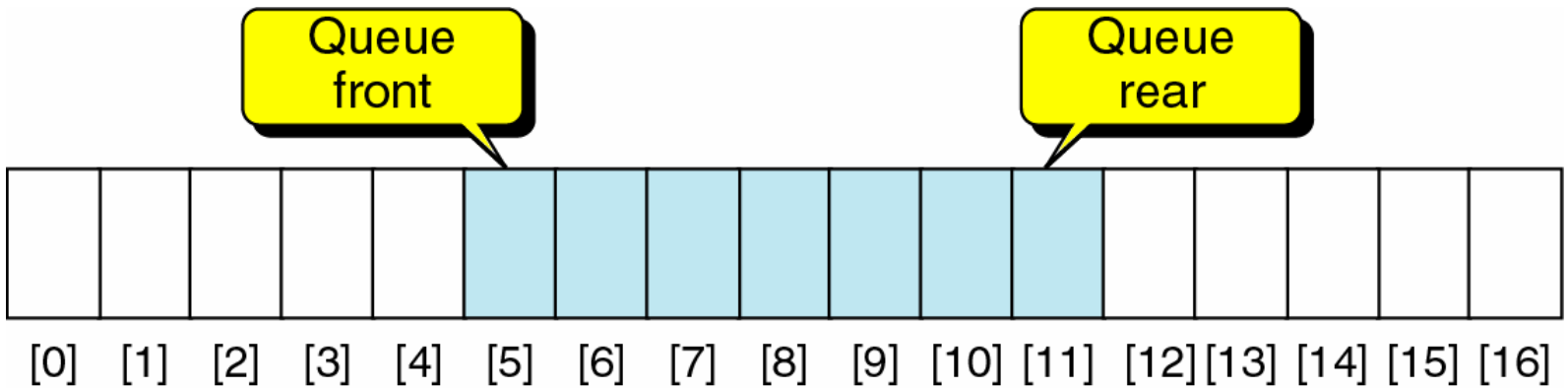
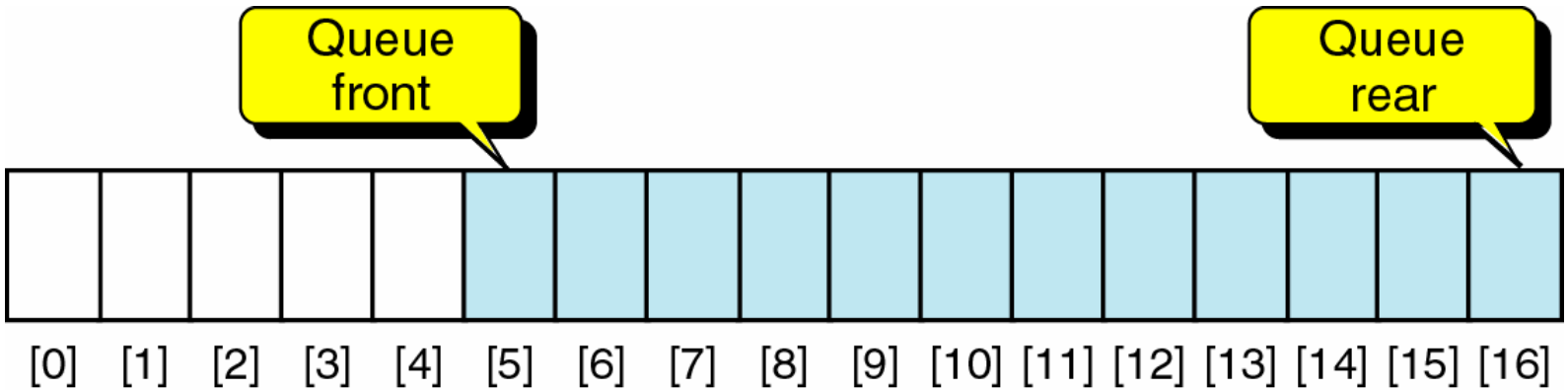
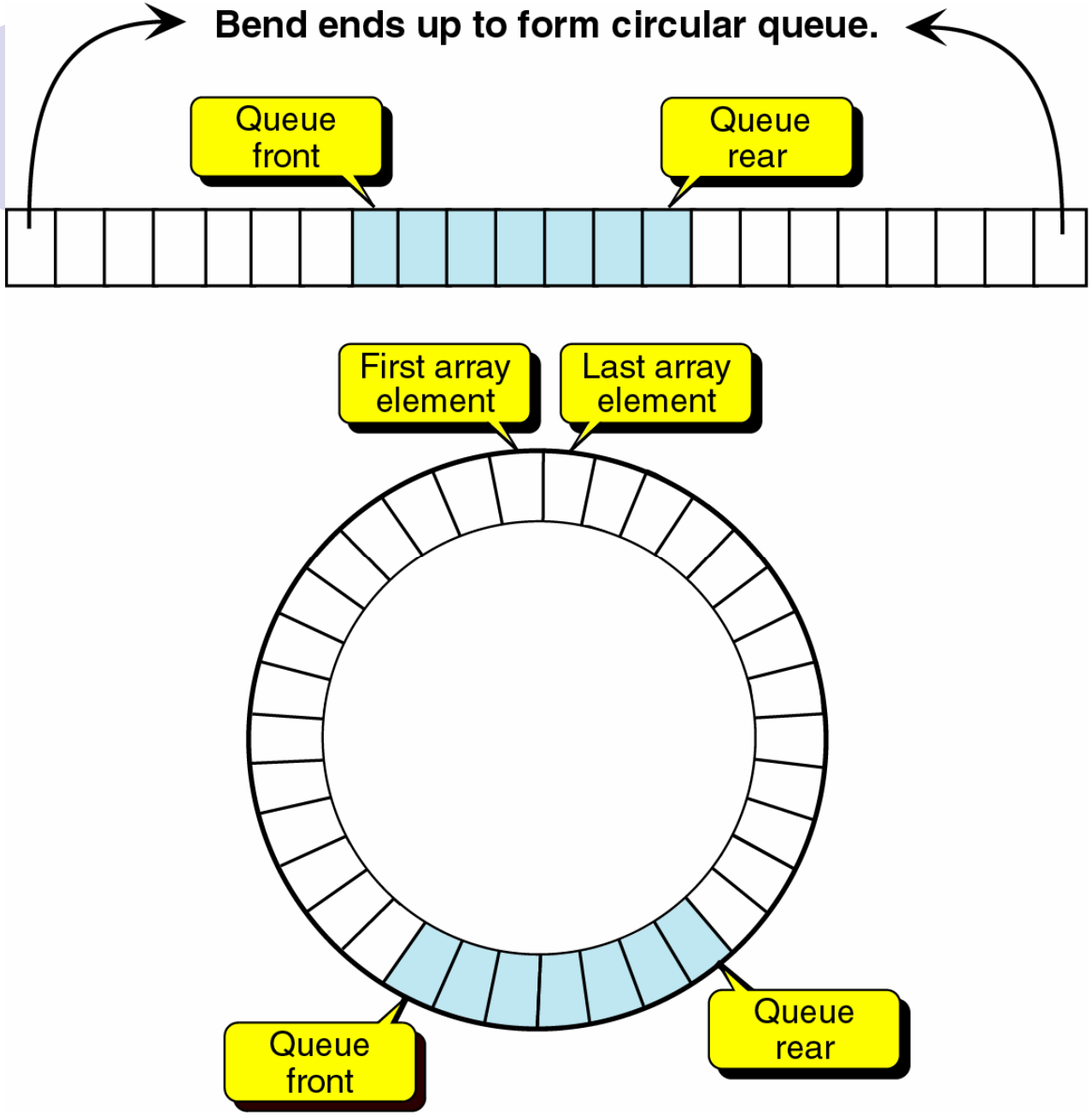


Figure 5-16

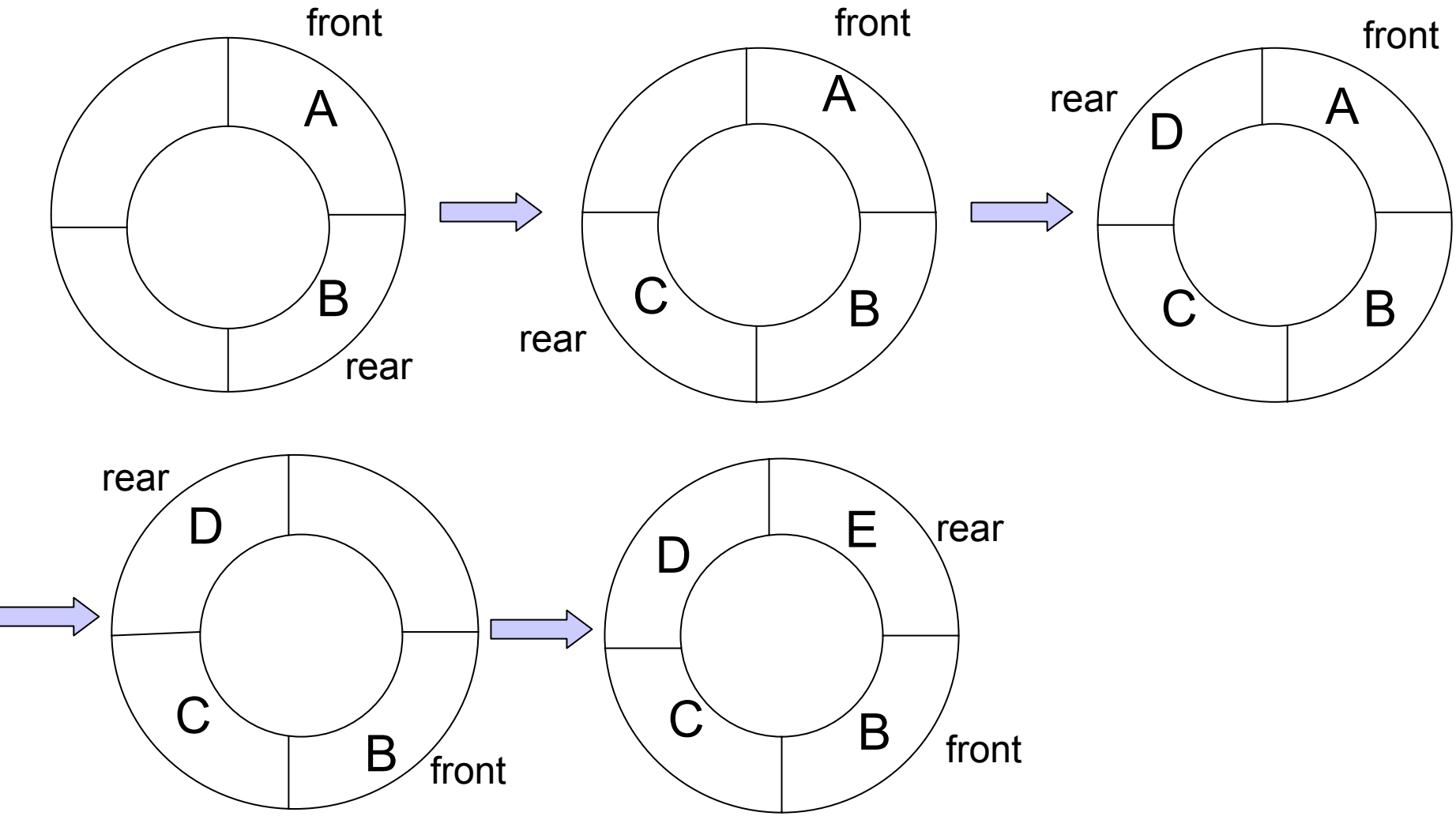
Queue tăng hết mảng



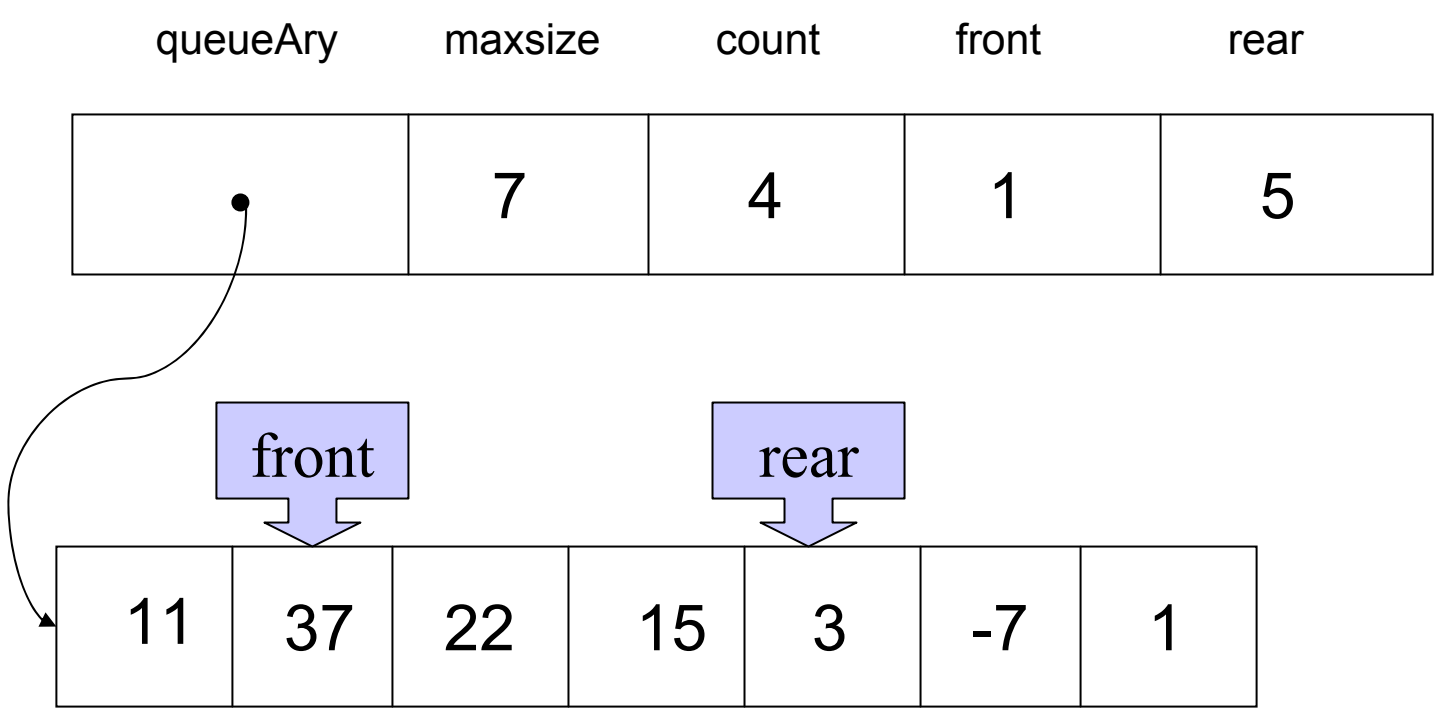
- Do đó cần sử dụng một mảng rất lớn?


Bend ends up to form circular queue.
Queue front
Queue rear
First array element
Last array element
Queue front
Queue rear

Queue dạng vòng



Queue thực hiện trên mảng



Định nghĩa cấu trúc Queue

```
typedef struct intqueue {
      int *queueAry;
      int maxSize;
      int count;
      int front;
      int rear;
}IntQueue;
```

Tạo Queue

```
IntQueue* CreateQueue (int max) {
  IntQueue *queue;
  queue = (IntQueue *)malloc(sizeof(IntQueue));

  /* Cấp phát cho mảng */
  queue->queueAry = malloc(max * sizeof(int));

  /* Khởi tạo queue rỗng */
  queue->front   = -1;
  queue->rear    = -1;
  queue->count   =  0;
  queue->maxSize = maxSize;

  return queue;
} /* createQueue */
```

Enqueue

```
int enqueue (struct intqueue  *queue, int   datain) {
    if (queue->count == queue->maxSize)
          return 0; /* queue is full */
    (queue->rear)++;
    if (queue->rear  == queue->maxSize)
       /* Queue wraps to element 0 */
       queue->rear  =  0;
    queue->queueAry[queue->rear] = datain;
    if (queue->count == 0) {
          /* Inserting into null queue */
          queue->front  = 0;
          queue->count  = 1;
         } /* if */
    else (queue->count)++;
    return 1;
```

Dequeue

```
int dequeue (struct intqueue *queue, int *dataOutPtr) {
    if (!queue->count)
          return 0;
     *dataOutPtr = queue->queueAry[queue->front];
    (queue->front)++;
    if (queue->front == queue->maxSize)
        /* queue front has wrapped to element 0 */
        queue->front =  0;
    if (queue->count == 1)
        /* Deleted only item in queue */
        queue->rear = queue->front = -1;
    (queue->count)--;

    return 1;
```

queueFront

```
int queueFront (struct intqueue  *queue,
                int *dataOutPtr) {
    if (!queue->count)
        return 0;
    else {
          *dataOutPtr = queue->queueAry[queue->front];
          return 1;
     } /* else */
}   /* queueFront */
```

queueRear

```
int queueRear (struct intqueue   *queue,
                    int *dataOutPtr) {
     if (!queue->count)
         return 0;
     else {
             *dataOutPtr = queue->queueAry[queue->rear];
             return 1;
     } /* else */

}    /* queueRear */
```

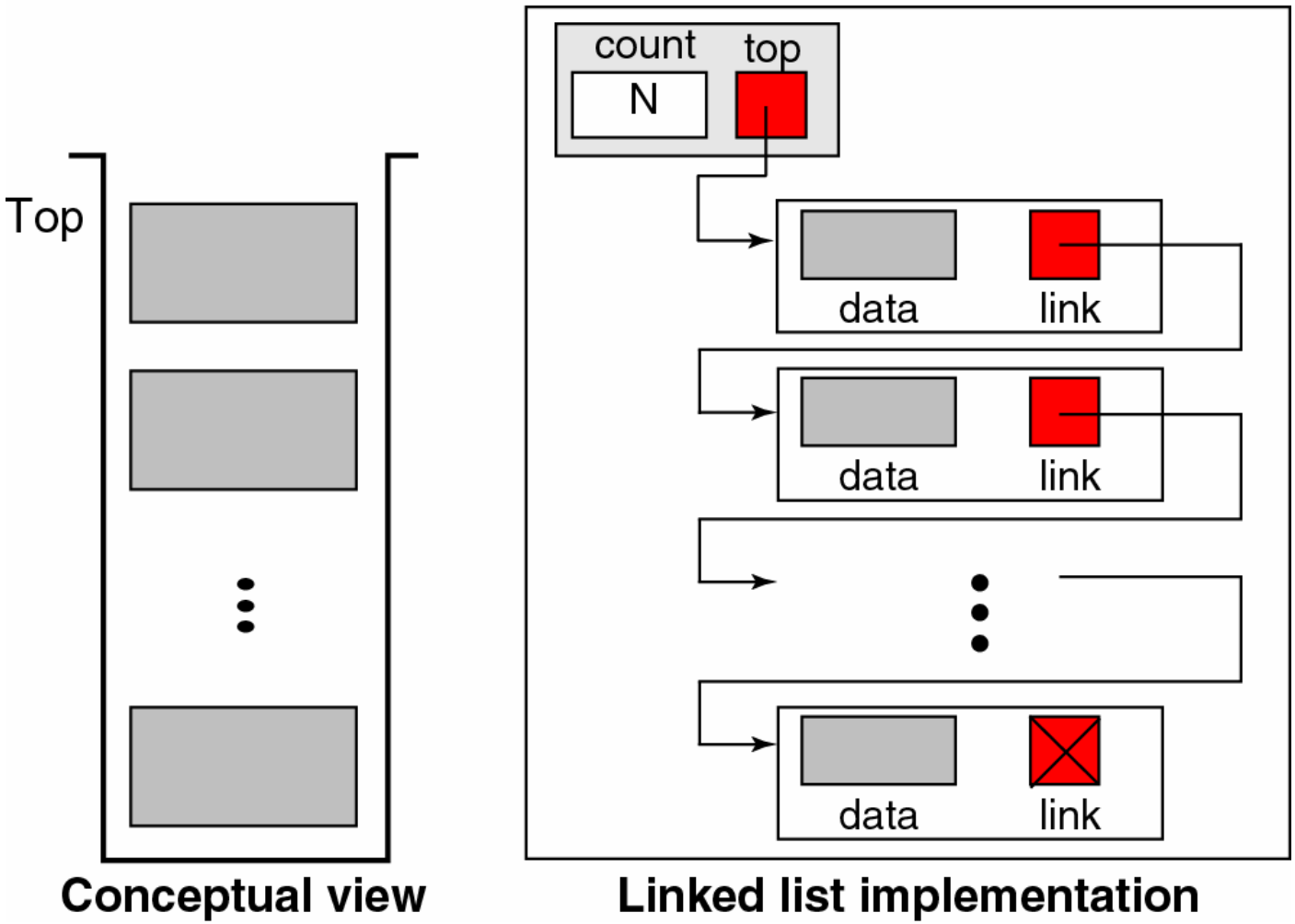
emptyQueue and fullQueue

```
int emptyQueue (struct intqueue *queue)
{
   return (queue->count == 0);
}  /* emptyQueue */

int fullQueue (struct intqueue  *queue )
{
   return ( queue->count == queue->maxSize);
}  /* fullQueue */
```

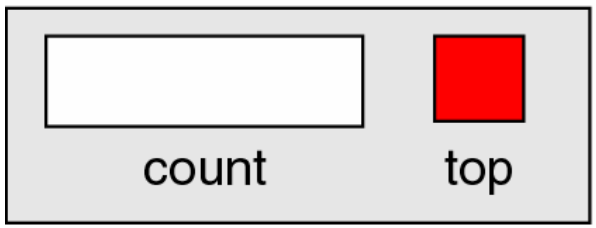
destroyQueue

```
struct intqueue *destroyQueue (struct intqueue
   *queue)
{
   if (queue)
       {
        free (queue->queueAry);
        free (queue);
       } /* if */
   return NULL;
}  /* destroyQueue */
```

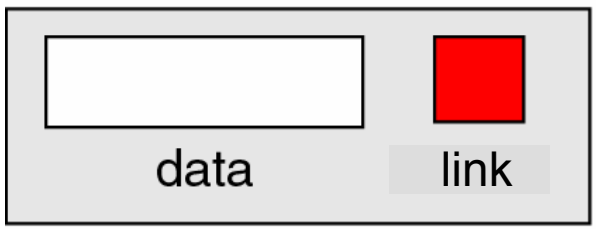
6. Lưu trữ móc nối với Stack



Các cấu trúc của head và node



Stack head structure



Stack node structure

```
stack
  count  <integer>
  top    <node pointer>
end stack



node
  data   <dataType>
  link   <node pointer>
end node
```

Khai báo stack

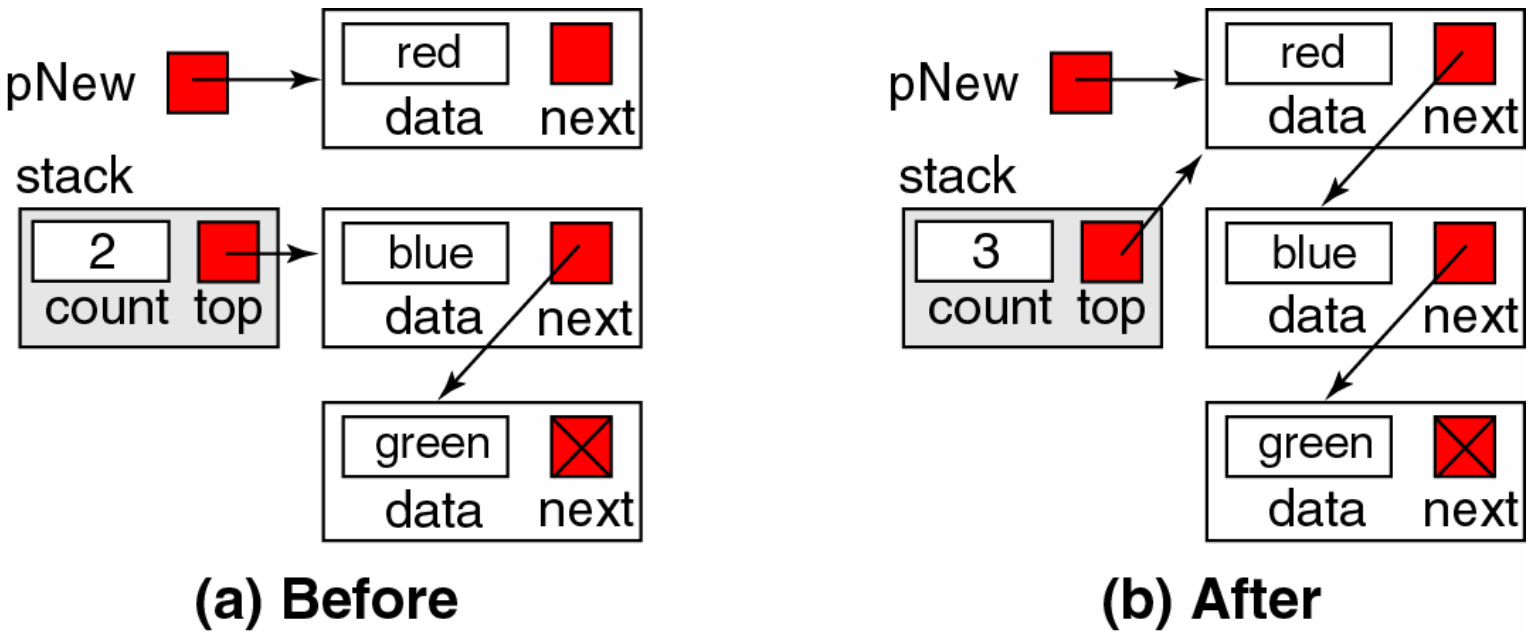
```
typedef struct node
      {
       int data ;
       struct node *link ;
      } STACK_NODE;

typedef struct stack
      {
       int count ;
       STACK_NODE *top ;
      } STACK;
```

createStack

```
static STACK  *createStack ()
{
   STACK  *stack ;
   stack = (STACK *) malloc( sizeof (STACK) ) ;
   if (stack)
         {
          stack->top   = NULL ;
          stack->count = 0;
         } /* if */

   return stack ;
}  /* createStack */
```

Push

- Giống như Thêm một phần tử mới vào danh sách trước phần tử đầu



pushStack

```
static int pushStack(STACK *stack, int dataIn) {
    STACK_NODE *newPtr;
    newPtr = (STACK_NODE *) malloc(sizeof(
    STACK_NODE));
    if (newPtr == NULL)
          return 0; /* no more space */
    newPtr->data = dataIn;
    newPtr->link = stack->top;
    stack->top = newPtr;
    ( stack->count )++;

    return 1;
}  /* pushStack */
```

7. Lưu trữ móc nối với Queue

- Bài tập về nhà: Xây dựng Queue móc nối

Cấu trúc dữ liệu và giải thuật

Đỗ Tuấn Anh

anhdt@it-hut.edu.vn

Nội dung

- Chương 1 – Thiết kế và phân tích (5 tiết)
- Chương 2 – Giải thuật đệ quy (10 tiết)
- Chương 3 – Mảng và danh sách (5 tiết)
- Chương 4 – Ngăn xếp và hàng đợi (10 tiết)
- Chương 5 – Cấu trúc cây (10 tiết)
- Chương 8 – Tìm kiếm (5 tiết)
- Chương 7 – Sắp xếp (10 tiết)
- Chương 6 – Đồ thị (5 tiết)

Chương 5 – Cấu trúc cây

1. Định nghĩa và khái niệm
2. Cây nhị phân
   - Định nghĩa và Tính chất
   - Lưu trữ
   - Duyệt cây
3. Cây tổng quát
   - Biểu diễn cây tổng quát
   - Duyệt cây tổng quát (nói qua)
4. Ứng dụng của cấu trúc cây
   - Cây biểu diễn biểu thức (tính giá trị, tính đạo hàm)
   - Cây quyết định

1. Định nghĩa và khái niệm

- Danh sách chỉ thể hiện được các mối quan hệ tuyến tính.
- Thông tin còn có thể có quan hệ dạng phi tuyến, ví dụ:
  - Các thư mục file
  - Các bước di chuyển của các quân cờ
  - Sơ đồ nhân sự của tổ chức
  - Cây phả hệ
- Sử dụng cây cho phép tìm kiếm thông tin nhanh

Cây là gì?

đỉr

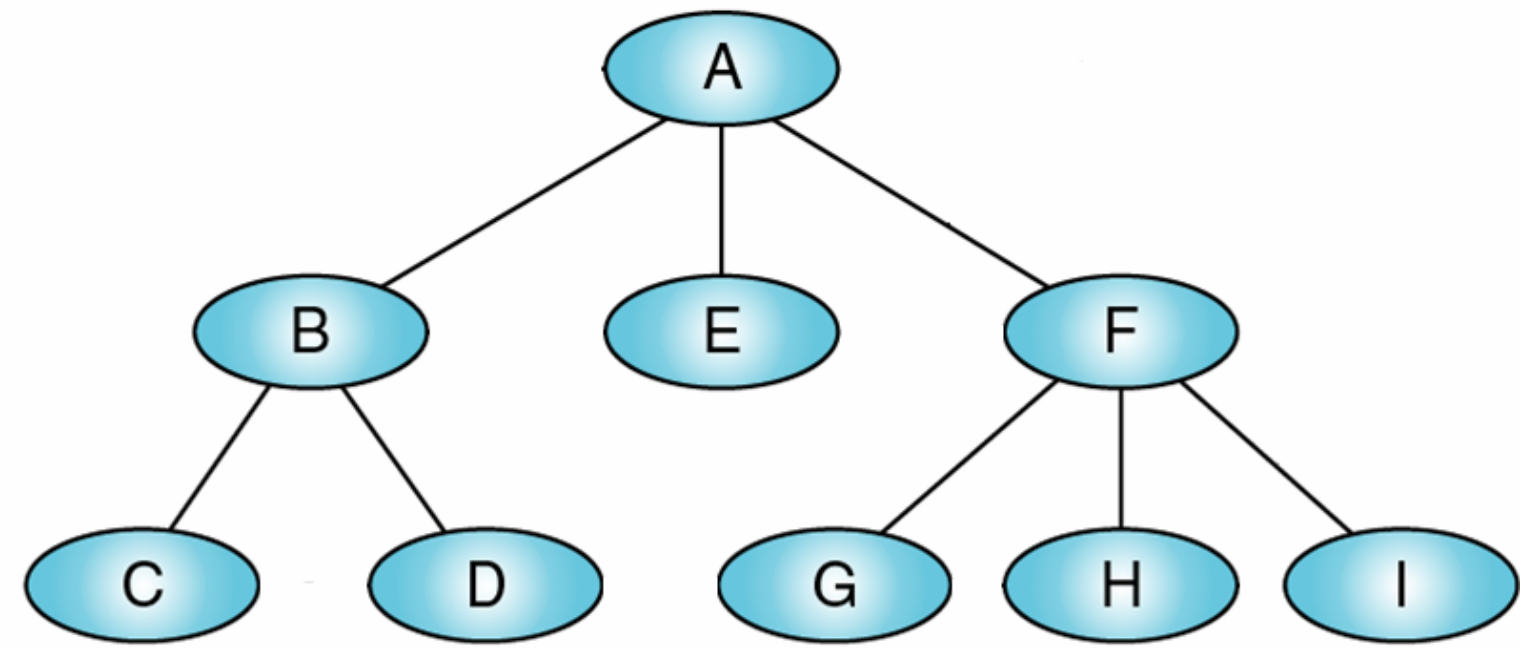
#cạ

Kết nố

Không có chu trình --- *T* sẽ chứa chu trình nếu thêm bất kỳ cạnh nào.

Cây là gì?

- Tập các nút (đỉnh), trong đó:
  - Hoặc là rỗng
  - Hoặc có một nút gốc và các cây con kết nối với nút gốc bằng một cạnh



Ví dụ: Cây thư mục

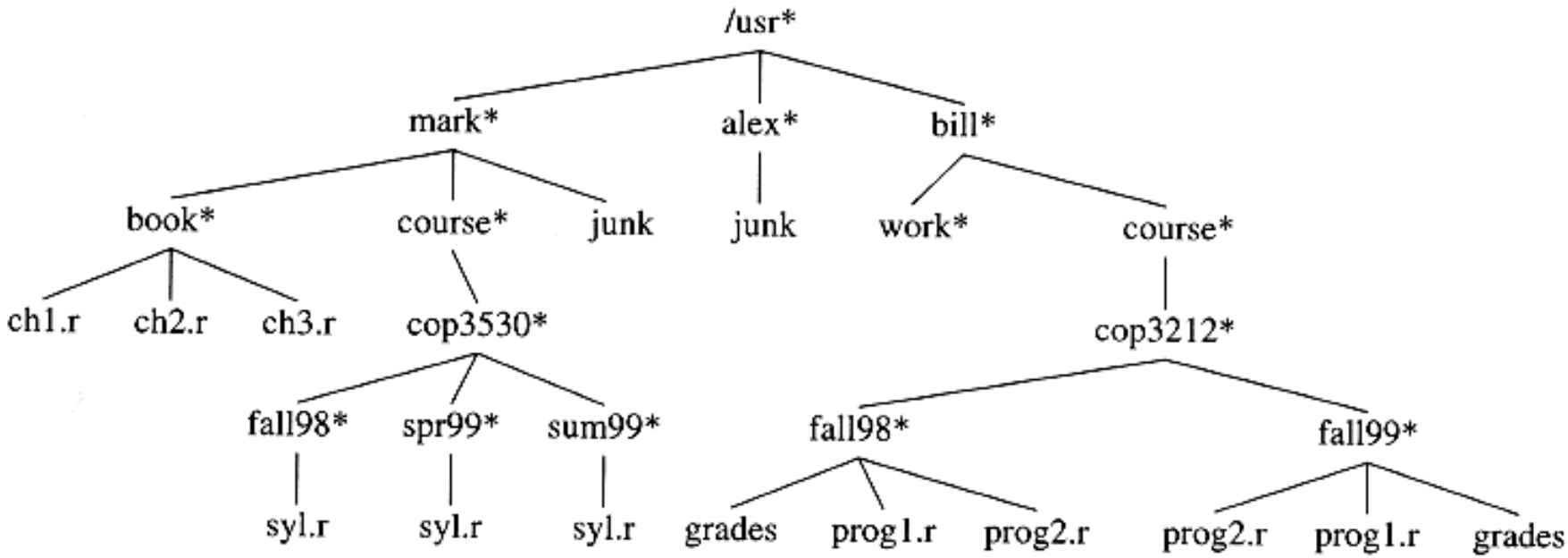


Figure 4.5 UNIX directory

Ví dụ: Cây biểu thức

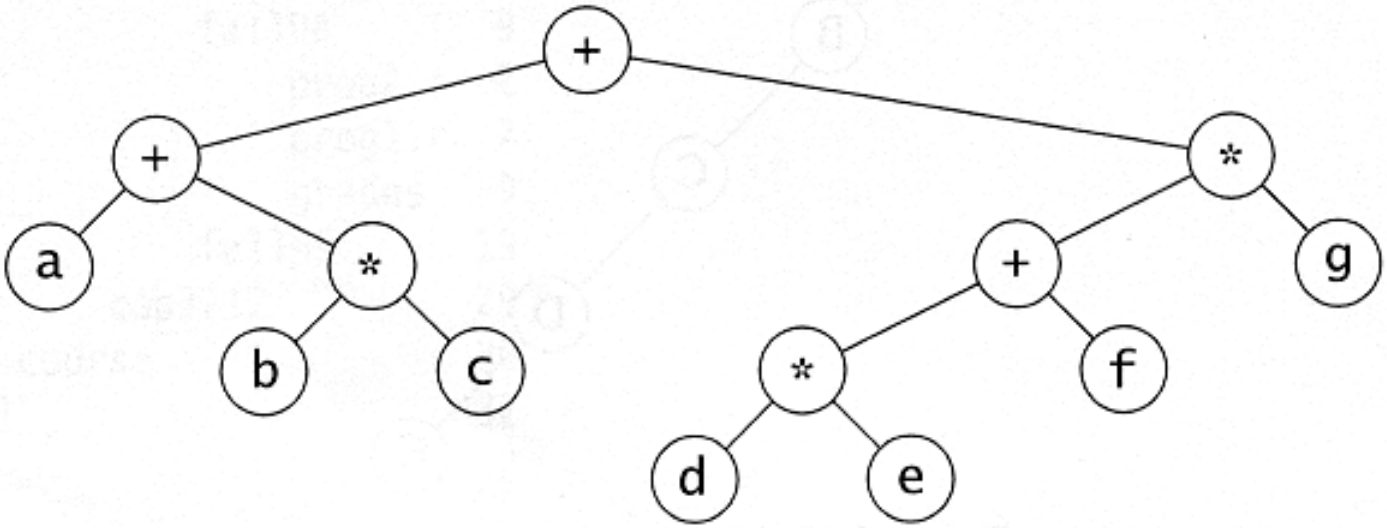
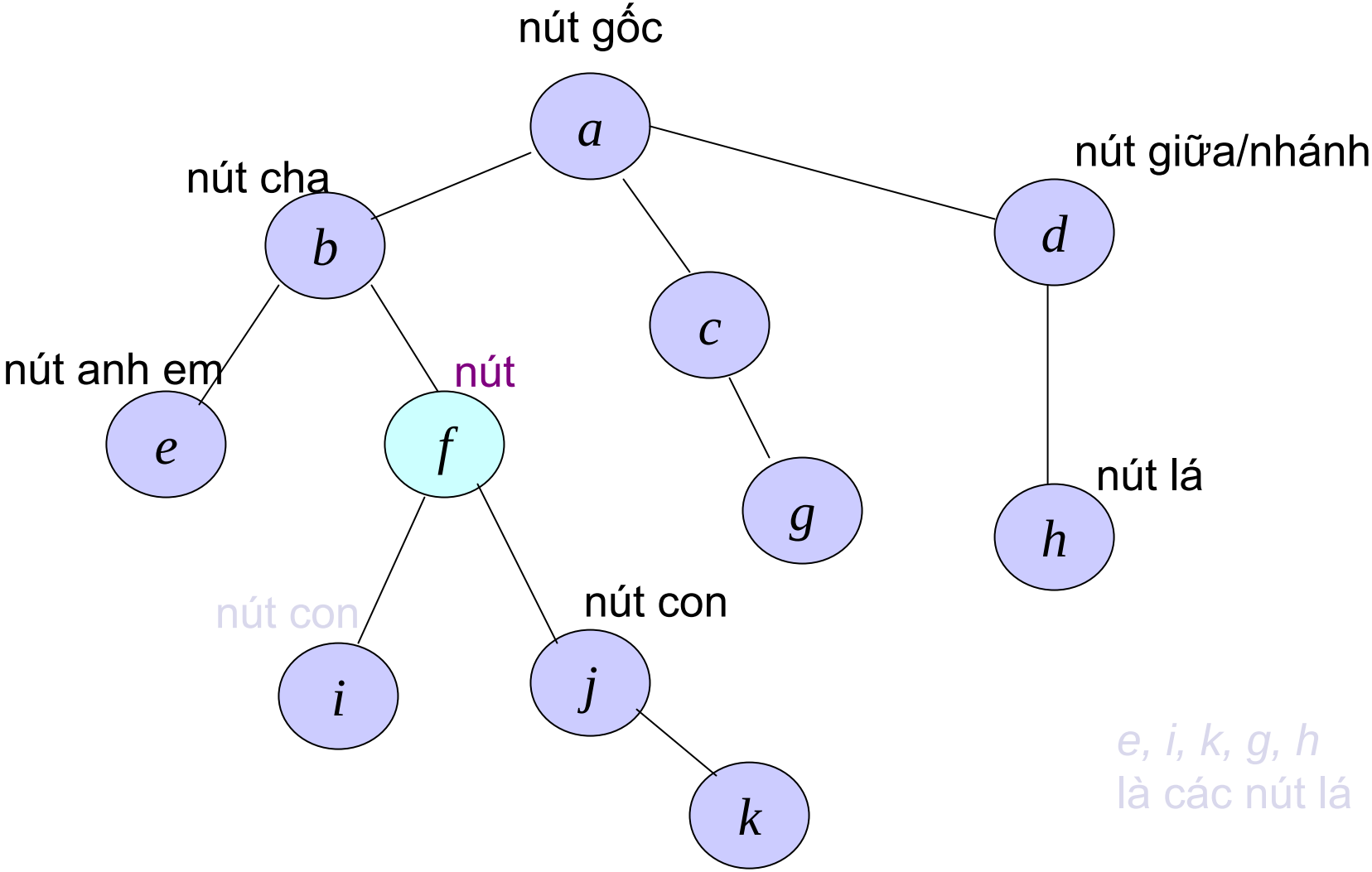
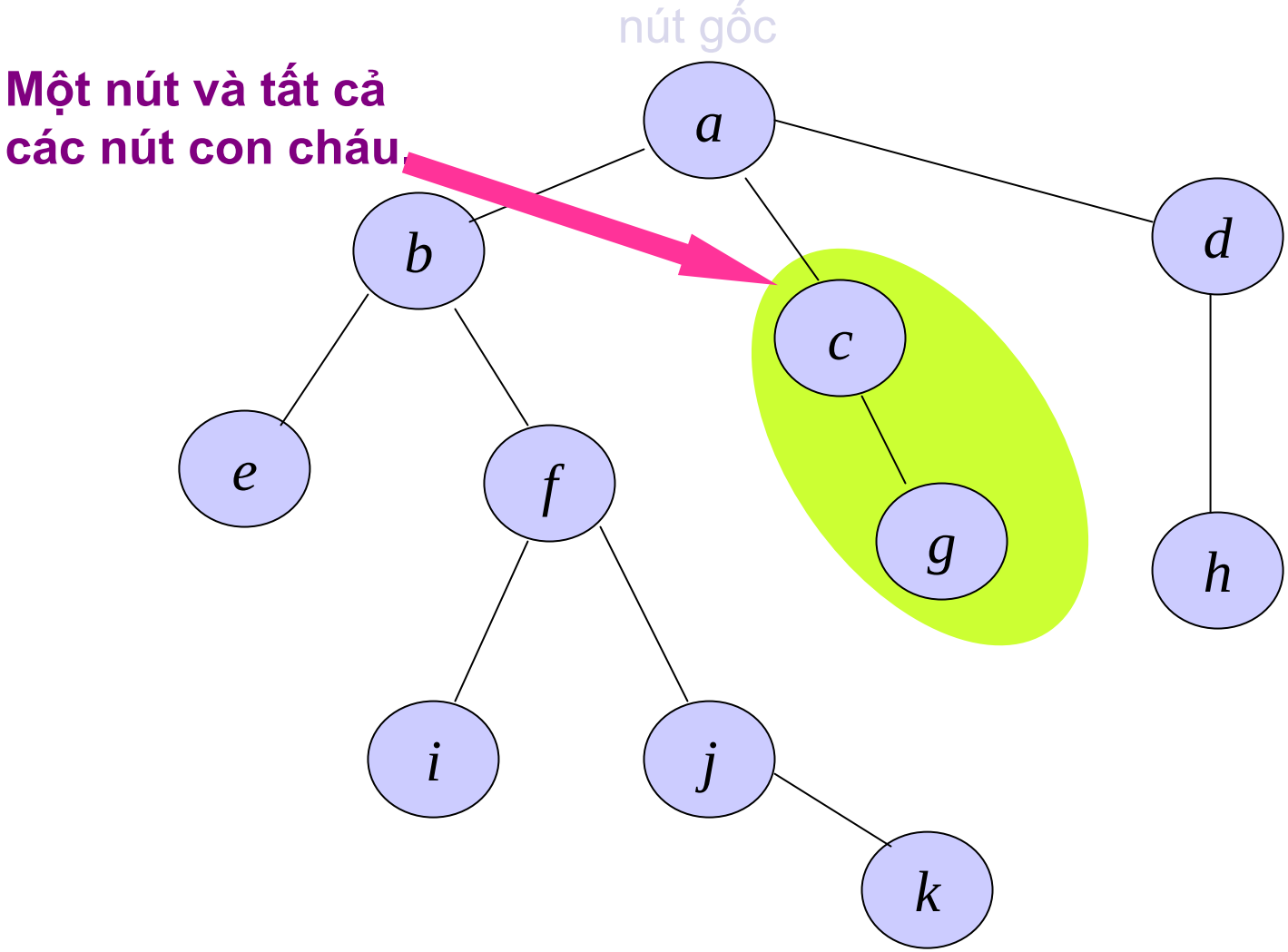


Figure 4.14 Expression tree for (a + b * c) + ((d * e + f) * g)

Các khái niệm



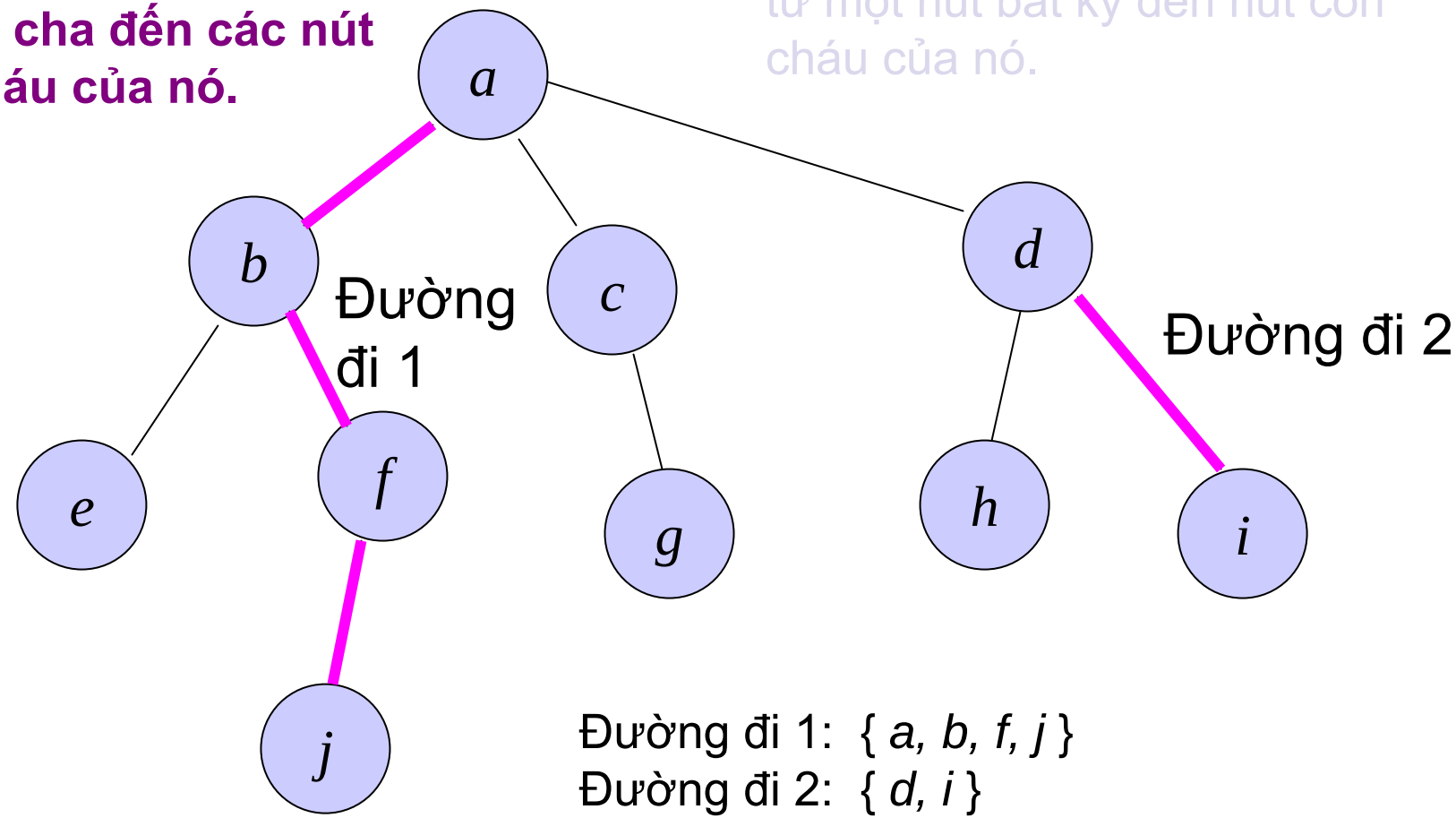
Cây con



Đường đi

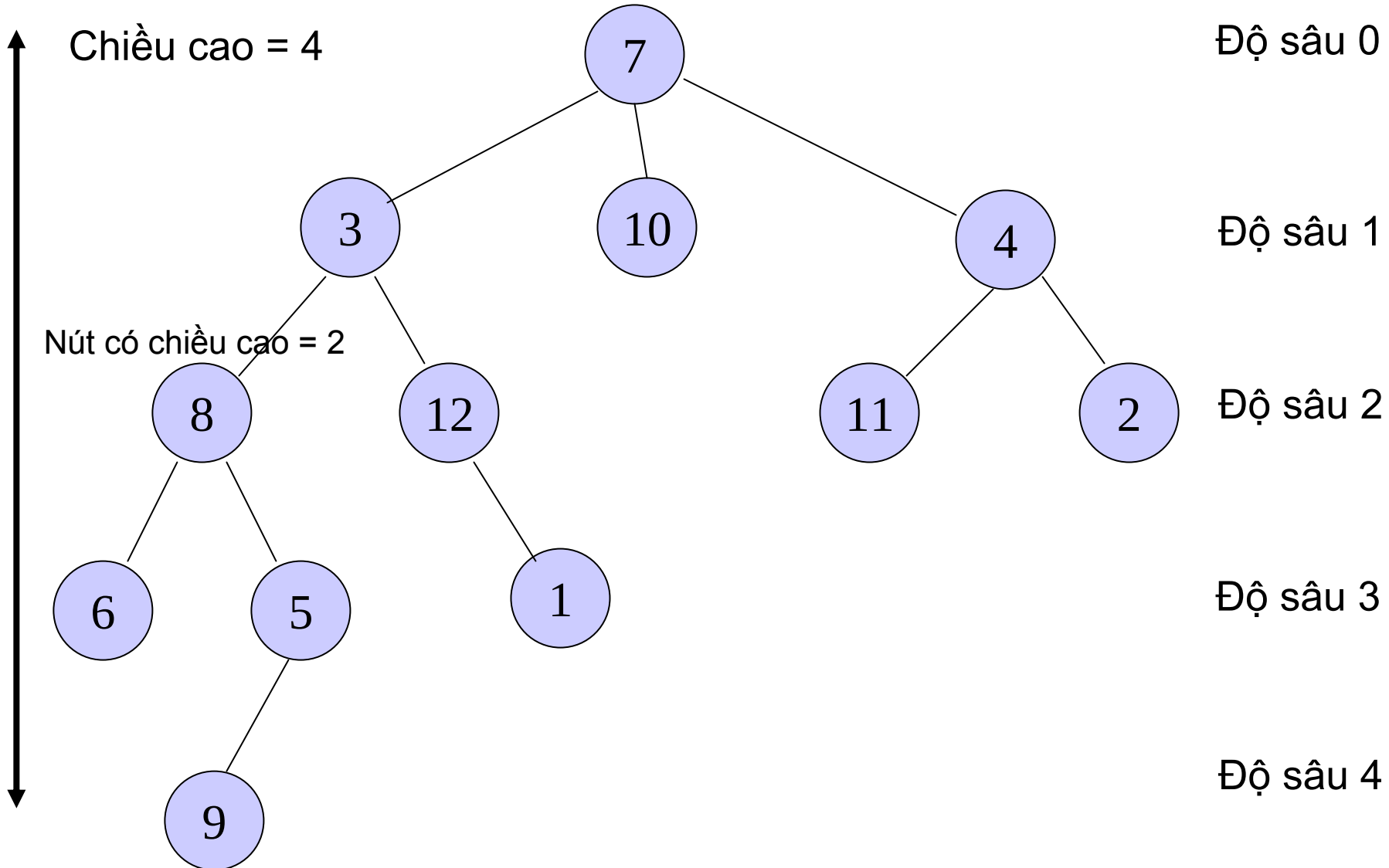
Tồn tại một đường đi duy nhất từ một nút bất kỳ đến nút con cháu của nó.

Từ nút cha đến các nút con cháu của nó.



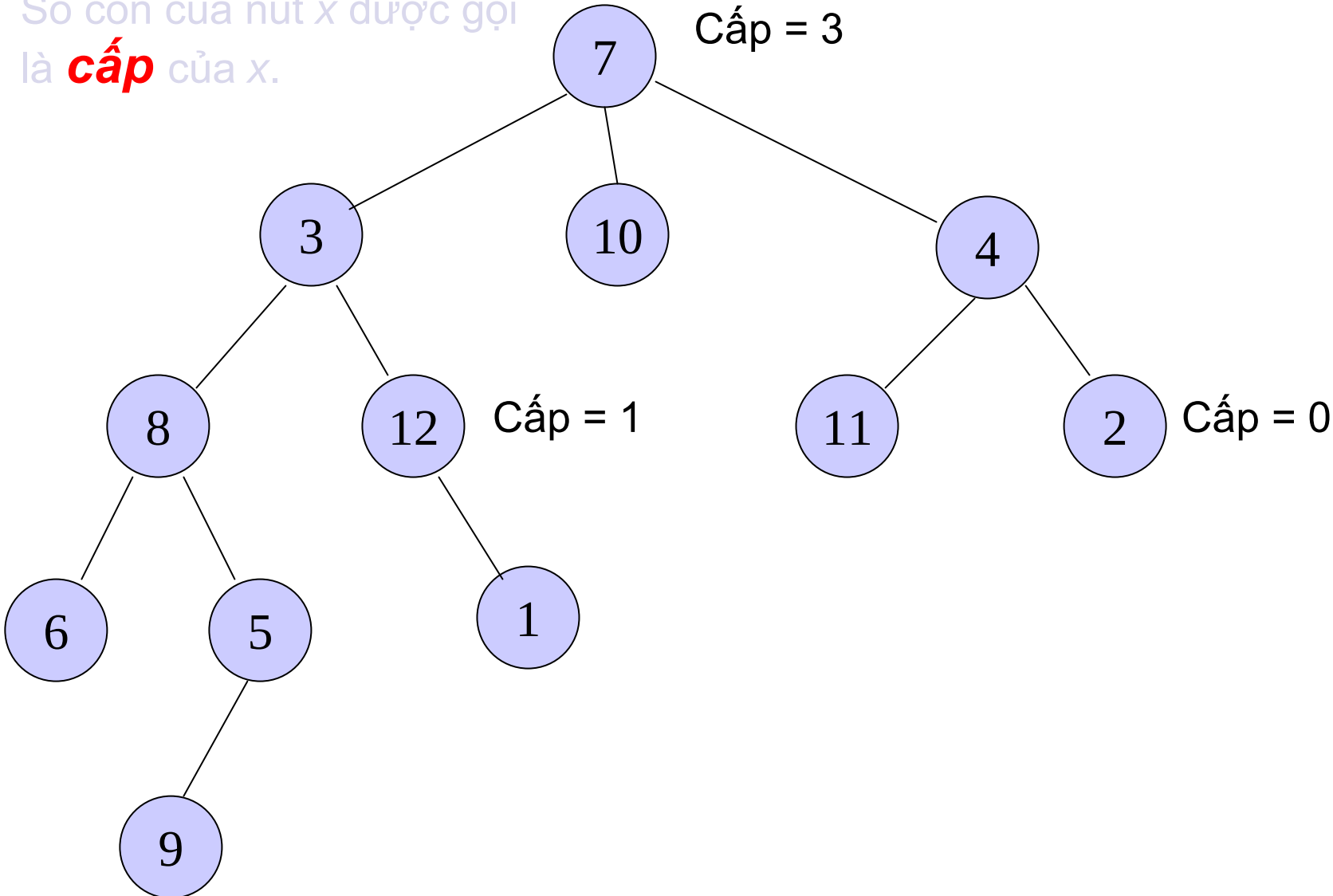
Đường đi 1: { *a, b, f, j* }
Đường đi 2: { *d, i* }

Độ sâu và độ cao



Cấp (degree)

Số con của nút *x* được gọi là ***cấp*** của *x*.



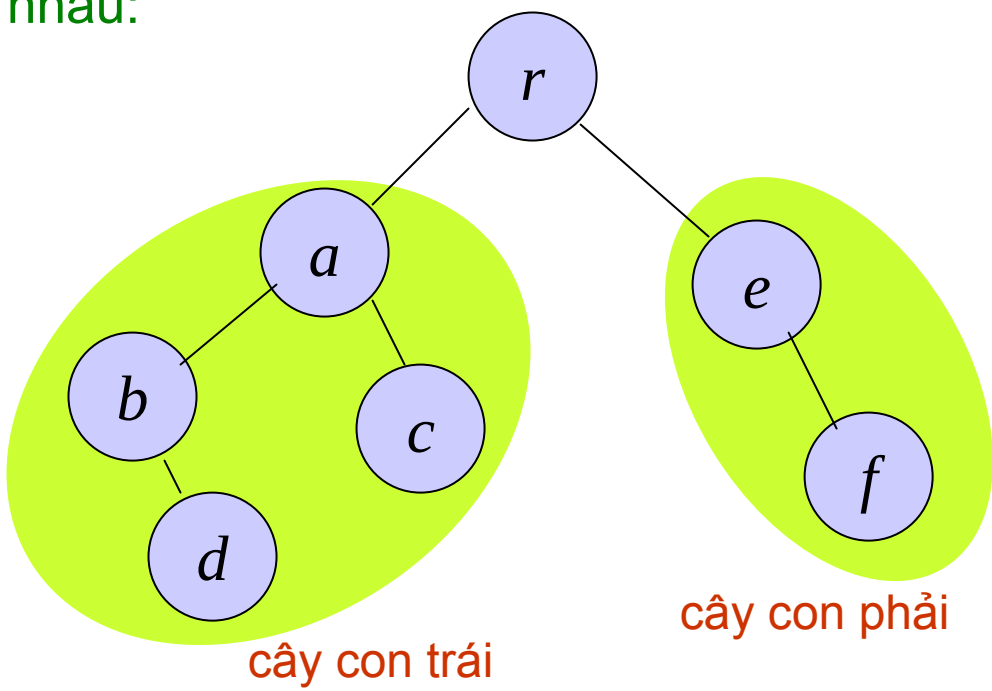
2. Cây nhị phân
2.1. Định nghĩa và tính chất

Mỗi nút có nhiều nhất 2 nút con. Con trái và Con phải

Một tập các nút *T* được gọi là cây nhị phân nếu

a) Nó là cây rỗng, hoặc

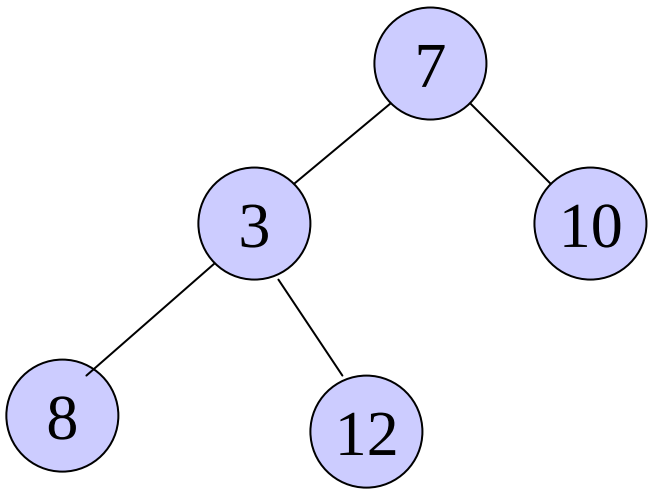
b) Gồm 3 tập con không trùng nhau:
1) một nút gốc
2) Cây nhị phân con trái
3) Cây nhị phân con phải



Cây nhị phân đầy đủ và Cây nhị phân hoàn chỉnh

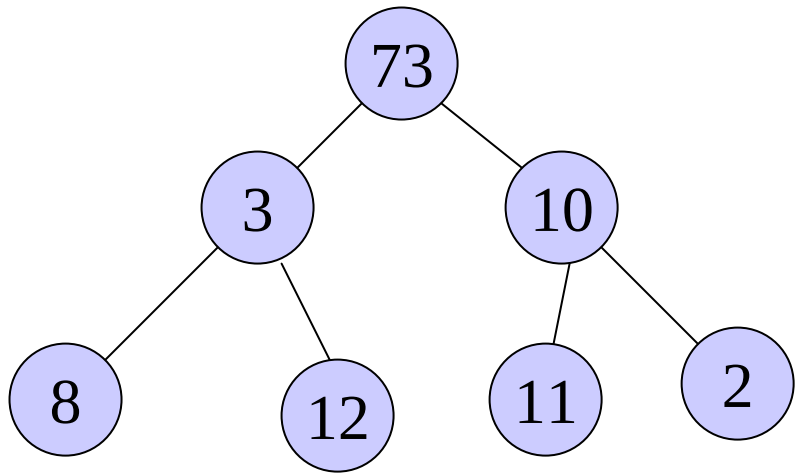
Cây nhị phân đầy đủ:

Các nút hoặc là nút lá hoặc có cấp = 2.

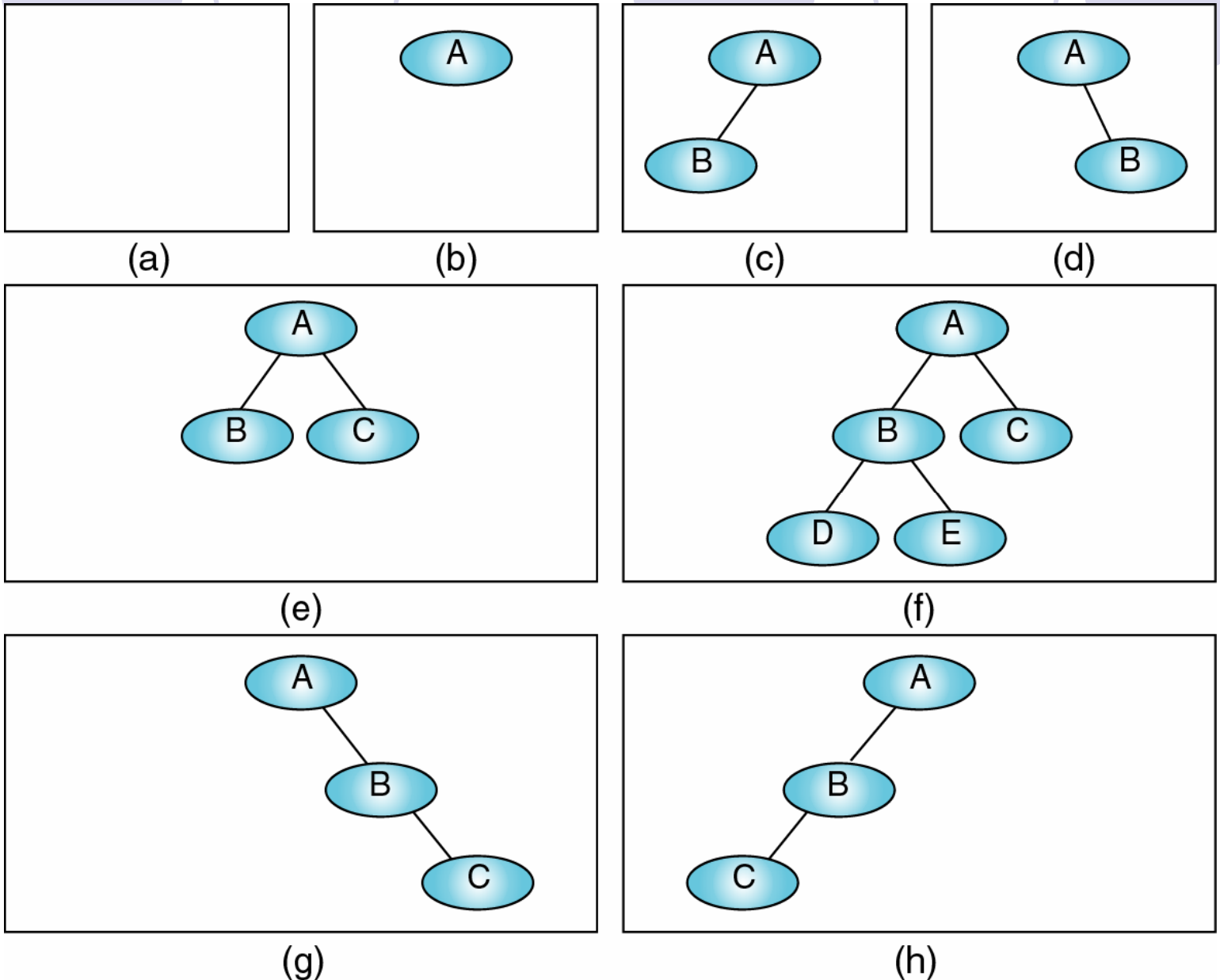


Cây nhị phân hoàn chỉnh:

Tất cả nút lá đều có cùng độ sâu và tất cả nút giữa có cấp = 2.



Một số dạng cây nhị phân



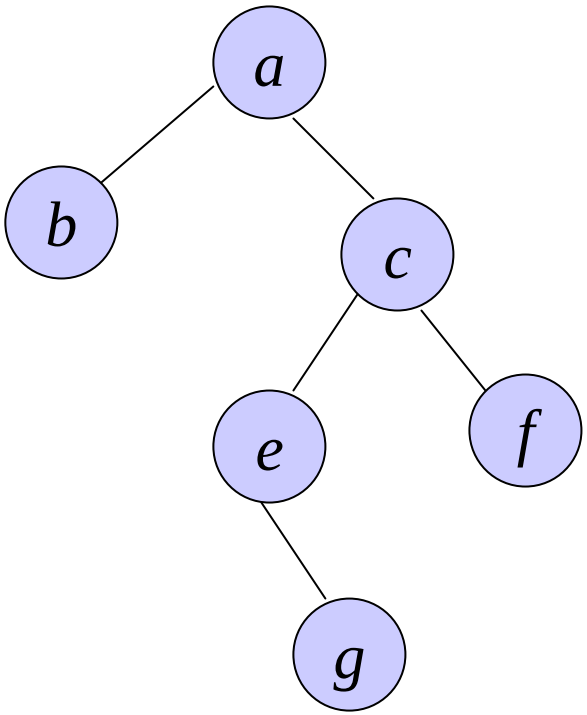
Một số tính chất

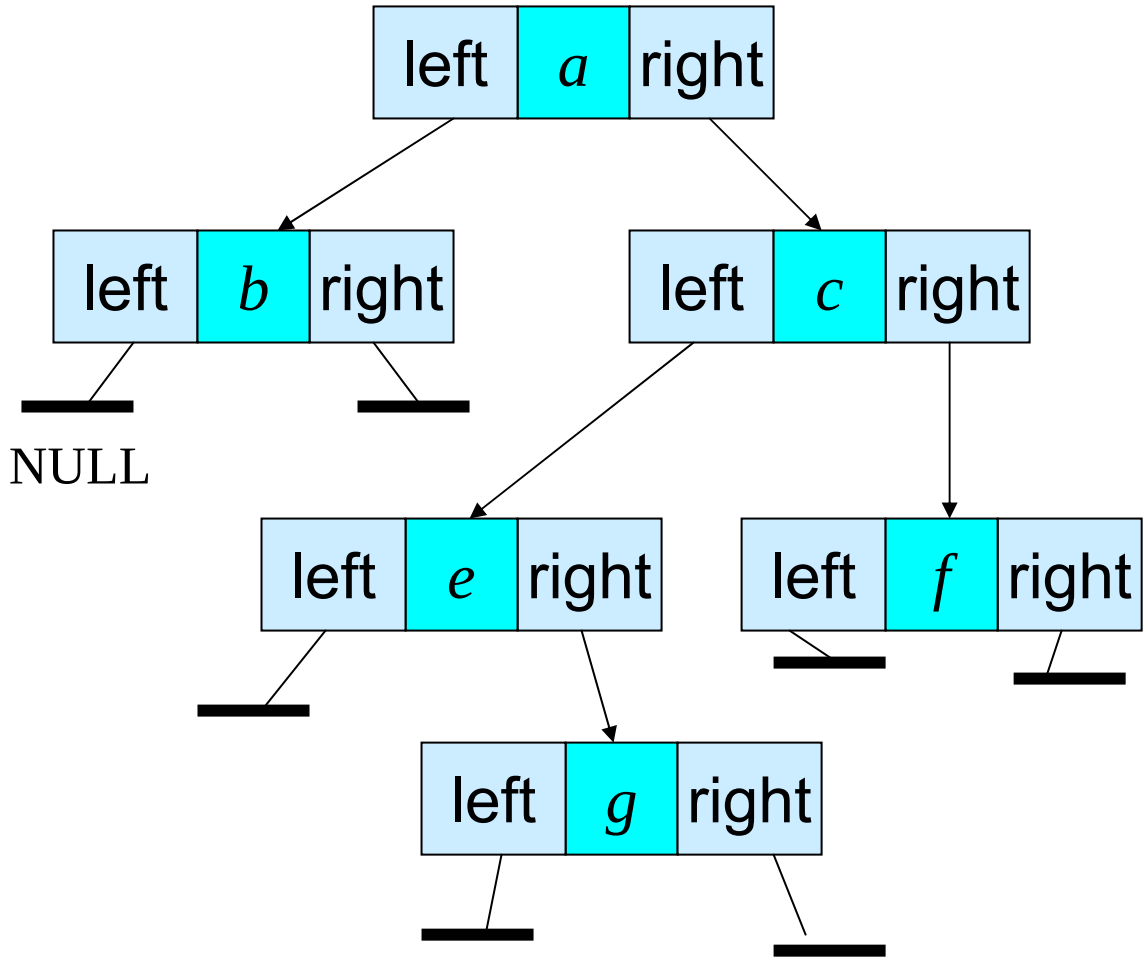
- Số nút tối đa có độ sâu i : 2^i
- Số nút tối đa (với cây nhị phân độ cao H) là: $2^{H+1} - 1$
- Độ cao (với cây nhị phân gồm N nút): H
  - Tối đa = N
  - Tối thiểu = $[\log_2(N+1)] - 1$

2.2 Lưu trữ cây nhị phân

- Lưu trữ kế tiếp:
  - Sử dụng mảng

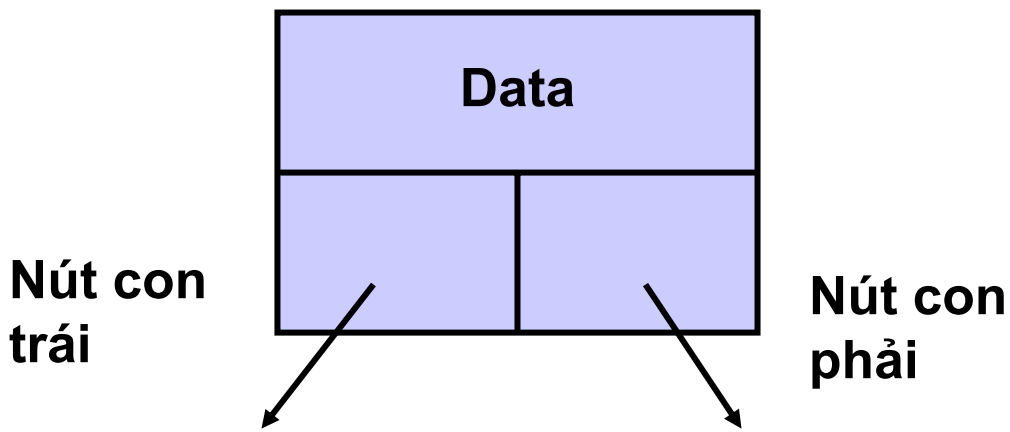
Lưu trữ móc nối





Xây dựng cấu trúc cây nhị phân

- Mỗi nút chứa :
  - Dữ liệu
  - 2 con trỏ trỏ đến 2 nút con của nó

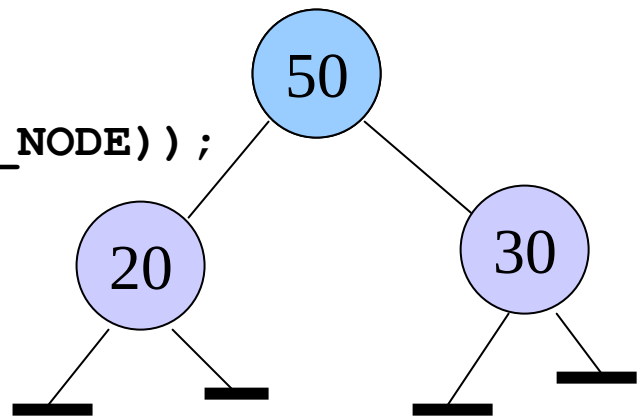


Cấu trúc cây nhị phân

```
typedef struct tree_node
{
    int data ;
    struct tree_node *left ;
    struct tree_node *right ;
} TREE_NODE;
```

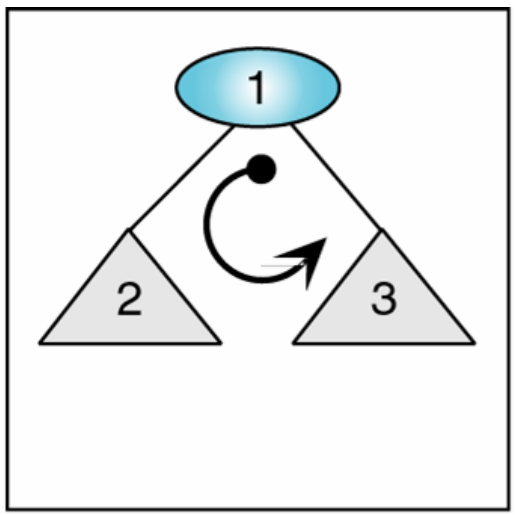
Tạo cây nhị phân

```
TREE_NODE *root, *leftChild, *rightChild;

// Tạo nút con trái
leftChild = (TREE_NODE*)malloc(sizeof(TREE_NODE));
leftChild->data = 20;
leftChild->left = leftChild->right = NULL;

// Tạo nút con phải
rightChild = (TREE_NODE*)malloc(sizeof(TREE_NODE));
rightChild->data = 30;
rightChild->left = rightChild->right = NULL;

// Tạo nút gốc
root = (TREE_NODE*)malloc(sizeof(TREE_NODE));
root->data = 10;
root->left = leftChild;
root->right = rightChild;


root -> data  = 50; // gán 50 cho root
```

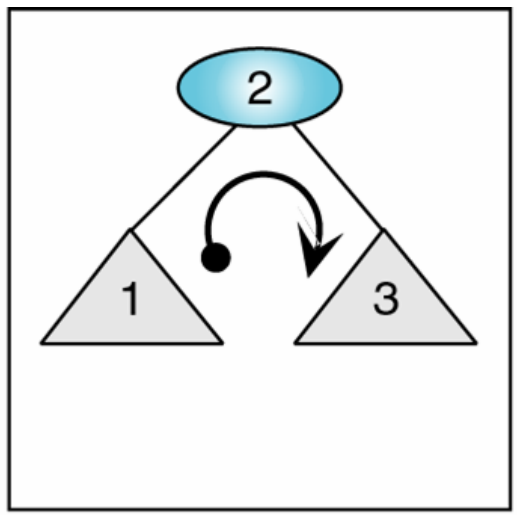


2.3. Duyệt cây nhị phân

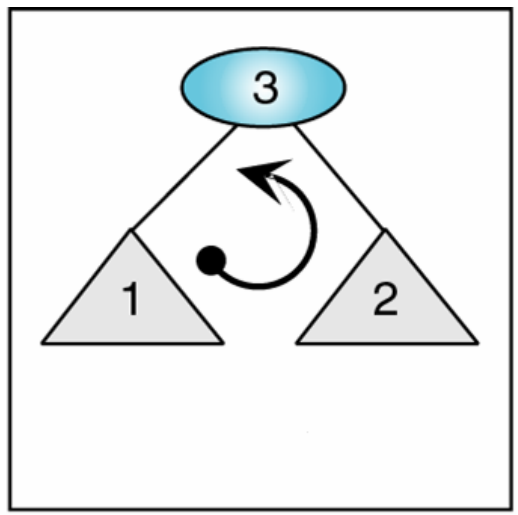
- Duyệt cây: lần lượt duyệt toàn bộ nút trên cây
- Có 3 cách duyệt cây :
  - Duyệt theo thứ tự trước
  - Duyệt theo thứ tự giữa
  - Duyệt theo thứ tự sau
- Định nghĩa duyệt cây nhị phân là những định nghĩa đệ quy.



(a) Thứ tự trước



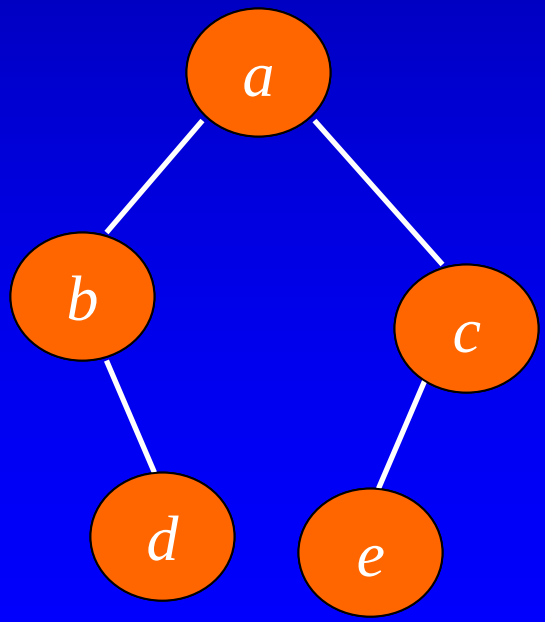
(b) Thứ tự giữa



(c) Thứ tự sau

Duyệt theo thứ tự trước

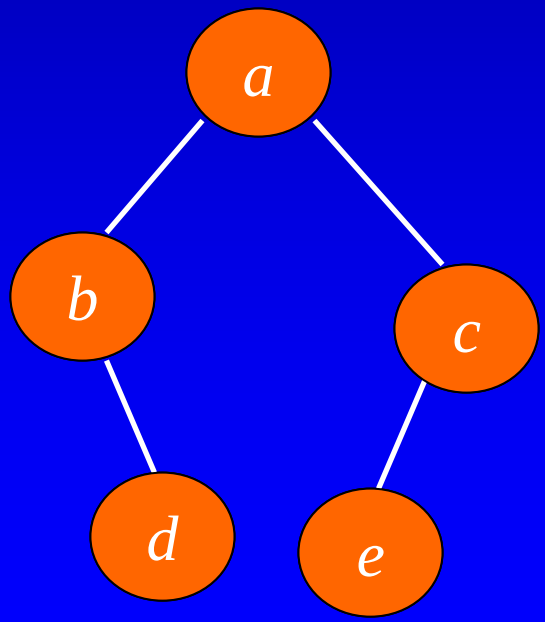
1. Thăm nút.
2. Duyệt cây con trái theo thứ tự trước.
3. Duyệt cây con phải theo thứ tự trước.



Traversal order: *abdce*

Duyệt theo thứ tự sau

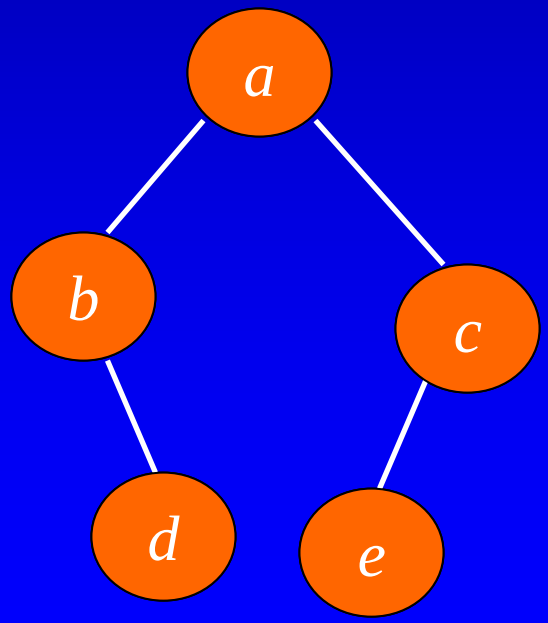
1. Duyệt cây con trái theo thứ tự sau.
2. Duyệt cây con phải theo thứ tự sau.
3. Thăm nút.



Thứ tự duyệt: *dbeca*

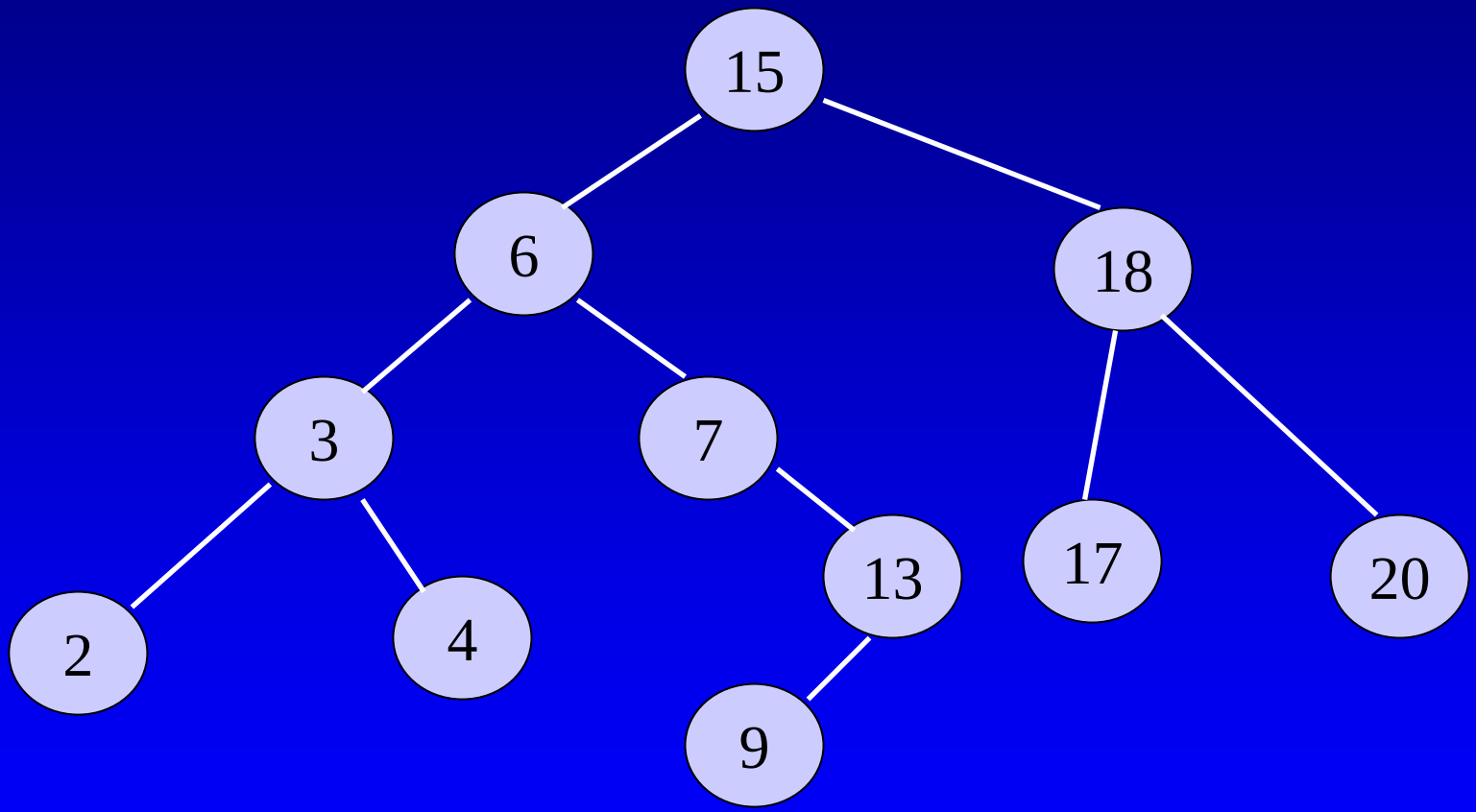
Duyệt theo thứ tự giữa

1. Duyệt cây con trái theo thứ tự giữa
2. Thăm nút.
3. Duyệt cây con phải theo thứ tự giữa.



Thứ tự duyệt: *bdaec*

Ví dụ



Thứ tự trước: 15, 6, 3, 2, 4, 7, 13, 9, 18, 17, 20
Thứ tự giữa: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20
Thứ tự sau: 2, 4, 3, 9, 13, 7, 6, 17, 20, 18, 15

Duyệt theo thứ tự trước – Đệ quy

```
void Preorder(TREE_NODE* root)
{
    if (root!=NULL)
    {
        // tham aNode
        printf("%d ", root->data);
        // duyet cay con trai
        Preorder(root->left);
        // duyet cay con phai
        Preorder(root->right);
    }
}
```

- Bài tập: Viết giải thuật đệ quy của
  - Duyệt theo thứ tự giữa
  - Duyệt theo thứ tự sau

Duyệt theo thứ tự trước – Vòng lặp

```
void Preorder_iter(TREE_NODE* treeRoot)
{
    TREE_NODE* curr = treeRoot;
    STACK* stack = createStack(MAX); // khởi tạo stack
    while (curr!=NULL || !IsEmpty(stack))
    {
       printf("%d ", curr->data); // thăm curr
       // nếu có cây con phải, đẩy cây con phải vào stack
       if (curr->right!=NULL)
           pushStack(stack, curr->right);
       if (curr->left!=NULL)
           curr = curr->left; // duyệt cây con trái
       else
           popStack(stack, &curr);// duyệt cây con phải
    }
    destroyStack(&stack);       // giải phóng stack
}
```

Duyệt theo thứ tự giữa

```
void Inorder_iter(TREE_NODE* root){
   TREE_NODE* curr = root;
   STACK* stack = createStack(MAX);// ktạo stack
   while (curr!=NULL || !IsEmpty(stack))
   {
        if (curr==NULL){
            popStack(stack, &curr);
            printf("%d", curr->data);
            curr = curr->right;
        }
        else
        {
            pushStack(stack, curr);
            curr = curr->left; // duyệt cây con trái
        }
   }
   destroyStack(stack);// giải phóng stack
}
```

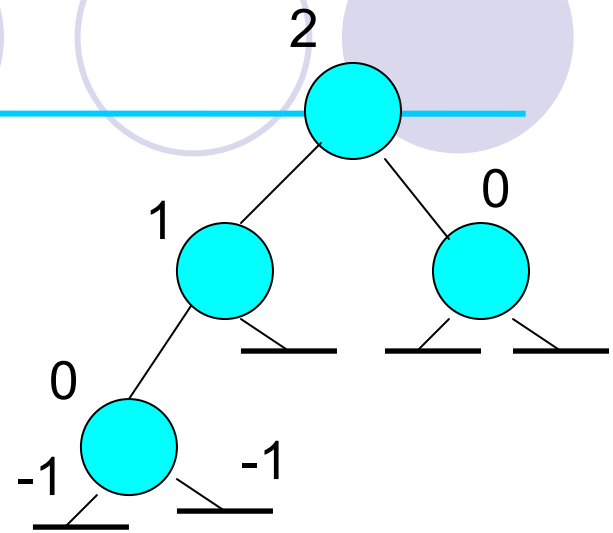
Duyệt theo thứ tự cuối

```
void Postorder_iter(TREE_NODE* treeRoot)
{
   TREE_NODE* curr = treeRoot;
   STACK* stack = createStack(MAX);// ktạo một stack

   while(curr != NULL || !IsEmpty(stack)) {
       if (curr == NULL) {

            while(!IsEmpty(stack) && curr==Top(stack)->right)
                PopStack(stack, &curr);
                printf("%d", curr->data);
            }

            curr = isEmpty(stack)? NULL: Top(stack)->right;
       }
       else {
            PushStack(stack, curr);
            curr = curr->left;
       }
   }
   destroyStack(&stack); // giải phóng stack
}
```

Một vài ứng dụng của phương pháp duyệt cây

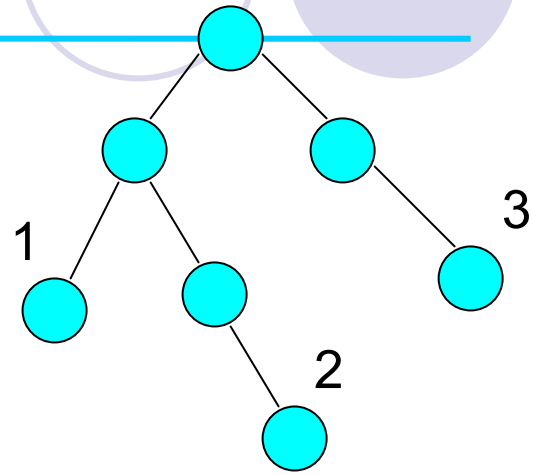
1. Tính độ cao của cây
2. Đếm số nút lá trong cây
3. Tính kích thước của cây (số nút)
4. Sao chép cây
5. Xóa cây
6. …

Tính độ cao của cây




```
int Height(TREE_NODE *tree)
{
    int heightLeft, heightRight, heightval;
    if ( tree == NULL )
       heightval = -1;
    else
    {  // Sử dụng phương pháp duyệt theo thứ tự sau
       heightLeft = Height (tree->left);
       heightRight = Height (tree->right);
       heightval = 1 + max(heightLeft, heightRight);
    }
    return heightval;
}
```

Đếm số nút lá

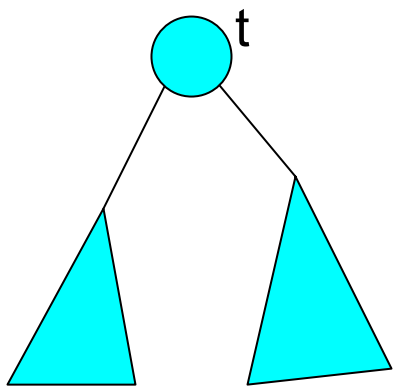


thứ tự đếm

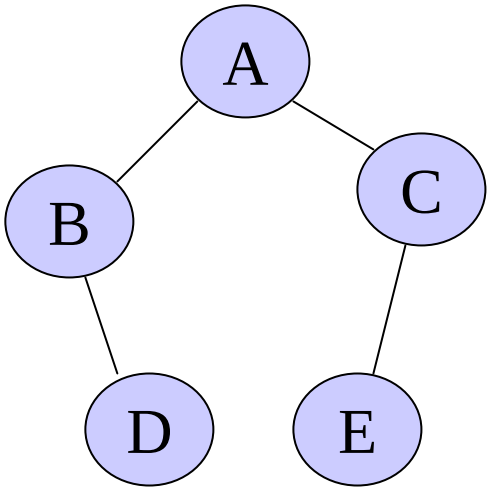
```
int CountLeaf(TREE_NODE *tree)
{
      if (tree == NULL)
            return 0;
      int count = 0;
      // Đếm theo thứ tự sau
      count +=  CountLeaf(tree->left);    // Đếm trái
      count += CountLeaf(tree->right);   // Đếm phải

      // nếu nút tree là nút lá, tăng count
      if (tree->left == NULL && tree->right == NULL)
            count++;
      return count;
}
```

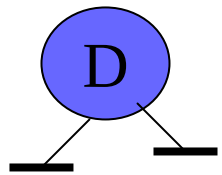
Kích thước của cây

```
int TreeSize(TREE_NODE *tree)
{
      if (tree == NULL)
            return 0;
      else
            return (  TreeSize(tree->left) +
                      TreeSize(tree->right) + 1 );
}
```

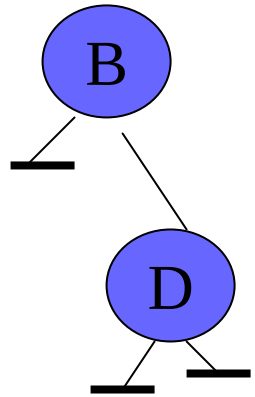


Sao chép cây

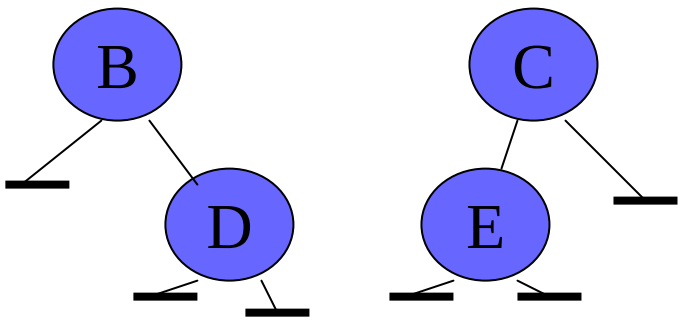




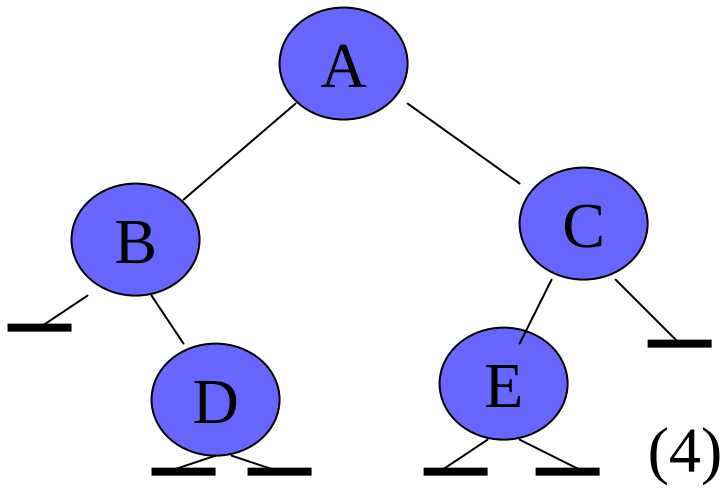
(1)



(2)



(3)



(4)

Sao chép cây

```
TREE_NODE* CopyTree(TREE_NODE *tree)
{
      // Dừng đệ quy khi cây rỗng
      if (tree == NULL)      return NULL;

      TREE_NODE *leftsub, *rightsub, *newnode;
      leftsub  = CopyTree(tree->left);
      rightsub = CopyTree(tree->right);

      // tạo cây mới
      newnode =  malloc(sizeof(TREE_NODE));
      newnode->data  =  tree->data;
      newnode->left  =  leftsub;
      newnode->right =  rightsub;
      return newnode;
}
```

Xóa cây

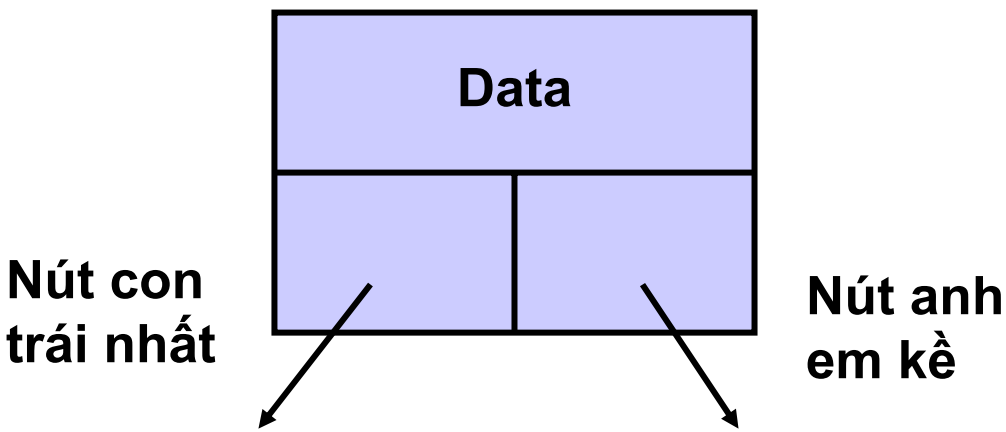
```
void DeleteTree(TREE_NODE *tree)
{
      // xóa theo thứ tự sau
      if (tree != NULL)
      {
            DeleteTree(tree -> left);
            DeleteTree(tree -> right);
            free (tree);
      }
}
```

3. Cây tổng quát
3.1. Biểu diễn cây tổng quát

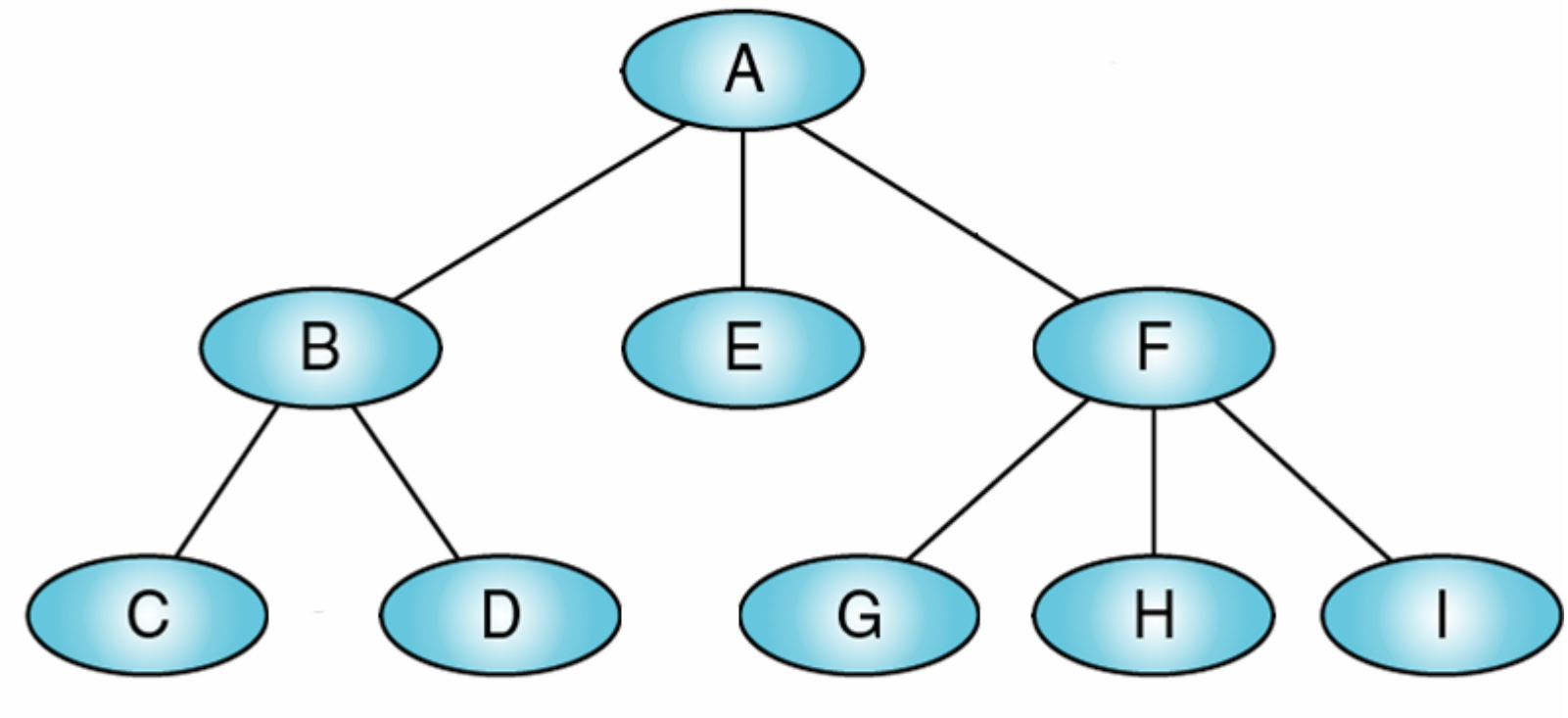
- Biểu diễn giống như cây nhị phân?
  - Mỗi nút sẽ chứa giá trị và các con trỏ trỏ đến các nút con của nó?
  - Bao nhiêu con trỏ cho một nút? – Không hợp lý
- Mỗi nút sẽ chứa giá trị và một con trỏ trỏ đến một "tập" các nút con
  - Xây dựng "tập" như thế nào?

Biểu diễn cây tổng quát

- Sử dụng con trỏ nhưng mở rộng hơn:
  - Mỗi nút sẽ có 2 con trỏ: một con trỏ trỏ đến nút con đầu tiên của nó, con trỏ kia trỏ đến nút anh em kề với nó
  - Cách này cho phép quản lý số lượng tùy ý của các nút con



Ví dụ



3.2. Duyệt cây tổng quát

1. Thứ tự trước:
   1. Thăm gốc
   2. Duyệt cây con thứ nhất theo thứ tự trước
   3. Duyệt các cây con còn lại theo thứ tự trước
2. Thứ tự giữa
   1. Duyệt cây con thứ nhất theo thứ tự giữa
   2. Thăm gốc
   3. Duyệt các cây con còn lại theo thứ tự giữa
3. Thứ tự sau:
   1. Duyệt cây con thứ nhất theo thứ tự sau
   2. Duyệt các cây con còn lại theo thứ tự sau
   3. Thăm gốc

4. Ứng dụng của cây nhị phân

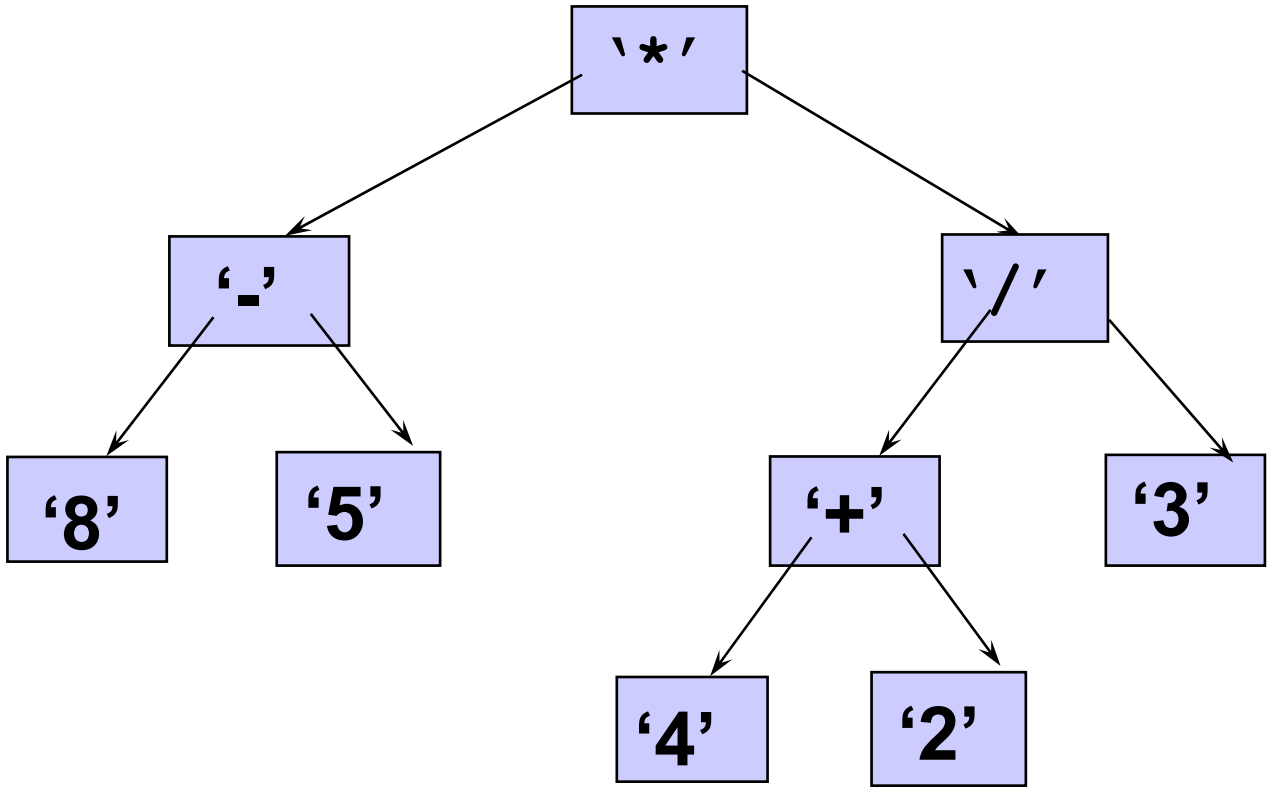
- Cây biểu diễn biểu thức
  - Tính giá trị biểu thức
  - Tính đạo hàm
- Cây quyết định

Cây biểu diễn biểu thức là . . .

Một loại cây nhị phân đặc biệt, trong đó:

1. **Mỗi nút lá chứa một toán hạng**
2. **Mỗi nút giữa chứa một toán tử**
3. **Cây con trái và phải của một nút toán tử thể hiện các biểu thức con cần được đánh giá trước khi thực hiện toán tử tại nút gốc**

Biểu thức nhị phân



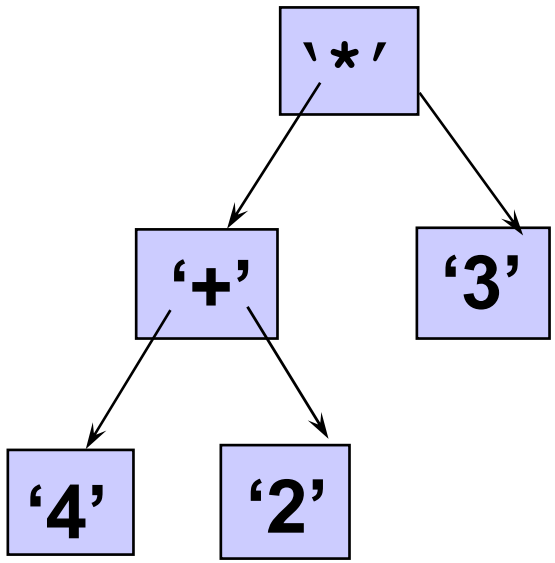
Các mức chỉ ra thứ tự ưu tiên

Các mức(độ sâu) của các nút chỉ ra thứ tự ưu tiên tương đối của chúng trong biểu thức (không cần dùng ngoặc để thể hiện thứ tự ưu tiên).

Các phép toán tại mức cao hơn sẽ được tính sau các các phép toán có mức thấp.

Phép toán tại gốc luôn được thực hiện cuối cùng.

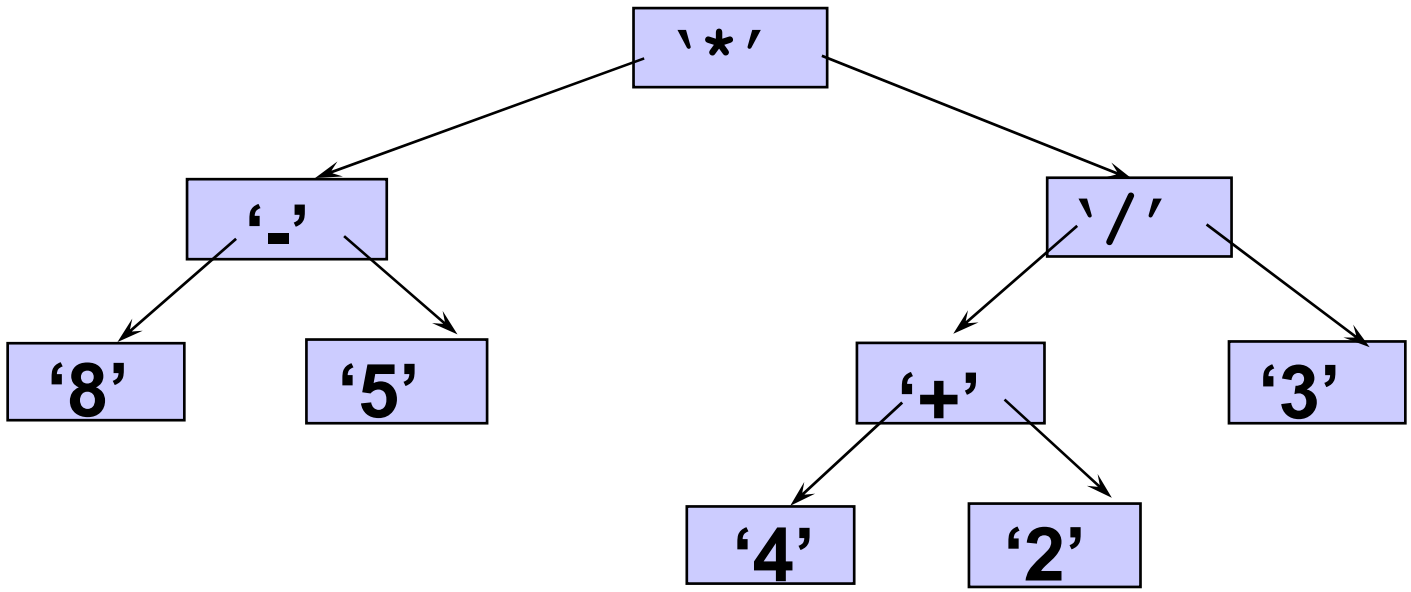
Cây biểu diễn biểu thức



Giá trị kết quả?

(4 + 2) * 3 = 18

Dễ dàng để tạo ra các biểu thức tiền tố, trung tố, hậu tố

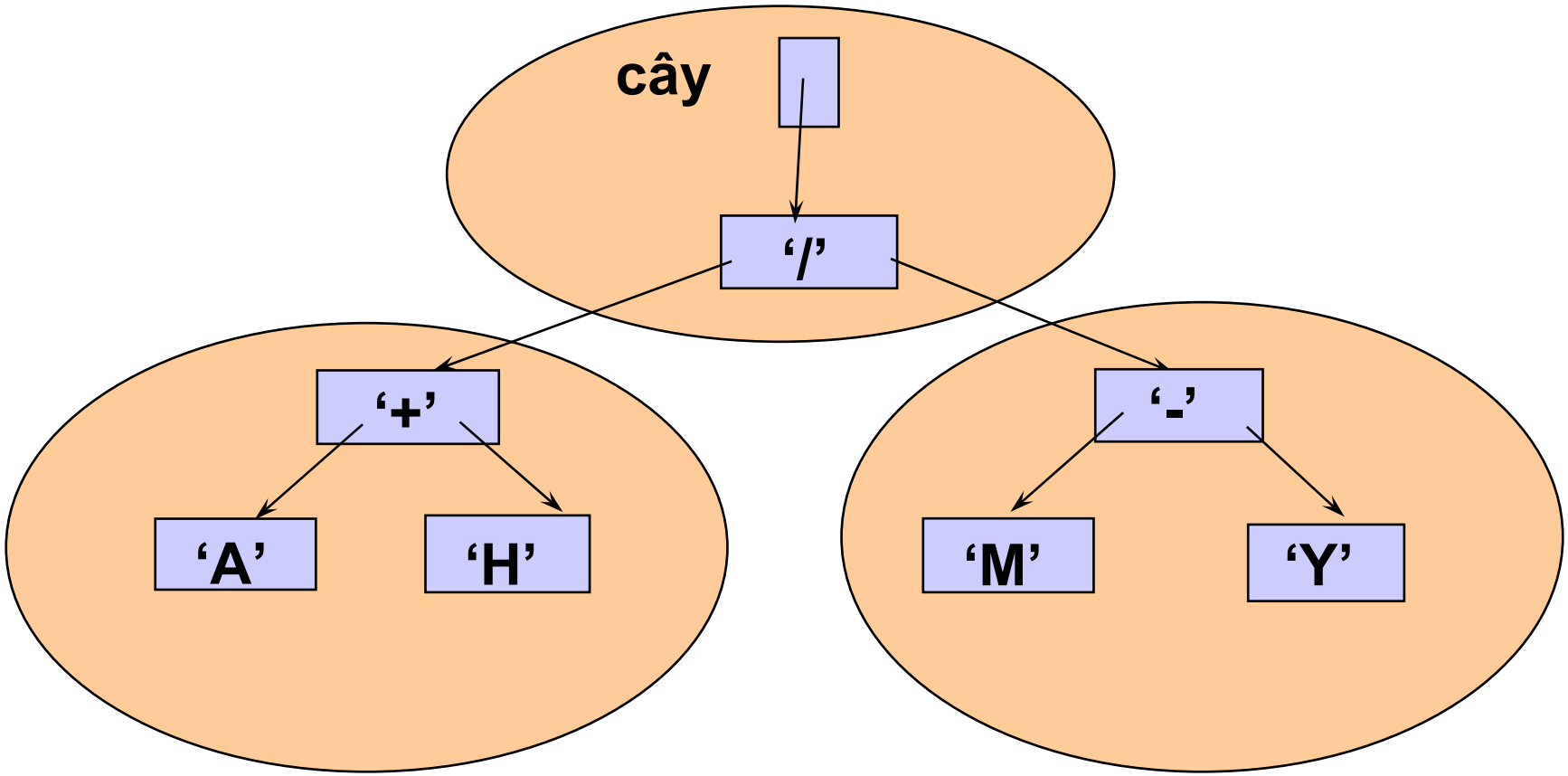


Trung tố: **((8 - 5) * ((4 + 2) / 3))**

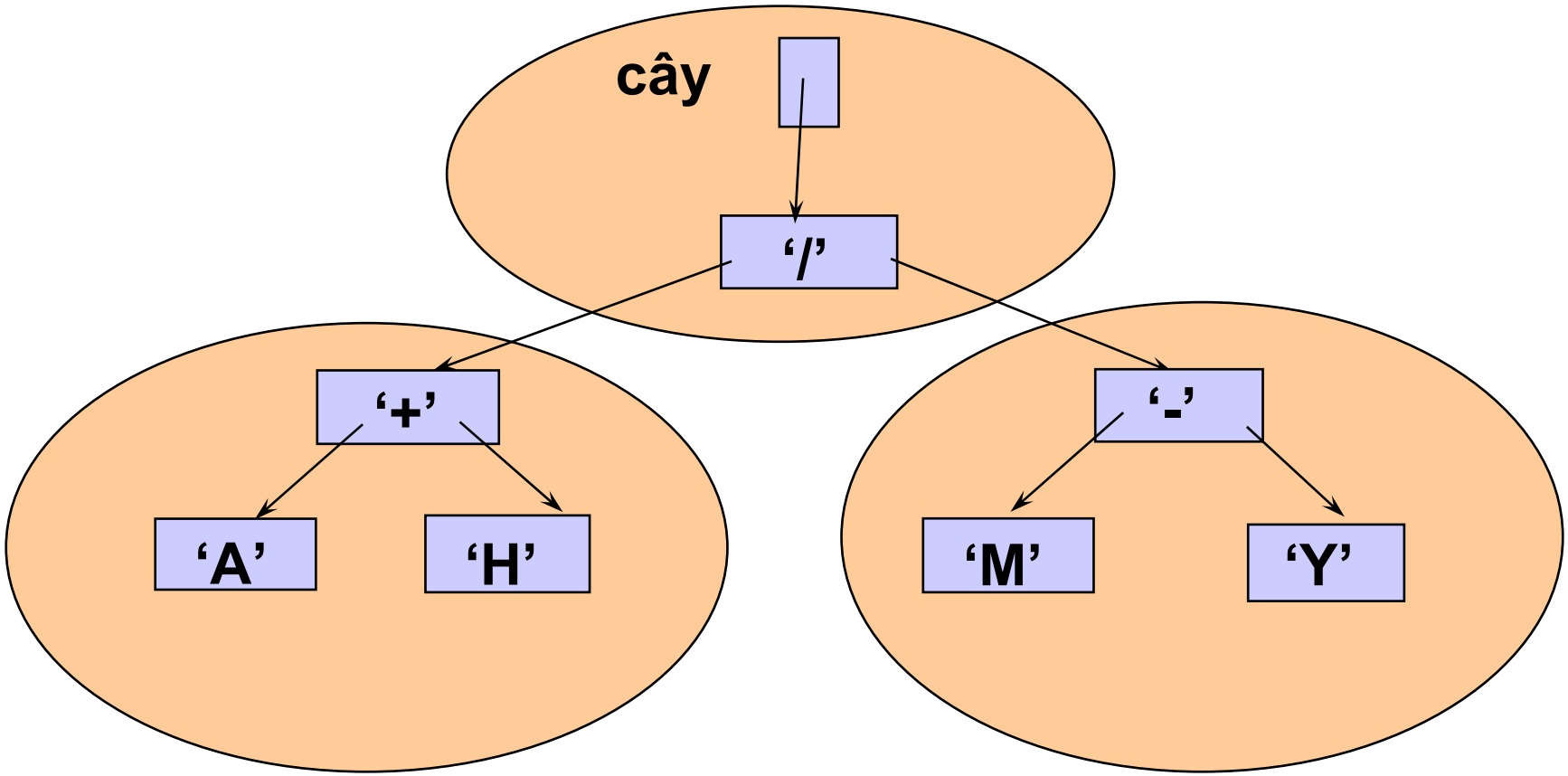
Tiền tố: *** - 8 5 / + 4 2 3**

Hậu tố: **8 5 - 4 2 + 3 / ***

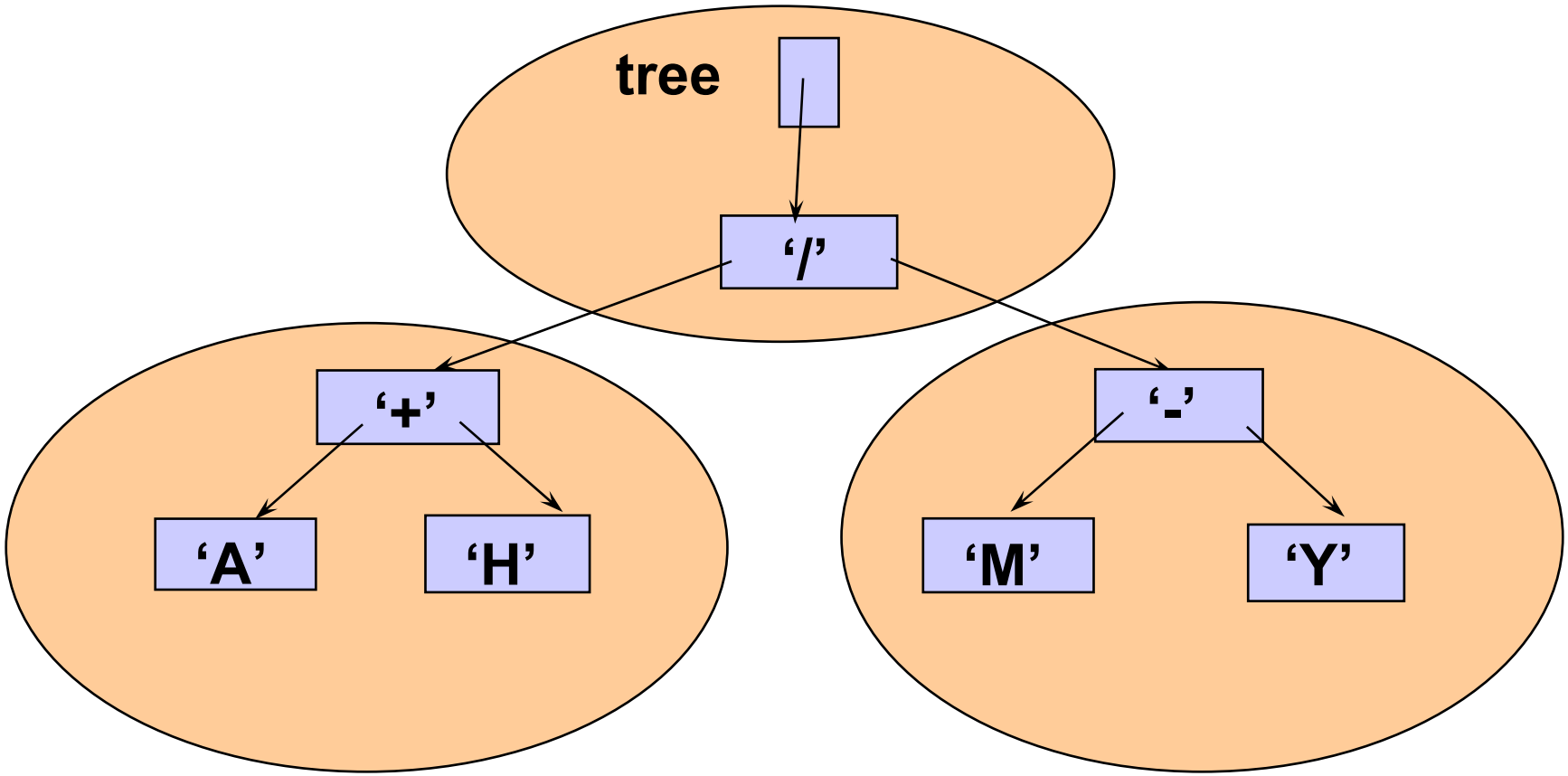
Duyệt theo thứ tự giữa
(A + H) / (M - Y)



Duyệt theo thứ tự trước
/ + A H - M Y

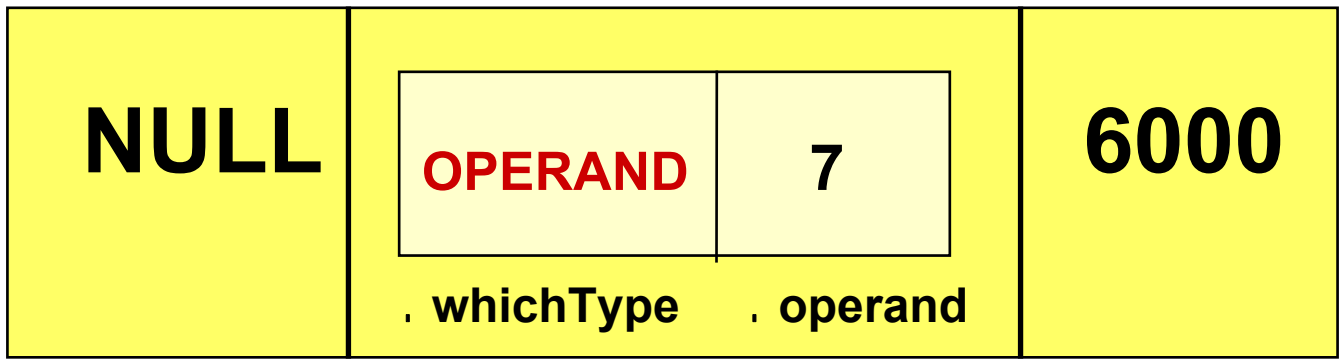


Duyệt theo thứ tự sau
A H + M Y - /



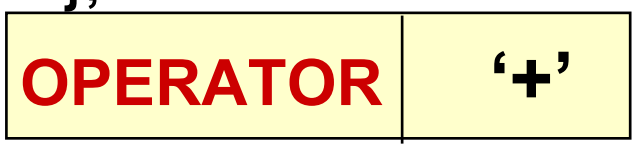
Mỗi nút có 2 con trỏ

```
struct   TreeNode
{
   InfoNode     info ;          // Dữ liệu
   TreeNode*    left ;          // Trỏ tới nút con trái
   TreeNode*    right ;         // Trỏ tới nút con phải
};
```

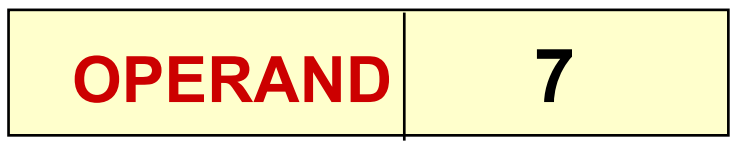


InfoNode có 2 dạng

```
enum  OpType  { OPERATOR,  OPERAND } ;

struct   InfoNode
{
   OpType       whichType;
   union                                    // ANONYMOUS union
   {
         char    operator ;
         int     operand ;
   }
};
```



. whichType . operation

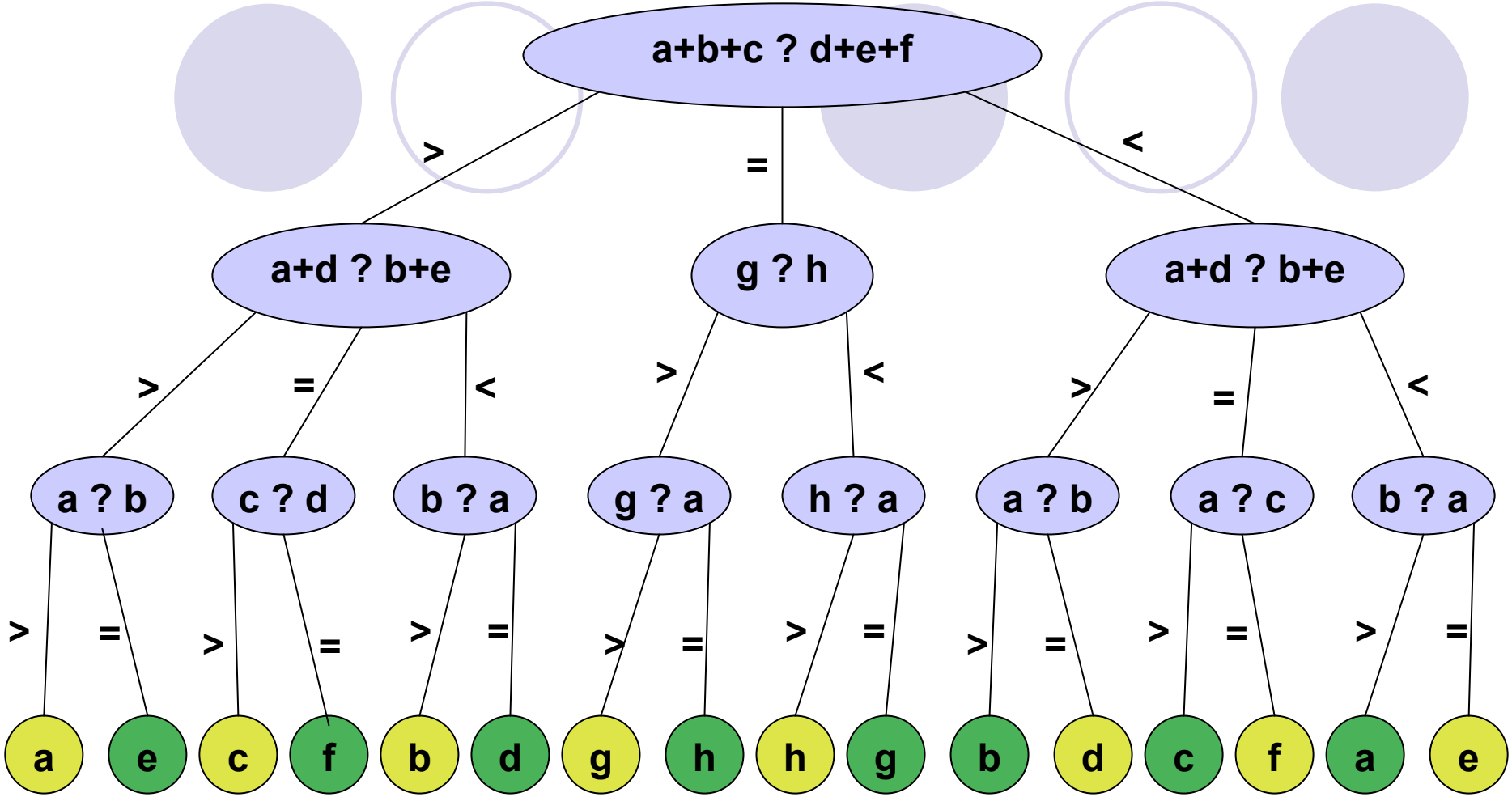


. whichType . operand

```
int Eval (TreeNode*  ptr)
{
   switch ( ptr->info.whichType )
     {
         case OPERAND :  return  ptr->info.operand ;
         case OPERATOR :
            switch ( tree->info.operation )
            {
               case  '+'  :  return  ( Eval ( ptr->left ) + Eval ( ptr->right ) ) ;

               case  '-'  :  return  ( Eval ( ptr->left )  -  Eval ( ptr->right ) ) ;

               case  '*'  :  return  ( Eval ( ptr->left )  *  Eval ( ptr->right ) ) ;

               case  '/'  :  return  ( Eval ( ptr->left )  /  Eval ( ptr->right ) ) ;
            }
     }
}
```

Cây quyết định

- Dùng để biểu diễn lời giải của bài toán cần quyết định lựa chọn
- Bài toán 8 đồng tiền vàng:
  - Có 8 đồng tiền vàng a, b, c, d, e, f, g, h
  - Có một đồng có trọng lượng không chuẩn
  - Sử dụng một cân Roberval (2 đĩa)
  - Output:
    - Đồng tiền k chuẩn là nặng hơn hay nhẹ hơn
    - Số phép cân là ít nhất


a+b+c ? d+e+f
>
=
<
a+d ? b+e
g ? h
a+d ? b+e
>
=
<
>
<
>
=
<
a ? b
c ? d
b ? a
g ? a
h ? a
a ? b
a ? c
b ? a
>
=
>
=
>
=
>
=
>
=
>
=
>
=
a
e
c
f
b
d
g
h
h
g
b
d
c
f
a
e

```
void EightCoins(a, b, c, d, e, f, g, h) {
   if (a+b+c == d+e+f) {
      if (g > h)                    Compare(g, h, a);
      else                          Compare(h, g, a);
   }
   else if (a+b+c > d+e+f){
      if (a+d == b+e)               Compare(c, f, a);
      else if (a+d > b+e)           Compare(a, e, b);
      else                          Compare(b, d, a);
   }
   else{
      if (a+d == b+e)    Compare(f,c,a);
      else if (a+d > b+e)           Compare(d, b, a);
      else                          Compare(e, a, b);
   }
}

// so sánh x với đồng tiền chuẩn z
void Compare(x,y,z){
   if(x>y) printf("x nặng");
   else printf("y nhẹ");
}
```

Cấu trúc dữ liệu và giải thuật

Đỗ Tuấn Anh

anhdt@it-hut.edu.vn

Nội dung

- Chương 1 – Thiết kế và phân tích (5 tiết)
- Chương 2 – Giải thuật đệ quy (10 tiết)
- Chương 3 – Mảng và danh sách (5 tiết)
- Chương 4 – Ngăn xếp và hàng đợi (10 tiết)
- Chương 5 – Cấu trúc cây (10 tiết)
- Chương 8 – Tìm kiếm (5 tiết)
- Chương 7 – Sắp xếp (10 tiết)
- Chương 6 – Đồ thị và một vài cấu trúc phi tuyến khác (5 tiết)
- Chương 9 – Sắp xếp và tìm kiếm ngoài (after)

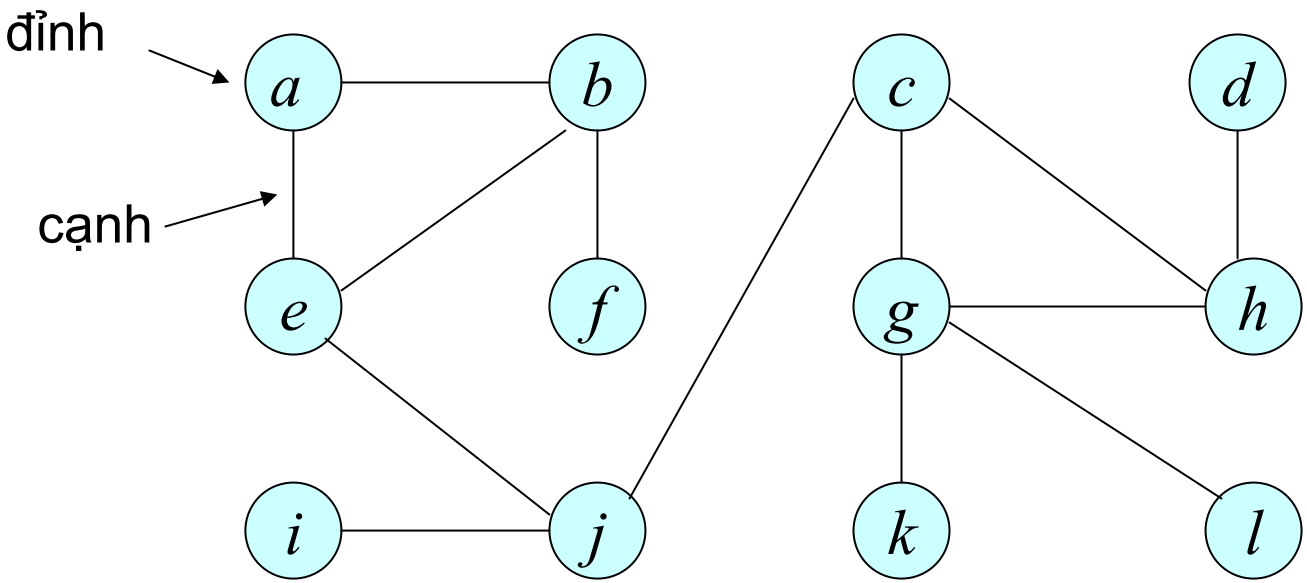
Chương 6 – Đồ thị và một vài cấu trúc phi tuyến khác

1. Định nghĩa và khái niệm
2. Biểu diễn đồ thị
   - Ma trận lân cận
   - Danh sách lân cận
3. Phép duyệt đồ thị
   - Theo chiều sâu
   - Theo chiều rộng
4. Ứng dụng
   - Bài toán bao đóng truyền ứng
   - Bài toán sắp xếp topo
5. Giới thiệu về danh sách tổng quát, đa danh sách (not yet)

1. Định nghĩa và khái niệm

Đồ thị

Một *đồ thị* *G* bao gồm một tập *V*(*G*) *các đỉnh* (*nút*) và một tập *E*(*G*) *các cạnh* (*cung*) là các cặp đỉnh.

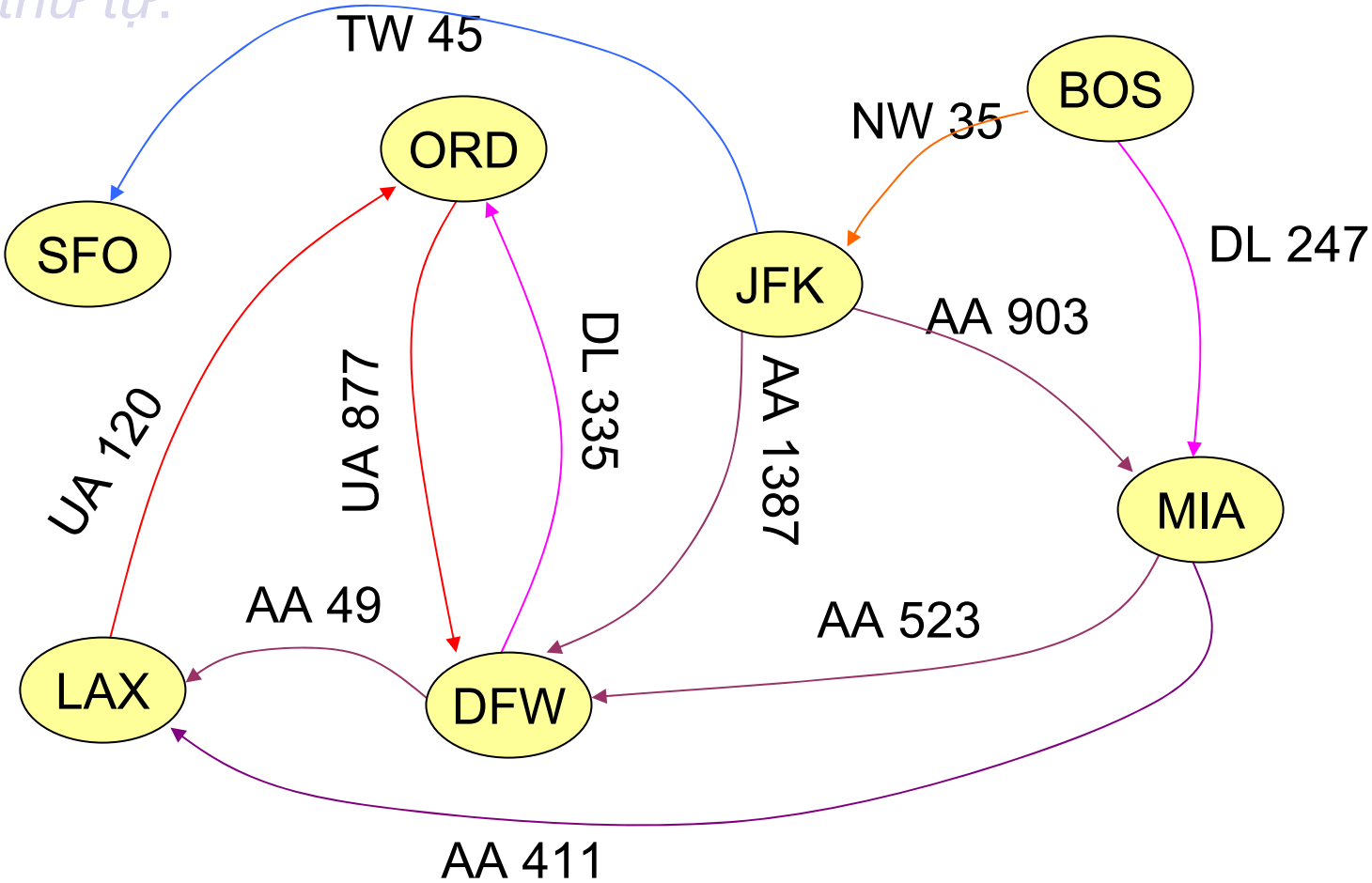


V = { *a*, *b*, *c*, *d*, *e*, *f*, *g*, *h*, *i*, *j*, *k*, *l* } 12 đỉnh

E = { (*a*, *b*), (*a*, *e*), (*b*, *e*), (*b*, *f*), (*c*, *j*), (*c*, *g*), (*c*, *h*), (*d*, *h*), (*e*, *j*),
(*g*, *k*), (*g*, *l*), (*g*, *h*), (*i*, *j*) } 13 cạnh

Đồ thị định hướng

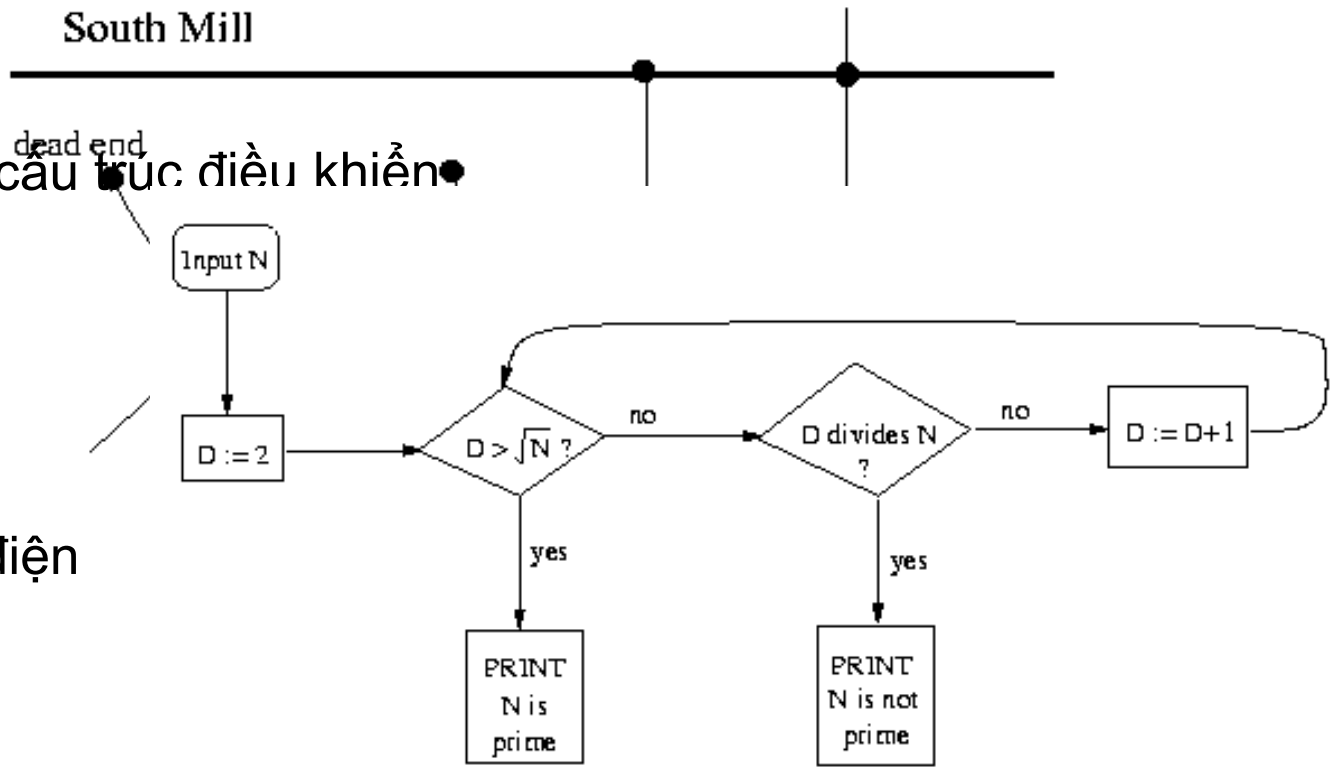
Trong *đồ thị định hướng* (*digraph*), các cạnh là những cặp *có thứ tự*.



Ứng dụng của đồ thị

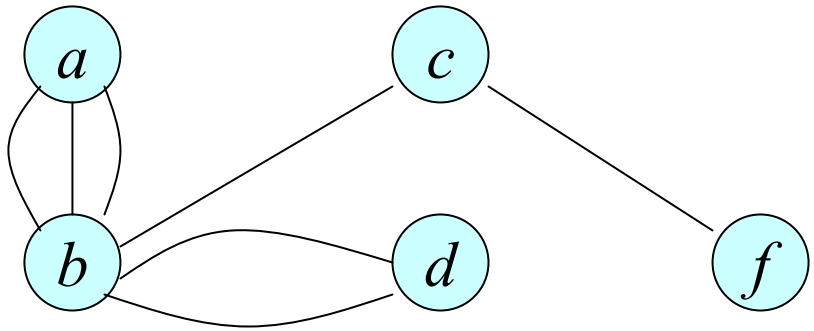
Đồ thị mô tả *các mối quan hệ*

- Mạng Internet
- Mạng lưới đường giao thông
- Nguyé
- Sơ đồ cấu trúc điều khiển
- Mạng
- Bề mặ
- Mạch điện
- …

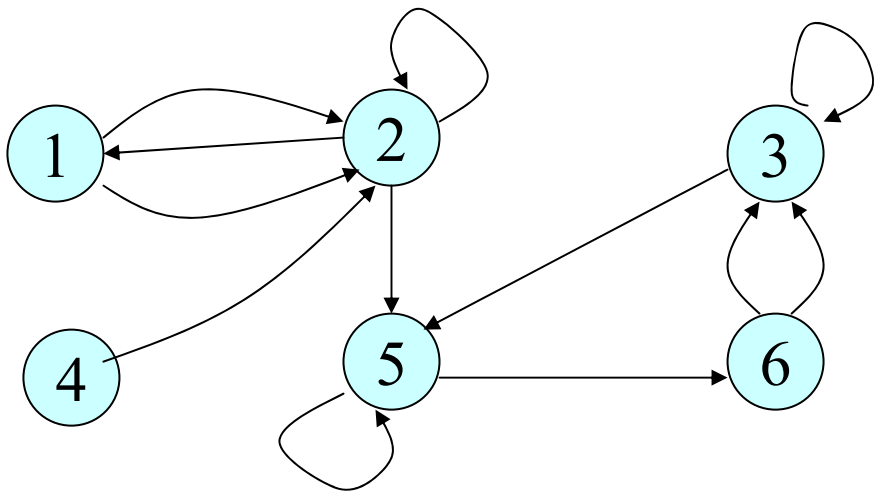


Các loại đồ thị khác

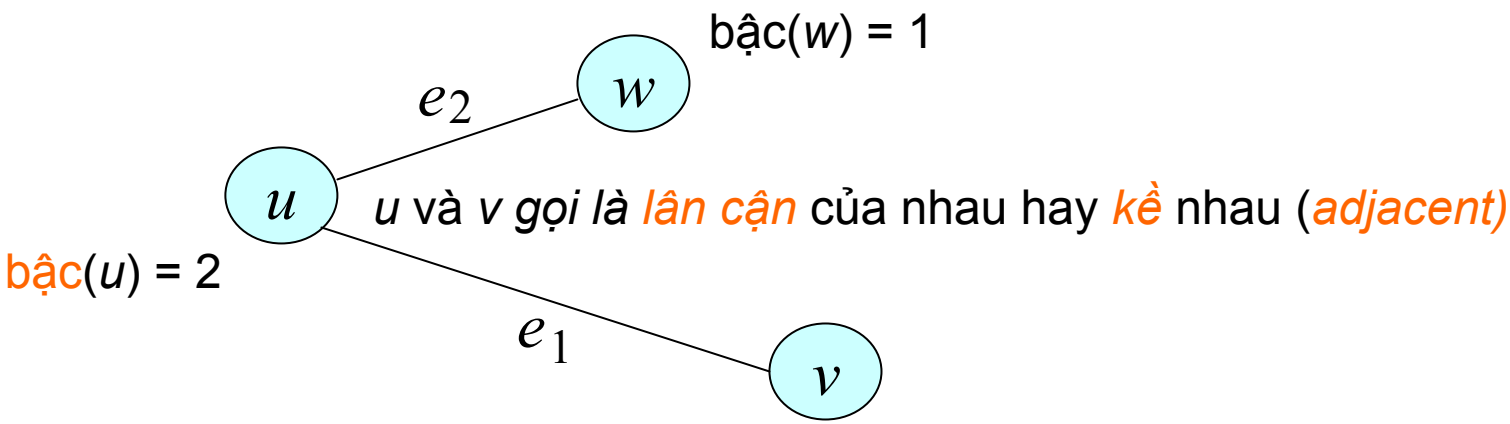
Đa đồ thị cho phép có thể có nhiều cạnh giữa 2 đỉnh.

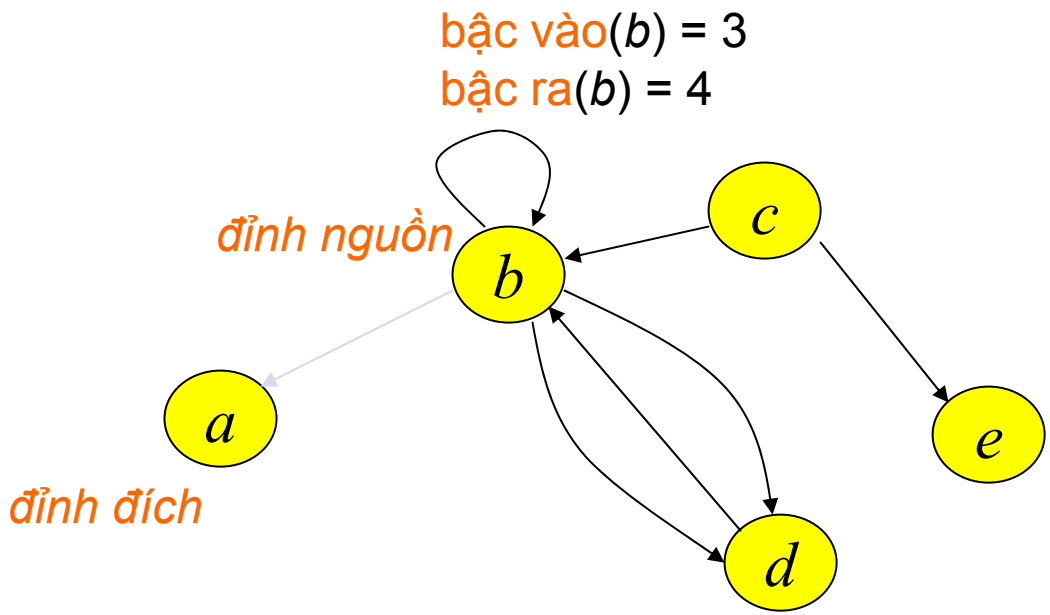


Giả đồ thị là một đa đồ thị cho phép *vòng lặp* (là các cạnh từ một đỉnh đến chính nó).



Cạnh và Đỉnh


bậc(*w*) = 1
e_2
w
u
u và *v* gọi là *lân cận* của nhau hay *kề* nhau (*adjacent)*
bậc(*u*) = 2
e_1
v


bậc vào(*b*) = 3
bậc ra(*b*) = 4
đỉnh nguồn
b
c
a
e
đỉnh đích
d

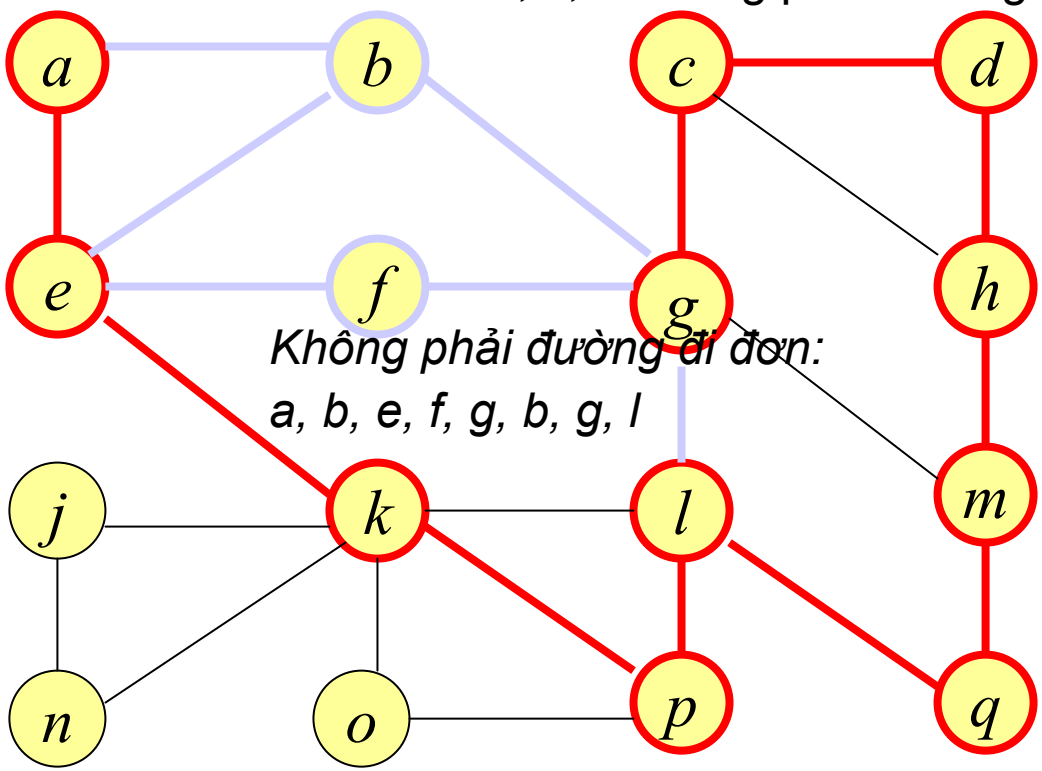
Đường đi

Một *đường đi* có độ dài *k* là một chuỗi các đỉnh v_0 , v_1 , …, v_k mà (v_i , v_{i+1}) với *i* = 0, 1, …, *k* – 1 là cạnh của *G*.

b, c, d không phải đường đi

***Đường đi đơn**:*
a, e, k, p, l, q
m, h, d, c, g

(không có đỉnh lặp lại)

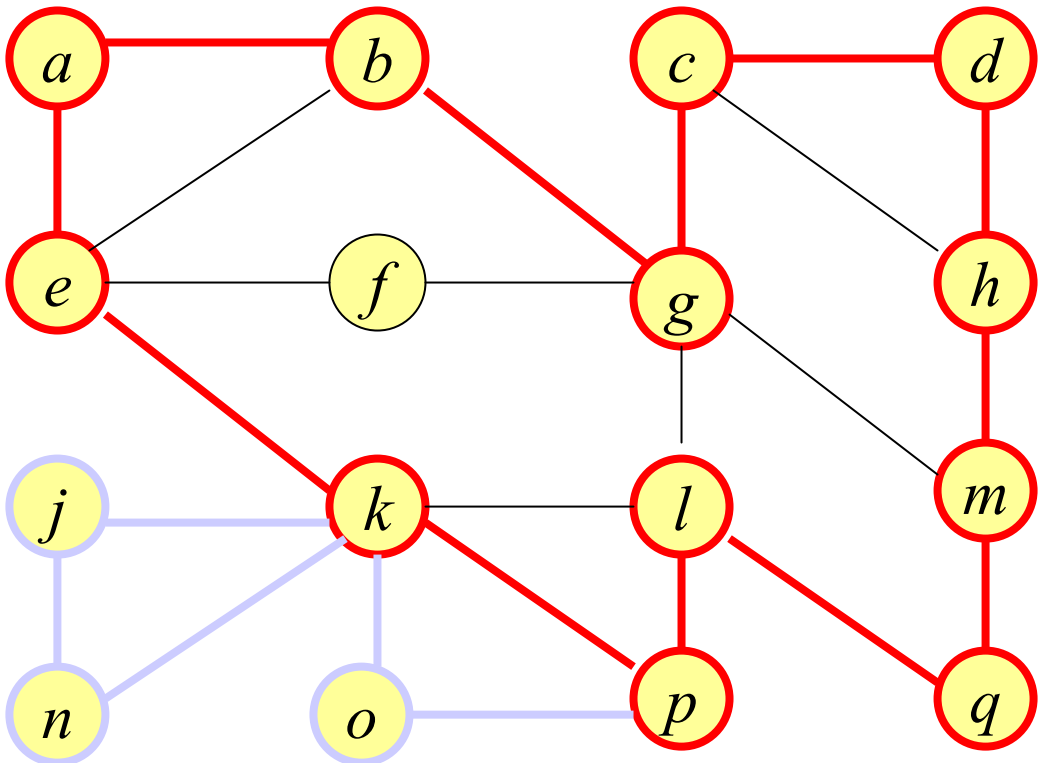


Không phải đường đi đơn:
a, b, e, f, g, b, g, l

Chu trình

Một *chu trình* là một đường đi có đỉnh đầu và đỉnh cuối trùng nhau.

Một *chu trình đơn* không có đỉnh trùng nhau trừ đỉnh đầu và đỉnh cuối.

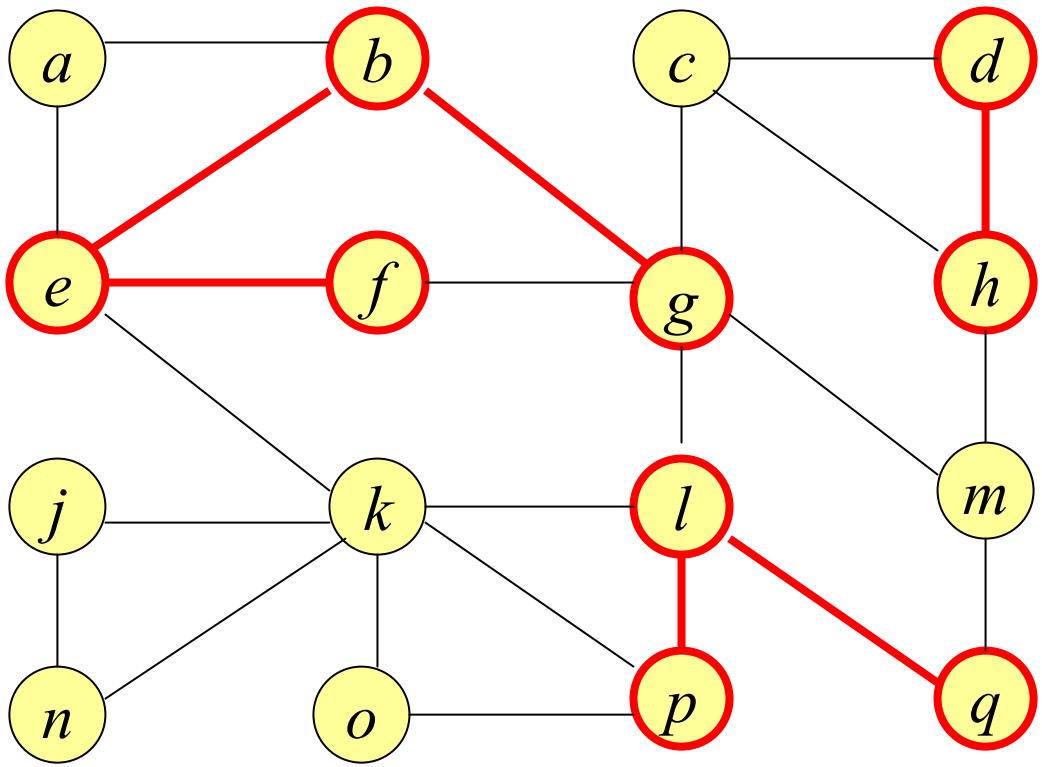


k, j, n, k, p, o,k không phải chu trình đơn.

Đồ thị con

Một *đồ thị con* *H* của *G*

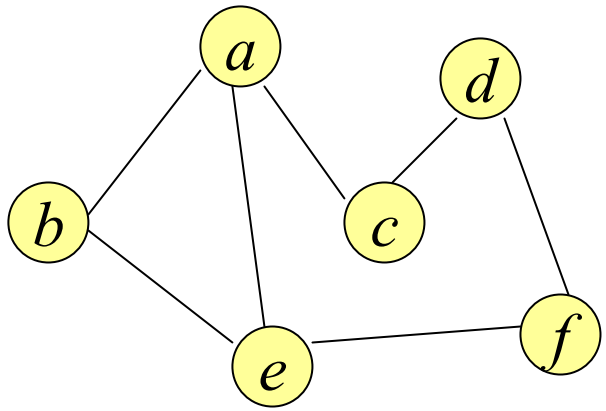
- là một đồ thị;
- các cạnh và các đỉnh của nó là tập con của *G*.



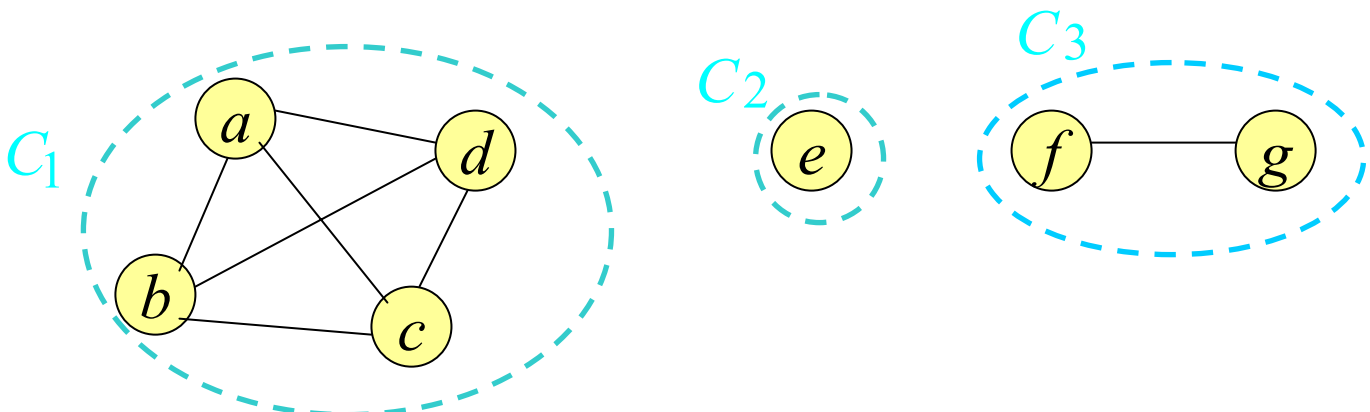
$V(H) = \{b, d, e, f, g, h, l, p, q\}$ $E(H) = \{(b, e), (b, g), (e, f), (d, h), (l, p), (l, q)\}$

Liên thông

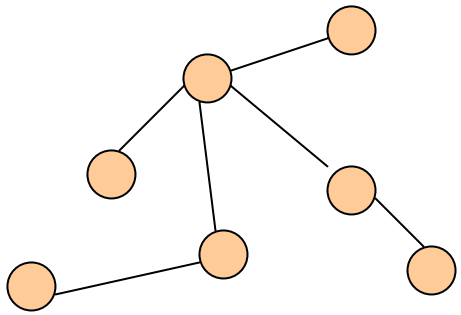
G được gọi là *liên thông* nếu giữa mọi cặp đỉnh của G đều có 1 đường đi..



Nếu *G* là không liên thông, các đồ thị con liên thông lớn nhất được gọi là *các thành phần liên thông* của *G*.



Cây có phải là liên thông?

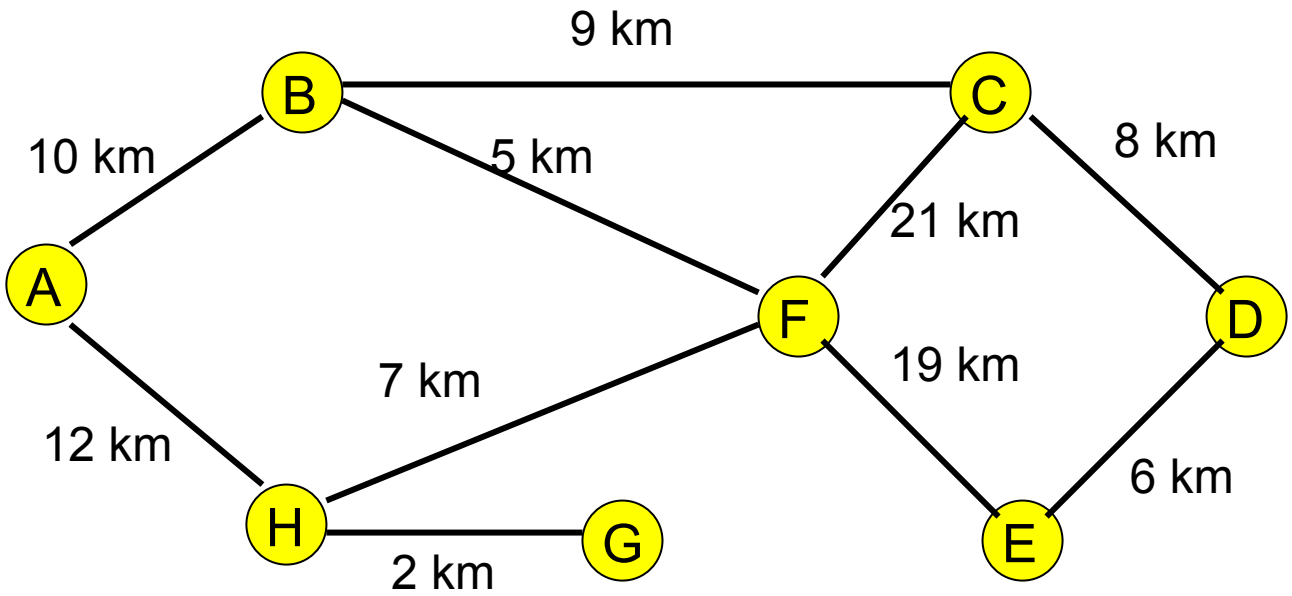


- Có, và $|E| = |V| - 1$.
  #số cạnh #số đỉnh

- Nếu G là liên thông, thì $|E| \geq |V| - 1$.

Đồ thị có trọng số

VD. Mạng lưới giao thông



2. Biểu diễn đồ thị

- 2 cách biểu diễn đồ thị phổ biến.
  1. Ma trận kề (Adjacency Matrix)

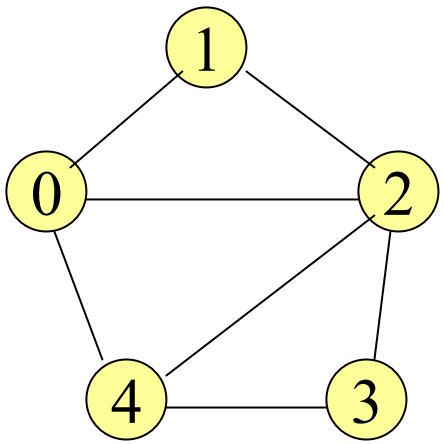
     Sử dụng một ma trận 2 chiều

  2. Danh sách kề (Adjacency List)

     Sử dụng một mảng của danh sách móc nối

Ma trận kề

bool *A* [n][n];



$$A[i][j] = \begin{cases} 1 & \text{nếu } (i, j) \in E(G) \\ 0 & \text{ngược lại} \end{cases}$$

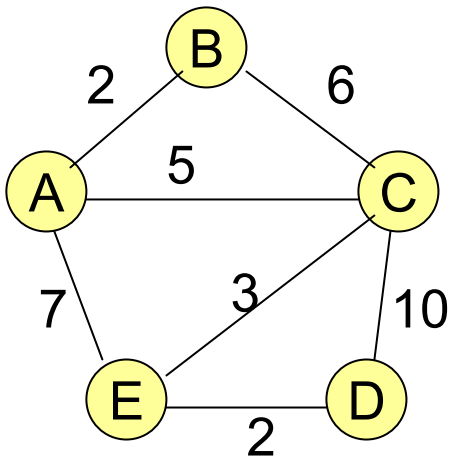
	0	1	2	3	4
0	0	1	1	0	1
1	1	0	1	0	0
2	1	1	0	1	1
3	0	0	1	0	1
4	1	0	1	1	0

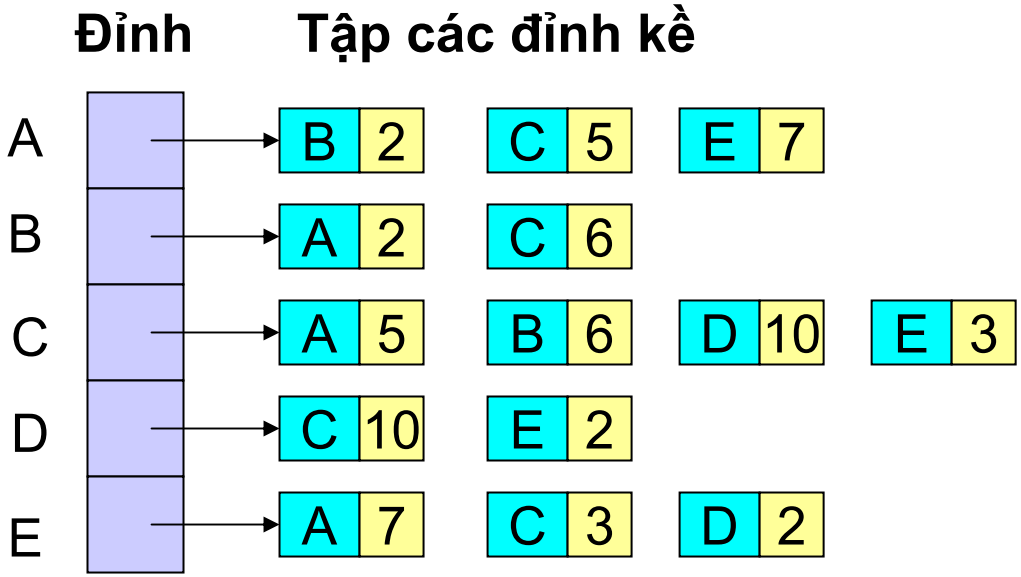
Lưu trữ: $O(|V|^2)$.

Đồ thị không định hướng

Thường được sử dụng với đồ thị nhỏ, không hiệu quả với đồ thị có ít cạnh

Danh sách kề





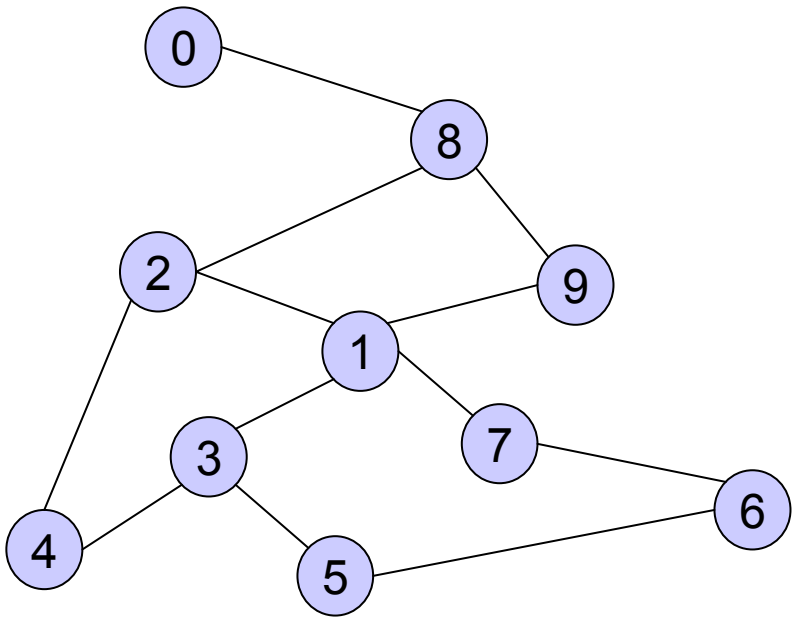
- **Danh sách kề là một mảng A[0..n-1] các danh sách, với n là số đỉnh của đồ thị.**
- **Chỉ số của mảng tương ứng với chỉ số của đỉnh**
- **Mỗi danh sách A[i] lưu trữ các chỉ số của các đỉnh kề với đỉnh i.**

Nếu G là đồ thị vô hướng, tổng độ dài của tất cả danh sách kề = $2|E|$.

Nếu G là đồ thị có hướng, tổng độ dài của tất cả danh sách kề = $|E|$.

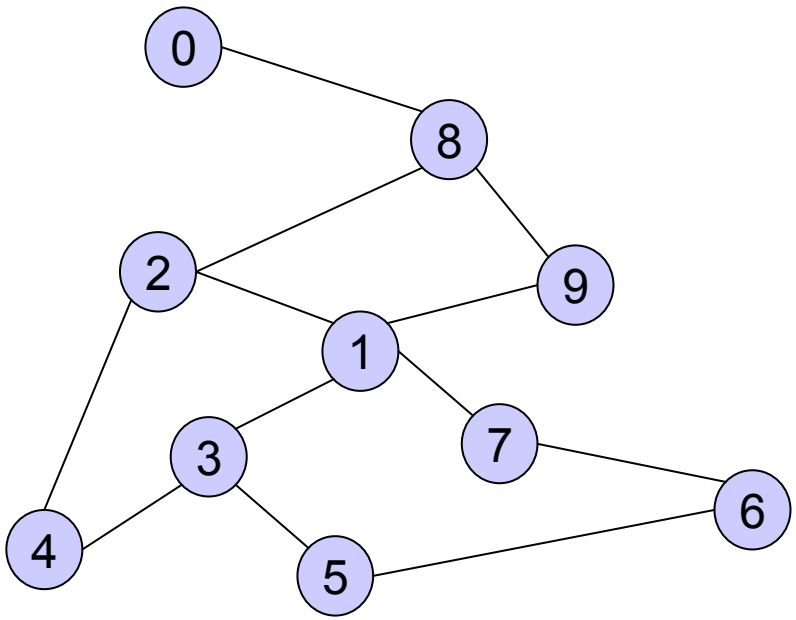
Chi phí bộ nhớ: $O(|V| + |E|)$. (Tốn ít bộ nhớ hơn).

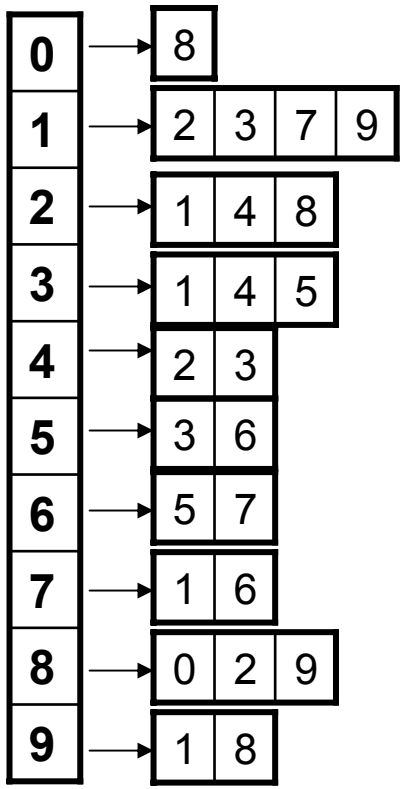
Ví dụ



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

Ví dụ





Phân tích độ phức tạp

Thao tác	Danh sách kề	Ma trận kề
Duyệt cạnh qua v	O(bậc(v))	$O(\|V\|)$
Duyệt cạnh ra của v	O(bậc ra(v))	$O(\|V\|)$
Duyệt cạnh vào của v	O(bậc vào(v))	$O(\|V\|)$
Kiểm tra u kề với v	O(min(bậc(u), bậc(v))	$O(1)$
Lưu trữ	$O(\|V\|+\|E\|)$	$O(\|V\|^2)$

Ma trận kề và danh sách kề

- **Danh sách kề**
  - Tiết kiệm bộ nhớ hơn ma trận kề nếu đồ thị có ít cạnh
  - Thời gian kiểm tra một cạnh có tồn tại lớn hơn

- **Ma trận kề**
  - Luôn luôn mất n^2 không gian bộ nhớ
    - Điều này có thể làm lãng phí bộ nhớ khi đồ thị thưa
  - Tìm một cạnh có tồn tại hay không trong thời gian hằng số

3. Phép duyệt đồ thị

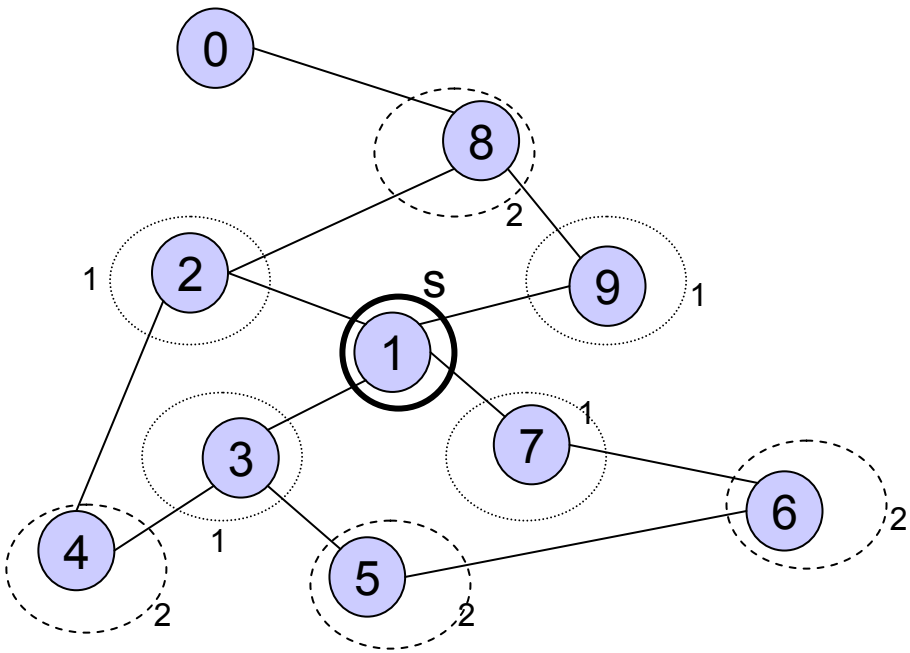
- Ứng dụng
  - Cho một đồ thị và một đỉnh **s** thuộc đồ thị
  - Tìm tất cả đường đi từ **s** tới các đỉnh khác
- 2 thuật toán duyệt đồ thị phổ biến nhất
  - Tìm kiếm theo chiều rộng (BFS)
    - Tìm đường đi ngắn nhất trong một đồ thị không có trọng số
  - Tìm kiếm theo chiều sau (DFS)
    - Bài toán sắp xếp topo
    - Tìm các thành phần liên thông mạnh
- Trước tiên ta sẽ xem xét BFS

Tìm kiếm theo chiều rộng

Tìm đường đi ngắn nhất từ đỉnh nguồn *s* tới tất cả các nút.

Ý tưởng: Tìm tất cả các nút tại khoảng cách 0, rồi tại khoảng cách 1, rồi đến khoảng cách 2, …

- Khoảng cách là số cạnh trên đường đi bắt đầu từ s



Ví dụ

Với s là đỉnh 1

Các nút tại khoảng cách 1?
2, 3, 7, 9

Các nút tại khoảng cách 2?
8, 6, 5, 4

Các nút tại khoảng cách 3?
0

BFS – Giải thuật

- Một hàng đợi Q để lưu trữ các đỉnh đang đợi được thăm.

  Tại mỗi bước, một đỉnh sẽ bị xóa khỏi Q và được đánh dấu là đã thăm.

- Một mảng flag lưu trạng thái các đỉnh đã được thăm.
- Mỗi đỉnh có trạng thái "thăm" như sau:

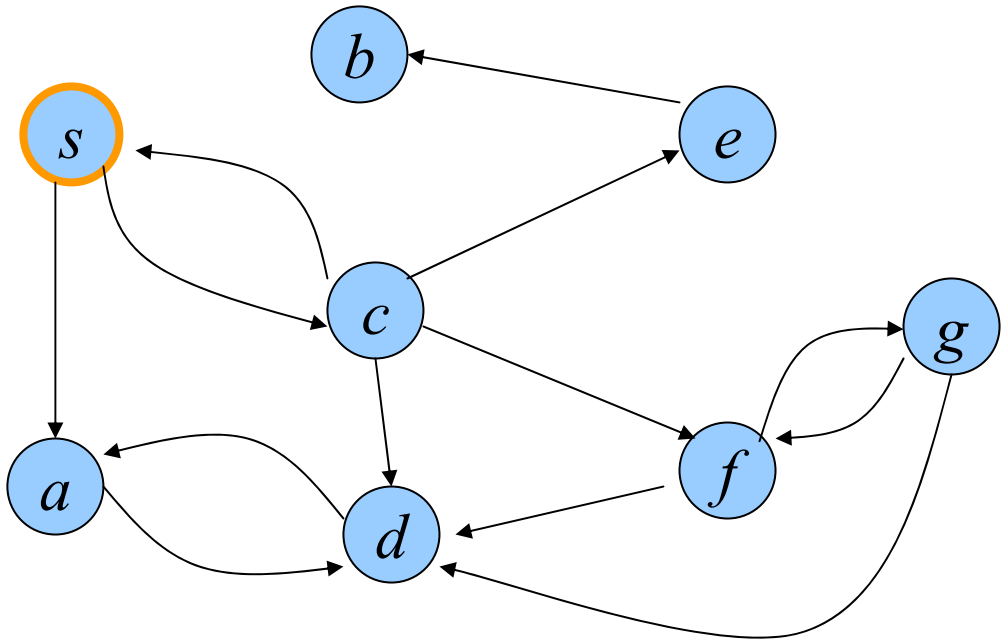
  FALSE: đỉnh là chưa được thăm.
  TRUE: đỉnh được đưa vào hàng đợi

BSF algorithm

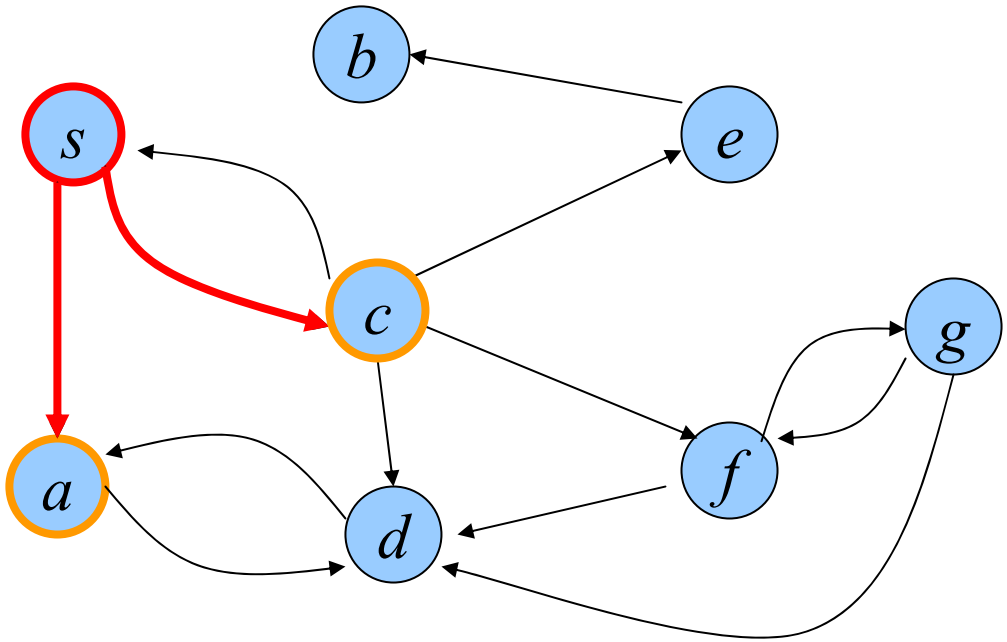
Algorithm $BFS(s)$
Input: s is the source vertex
Output: Mark all vertices that can be visited from s.

```
1.   for each vertex v
2.       do flag[v] := false;        // chưa được thăm
3.   Q = empty queue;
4.   flag[s] := true;
5.   enqueue(Q, s);
6.   while Q is not empty
7.      do v := dequeue(Q);
8.         for each w adjacent to v
9.            do if flag[w] = false
10.                  then flag[w] := true;
11.                       enqueue(Q, w)
```

Ví dụ BFS

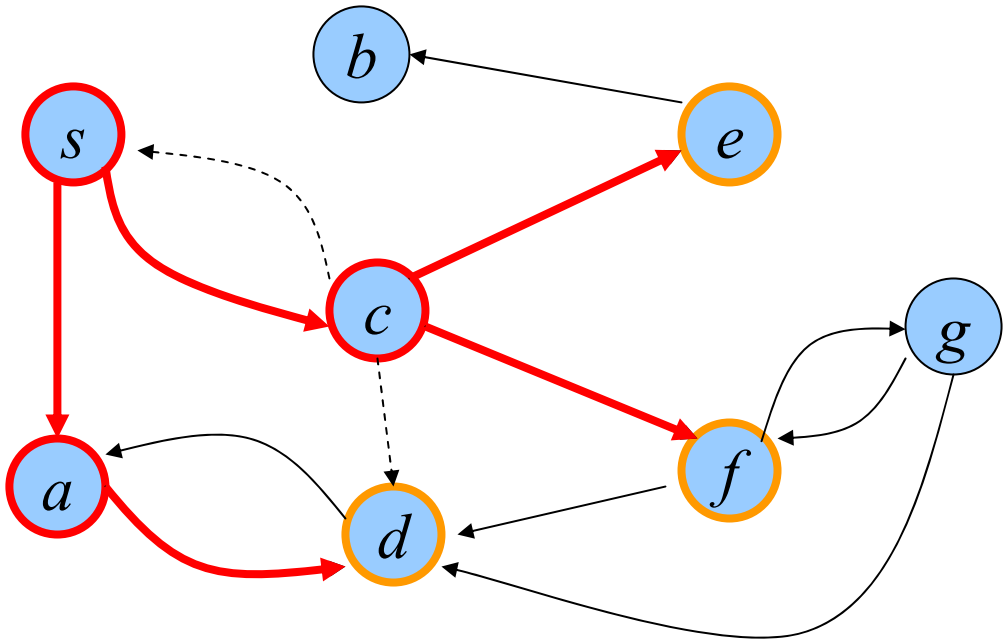


Thứ tự thăm:
Q: *s*



Thứ tự thăm: *s*

Q: *a, c*

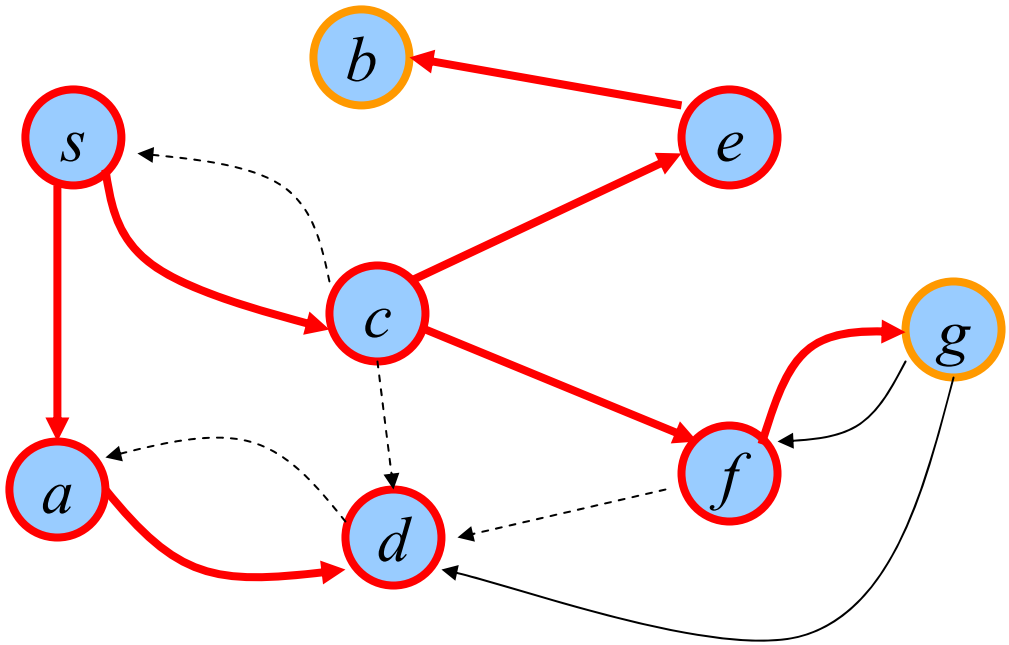


TT thăm: *s*, *a*
Q: *c, d*

⇩

TT thăm: *s*, *a, c*
Q: *d, e, f*

Các cạnh có nét đứt chỉ ra rằng đỉnh được xét nhưng đỉnh đã được thăm.

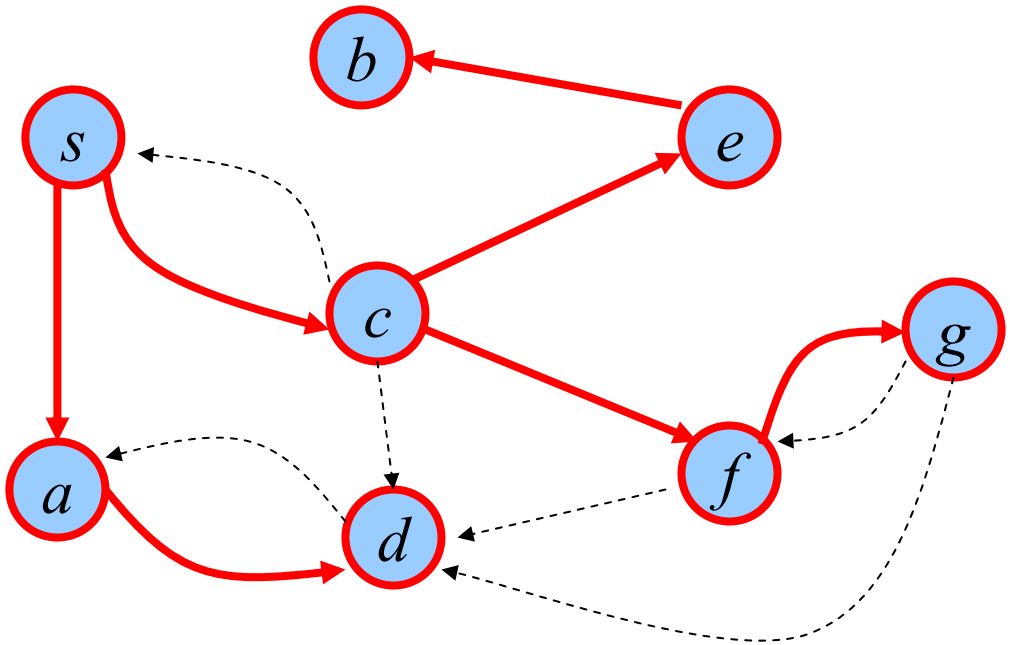


TT thăm: *s, a, c, d*
Q: *e, f*

TT thăm: *s, a, c, d, e*
Q: *f, b*

TT thăm: *s, a, c, d, e,*
Q: *b, g*

Kết thúc

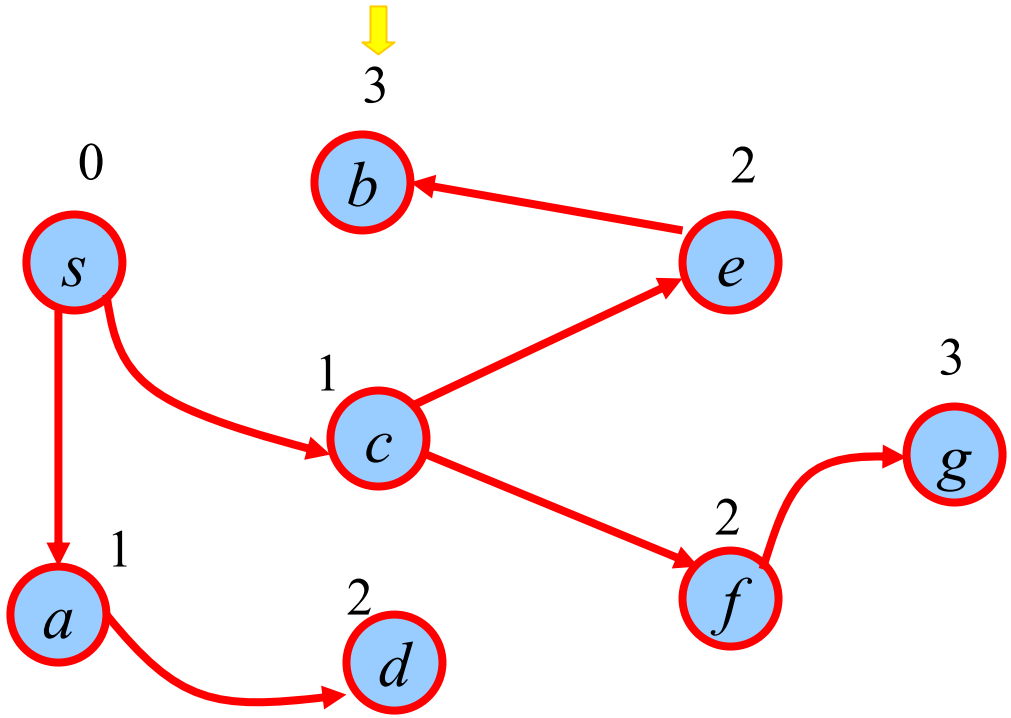


TT thăm: *s, a, c, d, e, f, b*
Q: *g*

TT thăm: *s, a, c, d, e, f, b, g*
Q:

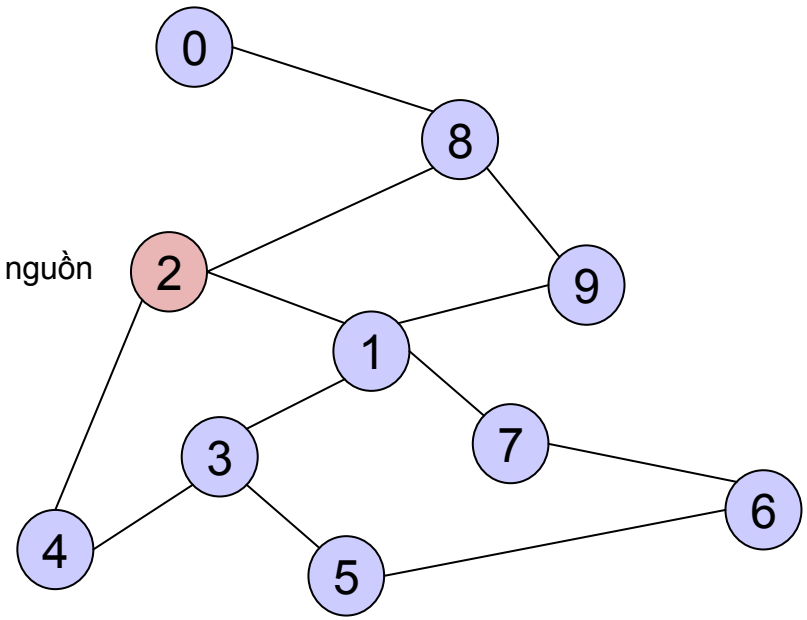
Cây duyệt theo chiều rộng

d(*b*): đường đi ngắn nhất từ *s* đến *b*



BFS chỉ thăm các tập con có thể đến được từ đỉnh ban đầu

Ví dụ



Danh sách kề

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

Flag (T/F)

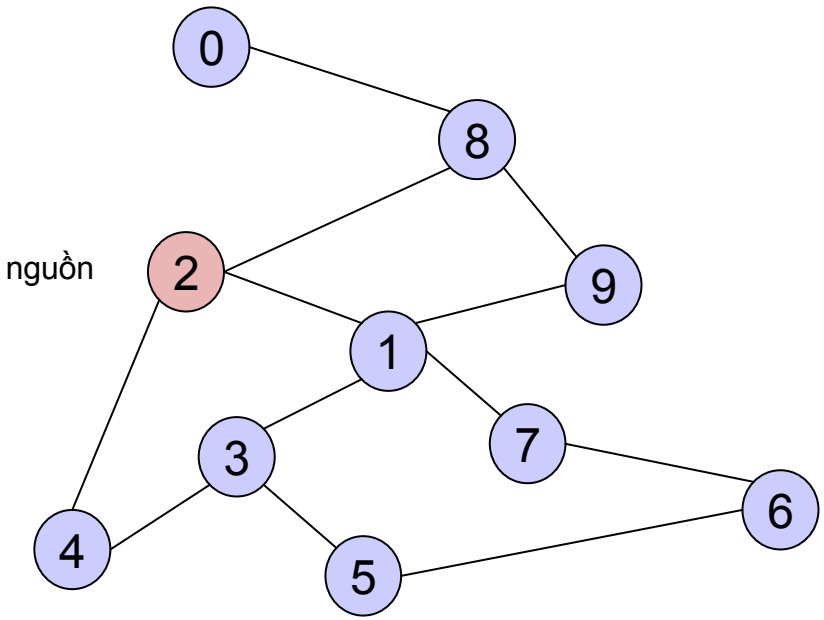
0	F
1	F
2	F
3	F
4	F
5	F
6	F
7	F
8	F
9	F

Khởi tạo ban đầu
(tất cả = F)

Q = { }

Khởi tạo **Q** rỗng

Ví dụ



Danh sách kề

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

Flag (T/F)

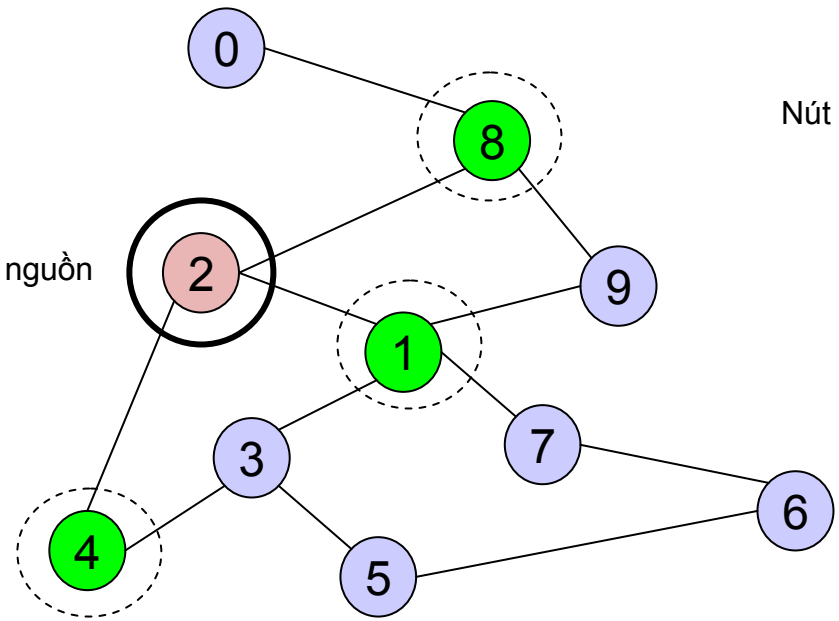
0	F
1	F
2	T
3	F
4	F
5	F
6	F
7	F
8	F
9	F

2 đã được thăm
Flag(2) = T.

Q = { 2 }

Đặt đỉnh nguồn 2 vào queue.

Ví dụ



Danh sách kề

Nút kề →

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

Flag (T/F)

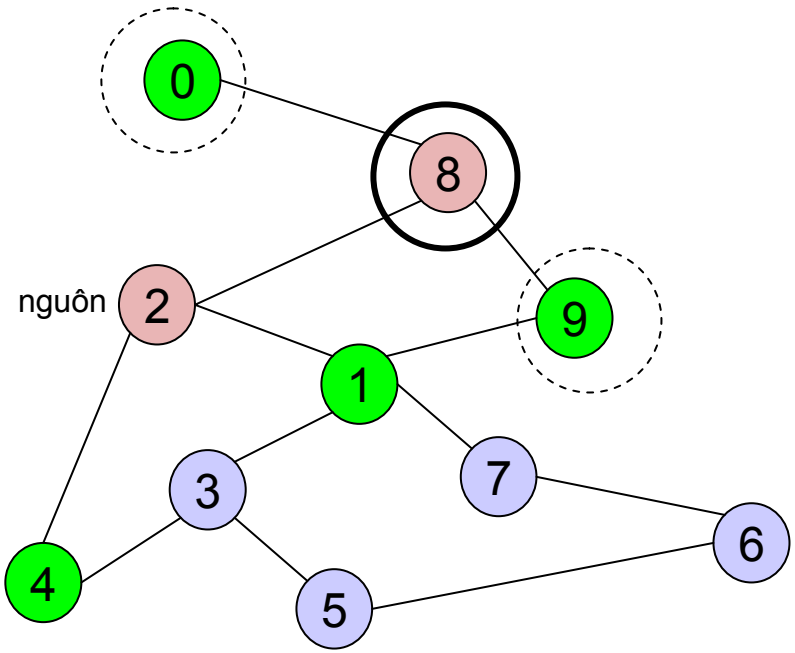
0	F
1	T
2	T
3	F
4	T
5	F
6	F
7	F
8	T
9	F

Đánh dấu các nút kề là đã thăm.

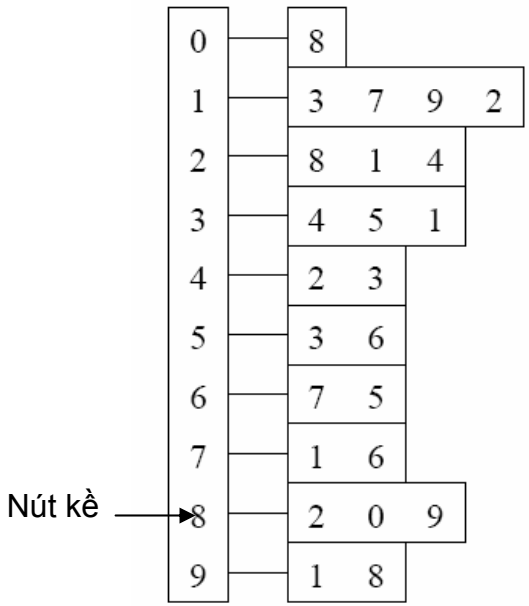
Q = {2} → { 8, 1, 4 }

Lấy 2 ra khỏi queue.
Đặt tất cả các nút kề chưa được thăm của 2 vào queue

Ví dụ



Danh sách kề



Flag (T/F)

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

Đánh dấu các nút mới được thăm.

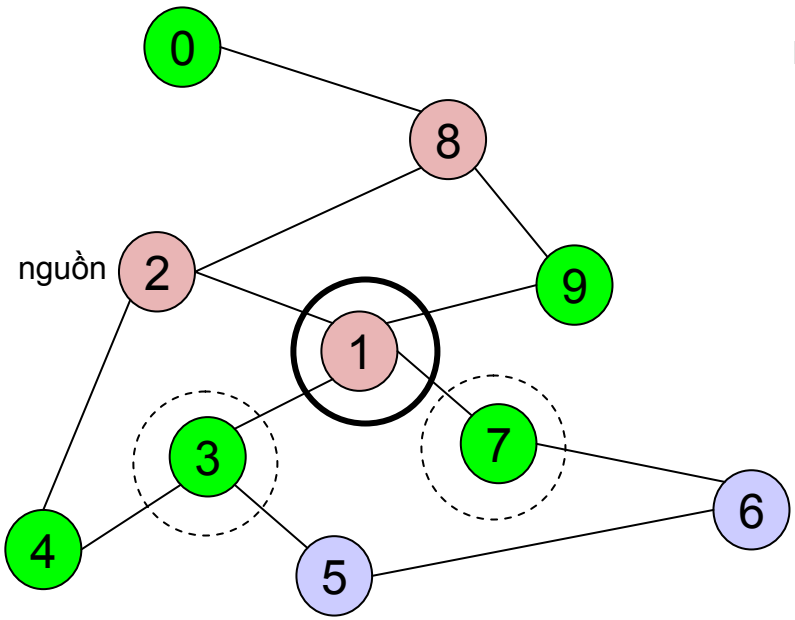
Q = { 8, 1, 4 } → { 1, 4, 0, 9 }

Lấy 8 ra khỏi queue.

-- Đặt tất cả các nút kề chưa được thăm của 8 vào queue.

-- Chú ý là 2 không được đặt vào queue nữa vì nó đã được thăm

Ví dụ



Danh sách kề

Flag (T/F)

Nút kề →

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

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

Đánh dấu các nút mới được thăm.

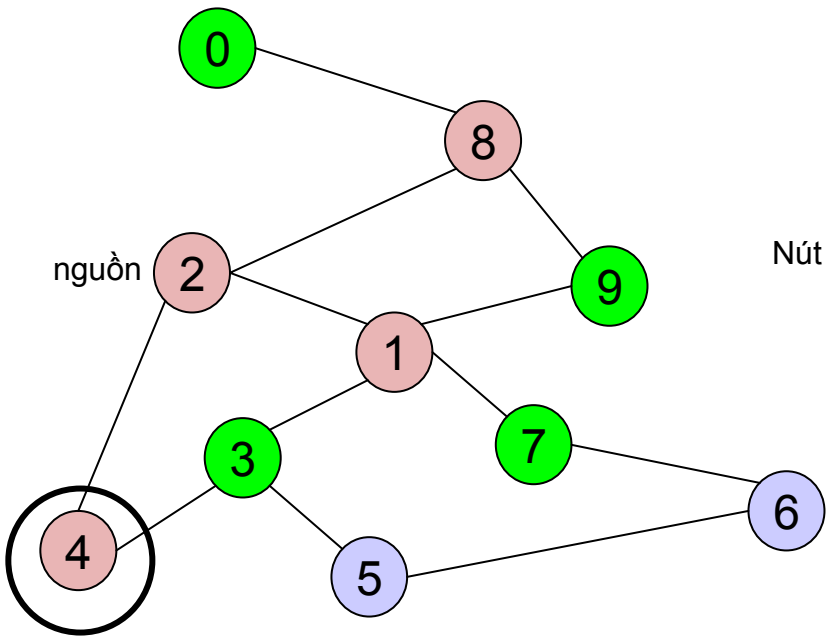
Q = { 1, 4, 0, 9 } → { 4, 0, 9, 3, 7 }

Rút 1 ra khỏi queue.

-- Đặt tất cả các nút kề chưa được thăm của 1 vào queue.

-- Chỉ có nút 3 và 7 là chưa được thăm.

Ví dụ



Danh sách kề

Nút kề →

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

Flag (T/F)

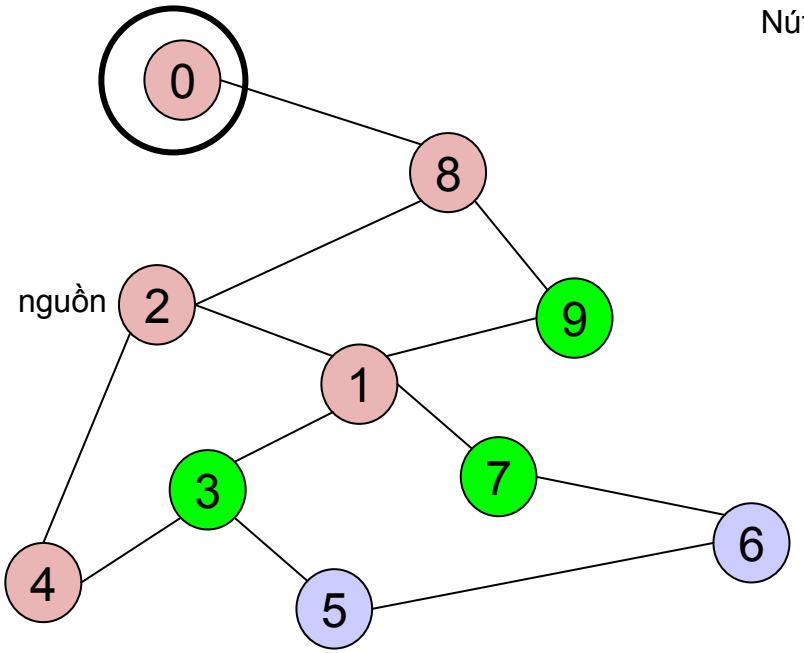
0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Q = { 4, 0, 9, 3, 7 } → { 0, 9, 3, 7 }

Rút 4 ra khỏi queue.

-- 4 không có nút kề nào là chưa được thăm!

Ví dụ



Danh sách kề

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

Flag (T/F)

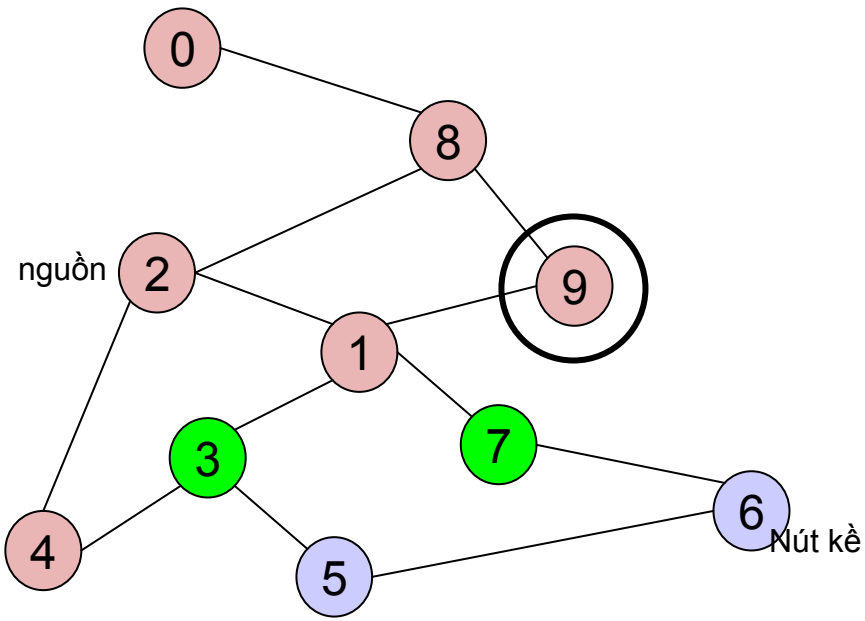
0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Q = { 0, 9, 3, 7 } → { 9, 3, 7 }

Rút 0 ra khỏi queue.

-- 0 không có nút kề nào là chưa được thăm!

Ví dụ



Danh sách kề

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

Flag (T/F)

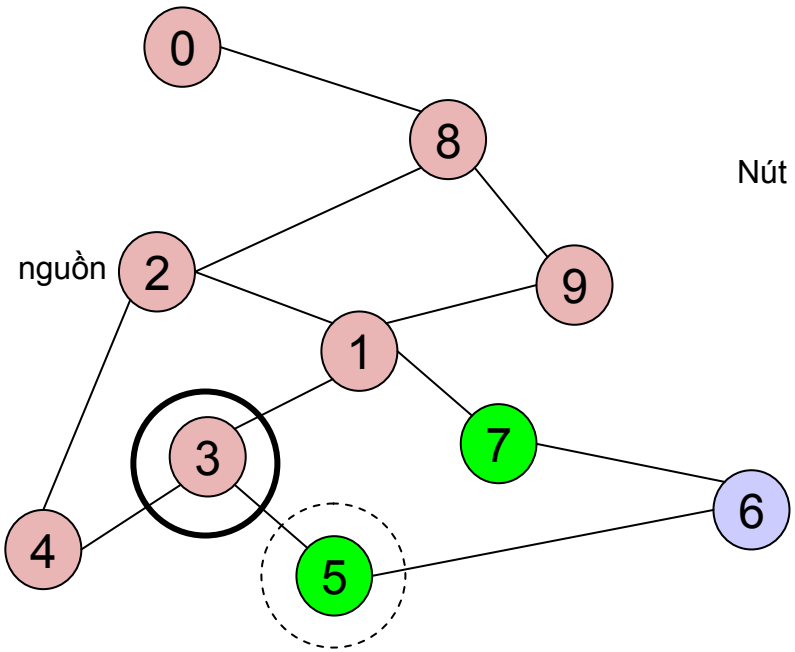
0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	T
8	T
9	T

Q = { 9, 3, 7 } → { 3, 7 }

Rút 9 ra khỏi queue.

-- 9 không có nút kề nào chưa được thăm!

Ví dụ



Danh sách kề

Nút kề →

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

Flag (T/F)

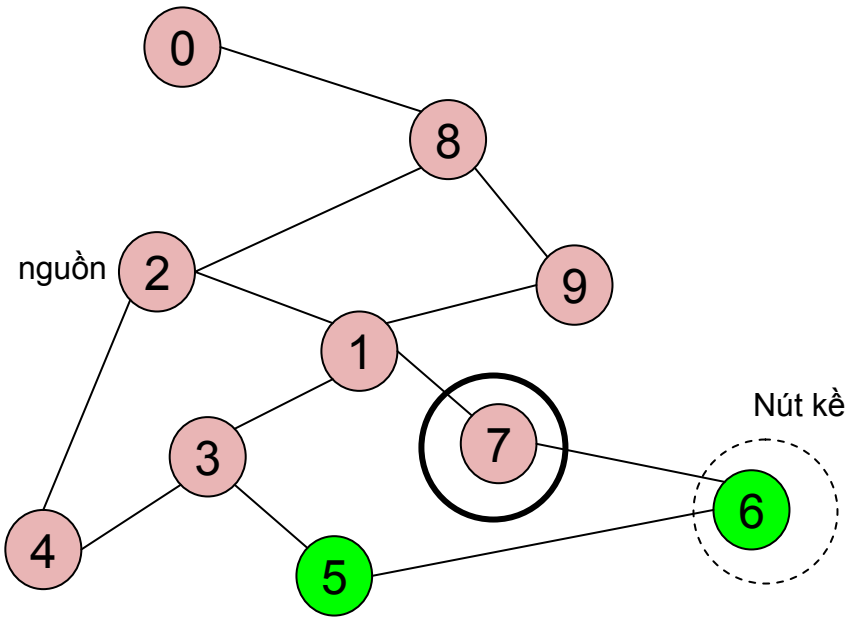
0	T
1	T
2	T
3	T
4	T
5	T
6	F
7	T
8	T
9	T

Đánh dấu đỉnh 5 đã được thăm.

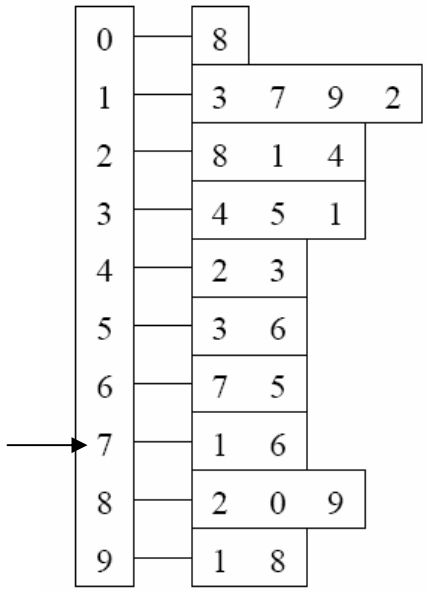
Q = { 3, 7 } → { 7, 5 }

Rút 3 ra khỏi queue.
-- Thêm nút 5 vào queue.

Ví dụ



Danh sách kề



Flag (T/F)

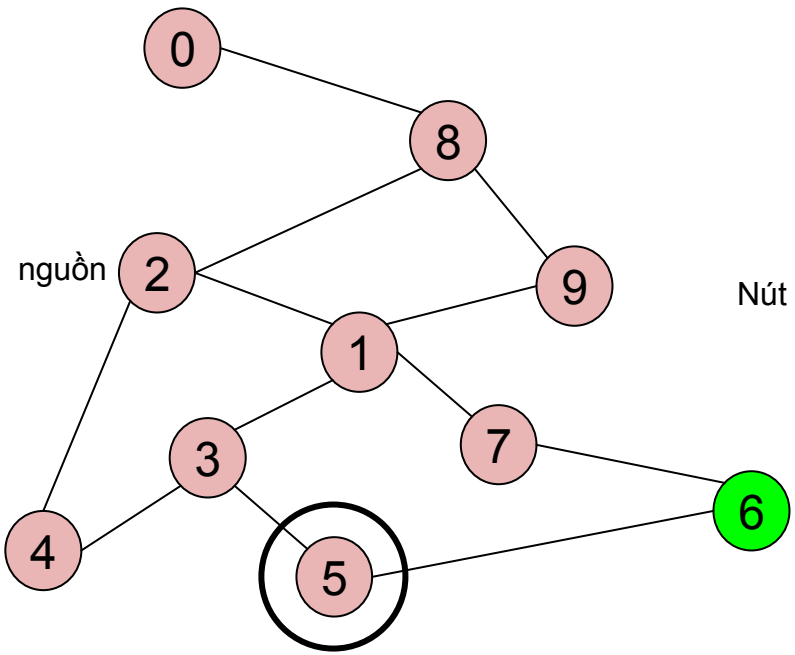
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Đánh dấu đỉnh 6 đã được thăm.

Q = { 7, 5 } → { 5, 6 }

Rút 7 ra khỏi queue.
-- Thêm nút 6 vào queue.

Ví dụ



Danh sách kề

Nút kề →

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

Flag (T/F)

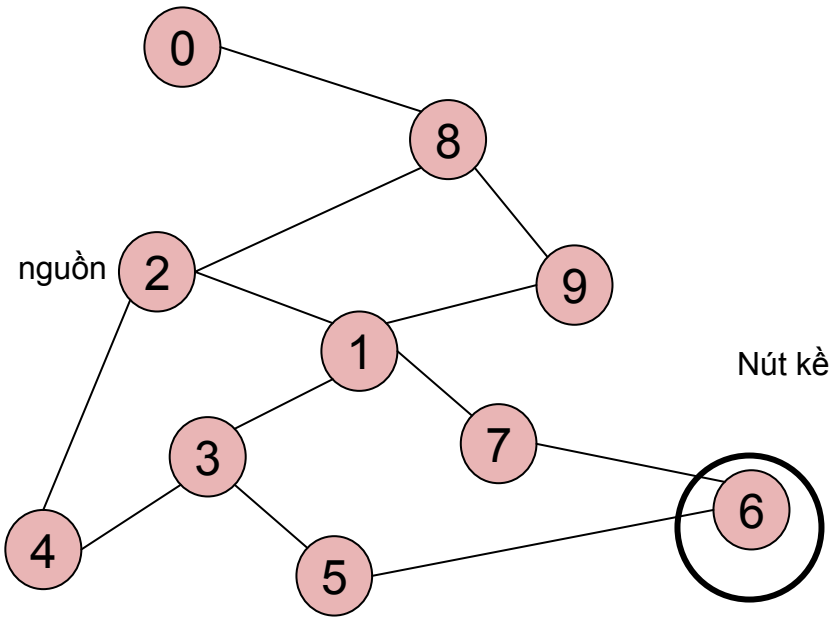
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Q = { 5, 6} → { 6 }

Rút 5 ra khỏi queue.

-- không có nút kề nào của 5 là chưa được thăm.

Ví dụ



Nút kề

Danh sách kề

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

Flag (T/F)

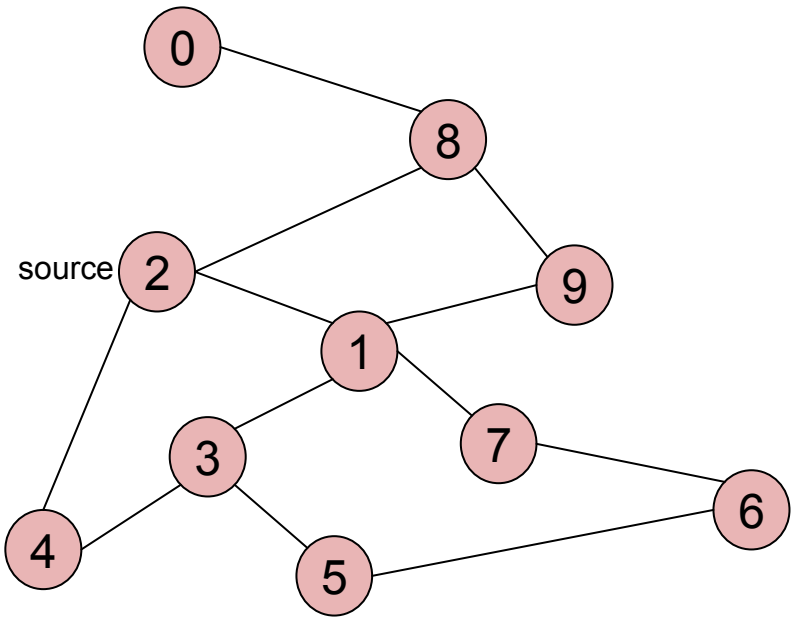
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Q = { 6 } → { }

Rút 6 ra khỏi queue.

-- không có nút kề nào của 6 là chưa được thăm.

Ví dụ



Danh sách kề

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

Flag (T/F)

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

Q = { } Dừng!!! Q rỗng!!!

Kết quả?

Nhìn vào Flag.

Tồn tại một đường đi từ 2 tới tất cả các đỉnh khác trong đồ thị

Độ phức tạp thời gian của BFS
(Sử dụng danh sách kề)

- Giả sử đồ thị được lưu dưới dạng danh sách kề
  - n = số đỉnh, m = số cạnh

Algorithm *BFS*(s)
Input: s is the source vertex
Output: Mark all vertices that can be visited from s.

```
1.  for each vertex v
2.      do flag[v] := false;
3.  Q = empty queue;
4.  flag[s] := true;
5.  enqueue(Q, s);
6.  while Q is not empty
7.      do v := dequeue(Q);
8.          for each w adjacent to v
9.              do if flag[w] = false
10.                 then flag[w] := true;
11.                     enqueue(Q, w)
```

O(n + m)

Mỗi đỉnh vào Q duy nhất một lần.

Mỗi lần lặp, thời gian tính tỉ lệ với bậc(v) + 1

Thời gian tính

- Nhắc lại: Một đồ thị có m cạnh, tổng số bậc = ?

$$\Sigma_{\text{vertex } v} \text{ bậc}(v) = 2m$$

- Tổng thời gian tính của vòng lặp while:

$$O(\Sigma_{\text{vertex } v} (\text{bậc}(v) + 1)) = O(n+m)$$

được tính trên tổng của tất cả các lần lặp trong while!

Độ phức tạp thời gian của BFS
(Sử dụng ma trận kề)

- Giả sử đồ thị được lưu dưới dạng ma trận kề
  - n = số đỉnh, m = số cạnh

```
Algorithm BFS(s)
Input: s is the source vertex
Output: Mark all vertices that can be visited from s.
1.   for each vertex v
2.       do flag[v] := false;
3.   Q = empty queue;
4.   flag[s] := true;
5.   enqueue(Q, s);
6.   while Q is not empty
7.      do v := dequeue(Q);
8.         for each w adjacent to v
9.            do if flag[w] = false
10.                 then flag[w] := true;
11.                      enqueue(Q, w)
```

O(n²)

Tìm các đỉnh kề của v phải duyệt toàn bộ hàng, mất thời gian O(n).

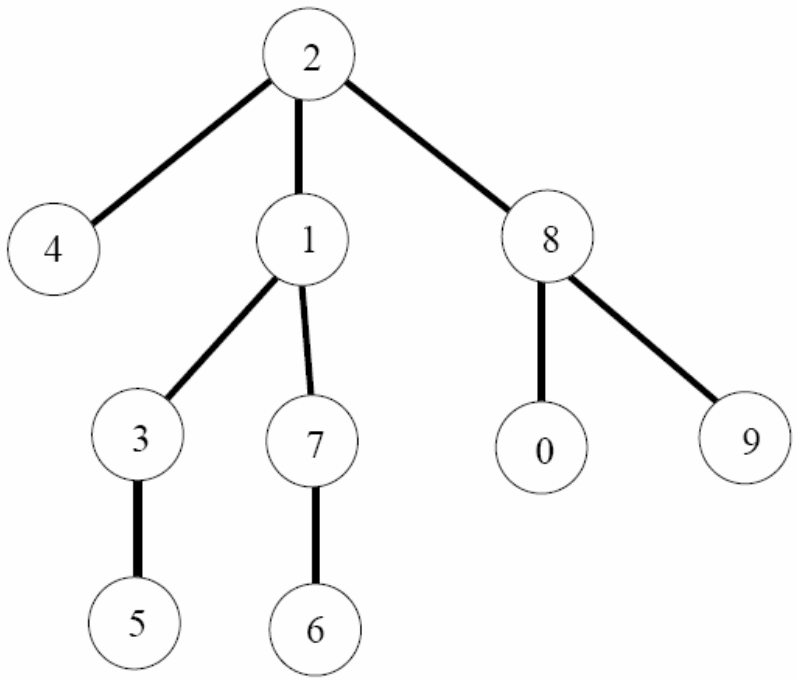
Tổng trên n lần lặp, tổng thời gian tính là O(n²).

Với ma trận kề, BFS là O(n²) độc lập với số cạnh m.

Cây khung theo chiều rộng

- Những đường đi được tạo ra bởi phép duyệt BFS thường được vẽ như một cây (gọi là cây khung theo chiều rộng), với đỉnh bắt đầu là gốc của cây.

Cây BFS với đỉnh s=2.

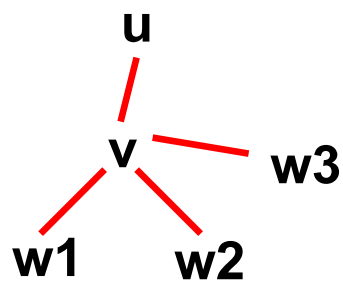


Tìm kiếm theo chiều sâu Depth-First Search (DFS)

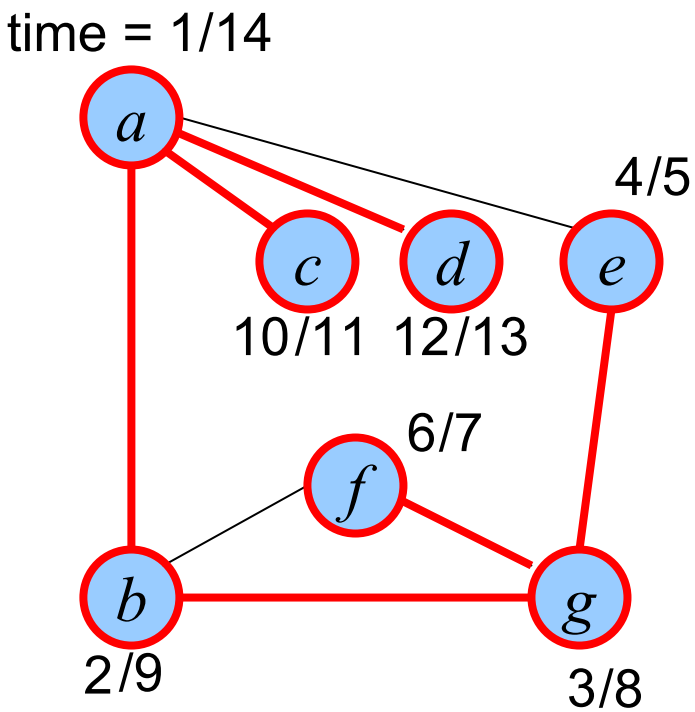
- DFS là một phép tìm kiếm trên đồ thị phổ biến khác
  - Về mặt ý tưởng, tương tự như phép duyệt theo thứ tự trước (thăm nút, rồi thăm các nút con một cách đệ quy)
- DFS có thể chỉ ra một số thuộc tính của đồ thị mà BFS không thể
  - Nó có thể cho biết đồ thị có chu trình hay không
  - Học sâu hơn trong Toán Rời Rạc

Giải thuật DFS

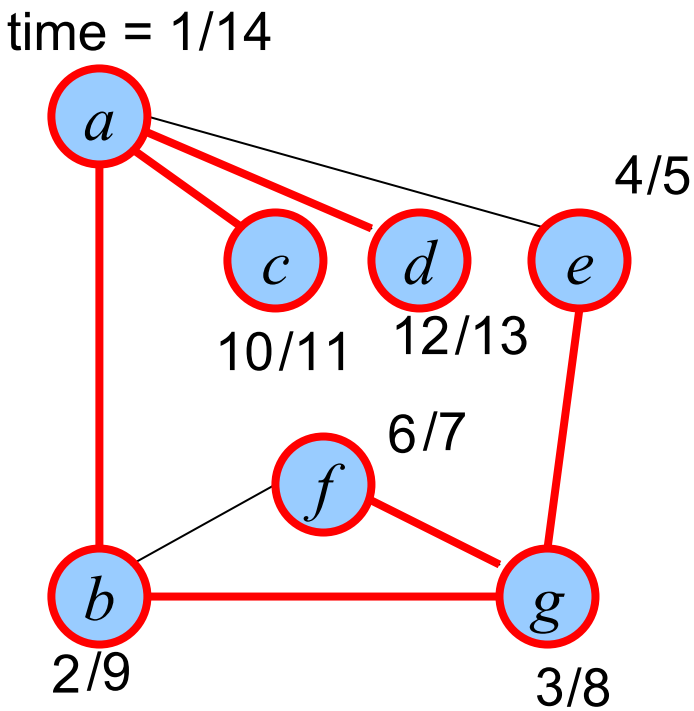
- DFS tiếp tục thăm các nút kề một cách đệ quy
  - Khi thăm v là kề với u, tiếp tục đệ quy để thăm tất cả các nút kề chưa được thăm của v. Sau đó quay lui lại u.



Ví dụ

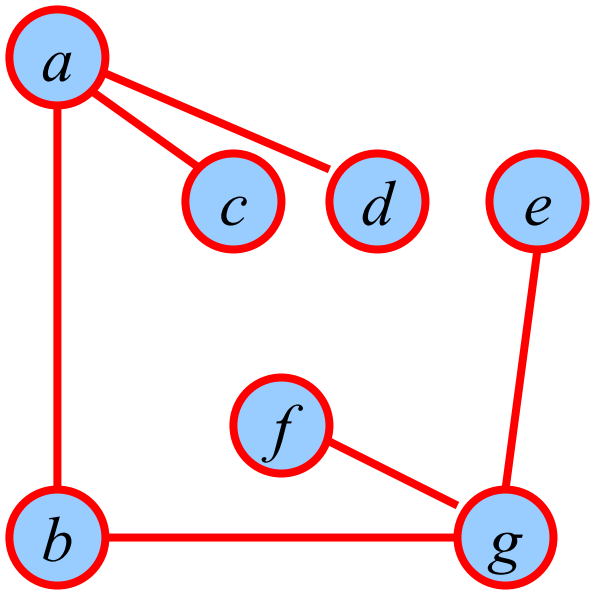


Thứ tự thăm



a d c b g f e

Cây DFS



Giải thuật DFS

Algorithm *DFS*(s)
1. **for** each vertex v
2. **do** $flag[v]$:= false;
3. *RDFS*(s);

Đánh dấu tất cả các đỉnh là chưa được thăm

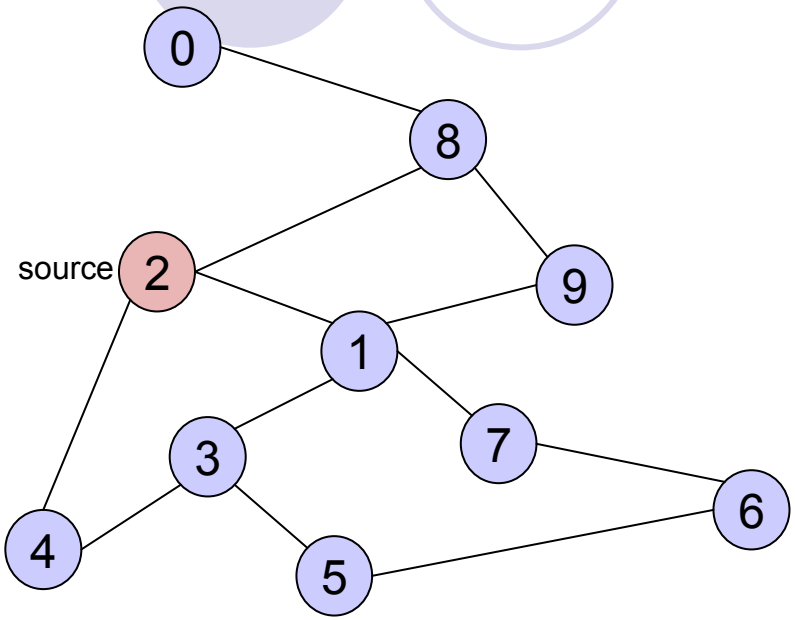
Algorithm *RDFS*(v)
1. $flag[v]$:= true;
2. **for** each neighbor w of v
3. **do if** $flag[w]$ = false
4. **then** *RDFS*(w);

v đã được thăm

Với các nút hàng xóm chưa được thăm, đệ quy RDFS(w)

Chúng ta cũng có thể đánh dấu đường đi bằng pred[ ].

Ví dụ



Danh sách kề

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

Flag (T/F)

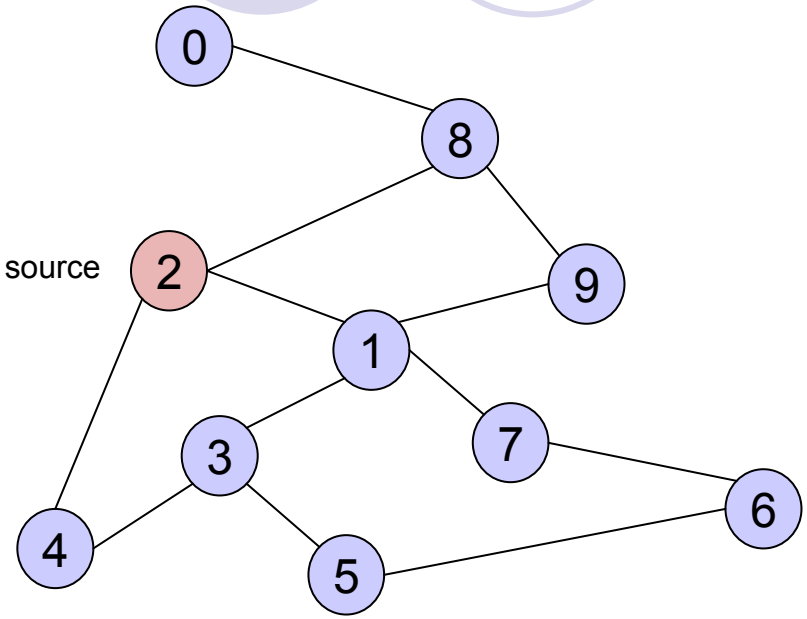
0	F	-
1	F	-
2	F	-
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	F	-
9	F	-

Pred

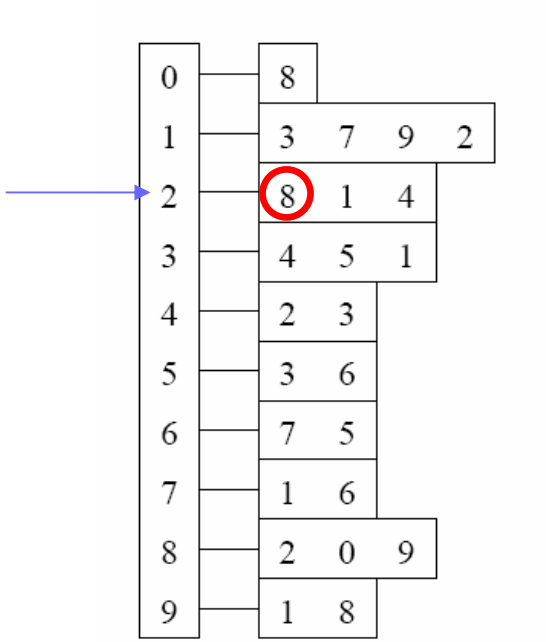
Initialize visited table (all False)

Initialize Pred to -1

Ví dụ



Danh sách kề



Flag (T/F)

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

-
-
-
-
-
-
-
-
-
-

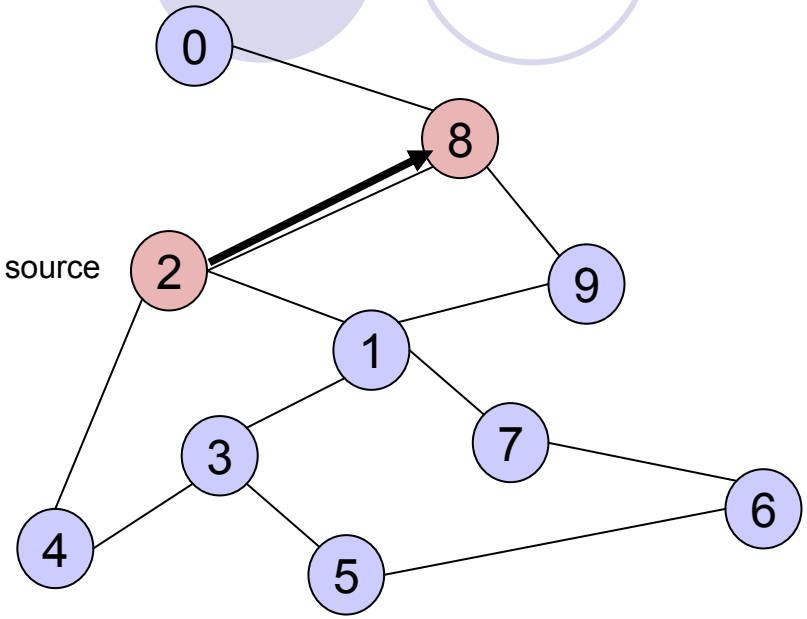
Pred

Mark 2 as visited

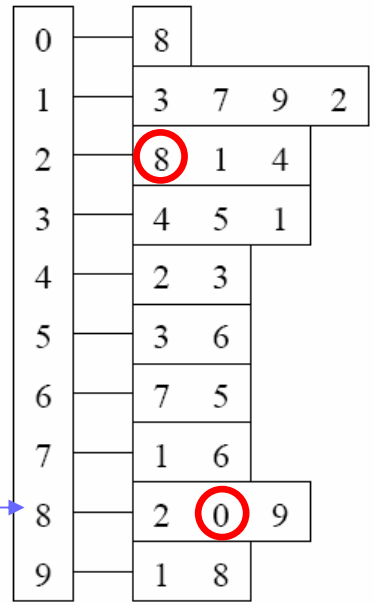
RDFS(2)

Now visit RDFS(8)

Ví dụ



Danh sách kề



Flag (T/F)

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

Pred
-
-
-
-
-
-
-
-
2
-

Mark 8 as visited

mark Pred[8]

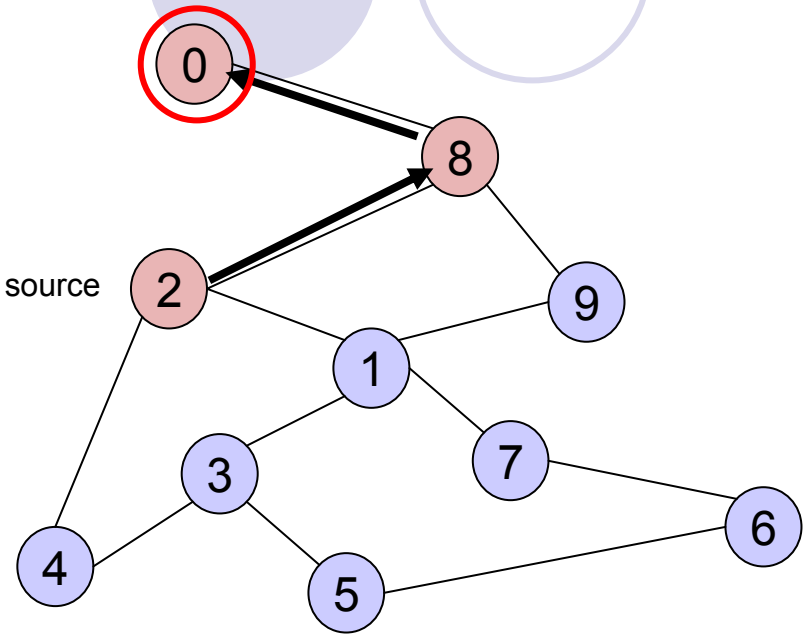
Recursive calls

RDFS(2)

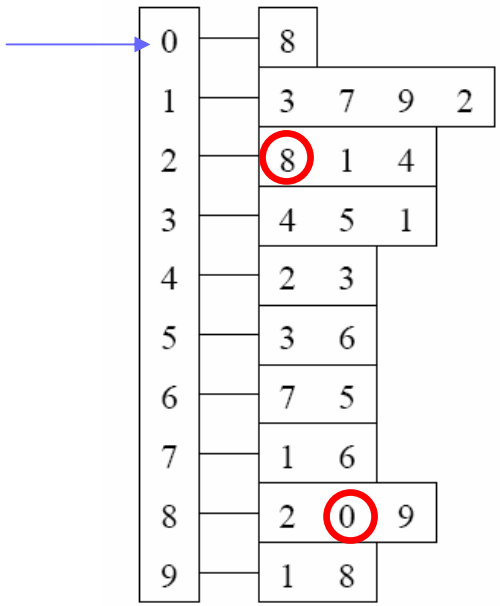
RDFS(8)

2 is already visited, so visit RDFS(0)

Example



Adjacency List



Visited Table (T/F)

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

Pred
8
-
-
-
-
-
-
-
2
-

Mark 0 as visited

Mark Pred[0]

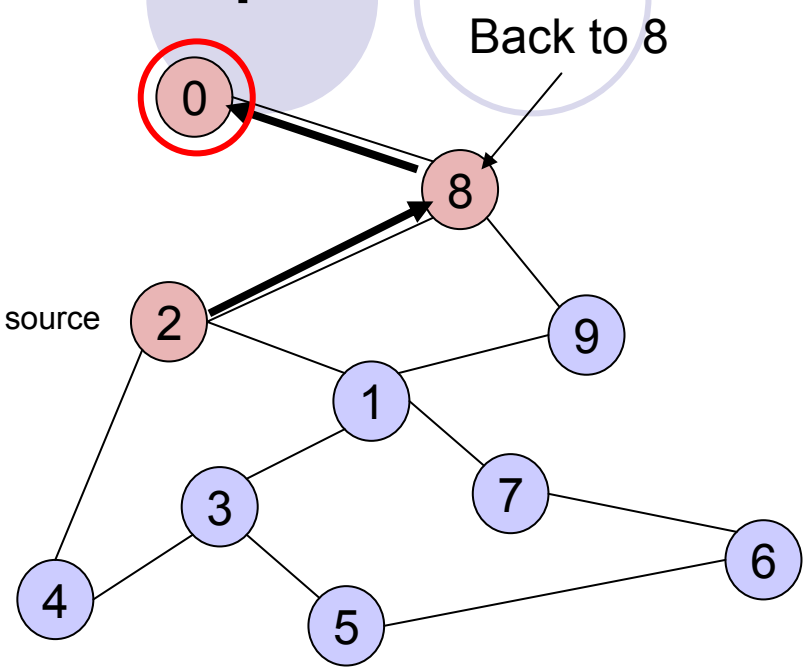
Recursive calls

RDFS(2)

RDFS(8)

RDFS(0) -> no unvisited neighbors, return to call RDFS(8)

Example



Adjacency List

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

Visited Table (T/F)

0	T	8
1	F	-
2	T	-
3	F	-
4	F	-
5	F	-
6	F	-
7	F	-
8	T	2
9	F	-

Pred

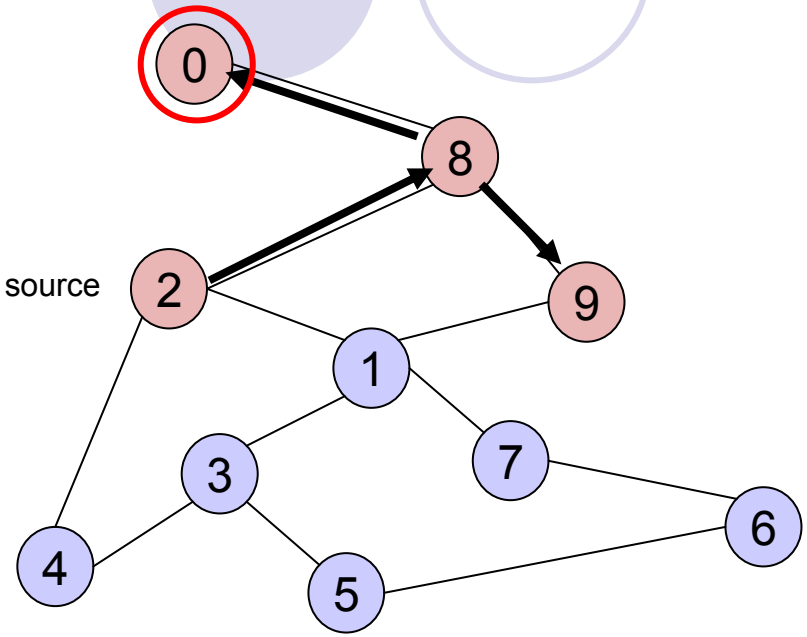
Recursive calls

RDFS(2)

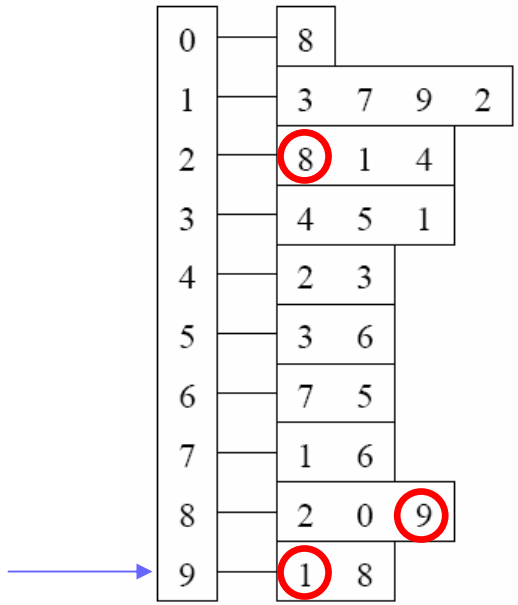
RDFS(8)

Now visit 9 -> RDFS(9)

Example



Adjacency List



Visited Table (T/F)

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

Pred
8
-
-
-
-
-
-
-
2
8

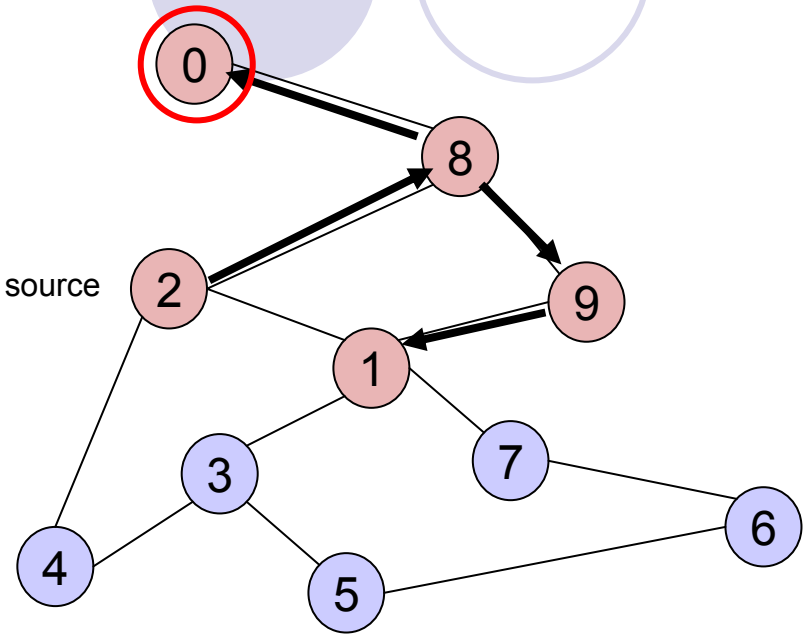
Mark 9 as visited

Mark Pred[9]

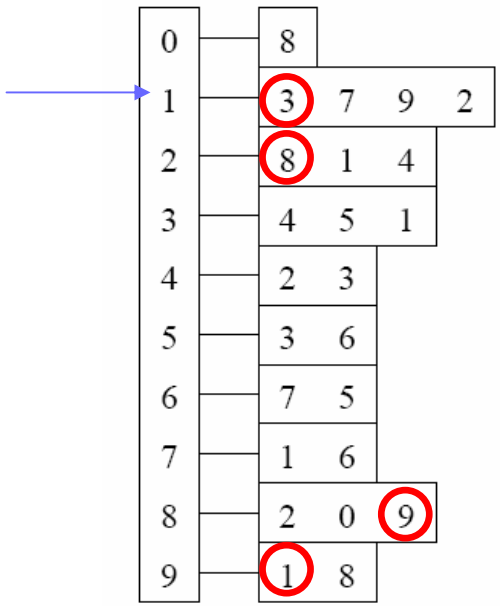
Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
-> visit 1, RDFS(1)

Example



Adjacency List



Visited Table (T/F)

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

8
9
-
-
-
-
-
-
2
8

Pred

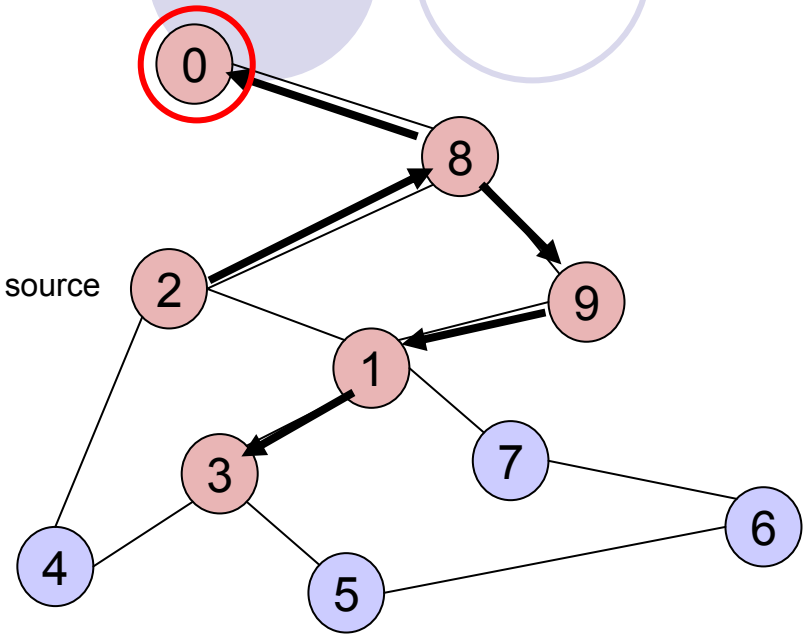
Mark 1 as visited

Mark Pred[1]

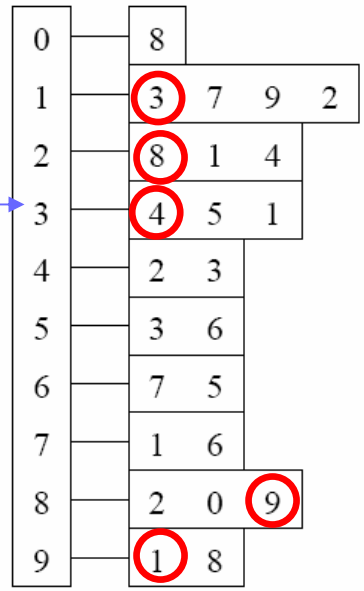
Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
visit RDFS(3)

Example



Adjacency List



Visited Table (T/F)

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

8
9
-
1
-
-
-
-
2
8

Pred

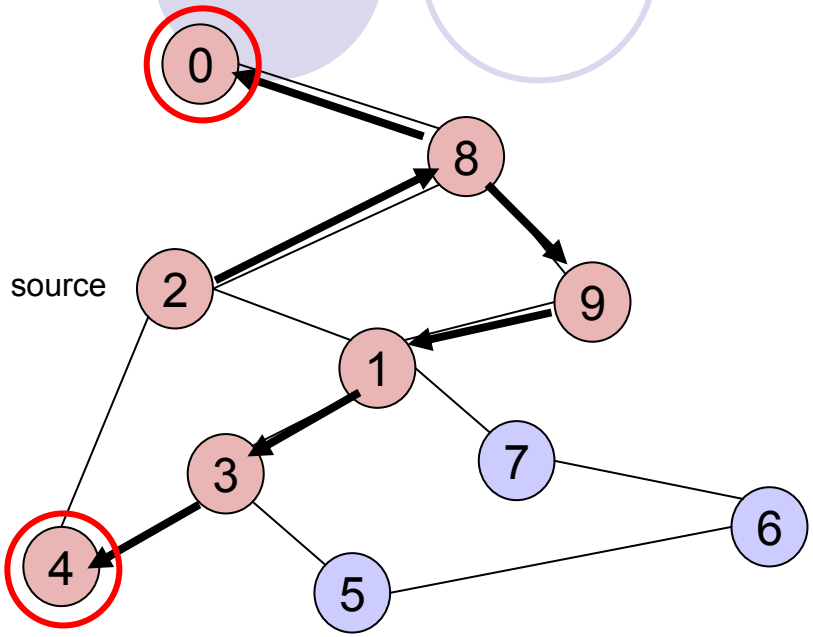
Mark 3 as visited

Mark Pred[3]

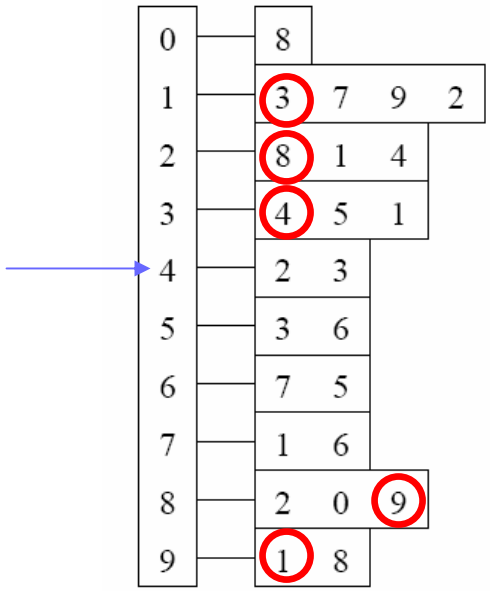
Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
visit RDFS(4)

Example



Adjacency List



Visited Table (T/F)

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

Pred
8
9
-
1
3
-
-
-
2
8

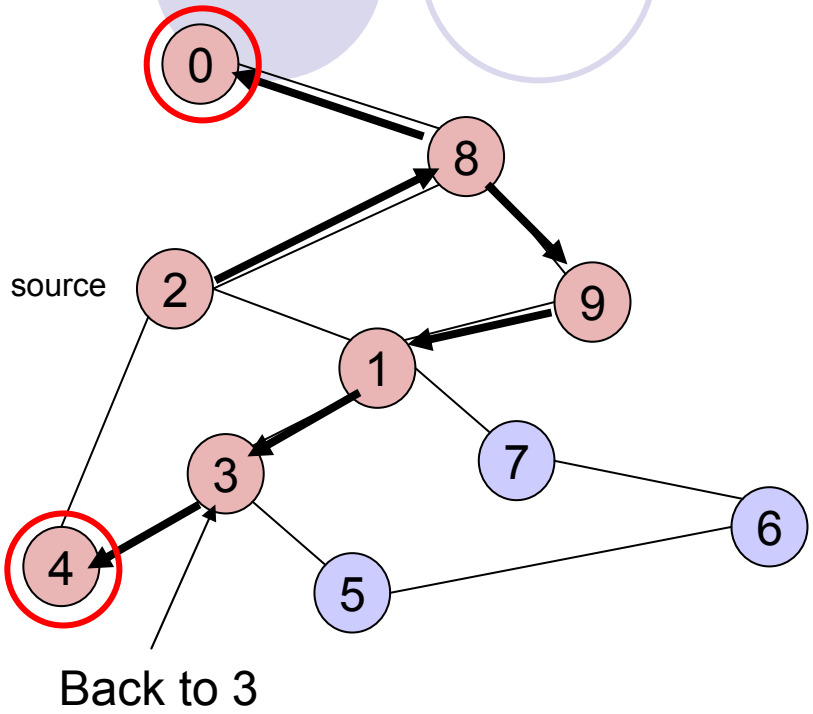
Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(4) → STOP all of 4's neighbors have been visited
return back to call RDFS(3)

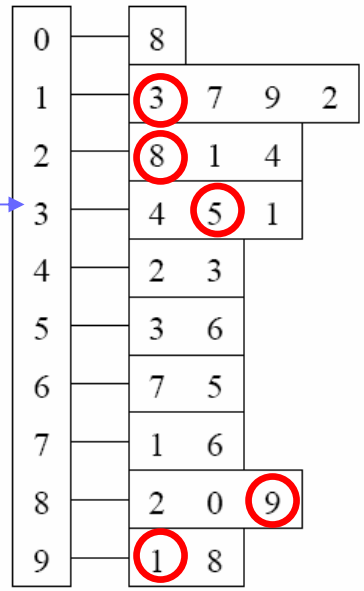
Mark 4 as visited

Mark Pred[4]

Example



Adjacency List



Visited Table (T/F)

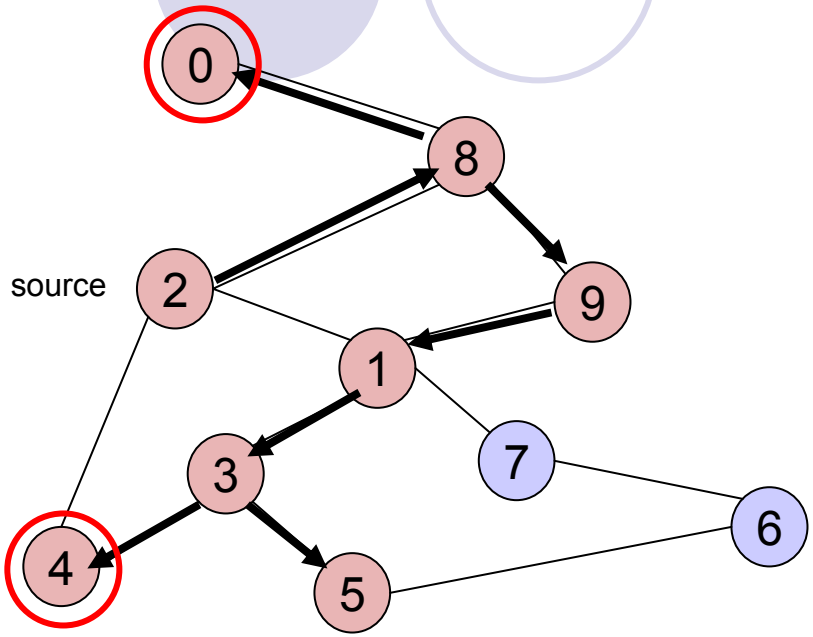
0	T
1	T
2	T
3	T
4	T
5	F
6	F
7	F
8	T
9	T

Pred
8
9
-
1
3
-
-
-
2
8

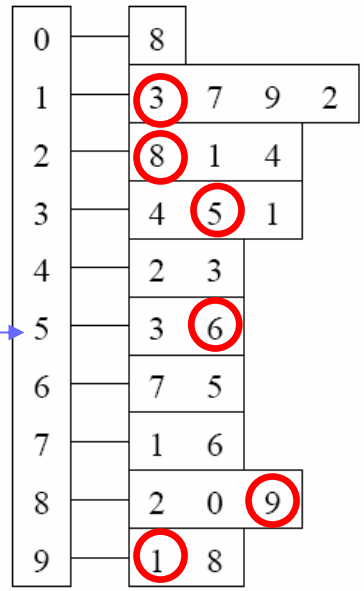
Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
visit 5 -> RDFS(5)

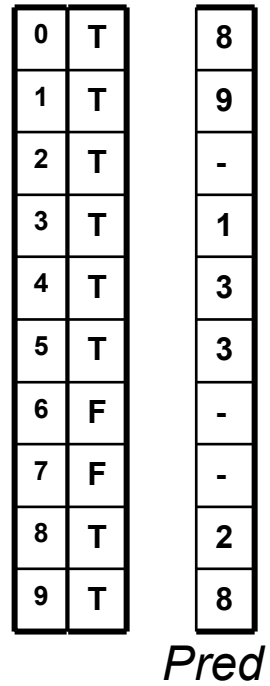
Example



Adjacency List



Visited Table (T/F)



Recursive calls

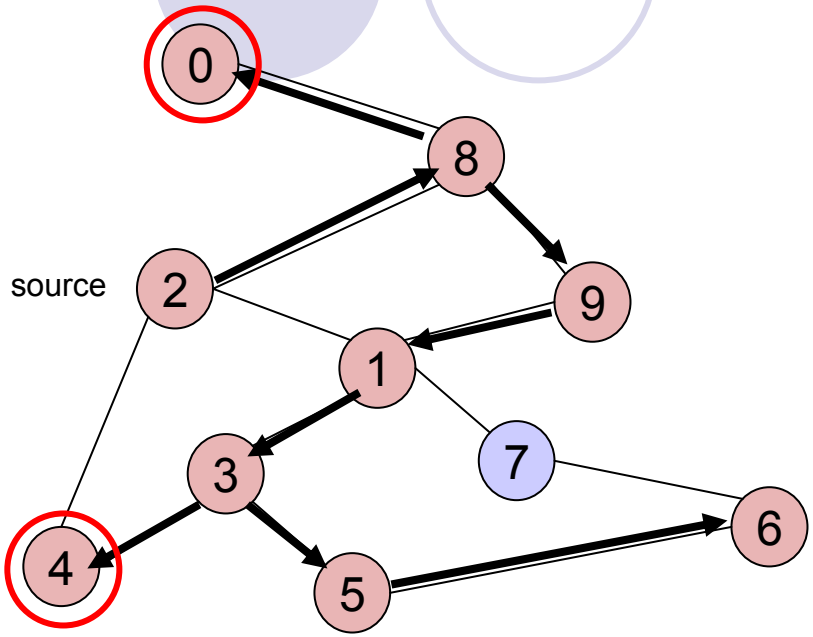
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)

Mark 5 as visited

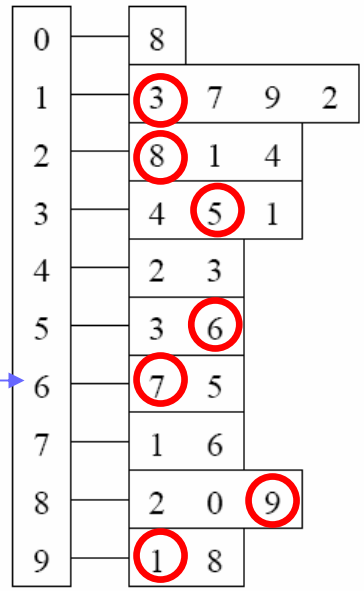
Mark Pred[5]

3 is already visited, so visit 6 -> RDFS(6)

Example



Adjacency List



Visited Table (T/F)

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

Pred
8
9
-
1
3
3
5
-
2
8

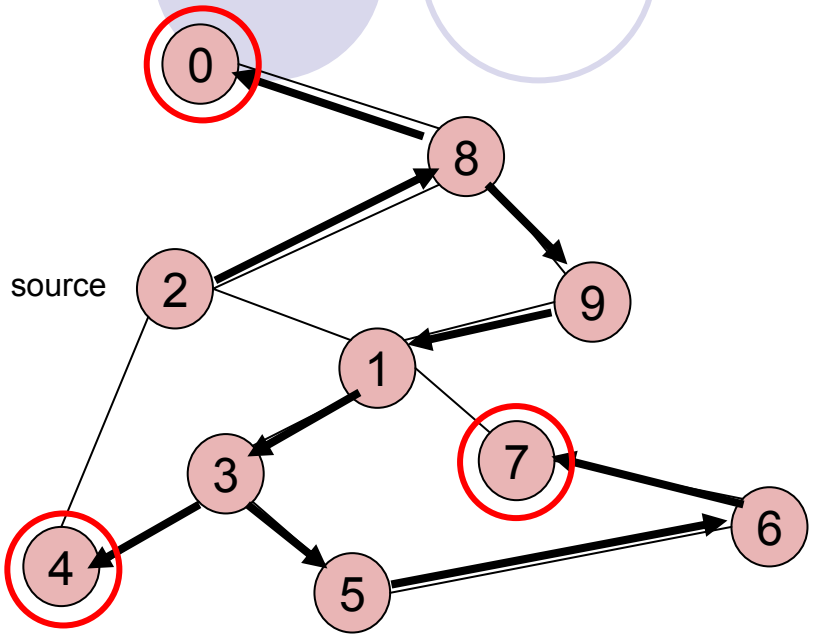
Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(6)
visit 7 -> RDFS(7)

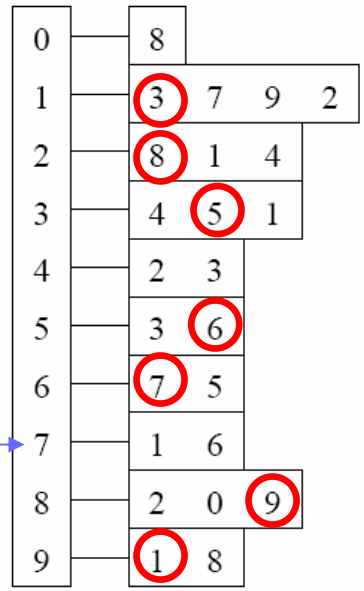
Mark 6 as visited

Mark Pred[6]

Example



Adjacency List



Visited Table (T/F)

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

8
9
-
1
3
3
5
6
2
8

Pred

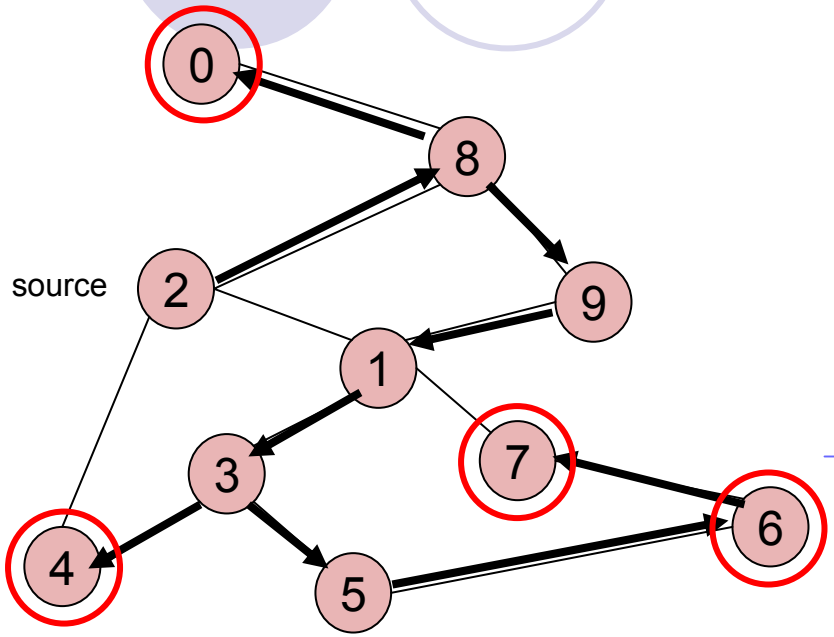
RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(6)
RDFS(7) -> Stop no more unvisited neighbors

Recursive calls

Mark 7 as visited

Mark Pred[7]

Example



Adjacency List

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

Visited Table (T/F)

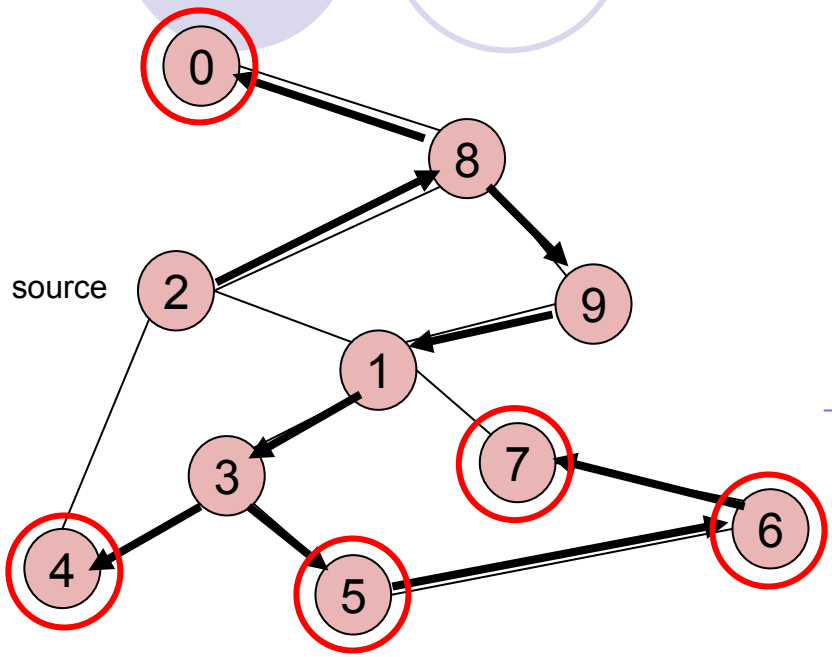
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred
8
9
-
1
3
3
5
6
2
8

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5)
RDFS(6) -> Stop

Example



Adjacency List

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

Visited Table (T/F)

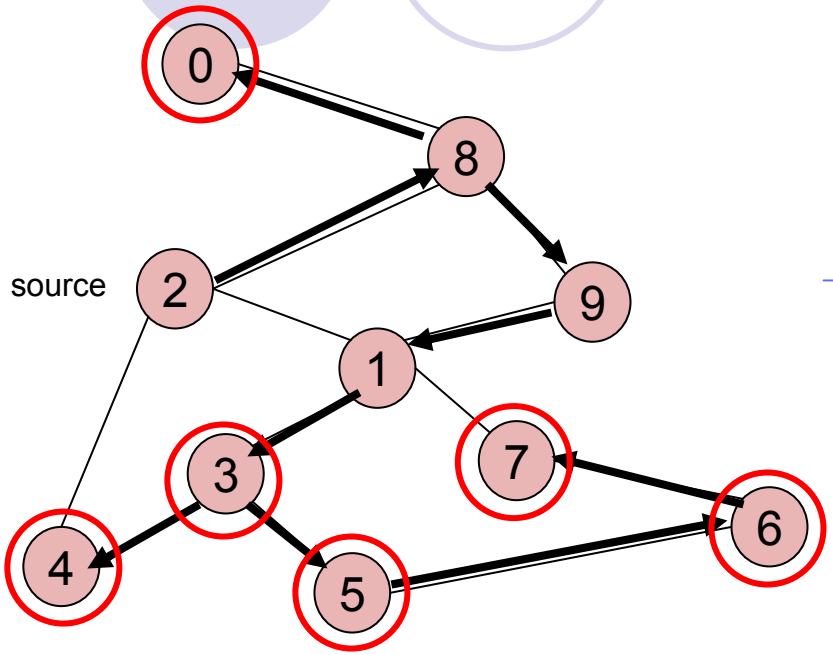
0	T
1	T
2	T
3	T
4	T
5	T
6	T
7	T
8	T
9	T

Pred
8
9
-
1
3
3
5
6
2
8

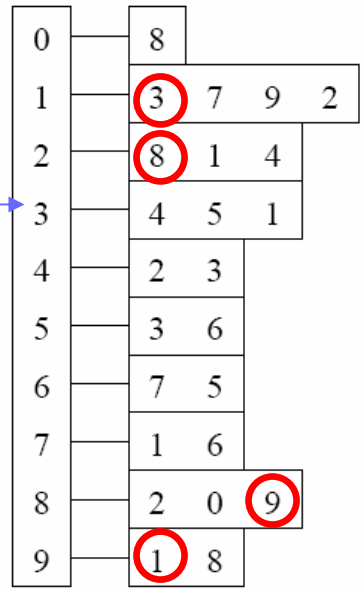
Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3)
RDFS(5) -> Stop

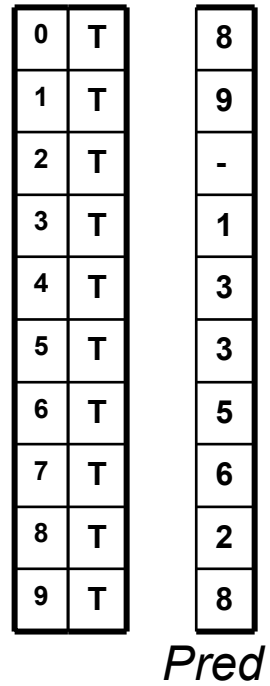
Example



Adjacency List



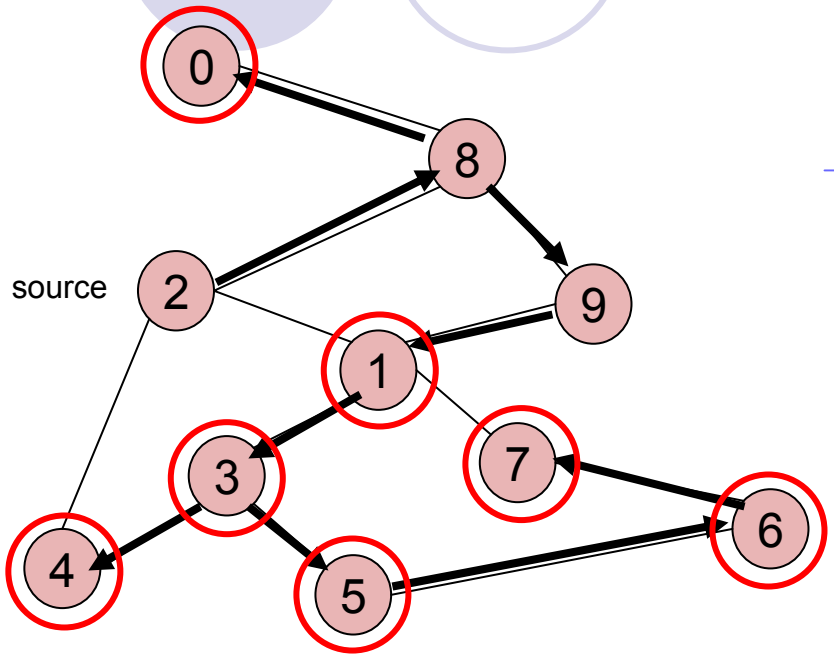
Visited Table (T/F)



Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1)
RDFS(3) -> Stop

Example



Adjacency List

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

Visited Table (T/F)

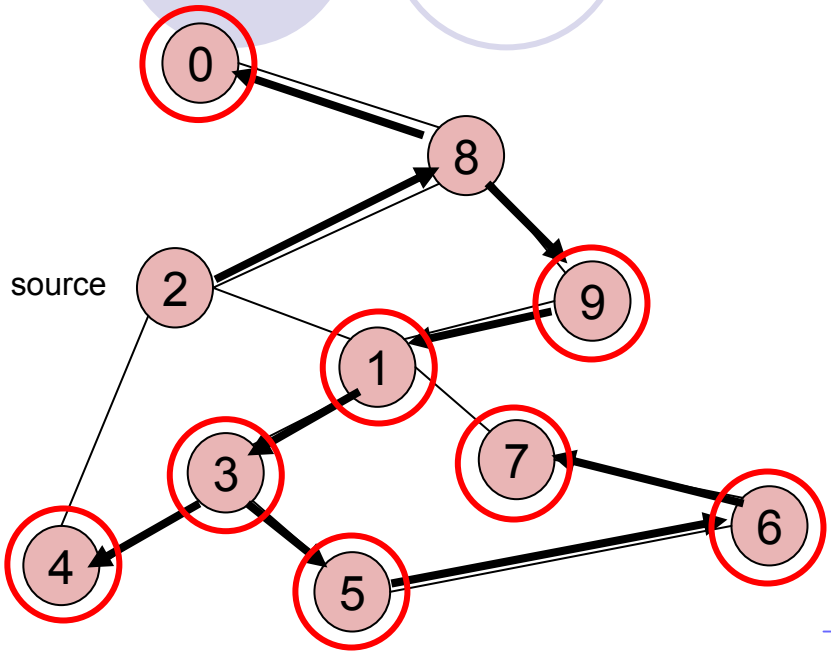
0	T	8
1	T	9
2	T	-
3	T	1
4	T	3
5	T	3
6	T	5
7	T	6
8	T	2
9	T	8

Pred

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9)
RDFS(1) -> Stop

Example



Adjacency List

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

Visited Table (T/F)

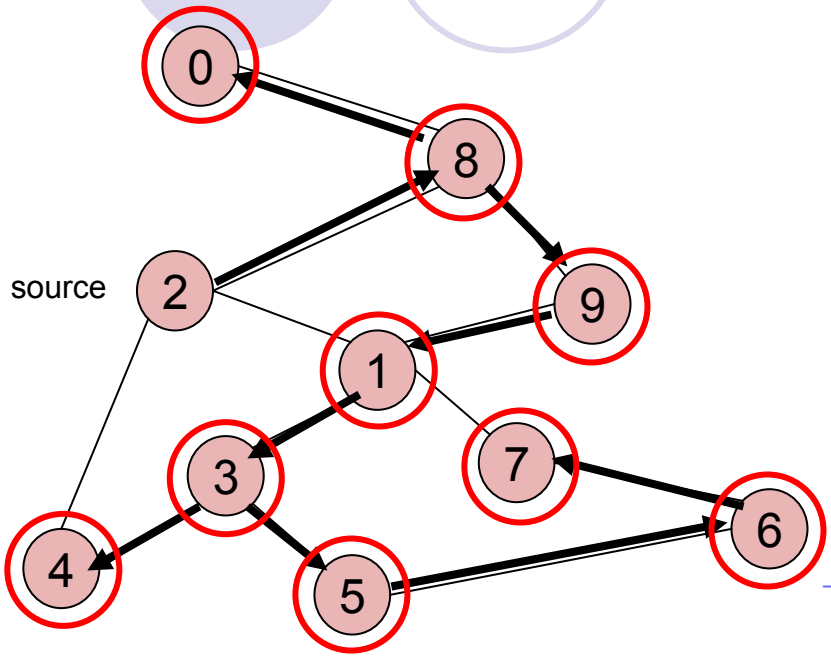
0	T	8
1	T	9
2	T	-
3	T	1
4	T	3
5	T	3
6	T	5
7	T	6
8	T	2
9	T	8

Pred

Recursive calls

RDFS(2)
RDFS(8)
RDFS(9) -> Stop

Example



Adjacency List

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

Visited Table (T/F)

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

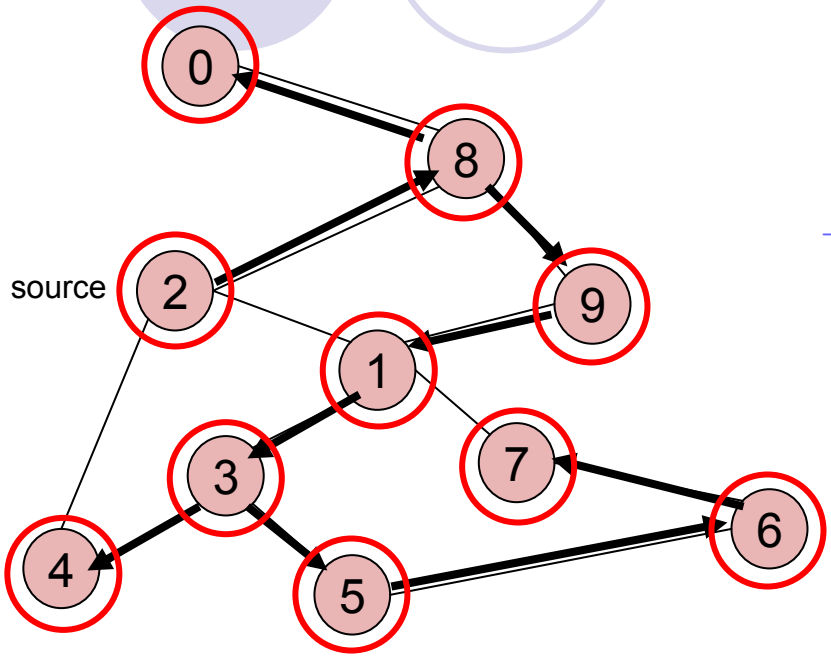
Pred
8
9
-
1
3
3
5
6
2
8

RDFS(2)

RDFS(8) -> Stop

Recursive calls

Example



Adjacency List

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

Visited Table (T/F)

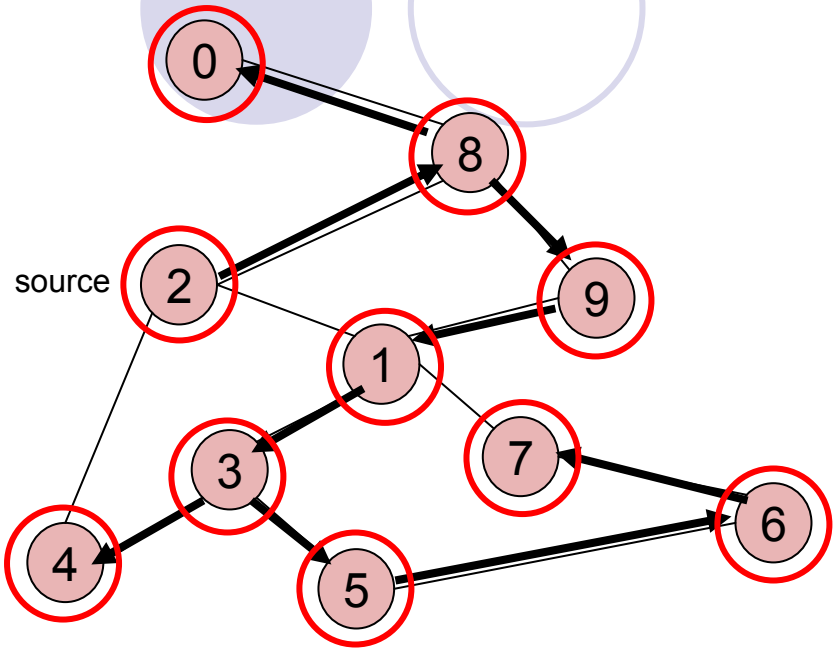
0	T	8
1	T	9
2	T	-
3	T	1
4	T	3
5	T	3
6	T	5
7	T	6
8	T	2
9	T	8

Pred

RDFS(2) -> Stop

Recursive calls

Example



Adjacency List

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

Visited Table (T/F)

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

Pred
8
9
-
1
3
3
5
6
2
8

Check our paths, does DFS find valid paths? Yes.

Algorithm *Path*(w)

1. **if** $pred[w] \neq -1$
2. **then**
3. *Path*$(pred[w])$;
4. output w

Try some examples.
Path(0) ->
Path(6) ->
Path(7) ->

Độ phức tạp thời gian của DFS
(Sử dụng danh sách kề)

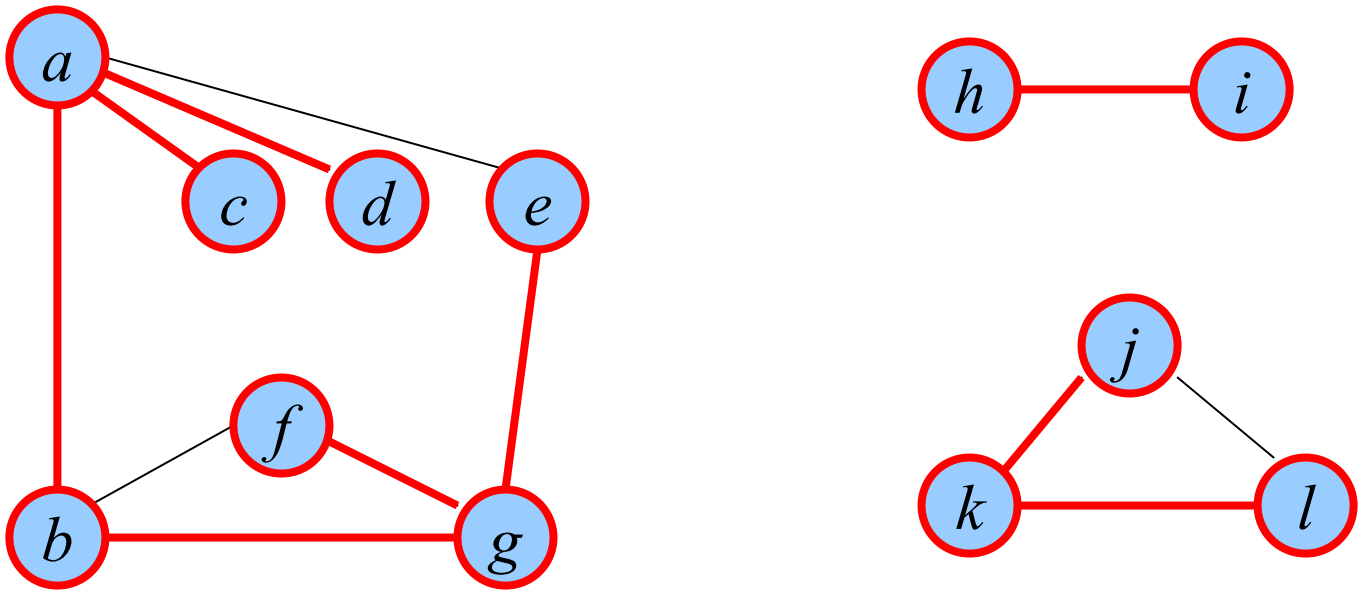
- Không bao giờ thăm một nút quá 1 lần
- Thực hiện kiểm tra tất cả cạnh của một đỉnh
  - Σ_v bậc(v) = 2m với m là tổng số cạnh

- Do vậy thời gian tính của DFS tỉ lệ thuận với số cạnh và số đỉnh (giống BFS)
  - O(n + m)

- Hoặc viết:
  - O(|v|+|e|)

|v| = tổng số đỉnh (n)
|e| = tổng số cạnh (m)

Depth-First Search



- DFS đảm bảo thăm mọi đỉnh liên thông với đỉnh ban đầu.
- Cho phép xác định đồ thị có liên thông không, và tìm các thành phần liên thông của đồ thị.

Ứng dụng của đồ thị

- Bài toán bao đóng truyền ứng (transitive closure)
- Bài toán sắp xếp topo (topological sort)

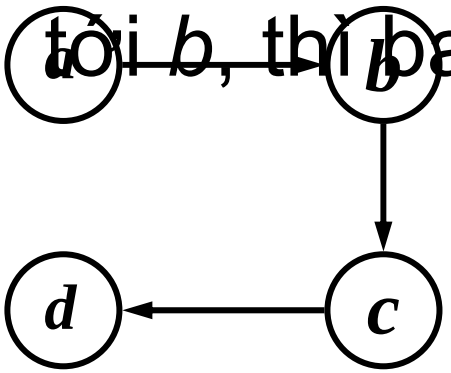
Bài toán bao đóng truyền ứng

- Đặt vấn đề: Cho đồ thị G
  - Có đường đi từ A đến B không?

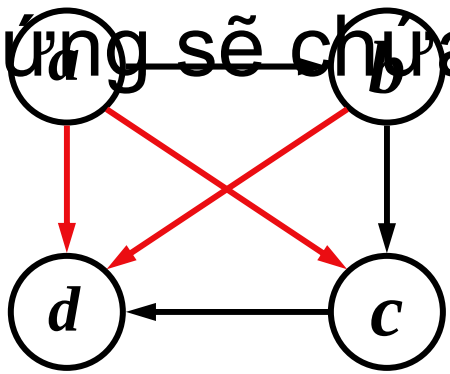
Bao đóng truyền ứng là gì?

- *Bao đóng truyền ứng* của một đồ thị định hướng có cùng số nút với đồ thị ban đầu.
- Nếu có một đường đi định hướng từ nút *a* tới *b*, thì bao đóng truyền ứng sẽ chứa một

Đồ thị



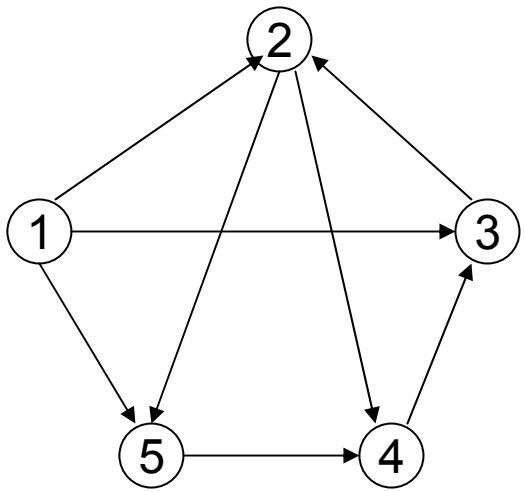
Bao đóng truyền ứng



tới *b*, thì bao đóng truyền ứng sẽ chứa một

Biểu diễn đồ thị dưới dạng ma trận kề

Kích thước của ma trận $|V| \times |V|$



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

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

Bao đóng truyền ứng của *G(V,E)* là

$$A^* = \begin{matrix} 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 \end{matrix}$$

$$a^*_{ij} = \begin{cases} 1 & \text{if there is a path from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$

Bao đóng truyền ứng và phép nhân ma trận

Xét $A^2 = A \times A$

Phép toán logic, AND, OR

$$a_{ij}^2 = V_{x=1}^{n} a_{ix} a_{xj}$$

$$a_{ij}^2 = \begin{cases} 1 & \text{if } (i,j) \in E \\ 1 & \text{if, for some } x,\ (i,x) \in E \text{ and } (x,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Bao đóng truyền ứng và phép nhân ma trận

Xét $A^3 = A^2 \times A$

$$a_{ij}^3 = V_{x=1}^{n} a_{ix}^2 a_{xj}$$

$$a_{ij}^3 = \begin{cases} 1 & \text{if there is a path of length} \\ & \text{at most 3 from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$

Bao đóng truyền ứng và phép nhân ma trận

Xét $A^k = A^{k-1} \times A$

$$a_{ij}^k = V_{x=1}^n a_{ix}^{k-1} a_{xj}$$

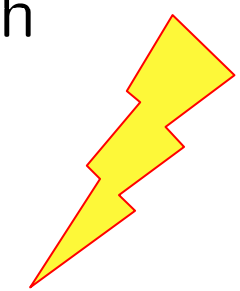
$$a_{ij}^k = \begin{cases} 1 & \text{if there is a path of length} \\ & \text{at most k from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$

Bao đóng truyền ứng và phép nhân ma trận

Xét $A^n = A \times A \times \ldots \times A$,

$$a_{ij}^n = V_{x=1}^n a_{ix}^{n-1} a_{xj}$$

$$a_{ij}^n = \begin{cases} 1 & \text{if there is a path of length} \\ & \text{at most n from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$



$$A^n = A \times A \times \ldots \times A = A^*$$

Giải thuật Warshall

Algorithm *Warshall* (*A*, *P*, *n*)
Input: *A* là ma trận kề biểu diễn đồ thị,
n là số đỉnh của đồ thị
Output: *P* là bao đóng truyền ứng của đồ thị

```
1.  P = A;
2.  for k = 1 to n do
3.      for  i = 1 to n do
4.          for j = 1 to n do
5.              P[i][j] = P[i][j] | P[i][k] & P[k][j];
```

Độ phức tạp của phép nhân 2 ma trận kích thước *nxn*?

$$O(n^3)$$

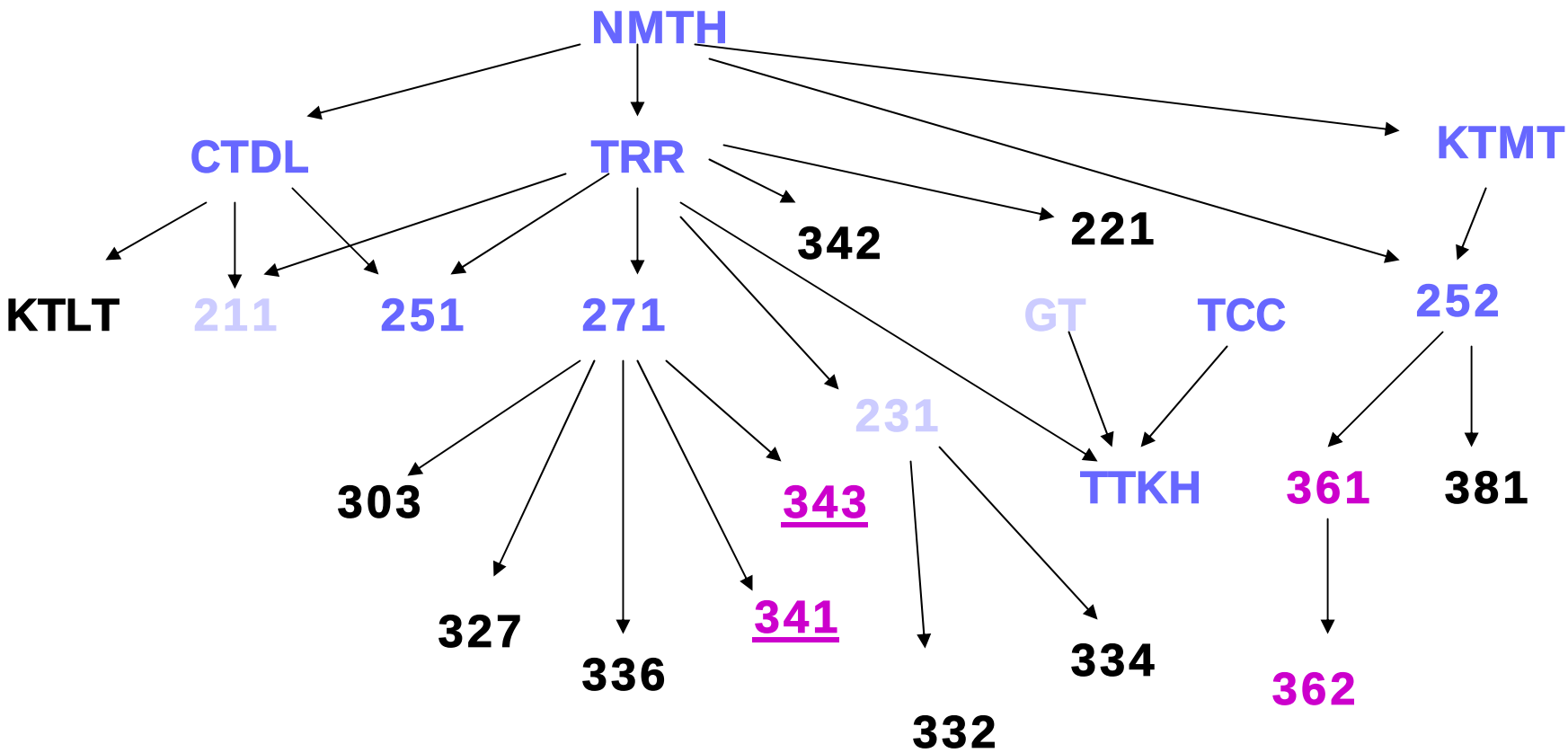
Thực hiện bao nhiêu phép nhân ma trận kích thước *nxn*?

$$O(n)$$

Độ phức tạp $O(n^4)$

Bài toán sắp xếp topo

Ví dụ: Cấu trúc môn học



Đồ thị định hướng, không có chu trình

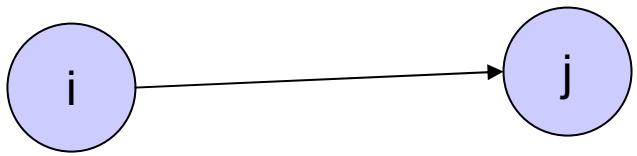
- Một đồ thị định hướng là một chuỗi các đỉnh $(v_0, v_1, \ldots, v_k)$
  - (v_i, v_{i+1}) được gọi là một *cung* (k gọi là cạnh)

- *Một chu trình định hướng* là một đường đi định hướng với đỉnh đầu trùng với đỉnh cuối.

- Một đồ thị định hướng *không có chu trình* nếu nó không chứa bất kỳ chu trình định hướng nào

Ứng dụng của đồ thị định hướng

- Đồ thị định hướng thường được sử dụng để thể hiện các công việc có thứ tự phụ thuộc
- Có nghĩa là công việc này chỉ bắt đầu khi công việc kia kết thúc
- Các quan hệ thứ tự ràng buộc đó được thể hiện bằng các *cung*
- Một *cung (i,j)* có nghĩa là *công việc j* không thể bắt đầu cho đến khi *công việc i* kết thúc



- Rõ ràng, để các ràng buộc không bị lặp vô hạn thì đồ thị phải là không có chu trình.

Bậc vào và bậc ra

- Vì các cạnh là có định hướng
- Phải xem xét các cung là “đi vào” hay “đi ra”
  - Khái niệm
    - Bậc vào(v)
    - Bậc ra(v)

Bậc ra

- Tổng số cung đi “ra” khỏi *v*
- Tổng số bậc ra? (m=#số cạnh)

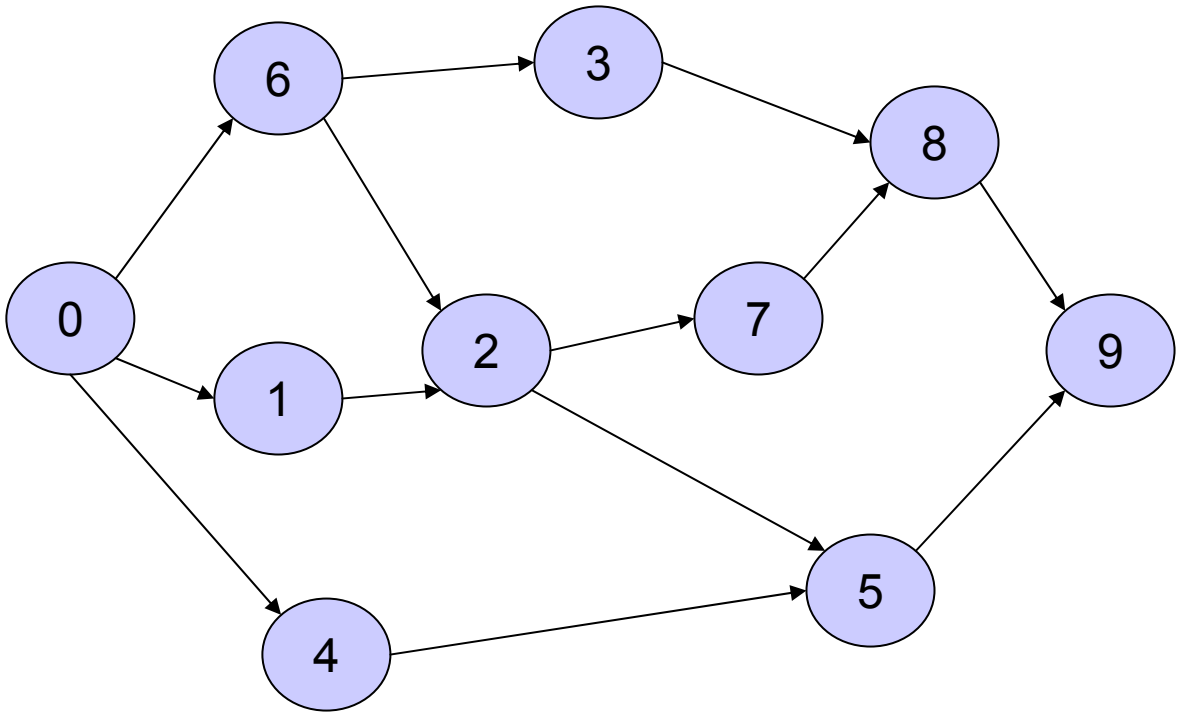
$$\sum_{\text{vertex } v} \text{bac\_ra}(v) = m$$

Bậc vào

- Tổng số cung đi *"vào"* *v*
- Tổng số bậc vào?

$$\sum_{\text{vertex } v} \text{bac\_vao}(v) = m$$

Ví dụ



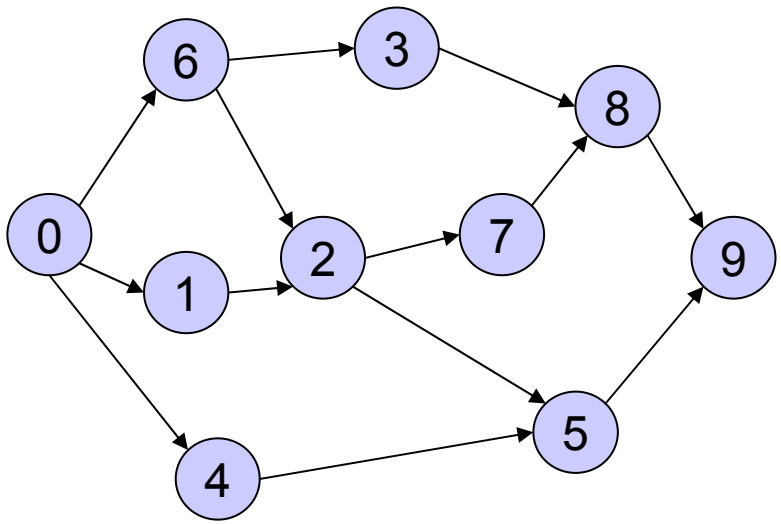
Bậc_vào(2)?

Bậc_vào(8)?

Bậc_ra(0)?

Sắp xếp topo

- Sắp xếp topo là thuật toán cho đồ thị định hướng không có chu trình
- Nó có thể được xem như việc định ra một thứ tự tuyến tính cho các đỉnh, với các quan hệ thứ tự thể hiện bởi các cung



Ví dụ:
0, 1, 2, 5, 9

0, 4, 5, 9

0, 6, 3, 7 ?

Sắp xếp topo

- Ý tưởng:
  - Bắt đầu với đỉnh có bậc vào = 0!
  - Nếu không tồn tại, đồ thị là có chu trình

1. Tìm đỉnh *i* có bậc vào = 0. Ghi vào dãy thứ tự tuyến tính
2. Xóa đỉnh *i* và các cung đi ra khỏi đỉnh *i* khỏi đồ thị
3. Đồ thị mới vẫn là định hướng không có chu trình. Do đó, lặp lại bước 1-2 cho đến khi không còn đỉnh nào trong đồ thị.

Giải thuật

Algorithm $TSort(G)$
Input: a directed acyclic graph G
Output: a topological ordering of vertices

```
1.   initialize Q to be an empty queue;
2.   for each vertex v
3.       do if indegree(v) = 0
4.             then enqueue(Q, v);
5.   while Q is non-empty
6.      do v := dequeue(Q);
7.         output v;
8.         for each arc (v, w)
9.             do indegree(w) = indegree(w) − 1;
10.               if indegree(w) = 0
11.                  then enqueue(w);
```

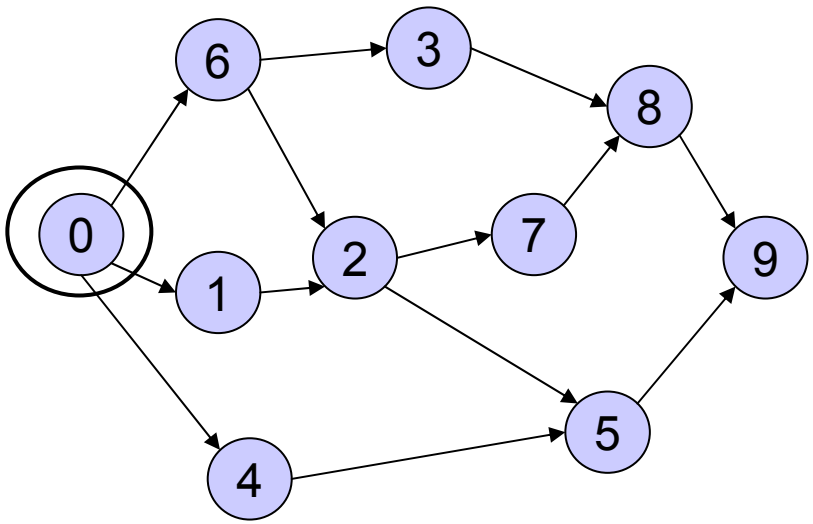
Tìm tất cả đỉnh bắt đầu

Giảm bậc vào(w)

Thêm các đỉnh bắt đầu mới vào Q

The running time is $O(n + m)$.

Ví dụ



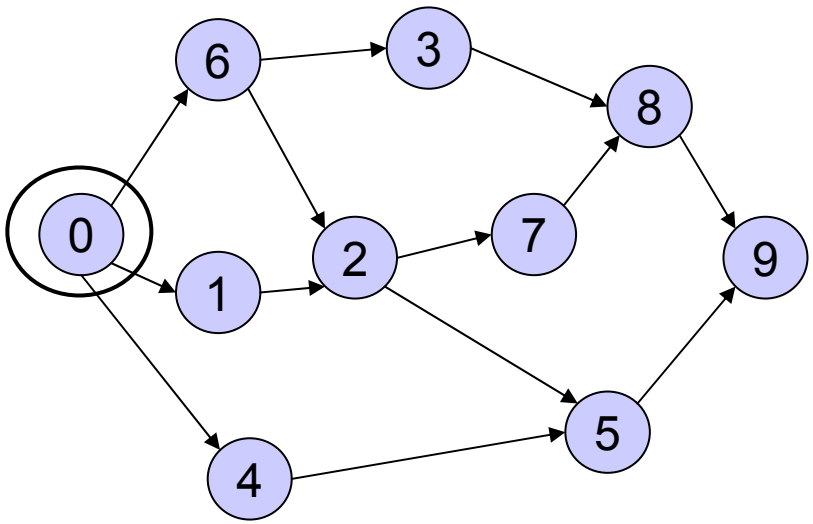
0	6 → 1 → 4
1	2
2	7 → 5
3	8
4	5
5	9
6	3 → 2
7	8
8	9
9	

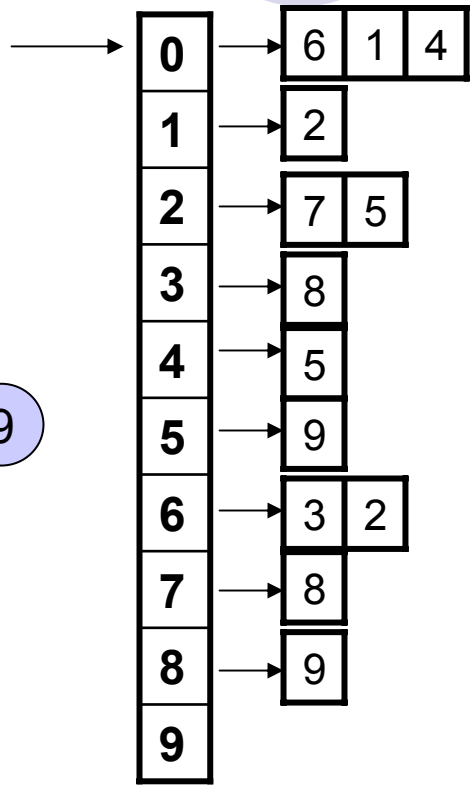
	Bậc vào
0	0 ← start
1	1
2	2
3	1
4	1
5	2
6	1
7	1
8	2
9	2

Q = { 0 }

OUTPUT: **0**

Ví dụ





Bậc vào

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

-1

-1

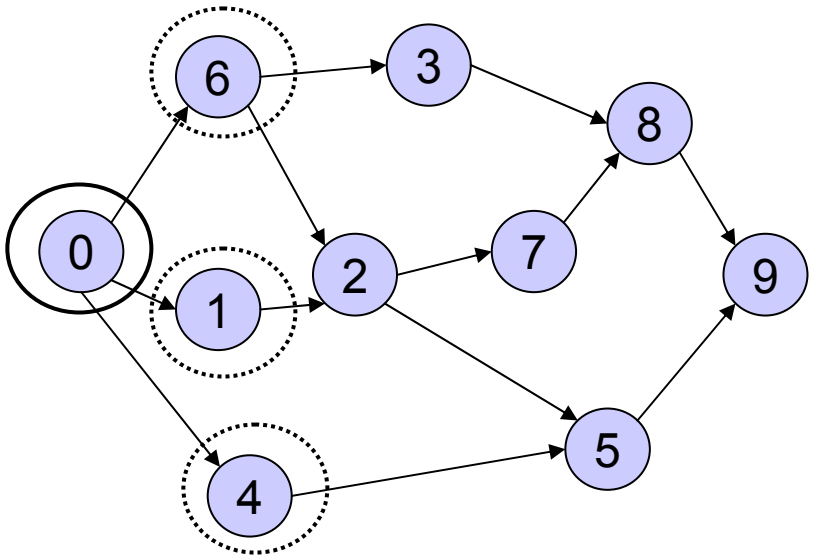
-1

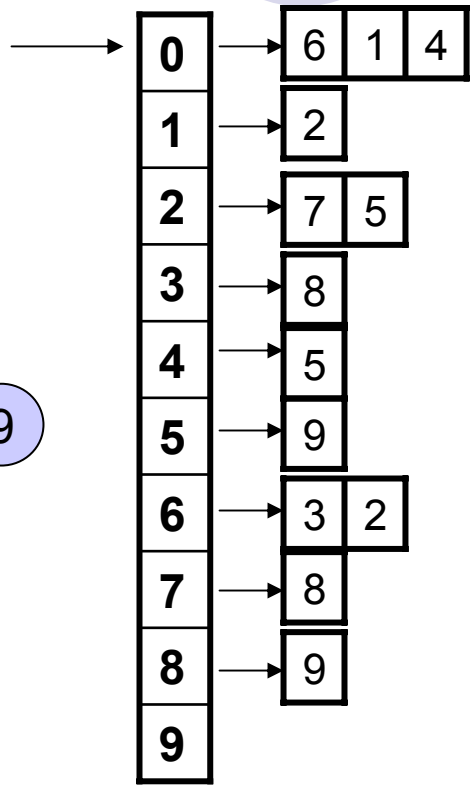
Dequeue 0 Q = { }
-> remove 0's arcs – adjust indegrees of neighbors

Decrement 0's neighbors

OUTPUT:

Ví dụ





Bậc vào

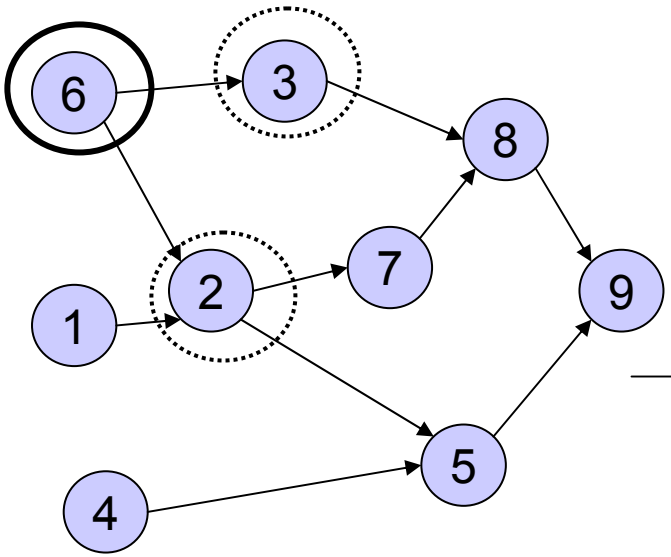
0	0
1	0
2	2
3	1
4	0
5	2
6	0
7	1
8	2
9	2

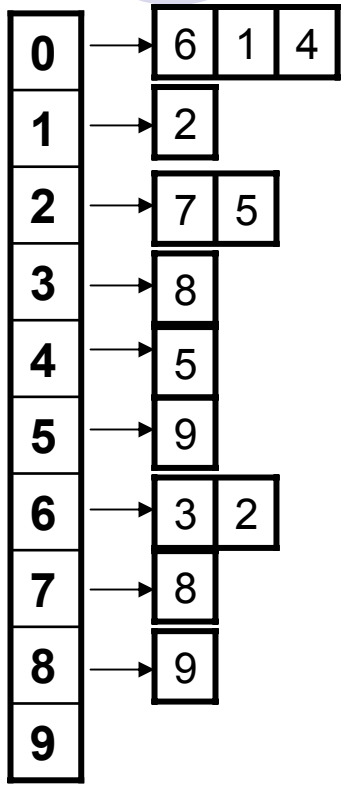
Dequeue 0 Q = { 6, 1, 4 }
Enqueue all starting points

Enqueue all
new start points

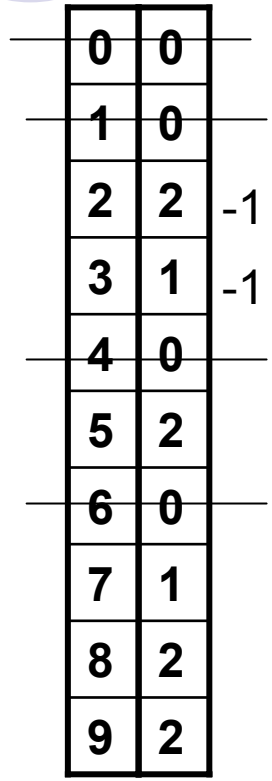
OUTPUT: 0

Ví dụ





Bậc vào



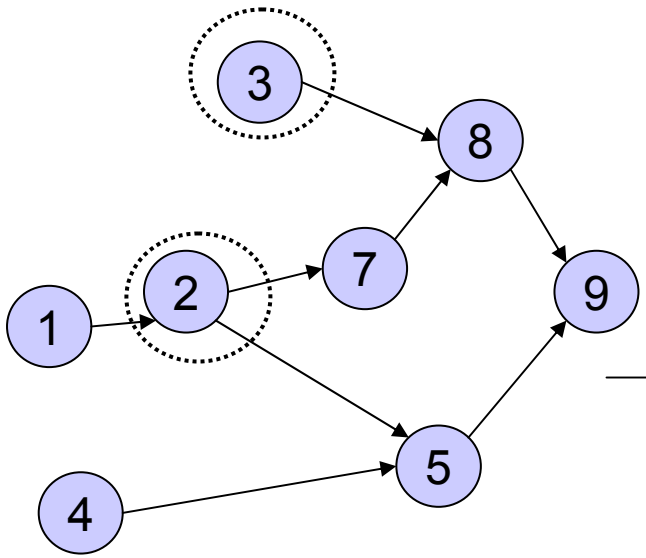
Dequeue 6 Q = { 1, 4 }

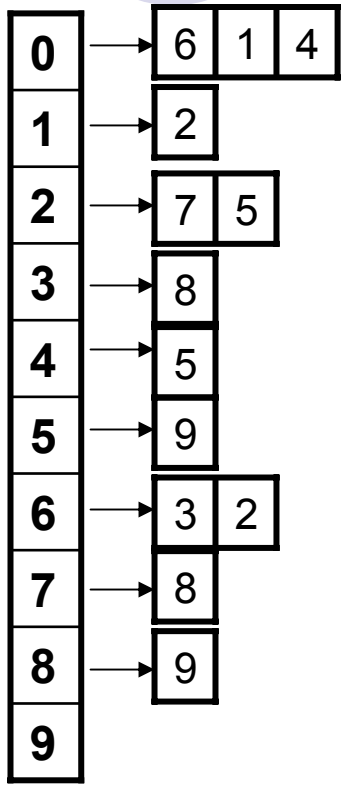
Remove arcs .. Adjust indegrees of neighbors

Adjust neighbors indegree

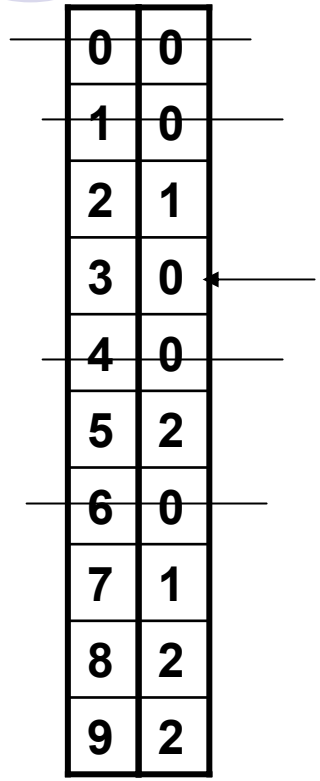
OUTPUT: 0 6

Ví dụ





Bậc vào

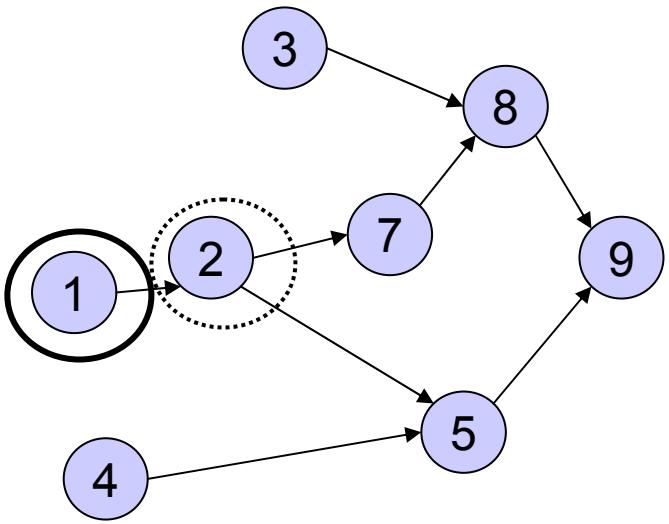


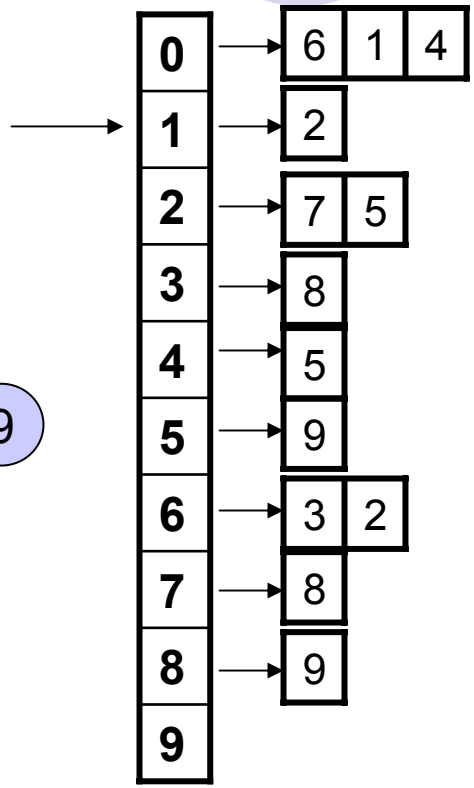
Dequeue 6 Q = { 1, 4, 3 }
Enqueue 3

Enqueue new start

OUTPUT: 0 6

Ví dụ





Bậc vào

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

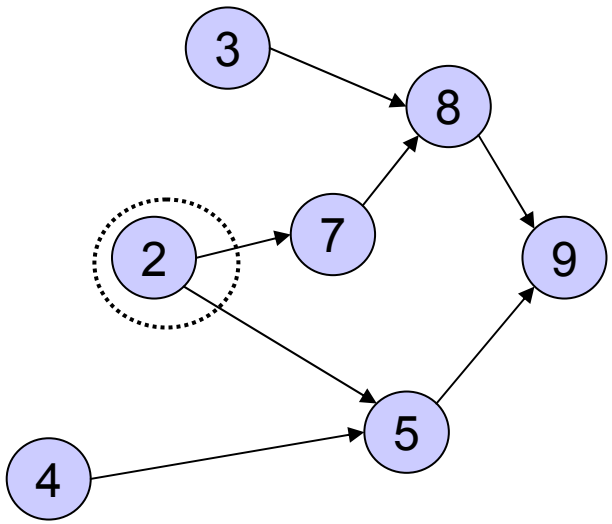
-1

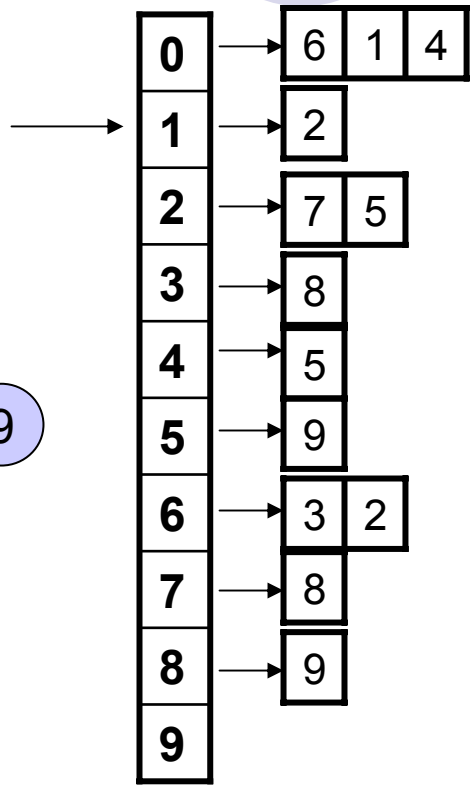
Dequeue 1 Q = { 4, 3 }
Adjust indegrees of neighbors

Adjust neighbors of 1

OUTPUT: 0 6 1

Ví dụ





Bậc vào

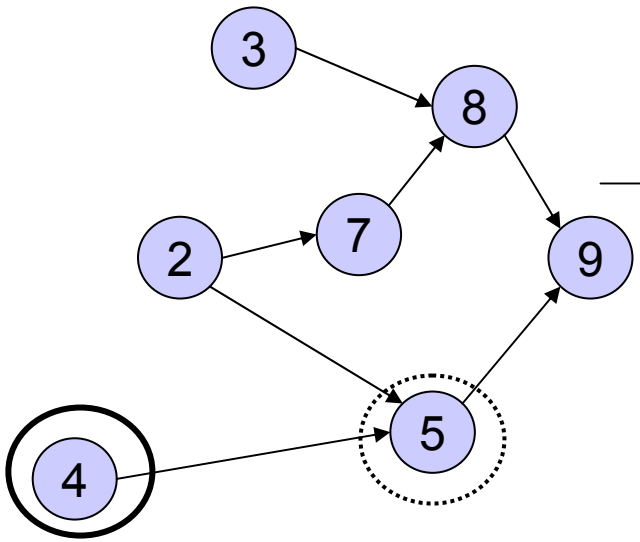
0	0
1	0
2	0
3	0
4	0
5	2
6	0
7	1
8	2
9	2

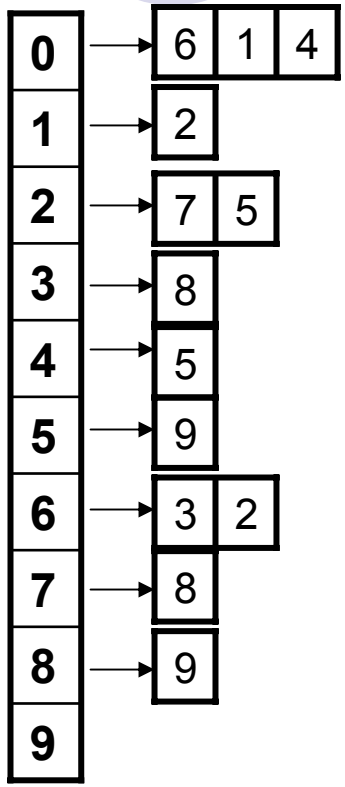
Dequeue 1 Q = { 4, 3, 2 }
Enqueue 2

Enqueue new starting points

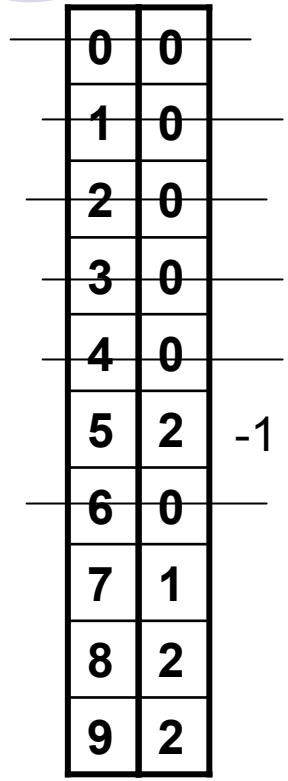
OUTPUT: 0 6 1

Ví dụ





Bậc vào



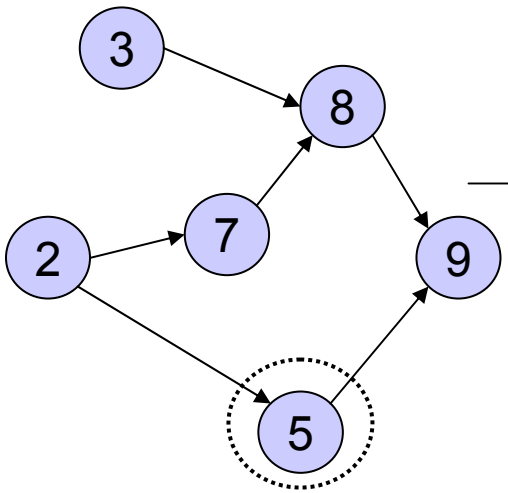
-1

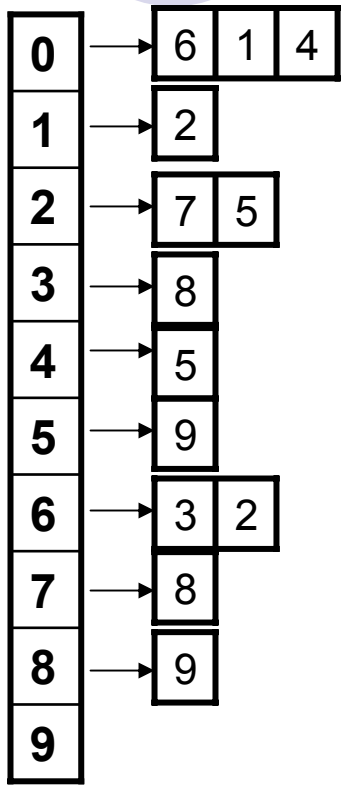
Dequeue 4 Q = { 3, 2 }
Adjust indegrees of neighbors

Adjust 4's neighbors

OUTPUT: 0 6 1 4

Ví dụ





Bậc vào

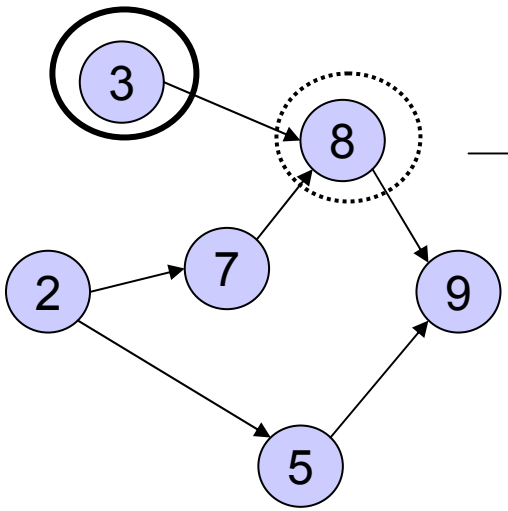
0	0
1	0
2	0
3	0
4	0
5	1
6	0
7	1
8	2
9	2

Dequeue 4 Q = { 3, 2 }
No new start points found

NO new start points

OUTPUT: 0 6 1 4

Ví dụ



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

Bậc vào

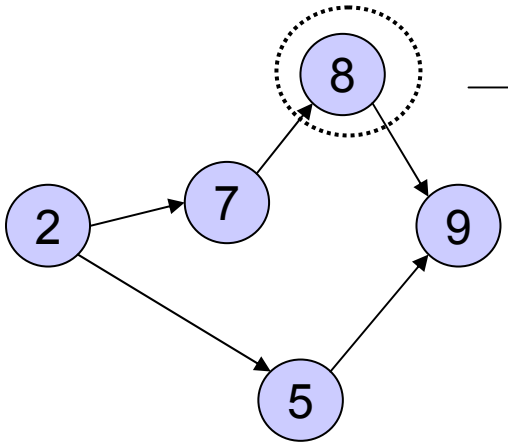
0	0
1	0
2	0
3	0
4	0
5	1
6	0
7	1
8	2
9	2

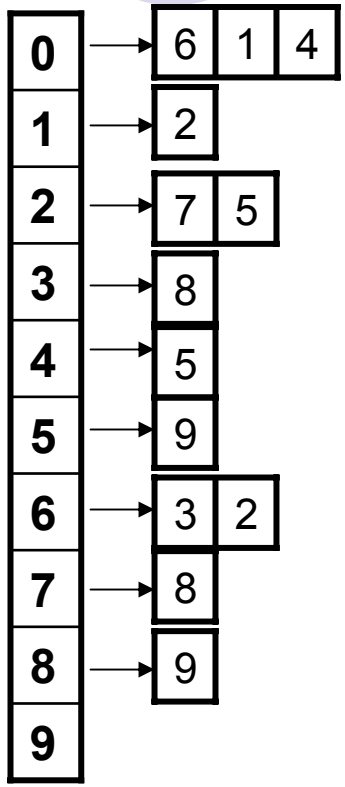
-1

Dequeue 3 Q = { 2 }
Adjust 3's neighbors

OUTPUT: 0 6 1 4 3

Ví dụ





Bậc vào

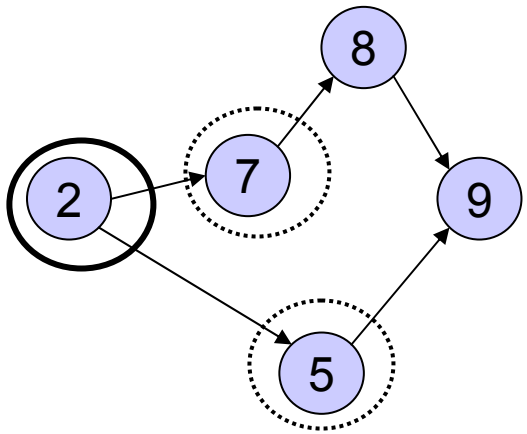
0	0
1	0
2	0
3	0
4	0
5	1
6	0
7	1
8	1
9	2

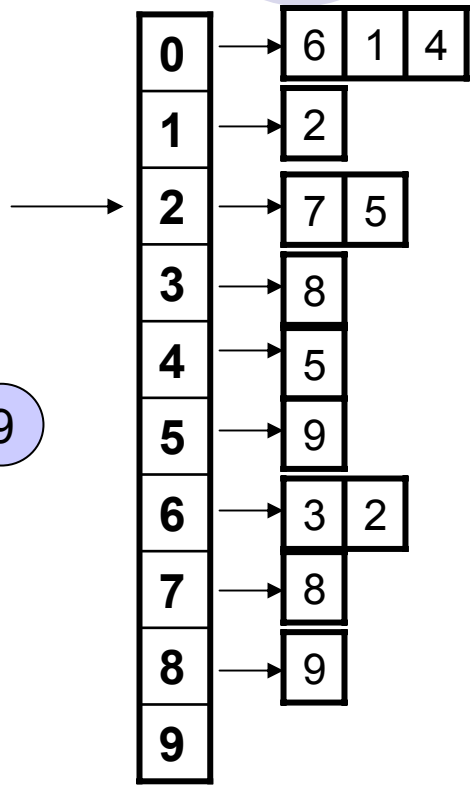
Dequeue 3 Q = { 2 }

No new start points found

OUTPUT: 0 6 1 4 3

Ví dụ





Bậc vào

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

-1

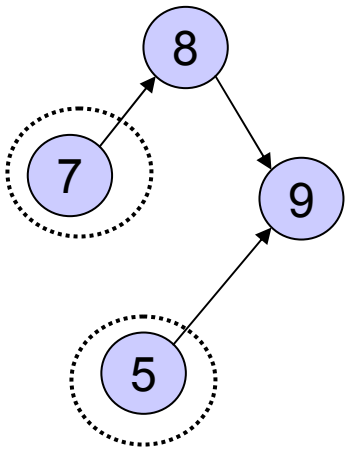
-1

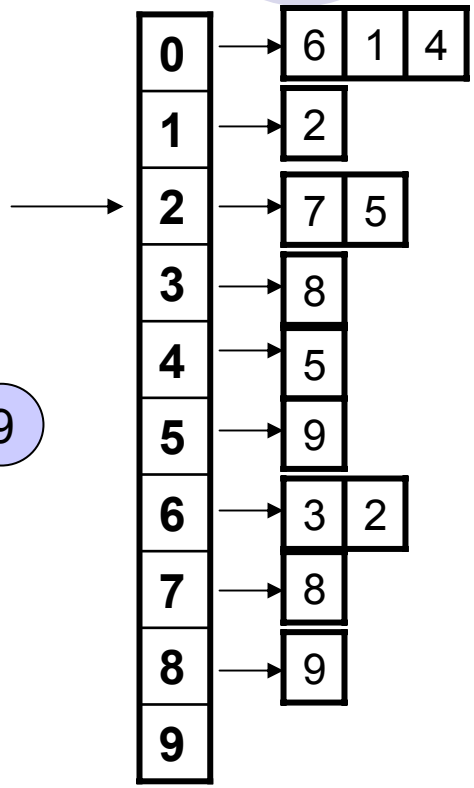
Dequeue 2 Q = { }

Adjust 2's neighbors

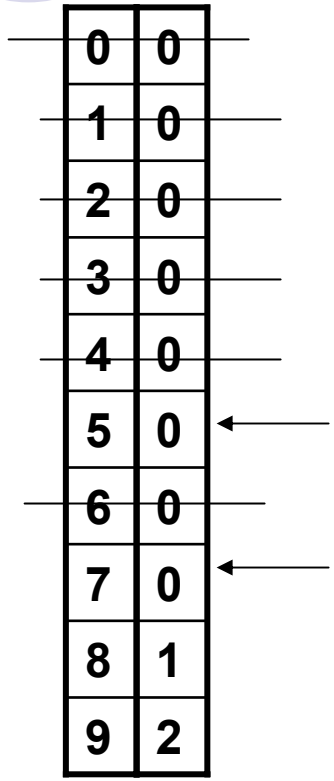
OUTPUT: 0 6 1 4 3 2

Ví dụ





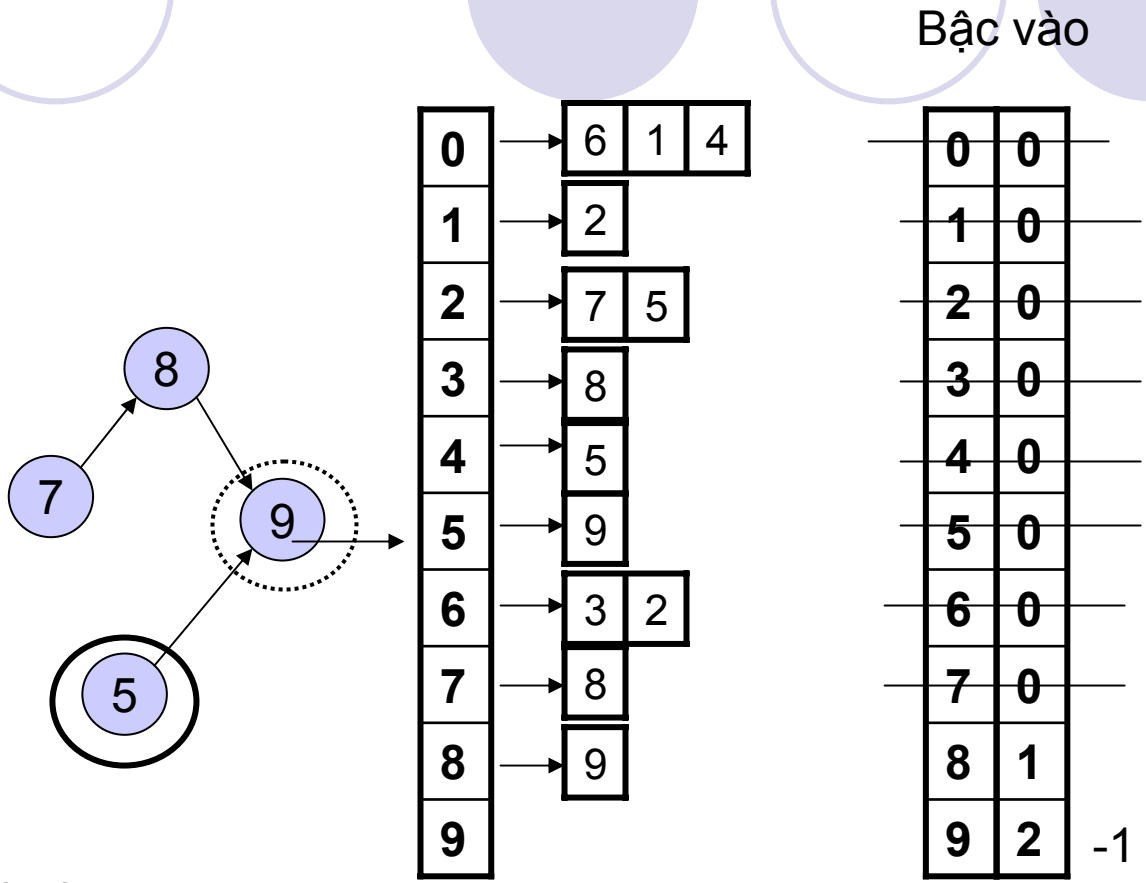
Bậc vào



Dequeue 2 Q = { 5, 7 }
Enqueue 5, 7

OUTPUT: 0 6 1 4 3 2

Ví dụ

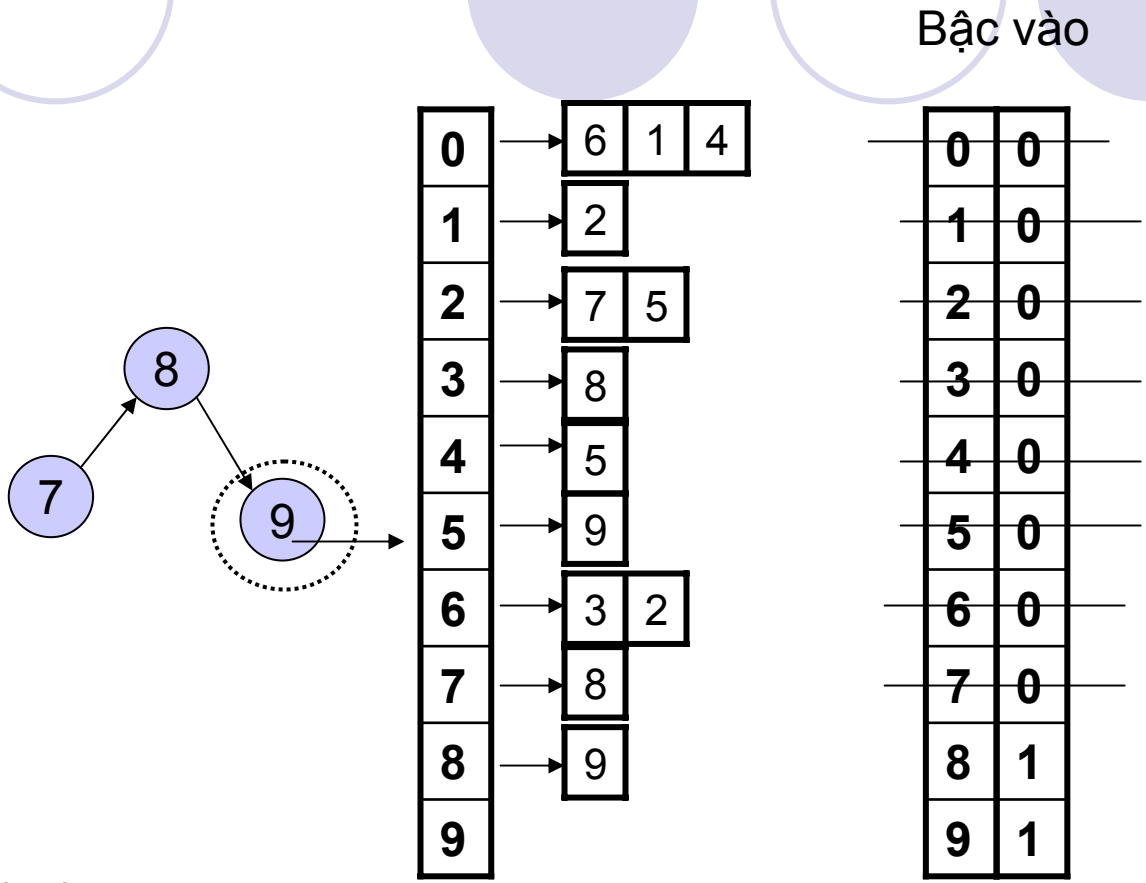


Dequeue 5 Q = { 7 }

Adjust neighbors

OUTPUT: 0 6 1 4 3 2 5

Ví dụ

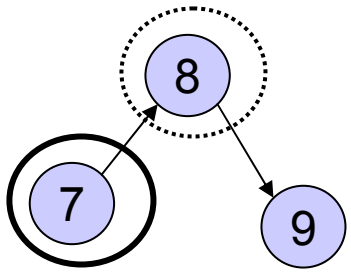


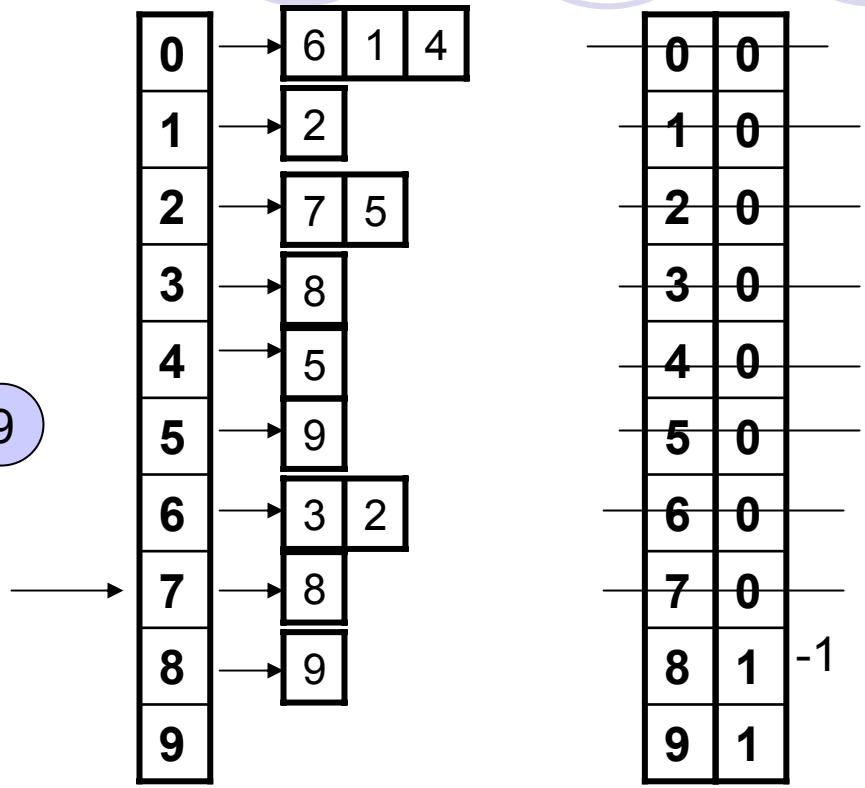
Dequeue 5 Q = { 7 }

No new starts

OUTPUT: 0 6 1 4 3 2 5

Ví dụ



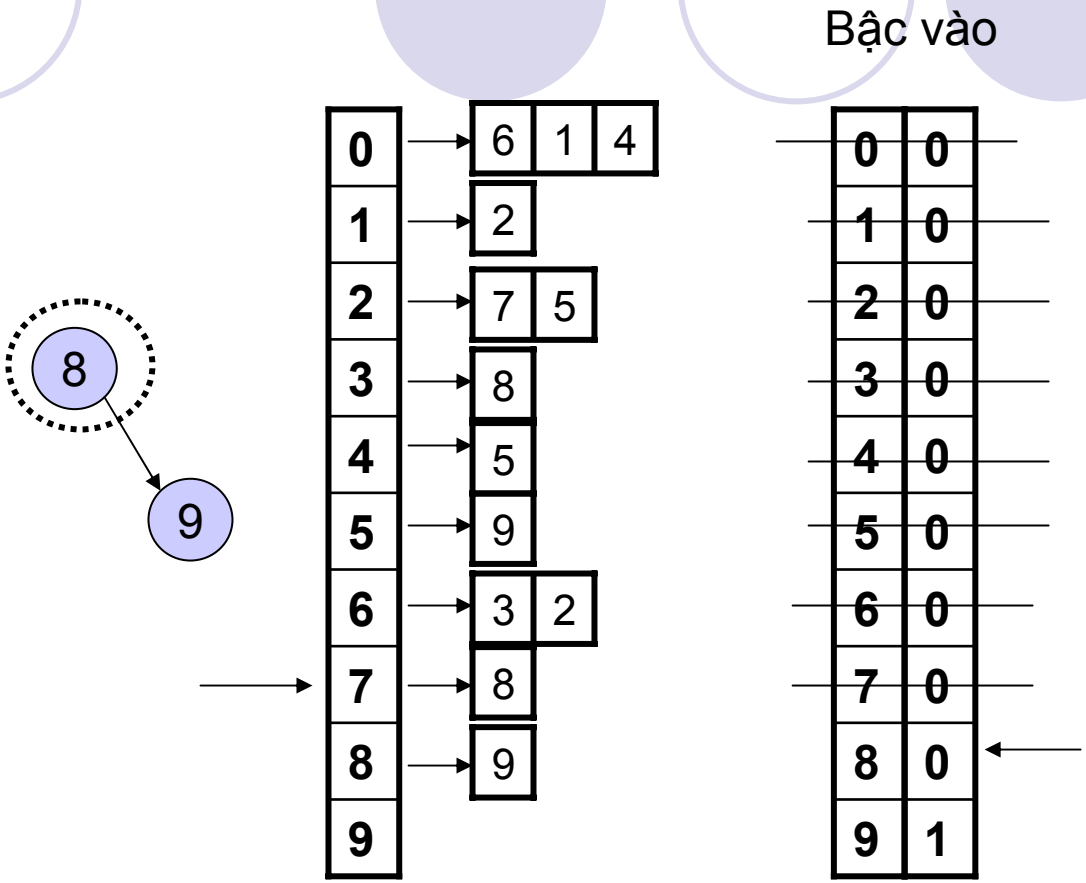


Dequeue 7 Q = { }

Adjust neighbors

OUTPUT: 0 6 1 4 3 2 5 7

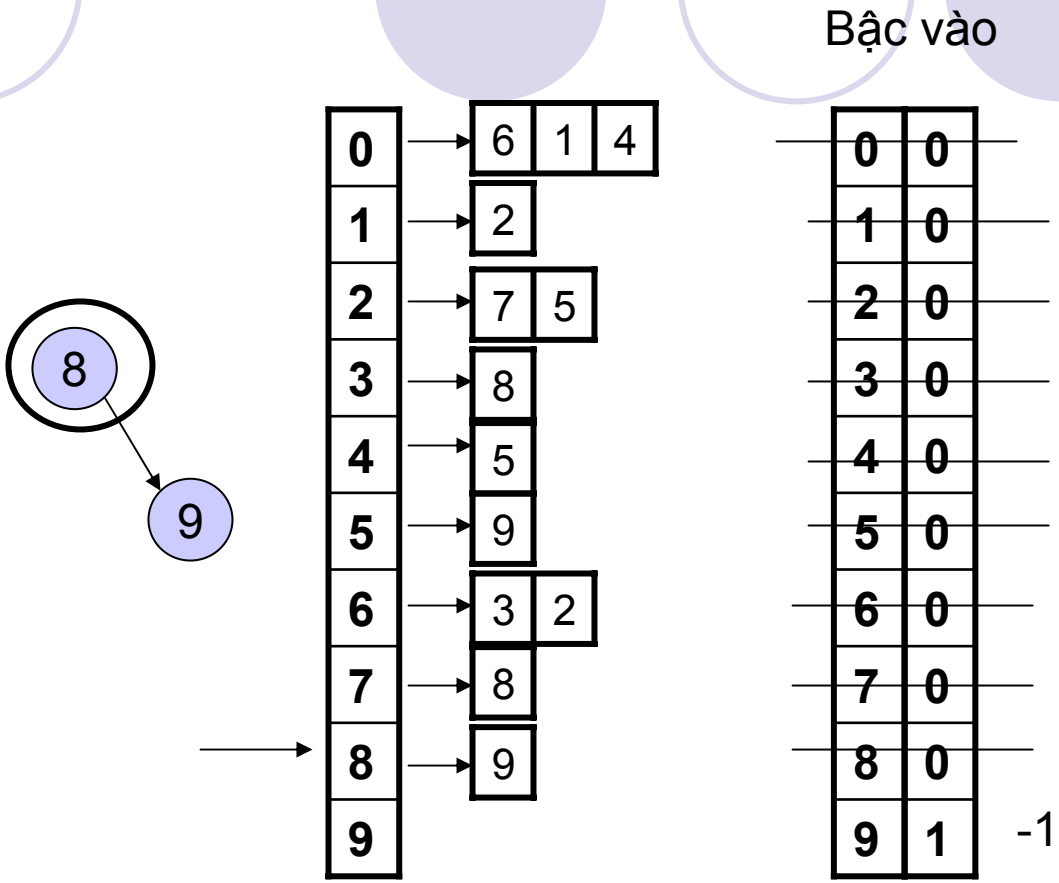
Ví dụ



Dequeue 7 Q = { 8 }
Enqueue 8

OUTPUT: 0 6 1 4 3 2 5 7

Ví dụ

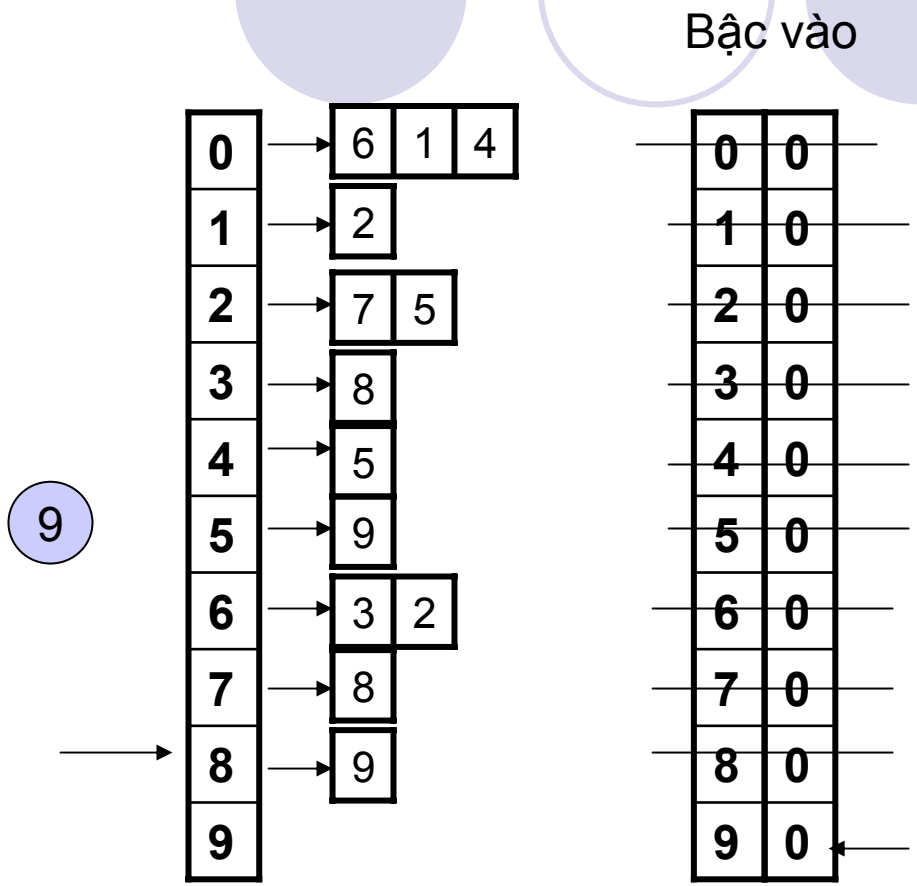


Dequeue 8 Q = { }

Adjust indegrees of neighbors

OUTPUT: 0 6 1 4 3 2 5 7 8

Ví dụ



Dequeue 8 Q = { 9 }

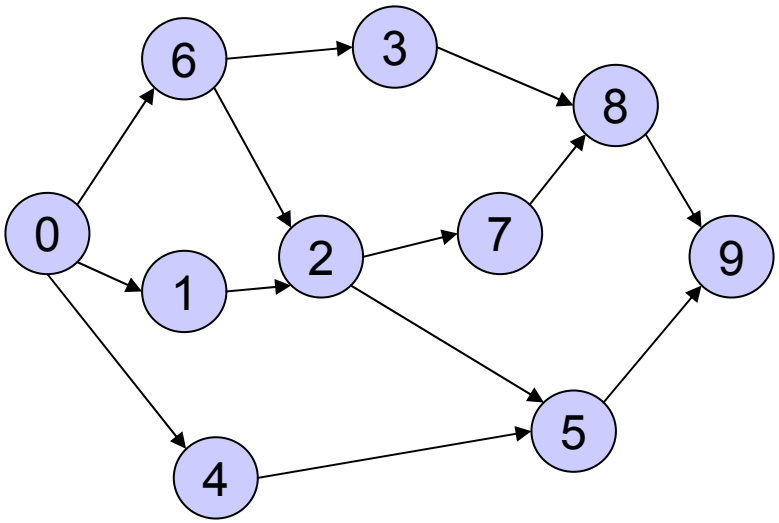
Enqueue 9

Dequeue 9 Q = { }

STOP – no neighbors

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

Ví dụ



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

Sắp xếp topo: Độ phức tạp

- Không bao giờ thăm 1 đỉnh nhiều hơn 1 lần
- Với mỗi đỉnh, phải xét tất cả các cung ra
  - Σ *bậc_ra(v)* = *m*
- Độ phức tạp về thời gian: O(n + m)

Cấu trúc dữ liệu và giải thuật

Đỗ Tuấn Anh

anhdt@it-hut.edu.vn

Nội dung

- Chương 1 – Thiết kế và phân tích (5 tiết)
- Chương 2 – Giải thuật đệ quy (10 tiết)
- Chương 3 – Mảng và danh sách (5 tiết)
- Chương 4 – Ngăn xếp và hàng đợi (10 tiết)
- Chương 5 – Cấu trúc cây (10 tiết)
- Chương 8 – Tìm kiếm (5 tiết)
- Chương 7 – Sắp xếp (10 tiết)
- Chương 6 – Đồ thị (5 tiết)
- Chương 9 – Sắp xếp và tìm kiếm ngoài (after)

Chương 7 – Sắp xếp

1. Đặt vấn đề
2. Ba phương pháp sắp xếp cơ bản
   - Sắp xếp lựa chọn – Selection Sort
   - Sắp xếp thêm dần – Insertion Sort
   - Sắp xếp nổi bọt/đổi chỗ - Bubble Sort
3. Sắp xếp hòa nhập – Merge Sort
4. Sắp xếp nhanh/phân đoạn – Quick Sort
5. Sắp xếp vun đống – Heap Sort

1. Đặt vấn đề

Sắp xếp là các thuật toán bố trí lại các phần tử của một mảng A[n] theo một thứ tự nhất định.

Việc sắp xếp được tiến hành dựa trên khóa của phần tử. Ví dụ: danh mục điện thoại gồm: Tên cơ quan, địa chỉ, số điện thoại.

Đơn giản bài toán:

-Khóa là các giá trị số

-Phần tử chỉ có trường khóa, không có các thành phần khác

-Sắp xếp theo thứ tự tăng dần

2. Ba phương pháp sắp xếp cơ bản

- Sắp xếp lựa chọn – Selection Sort
- Sắp xếp thêm dần – Insertion Sort
- Sắp xếp nổi bọt/đổi chỗ - Bubble Sort

Sắp xếp lựa chọn (Selection Sort)

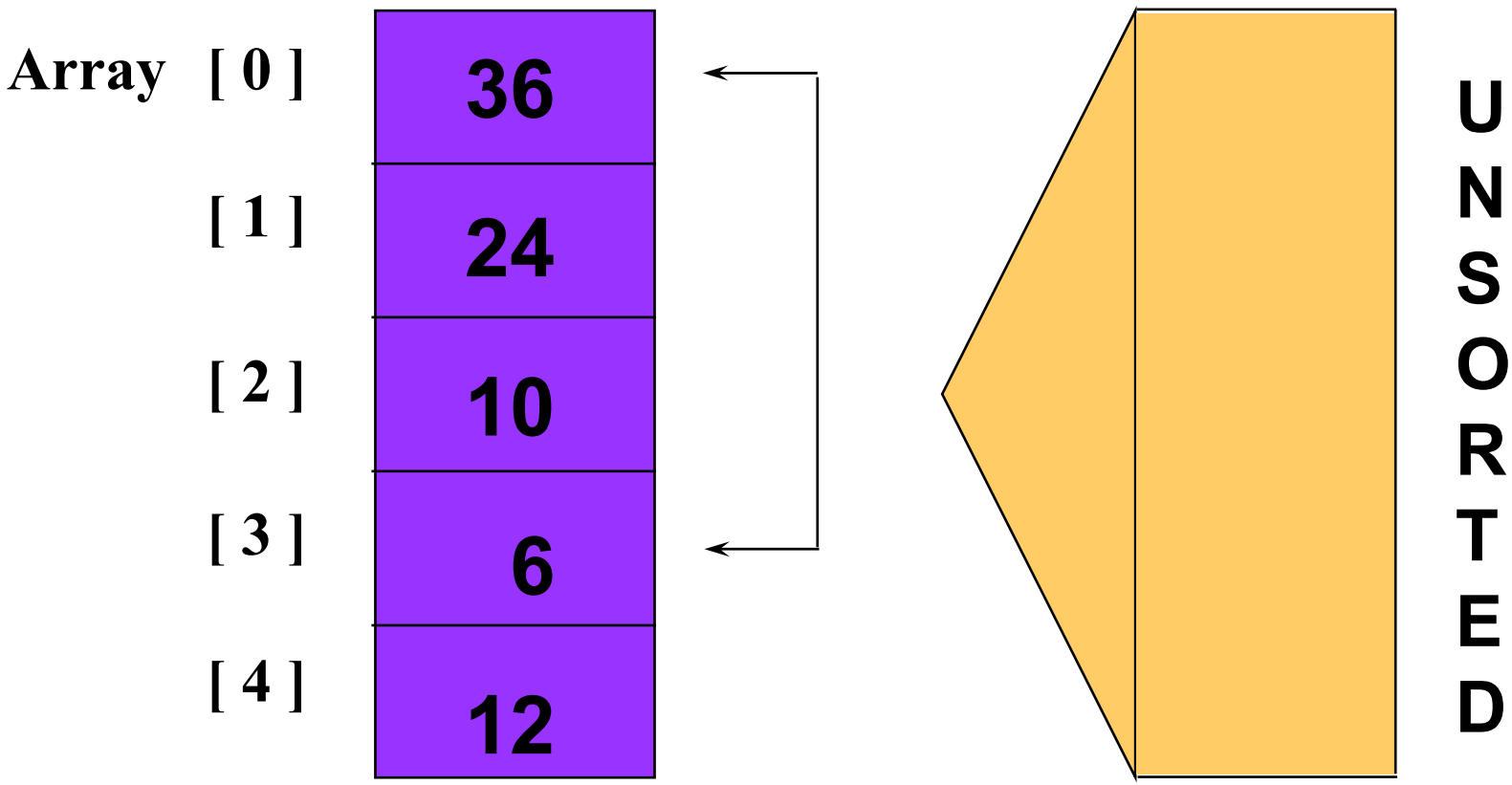
- Là phương pháp đơn giản nhất

Sắp xếp lựa chọn:

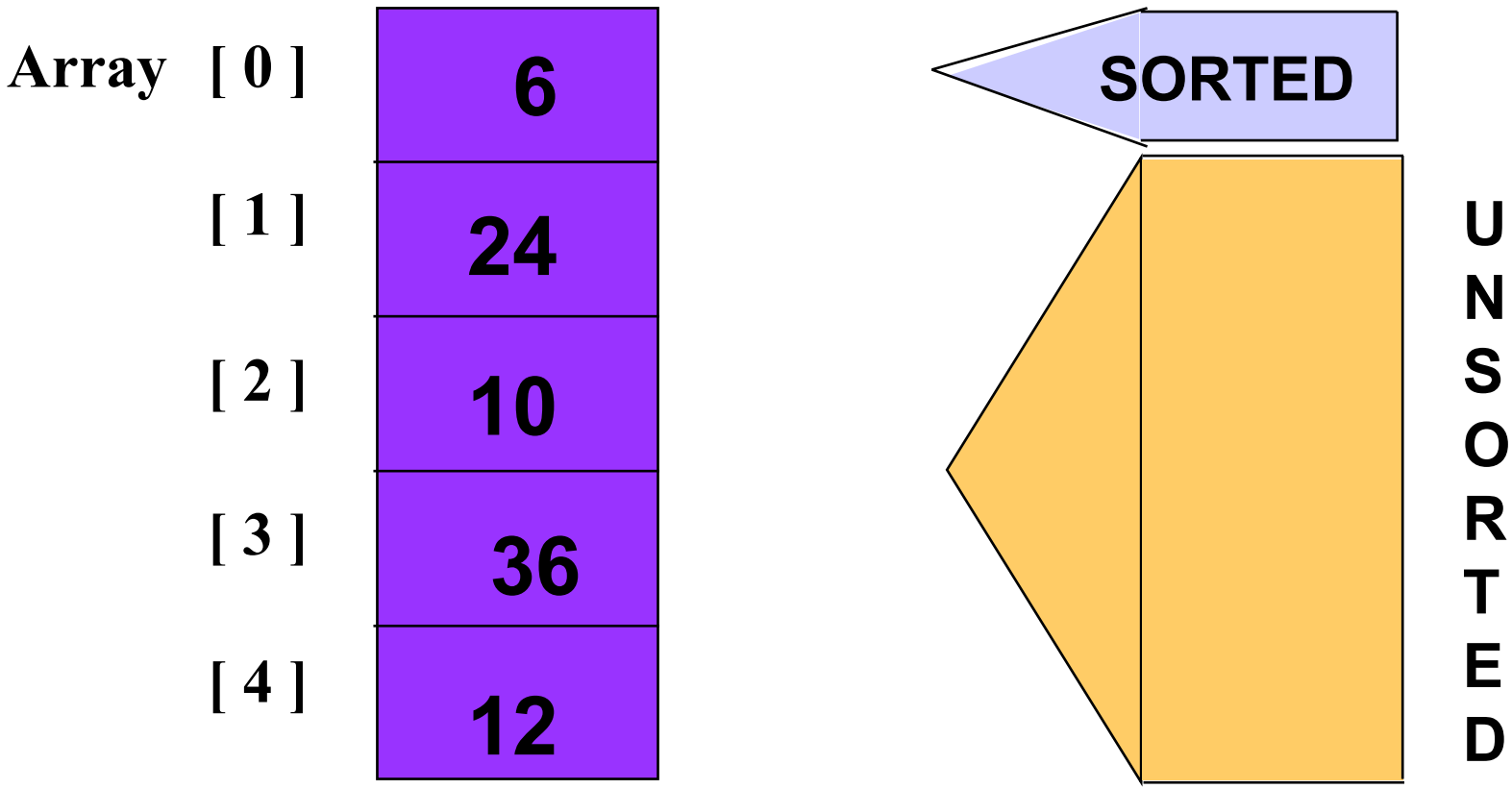
- Tìm phần tử có giá trị nhỏ nhất và đổi chỗ với phần tử chỉ số 0 (phần tử đầu của mảng).
- Tìm phần tử có giá trị nhỏ nhất trong số các phần tử chỉ số 1 đến chỉ số n-1 và đổi chỗ với phần tử chỉ số 1.
- Tìm phần tử có giá trị nhỏ nhất trong số các phần tử chỉ số 2 đến chỉ số n-1 và đổi chỗ với phần tử chỉ số 2.

…

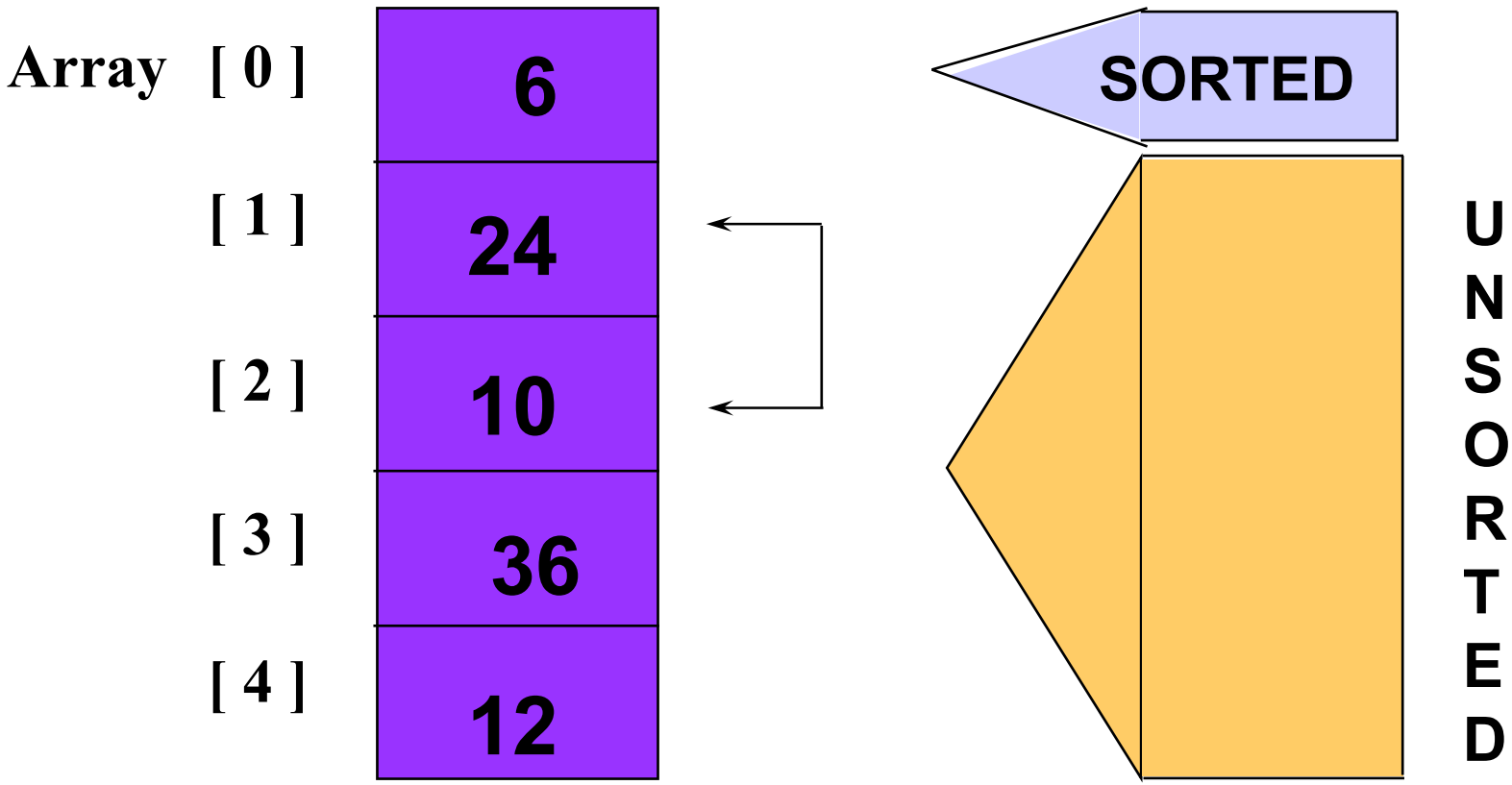
Selection Sort: Lượt 1



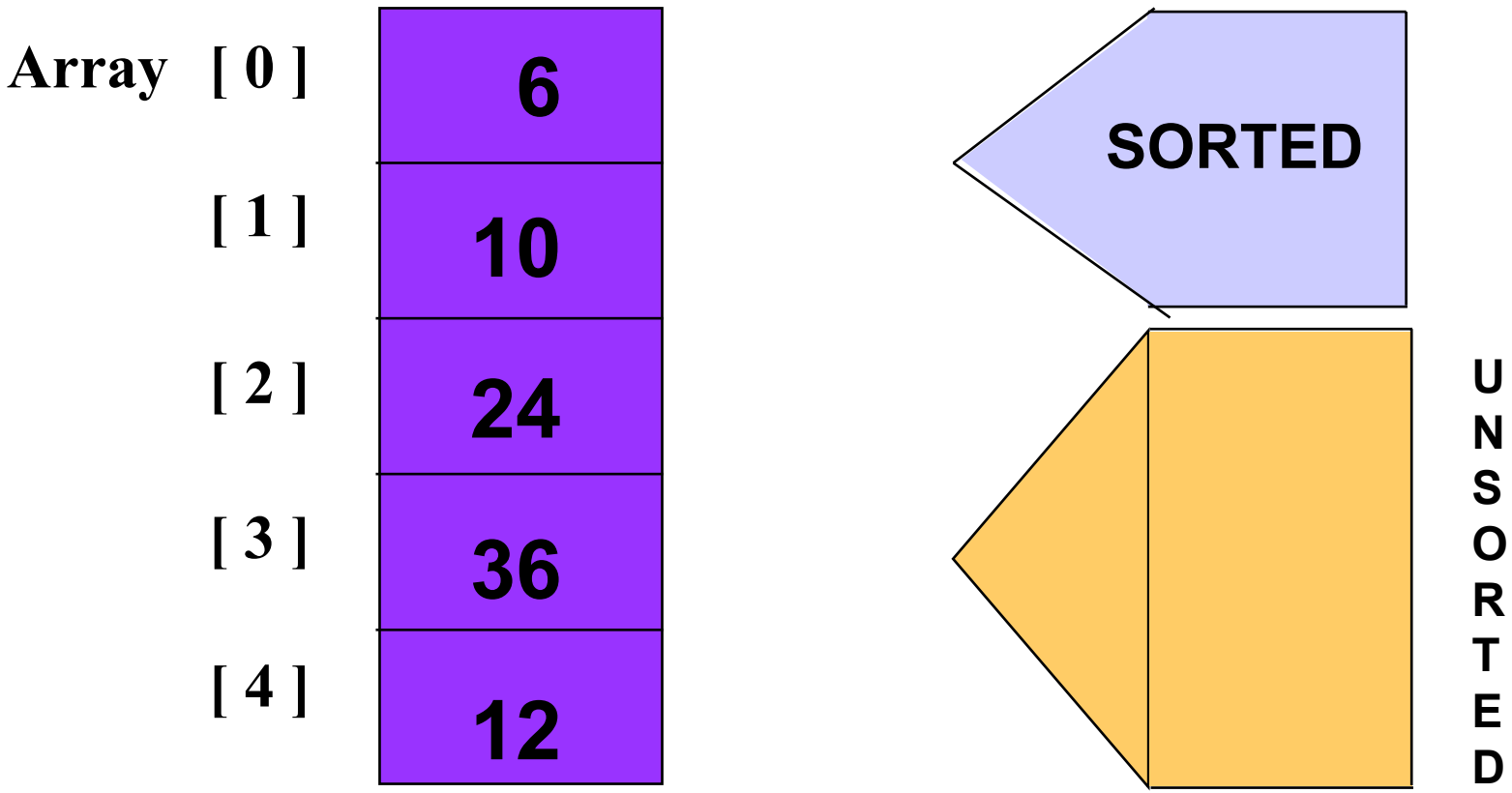
Selection Sort: Kết thúc lượt 1



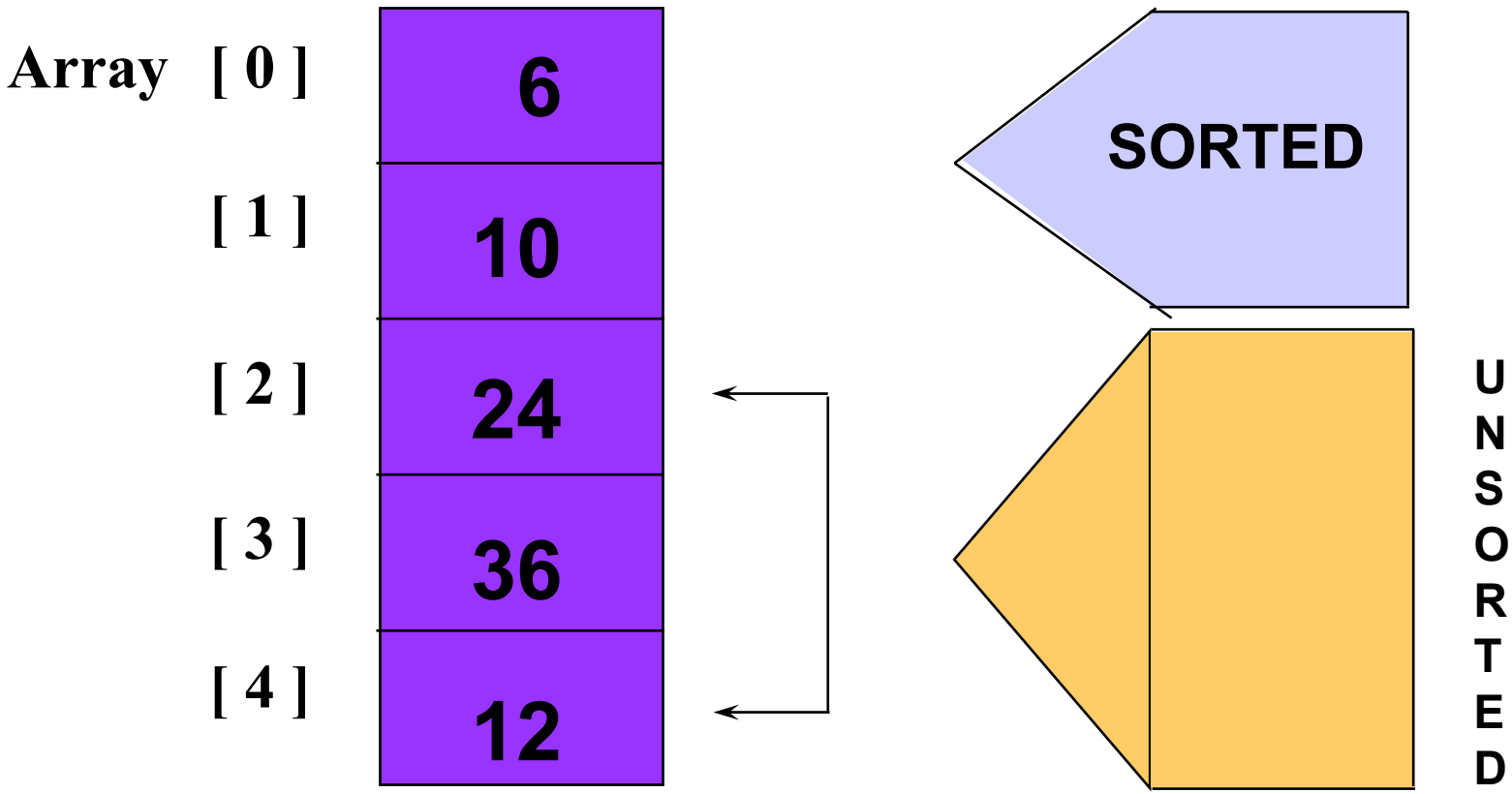
Selection Sort: Lượt 2



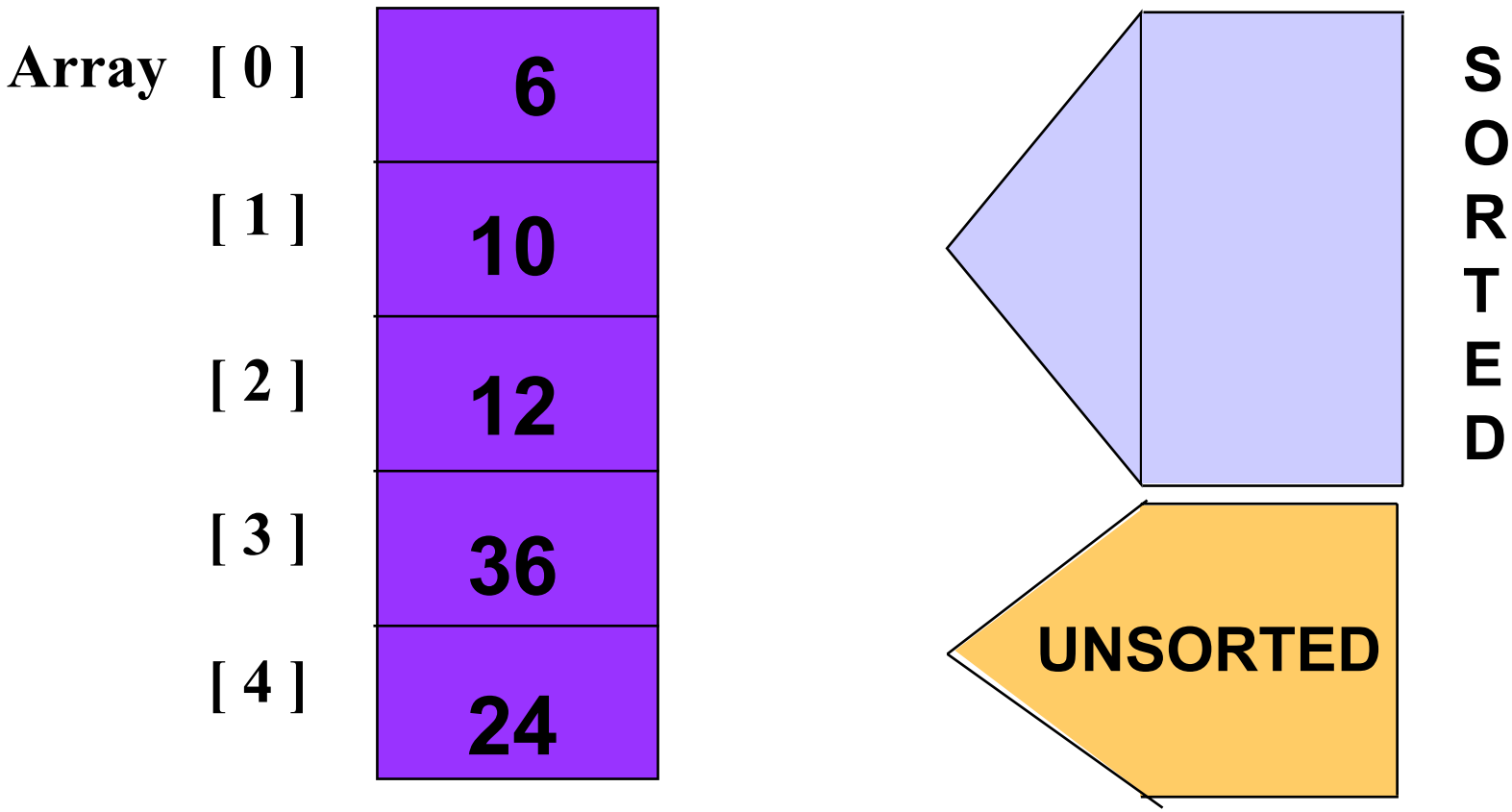
Selection Sort: Kết thúc lượt 2



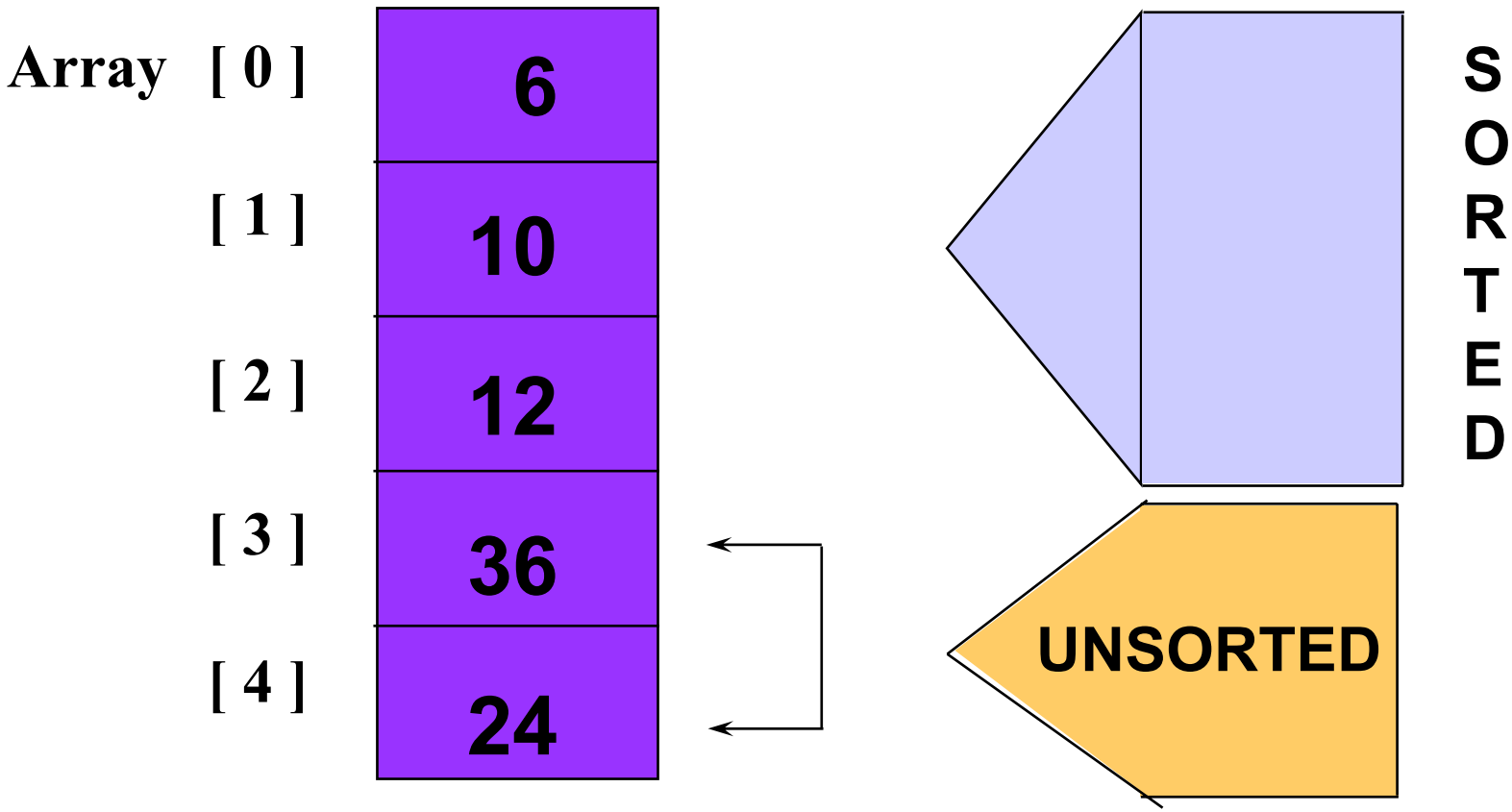
Selection Sort: Lượt 3



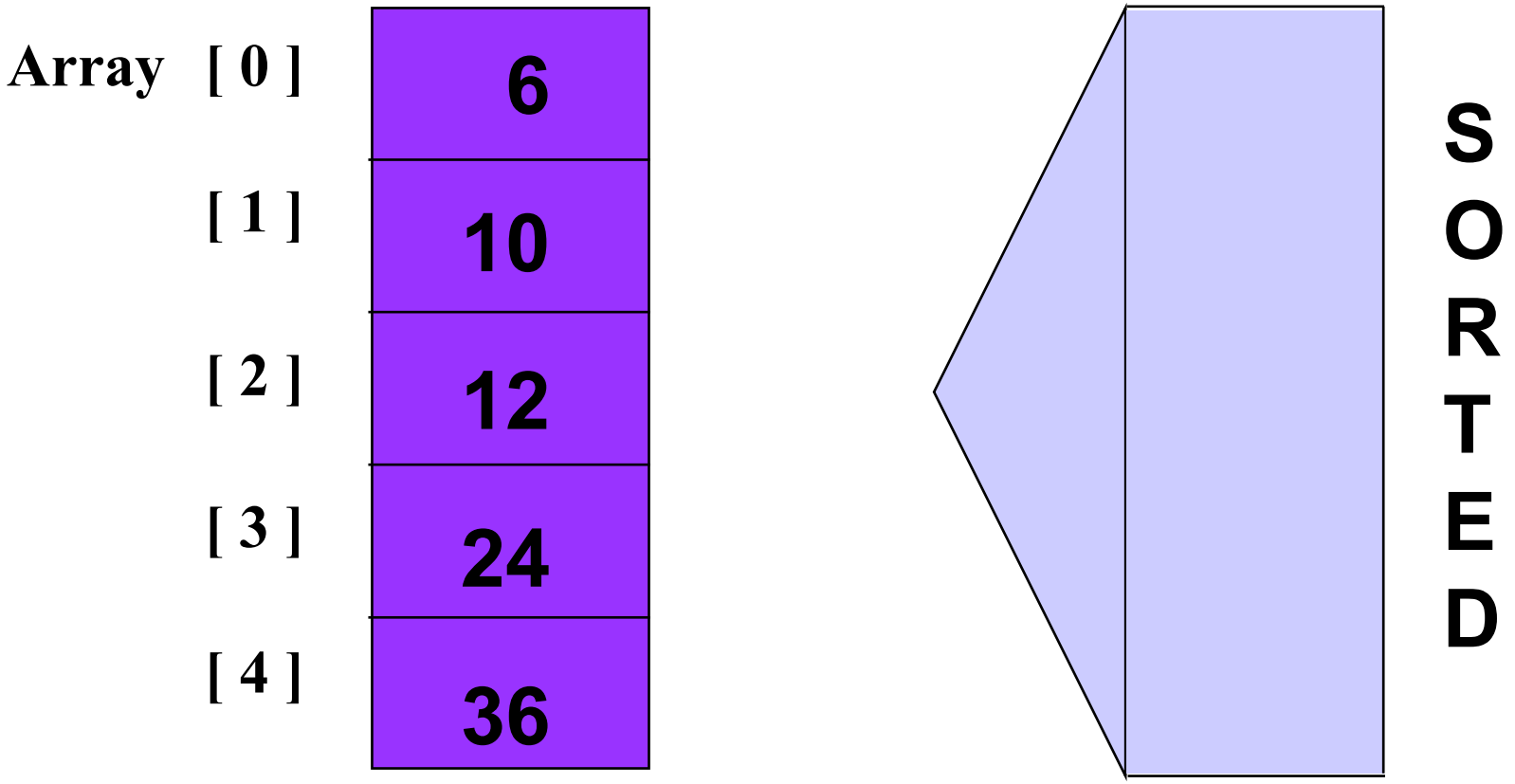
Selection Sort: Kết thúc lượt 3



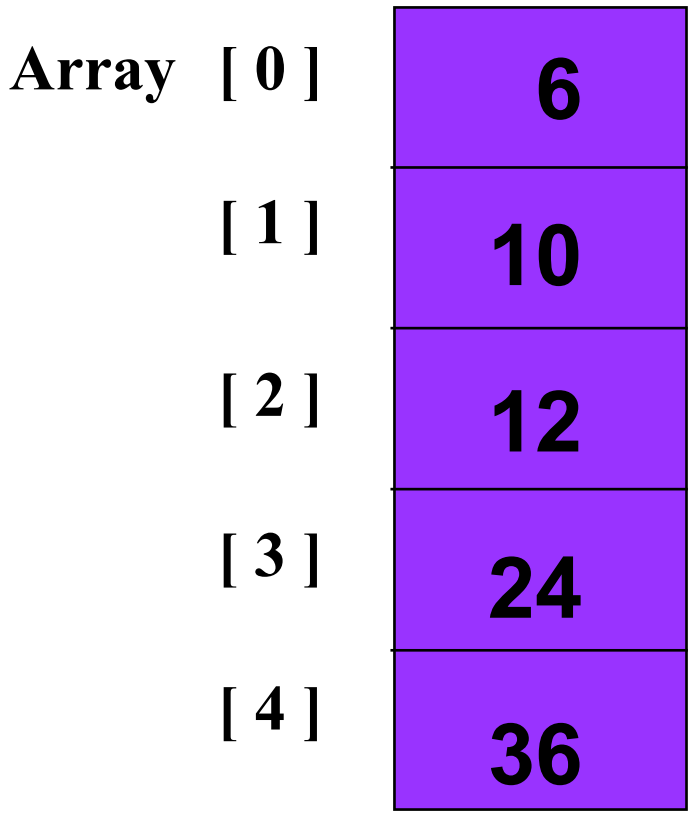
Selection Sort: Lượt 4



Selection Sort: Kết thúc lượt 4



Selection Sort: Số phép so sánh?



4 so sánh cho phần tử [0]

3 so sánh cho phần tử [1]

2 so sánh cho phần tử [2]

1 so sánh cho phần tử [3]

= 4 + 3 + 2 + 1

Độ phức tạp về thời gian

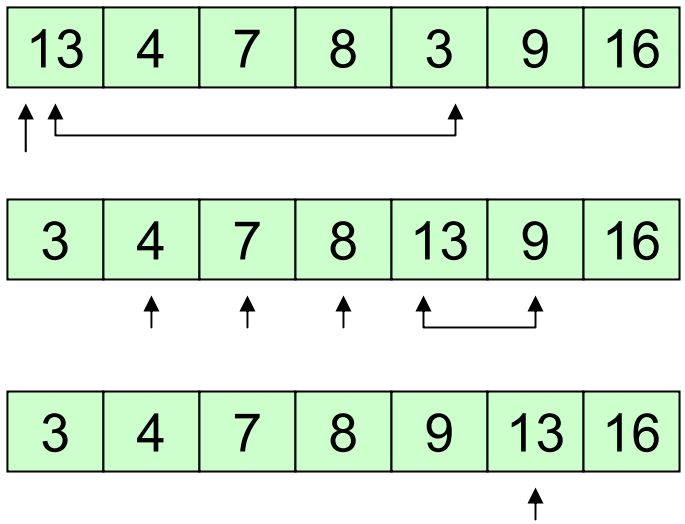
- **Số phép so sánh khi mảng có N phần tử:**

Sum = (N-1) + (N-2) + . . . + 2 + 1

$$Sum = \sum_{i=1}^{N-1} i = \frac{(N-1)N}{2}$$

$O(N^2)$

Ví dụ: Sắp xếp lựa chọn



Sắp xếp lựa chọn là thuật toán *sắp xếp ngay tại chỗ* : không cần sử dụng thêm bộ nhớ.

Xấu nhất: $O(n^2)$

Tốt nhất: $O(n^2)$

Trung bình: $O(n^2)$

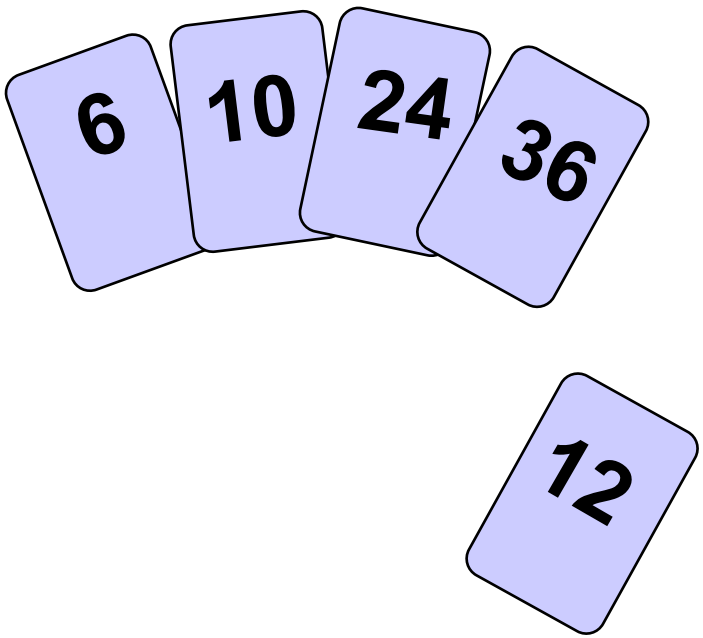
Sắp xếp lựa chọn

```
void  SelectionSort  ( int  A [ ] ,   int  n )

//  Sắp xếp mảng A[0 . . n-1 ] theo thứ tự tăng dần
{

    for  ( int  current = 0 ; current < n - 1 ; current++ )

       Swap ( A [ current ] , A [ GetMin ( A, current, n-1 ) ] ) ;

}
```

```
int  GetMin  ( int  A [ ] ,   int  start ,  int end )
// Tìm chỉ số của phần tử có giá trị nhỏ nhất trong mảng
//          A [start]  . . A [end].
{
    int  indexOfMin = start ;
    for  ( int  i  = start + 1 ;  i <= end ; i++ )
         if  ( A [ i ] < A [ indexOfMin ] )
              indexOfMin = i ;
    return   indexOfMin;
}
```

Sắp xếp thêm dần
Insertion Sort

Sắp xếp các quân bài?

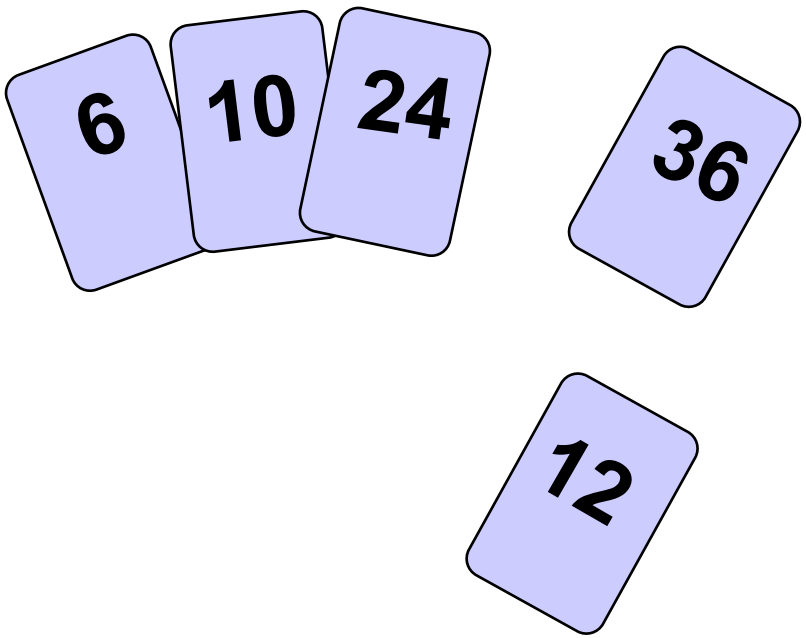


Mỗi lần "chèn" thêm một quân bài vào tay cầm bài, các quân bài trên tay đã được sắp xếp.

Để chèn 12, cần phải tạo khoảng trống cho nó bằng cách dịch chuyển 36 trước và sau đó dịch chuyển 24.

Sắp xếp thêm dần
Insertion Sort

Sắp xếp các quân bài?

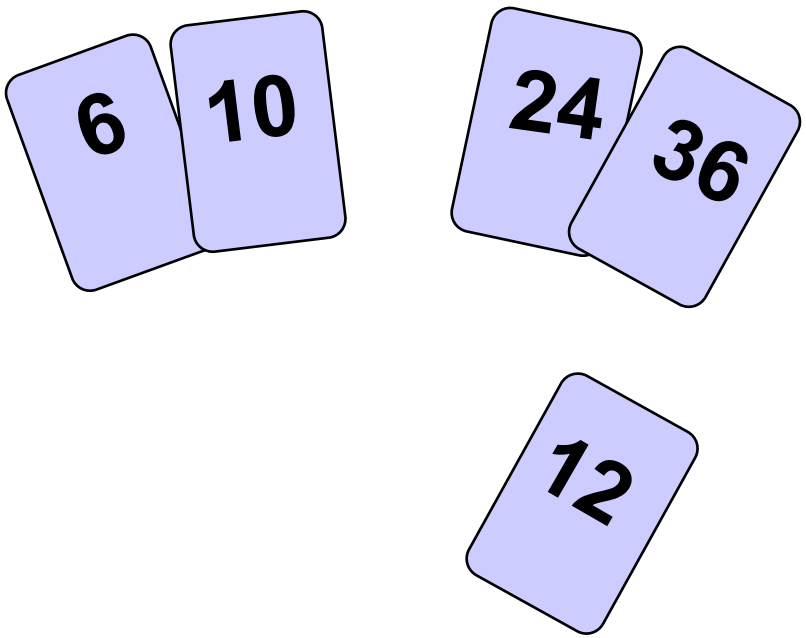


Mỗi lần "chèn" thêm một quân bài vào tay cầm bài, các quân bài trên tay đã được sắp xếp.

Để chèn 12, cần phải tạo khoảng trống cho nó bằng cách dịch chuyển 36 trước và sau đó dịch chuyển 24.

Sắp xếp thêm dần
Insertion Sort

Sắp xếp các quân bài?

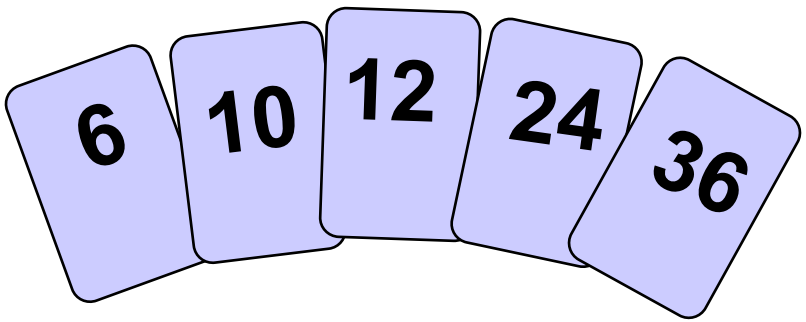


Mỗi lần "chèn" thêm một quân bài vào tay cầm bài, các quân bài trên tay đã được sắp xếp.

Để chèn 12, cần phải tạo khoảng trống cho nó bằng cách dịch chuyển 36 trước và sau đó dịch chuyển 24.

Sắp xếp thêm dần
Insertion Sort

Sắp xếp các quân bài?



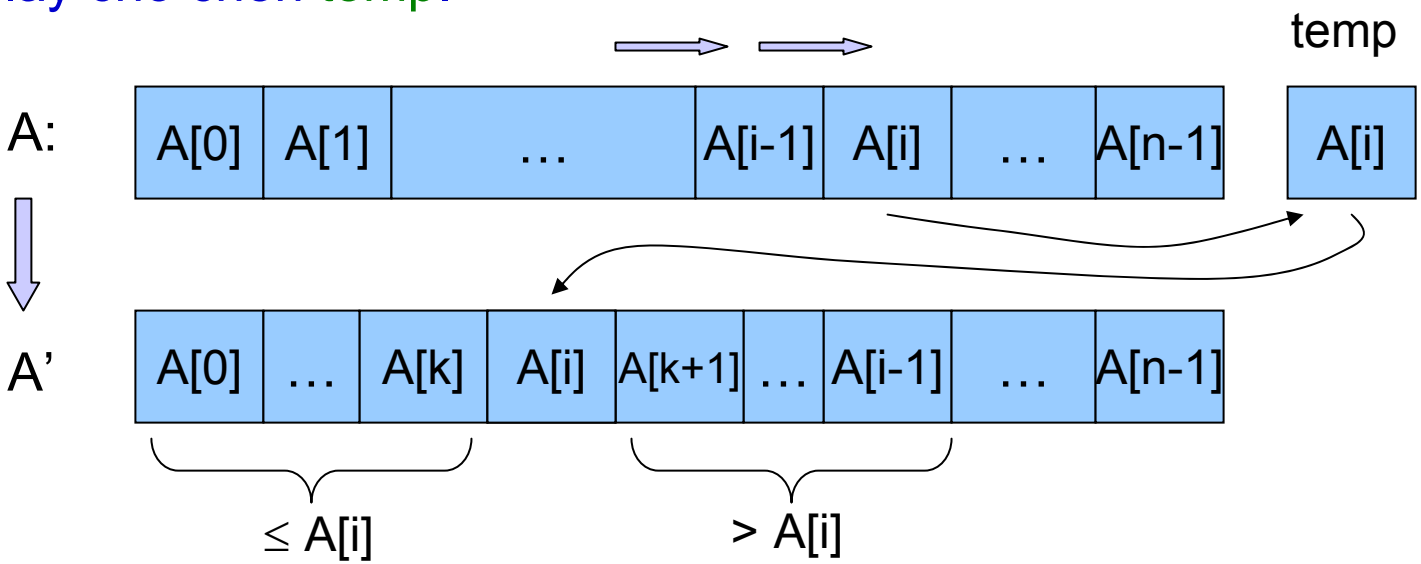
Mỗi lần "chèn" thêm một quân bài vào tay cầm bài, các quân bài trên tay đã được sắp xếp.

Để chèn 12, cần phải tạo khoảng trống cho nó bằng cách dịch chuyển 36 trước và sau đó dịch chuyển 24.

Sắp xếp thêm dần (Insertion Sort)

Tương tự như cách sắp xếp các quân bài.

- Rút lần lượt từng phần tử A[1], …, A[n-1], đưa vào một biến temp.
- Với mỗi phần tử A[i] được xét, dịch chuyển những phần tử lớn hơn temp trong số các phần tử từ A[0] .. A[i-1] sang bên phải một vị trí để lấy chỗ chèn temp.



Ví dụ


19 7 5 23 13 17
7 19 5 23 13 17
5 7 19 23 13 17
5 7 19 23 13 17
5 7 13 19 23 17
5 7 13 17 19 23

Thuật toán

```
void InsertionSort(int A[ ], int n)
{
      int i, j;
      int temp;

      for (j=1; j < n; j++)  {
            temp = A[j];
            // chèn A[j] vào chuỗi A[0], …, A[j-1]
            i = j-1;
            while (i >=0 && A[i] > temp) {
                  A[i+1] = A[i];
                  i = i -1;
            }
            A[i+1] = temp;
      }
}
```

Từng bước thực hiện

Mảng đầu vào:

34 8 64 51 32 21

j = 1; temp = 8;

34 > temp, dịch chuyển 34 một vị trí (sang vị trí thứ 2).

Đạt đến đầu danh sách. Do đó, vị trí thứ nhất = temp

Sau bước lặp đầu tiên: 8 34 64 51 32 21

(2 phần tử đầu đã được sắp xếp)

j = 2; temp = 64;

34 < 64, không cần dịch chuyển và thiết lập vị trí thứ 3 = 64

Sau bước lặp 2: 8 34 64 51 32 21

(3 phần tử đầu đã được sắp xếp)

j = 3; temp = 51;

51 < 64, dịch chuyển 64, ta có: 8 34 64 64 32 21,

34 < 51, dừng lại, thiết lập vị trí thứ 3 = temp,

Sau bước lặp thứ 3: 8 34 51 64 32 21

(4 phần tử đầu đã được sắp xếp)

j = 4; temp = 32,

32 < 64, ta có: 8 34 51 64 64 21,

32 < 51, ta có: 8 34 51 51 64 21,

32 < 34, ta có: 8 34 34 51 64 21,

32 > 8, dừng tại vị trí thứ 1, thiết lập vị trí thứ 2 = 32,

Sau bước lặp thứ 4: 8 32 34 51 64 21

j = 5; temp = 21, . . .

Sau bước lặp thứ 5: 8 21 32 34 51 64

Phân tích InsertionSort

Tốt nhất? Mảng đã được sắp xếp theo thứ tự tăng dần.

$$A[0] \le A[1] \le \ldots \le A[n-1]$$

Mỗi bước lặp chỉ cần một phép so sánh. Không dịch chuyển.

Thời gian tính: $O(n)$

Xấu nhất? Mảng đã được sắp xếp theo thứ tự giảm dần.

$$A[0] > A[1] > \ldots > A[n-1]$$

Bước lặp thứ j cần dịch chuyển j lần.
1+2+ … + n-1 = n(n-1)/2 số lần dịch chuyển.

Thời gian tính: $O(n^2)$

Phân tích InsertionSort

```
void InsertionSort(int A[ ], int n) {
   int i, j;
   int temp;

  for (j=1; j < n; j++)  {
      temp = A[j];
      // chèn A[j] vào chuỗi A[0], …, A[j-1]
      i = j-1;
      while (i >=0 && A[i] > temp) {
            A[i+1] = A[i];
            i = i -1;
      }
      A[i+1] = temp;
  }
}
```

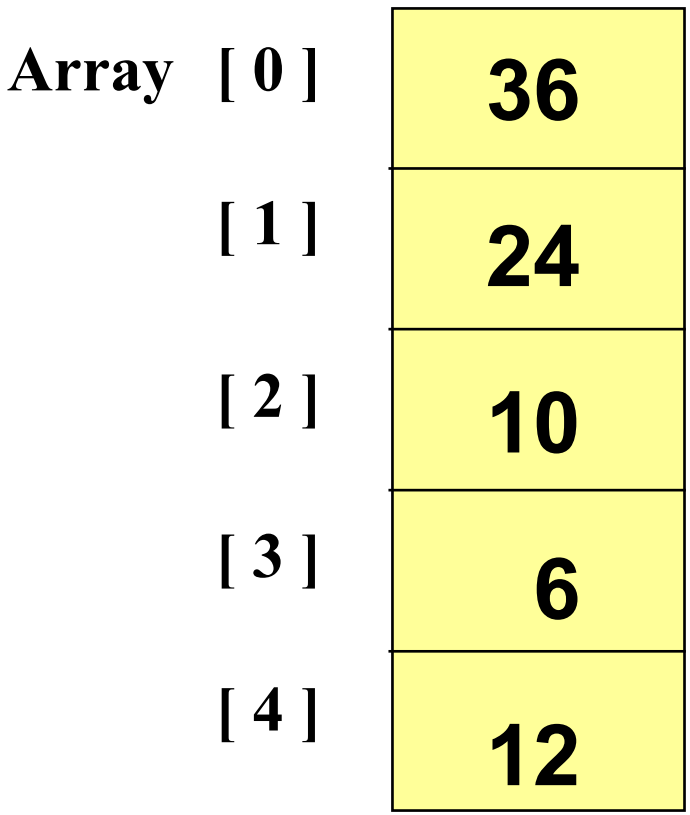
Tại bước lặp thứ j, một nửa số phần tử đứng trước A[j] là lớn hơn A[j] và cần dịch chuyển.

Trung bình, tổng số dịch chuyển:

$1/2 + 2/2 + \ldots + (n-1)/2 = O(n^2)$

Săp xếp thêm dần có thời gian tính trung bình và xấu nhất là xấp xỉ nhau

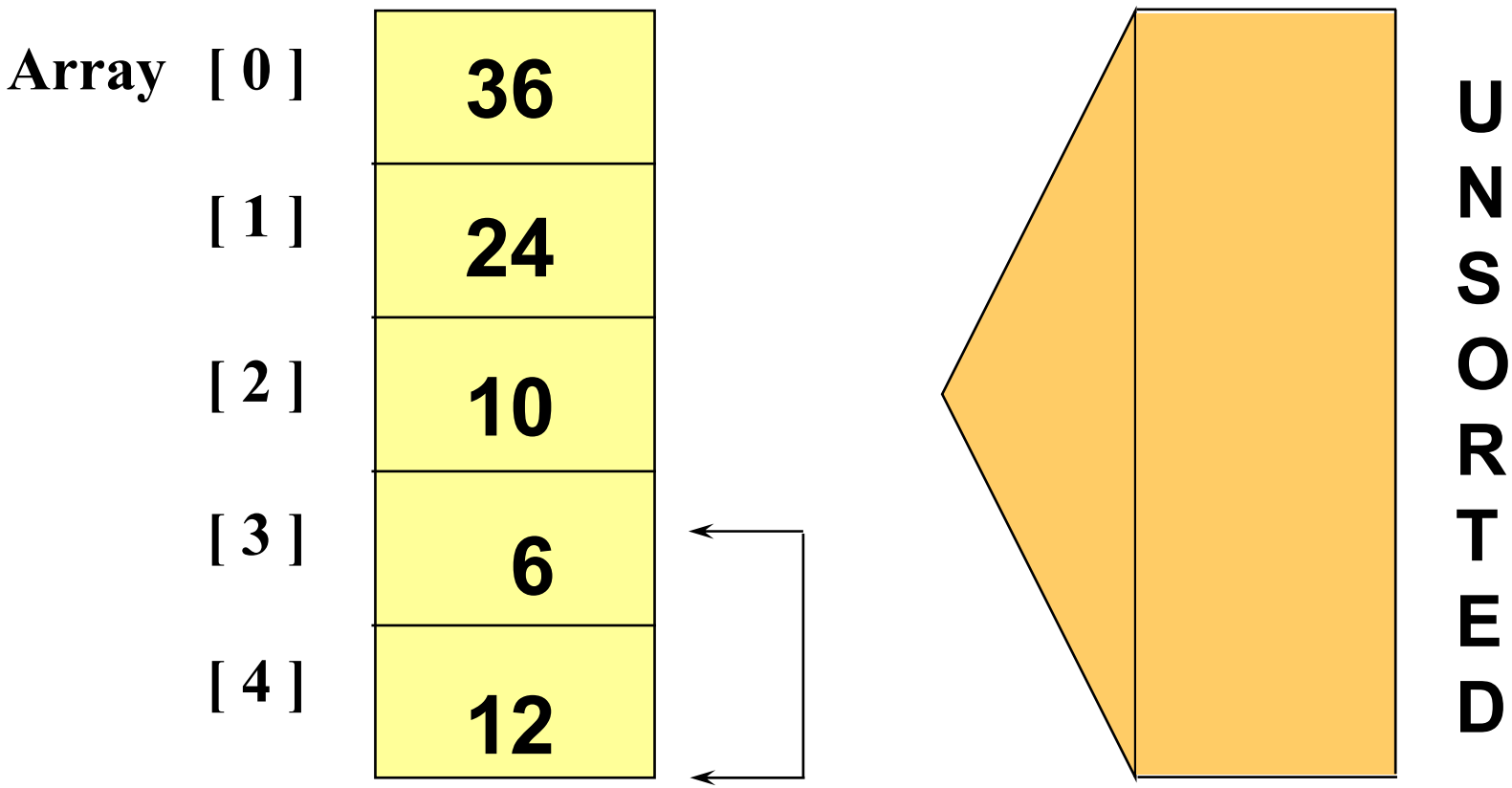
Sắp xếp nổi bọt/đổi chỗ
Bubble Sort/Exchange Sort



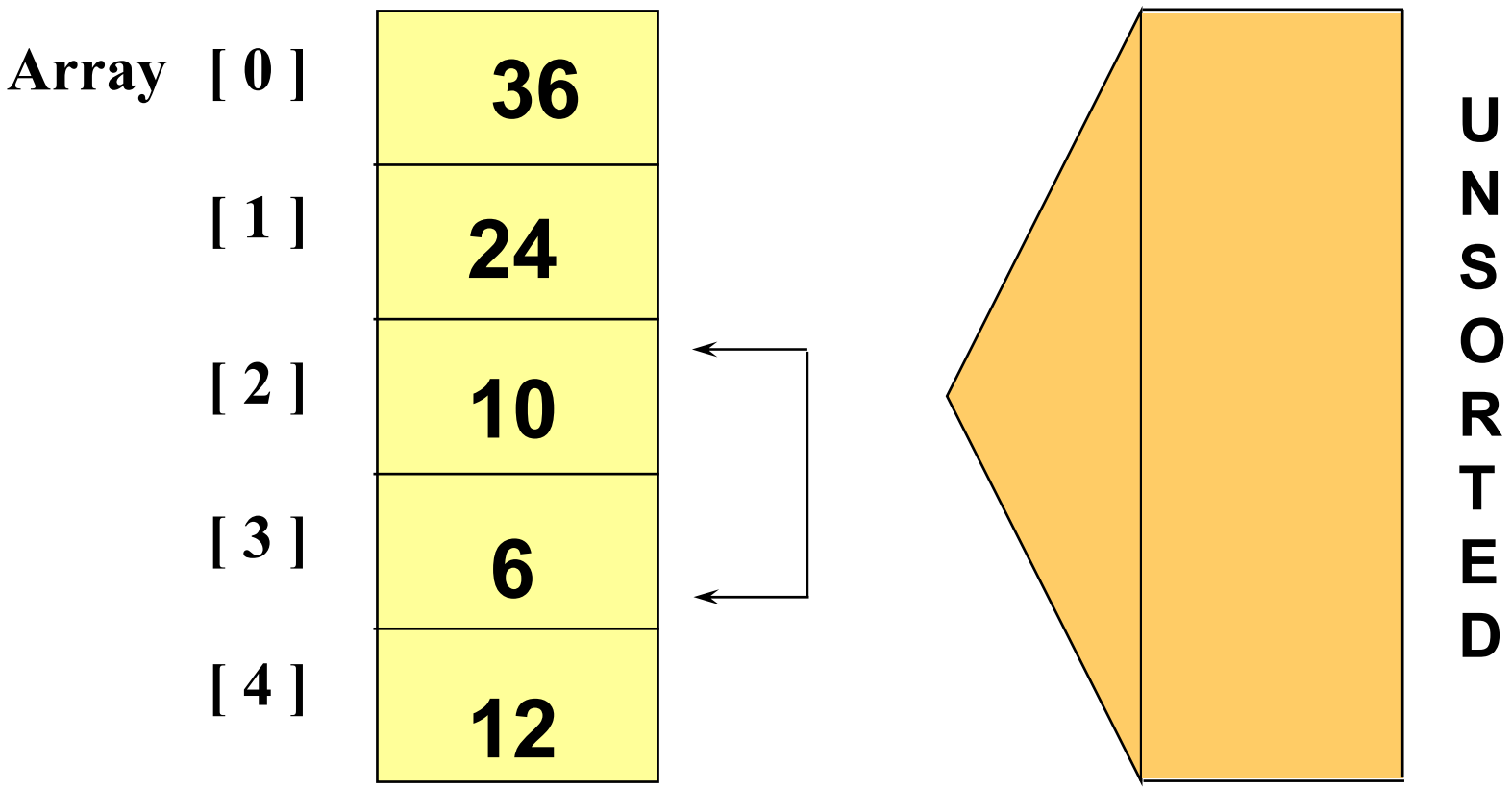
So sánh từng cặp phần tử kế cận, bắt đầu với từ cuối mảng, nếu ngược thứ tự thì đổi chỗ chúng cho nhau.

Qua mỗi lượt, phần tử nhỏ nhất sẽ "nổi lên trên" và chuyển đến đúng vị trí của nó.

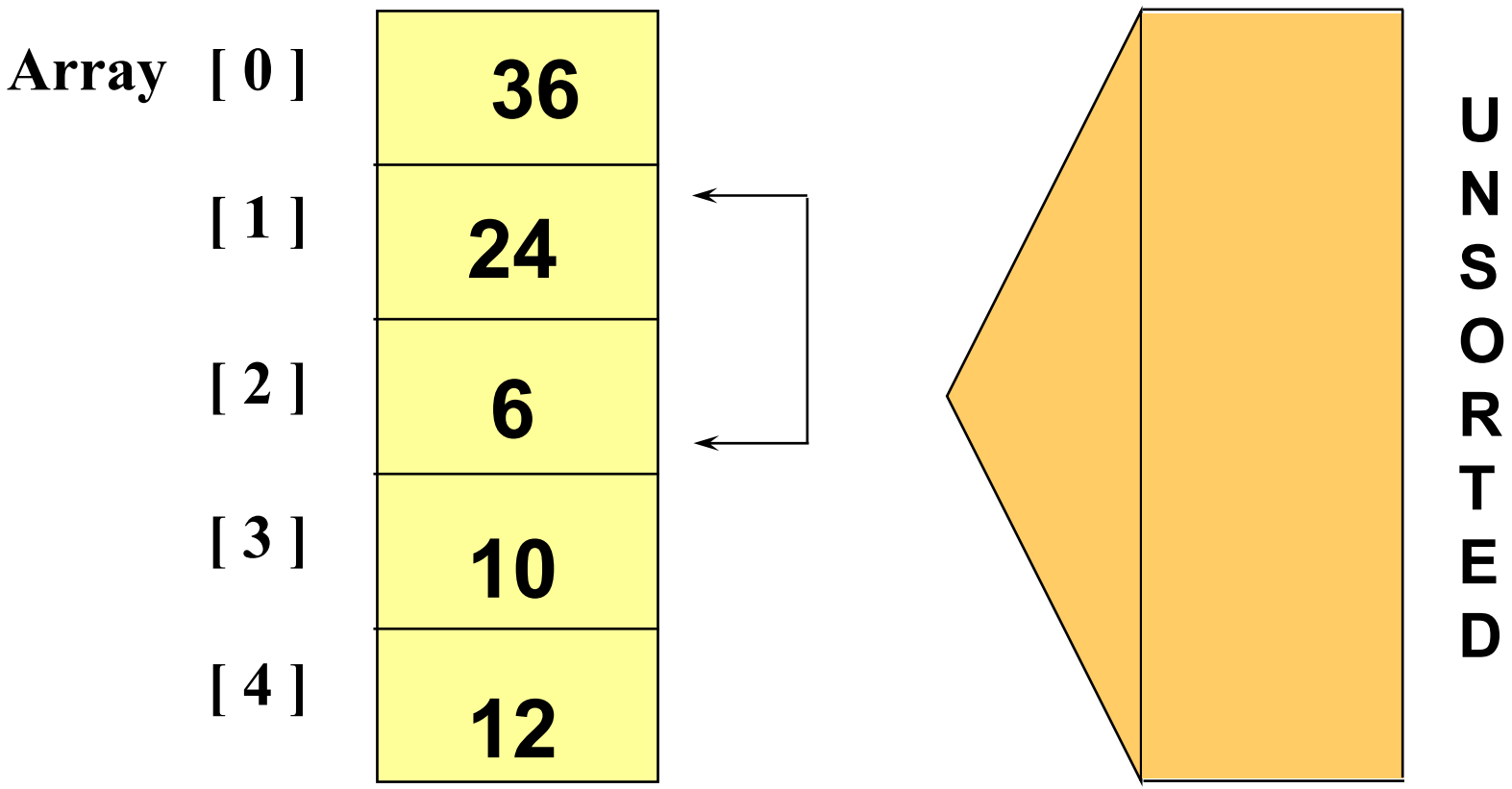
Bubble Sort: Lượt thứ 1



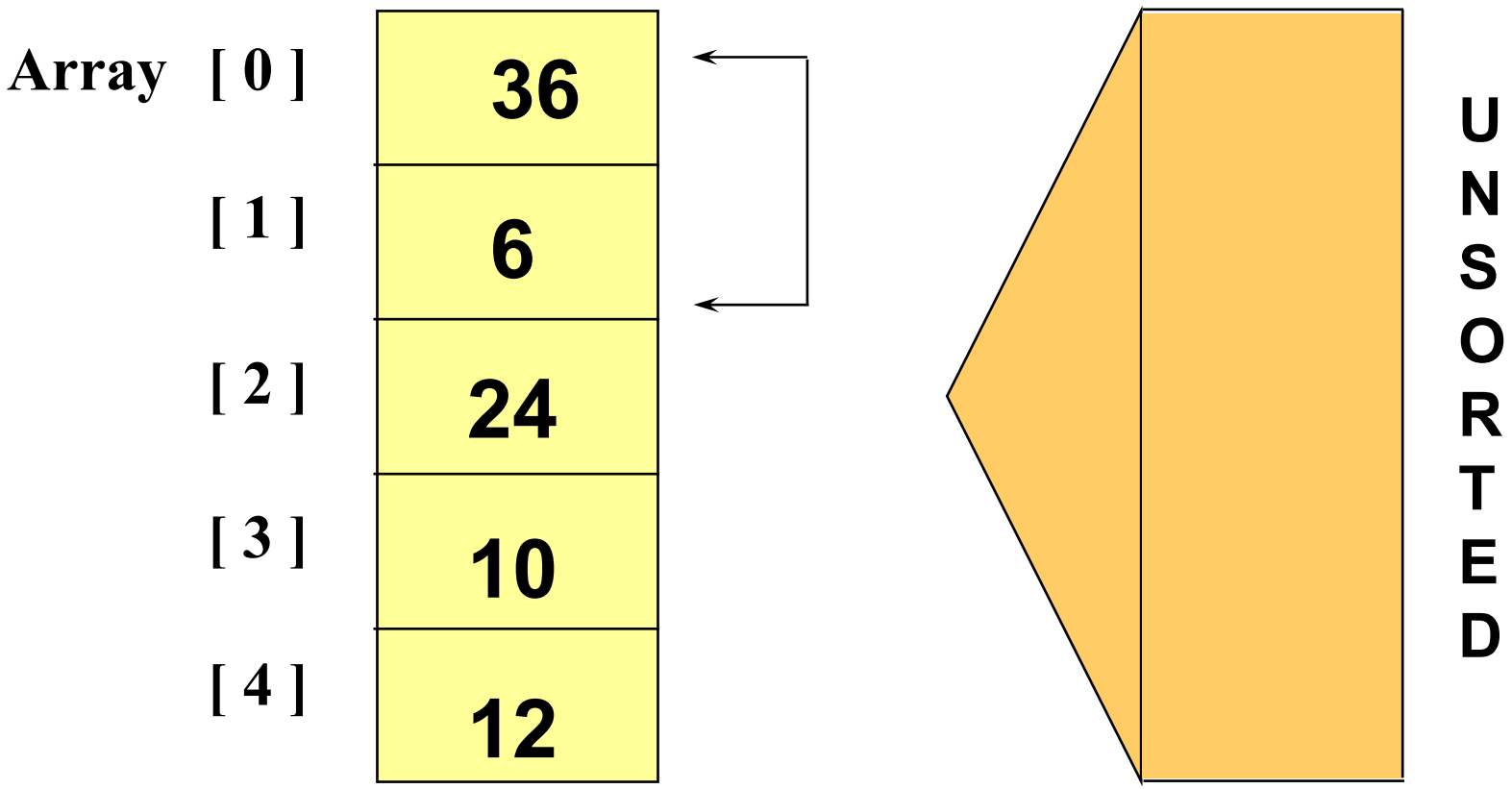
Bubble Sort: Lượt thứ nhất



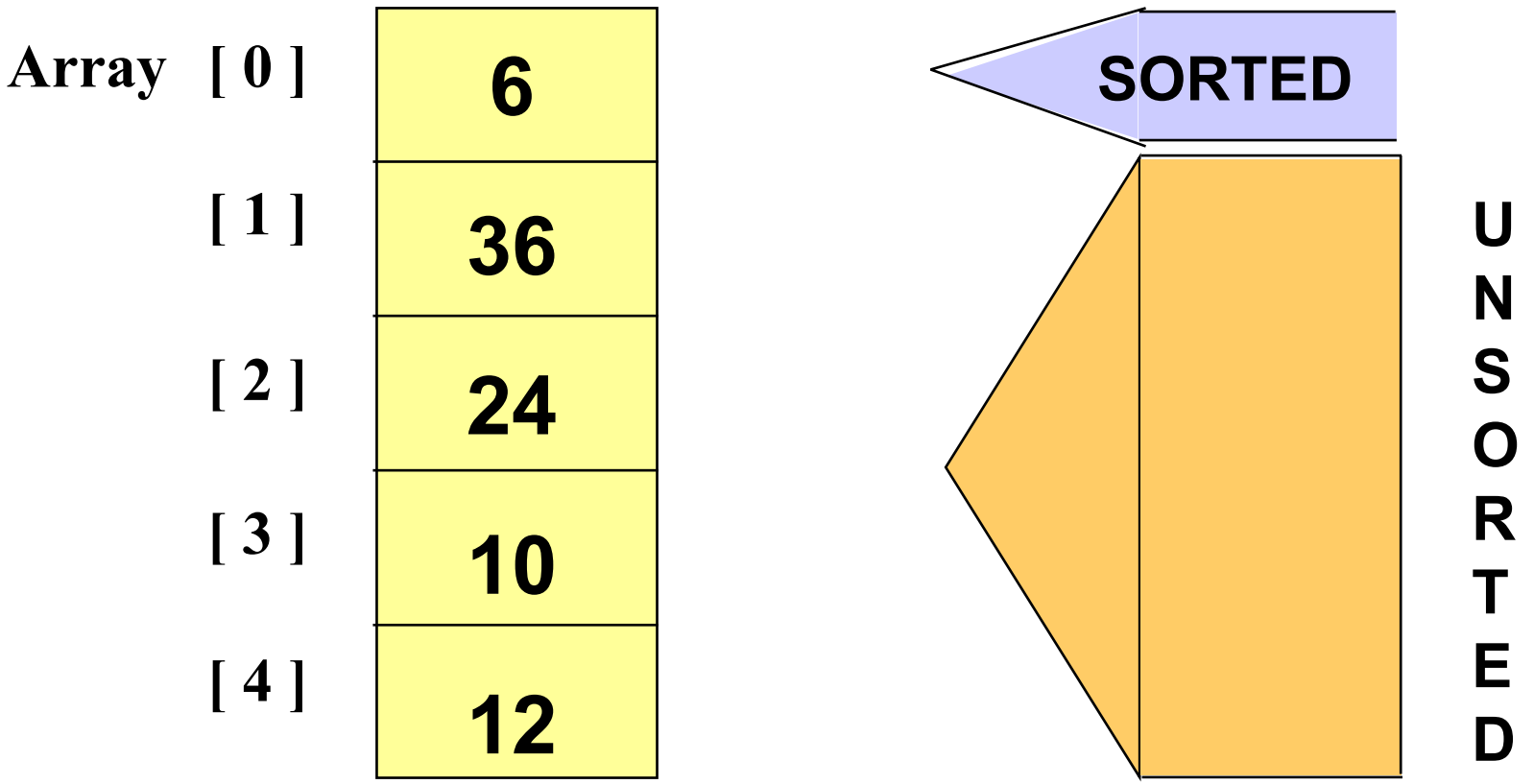
Bubble Sort: Lượt thứ nhất



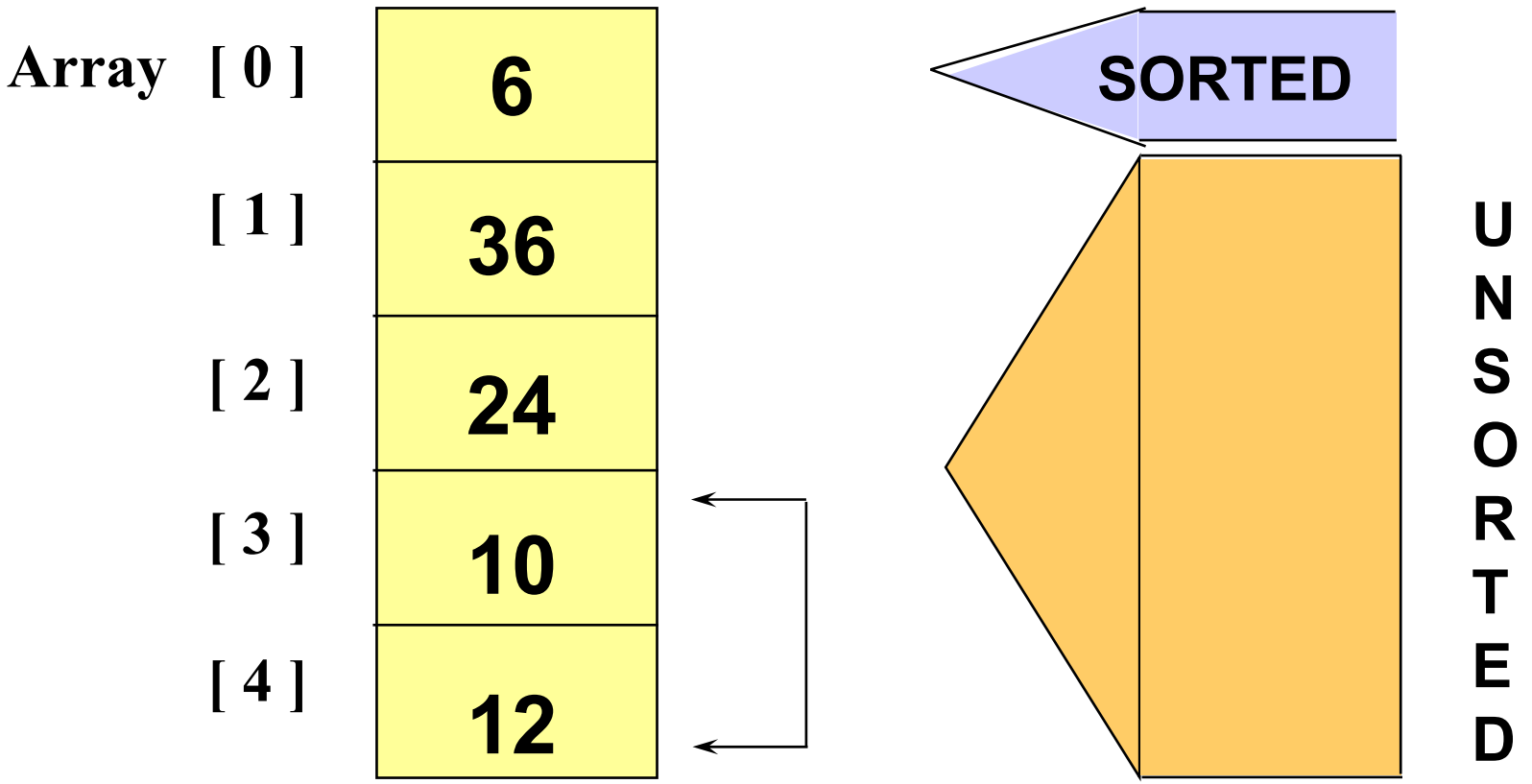
Bubble Sort: Lượt thứ nhất



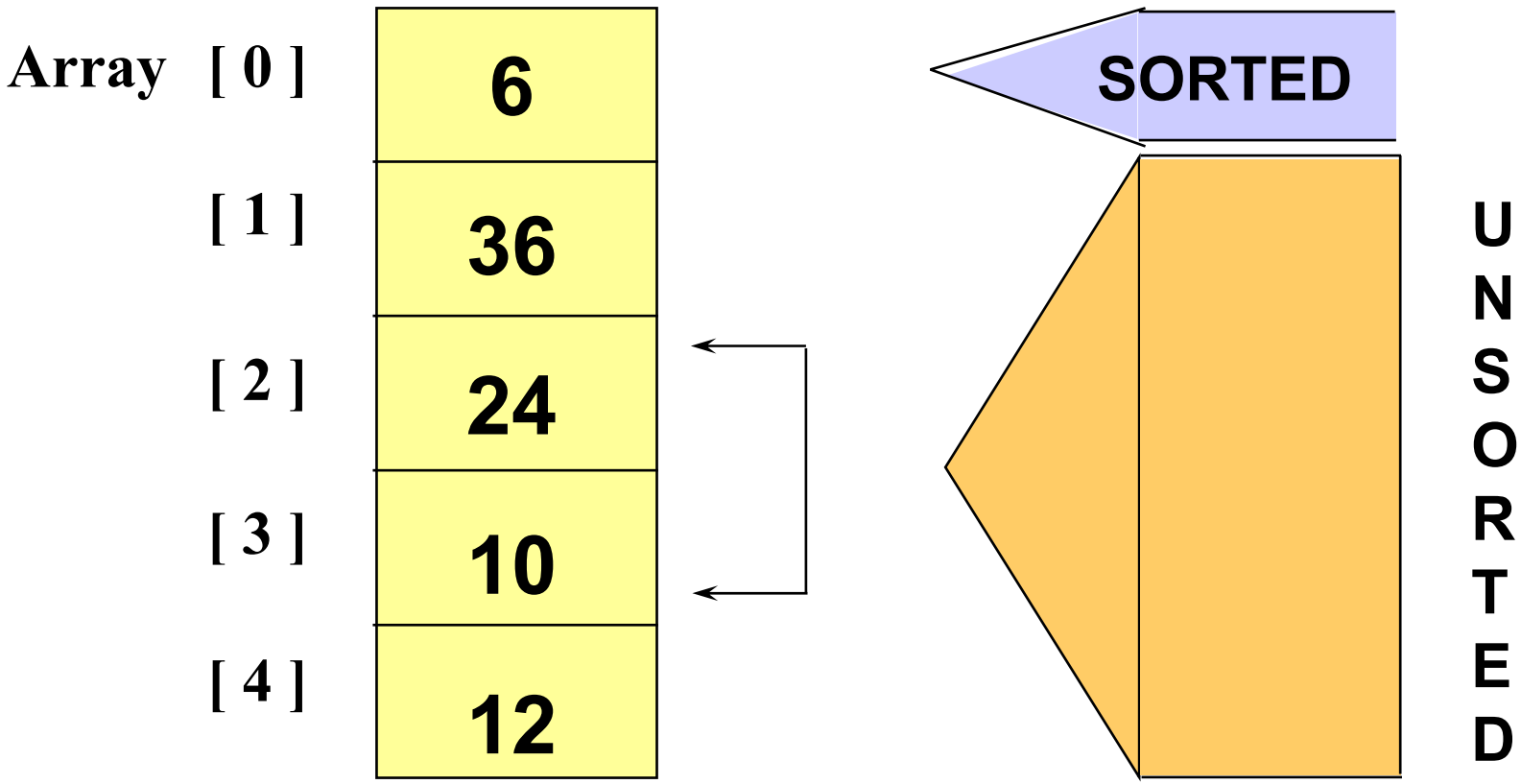
Bubble Sort: Kết thúc lượt 1



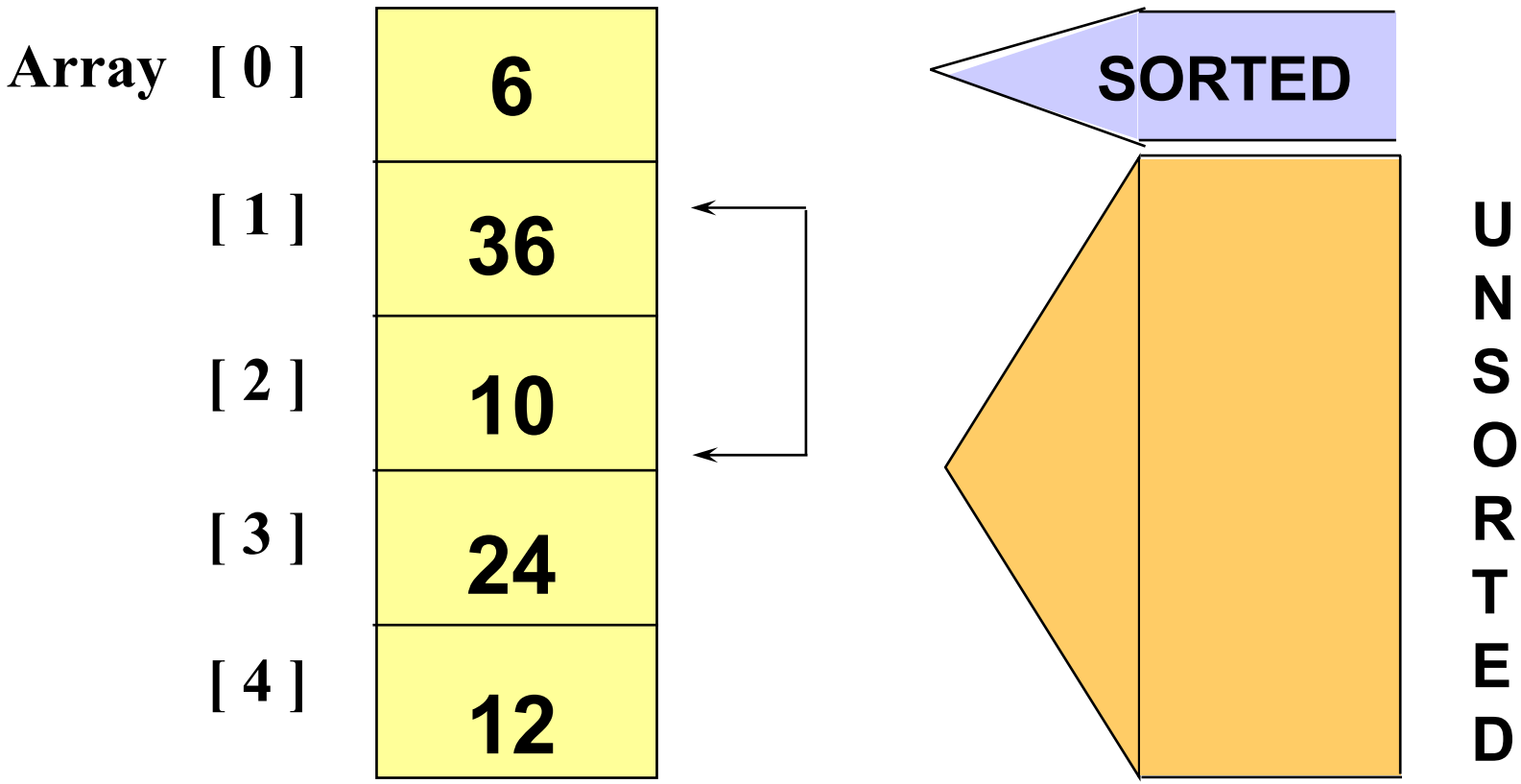
Bubble Sort: Lượt 2



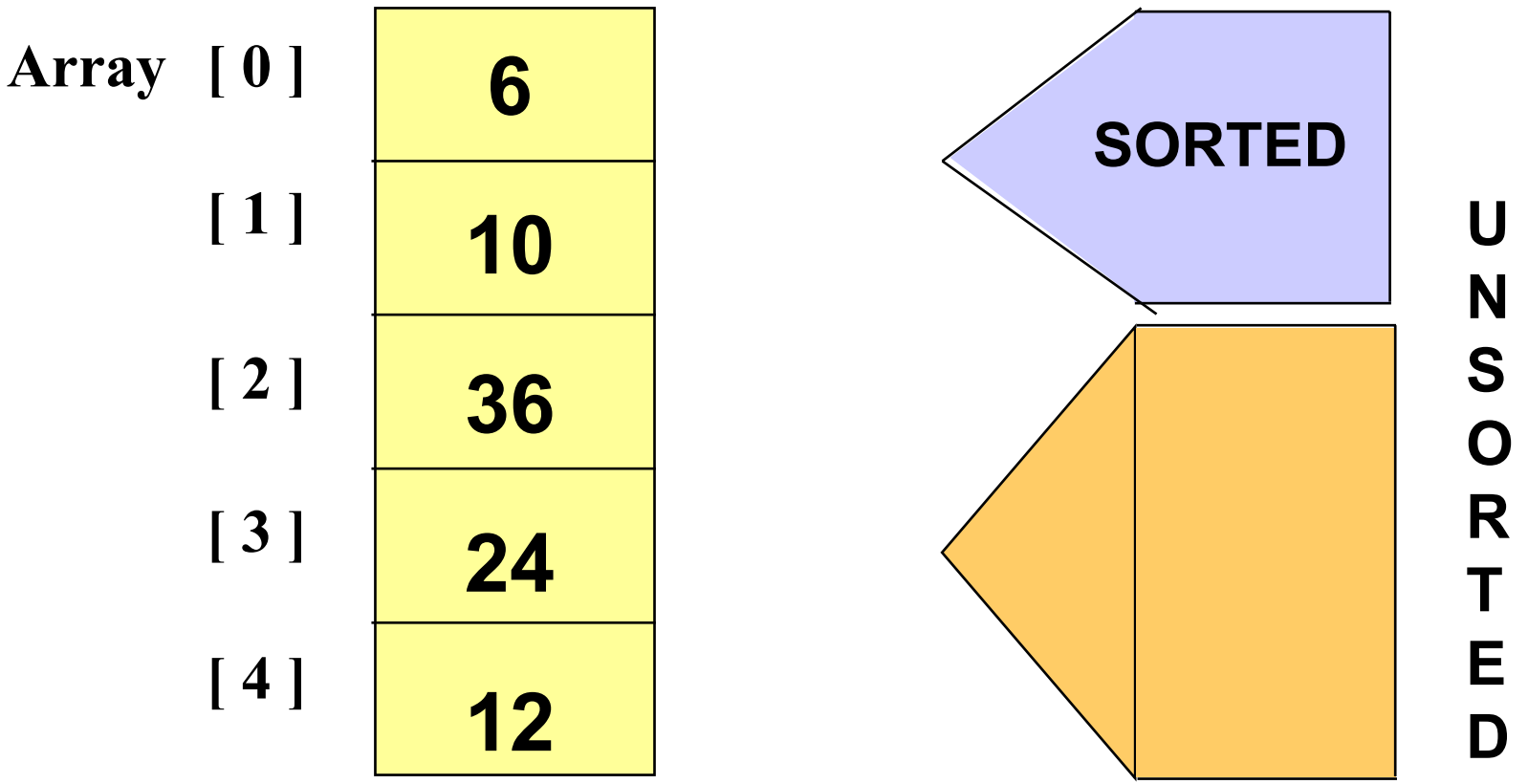
Bubble Sort: Lượt 2



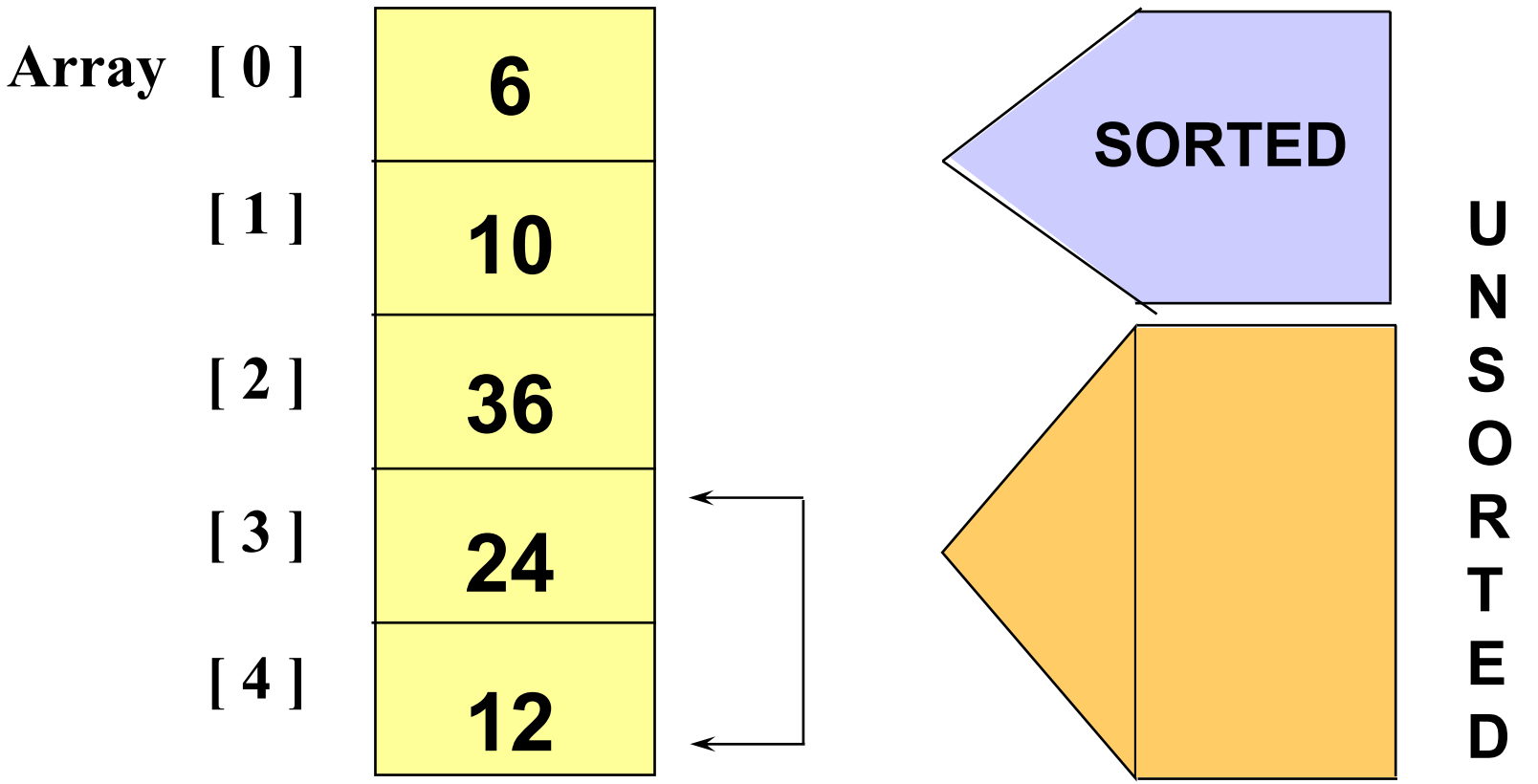
Bubble Sort: Lượt 2



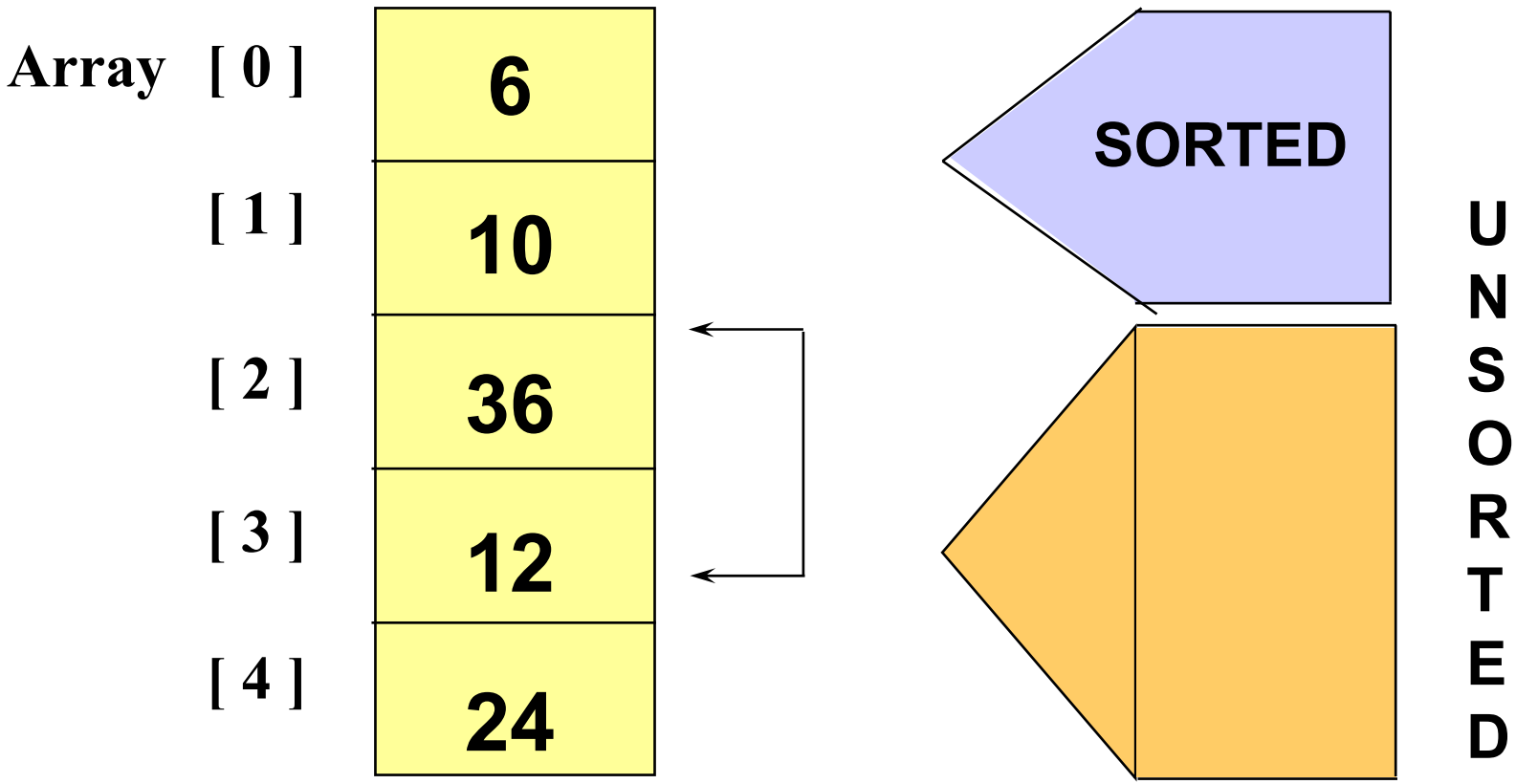
Bubble Sort: Kết thúc lượt 2



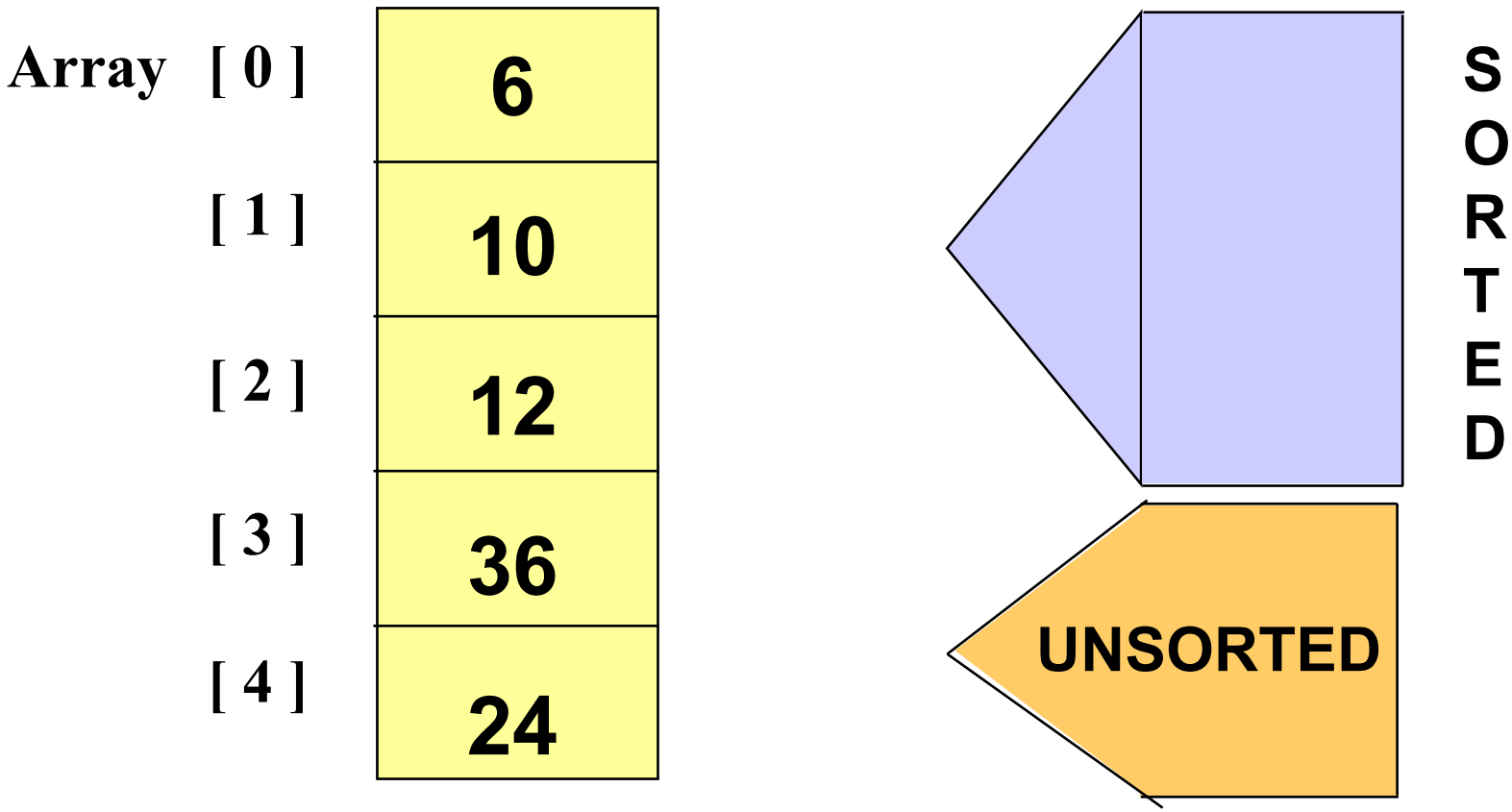
Bubble Sort: Lượt 3



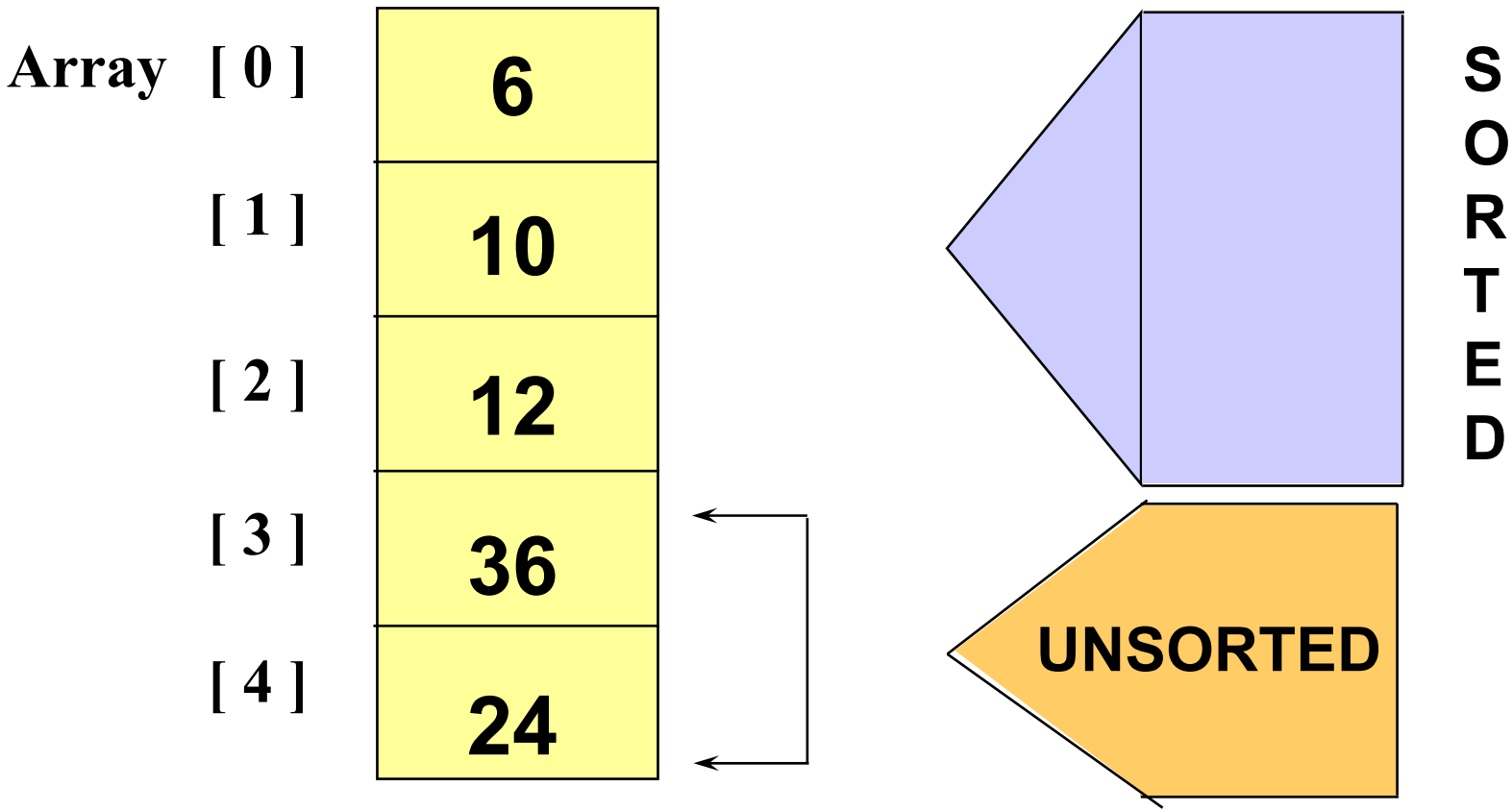
Bubble Sort: Lượt 3



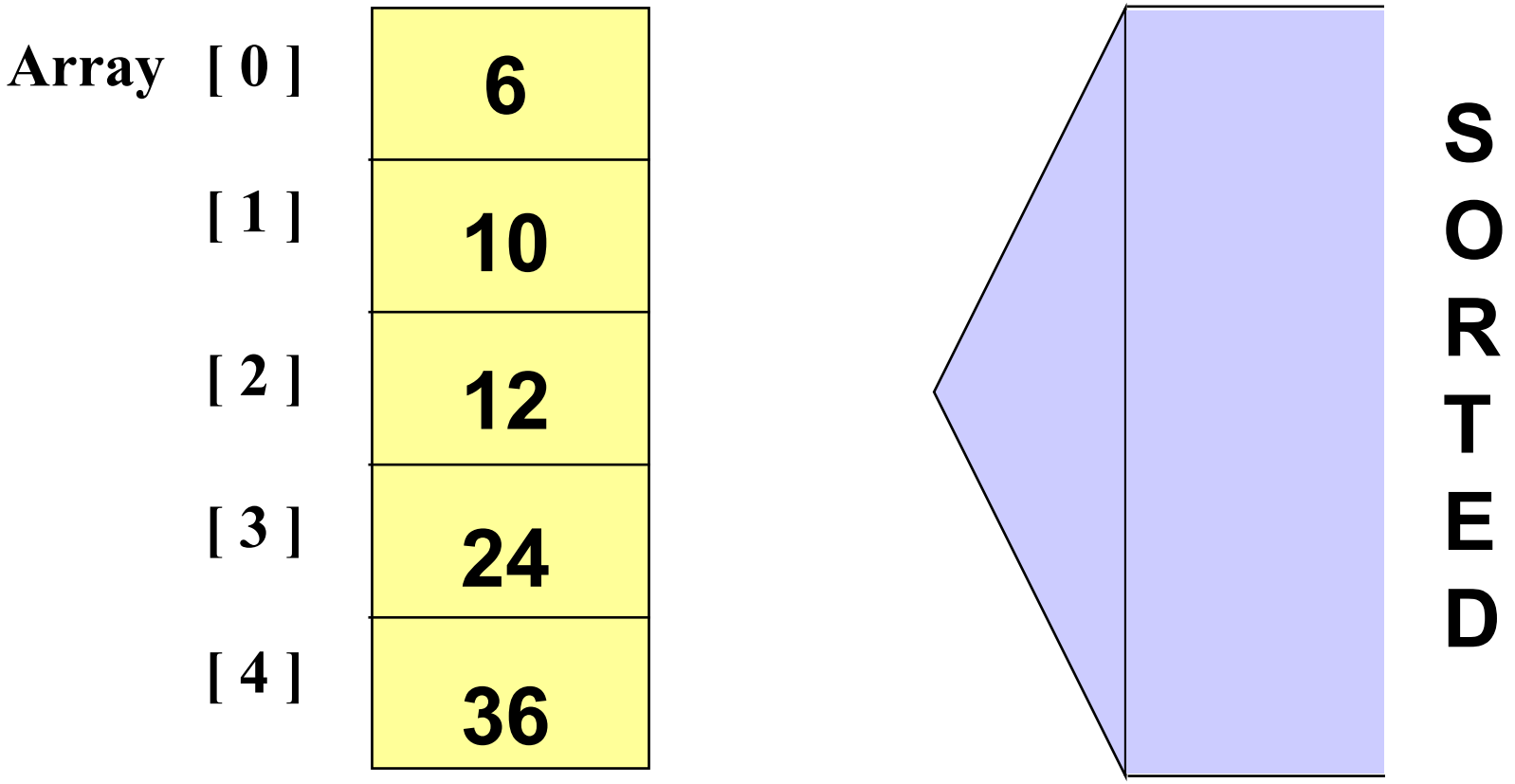
Bubble Sort: Kết thúc lượt 3



Bubble Sort: Lượt 4



Bubble Sort: Kết thúc lượt 4



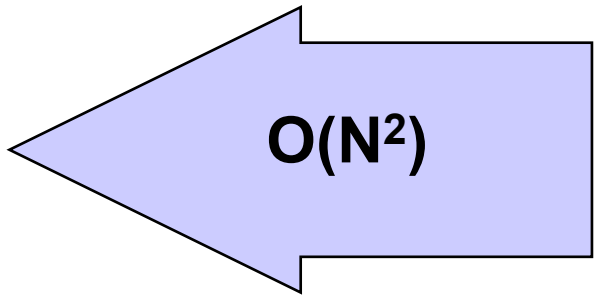
```
void  BubbleSort  (int A [ ] ,   int  n )
/*  Sắp xếp mảng A[n ] theo thứ tự tăng
   dần */
{
   for  ( int i = 0; i < n – 1; i++ )
          BubbleUp ( A , i , n - 1 ) ;
}
```

```
void BubbleUp  (int  A [ ] ,  int start , int end )

// Đổi chỗ các phần tử kề cần ngược thứ tự
   trong dãy từ A[start] và  A[end], bắt đầu tại
   A[end].
{
   for  ( int  j = end ;  j > start ; j-- )
         if  (A [ j ] < A [ j - 1 ] )
              Swap ( A [ j ], A [ j - 1 ] ) ;
}
```

Các thuật toán sắp xếp và số phép so sánh trung bình

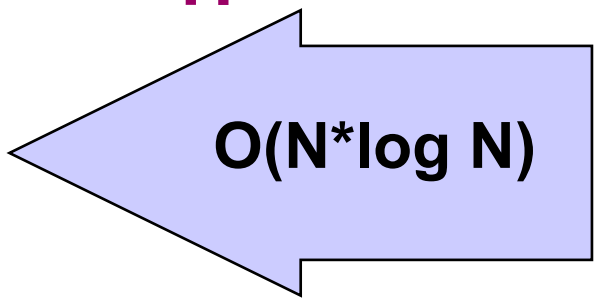
Các thuật toán sắp xếp đơn giản

- **Selection Sort**
- **Bubble Sort**
- **Insertion Sort**



Các thuật toán sắp xếp phức tạp hơn

- **Quick Sort**
- **Merge Sort**
- **Heap Sort**



3. Sắp xếp kiểu hòa nhập Mergesort

Dựa trên chiến lược chia để trị

- Chia dãy khóa thành 2 dãy nhỏ hơn có kích thước bằng nhau
- Trị: Sắp xếp mỗi dãy khóa *một cách đệ quy*
- Hòa nhập 2 dãy đã sắp xếp thành 1 dãy đã sắp xếp

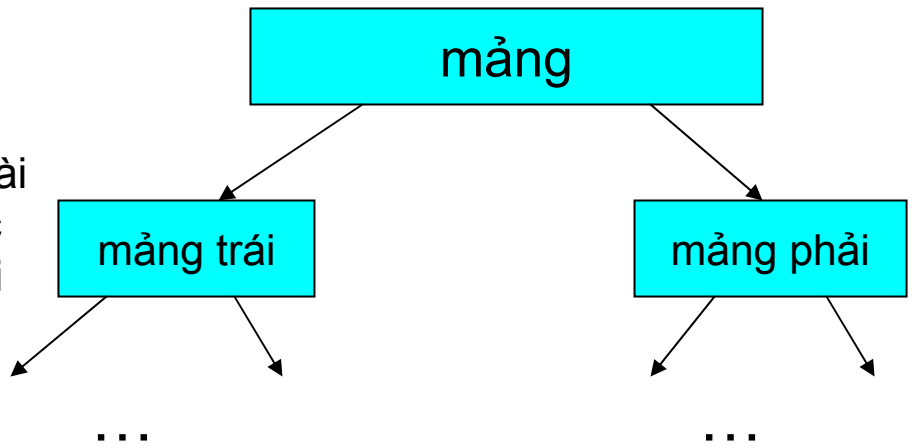
MergeSort: Bước chia và trị

Sắp xếp một mảng n số nguyên.

Dựa trên chiến lược *chia để trị* :

- *Chia*: chia đôi mảng thành 2 mảng con.

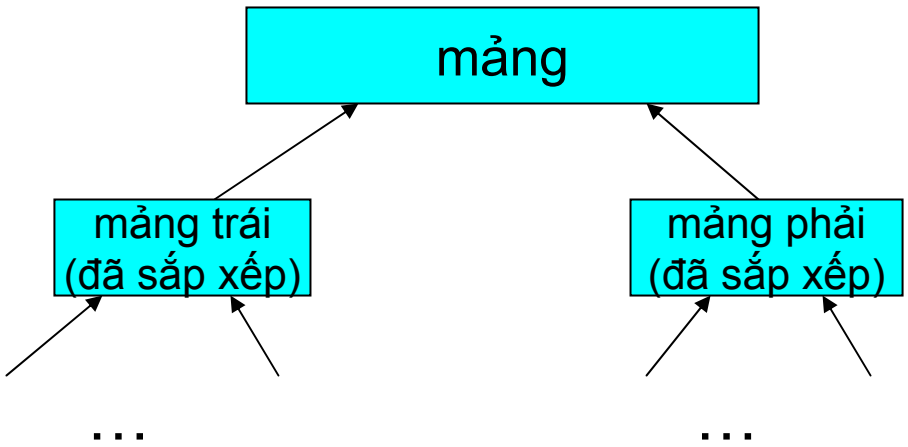
hai bài toán, mỗi bài toán có kích thước = ½ kích thước bài toán ban đầu



- Trị: sắp xếp 2 mảng con. Được thực hiện bằng đệ quy chia tiếp cho đến khi mảng có kích thước = 1

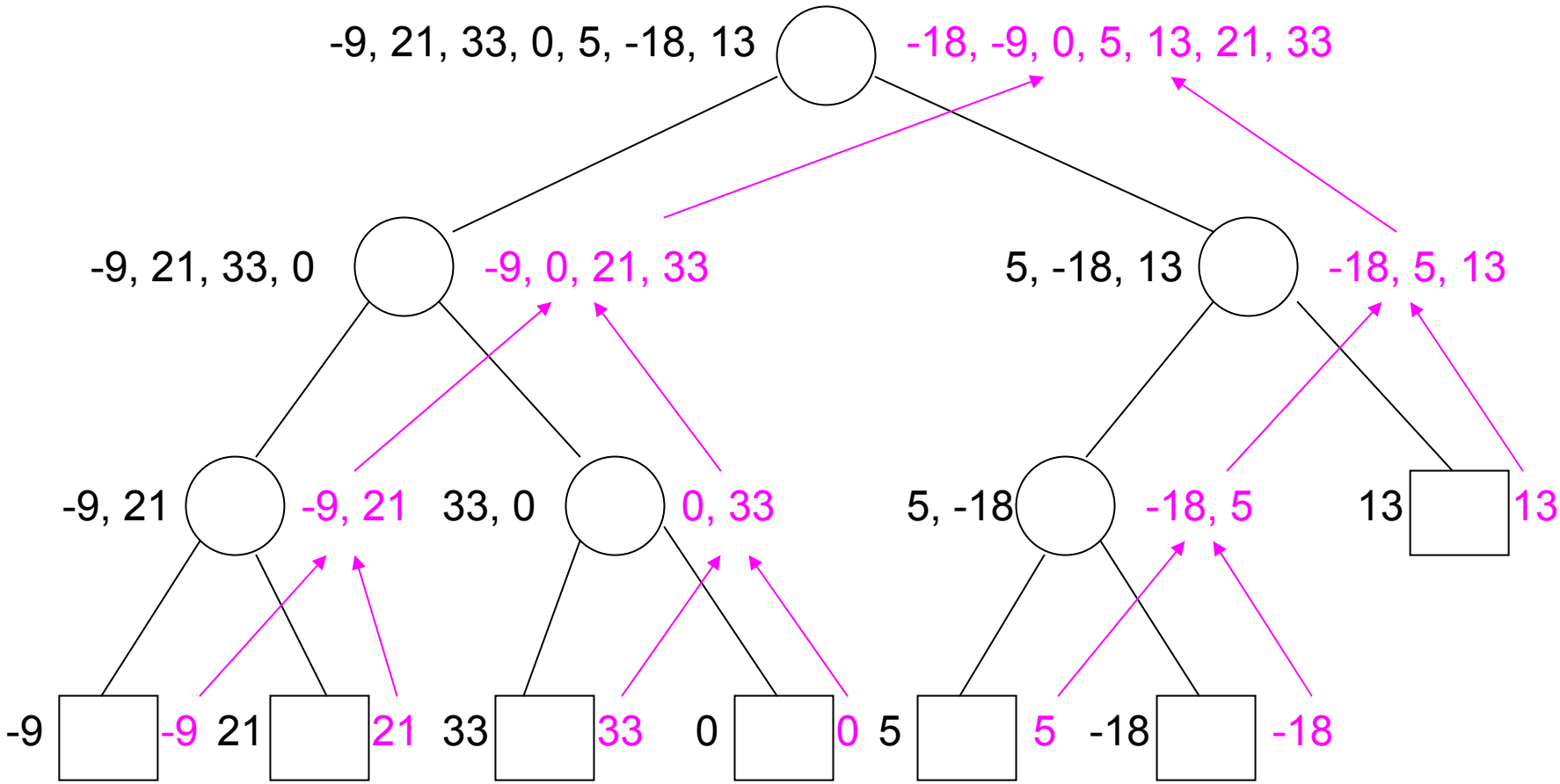
Mergesort: bước hòa nhập

- Hòa nhập: Hòa nhập 2 mảng con thành 1 mảng đã được săp xếp.



Chia dãy như thế nào? Thời gian tính?
Hòa nhập 2 dãy đã sắp xếp như thế nào? Thời gian tính?

Ví dụ


-9, 21, 33, 0, 5, -18, 13
-18, -9, 0, 5, 13, 21, 33
-9, 21, 33, 0
-9, 0, 21, 33
5, -18, 13
-18, 5, 13
-9, 21
-9, 21
33, 0
0, 33
5, -18
-18, 5
13
13
-9
-9
21
21
33
33
0
0
5
5
-18
-18

Chia như thế nào?

- Nếu dãy đầu vào được lưu dưới dạng danh sách móc nối: thao tác chia mất $\Theta(N)$
  - Quét toàn bộ danh sách, dừng lại tại phần tử thứ $\lfloor N/2 \rfloor$ và cắt móc nối
- Nếu dãy đầu vào là mảng: thao tác chia mất $O(1)$
  - Đầu vào `A[left..Right]`
  - Đầu ra:

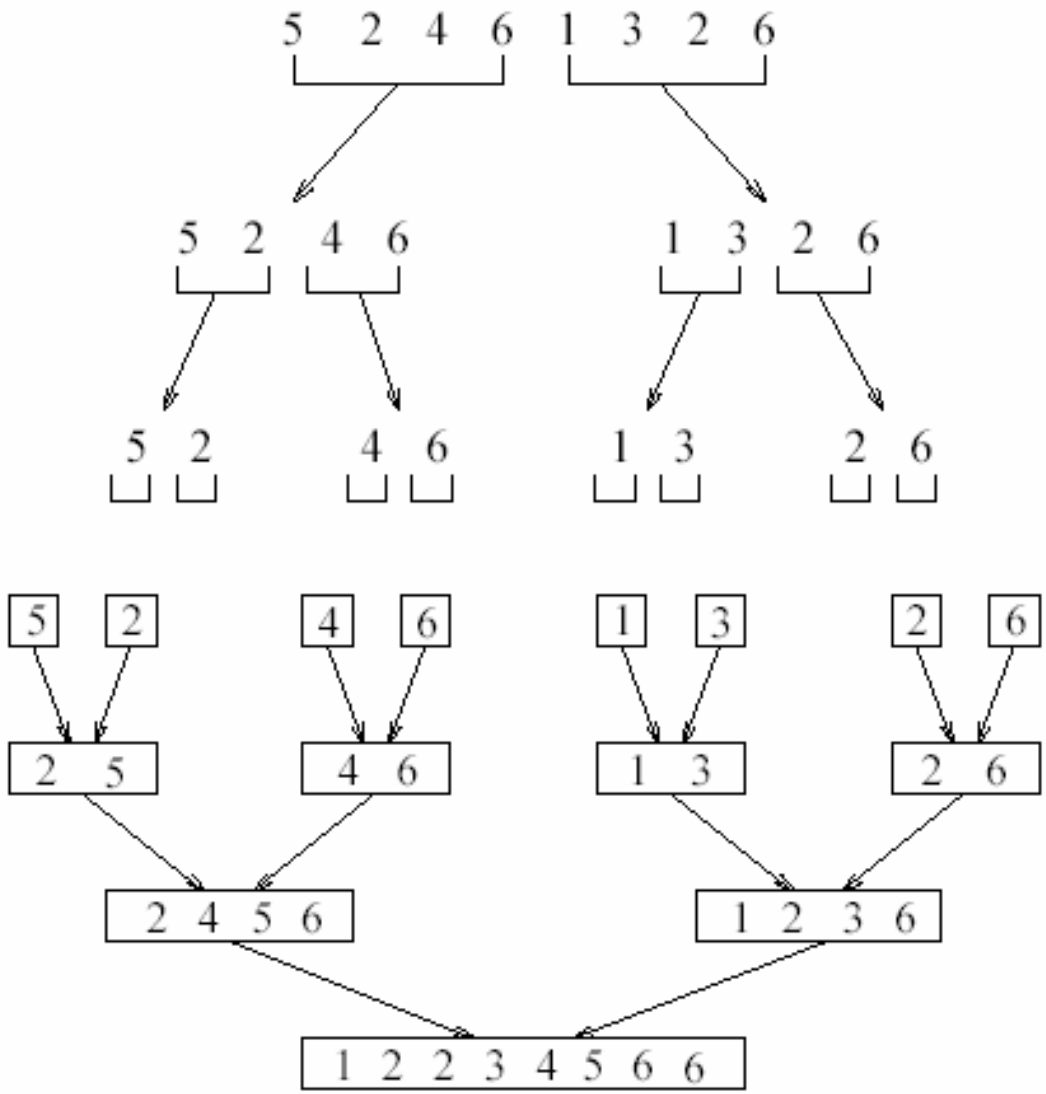
    `A[left..center]` và `A[center+1..Right]`

    với `center=(left+right)/2`

Giải thuật Mergesort

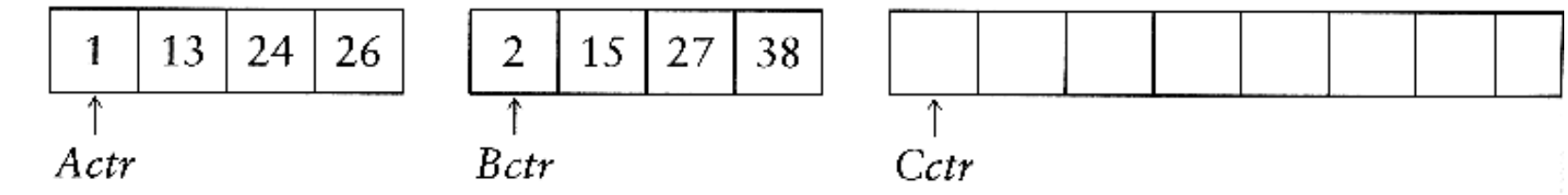
- Chia để trị
  - Sắp xếp một cách đệ quy với cả hai nửa chia được
  - hòa nhập 2 nửa sau khi sắp xếp lại với nhau

```
void mergesort(vector<int> & A, int left, int right)
{
   if (left < right) {
      int center = (left + right)/2;
      mergesort(A,left,center);
      mergesort(A,center+1,right);
      merge(A,left,center+1,right);
   }
}
```


5 2 4 6 1 3 2 6
5 2 4 6
1 3 2 6
5 2
4 6
1 3
2 6
5
2
4
6
1
3
2
6
2 5
4 6
1 3
2 6
2 4 5 6
1 2 3 6
1 2 2 3 4 5 6 6

Hòa nhập như thế nào?

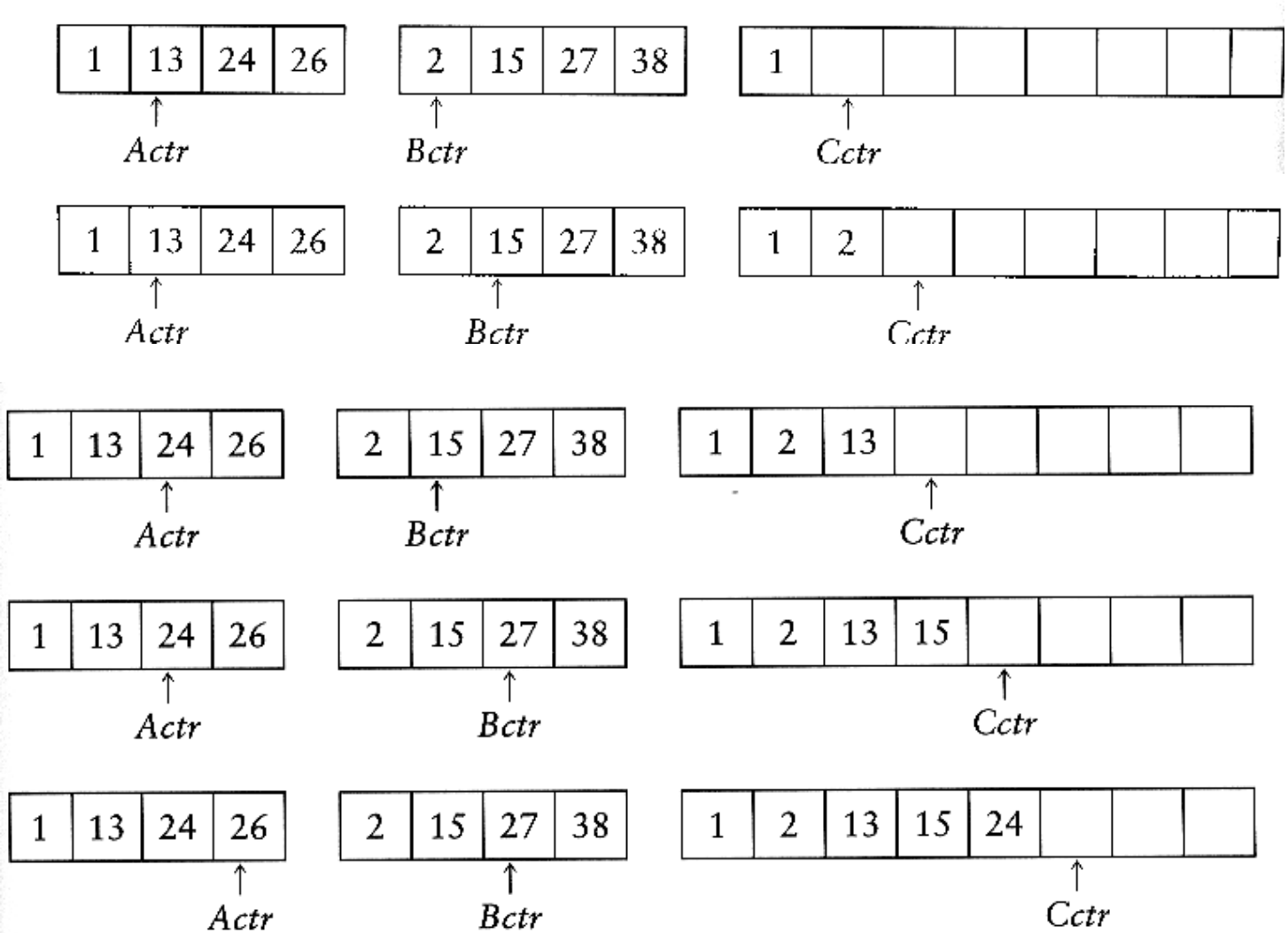
- Đầu vào: 2 mảng đã sắp xếp A và B
- Đầu ra: một mảng tổng hợp C được sắp xếp
- Ba biến chỉ số: Actr, Bctr, and Cctr
  - ban đầu chỉ đến phần tử đầu của từng mảng tương ứng



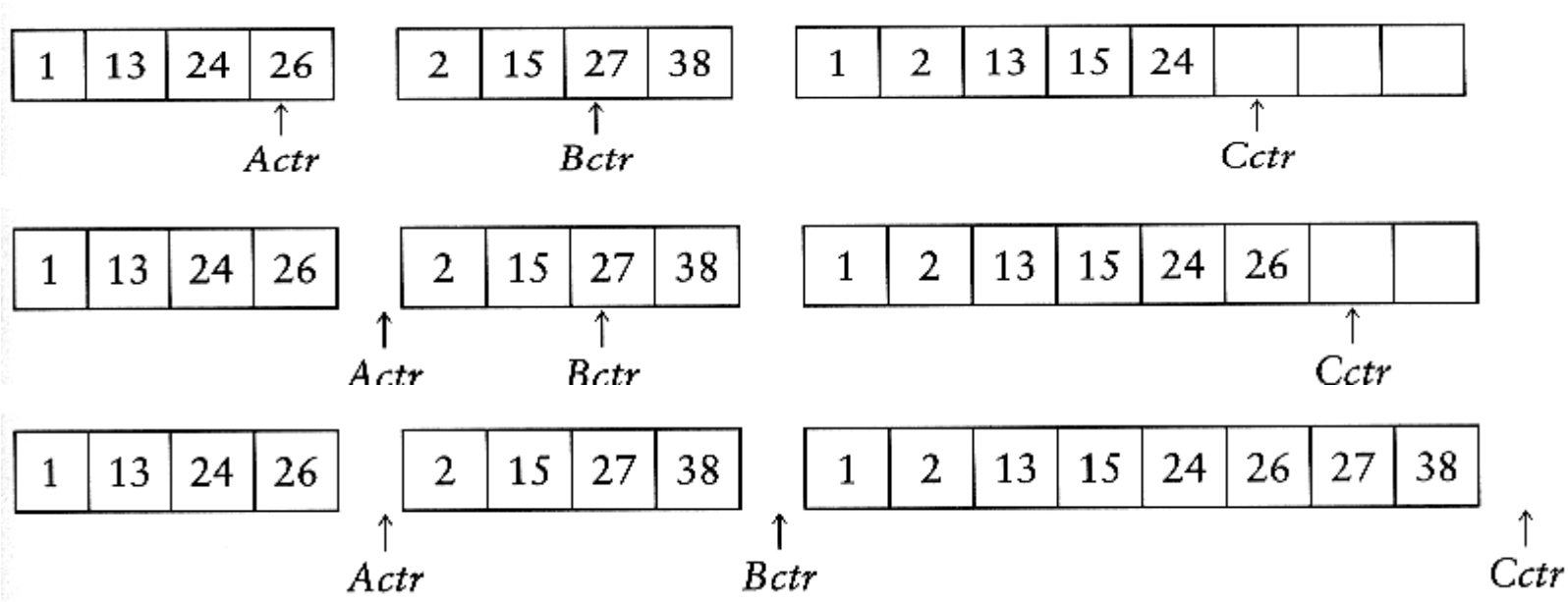
vị trí tiếp theo trong C, tăng biến chỉ số của mảng tương ứng.

(2) Nếu một trong hai dãy kết thúc, copy phần còn lại của dãy kia vào C

Ví dụ: Phép hòa nhập



Ví dụ: Phép hòa nhập



- Thời gian tính:
  - Thao tác hòa nhập mất O(m1 + m2) với m1 và m2 là kích thước của hai mảng.
- Bộ nhớ:
  - việc hòa nhập hai dãy cần sử dụng thêm O(m1+m2) không gian nhớ
  - ngoài ra còn phải copy sang mảng tạm C và copy ngược lại

Algorithm *merge*(A, p, q, r)
Input: Subarrays $A[p..l]$ and $A[q..r]$ s.t. $p \leq l = q - 1 < r$.
Output: $A[p..r]$ is sorted.
$(*\ T$ is a temporary array. $*)$

1. $k = p;\ i = 0;\ l = q - 1;$
2. **while** $p \leq l$ and $q \leq r$
3. **do if** $A[p] \leq A[q]$
4. **then** $T[i] = A[p];\ i = i + 1;\ p = p + 1;$
5. **else** $T[i] = A[q];\ i = i + 1;\ q = q + 1;$
6. **while** $p \leq l$
7. **do** $T[i] = A[p];\ i = i + 1;\ p = p + 1;$
8. **while** $q \leq r$
9. **do** $T[i] = A[q];\ i = i + 1;\ q = q + 1;$
10. **for** $i = k$ to r
11. **do** $A[i] = T[i - k];$

Độc phức tạp của mergesort

Gọi T(N) là thời gian tính trong trường hợp xấu nhất của mergesort để sắp xếp dãy có N phần tử.

Giả sử N là lũy thừa của 2.

- Bước chia: O(1)
- Bước trị: 2 T(N/2)
- Bước hòa nhập: O(N)

Công thức đệ quy:

$T(1) = 1$

$T(N) = 2T(N/2) + N$

Độ phức tạp của mergesort

$$T(N) = 2T(\frac{N}{2}) + N$$

$$= 2(2T(\frac{N}{4}) + \frac{N}{2}) + N$$

$$= 4T(\frac{N}{4}) + 2N$$

$$= 4(2T(\frac{N}{8}) + \frac{N}{4}) + 2N$$

$$= 8T(\frac{N}{8}) + 3N = \Lambda$$

$$= 2^k T(\frac{N}{2^k}) + kN$$

Vì $N=2^k$, ta có $k=\log_2 n$

$$T(N) = 2^k T(\frac{N}{2^k}) + kN$$

$$= N + N\log N$$

$$= O(N\log N)$$

- Độ phức tạp về bộ nhớ:

$O(n)$ đơn vị bộ nhớ cần cấp phát thêm.

3. Sắp xếp nhanh/phân đoạn (Quick Sort/ Partition Sort)

- Được coi là thuật toán nhanh nhất trong thực tế
- Thời gian tính trung bình: O(N log N)
- Xấu nhất: $O(N^2)$
  - Nhưng trường hợp xấu nhất hiếm khi xảy ra.
- Là một thuật toán đệ quy chia để trị, giống như sắp xếp hòa nhập (merge sort)

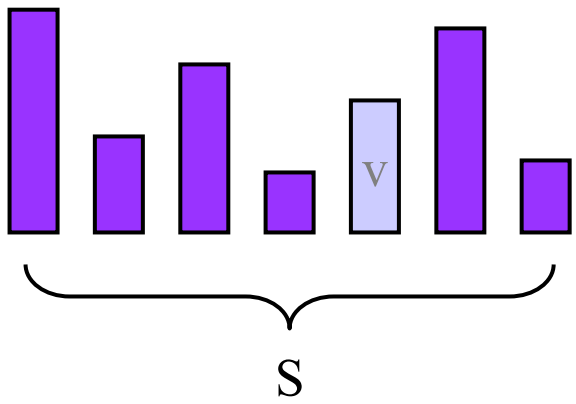
Quicksort

- Bước chia:
  - Chọn một phần tử bất kỳ (***chốt-pivot***) v trong S
  - Phân S thành 3 nhóm (phân đoạn):

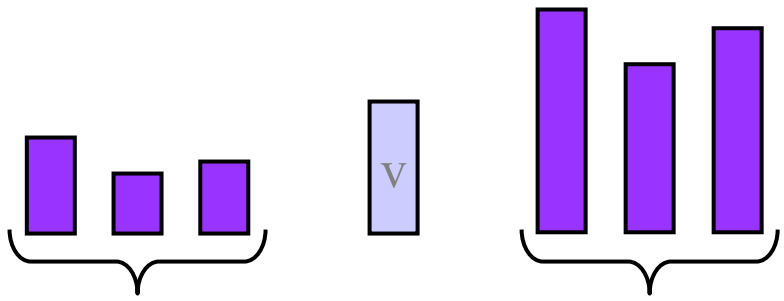
    S1 = {x ∈ S – {v} | x ≤ v}

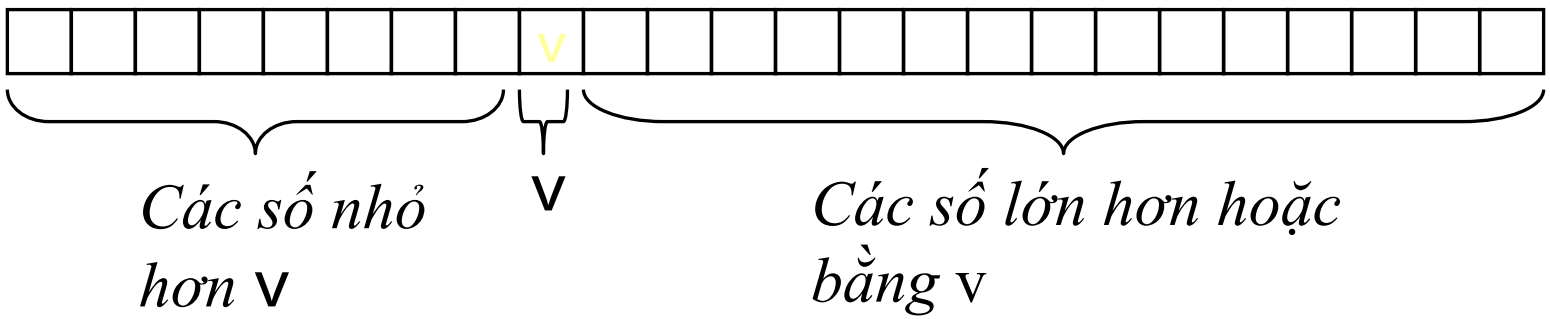
    v

    S2 = {x ∈ S – {v} | x ≥ v}
- Bước trị: đệ quy với S1 và S2

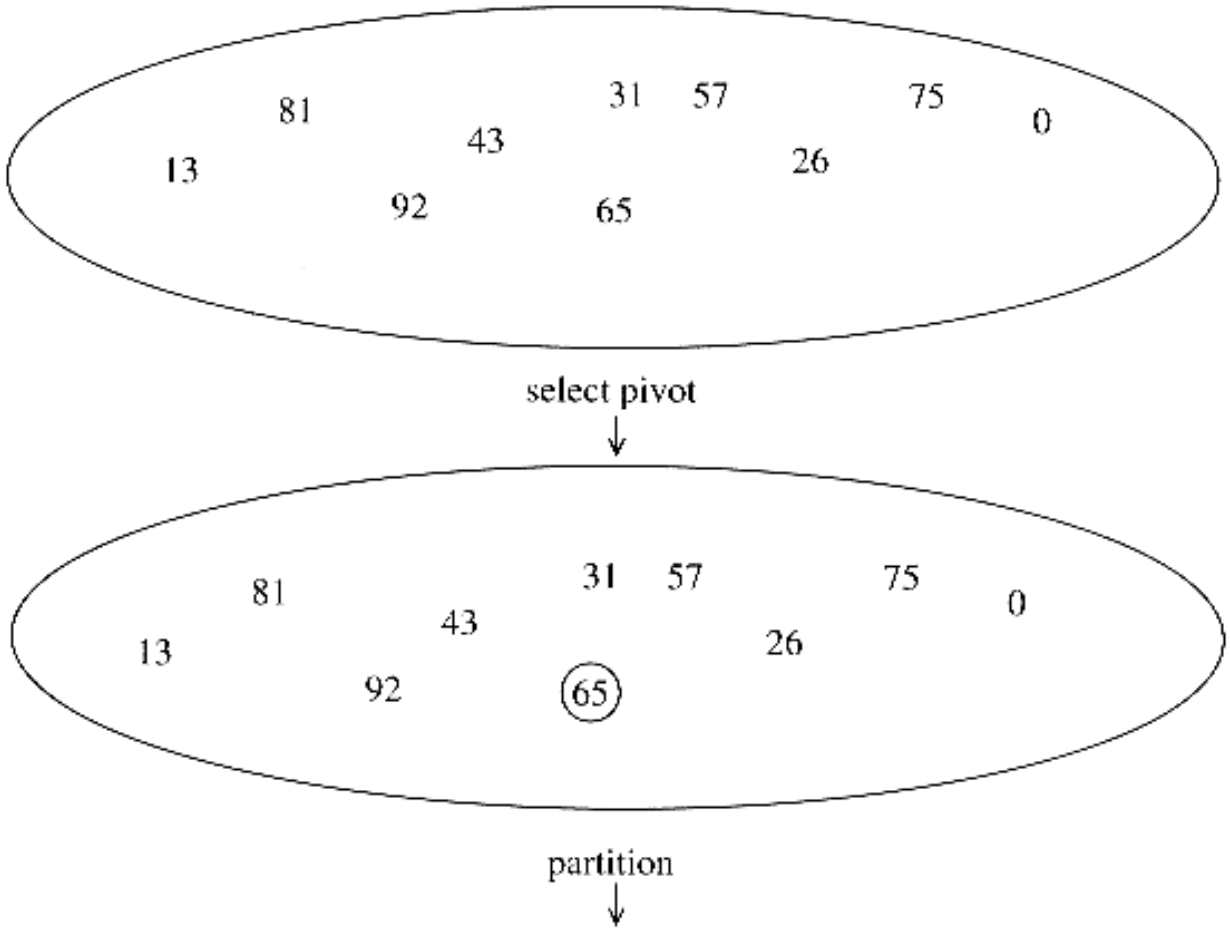


- Bước phối hợp: đoạn S1 đã được sắp xếp (tại thời điểm trở về từ hàm đệ quy), tiếp đến là v, và cuối cùng là S2 đã được sắp xếp

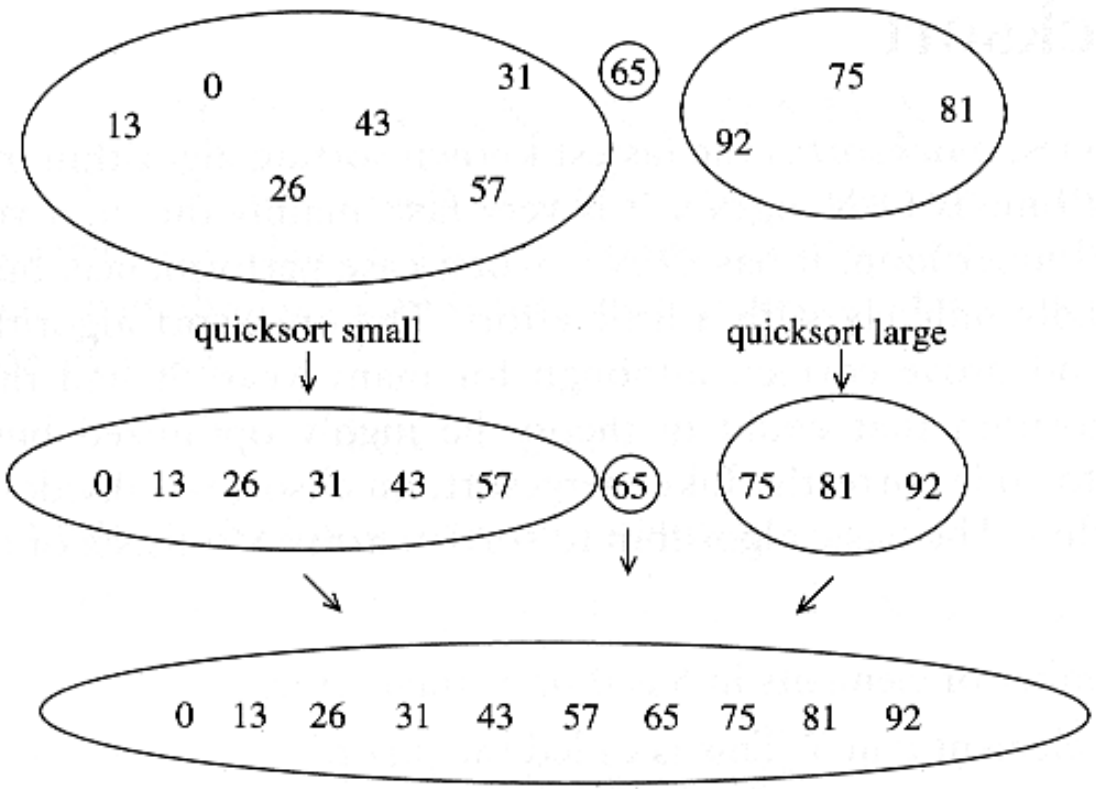




Example: Quicksort



Example: Quicksort...



Giải thuật

Đầu vào: mảng A[p, r]

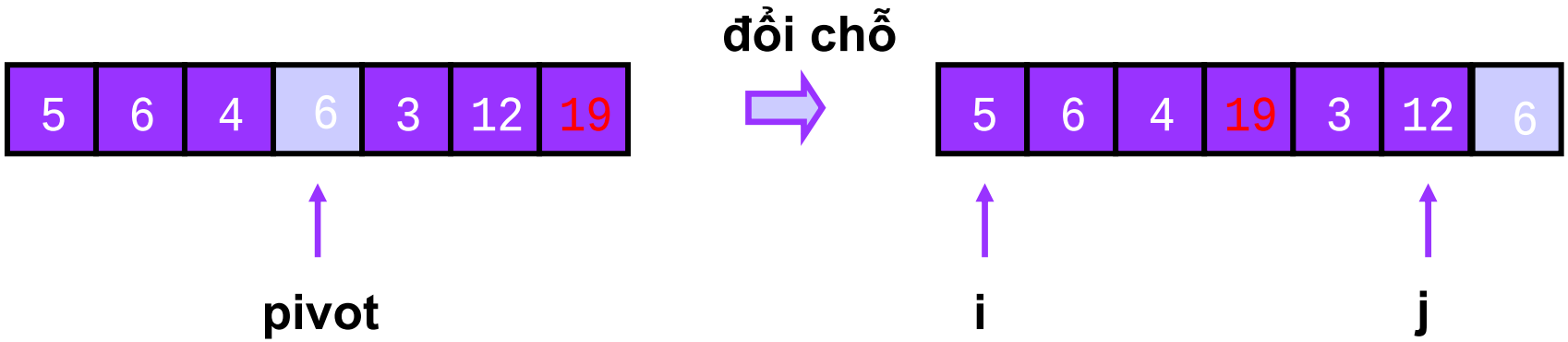
```
Quicksort (A, p, r) {
    if (p < r) {
        q = Partition (A, p, r)   // q là vị trí của phần tử chốt
        Quicksort (A, p, q-1)
        Quicksort (A, q+1, r)
    }
}
```

Phân đoạn

- Phân đoạn
  - Bước chính của thuật toán sắp xếp nhanh
  - Mục đích: với một khóa đã cho, phân các phần tử còn lại thành 2 phần nhỏ hơn
  - Có nhiều cách thực hiện
- Chúng ta sẽ xem xét một cách phân đoạn đơn giản và hiệu quả.
- Làm cách nào để chọn “chốt” sẽ được xem xét sau

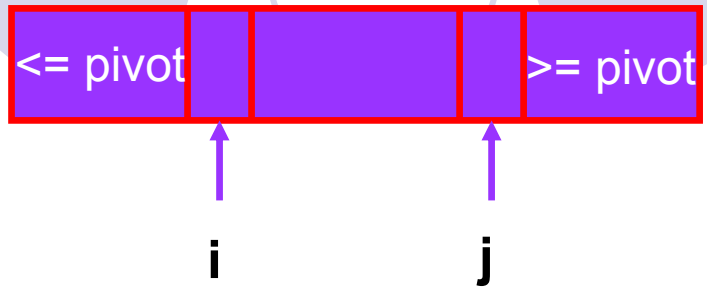
Giải thuật Phân đoạn

- Phân đoạn mảng A[left .. right]
- Đầu tiên, rút phần tử chốt pivot ra bằng cách đổi chỗ nó với phần tử cuối. (đổi chỗ pivot và A[right])
- Thiết lập i = left, j = right – 1

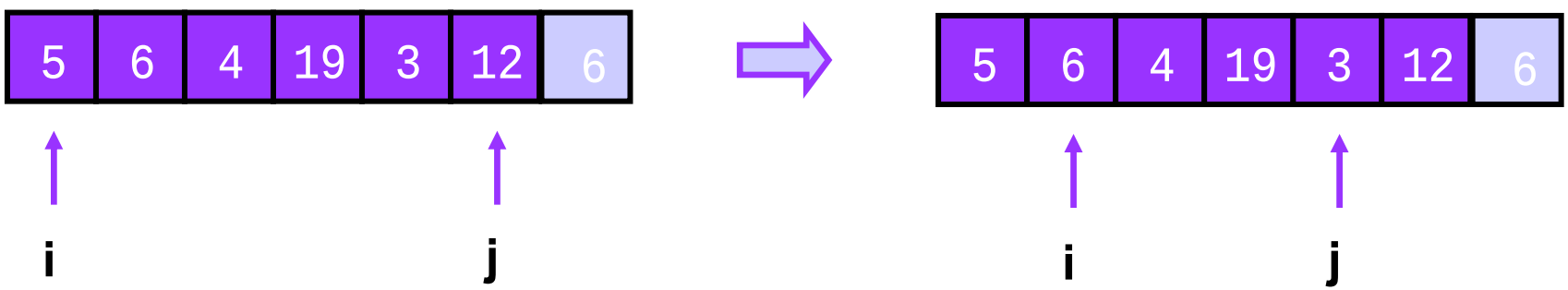


Giải thuật Phân đoạn

- Muốn có:
  - A[p] <= pivot, với p < i
  - A[p] >= pivot, với p > j



- Khi i < j
  - Dịch chuyển i sang phải, bỏ qua những phần tử nhỏ hơn pivot
  - Dịch chuyển j sang trái, bỏ qua những phần tử lớn hơn pivot
  - Khi cả i và j đều dừng
    - A[i] >= pivot
    - A[j] <= pivot

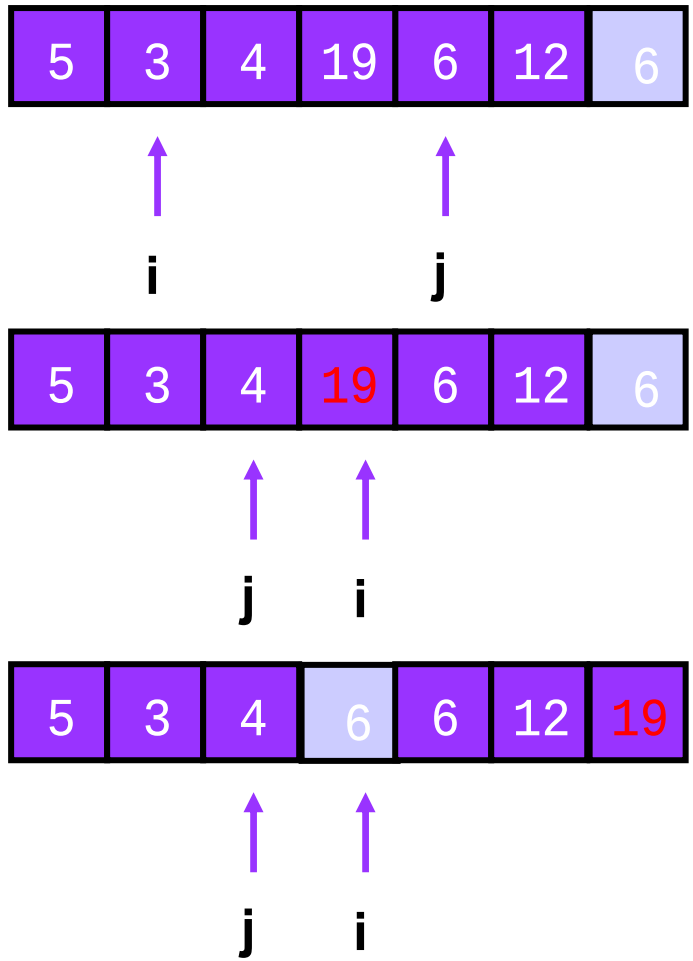


Giải thuật phân đoạn

- Khi i và j dừng và i < j
  - Đổi chỗ A[i] và A[j]
    - Phần tử lớn được đẩy sang bên phải và phần tử nhỏ được đẩy sang bên trái
  - Sau khi đổi chỗ
    - A[i] <= pivot
    - A[j] >= pivot
  - Lặp lại cho đến khi i và j giao nhau

Giải thuật phân đoạn

- Khi i và j giao nhau
  - Đổi chỗ A[i] và pivot
- Kết quả là:
  - A[p] <= pivot, với p < i
  - A[p] >= pivot, với p > i



Phân tích

- Với mảng có kích thước nhỏ, quicksort không hiệu quả bằng insertion sort
  - Vì quicksort mất thời gian để gọi hàm đệ quy
- Không nên sử dụng quicksort cho các mảng có kích thước nhỏ
  - Insertion Sort khá hiệu quả đối với các mảng có kích thước nhỏ

Lựa chọn chốt - pivot

- Chọn phần tử đầu tiên là chốt
  - Nếu đầu vào là ngẫu nhiên, ok
  - Nếu đầu vào là đã sắp xếp (hoặc theo thứ tự ngược lại)
    - tất cả phần tử được sẽ nằm trong S2 (hoặc S1)
    - Điều này lại tiếp tục lặp lại trong các lần đệ quy tiếp theo
    - Thời gian tính là $O(n^2)$
- Chọn chốt ngẫu nhiên
  - tương đối an toàn
  - việc chọn số ngẫu nhiên có thể tốn thời gian

Lựa chọn chốt - Pivot

- Lựa chọn phần tử trung vị của mảng làm chốt
  - Việc phân đoạn luôn tách đôi thành hai mảng có độ dài (gần) bằng nhau
  - Thuật toán quicksort tối ưu (O(N log N))
  - Tuy nhiên, rất khó để tìm được trung vị

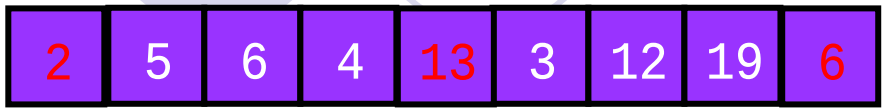
Chốt: trung vị của ba khóa

- Thay vào đó, ta chọn chốt là trung vị của ba khóa
  - So sánh ba phần tử: trái nhất, phải nhất và phần tử giữa mảng
  - Đổi chỗ các phần tử để thu được
    - A[left] = Nhỏ nhất trong ba phần tử
    - A[right] = Lớn nhất trong ba phần tử
    - A[center] = Trung vị của ba phần tử
  - Chọn A[center] làm chốt
  - Đổi chỗ A[center] và A[right – 1] để chốt nằm ở vị trí cạnh vị trí phải nhất (tại sao?)

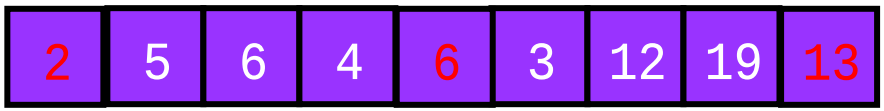
median3

```
int center = ( left + right ) / 2;
if( a[ center ] < a[ left ] )
    swap( a[ left ], a[ center ] );
if( a[ right ] < a[ left ] )
    swap( a[ left ], a[ right ] );
if( a[ right ] < a[ center ] )
    swap( a[ center ], a[ right ] );

    // Place pivot at position right - 1
swap( a[ center ], a[ right - 1 ] );
```

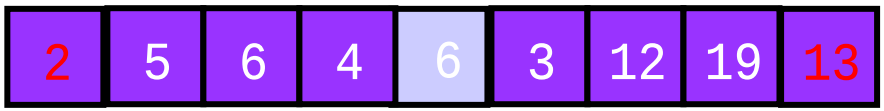
Chốt: trung vị của 3 khóa



A[left] = 2, A[center] = 13,
A[right] = 6

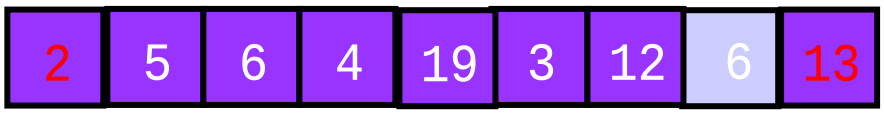


Đổi chỗ A[center] và A[right]



Chọn A[center] làm chốt

pivot



Đổi chỗ chốt và A[right – 1]

pivot

Chú ý: chỉ cần phân đoạn trên A[left + 1, …, right – 2]. Tại sao?

Thủ tục Quicksort

```
if( left + 10 <= right )
{
    Comparable pivot = median3( a, left, right );

        // Begin partitioning
    int i = left, j = right - 1;
    for( ; ; )
    {
        while( a[ ++i ] < pivot ) { }
        while( pivot < a[ --j ] ) { }
        if( i < j )
            swap( a[ i ], a[ j ] );
        else
            break;
    }

    swap( a[ i ], a[ right - 1 ] );  // Restore pivot

    quicksort( a, left, i - 1 );     // Sort small elements
    quicksort( a, i + 1, right );    // Sort large elements
}
else  // Do an insertion sort on the subarray
    insertionSort( a, left, right );
```

Chọn chốt

Phân đoạn

Đệ quy

Với mảng KT nhỏ

Giải thuật phân đoạn

- Chỉ làm việc với chốt là trung vị của ba khóa.
  - A[left] <= pivot và A[right] >= pivot
  - Do đó, chỉ cần phân đoạn mảng A[left + 1, …, right – 2]
- j không cần bắt đầu từ phần tử đầu
  - vì a[left] <= pivot
- i không cần bắt đầu từ phần tử cuối
  - vì a[right-1] = pivot

```
int i = left, j = right - 1;
for( ; ; )
{
    while( a[ ++i ] < pivot ) { }
    while( pivot < a[ --j ] ) { }
    if( i < j )
        swap( a[ i ], a[ j ] );
    else
        break;
}
```

Quicksort nhanh hơn Mergesort

- Cả Quicksort và MergeSort đều mất O(N log N) trong trường hợp trung bình.
- Tại sao quicksort nhanh hơn mergesort?
  - Vòng lặp trong gồm một phép tăng dần/giảm dần (1 đơn vị, nhanh), một phép kiểm tra điều kiện và một lệnh and a jump.
  - Không có thao tác hòa nhập (juggling) như mergesort.

```
int i = left, j = right - 1;
for( ; ; )
{
    while( a[ ++i ] < pivot ) { }
    while( pivot < a[ --j ] ) { }
    if( i < j )
        swap( a[ i ], a[ j ] );
    else
        break;
}
```

vòng lặp trong

Phân tích độ phức tạp

- Giả sử:
  - Chốt ngẫu nhiên (không sử dụng phân đoạn trung vị của ba khóa)
  - Không cắt những đoạn có kích thước nhỏ
- Thời gian tính
  - Lựa chọn chốt: hằng số - O(1)
  - Phân đoạn: tuyến tính O(N)
  - Thời gian tính của 2 lời gọi đệ quy
- $T(N)=T(i)+T(N-i-1)+cN$ trong đó c là hằng số
  - i: là số phần tử trong đoạn S1

Trường hợp xấu nhất

- Trường hợp nào là xấu nhất?
  - Chốt được chọn luôn là phần tử nhỏ nhất
  - Việc phân đoạn luôn bị lệch (một đoạn có 0 phần tử, một đoạn có N-1 phần tử)

$$
\begin{aligned}
T(N) &= T(N-1) + cN \\
T(N-1) &= T(N-2) + c(N-1) \\
T(N-2) &= T(N-3) + c(N-2) \\
&\vdots \\
T(2) &= T(1) + c(2) \\
T(N) &= T(1) + c\sum_{i=2}^{N} i = O(N^2)
\end{aligned}
$$

Trường hợp tốt nhất

- Trường hợp nào là tốt nhất?
  - Việc phân đoạn luôn tạo ra 2 đoạn có kích thước cân bằng.
  - Chốt được chọn luôn là trung vị của mảng

$$
\begin{aligned}
T(N) &= 2T(N/2) + cN \\
\frac{T(N)}{N} &= \frac{T(N/2)}{N/2} + c \\
\frac{T(N/2)}{N/2} &= \frac{T(N/4)}{N/4} + c \\
\frac{T(N/4)}{N/4} &= \frac{T(N/8)}{N/8} + c \\
&\vdots \\
\frac{T(2)}{2} &= \frac{T(1)}{1} + c \\
\frac{T(N)}{N} &= \frac{T(1)}{1} + c \log N \\
T(N) &= cN \log N + N = O(N \log N)
\end{aligned}
$$

Trường hợp trung bình

- Giả sử:
  - Mỗi lần phân đoạn, kích thước của S1 và S2 là tương đối cân bằng
- Giả định này khá đúng khi chọn chốt là trung vị của ba khóa
- Trong trường hợ trung bình, thời gian tính là O(N log N)

5. Sắp xếp vun đống HeapSort

Đặt vấn đề

3 yêu cầu công việc được gửi tới cho máy in A, B, C.

Số trang: Công việc A – 100 trang

Công việc B – 10 trang

Công việc C – 1 trang

Thời gian đợi nếu sử dụng FIFO:

(100+110+111) / 3 = 107 đơn vị thời gian

Thời gian đợi trung bình nếu lựa chọn công việc ngắn nhất trước:

(1+11+111) / 3 = 41 time units

Cần có một hàng đợi cho phép thêm và xóa phần tử nhỏ nhất

Hàng đợi ưu tiên

Hàng đợi ưu tiên
Priority Queue

3 thao tác:

- Thêm một phần tử theo mức ưu tiên phù hợp.
- Xóa phần tử có mức ưu tiên cao nhất
- Truy cập phần tử có mức ưu tiên cao nhất (không xóa)

Ứng dụng của Priority Queue

Trong phần lớn các ứng dụng, các phần tử của PQ là một cặp *khóa-giá trị*.

Mức ưu tiên	Nhiệm vụ

- Hệ thống mô phỏng các sự kiện phụ thuộc thời gian (ví dụ: bài toán giao thông trong sân bay)
- Lịch trình các tiến trình của hệ điều hành (chia sẻ thời gian)
- Là cấu trúc dữ liệu quan trọng thực hiện nhiều giải thuật (bài toán tìm cây khung nhỏ nhất, tìm đường đi ngắn nhất, ...)

Xây dựng Priority Queue

- Danh sách kết nối không có thứ tự
  - Thêm: O(1), xóa: O(N), truy cập: O(N)
- Mảng có thứ tự:
  - Thêm: O(N), xóa: O(N), truy cập: O(1)
- Danh sách kết nối có thứ tự:
  - Thêm: O(N), xóa: O(1), truy cập: O(1)
- Cây nhị phân tìm kiếm
  - Thêm, xóa: O(log N) – trung bình, O(N) – xấu nhất; truy cập: O(1)

Xây dựng PriorityQueue sử dụng Đống

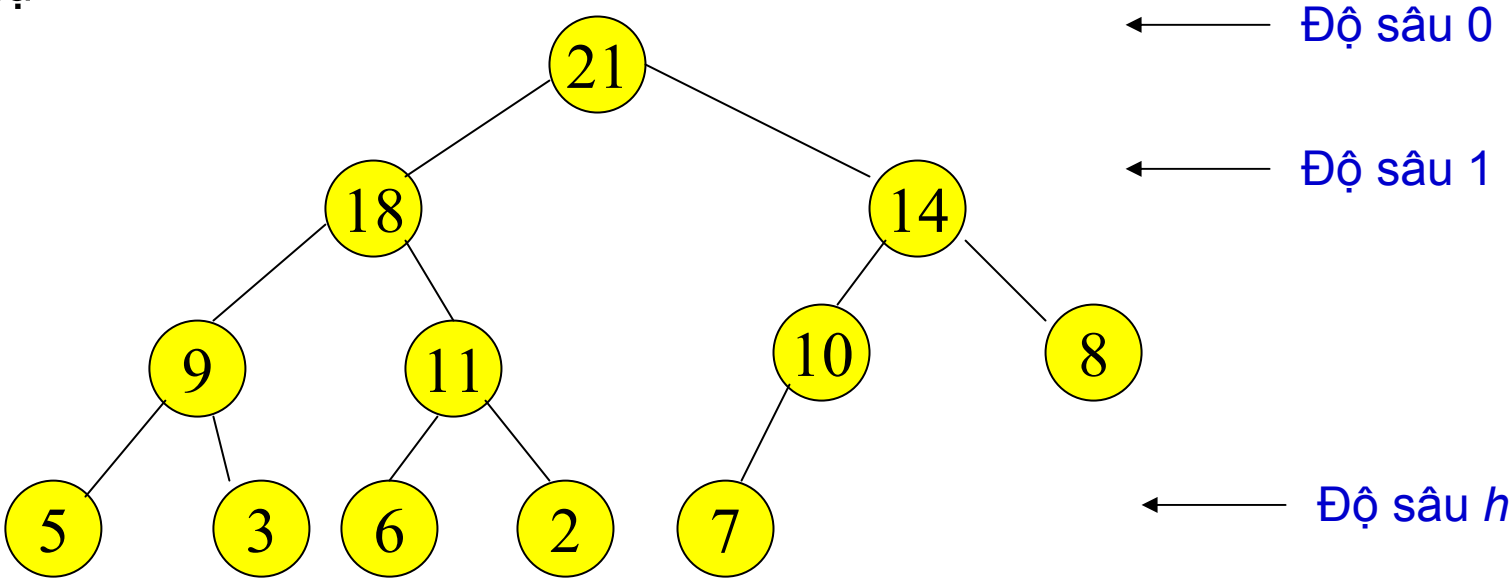
- Hỗ trợ các thao tác rất hiệu quả
  - Truy cập phần tử nhỏ nhất: O(1)
  - Thêm phần tử: O(log N)
  - Xóa phần tử nhỏ nhất: O(log N)

Đống (Heap)

Cây nhị phân có độ cao h

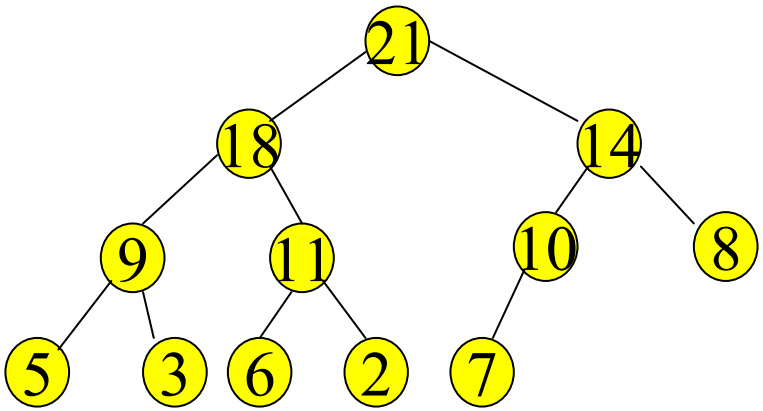
- *đầy đủ ít nhất đến độ sâu h – 1*
- có thể thiếu một số nút bên phải nhất ở độ sâu *h*

Ví dụ



Thuộc tính của Heap

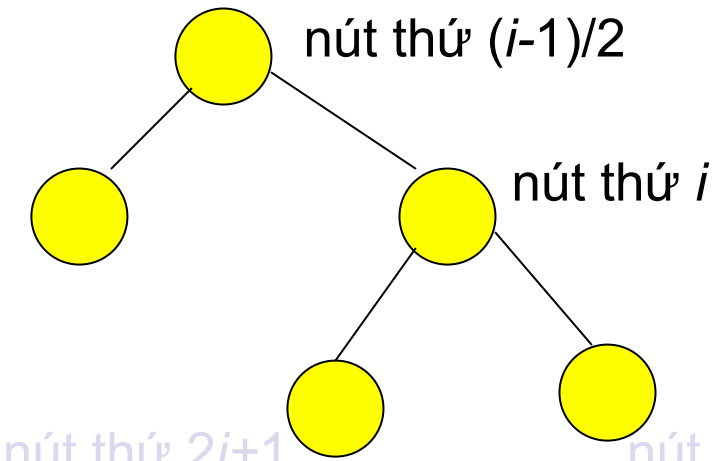
Giá trị [NútCha(*i*)] ≥ Giá trị [*i*], *i* > 1



Lưu trữ Heap sử dụng Mảng

21	18	14	9	11	10	8	5	3	6	2	7
Chỉ số	0	1	2	3	4	5	6	7	8	9	10	11

Dễ dàng truy cập các nút quan hệ cha con:



Nút Cha(i) = (i – 1)/2

Nút Con trái(i) = 2i + 1

Nút Con phải(i) = 2i + 2

```
int arr[] = { 21, 18, 14, 9, 11, 10, 8, 5, 3, 6, 2, 7 };
int arrSize = sizeof(arr) / sizeof(int);
```

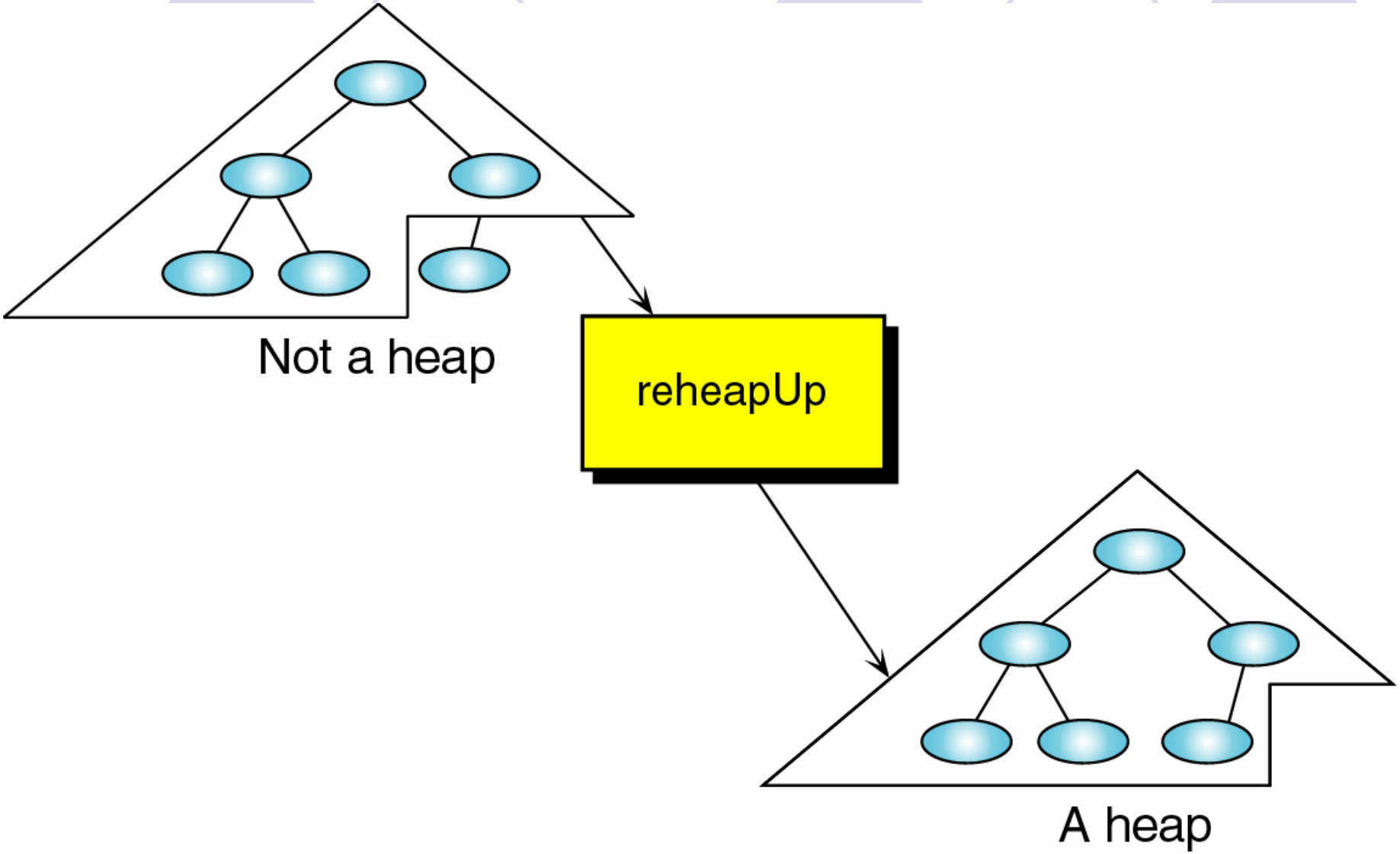
- Liệu có nên lưu trữ heap bằng danh sách liên kết?

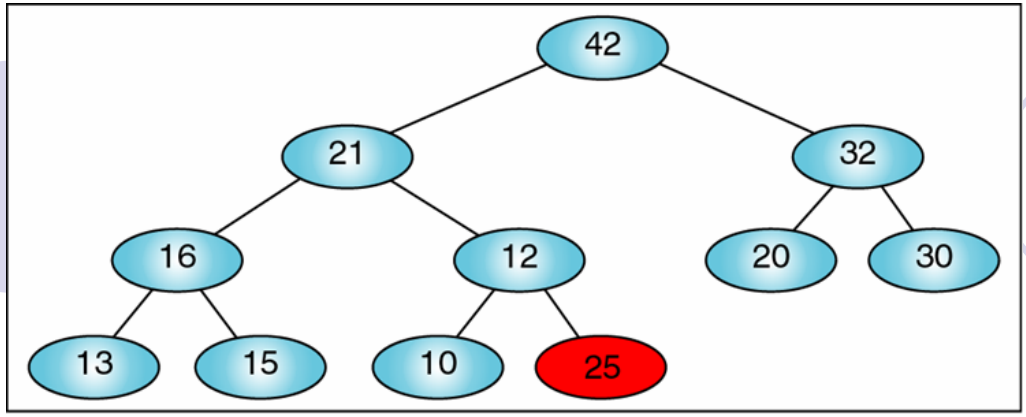
Các thao tác với Heap

- Thêm một phần tử
- Xóa phần tử đỉnh

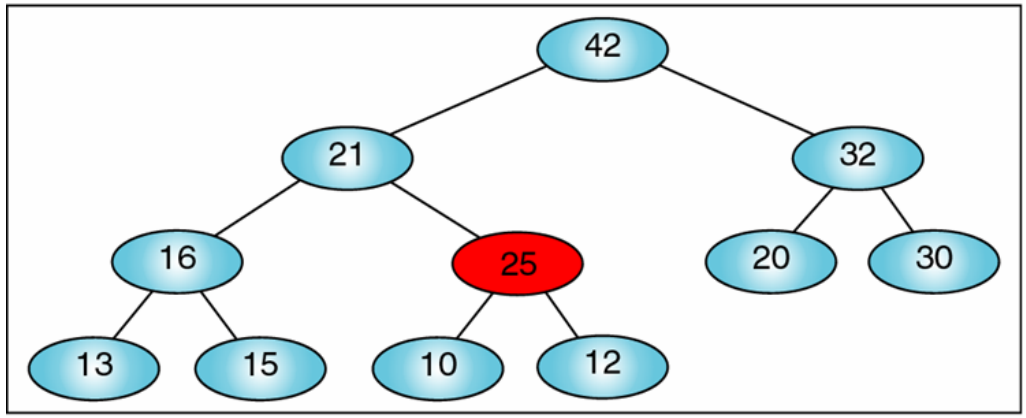
Thêm một phần tử vào đống

- Thêm một phần tử mới vào đáy của đống
- Vun lại đống từ dưới lên (reheapUp)

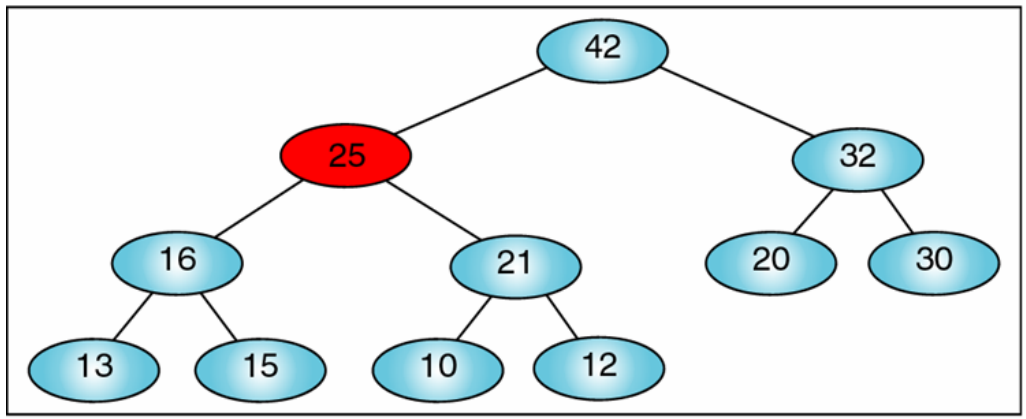

Not a heap
reheapUp
A heap



(a) Thêm phần tử mới vào nút lá cuối cùng



(b) "Vun" phần tử cuối (25) lên trên



(c) Tiếp tục "vun" đến khi cây là một đống

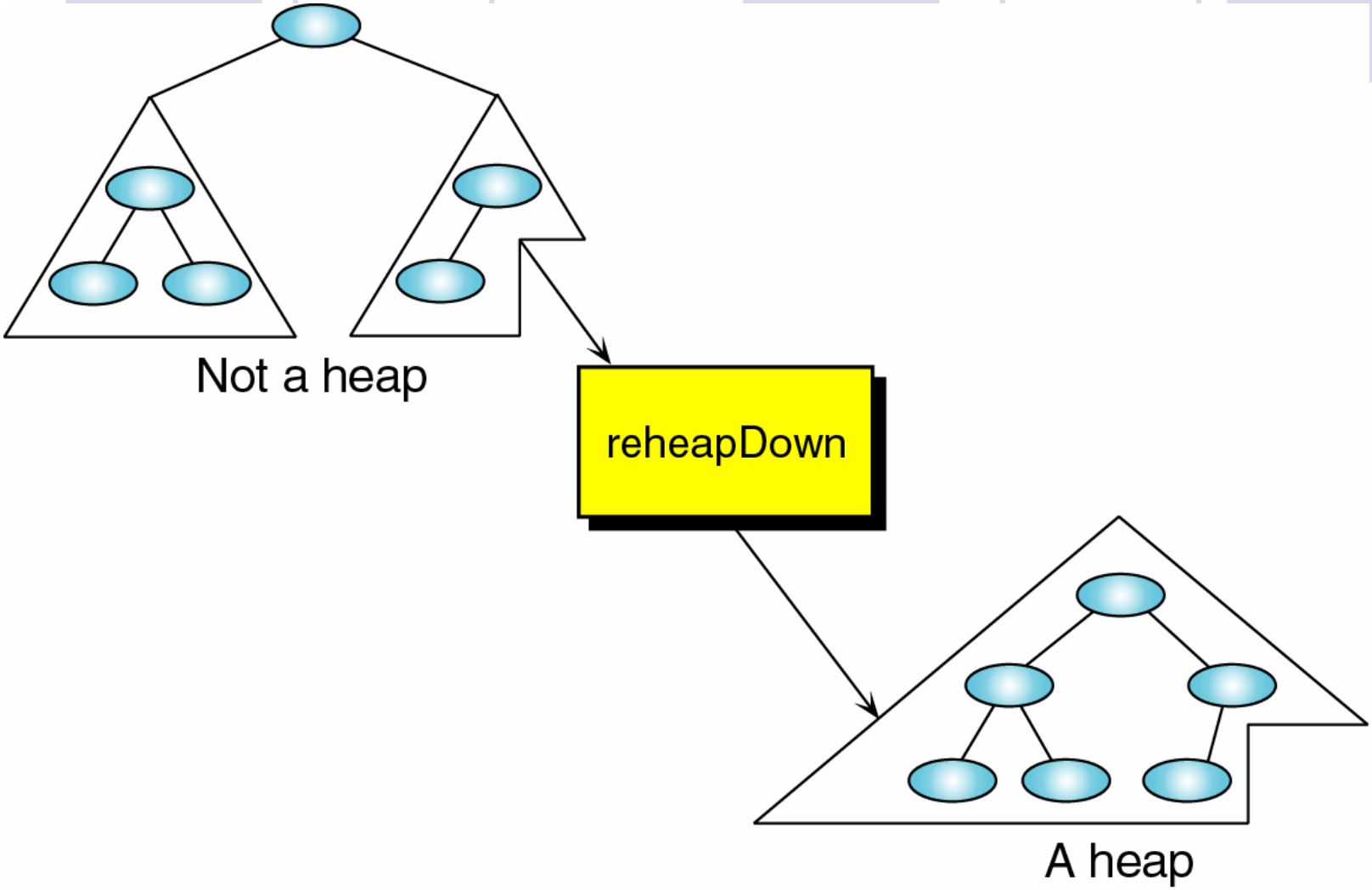
reheapUp

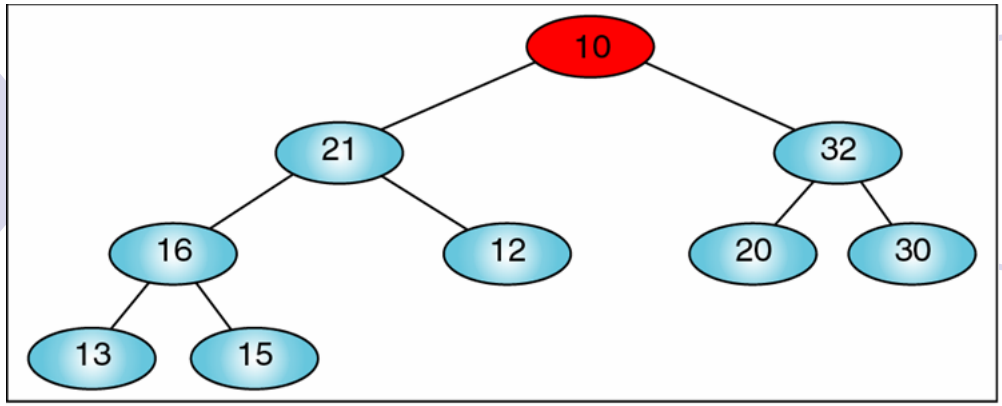
```
void reheapUp(int a[], int k)
{
  while(k>0 && a[(k-1)/2] < a[k])
  {
     exchange(a[k], a[(k-1)/2]);
     k = (k-1)/2;
  }
}
```

- Độ phức tạp của reheapUp = O(logN)

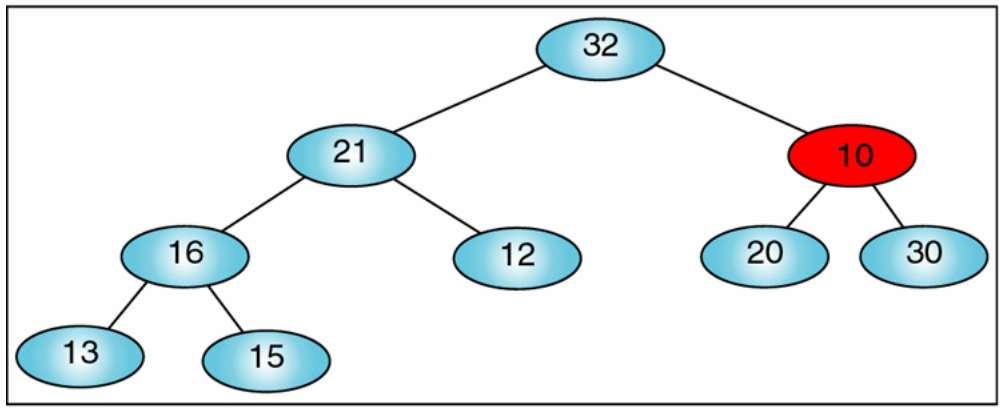
Xóa phần tử đỉnh của đống

- Đổi chỗ phần tử đỉnh bằng phần tử đáy cuối cùng
- Thiết lập kích thước đống giảm đi một phần tử
- Xếp lại đống từ trên xuống (reheapDown)

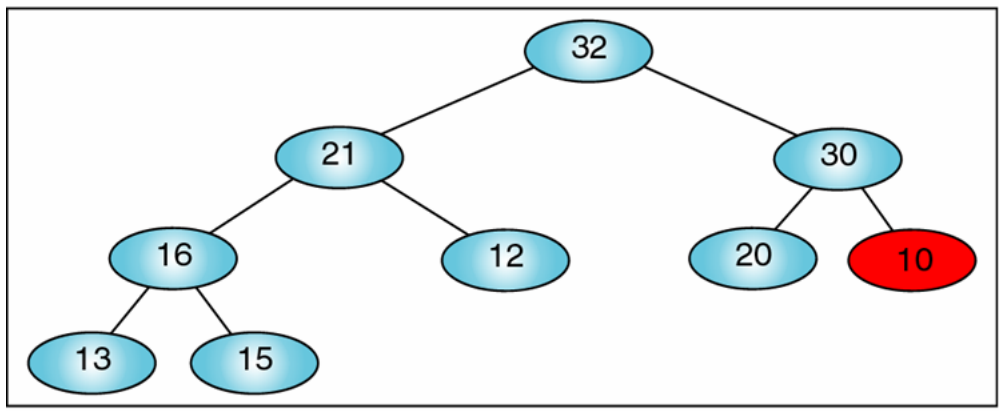

Not a heap
reheapDown
A heap



(a) Cây ban đầu: không phải đống



(b) Chuyển nút gốc xuống dưới



(c) Chuyển đến khi cây là đống

reheapDown

```
void reheapDown(int a[], int k, int
  N){
  while(2*k <= N-1){
      int j = 2*k+1;
      if(j<N-1 && a[j]<a[j+1]) j++;
      if(!(a[k]<a[j])) break;
      exchange(a[k], a[j]);
      k = j;
  }
}
```

- Độ phức tạp của reheapDown: O(log N)

Sắp xếp vun đống - Heapsort

(1) Tạo đống ban đầu gồm N phần tử

- Phần tử nhỏ nhất (lớn nhất) sẽ nằm tại đỉnh của đống

(2) Thực hiện N-1 lần thao tác xóa phần tử đỉnh

- Các phần tử sẽ được loại ra thứ tự

Bước 1: Tạo đống

- Đầu vào: N phần tử
- Đầu ra: Một đống của N phần tử

Thực hiện: 2 cách

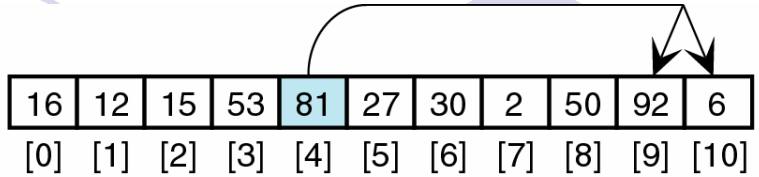
Cách 1:

- Thêm dần từng phần tử vào đống (sử dụng reheapUp)

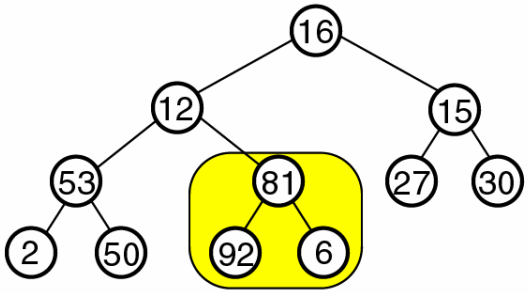
Cách 2:

- Xem mỗi phần tử mảng như gốc của một đống con (bỏ qua đống có kích thước 1)
- Sử dụng reheapDown để tạo đống lớn hơn

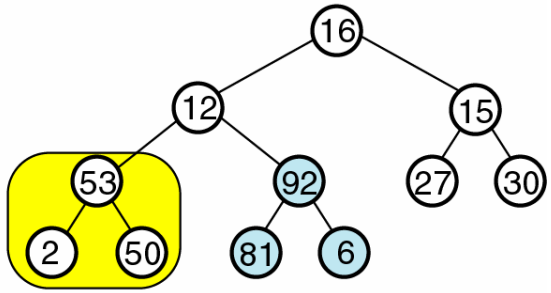
Thuật toán HeapSort sử dụng cách 2



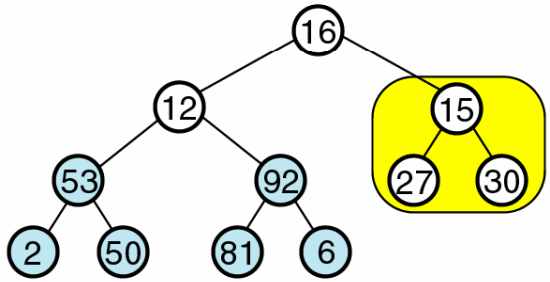
(a) Nonheap array



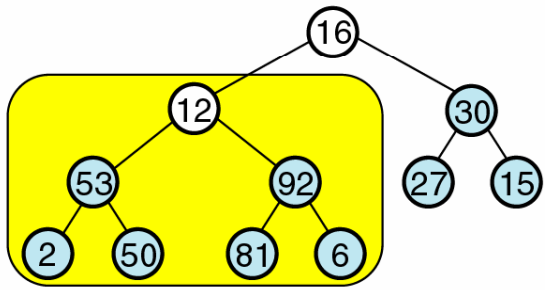
(b) Before first reheap down



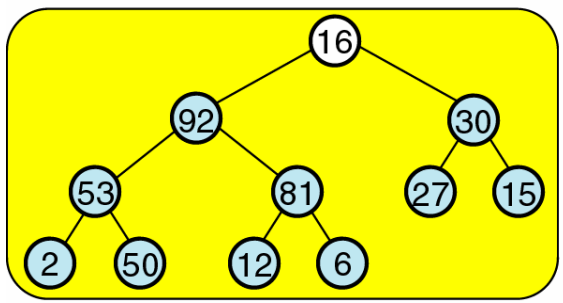
(c) After first reheap down



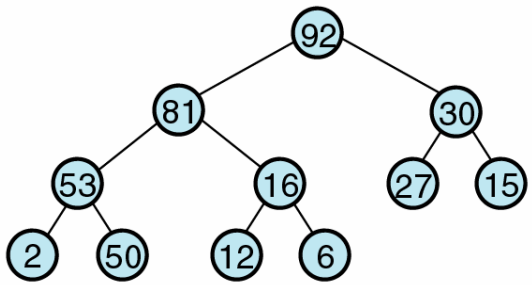
(d) After second reheap down



(e) After third reheap down



(f) After fourth reheap down



(g) Complete heap

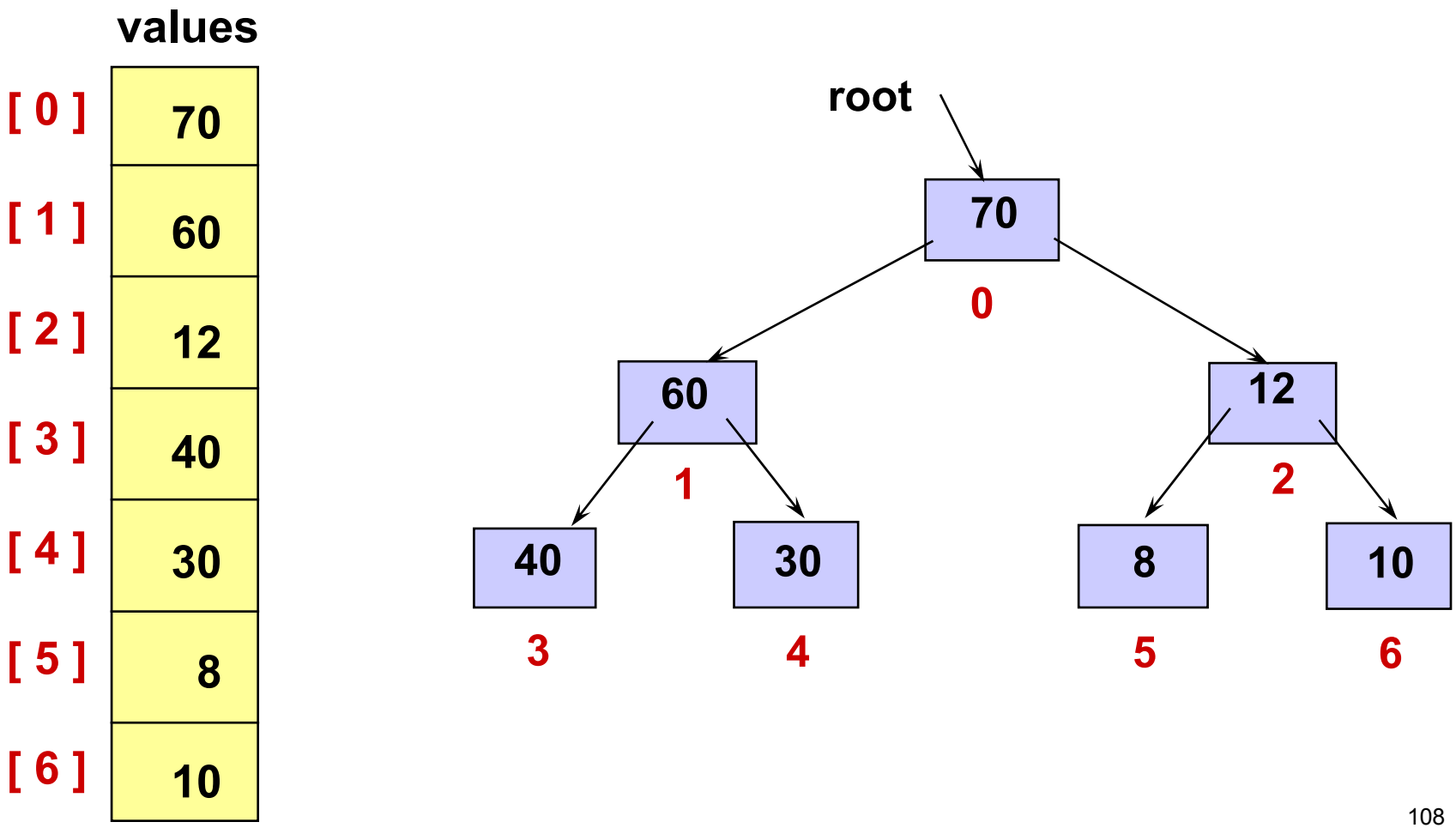
Heapsort

```
void heapsort(int a[], int left, int
  right)
{
  int k, N = right-left+1;
  int *pq = a+left-1;
  for(k=(N-1)/2; k>=0; k--)
     reheapDown(pq, k, N);
  while(N > 0){
     exchange(pq[0], pq[N-1]);
     reheapDown(pq, 0, --N);
  }
}
```

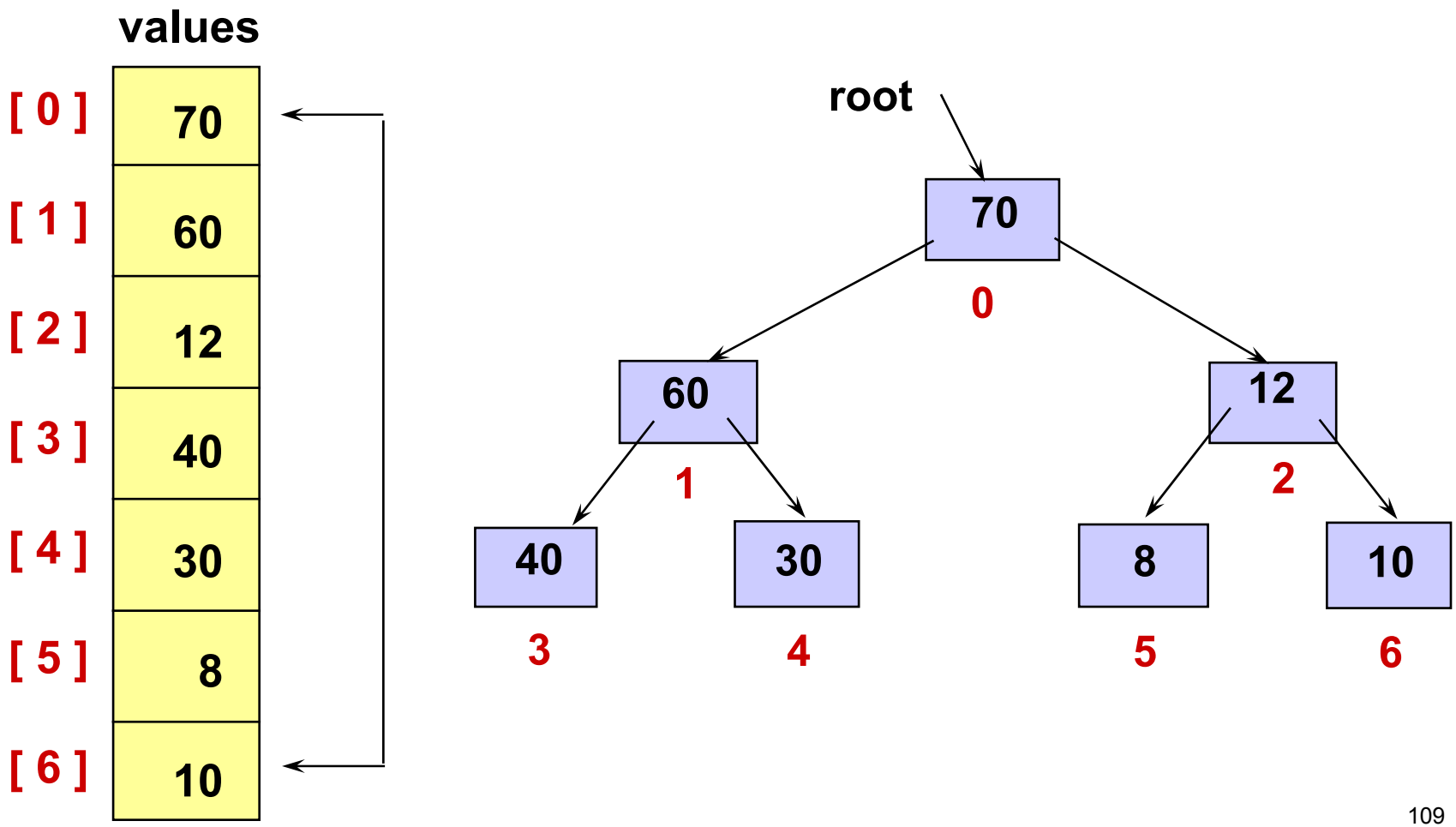
Tạo đống

Xóa lần lượt phần tử đỉnh

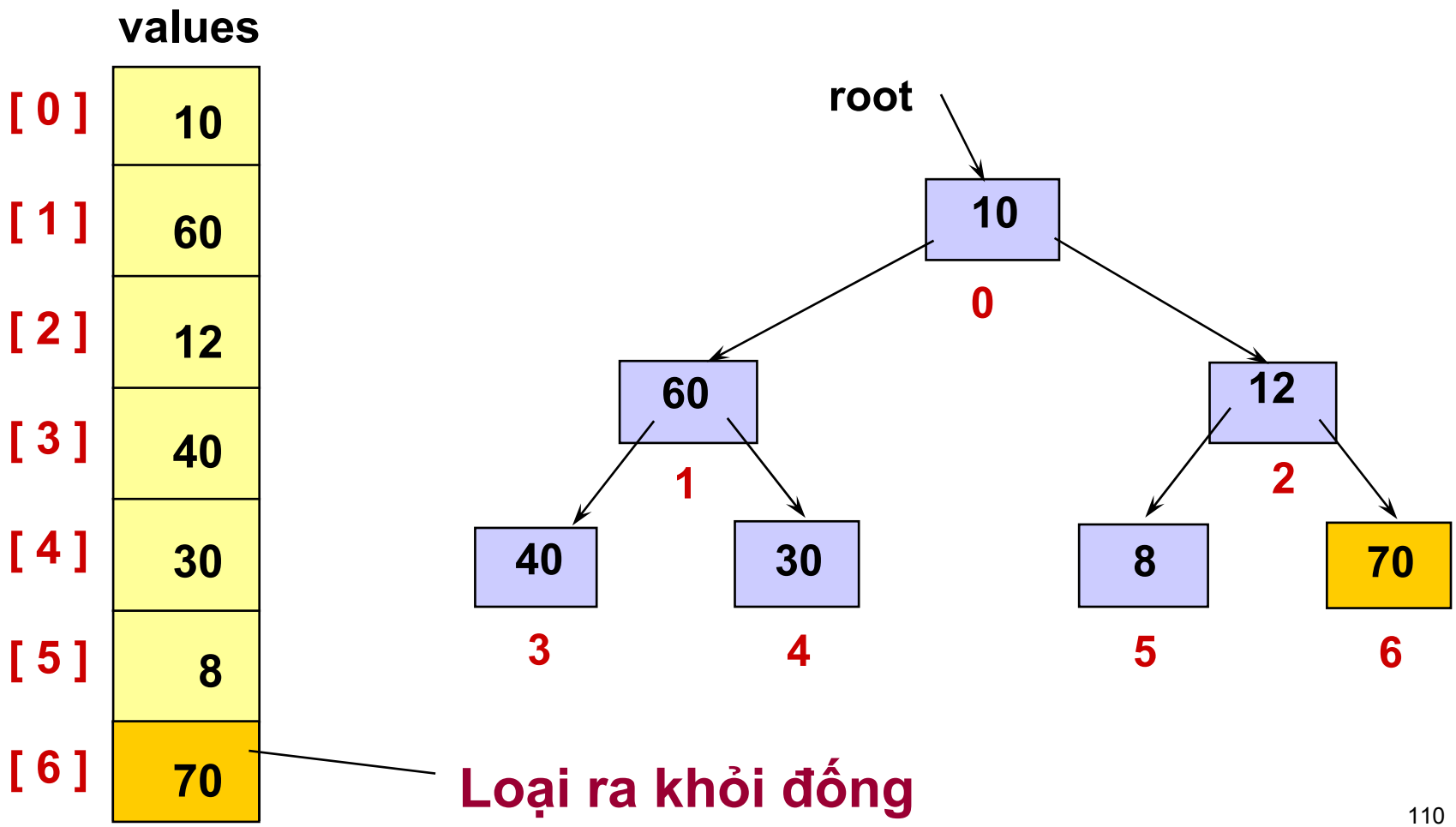
Sau khi tạo đống



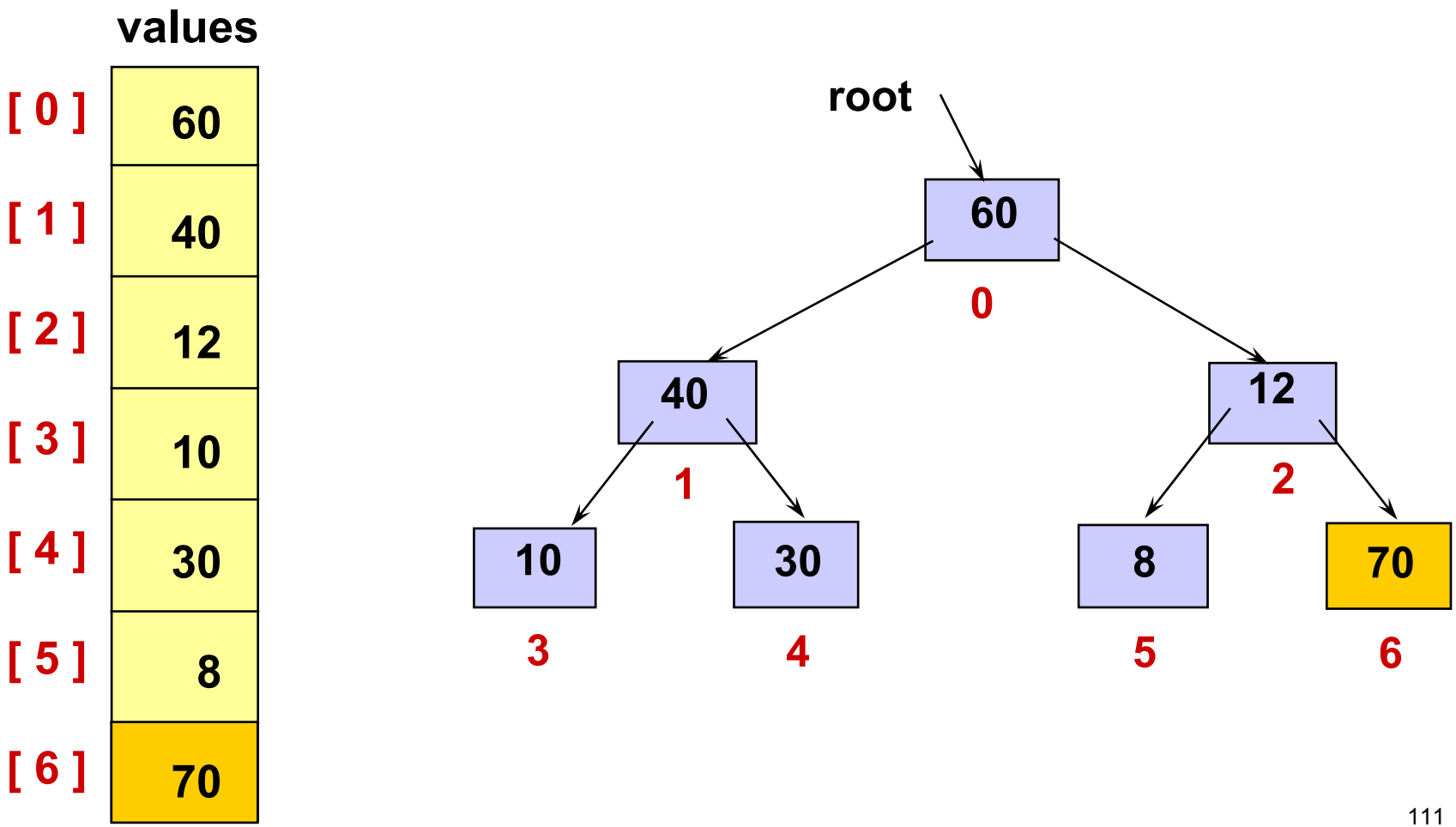
Đổi chỗ phần tử đỉnh và phần tử đáy



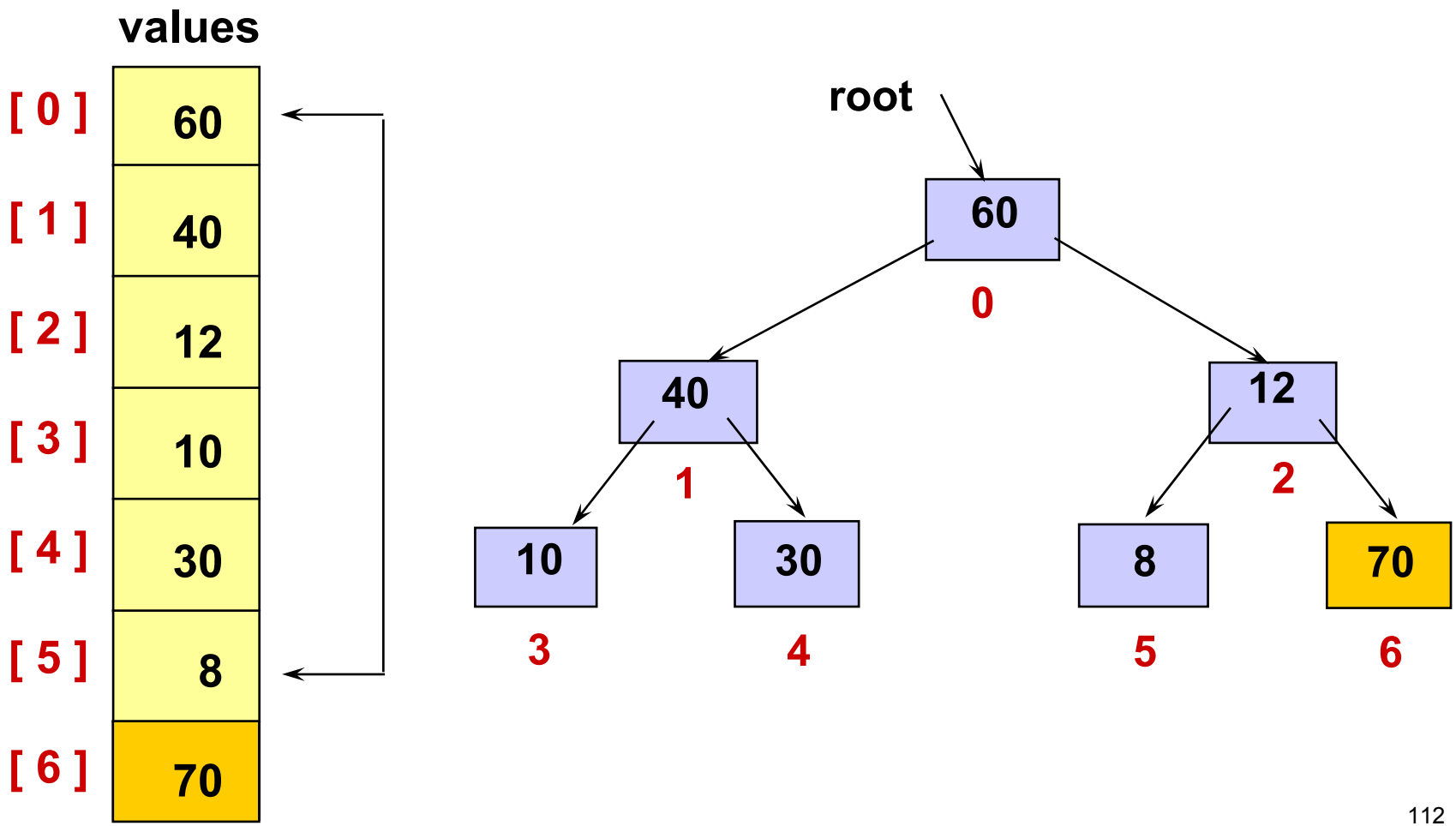
Sau khi đổi chỗ



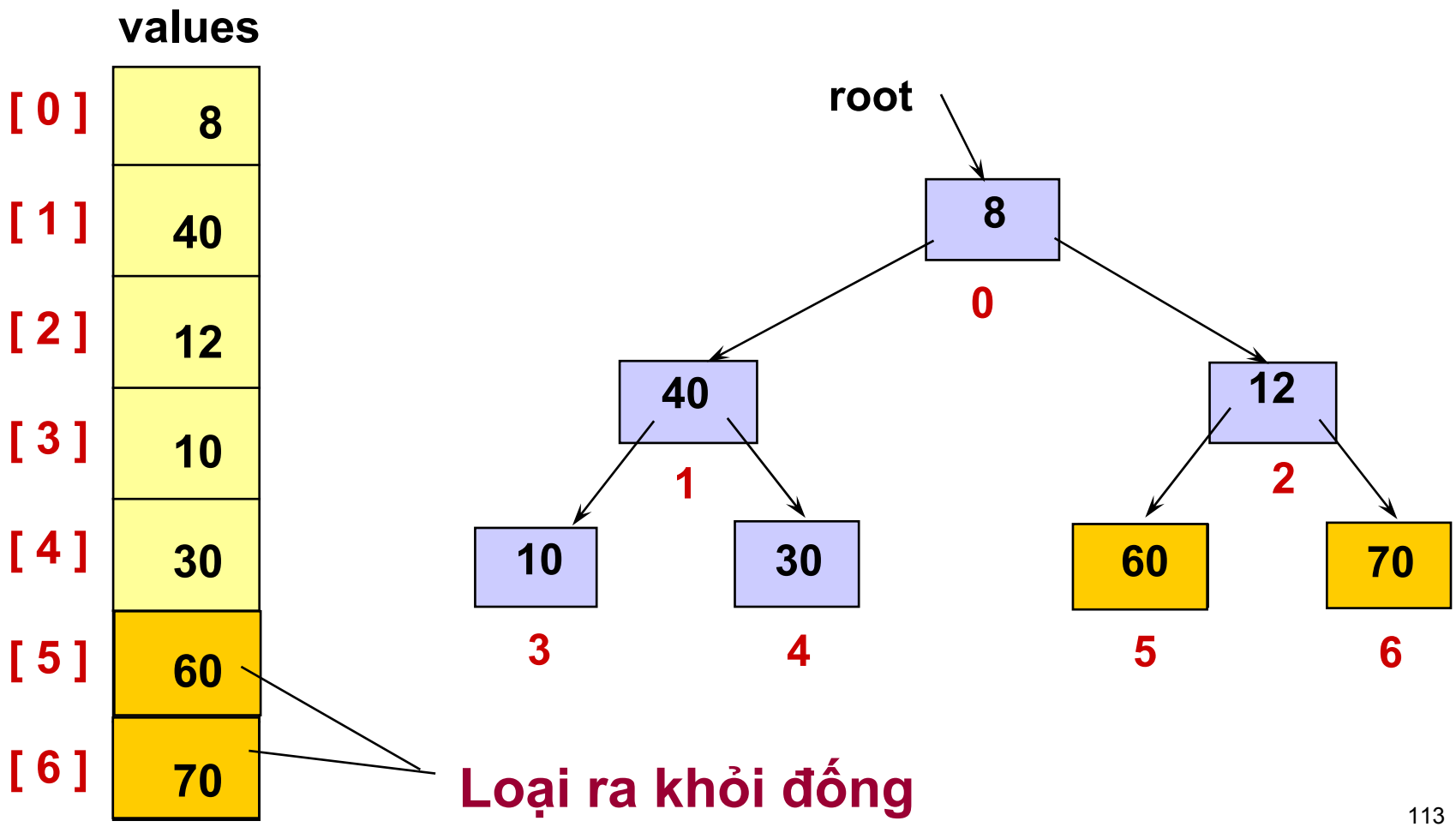
Vun lại đống


values
[0]
60
[1]
40
[2]
12
[3]
10
[4]
30
[5]
8
[6]
70
root
60
0
40
1
12
2
10
3
30
4
8
5
70
6

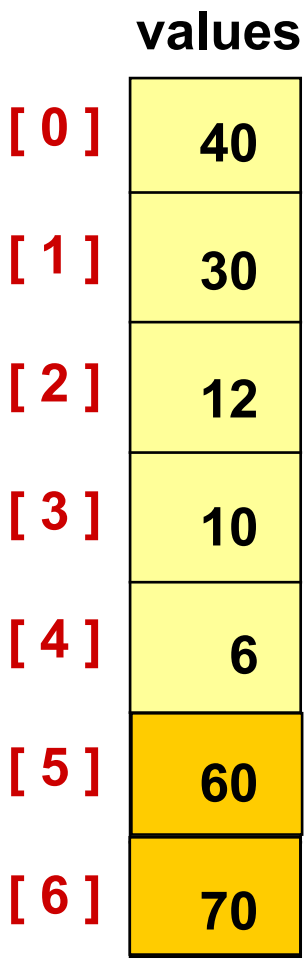
Đổi chỗ phần tử đỉnh

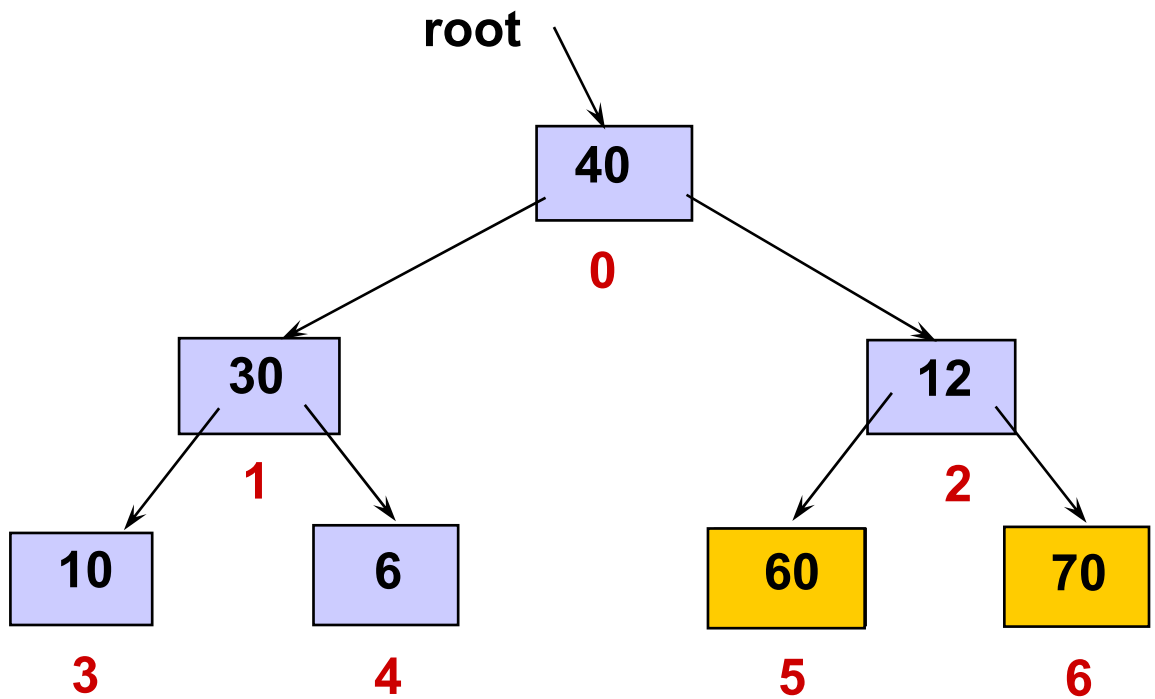


Sau khi đổi chỗ

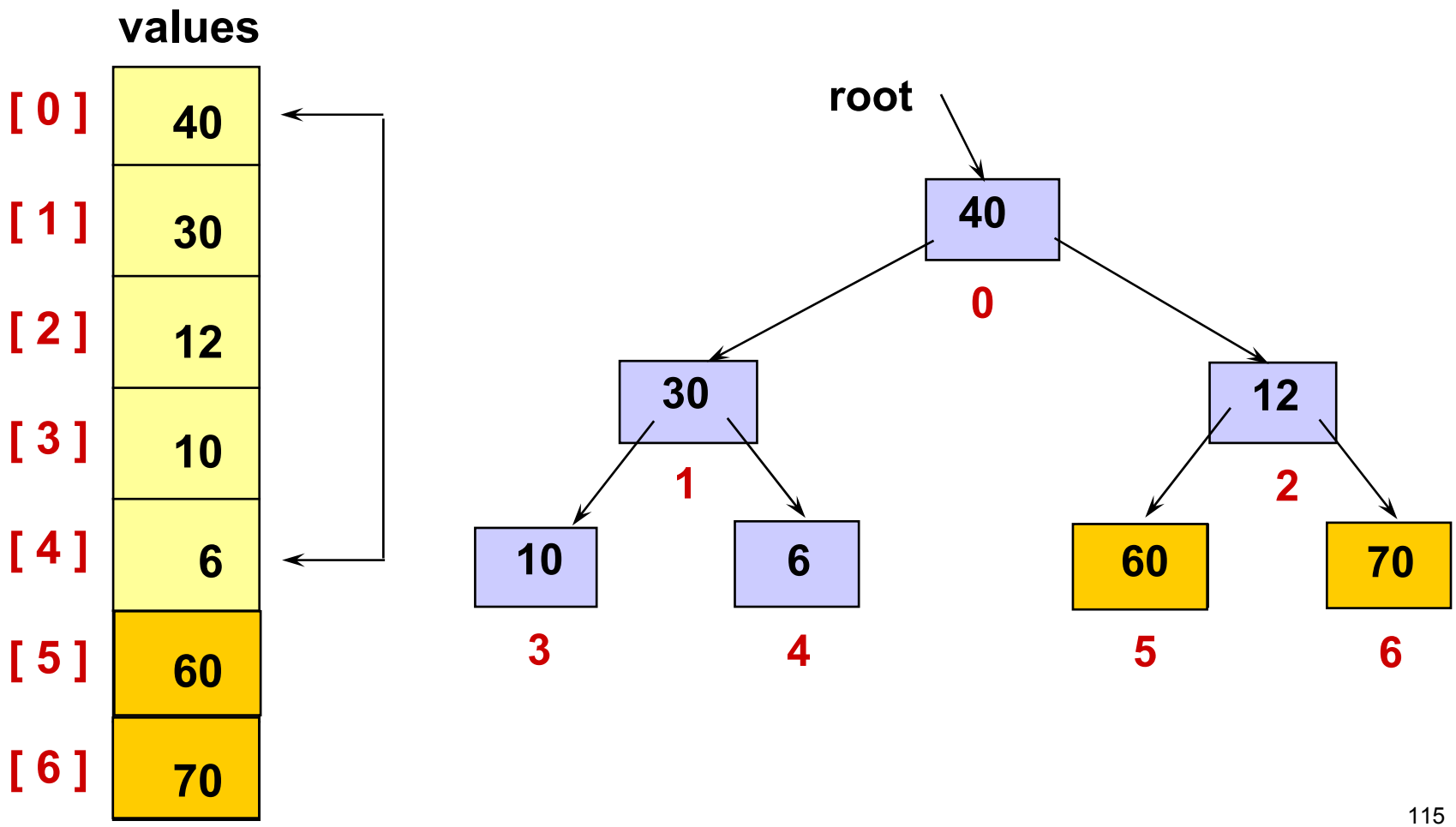


Vun lại đống

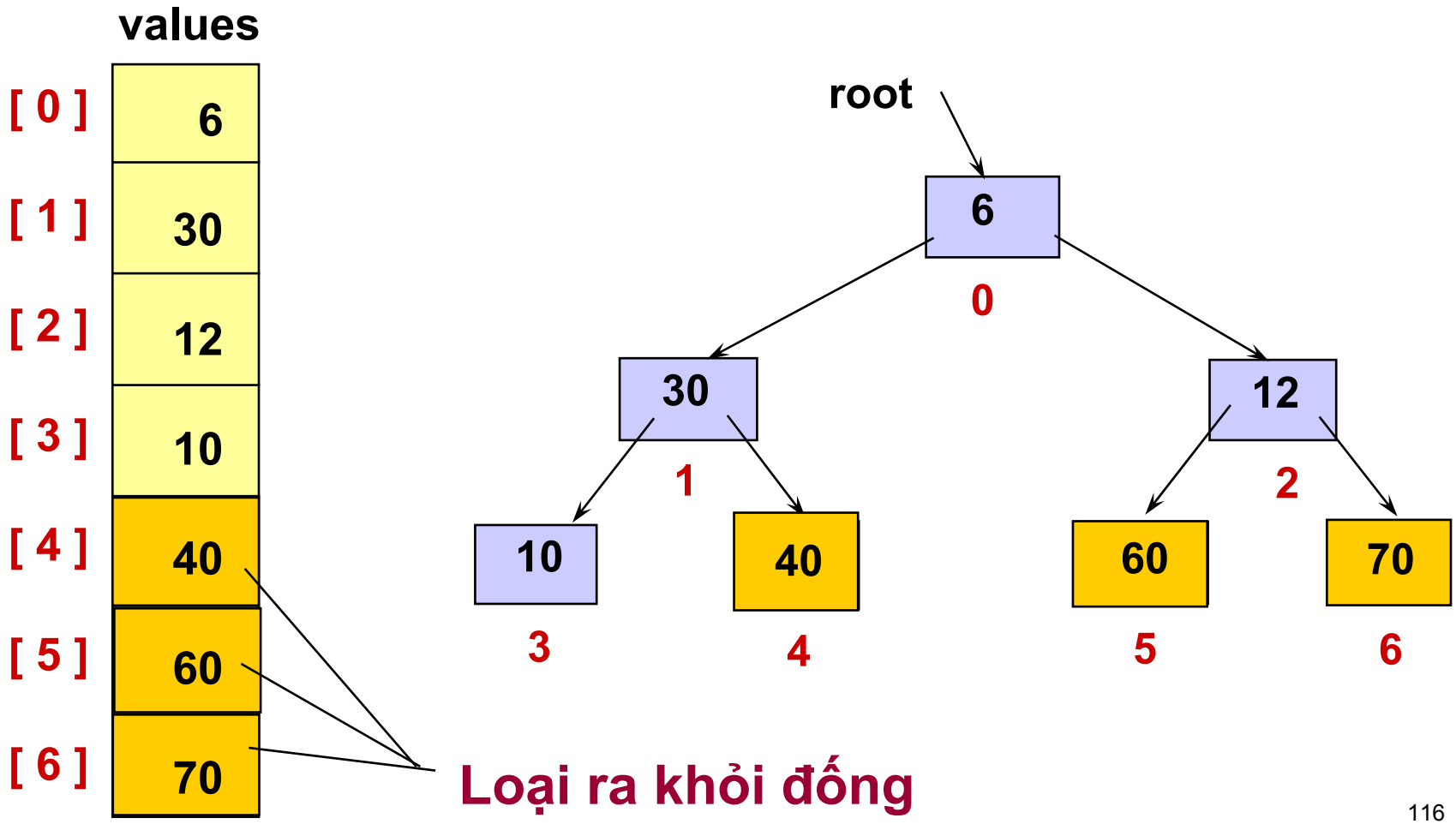

values
[0]
40
[1]
30
[2]
12
[3]
10
[4]
6
[5]
60
[6]
70


root
40
0
30
1
12
2
10
3
6
4
60
5
70
6

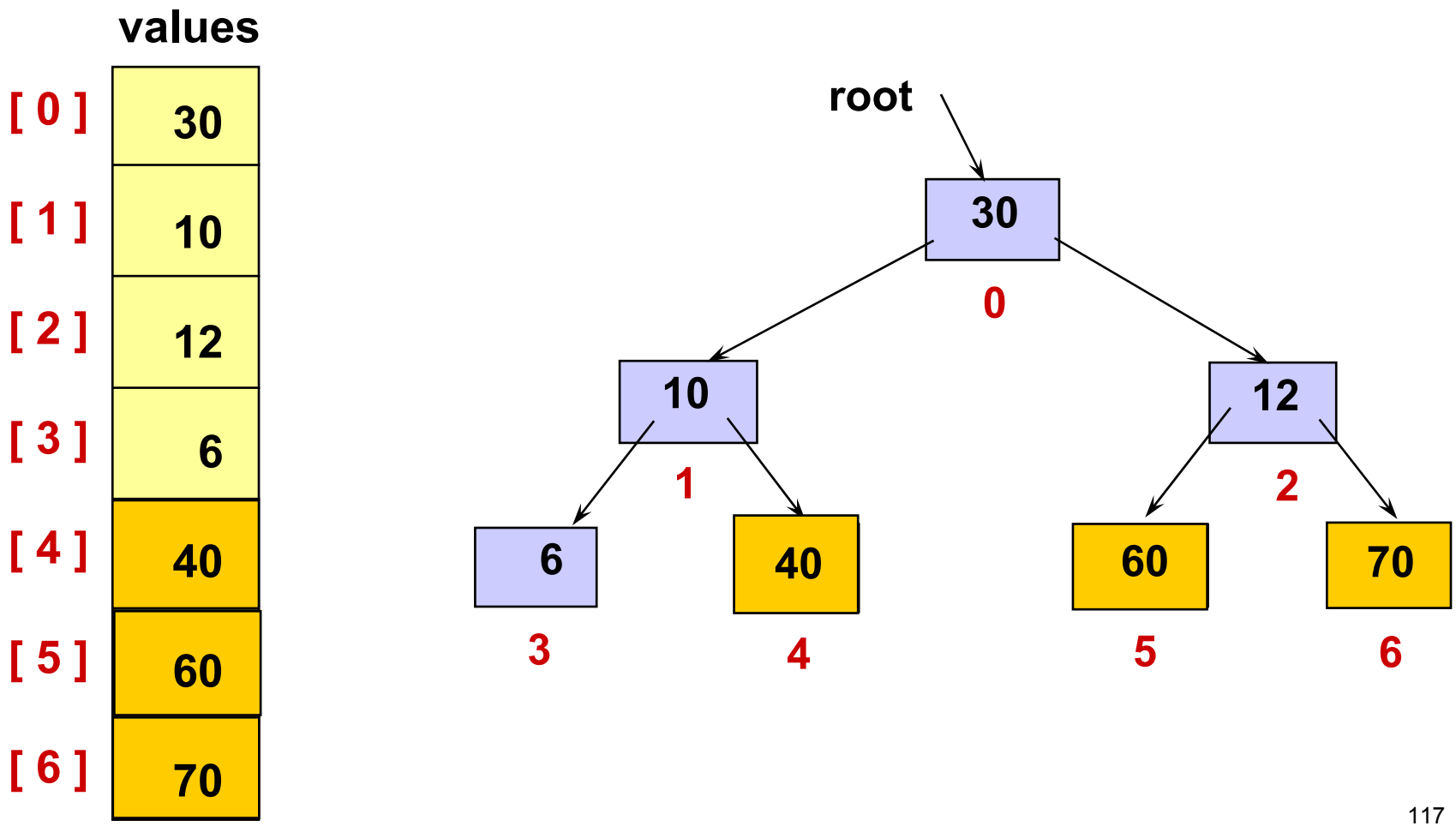
Đổi chỗ phần tử đỉnh



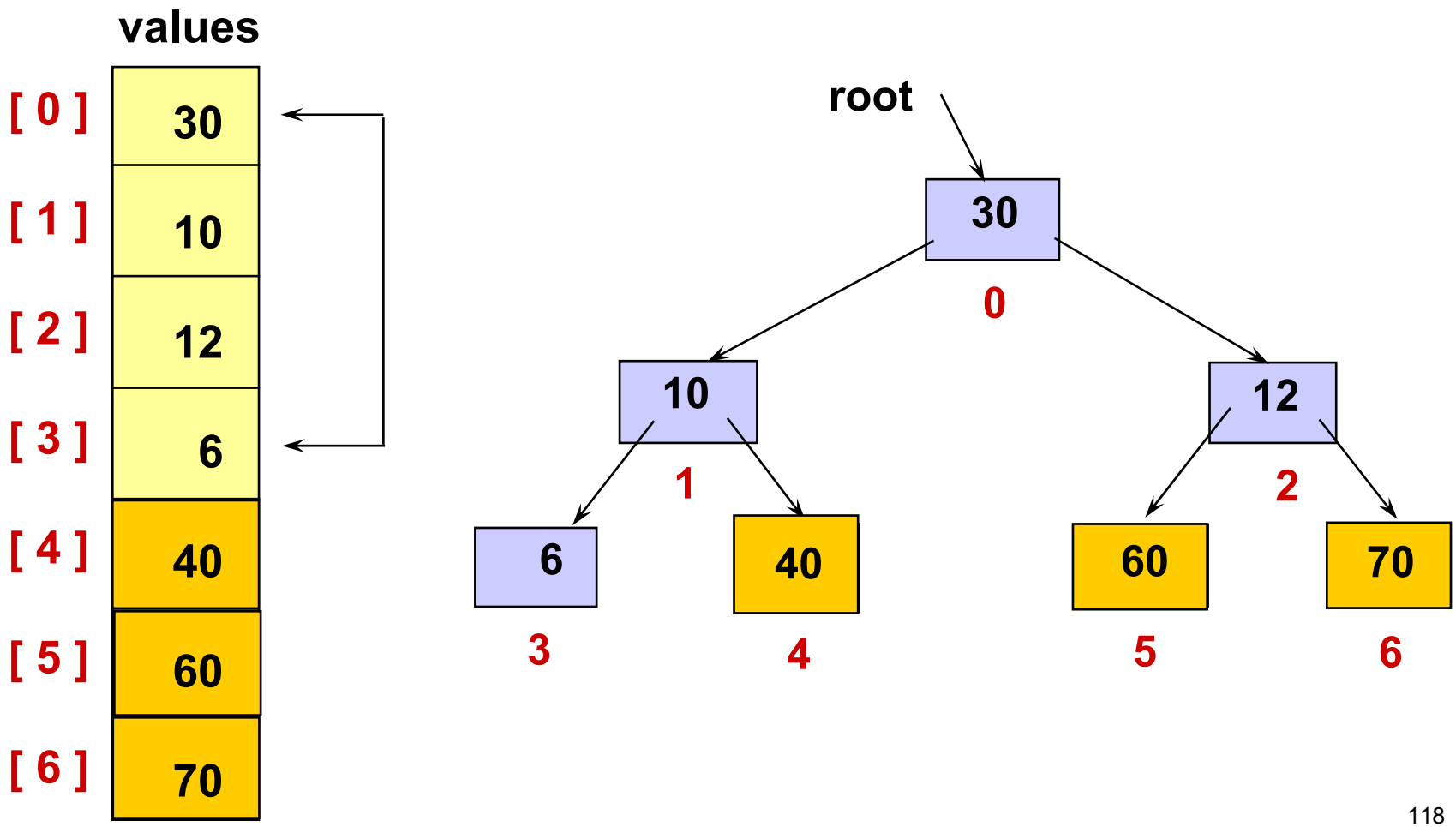
Sau khi đổi chỗ



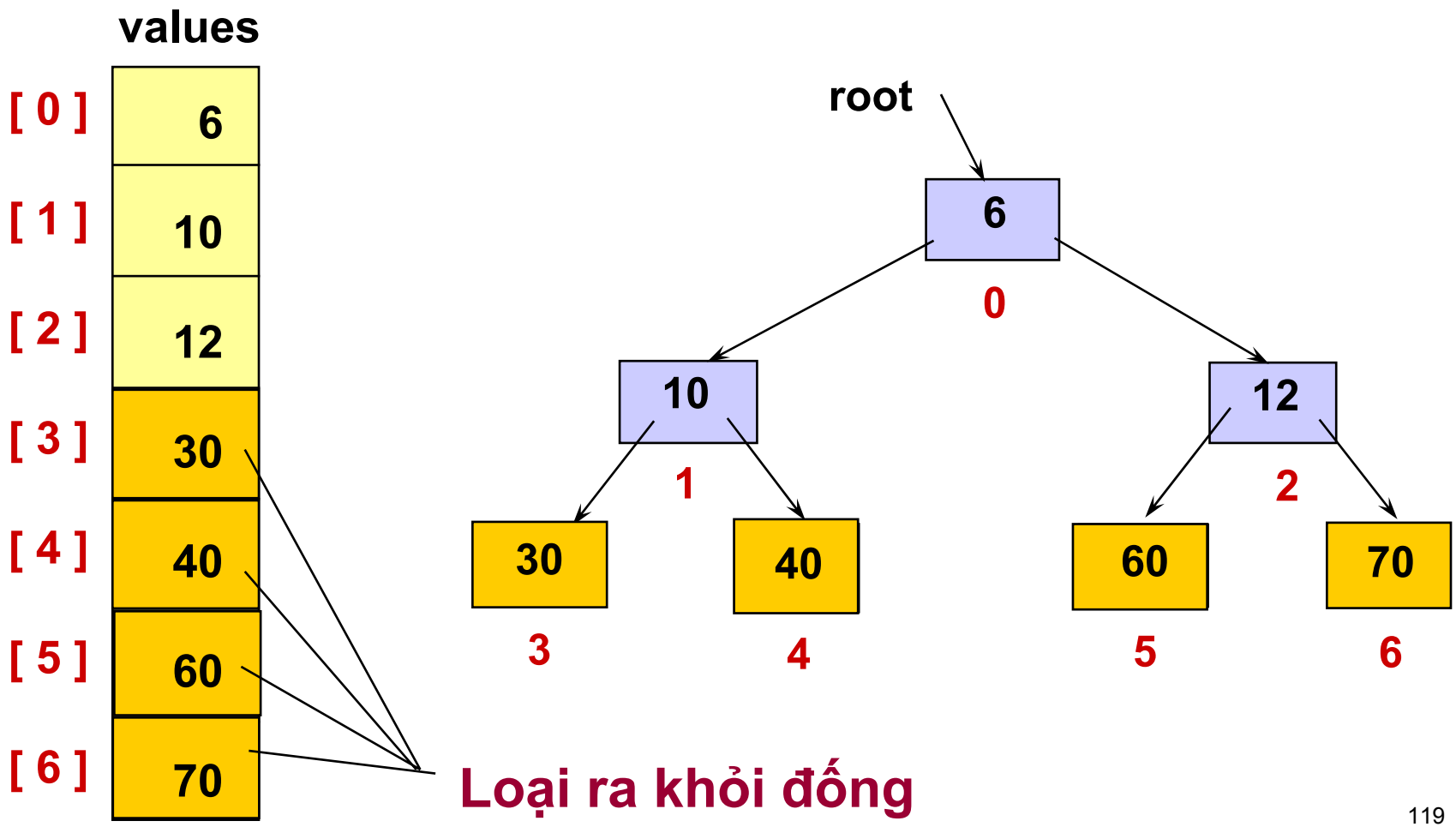
Vun lại đống


values
[0]
30
[1]
10
[2]
12
[3]
6
[4]
40
[5]
60
[6]
70
root
30
0
10
1
12
2
6
3
40
4
60
5
70
6

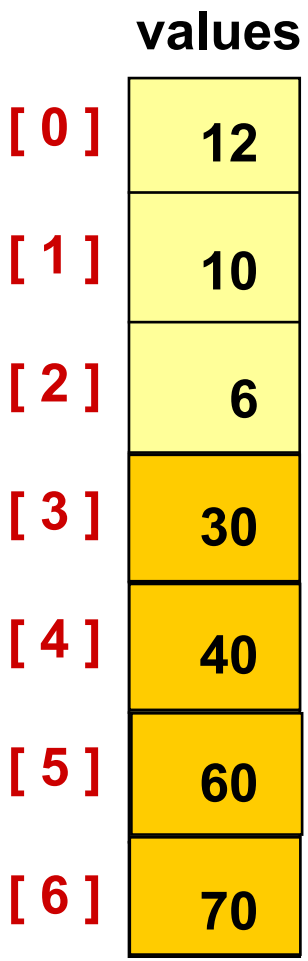
Đổi chỗ phần tử đỉnh

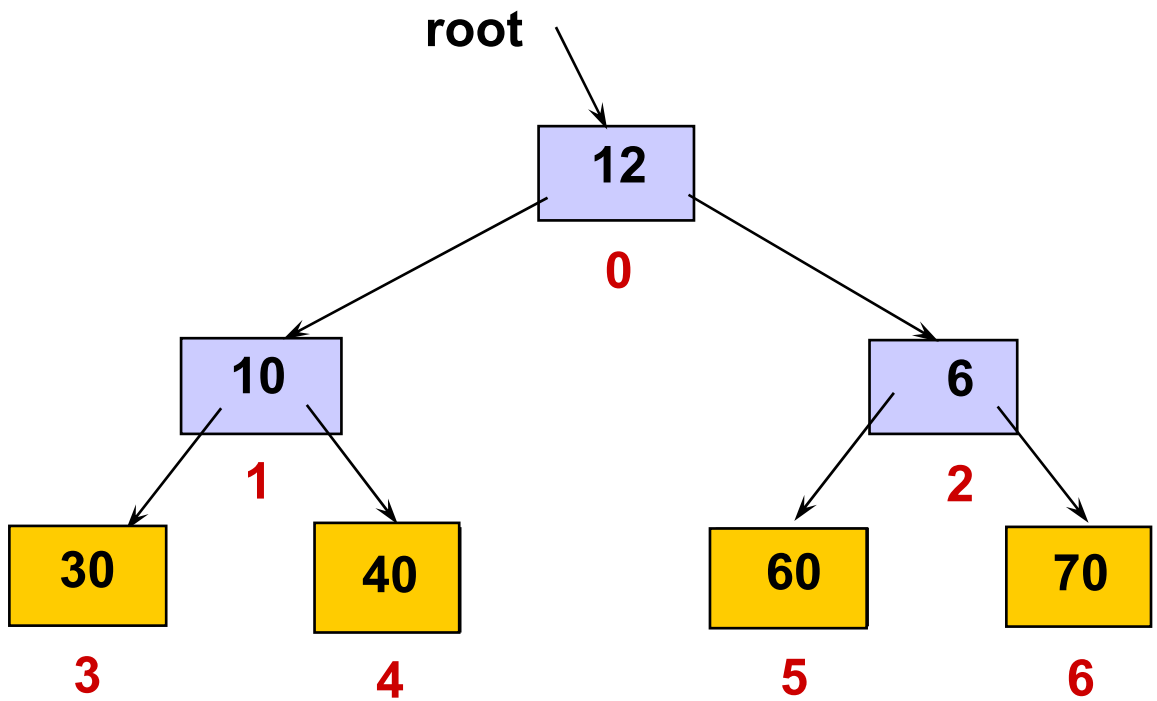


Sau khi đổi chỗ

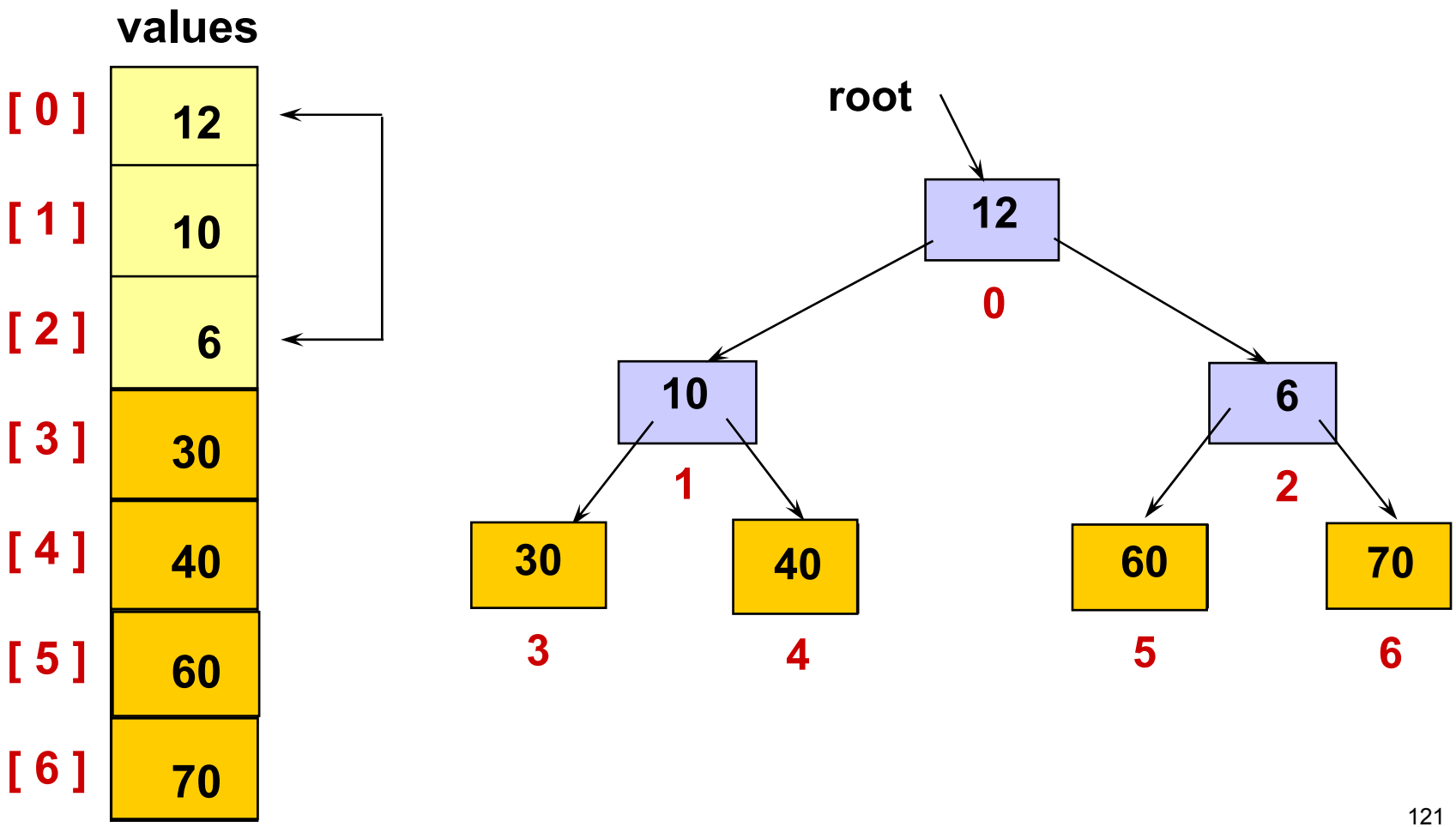


Vun lại đống

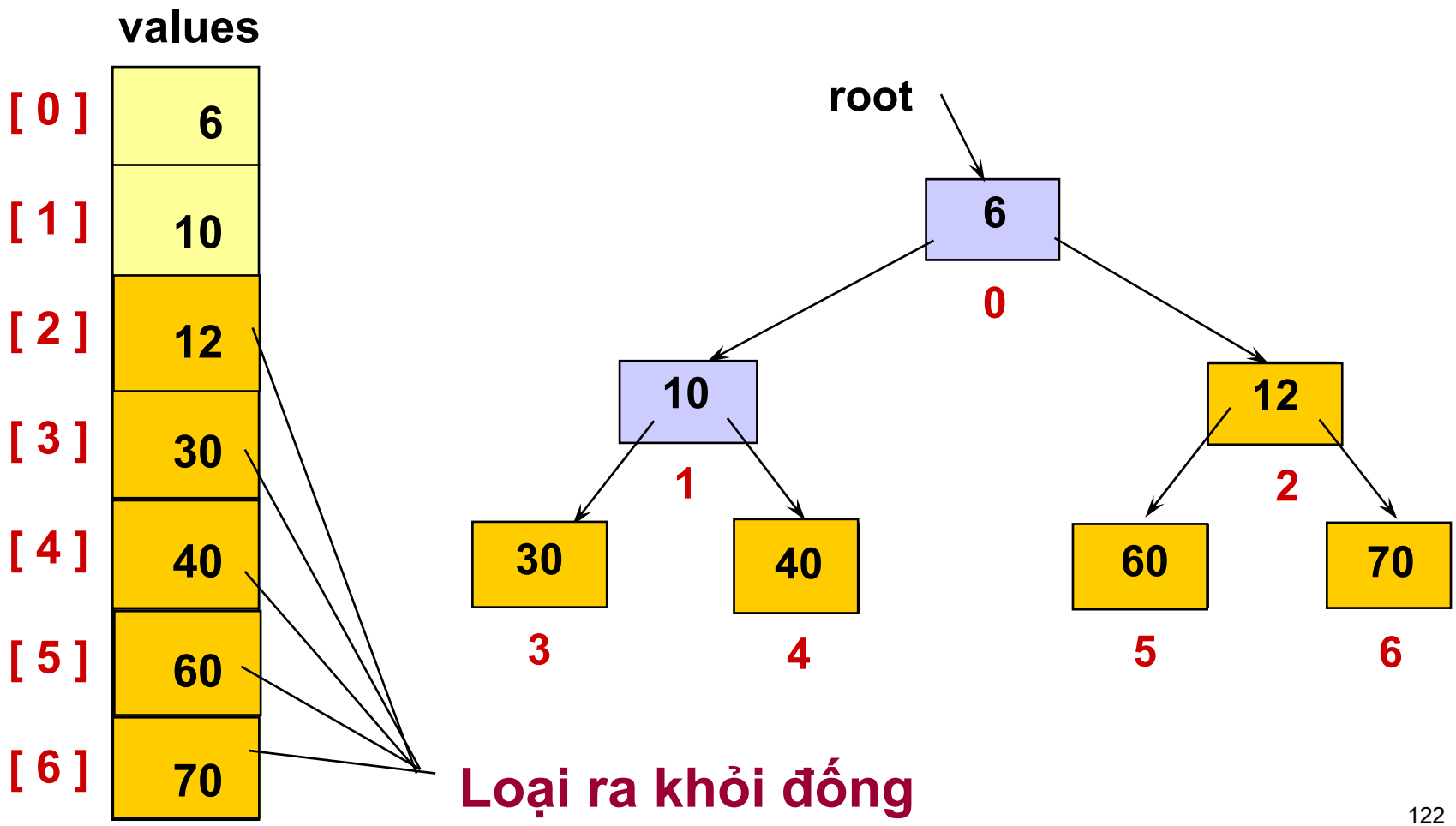

values
[0]
12
[1]
10
[2]
6
[3]
30
[4]
40
[5]
60
[6]
70


root
12
0
10
1
6
2
30
3
40
4
60
5
70
6

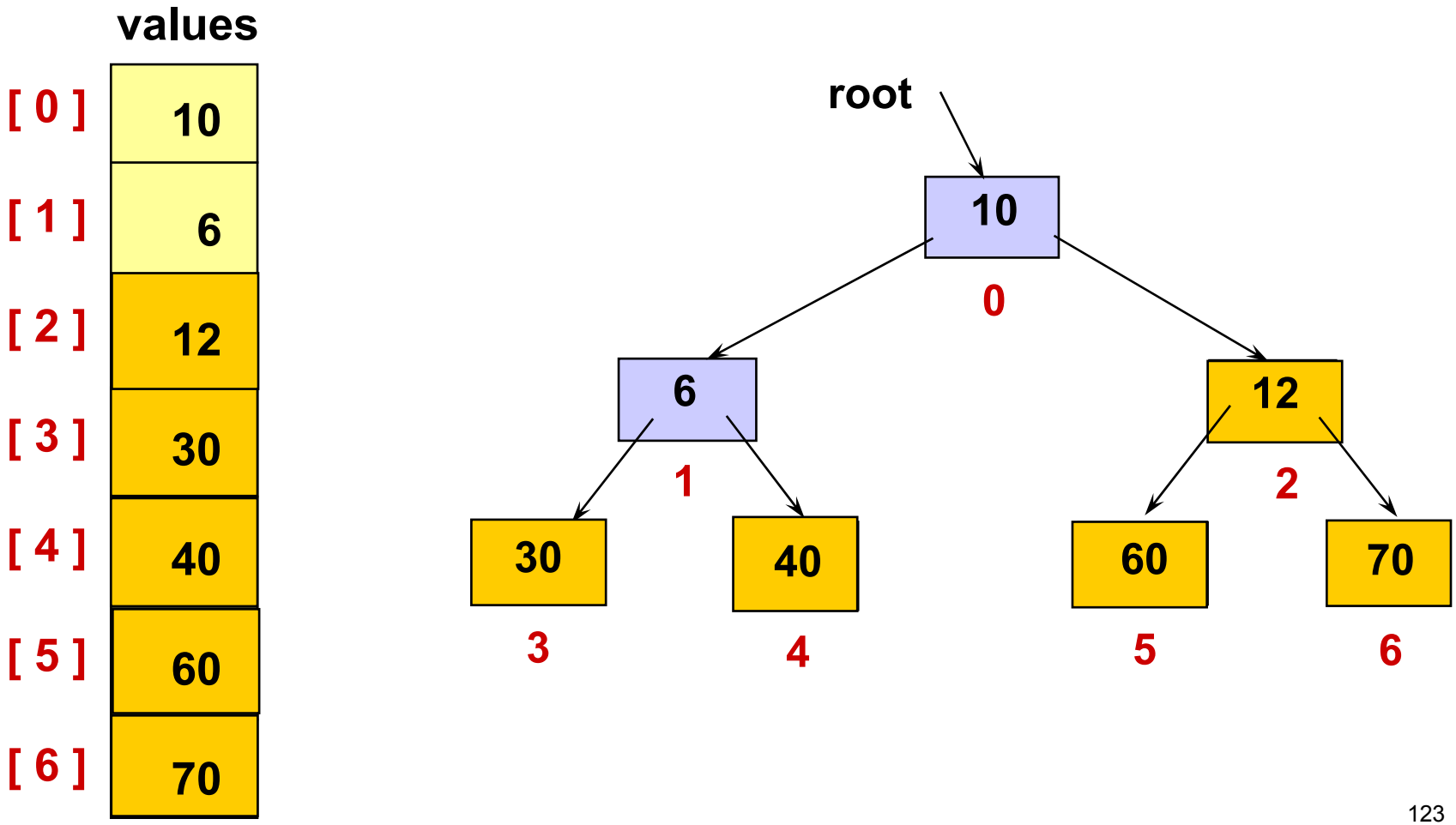
Đổi chỗ


values
[0]
12
[1]
10
[2]
6
[3]
30
[4]
40
[5]
60
[6]
70
root
12
0
10
1
6
2
30
3
40
4
60
5
70
6

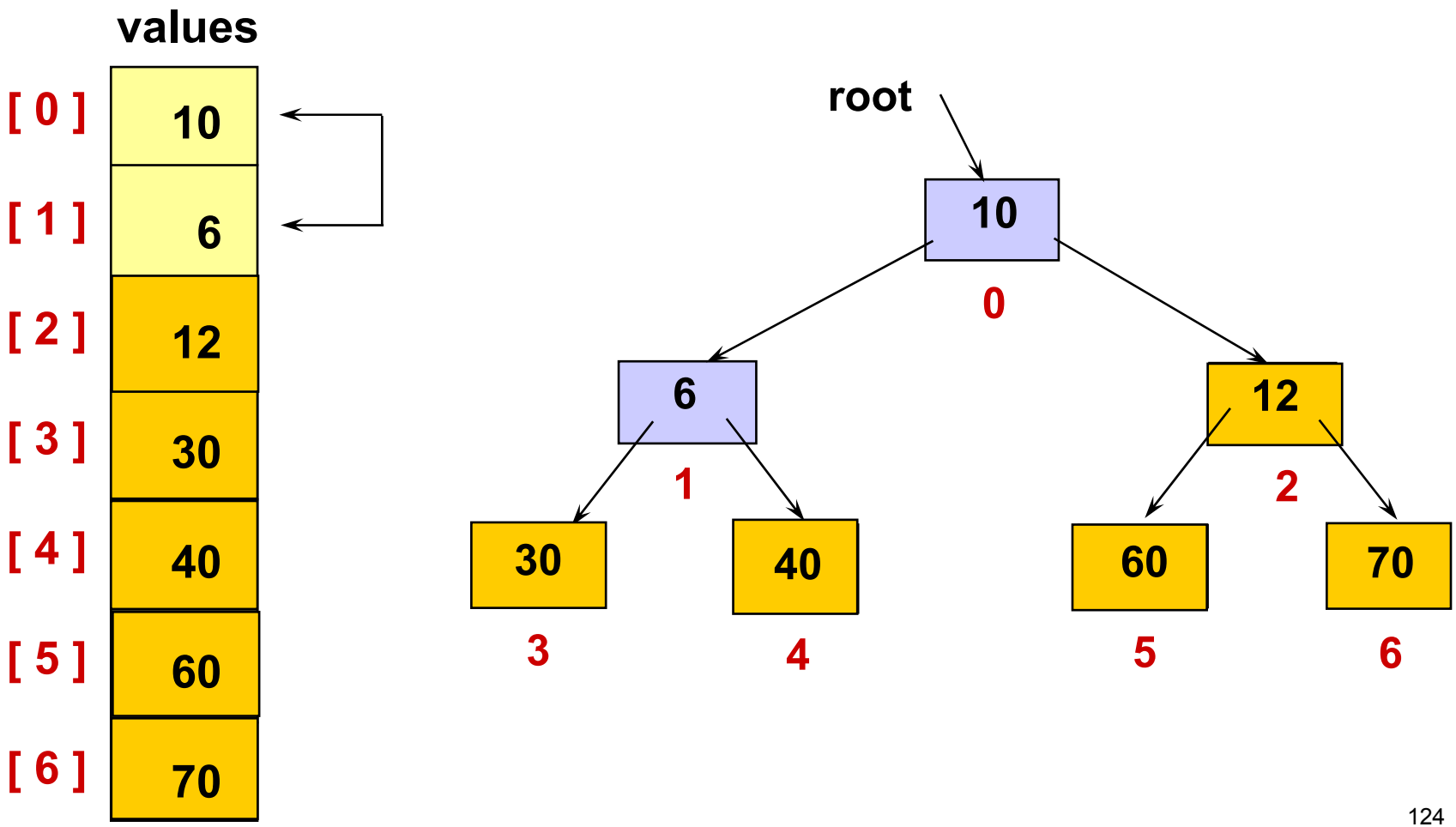
Sau khi đổi chỗ



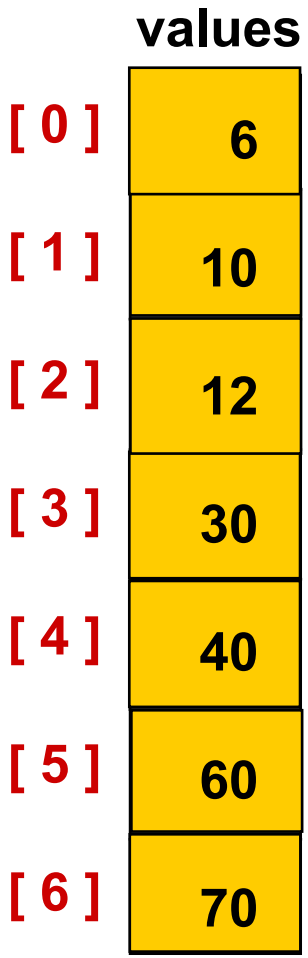
Vun lại đống

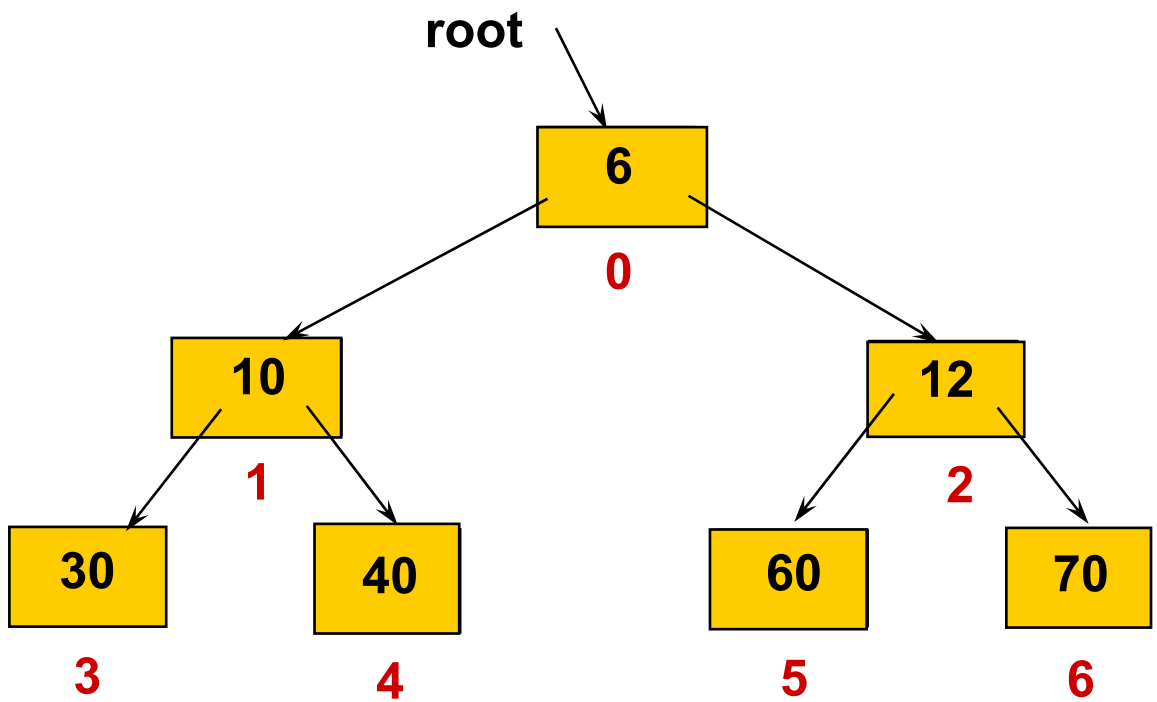

values
[0]
10
[1]
6
[2]
12
[3]
30
[4]
40
[5]
60
[6]
70
root
10
0
6
1
12
2
30
3
40
4
60
5
70
6

Đổi chỗ


values
[0]
10
[1]
6
[2]
12
[3]
30
[4]
40
[5]
60
[6]
70
root
10
0
6
1
12
2
30
3
40
4
60
5
70
6

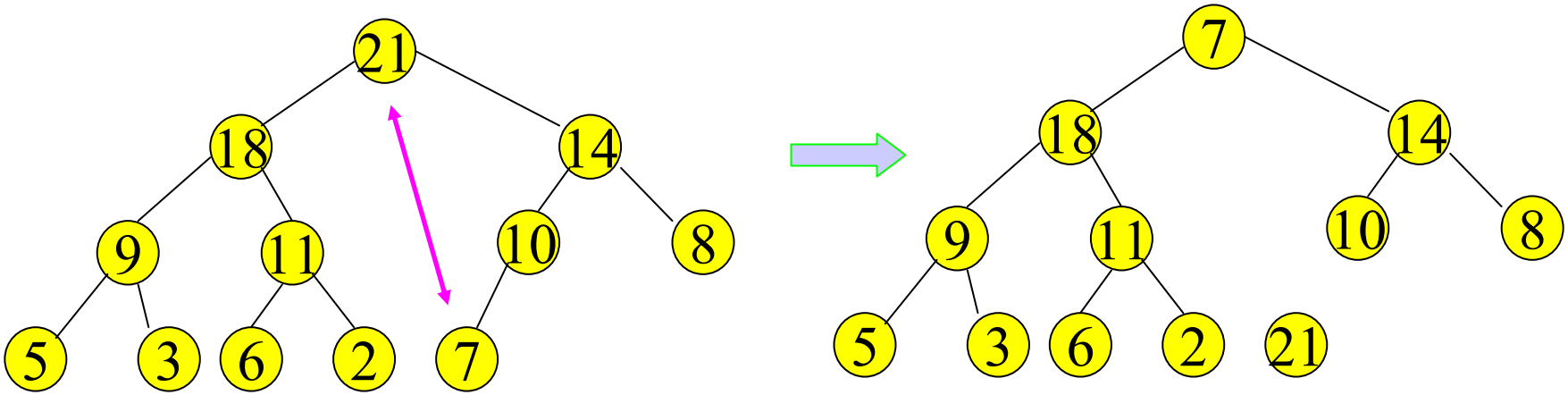
Sau khi đổi chỗ



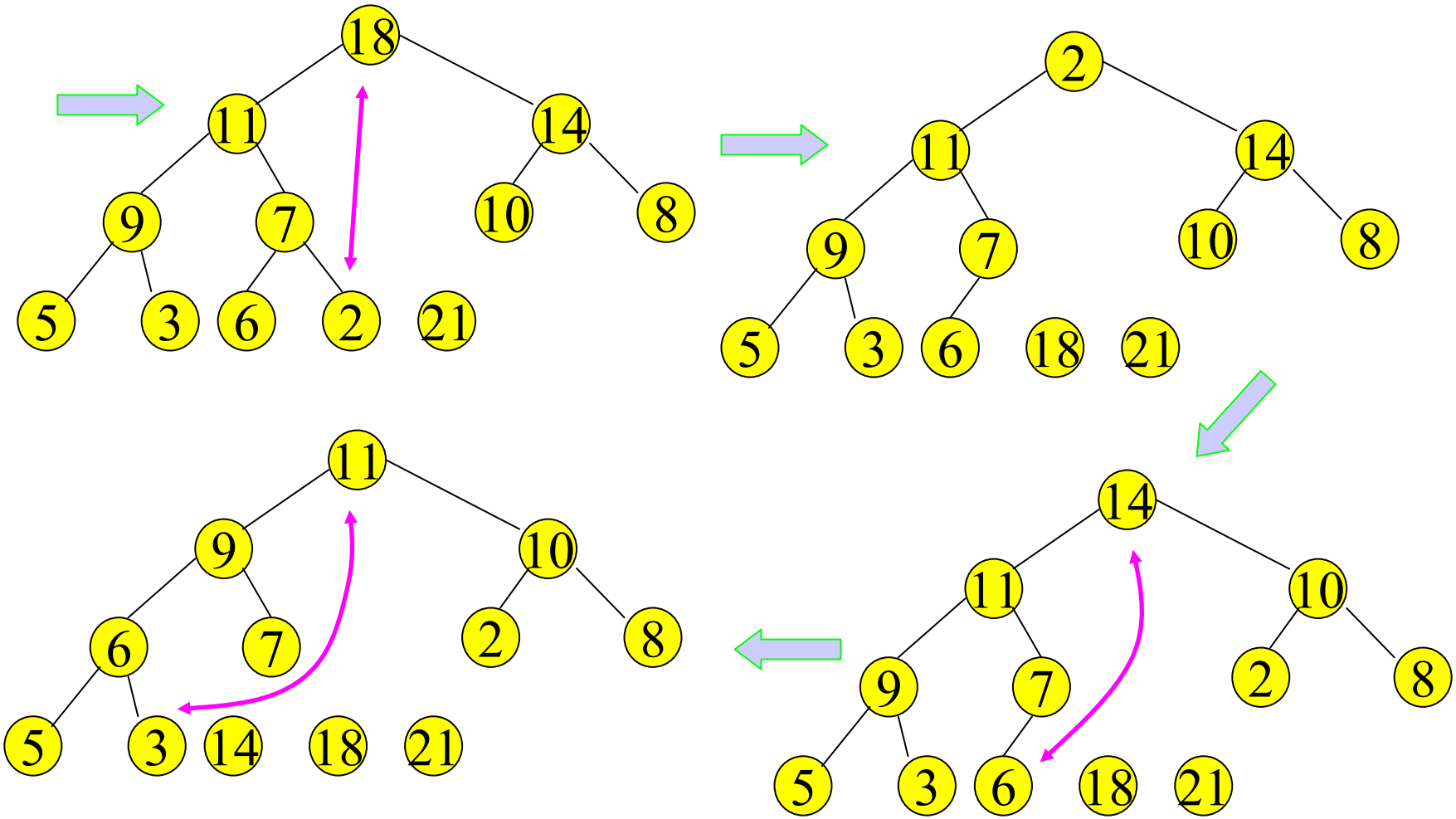


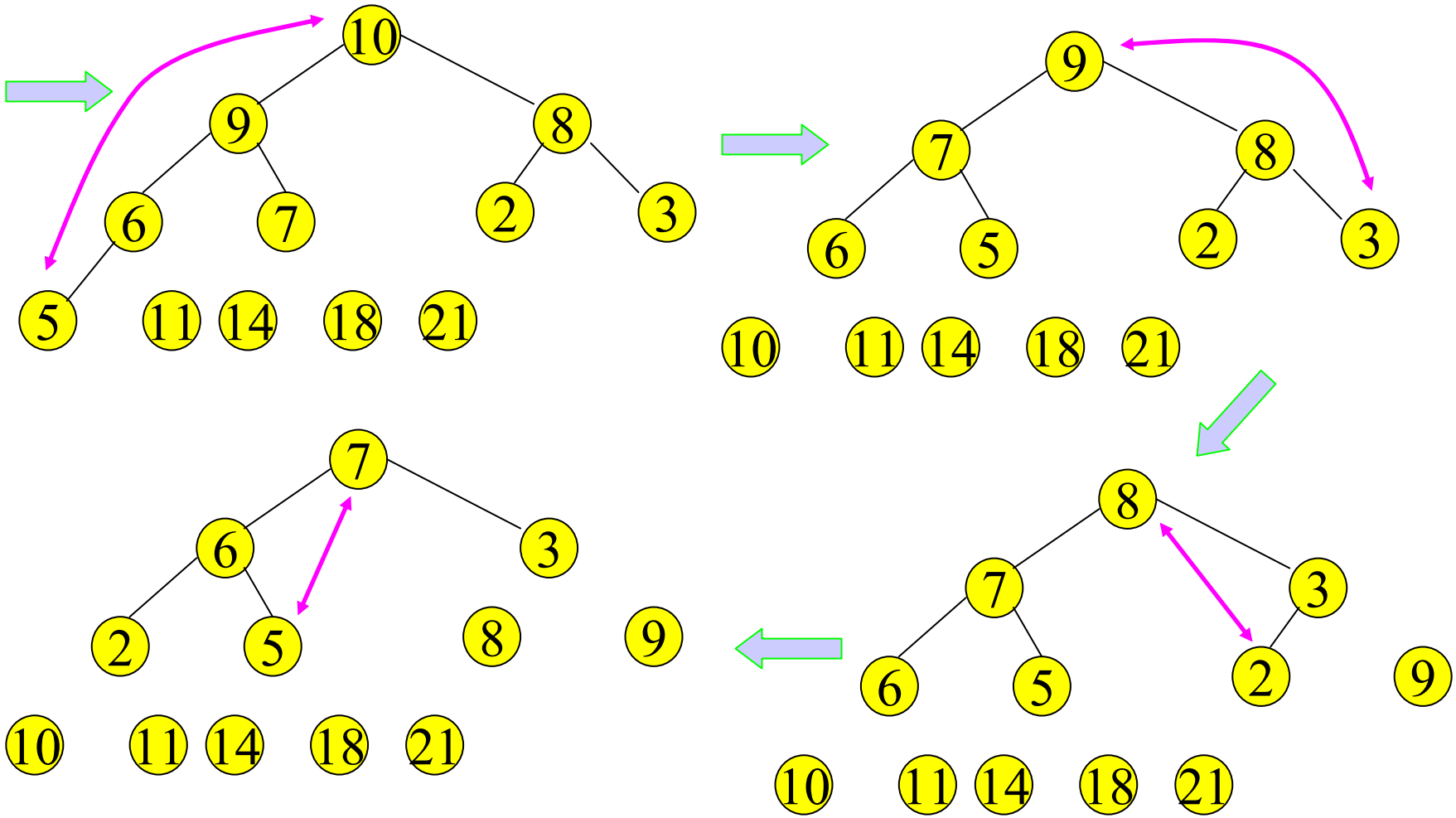
Tất cả các phần tử đã được sắp xếp

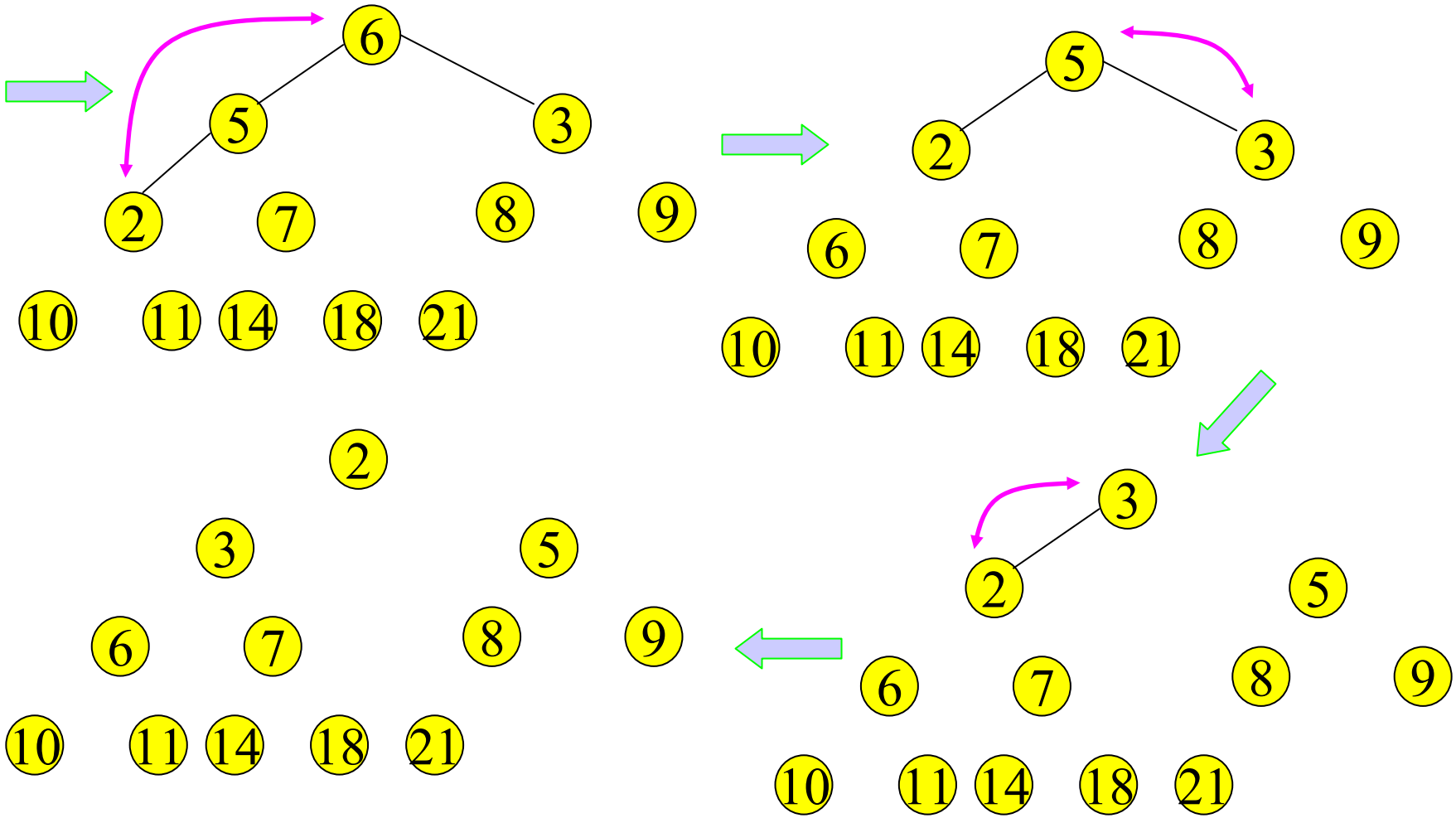
Heapsort



Heapsort


18
11
14
9
7
10
8
5
3
6
2
21
2
11
14
9
7
10
8
5
3
6
18
21
11
9
10
6
7
2
8
5
3
14
18
21
14
11
10
9
7
2
8
5
3
6
18
21


10
9
8
6
7
2
3
5
11
14
18
21
9
7
8
6
5
2
3
10
11
14
18
21
8
7
3
6
5
2
9
10
11
14
18
21
7
6
3
2
5
8
9
10
11
14
18
21



2, 3, 5, 6, 7, 8, 9, 10, 11, 14, 18, 21

Độ phức tạp về thời gian

(1) Tạo đống:

- N/2 * O(log N) $\Rightarrow$ O(N log N)

(2) Thực hiện N-1 lần thao tác xóa phần tử đỉnh

- Mỗi lần xóa mất O(log N) $\Rightarrow$ O(N log N)

- Độ phức tạp thời gian: O(N log N)