Cấu Trúc Dữ Liệu I

Khoa Toán – Tin học

Email: ptbao@mathdep.hcmuns.edu.vn

Nội dung chương trình

1. Tìm kiếm và sắp xếp (Search & Sort)
2. CTDL Danh sách liên kết (Linked List)
3. CTDL Cây (Tree)
4. Nội dung thêm

Tài liệu tham khảo

1. Data Structures and Algorithm Analysis in C, Mark Allen Weiss, Addison Wesley Longman Inc. – 1997.
2. Nhập môn Cấu Trúc Dữ Liệu và Thuật Toán, TS. Dương Anh Đức, ĐHQG Tp.HCM
3. Data Structures and Program Design in C++, Robert L. Kruse and Alexander J. Ryba, Prentice – Hall International Inc. – 1999.
4. Data Structures and C Programs, Christopher J. Van Wyk, Addison Wesley – 1992.
5. Cấu Trúc Dữ Liệu + Thuật Giải = Chương Trình, người dịch Nguyễn Quốc Cường, nhà xuất bản Đại học và Giáo dục chuyên nghiệp.
6. Internet

Tìm kiếm và sắp xếp

- 02 thuật toán tìm kiếm
- 11 thuật toán sắp xếp

Cấu trúc dữ liệu DSLK

- DSLK đơn
- DSLK đôi, đa
- Ngăn xếp (Stack)
- Hàng đợi (Queue)
- DSLK có thứ tự
- DSLK vòng

Cấu trúc dữ liệu cây

- Cây tổng quát
- Cây nhị phân
- Cây nhị phân tìm kiếm

Nội dung thêm

- Cây AVL (AVL Tree)
- Cây đỏ đen (Red-Black Tree)
- Bảng băm (Hashing)
- Tìm kiếm chuỗi (Pattern Matching)

Bài thực hành

1. Thuật toán tìm kiếm, sắp xếp
2. Mô phỏng hàng đợi mua vé xem phim
3. Thuật toán Balan ngược (Stack - DSLK) để tính giá trị biểu thức toán học.
4. Cộng, trừ, nhân, chia, luỹ thừa hai số nguyên lớn (DSLK)
5. Tính giao, hội, hiệu, phần bù của 02 tập hợp (DSLK)
6. Tính cộng, trừ, nhân, chia, giá trị 02 đa thức (DSLK)
7. Dùng DSLK cài đặt ma trận thưa, tính tổng, hiệu, nhân, nghịch đảo (nếu có) của ma trận.
8. Dùng cây nhị phân tìm kiếm để quản lý các điểm trong 2D: tìm kiếm, thêm, bớt, tạo cây, hủy cây, lưu dữ liệu xuống file, đọc dữ liệu từ file lên cây.

Chú ý: ***chỉ chọn 01 trong 05 bài: 03, 04, 05, 06, 07***

Tổng quan

- Khái niệm thuật giải
- Phân tích thuật giải
- Trừu tượng hoá dữ liệu

Khái niệm thuật giải

- Thuật giải là gì ?
- Tính chất của thuật giải:
  1. Dữ liệu
  2. Tính xác định
  3. Tính dừng
  4. Tính đúng đắn
  5. Tính khả thi

Khái niệm thuật giải (tt)

- Tất cả các bài toán đều giải được trên máy tính ?
- Có bao nhiêu thuật giải ?
- Mục tiêu của thuật giải là gì ?
- Phải chọn thuật giải nào ?

Phân tích thuật giải

- Thời gian thực hiện (Running Time)
- Mã giả (Pseudo-Code)
- Phân tích thuật giải

Thời gian thực hiện (Running Time / Time Complexity)

- Thời gian thực hiện phụ thuộc gì ?
- Vậy làm sao đánh giá thuật giải ?
- Làm thế nào xác định thời gian thực hiện ?

Hướng tiếp cận thực nghiệm

- Các bước thực hiện:
  1. Viết chương trình cài đặt
  2. Thực thi chương trình với nhiều bộ dữ liệu
  3. Đo và thống kê thời gian
  4. Xấp xỉ biểu đồ
- Hạn chế:
  1. Cần phải cài đặt CT và đo thời gian
  2. Bộ dữ liệu không thể đặc trưng hết
  3. Khó so sánh 02 thuật giải

Mã giả

- Mã giả là gì ?
- Đặc tính cơ bản:
  1. Mô tả thuật giải có cấu trúc hơn NNTN
  2. Kém hình thức hơn NNLT
  3. Công cụ chủ yếu mô tả thuật giải
  4. Tàng ẩn dưới khái niệm thiết kế CT

Mã giả (tt)

- Một số quy ước:
  1. Các biểu thức toán học
  2. Lệnh gán: "←"
  3. So sánh: "="
  4. Khai báo hàm (thuật giải)

     Thuật giải <tên TG> (<tham số>)

     Input: <dữ liệu vào>

     Output: <dữ liệu ra>

     <Các câu lệnh>

     End <tên TG>

Mã giả (tt)

5. Các cấu trúc:
   - Cấu trúc chọn:
     If ... then ... [else ...]
   - Vòng lặp:
     While ... do
     Do ... while (...)
     For ... do ...
6. Một số câu lệnh khác:
   - Trả giá trị về: Return [*giá trị*]
   - Lời gọi hàm: <Tên>(tham số)

Phân tích thuật giải

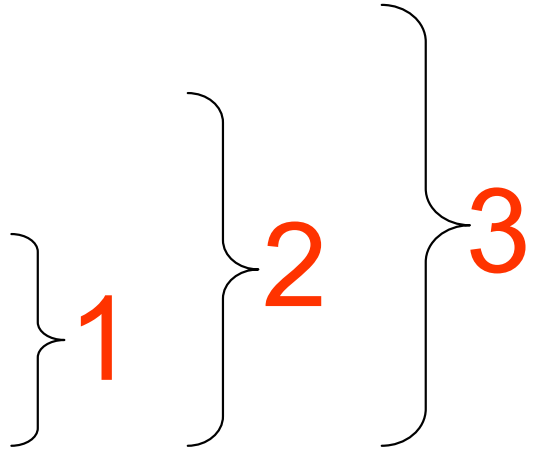
- Khái niệm độ phức tạp của thuật toán
- Các thao tác cơ sở
- Khảo sát mã giả
- Ví dụ

Thuật giải MaxOfArray(A,n)

Input: Mảng A có n phần tử

Output: Phần tử lớn nhất của A

```
currentMax ← A[0]
for (i ← 1 to n-1 ) do
        if(currentMax < A[i])
                currentMax ← A[i]
return currentMax
```



End MaxOfArray

TT	Thao tác	Tốt nhất	Xấu nhất	TB
1	So sánh	n-1	n-1	n-1
2	Gán	1	n	n/2
3	Return	1	1	1

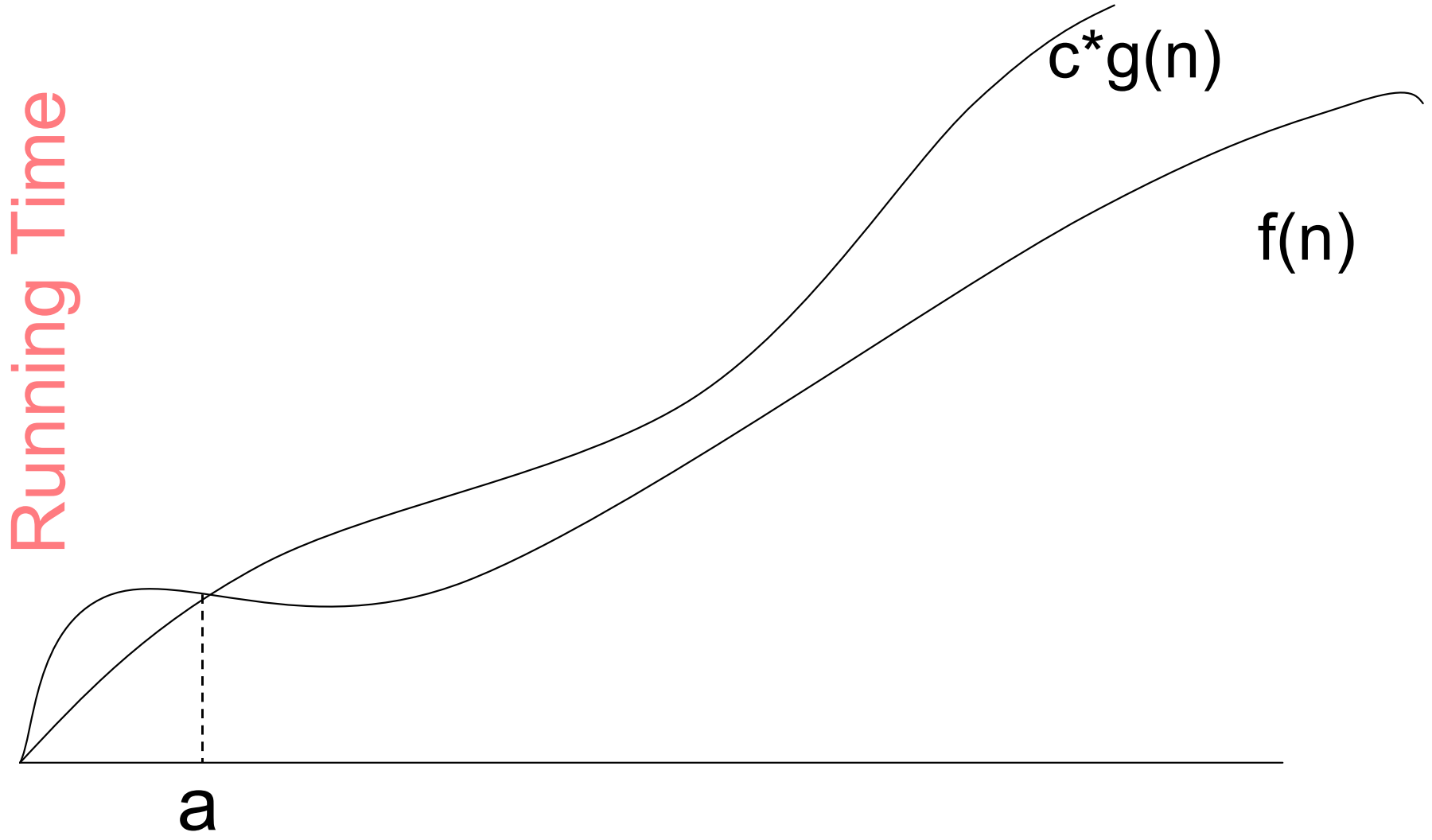
Xấp xỉ tiệm cận

- Mục đích
  - Ví dụ:

$$1{,}000{,}001 \approx 1{,}000{,}000$$

$$3n^2 \approx n^2$$

- Khái niệm "O":
  - Cho f(n) và g(n) ta nói $f(n) \in O(g(n))$ khi và chỉ khi $f(n) \leq c*g(n)$, $\forall\ n \geq a$ với c và a là hằng số, và f(n) và g(n) là các hàm trên miền số tự nhiên


Running Time
c*g(n)
f(n)
a

Xấp xỉ tiệm cận (tt)

- Nếu $7n^2 - 3 \in O(n^5)$ nhưng ta chỉ quan tâm đến xấp xỉ bậc thấp nhất $\Rightarrow 7n^2 - 3 \in O(n^2)$
- Quy tắc xác định O(?): xét thành phần có bậc cao nhất của f

  Ví dụ: $12n - 2 \in O(n)$

  $3n^2\log(n) - 12n^2 + 19 \in O(n^2\log(n))$

  $3e^n - 10000n^2 + 2 \in O(e^n)$

Phân tích thời gian theo hướng tiệm cận

- Sử dụng ký hiệu "O" để biểu diễn khối lượng các thao tác cơ sở.

  Ví dụ: ta có thể nói hàm MaxOfArray chạy với thời gian O(n)

- Khi so sánh các thuật toán ta có thể nói:

  Thuật toán chạy với thời gian O(n) tốt hơn thuật toán $O(n^2)$

  Tương tự O(log(n)) tốt hơn O(n)

Phân tích thời gian theo hướng tiệm cận (tt)

- Có thể hệ số ứng với thành phần có bậc cao nhất có thể rất lớn nên đôi khi, với tập dữ liệu Input xác định thuật toán O(n) sẽ kém hiệu quả hơn thuật toán $O(n^2)$ hoặc ngay cả $O(2^n)$

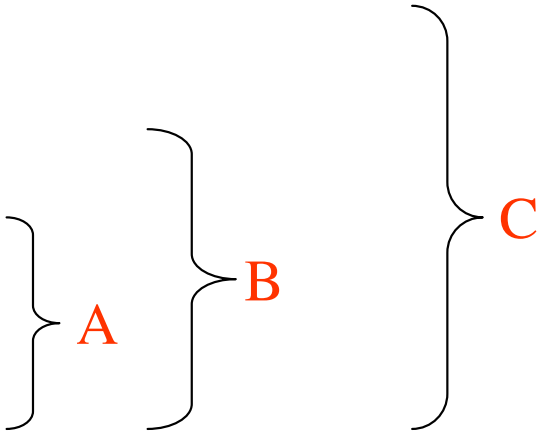
  Ví dụ: 10,000,000n là O(n) nhưng kém hiệu quả hơn $O(n^2)$ khi $n << 10,000,000$

Phương pháp thực hiện

- Lý thuyết hàm sinh
- Khảo sát thực tế

  Ví dụ: log(n) < thực tế <<< n
- Xét ví dụ đoạn chương trình:

```
for(i ← 1 to n) do
    if(i<10) then
        for(j← 1 to log(n)) do
            print j
    else print i
```



Phân tích ví dụ

$$T(A) \in O(\log(n))$$

$$T(B) \in O(1) + \text{Max}\{O(\log(n)), O(1)\}$$

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

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

$$= O(\log(n))$$

$$T(C) \in n * O(\log(n))$$

$$= O(n * \log(n))$$

Trừu tượng hoá dữ liệu

- Đời sống của một chương trình:
  1. Mô tả bài toán (Task specification)
  2. Thiết kế (Design)
  3. Kiểm tra (Verification)
  4. Cài đặt (Implementation, Coding)
  5. Thử nghiệm, bảo trì, nâng cấp, hoàn thiện và sử dụng (Testing Maintenance, Evolution, Refinement and Use)
  6. Lỗi thời và biến mất

Trừu tượng hoá dữ liệu (tt)

- Thế nào là một chương trình tốt ?
- Các tính chất của chương trình tốt:
  1. Đúng đắn (Correct)
  2. Hiệu quả (Efficient)
  3. Chi phí thấp (Cheap)
  4. Dễ hiểu (Understandable)
  5. Dễ bảo trì, nâng cấp (Maintainable)
  6. Mạnh mẽ (robust)
  7. Đơn thể (Modular)

Trừu tượng hoá dữ liệu (tt)

- Kỹ thuật *Lập Trình Đơn Thể* là gì ?
- Dùng phương pháp thiết kế nào để xác định các đơn thể ?
  1. Top-Down
  2. Hoặc Bottom-Up
- LTĐT & trừu tượng hoá để làm gì ?
- Cơ sở chính của trừu tượng hoá:
  - ✓ Quan tâm tổng thể
  - ✓ Không quan tâm chi tiết
  - ✓ Tương tự khái niệm "*Hộp đen*"

Trừu tượng hoá dữ liệu (tt)

- Phân loại trừu tượng hoá:
  1. Trừu tượng hoá chức năng (Functional Abstraction), phân tách giao tiếp của chức năng (hàm) với cài đặt cụ thể.
  2. Trừu tượng hoá dữ liệu (Data Abstraction), phân biệt thuộc tính dữ liệu với cài đặt cụ thể.

Trừu tượng hoá dữ liệu (tt)

- Một kiểu dữ liệu bao gồm những gì ?
  1. Một tập hợp các giá trị
  2. Một tập hợp các phép toán (thao tác) trên các giá trị này
- Kiểu dữ liệu trừu tượng (*Abstract Data Type* - **ADT**) sẽ do người dùng định nghĩa
- Cần chú ý khi cài đặt ADT:
  1. Chọn biểu diễn thích hợp của dữ liệu thông qua các kiểu dữ liệu đã biết.
  2. Cài đặt các thao tác trên dữ liệu

Trừu tượng hoá dữ liệu (tt)

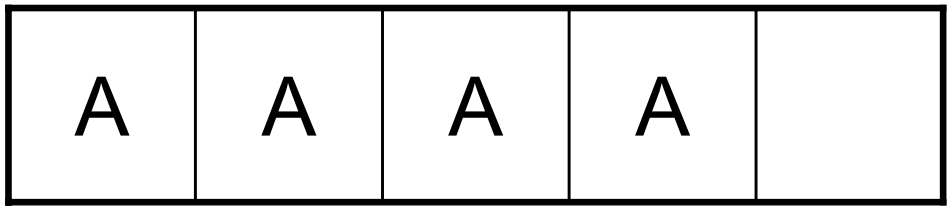
- Kiểu dữ liệu cơ sở

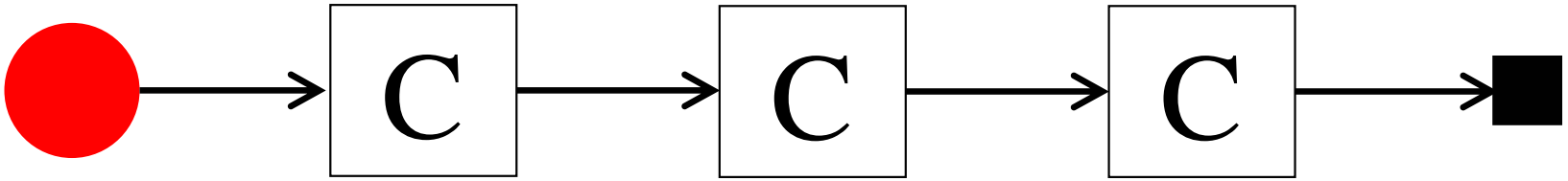
  Ví dụ : char, int, boolean, ...

  được định nghĩa sẵn trong NNLT
- Kiểu dữ liệu (KDL) có cấu trúc
- Phân loại KDL:
  1. Tuyến tính
  2. Phi tuyến

Kiểu dữ liệu tuyến tính

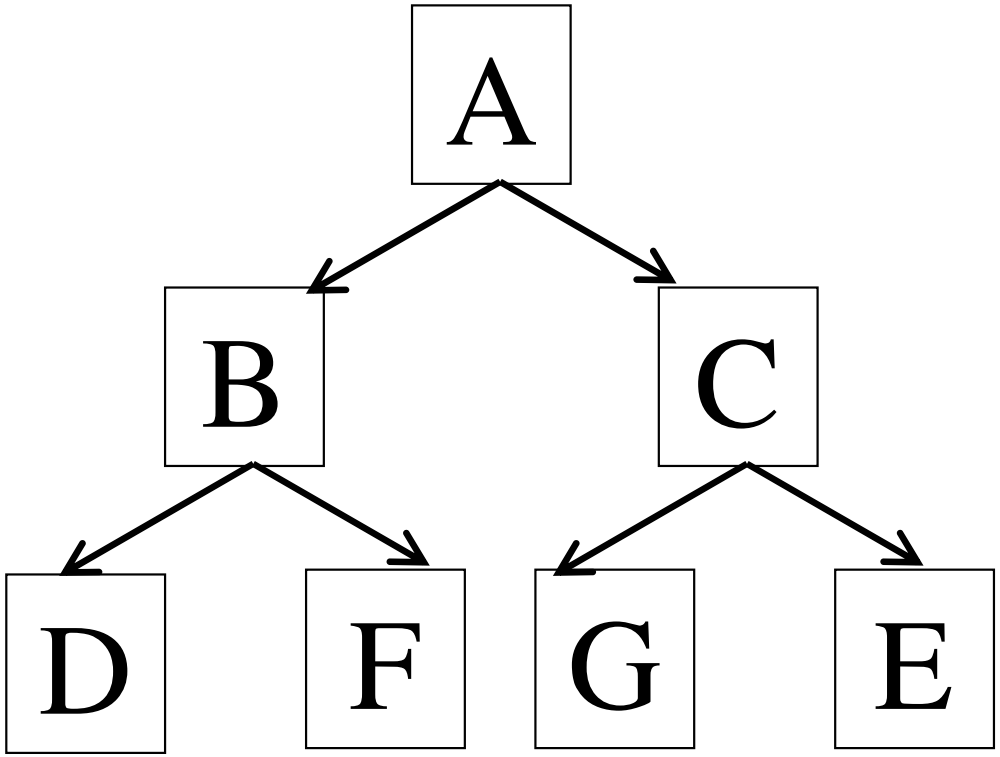
- Là kiểu dữ liệu thoả 02 tính chất sau:
  1. Đứng sau mỗi phần tử tối đa có 01 phần tử
  2. Mỗi phần tử có không quá 01 phần tử đứng trước nó.
- Ví dụ:



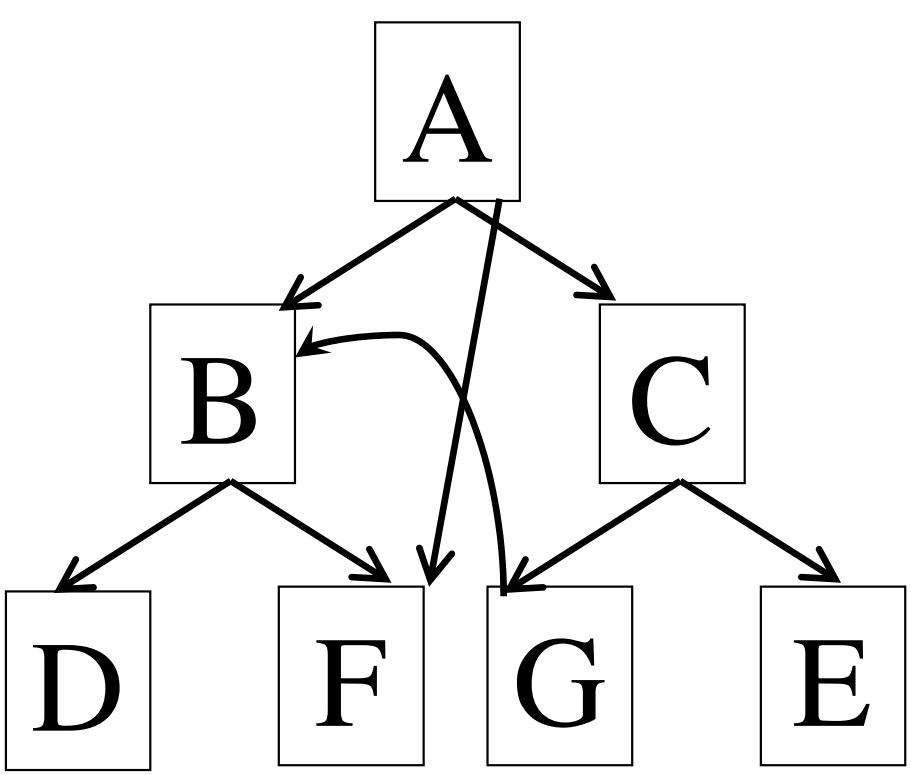


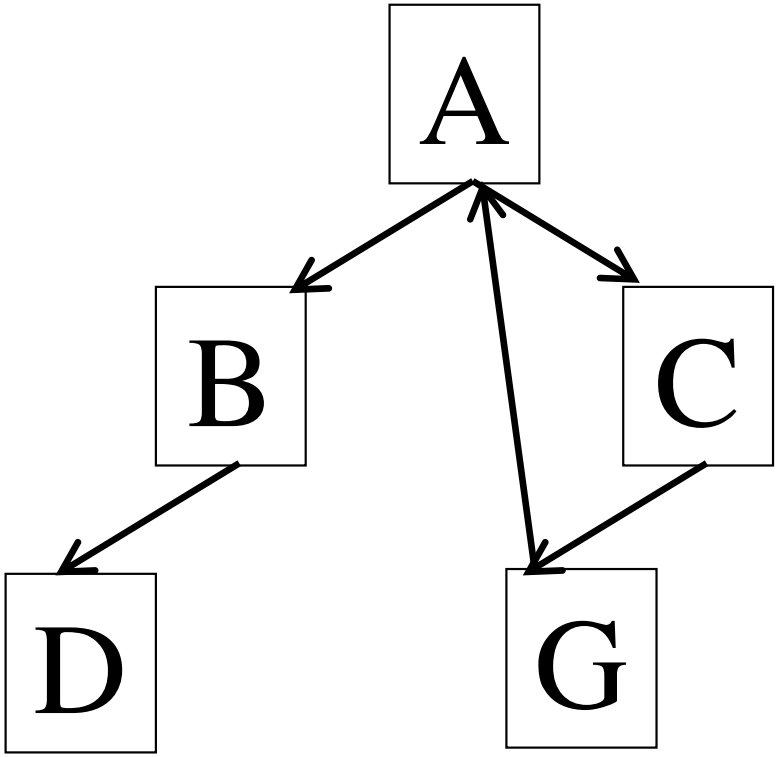
Kiểu dữ liệu phi tuyến

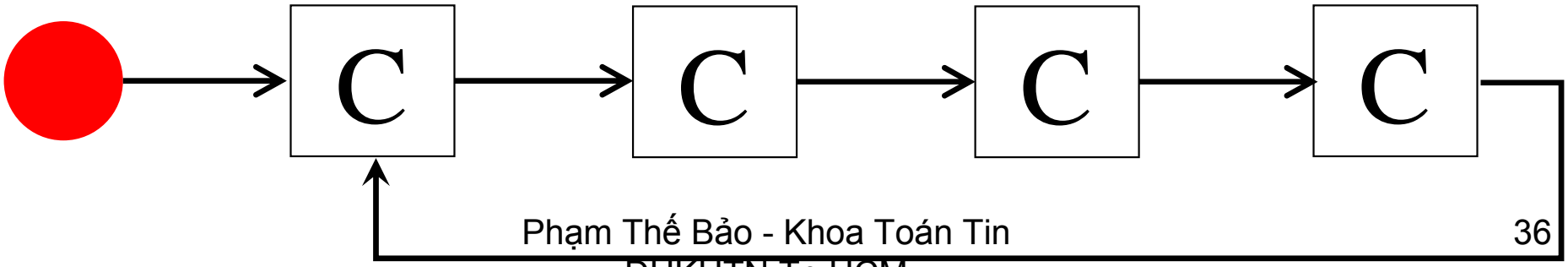
- Là KDL không thoả ít nhất 01 trong 02 tính chất của KDL tuyến tính trên
- Ví dụ:



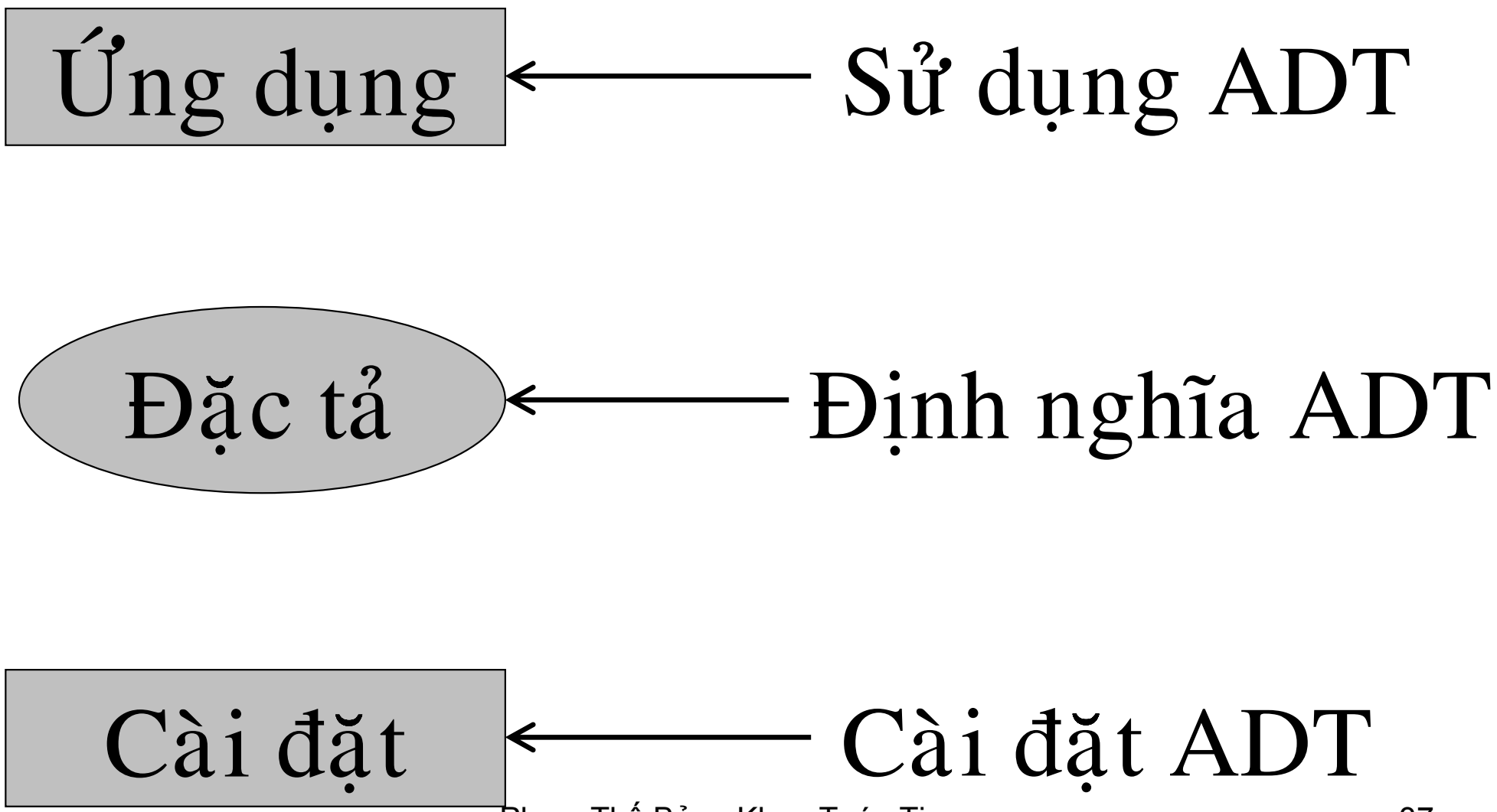
Một số ví dụ







Sơ đồ xây dựng ADT



Ví dụ xây dựng một ADT đơn giản (KDL số phức)

- Đặc tả bài toán:

  Số phức là một số biểu diễn bởi 01 cặp số nguyên (a,b), a là phần thực và b là phần ảo. Ta có thể áp dụng các phép toán số học như: +, -, *, /. Ngoài ra ta có tính chất i*i = -1

- Cài đặt 1:

```
typedef struct {
    int  imaginary_number, real_number;
    }complex;
```

Ví dụ (tt)

- Cài đặt 2:

```
#define imaginary_number        0
#define real_number             1
typedef int complex[2];
```

- Các hàm cơ sở:
  - ✓ createComplex(a,b): nhận 02 số nguyên a,b và trả ra số phức tương ứng.
  - ✓ getImaginaryNumber(c): nhận vào 01 số phức c và trả ra phần ảo của nó.
  - ✓ getRealNumber(c): nhận vào 01 số phức c và trả ra phần thực của nó.

Ví dụ (tt)

- Hàm mở rộng:
  - ✓addComplex(complex c1,complex c2): cộng 02 số phức. Cài đặt như sau:

```
complex addComplex(complex c1,complex c2){
        int r1 = getRealNumber(c1);
        int r2 = getRealNumber(c2);
        int i1 = getImaginaryNumber(c1);
        int i2 = getImaginaryNumber(c2);
        return createComplex(r1+r2,i1+i2);
        }
```

TÌM KIẾM
(SEARCH)

Cấu trúc dữ liệu mảng

- Định nghĩa: CTDL mảng (array) là CTDL gồm một dãy liên tiếp các phần tử. Các phần tử này có thể được truy cập ngẫu nhiên.
- Các thao tác thông thường trên mảng:
  - ❖ Khởi tạo mảng
  - ❖ Lấy giá trị của phần tử thứ i trong mảng
  - ❖ Đặt giá trị cho phần tử thứ i trong mảng
  - ❖ Xác định kích thước mảng (số phần tử trong mảng)
  - ❖ Thêm một phần tử vào mảng
  - ❖ Huỷ một phần tử ra khỏi mảng
  - ❖ Tìm kiếm một phần tử trong mảng
  - ❖ Sắp xếp mảng

Tìm kiếm

- Phương pháp cơ bản:
  1. Tuần tự (tuyến tính – Linear Search)
  2. Nhị phân (Binary Search)
- Ý tưởng:
  1. Tuần tự: duyệt từ phần tử đầu tiên cho đến phần tử cuối cùng để tìm phần tử thoả điều kiện cần tìm.
  2. Nhị phân: phân hoạch làm 02 phần, 01 phần chắc chắn không có, phần còn lại nếu có sẽ tồn tại. Quá trình tìm kiếm sẽ có không gian tìm kiếm thu hẹp, nên việc tìm kiếm sẽ nhanh chóng. Điều kiện các phần tử phải có quan hệ thứ tự nào đó.

Tìm kiếm (tt)

- Thuật giải:

1. Tuần tự:

Thuật giải LSearch(A,n, α)

Input: Mảng A có n phần tử và điều kiện α

Ouput: Vị trí phần tử cần tìm, hoặc -1 nếu không tìm thấy

```
for(i ← 0 to n-1)
        if(A[i] thoả điều kiện α) return i;
return -1;
End LSearch;
```

Tìm kiếm (tt)

Ưu Khuyết điểm của thuật toán tìm kiếm tuần tự

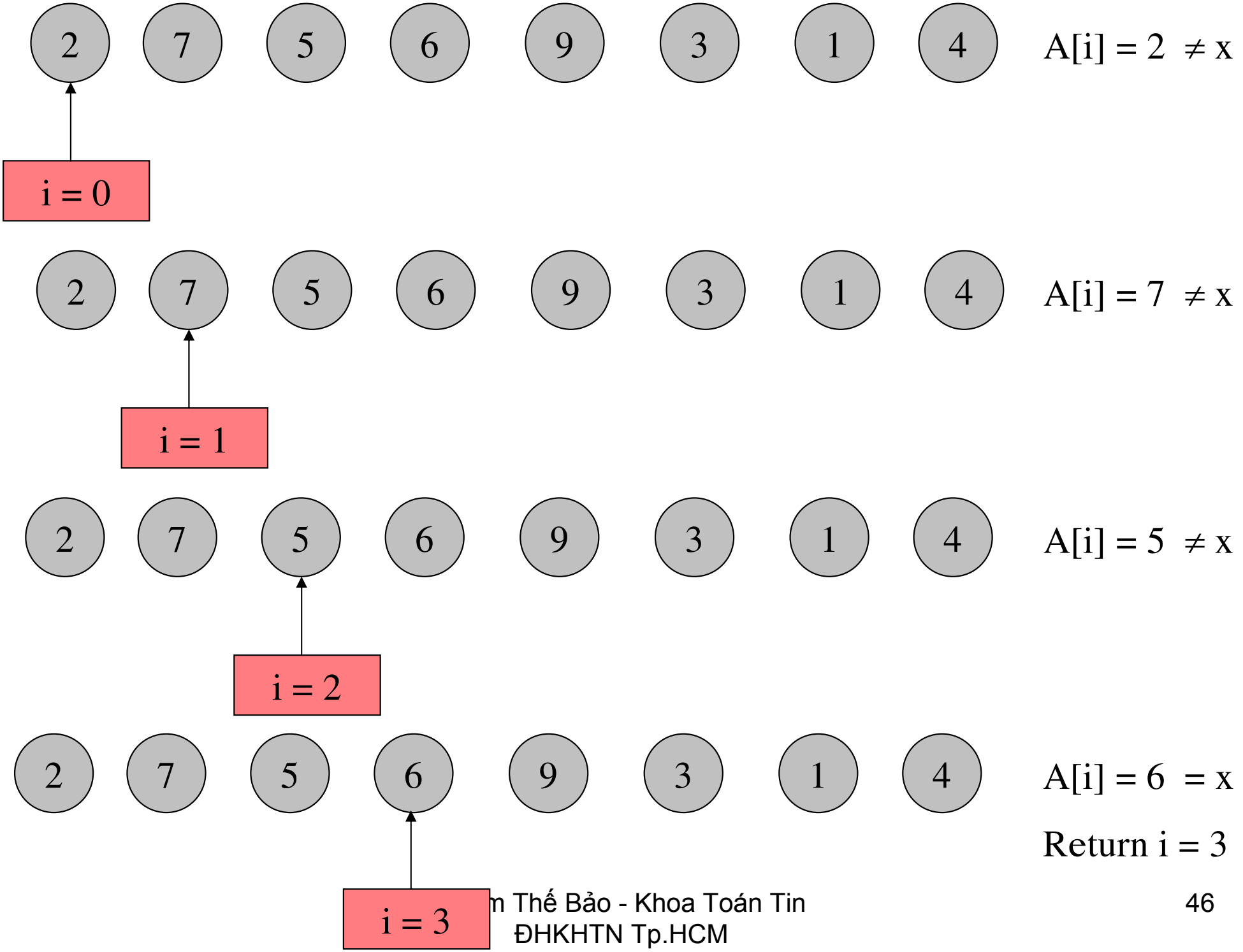
Ưu điểm:

tổng quát, có thể áp dụng trên mọi tiêu chuẩn tìm kiếm cũng như mọi dữ liệu.

Khuyết điểm:

Chi phí cao (tỉ lệ O(n))

Ví dụ: tìm phần tử x = 6 trong mảng dưới đây


2 7 5 6 9 3 1 4
A[i] = 2 ≠ x
i = 0
2 7 5 6 9 3 1 4
A[i] = 7 ≠ x
i = 1
2 7 5 6 9 3 1 4
A[i] = 5 ≠ x
i = 2
2 7 5 6 9 3 1 4
A[i] = 6 = x
Return i = 3
i = 3

Tìm kiếm (tt)

2. Nhị phân

Thuật giải BSearch(A,n, α)

Input: Mảng A có n phần tử và điều kiện α

Ouput: Vị trí phần tử cần tìm, hoặc -1 nếu không tìm thấy

```
left ← 0; right ← n-1;
while(left ≤ right) do
        mid ← (left+right)/2;
        if(A[mid] thoả α) then return mid;
        if(phần tử thoả α ở bên trái) then
                right ← mid-1;
        else left ← mid +1;
return -1;
End BSearch;
```

Tìm kiếm (tt)

Ưu Khuyết điểm của thuật toán tìm kiếm nhị phân

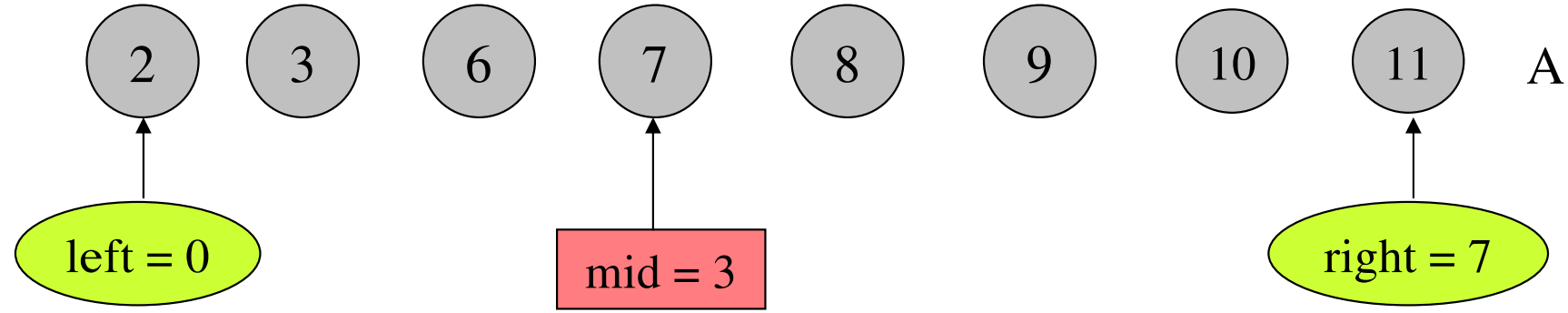
Ưu điểm:

Chi phí thấp (tỉ lệ $O(\log_2(n))$

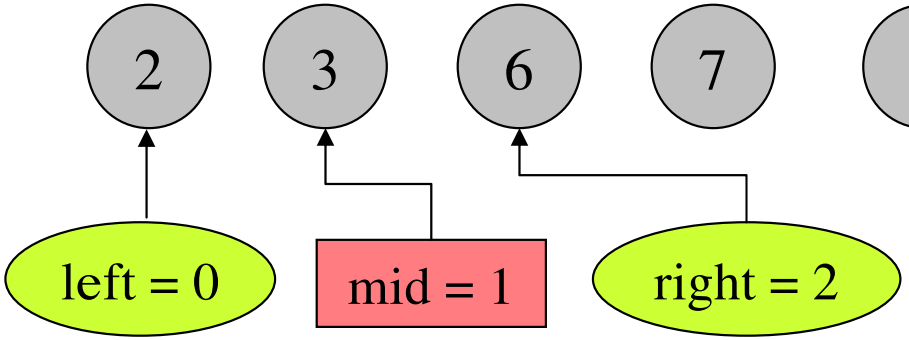
Khuyết điểm:

Thuật toán không tổng quát, không thể áp dụng trên mọi tiêu chuẩn tìm kiếm cũng như mọi dữ liệu. Dữ liệu bắt buộc phải có quan hệ thứ tự và việc tìm kiếm phải theo khoá thứ tự này.

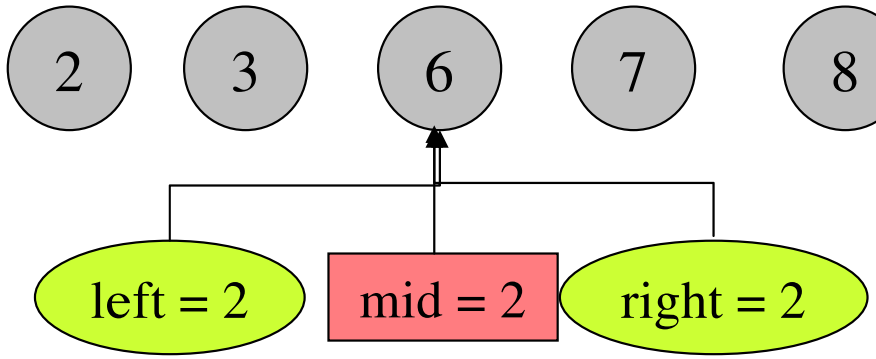
Ví dụ: tìm phần tử x = 6 trong mảng dưới đây



A[mid] = 7 > x



A[mid] = 3 < x



A[mid] = 6 = x

Return mid = 2

SẮP XẾP

(SORT)

1. Đổi chỗ trực tiếp – Interchange Sort
2. Nổi bọt – Bubble Sort
3. Shaker Sort
4. Chèn trực tiếp – Insertion Sort
5. Chèn nhị phân – Binary Insertion Sort
6. Shell Sort
7. Chọn trực tiếp – Selection Sort
8. Heap Sort
9. Quick Sort
10. Merge Sort
11. Radix Sort

Phương pháp đổi chỗ trực tiếp

- Ý tưởng thuật toán:

Khái niệm nghịch thế

Xét một mảng các số a_0 a_1 ... a_n

Nếu có i<j và $a_j < a_i$ thì ta gọi đó là một nghịch thế

Vậy nếu mảng chưa sắp xếp: sẽ có nghịch thế

Nếu mảng đã sắp xếp (có thứ tự): không có nghịch thế và khi đó a_0 sẽ là phần tử bé nhất rồi đến a_1 a_2 ...

Đổi chỗ trực tiếp (tt)

Để một mảng được sắp xếp thì số nghịch thế phải giảm dần, để làm được ta sẽ hoán vị cặp phần tử xảy ra nghịch thế.

- Thuật toán:

Thuật toán InterchangeSort(A,n)

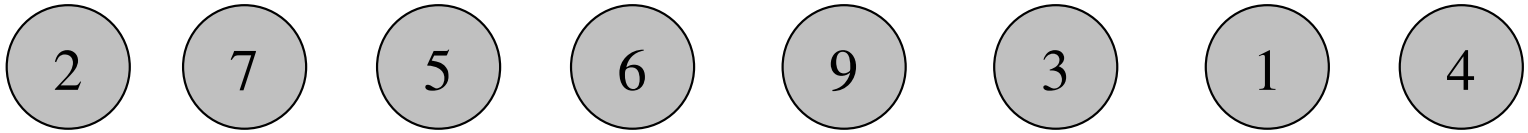
Input: Mảng A có n phần tử

Output: Mảng A được sắp xếp

```
for(i ← 0 to n-2) do
   for(j ← i+1 to n-1) do
         if(A[i]>A[j]) then A[i] ↔ A[j]
return;
```

Đổi chỗ trực tiếp (tt)

- Nhận xét:
  - Sau mỗi lần lặp của vòng lặp for i, phần tử A[i] = min (A[i], A[i+1], ..., A[n-1])
  - Thuật toán có độ phức tạp $O(n^2)$
- Ví dụ:



Phương pháp nổi bọt

- Ý tưởng thuật toán:

  Từ một số gợi ý trong thiên nhiên như:

  - ✓ Bọt khí
  - ✓ Hỗn hợp dầu và nước
  - ✓ Thả hỗn hợp gạo và mạt cưa vào nước

  Ta sẽ đổi chỗ cho 02 phần tử liên tiếp nếu 02 phân tử đó tạo thành 01 nghịch thế.

Phương pháp nổi bọt (tt)

- Thuật toán:

Thuật toán BubbleSort(A,n)

Input: Mảng A có n phần tử

Output: Mảng A được sắp xếp

for(i ← 0 to n-2) do

for(j ← n-1 to i+1) do

if(A[j-1] > A[j]) then A[j-1] ↔ A[j]

return;

Phương pháp nổi bọt (tt)

- Nhận xét:
  - ✓ Sau mỗi lần lặp của vòng lặp for i, phần tử nhẹ nhất sẽ nổi lên đến mức i; nghĩa là A[i] = min(A[i], A[i+1], ..., A[n-1])
  - ✓ Thuật toán có độ phức tạp $O(n^2)$
  - ✓ Thay vì cho vật nhẹ nổi lên ta cũng có thể cho vật nặng chìm xuống

Phương pháp nổi bọt (tt)

Thuật toán BubbleSort1(A,n)

Input: Mảng A có n phần tử

Output: Mảng A được sắp xếp

```
for(i ← n-1 to 1) do
        for(j ← 0 to i-1) do
            if(A[j] > A[j+1]) then A[j] ↔ A[j+1]

return;
```

Phương pháp nổi bọt (tt)

✓ Hay đồng thời cho vật nặng chìm xuống và vật nhẹ nổi lên.

Thuật toán BubbleSort2(A,n)

Input: Mảng A có n phần tử

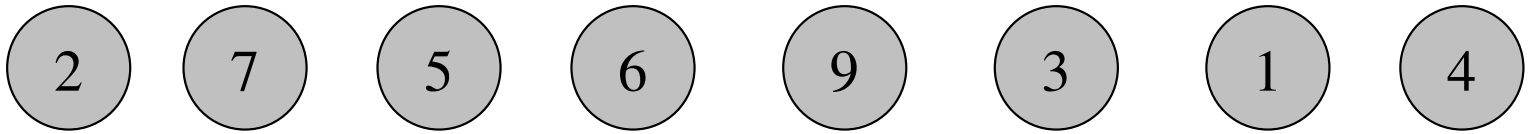
Output: Mảng A được sắp xếp

```
up ← 0; down ← n-1;
while (up < down) do
        for (j ← up to down-1) do
                if(A[j] > A[j+1]) then A[j] ↔ A[j+1]
        down ← down -1;
        for(j ← down to up+1) do
                if(A[j-1] > A[j]) then A[j-1] ↔ A[j]
        up ← up + 1;
return;
```

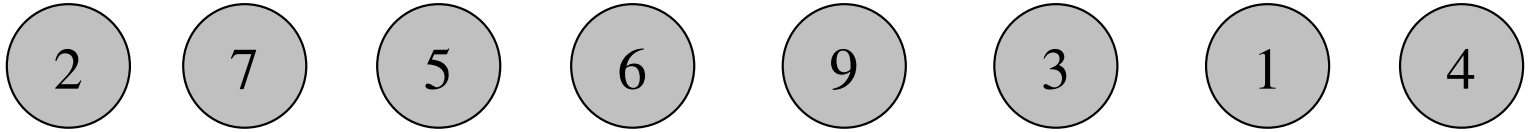
Phương pháp nổi bọt (tt)

- ✓ Dù thuật toán vẫn có độ phức tạp $O(n^2)$ nhưng chi phí sẽ giảm nhiều, vì ta đã lợi dụng tính đối ngẫu.

- Ví dụ 1 (nhẹ nổi lên)



- Ví dụ 2 (nhẹ nổi lên và nặng chìm xuống)



Phương pháp Shaker Sort

- Ý tưởng thuật toán:

  Phương pháp Bubble Sort có một nhược điểm rất lớn là khi dữ liệu có thể đã được sắp xếp cục bộ thì phương pháp Bubble Sort vẫn phải làm tuần tự mà không lợi dụng được tính chất này. Do đó Shaker Sort là phương pháp cải tiến của Bubble Sort để giảm số lần lặp.

Phương pháp Shaker Sort (tt)

- Thuật toán:

Thuật toán ShakerSort(A,n)

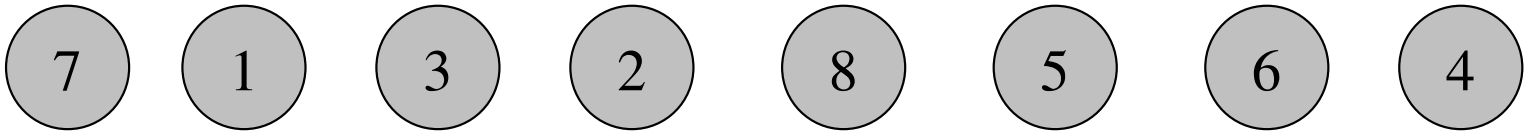
Input: Mảng A có n phần tử

Output: Mảng A được sắp xếp

```
up ← 0; down ← n-1;hv ← 0;
while (up < down) do
        for (j ← up to down-1) do
                if(A[j] > A[j+1]) then
                        A[j] ↔ A[j+1];
                        hv ← j;
        down ← hv;
        for(j ← down to up+1) do
                if(A[j-1] > A[j]) then
                        A[j-1] ↔ A[j];
                        hv ← j;
        up ← hv;
return;
```

Phương pháp Shaker Sort (tt)

- Nhận xét:
  - ✓ Thuật toán có độ phức tạp $O(n^2)$
  - ✓ Tuy nhiên qua thực nghiệm, nếu dữ liệu có thứ tự cục bộ nhiều thì Shaker Sort sẽ nhanh hơn rất nhiều so với Bubble Sort.
- Ví dụ:



Phương pháp chèn trực tiếp

- Ý tưởng thuật toán:

  Xét mảng A có các số a_0 a_1 ... a_{i-1} a_i a_n

  Giả sử tại thời điểm thứ i đã có i phần tử đầu tiên của mảng đã có thứ tự $a_0 \le a_1 \le \dots \le a_{i-1}$

  Nếu ta tìn được vị trí k ($0 \le k \le i$) sao cho $a_{k-1} \le a_i \le a_k$

  ta sẽ chèn a_i vào giữa a_{k-1} và a_k ta sẽ nhận được trạng thái mới của mảng A trong đó i+1 phần tử đầu tiên có thứ tự tăng.

Phương pháp chèn trực tiếp (tt)

- Thuật toán:

Thuật toán InsertionSort(A,n)

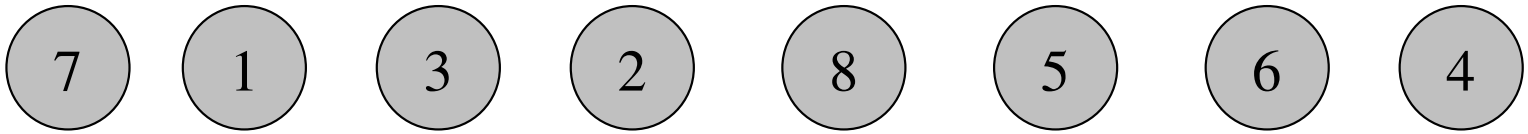
Input: Mảng A có n phần tử

Output: Mảng A có thứ tự

```
for(i ← 1 to n-1) do
        x ← A[i];
        j ← i-1;
        while (x<A[j] && j ≥ 0) do
                A[j+1] ← A[j];
                j ← j-1;
        A[j+1] ← x;
return;
```

Phương pháp chèn trực tiếp (tt)

- Ví dụ:



Phương pháp chèn trực tiếp (tt)

- Nhận xét:
  - ✓ Thuật toán có độ phức tạp $O(n^2)$
  - ✓ Dữ liệu càng ít nghịch thế, thuật toán chạy càng nhanh.

Phương pháp chèn nhị phân

- Ý tưởng thuật toán:

  Trong thuật toán chèn trực tiếp, khi tìm vị trí k thỏa $a_{k-1} \le a_i \le a_k$, phương pháp tìm vị trí là *tuần tự* , như thế ta có thể dùng phương pháp tìm kiếm *nhị phân* để cải tiến do $a_0 \le a_1 \le \ldots \le a_{i-1}$

Phương pháp chèn nhị phân (tt)

- Thuật toán:

Thuật toán BinaryInsertionSort(A,n)

Input: Mảng A có n phần tử

Output: Mảng A có thứ tự

```
for(i ← 1 to n-1) do
        x ← A[i];
        k ← BinarySearch(A,i,x);
        for(j ←i to k+1) do
                A[j] ← A[j-1];
        A[k] ← x;
return;
```

Phương pháp chèn nhị phân (tt)

- Nhận xét:
  - ✓ Thuật toán có độ phức tạp $O(n^2)$ với thao tác gán nhưng có độ phức tạp $O(n\log_2 n)$ với thao tác so sánh.
  - ✓ Nhưng phương pháp chèn nhị phân sẽ nhanh hơn đáng kể khi dữ liệu rất nhiều và đặc biệt khi chi phí so sánh giữa 02 phần tử là cao, ví dụ: string, cấu trúc, ...

Phương pháp Shell Sort

- Ý tưởng thuật toán:

  Khi dữ liệu càng ít hỗn độn (ít nghịch thế) thì thuật toán chèn trực tiếp càng chạy nhanh, ta có thuật toán chèn nhị phân là một cải tiến. Một hướng tiếp cận khác là làm sao giảm bớt số nghịch thế trước khi thực sự sắp xếp. Shell sort là cải tiến của chèn trực tiếp theo hướng tiếp cận này.

  Ta sẽ phân hoạch mảng thành nhiều dãy con đan xen nhau, sắp xếp các dãy con này trước khi ta dùng phương pháp chèn trực tiếp để sắp xếp mảng.

Phương pháp Shell Sort (tt)

Chúng ta phải phân hoạch các bước nhảy như sau $h_0 > h_1 > \ldots > h_k$, và ở bước cuối cùng phải là 1. Tại các bước ta sẽ dùng thuật toán chèn trực tiếp để sắp xếp các dãy con.

Phương pháp Shell Sort (tt)

- Thuật toán:

Thuật toán ShellSort(A,n)

Input: Mảng A có n phần tử

Output: Mảng A có thứ tự

Chọn h[0], h[1], ..., h[k-1] = 1;

for(i ← 0 to k-1) do

Phân hoạch mảng A thành t dãy có chiều dài h[i]

for(j ← 0 to t-1) do

InsertionSort(dãy con đã phân hoạch);

return;

Phương pháp Shell Sort (tt)

Nhưng trong cài đặt, chúng ta sẽ đồng thời sắp xếp từng dãy con.

- Thuật toán:

Thuật toán ShellSort(A,n)

Input: Mảng A có n phần tử

Output: Mảng A có thứ tự

Chọn h[0], h[1], ..., h[k-1] = 1;

```
for(p ← 0 to k-1) do
   for(i ← h[p] to n-1) do
            x ← A[i]; j ← i;
            while(x<A[j-h[p]] && j ≥ h[p]) do
                     A[j] ← A[j-h[p]];
                     j ← j – h[p];
            A[j] ← x;
return;
```

Phương pháp Shell Sort (tt)

- Nhận xét:
  - ✓ Thuật toán có độ phức tạp nhỏ hơn O($n^{3/2}$) Tuy nhiên Shell Sort hiệu quả hơn nhiều so với chèn trực tiếp.
  - ✓ Tính hiệu quả sẽ phụ thuộc rất nhiếu vào cách chọn các bước nhảy. Tuy nhiên, hiện nay không có tiêu chuẩn tổng quát để chọn các bước nhảy, mà thông qua kinh nghiệm ta sẽ có cách chọn. Nhưng chọn như thế nào đi nữa, ta luôn làm sao chọn các bước nhảy mà phân hoạch mảng thành các dãy ít trùng lặp càng tốt.

Phương pháp Shell Sort (tt)

Một số cách chọn:

- h là dãy các số nguyên tố giảm dần đến 1: 13, 11, 7, 5, 3, 1. Đôi khi không nhất thiết các số nguyên tố này liên tục, ví dụ: 13, 5, 3, 1.
- h có dạng 3i+1: 364, 121, 40, 13, 4, 1
- Dãy fibonaci: 34, 21, 13, 8, 5, 3, 2, 1
- Dãy: 4193, 1037, 281, 77, 23, 8, 1
- Dãy 1968, 861, 336, 112, 48, 21, 7, 3, 1
- Dãy 511, 255, 127, 63, 31, 15, 7, 3, 1

Phương pháp Shell Sort (tt)

- Ví dụ: xét dãy 5,3,1

Với h[0] = 5

Phương pháp chọn trực tiếp

- Ý tưởng thuật toán:

  Ta thấy rằng nếu mảng có thứ tự thì A[i] luôn là phần tử bé nhất của tập (A[i], A[i+1], ..., A[n]). Vậy một cách đơn giản ta đặt phần tử nhỏ nhất vào đầu mảng, tìm phần tử nhỏ kế tiếp trong tập còn lại rồi đặt vào vị trí kế tiếp, cứ như vậy cho đến hết ta có mảng được sắp có thứ tự.

Phương pháp chọn trực tiếp (tt)

- Thuật toán:

Thuật toán SelectionSort(A,n)

Input: Mảng A có n phần tử

Output: Mảng A có thứ tự

```
for(i ← 0 to n-2) do
        k ← i;
        for(j ← i+1 to n-1) do
                if( A[k] > A[j] ) then k← j;
        A[k] ↔ A[i];
return;
```

Phương pháp chọn trực tiếp (tt)

- Nhận xét:
  - ✓ Thuật toán có độ phức tạp $O(n^2)$.

Phương pháp chọn trực tiếp (tt)

- Ví dụ:

Phương pháp Heap Sort

- Ý tưởng thuật toán:

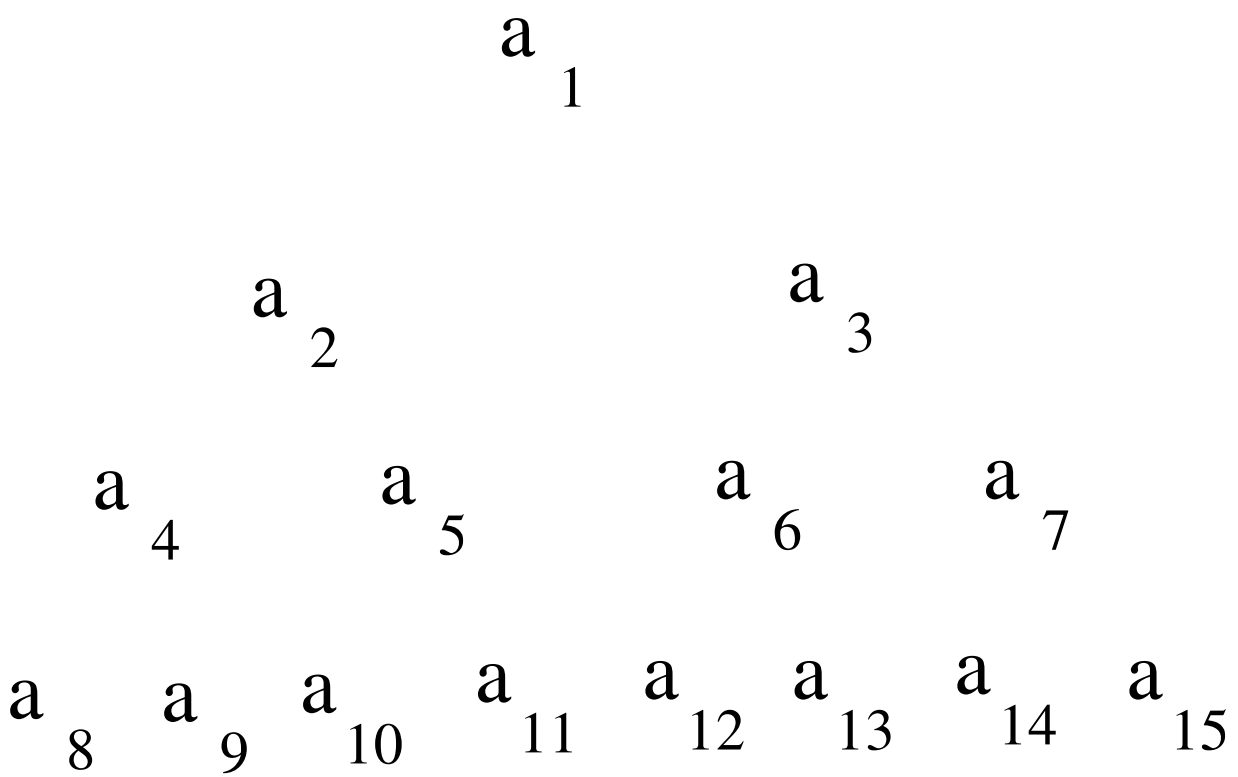
  Ta thấy rằng trong thuật toán chọn trực tiếp, cứ mỗi vòng lặp ta phải tìm phần tử nhỏ nhất trong số các phần tử còn lại. Trong quá trình tìm kiếm ta không tận dụng được các thông tin ở các lần tìm trước đó.

  Một các tiếp cận khác là ta sẽ làm sao lưu trữ thông tin các lần tìm kiếm trước đó để có thể sử dụng sau này.

Phương pháp Heap Sort (tt)

- Khái niệm Heap:
  - Nếu có dãy các phần tử $a_l\ a_{l+1}\ \dots\ a_r$ $(l \le r)$ mà a_i có quan hệ với $a_{2i}\ a_{2i+1}$ lớn hơn hay nhỏ hơn $\forall\ i = l\ ..\ r$ thì ta nói đây là heap.
  - Heap max là heap mà $a_i \ge \max(a_{2i}, a_{2i+1})$
  - Heap min là heap mà $a_i \le \min(a_{2i}, a_{2i+1})$
  - Mọi dãy $a_l\ a_{l+1}\ \dots\ a_r$ với $(l \le r < 2l)$ đều là heap.

Phương pháp Heap Sort (tt)



Phương pháp Heap Sort (tt)

- Tính chất:
  - Nếu có dãy $a_l\ a_{l+1}\ \dots\ a_r$ là heap thì $a_i\ a_{i+1}\ \dots\ a_j$ $(l \le i \le j \le r)$ cũng là heap.
  - Nếu $a_1\ a_2\ \dots\ a_k$ là heap max thì a_1 là phần tử lớn nhất.
  - Nếu $a_1\ a_2\ \dots\ a_k$ là heap min thì a_1 là phần tử bé nhất.

Phương pháp Heap Sort (tt)

- Ứng dụng heap:

  Nếu ta tổ chức được mảng $a_0\ a_1\ \ldots\ a_{n-1}$ thành heap max thì a_0 là phần tử lớn nhất, vậy ta chỉ cần hoán đổi a_0 và a_{n-1} thì ta sẽ có a_{n-1} ở đúng vị trí. Và ta vẫn có $a_1\ a_2\ \ldots\ a_{n-2}$ là một heap, vậy ta chỉ cần cập nhật thêm a_0 vào để tạo ra một heap nữa thì ta tiếp tục quá trình như trên ta sẽ được mảng có thứ tự tăng dần.

Phương pháp Heap Sort (tt)

Thuật toán:

Thuật toán HeapSort(A,n)

Input: Mảng A có n phần tử

Output: Mảng A có thứ tự

CreateHeap(A,n);

for(k ← n-1 to 1) do

A[0] ↔ A[k];

InsertHeap(A,0,k-1);

return;

Phương pháp Heap Sort (tt)

Thuật toán CreateHeap(A,n)

Input: Mảng A có n phần tử

Output: Mảng A là heap max

for(k ← (n+1)/2 - 1 to 0) do

InsertHeap(A,k,n-1);

return;

Phương pháp Heap Sort (tt)

Thuật toán InsertHeap(A,l,r)

Input: Mảng A chứa heap $a_{l+1} \dots a_r$

Output: Mảng A chứa heap $a_l\ a_{l+1} \dots a_r$

```
p ← 2*l;
if(p > r) then return;
if(p < r) then
        if(A[p] < A[p+1]) then p ← p+1;
if(A[l] < A[p])
        A[l] ↔ A[p];
        InsertHeap(A,p,r);
return;
```

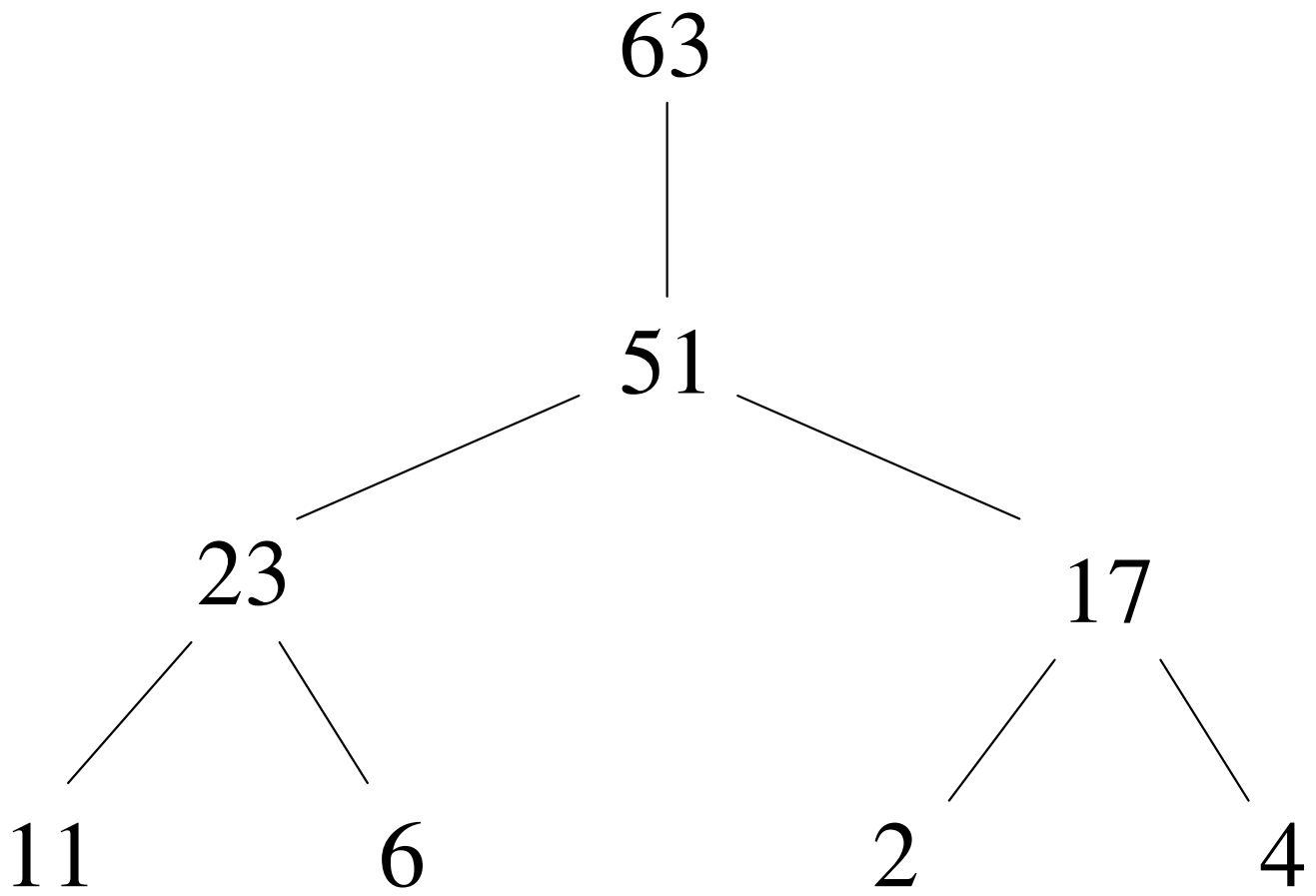
Phương pháp Heap Sort (tt)

- Nhận xét:
  - Thuật toán có độ phức tạp $O(\log_2 n)$
  - Muốn sắp xếp mảng tăng dùng heap max
  - Muốn Sắp xếp mảng giảm dùng heap min

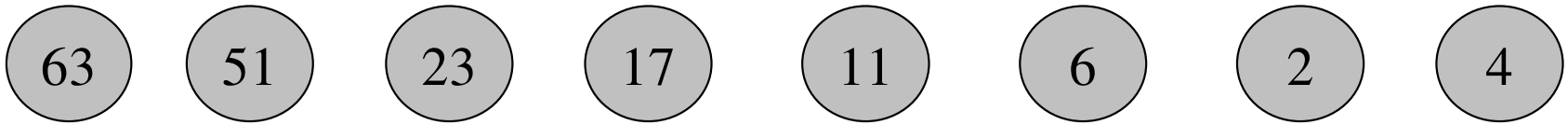
Phương pháp Heap Sort (tt)

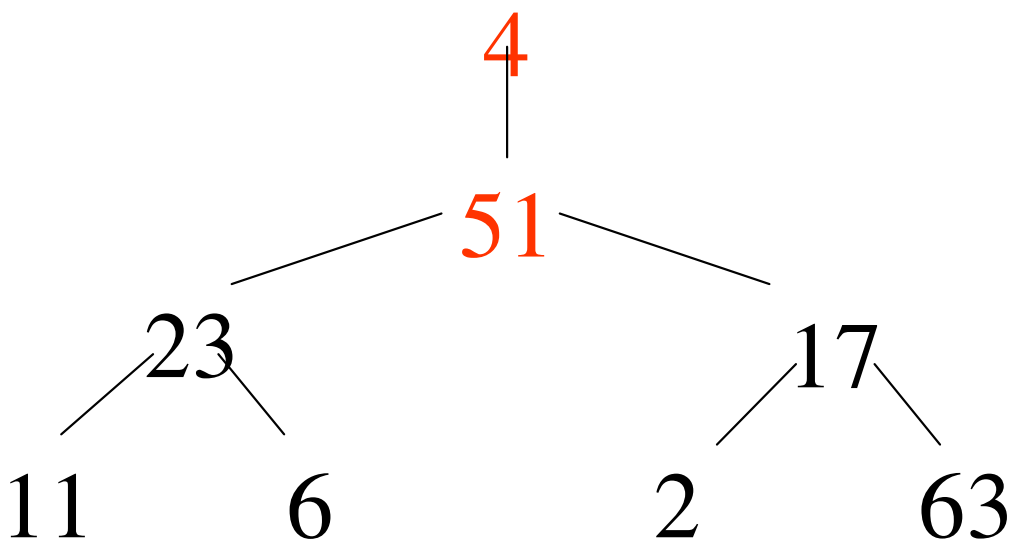
- Ví dụ:

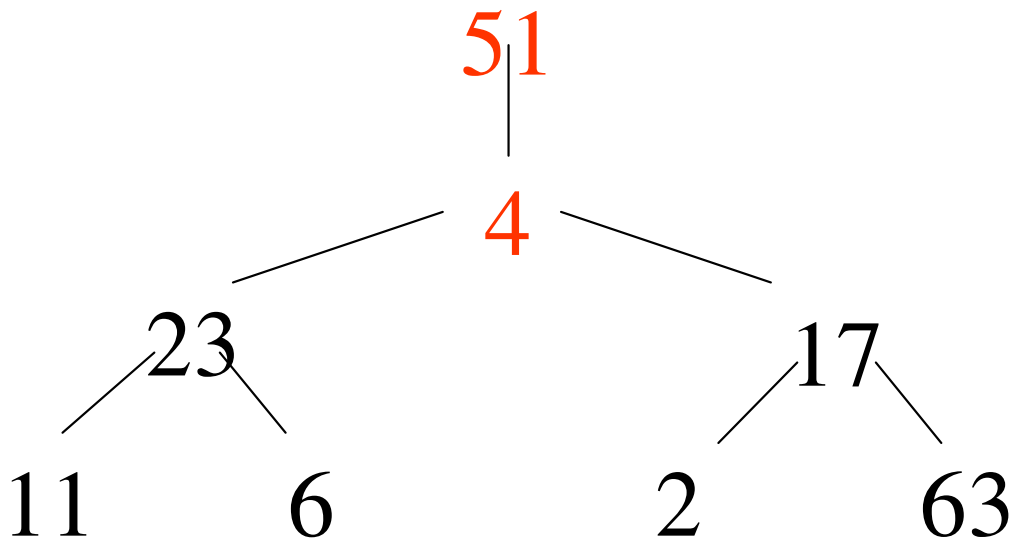
  Bước 1 : Tạo Heap

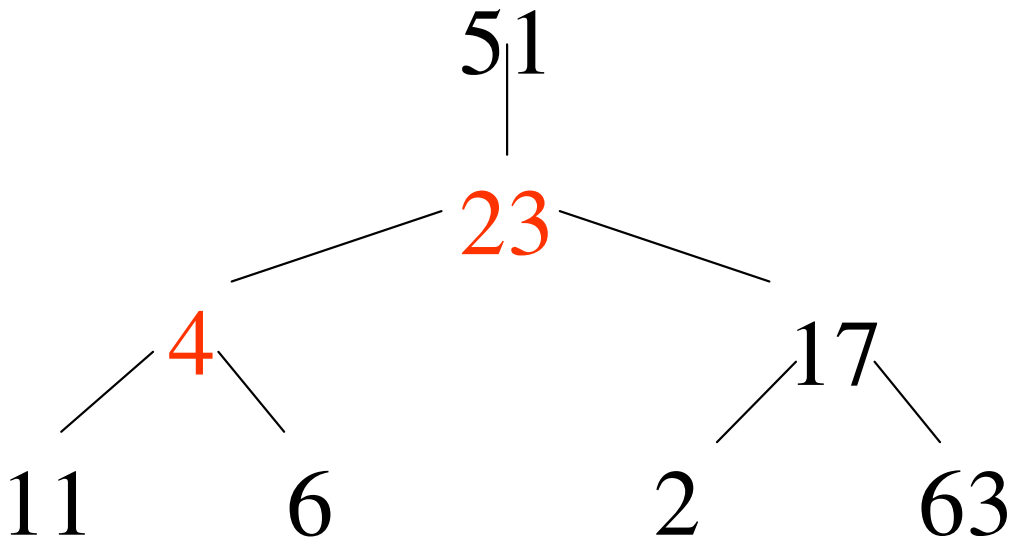

63
51
23
17
11
6
2
4

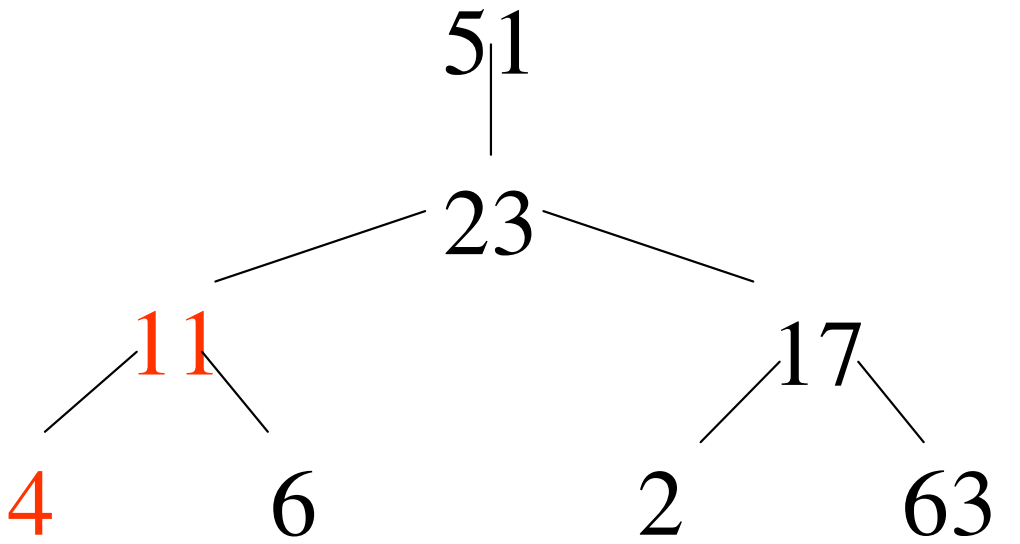
Sắp xếp




4
51
23
17
11
6
2
63


51
4
23
17
11
6
2
63


51
23
4
17
11
6
2
63


51
23
11
17
4
6
2
63

Phương pháp Quick Sort

- Tổng quan:

  Quick Sort do Hoare đưa ra vào những năm 60, được nhiều nhà toán học, tin học nghiên cứu và phát triển như D. Knuth, N. Wirth

- Ý tưởng thuật toán:

  Nguyên tắc chính là chia để trị. Đây là cơ sở của nhiều thuật toán hiệu quả.

  Mảng A nếu ta chia làm 02 phần, phần đầu gồm các phần tử $\leq X$, phần sau là các phần tử $\geq X$, với X là số bất kỳ trong A. Nếu ta sắp xếp 02 phần này thì A sẽ có thứ tự.

Phương pháp Quick Sort (tt)

- Thuật toán:

Thuật toán QuickSort(A,l,r)

Input: Mảng A có các phần tử ở vị trí tử l đến r.

Output: Mảng A được sắp xếp từ l đến r.

Chọn phần tử X;

i ← l; j ← r;

while (i ≤ j) do

while (A[i] < X) do i ← i + 1;

while (A[j] > X) do j ← j - 1;

if (i ≤ j) then

A[i] ↔ A[j];

i ← i + 1;

j ← j - 1;

if (j > l) then QuickSort(A,l,j);

if(r > i) then QuickSort(A,I,r);

return;

Phương pháp Quick Sort (tt)

- Nhận xét:
  - Sau mỗi lần phân hoạch thì mảng A có các phần tử từ vị trí l đến j sẽ có giá trị ≤ X và từ i đến r sẽ ≥ X.
  - Khi phân chia bắt buộc phần tử X phải là phần tử trong mảng để đảm bảo tính dừng của thuật toán (cả hai phần sau khi chia sẽ không bị rỗng).
  - Trung bình thuật toán có độ phức tạp $O(n \log_2 n)$

Phương pháp Quick Sort (tt)

- Trong trường hợp xấu nhất là $O(n^2)$
- Trường hợp xấu nhất là sau khi phân hoạch thì một phần chứa 01 phần tử và phần kia chứa l-r phần tử còn lại. Trường hợp tốt nhất xảy ra khi chia mảng được 02 phần xấp xỉ gần bằng nhau về số lượng (chỉ sai khác 01 phần tử). Do đó, trong cài đặt lập trình viên thường chọn phần tử X = A[(l+r)/2] với hy vọng sẽ là phần tử trung vị (median)

Phương pháp Quick Sort (tt)

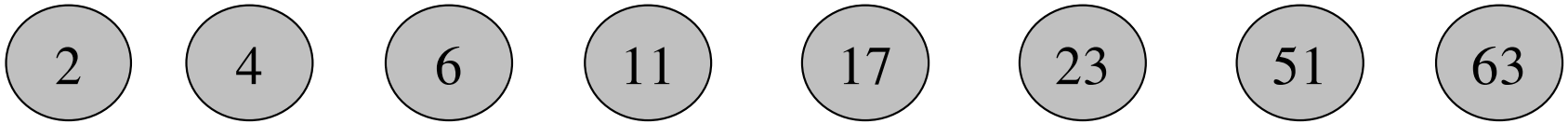
- Trong nhiều cải tiến, người ta cố gắng chọn phần tử X không phải là phần tử lớn nhất hay nhỏ nhất.
- Cuối cùng, khi n nhỏ, lập trình viên sẽ không dùng quick sort mà sẽ dùng thuật toán khác hiệu quả hơn như: chọn trực tiếp, nổi bọt, ... Nhưng khi n khá lớn chúng ta sẽ gặp khó khăn với vấn đề đệ quy.

Phương pháp Quick Sort (tt)

- Ví dụ:

Chọn X = A[(7+0)/2] = 4

Cuối cùng ta được



Phương pháp Merge Sort

- Tổng quan:

  Merge Sort do D. Knuth giới thiệu vào những năm 60

- Ý tưởng thuật toán:

  Nguyên tắc chính cũng là chia để trị. Nhưng cách chia sẽ khác Quick Sort.

  Mảng A nếu ta chia làm 02 phần bằng nhau, nếu ta sắp xếp 02 phần lại rồi trộn 02 nửa lại ta sẽ có mảng có thứ tự.

Phương pháp Merge Sort (tt)

- Thuật toán:

Thuật toán MergeSort(A,l,r)

Input: Mảng A có các phần tử ở vị trí tử l đến r.

Output: Mảng A được sắp xếp từ l đến r.

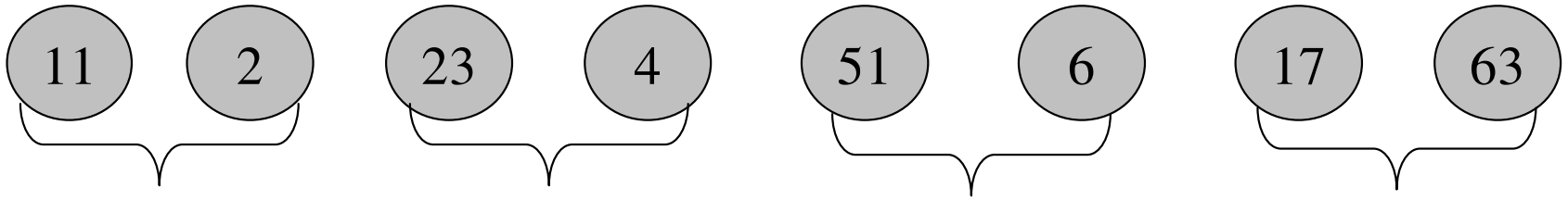
```
if(l < r) then
        mid ← (l+r)/2 ;
        MergeSort(A,l,mid);
        MergeSort(A,mid+1,r)
        Trộn A[l,mid] và A[mid+1,r] thành A[l,r];
return;
```

Phương pháp Merge Sort (tt)

- Nhận xét:
  - Sau mỗi lần phân hoạch thì số lượng sắp xếp sẽ giảm ½, vậy cần $\log_2 n$ lần phân hoạch.
  - Chí phí trung bình là $O(n\log_2 n)$

Phương pháp Merge Sort (tt)

- Ví dụ:



Phương pháp Merge Sort (tt)

- Cải tiến:
  - Trộn k dãy
  - Trộn dãy có thứ tự
  - ...

Phương pháp Radix Sort

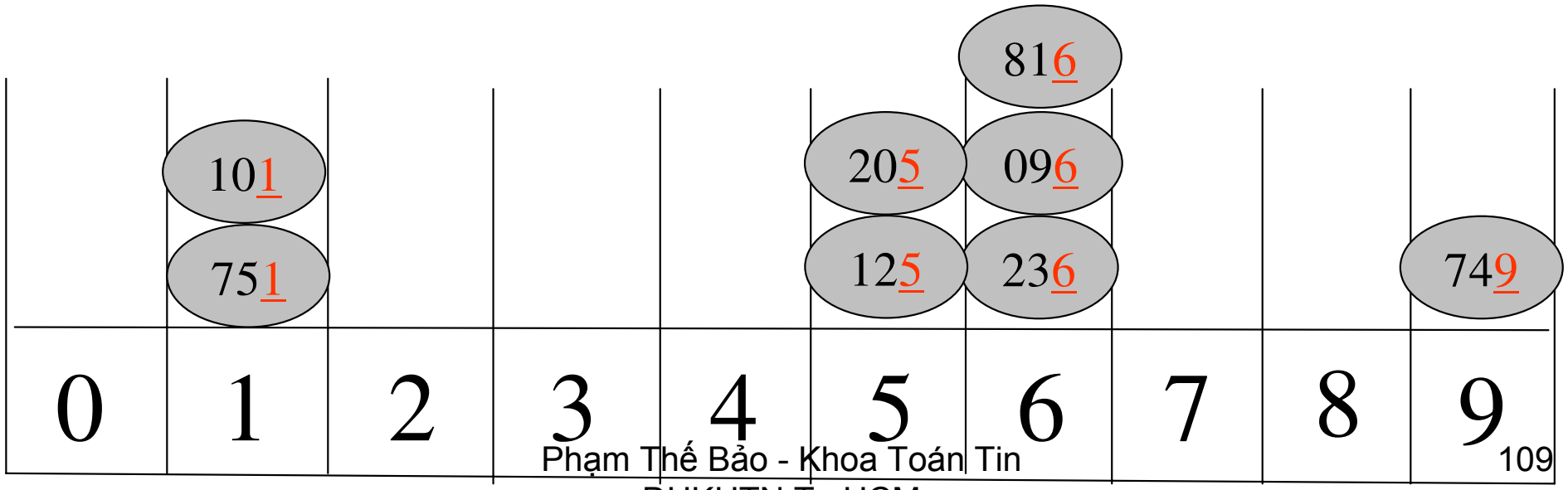
- Ý tưởng thuật toán:

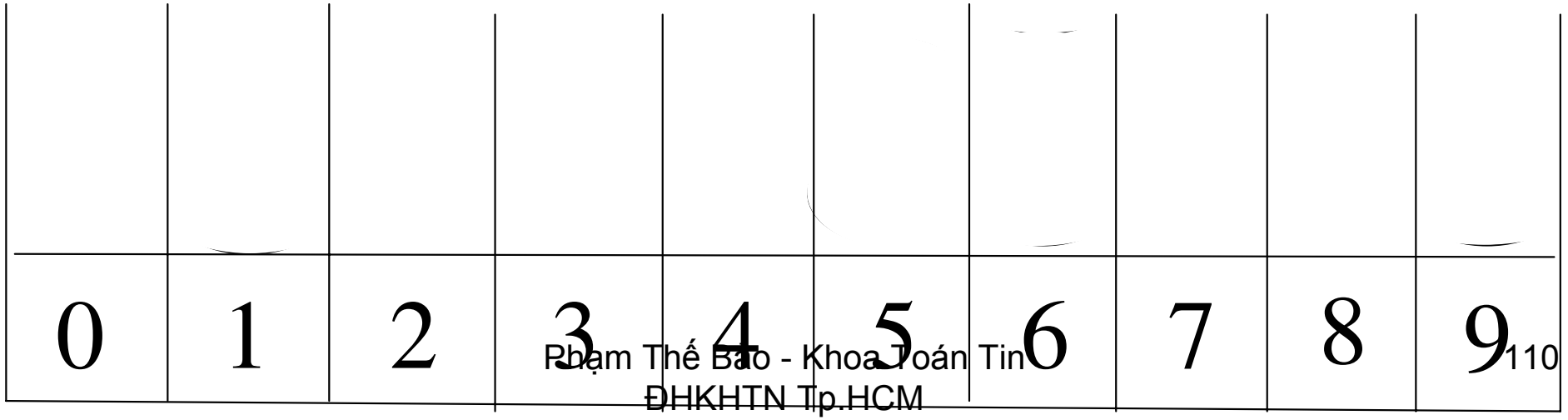
  Radix Sort là thuật toán tiếp cận theo ý tưởng không so sánh giá trị các phần tử; cơ sở để sắp xếp của các thuật toán cũ; mà dùng nguyên tắc phân loại thư như bưu điện. Do đó thuật toán này còn có tên là Postnam's Sort.

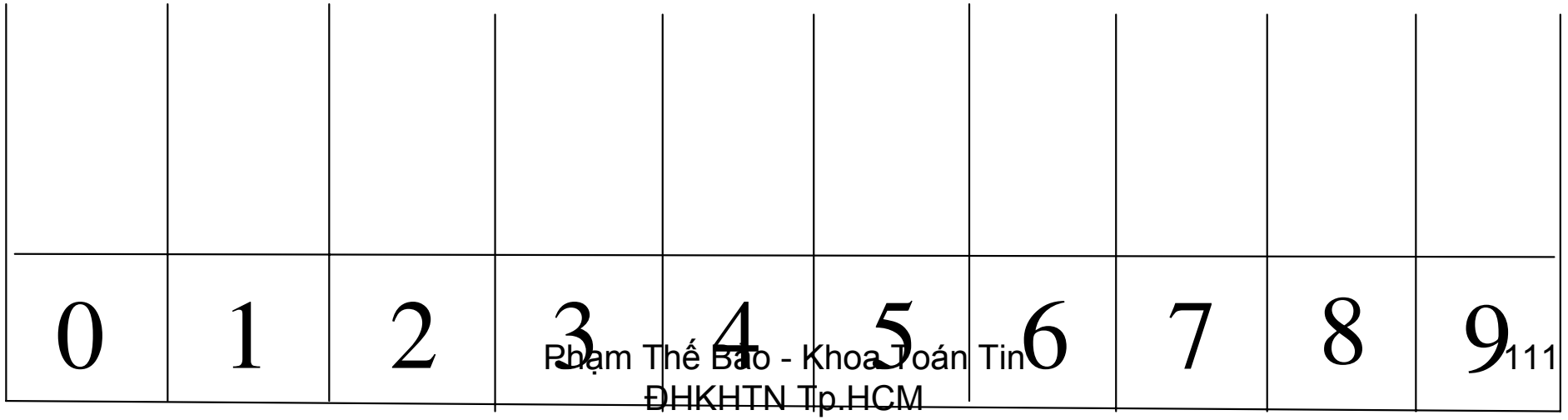
  Nguyên tắc chính là phân loại và trình tự phân loại sẽ tạo ra thứ tự cho các phần tử.

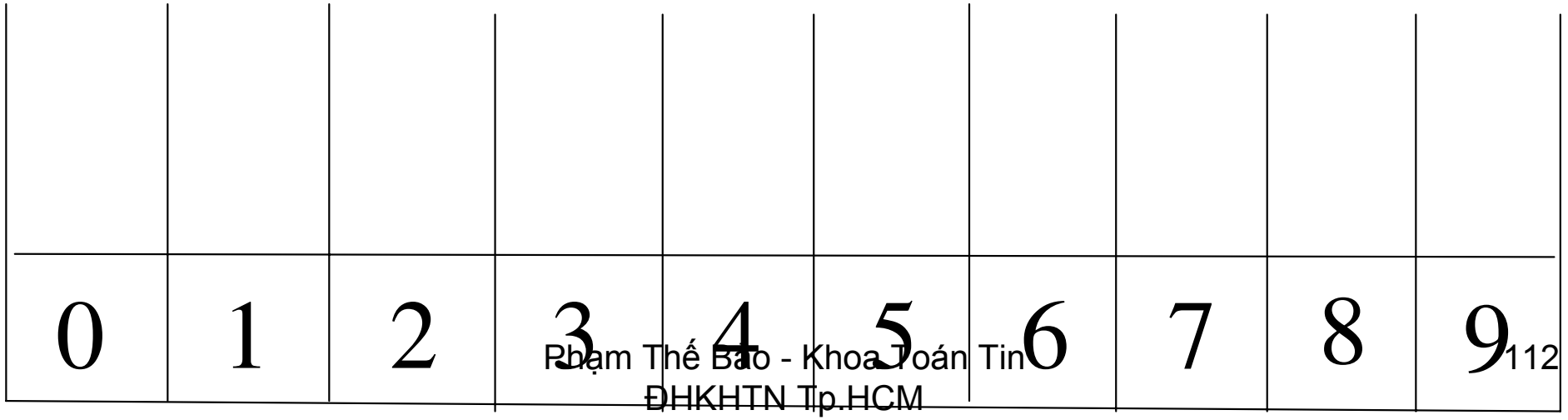
Phương pháp Radix Sort (tt)

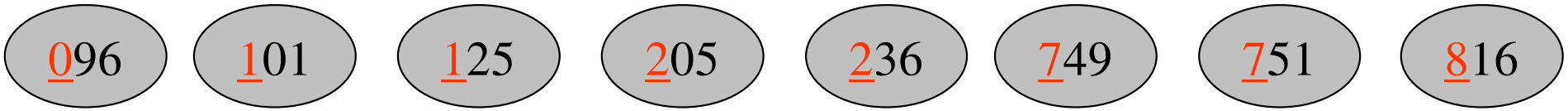
- Ví dụ:




0 1 2 3 4 5 6 7 8 9


0 1 2 3 4 5 6 7 8 9


0 1 2 3 4 5 6 7 8 9



Kết thúc → Mảng có thứ tự

Phương pháp Radix Sort (tt)

- Thuật toán:

Thuật toán RadixSort(A,n,k)

Input: Mảng A có n phần tử, có tối đa k chữ số.

Output: Mảng A được sắp xếp.

Khởi tạo 10 ngăn chứa B[0..9] rỗng;

for (t ← 0 to k-1) do

for (j ← 0 to n-1) do

Thêm A[j] vào B[Digit(A[j],t)];

for (k ← 0 to 9) do

Lấy ngược các phần tử từ B[i] đưa vào A

return;

Phương pháp Radix Sort (tt)

Thuật toán Digit(n,k)

Input: số nguyên n và k.

Output: giá trị tại vị trí k của số n.

```
value ← 1;
for (i ← 0 to k-1 ) do
        value ←  value * 10;
return (n / value)% 10;
```

Phương pháp Radix Sort (tt)

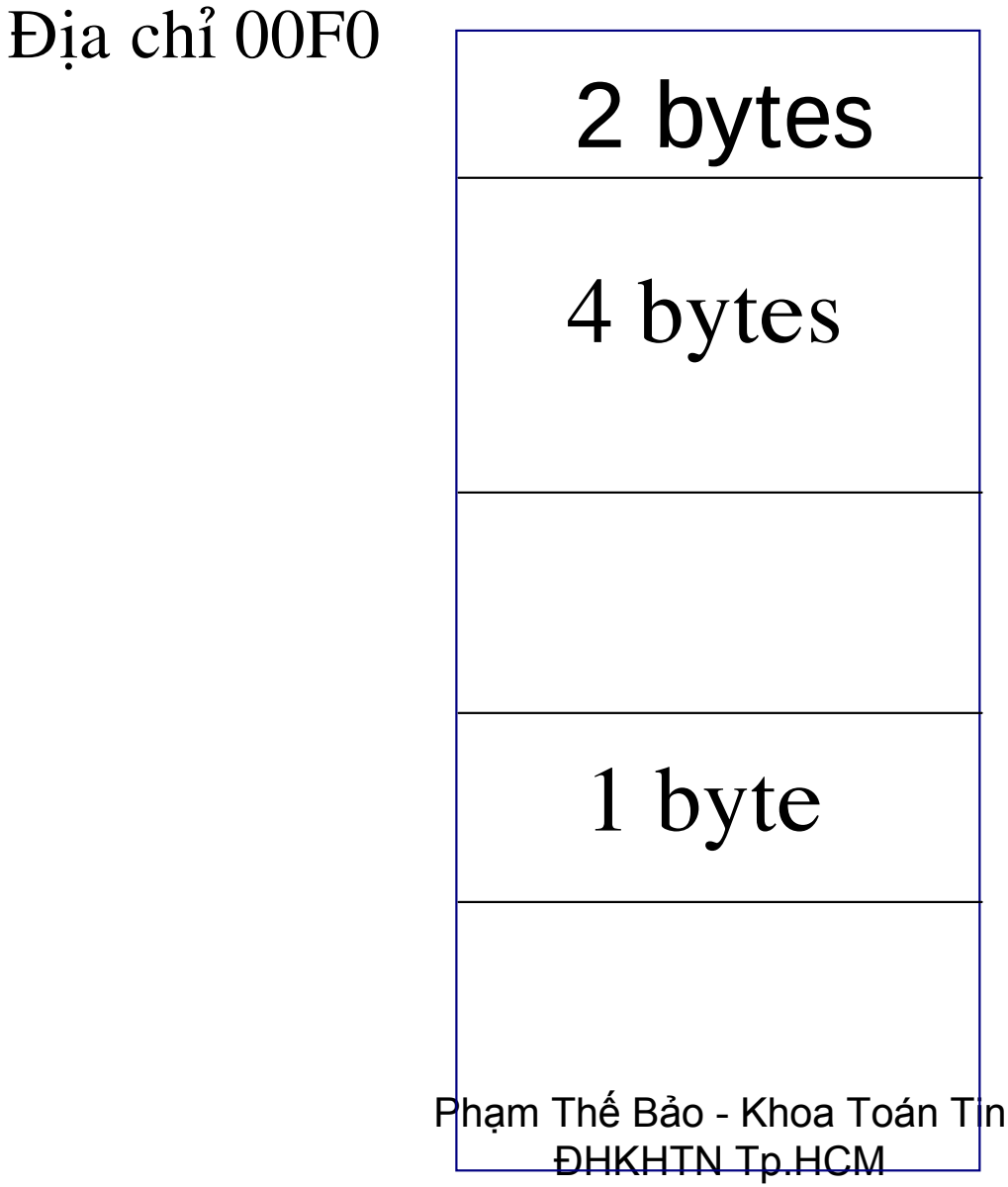
- Nhận xét:
  - Độ phức tạp của thuật toán là O(n).
  - Tốn kém, vì phải có 10 ngăn chứa mà mỗi ngăn chứa phải có tối đa đúng bằng số phần tử của dữ liệu với kiểu dữ liệu số, và nhiều hơn nữa với kiểu dữ liệu chuỗi.
  - Thuận tiện cài đặt với các kiểu dữ liệu chỉ so sánh trên số, ký tự, chuỗi.
  - Rất thích hợp cho sắp xếp trên cấu trúc dữ liệu danh sách liên kết

DANH SÁCH LIÊN KẾT
(LINKED LIST)

- Đặc tính của CTDL mảng:

1. Ưu điểm
   - Tốc độ xử lý cao do truy cập ngẫu nhiên.
   - Dễ dàng cài đặt, sử dụng.
2. Khuyết điểm
   - Cấp phát tĩnh, khi thì phí phạm bộ nhớ khi thì không đủ.
   - Đỏi hỏi vùng nhớ liên tục.

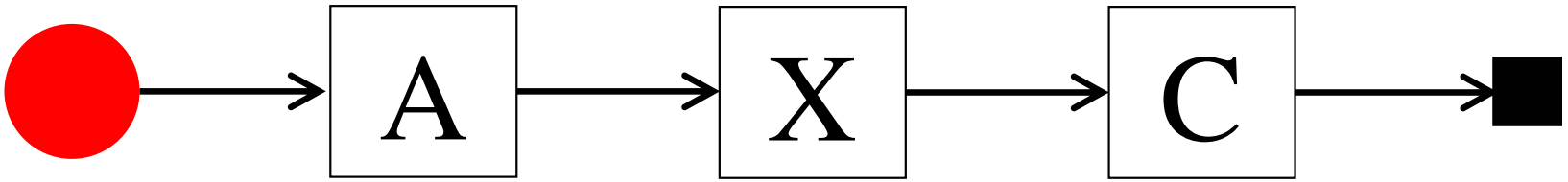
Mô hình cấp phát bộ nhớ



Cấu trúc dữ liệu DSLK

- Định nghĩa:

  Là một CTDL gồm một dãy các phần tử (gọi là node) có cùng cấu trúc được liên kết với nhau tạo thành một chuỗi tuyến tính.



Cấu trúc dữ liệu DSLK (tt)

- Tính chất:
  1. DSLK sẽ có một số đặc tính như mảng 01 chiều. Đây là CTDL động, giúp tránh được nhược điểm về bộ nhớ.
  2. Việc truy xuất dữ liệu thông qua phần tử đầu DSLK, và tuần tự từ phần tử này đến phần tử khác thông qua mối liên kết giữa các node. Node cuối cùng của DSLK sẽ trỏ đến một địa chỉ (giá trị) đặc biệt (với C là NULL). Tại sao phai trỏ đến 01 địa chỉ đặc biệt ?
  3. DSLK là một CTDL tuyến tính, truy cập tuần tự, kích thước thay đổi theo nhu cầu.
  4. Không cần vùng nhớ liên tục, hiệu quả ngay khi vùng nhớ bị phân mảnh.

Cấu trúc dữ liệu DSLK (tt)

- Các thao tác thường dùng:
  1. Khởi tạo DSLK
  2. Kiểm tra DSLK rỗng hay không ?
  3. Thêm một phần tử vào DSLK, thêm: vào đầu, cuối, giữa.
  4. Huỷ một phần tử ra khỏi DSLK, huỷ: ở đầu, cuối, giữa.
  5. Tìm một phần tử trong DSLK.
  6. Sắp xếp DSLK.

Cấu trúc dữ liệu DSLK (tt)

- Khai báo:

  DSLK được định nghĩa như một CTDL đệ quy: mỗi phần tử gồm dữ liệu và một con trỏ trỏ đến địa chỉ của node kế tiếp.

- Quản lý:

  Để quản lý một DSLK ta sẽ thông qua một phần tử đặc biệt là phần tử đầu tiên.

  Từ phần tử đầu tiên này ta có thể truy cập bất cứ phần tử nào trong DSLK. Vì vậy dù làm gì ta sẽ không thể mất dấu phần tử đầu tiên nếu không ta sẽ không quản lý được DSLK.

  Trong nhiều trường hợp ta cần xử lý nhiều ở cuối DSLK ta có thể thêm phần tử cuối DSLK để dễ quản lý.

DANH SÁCH LIÊN KẾT ĐƠN

Khai báo kiểu CTDL

- Kiểu cấu trúc trong C

```
typedef struct tagNode{
    DataType Data;              // khai báo dữ liệu
    struct tagNode *Next;
};
typedef tagNode *Node
typedef struct tagList{
    Node Head;                  // phần tử đầu tiên
    Node Tail;                  // phần tử cuối, có thể không
                                // cần dùng cũng được
}LinkedList;
```

Khai báo kiểu CTDL (tt)

- Kiểu đối tượng trong C

```
class Node{
friend class LinkedList;
    private:
        DataType data;
        Node *Next;
    public:
        Node(DataType d=giá trị đặc biệt ) {
                data=d;Next=NULL;}
        ~Node(){if(Next)delete Next;}

};
typedef Node *LinkedNode;
```

```
class LinkedList{
    protected:
        LinkedNode Head,Tail;
    public:
        LinkedList(){Head=Tail=NULL;}
        // Vài constructor cần
        ~LinkedList(){if(Head)delete Head;}
        void insertNode(DataType);
        int searchNode(DataType);
        void deleteNode(DataType);
        void addTail(DataType);
        void printList();
        void addHead(DataType);
        void addAfter(DataType,DataType);
    private:
        void addTail(LinkedNode);
        void addHead(LinkedNode);
        LinkedNode createNode(DataType);
        void insertNode(LinkedNode);
        int searchNode(LinkedNode);
        int deleteNode(DataType);
        int addAfter(LinkedNode,DataType);
};
```

Các thao tác

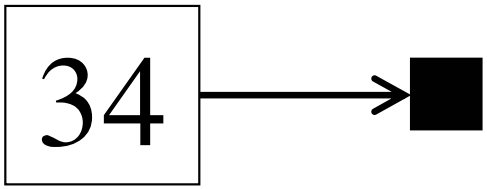
- Khởi tạo DSLK rỗng

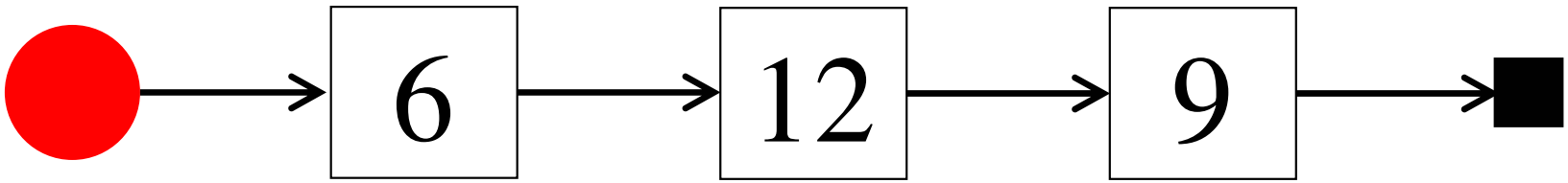
```
void initLinkedList(LinkedList &l){
        l.Head = l.Tail = NULL;
}
```

- Kiểm ra DSLK rỗng ?

```
int isEmpty(LinkedList l){
        return (l.Head==NULL);
}
```

Thêm một phần tử vào đầu DSLK đơn

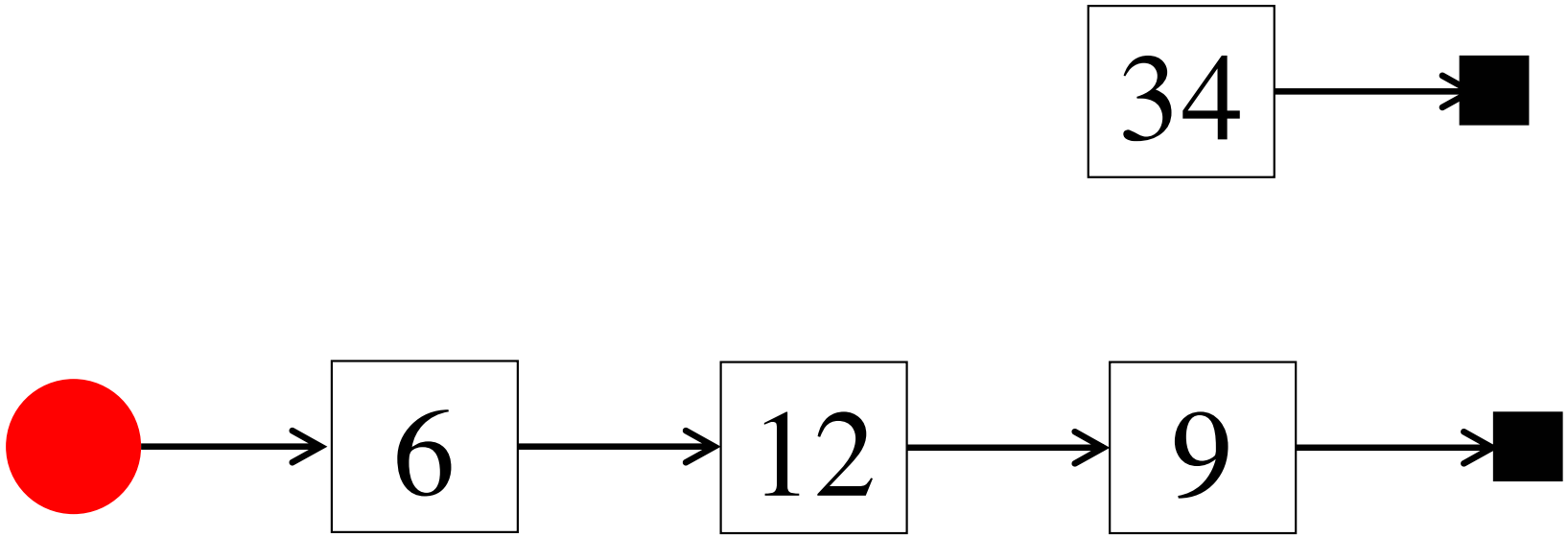




- Thêm 01 phần tử vào đầu DSLK

```
int insertNode2Head(LinkedList &l, DataType x){
        Node p = new tagNode;
        if(p==NULL) return 0;
        p->Data = x;
        p->Next = NULL;
        if(isEmpty(l)){
                l.Head = l.Tail = p;
                }
        else {
                p->Next = l.Head;
                l.Head = p;
                }
        return 1;
}
```

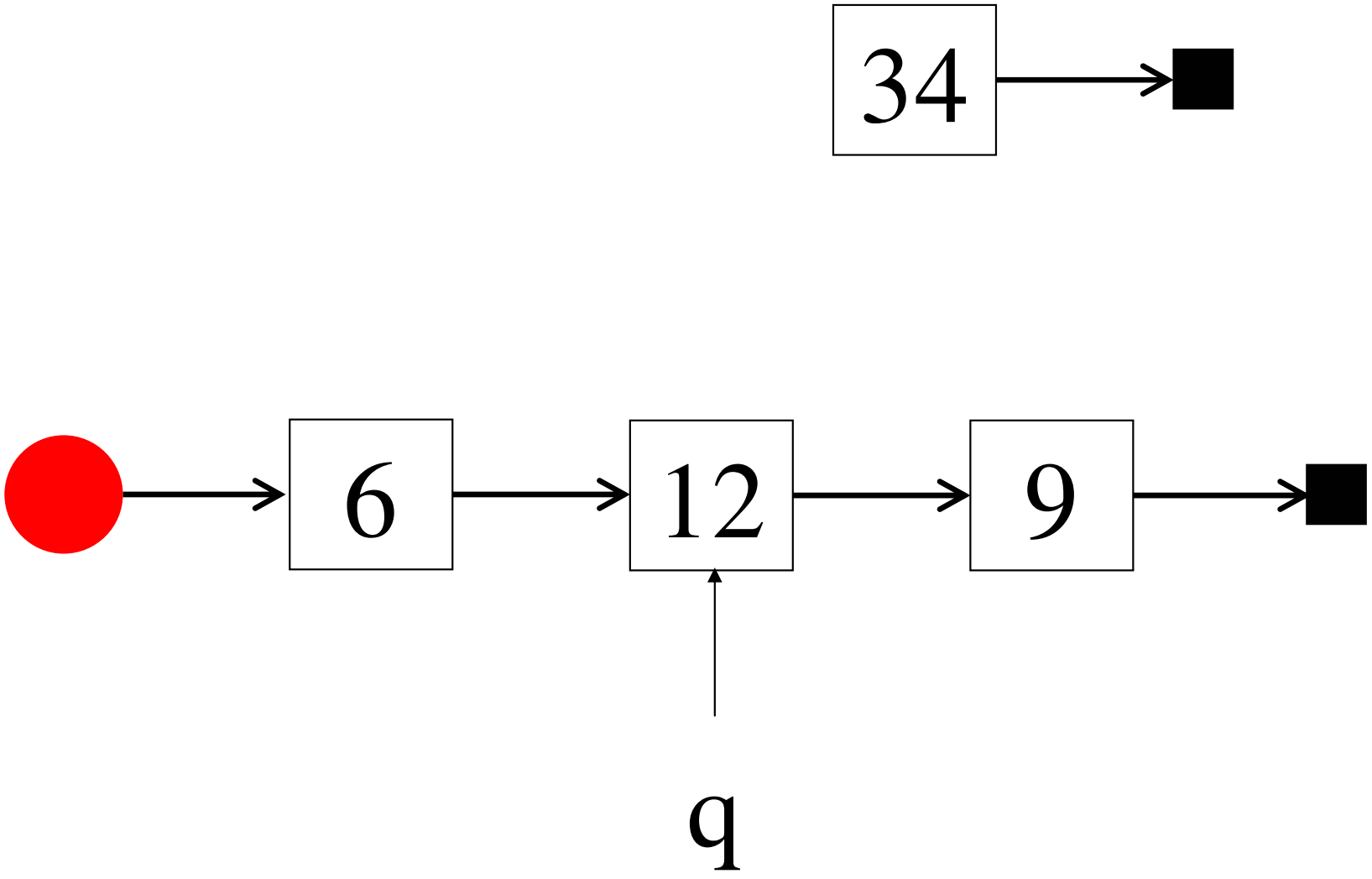
Thêm một phần tử vào cuối DSLK đơn



- Thêm 01 phần tử vào cuối DSLK

```
int insertNode2Tail(LinkedList &l, DataType x){
        Node p = new tagNode;
        if(p==NULL)return 0;
        p->Data = x;
        p->Next = NULL;
        if(isEmpty(l)){
                l.Head = l.Tail = p;
                }
        else{
                l.Tail->Next = p;
                l.Tail = p;
                }
        return 1;
}
```

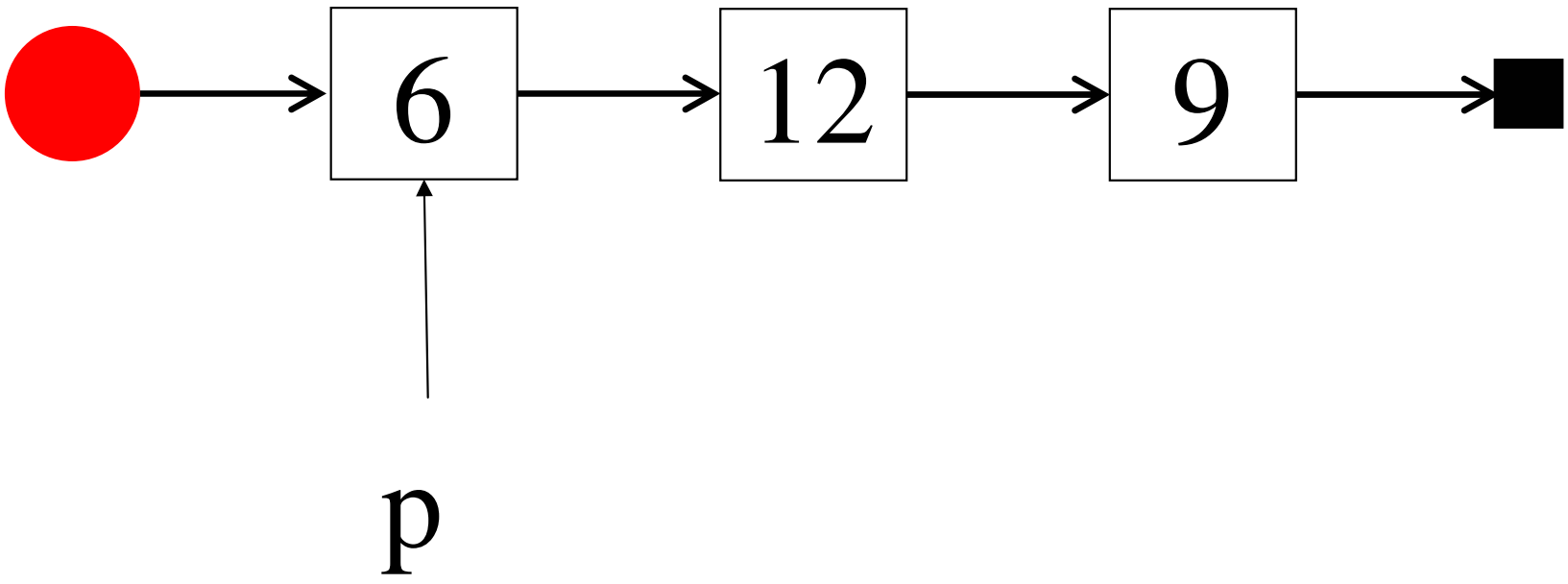
Thêm một phần tử vào sau phần tử q của DSLK đơn



- Thêm 01 phần tử vào sau node q trong DSLK

```
int insertNode2After(LinkedList &l, Node q, DataType x){
        Node p = new tagNode;
        if(p==NULL)return 0;
        p->Data = x;
        p->Next = NULL;
        if(isEmpty(l)){
                l.Head = l.Tail = p;
                }
        else{
                p->Next = q->Next;
                q->Next = p;
                if(q==l.Tail)l.Tail = p;
                }
        return 1;
}
```

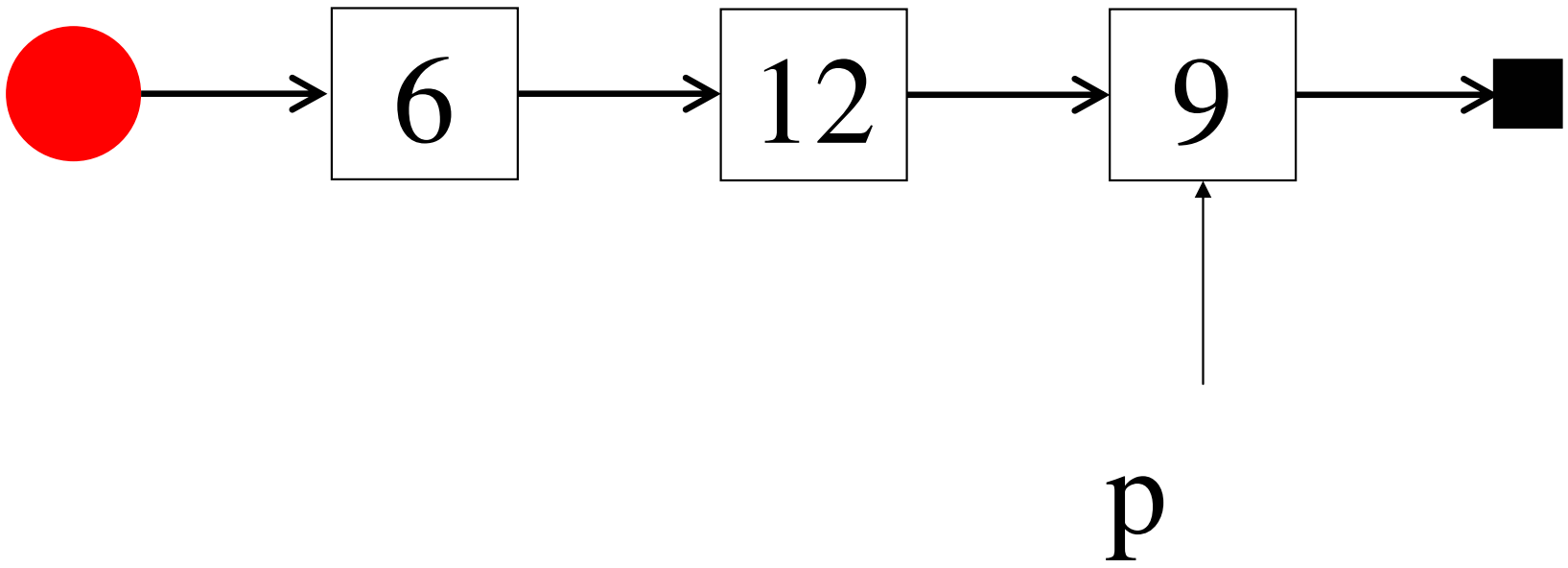
Hủy một phần tử ở đầu DSLK đơn



- Hủy 01 phần tử ở đầu DSLK

```
int removeNodeAtHead(LinkedList &l){
        Node p = l.Head;
        if(isEmpty(l)) return -1
        l.Head = p->Next;
        if(l.Head==NULL)l.Tail=NULL;
        p->Next = NULL;
        delete p;
        return 1;
}
```

Hủy một phần tử ở cuối DSLK đơn



- Hủy 01 phần tử ở cuối DSLK

```
int removeNodeAtTail(LinkedList &l){
        Node p = l.Head;
        Node q = NULL;
        while(p!=l.Tail){
                q = p;
                p = p->Next;
                }
        if(!p) return -1;
        if(q){
                q->Next = NULL;
                l.Tail = q;
                }
        else {
                l.Head = l.Tail = NULL;
                }
        delete p;
        return 1;
}
```

- Hủy 01 phần tử có giá trị X trong DSLK

```
int removeNode (LinkedList &l, DataType X){
        Node p = l.Head;
        if(!p) return -1;
        if(p->Data!=X && l.Tail->Data!=X){
                Node q= NULL;
                while(p->Data!=X && p){
                        q = p; p = p->Next;
                        }
                if(!p)return 0;
                q –> Next = p->Next;
                p->Next = NULL;
                delete p;
                }
        else{
                if(p->Data bằng X)return removeNodeAtHead(l);
                if(l.Tail->Data bằng X)return removeNodeAtTail(l);
                }
        return 1;
}
```

- Tìm 01 phần tử trong DSLK

```
Node searchNode(LinkedList l, DataType x){
        Node p = l.Head;
        if(p==NULL) return NULL;
        while (p){
                if(p->Data bằng x)return p;
                p = p->Next;
                }
        return NULL;
}
```

- Nối DSLK này vào cuối DSLK kia

```
void appendList(LinkedList &l, LinkedList &l1){
        If(isEmpty(l))l = l1;
        else{
                l.Tail->Next = l1.Head;
                if(!isEmpty(l1))l.Tail = l1.Tail;
                }
}
```

- Sắp xếp DSLK theo PP MergeSort

```
void mergeSort(LinkedList &l){
   LinkedList l1,l2;
   if(l.Head==l.Tail)return;
   initLinkedList(l1);
   initLinkedList(l2);
   divideList(l,l1,l2);
   mergeSort(l1);
   mergeSort(l2);
   mergeList(l,l1,l2);
}
```

Sắp xếp DSLK theo PP MergeSort (tt)

```
void divideList(LinkedList &l,LinkedList &l1,LinkedList &l2){
    Node p;
    while(!isEmpty(l)){
            p = l.Head;
            l.Head = p->Next;
            p->Next = NULL;
            insertNodeAtTail(l1,p);
            if(!isEmpty(l)){
                    p = l.Head;
                    l.Head = p->Next;
                    p->Next = NULL;
                    insertNodeAtTail(l2,p);
                    }
            }
    initLinkedList(l);
}
```

- Sắp xếp DSLK theo PP MergeSort (tt)

```
void mergeList(LinkedList &l,LinkedList &l1,LinkedList &l2){
    Node p;
    while(!isEmpty(l1)&&!isEmpty(l2)){
        if(l1.Head->Data nhỏ hơn bằng l2.Head->Data){
                p = l1.Head;
                l1.Head = p->Next;
                p->Next = NULL;
                }
        else{
                p = l2.Head;
                l2.Head = p->Next;
                p->Next = NULL;
                }
        insertNodeAtTail(l,p);
        }
    if(!isEmpty(l1))appendList(l,l1);
    if(!isEmpty(l2))appendList(l,l2);
    initLinkedList(l1);
    initLinkedList(l2);
}
```

- Sắp xếp DSLK theo PP QuickSort

```
void quickSort(LinkedList &l){
   LinkedList l1,l2;
   if(l.Head==l.Tail)return;
   initLinkedList(l1);
   initLinkedList(l2);
   splitList(l,l1,l2,X);
   quickSort(l1);
   quickSort(l2);
   concatList(l,l1,p,l2);
   initLinkedList(l1);
   initLinkedList(l2);
}
```

Sắp xếp DSLK theo PP QuickSort (tt)

```
void splitList(LinkedList &l, LinkedList &l1, LinkedList &l2,Node &x){
    Node p;
    x = l.Head;
    l.Head = x->Next;
    x->Next = NULL;
    while(!isEmpty(l)){
            x = l.Head;
            l.Head = x->Next;
            x->Next = NULL;
            if(p->Data nhỏ hơn bằng x->Data)insertNodeAtTail(l1,p);
            else insertNodeAtTail(l2,p);
            }
     initLinkedList(l);
}
```

Sắp xếp DSLK theo PP QuickSort (tt)

```
void concatList(LinkedList &l, LinkedList &l1,Node &x, LinkedList &l2){
    appendList(l,l1);
    insertNodeAtTail(l,x);
    appendList(l,l2);
    initLinkedList(l1);
    initLinkedList(l2);
}
```

- Sắp xếp DSLK theo PP RadixSort

```
void radixSort(LinkedList &l){
    LinkedList temp[10];
    if(l.Head==l.Tail)return;
    for(int i=0;i<10;i++)initLinkedList(temp[i]);
    int numDigit = getMaxDigit(l);
    for(i=0;i<numDigit;i++){
            send2Box(i,l,temp);
            for(int j=0;j<10;j++){
                    appendList(l,temp[j]);
                     initLinkedList(temp[j]);
                    }
            }
}
```

- Sắp xếp DSLK theo PP RadixSort (tt)

```
void send2Box(int n, LinkedList &l, LinkedList temp[]){
   Node p;
   while(!isEmpty(l)){
         p = getHead(l);
         addTail(temp[getDigit(n,p->Data)],p);
         }
    initLinkedList(l);
   }
```

- Sắp xếp DSLK theo PP RadixSort (tt)

```
int getDigit(int n, int t){
    long temp=1;
    for(int i=0;i<n;i++)temp*=10;
    return (t/temp)%10;
    }

int getMaxDigit(l){
    int max = findMax(l);
    return countDigit(max);
    }
```

- Sắp xếp DSLK theo PP RadixSort (tt)

```
int findMax(LinkedList l){

    }

int countDigit(int m){
    int count = 0;
    while(m){
        m/=10;
        count++;
        }
    return count;
    }
```

NGĂN XẾP
(STACK)

CTDL Ngăn xếp

- Định nghĩa:

  là một vật chứa (container) các đối tượng làm việc theo cơ chế **LIFO** (**L**ast **I**n **F**irst **O**ut), nghĩa là đối tượng được đưa vào sau sẽ được lấy ra trước.

CTDL Ngăn xếp (tt)

- Tính chất:
  1. Các đối tượng được thêm vào bất kỳ lúc nào nhưng chỉ có đối tượng được thêm sau cùng mới được phép lấy ra.
  2. Thao tác thêm 01 phần tử được gọi là “Push”, thao tác lấy 01 phần tử ra là “Pop”.

CTDL Ngăn xếp (tt)

- Ví dụ:

  Băng đạn súng AK, tháp Hà Nội, ...

- Ứng dụng:

  Khử đệ quy, tổ chức các quá trình tìm kiếm theo chiều sâu và quay lui, vét cạn, tính toán biểu thức, ...

CTDL Ngăn xếp (tt)

- Định nghĩa

Stack là một CTDL ADT hỗ trợ 02 thao tác chính:

1. push(o): thêm đối tượng o vào đầu stack
2. pop(): lấy đối tượng ở đầu stack ra.

- Thao tác hỗ trợ:

1. size(): số phần tử trong stack
2. isEmpty(): kiểm tra stack rỗng
3. top(): trả giá trị phần tử ở đầu stack

CTDL Ngăn xếp (tt)

- CTDL
  1. Mảng
  2. DSLK
- Cài đặt
  1. Mảng 01 chiều: khai báo 01 mảng 01 chiều có kích thước N phần tử, và các phần tử được đánh số từ 0 đến N-1.

CTDL Ngăn xếp (tt)

2. DSLK đơn: DSLK có những đặc tính phù hợp để làm stack

STT	Stack	DSLKD
1	push	insertHead
2	pop	removeHead
3	size	getCount
4	isEmpty	isEmpty
5	top	getHead

CTDL Ngăn xếp (tt)

```
typedef LinkedList Stack;
int isEmpty(Stack st){
      return st.Head==NULL;
      }
int size(Stack st){
      Node p = st.Head;
      int count = 0;
      while(p){
         count++;
         p = p->Next;
         }
      }
```

CTDL Ngăn xếp (tt)

```
DataType top(Stack st){
    return st.Head->Data;
    }
int push(Stack st, DataType x){
    // chèn vào đầu DSLK st
    }
DataType pop(Stack st){
    // Lấy phần tử đầu DSLK st
    }
```

THUẬT TOÁN
BA LAN NGƯỢC

Thuật toán Ba Lan ngược (Reverse Polish Notation - RPN)

Bài toán:

$$(1+5)*(12 - (8+1))$$

$$\xrightarrow{\text{Biến đổi}} 1\ 5\ +\ 12\ 8\ 1\ +\ -\ *$$

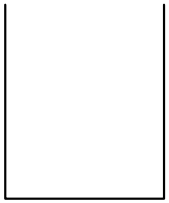
Thuật toán chuyển đổi:

1. Khởi tạo Stack rỗng (chứa các phép toán).
2. Lặp cho đến khi kết thúc biểu thức:
   - Đọc 01 phần tử của biểu thức (01 phần tử có thể là hằng, biến, phép toán, “)” hay “(”).
   - Nếu phần tử là:
     - ✓ “(”: đưa vào Stack.
     - ✓ “)”: lấy các phần tử của Stack ra cho đến khi gặp “(” trong Stack.
     - ✓ Một phép toán:
       - ❖ Nếu Stack rỗng: đưa vào Stack.
       - ❖ Nếu Stack khác rỗng và phép toán có độ ưu tiên cao hơn phần tử ở đầu Stack: đưa vào Stack.
       - ❖ Nếu Stack khác rỗng và phép toán có độ ưu tiên thấp hơn hoặc bằng phần tử ở đầu Stack: lấy phần tử từ Stack ra; sau đó lặp lại việc so sánh với phần tử ở đầu Stack.
     - ✓ Hằng hoặc biến: Lấy ra.
3. Lấy hết tất cả các phần tử của Stack ra.

(Ta xem “(” có độ ưu tiên thấp hơn độ ưu tiên của các phép toán)

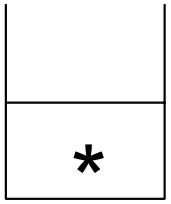
Ví dụ

7*8-(2+3)



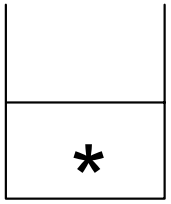
7

*8-(2+3)



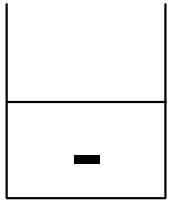
7

8-(2+3)



7 8

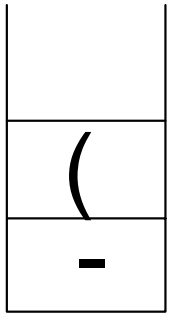
-(2+3)



7 8 *

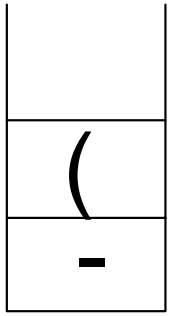
Lấy * ra, đưa - vào

(2+3)



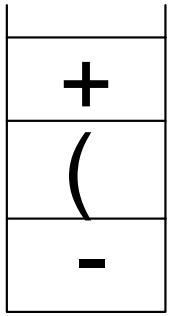
7 8 *

2+3)



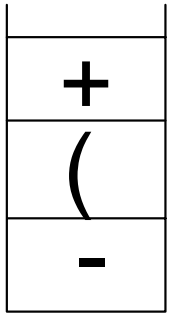
7 8 * 2

+3)



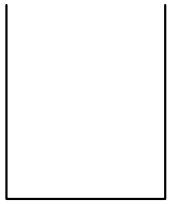
7 8 * 2

3)



7 8 * 2 3

)



Lấy + (- ra

7 8 * 2 3 + -

7*8-(2+3)

Kết Quả

7 8 * 2 3 + -

Thuật toán tính giá trị biểu thức Ba lan ngược

7 8 * 2 3 + -

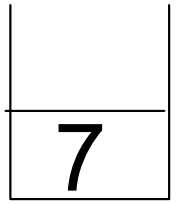
51

Thuật toán tính giá trị:

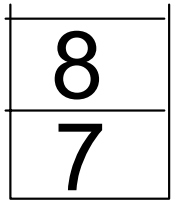
1. Khởi tạo Stack rỗng (chứa hằng hoặc biến).
2. Lặp cho đến khi kết thúc biểu thức:
   - Đọc 01 phần tử của biểu thức (01 phần tử có thể là hằng, biến, phép toán).
   - Nếu phần tử là hằng hay biến: đưa vào Stack.
   - Ngược lại:
     - ✓ Lấy ra 02 phần tử của Stack.
     - ✓ Áp dụng phép toán cho 02 phần tử vừa lấy ra.
     - ✓ Đưa kết quả vào Stack.
3. Giá trị của biểu thức chính là phần tử cuối cùng của Stack.

Ví dụ

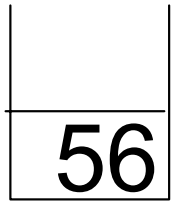
7 8 * 2 3 + -



8 * 2 3 + -

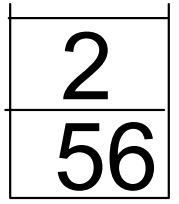


*2 3 + -

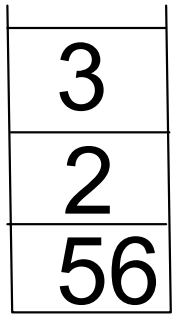


Thực hiện phép toán *

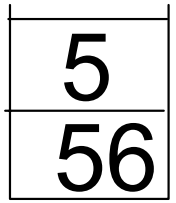
2 3 + -



3 + -

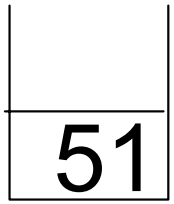


+ -



Thực hiện phép toán +

-



Thực hiện phép toán -

7 8 * 2 3 + -

Kết Quả →

51

HÀNG ĐỢI
(QUEUE)

CTDL Hàng đợi

- Định nghĩa:

  Là một vật chứa (Container) các đối tượng làm việc theo cơ chế **FIFO** (**F**irst **I**n **F**irst **O**ut), nghĩa là phần tử nào vào trước sẽ là phần tử ra trước.

CTDL Hàng đợi (tt)

- Tính chất:
  1. Các đối tượng được thêm vào bất kỳ lúc nào ở cuối nhưng chỉ có đối tượng được thêm vào đầu tiên mới được phép lấy ra.
  2. Thao tác thêm 01 phần tử được gọi là "enqueue", thao tác lấy 01 phần tử ra là "dequeue".

CTDL Hàng đợi (tt)

- Ví dụ:

Việc xếp hàng mua vé, quản lý và phân phối tiến trình trong hệ điều hành, ...

- Ứng dụng:

Khử đệ quy, tổ chức các quá trình tìm kiếm theo chiều rộng và quay lui, vét cạn, tổ chức quản lý và phân phối tiến trình trong các hệ điều hành, tổ chức bộ đệm của bàn phím, ...

CTDL Hàng đợi (tt)

- Định nghĩa

Queue là một CTDL ADT hỗ trợ 02 thao tác chính:

1. enqueue(o): thêm đối tượng o vào cuối queue
2. dequeue(): lấy đối tượng ở đầu queue ra.

- Thao tác hỗ trợ:

1. size(): số phần tử trong queue
2. isEmpty(): kiểm tra queue rỗng
3. front(): trả giá trị phần tử ở đầu queue

CTDL Hàng đợi (tt)

- CTDL
  1. Mảng
  2. DSLK
- Cài đặt
  1. Mảng 01 chiều: khai báo 01 mảng 01 chiều có kích thước N phần tử, và các phần tử được đánh số từ 0 đến N-1.

CTDL Hàng đợi (tt)

2. DSLK đơn: DSLK có những đặc tính phù hợp để làm stack

STT	Queue	DSLKD
1	enqueue	insertTail
2	dequeue	removeHead
3	size	getCount
4	isEmpty	isEmpty
5	front	getHead

CTDL Hàng đợi (tt)

```
typedef LinkedList Queue;
int isEmpty(Queue qu){
      return qu.Head==NULL;
      }
int size(Queue qu){
      Node p = qu.Head;
      int count = 0;
      while(p){
         count++;
         p = p->Next;
         }
      }
```

CTDL Hàng đợi (tt)

```
DataType front(Queue qu){
    return qu.Head->Data;
    }
int enqueue (Queue qu, DataType x){
    // chèn vào cuối DSLK qu
    }
DataType dequeue (Queue qu){
    // Lấy phần tử đầu DSLK qu
    }
```

DSLK CÓ THỨ TỰ
(ORDERED LIST)

CTDL DSLK Có thứ tự

- Định nghĩa:

  Là một vật chứa (Container) các đối tượng theo một trình tự nhất định. Khi thêm phần tử ta phải đảm bảo quan hệ thứ tự không thay đổi.

DSLK VÒNG

CTDL DSLK Đơn vòng

- Định nghĩa:

  Là một DSLK mà phần tử kế tiếp của phần tử cuối DSLK chính là phần tử đầu DSLK.

DSLK Đơn vòng (tt)

- Thêm ở đầu
  - Như bình thường
  - l.Tail->Next = l.Head
- Thêm ở cuối
  - Như bình thường
  - l.Tail->Next = l.Head
- Thêm ở giữa
  - Như bình thường

DSLK Đơn vòng (tt)

- Hủy ở đầu
  - Như bình thường
  - l.Tail->Next = l.Head
- Hủy ở cuối
  - Như bình thường
  - l.Tail->Next = l.Head
- Hủy ở giữa
  - Như bình thường

DSLK Đơn vòng (tt)

- Duyệt

```
p = l.Head;
do{
   // xử lý, ví dụ in giá trị các node: printf("%d",p->Data);
   p = p->Next;
}while(p!=l.Head);
```

DSLK ĐÔI
(DOUBLE LINKED LIST)

DSLK Đôi

- Định nghĩa:

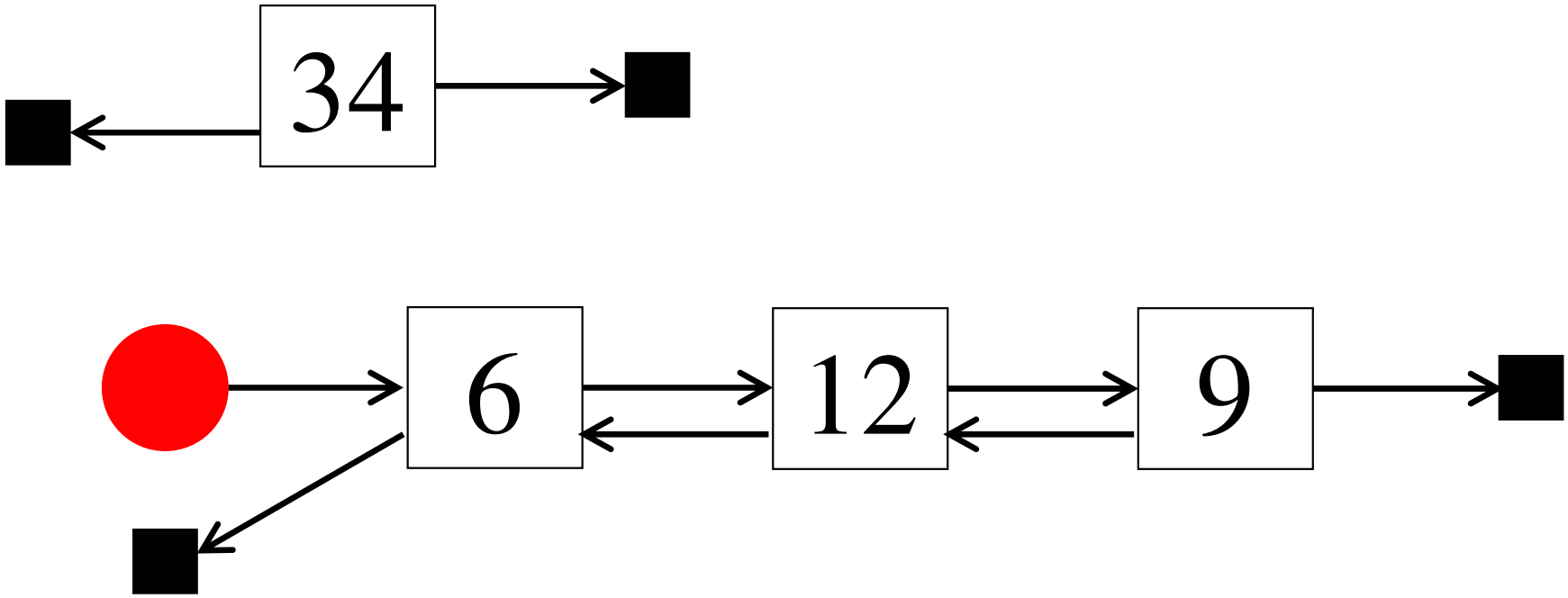
  Là DSLK mà mỗi phần tử sẽ có 02 mối liên kết.

- Khai báo:

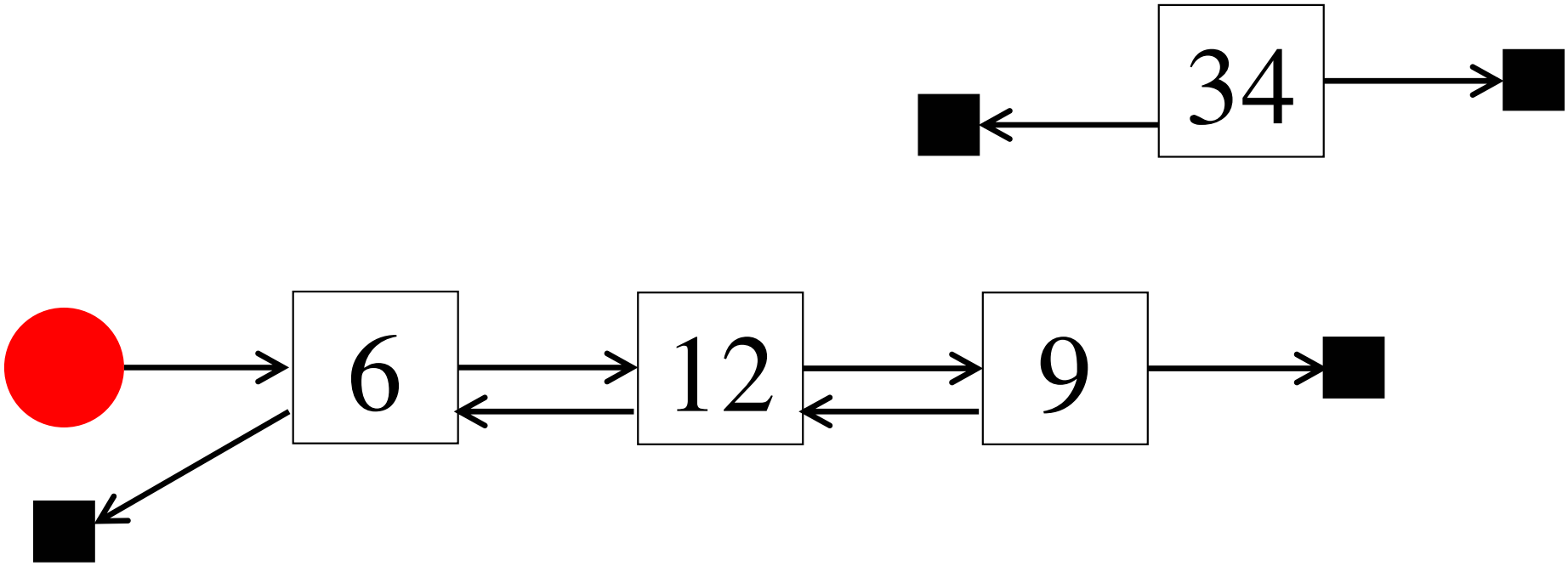
```
typedef struct tagNode{
DataType Data;
tagNode *Next;
tagNode *Prev;
};
```

```
typedef struct tagNode{
DataType Data;
tagNode *Left;
tagNode *Right;
};
```

Thêm một phần tử vào đầu DSLK đôi



Thêm một phần tử vào cuối DSLK đôi



DSLK Đôi (tt)

- Thêm ở đầu

```
p->Next = l.Head;
l.Head->Prev = p;
l.Head = p;
```

- Thêm ở cuối

```
l.Tail->Next = p;
p->Prev = l.Tail;
l.Tail = p;
```

DSLK Đôi (tt)

- Thêm p ở sau phần tử q

```
p->Next = q->Next;
p->Prev = q;
q->Next->Prev = p;
q->Next = p;
```

DSLK Đôi (tt)

- Hủy ở đầu

```
p = l.Head;
l.Head = p->Next;
l.Head->Prev = NULL;
p->Next = NULL;
delete p;
```

DSLK Đôi (tt)

- Hủy ở cuối

```
p = l.Tail;
Tìm q trước p
l.Tail = q;
l.Tail->Next = NULL;
p->Next = p->Prev = NULL;
delete p;
```

DSLK Đôi (tt)

- Hủy p ở sau q

```
q->Next = p->Next;
p->Next->Prev = q;
p->Next = p->Prev = NULL;
delete p;
```

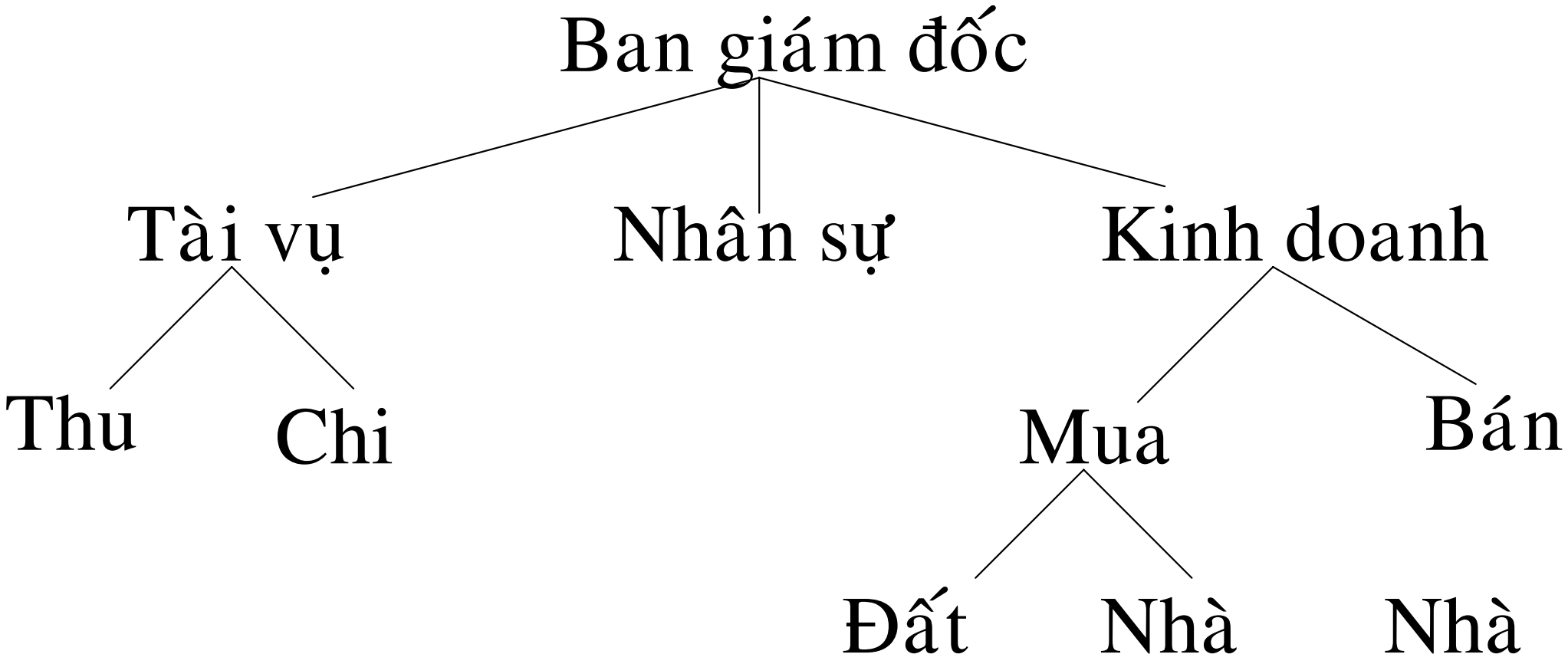
DANH SÁCH LIÊN KẾT ĐA

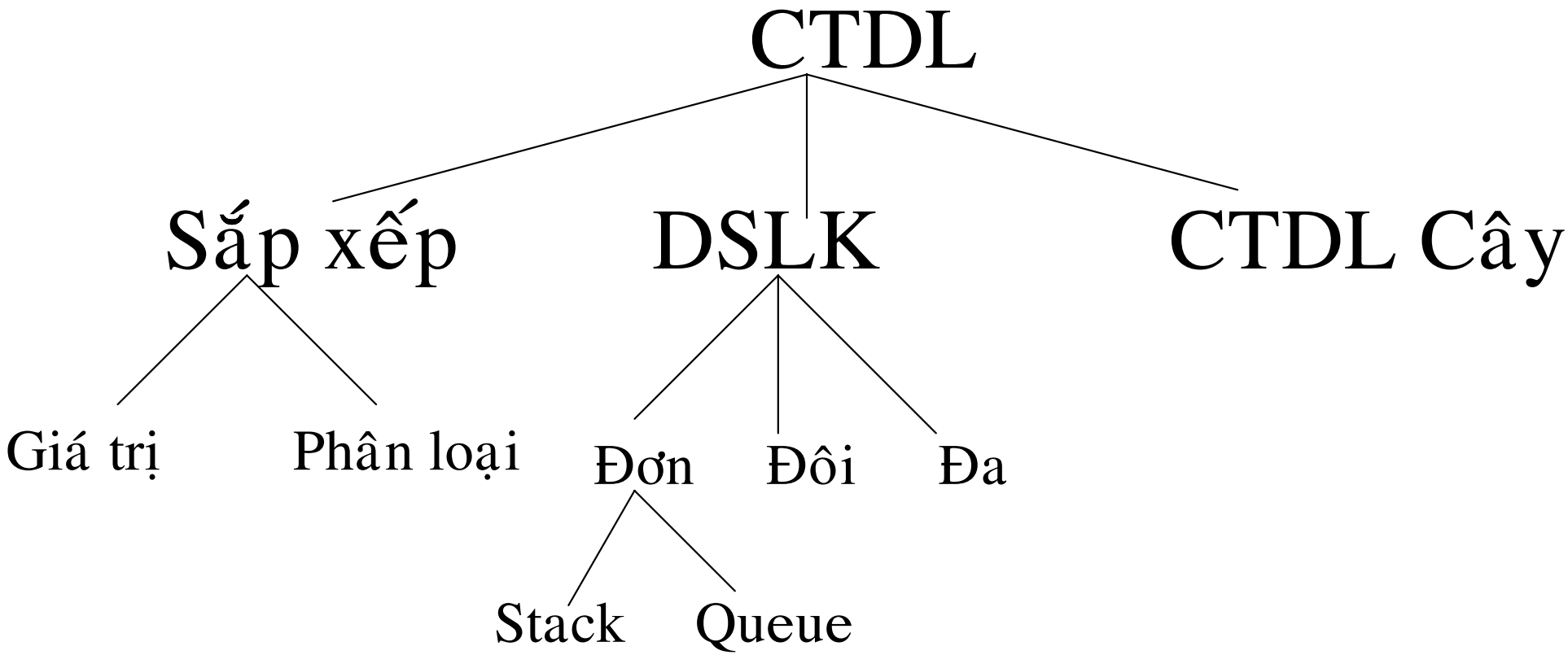
CẤU TRÚC DỮ LIỆU
CÂY
(TREE)

- Giới thiệu CTDL cây:

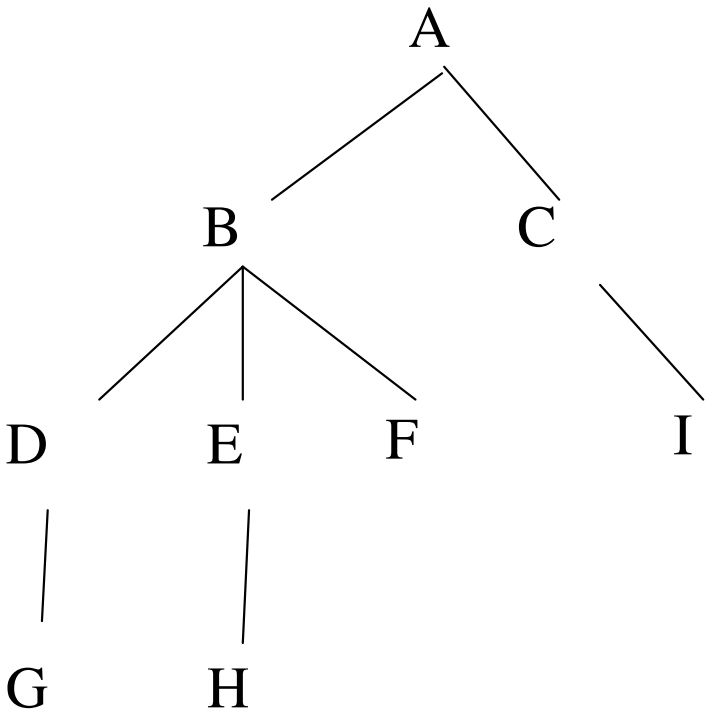
Cây là một CTDL biểu diễn các mô hình phân cấp như :

- Sơ đồ tổ chức công ty
- Mục lục của sách
- Cấu trúc thư mục trong DOS/WIN/LINUX ...
- ...


Ban giám đốc
Tài vụ
Nhân sự
Kinh doanh
Thu
Chi
Mua
Bán
Đất
Nhà
Nhà


CTDL
Sắp xếp
DSLK
CTDL Cây
Giá trị
Phân loại
Đơn
Đôi
Đa
Stack
Queue

Một số thuật ngữ

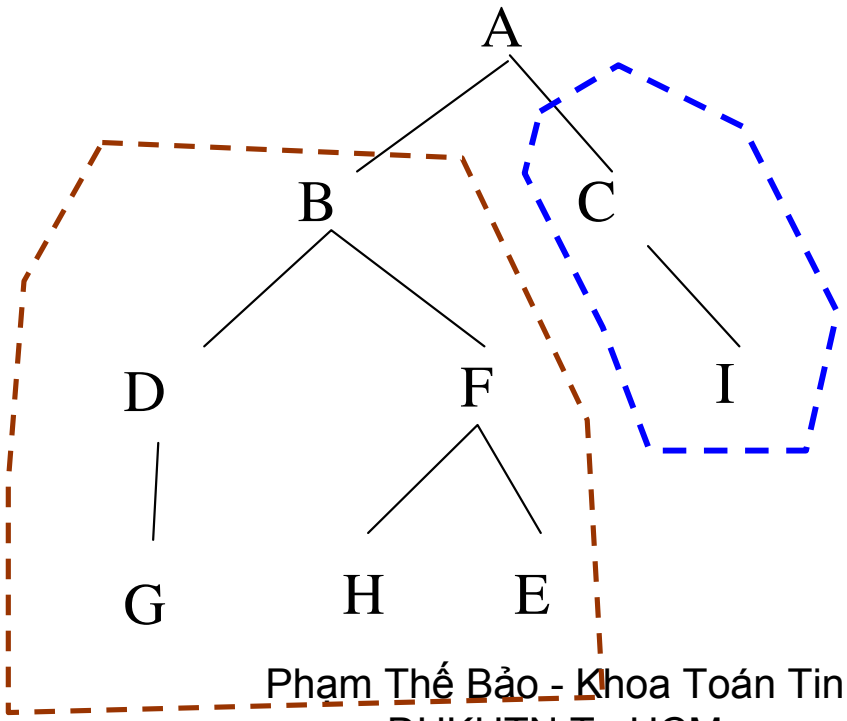


- ✓ Cây gồm tập các node. Hai node nối với nhau bởi 01 cạnh.
- ✓ Node A gọi là node gốc.
- ✓ B là cha của D, E, và F
- ✓ C là anh em của B
- ✓ G, H, F, và I là node ngoài, hay node lá.
- ✓ A, B, C, D, và E là node trong.
- ✓ Cây có chiều cao h = 4
- ✓ Bậc của node B là d(B) = 3
- ✓ Node E năm ở độ sâu 2 hay có mức là 2.
- ✓ Cây có N node thì sẽ có N-1 cạnh

- Nhược điểm:
  - ✓ Khi thêm phần tử, ta sẽ thêm bất kỳ, nên khi quản lý sẽ khó khăn.
  - ✓ Bậc các node trên cây giao động trong biên độ lớn nên rất khó khăn việc quản lý
  - ✓ Việc xoá sẽ mất nhiều thời gian tìm kiếm và khó khăn tổ chức lại khi xoá node trong.

CÂY NHỊ PHÂN
(BINARY TREE)

- Định nghĩa:
  1. Không đệ quy: Cây nhị phân là cây mà nỗi node có bậc tối đa là 2.
  2. Theo đệ quy:
     - Hoặc là một node lá,
     - Hoặc cây gồm 01 node gốc và 02 cây nhị phân con của nó (cây con trái và cây con phải).



- Tính chất:
  1. Số node nằm ở mức $i \leq 2^i$
  2. Số node lá $\leq 2^{h-1}$ với h là chiều cao của cây.
  3. Chiều cao của cây $h \geq \log_2(N)$ với N là số node trong cây.
  4. Số node trong cây $\leq 2^h - 1$

- Cài đặt bằng cấu trúc trong C:

```
typedef struct tagNode{
    DataType        Data;
    tagNode         *Left;
    tagNode         *Right;
};

typedef tagNode *BinaryTree;
```

- Cài đặt bằng đối tượng trong C:

```
class tagNode{
        friend class BinaryTree;
        private:
                DataType        Data;
                tagNode         *Left,*Right;
        public:
                tagNode(Data d=giá trị mặc định)
                        {Left=Right=NULL;Data = d}
                ~tagNode()
                        {if(Left)delete Left;if(Right)delete Right;}
};
typedef tagNode *Node;
```

```
class BinaryTree{
  private:
      Node Root;
  public:
      BinaryTree(){Root=NULL;}
      ~BinaryTree(){if(Root)delete Root;}
      // Các phương thức thêm, xoá, tìm kiếm,
      // đếm, ...
};
```

- Các thao tác thông thường:
  1. Kiểm tra cây rỗng
  2. Kiểm tra 01 node có phải là node lá không ?
  3. Duyệt cây
  4. Tìm kiếm 01 phần tử

- # Duyệt cây

1. Duyệt theo thứ tự trước (NLR)

Thuật toán NLR(T)

Input: Cây nhị phân T

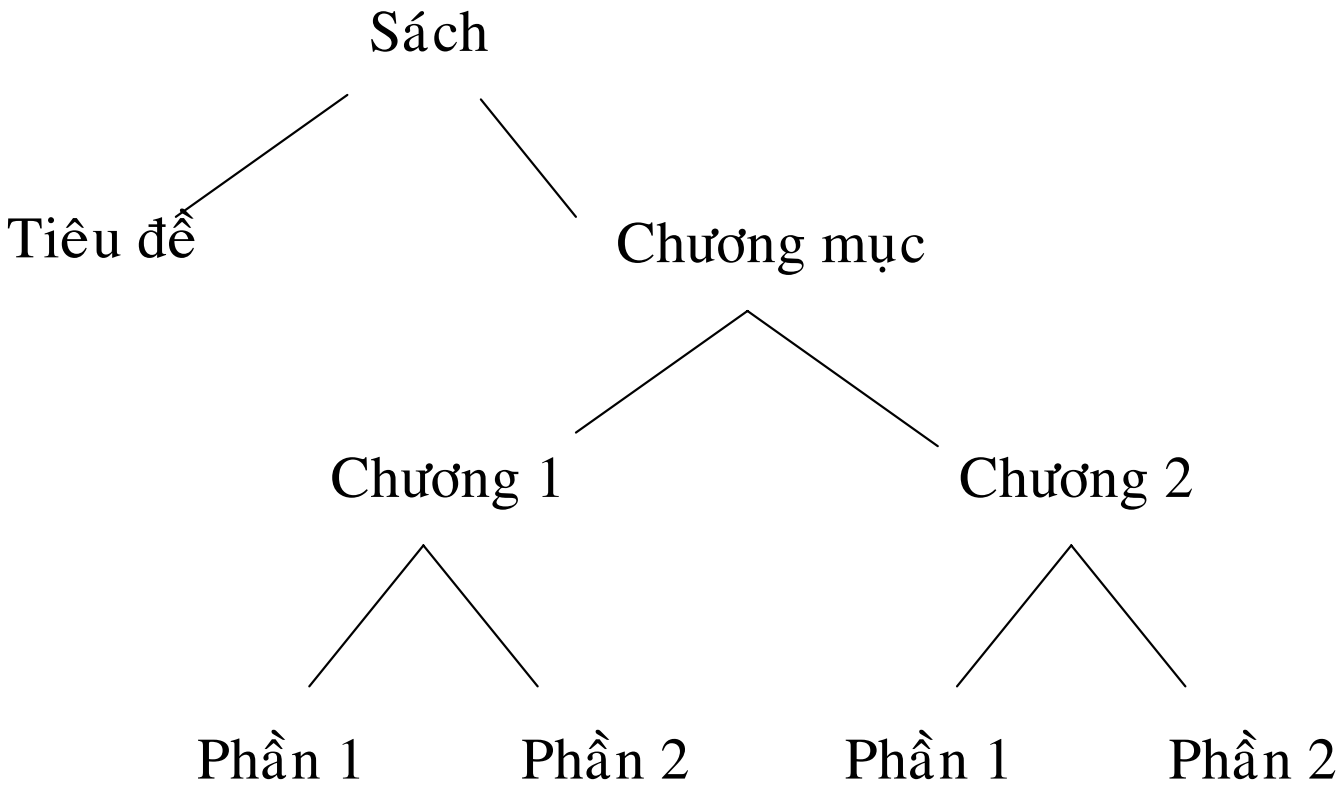
Output: tuỳ nhu cầu

"Thăm" node gốc T.

NLR(Con trái của T);

NLR(Con phải của T);

End.


Sách
Tiêu đề
Chương mục
Chương 1
Chương 2
Phần 1
Phần 2
Phần 1
Phần 2

2. Duyệt theo thứ tự sau (LRN)

Thuật toán LRN(T)

Input: Cây nhị phân T

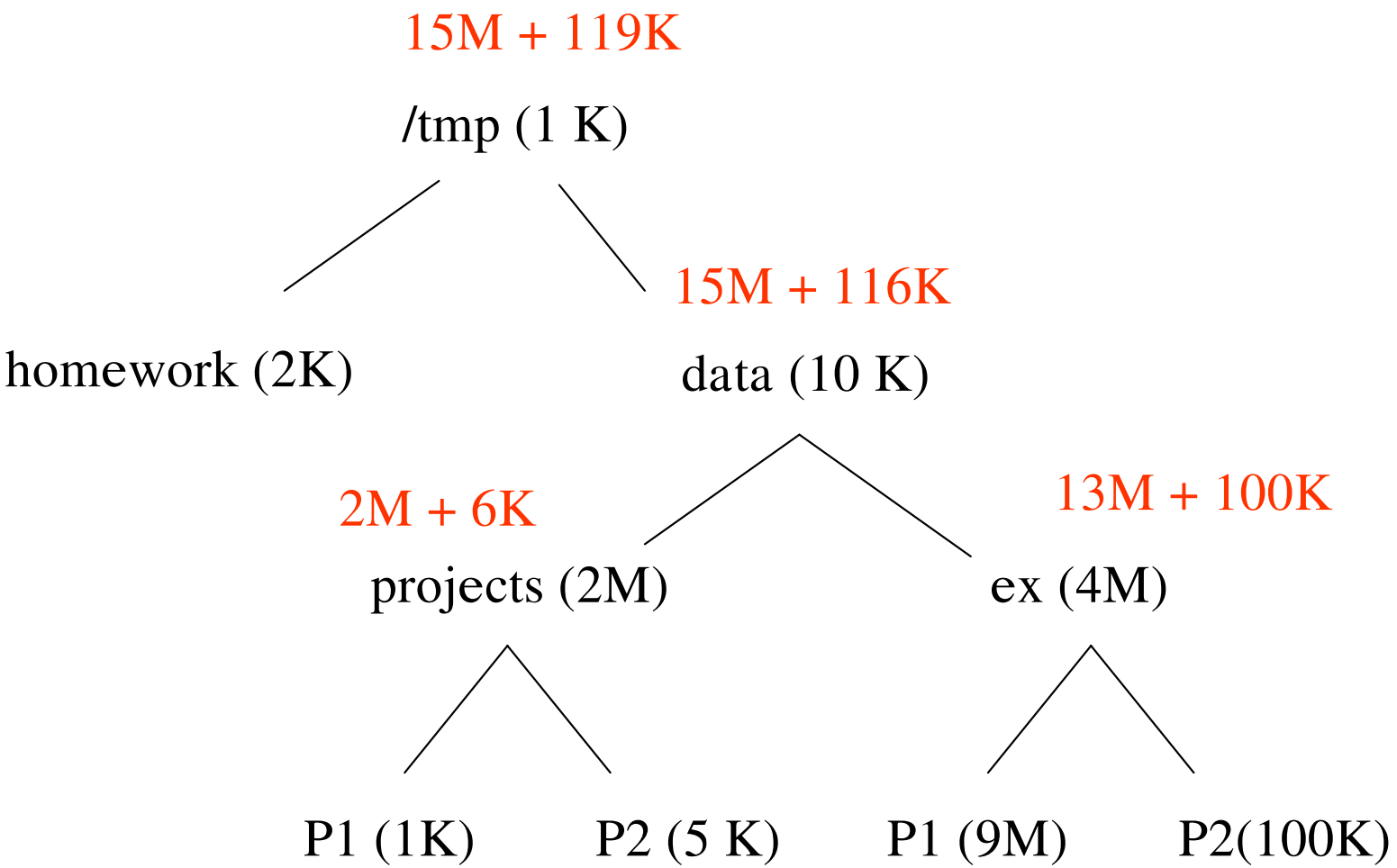
Output: tuỳ nhu cầu

NLR(Con trái của T);

NLR(Con phải của T);

"Thăm" node gốc T.

End.


15M + 119K
/tmp (1 K)
homework (2K)
15M + 116K
data (10 K)
2M + 6K
projects (2M)
13M + 100K
ex (4M)
P1 (1K)
P2 (5 K)
P1 (9M)
P2(100K)

3. Duyệt theo thứ tự giữa (LNR)

Thuật toán LNR(T)

Input: Cây nhị phân T

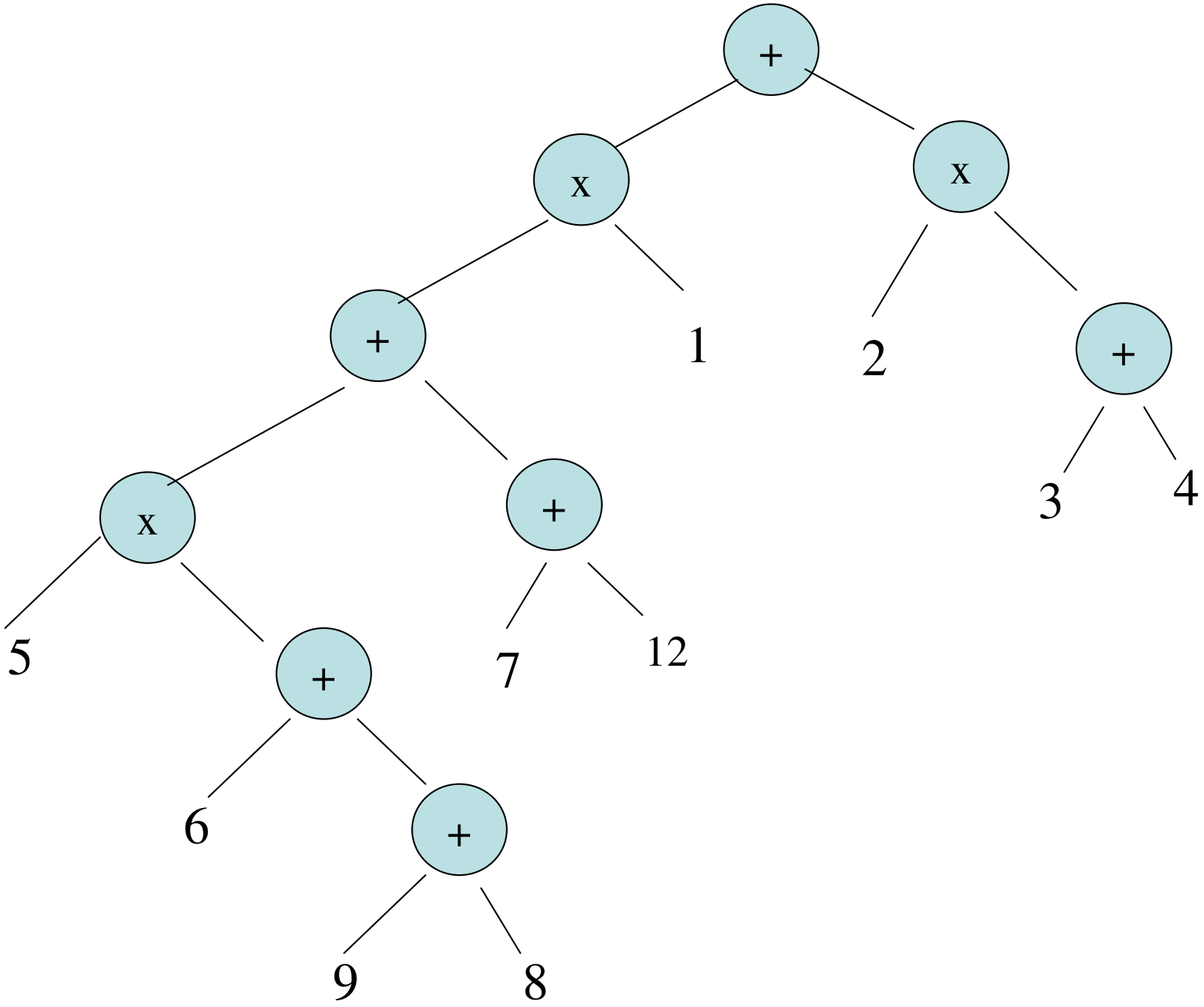
Output: tuỳ nhu cầu

LNR(Con trái của T);

"Thăm" node gốc T.

LNR(Con phải của T);

End.



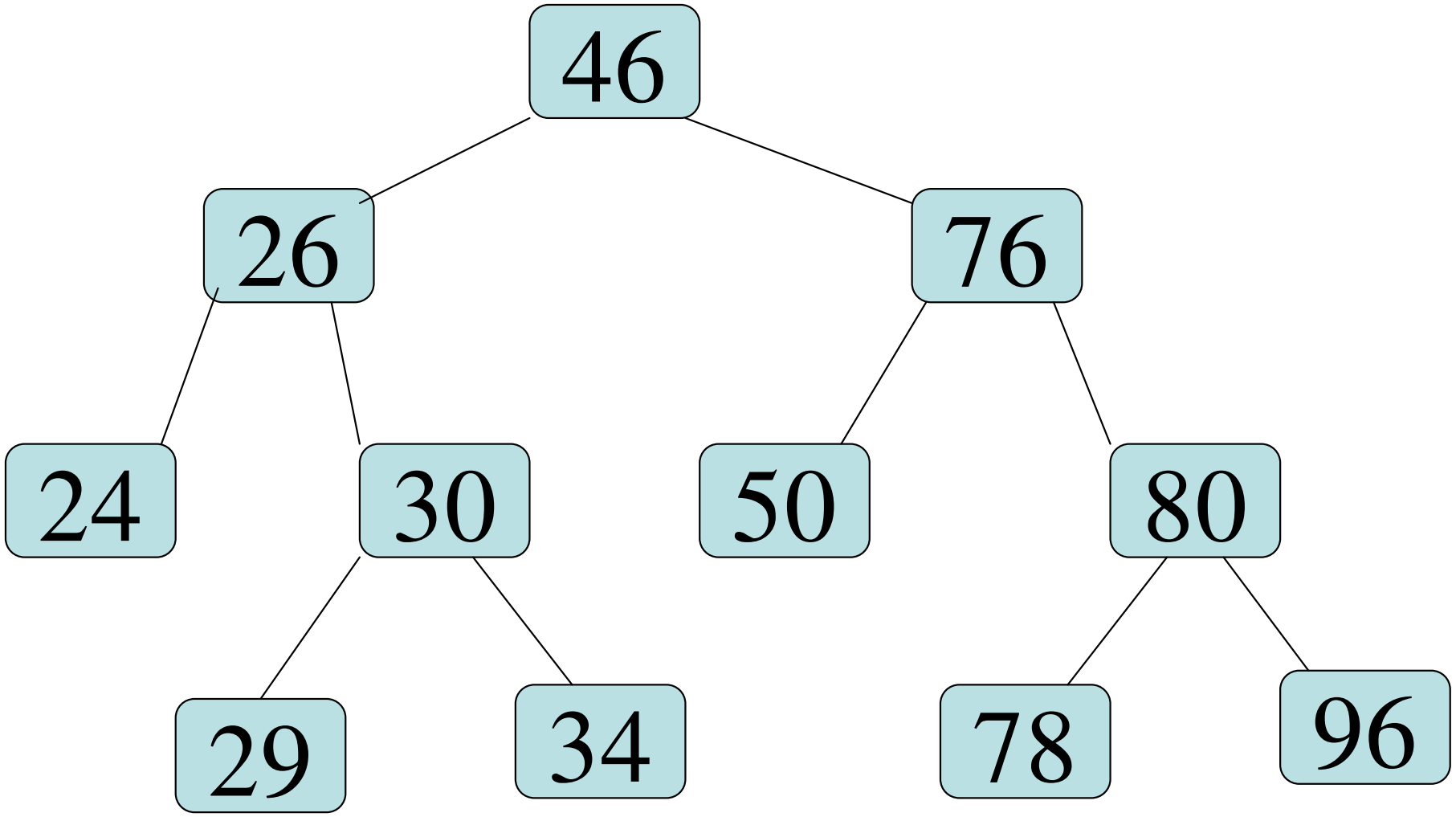
((((5x(6+(9+8)))+(7+12))x1)+(2x(3+4)))

- Đôi khi chúng ta quan tâm đến cả quan hệ giữa phần tử con và phần tử cha phần tử, nên chúng ta sẽ thêm 01 con trỏ Parent để biết quan hệ con cha.
- Còn khó khăn khi quản cây nhị phân.

CÂY NHỊ PHÂN TÌM KIẾM (BINARY SEARCH TREE)

- Định nghĩa:

  Là cây nhị phân mà ***khoá của node bất kỳ trên cây sẽ lớn hơn khoá các node trên cây con trái và nhỏ hơn khoá các node trên cây con phải của nó*.**


46
26
76
24
30
50
80
29
34
78
96

- Các thao tác:
  1. Duyệt cây
  2. Tìm kiếm 01 phần tử
  3. Thêm 01 phần tử
  4. Huỷ 01 phần tử
  5. Huỷ cây

- Duyệt thứ tự trước:

```
void NLR(BSTree T){
    if(T){
        “Thăm gốc T”; // printf(“%d”,T->Data);
        NLR(T->Left);
        NLR(T->Right);
    }
}
```

- Duyệt thứ tự sau:

```
void LRN(BSTree T){
    if(T){
        NLR(T->Left);
        NLR(T->Right);
        "Thăm gốc T"; // printf("%d",T->Data);
    }
}
```

- Duyệt thứ tự giữa:

```
void LRN(BSTree T){
    if(T){
        NLR(T->Left);
        “Thăm gốc T”; // printf(“%d”,T->Data);
        NLR(T->Right);
    }
}
```

- Tìm kiếm 01 phần tử:

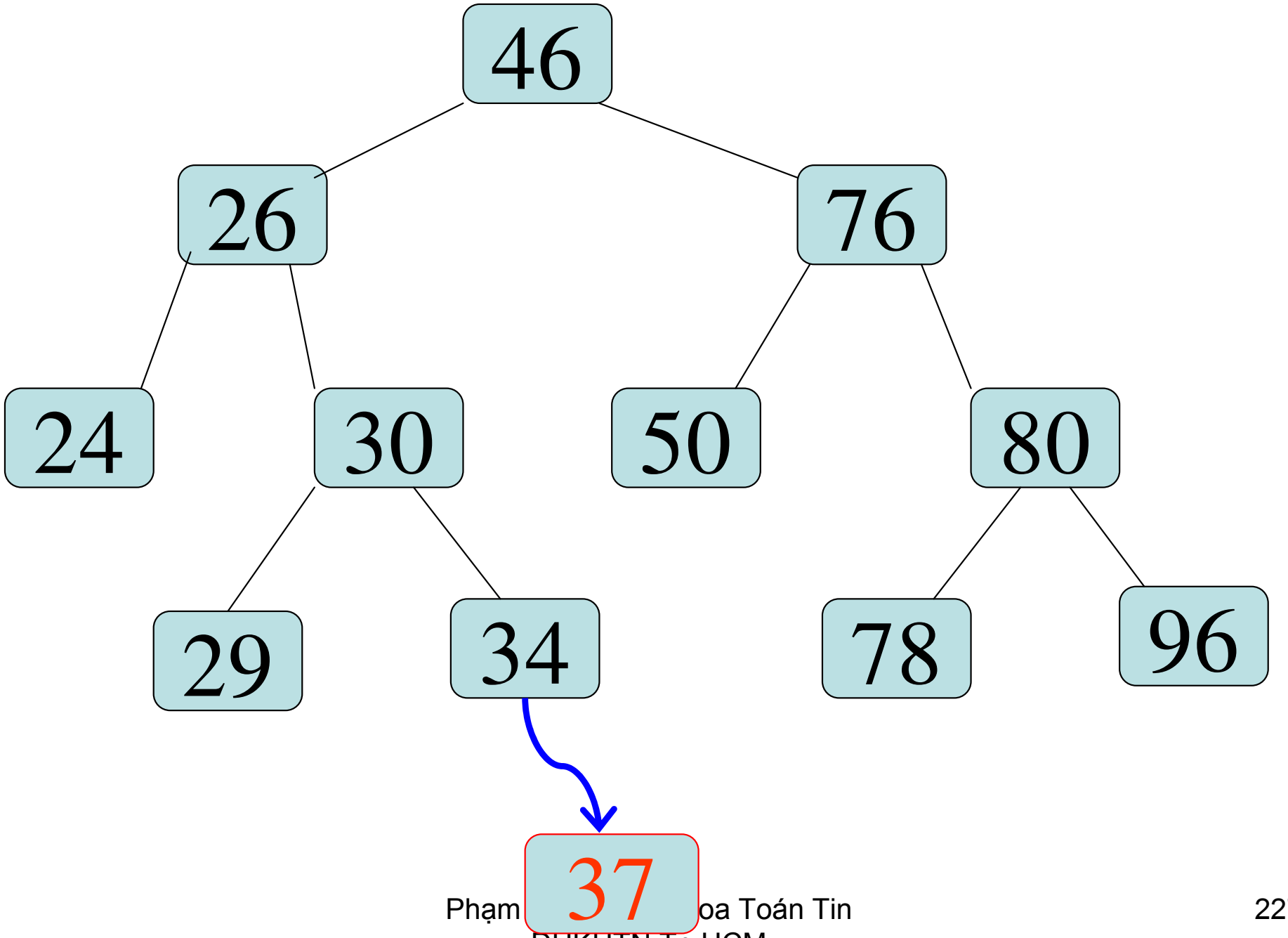
```
tagNode * searchNode(BSTree T, DataType x){
        if(T){
                if(T->Data bằng x) return T;
                if(T->Data lớn hơn x)
                        return searchNode(T->Left,x);
                else
                        return searchNode(T->Right,x)

        }
        return NULL;
}
```

- Thêm 01 phần tử vào CNPTK:
  - ✓ Đảm bảo điều kiện ràng buộc của CNPTK.
  - ✓ Có thể thêm vào nhiều chỗ khác nhau trên cây, nhưng nếu thêm ở node lá thì sẽ dễ nhất; vì quá trình sẽ tương tự tìm kiếm.
  - ✓ Hàm insertNode sẽ trả ra giá trị: -1, 0 , 1 thể hiện không đủ bộ nhớ, giá trị thêm đã có trong CNPTK rồi, và thêm thành công.

```
int insertNode(BSTree &T, DataType x){
   if(T){
        if(T->Data bằng x)return 0;
        if(T->Data lớn hơn x)
                return insertNode(T->Left,x);
        else
                return insertNode(T->Right,x);
   }
   T = new tagNode;
   if(!T) return -1;
   T->Data = x;
   T->Left = T->Right = NULL;
   return 1;
}
```

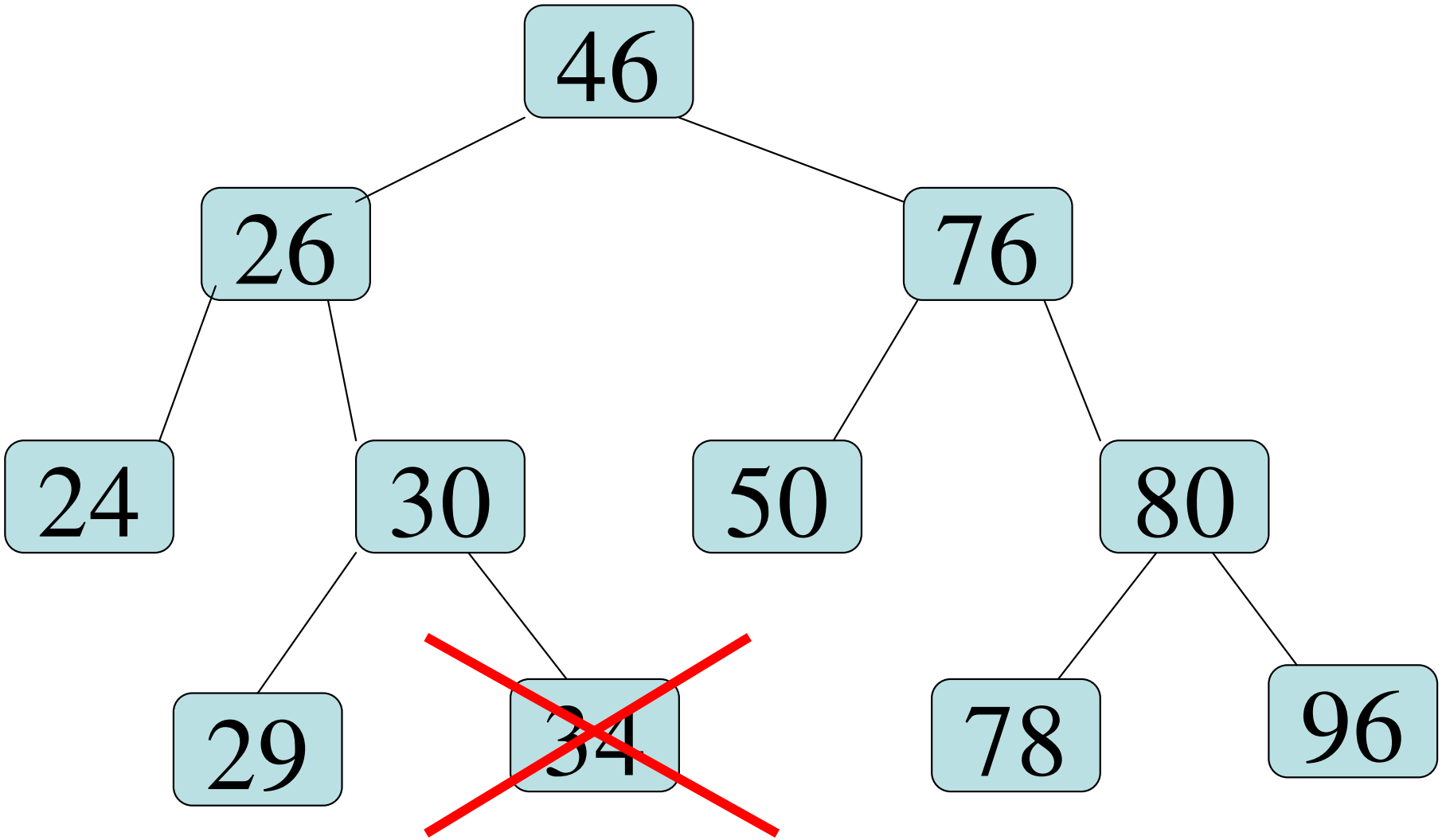
Thêm x = 37



- Hủy 01 phần tử trên CNPTK:
  - ✓ Đảm bảo điều kiện ràng buộc của CNPTK.
  - ✓ Có 03 trường hợp xảy ra:
    - ➢ x là node lá
    - ➢ x chỉ có 01 con (trái hay phải)
    - ➢ x có đủ 02 con

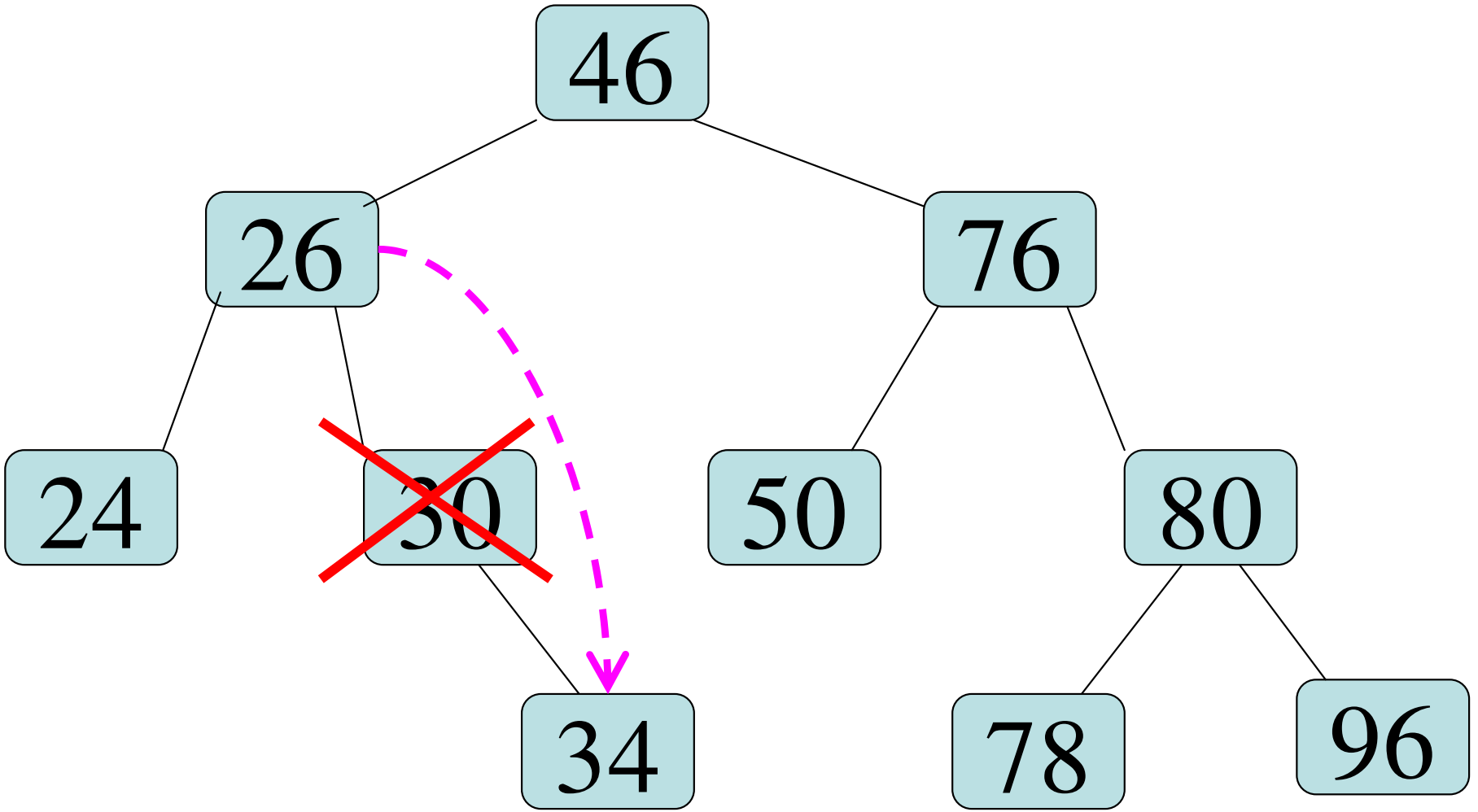
Trường hợp thứ nhất: đơn giản vì không có móc nối đến phần tử nào khác

Hủy x = 34



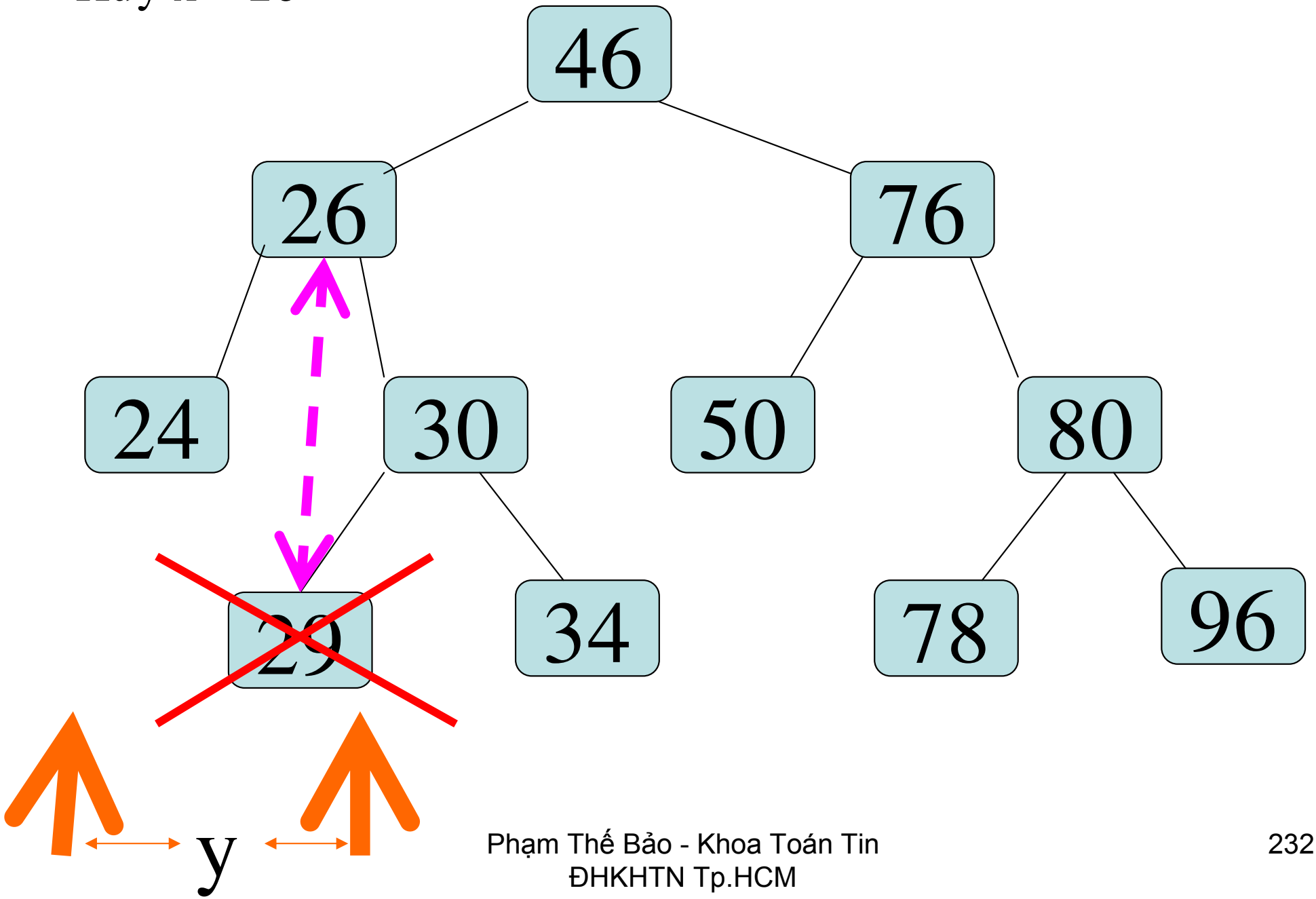
Trường hợp thứ hai: trước khi huỷ x ta móc nối cha của x và con duy nhất của nó.

Hủy x = 30



- Trường hợp cuối cùng: ta không thể huỷ trược tiếp x do x có đủ 02 con (vì tái tổ chức các mối nối sẽ phức tạp), nên ta sẽ huỷ gián tiếp. Thay vì huỷ x, ta sẽ tìm 01 phần tử thế mạng y, mà phần tử này có tối đa 01 con. Thông tin lưu tại y sẽ được chuyển lưu tại x. Sau đó sẽ huỷ thật sự y đi, như 02 trường hợp đầu.
- Vấn đề đặt ra bây giờ là làm sao chọn y mà khi chuyển dữ liệu từ y lên x vẫn đảm bảo tính chất của CNPTK.
- Vậy có 02 phần tử sẽ thoả mãn đó là:
  - ✓ Phần tử nhỏ nhất trên cây con phải.
  - ✓ Hay phần tử lớn nhất trên cây con trái.

Hủy x = 26



```
int deleteNode(BSTree &T, DataType x){
    if(!T)return 0;
    if(T->Data lớn hơn x)
                return deleteNode(T->Left,x);
     if(T->Data nhỏ hơn x)
                return deleteNode(T->Right,x);
    else{
          tagNode *p = T;
          if(!T->Left) T = T->Right;
          else
                if(!T->Right)T = T->Left;
                else searchStandFor(p,T->Right);
          delete p;
          return 1;
          }
    }
```

```
// phần tử nhỏ nhất trên cây con phải
void searchStandFor(BSTree &p, BSTree &q){
   if(q->Left)searchStandFor(p,q->Left);
   else{
        p->Data = q->Data;
        p = q;
        q = q->Right;
        }
   }
```

- Tạo CNPTK:
  - ✓ Lặp lại quá trình thêm 01 phần tử vào CNPTK.
- Huỷ toàn bộ cây NPTK
  - ✓ Thông qua thao tác duyệt theo thứ tự sau.
  - ✓ Vậy có thể huỷ cây theo thứ tự giữa hoặc trước được không ?

```
void removeTree(BSTree &T){
   if(T){
     removeTree(T->Left);
     removeTree(T->Right);
     delete T;
     }
}
```

- # Tính chất:

1. Tất cả thao tác tìm kiếm, thêm, huỷ trên CNPTK đều có độ phức tạp O(h).
2. Trong trường hợp tốt nhất, CNPTK có N node sẽ có độ cao $h = \log_2(N)$, chi phí tìm kiếm khi đó sẽ tương đương tìm kiếm nhị phân trên mảng có thứ tự.
3. Tuy nhiên trong trường hợp xấu nhất, cây có thể suy biến thành 01 DSLK, khi đó các thao tác trên sẽ có độ phức tạp O(N).

Bài tập

1. Đếm cây NPTK có bao nhiêu phần tử ?
2. Đếm cây NPTK có bao nhiêu node lá ?
3. Tính chiều cao cây NPTK.