

# Chương 5: CÂY (Tree)



# Nội dung

2

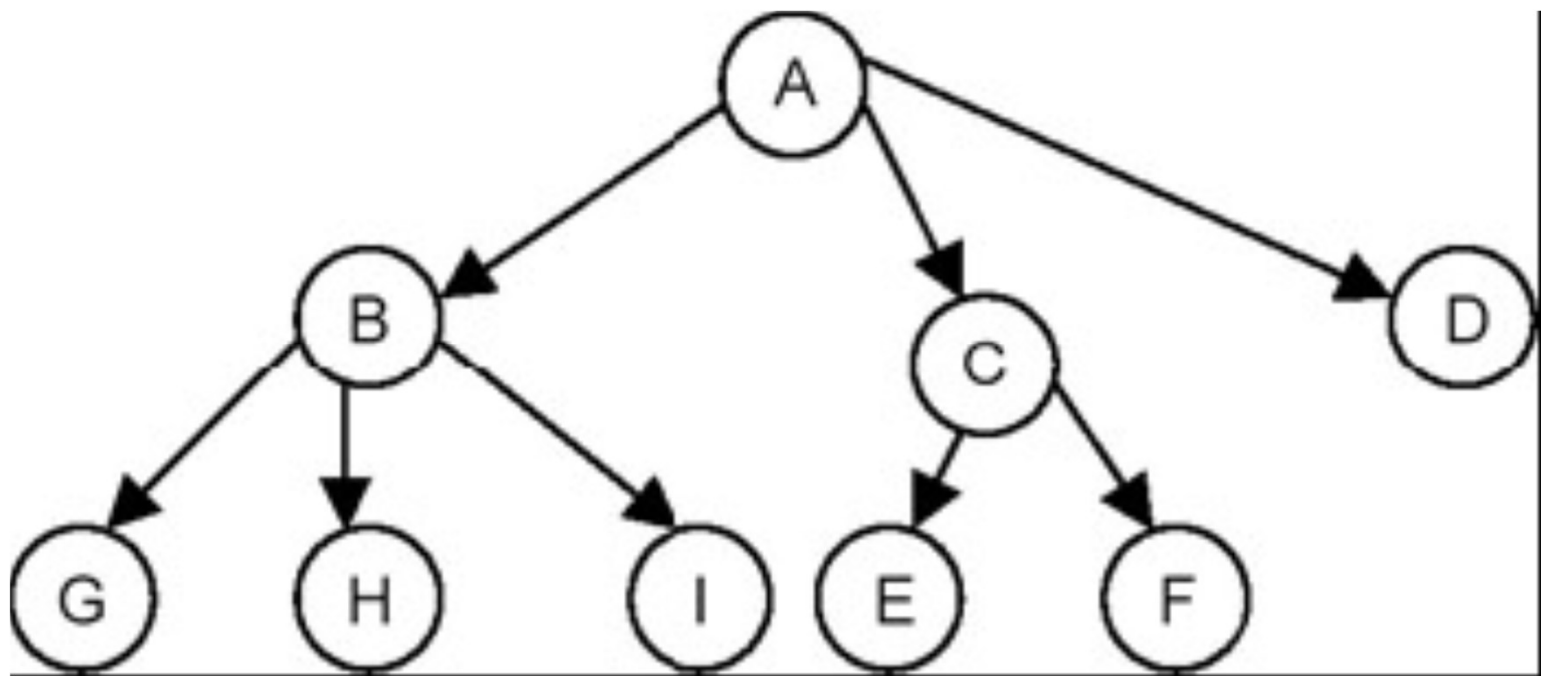
- Cấu trúc cây (**Tree**)
- Cấu trúc cây nhị phân (*Binary Tree*)
- Cấu trúc cây nhị phân tìm kiếm (*Binary Search Tree*)
- Cấu trúc cây nhị phân tìm kiếm cân bằng (**AVL Tree**)

# Tree – Định nghĩa

3

- Cây là một tập hợp  $T$  các phần tử (gọi là **nút** của cây) trong đó có 1 nút đặc biệt được gọi là **gốc**, các nút còn lại được chia thành những tập rời nhau  $T_1, T_2, \dots, T_n$  theo quan hệ phân cấp trong đó  **$T_i$  cũng là một cây**
  
- A tree is a set of one or more nodes  $T$  such that:
  - ▣ i. there is a specially designated node called a root
  - ▣ ii. The remaining nodes are partitioned into  $n$  *disjointed* set of nodes  $T_1, T_2, \dots, T_n$ , each of which is a tree

# Tree – Ví dụ



# Tree - Một số khái niệm cơ bản

5

- **Bậc của một nút (Degree of a Node of a Tree):**
  - ▣ Là **số cây con** của nút đó. Nếu bậc của một nút bằng 0 thì nút đó gọi là nút lá (**leaf node**)
- **Bậc của một cây (Degree of a Tree):**
  - ▣ Là **bậc lớn nhất** của các nút trong cây. Cây có bậc  $n$  thì gọi là cây  **$n$ -phân**
- **Nút gốc (Root node):**
  - ▣ Là nút **không** có nút cha
- **Nút lá (Leaf node):**
  - ▣ Là nút có **bậc bằng 0**

# Tree - Một số khái niệm cơ bản

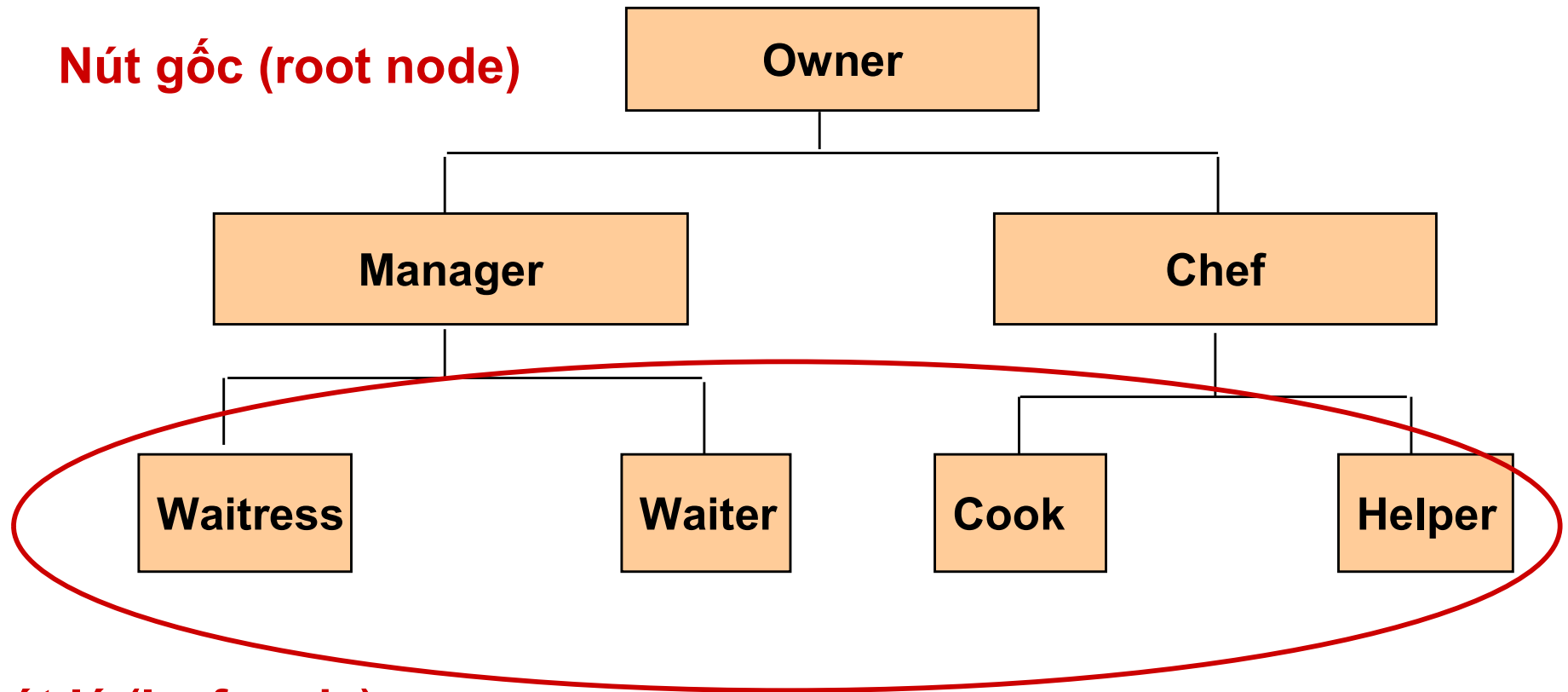
6

- Nút nhánh:
  - ▣ Là nút có **bậc khác 0** và **không phải là gốc**
- Mức của một nút (**Level of a Node**):
  - ▣ Mức (gốc (T) ) = 0
  - ▣ Gọi  $T_1, T_2, T_3, \dots, T_n$  là các cây con của  $T_0$   $\text{Mức}(T_1) = \text{Mức}(T_2) = \dots = \text{Mức}(T_n) = \text{Mức}(T_0) + 1$
- **Chiều cao** (height) hay **chiều sâu** (depth) của một cây là số mức lớn nhất của nút có trên cây đó

# Tree – Ví dụ

7

**Nút gốc (root node)**

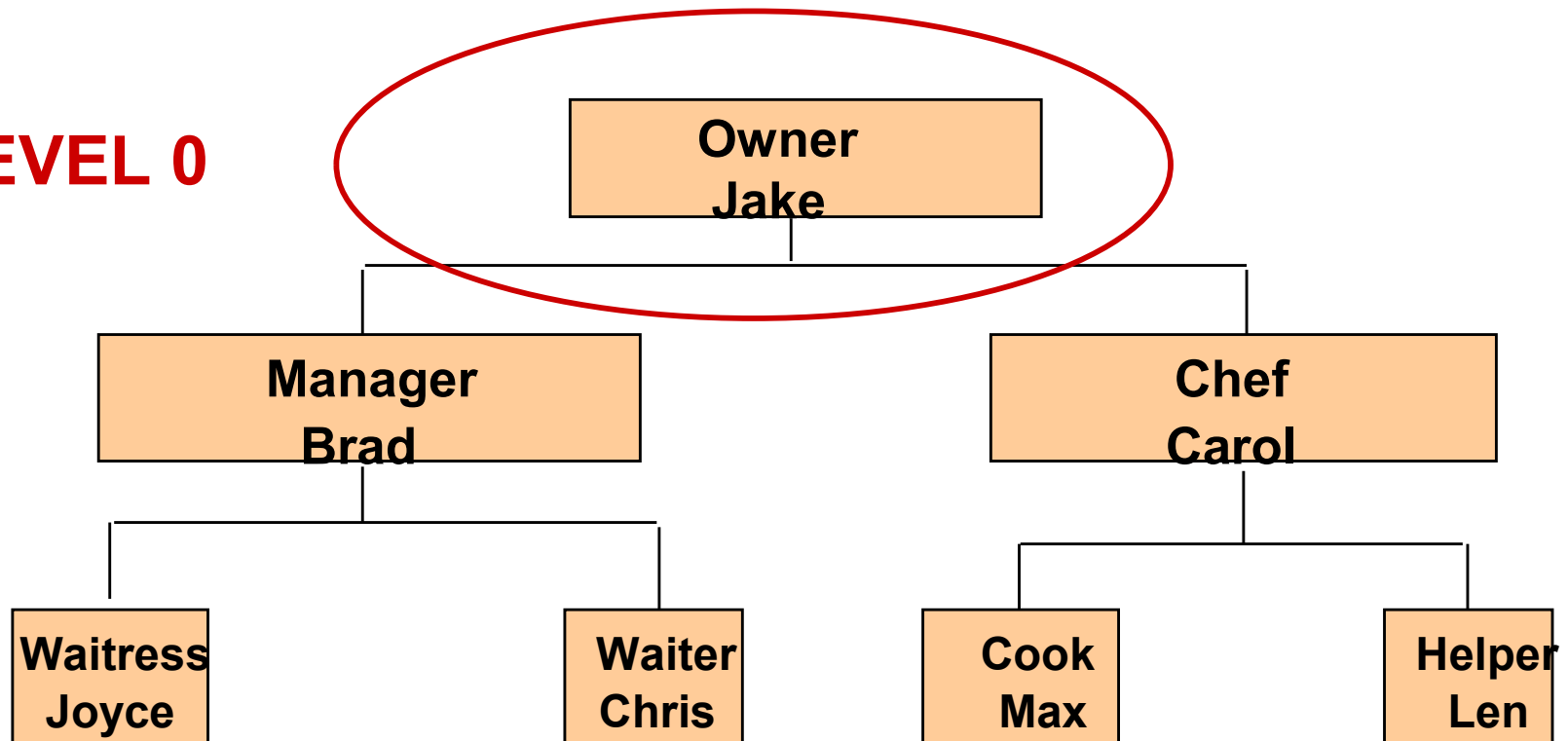


**nút lá (leaf node)**

# A Tree Has Levels

8

**LEVEL 0**



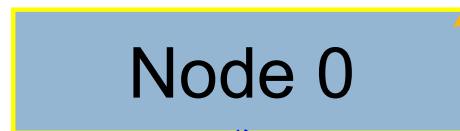


# Định nghĩa

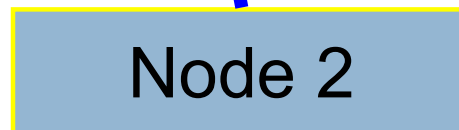
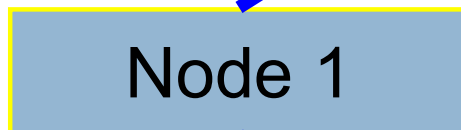
Node 0 là **tổ tiên** của tất cả các node

Nodes 1-6 là **con cháu** của node 0

Gốc

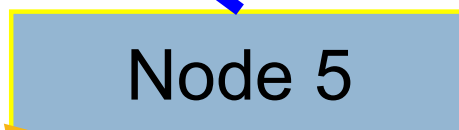
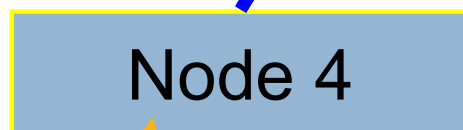


Node 1,2,3 **con** của gốc



Node 1 là **cha** của

Nodes 4,5



**lá**

Node 4, 5 là **anh em**

# Một số khái niệm cơ bản

10

- **Độ dài đường đi từ gốc đến nút x:**

$P_x =$  **số nhánh** cần đi qua kể từ gốc đến x

- **Độ dài đường đi tổng của cây:**  $P_T = \sum_{X \in T} P_X$

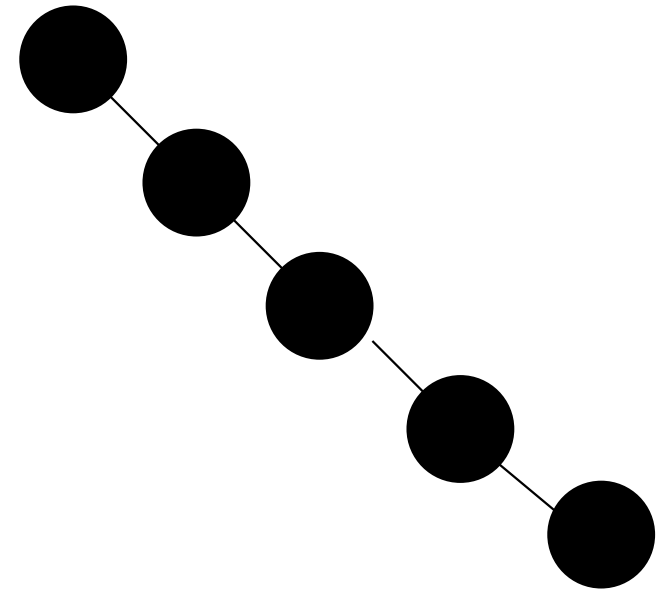
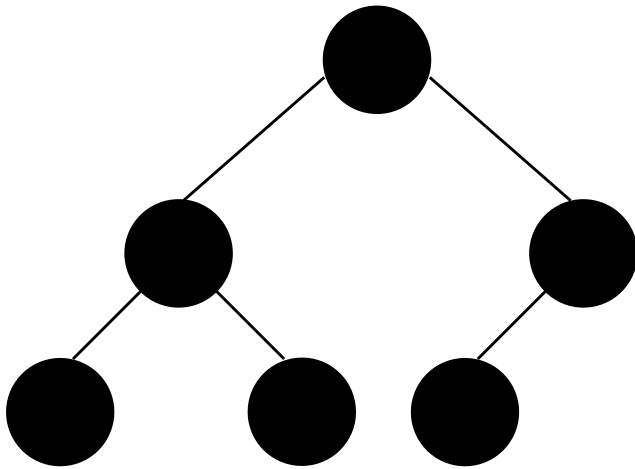
trong đó  $P_x$  là độ dài đường đi từ gốc đến X

- **Độ dài đường đi trung bình:**  $P_I = P_T/n$  (n là số nút trên cây T)
- **Rừng cây:** là tập hợp nhiều cây trong đó thứ tự các cây là quan trọng

# Binary Tree – Định nghĩa

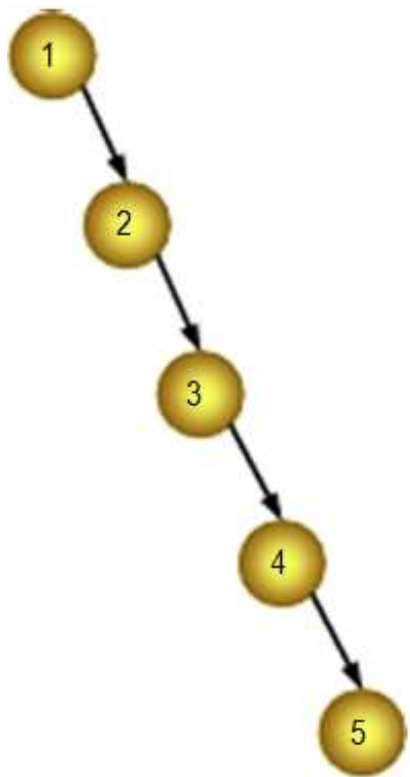
11

- Cây nhị phân là cây mà mỗi nút có tối đa **2 cây con**

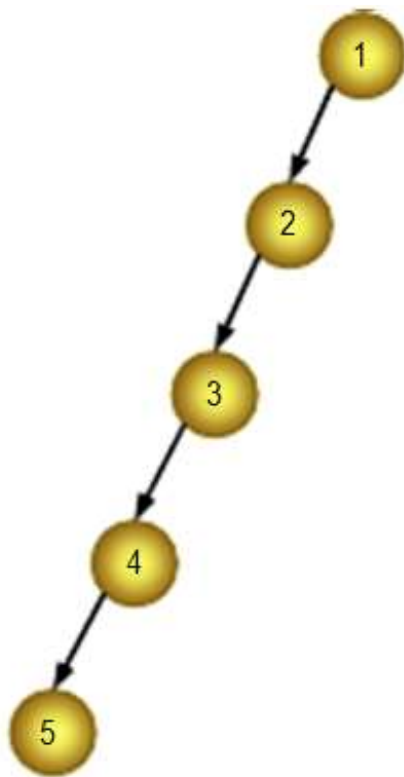


# Binary Tree – Ví dụ

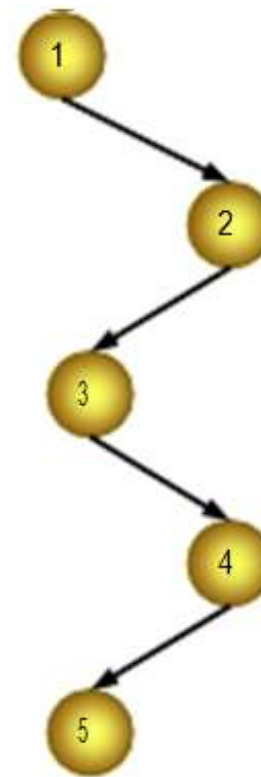
- Cây lệch trái, cây lệch phải, cây zíc-zắc



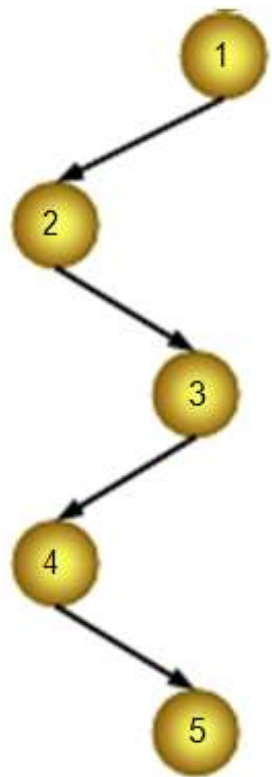
a)



b)



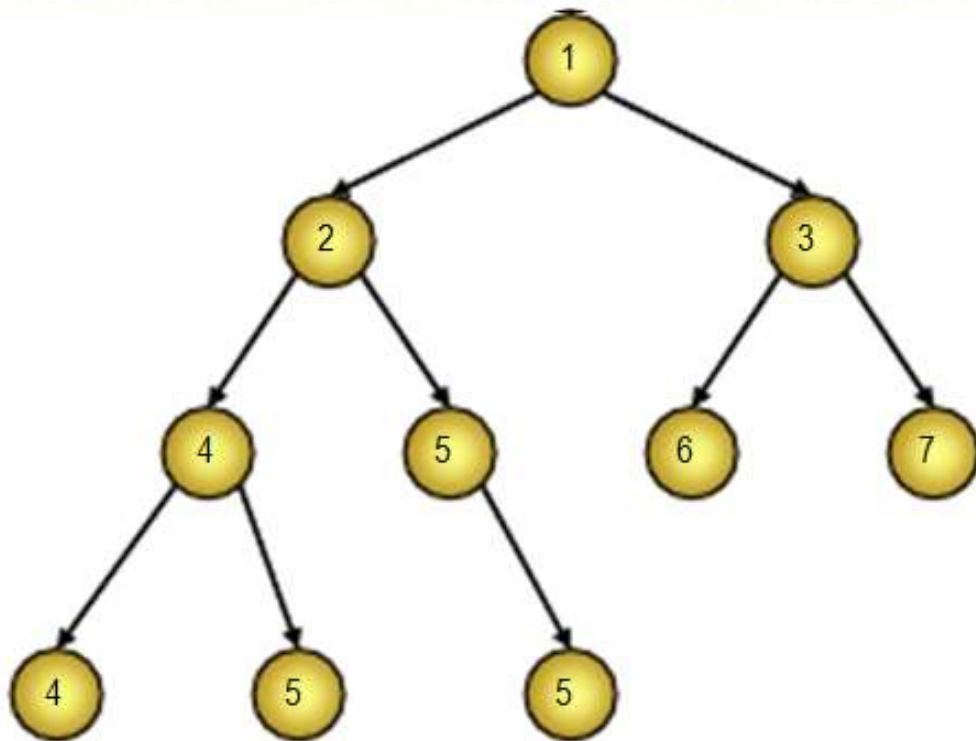
c)



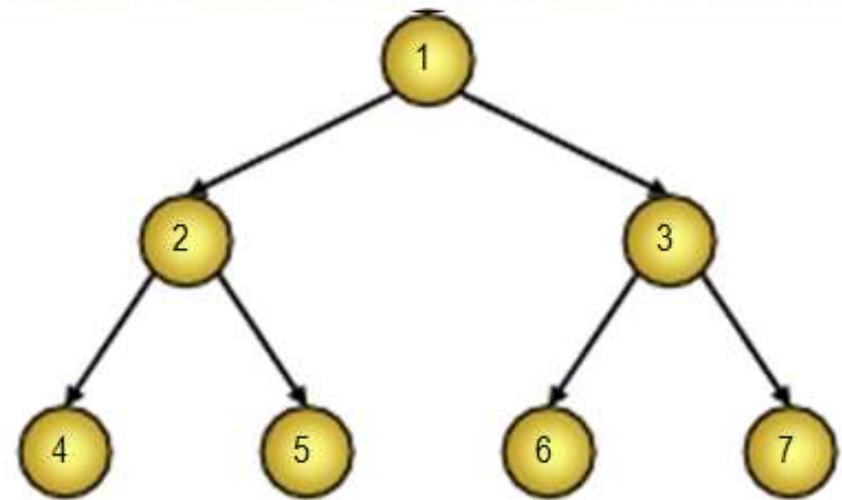
d)

## 2. Cây nhị phân (Binary tree)

- **Cây nhị phân hoàn chỉnh** (complete binary tree) có chiều cao là  $h$  thì mọi nút có mức  $< h-1$  đều có đúng 02 nút con.
- **Cây nhị phân đầy đủ** (full Binary tree) có chiều cao  $h$  thì mọi nút có mức  $\leq h-1$  đều có đúng 02 nút con



e)

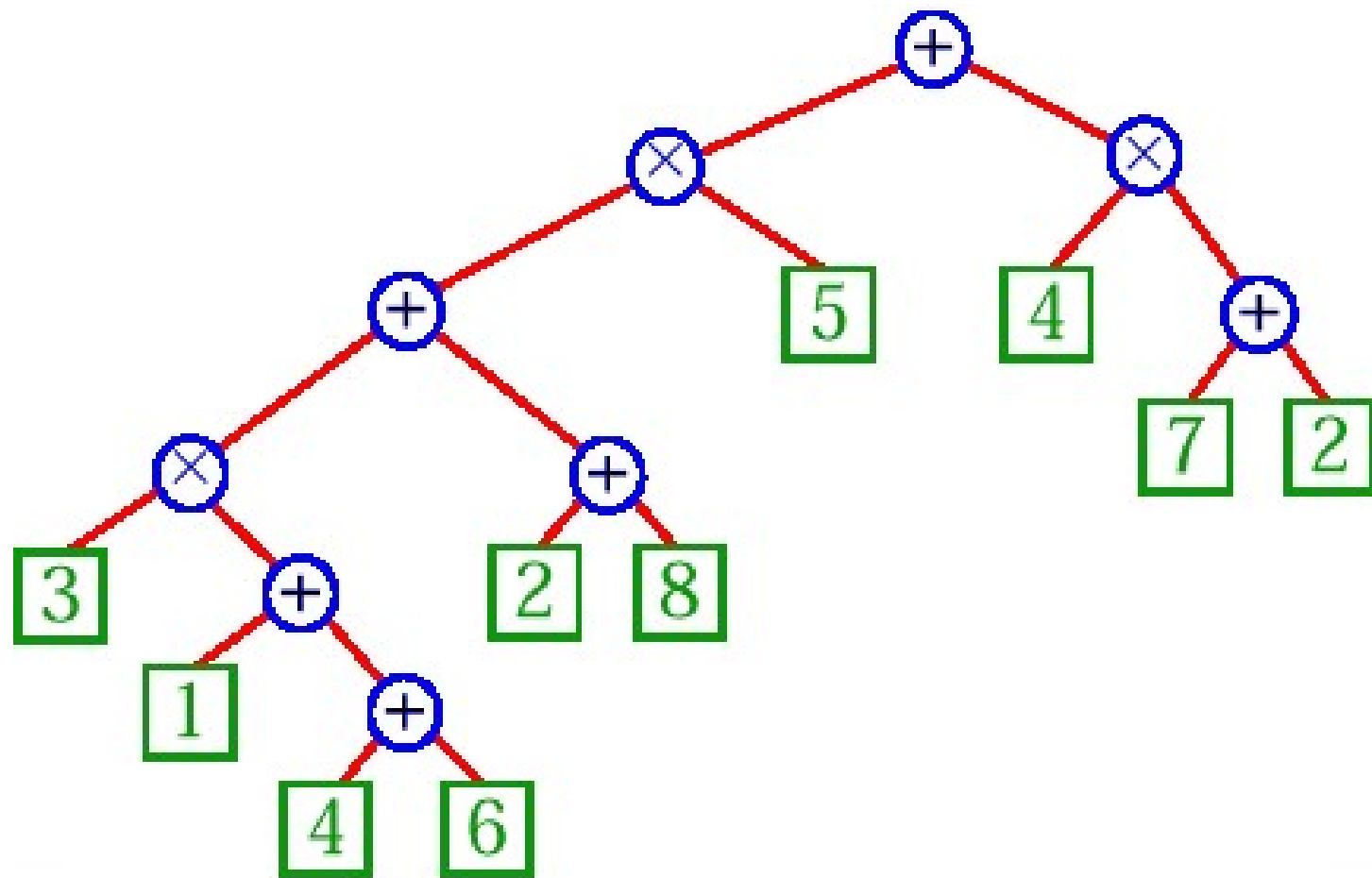


f)

# Binary Tree – Ví dụ

14

- Cây nhị phân dùng để biểu diễn một biểu thức toán học:



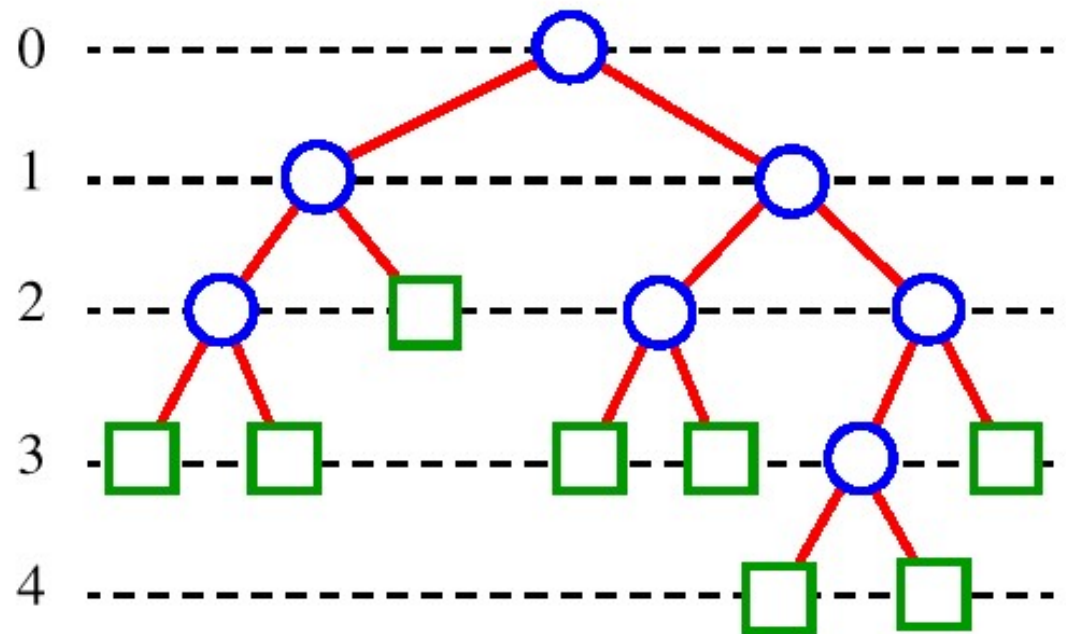
$(((((3 \times (1 + (4 + 6))) + (2 + 8)) \times 5) + (4 \times (7 + 2))))$

# Binary Tree – Một số tính chất

15

- Số nút nằm ở mức  $i \leq 2^i$
- Số nút lá  $\leq 2^{h-1}$ , với  $h$  là chiều cao của cây
- Chiều cao của cây  $h \geq \log_2 N$ , với  $N$  là số nút trong cây
- Số nút trong cây  $\leq 2^h - 1$ , với  $h$  là chiều cao của cây

Xem thêm gtrình trang 142



## 2. Cây nhị phân (Binary tree)

### b) Một số các tính chất:

- ▣ Nếu số lượng nút là như nhau thì cây nhị phân suy biến có chiều cao lớn nhất, cây nhị phân đầy đủ có chiều cao nhỏ nhất.
- ▣ Số lượng tối đa các nút trên mức  $i$  là  $2^{i-1}$  và tối thiểu là 1 ( $i \geq 1$ )
- ▣ Số lượng tối đa các nút trên cây nhị phân có chiều cao  $h$  là  $2^h - 1$  và tối thiểu là  $h$  ( $h \geq 1$ )
- ▣ Cây nhị phân hoàn chỉnh có  $n$  nút thì chiều cao của nó là

$$h = \lceil \lg_2(n) \rceil + 1$$



## 2. Cây nhị phân tương đương (Binary tree)- 119

- ▣ Biểu diễn cây tổng quát bằng cây nhị phân tương đương:
- ▣ Mỗi node có cấu trúc
  - Node con trái là node con cực trái (CHILD) của node đó
  - Node con phải là node em kế cận phải (SIBLING) của node đó

**Địa chỉ con cực trái**

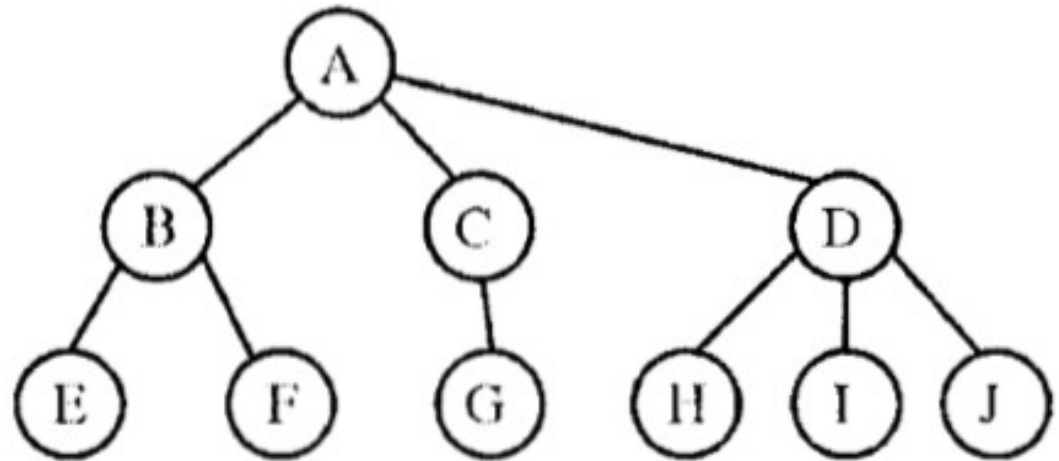
**Data**

**Địa chỉ em kế cận phải**

## 2. Cây nhị phân tương đương (Binary tree)- 119

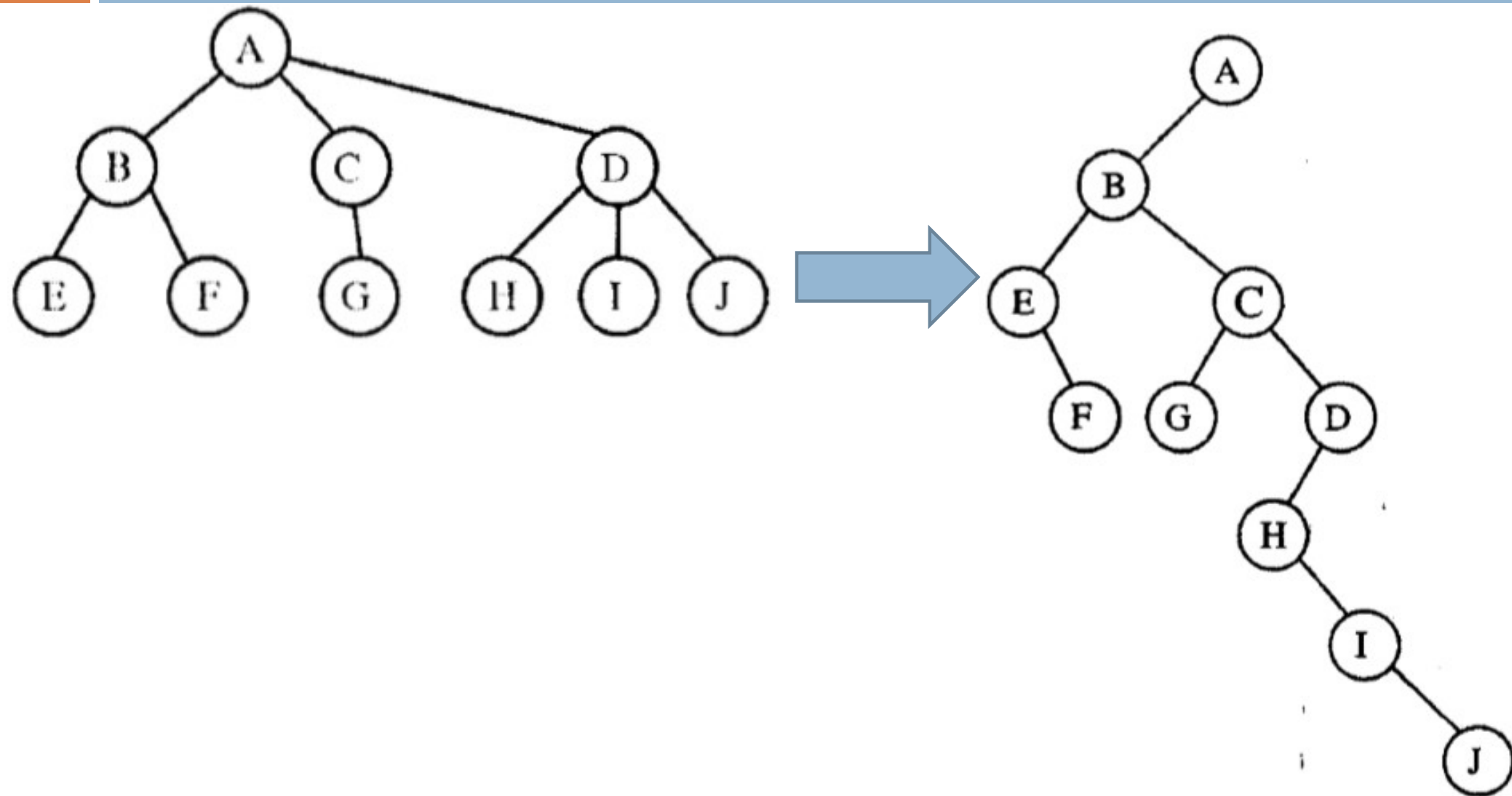
**CHILD**: node cực trái

**SIBLING** : node em kế cận phải



Node	A	B	C	D	E	F	G	H	I	J
CHILD	B	E	G	H						
SIBLING		C	D		F			I	J	

## 2. Cây nhị phân tương đương (Binary tree)- 119



# Binary Tree - Biểu diễn bằng mảng

20

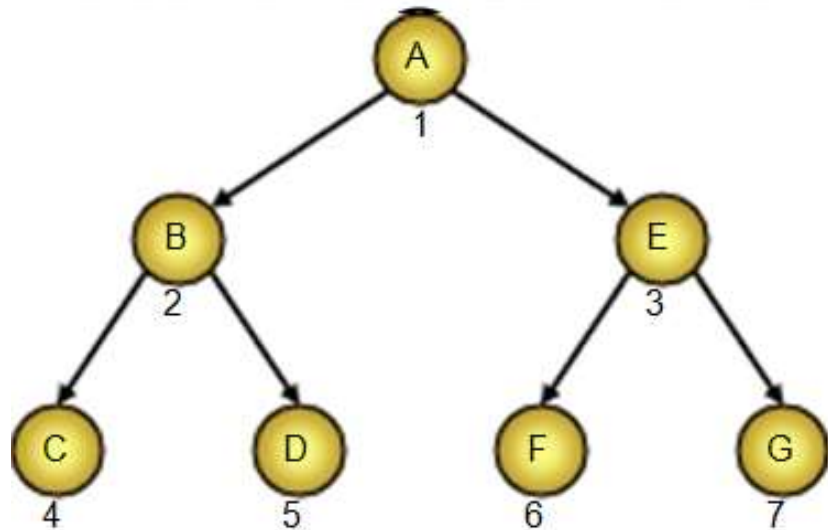
- Một cây nhị phân đầy đủ, ta có thể dễ dàng đánh số các nút: theo thứ tự lần lượt từ mức 0 trở đi. từ trái sang phải đối với các nút ở mỗi mức.

- Nút thứ  $i$ :

- Con trái:  $2i$
- Con phải:  $2i + 1$

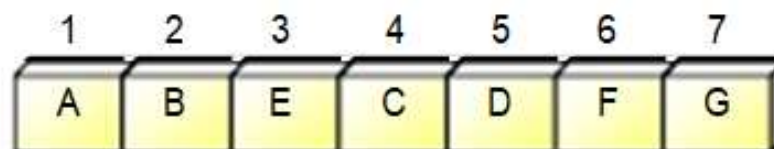
- Nút thứ  $j$ :

- Nút cha :  $j \text{ div } 2$



- Sử dụng một mảng  $T$  để lưu trữ các thông tin của một nút:

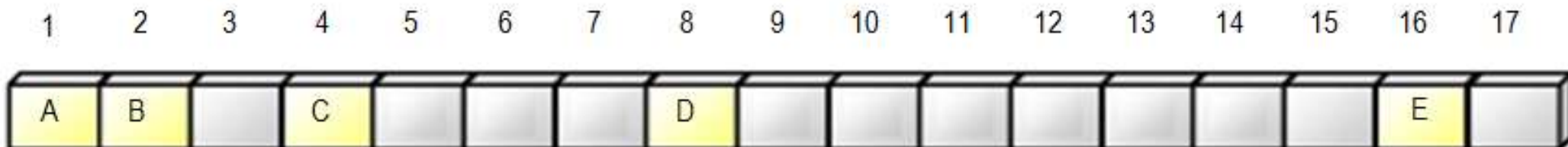
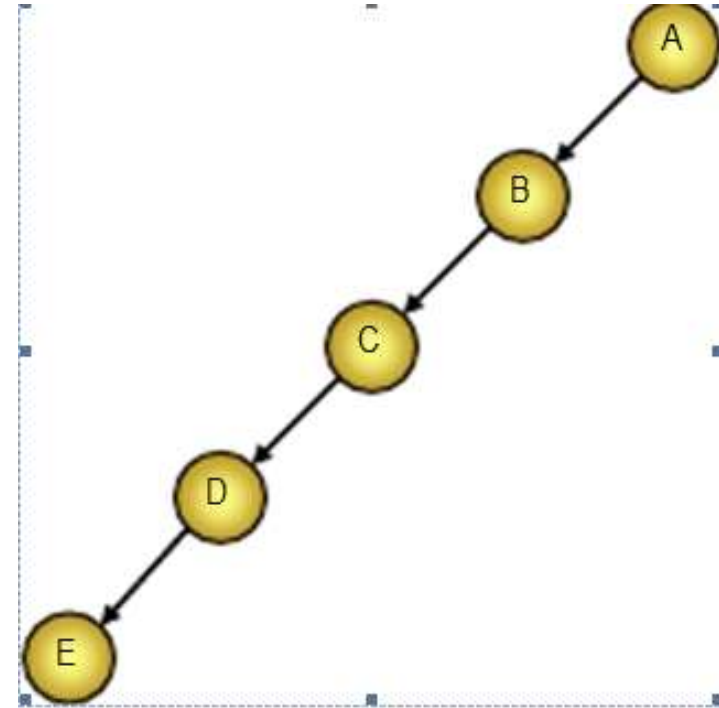
- Nút thứ  $i$  của cây được lưu trữ bằng phần tử  $T[i]$



# Binary Tree - Biểu diễn bằng mảng

21

- Cây nhị phân không đầy đủ, ta có thể thêm vào một số nút giả để được cây nhị phân đầy đủ
- Gán những giá trị đặc biệt cho những phần tử trong mảng T tương ứng nút giả
- Hoặc, dùng thêm một mảng phụ để đánh dấu những nút nào là nút giả được thêm vào
- => sự lãng phí bộ nhớ

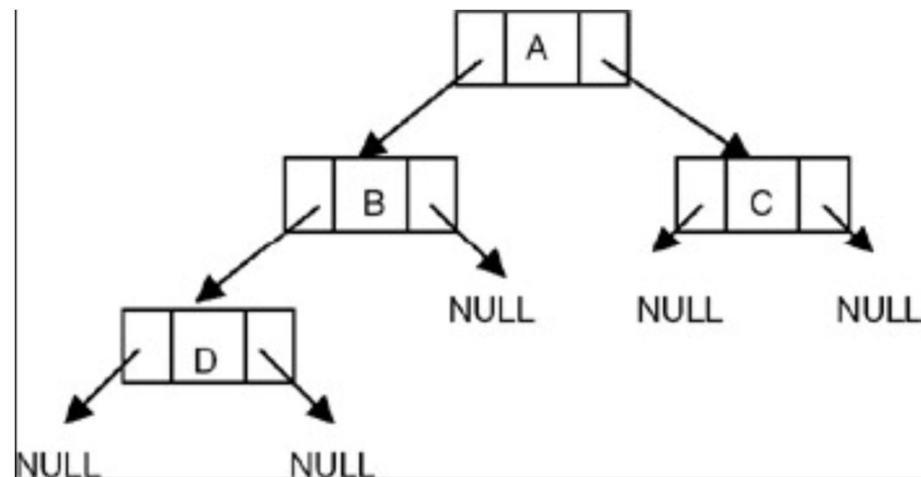


# Binary Tree - Biểu diễn bằng cấu trúc liên kết

22

- Sử dụng một biến động để lưu trữ các thông tin của một nút:
  - ▣ Thông tin lưu trữ tại nút
  - ▣ Địa chỉ nút gốc của cây con trái trong bộ nhớ
  - ▣ Địa chỉ nút gốc của cây con phải trong bộ nhớ
- Khai báo cấu trúc cây nhị phân:

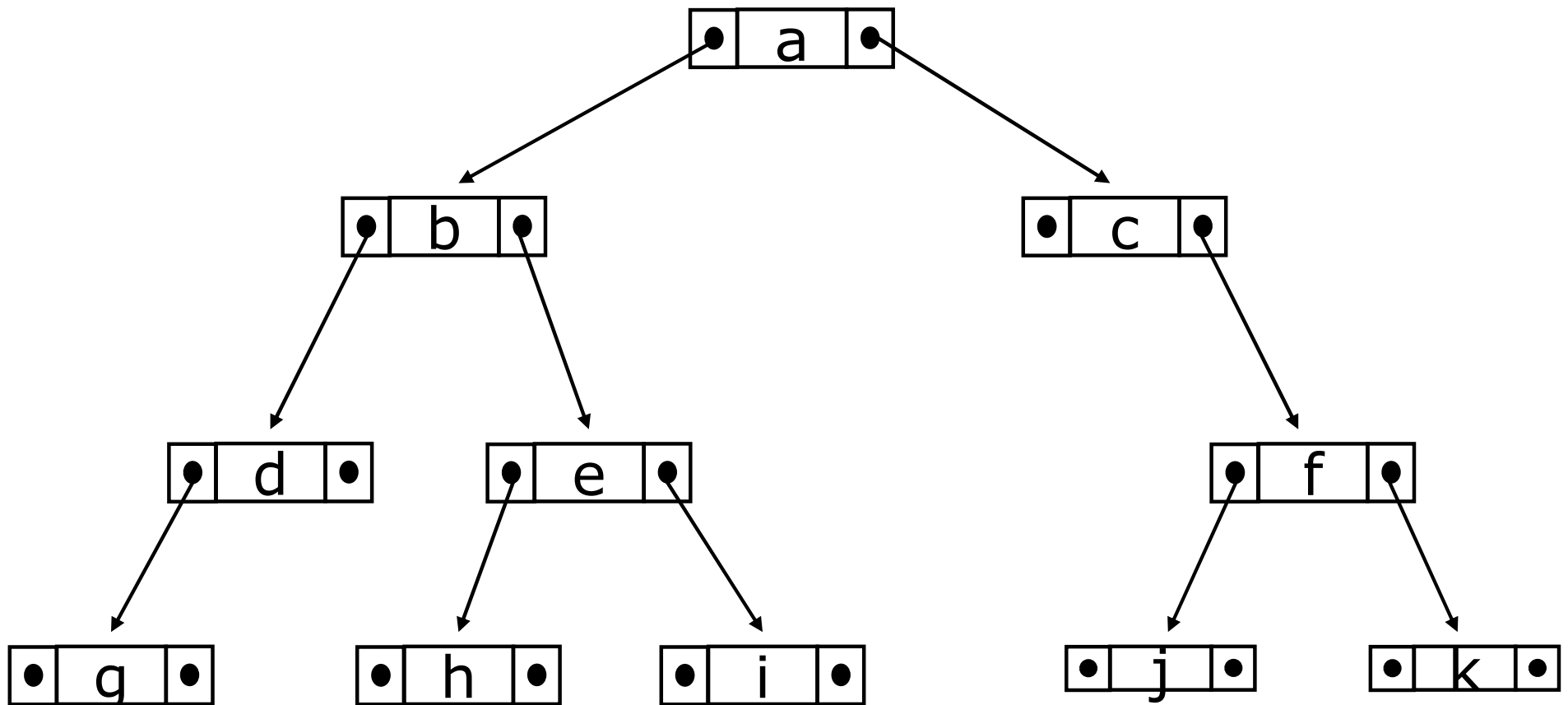
```
struct TNode
{
    DataType data;
    TNode *pLeft, *pRight;
};
typedef TNode *Tree;
```



- Để quản lý cây nhị phân chỉ cần quản lý địa chỉ nút gốc:  
`Tree root;`

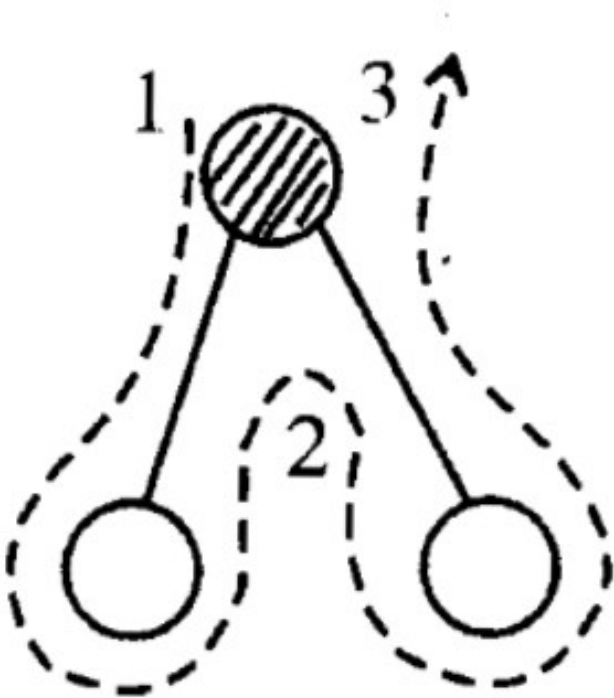
# Binary Tree - Biểu diễn

23



# Cây nhị phân – Binary tree

24

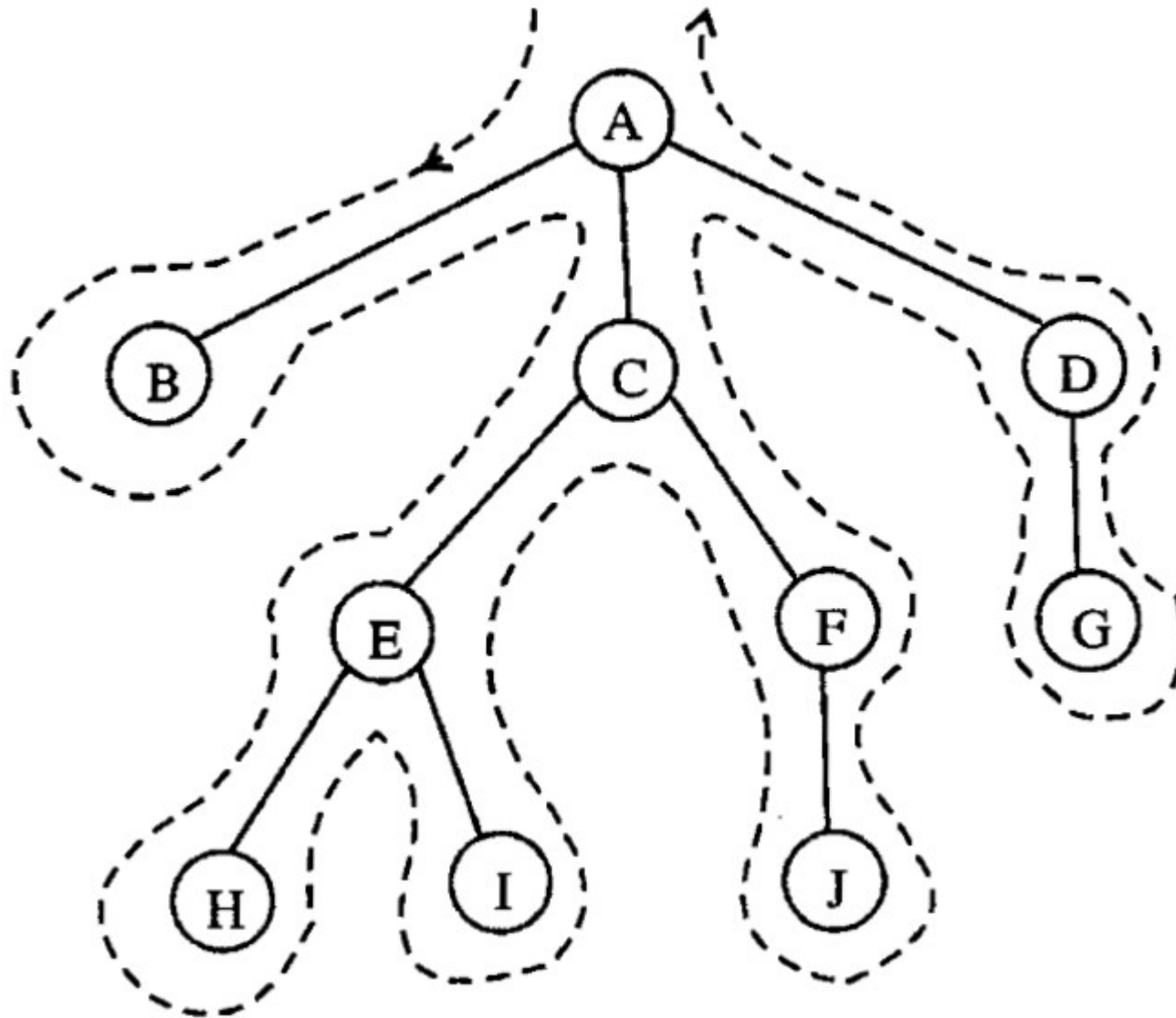


- Một nút với 2 con
- Khi đi qua nút ấy và các con nó theo đường mũi tên, ta sẽ gặp nút ấy tới ba lần



# Duyệt cây

25



# Binary Tree - Duyệt cây nhị phân

26

- Có 3 kiểu duyệt chính có thể áp dụng trên cây nhị phân:
  - ▣ Duyệt theo thứ tự trước - **preorder** (Node-Left-Right: **NLR**)
  - ▣ Duyệt theo thứ tự giữa - **inorder** (Left-Node-Right: **LNR**)
  - ▣ Duyệt theo thứ tự sau - **postorder** (Left-Right-Node: **LRN**)
- Tên của 3 kiểu duyệt này được đặt dựa trên trình tự của việc thăm nút gốc so với việc thăm 2 cây con

# Binary Tree - Duyệt cây nhị phân

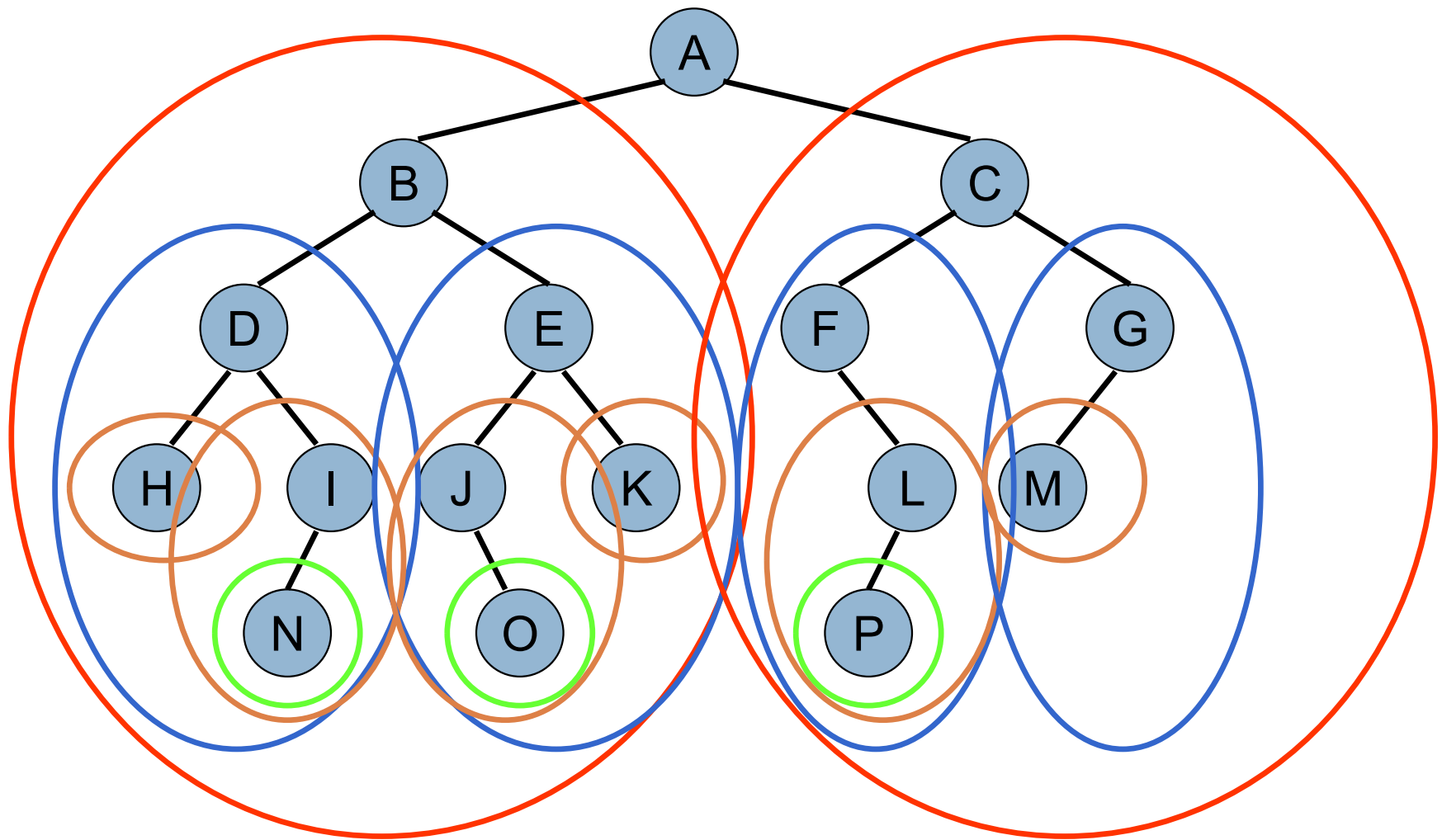
27

- ❑ Duyệt theo thứ tự trước NLR (Node-Left-Right)
  - ❑ Kiểu duyệt này trước tiên thăm nút gốc sau đó thăm các nút của cây con trái rồi đến cây con phải
  - ❑ Thủ tục duyệt có thể trình bày đơn giản như sau:

```
void NLR (Tree t)
{
    if (t != NULL)
    {
        // Xử lý t tương ứng theo nhu cầu
        NLR (t->pLeft) ;
        NLR (t->pRight) ;
    }
}
```

# Binary Tree - Duyệt cây nhị phân NLR

28



Kết quả: A B D H I N E J O K C F L P G M

# Binary Tree - Duyệt cây nhị phân

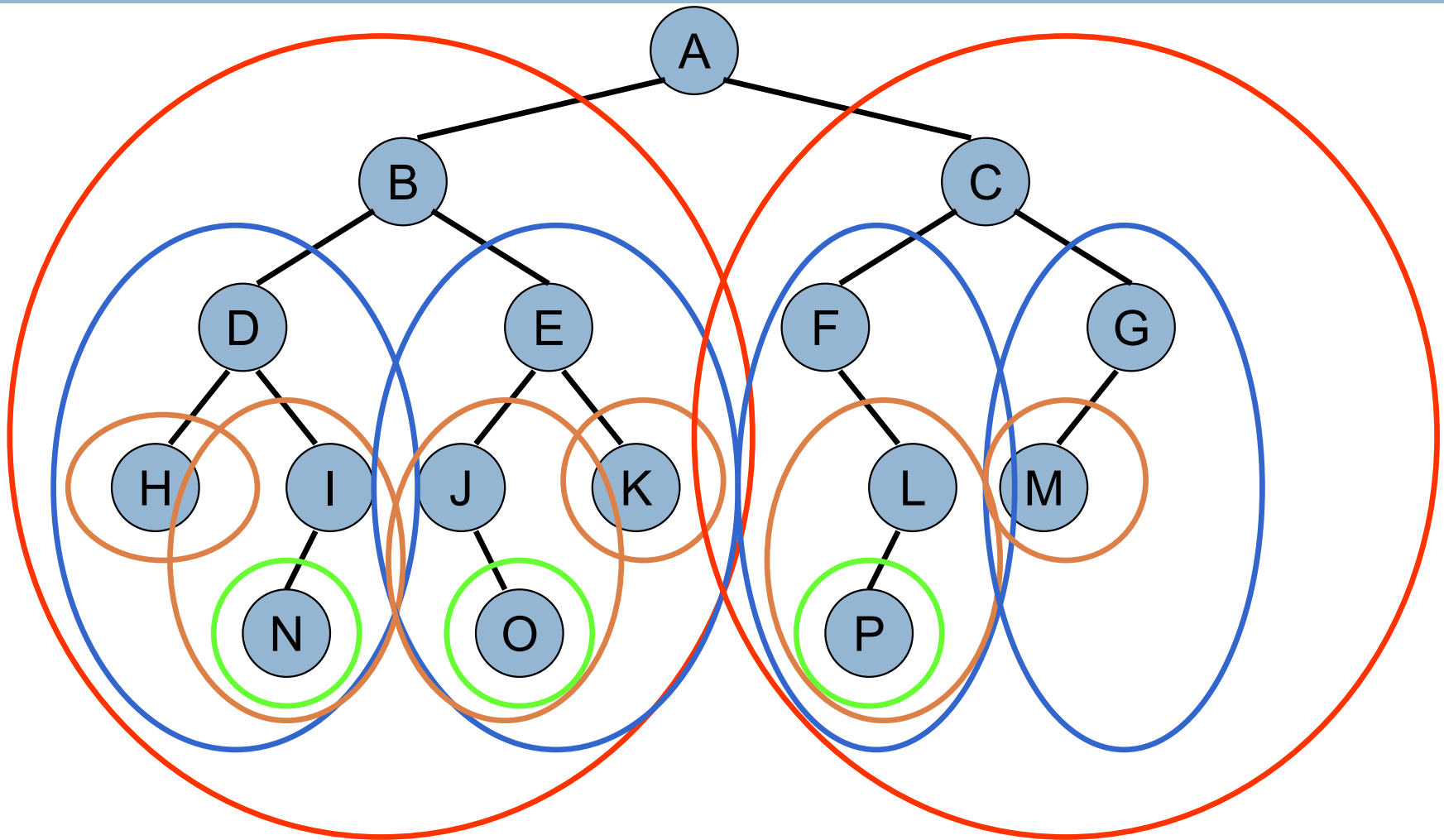
29

- ❑ Duyệt theo thứ tự giữa LNR (Left-Node-Right)
  - ❑ Kiểu duyệt này trước tiên thăm các nút của **cây con trái** sau đó thăm **nút gốc** rồi đến **cây con phải**
  - ❑ Thủ tục duyệt có thể trình bày đơn giản như sau:

```
void LNR(Tree t)
{
    if (t != NULL)
    {
        LNR(t->pLeft) ;
        Tham Node (t) //Xử lý nút t theo nhu cầu
        LNR(t->pRight) ;
    }
}
```

# Binary Tree - Duyệt cây nhị phân LNR

30



Kết quả: H D N I B J O E K A F P L C M G

# Binary Tree - Duyệt cây nhị phân

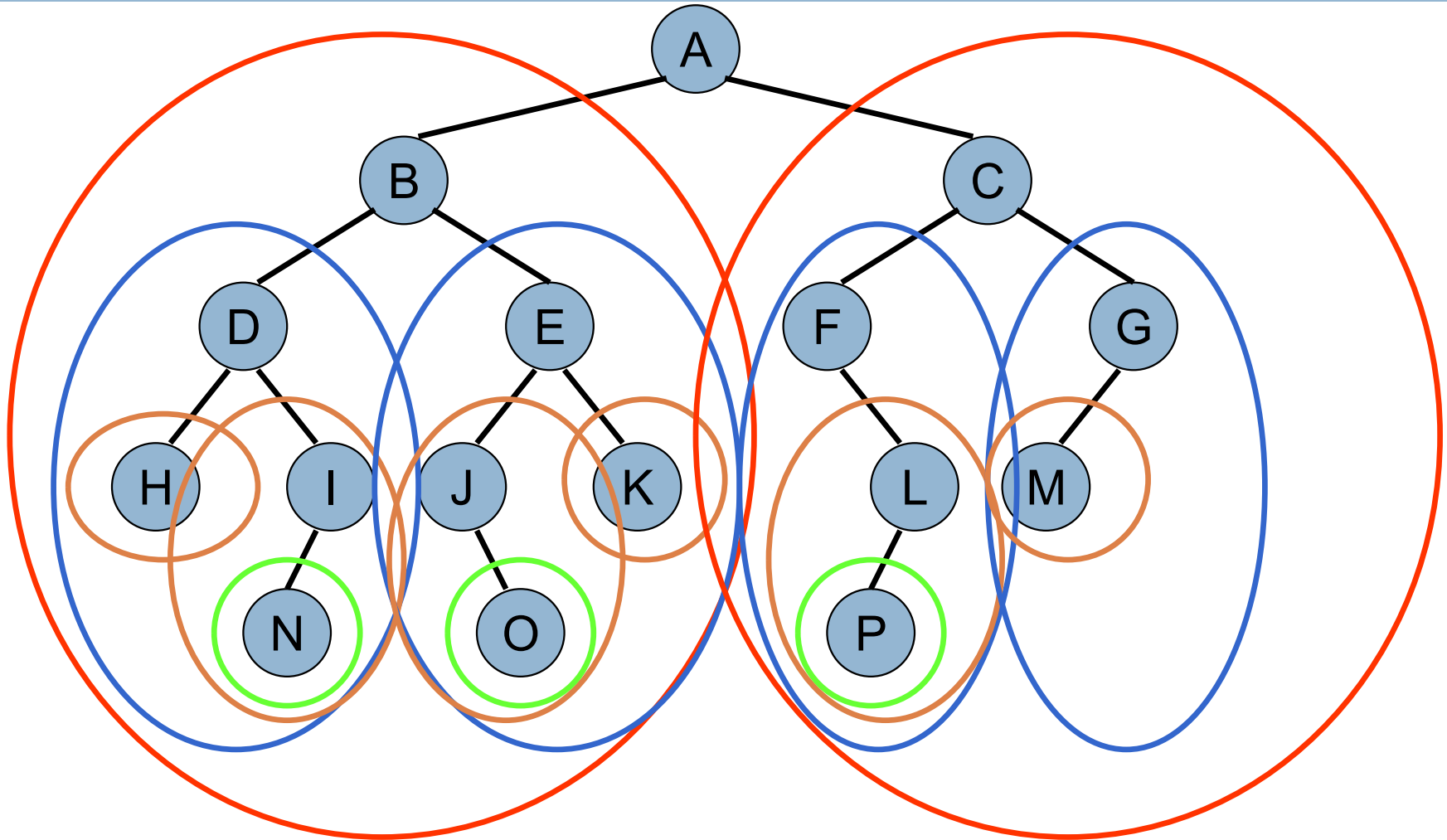
31

- Duyệt theo thứ tự sau LRN (Left-Right-Node)
  - ▣ Kiểu duyệt này trước tiên thăm các nút của **cây con trái** sau đó thăm đến **cây con phải** rồi cuối cùng mới thăm **nút gốc**
  - ▣ Thủ tục duyệt có thể trình bày đơn giản như sau:

```
void LRN(Tree t)
{
    if (t != NULL)
    {
        LRN(t->pLeft) ;
        LRN(t->pRight) ;
        Tham Node(t) // Xử lý tương ứng t theo nhu cầu
    }
}
```

# Binary Tree - Duyệt cây nhị phân LRN

32



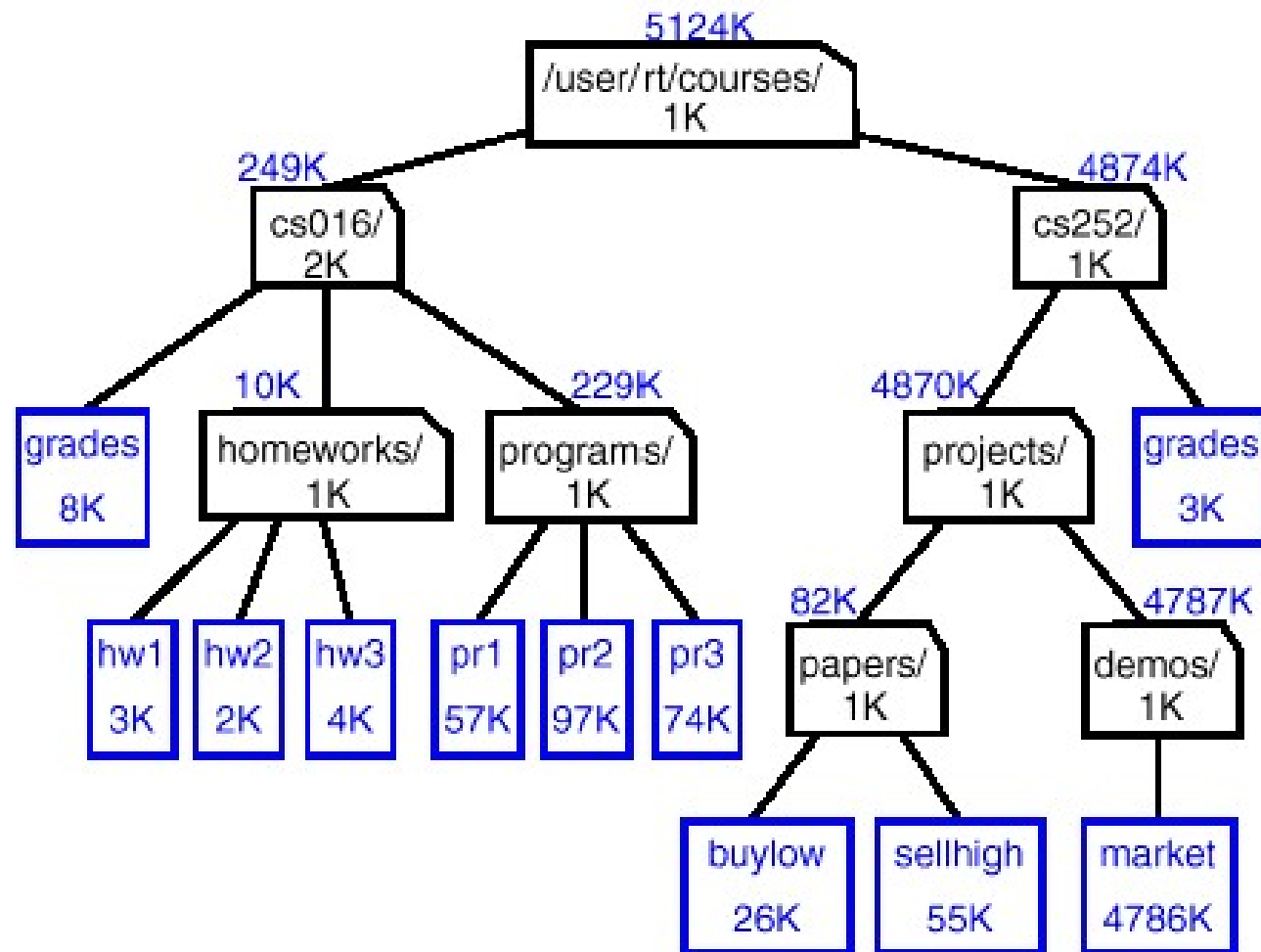
Kết quả: H N I D O J K E B P L F M G C A



# Binary Tree - Duyệt cây nhị phân LRN

33

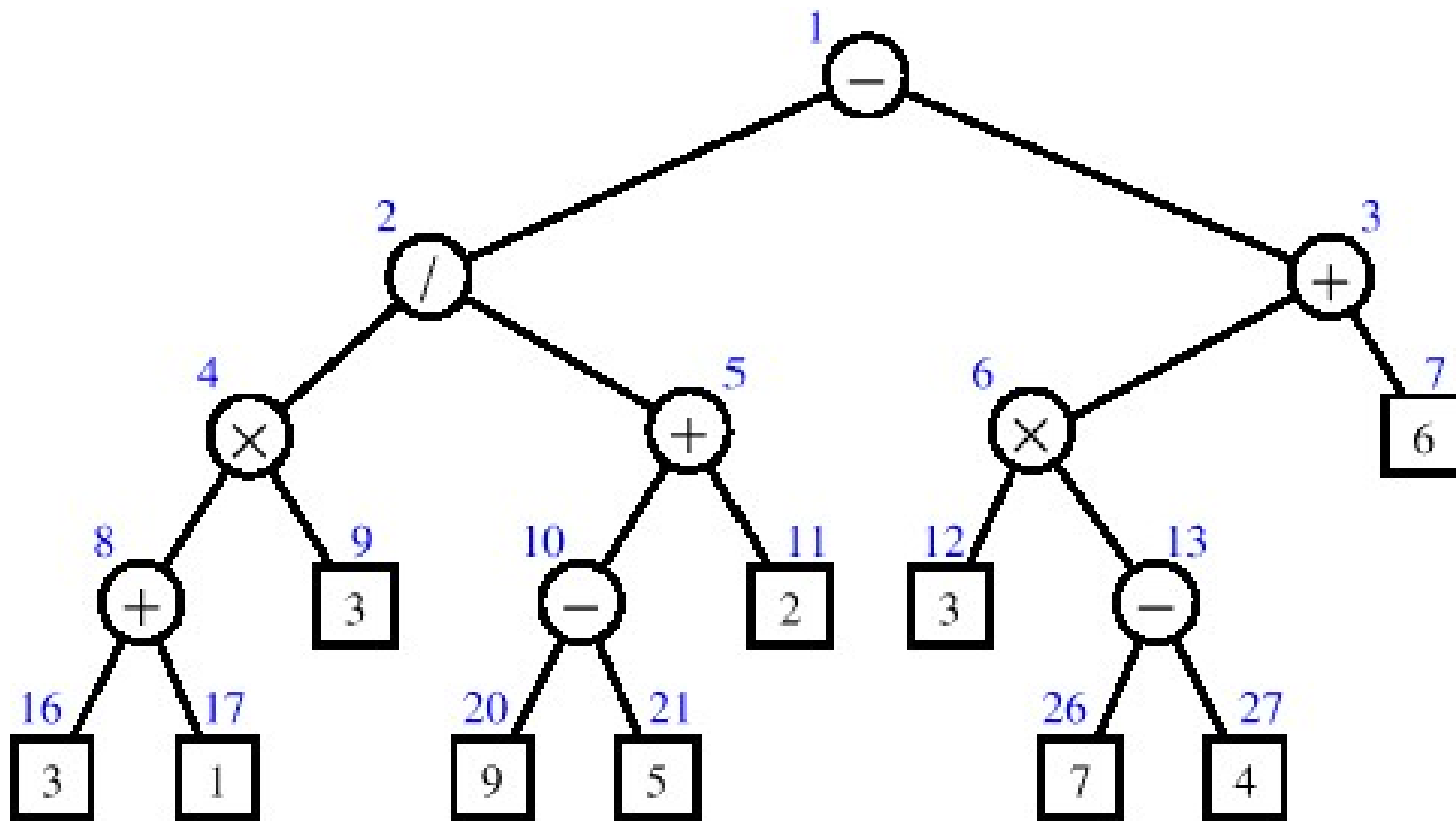
- Một ví dụ quen thuộc trong tin học về ứng dụng của duyệt theo thứ tự sau là việc xác định tổng kích thước của một thư mục trên đĩa



# Binary Tree - Duyệt cây nhị phân LRN

34

- Tính toán giá trị của biểu thức dựa trên cây biểu thức



$$(3 + 1) \times 3 / (9 - 5 + 2) - (3 \times (7 - 4) + 6) = -13$$

# Một cách biểu diễn cây nhị phân khác

35

- Đôi khi, khi định nghĩa cây nhị phân, người ta quan tâm đến cả quan hệ 2 chiều cha con chứ không chỉ một chiều như định nghĩa ở phần trên.
- Lúc đó, cấu trúc cây nhị phân có thể định nghĩa lại như sau:

```
typedef struct tagTNode
{
    DataType                Key;
    struct tagTNode*        pParent;
    struct tagTNode*        pLeft;
    struct tagTNode*        pRight;
} TNode;

typedef TNode    *TREE;
```

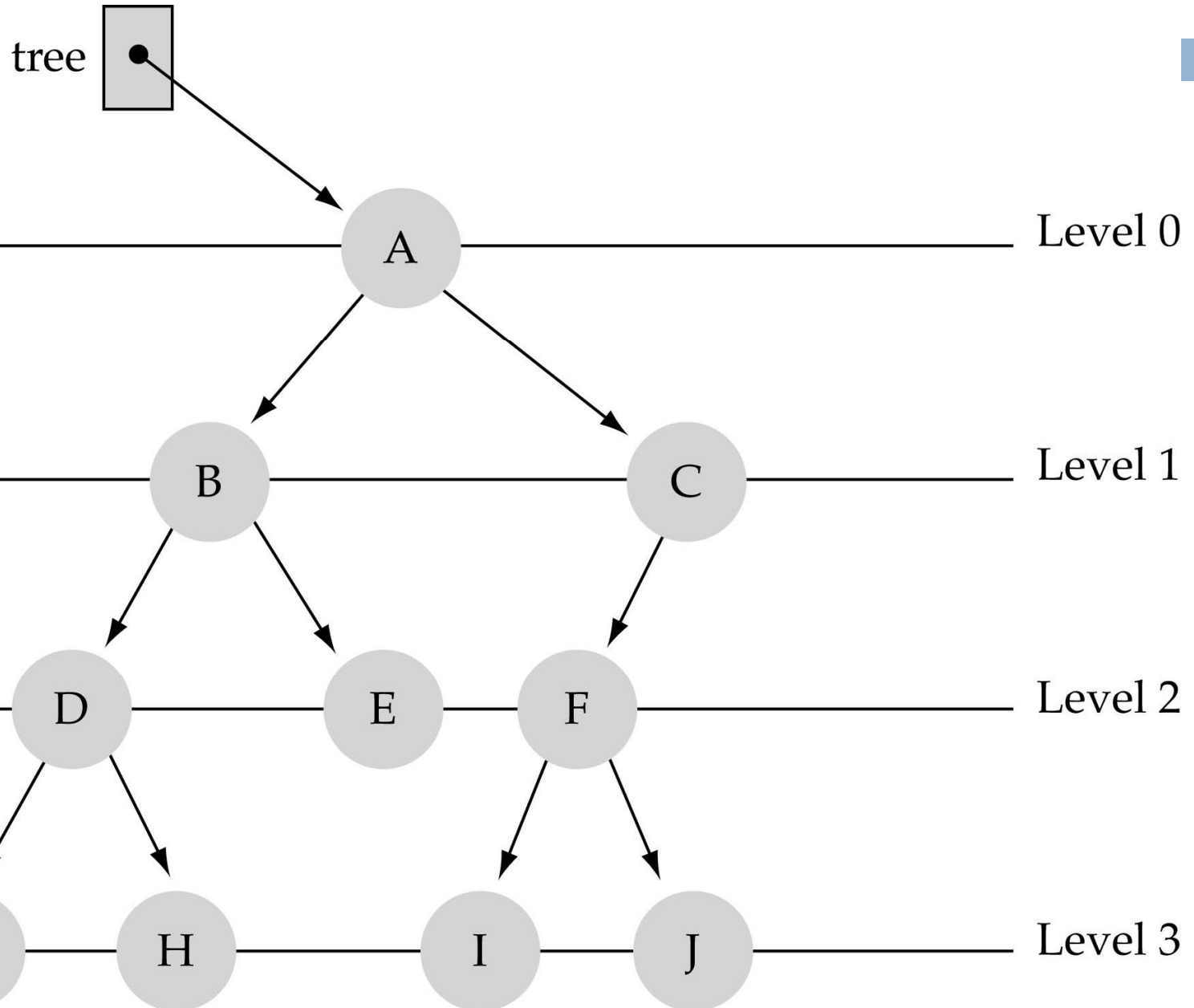
# Một số thao tác trên cây

36

- Đếm số node
- Đếm số node lá
- Tính chiều cao

# Đếm số node

37



# Đếm số node

38

- Số node (EmptyTree) = 0
- Số node (Tree) = 1 + Số node (Tree.Left)  
+ Số node (Tree.Right)

# Đếm số node lá

39

$$\text{Số nút lá}(\text{EmptyTree}) = 0$$

$$\text{Số nút lá}(\text{RootOnly}) = 1$$

$$\begin{aligned} \text{Số nút lá}(\text{Tree}) = & \text{Số nút lá}(\text{Tree.Left}) + \\ & \text{Số nút lá}(\text{Tree.Right}) \end{aligned}$$

# Tính chiều cao

40

$$\textit{Height}(\textit{Tree}) = 1 + \text{maximum}(\textit{Height}(\textit{Tree}.\textit{Left}), \textit{Height}(\textit{Tree}.\textit{Right}))$$

$$\textit{Depth}(\textit{EmptyTree}) = -1$$



# Binary Search Tree

- Trong chương 6, chúng ta đã làm quen với một số cấu trúc dữ liệu động. Các cấu trúc này có sự mềm dẻo nhưng lại bị hạn chế trong việc tìm kiếm thông tin trên chúng (chỉ có thể tìm kiếm tuần tự)
- Nhu cầu tìm kiếm là rất quan trọng. Vì lý do này, người ta đã đưa ra cấu trúc cây để thỏa mãn nhu cầu trên
- Tuy nhiên, nếu chỉ với cấu trúc cây nhị phân đã định nghĩa ở trên, việc tìm kiếm còn rất mơ hồ
- Cần có thêm một số ràng buộc để cấu trúc cây trở nên chặt chẽ, dễ dùng hơn
- Một cấu trúc như vậy chính là **cây nhị phân tìm kiếm**

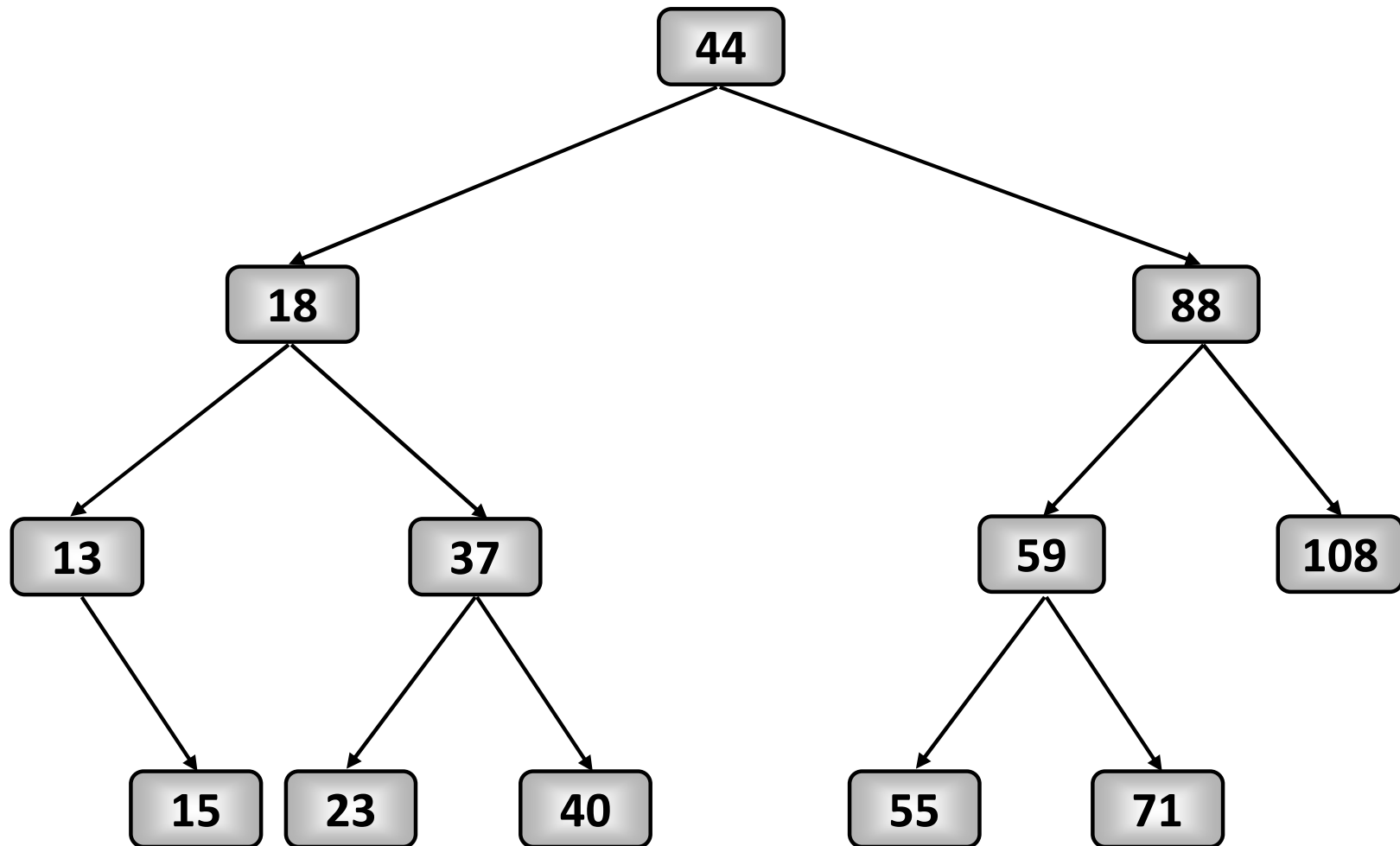
# Binary Search Tree - Định nghĩa

42

- Cây nhị phân tìm kiếm (CNPTK) là cây nhị phân trong đó **tại mỗi nút**, khóa của nút đang xét **lớn hơn** khóa của tất cả các nút thuộc **cây con trái** và **nhỏ hơn** khóa của tất cả các nút thuộc **cây con phải**
- Nhờ ràng buộc về khóa trên CNPTK, việc tìm kiếm trở nên có định hướng
- Nếu số nút trên cây là  $N$  thì chi phí tìm kiếm trung bình chỉ khoảng  $\log_2 N$
- Trong thực tế, khi xét đến cây nhị phân chủ yếu người ta xét CNPTK

# Binary Search Tree – Ví dụ

43



# Binary Search Tree – Biểu diễn

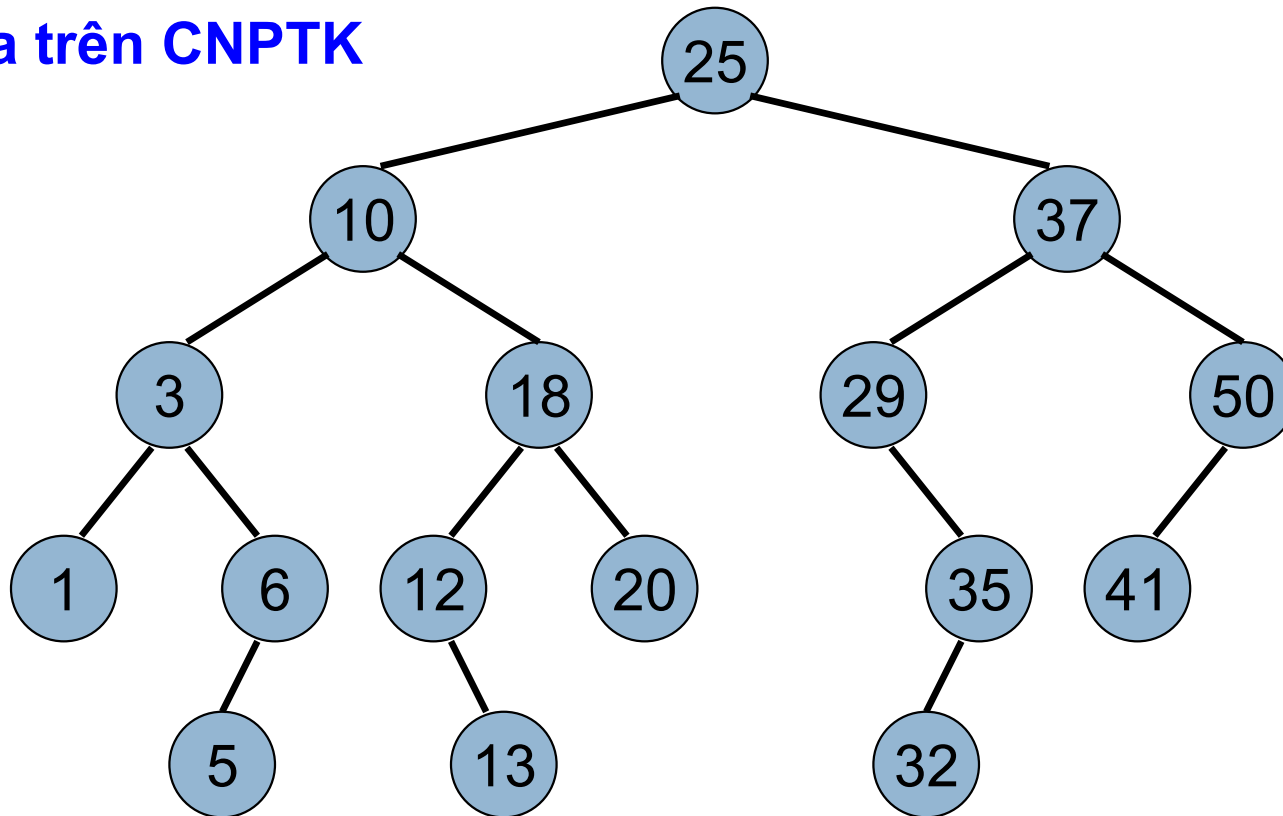
44

- Cấu trúc dữ liệu của CNPTK là cấu trúc dữ liệu biểu diễn cây nhị phân nói chung (???)
- Thao tác duyệt cây trên CNPTK hoàn toàn giống như trên cây nhị phân (???)
  - ▣ Chú ý: khi duyệt theo thứ tự giữa, trình tự các nút duyệt qua sẽ cho ta một dãy các nút theo thứ tự tăng dần của khóa

# Binary Search Tree – Duyệt cây

45

Duyệt giữa trên CNPTK

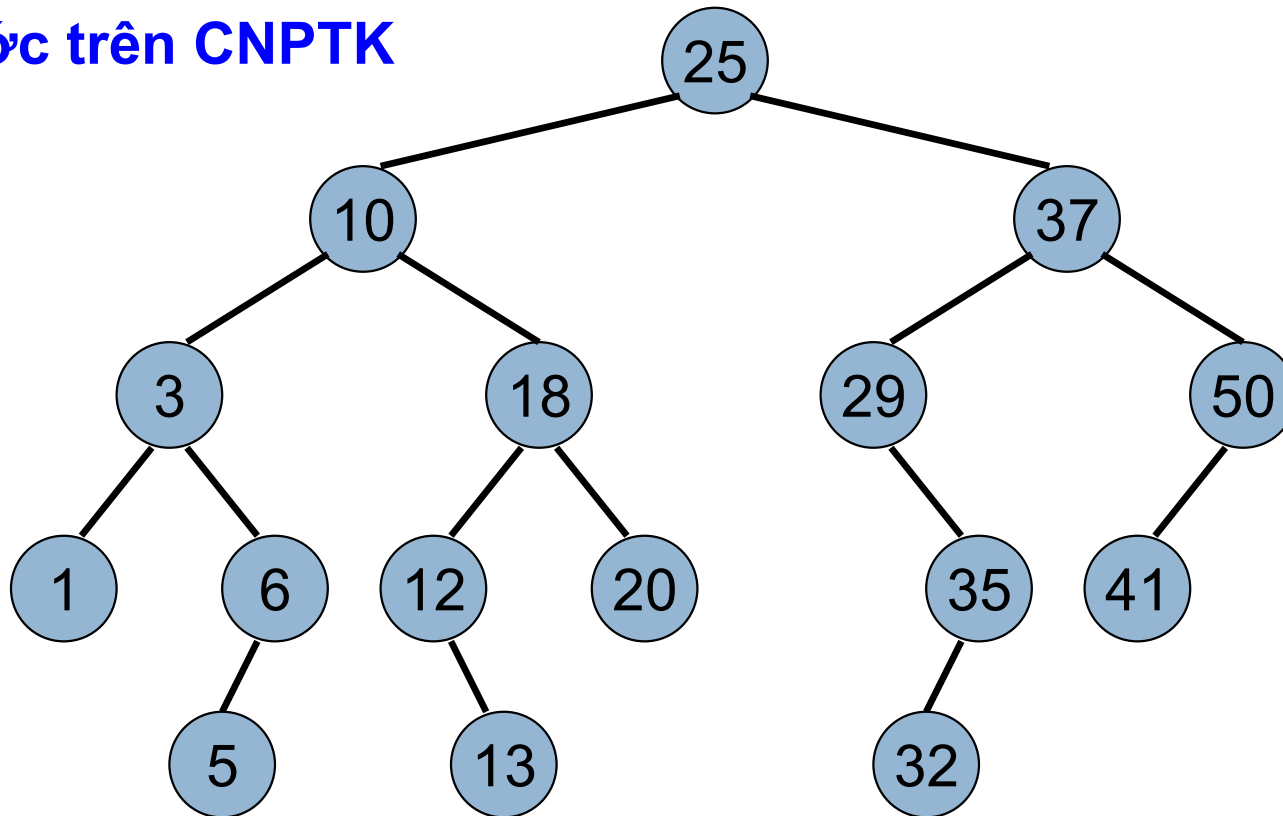


Duyệt inorder: 1 3 5 6 10 12 13 18 20 25 29 32 35 37 41 50

# Binary Search Tree – Duyệt cây

46

Duyệt trước trên CNPTK

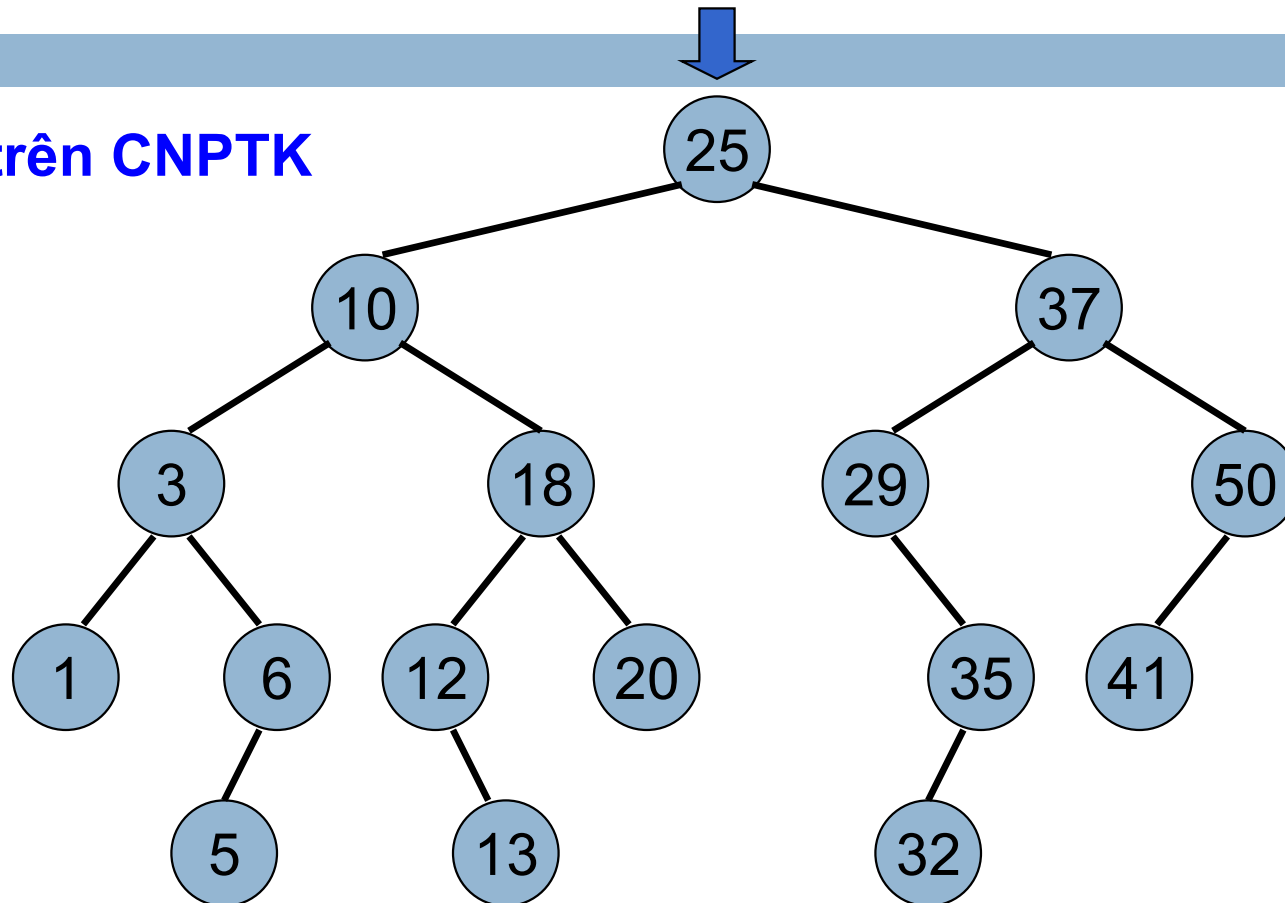


Duyệt preorder:

# Binary Search Tree – Tìm kiếm

47

Tìm kiếm trên CNPTK



Không phải là cây tìm kiếm nhị phân

Tìm kiếm 13

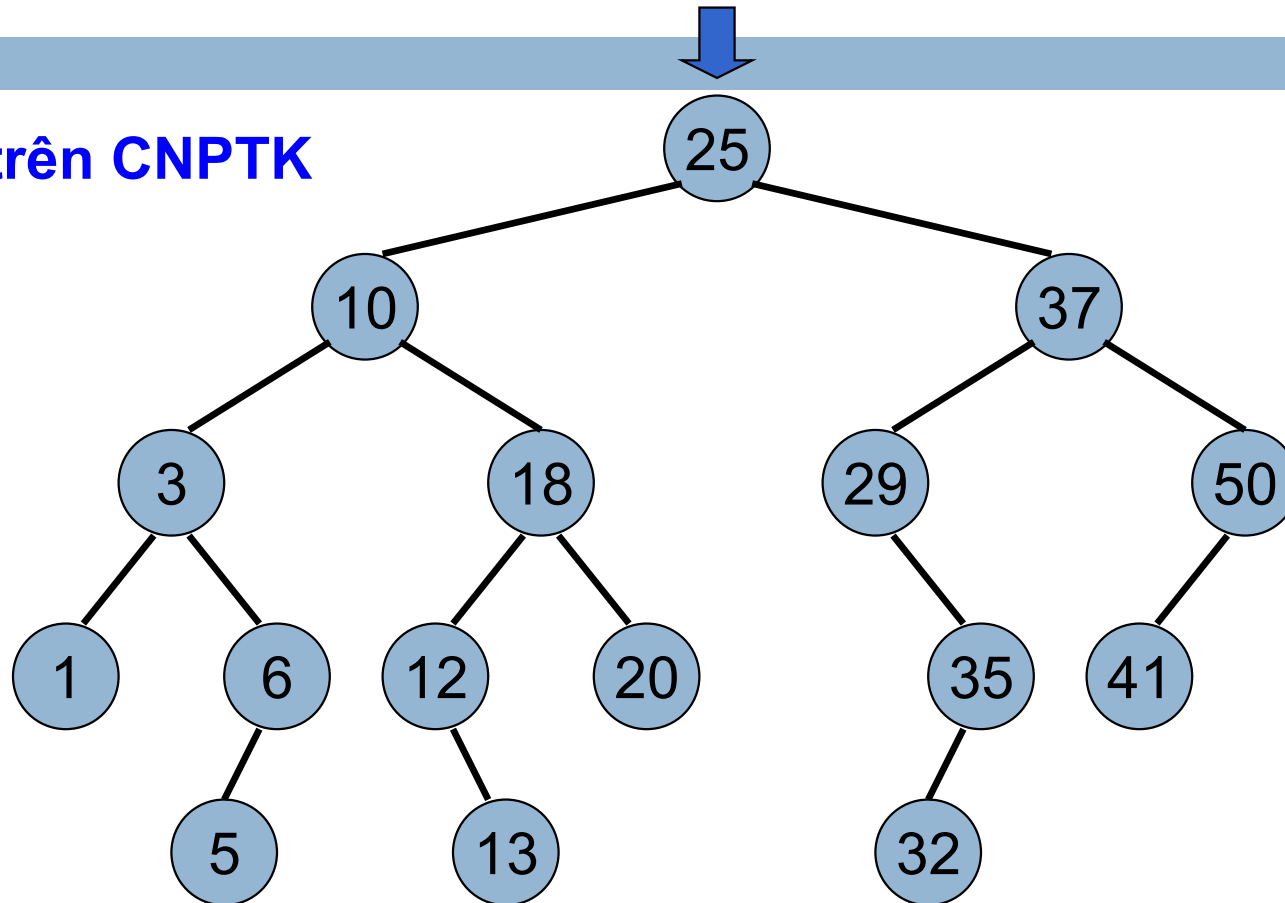
Tìm thấy

Số node duyệt: 5  
Số lần so sánh: 9

# Binary Search Tree – Tìm kiếm

48

Tìm kiếm trên CNPTK



Khóa gốc nhỏ hơn

Tìm kiếm 14 Không tìm thấy

Số node duyệt: 5  
Số lần so sánh: 10



# Binary Search Tree – Tìm kiếm

49

- Tìm một phần tử x trong CNPTK (dùng đệ quy):

```
TNode* searchNode (Tree T, DataType X)
{
    if (T)
    {
        if (T->Key == X)
            return T;
        if (T->Key > X)
            return searchNode (T->pLeft, X);
        return searchNode (T->pRight, X);
    }
    return NULL;
}
```

# Binary Search Tree – Tìm kiếm

50

- Tìm một phần tử x trong CNPTK (dùng vòng lặp):

```
TNode * searchNode (Tree T, DataType x)
{
    TNode *p = T;
    while (p != NULL)
    {
        if (x == p->Key)    return p;
        else
            if (x < p->Key) p = p->pLeft;
            else p = p->pRight;
    }
    return NULL;
}
```

# Binary Search Tree – Tìm kiếm

51

- Nhận xét:
  - ▣ Số lần so sánh tối đa phải thực hiện để tìm phần tử  $X$  là  $h$ , với  $h$  là chiều cao của cây
  - ▣ Như vậy thao tác tìm kiếm trên CNPTK có  $n$  nút tốn chi phí trung bình khoảng  $O(\log_2 n)$

# Binary Search Tree – Thêm

52

- Việc thêm một phần tử X vào cây phải bảo đảm **điều kiện ràng buộc** của CNPTK
- Ta có thể thêm vào nhiều chỗ khác nhau trên cây, nhưng nếu thêm vào **một nút ngoài** sẽ là tiện lợi nhất do ta có thể thực hiện quá trình tương tự thao tác tìm kiếm
- Khi chấm dứt quá trình tìm kiếm cũng chính là lúc tìm được chỗ cần thêm
- Hàm insert trả về giá trị:
  - 1          khi không đủ bộ nhớ
  - 0          khi gặp nút cũ
  - 1          khi thêm thành công

# Binary Search Tree – Thêm

53

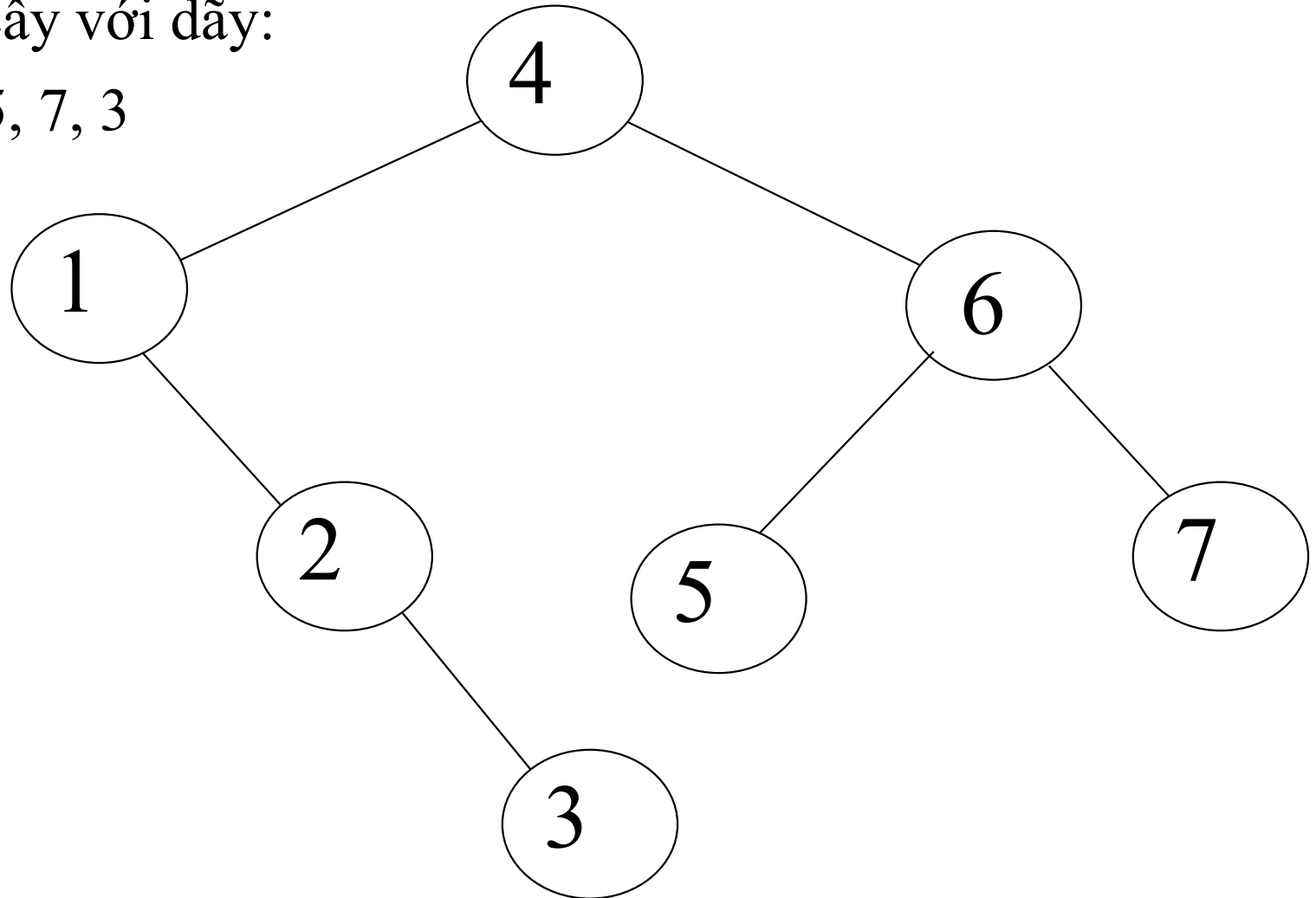
- Thêm một phần tử vào cây

```
int insertNode (Tree &T, DataType X)
{   if (T) {
        if(T->data == X) return 0;
        if(T->data > X)
            return insertNode(T->pLeft, X);
        else
            return insertNode(T->pRight, X);
    }
    T = new TNode;
    if (T == NULL) return -1;
    T->data = X;
    T->pLeft = T->pRight = NULL;
    return 1;
}
```

# Binary Search Tree – Thêm

54

- Ví dụ tạo cây với dãy:  
4, 6, 1, 2, 5, 7, 3

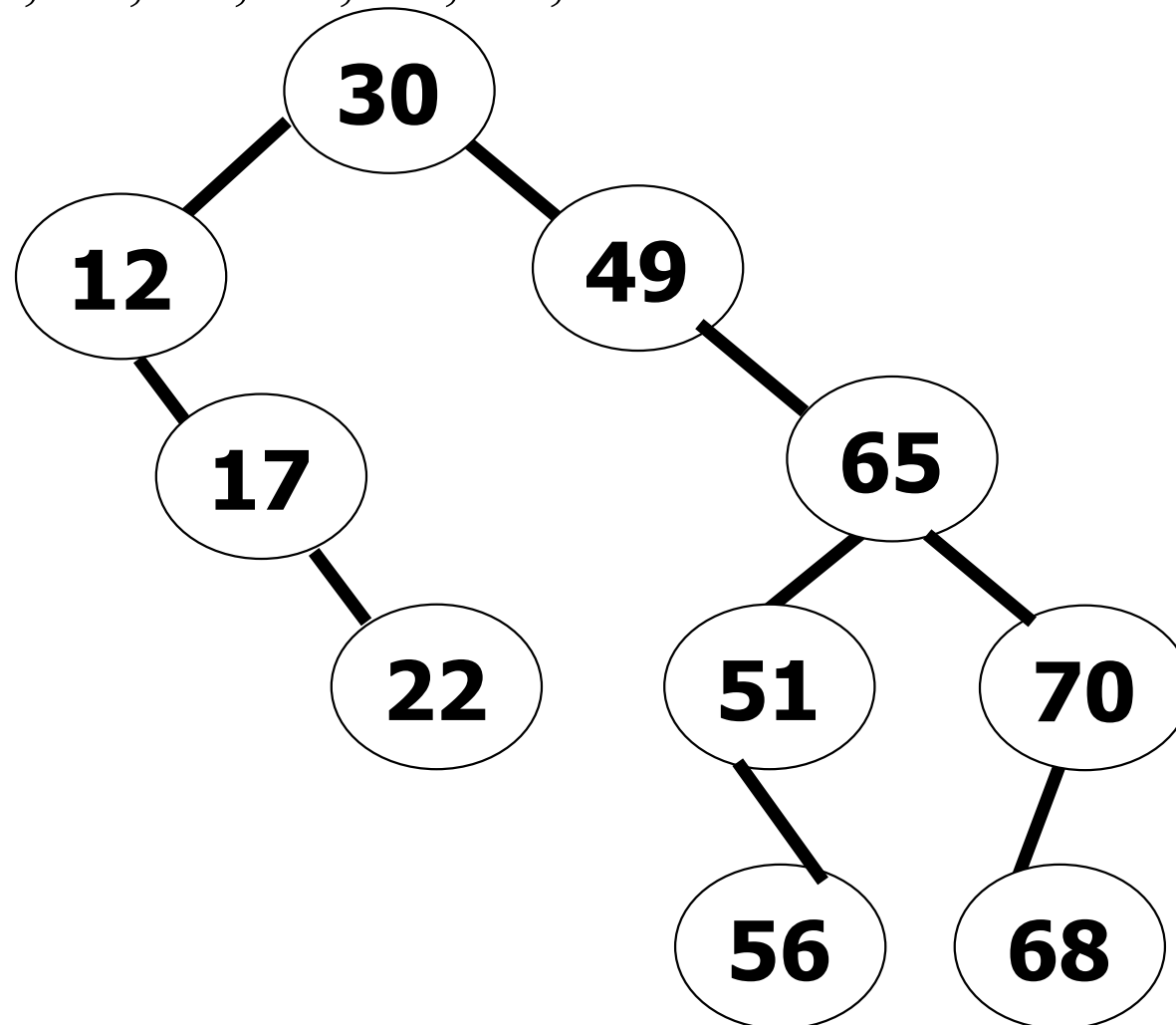


# Binary Search Tree – Thêm

55

- Ví dụ tạo cây với dãy:

30, 12, 17, 49, 22, 65, 51, 56, 70, 68



# Binary Search Tree – Hủy một phần tử có khóa X

56

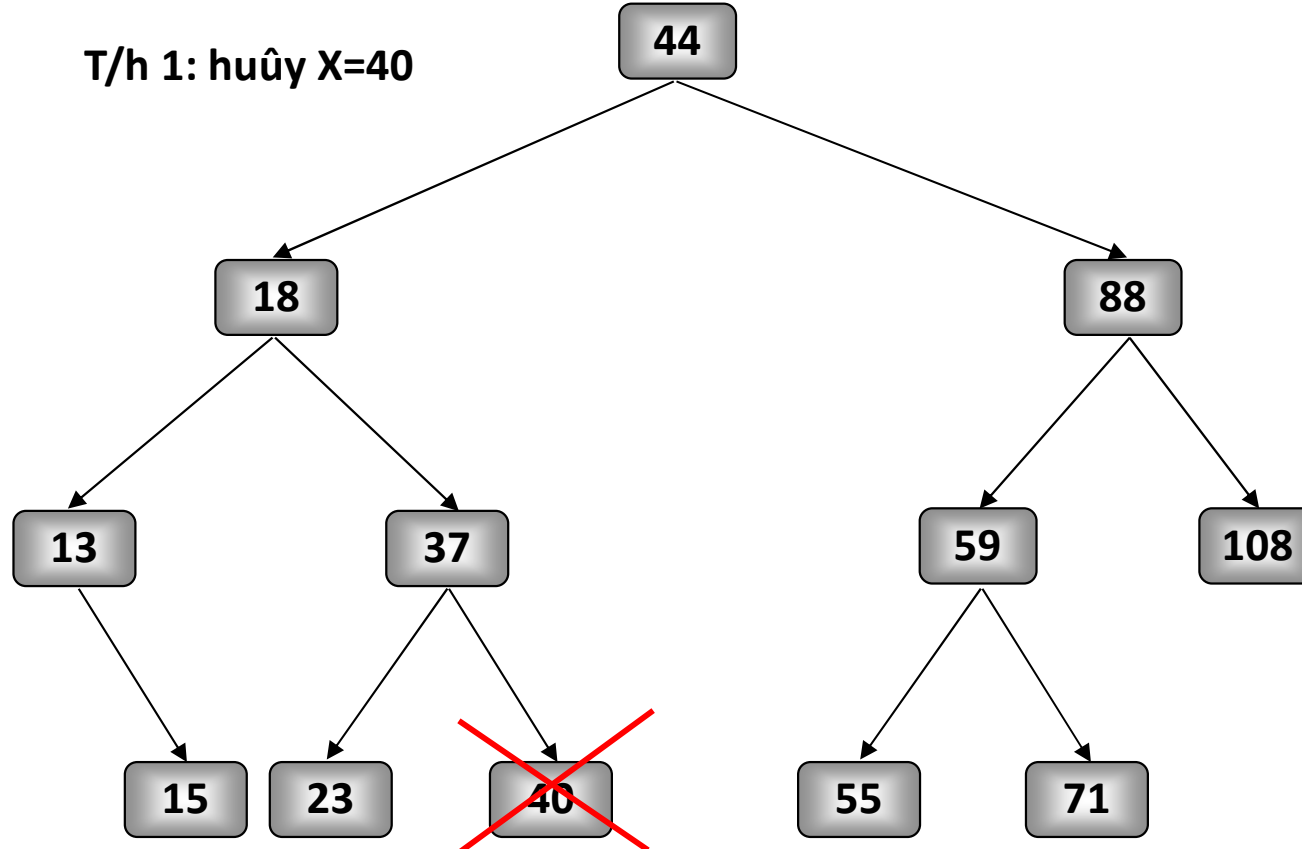
- Việc hủy một phần tử X ra khỏi cây phải bảo đảm điều kiện ràng buộc của CNPTK
- Có 3 trường hợp khi hủy nút X có thể xảy ra:
  - ▣ X là nút lá
  - ▣ X chỉ có 1 con (trái hoặc phải)
  - ▣ X có đủ cả 2 con



# Binary Search Tree – Hủy một phần tử có khóa X

57

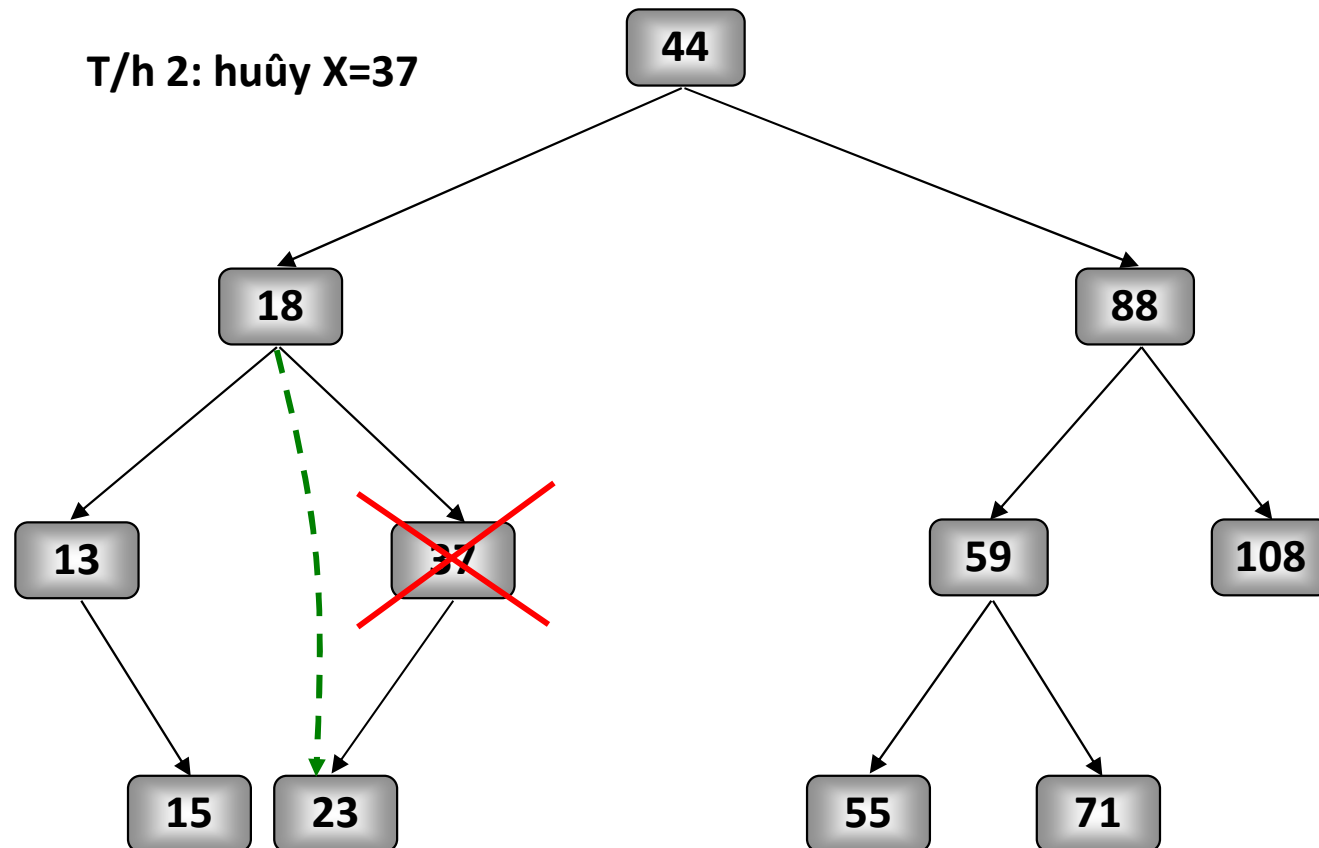
- Trường hợp 1: X là nút lá
  - ▣ Chỉ đơn giản hủy X vì nó không móc nối đến phần tử nào khác



# Binary Search Tree – Hủy một phần tử có khóa X

58

- Trường hợp 2: X chỉ có 1 con (trái hoặc phải)
  - ▣ Trước khi hủy X ta móc nối cha của X với con duy nhất của nó



# Binary Search Tree – Hủy một phần tử có khóa X

59

- Trường hợp 3: X có đủ 2 con:
  - ▣ Không thể hủy trực tiếp do X có đủ 2 con
  - ▣ Hủy gián tiếp:
    - Thay vì hủy X, ta sẽ tìm một phần tử thế mạng Y. Phần tử này có tối đa một con
    - Thông tin lưu tại Y sẽ được chuyển lên lưu tại X
    - Sau đó, nút bị hủy thật sự sẽ là Y giống như 2 trường hợp đầu
- ▣ Vấn đề: chọn Y sao cho khi lưu Y vào vị trí của X, cây vẫn là CNPTK

# Binary Search Tree – Hủy một phần tử có khóa X

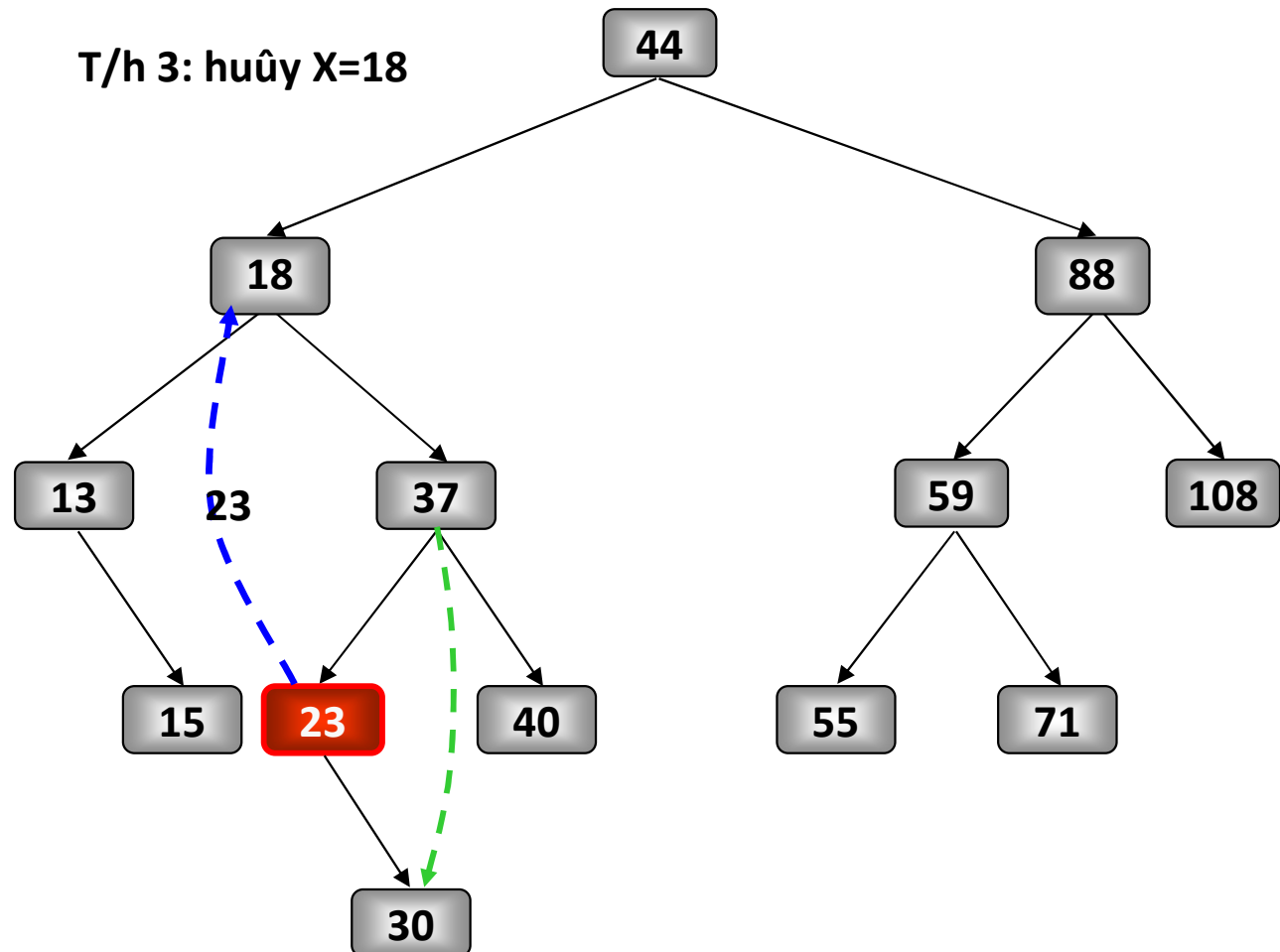
60

- Trường hợp 3: X có đủ 2 con:
  - ▣ Có 2 phần tử thỏa mãn yêu cầu:
    - Phần tử trái nhất trên cây con phải
    - Phần tử phải nhất trên cây con trái
  - ▣ Việc chọn lựa phần tử nào là phần tử thế mạng hoàn toàn phụ thuộc vào ý thích của người lập trình
  - ▣ Ở đây, ta sẽ chọn phần tử phải nhất trên cây con trái làm phần tử thế mạng

# Binary Search Tree – Hủy một phần tử có khóa X

61

- Trường hợp 3: X có đủ 2 con:
  - ▣ Khi hủy phần tử  $X=18$  ra khỏi cây, phần tử 23 là phần tử thế mạng:



# Binary Search Tree – Hủy một phần tử có khóa X

62

- Trường hợp 3: X có đủ 2 con:
  - ▣ Hàm *delNode* trả về giá trị 1, 0 khi hủy thành công hoặc không có X trong cây:  
int *delNode*(Tree &T, DataType X)
  - ▣ Hàm *searchStandFor* tìm phần tử thế mạng cho nút p  
void *searchStandFor*(Tree &p, Tree &q)

# Binary Search Tree – Hủy một phần tử có khóa X

63

```
int delNode(Tree &T, DataType X)
{
    if (T == NULL)    return 0;
    if (T->data > X)   return delNode(T->pLeft, X);
    if (T->data < X)   return delNode(T->pRight, X);
    TNode* p = T;
    if (T->pLeft == NULL)
        T = T->pRight;
    else
        if (T->pRight == NULL)    T = T->pLeft;
        else // T có đủ 2 con
            { TNode*    q = T->pRight;
              searchStandFor(p, q);
            }
    delete p;
}
```

# Binary Search Tree – Hủy một phần tử có khóa X

64

- Tìm phần tử thế mạng

```
void searchStandFor(Tree &p, Tree &q)
{
    if (q->pLeft)
        searchStandFor(p, q->pLeft);
    else
    {
        p->data = q->data;
        p = q;
        q = q->pRight;
    }
}
```



# Xóa node (Ko đệ qui)

65

```
procedure BSTDelete(X: TKey)
var
p, q, Node_xoa, Child: NODE;
begin
p := Root; q := NULL; {Về sau, khi p trở sang nút khác,
ta luôn giữ cho q^ luôn là cha của p^}
while p ≠ NULL do {Tìm xem trong cây có khoá X không?}
begin
if p->data= X then Break; {Tìm thấy}
q := p;
if X < p->data then p := p->pLeft
else p := p->pRight;
end;
```

# Xóa node (Ko đệ qui)

66

```
if p = NULL then Exit; {X không tồn tại trong BST nên
không xoá được}

if (p->pLeft ≠ NULL) and (p->pRight ≠ NULL) then {p^ có cả
con trái và con phải}

begin

Node_xoa := p; {Giữ lại nút chứa khoá X}

q := p; p := p->pLeft; {Chuyển sang nhánh con trái để tìm
nút cực phải}

while p->pRight ≠ NULL do

begin

    q := p; p := p-> pRight;

end;

Node_xoa-> data := p-> data; {Chuyển giá trị từ nút cực
phải trong nhánh con trái lên Node^}
```

# Xóa node (Ko đệ qui)

67

{Nút bị xoá giờ đây là nút  $p^{\wedge}$ , nó chỉ có nhiều nhất một con}

{Nếu  $p^{\wedge}$  có một nút con thì đem Child trở tới nút con đó, nếu không có thì Child = nil}

**if**  $p \rightarrow pLeft \neq \text{NULL}$  **then** Child :=  $p \rightarrow pLeft$

**else** Child :=  $p \rightarrow pRight$ ;

**if**  $p = \text{Root}$  **then** Root := Child; {Nút  $p^{\wedge}$  bị xoá là gốc cây}

**else** {Nút bị xoá  $p^{\wedge}$  không phải gốc cây thì lấy mỗi nối từ cha của nó là  $q^{\wedge}$  nối thẳng tới Child}

**if**  $q \rightarrow pLeft = p$  **then**  $q \rightarrow pLeft := \text{Child}$

**else**  $q \rightarrow pRight := \text{Child}$ ;

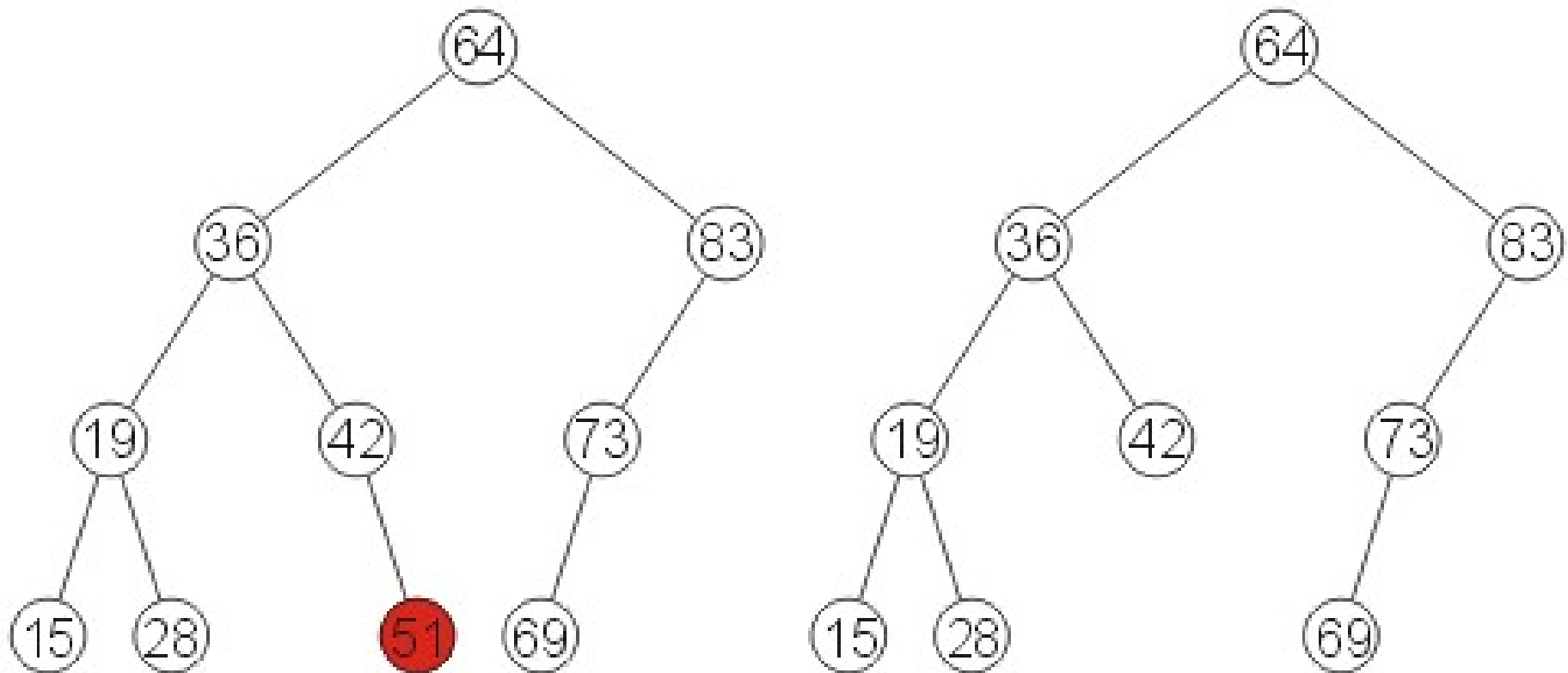
Delete( $p$ ) ;

**end**;

# Binary Search Tree – Hủy một phần tử có khóa X

68

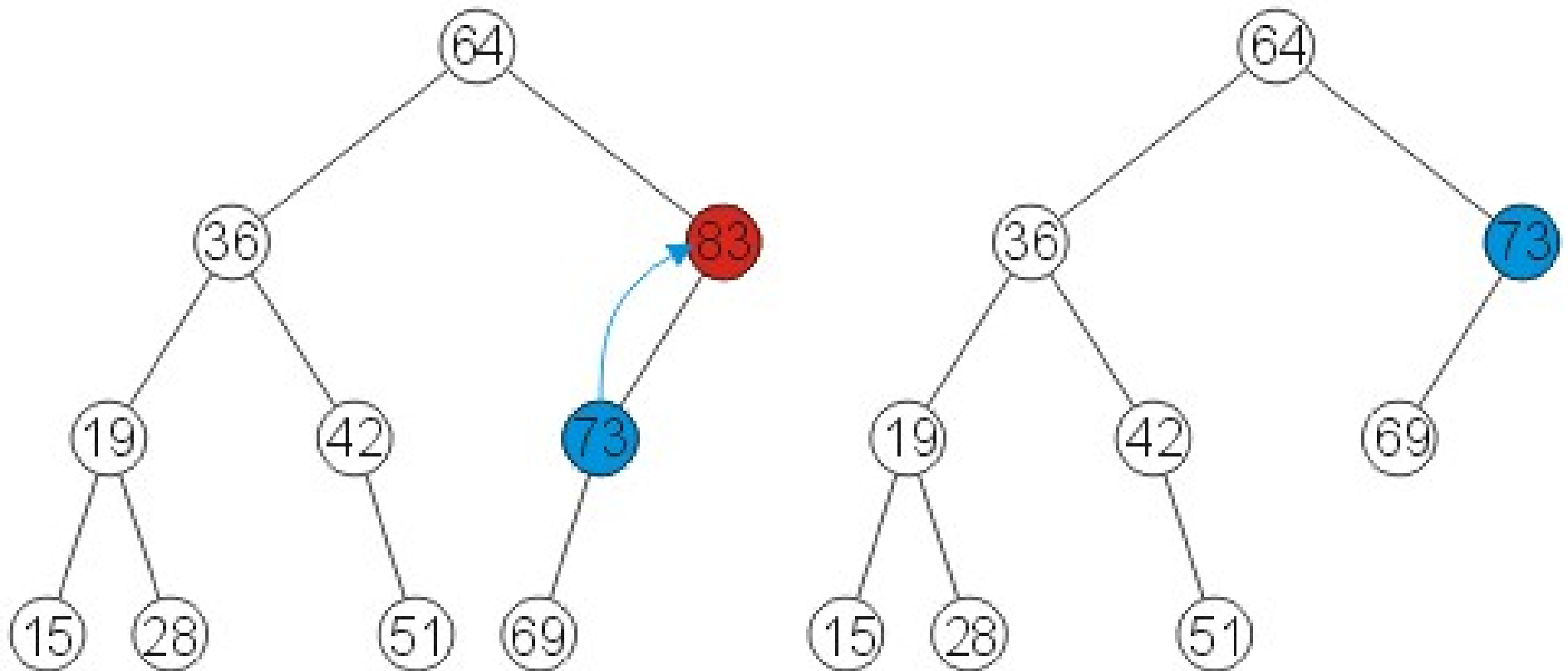
□ Ví dụ xóa 51:



# Binary Search Tree – Hủy một phần tử có khóa X

69

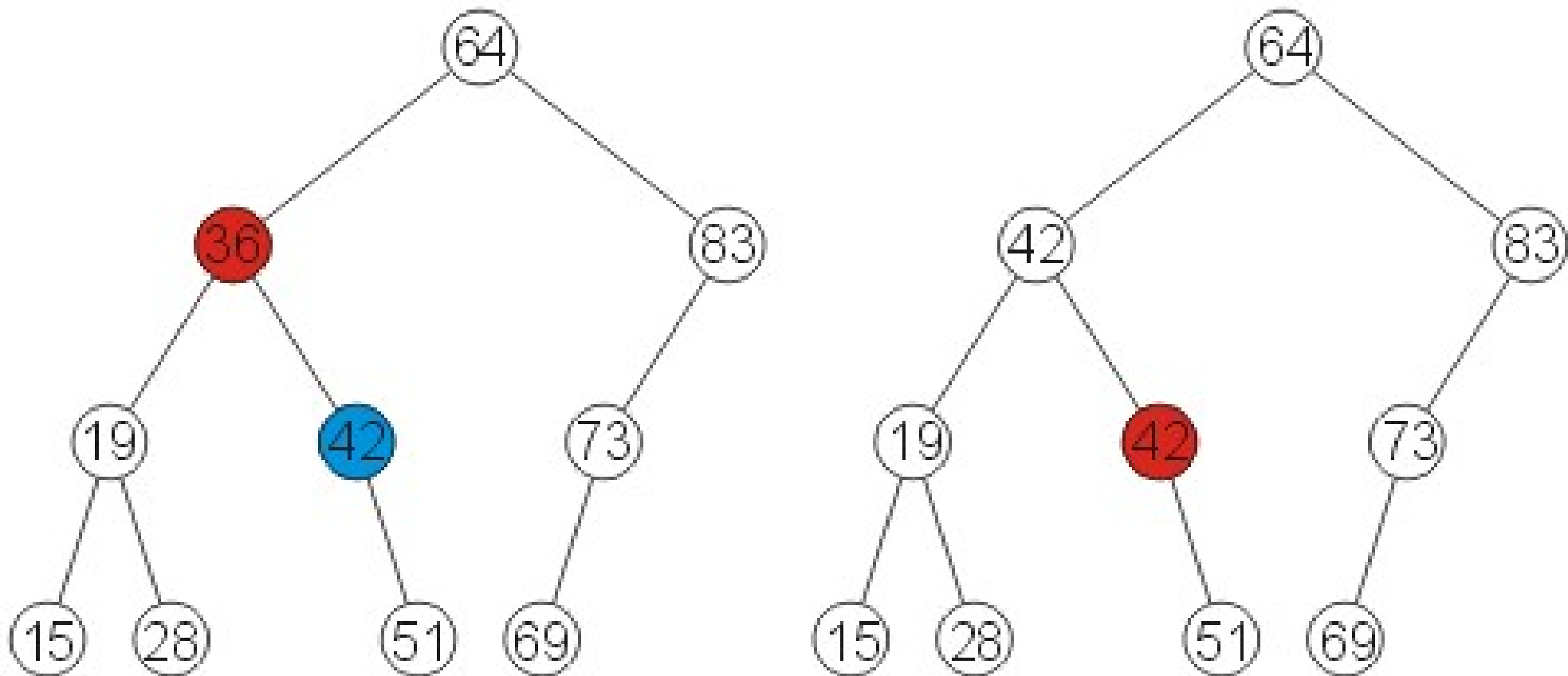
□ Ví dụ xóa 83:



# Binary Search Tree – Hủy một phần tử có khóa X

70

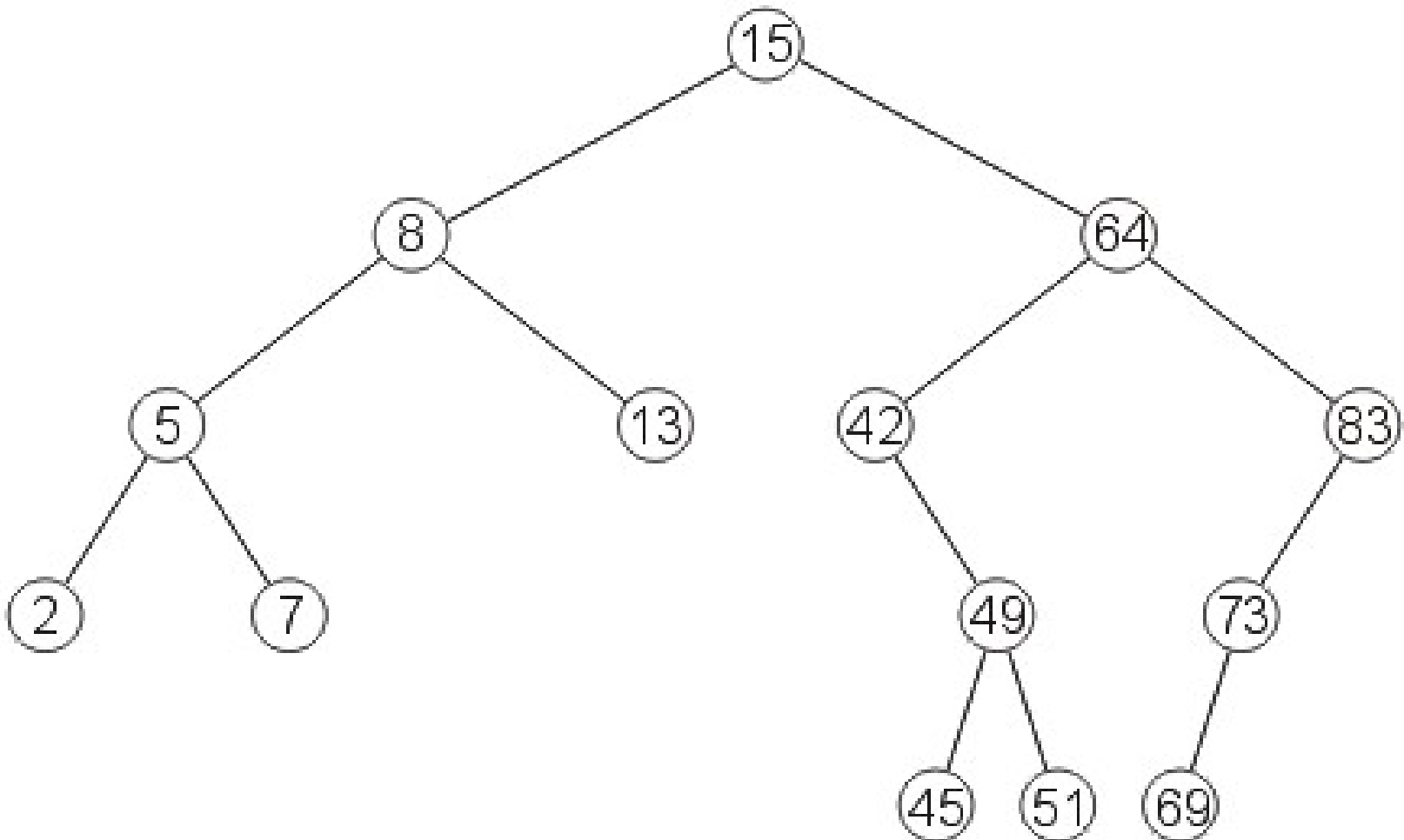
□ Ví dụ xóa 36:



# Binary Search Tree – Hủy một phần tử có khóa X

71

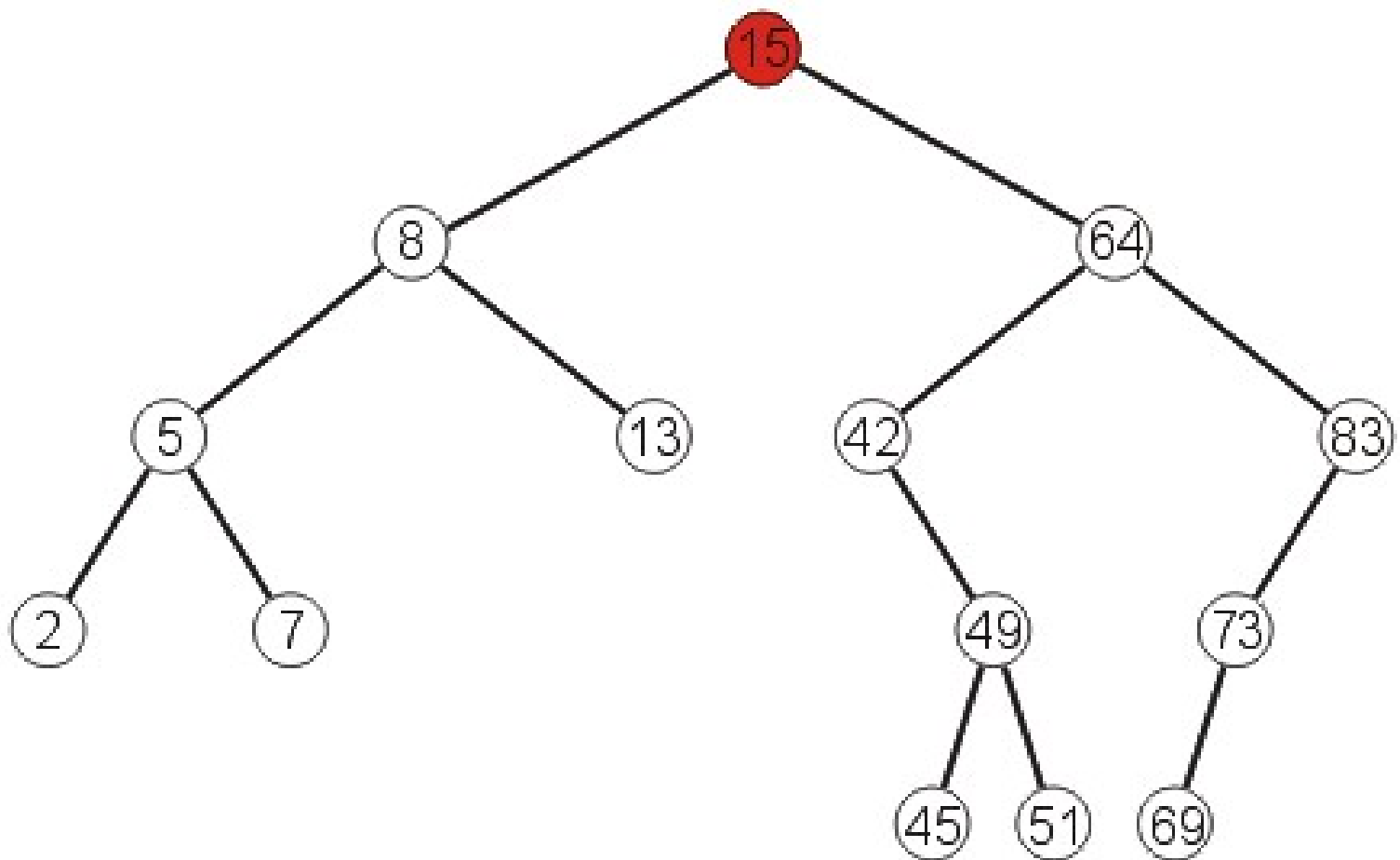
- Xóa nút gốc (2 lần):



# Binary Search Tree – Hủy một phần tử có khóa X

72

- Ví dụ xóa 15:

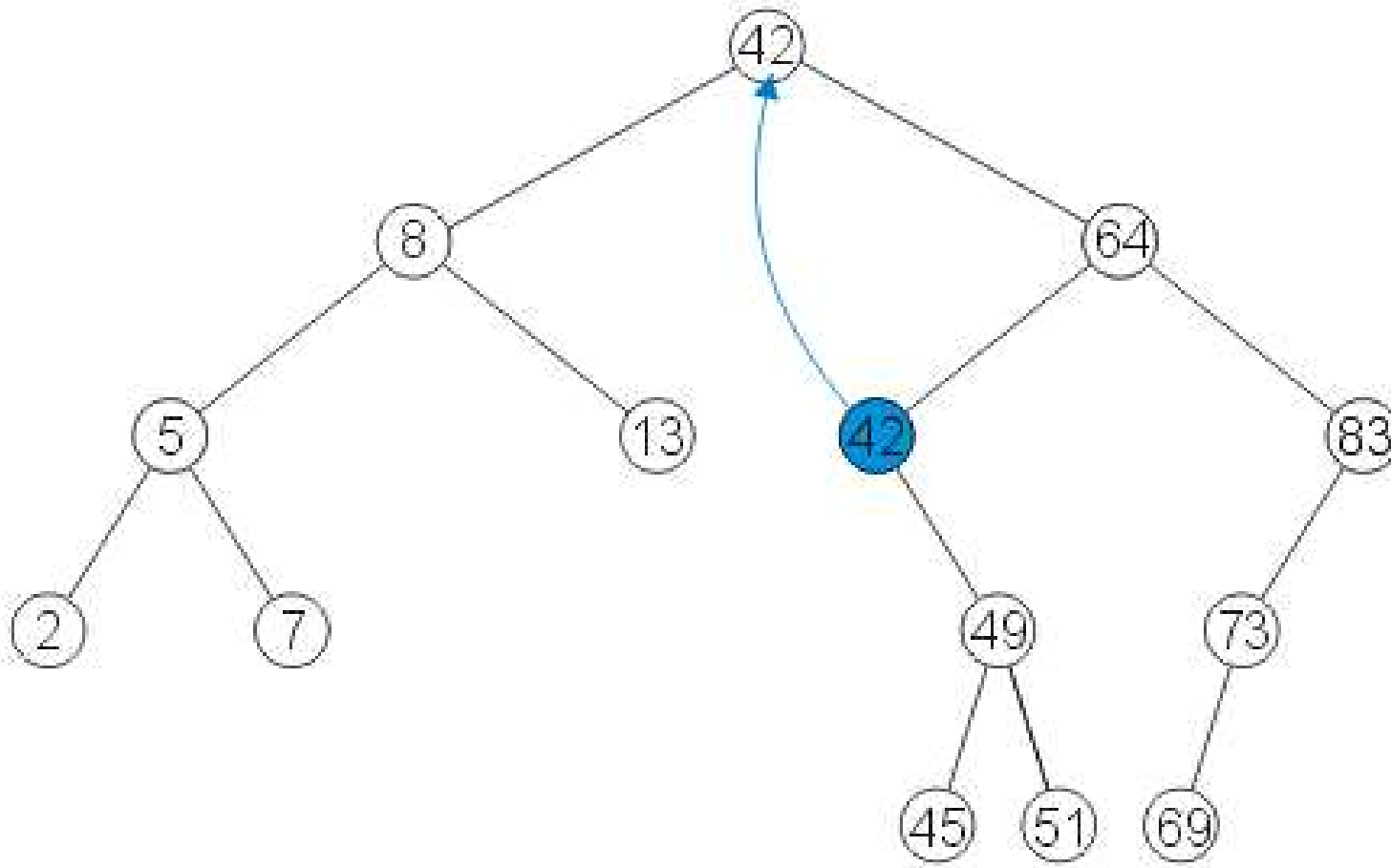




# Binary Search Tree – Hủy một phần tử có khóa X

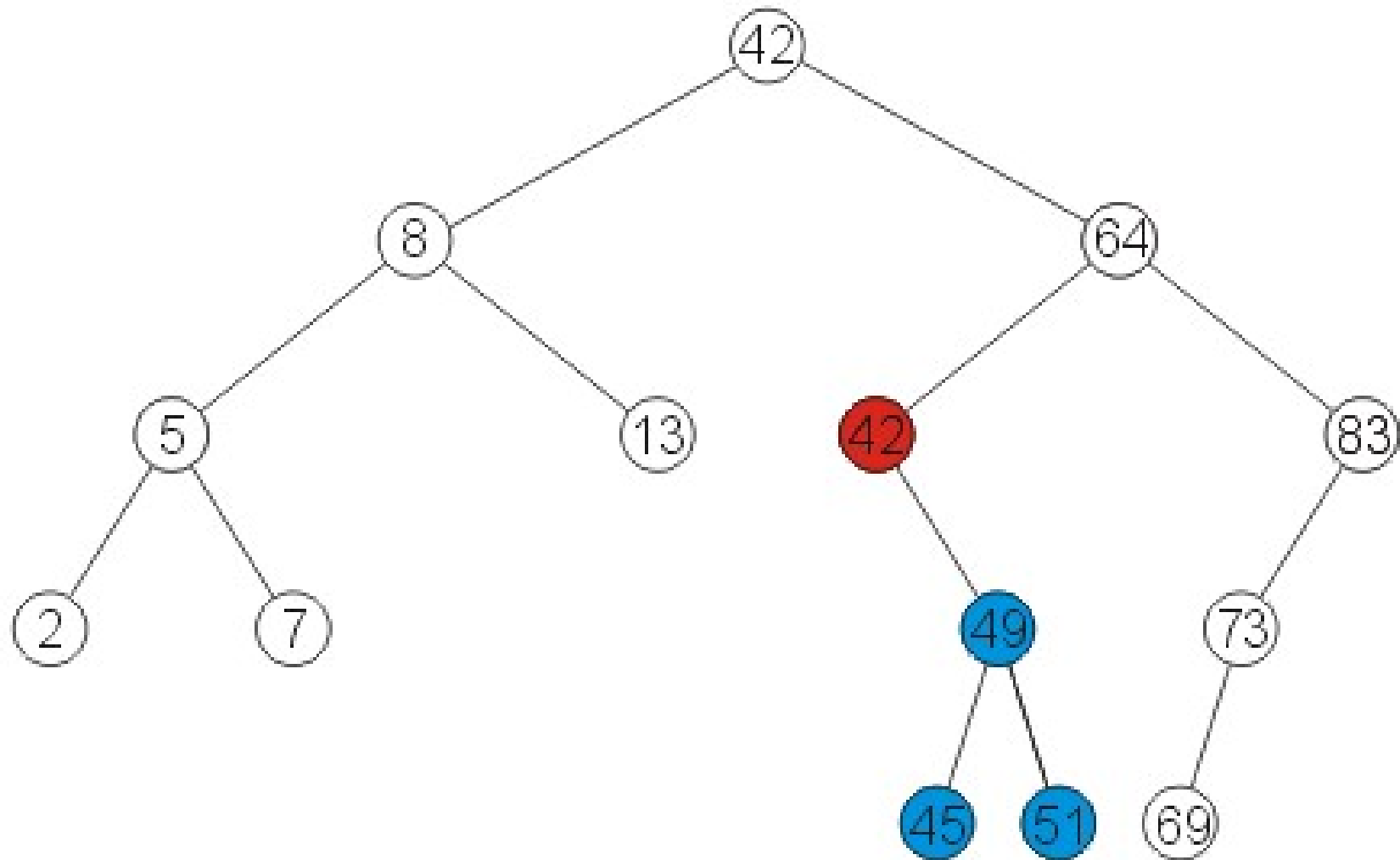
73

- 42 là thể mạng



# Binary Search Tree – Hủy một phần tử có khóa X

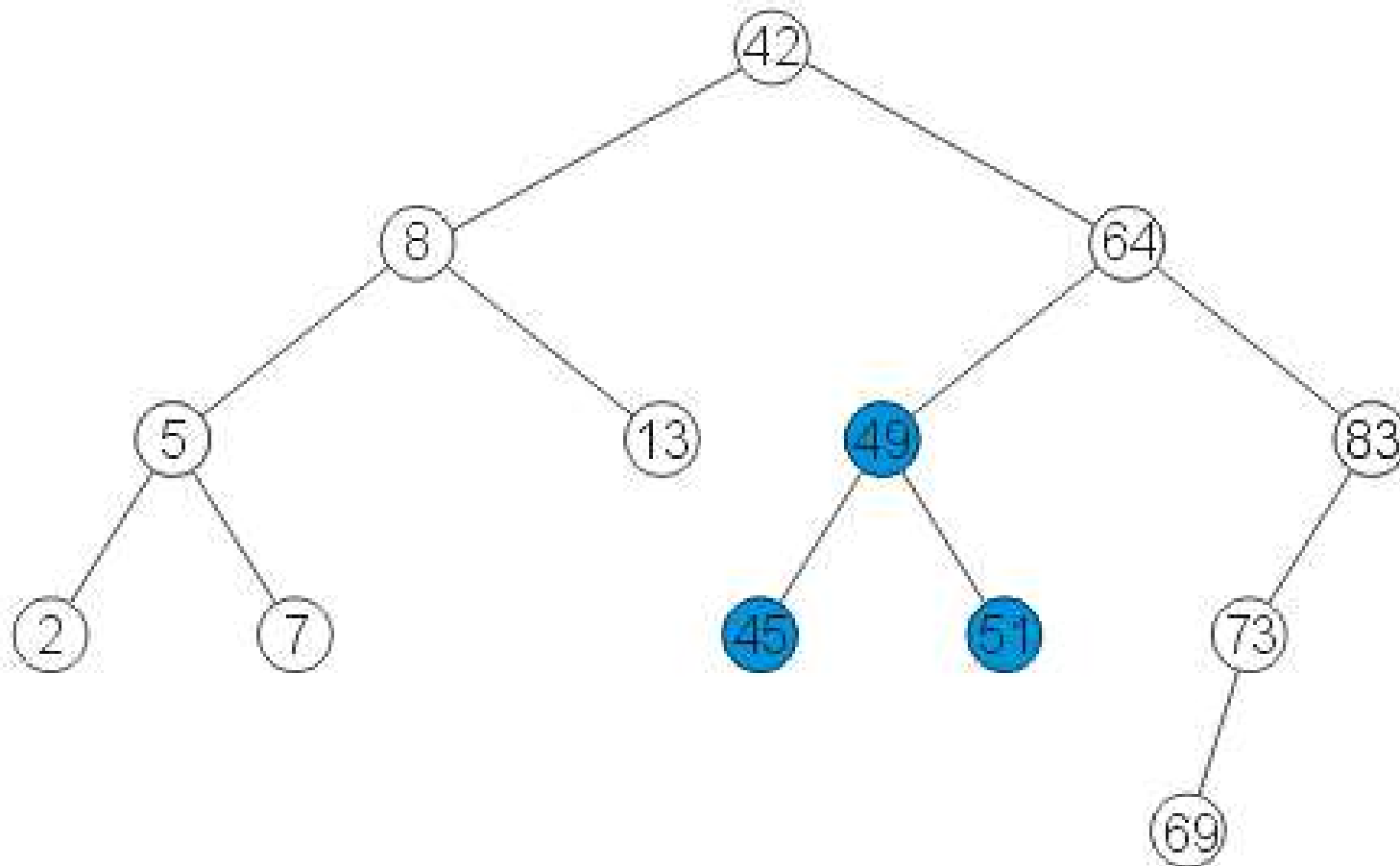
74



# Binary Search Tree – Hủy một phần tử có khóa X

75

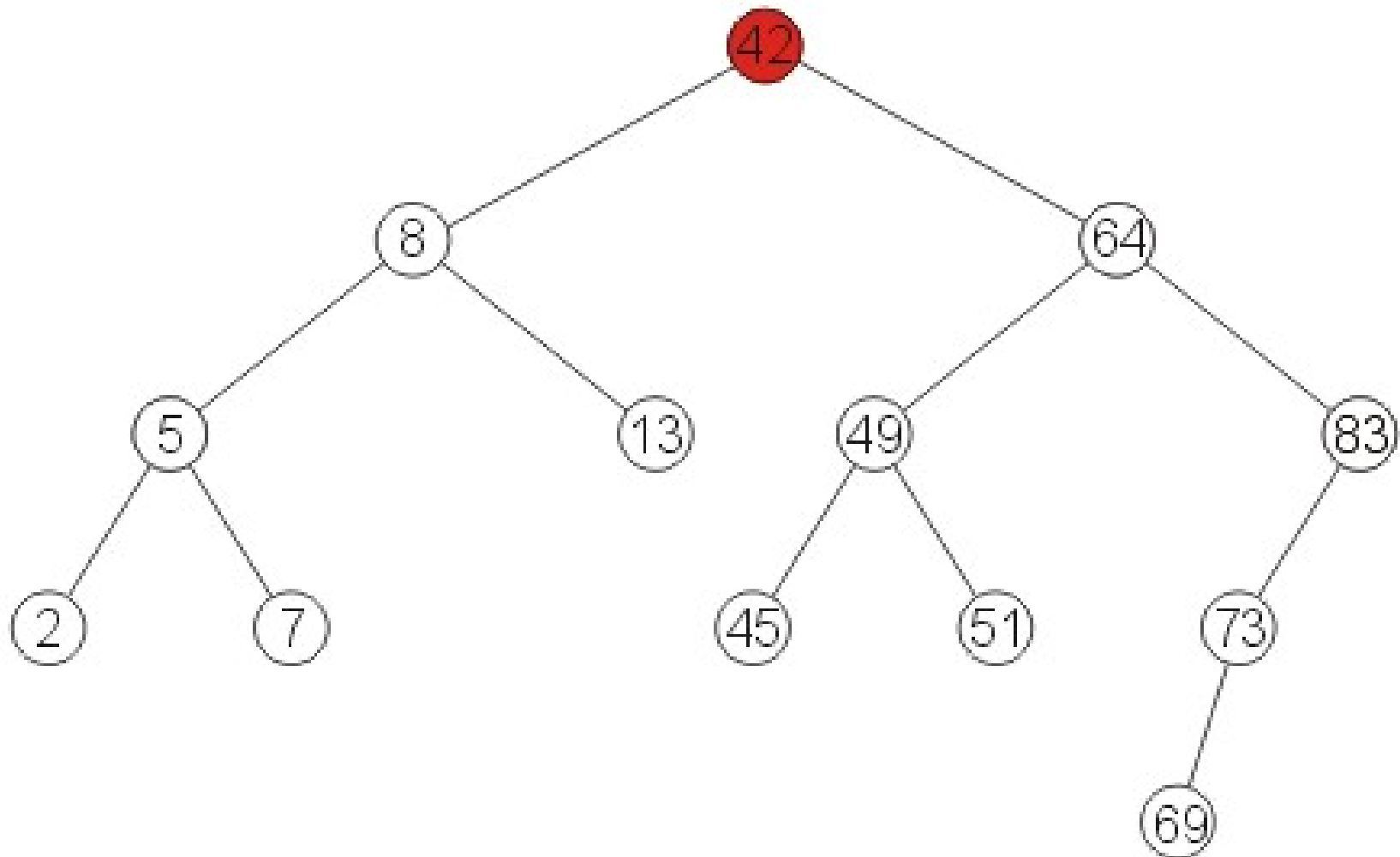
- Kết quả xóa lần 1:



# Binary Search Tree – Hủy một phần tử có khóa X

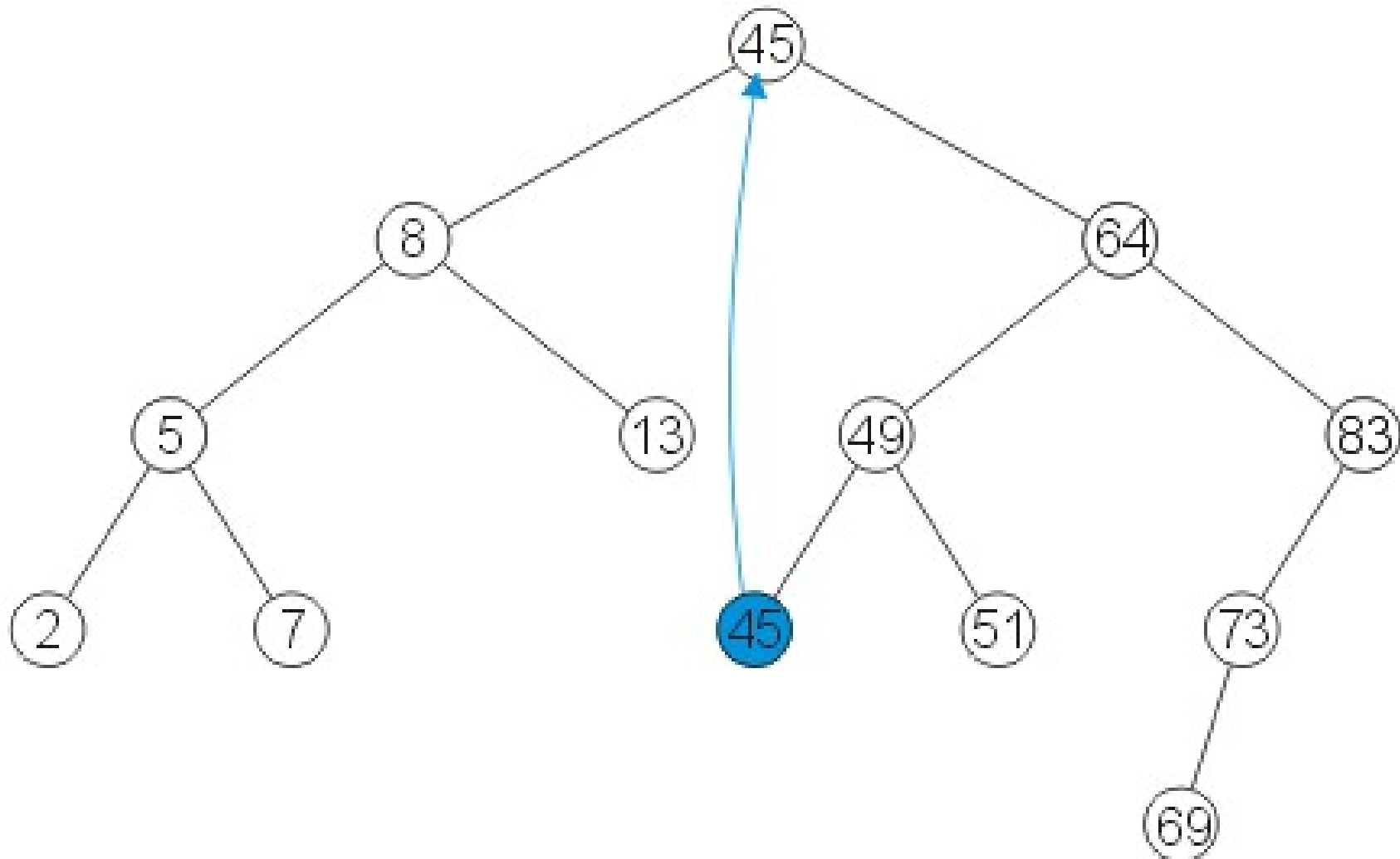
76

- Ví dụ xóa 42



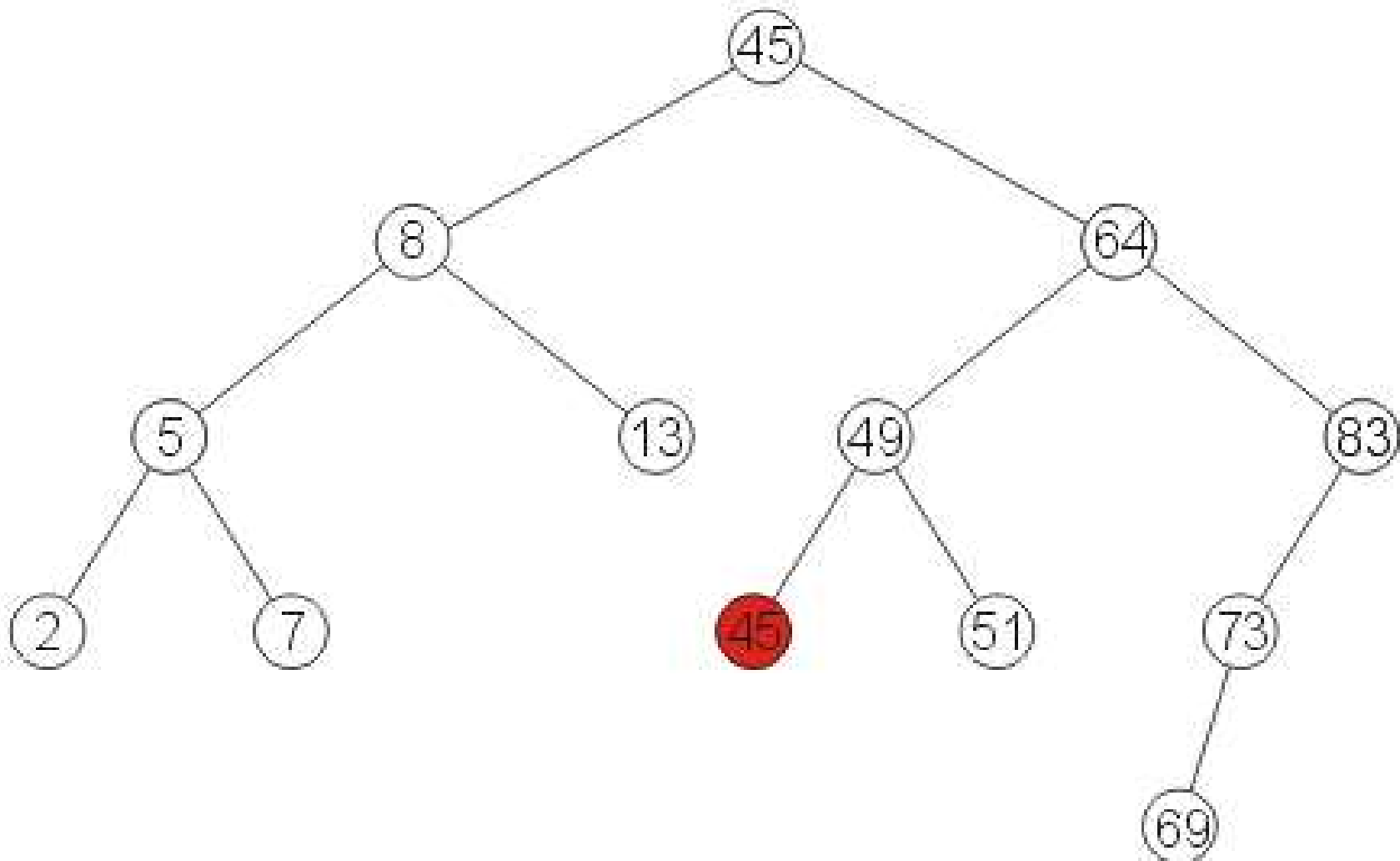
# Binary Search Tree – Hủy một phần tử có khóa X

77



# Binary Search Tree – Hủy một phần tử có khóa X

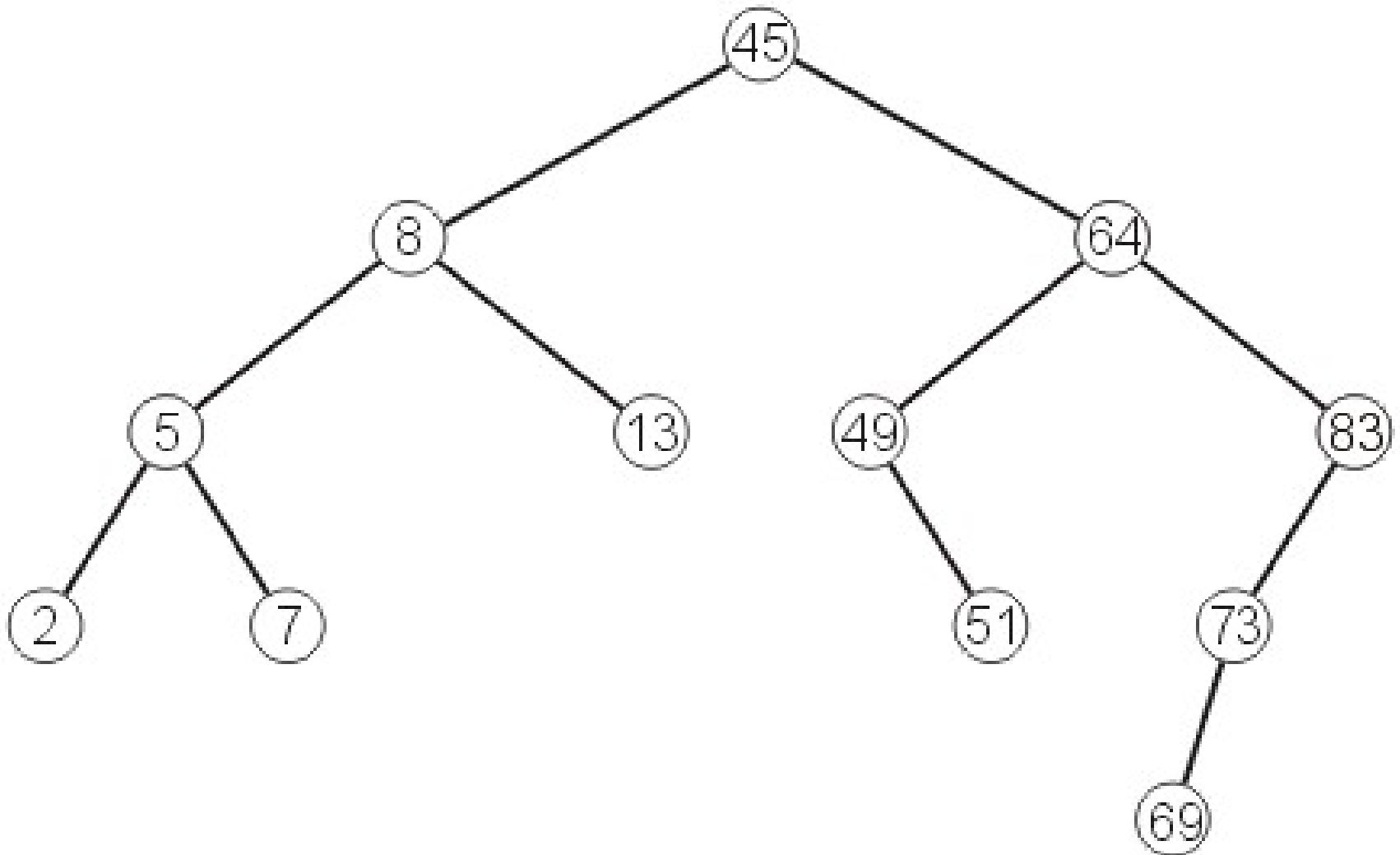
78



# Binary Search Tree – Hủy một phần tử có khóa X

79

- Xóa 15, sau đó 42:



# Binary Search Tree – Hủy toàn bộ cây

80

- Việc hủy toàn bộ cây có thể được thực hiện thông qua thao tác duyệt cây theo thứ tự sau. Nghĩa là ta sẽ hủy cây con trái, cây con phải rồi mới hủy nút gốc

```
void removeTree(Tree &T)
{
    if(T)
    {
        removeTree(T->pLeft);
        removeTree(T->pRight);
        delete(T);
    }
}
```



# Binary Search Tree

81

## □ Nhận xét:

- Tất cả các thao tác **searchNode**, **insertNode**, **delNode** đều có độ phức tạp trung bình  $O(h)$ , với  $h$  là chiều cao của cây
- Trong trường hợp tốt nhất, CNPTK có  $n$  nút 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
- Trong trường hợp xấu nhất, cây có thể bị suy biến thành 1 danh sách liên kết (khi mà mỗi nút đều chỉ có 1 con trừ nút lá). Lúc đó các thao tác trên sẽ có độ phức tạp  **$O(n)$**
- Vì vậy cần có cải tiến cấu trúc của CNPTK để đạt được chi phí cho các thao tác là  **$\log_2(n)$**

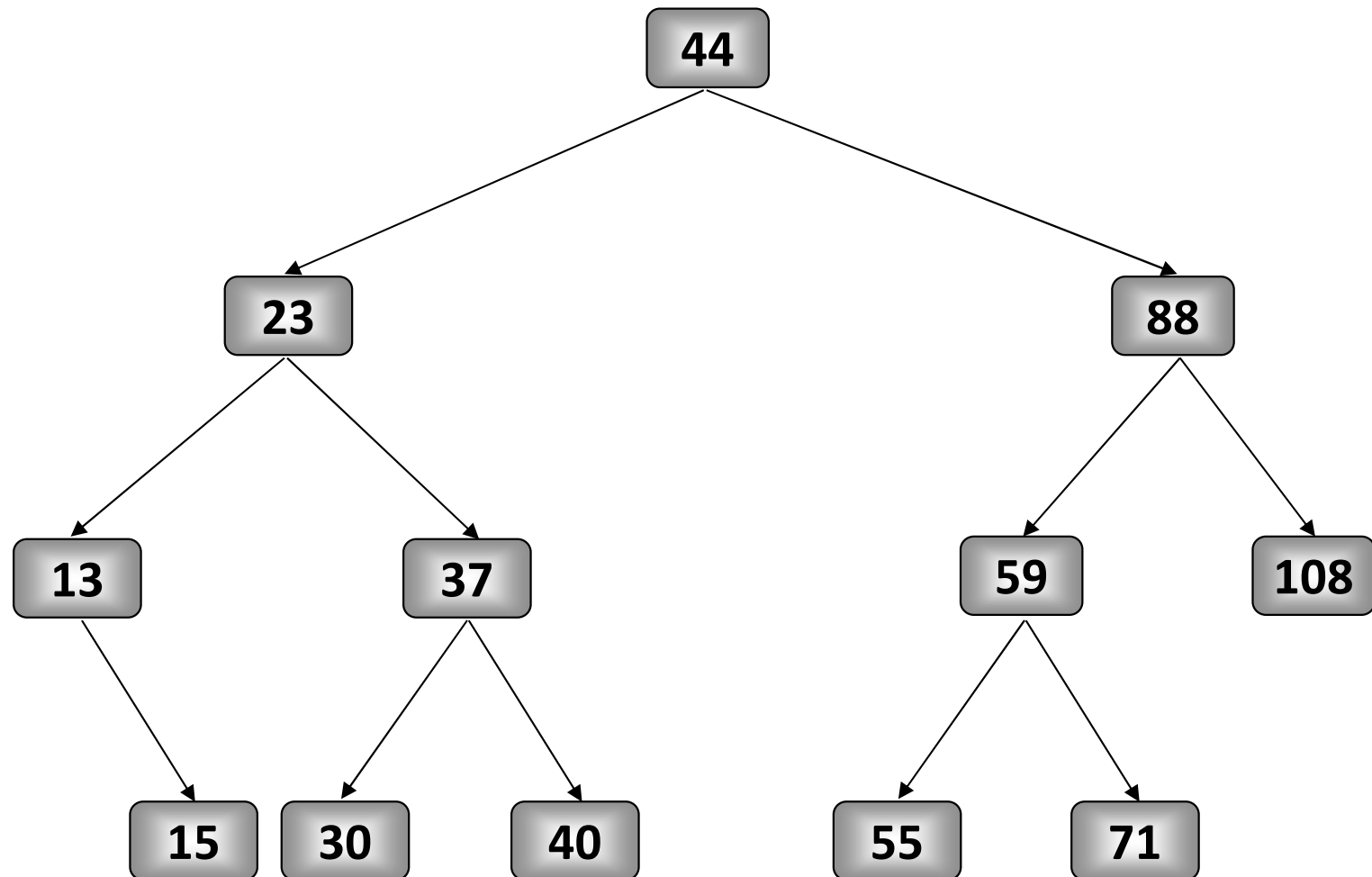
# AVL Tree - Định nghĩa

82

- Cây nhị phân tìm kiếm cân bằng là cây mà tại mỗi nút của nó độ cao của cây con trái và của cây con phải chênh lệch không quá một.

# AVL Tree – Ví dụ

83



# AVL Tree

84

- Lịch sử cây cân bằng (AVL Tree):
  - ▣ AVL là tên viết tắt của các tác giả người Nga đã đưa ra định nghĩa của cây cân bằng **A**delson-**V**elskii và **L**andis (1962)
  - ▣ Từ cây AVL, người ta đã phát triển thêm nhiều loại CTDL hữu dụng khác như cây đỏ-đen (Red-Black Tree), B-Tree, ...
  
- Cây AVL có chiều cao  **$\log_2(n)$**

# AVL Tree

85

- Chỉ số cân bằng của một nút:
  - ▣ Định nghĩa: Chỉ số cân bằng của một nút p. Viết tắt:  $CSCB(p)$   
$$CSCB(p) = \text{Độ cao cây phải (p)} - \text{Độ cao cây trái (p)}$$
  - ▣ Đối với một cây cân bằng, chỉ số cân bằng của mỗi nút chỉ có thể mang một trong ba giá trị sau đây:
    - $CSCB(p) = 0 \Leftrightarrow \text{Độ cao cây trái (p)} = \text{Độ cao cây phải (p)}$
    - $CSCB(p) = 1 \Leftrightarrow \text{Độ cao cây trái (p)} < \text{Độ cao cây phải (p)}$
    - $CSCB(p) = -1 \Leftrightarrow \text{Độ cao cây trái (p)} > \text{Độ cao cây phải (p)}$
- Để tiện trong trình bày, chúng ta sẽ ký hiệu như sau:  
$$p \rightarrow \text{balFactor} = CSCB(p);$$
- Độ cao cây trái (p) ký hiệu là **hL**
- Độ cao cây phải(p) ký hiệu là **hR**

# AVL Tree – Biểu diễn

86

```
#define LH    -1    /* Cây con trái cao hơn */
#define EH     0    /* Hai cây con bằng nhau */
#define RH     1    /* Cây con phải cao hơn */
```

```
struct AVLNode{
    char          balFactor; // Chỉ số cân bằng
    DataType      data;
    AVLNode*      pLeft;
    AVLNode*      pRight;
};
typedef AVLNode*  AVLTree;
```

# AVL Tree – Biểu diễn

87

- Trường hợp thêm hay hủy một phần tử trên cây có thể làm cây tăng hay giảm chiều cao, khi đó phải cân bằng lại cây
- Việc cân bằng lại một cây sẽ phải thực hiện sao cho chỉ ảnh hưởng tối thiểu đến cây nhằm giảm thiểu chi phí cân bằng
- Các thao tác đặc trưng của cây AVL:
  - ▣ Thêm một phần tử vào cây AVL
  - ▣ Hủy một phần tử trên cây AVL
  - ▣ Cân bằng lại một cây vừa bị mất cân bằng

# AVL Tree

88

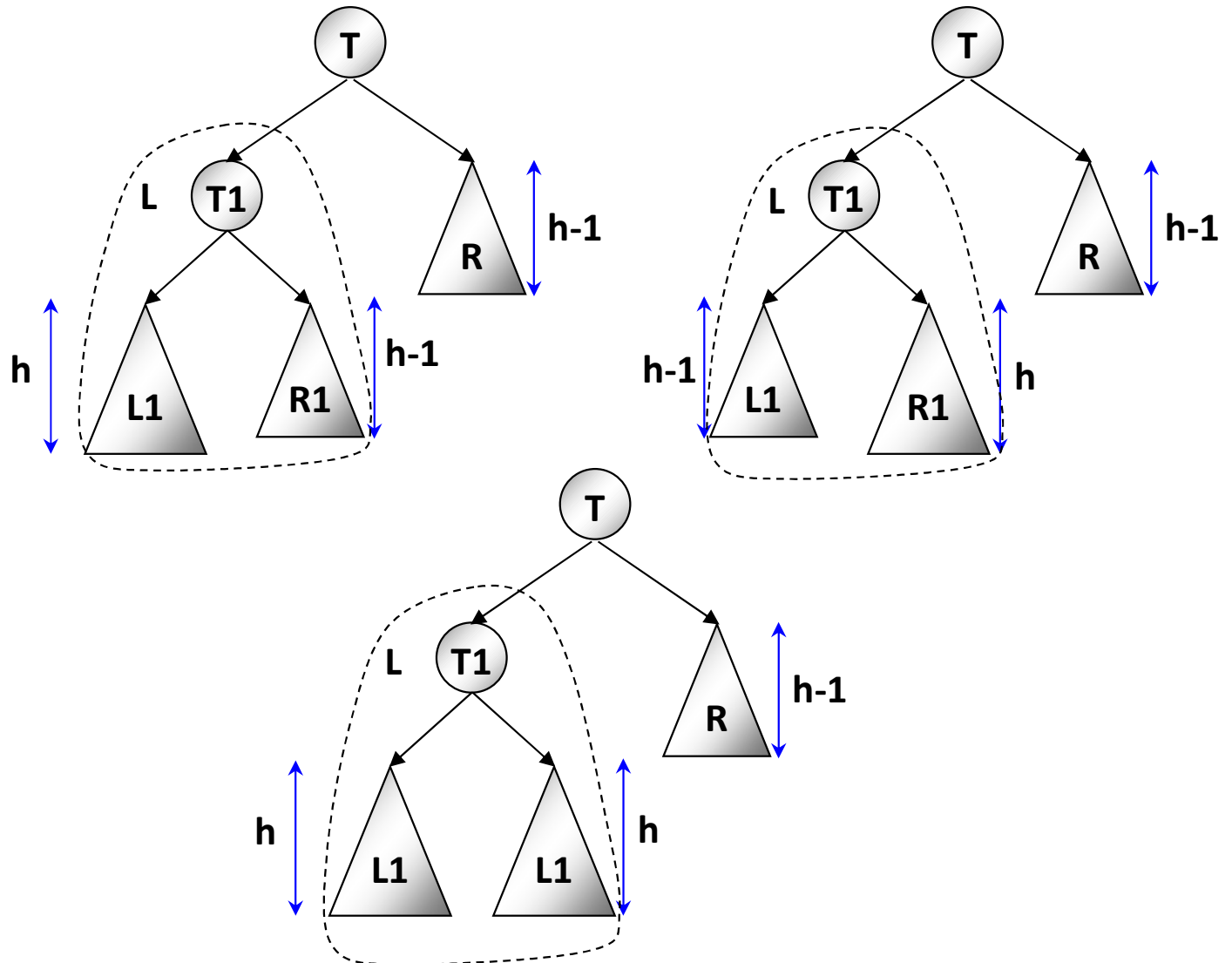
- Các trường hợp mất cân bằng:
  - ▣ Ta sẽ không khảo sát tính cân bằng của 1 cây nhị phân bất kỳ mà chỉ quan tâm đến các khả năng mất cân bằng xảy ra khi thêm hoặc hủy một nút trên cây AVL
  - ▣ Như vậy, khi mất cân bằng, độ lệch chiều cao giữa 2 cây con sẽ là 2
  - ▣ Có 6 khả năng sau:
    - Trường hợp 1 - Cây T lệch về bên trái : 3 khả năng
    - Trường hợp 2 - Cây T lệch về bên phải: 3 khả năng



# AVL Tree

89

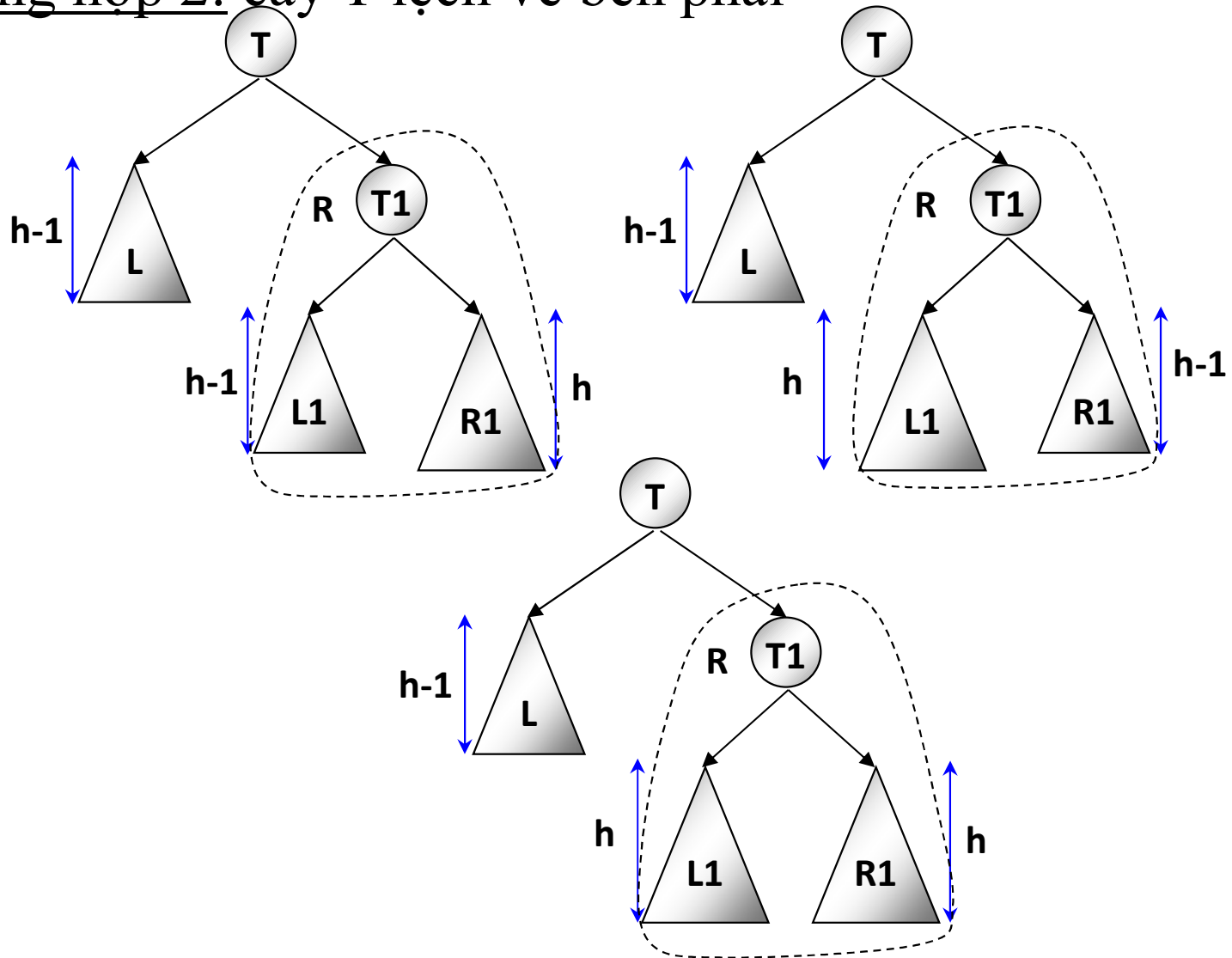
- Trường hợp 1: cây T lệch về bên trái



# AVL Tree

90

- Trường hợp 2: cây T lệch về bên phải



# AVL Tree

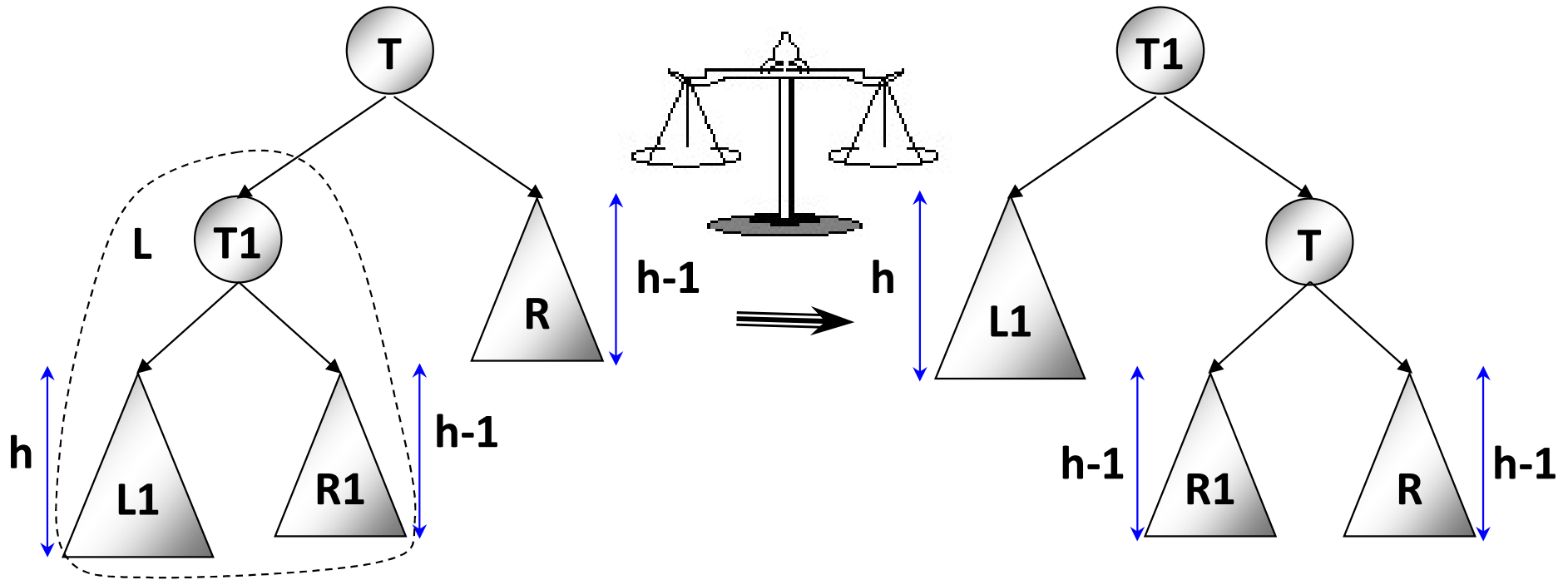
91

- Các trường hợp mất cân bằng:
  - ▣ Các trường hợp lệch về bên phải hoàn toàn đối xứng với các trường hợp lệch về bên trái.
  - ▣ Vì vậy, chỉ cần khảo sát trường hợp lệch về bên trái.
  - ▣ Trong 3 trường hợp lệch về bên trái, trường hợp T1 lệch phải là phức tạp nhất. Các trường hợp còn lại giải quyết rất đơn giản.

# AVL Tree - Cân bằng lại cây AVL

92

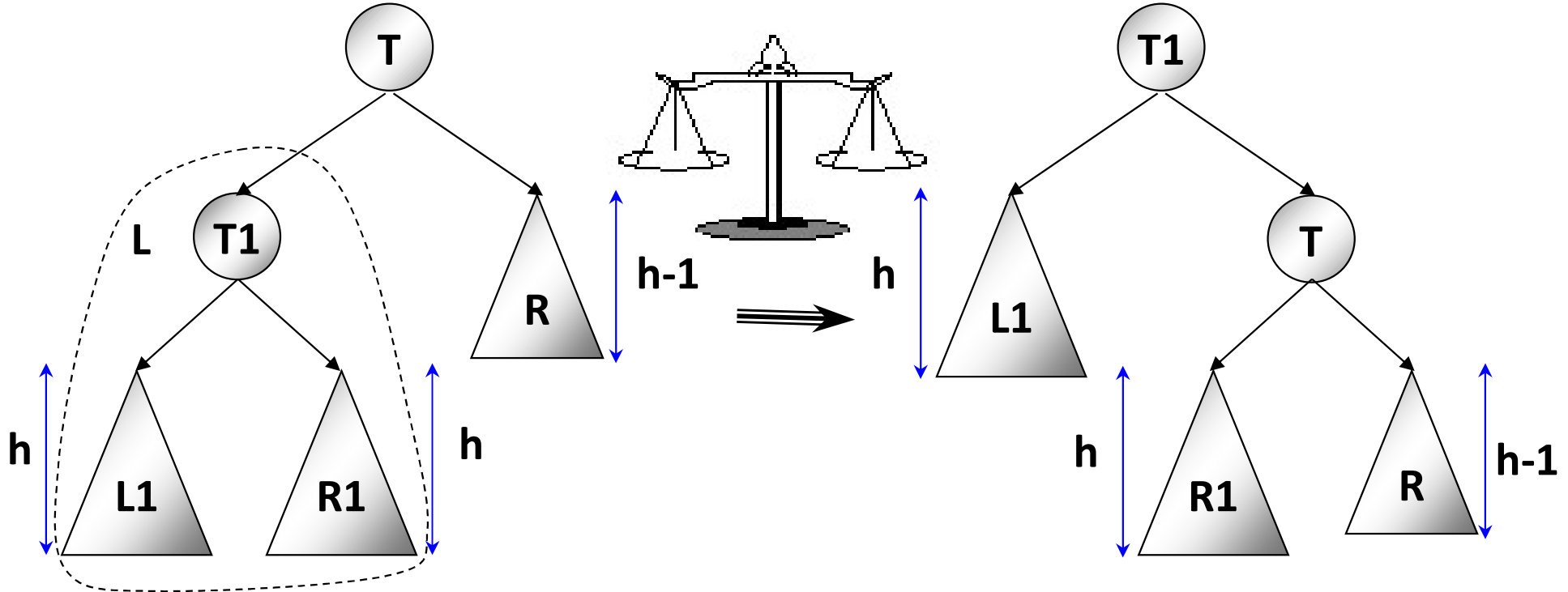
- T/h 1.1: cây T1 lệch Left-Left



# AVL Tree - Cân bằng lại cây AVL

93

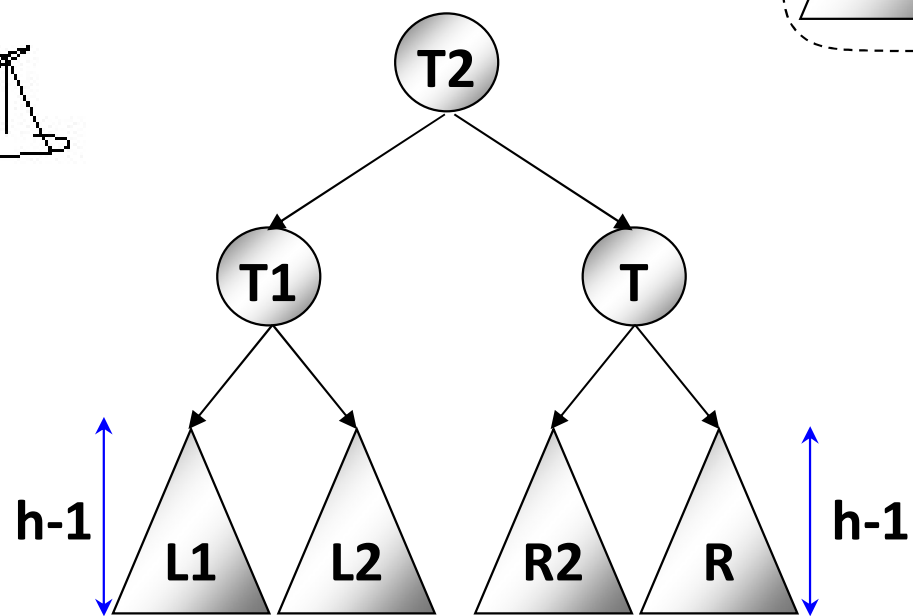
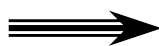
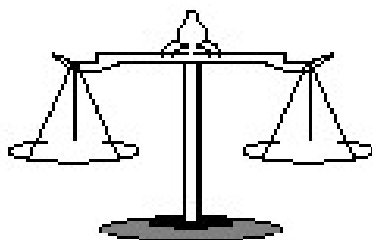
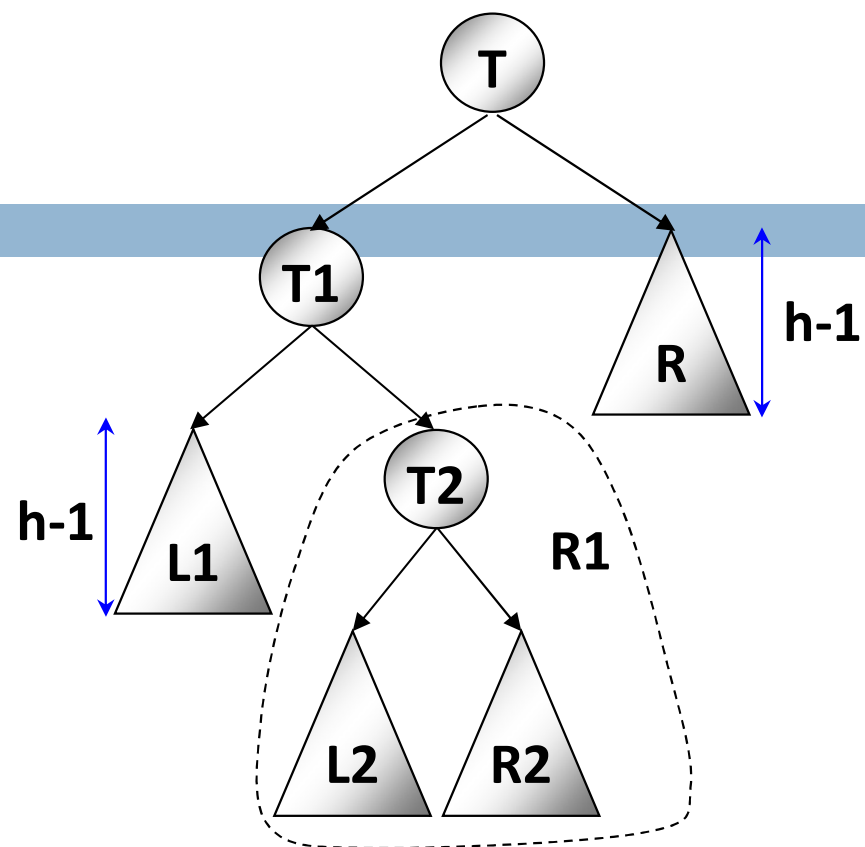
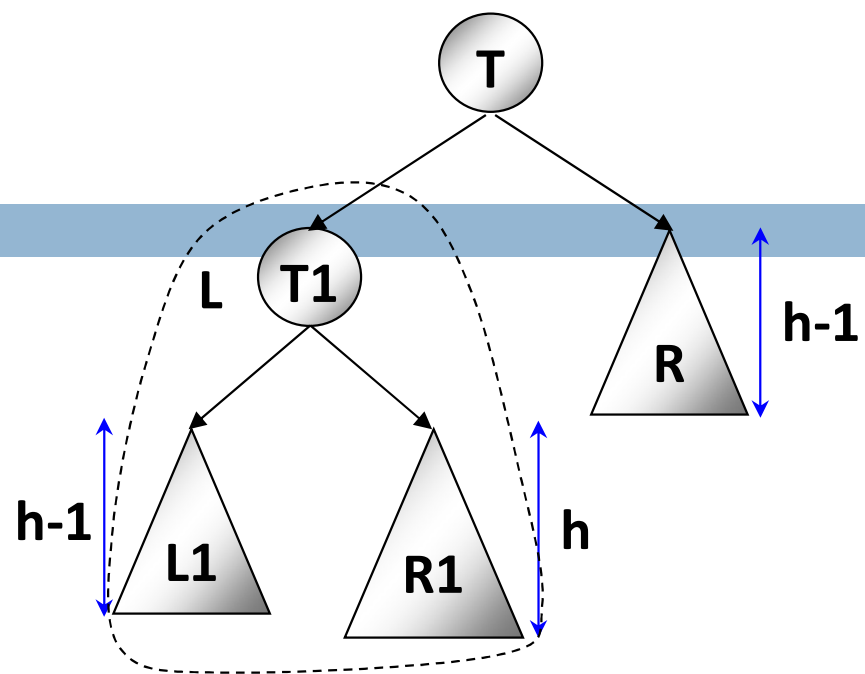
- T/h 1.2: cây T1 không lệch. Ta thực hiện phép quay đơn Left-Left



# AVL Tree - Cân bằng lại cây AVL

94

- T/h 1.3: cây T1 lệch về bên phải. Ta thực hiện phép quay kép Left-Right
- Do T1 lệch về bên phải ta không thể áp dụng phép quay đơn đã áp dụng trong 2 trường hợp trên vì khi đó cây T sẽ chuyển từ trạng thái mất cân bằng do lệch trái thành mất cân bằng do lệch phải ? cần áp dụng cách khác



# AVL Tree - Cân bằng lại cây AVL

96

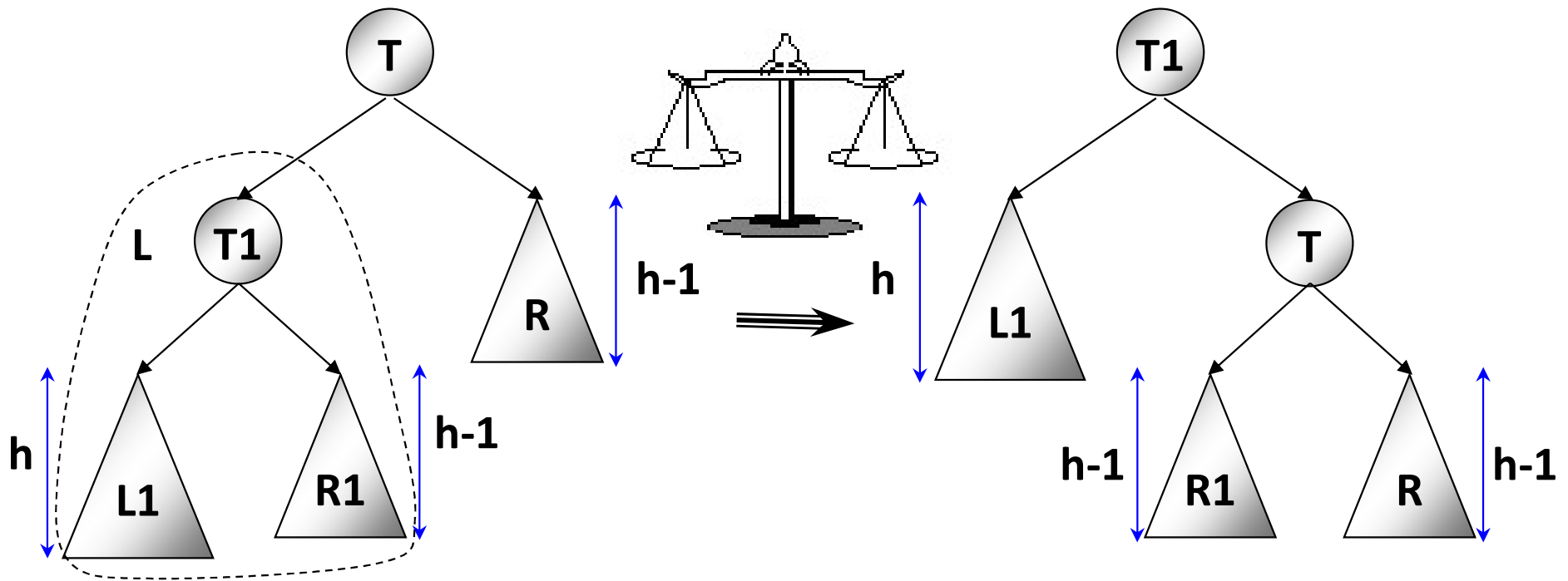
- Lưu ý:
  - ▣ Trước khi cân bằng cây T có chiều cao  $h+2$  trong cả 3 trường hợp 1.1, 1.2 và 1.3
  - ▣ Sau khi cân bằng:
    - Trường hợp 1.1 và 1.3 cây có chiều cao  $h+1$
    - Trường hợp 1.2 cây vẫn có chiều cao  $h+2$ . Đây là trường hợp duy nhất sau khi cân bằng nút T cũ có chỉ số cân bằng  $\neq 0$
    - Thao tác cân bằng lại trong tất cả các trường hợp đều có độ phức tạp  $O(1)$



# AVL Tree - Cân bằng lại cây AVL

97

- T/h 1.1: cây T1 lệch về bên trái. Ta thực hiện phép quay đơn Left-Left



# AVL Tree - Cân bằng lại cây AVL

98

```
void rotateLL(AVLTree &T) //quay đơn Left-Left
{
    AVLNode* T1 = T->pLeft;
    T->pLeft      = T1->pRight;
    T1->pRight    = T;
    switch(T1->balFactor) {
        case LH: T->balFactor = EH;
                 T1->balFactor = EH;
                 break;
        case EH: T->balFactor = LH;
                 T1->balFactor = RH;
                 break;
    }
    T = T1;
}
```

# AVL Tree - Cân bằng lại cây AVL

99

## □ Quay đơn Right-Right:

```
void rotateRR (AVLTree &T) //quay đơn Right-Right
{
    AVLNode*    T1 = T->pRight;
    T->pRight    = T1->pLeft;
    T1->pLeft    = T;
    switch(T1->balFactor) {
        case RH: T->balFactor = EH;
                  T1->balFactor= EH;
                  break;
        case EH: T->balFactor = RH;
                  T1->balFactor= LH;
                  break;
    }
    T = T1;
}
```

# AVL Tree - Cân bằng lại cây AVL

100

## □ Quay kép Left-Right:

```
void rotateLR(AVLTree &T)//quay kép Left-Right
{
    AVLNode* T1 = T->pLeft;
    AVLNode* T2 = T1->pRight;
    T->pLeft = T2->pRight;
    T2->pRight = T;
    T1->pRight = T2->pLeft;
    T2->pLeft = T1;
    switch(T2->balFactor) {
        case LH: T->balFactor = RH; T1->balFactor = EH; break;
        case EH: T->balFactor = EH; T1->balFactor = EH; break;
        case RH: T->balFactor = EH; T1->balFactor = LH; break;
    }
    T2->balFactor = EH;
    T = T2;
}
```

# AVL Tree - Cân bằng lại cây AVL

101

## □ Quay kép Right-Left

```
void rotateRL(AVLTree &T)    //quay kép Right-Left
{
    AVLNode*    T1 = T->pRight;
    AVLNode*    T2 = T1->pLeft;
    T->pRight    = T2->pLeft;
    T2->pLeft    = T;
    T1->pLeft    = T2->pRight;
    T2->pRight    = T1;
    switch(T2->balFactor) {
        case RH: T->balFactor = LH; T1->balFactor = EH; break;
        case EH: T->balFactor = EH; T1->balFactor = EH; break;
        case LH: T->balFactor = EH; T1->balFactor = RH; break;
    }
    T2->balFactor = EH;
    T = T2;
}
```

# AVL Tree - Cân bằng lại cây AVL

102

- Cân bằng khi cây bị lệch về bên trái:

```
int balanceLeft(AVLTree &T)
//Cân bằng khi cây bị lệch về bên trái
{
    AVLNode*    T1 = T->pLeft;

    switch(T1->balFactor)
    {
        case LH:    rotateLL(T); return 2;
        case EH:    rotateLL(T); return 1;
        case RH:    rotateLR(T); return 2;
    }
    return 0;
}
```

# AVL Tree - Cân bằng lại cây AVL

103

- Cân bằng khi cây bị lệch về bên phải

```
int balanceRight(AVLTree &T )  
//Cân bằng khi cây bị lệch về bên phải  
{  
    AVLNode*      T1 = T->pRight;  
  
    switch(T1->balFactor)      {  
    case LH:      rotateRL(T); return 2;  
    case EH:      rotateRR(T); return 1;  
    case RH:      rotateRR(T); return 2;  
    }  
    return 0;  
}
```

# AVL Tree - Thêm một phần tử trên cây AVL

104

- Việc thêm một phần tử vào cây AVL diễn ra tương tự như trên CNPTK
- Sau khi thêm xong, nếu chiều cao của cây thay đổi, từ vị trí thêm vào, ta phải lần ngược lên gốc để kiểm tra xem có nút nào bị mất cân bằng không. Nếu có, ta phải cân bằng lại ở nút này
- Việc cân bằng lại chỉ cần thực hiện 1 lần tại nơi mất cân bằng
- Hàm *insertNode* trả về giá trị  $-1$ ,  $0$ ,  $1$  khi không đủ bộ nhớ, gặp nút cũ hay thành công. Nếu sau khi thêm, chiều cao cây bị tăng, giá trị  $2$  sẽ được trả về

**int *insertNode*(AVLTree &T, DataType X)**

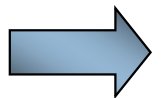


# AVL Tree - Thêm một phần tử trên cây AVL

105

```
int insertNode(AVLTree &T, DataType X)
{ int res;
  if (T)
  {   if (T->key == X) return 0; //đã có
      if (T->key > X)
      {   res      = insertNode(T->pLeft, X);
          if(res < 2) return res;
          switch(T->balFactor)
          {   case RH: T->balFactor = EH; return 1;
              case EH: T->balFactor = LH; return 2;
              case LH: balanceLeft(T); return 1;
          }
      }
  }
```

```
.....
}
```

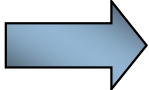


insertNode2

# AVL Tree - Thêm một phần tử trên cây AVL

106

```
int insertNode (AVLTree &T, DataType X)
{
    .....
    else // T->key < X
    {
        res      = insertNode (T-> pRight, X) ;
        if(res < 2) return res;
        switch (T->balFactor)
        {
            case LH: T->balFactor      = EH; return 1;
            case EH: T->balFactor      = RH; return 2;
            case RH: balanceRight (T) ; return 1;
        }
    }
    .....
}
```



insertNode3

# AVL Tree - Thêm một phần tử trên cây AVL

107

```
int insertNode (AVLTree &T, DataType X)
{
    .....
    .....
    T = new TNode;
    if (T == NULL) return -1; //thiếu bộ nhớ
    T->key = X;
    T->balFactor = EH;
    T->pLeft = T->pRight = NULL;
    return 2; // thành công, chiều cao tăng
}
```

# AVL Tree - Hủy một phần tử trên cây AVL

108

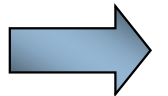
- Cũng giống như thao tác thêm một nút, việc hủy một phần tử X ra khỏi cây AVL thực hiện giống như trên CNPTK
- Sau khi hủy, nếu tính cân bằng của cây bị vi phạm ta sẽ thực hiện việc cân bằng lại
- Tuy nhiên việc cân bằng lại trong thao tác hủy sẽ phức tạp hơn nhiều do có thể xảy ra phản ứng dây chuyền
- Hàm *delNode* trả về giá trị 1, 0 khi hủy thành công hoặc không có X trong cây. Nếu sau khi hủy, chiều cao cây bị giảm, giá trị 2 sẽ được trả về:

**int delNode(AVLTree &T, DataType X)**

# AVL Tree - Hủy một phần tử trên cây AVL

109

```
int delNode(AVLTree &T, DataType X)
{ int res;
  if (T==NULL)          return 0;
  if (T->key > X)
  { res = delNode (T->pLeft, X);
    if (res < 2)      return res;
    switch (T->balFactor)
    { case LH: T->balFactor = EH; return 2;
      case EH: T->balFactor = RH; return 1;
      case RH: return balanceRight (T);
    }
  } // if (T->key > X)
  .....
}
```

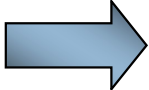


delNode2

# AVL Tree - Hủy một phần tử trên cây AVL

110

```
int delNode(AVLTree &T, DataType X)
{
    .....
    if (T->key < X)
    {
        res = delNode (T->pRight, X);
        if (res < 2) return res;
        switch (T->balFactor)
        {
            case RH: T->balFactor = EH; return 2;
            case EH: T->balFactor = LH; return 1;
            case LH: return balanceLeft (T);
        }
    } // if (T->key < X)
    .....
}
```



delNode3

# AVL Tree - Hủy một phần tử trên cây AVL

111

```
int delNode(AVLTree &T, DataType X)
{.....
    else //T->key == X
    { AVLNode*    p = T;
      if(T->pLeft == NULL)          { T = T->pRight; res = 2; }
      else if(T->pRight == NULL) { T = T->pLeft;   res = 2; }
      else //T có đủ cả 2 con
      { res = searchStandFor(p,T->pRight);
        if(res < 2) return res;
        switch(T->balFactor)
        { case RH: T->balFactor = EH; return 2;
          case EH: T->balFactor = LH; return 1;
          case LH: return balanceLeft(T);
          }
        }
      delete p; return res;
    }
}
```

# AVL Tree - Hủy một phần tử trên cây AVL

112

```
int searchStandFor(AVLTree &p, AVLTree &q)
//Tìm phần tử thể mạng
{ int res;
  if(q->pLeft)
  { res = searchStandFor(p, q->pLeft);
    if(res < 2) return res;
    switch(q->balFactor)
    { case LH: q->balFactor = EH; return 2;
      case EH: q->balFactor = RH; return 1;
      case RH: return balanceRight(T);
    }
  } else
  { p->key = q->key; p = q; q = q->pRight; return 2;
  }
}
```



# AVL Tree

113

- Nhận xét:
  - ▣ Thao tác thêm một nút có độ phức tạp  $O(1)$
  - ▣ Thao tác hủy một nút có độ phức tạp  $O(h)$
  - ▣ Với cây cân bằng trung bình 2 lần thêm vào cây thì cần một lần cân bằng lại; 5 lần hủy thì cần một lần cân bằng lại

# AVL Tree

114

- Nhận xét:
  - ▣ Việc hủy 1 nút có thể phải cân bằng dây chuyền các nút từ gốc cho đến phần tử bị hủy trong khi thêm vào chỉ cần 1 lần cân bằng cục bộ
  - ▣ Độ dài đường tìm kiếm trung bình trong cây cân bằng gần bằng cây cân bằng hoàn toàn  $\log_2 n$ , nhưng việc cân bằng lại đơn giản hơn nhiều
  - ▣ Một cây cân bằng không bao giờ cao hơn 45% cây cân bằng hoàn toàn tương ứng dù số nút trên cây là bao nhiêu