

Chương 3: 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

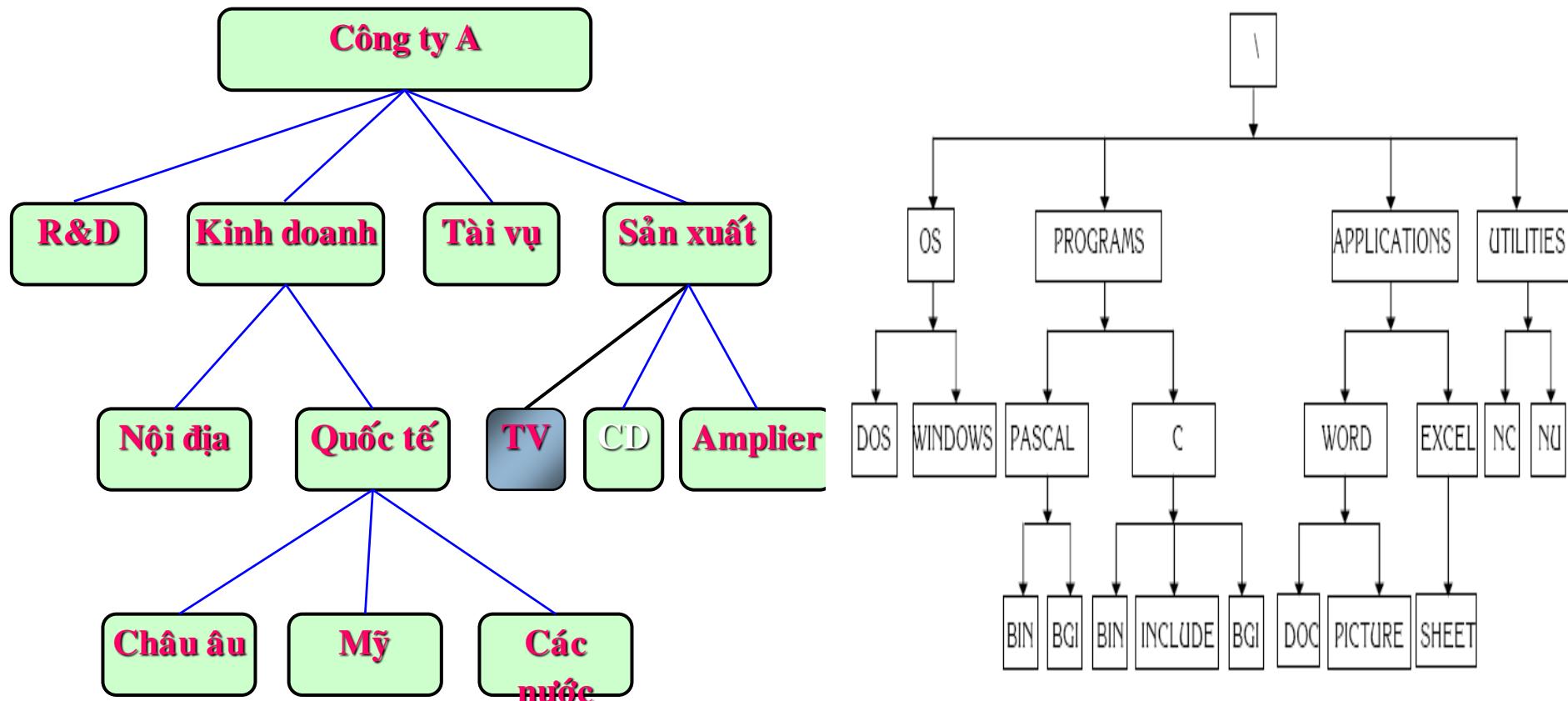
3

- Cây là một tập gồm 1 hay nhiều **nút** T, trong đó có một 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

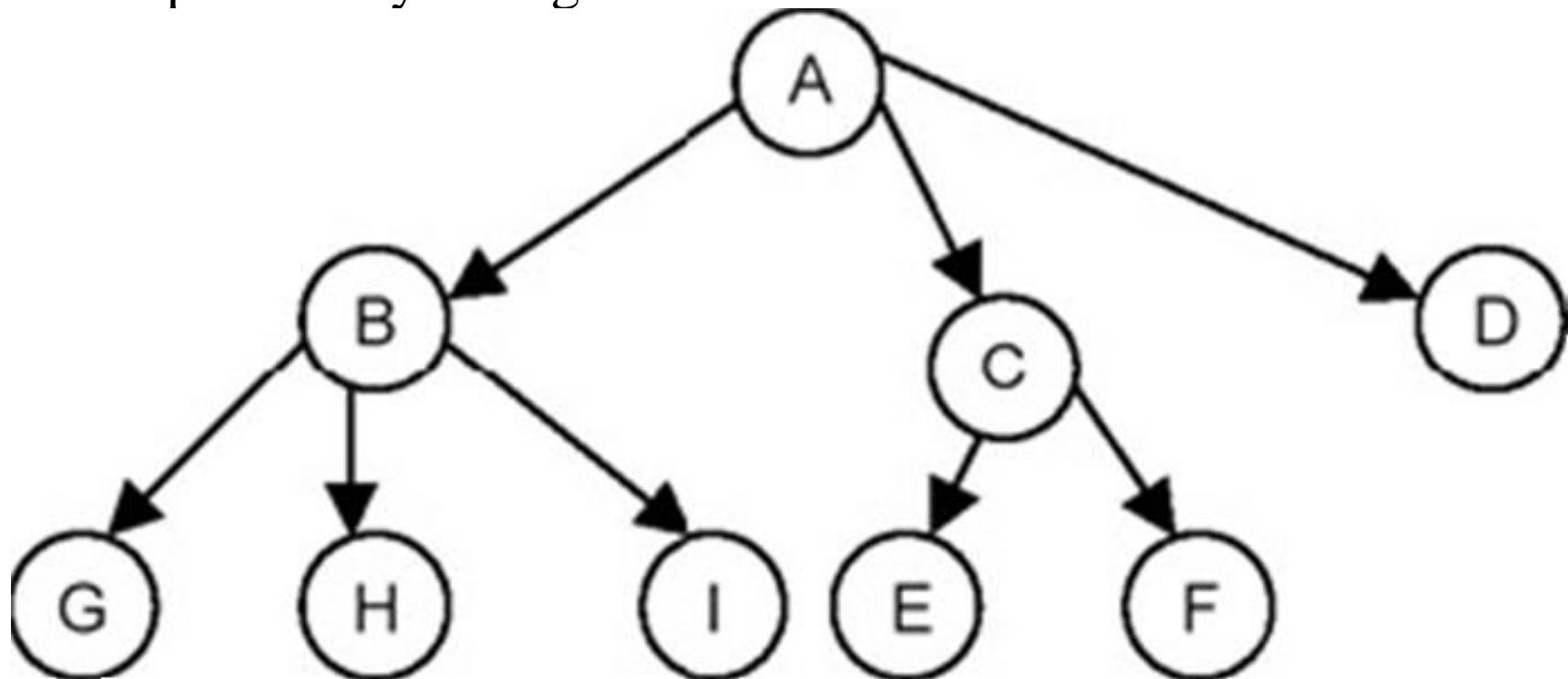
4

- ☐ Sơ đồ tổ chức của một công ty



Tree – Ví dụ

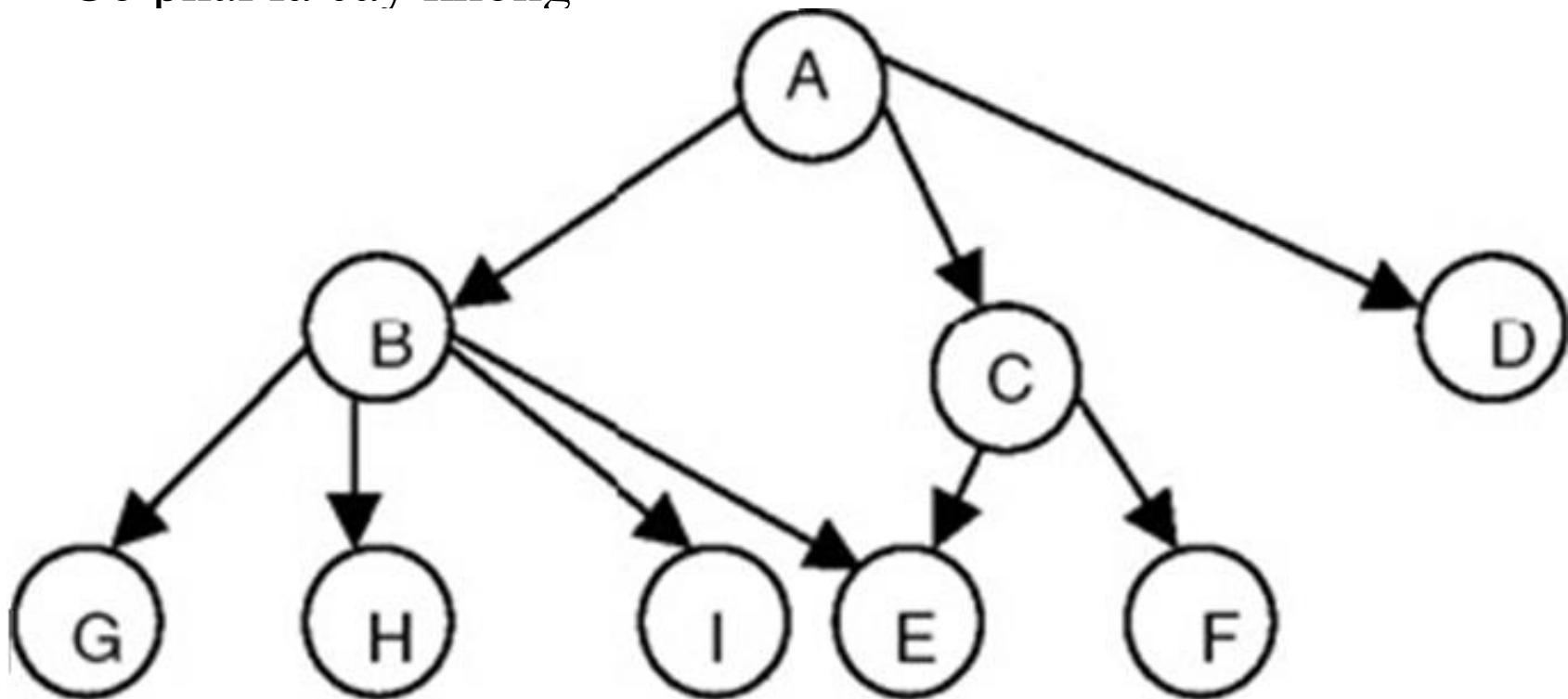
- ☐ Có phải là cây không



Tree – Ví dụ

6

- ☐ Có phải là cây không

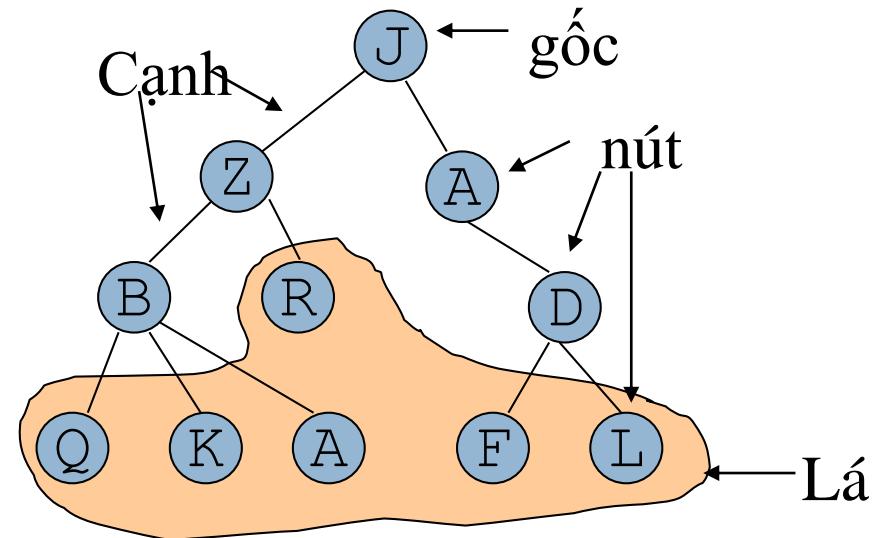


Không vì trong cấu trúc cây không tồn tại chu trình

Tree - Một số khái niệm cơ bản

7

- 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

8

- Nút nhánh:

- Là nút có **bậc khác 0** và **không phải là gốc**

- Mức của một nút (**Level of a Node**):

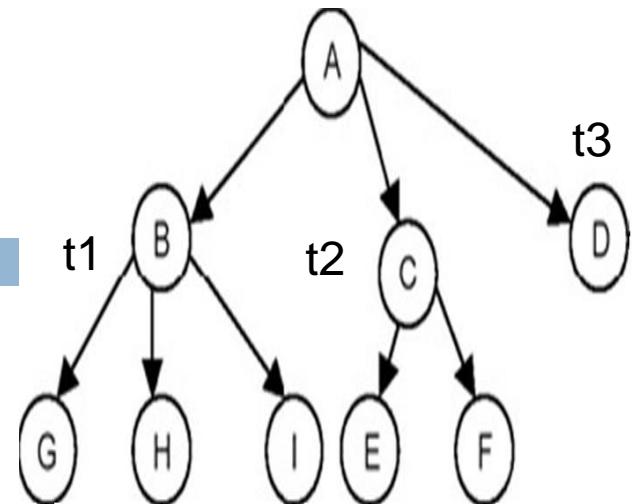
- Mức (T_0) = 0, với T_0 là gốc
 - Gọi $T_1, T_2, T_3, \dots, T_n$ là các cây con của T_0 :

$$\text{Mức}(T_1) = \text{Mức}(T_2) = \dots = \text{Mức}(T_n) = \text{Mức}(T_0) + 1$$

We define the level of the node by taking the level of the root node as 0, and incrementing it by 1 as we move from the root towards the subtrees.

- Chiều cao của cây (độ sâu) (**Height – Depth of a Tree**):

- Là mức cao nhất của nút + 1 (mức cao nhất của 1 nút có trong cây)



Tree – Ví dụ

9

Nút gốc (root node)

Owner

Manager

Chef

Waitress

Waiter

Cook

Helper

nút lá (leaf node)

A Tree Has Levels

10

LEVEL 0

Owner
Jake

LEVEL 1

Manager
Brad

Chef
Carol

LEVEL 2

Waitress
Joyce

Waiter
Chris

Cook
Max

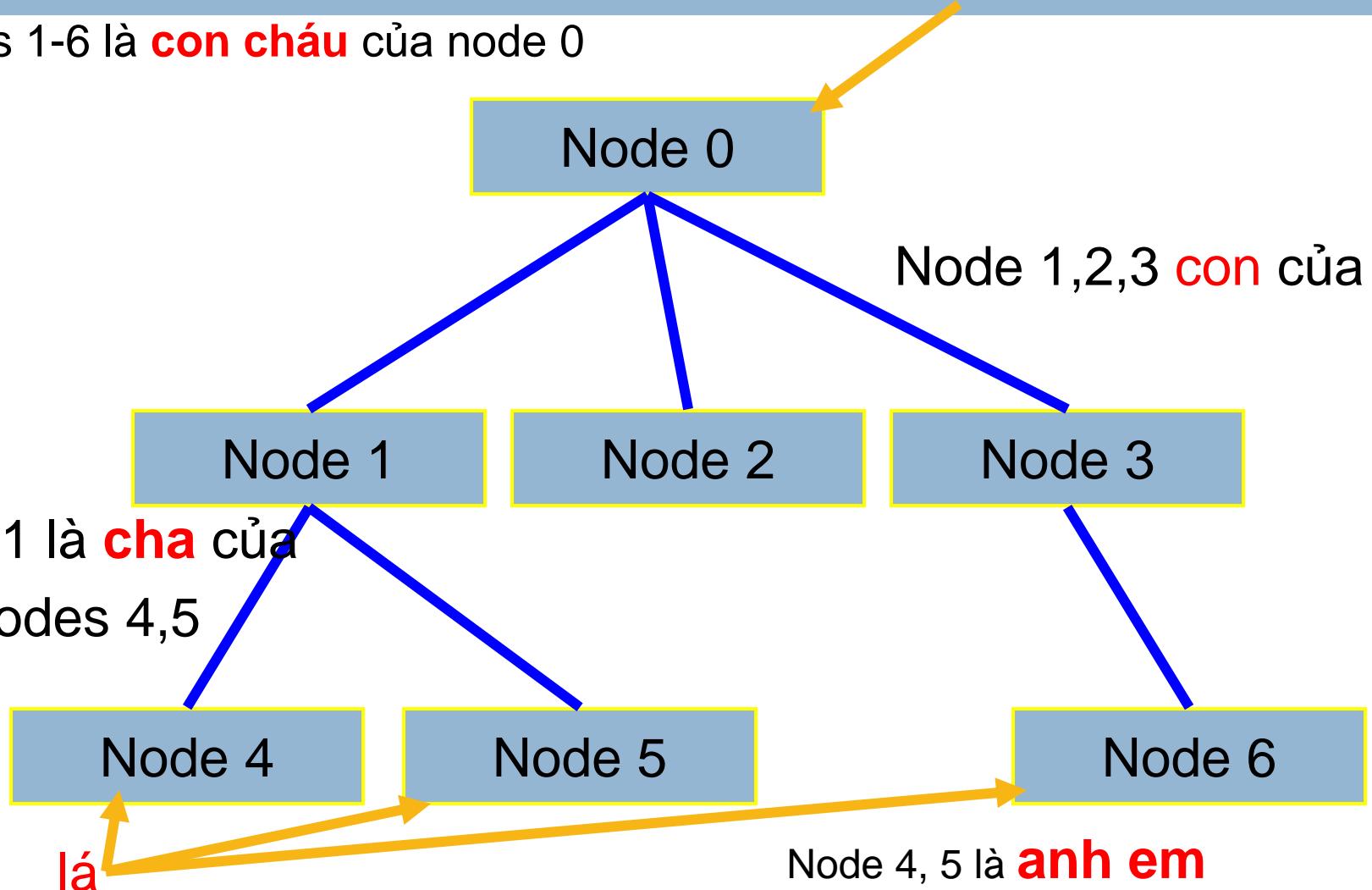
Helper
Len

Định nghĩa

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

Nodes 1-6 là **con cháu** của node 0

Gốc



Một số khái niệm cơ bản

12

- **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

- Rừng (forest) là tập hợp các cây.
- Khi cây mất gốc → rừng

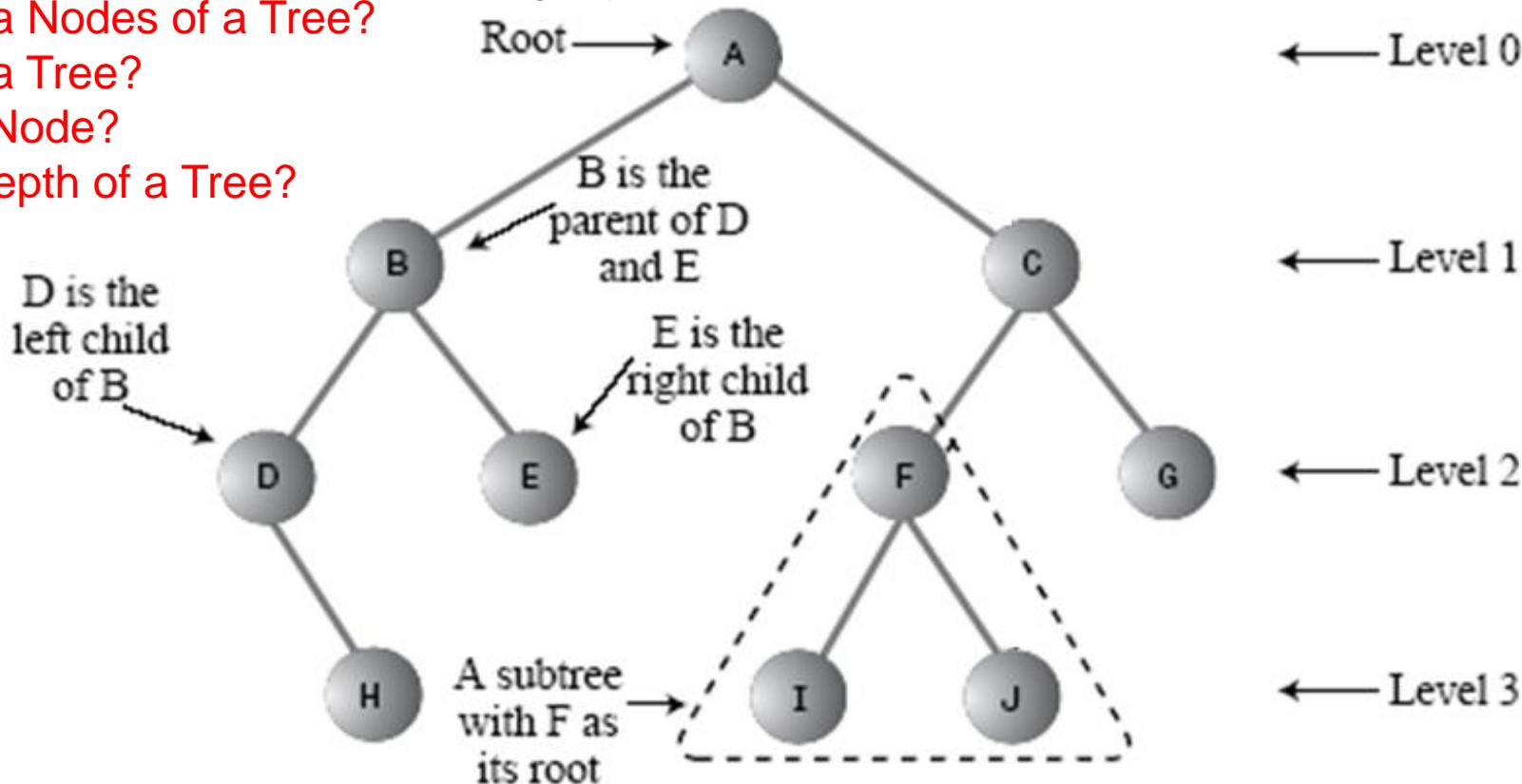
Tree – Ví dụ

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

Chiều cao của cây (độ sâu) (Height – Depth of a Tree): Là mức cao nhất của nút + 1 (mức cao nhất của 1 nút có trong cây)

- Leaf node?
- Degree of a Nodes of a Tree?
- Degree of a Tree?
- Level of a Node?
- Height – Depth of a Tree?



Trắc nghiệm

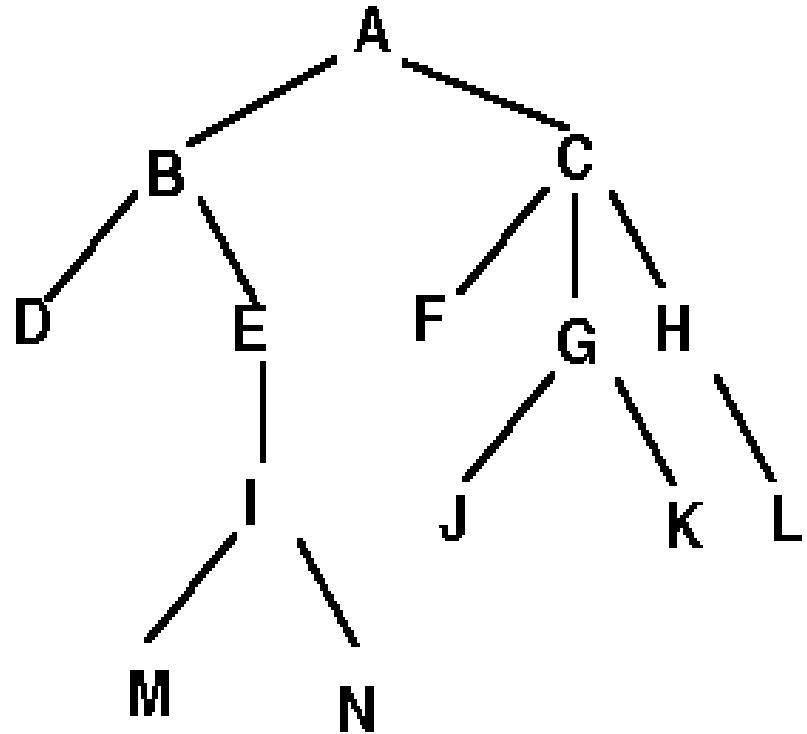
14

- The depth of a tree is the _____ of a tree
 - a) number of nodes on the tree
 - b) number of levels of a tree
 - c) number of branches
 - d) level
- Give the binary tree with root A. The root has left child B and right child C. B has left child D and right child E. There are no other nodes in the tree. The height of the tree is _____.
 - a) 0
 - b) 3
 - c) 1
 - d) 2

Bài tập

15

- Leaf node?
- Numbers of Leaf node?
- Degree of a Nodes of a Tree
- Degree of a Tree?
- Level of a Node?
- Height – Depth of a Tree?



Depth-first Search

16

1.3. Biểu diễn cây

- Dùng đồ thị, Dùng giản đồ tập hợp, Sử dụng dạng phân cấp chỉ số
BIỂU DIỄN CÂY TRONG BỘ NHỚ MÁY TÍNH
- Để biểu diễn cây trong bộ nhớ máy tính dùng danh sách liên kết.
- Để biểu diễn cây N-phân dùng danh sách có N mồi liên kết để quản lý N địa chỉ nút con.
- Cấu trúc dữ liệu của cây N-phân tương tự cấu trúc dữ liệu đa liên kết.

```
const int N = 100;  
typedef struct NTNode  
{ T Key;  
    NTNode * SubNode[N];  
}NTOneNode;  
typedef struct NTOneNode * NTTType;
```

Để quản lý cây, chỉ cần quản lý địa chỉ nút gốc **NTType** NTree;

Nội dung

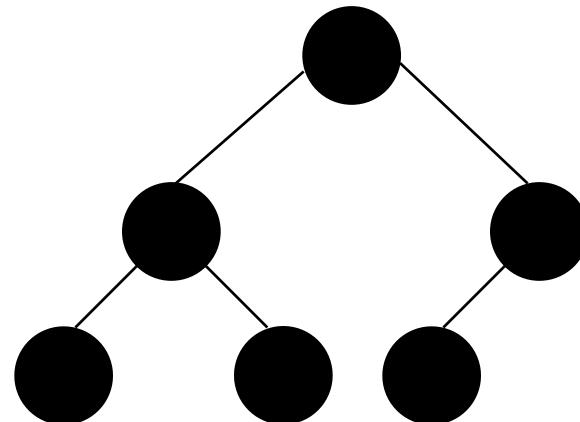
25

- 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*)

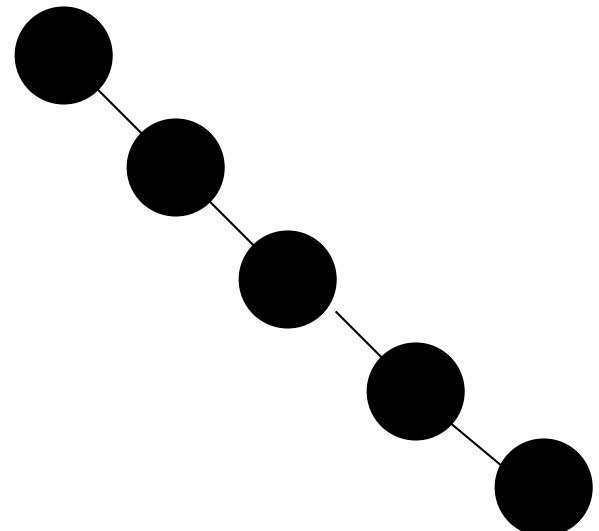
Cây nhị phân (Binary Tree) – Định nghĩa

26

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

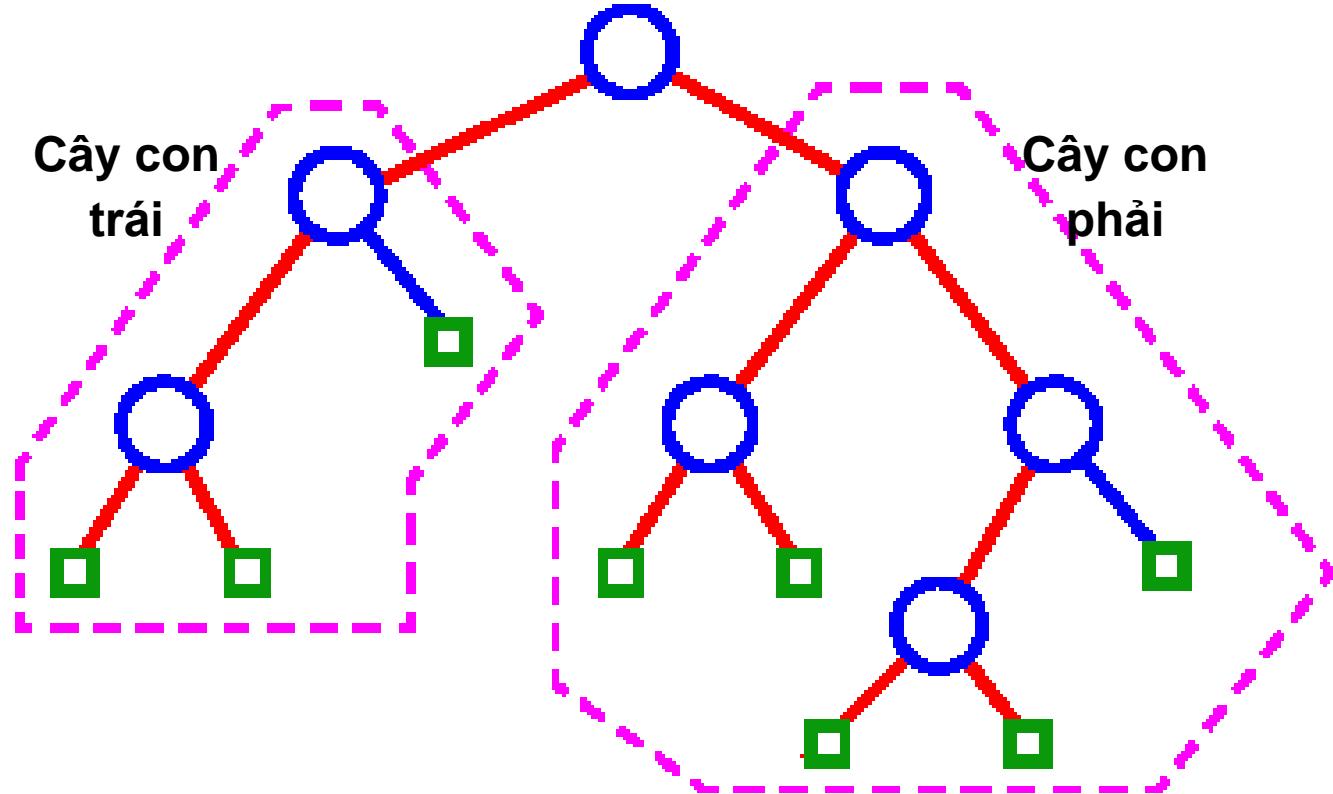


Link List



Cây nhị phân (Binary Tree) – Định nghĩa

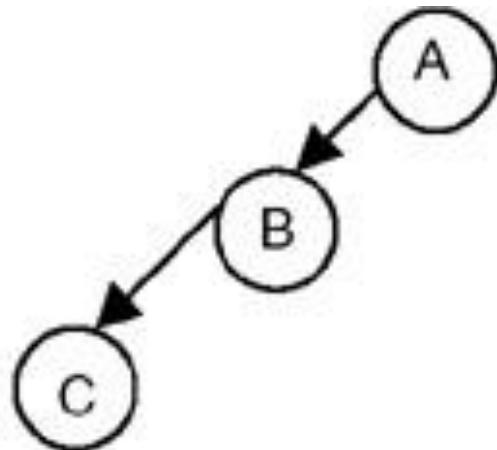
27



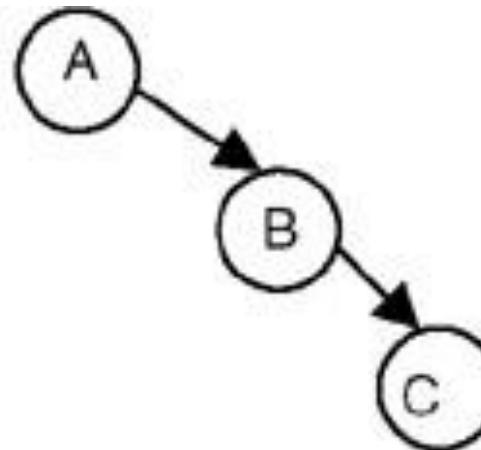
Hình ảnh một cây nhị phân

Cây nhị phân (Binary Tree) – Định nghĩa

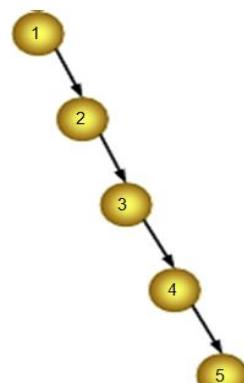
- Cây lệch trái và cây lệch phải, cây zíc-zắc



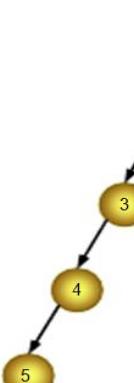
Left Skewed



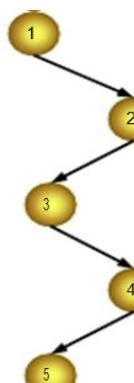
Right Skewed



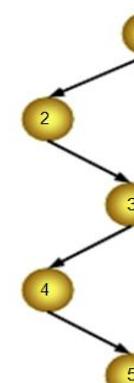
a)



b)



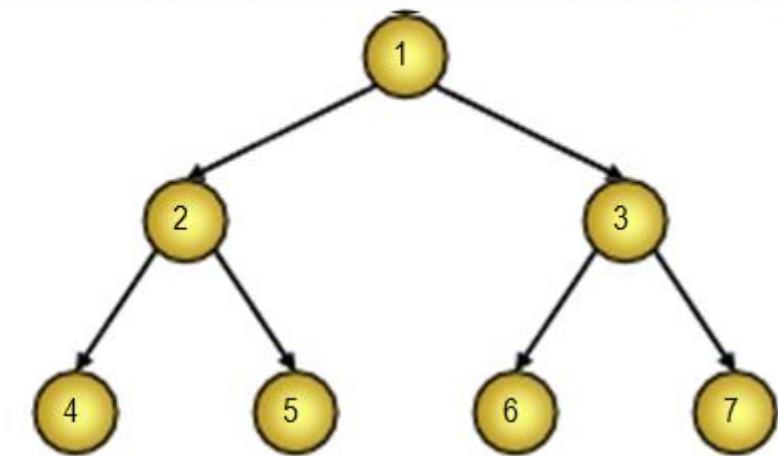
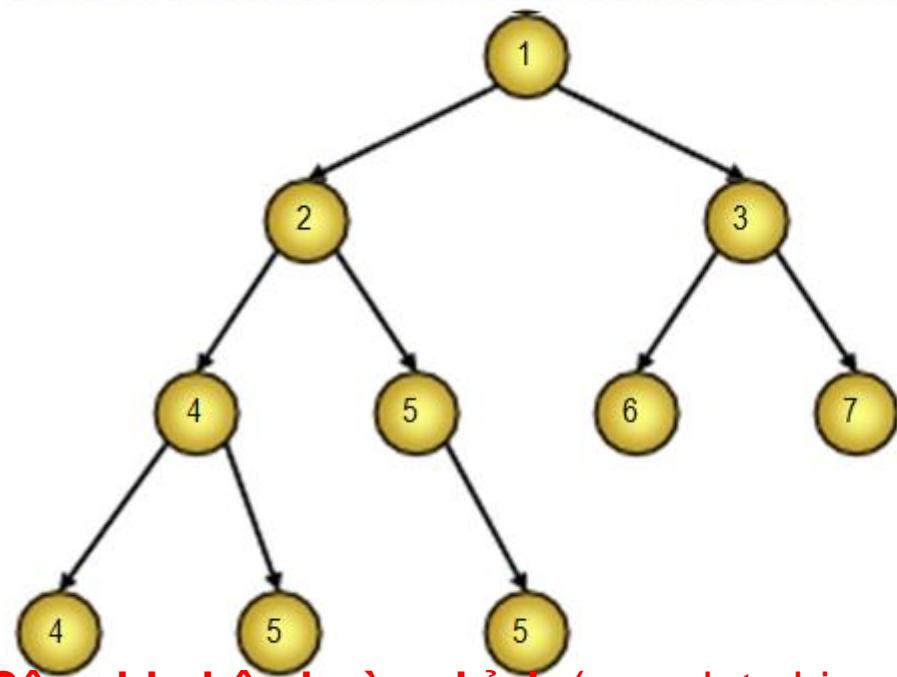
c)



d)

Cây nhị phân (Binary Tree) – Định nghĩa

- **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



Cây nhị phân đầy đủ (A full binary tree)

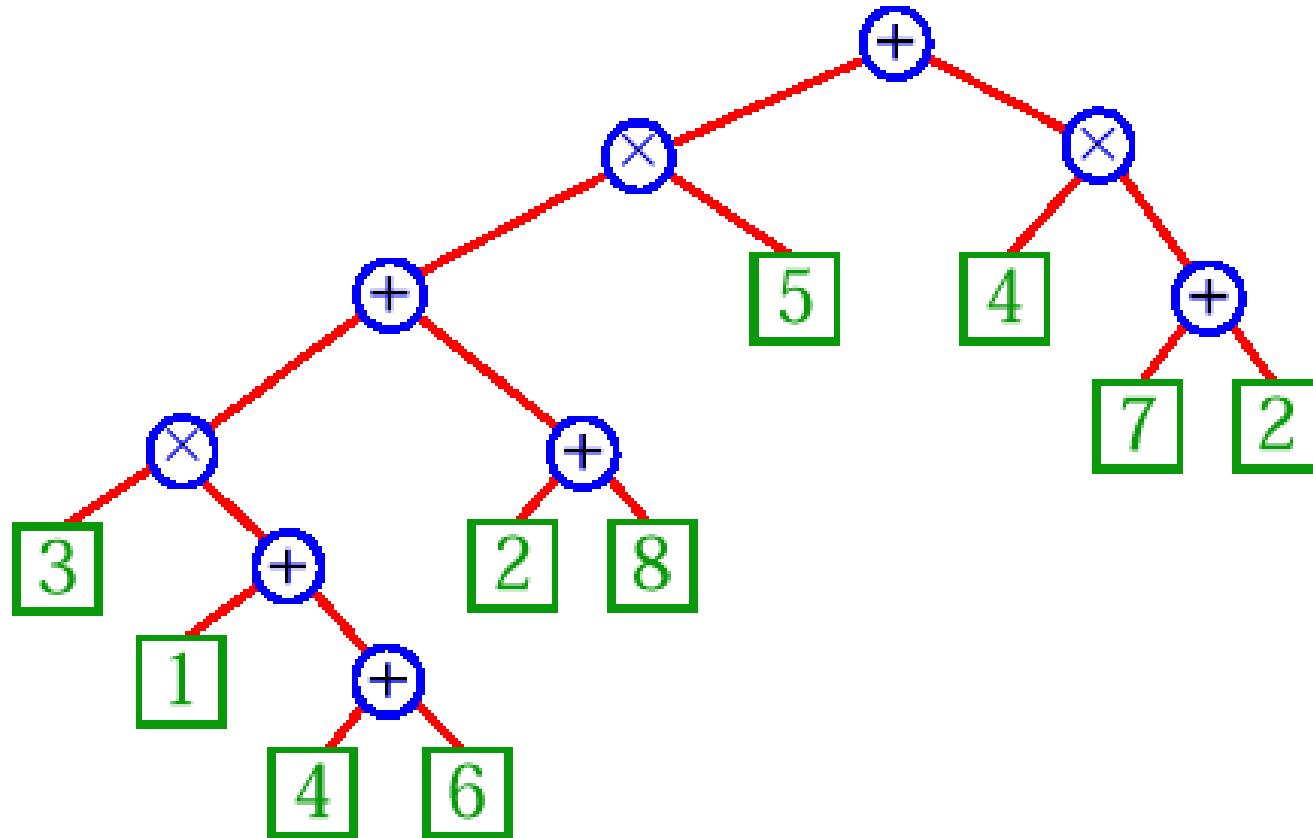
Cây nhị phân hoàn chỉnh (complete binary tree)

f)

Cây nhị phân (Binary Tree) – Ứng dụng

30

- Cây biểu thức: được dùng để biểu diễn một biểu thức toán học

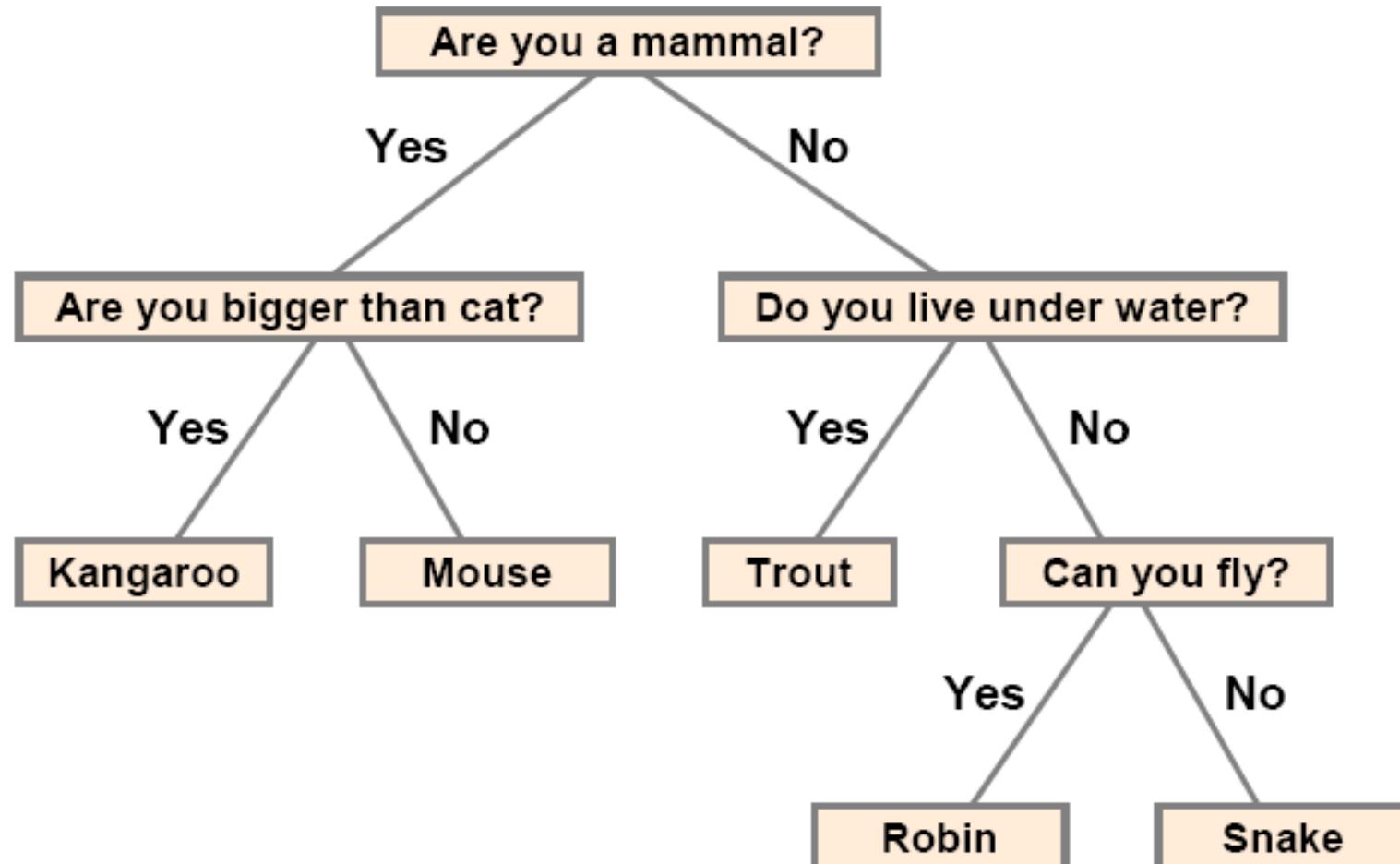


$$((((3 \times (1 + (4 + 6)))) + (2 + 8)) \times 5) + (4 \times (7 + 2)))$$

Binary Tree – Ví dụ

31

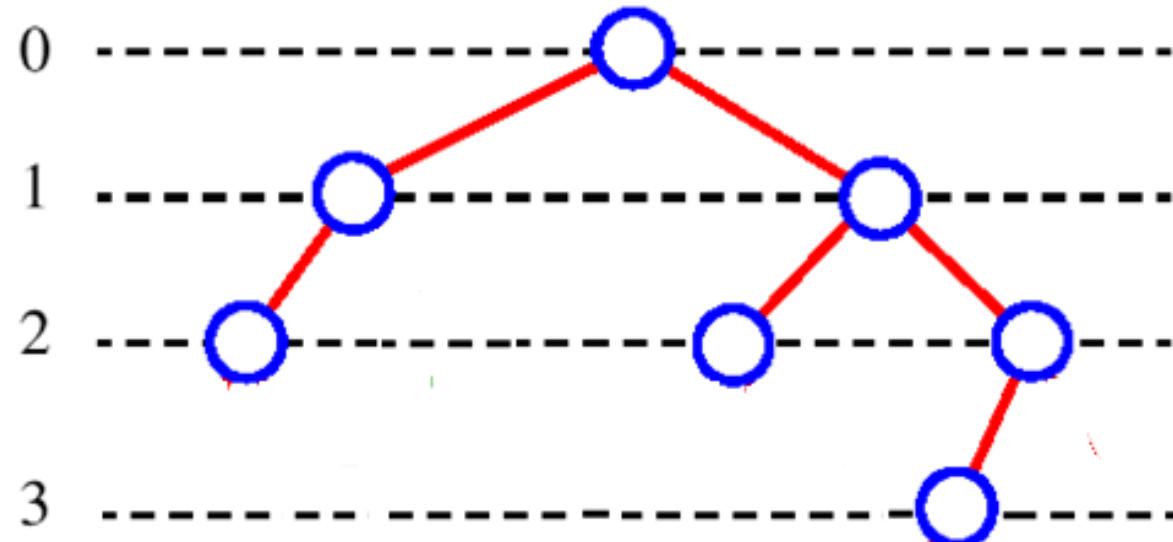
- Cây quyết định: được dùng để hỗ trợ quá trình ra quyết định



Binary Tree – Một số tính chất

32

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



Trắc nghiệm

33

- A binary tree is a tree in which each node references at most _____ node(s)

1

0

3

2



- The maximum number of leaf-nodes in a binary tree of height 4 is:

2

4

6

8



- What is the minimum height of a binary tree with 31 nodes?

4



7

5

3

- If the depth of a binary tree is 3, then what is the maximum size of the tree?

3

4

6

8



Cây nhị phân tương đương (Binary tree)

- 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

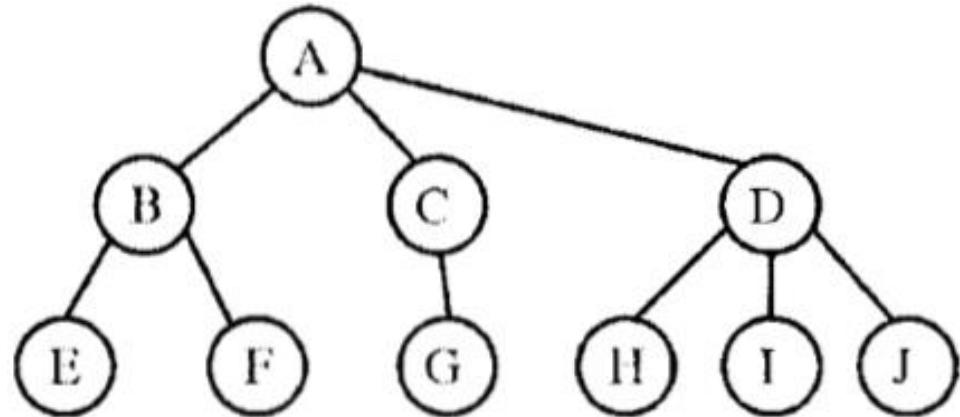
Data

Địa chỉ em kế cận phải

Cây nhị phân tương đương (Binary tree)-

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	

Binary Tree - Biểu diễn bằng mảng

37

- 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 :

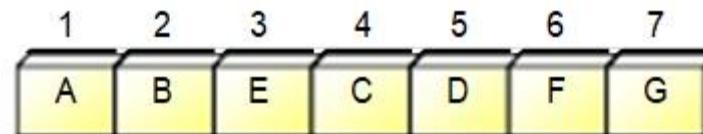
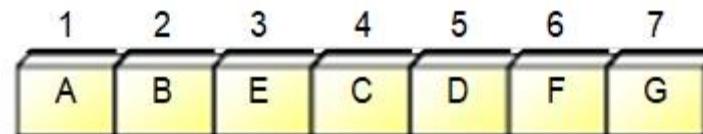
- Con trái: $2i$
 - Con phải: $2i + 1$

- Nút thứ j :

- Nút cha : $j \text{ div } 2$

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

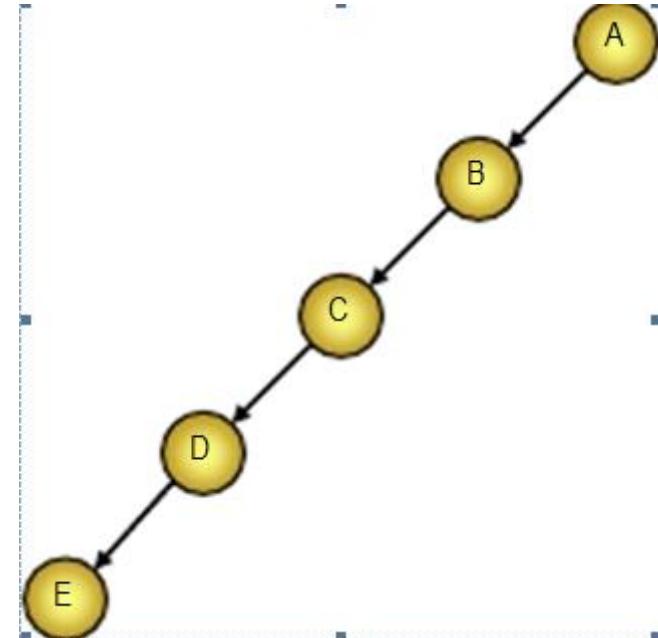
- Nút thứ i của cây được lưu trữ bằng phần tử $T[i]$



Binary Tree - Biểu diễn bằng mảng

38

- 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

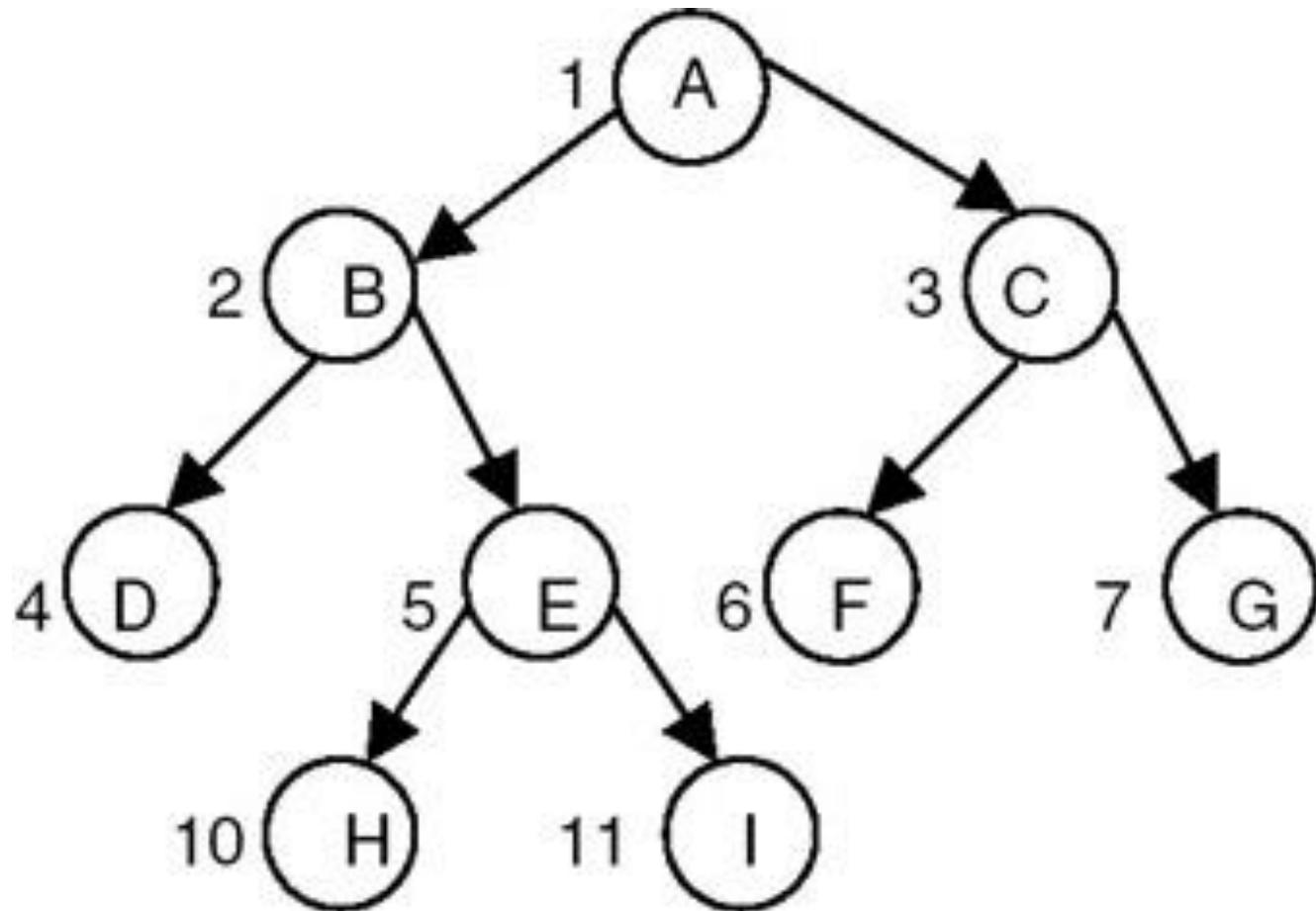


1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17



Binary Tree - Biểu diễn

39



1	A
2	B
3	C
4	D
5	E
6	F
7	G
8	
9	
10	H
11	I

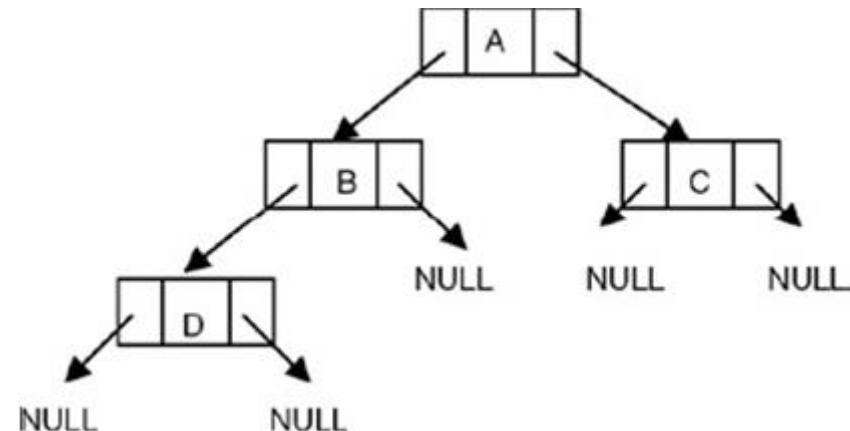
Array tree

Binary Tree - Biểu diễn

40

- Sử dụng cấu trúc để lưu trữ các thông tin của một nút gồm:
 - Dữ liệu của nút
 - Địa chỉ nút gốc của cây con trái
 - Địa chỉ nút gốc của cây con phải
- Khai báo cấu trúc cây nhị phân:

```
struct TNode
{
    DataType data;
    TNode *left, *right;
};
typedef TNode* Tree; //??
```



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

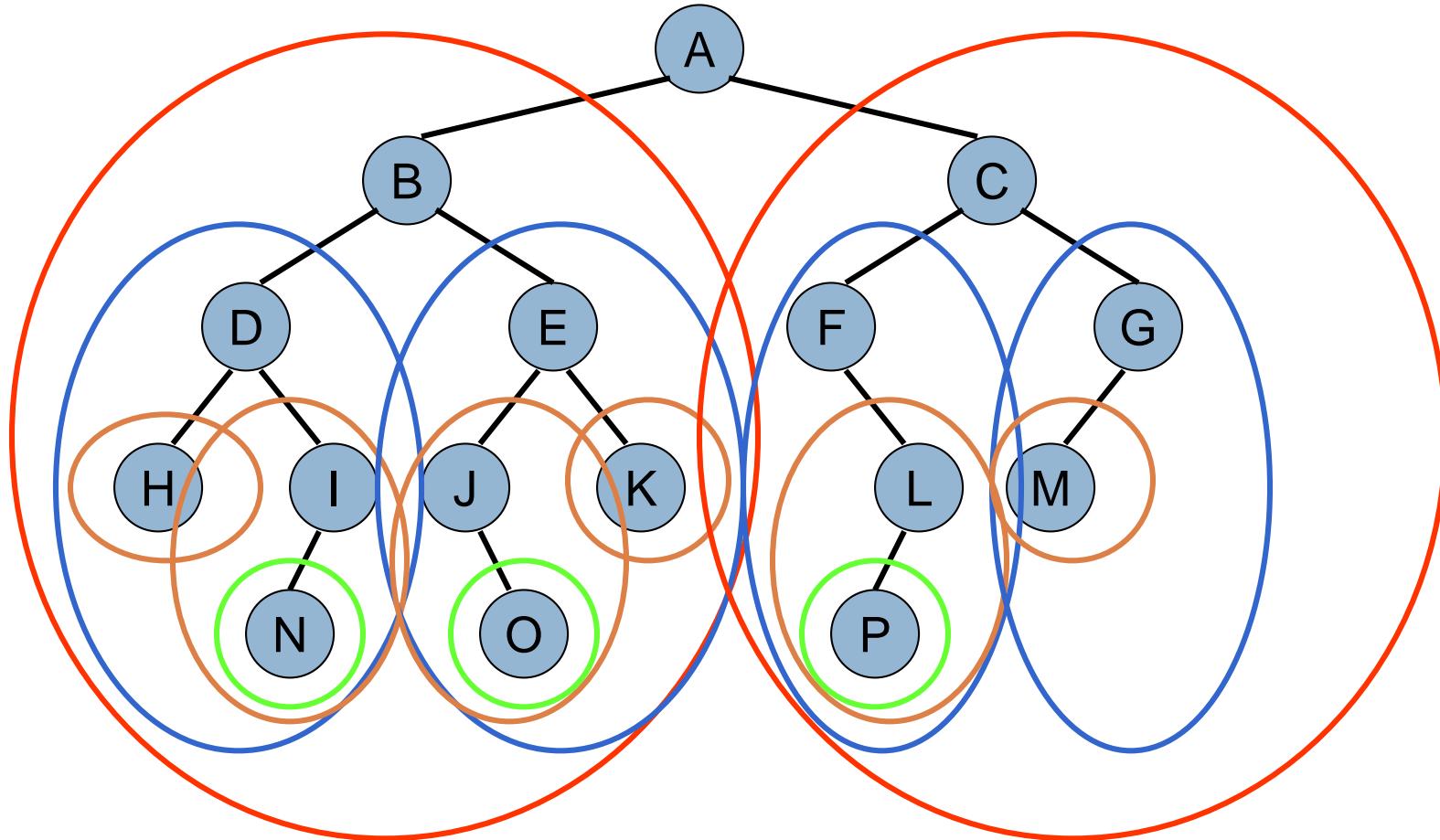
Binary Tree - Duyệt cây nhị phân

42

- 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 NLR

43



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

Binary Tree - Duyệt cây nhị phân

44

- 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 root)
{
    if (root != NULL)
    {
        // Xử lý tương ứng theo nhu cầu
        NLR (root->left) ;
        NLR (root->right) ;
    }
}
```

Binary Tree - Duyệt cây nhị phân

Duyệt theo thứ tự trước NLR (Node-Left-Right)

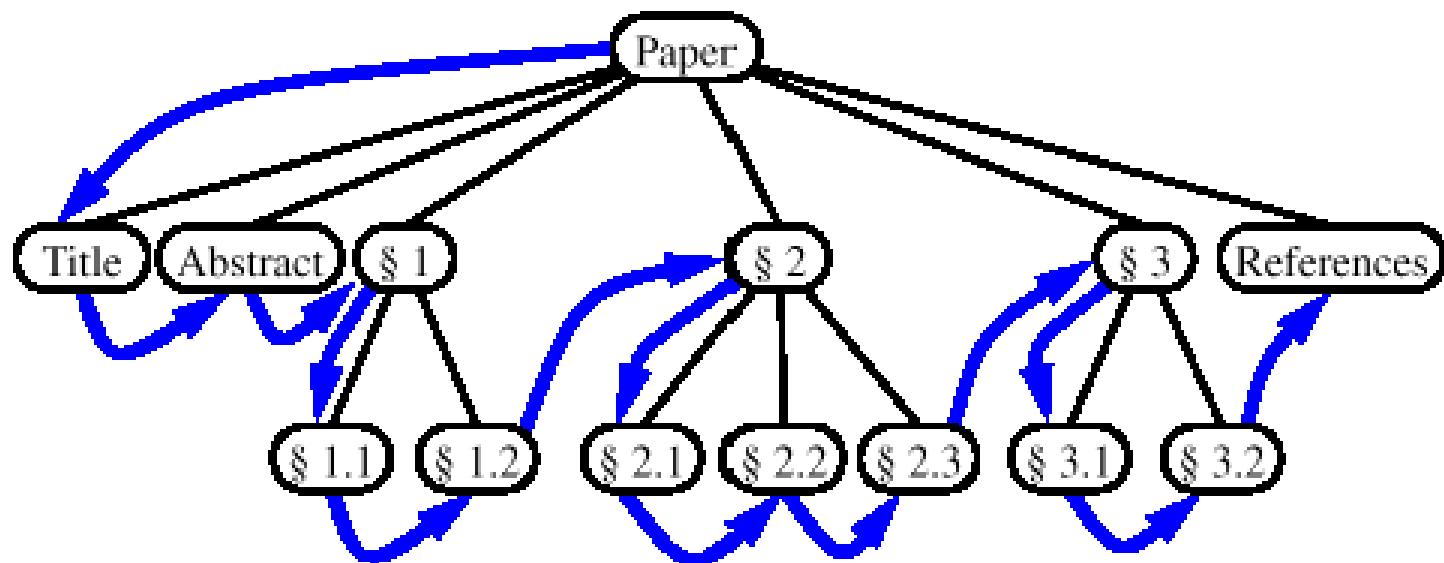
45

```
161     #Duyệt NLR
162     def duyet_nut_trai_phai(self,goc=0):
163         #Duyệt theo NLR
164         nut_ht = goc
165         if goc == None:
166             nut_ht = self.goc
167             #if
168             #kiểm tra nút hiện tại có bằng None không
169             if nut_ht == None:
170                 return []
171             else: #cây có giá trị
172                 kq = []
173                 kq.append(nut_ht.khoa)
174                 kq_trai = self.duyet_nut_trai_phai(nut_ht.trai)
175                 for x in kq_trai:
176                     kq.append(x)
177                     #For duyệt trái
178
179                     #Duyệt phải
180                     kq_phai = self.duyet_nut_trai_phai(nut_ht.phai)
181                     for x in kq_phai:
182                         kq.append(x)
183                         #for
184                     return kq
```

Cây nhị phân

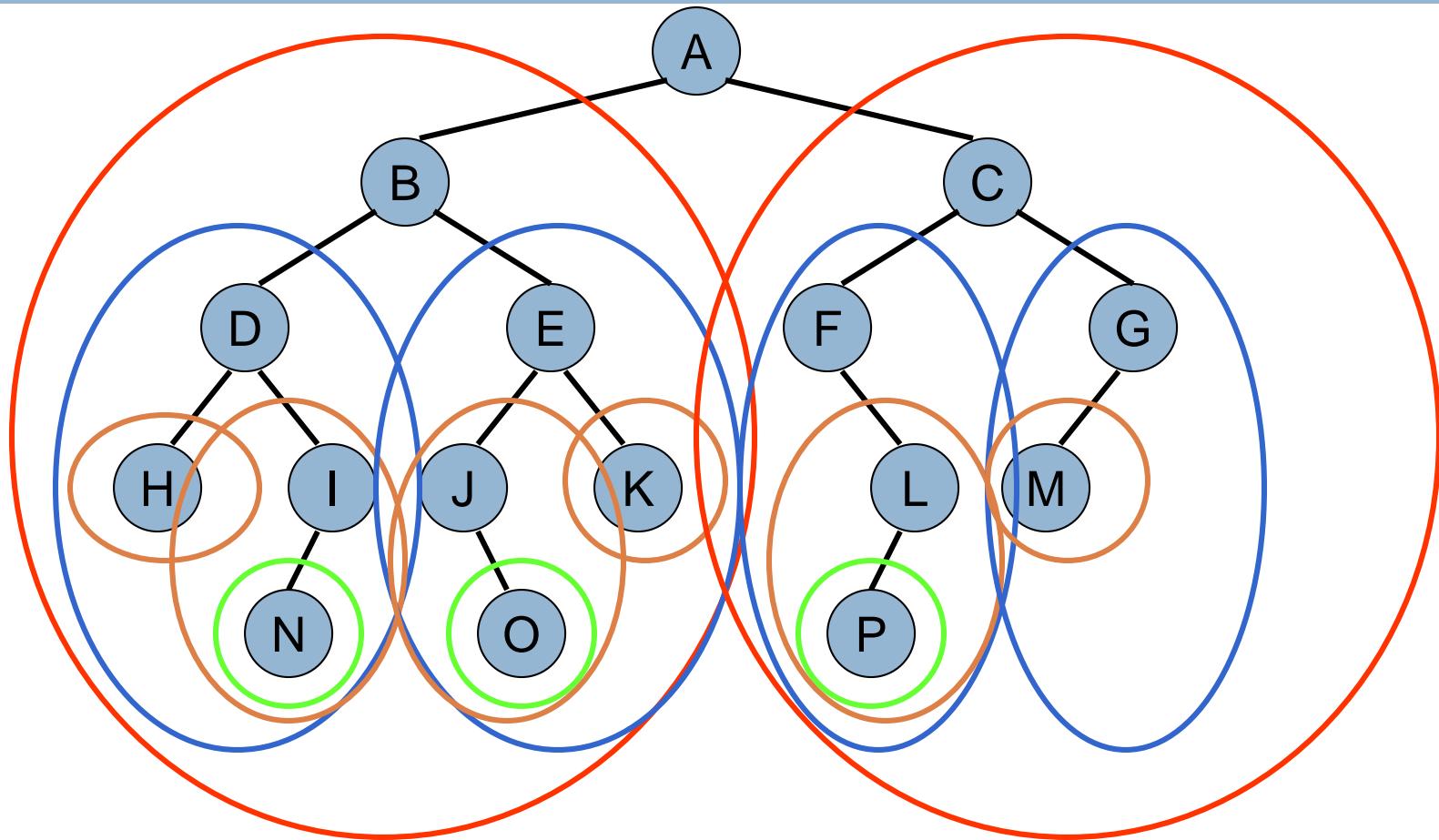
Duyệt theo thứ tự trước (Node-Left-Right)

- Một ví dụ: đọc một quyển sách hay bài báo từ đầu đến cuối như minh họa trong hình bên dưới:



Binary Tree - Duyệt cây nhị phân LNR

47



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

Binary Tree - Duyệt cây nhị phân

48

- ❑ 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 root)
{
    if (root != NULL)
    {
        LNR(root->left) ;
        //Xử lý nút t theo nhu cầu
        LNR(root->right) ;
    }
}
```

Binary Tree - Duyệt cây nhị phân

Duyệt theo thứ tự giữa LNR (Left-Node-Right)

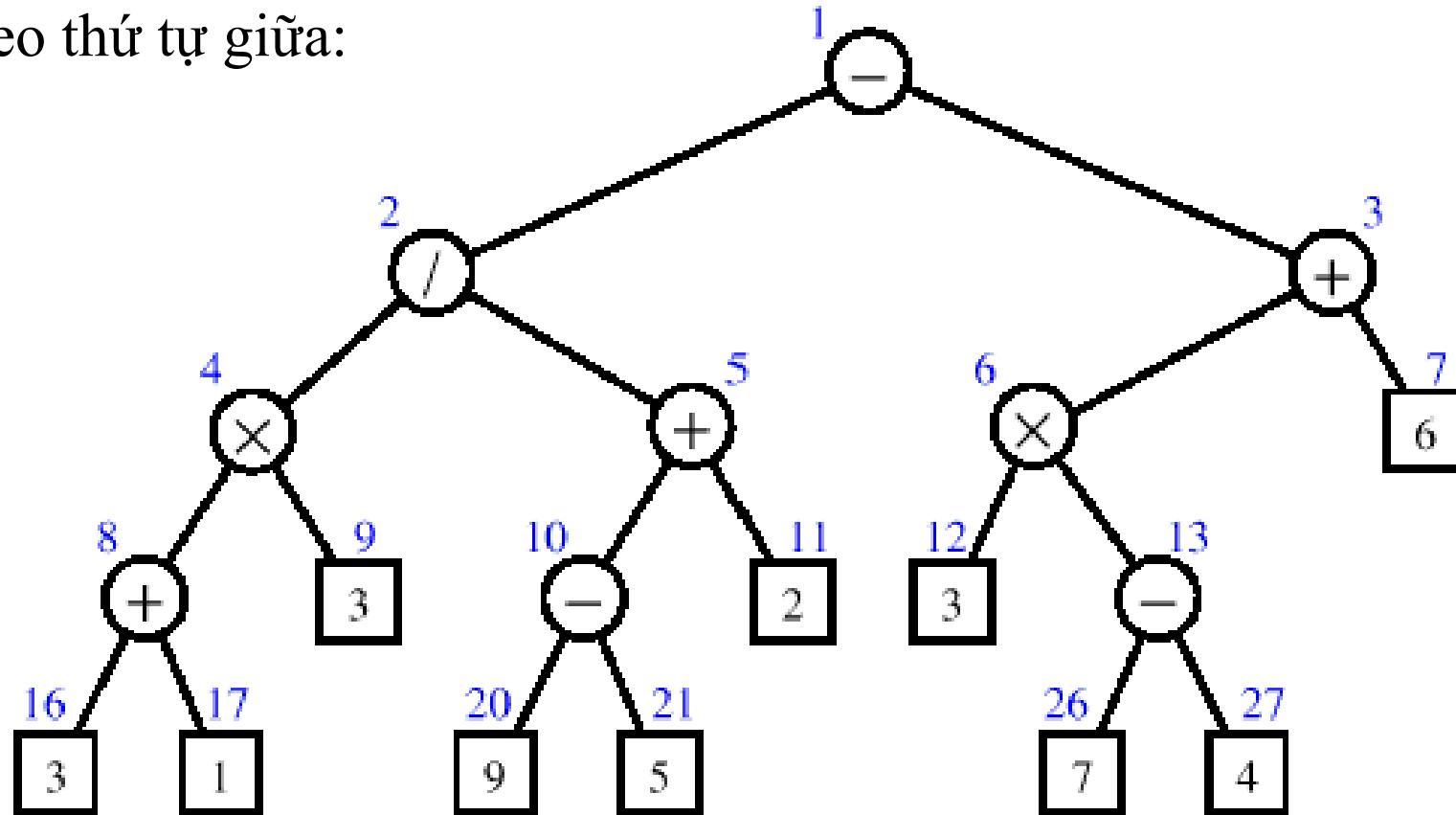
49

```
135     #Duyệt cây
136     def duyet_trai_nut_phai(self,goc=0):
137         #Duyệt theo LNR
138         nut_ht = goc
139         if goc == 0:
140             nut_ht = self.goc
141         #if
142         #kiểm tra nút hiện tại có bằng None không
143         if nut_ht == None:
144             return []
145         else: #cây có giá trị
146             kq = []
147             kq_trai = self.duyet_trai_nut_phai(nut_ht.trai)
148             for x in kq_trai:
149                 kq.append(x)
150             #For duyệt trái
151             kq.append(nut_ht.khoa)
152             #Duyệt phải
153             kq_phai = self.duyet_trai_nut_phai(nut_ht.phai)
154             for x in kq_phai:
155                 kq.append(x)
156             #for
157             return kq
```

Binary Tree - Duyệt cây nhị phân LNR

50

- Tính toán giá trị của biểu thức dựa trên cây biểu thức: duyệt cây theo thứ tự giữa:

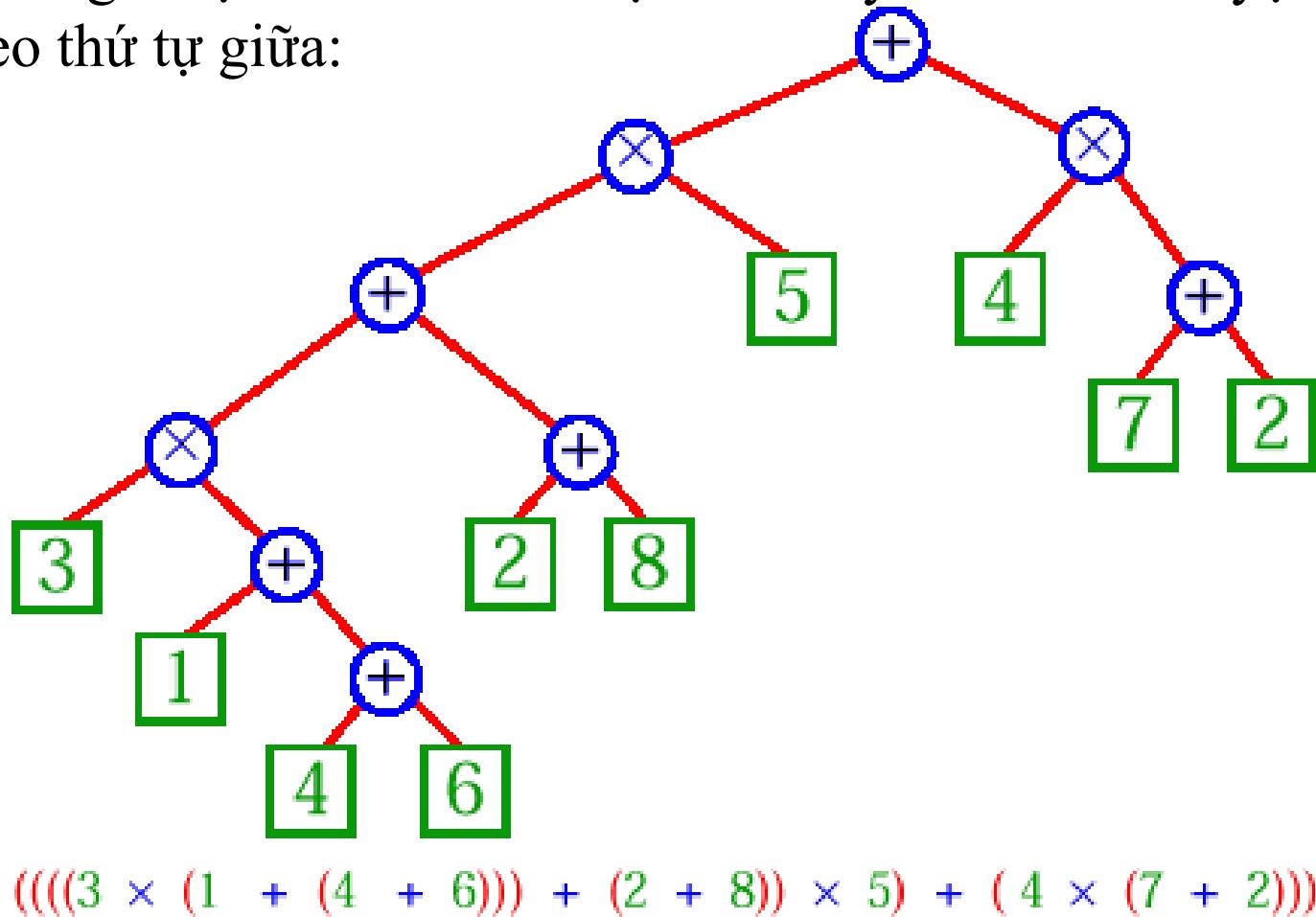


$$(3 + 1) \times 3 / (9 - 5 + 2) - (3 \times (7 - 4) + 6) = -13$$

Binary Tree – Ứng dụng

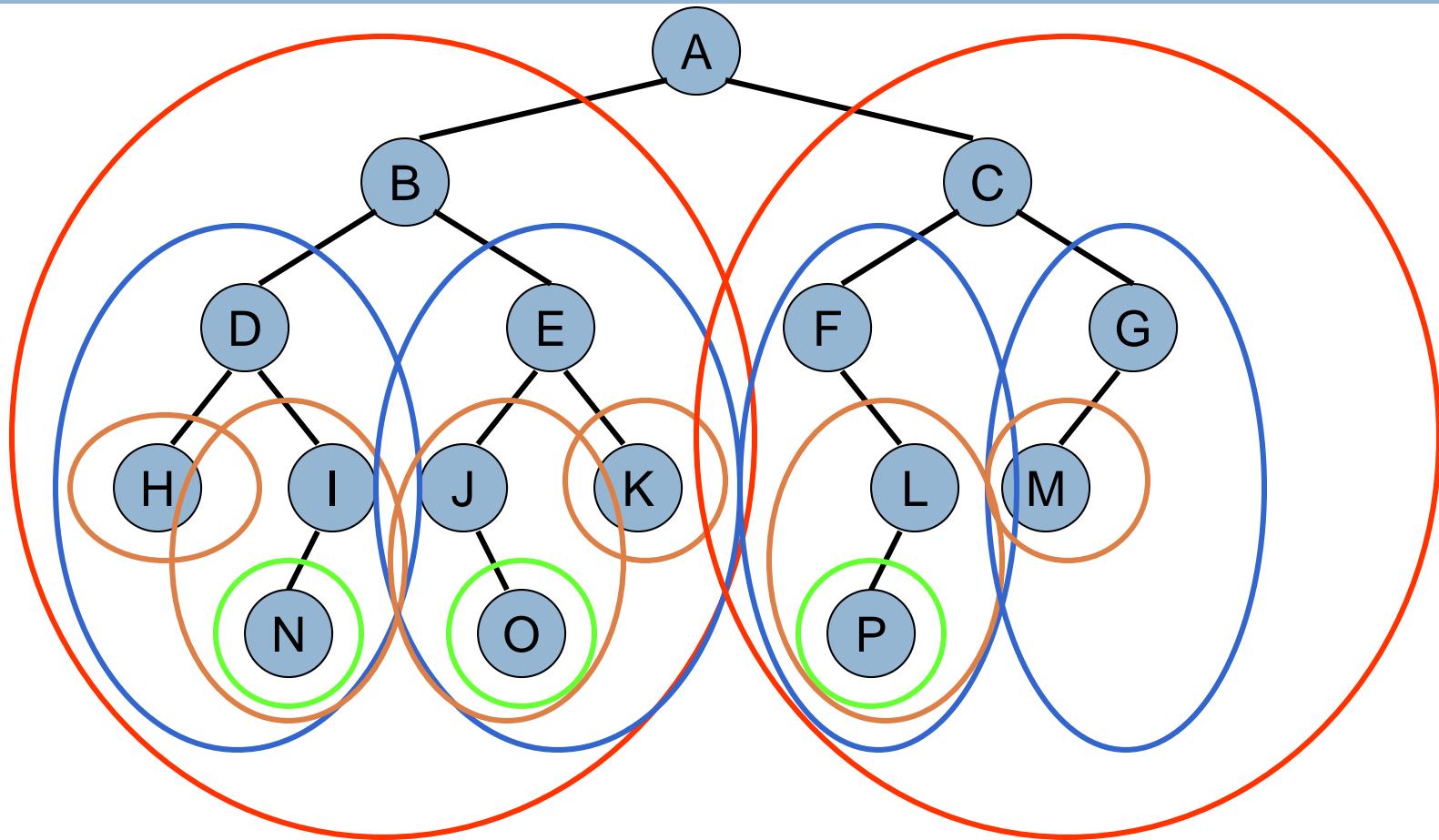
51

- Tính toán giá trị của biểu thức dựa trên cây biểu thức: duyệt cây theo thứ tự giữa:



Binary Tree - Duyệt cây nhị phân LRN

52



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

Binary Tree - Duyệt cây nhị phân

53

- 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 root)
{
    if (root != NULL)
    {
        LRN(root->left) ;
        LRN(root->right) ;
        // Xử lý tương ứng t theo nhu cầu
    }
}
```

Binary Tree - Duyệt cây nhị phân

Duyệt theo thứ tự sau LRN (Left-Right-Node)

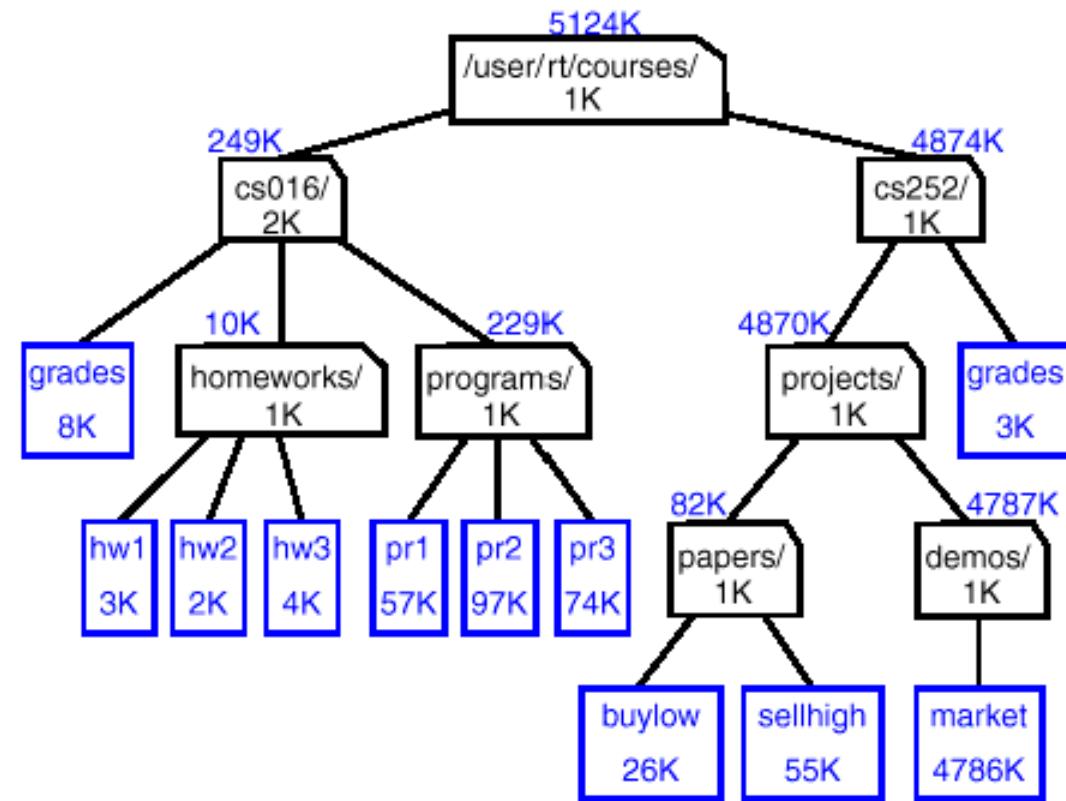
54

```
188     def duyet_trai_phai_nut(self,goc=0):
189         #Duyệt theo LRN
190         nut_ht = goc
191         if goc == None:
192             nut_ht = self.goc
193         #if
194         #kiểm tra nút hiện tại có bằng None không
195         if nut_ht == None:
196             return []
197         else: #cây có giá trị
198             kq = []
199             kq_trai = self.duyet_trai_phai_nut(nut_ht.trai)
200             for x in kq_trai:
201                 kq.append(x)
202                 #For duyệt trái
203
204                 #Duyệt phải
205                 kq_phai = self.duyet_trai_phai_nut(nut_ht.phai)
206                 for x in kq_phai:
207                     kq.append(x)
208                     #Not
209                     kq.append(nut_ht.khoa)
210                     #for
211                     return kq
```

Cây nhị phân

Duyệt theo thứ tự sau (Left-Right-Node)

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



Trắc nghiệm

56

- Give the binary tree with root A. The root has left child B and right child C. B has left child D and right child E. There are no other nodes in the tree.

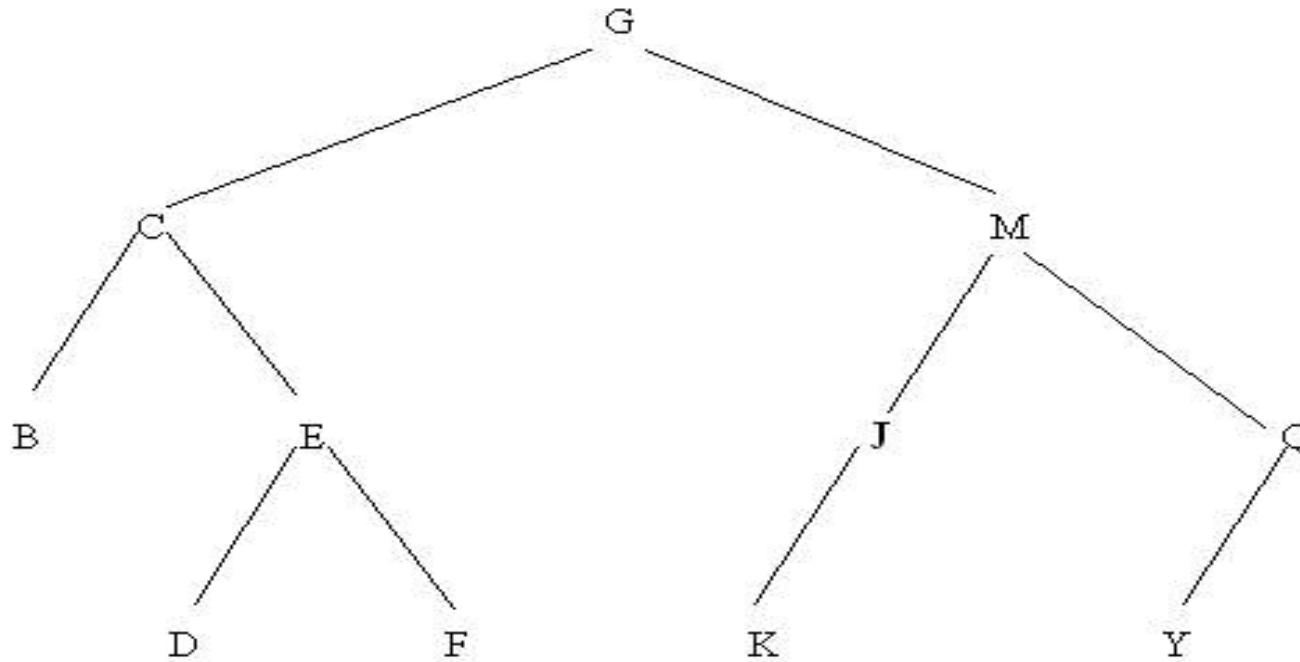
Which of the following traversals yields ABCDE?

- a) Inorder
- b) Preorder
- c) All of the others answers
- d) None of the others answers

Trắc nghiệm

57

- The order in which the nodes of this tree would be visited by a post-order traversal is

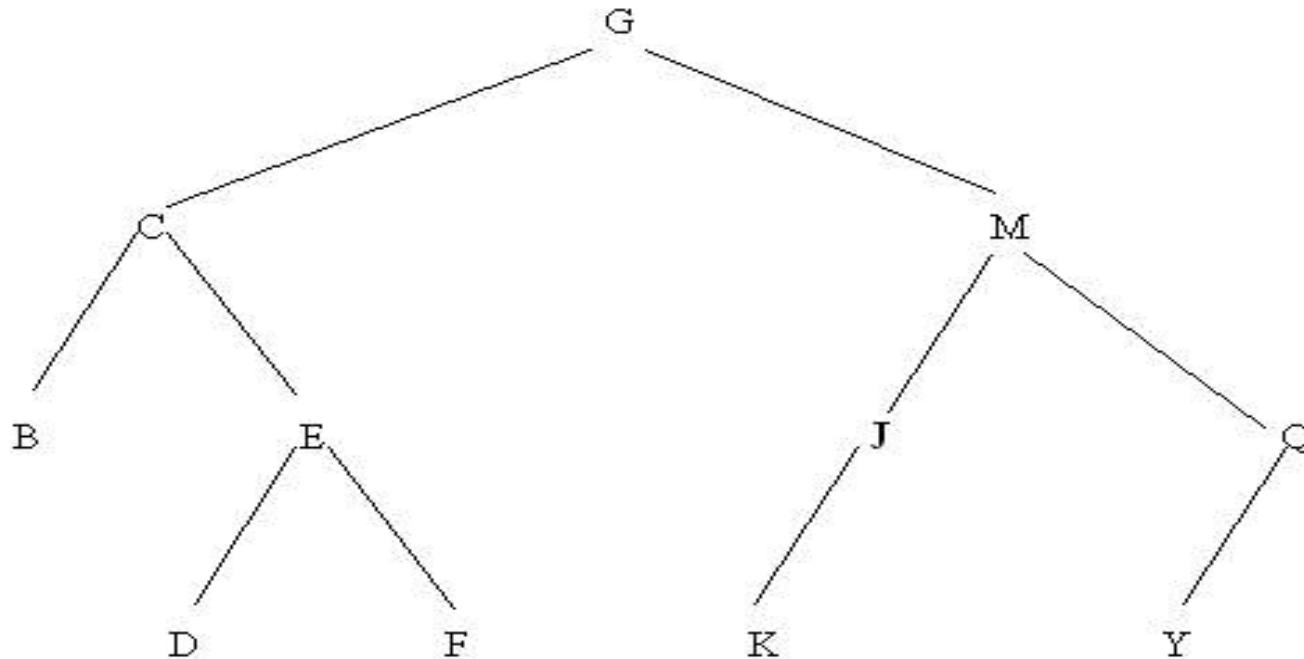


- a) GCMBEJQDFKY
- b) BCDFEJKYQMG
- c) GCBEDFMJKQY
- d) BDFECKJYQMG

Trắc nghiệm

58

- The order in which the nodes of this tree would be visited by a pre-order traversal is

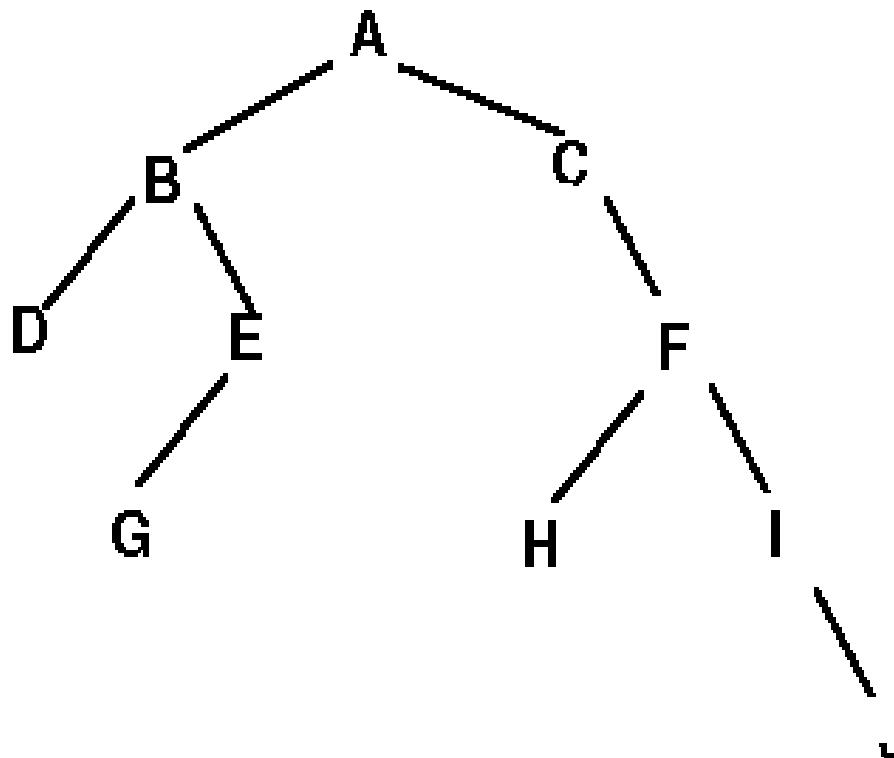


- a) GCMBEJQDFKY
- b) BCDFEJKYQMG
- c) BCDEFGKJMYQ
- d) GCBEDFMJKQY

Bài tập

59

- Trình bày các kết quả thức duyệt pre-order, inorder, post-order của cây sau:



Bài tập

60

Cho biểu thức $(x-y+z)^*t + (u/v - x/z)$

1. Viết lại biểu thức dưới dạng tiền tố, hậu tố
2. Vẽ cây nhị phân biểu diễn biểu thức.

Bài tập

61

- Vẽ cây biểu diễn cho biểu thức $((a+b)+c*(d+e)+f)*(g+h)$
- Trình bày biểu thức tiền tố và hậu tố của biểu thức đã cho.

Cây nhị phân

Biểu diễn cây tổng quát bằng cây nhị phân

- Nhược điểm của các cấu trúc cây tổng quát:
 - Bậc của các nút trên cây có thể dao động trong một biên độ lớn
⇒ việc biểu diễn gấp nhiều khó khăn và lãng phí.
 - Việc xây dựng các thao tác trên cây tổng quát phức tạp hơn trên cây nhị phân nhiều.
- Vì vậy, thường nếu không quá cần thiết phải sử dụng cây tổng quát, người ta chuyển cây tổng quát thành cây nhị phân.

Cây nhị phân

Biểu diễn cây tổng quát bằng cây nhị phân

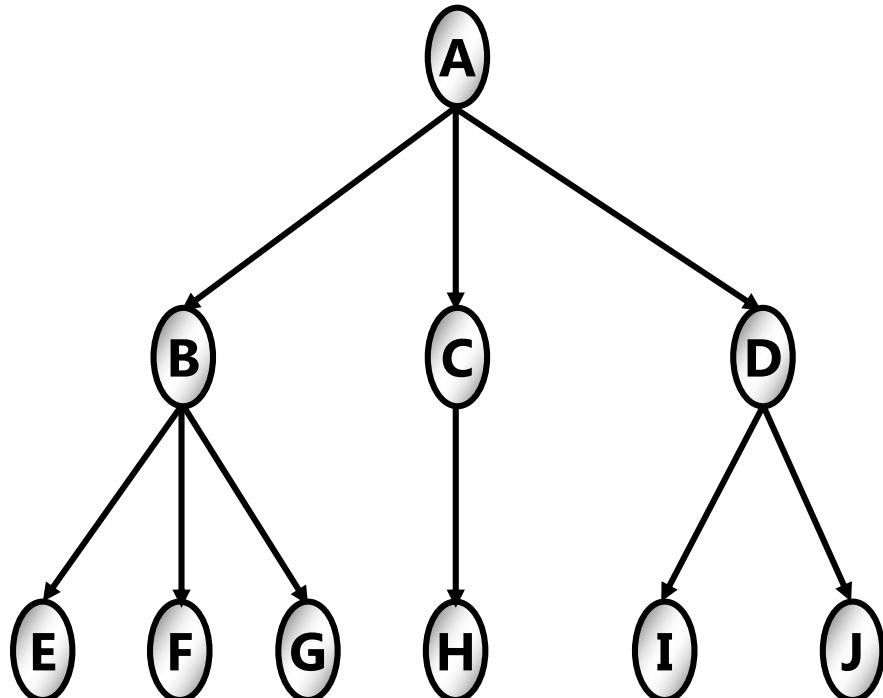
- Ta có thể biến đổi một cây bất kỳ thành một cây nhị phân theo qui tắc sau:
 - Giữ lại nút con trái nhất làm nút con trái.
 - Các nút con còn lại chuyển thành nút con phải.
 - Như vậy, trong cây nhị phân mới, con trái thể hiện quan hệ cha con và con phải thể hiện quan hệ anh em trong cây tổng quát ban đầu.

Giữ lại nút con trái nhất làm nút con trái.

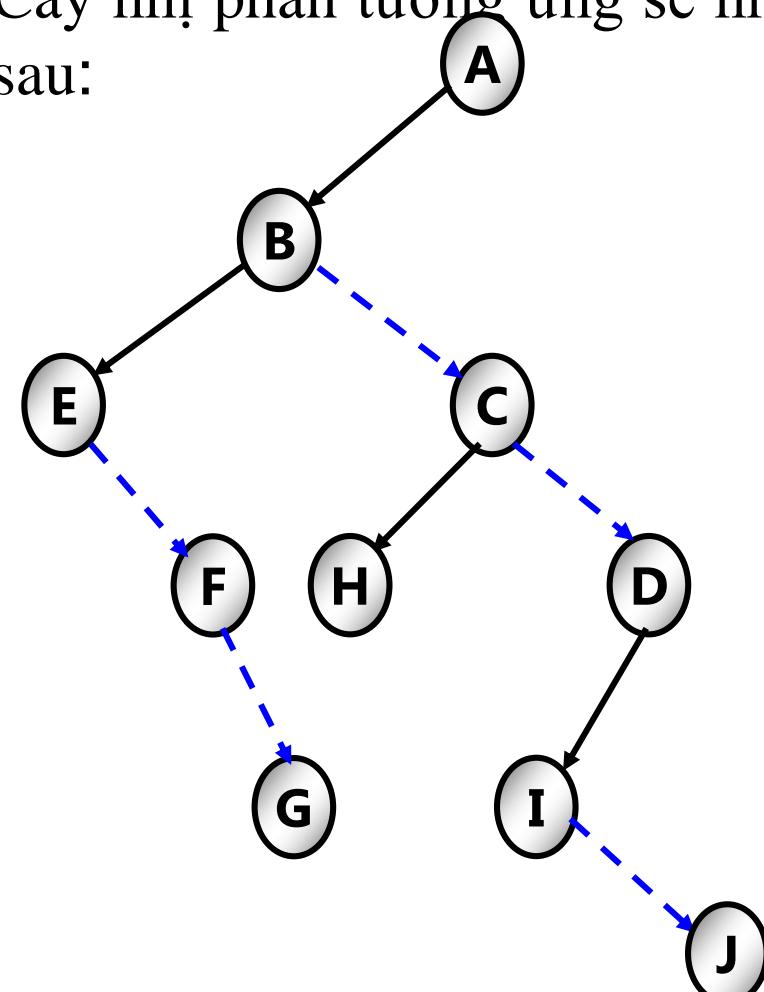
Các nút con còn lại chuyển thành nút con phải.

Như vậy, trong cây nhị phân mới, con trái thể hiện quan hệ cha con và con phải thể hiện quan hệ anh em trong cây tổng quát ban đầu.

- Giả sử có cây tổng quát như hình bên dưới:



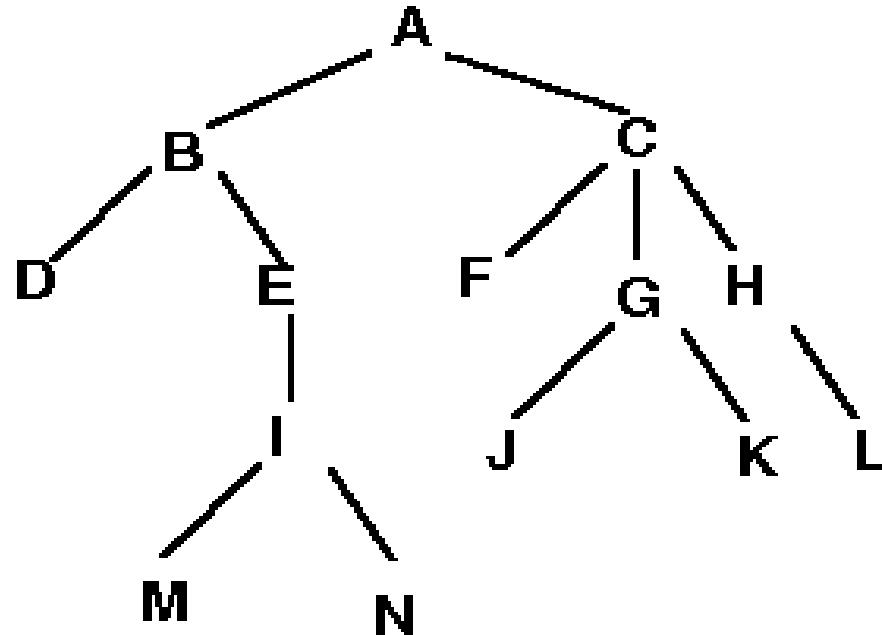
- Cây nhị phân tương ứng sẽ như sau:



Cây nhị phân

Biểu diễn cây tổng quát bằng cây nhị phân

- Chuyển cây tổng quát sau sang dạng nhị phân



Một cách biểu diễn cây nhị phân khác

66

- BinaryTree class still remains abstract
- It inherits all properties from the Tree class (e.g., parent(), is_leaf(), root() etc)
- Provides two declarations for two new abstract methods:
 - left(p) - position of p's left child
 - right(p) - position of p's right child
- Provides concrete implementation of the methods:
 - sibling(p) - other child of p's parent
 - children(p) - generator for the ordered children of a node.

```
1 class BinaryTree(Tree):  
2     '''Abstract base class representing a binary tree structure.'''  
3  
4     # ----- additional abstract methods -----  
5     def left(self, p):  
6         ...  
7         Return a Position representing p's left child.  
8         Return None if p does not have a left child.  
9  
10    raise NotImplementedError('Must be implemented by subclass')  
11  
12    def right(self, p):  
13        ...  
14        Return a Position representing p's right child.  
15        Return None if p does not have a right child  
16  
17    raise NotImplementedError('Must be implemented by subclass')  
18  
19    # ----- concrete methods implemented in this class -----  
20  
21    def sibling(self, p):  
22        '''Return a Position representing p's sibling (or None if no sibling).'''  
23        parent = p.parent()  
24        if parent is None:          # p must be the root  
25            return None             # root has no sibling  
26        else:  
27            if p == self.left(parent):  
28                return self.right(parent)  # possibly None  
29            else:  
30                return self.left(parent) # possibly None  
31  
32  
33    def children(self, p):  
34        '''Generate an iteration of Positions representing p's children'''  
35        if self.left(p) is not None:  
36            yield self.left(p)  
37        if self.right(p) is not None:  
38            yield self.right(p)
```

Một cách biểu diễn cây nhị phân khác

67

- A sub-class of BinaryTree class.
- It defines two nested classes:
 - A nonpublic _Node class to represent node.
 - A public Position class that wraps the node - inherited from Tree.Position class
- It provides following two utility methods:
 - _validity utility for robustly checking the validity of a given position while unwrapping it.
 - _make_position utility for wrapping a node as a position to return to a caller.
- Operations for updating Linked Binary Tree
 - T._add_root(e) , T._add_left(e)
 - T._add_right(e) , T._replace(p, e)
 - T._delete(p) , T._attach(p, T1, T2)

```
5     '''Linked representation of a binary tree structure.'''
6
7     class _Node:
8         __slots__ = '_element', '_parent', '_left', '_right'
9         def __init__(self, element, parent=None, left=None, right=None):
10             self._element = element
11             self._parent = parent
12             self._left = left
13             self._right = right
14
15     class Position(BinaryTree.Position):
16         '''An abstraction representing the location of a single element.'''
17
18         def __init__(self, container, node):
19             '''Constructor should not be invoked by user.'''
20             self._container = container
21             self._node = node
22
23         def element(self):
24             '''Return the element stored at this position.'''
25             return self._node._element
26
27         def __eq__(self, other):
28             '''Return True if other is a position representing the same location.'''
29             return type(other) is type(self) and other._node is self._node
30
31     # ----- hidden utility functions for LinkedBinary Tree -----
32
33     def validate(self, p):
34         '''Return associated node, if position is valid.'''
35         if not isinstance(p, self.Position):
36             raise TypeError('p must be proper Position type')
37         if p._container is not self:
38             raise ValueError('p does not belong to this container')
39         if p._node._parent is p._node: # convention for deprecated nodes
40             raise ValueError('p is no longer valid')
41         return p._node
42
43     def _make_position(self, node):
44         '''Return Position instance for a given node (or None if no node)'''
45         return self.Position(self, node) if node is not None else None
```

Một cách biểu diễn cây nhị phân khác

68

```
# ----- binary tree constructor -----
def __init__(self):
    '''Create an initially empty binary tree'''
    self._root = None
    self._size = 0

# ----- Public Accessors -----
def __len__(self):
    '''Return the total number of elements in the tree.'''
    return self._size

def root(self):
    '''Return the root Position of the tree (or None if tree is empty).'''
    return self._make_position(self._root)

def parent(self, p):
    '''Return the position P's parent (or None if p is root)'''
    node = self._validate(p)
    return self._make_position(node._parent)

def left(self, p):
    '''Return the Position P's left child (or None if no left child).'''
    node = self._validate(p)
    return self._make_position(node._left)

def right(self, p):
    '''Return the Position P's right child (or None if no right child).'''
    node = self._validate(p)
    return self._make_position(node._right)

def num_children(self, p):
    '''Return the number of children of Position P.'''
    node = self._validate(p)
    count = 0
    if node._left is not None:      # left child exists
        count += 1
    if node._right is not None:     # right child exists
        count += 1
    return count
```

```
# ----- NonPublic tree update Methods -----
def _add_root(self, e):
    ...
    Place element e at the root of an empty tree and return new Position.
    Raise ValueError if tree nonempty.
    ...
    if self._root is not None: raise ValueError('Root Exists.')
    self._size = 1
    self._root = self._Node(e)
    return self._make_position(self._root)

def _add_left(self, p, e):
    ...
    Create a new left child for Position P, storing element e.
    Return the position of a new node.
    Raise ValueError if Position p is invalid or p already has a left child.
    ...
    node = self._validate(p)
    if node._left is not None: raise ValueError('Left child exists')
    self._size += 1
    node._left = self._Node(e, node) # node is its parent
    return self._make_position(node._left)

def _add_right(self, p, e):
    ...
    Create a new right child for Position p, storing element e.
    Return the position of new node.
    Raise ValueError if Position p is invalid or p already has a right child.
    ...
    node = self._validate(p)
    if node._right is not None: raise ValueError('Right Child Exists')
    self._size += 1
    node._right = self._Node(e, node) # node is its parent
    return self._make_position(node._right)

def _replace(self, p, e):
    ...
    replace the element at position P with e and return the old element.'''
    node = self._validate(p)
    old = node._element
    node._element = e
    return old
```

Một cách biểu diễn cây nhị phân khác

69

```

148     def _delete(self, p):
149         """
150             Delete the node at Position p, and replace it with its child, if any.
151             Return the element that had been stored at Position p.
152             Raise ValueError if Position p is invalid or p has two children.
153         """
154
155         node = self._validate(p)
156         if self.num_children(p) == 2: raise ValueError('p has two children')
157         child = node._left if node._left else node._right
158         if child is not None:
159             child._parent = node._parent # child's grandparent becomes parent
160             if node is self._root:
161                 self._root = child # child becomes root if its parent is deleted
162             else:
163                 parent = node._parent
164                 if node is parent._left:
165                     parent._left = child
166                 else:
167                     parent._right = child
168             self._size -= 1
169             node._parent = None # convention for deprecated node
170             return node._element
171
172     def _attach(self, p, t1, t2):
173         """Attach trees t1 and t2 as left and right subtrees of external p."""
174
175         node = self._validate(p)
176
177         if not self.is_leaf(p): raise ValueError('position must be leaf')
178         if not type(self) is type(t1) is type(t2): # all 3 tree must be same type
179             raise TypeError('Tree types must match')
180         self._size += len(t1) + len(t2)
181
182         if not t1.is_empty(): # attach t1 as left subtree of node
183             t1._root._parent = node
184             node._left = t1._root
185             t1._root = None # set t1 instance to empty
186             t1.size = 0
187         if not t2.is_empty(): # attach t2 as right subtree of node
188             t2._root._parent = node
189             node._right = t2._root
190             t2._root = None # set t2 instance to empty
191             t2.size = 0

```

```
86     def _print_children(self, p, arr):
87         if p is not None:
88             arr += str(p.element()) + ': '
89             if self.num_children(p) > 0:
90                 for c in self.children(p):
91                     arr += str(c.element()) + '\t'
92                 arr += '\n'
93                 for c in self.children(p):
94                     arr = self._print_children(c, arr)
95             arr += '\n'
96         return arr
97
98     def __str__(self):
99         """ Provide a string representation of the tree """
100
101         start = self.root()
102         arr = ''
103         arr = self._print_children(start, arr)
104
105         return arr
```

Above function prints a binary tree.

Can you make it better by providing a more intuitive output?

Một cách biểu diễn cây nhị phân khác

7

```
192 #####
193
194 P = LinkedBinaryTree()
195 P._add_root('Providence')
196 P._add_left(P.root(), 'Chicago')
197 P._add_right(P.root(), 'Seattle')
198 P._add_left(P.left(P.root()), 'Baltimore')
199 P._add_right(P.left(P.root()), 'New York')
200 print(P)
201 print ('\n-----\n')
202
203 Q = LinkedBinaryTree()
204 Q._add_root('-')
205 Q._add_left(Q.root(), '/')
206 Q._add_right(Q.root(), '+')
207 Q._add_left(Q.left(Q.root()), 'X')
208 Q._add_right(Q.left(Q.root()), '+')
209
210 Q._add_left(Q.right(Q.root()), 'X')
211 Q._add_right(Q.right(Q.root()), 6)
212
213 Q._add_left(Q.left(Q.left(Q.root()))), '+')
214 Q._add_right(Q.left(Q.left(Q.root()))), '3')
215 Q._add_left(Q.left(Q.left(Q.left(Q.root())))), '3')
216 Q._add_right(Q.left(Q.left(Q.left(Q.root())))), '1')
217
218 Q._add_left(Q.right(Q.left(Q.root()))), '-')
219 Q._add_right(Q.right(Q.left(Q.root()))), '2')
220 Q._add_left(Q.left(Q.right(Q.left(Q.root())))), '9')
221 Q._add_right(Q.left(Q.right(Q.left(Q.root())))), '5')
222
223 Q._add_left(Q.left(Q.right(Q.root()))), '3')
224 Q._add_right(Q.left(Q.right(Q.root()))), '-')
225
226 Q._add_left(Q.right(Q.left(Q.right(Q.root())))), '7')
227 Q._add_right(Q.right(Q.left(Q.right(Q.root())))), '4')
228
229 print(Q)
```

Providence: Chicago
Chicago: Baltimore
Baltimore:
New York:

Seattle:

-: / +
/: X +
X: + 3
+: 3 1
3:
1:

3:
+: - 2
-: 9 5

9:
5:

2:

+: X 6
X: 3 -

3:
-: 7 4

7:
4:

6:

Một cách biểu diễn cây nhị phân khác

7

```
192 #####
193
194 P = LinkedBinaryTree()
195 P._add_root('Providence')
196 P._add_left(P.root(), 'Chicago')
197 P._add_right(P.root(), 'Seattle')
198 P._add_left(P.left(P.root()), 'Baltimore')
199 P._add_right(P.left(P.root()), 'New York')
200 print(P)
201 print ('\n-----\n')
202
203 Q = LinkedBinaryTree()
204 Q._add_root('-')
205 Q._add_left(Q.root(), '/')
206 Q._add_right(Q.root(), '+')
207 Q._add_left(Q.left(Q.root()), 'X')
208 Q._add_right(Q.left(Q.root()), '+')
209
210 Q._add_left(Q.right(Q.root()), 'X')
211 Q._add_right(Q.right(Q.root()), 6)
212
213 Q._add_left(Q.left(Q.left(Q.root()))), '+')
214 Q._add_right(Q.left(Q.left(Q.root()))), '3')
215 Q._add_left(Q.left(Q.left(Q.left(Q.root())))), '3')
216 Q._add_right(Q.left(Q.left(Q.left(Q.root())))), '1')
217
218 Q._add_left(Q.right(Q.left(Q.root()))), '-')
219 Q._add_right(Q.right(Q.left(Q.root()))), '2')
220 Q._add_left(Q.left(Q.right(Q.left(Q.root())))), '9')
221 Q._add_right(Q.left(Q.right(Q.left(Q.root())))), '5')
222
223 Q._add_left(Q.left(Q.right(Q.root()))), '3')
224 Q._add_right(Q.left(Q.right(Q.root()))), '-')
225
226 Q._add_left(Q.right(Q.left(Q.right(Q.root())))), '7')
227 Q._add_right(Q.right(Q.left(Q.right(Q.root())))), '4')
228
229 print(Q)
```

Providence: Chicago
Chicago: Baltimore
Baltimore:
New York:

Seattle:

-: / +
/: X +
X: + 3
+: 3 1
3:
1:

3:
+: - 2
-: 9 5

9:
5:

2:

+: X 6
X: 3 -

3:
-: 7 4

7:
4:

6:

Một cách biểu diễn cây nhị phân khác

72

Operation	Running Time
len, is_empty	$O(1)$
root, parent, left, right, sibling, children, num_children	$O(1)$
is_root, is_leaf	$O(1)$
depth(p)	$O(d_p + 1)$
height	$O(n)$
add_root, add_left, add_right, replace, delete, attach	$O(1)$

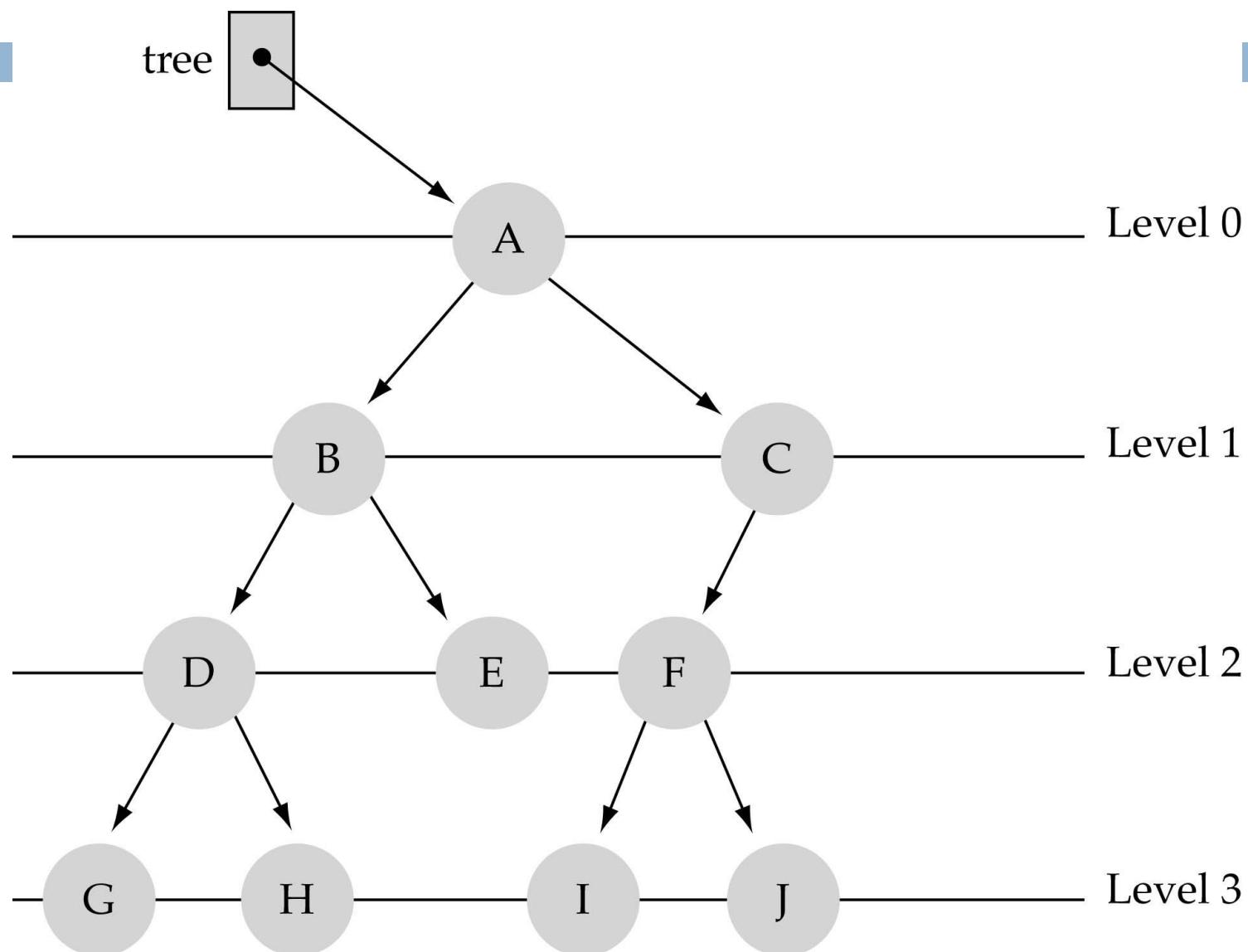
Một số thao tác trên cây

73

- Đếm số node
- Tính tổng của các nút có trong cây
- Đếm số node lá
- Tính chiều cao
- ...

Đếm số node

74



Đếm số node

75

- Thuật toán:
 - Nếu Tree rỗng, Số node (Tree) = 0
 - Ngược lại, Số node (Tree) = 1 + Số node (Tree.Left) + Số node (Tree.Right)

Đếm số node

76

```
int countnode(Tree root)
{
    if(root==NULL)
        return 0;
    return countnode(root->left)+countnode(root->right)+1;
}
```

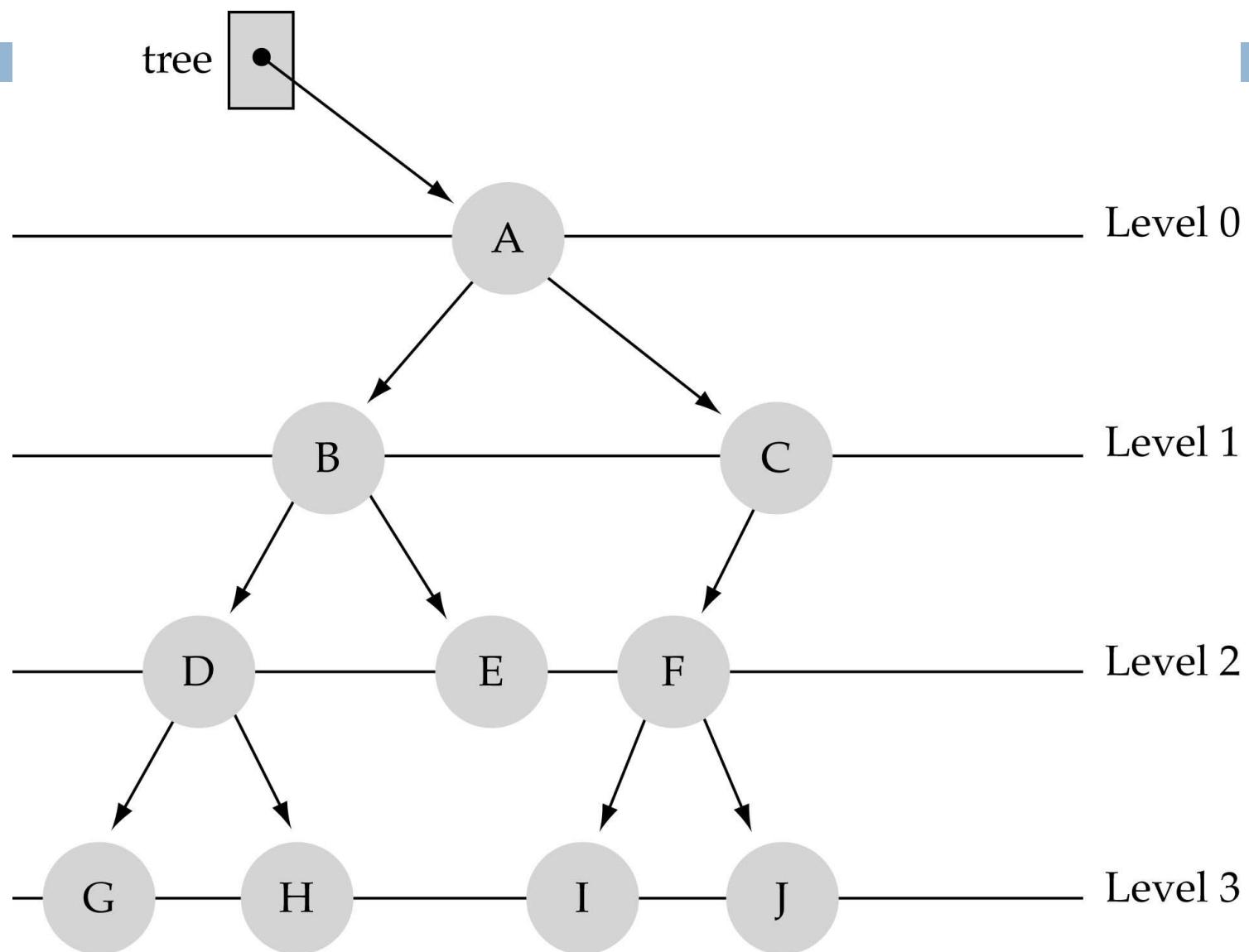
Tính tổng các node

77

```
int sum(Tree root)
{
    if(root==NULL)
        return 0;
    else
        return sum(root->left)+sum(root->right)+root->data;
}
```

Đếm số node lá

78



Đếm số node lá

79

- Thuật toán:
 - Nếu Tree rỗng, Số nút lá (Tree) = 0
 - Nếu Tree có nút lá, Số nút lá (Tree) = 1 + Số nút lá (Tree.Left) + Số nút lá (Tree.Right)
 - Nếu Tree không là nút lá, Số nút lá (Tree) = Số nút lá (Tree.Left) + Số nút lá (Tree.Right)

Đếm số node lá

80

```
int countleaf(Tree root)
{
    if(root==NULL)
        return 0;
    if((root->left==NULL)&&(root->right==NULL))
        return 1;
    return countleaf(root->left)+countleaf(root->right);
}
```

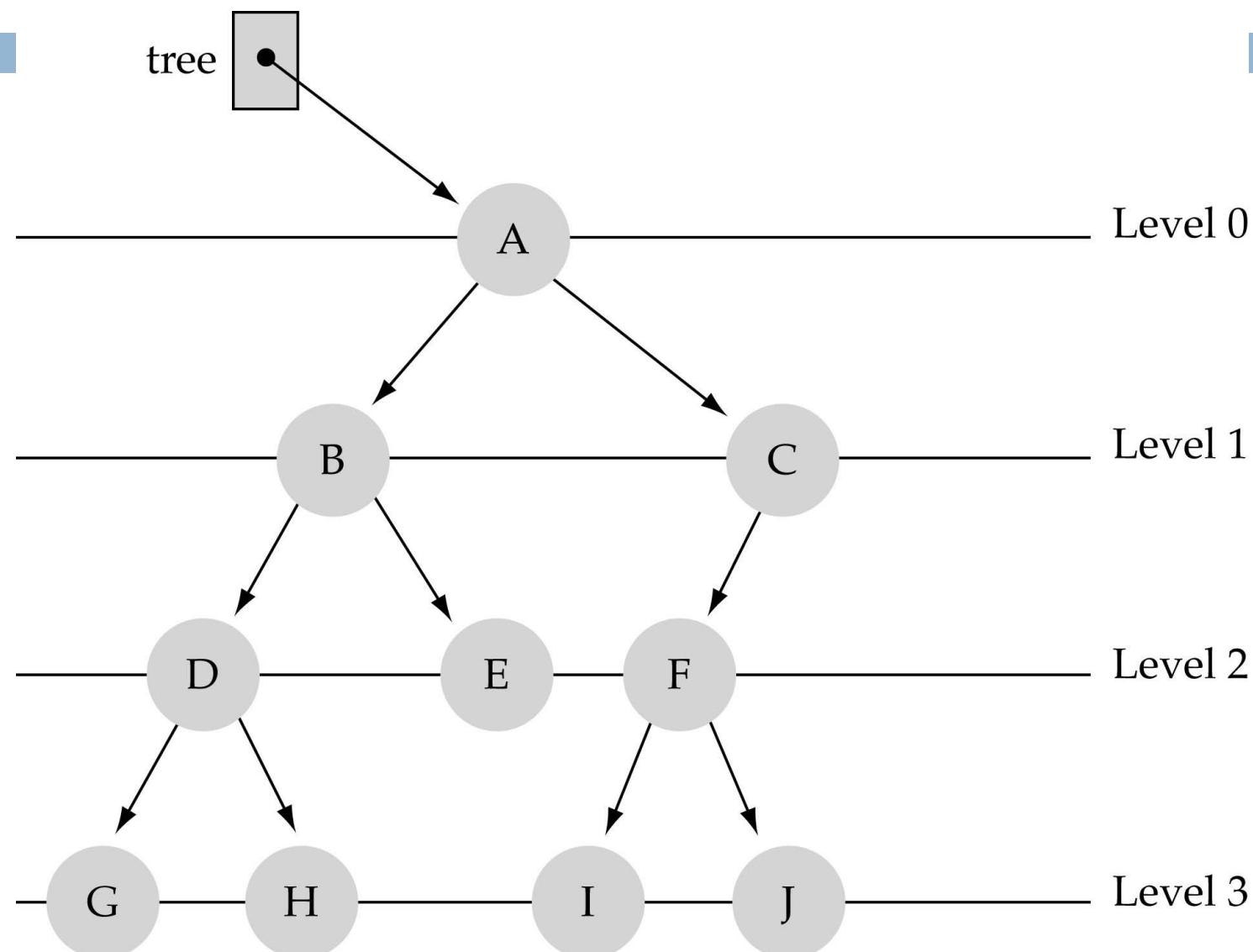
Đếm số node không là nút lá

81

```
int countnonleaf(Tree root)
{
    if(root==NULL)
        return 0;
    if((root->left!=NULL)|| (root->right!=NULL))
        return 1;
    return countnonleaf(root->left)+countnonleaf(root->right)+1;
}
```

Tính chiều cao

82



Tính chiều cao

83

- Thuật toán:
 - Nếu Tree rỗng, $\text{Height}(\text{Tree}) = 0$
 - *Ngược lại, $\text{Height}(\text{Tree}) = 1 + \max(\text{Height}(\text{Tree.Left}), \text{Height}(\text{Tree.Right}))$*

Tính chiều cao

84

```
int Height(Tree root)
{
    if (root==NULL) return 0;
    else
    {
        int HL=Height(root->left);
        int HR=Height(root->right);
        return 1+((HL<HR) ? HR:HL);
    }
}
```

Bài tập

85

Hãy viết các chương trình con sau thực hiện trên cây nhị phân:

1. Kiểm tra cây rỗng
2. Kiểm tra nút n có phải là nút lá không.
3. Kiểm tra nút n có phải là nút cha của nút m không.
4. Tính chiều cao của cây.
5. Tính số nút của cây
6. Duyệt tiền tự, trung tự, hậu tự.
7. Đếm số nút lá của cây.
8. Đếm số nút trung gian của cây.
9. Nút có giá trị lớn nhất, nhỏ nhất, tổng giá trị các nút, trung bình giá trị các nút

Nội dung

86

- 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*)

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**

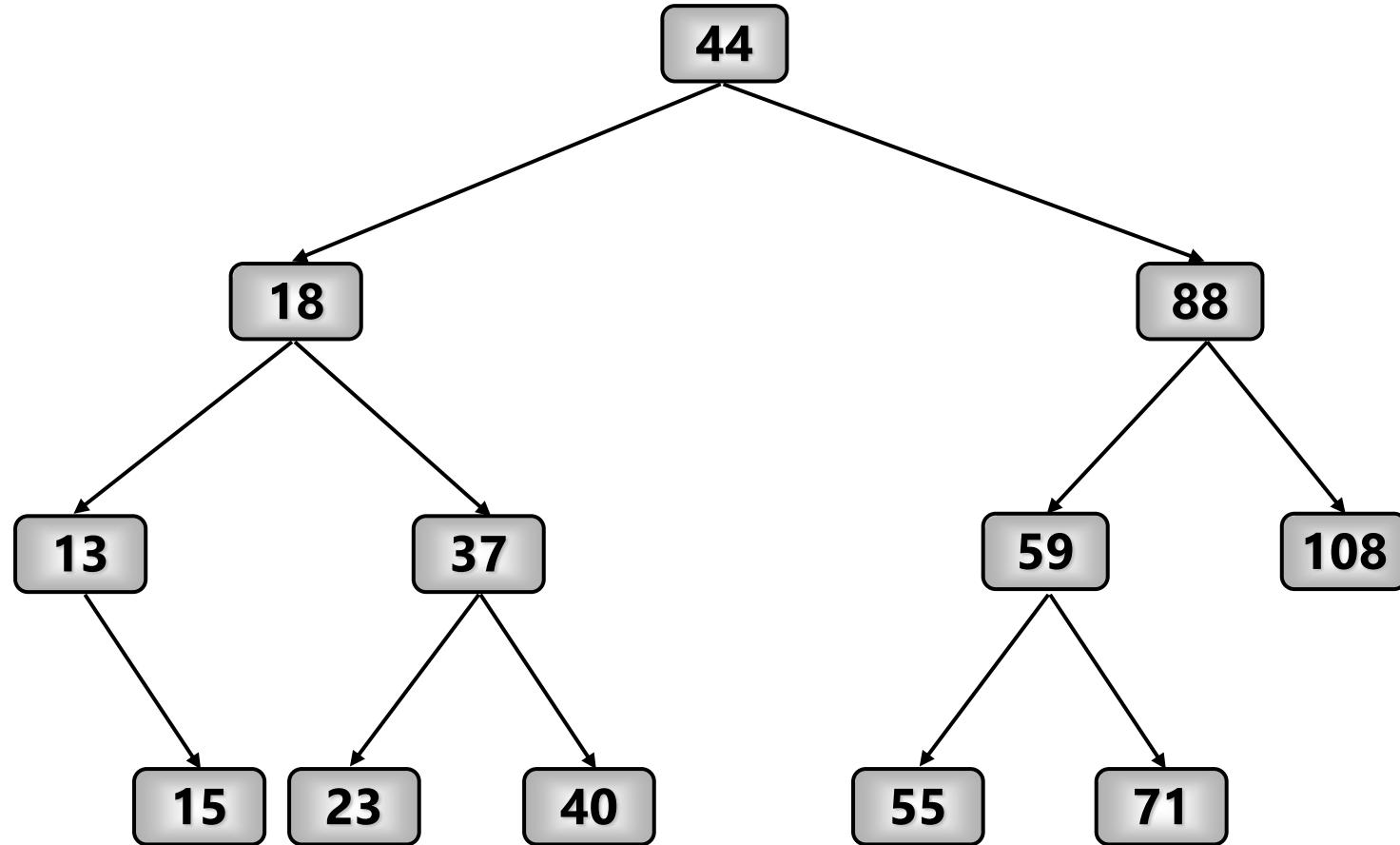
Binary Search Tree - Định nghĩa

88

- 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

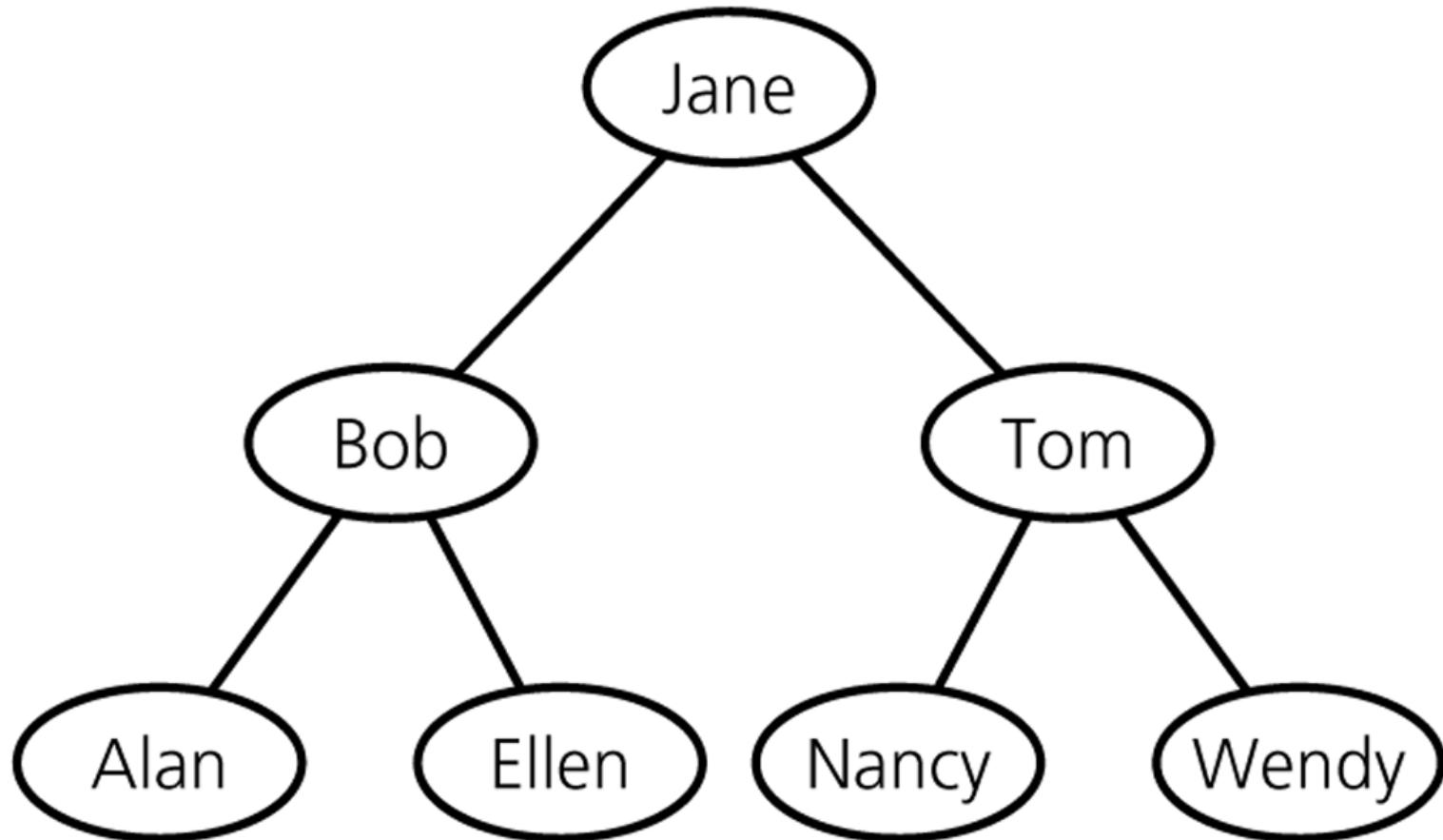
Binary Search Tree – Ví dụ

89



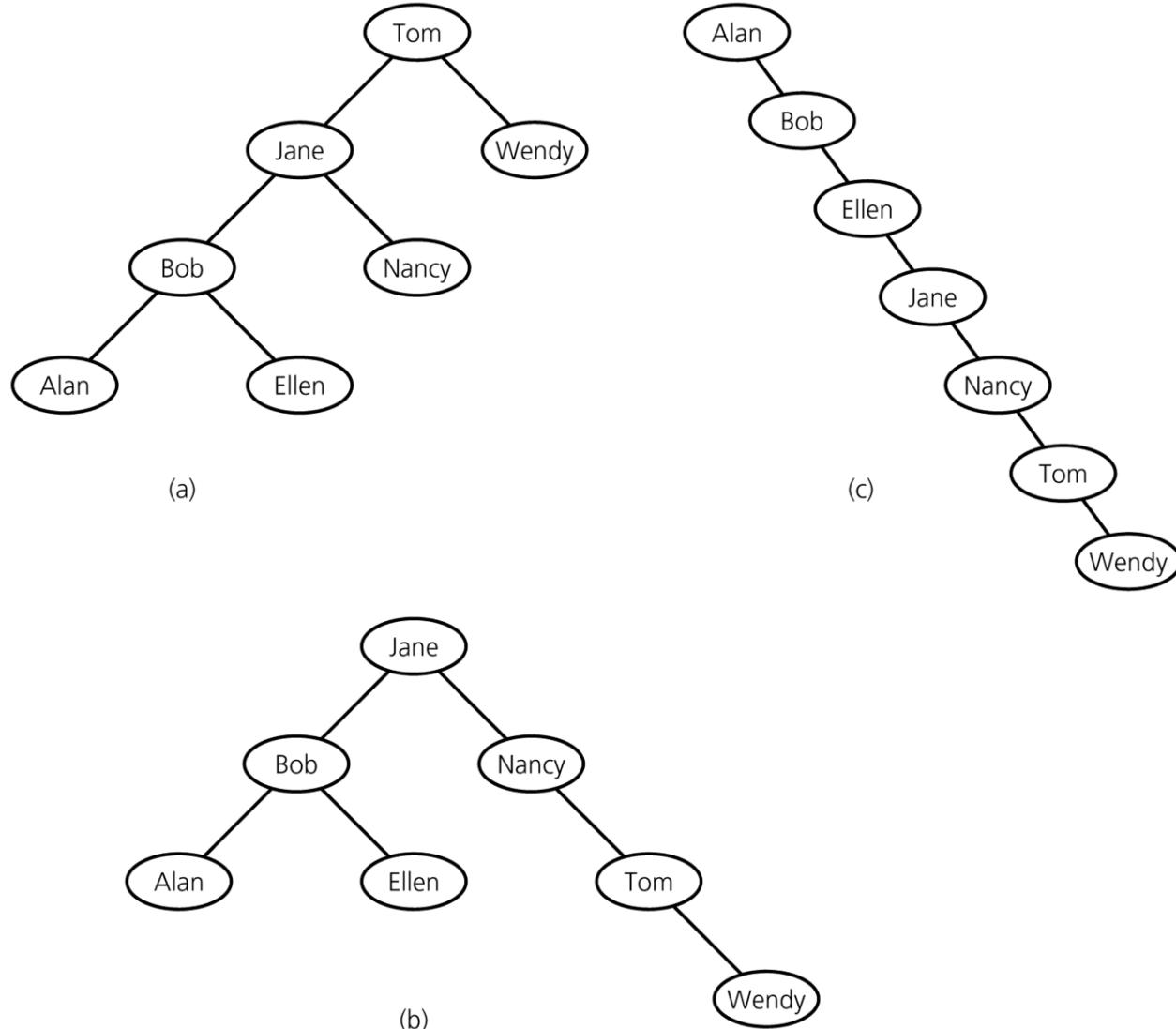
Binary Search Tree – Ví dụ

90



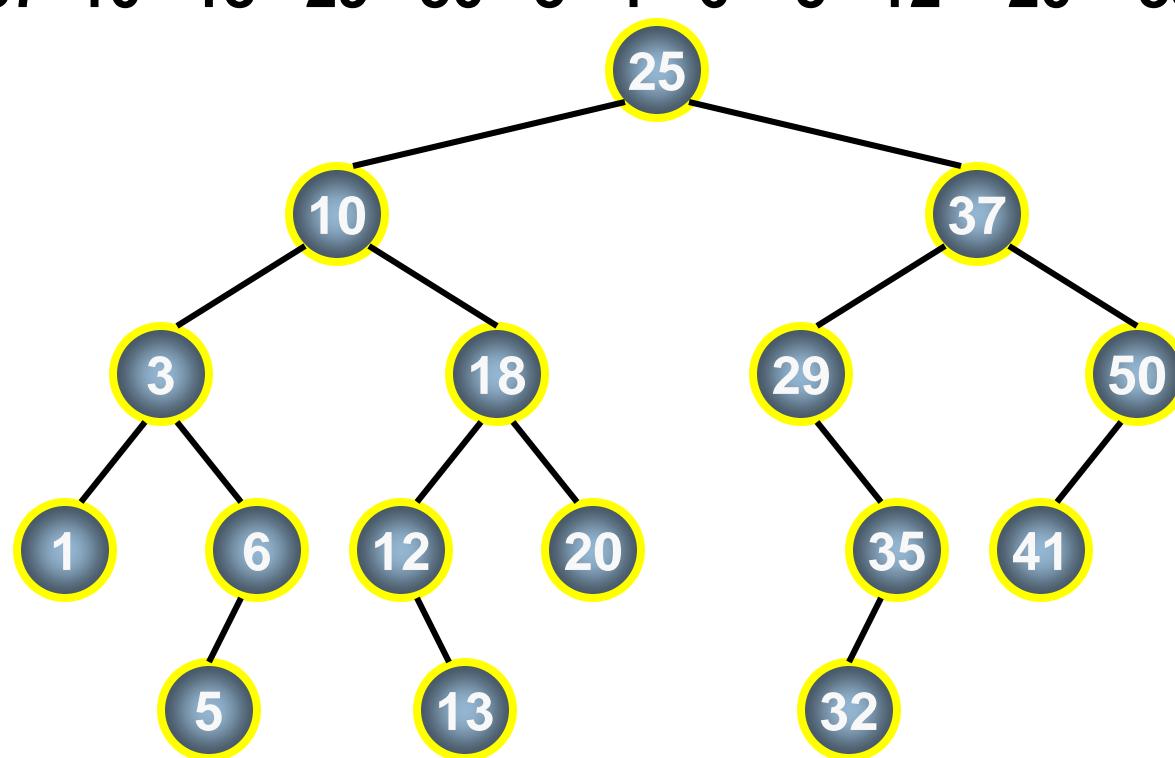
Binary Search Tree – Ví dụ

91



Tạo cây nhị phân tìm kiếm

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



Cấu trúc cây

Binary Search Tree

- We define the following three public methods for tree traversal:
- T.preorder()
- T.postorder()
- T.breadthfirst()
- T.positions() # uses preorder traversal by default. It is possible to access the elements of the tree as shown below:
 for p in T.positions():
 print(p.element())

```

1 class Tree:
2     '''Abstract Base Class representing a tree structure'''
3
4
5     def __iter__(self):
6         '''Generate an iteration of tree's elements'''
7         for p in self.positions():    # use same order as positions
8             yield p.element()        # but yield each element
9
10
11     def preorder(self):
12         '''Generate a preorder iteration of positions in the tree.'''
13         if not self.is_empty():
14             for p in self._subtree_preorder(self.root()):    # start recursion
15                 yield p
16
17     def _subtree_preorder(self, p):
18         '''Generate a preorder iteration of positions in subtree rooted at p.'''
19         yield p               # visit p before its subtrees
20         for c in self.children(p):    # for each child c
21             for other in self._subtree_preorder(c): # do preorder of c's subtree
22                 yield other          # yield each element
23
24
25     def positions(self):
26         '''Generate an iteration of the tree's positions.'''
27         return self.preorder()      # return entire preorder iteration
28
29
30     def postorder(self):
31         '''Generate post order iteration of positions in the tree.'''
32         if not self.is_empty():
33             for p in self._subtree_postorder(self.root()):    # start recursion
34                 yield p
35
36     def _subtree_postorder(self, p):
37         '''Generate a postorder iteration of positions in subtree rooted at p.'''
38         for c in self.children(p):    # for each child c
39             for other in self._subtree_postorder(c): # do postorder of c's subtree
40                 yield other          # yield each element
41         yield p                  # then visit p after visiting sub-trees.
42
43
44     def breadthfirst(self):
45         '''Generate a breadth-first iteration of the positions of the tree.'''
46         if not self.is_empty():
47             fringe = LinkedQueue()    # known positions not yet yielded
48             fringe.enqueue(self.root()) # store in queue starting with root
49             while not fringe.is_empty():
50                 p = fringe.dequeue()   # remove from front of the queue
51                 yield p              # report this position
52                 for c in self.children(p):
53                     fringe.enqueue(c)    # add children to back of queue
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137

```

Binary Search Tree – Ứng dụng

94

Table of Contents

- With Indent
- With explicit Numbering

Paper

Title

Abstract

§1

§1.1

§1.2

§2

§2.1

...

(a)

Paper

Title

Abstract

§1

§1.1

§1.2

§2

§2.1

...

(b)

Binary Search Tree – Ứng dụng

95

```
3 class LinkedTree(Tree):
4     '''Linked representation of general Tree Structure.'''
5
6     class _Node:
7         __slots__ = '_element', '_parent', '_children', '_noc'
8         def __init__(self, element, parent=None):
9             self._element = element
10            self._parent = parent
11            self._children = []
12            self._noc = 0 # number of children
13
14    class Position(Tree.Position):
15        '''An abstraction representing the location of a single element.'''
16
17        def __init__(self, container, node):
18            '''Constructor should not be invoked by user.'''
19            self._container = container
20            self._node = node
21
22        def element(self):
23            '''Return the element stored at this position.'''
24            return self._node._element
25
26        def __eq__(self, other):
27            '''Return True if other is a position representing the same location.'''
28            return type(other) is type(self) and other._node is self._node
29
30    # ----- hidden utility functions for LinkedTree -----
31    def _validate(self, p):
32        '''Return associated node, if position is valid.'''
33        if not isinstance(p, self.Position):
34            raise TypeError('p must be proper Position type')
35        if p._container is not self:
36            raise ValueError('p does not belong to this container')
37        if p._node._parent is p._node: # convention for deprecated nodes
38            raise ValueError('p is no longer valid')
39        return p._node
40
41    def _make_position(self, node):
42        '''Return Position instance for a given node (or None if no node)'''
43        return self.Position(self, node) if node is not None else None
44
45
46
47
48
49
50
51    # ----- Public Accessors -----
52    def __len__(self):
53        '''Return the total number of elements in the tree.'''
54        return self._size
55
56    def root(self):
57        '''Return the root Position of the tree (or None if tree is empty).'''
58        return self._make_position(self._root)
59
60    def parent(self, p):
61        '''return the position P's parent (or None if p is root)'''
62        node = self._validate(p)
63        return self._make_position(node._parent)
64
65    def children(self, p):
66        ''' Generate an iteration of Positions representing p's children'''
67        node = self._validate(p)
68        for i in range(node._noc):
69            if node._children[i] is not None:
70                yield self._make_position(node._children[i])
71
72    def num_children(self, p):
73        '''Return the number of children of Position P.'''
74        node = self._validate(p)
75        return node._noc
76
```

Binary Search Tree – Ứng dụng

96

```

77 # ----- Nonpublic tree update methods -----
78 def _add_root(self, e):
79     """
80     Place element e at the root of an empty tree and return new Position.
81     Raise ValueError if tree nonempty.
82     """
83     if self._root is not None: raise ValueError('Root Exists.')
84     self._size = 1
85     self._root = self._Node(e)
86     return self._make_position(self._root)
87
88 def _add_child(self, p, e):
89     """
90     Create a new left child for Position P, storing element e.
91     Return the position of a new node.
92     Raise ValueError if Position p is invalid or p already has a left child.
93     """
94     node = self._validate(p)
95     child = self._Node(e, node) # create a new child with node as its parent
96     node._children.append(child) # add child to the list with
97     self._size += 1
98     node._noc += 1
99     return self._make_position(child) # return current child position
100
101 def _replace(self, p, e):
102     """
103     replace the element at position P with e and return the old element.
104     """
105     node = self._validate(p)
106     old = node._element
107     node._element = e
108     return old

```

```

def _delete(self, p):
    """
    Delete the node at Position p, and replace it with its child, if any.
    Return the element that had been stored at Position p.
    Raise ValueError if Position p is invalid or p has two children.
    *** NOT TEST YET ***
    """

    node = self._validate(p)

    if node._noc > 1:
        raise ValueError('p has more than 1 child. Node can not be deleted')

    if node is self._root: # if we are deleting root node
        for i in range(node._noc):
            child = node._children[i]
            if child is not None:
                self._root = child
    else: # if not a root node
        for i in range(node._noc):
            child = node._children[i]
            if child is not None:
                child._parent = node._parent # child's grandparent becomes parent
    self._size -= 1
    node._parent._noc -= 1 # decrease the number of children for the parent
    node._parent = node # convention for deprecated node
    return node._element

def preorder_indent(self, p, d):
    """
    Print preorder representation of subtree T rooted a p at depth d.
    """
    print(2*d*' '+str(p.element()))
    if not self.is_leaf(p):
        for c in self.children(p):
            self.preorder_indent(c, d+1)

def print_tree(self):
    p = self.root()
    d = 0
    #set_trace()
    self.preorder_indent(p, d)

```

Binary Search Tree – Ứng dụng

97

```
A = LinkedTree()
A.add_root('Paper')
A.add_child(A.root(), 'Title')
A.add_child(A.root(), 'Abstract')
A.add_child(A.root(), 'Section1')
A.add_child(A.root(), 'Section2')
A.add_child(A.root(), 'Conclusion')
print('Root has {} children'.format(A.num_children(A.root())))
L1 = []
for child in A.children(A.root()):
    L1.append(child)

A.add_child(L1[2], 'Sec1.1')
A.add_child(L1[2], 'Sec1.2')
A.add_child(L1[2], 'Sec1.3')

print('L1[2] has {} children'.format(A.num_children(L1[2])))

A.add_child(L1[3], 'Sec2.1')
A.add_child(L1[3], 'Sec2.2')
A.add_child(L1[3], 'Sec2.3')

# Prints the labels with indent
A.print_tree_with_indent()

print("-----")

for p in A.preorder(): # preorder traversal
    print(p.element())
print("-----")

for p in A.postorder(): # postorder traversal
    print(p.element())
print("-----")

# Execute the cell containing LinkedQueue for this
for p in A.breadthfirst(): # breadthfirst traversal
    print(p.element())
```

↳ Root has 5 children
L1[2] has 3 children
Paper
 Title
 Abstract
 Section1
 Sec1.1
 Sec1.2
 Sec1.3
 Section2
 Sec2.1
 Sec2.2
 Sec2.3
 Conclusion

Paper
Title
Abstract
Section1
 Sec1.1
 Sec1.2
 Sec1.3
Section2
 Sec2.1
 Sec2.2
 Sec2.3
Conclusion

Title
Abstract
Section1
 Sec1.1
 Sec1.2
 Sec1.3
Section2
 Sec2.1
 Sec2.2
 Sec2.3
Conclusion

Title
Abstract
Section1
 Sec1.1
 Sec1.2
 Sec1.3
Section2
 Sec2.1
 Sec2.2
 Sec2.3
Conclusion

Paper
Title
Abstract
Section1
 Sec1.1
 Sec1.2
 Sec1.3
Section2
 Sec2.1
 Sec2.2
 Sec2.3
Conclusion

```
def preorder_label(self, p, d, path):
    '''Print a labeled representation of subtree T rooted at p at depth d.'''
    label='.'.join(str(j+1) for j in path) # displayed labels are one-indexed
    print(2*d*' '+label, p.element())
    path.append(0) # path entries are zero-indexed
    for c in self.children(p):
        self.preorder_label(c, d+1, path) # child depth is d+1
        path[-1] += 1
    path.pop()

def print_tree_with_labels(self):
    ''' Print tree with Labels.'''
    p = self.root()
    d = 0
    path = []
    self.preorder_label(p, d, path)
```

```
211 print("-----")
212
213 A.print_tree_with_labels()
...
-----  
Paper  
Title  
Abstract  
Section1  
    Section2  
        Conclusion  
    Sec1.1  
    Sec1.2  
    Sec1.3  
Section2  
    Sec2.1  
    Sec2.2  
    Sec2.3  
Conclusion  
-----  
Paper  
1 Title  
2 Abstract  
3 Section1  
    3.1 Sec1.1  
    3.2 Sec1.2  
    3.3 Sec1.3  
4 Section2  
    4.1 Sec2.1  
    4.2 Sec2.2  
    4.3 Sec2.3  
5 Conclusion
```

Binary Search Tree – Ứng dụng

98

```
A = LinkedTree()
A.add_root('Paper')
A.add_child(A.root(), 'Title')
A.add_child(A.root(), 'Abstract')
A.add_child(A.root(), 'Section1')
A.add_child(A.root(), 'Section2')
A.add_child(A.root(), 'Conclusion')
print('Root has {} children'.format(A.num_children(A.root())))
L1 = []
for child in A.children(A.root()):
    L1.append(child)

A.add_child(L1[2], 'Sec1.1')
A.add_child(L1[2], 'Sec1.2')
A.add_child(L1[2], 'Sec1.3')

print('L1[2] has {} children'.format(A.num_children(L1[2])))

A.add_child(L1[3], 'Sec2.1')
A.add_child(L1[3], 'Sec2.2')
A.add_child(L1[3], 'Sec2.3')

# Prints the labels with indent
A.print_tree_with_indent()

print("-----")

for p in A.preorder(): # preorder traversal
    print(p.element())
print("-----")

for p in A.postorder(): # postorder traversal
    print(p.element())
print("-----")

# Execute the cell containing LinkedQueue for this
for p in A.breadthfirst(): # breadthfirst traversal
    print(p.element())
```

↳ Root has 5 children
L1[2] has 3 children
Paper
 Title
 Abstract
 Section1
 Sec1.1
 Sec1.2
 Sec1.3
 Section2
 Sec2.1
 Sec2.2
 Sec2.3
 Conclusion

Paper
Title
Abstract
Section1
 Sec1.1
 Sec1.2
 Sec1.3
Section2
 Sec2.1
 Sec2.2
 Sec2.3
Conclusion

Title
Abstract
Section1
 Sec1.1
 Sec1.2
 Sec1.3
Section2
 Sec2.1
 Sec2.2
 Sec2.3
Conclusion

Title
Abstract
Section1
 Sec1.1
 Sec1.2
 Sec1.3
Section2
 Sec2.1
 Sec2.2
 Sec2.3
Conclusion

Paper
Title
Abstract
Section1
 Sec1.1
 Sec1.2
 Sec1.3
Section2
 Sec2.1
 Sec2.2
 Sec2.3
Conclusion

```
def preorder_label(self, p, d, path):
    '''Print a labeled representation of subtree T rooted at p at depth d.'''
    label='.'.join(str(j+1) for j in path) # displayed labels are one-indexed
    print(2*d*' '+label, p.element())
    path.append(0) # path entries are zero-indexed
    for c in self.children(p):
        self.preorder_label(c, d+1, path) # child depth is d+1
        path[-1] += 1
    path.pop()

def print_tree_with_labels(self):
    ''' Print tree with Labels.'''
    p = self.root()
    d = 0
    path = []
    self.preorder_label(p, d, path)
```

```
211 print("-----")
212
213 A.print_tree_with_labels()
...
-----  
Paper  
Title  
Abstract  
Section1  
Section2  
Conclusion  
-----  
Paper  
1 Title  
2 Abstract  
3 Section1  
    3.1 Sec1.1  
    3.2 Sec1.2  
    3.3 Sec1.3  
4 Section2  
    4.1 Sec2.1  
    4.2 Sec2.2  
    4.3 Sec2.3  
5 Conclusion
```

Binary Search Tree – Ứng dụng

99

- The *parenthetic string representation* $P(T)$ of tree T is recursively defined as follows:

- If T consists of a single position p , then $P(T) = \text{str}(p.\text{element}())$
- Otherwise, it is defined recursively as,

$$P(T) = \text{str}(p.\text{element}()) + '(' + P(T_1) + ', ' + \dots + ', ' + P(T_k) + ')'$$

where p is the root of T and T_1, T_2, \dots, T_k are the subtrees rooted at the children of p , which are given in order if T is an ordered tree.

Electronics R'Us

1 R&D
2 Sales
 2.1 Domestic
 2.2 International
 2.2.1 Canada
 2.2.2 S. America

Electronics R'Us (R&D, Sales (Domestic, International (Canada, S. America, Overseas (Africa, Europe, Asia, Australia))), Purchasing, Manufacturing (TV, CD, Tuner))

Parenthctic representation

Labeled representation

Binary Search Tree – Ứng dụng

100

```
167     def parenthesize(self, p):
168         '''Print parenthesized representation of subtree of T rooted at p.'''
169         print(p.element(), end='') # use of end avoids trailing newline
170         if not self.is_leaf(p):
171             first_time = True
172             for c in self.children(p):
173                 sep = ' (' if first_time else ', ' # determine proper separator
174                 print(sep, end='')
175                 first_time = False # any future passes will not be the first
176                 self.parenthesize(c) # recur on child
177                 print(')', end='') # include closing parenthesis
178
179     def print_tree_with_parenthesis(self):
180         '''Generate parenthetic representation of the tree.'''
181         self.parenthesize(self.root())
182
230     print("-----")
231     A.print_tree_with_parenthesis()
```

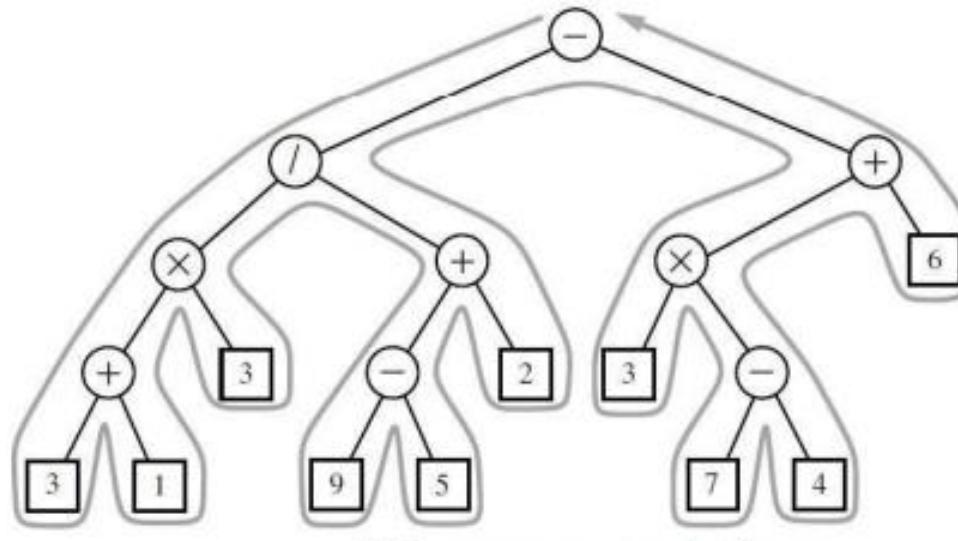
Euler Tour Traversal

101

- A more general tree traversal algorithm.
- “Walk” around the tree T where we start by going from the root toward its leftmost child, viewing the edges of T as being “walls” that we always keep to our left.
- The **complexity of the walk is $O(n)$**
 - because it progresses exactly two times along each of the $n-1$ edges of the tree—once going downward along the edge, and later going upward along the edge.
- Combines preorder and postorder traversals.
- There are two notable “visits” to each position p :
 - A “**pre visit**” occurs when first reaching the position, that is, when the walk passes immediately left of the node in our visualization.
 - A “**post visit**” occurs when the walk later proceeds upward from that position, that is, when the walk passes to the right of the node in our visualization.

Euler Tour Traversal

102



Euler tour traversal of a tree.

Algorithm eulertour(T , p):

perform the “pre visit” action for position p

for each child c in $T.\text{children}(p)$ **do**

eulertour(T, c)

{recursively tour the subtree rooted at c}

perform the “post visit” action for position p

Euler Tour Traversal

103

```

1 class EulerTour:
2     ...
3
4     Abstract Base Class for performing Euler Tour of a tree.
5     _hook_previsit and _hook_postvisit may be overriden by subclasses.
6
7     ...
8
9     def __init__(self, tree):
10        '''Prepare an Euler Tour template for given tree.'''
11        self._tree = tree
12
13    def tree(self):
14        '''Return reference to the tree being traversed.'''
15        return self._tree
16
17    def execute(self):
18        '''Perform the tour and return any result from post visit of root.'''
19        if len(self._tree) > 0:
20            return self._tour(self._tree.root(),0,[])
21
22    def _tour(self, p, d, path):
23        '''Perform tour of subtree rooted at Position P.
24
25        p: Position of current node being visited
26        d: depth of p in the tree
27        path: list of indices of children on path from root to p.
28
29        ...
30
31        self._hook_previsit(p, d, path)          # "pre visit" p
32        results = []
33        path.append(0)                         # add new index to end of path before recursion
34        for c in self._tree.children(p):
35            results.append(self._tour(c, d+1, path)) # recur on child's subtree
36            path[-1] += 1                      # increment index
37        path.pop()
38        answer = self._hook_postvisit(p, d, path, results) # "post visit" p
39
40    def _hook_previsit(self, p, d, path):      # can be overridden
41        pass
42
43    def _hook_postvisit(self, p, d, path, results): # can be overridden
44        pass

```

Binary Search Tree – Biểu diễn

104

- Cấu trúc dữ liệu của CNPTK là cấu trúc dữ liệu biểu diễn cây nhị phân nói chung

```
struct TNode
{
    DataType data;
    TNode *left, *right;
};
typedef TNode* Tree;
Tree root ;
```

```
1 class Nut:
2     #Định nghĩa lớp nút
3     def __init__(self, khoa=None):
4         self.khoa = khoa
5         self.trai = None
6         self.phai = None
7     #def
```

- 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 – Thêm một nút vào cây

105

```
int InsertTree(Tree &root , int x)
{
    if(root != NULL)
    {
        if(root->data==x) return 0;
        if(root->data>x) return InsertTree(root->left,x);
        else return InsertTree(root->right,x);
    }
    else
    {
        root=new node;
        if(root ==NULL) return -1;
        root->data=x;
        root->left=root->right=NULL;
        return 1;
    }
}
```

Binary Search Tree – Tạo cây

106

- Ta có thể tạo một cây nhị phân tìm kiếm bằng cách lặp lại quá trình thêm 1 phần tử vào một cây rỗng.

```
void CreateTree(Tree &root)
```

```
{
```

```
    int x,n;
```

```
    printf("Nhập n = "); scanf("%d",&n);
```

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

```
{
```

```
        scanf("%d",&x);
```

```
        InsertTree(root,x);
```

```
}
```

```
8     def chen(self,khoa):
9         if self is None:
10             nut = Nut(khoa)
11             self = nut
12             return
13 #if Nút chưa được khởi tạo
14 #nút đã khởi tạo rồi,Nút khác None
15 if khoa<self.khoa:
16     if self.trai == None:
17         self.trai = Nut(khoa) #Nút trái chưa có giá trị
18     else: #Nút trái đã có giá trị
19         self.trai.chen(khoa)
20     #if
21 elif khoa > self.khoa:
22     if self.phai == None:
23         self.phai = Nut(khoa)
24     else:
25         #có nút bên phải rồi
26         self.phai.chen(khoa)
27     #if|
28 else:
29     #Không lớn hơn hay không nhỏ hơn, bị trùng khoá
30     print (f'Bị trùng khoá {khoa}')
31     #if
32 # def
33 #class Nut
```

Binary Search Tree – Thêm một nút vào cây

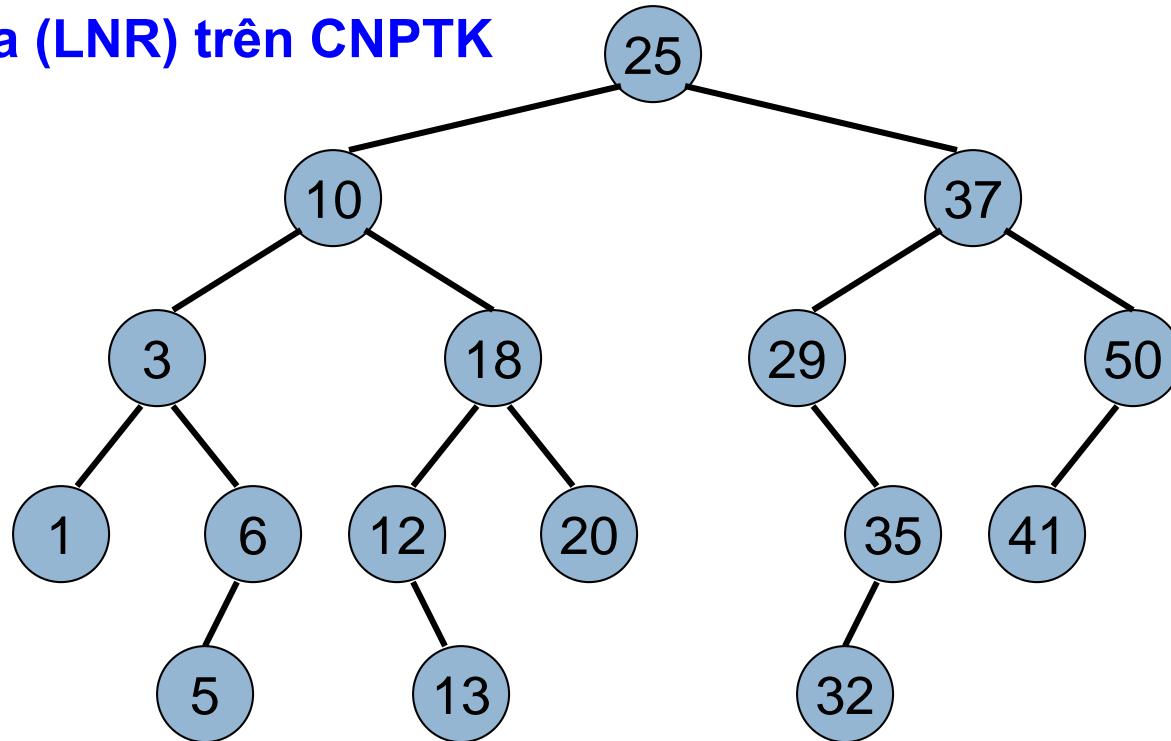
108

```
34 #Định nghĩa lớp cây nhị phân tìm kiếm
35 class CayNhiPhanTimKiem:
36     def __init__(self,khoa = None):
37         if khoa == None:#Không truyền vào tham số
38             self.goc = None
39         else:
40             self.goc = Nut(khoa)
41         #if
42     #def
43     #Chèn vào 1 giá trị khoá
44     def chen(self, khoa):
45         if self.goc == None:
46             self.goc = Nut(khoa)
47         else:#Có nút rồi
48             self.goc.chen(khoa)
49         #if
50     #def Chèn 1 nút vào cây
51     #Xoá 1 nút
```

Binary Search Tree – Duyệt cây

109

Duyệt giữa (LNR) 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

Duyệt sau (LNR) trên CNPTK

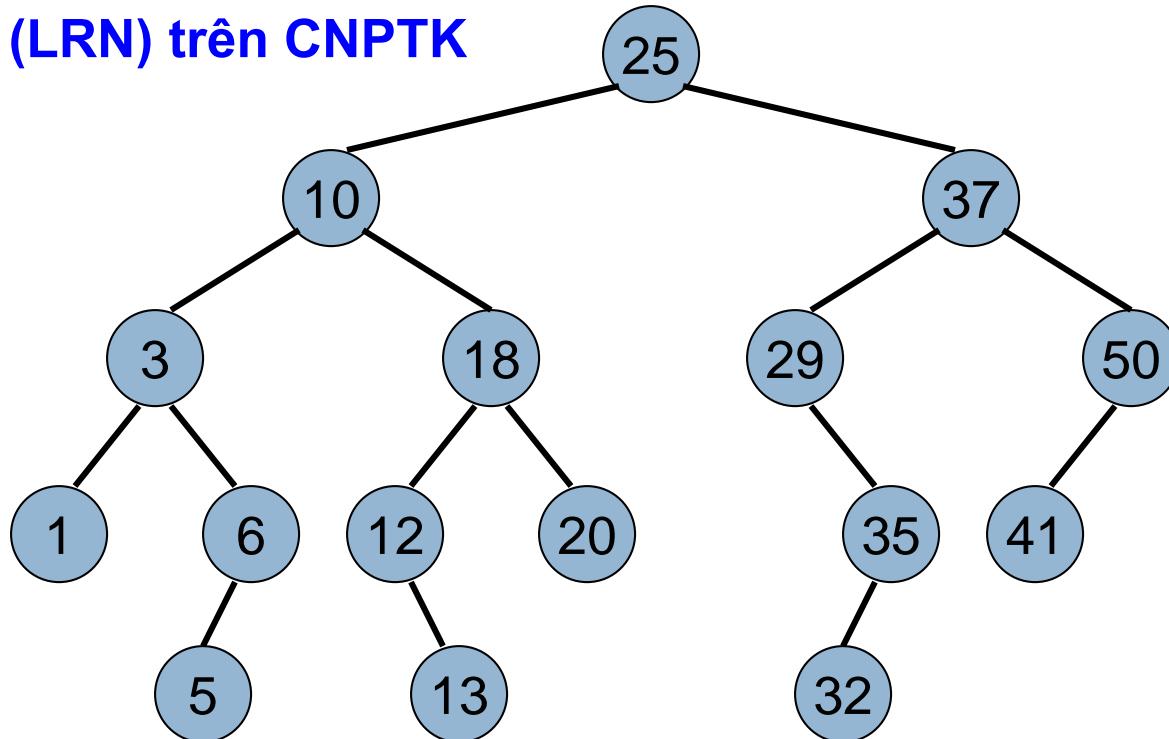
110

```
56     def duyet_trai_nut_phai(self,goc=0):
57         #Duyệt theo LNR
58         nut_ht = goc
59         if goc == None:
60             nut_ht = self.goc
61         #if
62         #kiểm tra nút hiện tại có bằng None không
63         if nut_ht == None:
64             return []
65         else: #cây có giá trị
66             kq = []
67             kq_trai = self.duyet_trai_nut_phai(nut_ht.trai)
68             for x in kq_trai:
69                 kq.append(x)
70                 #For duyệt trái
71                 kq.append(nut_ht.khoa)
72                 #Duyệt phải
73                 kq_phai = self.duyet_trai_nut_phai(nut_ht.phai)
74                 for x in kq_phai:
75                     kq.append(x)
76                     #for
77                     return kq
78             #if
79     #def
```

Binary Search Tree – Duyệt cây

111

Duyệt sau (LRN) trên CNPTK



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

Binary Search Tree – Duyệt cây

Duyệt sau (LRN) trên CNPTK

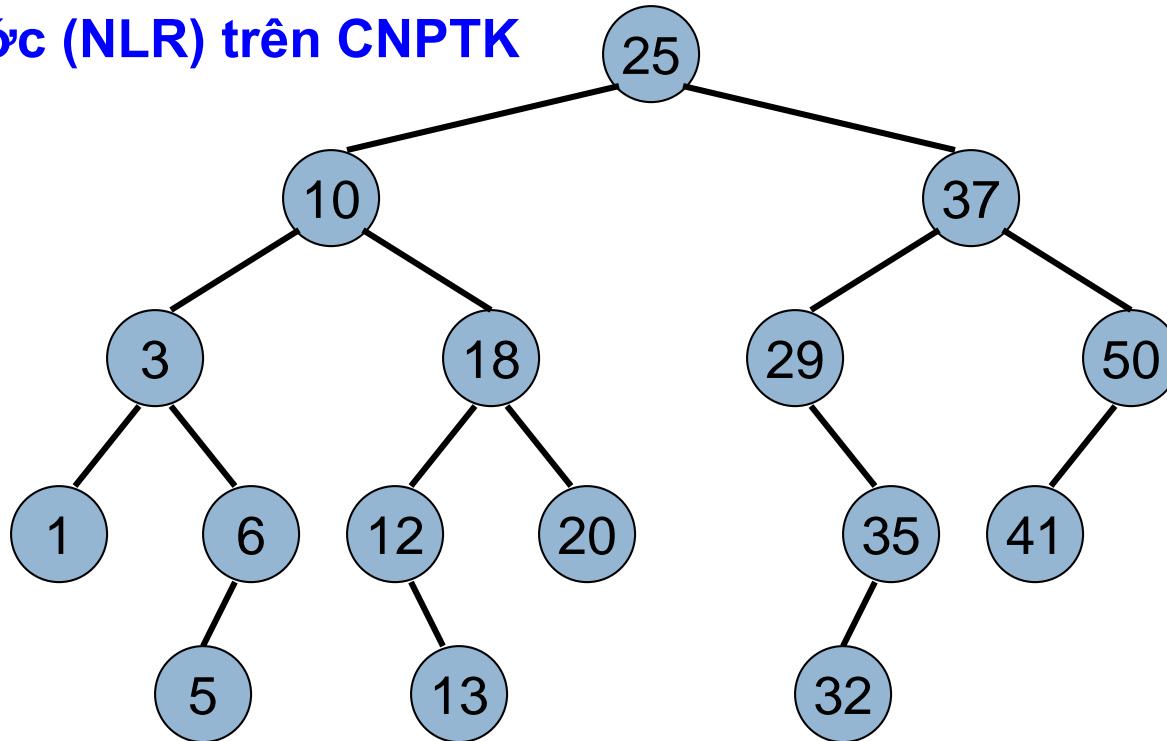
112

```
108     def duyet_trai_phai_nut(self,goc=0):
109         #Duyệt theo LRN
110         nut_ht = goc
111         if goc == 0:
112             nut_ht = self.goc
113             #if
114             #kiểm tra nút hiện tại có bằng None không
115             if nut_ht == None:
116                 return []
117             else: #cây có giá trị
118                 kq = []
119                 kq_trai = self.duyet_trai_phai_nut(nut_ht.trai)
120                 for x in kq_trai:
121                     kq.append(x)
122                     #For duyệt trái
123
124                     #Duyệt phải
125                     kq_phai = self.duyet_trai_phai_nut(nut_ht.phai)
126                     for x in kq_phai:
127                         kq.append(x)
128                         #Not
129                         kq.append(nut_ht.khoa)
130                         #for
131                         return kq
132             #if
133     #def
```

Binary Search Tree – Duyệt cây

113

Duyệt trước (NLR) trên CNPTK



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

Binary Search Tree – Duyệt cây

Duyệt sau (NLR) trên CNPTK

114

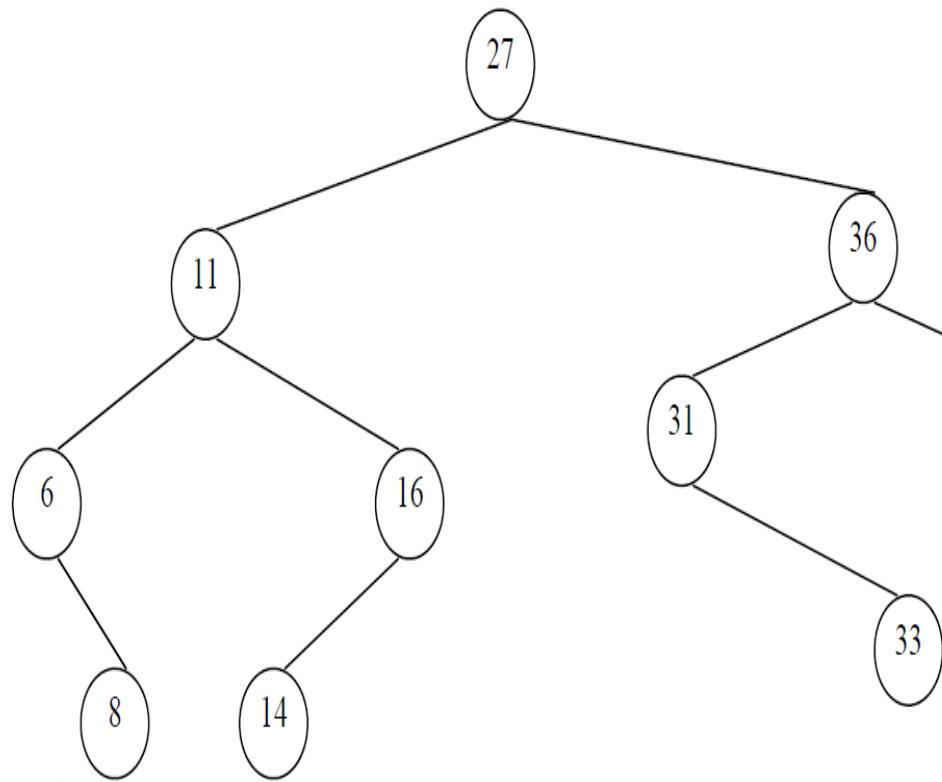
```
81      #Duyệt NLR
82      def duyet_nut_trai_phai(self,goc=0):
83          #Duyệt theo NLR
84          nut_ht = goc
85          if goc == None:
86              nut_ht = self.goc
87          #if
88          #kiểm tra nút hiện tại có bằng None không
89          if nut_ht == None:
90              return []
91          else: #cây có giá trị
92              kq = []
93              kq.append(nut_ht.khoa)
94              kq_trai = self.duyet_nut_trai_phai(nut_ht.trai)
95              for x in kq_trai:
96                  kq.append(x)
97                  #For duyệt trái
98
99                  #Duyệt phải
100                 kq_phai = self.duyet_nut_trai_phai(nut_ht.phai)
101                 for x in kq_phai:
102                     kq.append(x)
103                     #for
104                     return kq
105             #if
106             #def
```

Binary Search Tree – Duyệt cây

115

Bài tập

Kết quả của phép duyệt cây theo thứ tự NLR là:



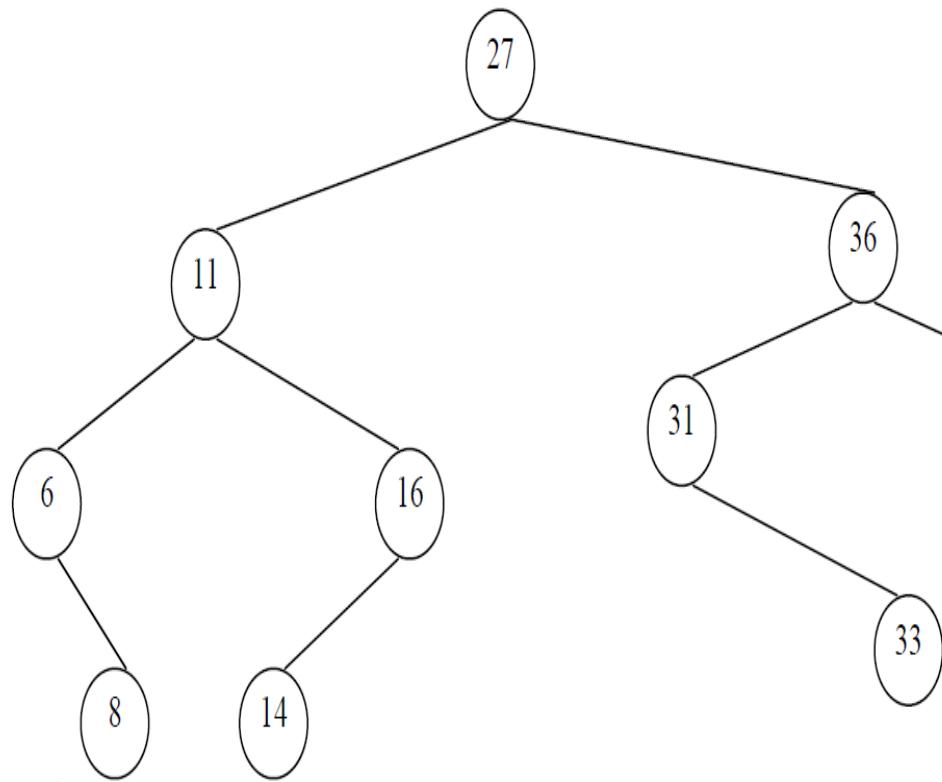
- A. 27,36,46,31,33,11,16,14,6,8
- B. 27,11,16,6,8,14,31,33,46
- C. 27,11,6,8,16,14,36,31, 46,33
- D. 27,11,6,8,16,14,36,31,33,46

Binary Search Tree – Duyệt cây

116

Bài tập

Kết quả của phép duyệt cây theo thứ tự LRN là:

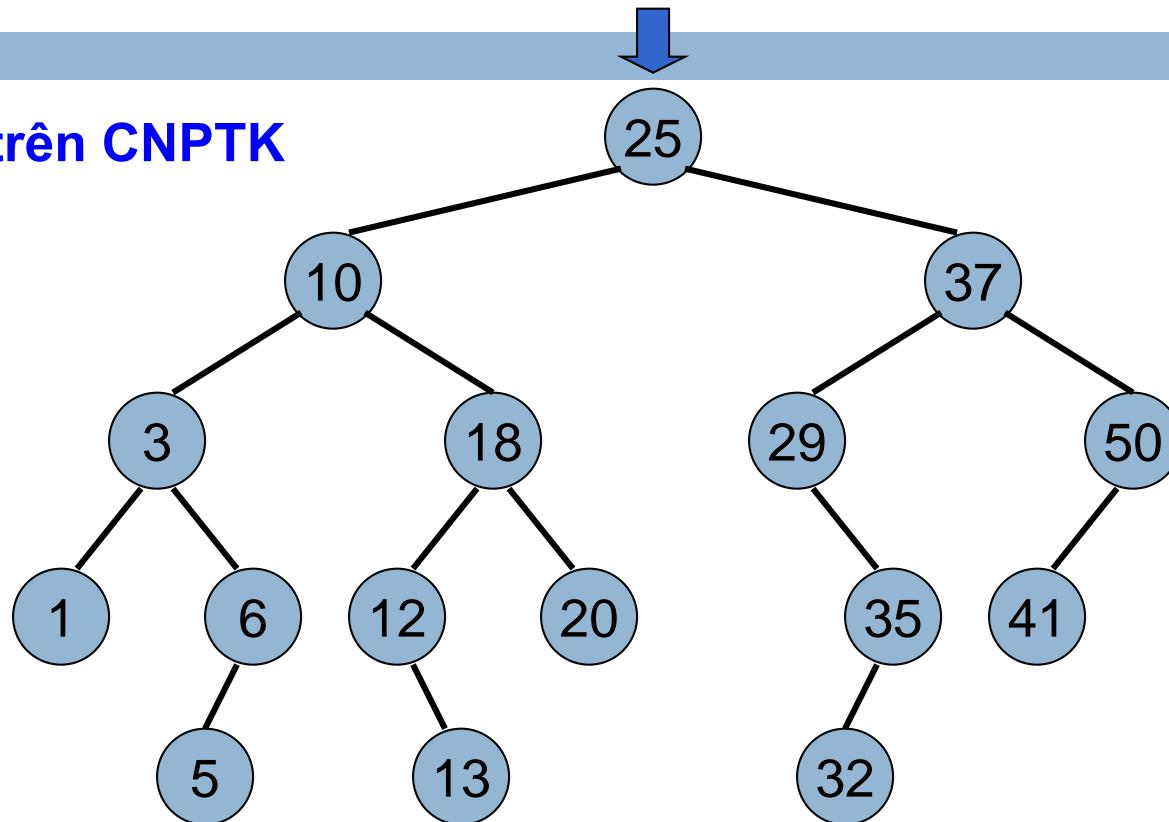


- A. 6,8,11,14,16,27,31,33,36,46
- B. 6,8,14,16,11,31,33,36,46,27
- C. 8,6,14,16,11,33,31,46,36,27
- D. 6,8,14,16,11,31,33,46,36,27

Binary Search Tree – Tìm kiếm

117

Tìm kiếm trên CNPTK



Tìm kiếm 13

Tìm thấy

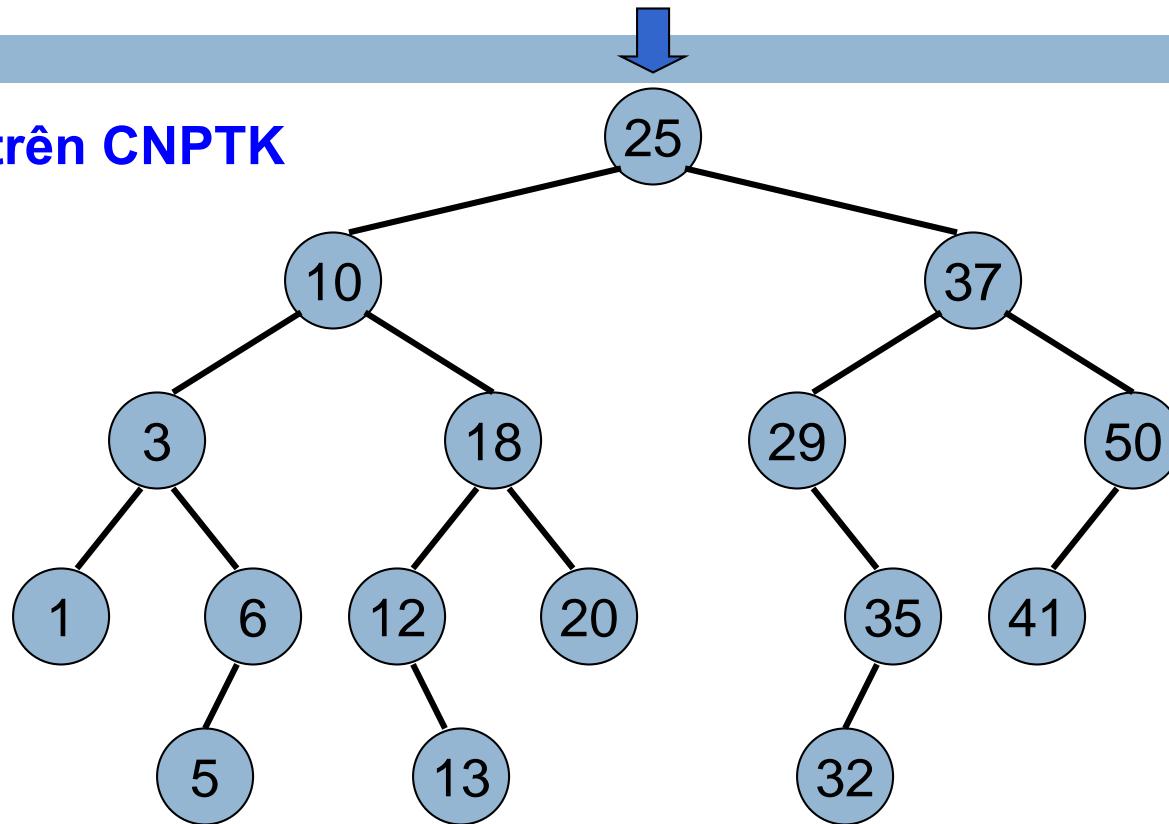
Số node duyệt: 5

Kết quả tìm kiếm
Nhập giá trị cần tìm

Binary Search Tree – Tìm kiếm

118

Tìm kiếm trên CNPTK



Tìm kiếm 14

Không tìm thấy

Số node duyệt: 5

Nháe gốc hổn loạn

Binary Search Tree – Tìm kiếm

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

119

```
134     #Tim
135     def tim(self,khoa):
136         if self.goc == None :
137             #Cây rỗng
138             return
139         #if
140         nut_ht =self.goc
141         kq = ''
142         while (nut_ht != None and nut_ht.khoa !=khoa):
143             kq = kq + f'{nut_ht.khoa} -> '
144             if khoa <=nut_ht.khoa:
145                 nut_ht = nut_ht.trai
146             else:
147                 nut_ht = nut_ht.phai
148             #if
149         #while
150         if nut_ht == None : #Tìm không thấy
151             return None
152         else:#Tìm thấy
153             kq =kq +f'{nut_ht.khoa}'
154             return kq
155         #if
156     #def
157 #class
```

Binary Search Tree – Tìm kiếm

120

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

```
TNode* searchNode(Tree root, DataType x)
{
    if (root)
    {
        if (root->data == x)
            return root;
        if (root->data > x)
            return searchNode(root->left, x);
        return searchNode(root->right, x);
    }
    return NULL;
}
```

Binary Search Tree – Tìm kiếm

121

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

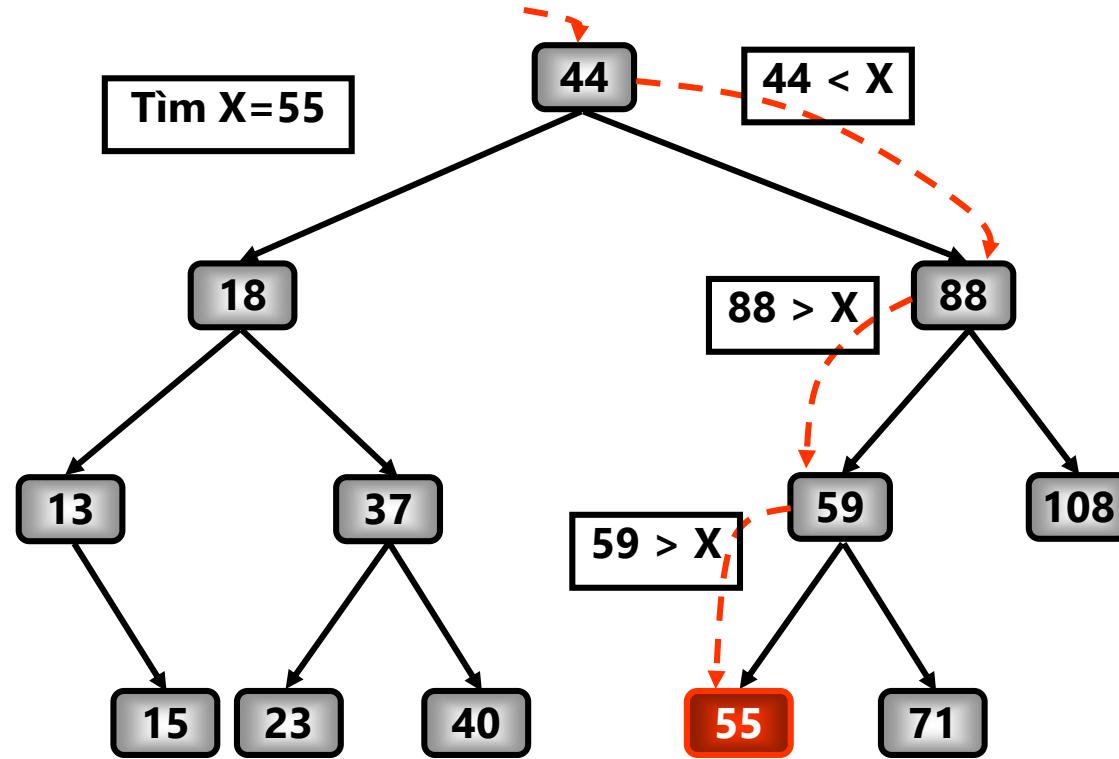
```
TNode* searchNode(Tree root, DataType x)
{
    TNode *p = root;
    while (p != NULL)
    {
        if (x == p->data)    return p;
        else if (x < p->data) p = p->left;
        else p = p->right;
    }
    return NULL;
}
```

Binary Search Tree – Tìm kiếm

122

- 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 tồn chí phí trung bình khoảng $O(\log_2 n)$

Tìm một phần tử x trong cây



Binary Search Tree – Thêm

124

- 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
- Cách thực hiện:
 - Tìm vị trí thêm (???)
 - Thực hiện thêm (???)

Binary Search Tree – Thêm

125

- 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
- Cách thực hiện:
 - Tìm vị trí thêm (???)
 - Thực hiện thêm (???)

Binary Search Tree – Thêm

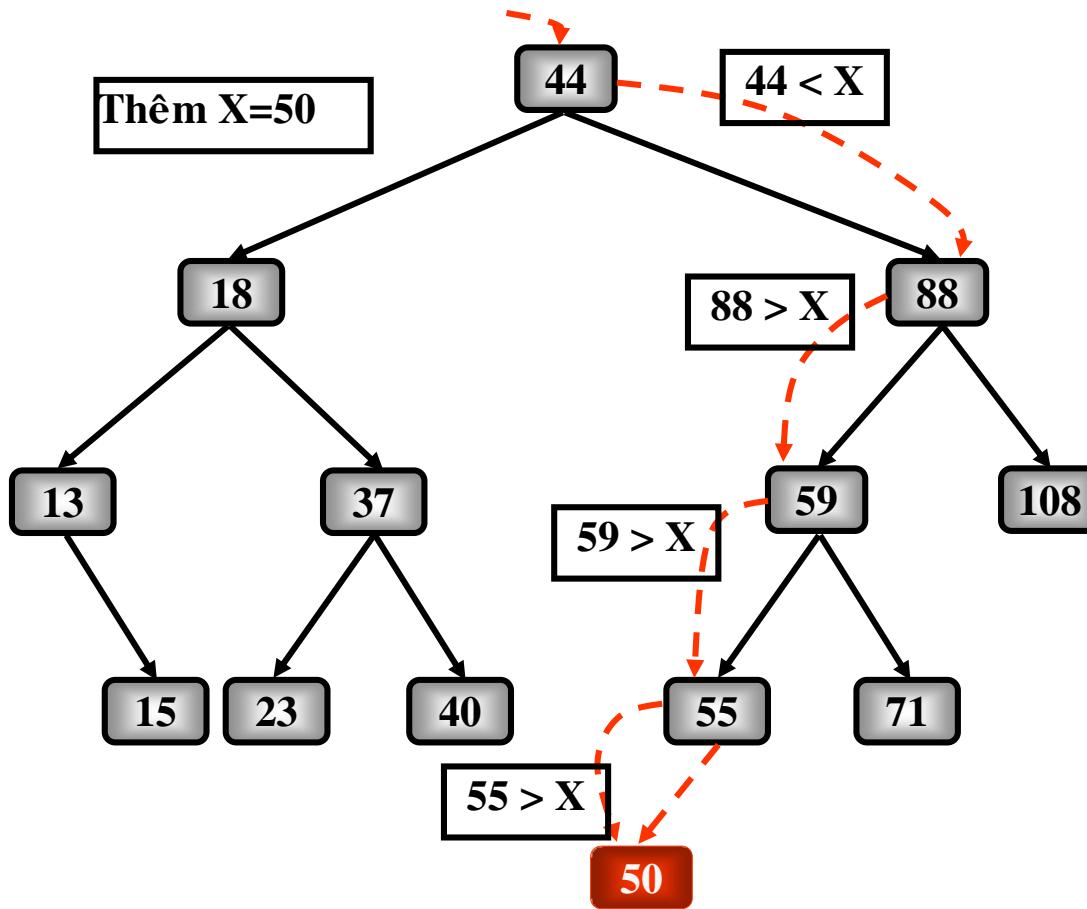
126

- Thêm một phần tử vào cây:

```
int insertNode (Tree &root, DataType X){  
    if (root) {  
        if (root->data == X) return 0;  
        if (root->data > X)  
            return insertNode(root->left, X);  
        else  
            return insertNode(root->right, X);  
    }  
    root = new TNode;  
    if (root == NULL) { cout<<“Khong du bo nho”; return -1; }  
    root->data = X;  
    root->left = root->right = NULL;  
    return 1;  
}
```

–1 : khi không đủ bộ nhớ
0 : khi nút đã có
1 : khi thêm thành công

Thêm một phần tử x vào cây

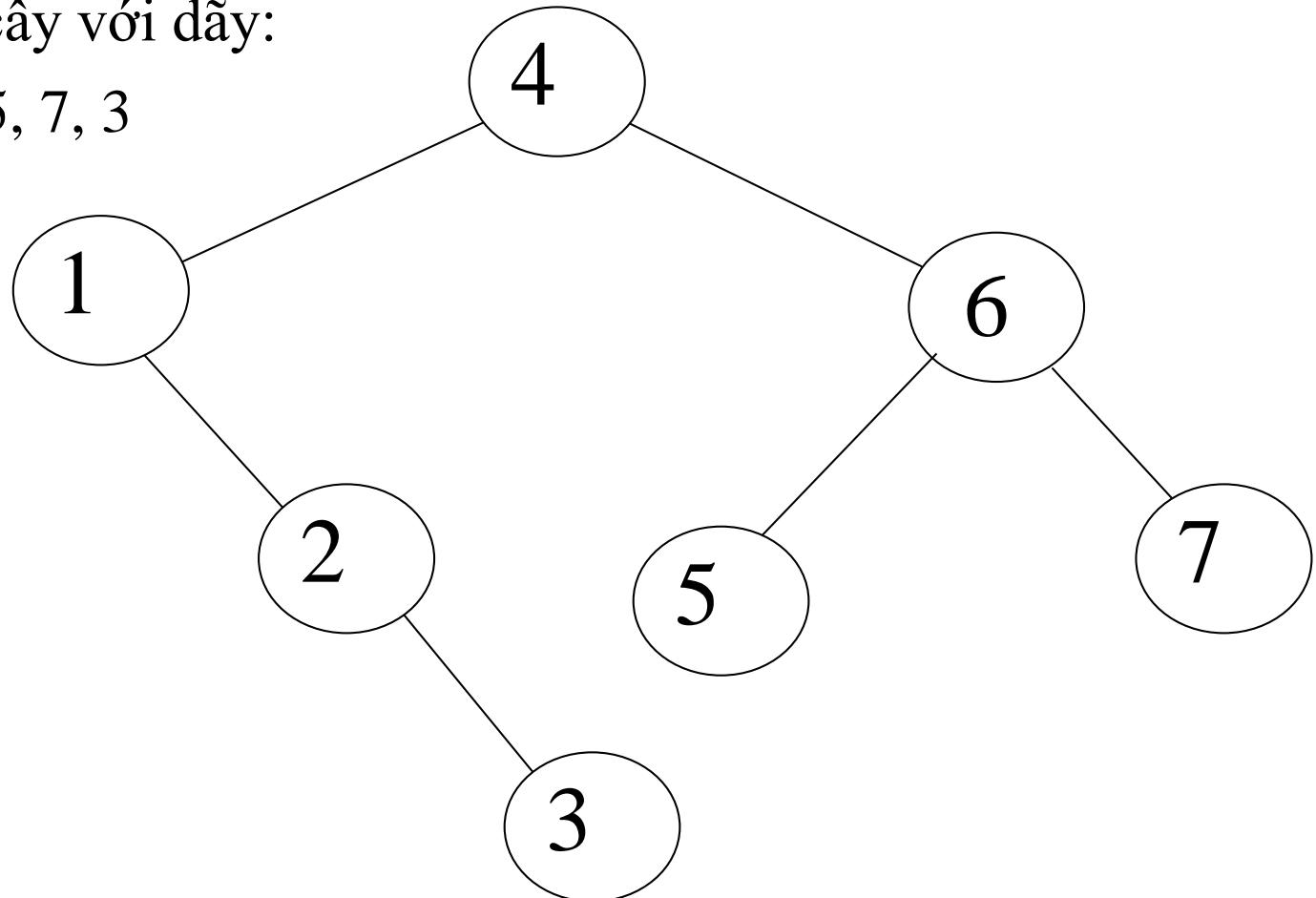


Binary Search Tree – Thêm

128

- Ví dụ tạo cây với dãy:

4, 6, 1, 2, 5, 7, 3

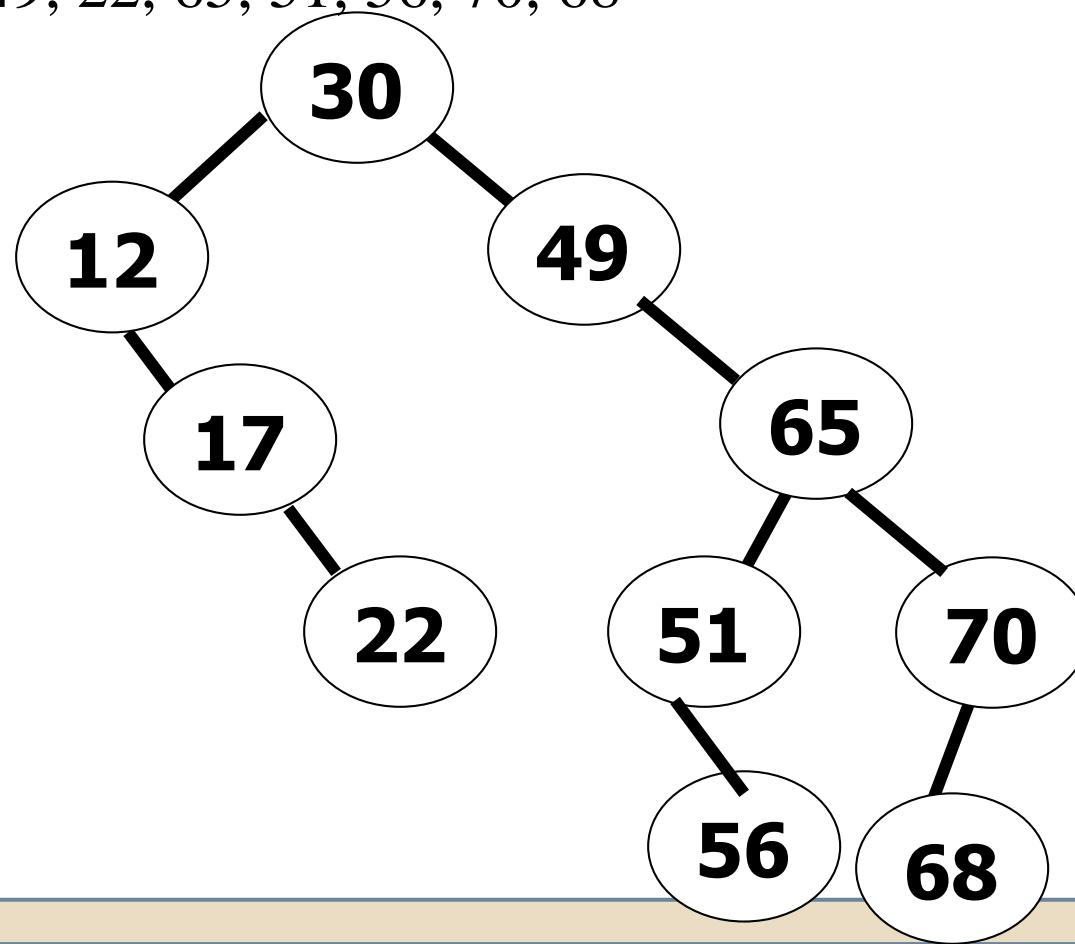


Binary Search Tree – Thêm

129

- 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

130

- 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

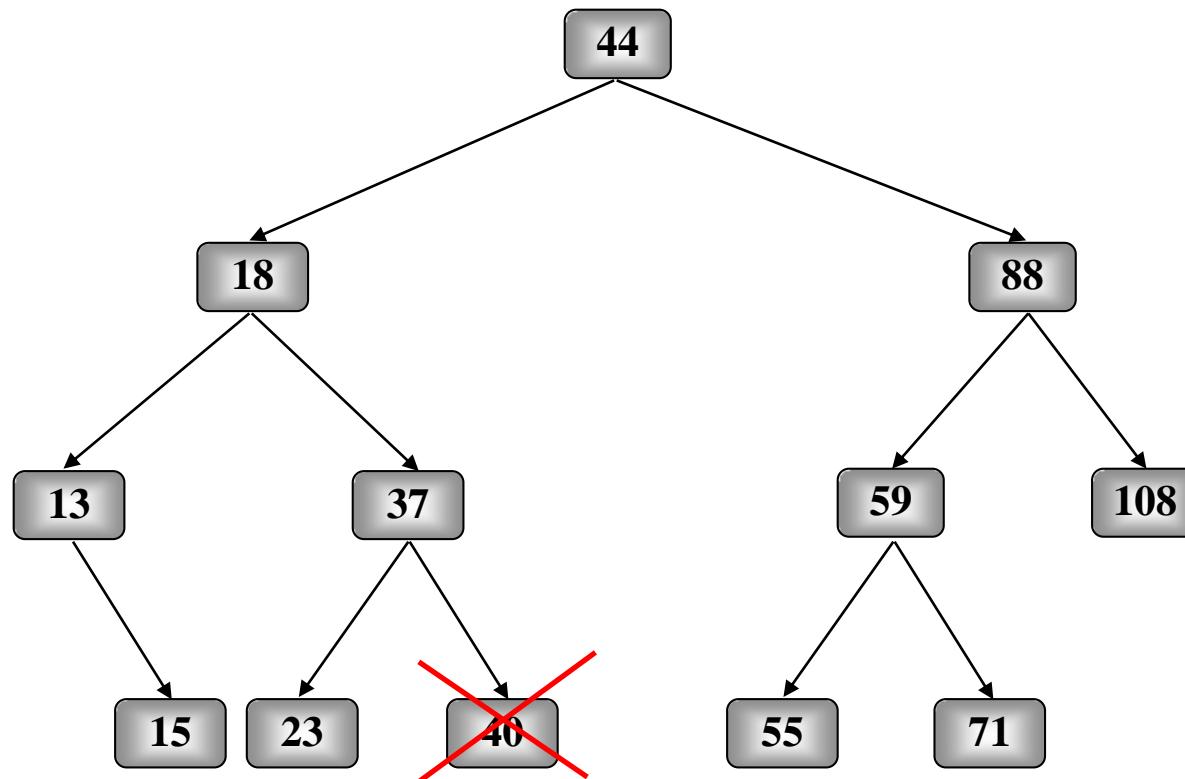
Binary Search Tree – Hủy một phần tử có khóa X

131

□ Trường hợp 1: X là nút lá

- Chỉ đơn giản hủy X vì nó không móc nối đến phần tử nào khác

Hủy X=40

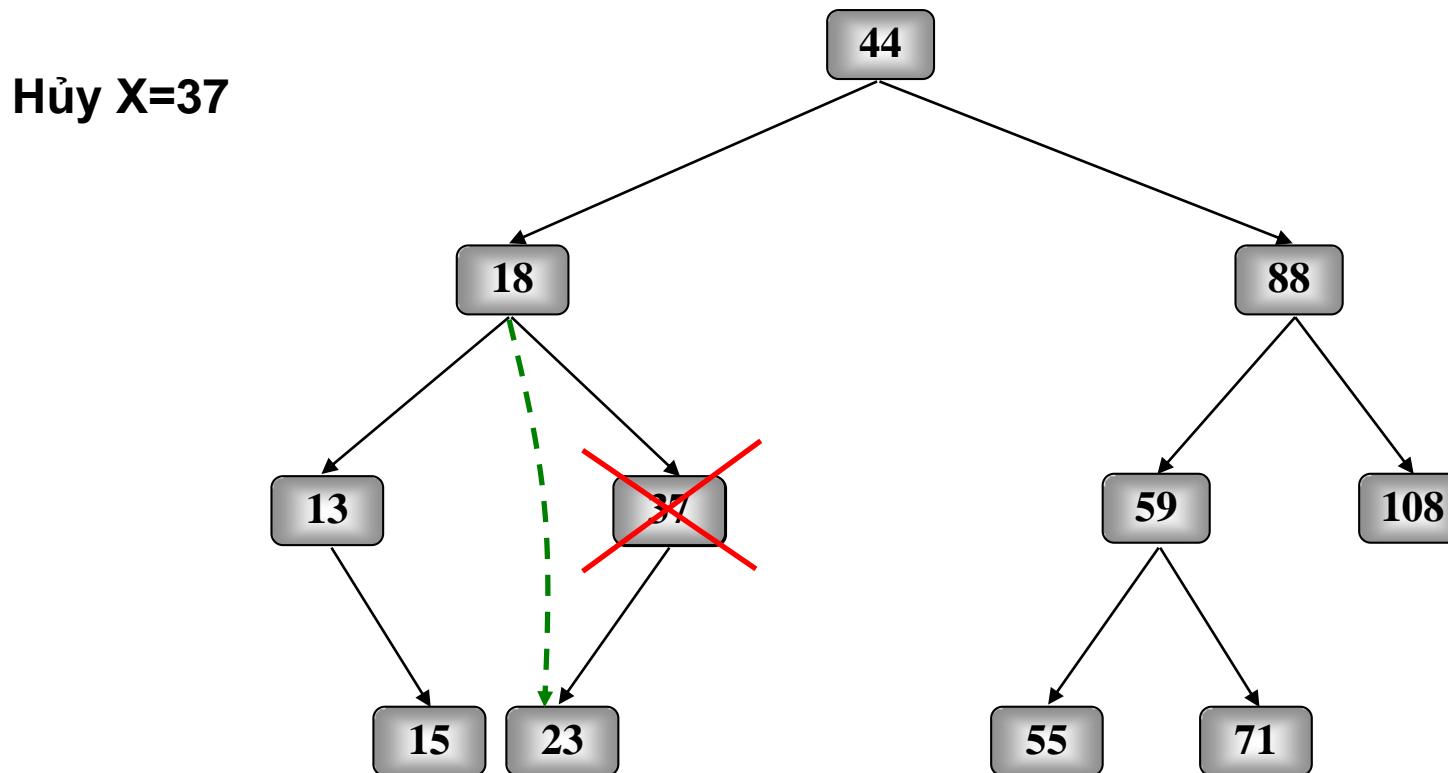


Binary Search Tree – Hủy một phần tử có khóa X

132

□ 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

133

- 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

134

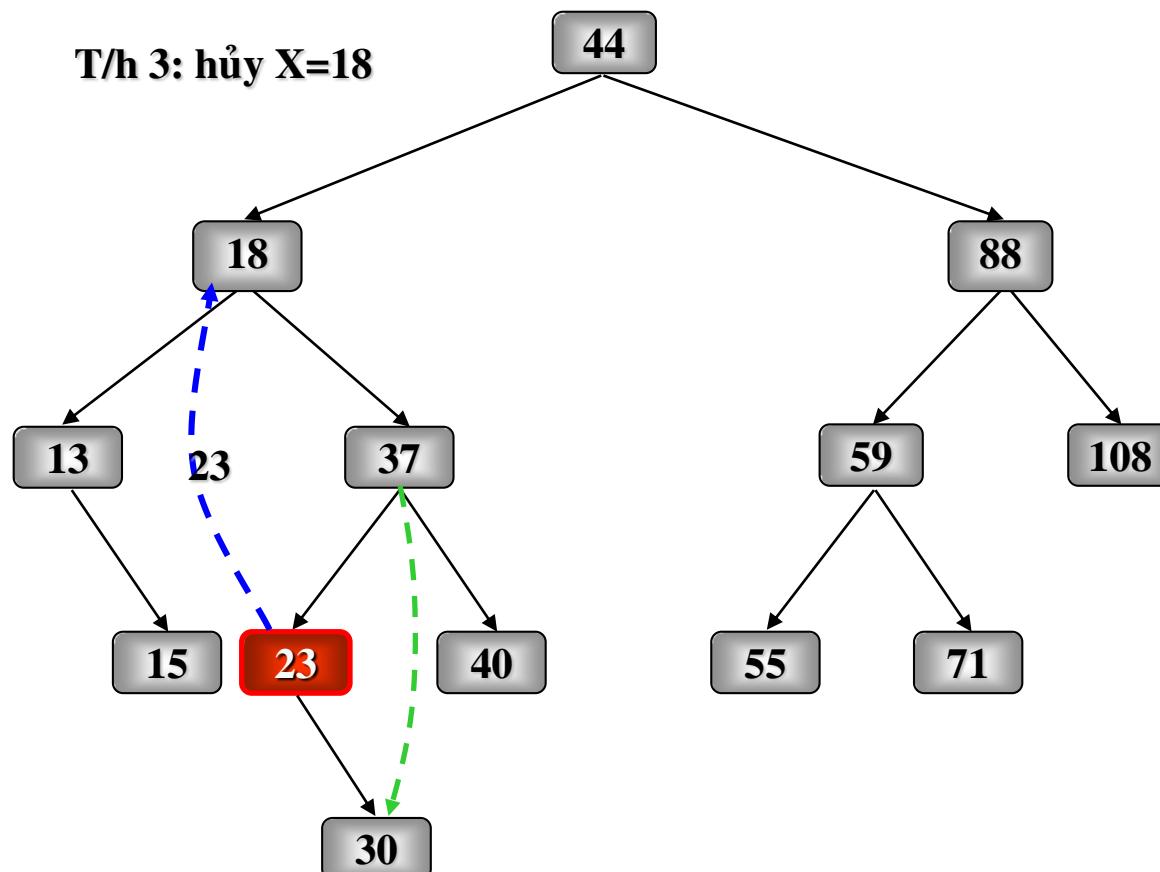
- Cách chọn phần tử thế mạng:
 - Phần tử **nhỏ nhất** (trái nhất) **trên cây con phải** (của nút muốn xóa)
 - Phần tử **lớn nhất** (phải nhất) **trên cây con trái** (của nút muốn xóa)
- Việc chọn lựa phần tử nào là phần tử thế mạng phụ thuộc vào ý thích của người lập trình
- Ở đây, ta sẽ chọn phần tử *nhỏ nhất trên cây con phải* làm phần tử thế mạng

Binary Search Tree – Hủy một phần tử có khóa X

135

- Trường hợp 3: X có đủ 2 con:

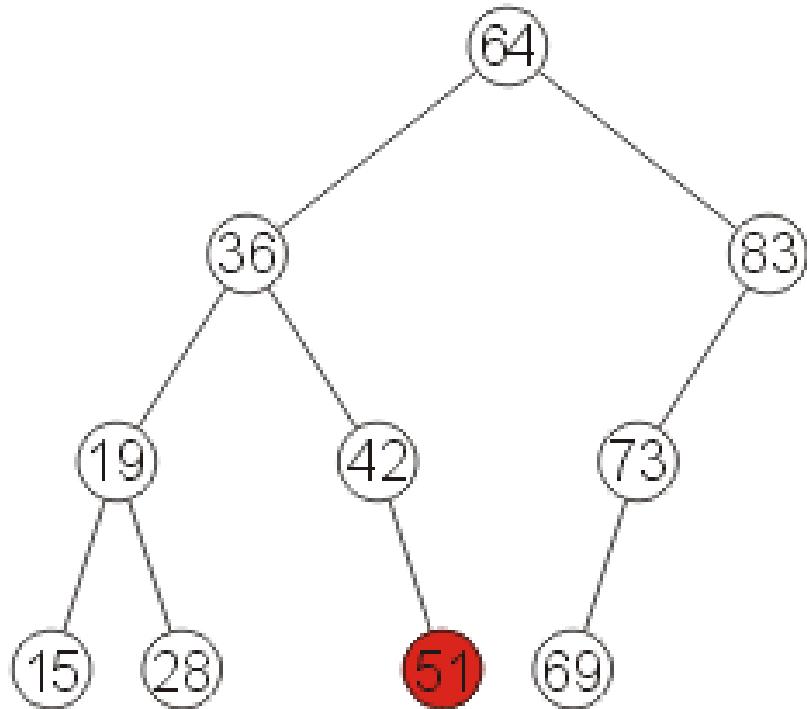
- Khi hủy phần tử X=18 ra khỏi cây, phần tử 23 là phần tử thế mạng:



Binary Search Tree – Hủy một phần tử có khóa X

136

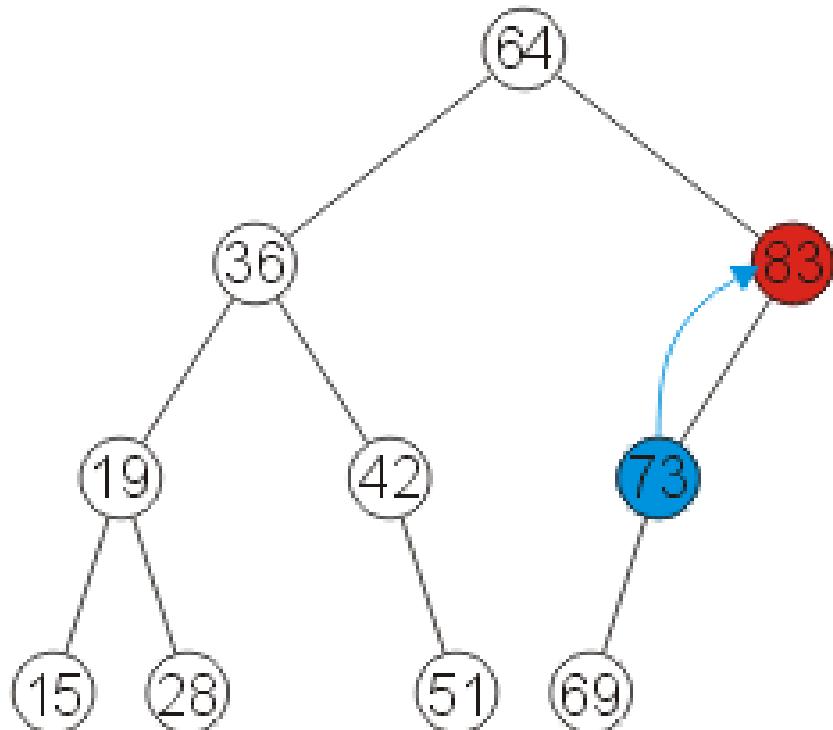
- Xóa 51



Binary Search Tree – Hủy một phần tử có khóa X

137

- Xóa 83

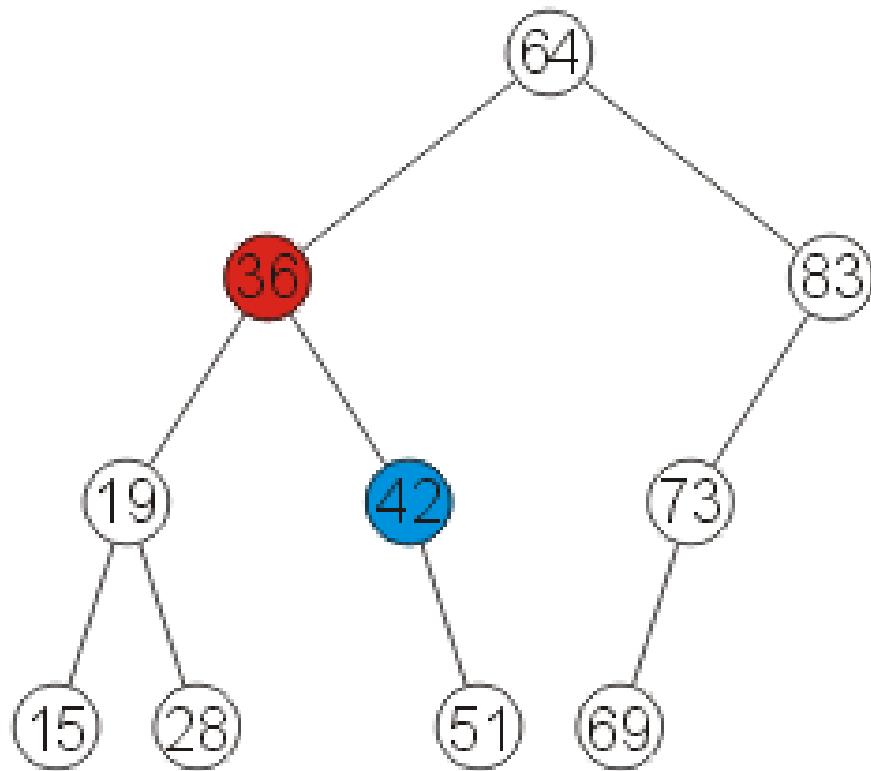


1

Binary Search Tree – Hủy một phần tử có khóa X

138

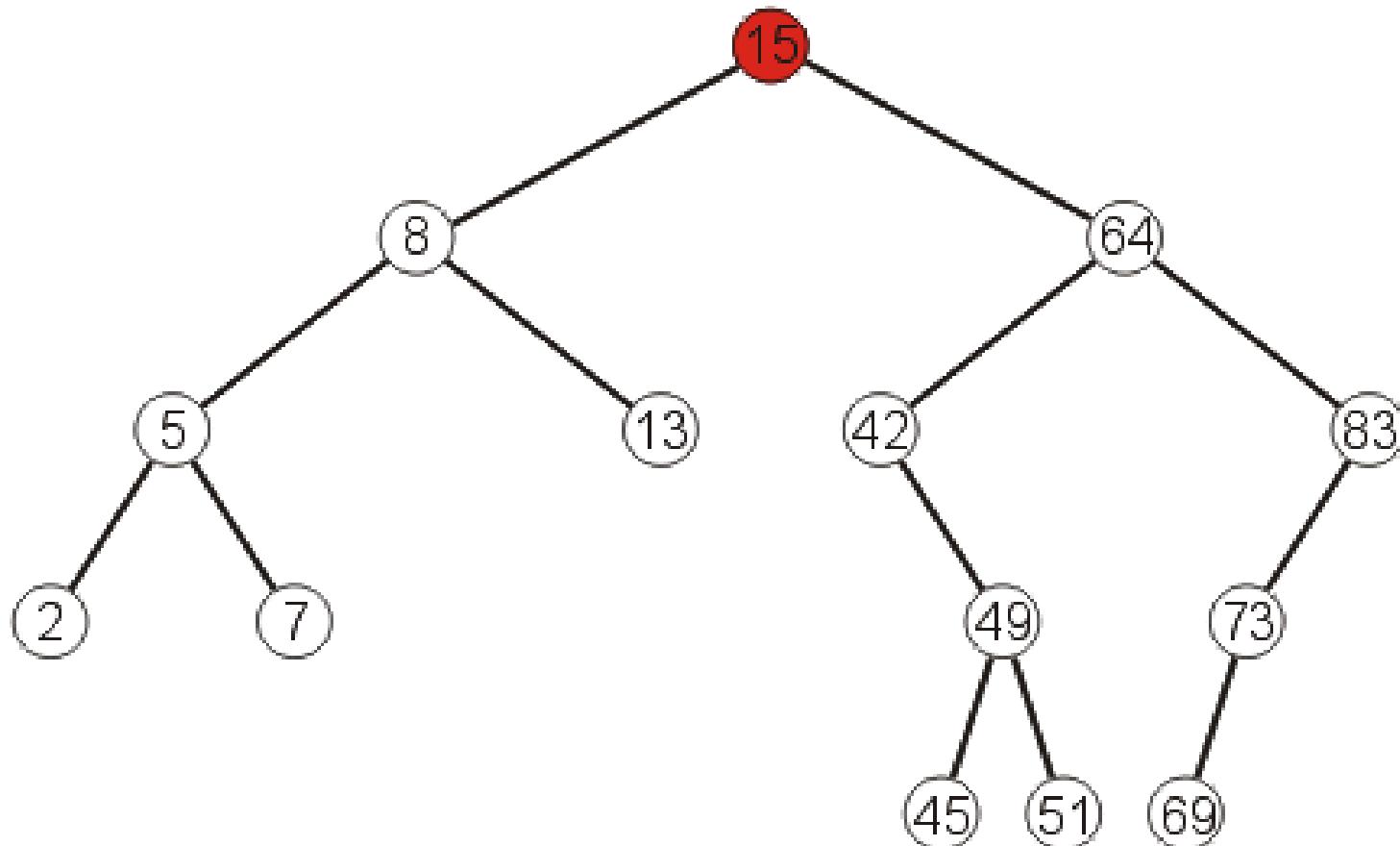
- Xóa 36



Binary Search Tree – Hủy một phần tử có khóa X

139

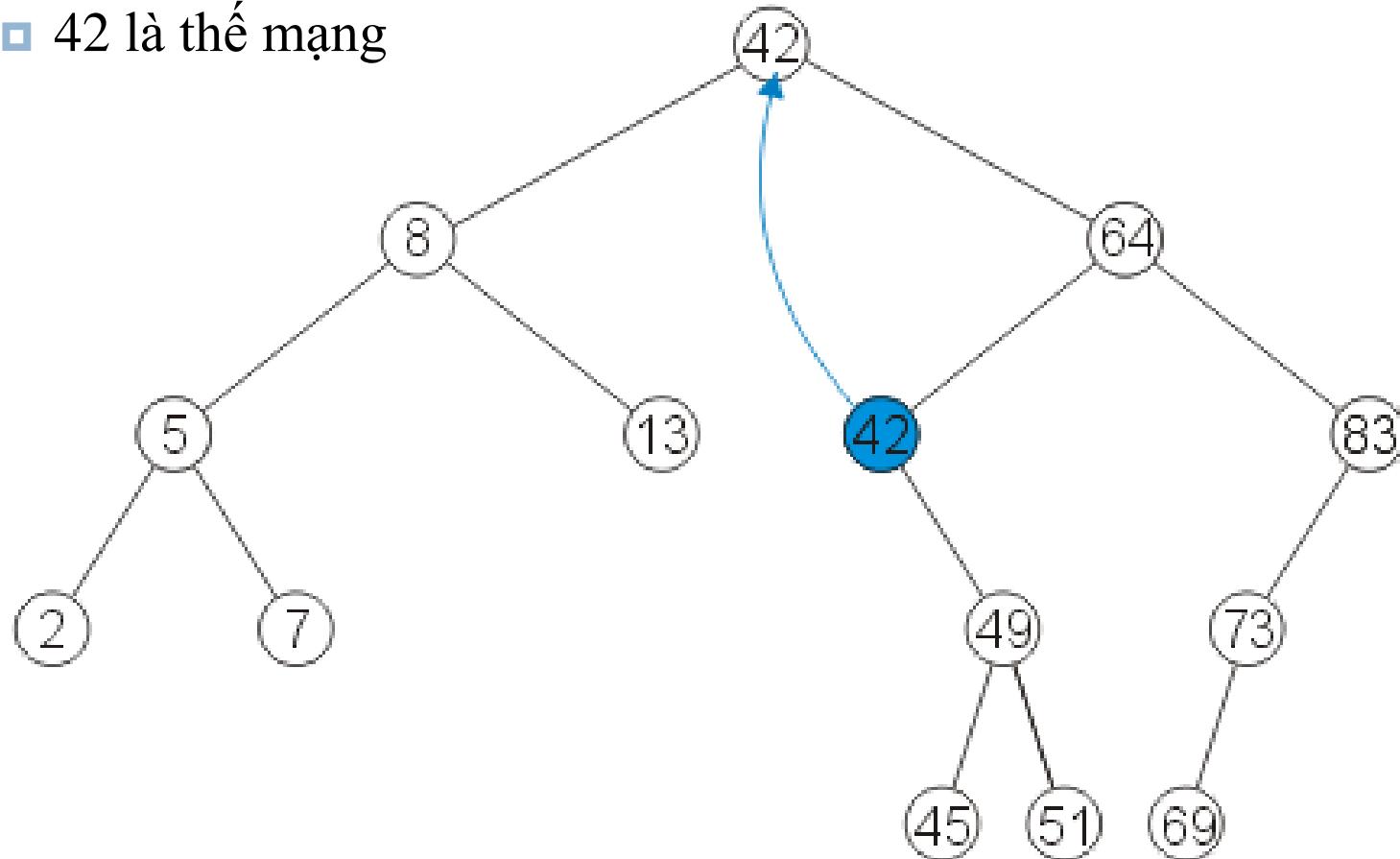
- Xóa nút gốc:



Binary Search Tree – Hủy một phần tử có khóa X

140

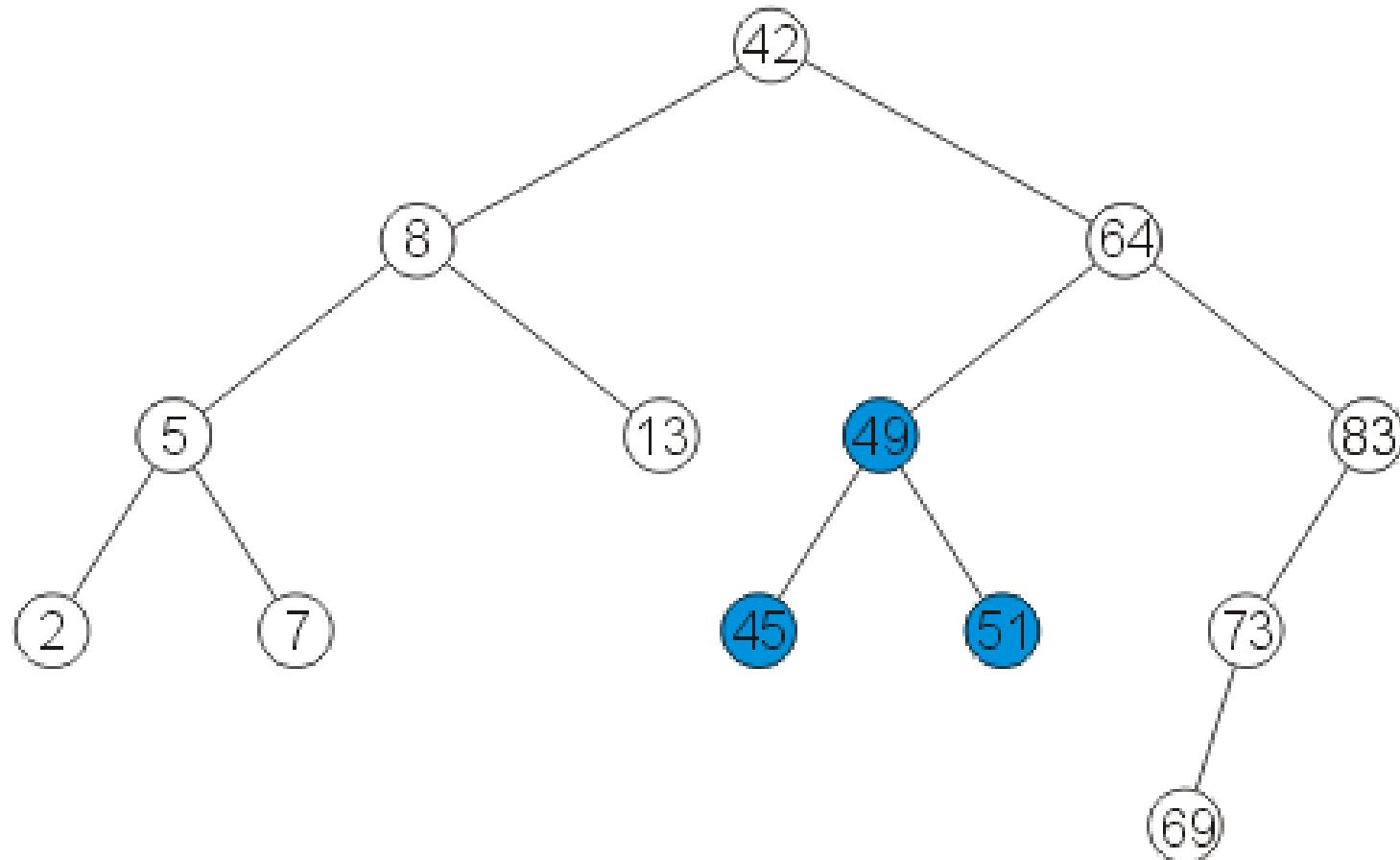
- Xóa nút gốc:
 - 42 là thẻ mạng



Binary Search Tree – Hủy một phần tử có khóa X

141

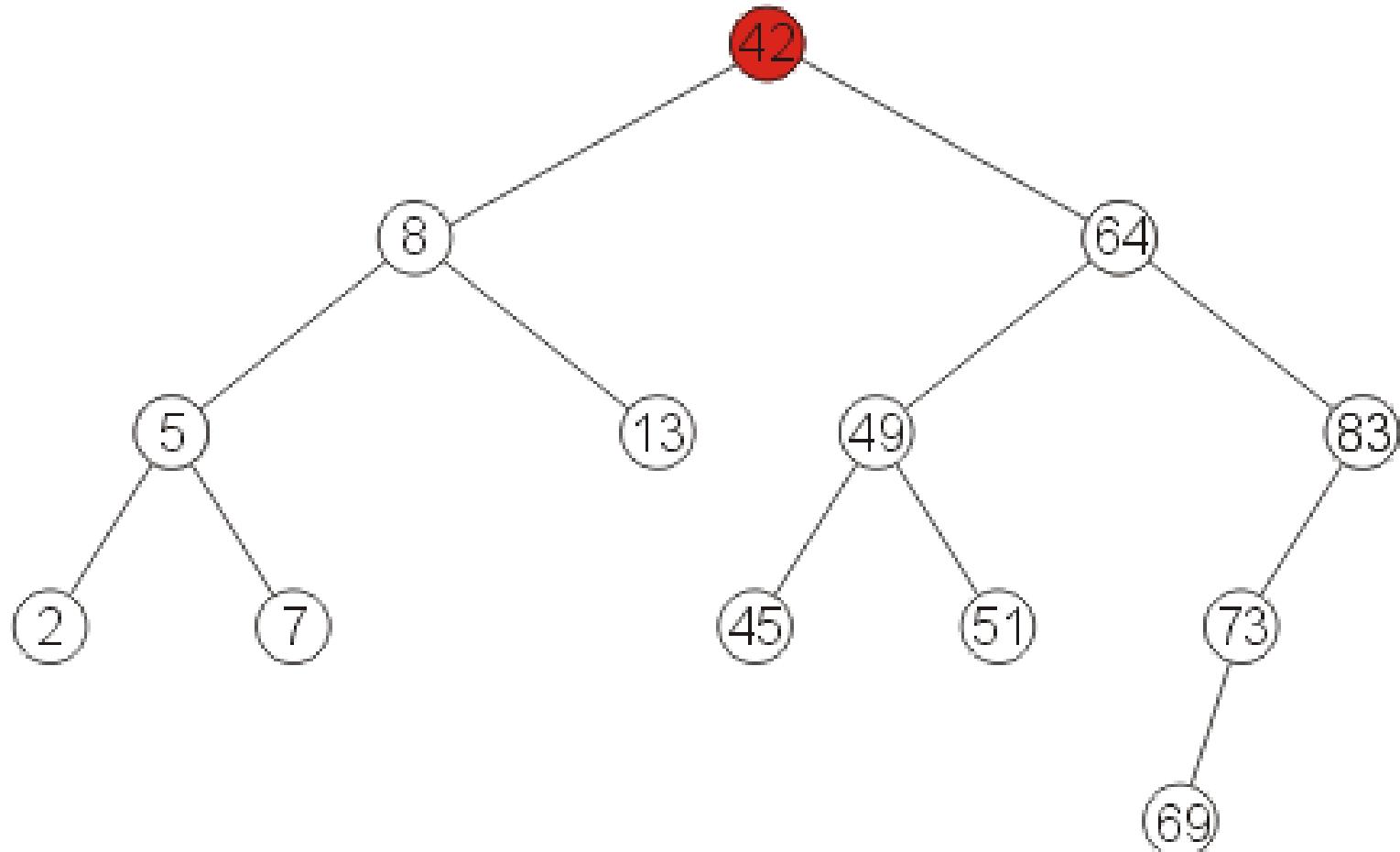
- Kết quả xóa:



Binary Search Tree – Hủy một phần tử có khóa X

142

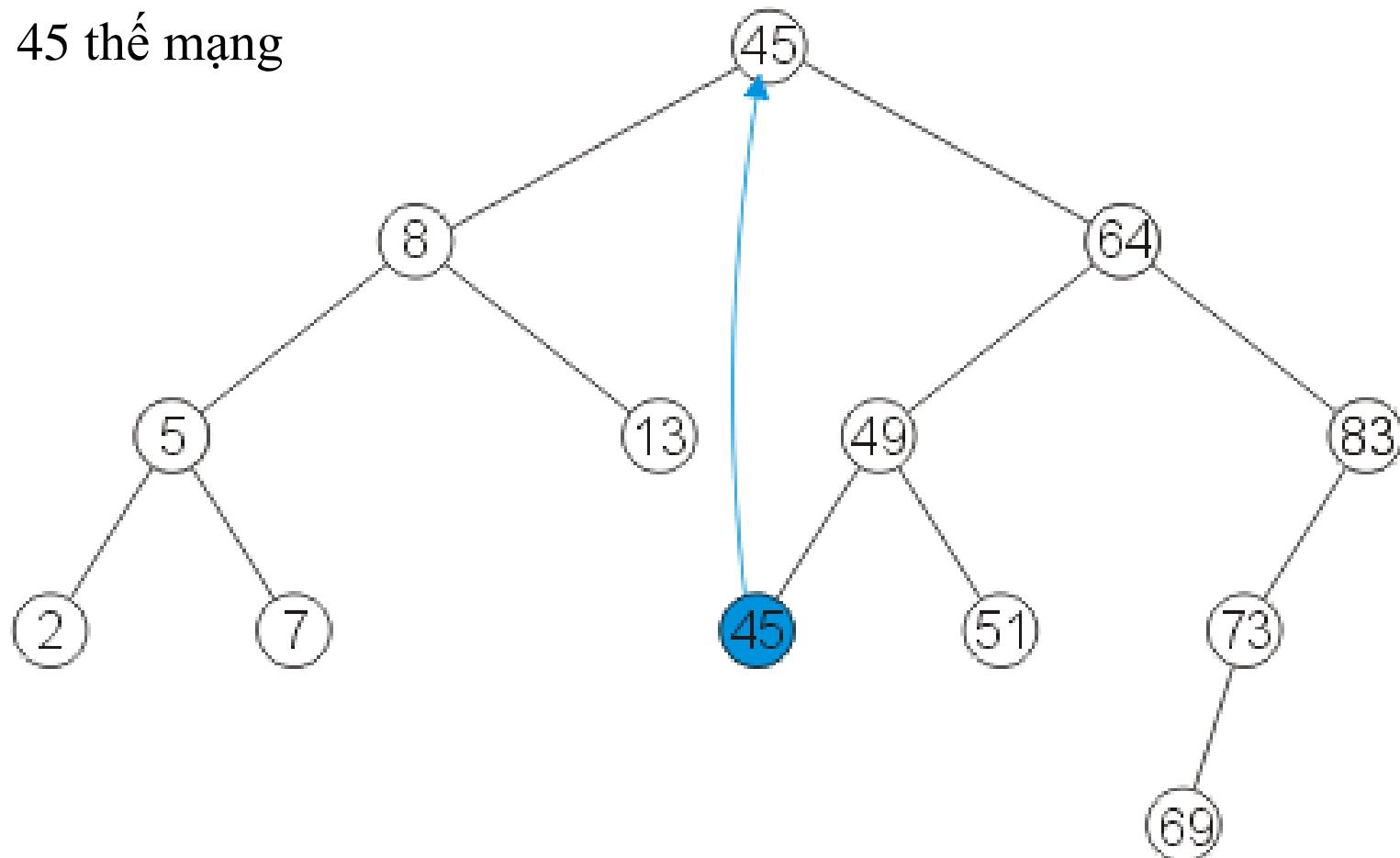
- Xóa gốc 42



Binary Search Tree – Hủy một phần tử có khóa X

143

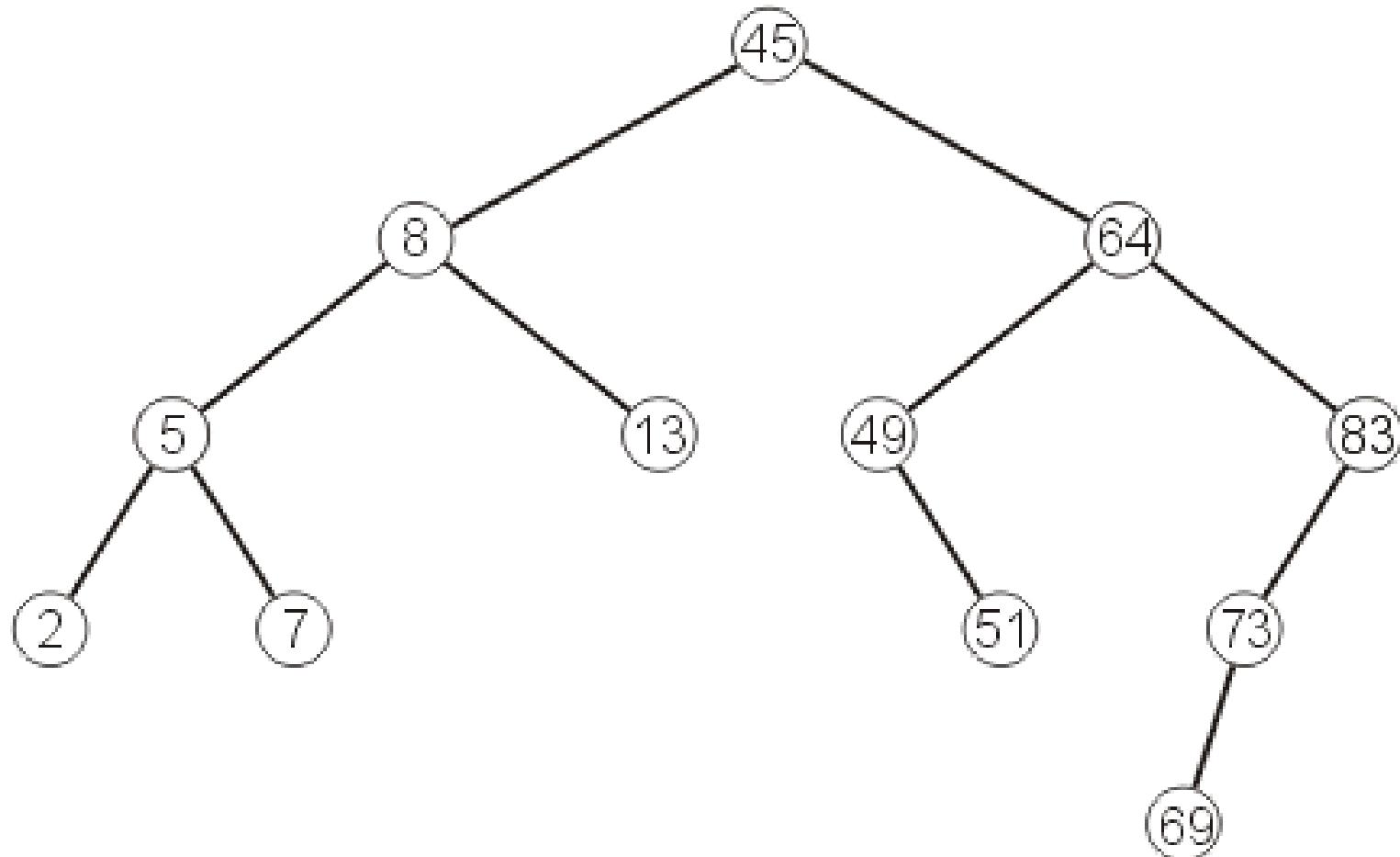
- Xóa gốc 42
 - 45 thê mạng



Binary Search Tree – Hủy một phần tử có khóa X

144

- ☐ Kết quả xóa:



Binary Search Tree – Hủy một phần tử có khóa X

145

□ Các hàm dùng để hủy 1 phần tử:

- 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 q và gán dữ liệu của q cho nút muốn xóa p

void **searchStandFor**(Tree &p, Tree &q)

Binary Search Tree – Hủy một phần tử có khóa X

146

- Hủy một nút

```
int delNode(Tree &root, DataType X)
{
    if (root == NULL)      return 0;
    if (root->data > X)   return delNode(root->left, X);
    if (root->data < X)   return delNode(root->right, X);

    TNode* p = root;
    if (root->left == NULL) root = root->right;
    else if (root->right == NULL) root = root->left;
        else // T có đủ 2 con
            searchStandFor(p, root->right);

    delete p;
}
```

Binary Search Tree – Hủy một phần tử có khóa X

147

- Tìm phần tử thế mạng (*nhỏ nhất trên cây con phải*):

```
void searchStandFor(Tree &p, Tree &q)
{
    if (q->left != NULL)
        searchStandFor (p, q->left);
    else
    {
        p->data = q->data;
        p = q;
        q = q->right;
    }
}
```

Binary Search Tree – Hủy toàn bộ cây

148

- Việc 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 &root)
{
    if (root)
    {
        removeTree(root->left);
        removeTree(root->right);
        delete(root);
    }
}
```

Phương thức Xoá nút bằng Python

149

```
51     #Xoá 1 nút
52     def xoa(self,khoa):
53         nut_cha = None
54         cha_con = None
55         nut_ht = self.goc
56         #tìm nút xoá,
57         #Các trường hợp xoá nút lá, xoá nút có 1 con trái, xoá nút có 1 con phải,
58         # xoá nút có cả 2 con, xoá nút gốc
59         while nut_ht != None:
60             if nut_ht.khoa > khoa:#khoá xoá nhỏ hơn
61                 nut_cha = nut_ht
62                 nut_ht = nut_ht.trai #tìm nhánh bên trái
63                 cha_con ='trai'
64             elif nut_ht.khoa < khoa:
65                 nut_cha = nut_ht
66                 nut_ht = nut_ht.phai
67                 cha_con ='phai'
68             else: #bằng, tìm thấy nghĩa là xoá nút này
69                 if nut_cha == None: #Nút gốc
70                     #xoá nút gốc
71                     #Nếu nút gốc không có 2 con
72                     if nut_ht.trai == None and nut_ht.phai == None:
73                         #xoá nút gốc mà không có con
74                         self.goc = None
75                         #if
76                         elif nut_ht.trai == None:
77                             #Nút trái không có con, xoá nút gốc chỉ có 1 nút con bên phải
78                             self.goc = nut_ht.phai
```

Phương thức Xoá nút bằng Python

```
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
    elif nut_ht.phai == None:
        #xoá nút chỉ có 1 con bên trái
        self.goc = nut_ht.trai
    else:
        #xoá nút gốc có đủ 2 con
        #xoay trái
        self.goc = nut_ht.phai
        tam = self.goc
        while tam.trai !=None:
            #Truy tìm đến cực trái để gắn nhánh trái xuống bên trái của nút cực trái
            |   tam = tam.trai
            #while
            tam.trai = nut_ht.trai
        #if
    elif nut_ht.trai == None and nut_ht.phai ==None:
        #Không phải nút gốc. Xoá nút lá, không có con trái và phải
        if cha_con == 'trai':
            nut_cha.trai = None
        else:
            nut_cha.phai = None
        #if
    elif nut_ht.trai == None:
        #Không phải nút lá mà là nút giữa.
        # Xoá nút chỉ có 1 con bên phai
        if cha_con == 'trai':
            nut_cha.trai = nut_ht.phai
```

Phương thức Xoá nút bằng Python

151

```
105             else:
106                 nut_cha.phai = nut_ht.phai
107             #if
108         elif nut_ht.phai == None:
109             #xoá nút giữa chỉ có 1 con bên trái
110             if cha_con == 'trai':
111                 nut_cha.trai = nut_ht.trai
112             else:
113                 nut_cha.phai = nut_ht.trai
114         else:
115             #xoá nút có đủ 2 con
116             #xoay trái
117             if cha_con == 'trai':
118                 nut_cha.trai = nut_ht.phai
119             else:
120                 nut_cha.phai = nut_ht.phai
121             #if
122             if nut_ht.phai.trai == None:
123                 nut_ht.phai.trai = nut_ht.trai
124             else:#nút chưa là None, truy tìm nút tận cùng bên trái
125                 tam = nut_ht.phai
126                 while tam.trai != None:
127                     tam = tam.trai
128                     #while
129                     tam.trai = nut_ht.trai
130             #if
131             #if
```

Phương thức Xoá nút bằng Python

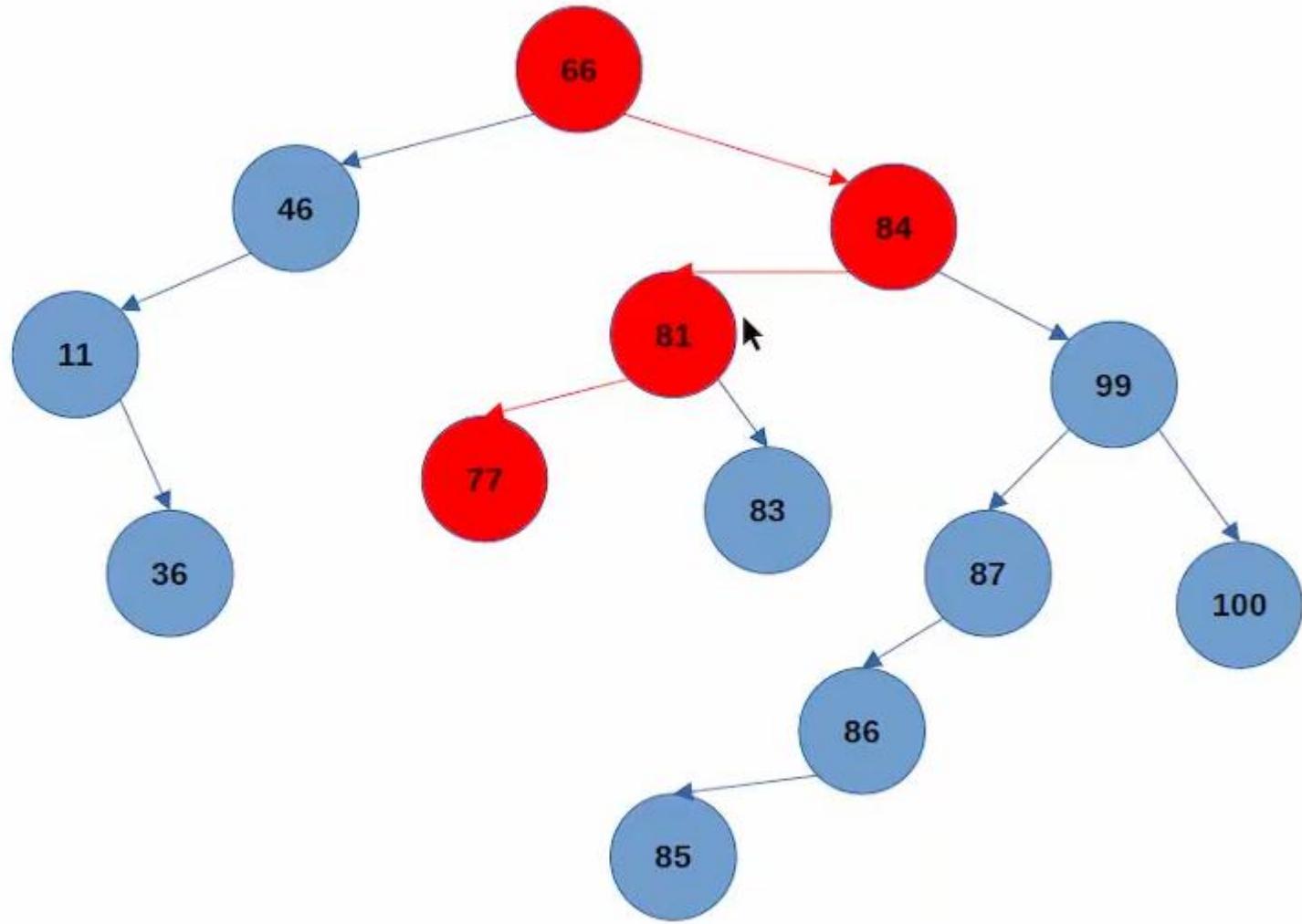
152

```
131      #if  
132      del nut_ht  
133      break  
134      #if  
135      #while  
136      #def
```

Binary Search Tree – Hủy một phần tử có khóa X

153

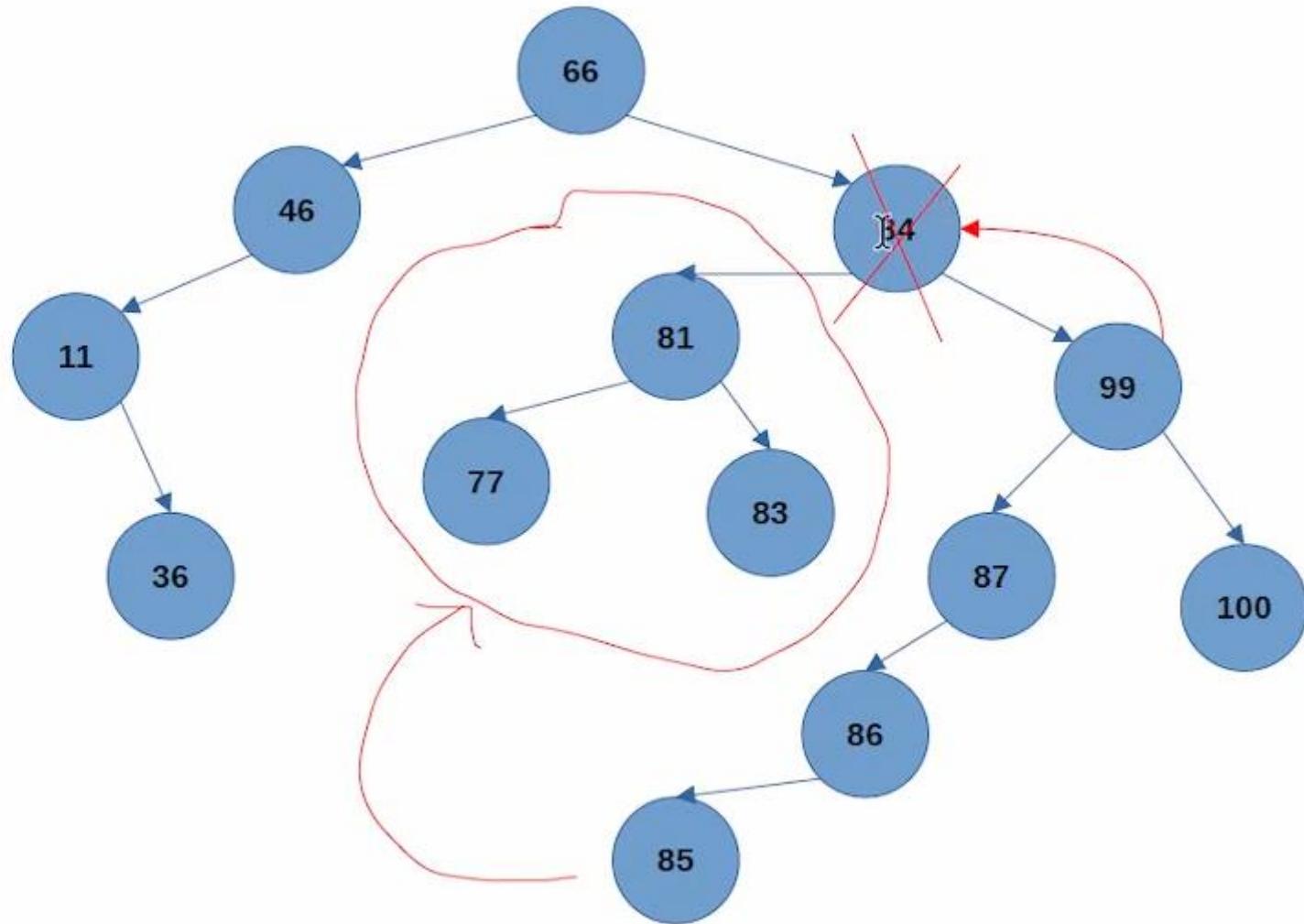
Tìm 77



Binary Search Tree – Hủy một phần tử có khóa X

154

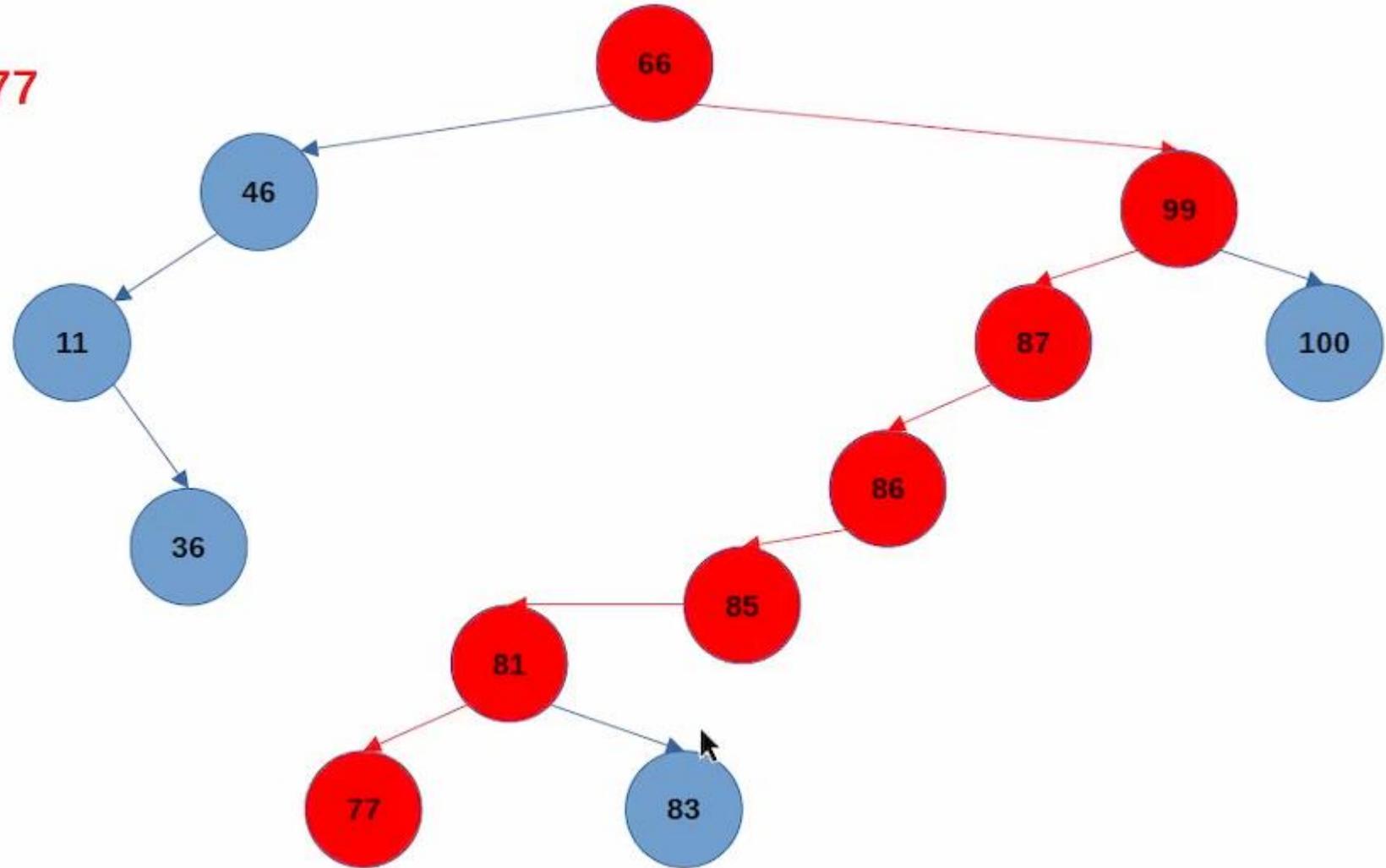
Xóa 84



Binary Search Tree – Hủy một phần tử có khóa X

155

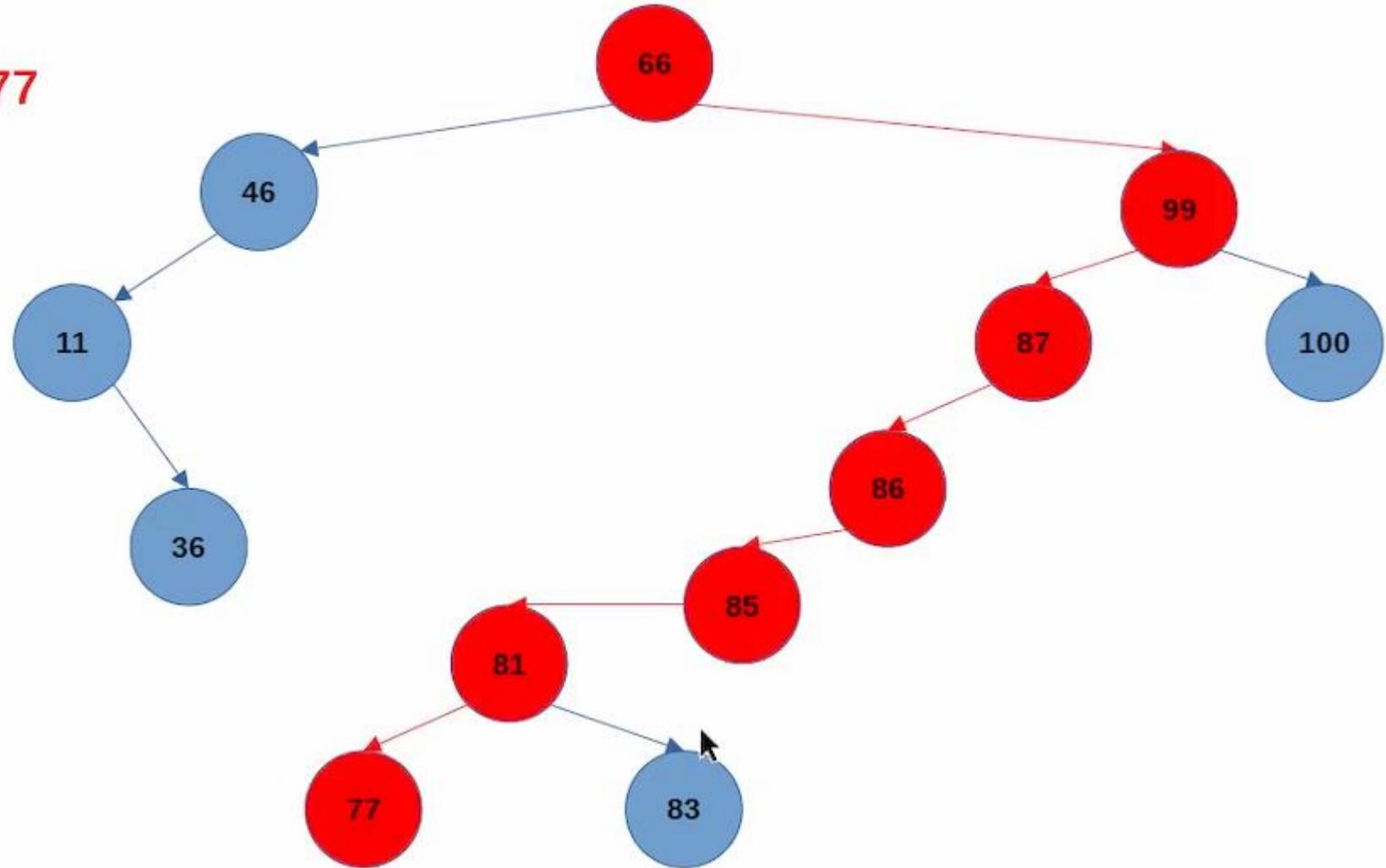
Tìm 77



Binary Search Tree – Hủy một phần tử có khóa X

156

Tìm 77



Binary Search Tree – Hàm Main()

15

```
240 #Hàm main
241 def main():
242     SO_PHAN_TU = 10
243     cay = CayNhiPhanTimKiem()
244     print('*****Chèn vào cây*****')
245     tap_gia_tri = set()
246     from random import randint
247     while len(tap_gia_tri) < SO_PHAN_TU:
248         gt = randint (1,100) #Lấy 10 phần tử không trùng nhau nên dùng tập hợp
249         tap_gia_tri.add(gt)
250     #while
251     #tap_gia_tri = list(tap_gia_tri) #Phát sinh danh sách ngẫu nhiên
252     tap_gia_tri = [66,46,84,11,81, 99,36,77,83, 87,100, 86, 85]
253     print('Chèn lần lượt', tap_gia_tri)
254     for x in tap_gia_tri:
255         cay.chen(x)
256     kq = cay.duyet_trai_nut_phai()
257     print('*****Duyệt cây theo Trai - Nut - Phai (LNR):',kq)
258
259     kq =cay.duyet_nut_trai_phai()
260     print('*****Duyệt cây theo Nut - Trai -Phai (NLR):',kq)
261
262     kq =cay.duyet_trai_phai_nut()
263     print('*****Duyệt cây theo Trai -Phai - Nut (LRN):',kq)
```

Chú

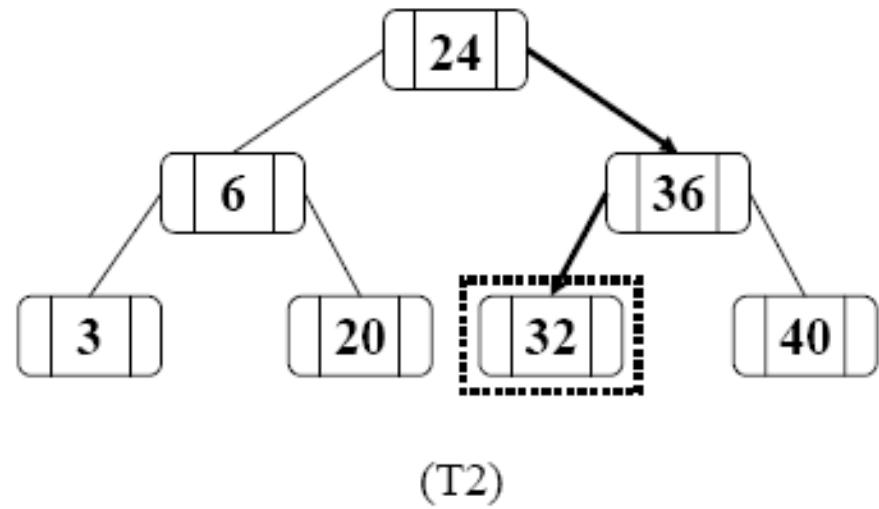
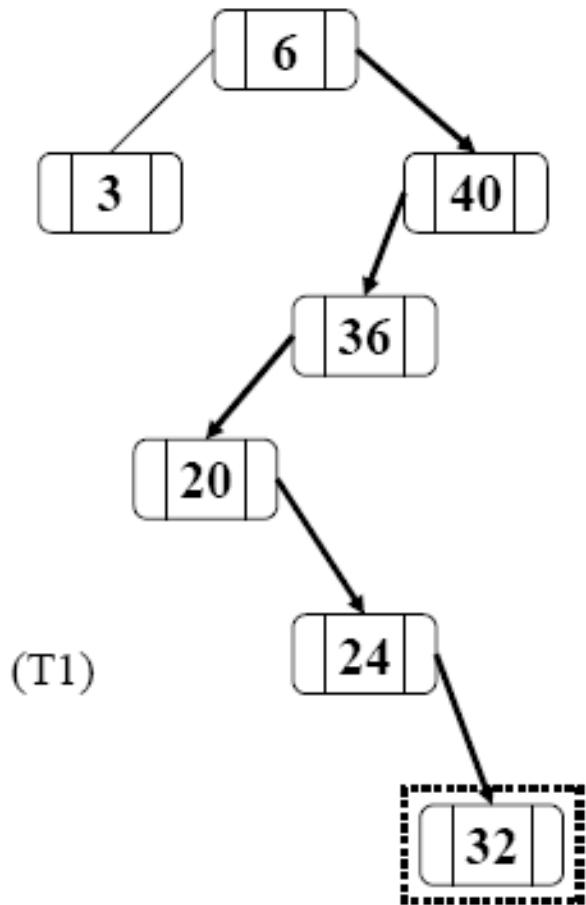
Binary Search Tree – Hàm Main()

```
264     print('*****Xoa 1 phần tử có trong cây: ')
265     gt = int(input('Nhập vào giá trị cần xoá '))
266     print(f'Xoa {gt}')
267     cay.xoa(gt)
268     kq = cay.duyet_trai_nut_phai()
269     print('*****Duyệt cây theo Trai - Nut - Phai (LNR): ',kq)
270
271     kq =cay.duyet_nut_trai_phai()
272     print('*****Duyệt cây theo Nut - Trai -Phai (NLR): ',kq)
273
274     kq =cay.duyet_trai_phai_nut()
275     print('*****Duyệt cây theo Trai -Phai - Nut (LRN): ',kq)
276     #Tìm
277     while True:
278         nhap = input('Nhập vào khoá cần tìm ')
279         if nhap == '':
280             break
281         #if
282         gt = int(nhap)
283         kq = cay.tim(gt)
284         if kq == None:
285             print (f'Không tìm thấy {gt}')
286         else:#Tìm thấy
287             print(f'Tìm thấy {gt}:{kq}')
288         #if
289     #while
290 #def
```

```
291 if __name__=='__main__':
292     main()
293 #if
```

Đánh giá tìm kiếm

159

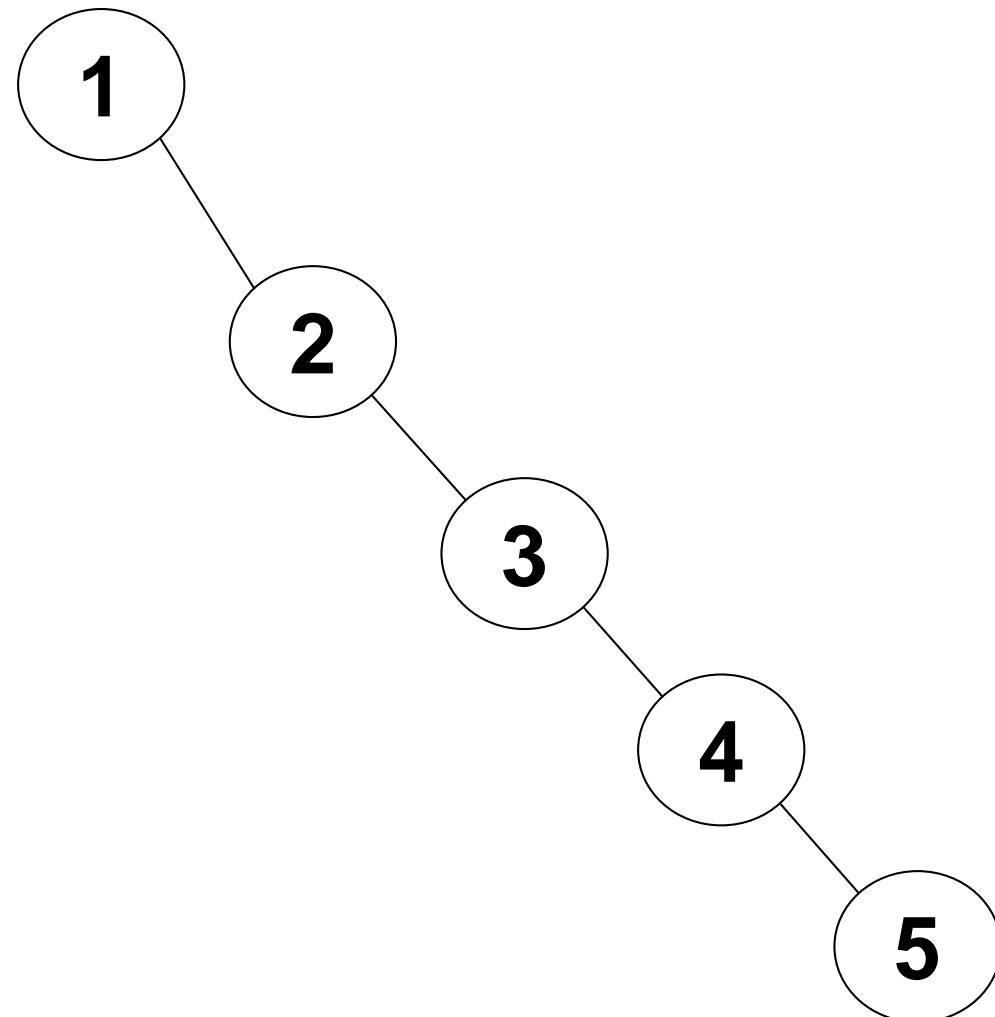


Tìm giá trị 32

Đánh giá tìm kiếm

160

- 1, 2, 3, 4, 5



Binary Search Tree

161

- 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 $\log_2(n)$

Trắc nghiệm

162

- The following items are inserted into a binary search tree:
3,6,5,2,4,7,1.

Which node is the deepest?

- a) 1
- b) 7
- c) 3
- d) 4

Bài tập

163

1. Vẽ hình cây TKNP tạo ra từ cây rỗng bằng cách lần lượt thêm vào các khoá là các số nguyên: 54, 31, 43, 29, 65, 10, 20, 36, 78, 59.
2. Vẽ lại hình cây TKNP ở câu 1 sau khi lần lượt xen thêm các nút 15, 45, 55.
3. Vẽ lại hình cây TKNP ở câu 2 sau khi lần lượt xoá các nút 10, 20, 43, 65, 54.
4. Hãy dựng cây TKNP ứng với dãy khóa (thứ tự tính theo qui tắc so sánh chuỗi): HAIPHONG, CANTHO, NHATRANG, DALAT, HANOI, ANGIANG, MINHHAI, HUE, SAIGON, VINHLONG. Đánh dấu đường đi trên cây khi tìm kiếm khóa DONGTHAP.

Bài tập

164

□ Bạn hãy thực hiện các câu dưới đây với cây nhị phân tìm kiếm sau:

1. Vẽ lại cây sau khi xóa

nút 14

2. Vẽ lại cây sau khi xóa

nút 31

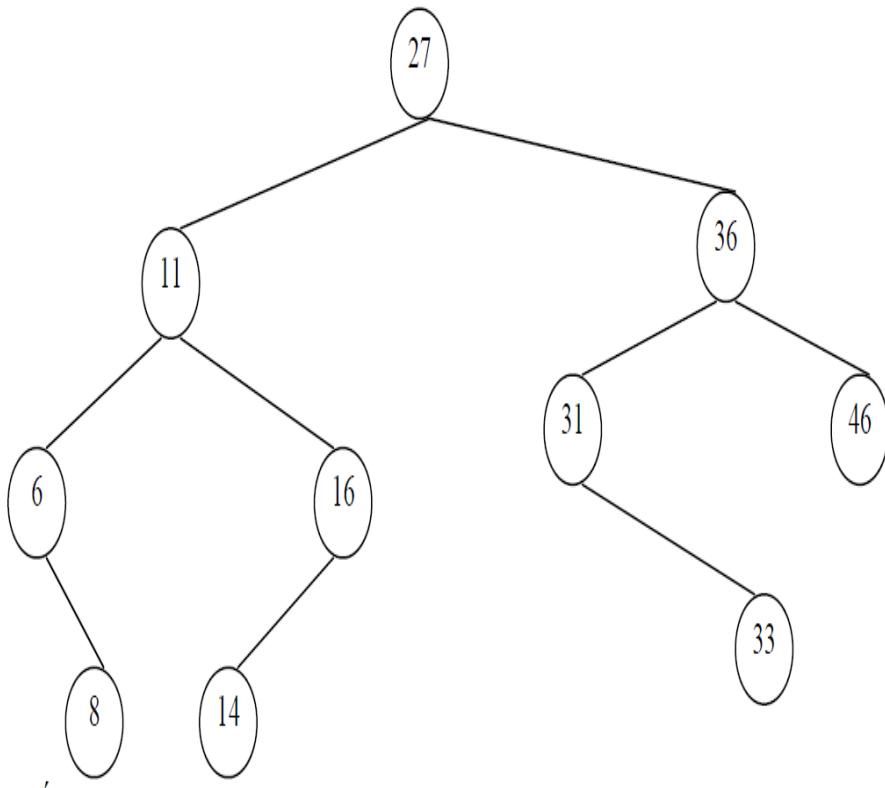
3. Vẽ lại cây sau khi xóa nút

11

4. Vẽ lại cây sau khi xóa nút

gốc

Chú ý: tất cả các câu đều
dùng lại cây nhị phân trên



Bài tập

165

- Viết các hàm:
 - Đếm số nút có đúng 1 con
 - Đếm số nút có đúng 2 con
 - Đếm số nguyên tố trên cây
 - Tính tổng các nút có đúng 1 con
 - Tính tổng các nút có đúng 2 con
 - Tính tổng các số chẵn
 - Nhập x, tìm giá trị nhỏ nhất trên cây mà lớn hơn x
 - Xuất số nguyên tố nhỏ nhất trên cây
 - Nhập x, tìm x trên cây, nếu tìm thấy x thì cho biết x có bao nhiêu con
 - Xóa 1 nút

Nội dung

166

- 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*)

Giới thiệu AVL Tree

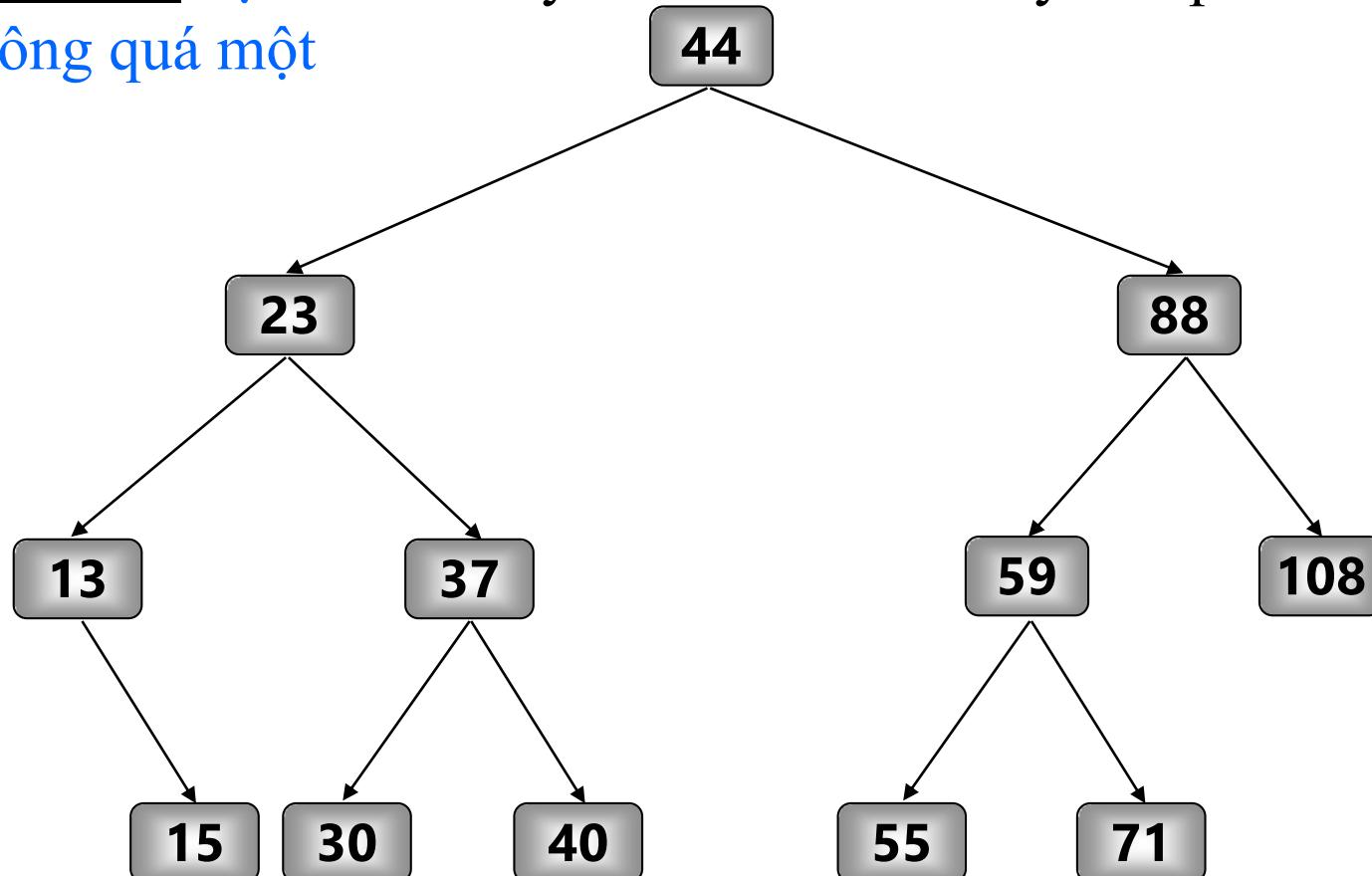
167

- Phương pháp chèn trên CNPTK có thể có những biến dạng mất cân đối nghiêm trọng
 - ▣ **Chi phí cho việc tìm kiếm** trong trường hợp xấu nhất đạt tới n
 - ▣ VD: 1 triệu nút \Rightarrow chi phí tìm kiếm = 1.000.000 nút
- Nếu có một cây tìm kiếm nhị phân cân bằng hoàn toàn, **chi phí cho việc tìm kiếm** chỉ xấp xỉ $\log_2 n$
 - ▣ VD: 1 triệu nút \Rightarrow chi phí tìm kiếm = $\log_2 1.000.000 \approx 20$ nút
- G.M. **Adelson-Velsky** và E.M. **Landis** đã đề xuất một tiêu chuẩn cân bằng (sau này gọi là cân bằng **AVL**)
 - ▣ Cây AVL có chiều cao $O(\log_2(n))$

AVL Tree - Định nghĩa

168

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



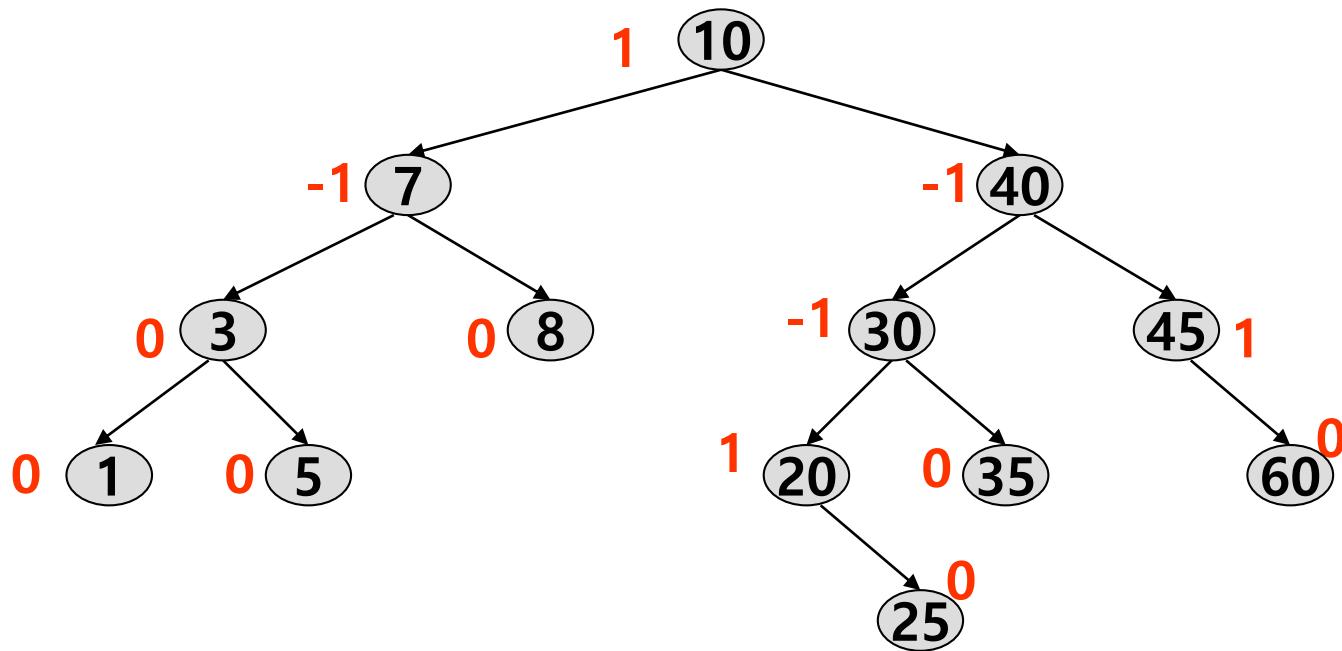
AVL Tree

169

- Chỉ số cân bằng của một nút:
 - Định nghĩa: Chỉ số cân bằng của một nút là **hiệu** của chiều cao cây con phải và cây con trái của nó
 - Đối với một cây cân bằng, chỉ số cân bằng (CSCB) của mỗi nút chỉ có thể mang một trong ba giá trị sau đây:
 - $CSCB(p) = 0 \Leftrightarrow$ Độ cao cây phải (p) = Độ cao cây trái (p)
 - $CSCB(p) = 1 \Leftrightarrow$ Độ cao cây phải (p) > Độ cao cây trái (p)
 - $CSCB(p) = -1 \Leftrightarrow$ Độ cao cây phải (p) < Độ cao cây trái (p)
- Để tiện trong trình bày, chúng ta sẽ ký hiệu như sau:

p->balFactor = CSCB(p);
- Độ cao cây trái (p) ký hiệu là **hL**
- Độ cao cây phải(p) ký hiệu là **hR**

Ví dụ - Chỉ số cân bằng của nút

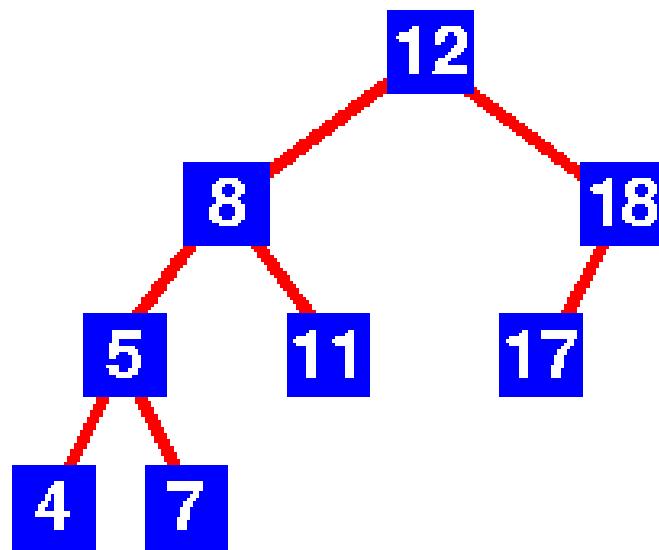


- What is the balance factor for each node in this AVL tree?
- Is this an AVL tree?

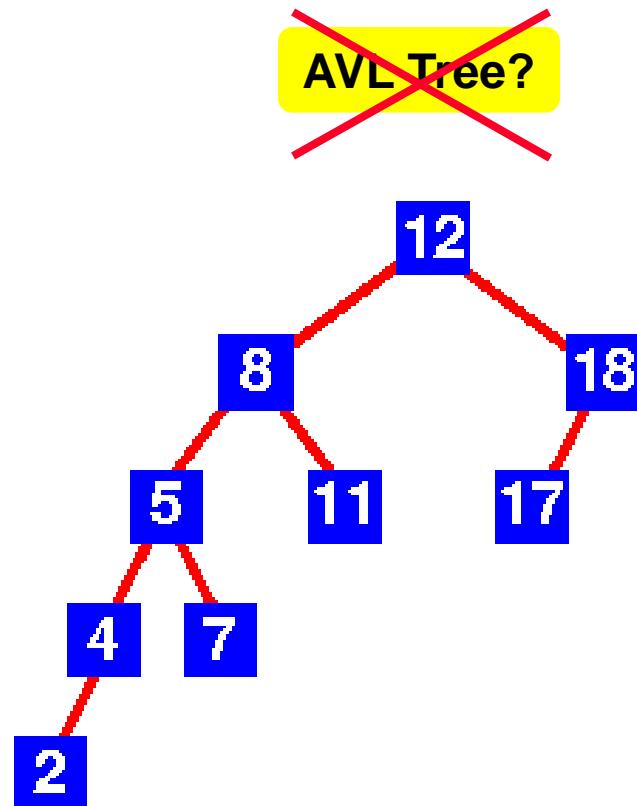
AVL Tree – Ví dụ

171

AVL Tree ?

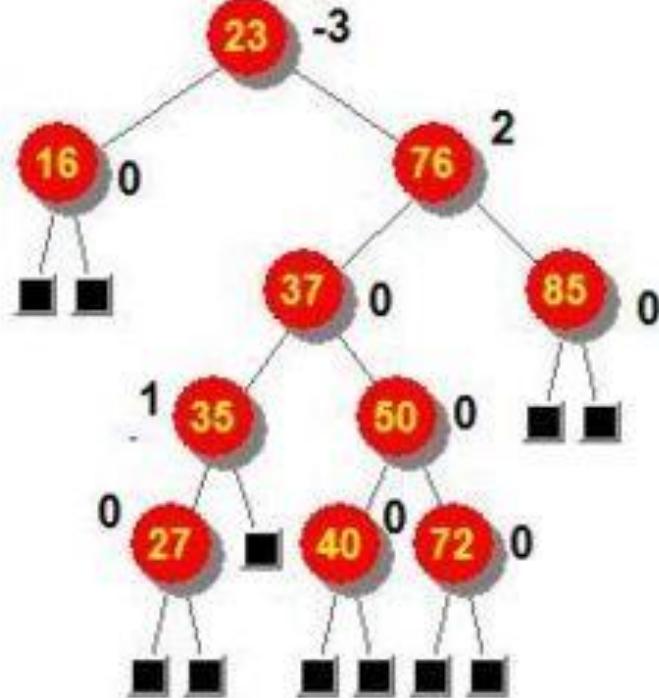


AVL Tree?



AVL Tree – Ví dụ

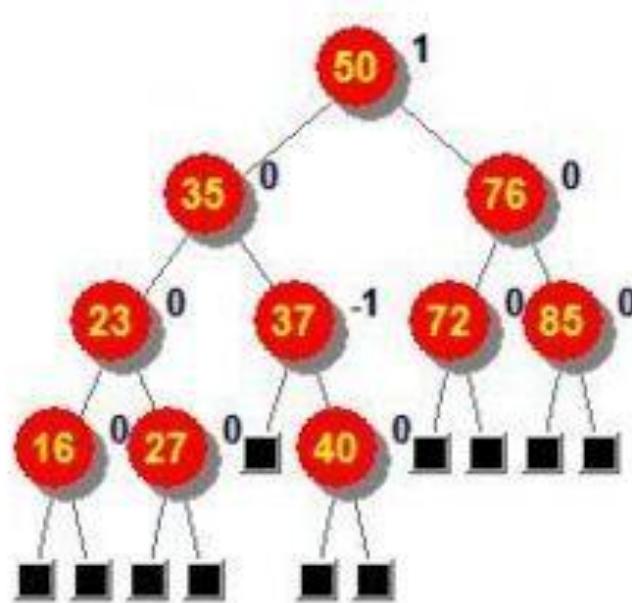
172



1. Cây tìm kiếm nhị phân
không là cây AVL

Hai cây được tạo từ cùng dãy khóa

23;76;37;85;50;40;72;35;16;27;



2. Cây AVL

AVL Tree – Biểu diễn

173

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

struct AVLNode{
    char          balFactor;      // Chỉ số cân bằng
    DataType      data;
    AVLNode*     left;
    AVLNode*     right;
};

typedef AVLNode* AVLTree;
```

AVL Tree – Biểu diễn

174

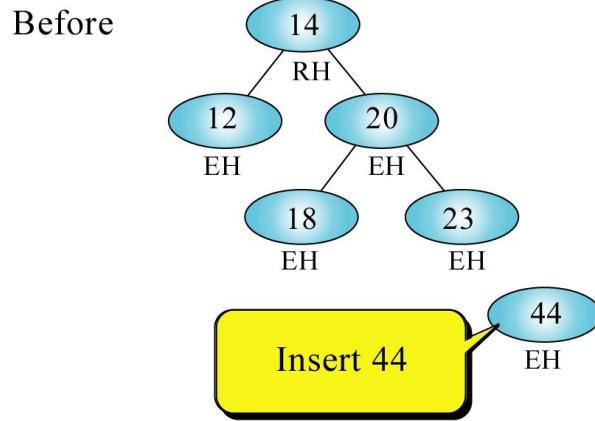
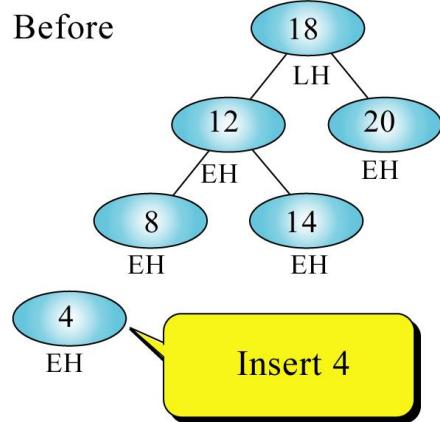
- Các thao tác đặc trưng của cây AVL:
 - **Thêm** một phần tử vào cây AVL
 - **Hủy** một phần tử trên cây AVL
 - **Cân bằng lại** một cây vừa bị mất cân bằng (**Rotation**)
- Trường hợp **thêm** một phần tử trên cây AVL được thực hiện giống như thêm trên CNPTK, tuy nhiên sau khi thêm phải cân bằng lại cây
- Trường hợp **hủy** một phần tử trên cây AVL được thực hiện giống như hủy trên CNPTK và cũng 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

AVL Tree - Các trường hợp mất cân bằng

175

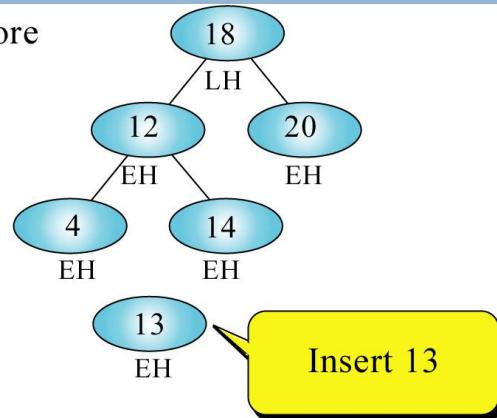
- Ta sẽ không khảo sát tính cân bằng của 1 cây nhị phân bất kỳ mà chỉ quan tâm đến các khả năng mất cân bằng xảy ra khi **chèn hoặc xóa** một nút trên cây AVL
- Các trường hợp mất cân bằng:
 - Sau khi chèn (xóa) cây con **trái lệch trái** (left of left)
 - Sau khi chèn (xóa) cây con **trái lệch phải** (right of left)
 - Sau khi chèn (xóa) cây con **phải lệch phải** (right of right)
 - Sau khi chèn (xóa) cây con **phải lệch trái** (left of right)

Ví dụ: Các trường hợp mất cân bằng

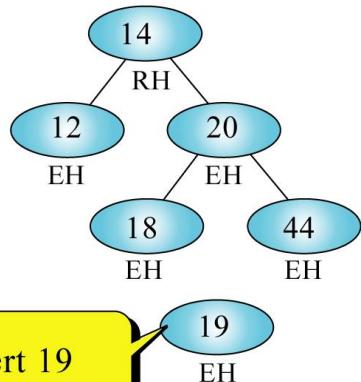


Ví dụ: Các trường hợp mất cân bằng

Before



Before



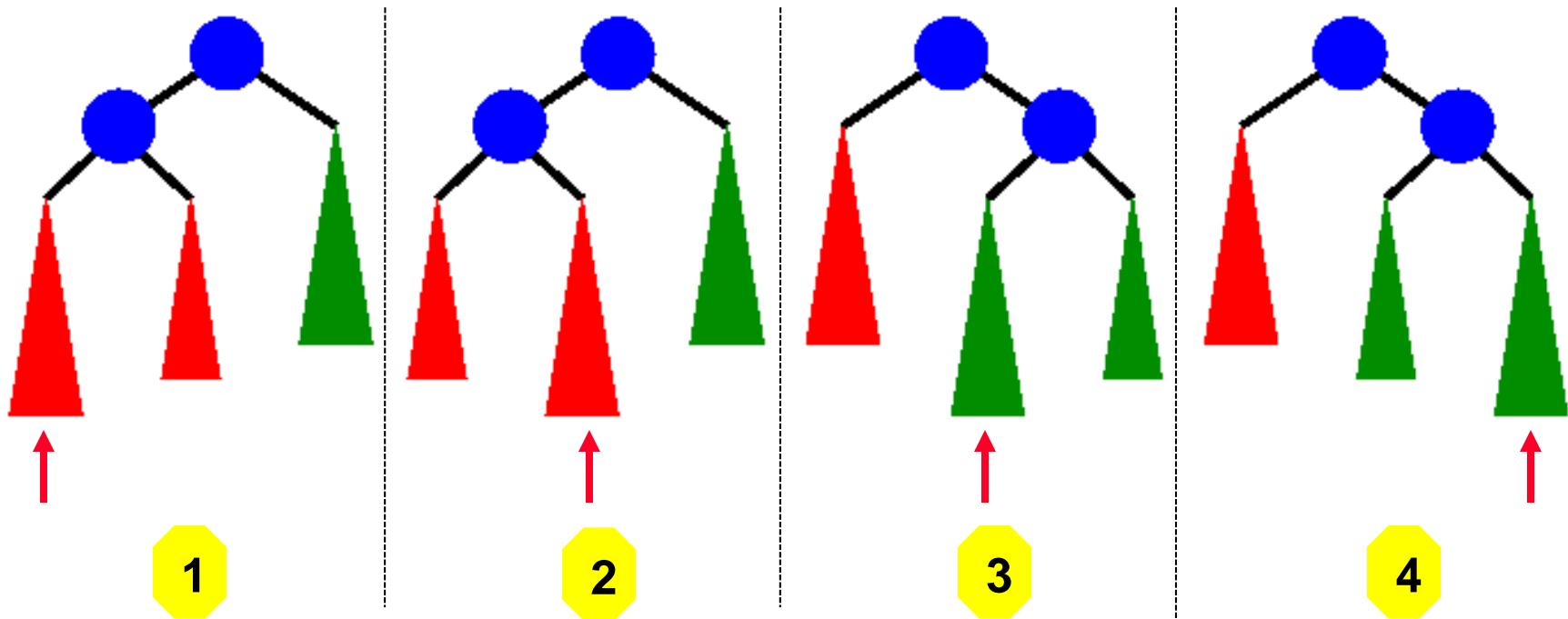
AVL Tree

178

- 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 - Các trường hợp mất cân bằng

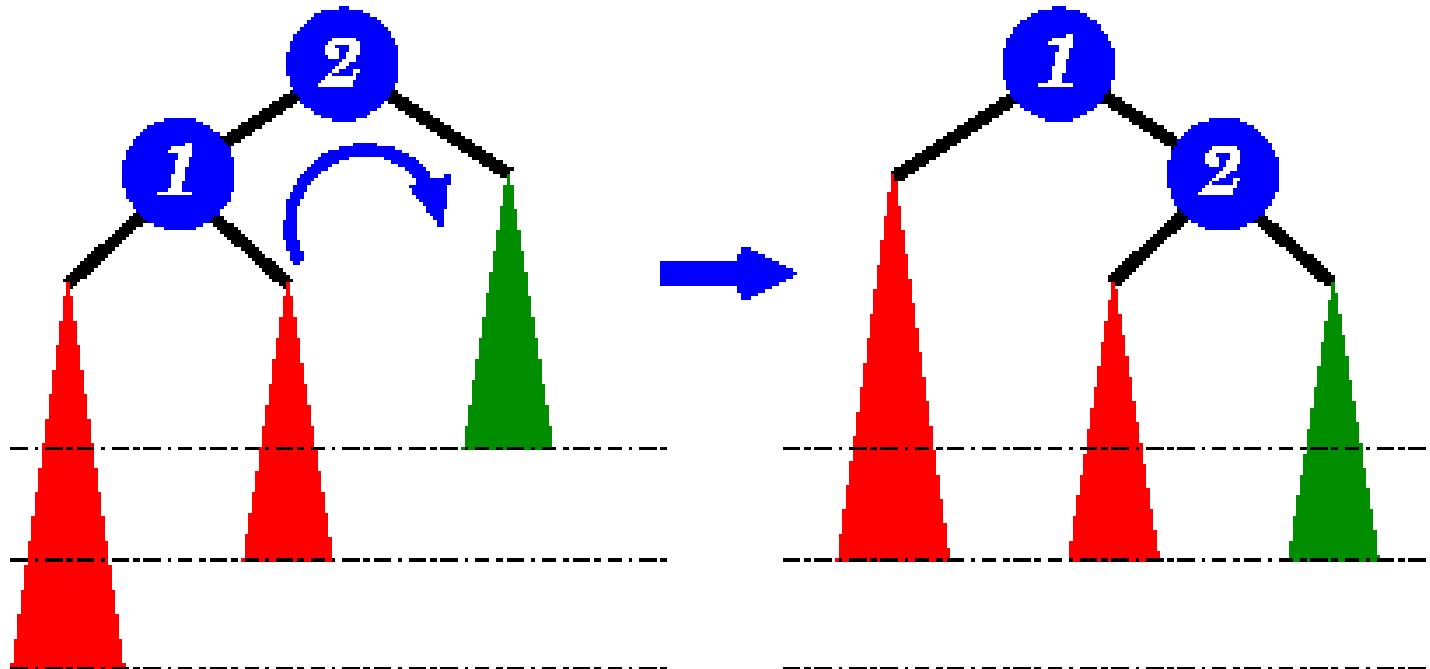
□ Chèn nút vào cây AVL



- 1 và 4 là các ảnh đối xứng
- 2 và 3 là các ảnh đối xứng

Cây AVL – Tái cân bằng

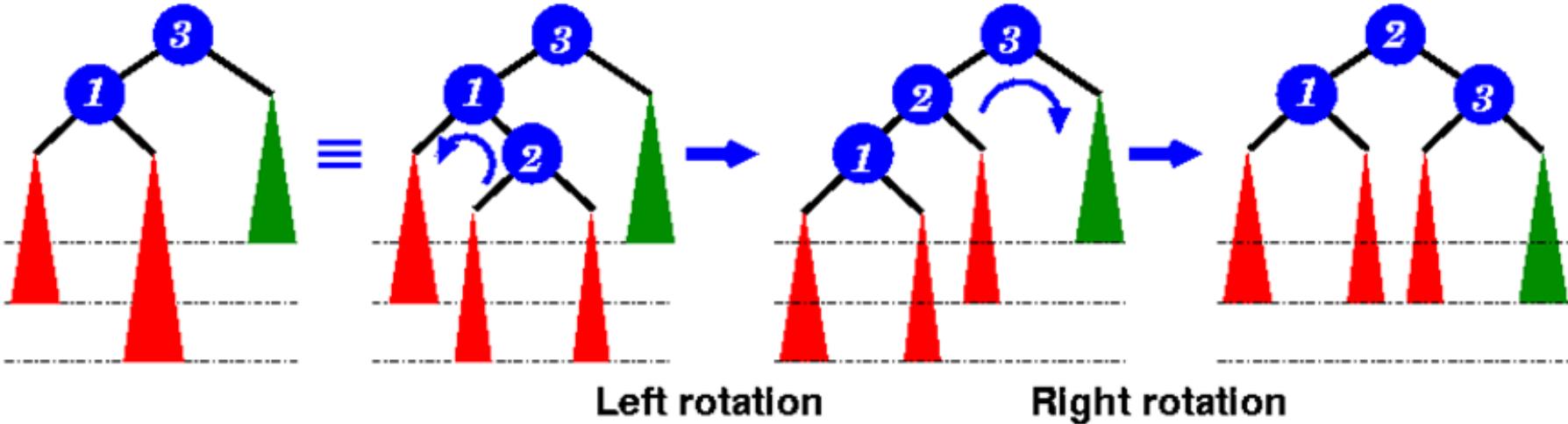
- Trường hợp 1 được giải bởi phép quay:



- Trường hợp 4 là quay một ảnh đối xứng

Cây AVL – Tái cân bằng

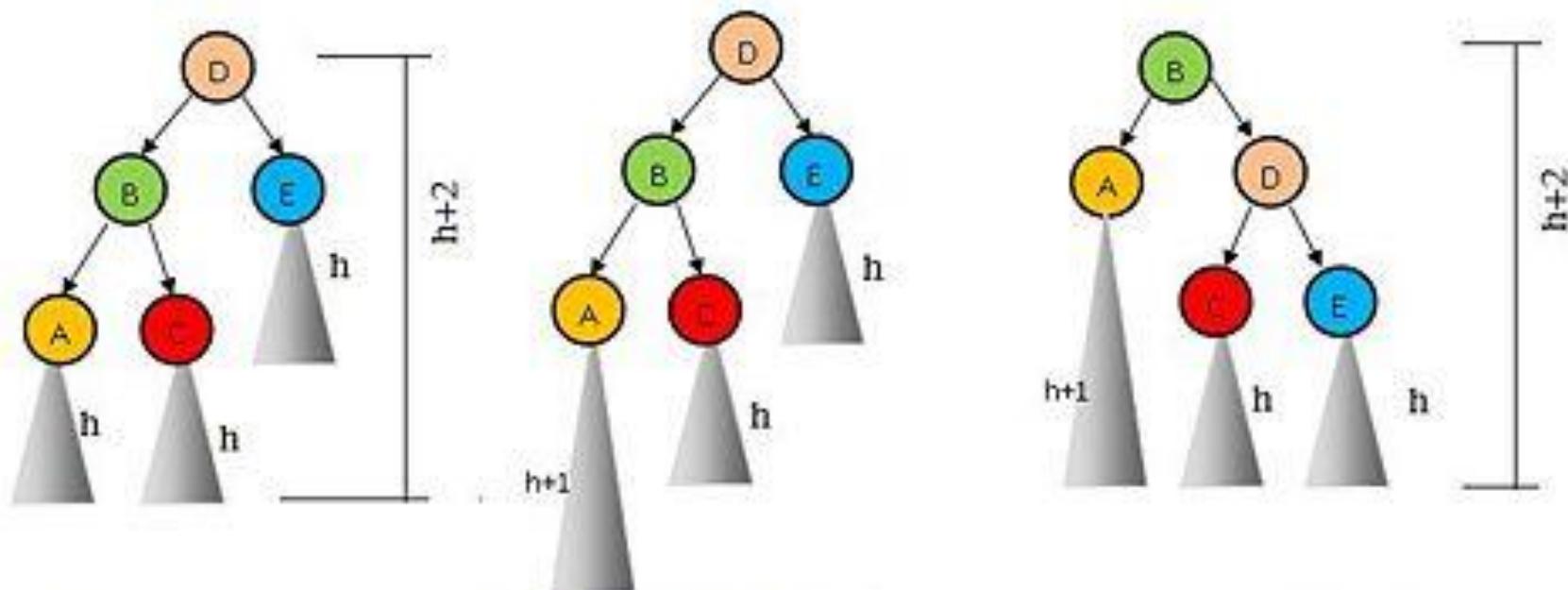
- Trường hợp 2 cần một phép quay kép (*double*)



- Trường hợp 3 là phép quay ảnh đối xứng

Cây AVL – Tái cân bằng

182



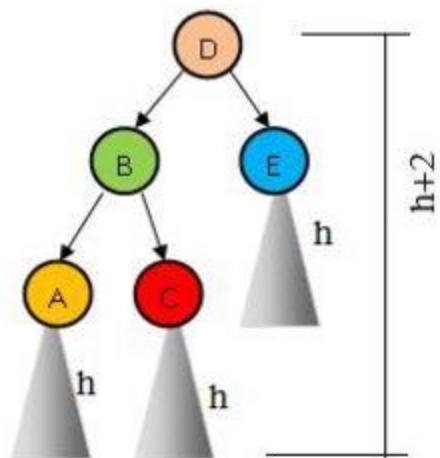
1. Cây lệch trái trước khi chèn . Chiều cao cây là $h+2$,

2. Chèn một phần tử vào cây con trái của cây con trái làm cho cây mất cân bằng AVL

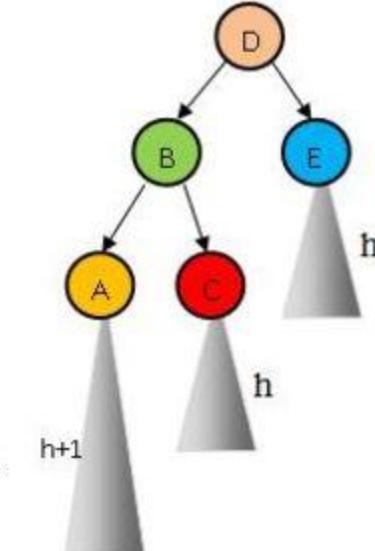
3. Sau phép quay phải cây trở thành cân bằng, chiều cao cây vẫn là $h+2$

Cây AVL – Tái cân bằng

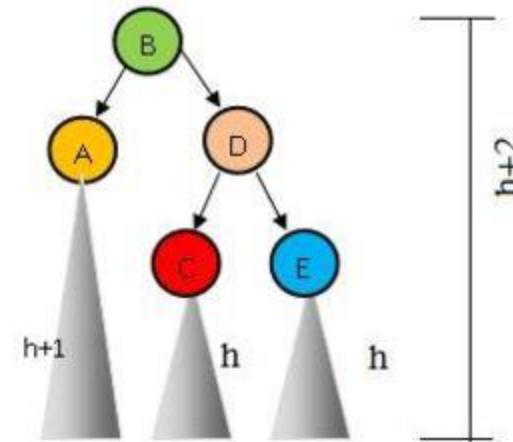
183



1. Cây lệch trái trước khi chèn . Chiều cao cây là $h+2$,



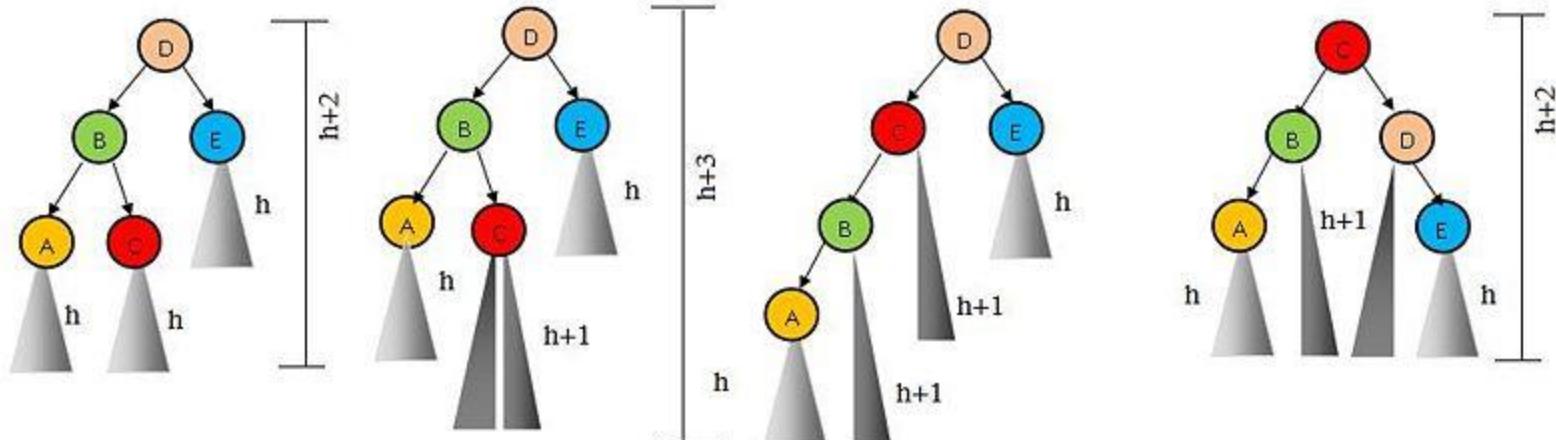
2. Chèn một phần tử vào cây con trái của cây con trái làm cho cây mất cân bằng AVL



3. Sau phép quay phải cây trở thành cân bằng, chiều cao cây vẫn là $h+2$

Cây AVL – Tái cân bằng

184



1. Cây ban đầu lệch trái,
cây con trái cân bằng.
Chiều cao cây là $h+2$

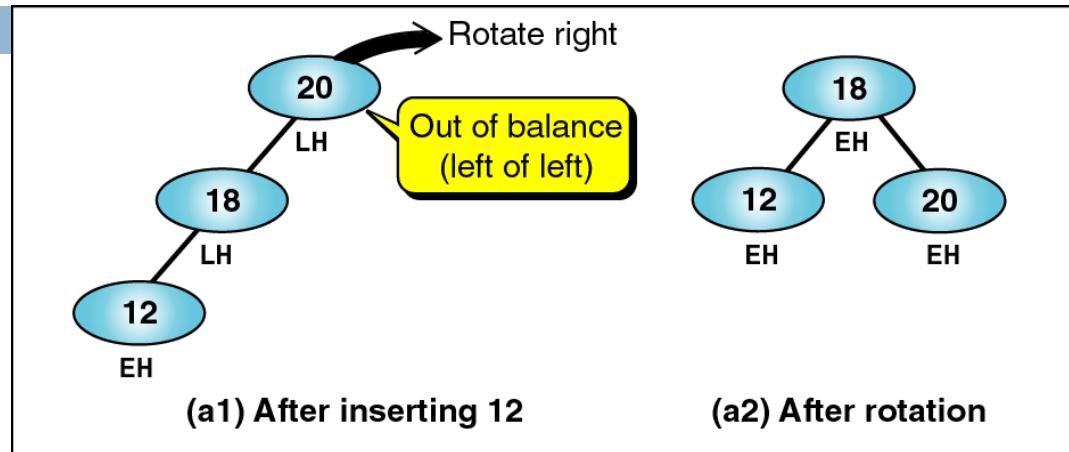
2. Phép chèn vào cây con
trái làm cho cây con trái
lệch phải.

3. Phép xoay trái cây con trái
đưa cây con trái thành cây lệch
trái

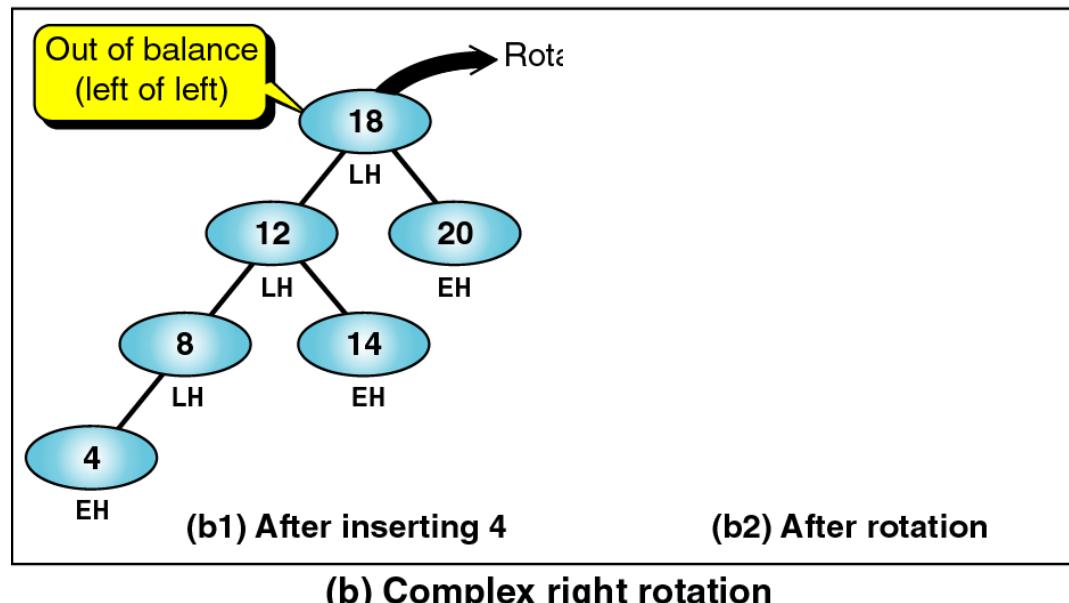
4.Sau phép xoay phải cây,
cây trở thành cân bằng AVL.
Chiều cao vẫn là $h+2$

Ví dụ: Tái cân bằng

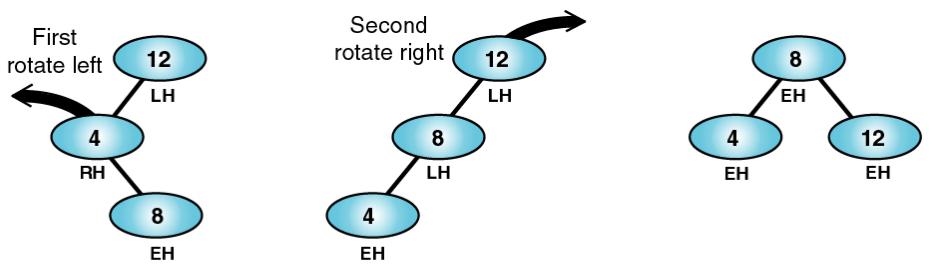
185



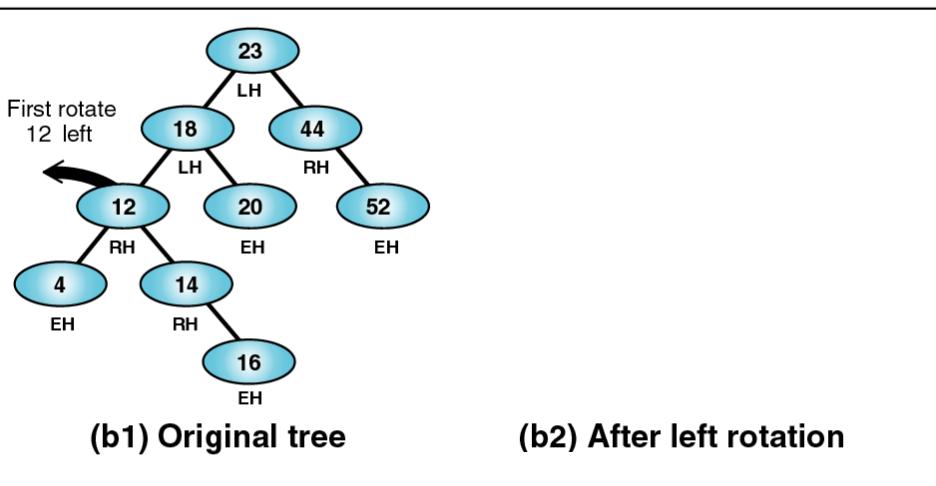
1. left of left



Ví dụ: Tái cân bằng

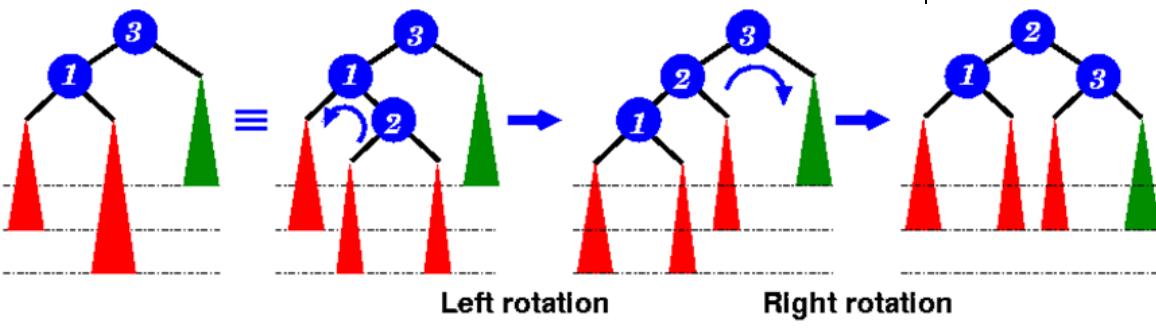


(a) Simple double rotation right



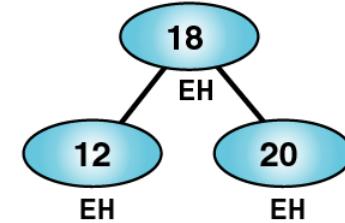
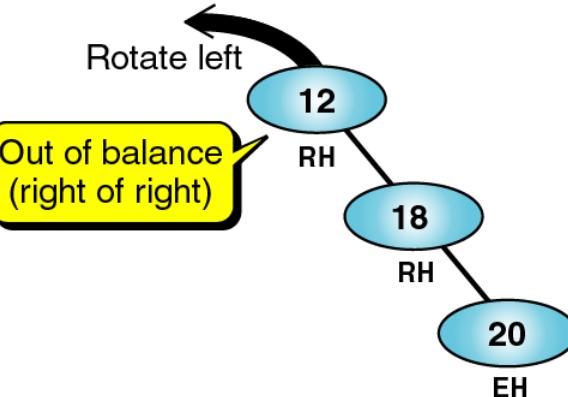
(b1) Original tree

(b2) After left rotation

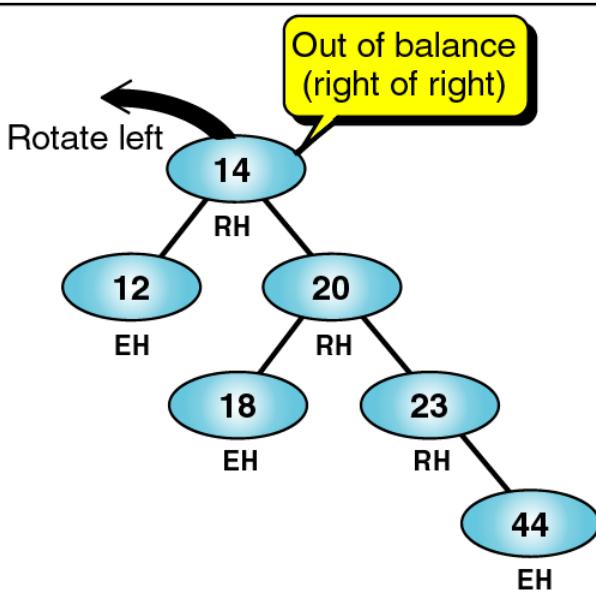


(b) Complex double rotation right

Ví dụ: Tái cân bằng



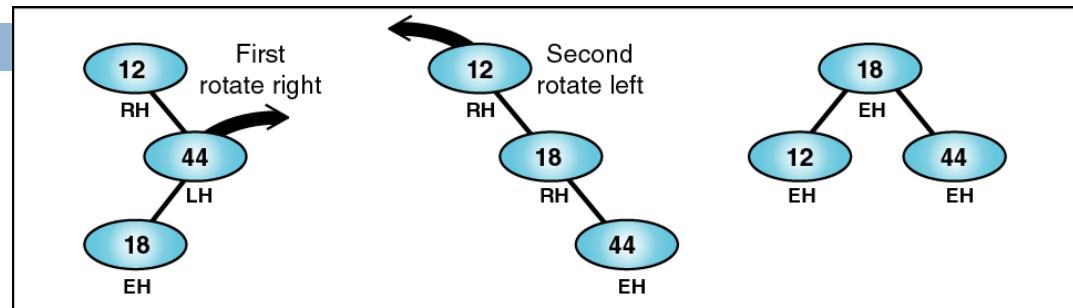
(a) Simple left rotation



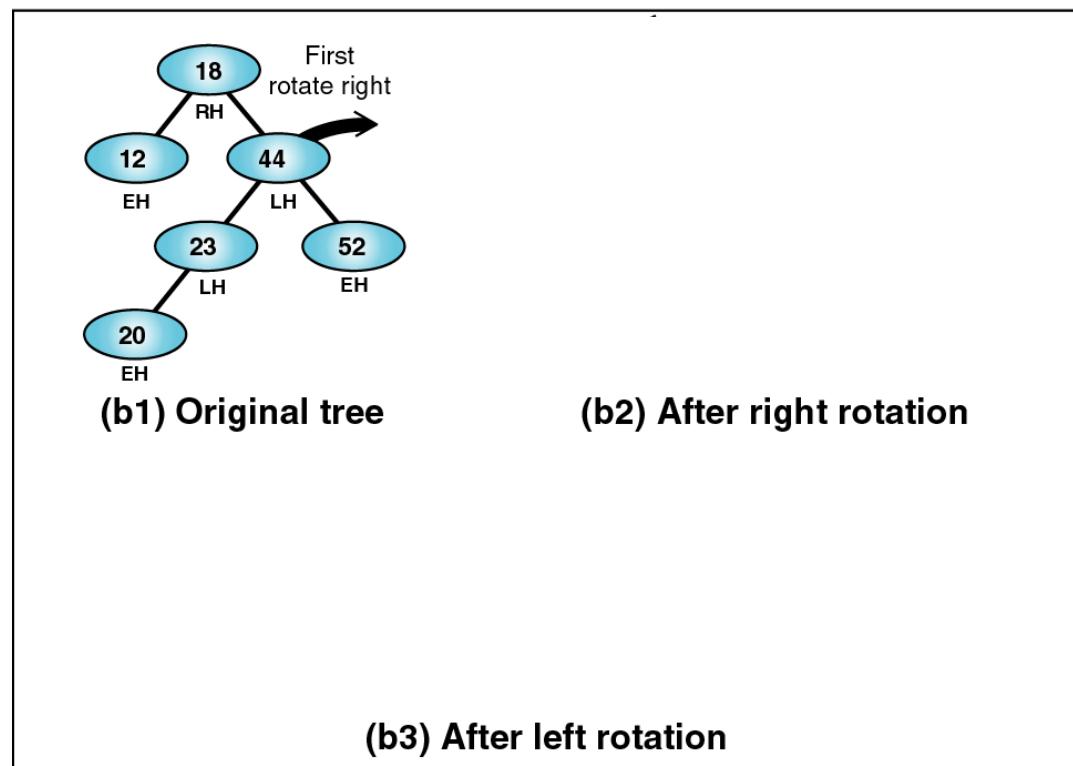
3. right of right

(b) Complex left rotation

Ví dụ: Tái cân bằng



(a) Simple double rotation right



4. left of right

AVL Tree - Cân bằng lại cây AVL

189

- Quay đơn Left-Left:

```
void rotateLL(AVLTree &root) //quay đơn Left-Left
{
    AVLNode* root1 = root->left;
    root->left = root1->right;
    root1->right = root;
    switch(root1->balFactor) {
        case LH: root->balFactor = EH;
                    root1->balFactor = EH;
                    break;
        case EH: root->balFactor = LH;
                    root1->balFactor = RH;
                    break;
    }
    root = root1;
}
```

AVL Tree - Cân bằng lại cây AVL

190

- Quay đơn Right-Right:

```
void rotateRR (AVLTree &root)      //quay đơn Right-  
Right  
{  
    AVLNode* root1 = root->right;  
    root->right = root1->left;  
    root1->left = root;  
    switch(root1->balFactor)      {  
        case RH: root->balFactor = EH;  
                    root1->balFactor= EH;  
                    break;  
        case EH: root->balFactor = RH;  
                    root1->balFactor= LH;  
                    break;  
    }  
    T = T1;  
}
```

AVL Tree - Cân bằng lại cây AVL

191

- Quay kép Left-Right:

```
void rotateLR(AVLTree &root)//quay kép Left-Right
{
    AVLNode* root1 = root->left;
    AVLNode* root2 = root1->right;
    root->left = root2->right;
    root2->right = root;
    root1->right = root2->left;
    root2->left = root1;
    switch(root2->balFactor)      {
        case LH: root->balFactor = RH; root1->balFactor = EH; break;
        case EH: root->balFactor = EH; root1->balFactor = EH; break;
        case RH: root->balFactor = EH; root1->balFactor = LH; break;
    }
    root2->balFactor = EH;
    root = root2;
}
```

AVL Tree - Cân bằng lại cây AVL

192

- Quay kép Right-Left

```
void rotateRL(AVLTree &root) //quay kép Right-Left
{
    AVLNode* root1 = root->right;
    AVLNode* root2 = root1->left;
    root->right = root2->left;
    root2->left = root;
    root1->left = root2->right;
    root2->right = root1;
    switch(root2->balFactor) {
        case RH: root->balFactor = LH; root1->balFactor = EH; break;
        case EH: root->balFactor = EH; root1->balFactor = EH; break;
        case LH: root->balFactor = EH; root1->balFactor = RH; break;
    }
    root2->balFactor = EH;
    root = root2;
}
```

AVL Tree - Cân bằng lại cây AVL

193

- Cân bằng khi cây bị lệch về bên trái:

```
int balanceLeft(AVLTree &root)
//Cân bằng khi cây bị lệch về bên trái
{
    AVLNode* root1 = root->left;

    switch(root1->balFactor) {
        case LH:      rotateLL(root); return 2;
        case EH:      rotateLL(root); return 1;
        case RH:      rotateLR(root); return 2;
    }
    return 0;
}
```

AVL Tree - Cân bằng lại cây AVL

194

- Cân bằng khi cây bị lệch về bên phải

```
int balanceRight(AVLTree &root )
//Cân bằng khi cây bị lệch về bên phải
{
    AVLNode* root1 = root->right;

    switch(root1->balFactor) {
        case LH:      rotateRL(root); return 2;
        case EH:      rotateRR(root); return 1;
        case RH:      rotateRR(root); return 2;
    }
    return 0;
}
```

AVL Tree - Thêm một phần tử trên cây AVL

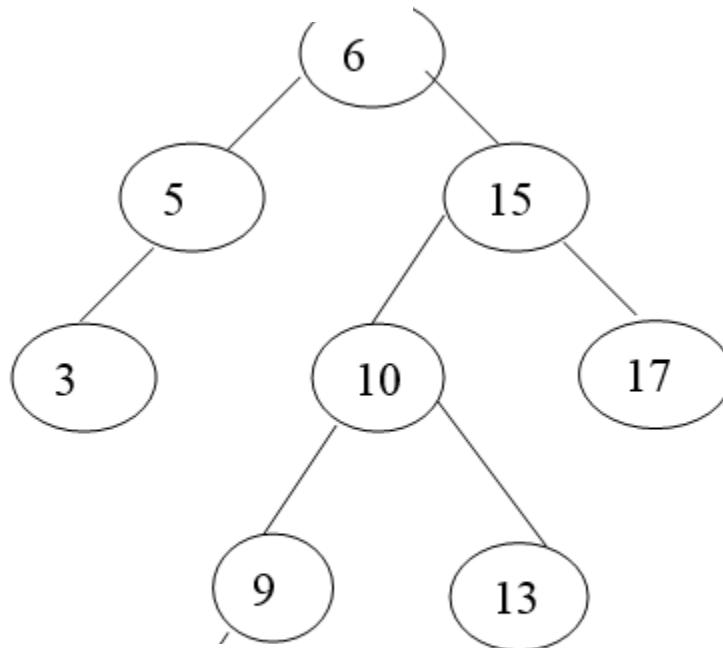
195

- 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, gấp 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)

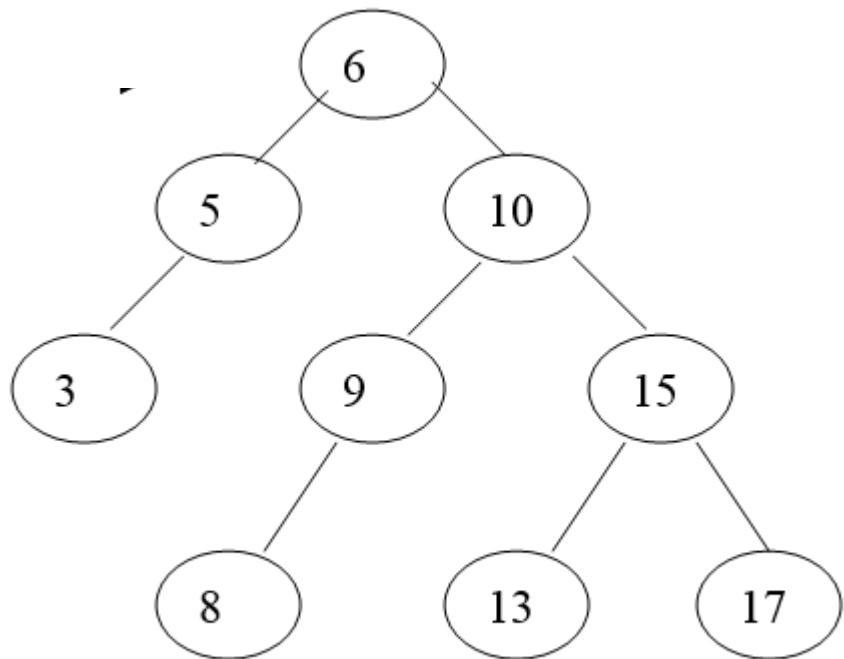
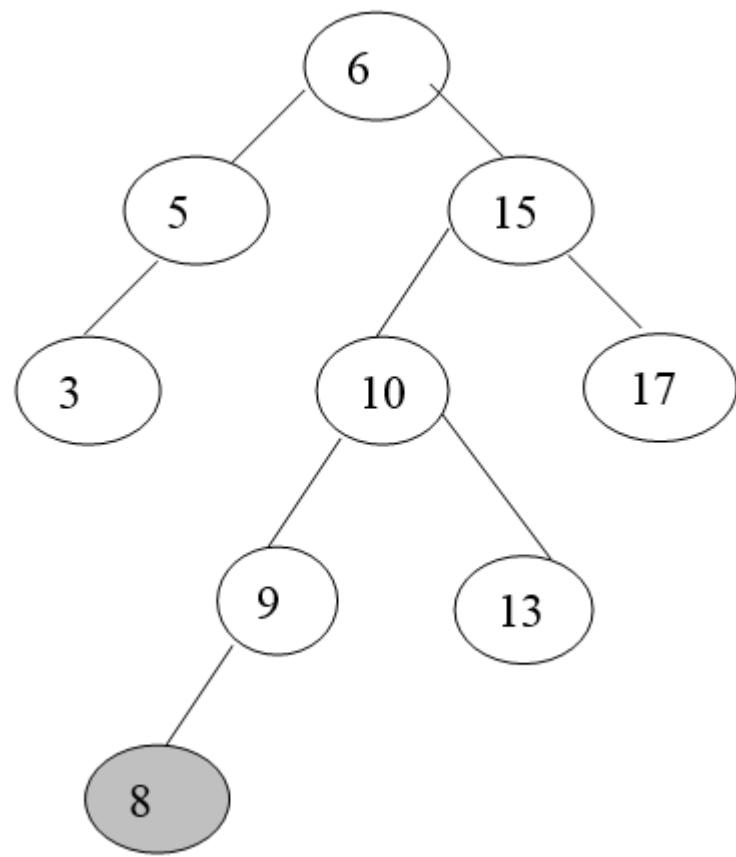
Bài tập

Cho cây BST, vẽ lại cây khi thêm nút có khóa là 8, sau khi thêm nếu cây không cân bằng thì hãy cân bằng lại



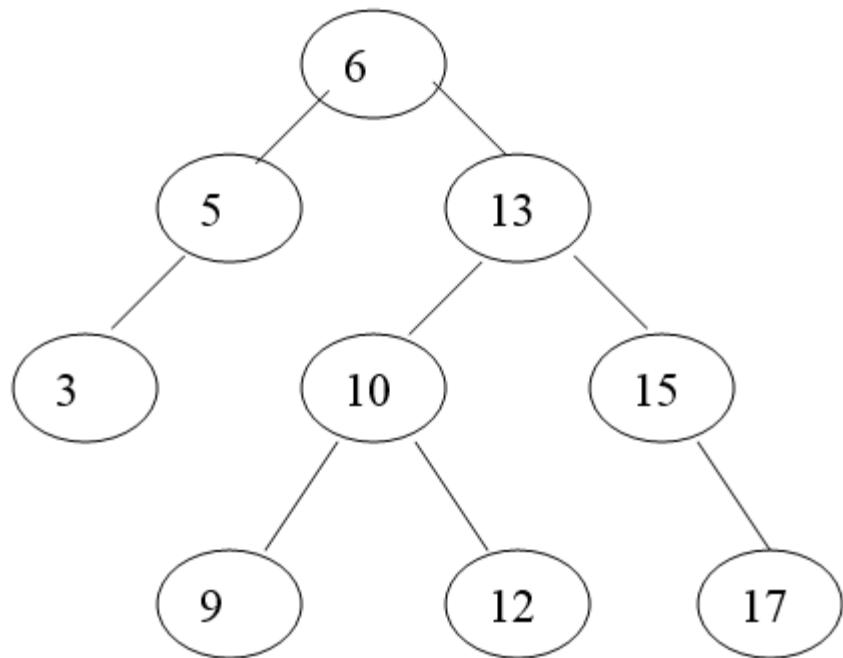
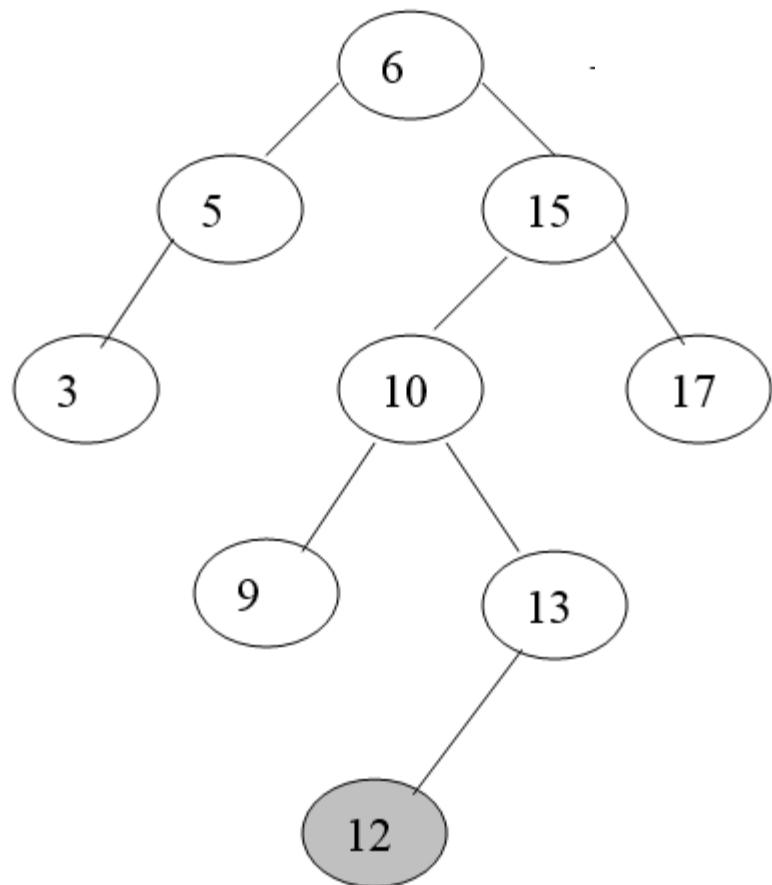
Bài tập

Cân bằng cây BST sau:



Bài tập Tái cân cân bằng

Cân bằng cây BST sau:



AVL Tree - Thêm một phần tử trên cây AVL

199

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

insertNode2 

AVL Tree - Thêm một phần tử trên cây AVL

200

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



insertNode3

AVL Tree - Thêm một phần tử trên cây AVL

201

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

AVL Tree - Hủy một phần tử trên cây AVL

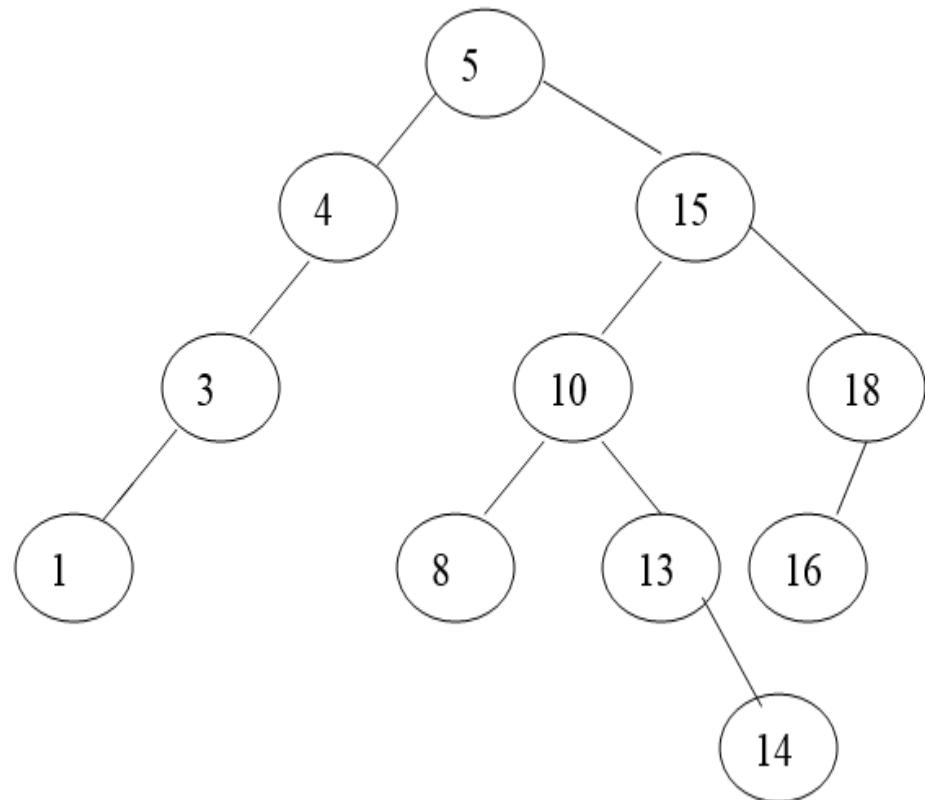
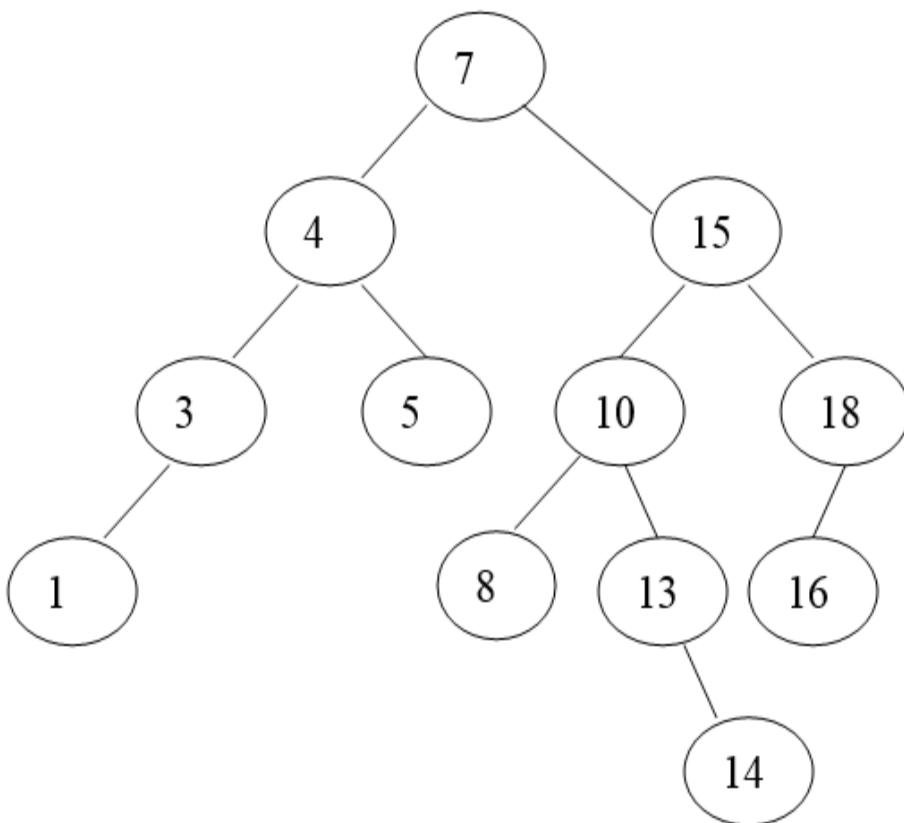
202

- 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 & *root*, DataType X)

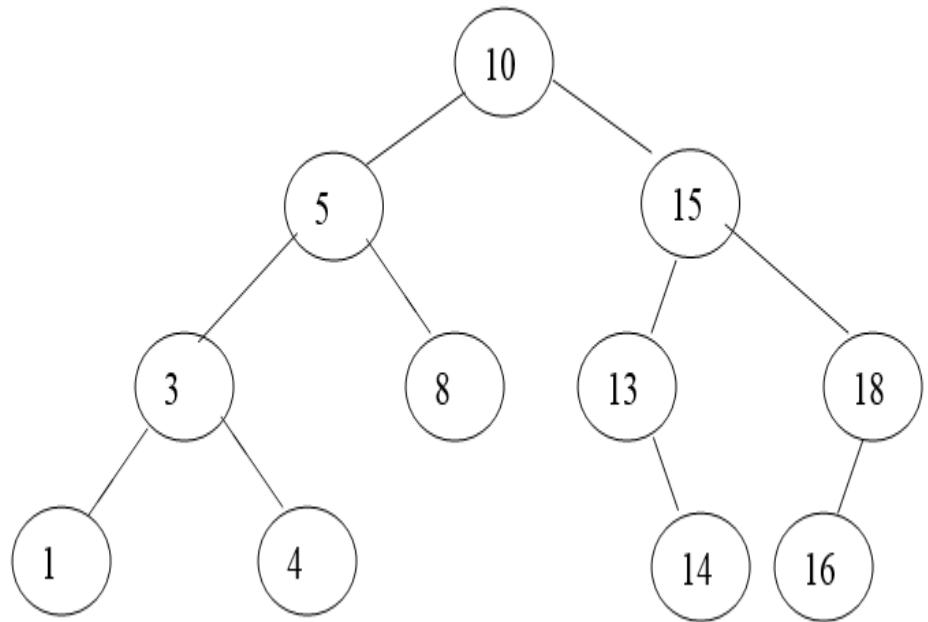
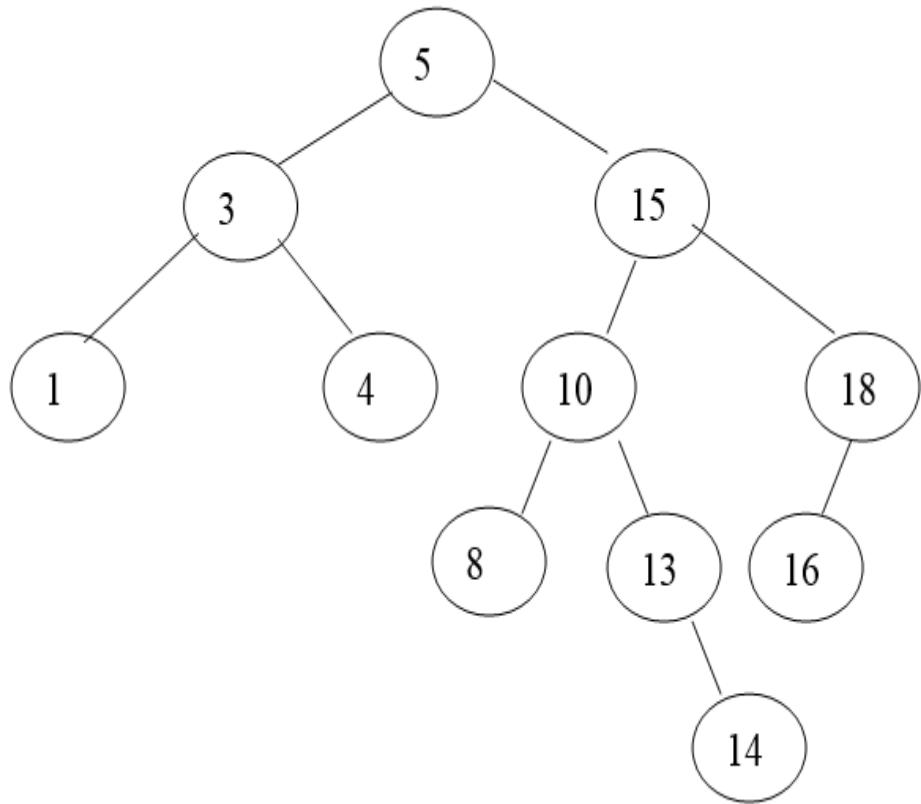
Bài tập

Cho cây BST sau, vẽ cây sau khi xóa nút có khóa là 7. Sau khi xóa, Kiểm tra cây có cân bằng không, nếu không cân bằng lại



Bài tập

Cho cây BST sau, vẽ cây sau khi xóa nút có khóa là 7. Sau khi xóa, Kiểm tra cây có cân bằng không, nếu không cân bằng lại



AVL Tree - Hủy một phần tử trên cây AVL

205

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



delNode2

AVL Tree - Hủy một phần tử trên cây AVL

206

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



delNode3

AVL Tree - Hủy một phần tử trên cây AVL

207

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

AVL Tree - Hủy một phần tử trên cây AVL

208

```
int searchStandFor (AVLTree &p, AVLTree &q)
//Tìm phần tử thế mang
{ int res;
  if (q->left)
    { res      = searchStandFor (p, q->left);
      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 (root);
        }
    } else
    { p->key = q->key; p = q; q = q->right; return 2;
    }
}
```

AVL Tree - Nhận xét

209

- 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 - Nhận xét

210

- 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