University of Science, VNU-HCM
Faculty of Information Technology

Data Structure and Algorithm

Binary Search Tree
Balanced Tree
AVL

Lecturer: Le Ngoc Thanh
Email: lnthanh@fit.hcmus.edu.vn

HCM City

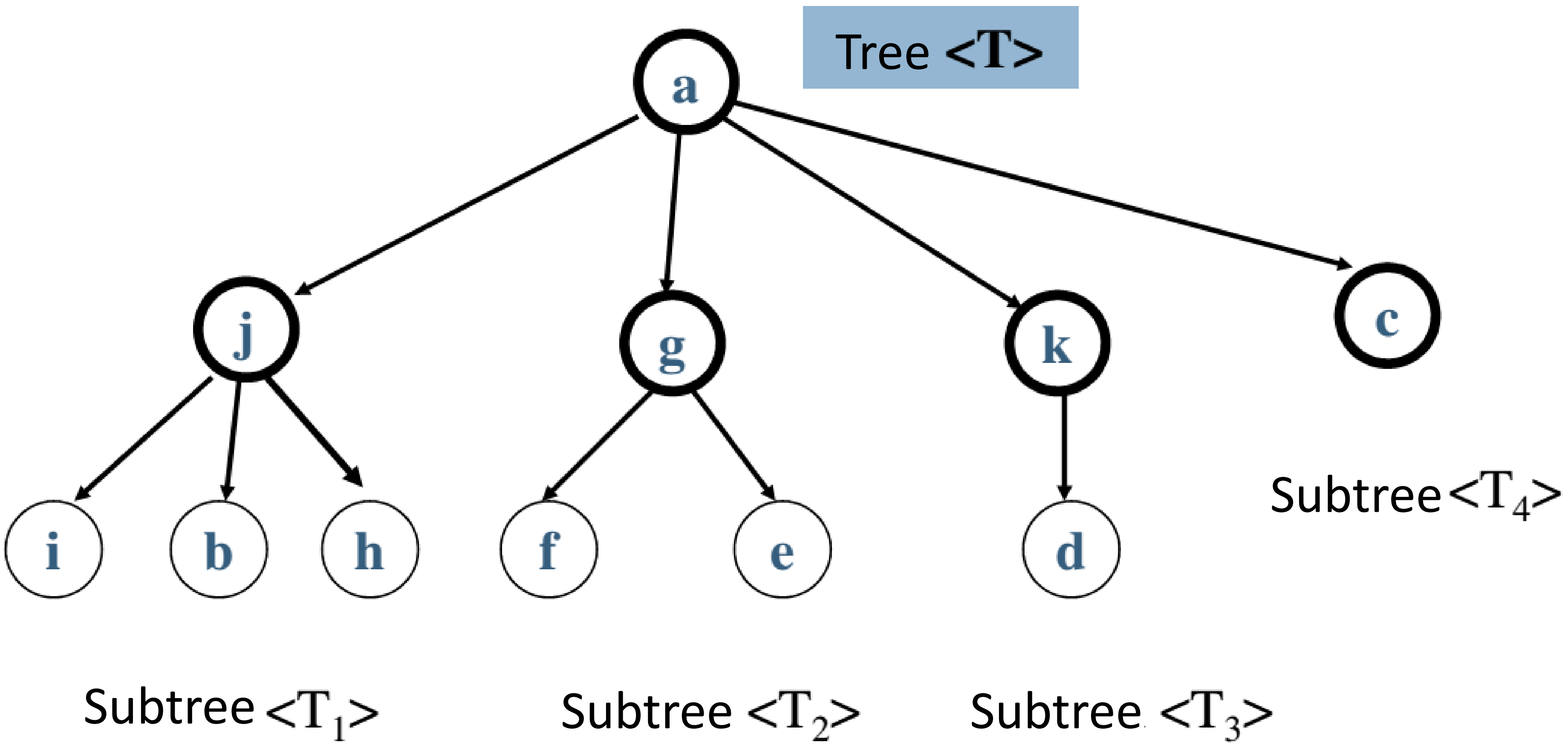
Outline

- Tree
- Binary Tree
- Binary Search Tree
- Balanced Binary Search Tree
  - AVL

Tree

- A tree <T> (Tree) is:
  - A set of elements, called nodes $p_1, p_2, \ldots, p_N$
  - If $N = 0$, the tree <T> is called an empty tree (NULL)
  - If $N > 0$:
    - There exists only one node p_k called the root of the tree
    - The remaining nodes are divided into m sets of non-intersections:
      - $T_1, T_2, \ldots, T_m$
      - Each $<T_i>$ is 1 subtree of the <T> tree

Tree



Tree <T>
a
j
g
k
c
i
b
h
f
e
d
Subtree $<T_4>$
Subtree $<T_1>$
Subtree $<T_2>$
Subtree $<T_3>$

Tree Properties

- The root node does not have a parent node.
- Each other node has only 1 parent node
- Each node can have multiple children.
- No cycle

Tree Properties

- Node: is an element in the tree.
  - Each node can contain any data
- Branch: is the connection between two nodes
- Parent node
- Child node
- Sibling nodes: are nodes that have the same parent node
- Degree of node p_i: is the number of children of p_i

Tree Properties

- Root node: A node that has no parent
- Leaf node (external node): node has degree = 0 (no child node)
- Internal node: is a node which has a parent node and a child node
- Subtree

Tree Properties

- Degree of tree: is the largest degree of the nodes in the tree
  - Degree (<T>) = max {degree (p_i) / $p_i \in$ <T>}
- Path between node p_i to node p_j: is a series of consecutive nodes from p_i to p_j such that there are branches between two adjacent nodes.
  - Path(a, d)?

Tree Properties

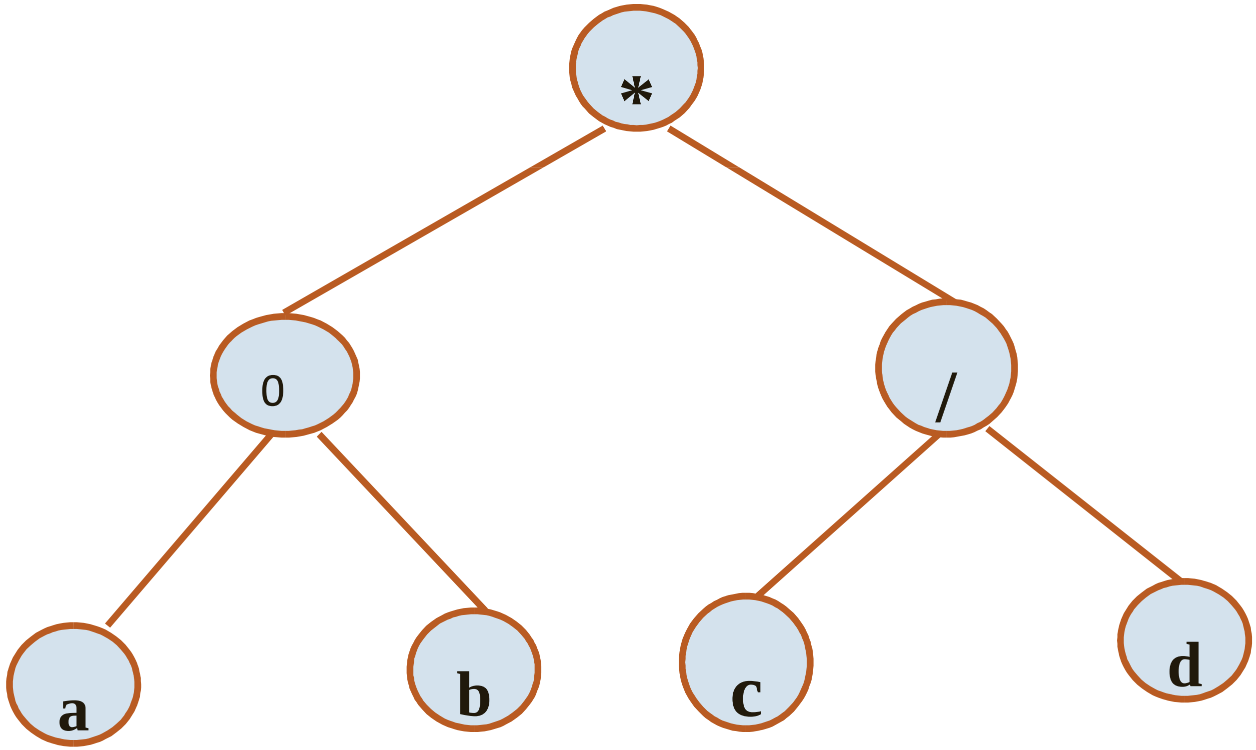
- Level:
  - Level(p) = 0 if p = root
  - Level(p) = 1 + level(parent(p)) if p! = Root
- Height of tree (h_T): the longest path from the root node to the leaf node
  - h_T = max {Path(root, p_i) | p_i is the leaf node ∈ <T>}

Outline

- Tree
- **Binary Tree**
- Binary Search Tree
- Balanced Binary Search Tree
  - AVL

Binary Tree

- A binary tree is a tree with degree = 2



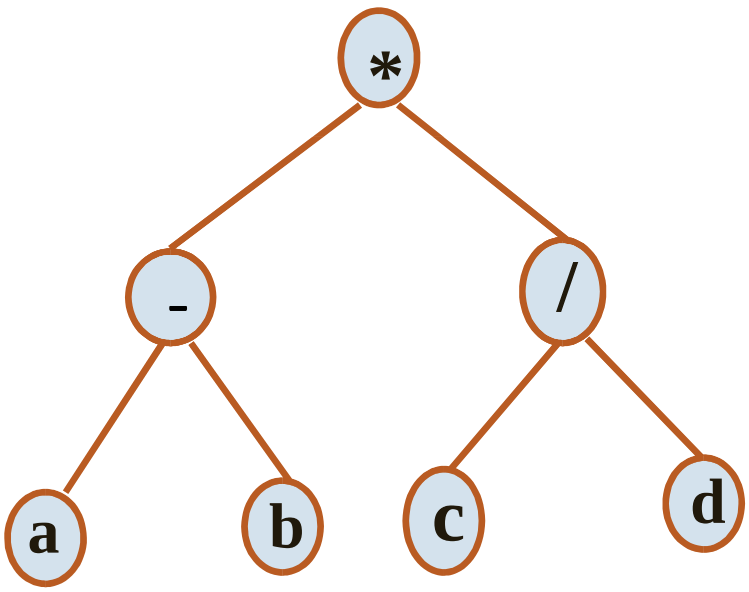
Binary Tree

- The height of a binary tree has N nodes:
  - $h_T(max) = N$
  - $h_T(min) = [\log_2 N] + 1$

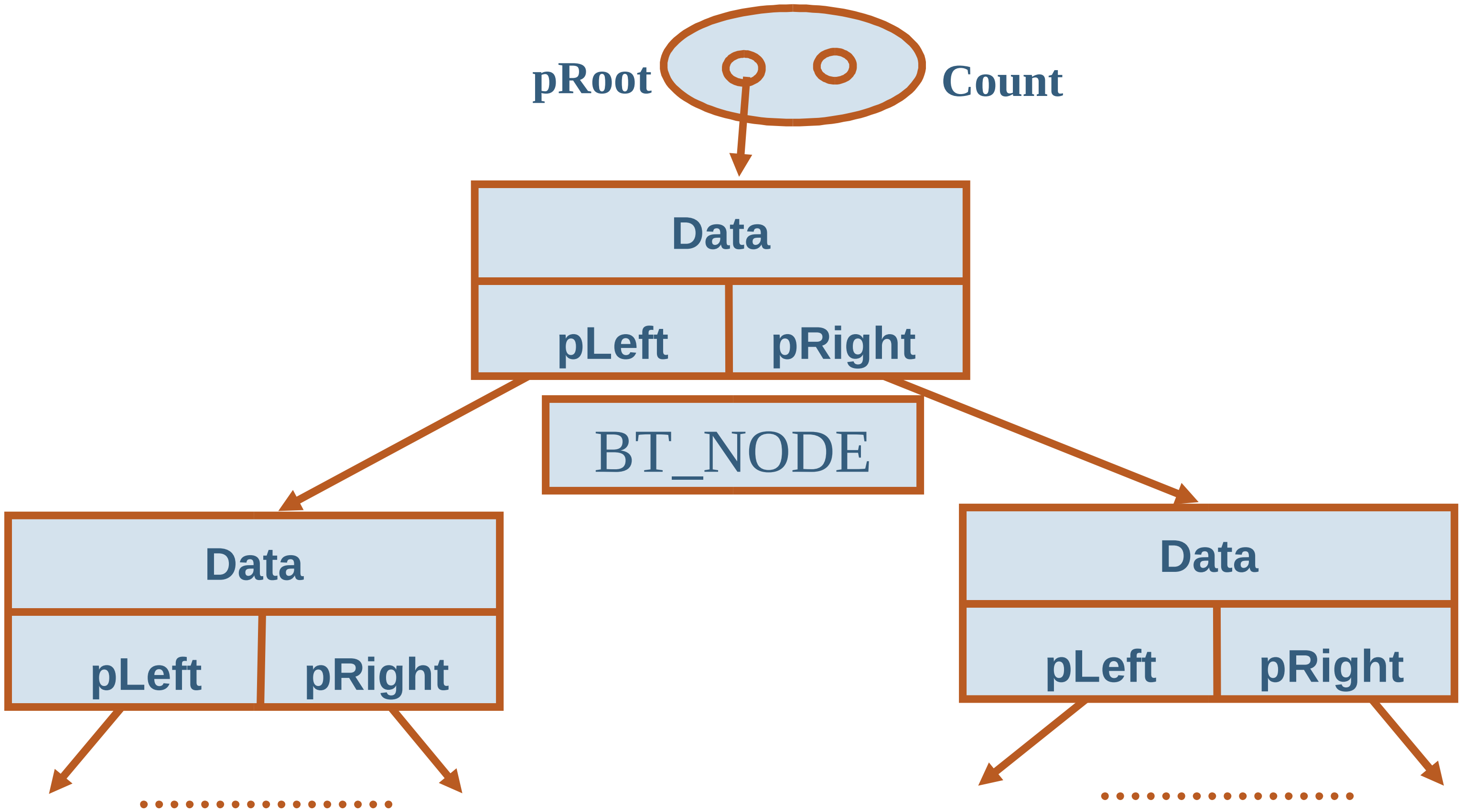
Binary Tree

- There are 2 ways to organize a binary tree:
  - Stored by array
  - Stored by structure pointers

#	Node	Left child	Right Child
0	*	1	2
1	-	3	4
2	/	5	6
3	a	-1	-1
4	b	-1	-1
5	c	-1	-1
6	d	-1	-1



Binary Tree



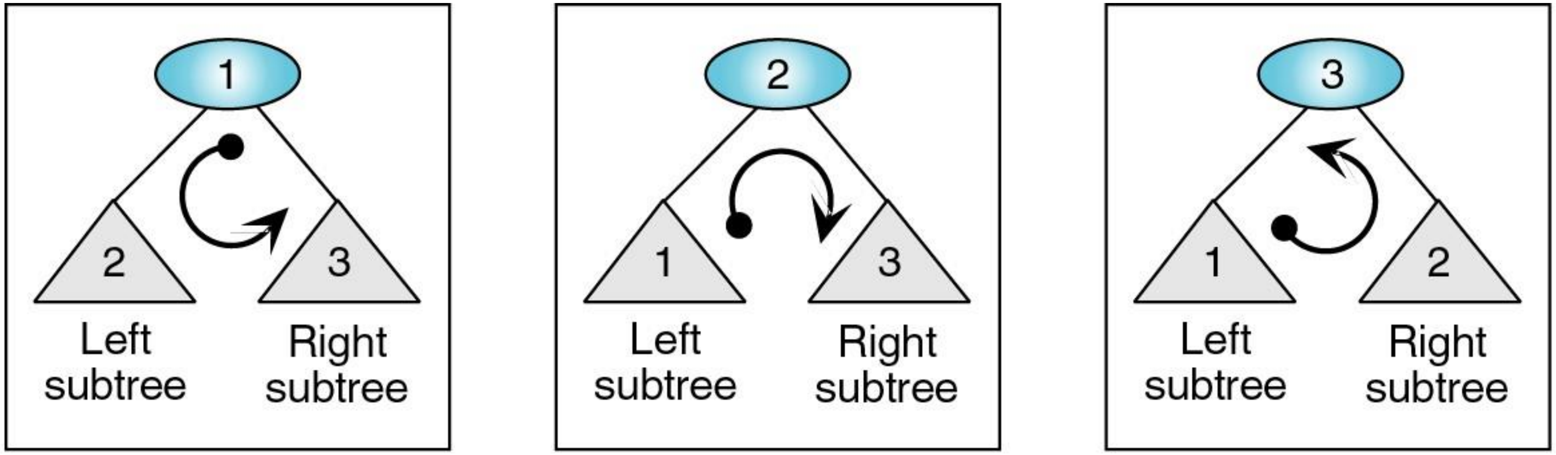
pRoot
Count
Data
pLeft
pRight
BT_NODE
Data
pLeft
pRight
Data
pLeft
pRight

Tree structure using pointers

```
typedef struct tagBT_NODE {
    int Data;
    tagBT_NODE *pLeft; //pointer to the left child node
    tagBT_NODE *pRight; //pointer to the right child node
} BT_NODE;             // binary tree node

typedef struct BIN_TREE {
   int     Count;   //Number of nodes in the tree
   BT_NODE *pRoot;  //the pointer to the root node
};     // binary tree
```

Traverse in Tree

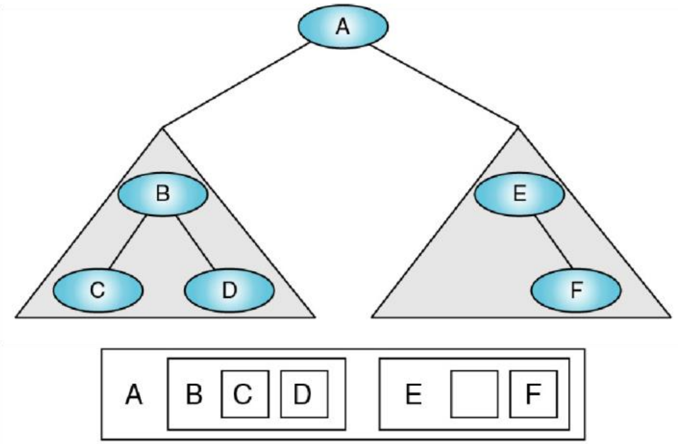
- There are 3 ways to traverse the tree:
  - Pre-Order (NLR)
  - In-Order (LNR)
  - Post-Order (LRN)



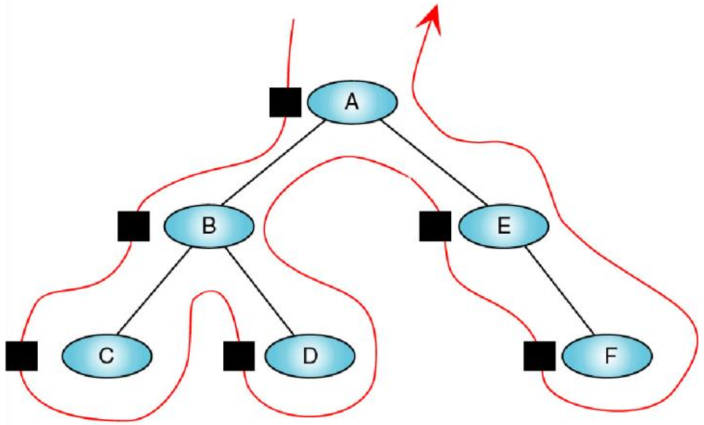
(a) Preorder traversal (b) Inorder traversal (c) Postorder traversal

Traverse in Tree - NLR

```
void NLR(const BT_NODE *pCurr)
{
   if (pCurr==NULL)
        return;
   "Do something at pCurr"
   NLR(pCurr->pLeft);
   NLR(pCurr->pRight);
}
```

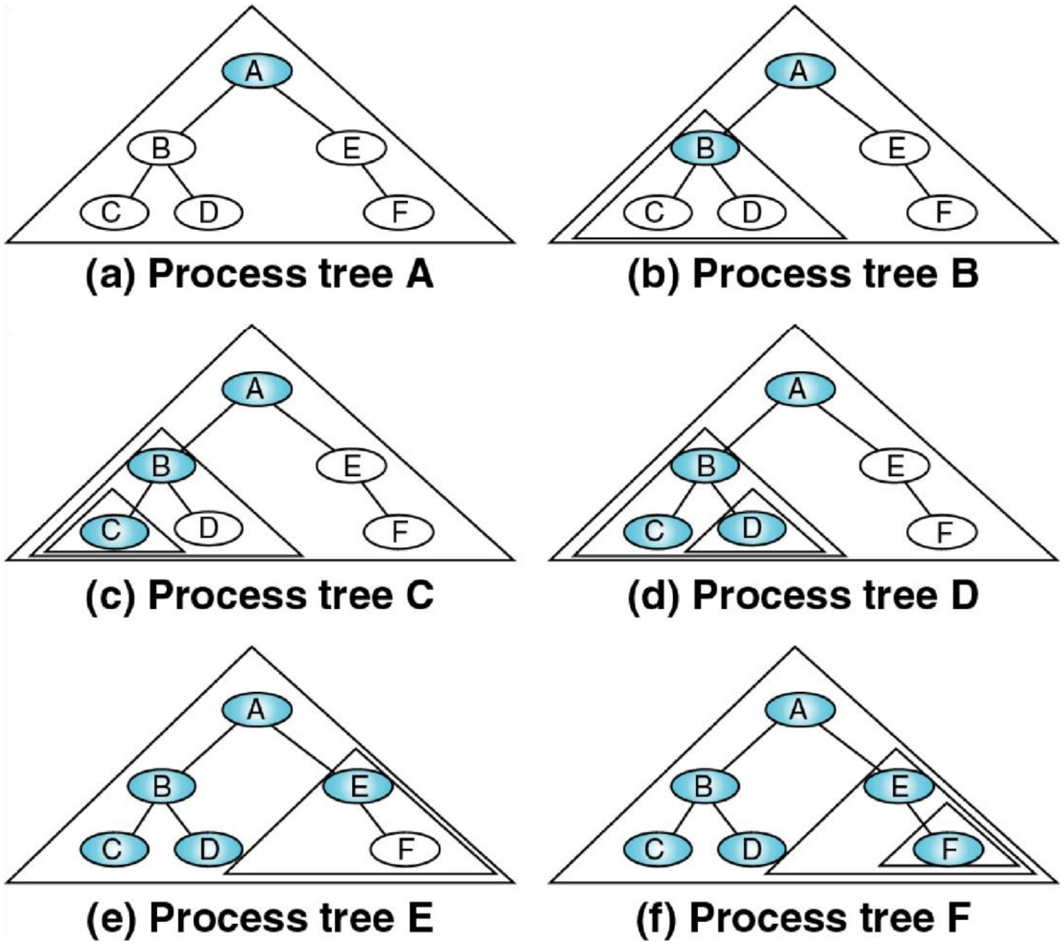


(a) Processing order



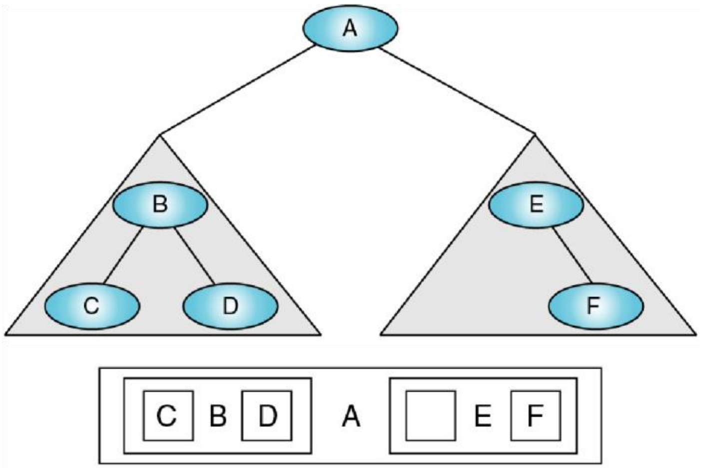
(b) "Walking" order

Traverse in Tree - NLR

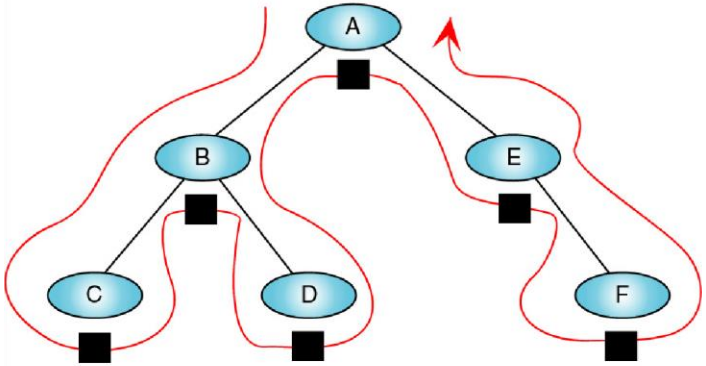


Traverse in Tree - LNR

```
void LNR(const BT_NODE *pCurr)
{
   if (pCurr==NULL)
        return;
   LNR(pCurr->pLeft);
   "Do something at pCurr"
   LNR(pCurr->pRight);
}
```



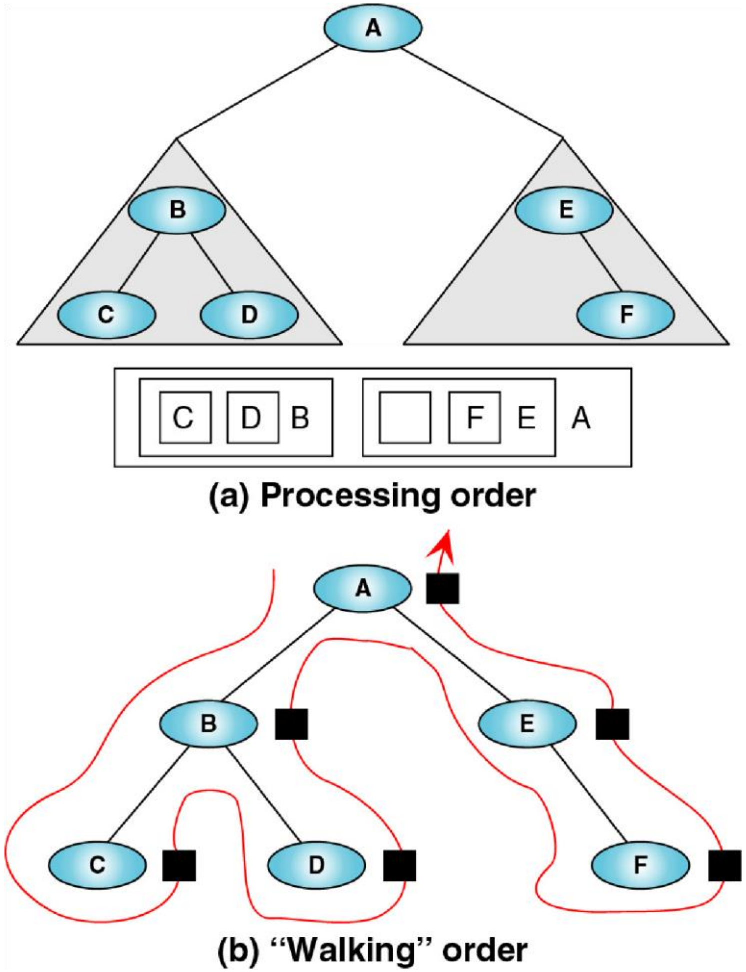
(a) Processing order



(b) "Walking" order

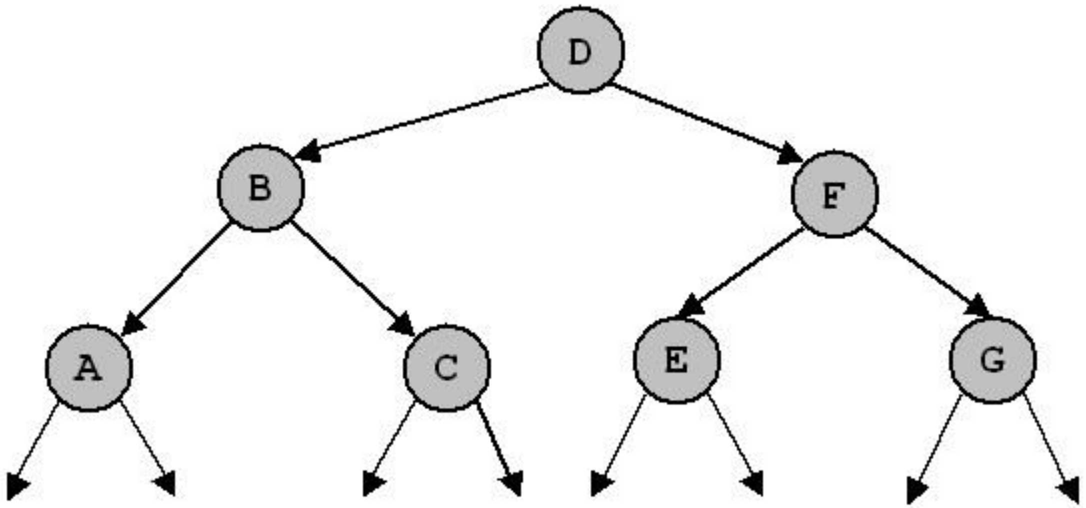
Traverse in Tree - LRN

```
void LRN(const BT_NODE *pCurr)
{
   if (pCurr==NULL)
        return;
   LRN(pCurr->pLeft);
   LRN(pCurr->pRight);
   "Do something at pCurr"
}
```



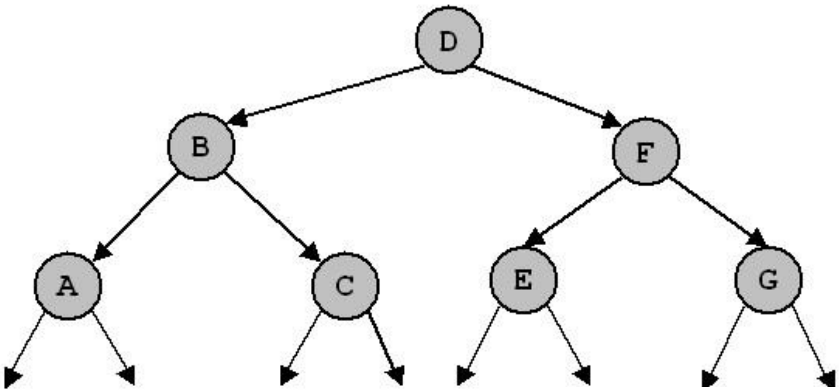
Quiz

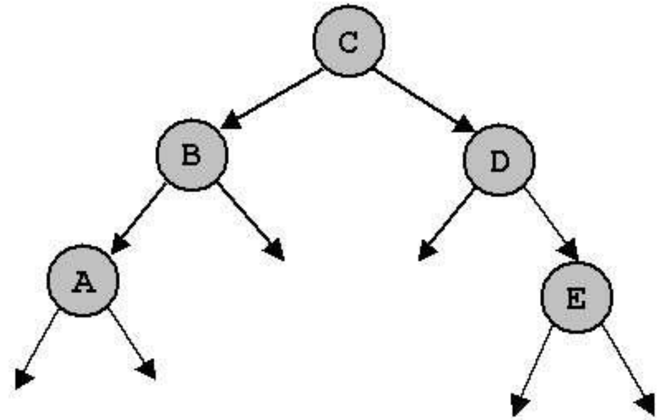
- Give the preorder, inorder, postorder, and level-order traversals of the following binary trees.

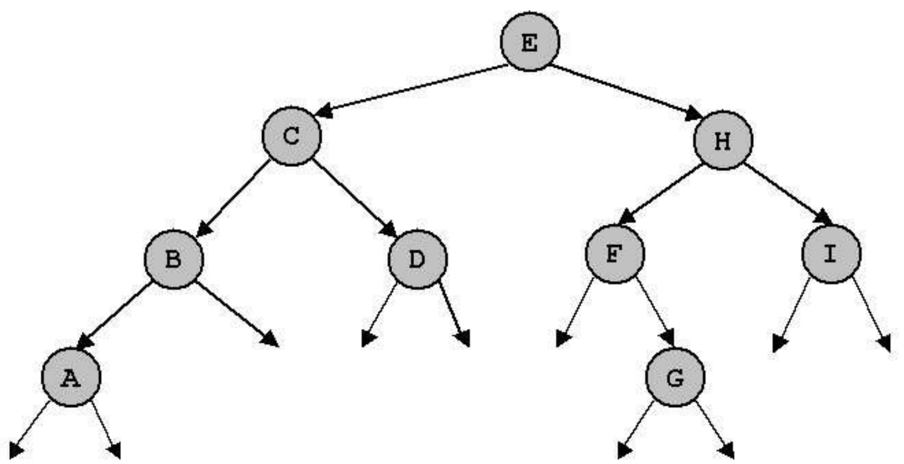


Quiz

- Give the preorder, inorder, postorder, and level-order traversals of the following binary trees.







Quiz

(a) Write a function that counts the number of items in a binary tree.

(b) Write a function that returns the sum of all the keys in a binary tree.

(c) Write a function that returns the maximum value of all the keys in a binary tree. Assume all values are nonnegative; return -1 if the tree is empty.

Quiz

(a) The height of a tree is the maximum number of nodes on a path from the root to a leaf node. Write a C function that returns the height of a binary tree.

(b) The cost of a path in a tree is sum of the keys of the nodes participating in that path. Write a C function that returns the cost of the most expensive path from the root to a leaf node.

Outline

- Tree
- Binary Tree
- **Binary Search Tree**
- Balanced Binary Search Tree
  - AVL

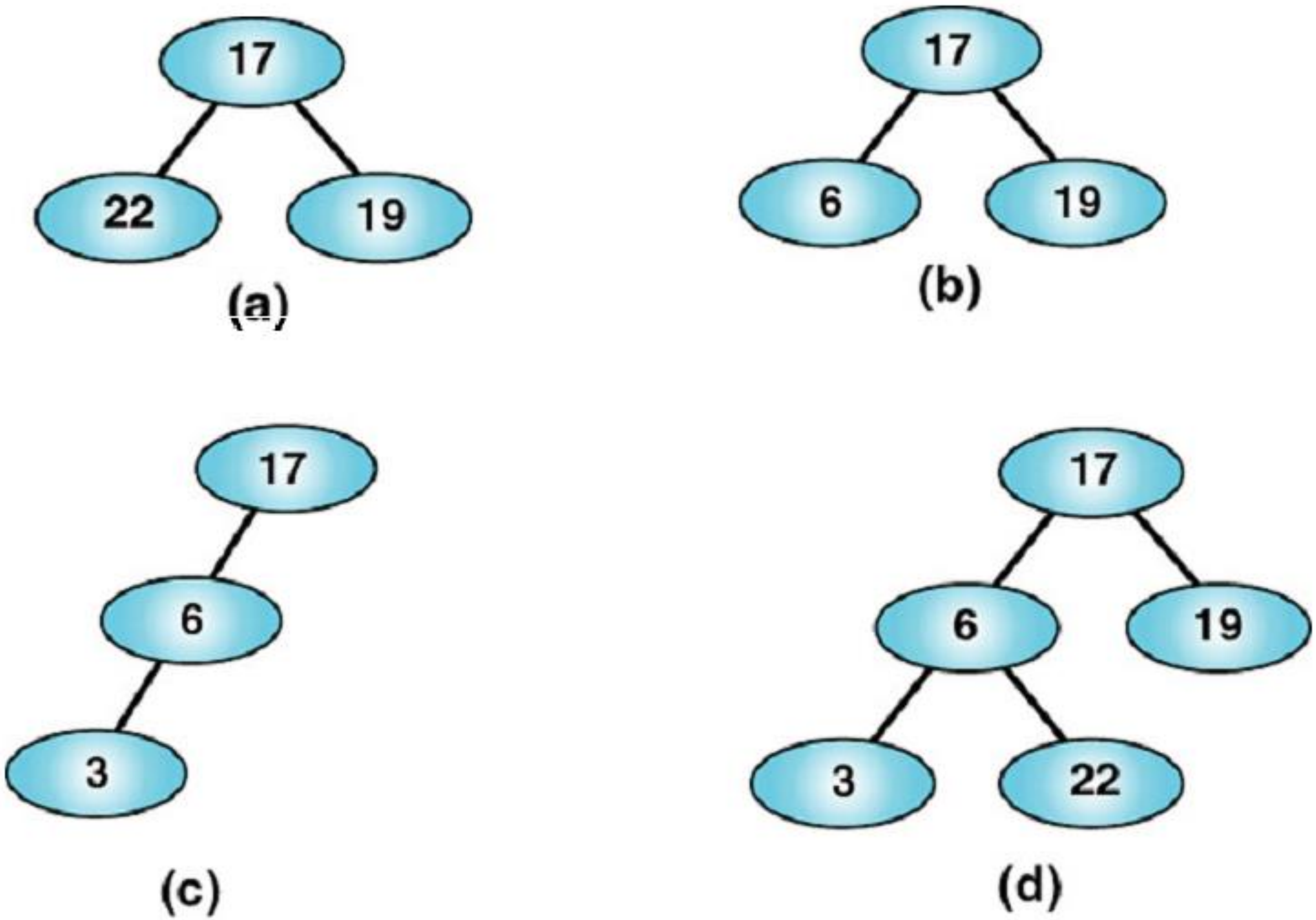
Binary Search Tree

- The binary search tree is:
  - A binary tree
  - Each node p of the tree satisfies:
    - All nodes in the left subtree (p-> pLeft) are less than the value of p

      $\forall q \in$ p-> pLeft: q-> Data <p-> Data
    - All nodes in the right subtree (p-> pRight) are greater than the value of p

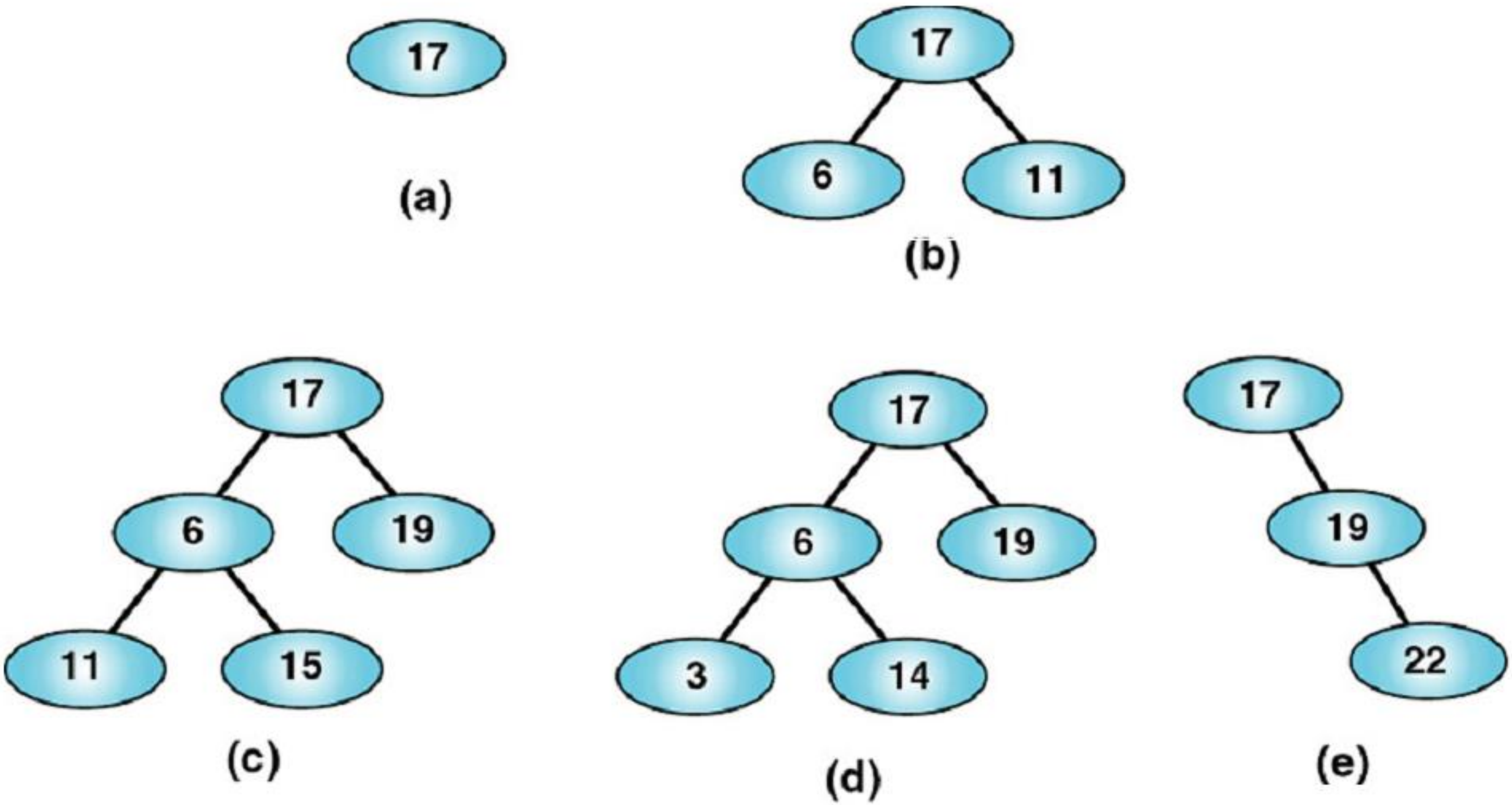
      $\forall q \in$ p-> pRight: q-> Data> p-> Data

Example



Which tree is Binary Search Tree (BST)?

Example



Which tree is Binary Search Tree (BST)?

Operations in BST

- Create a empty tree
- Check the empty tree
- Find an element
- Add 1 element
- Delete 1 element

Create and check empty trees

- Create a empty tree:

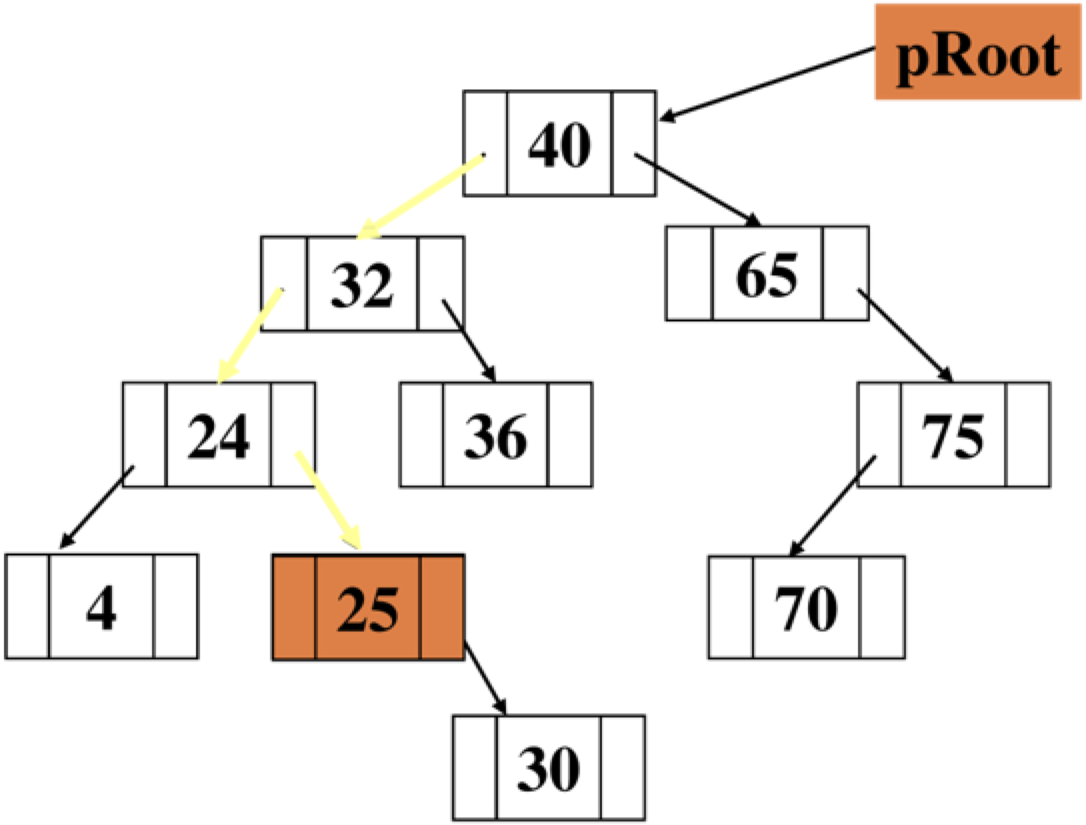
```
void BSTCreate(BIN_TREE &t)
{
    t.Count = 0;    // number of nodes in BST
    t.pRoot = NULL; // pointer of root node
}
```

- Check a empty tree:

```
int  BSTIsEmpty(const BIN_TREE &t)
{
    if (t.pRoot==NULL)
        return 1;
    return 0;
}
```

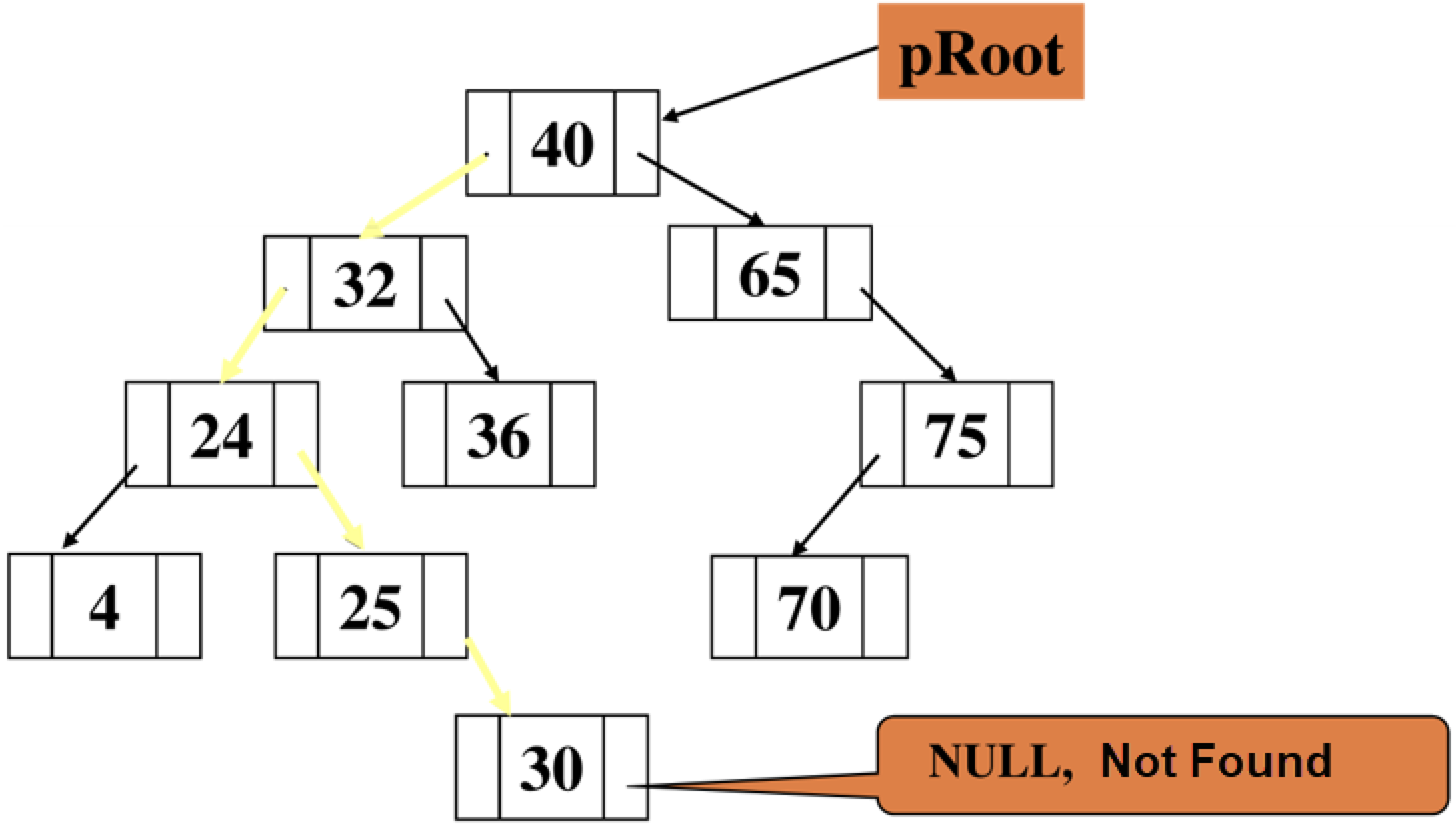
Search for an element

- Example search for element 25:



Search for an element

- Example search for element 31:

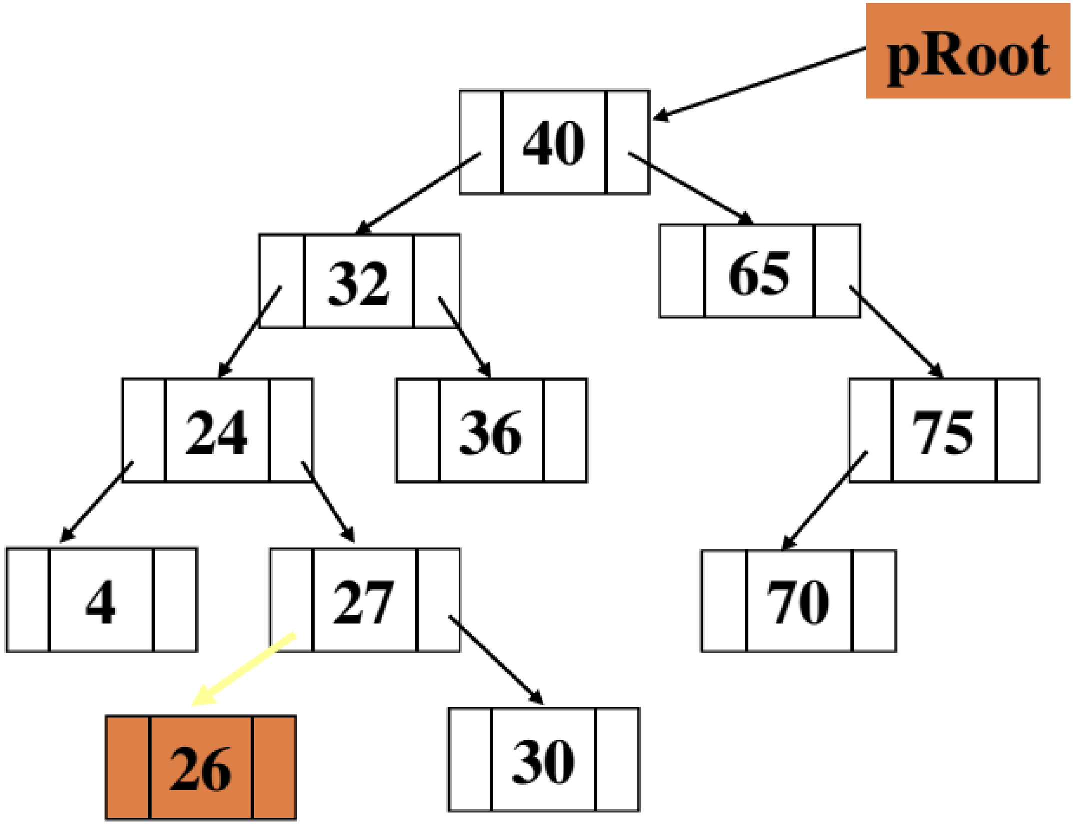


Search for an element

```
BT_NODE *BSTSearch(const BT_NODE *pCurr, int Key)
{
   if (pCurr==NULL)  return NULL; //Not Found
   if (pCurr->Data==Key) return pCurr; // Found
   else if (pCurr->Data > Key) // Search in left subtree
             return BSTSearch(pCurr->pLeft, Key);
      else // Search in right subtree
          return BSTSearch(pCurr->pRight, Key);
}
```

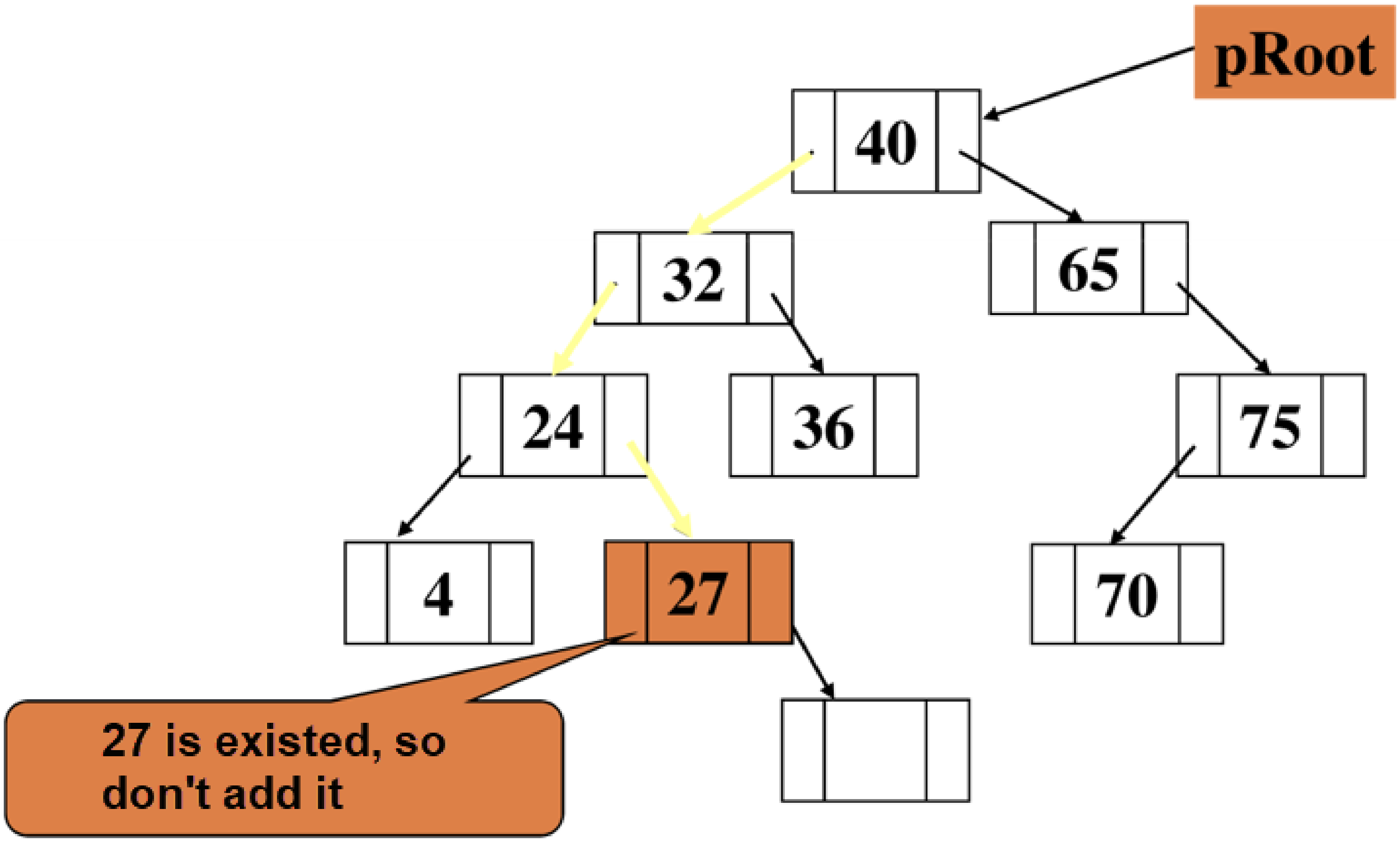
Add new element

- Example for adding element 26:



Add new element

- Example for adding element 27:



Add new element

```
int  BSTInsert(BT_NODE *&pCurr, int newKey)
{
        if (pCurr==NULL) {
                pCurr = new BT_NODE;  // Create new node
                pCurr->Data = newKey;
                pCurr->pLeft = pCurr->pRight = NULL;
                 return 1; // Success to add new element
        }
        if (pCurr->Data > newKey) // Add to left subtree
                return BSTInsert(pCurr->pLeft, newKey);
        else if (pCurr->Data < newKey)        // Add to right subtree
                return BSTInsert(pCurr->pRight, newKey);
        else return 0;  // Key is existed, don't add it
}
```

Quiz

Given a BST in pre-order as {13,5,3,2,11,7,19,23}, draw this BST and determine if this BST is the same as one described in post-order as {2,3,5,7,11,23,19,13}.

Quiz

(a) Insert items with the following keys (in the given order) into an initially empty binary search tree: 30, 40, 24, 58, 48, 26, 11, 13. Draw the tree after any two insertions.

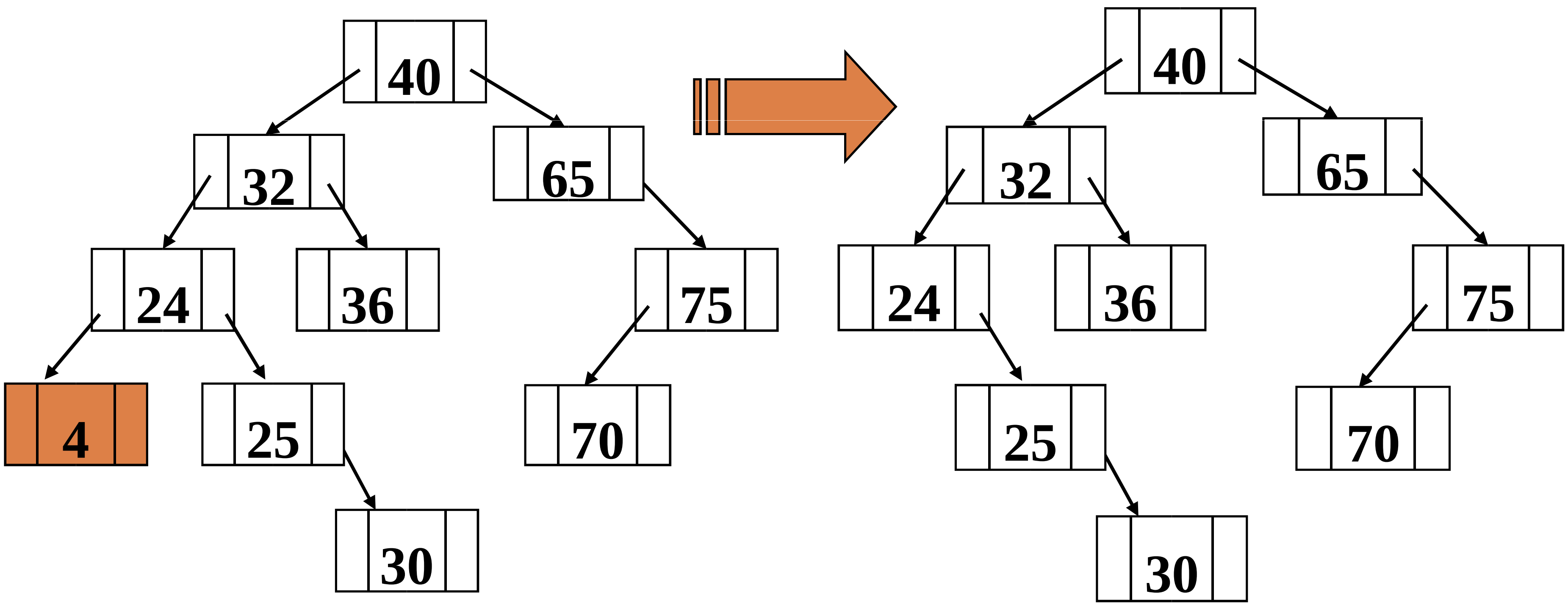
(b) Choose a set of 7 distinct, positive, integer keys. Draw binary search trees for your set of height 2, 5, and 6.

Delete an element

- Operation to delete an element:
  - Apply a search algorithm to determine which node contains the element to be deleted
  - If found, delete the element from the tree.
    - Delete node without any child node
    - Delete node with 1 child node
    - Delete node with 2 children

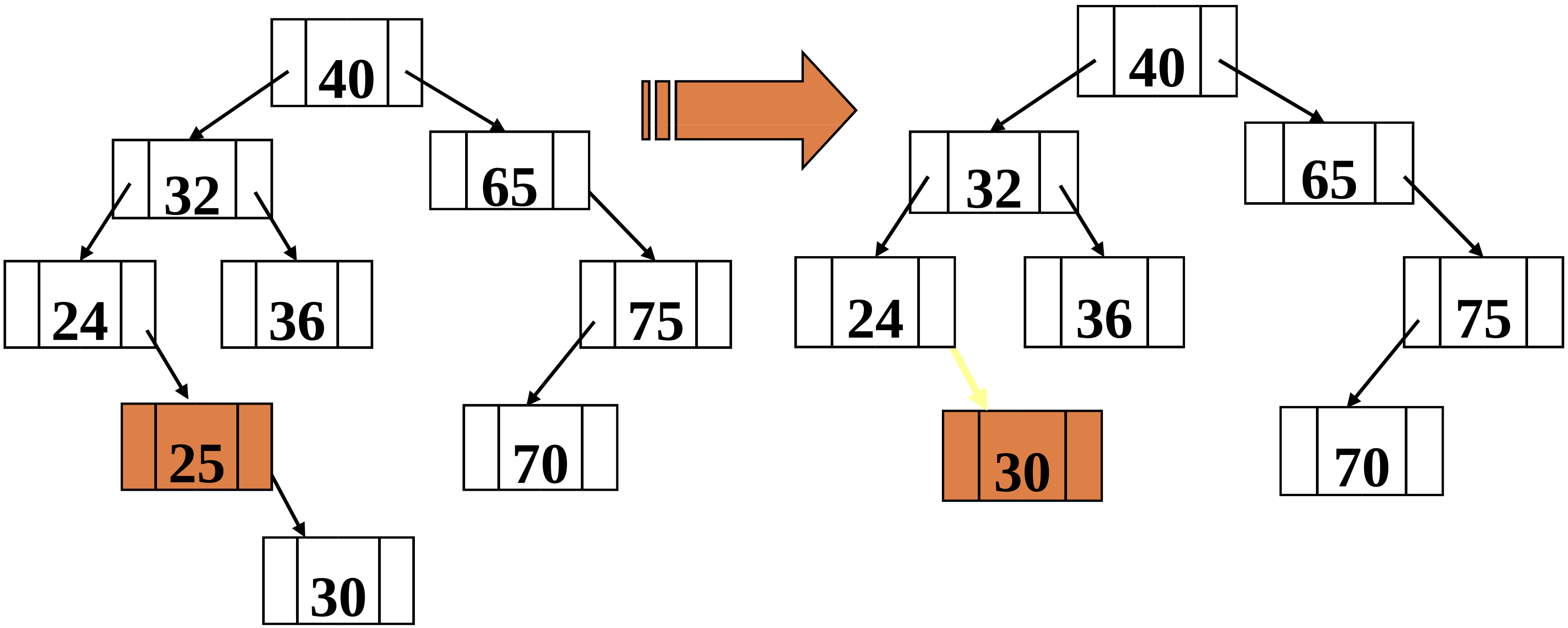
Delete an element without child

- Example of deleting element 4 (without child nodes)



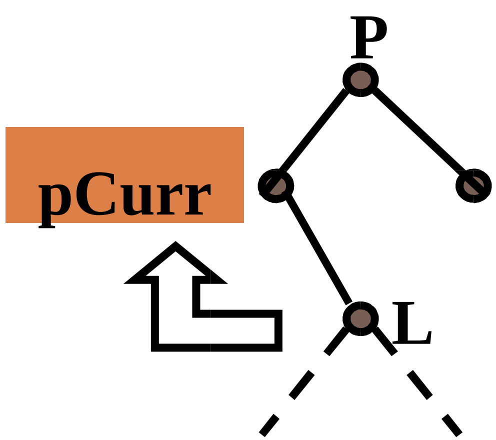
Delete an element with right child

- Example of deleting element 25 (with a right child node

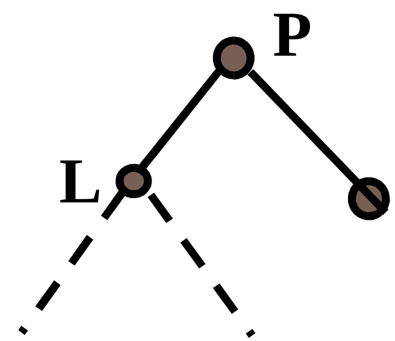


Delete an element with right child

- Delete node with only the right child node



Before delete pCurr

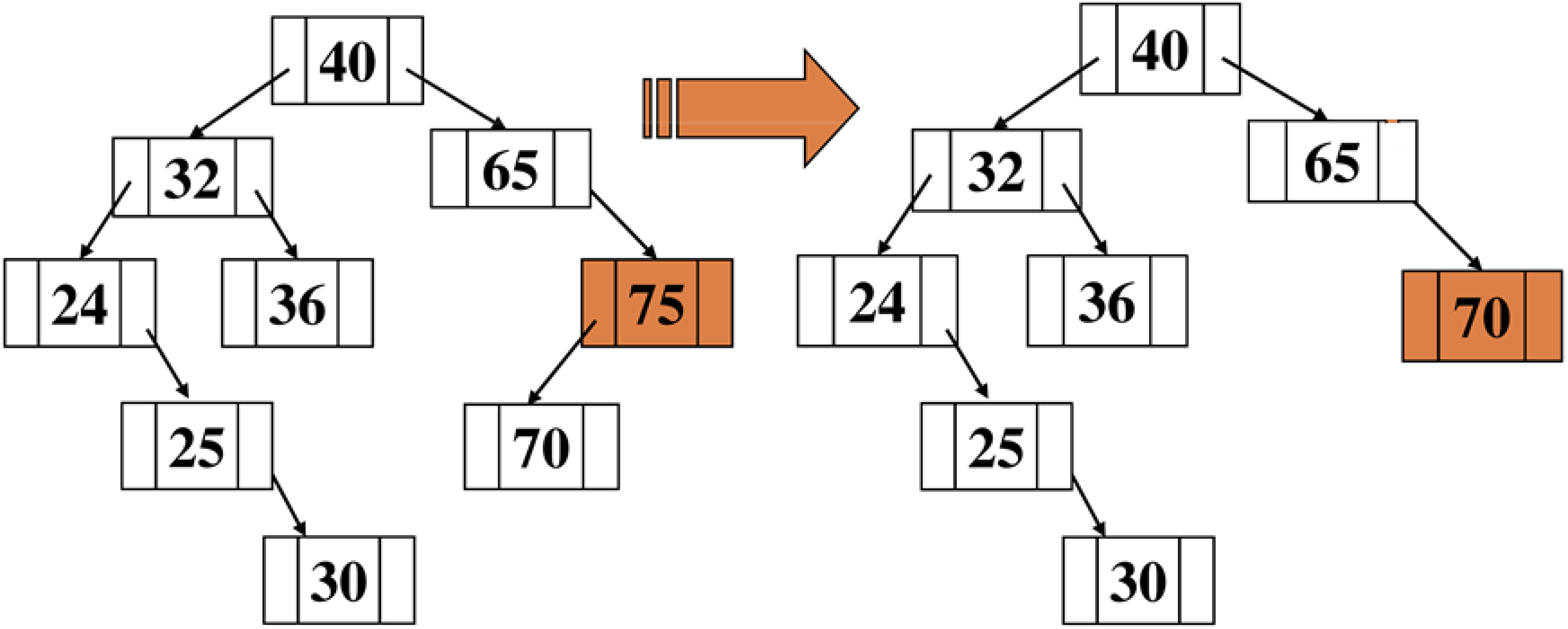


After delete pCurr

```
P->pLeft = pCurr->pRight;
delete pCurr;
```

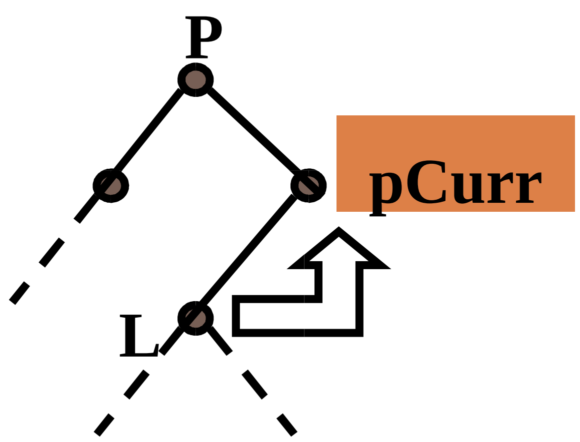
Delete an element with left child

- Example of deleting element 75 (with a left child node

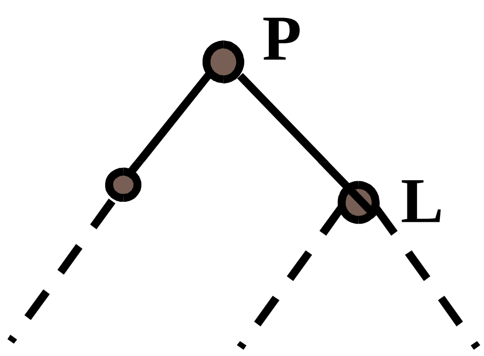


Delete an element with left child

- Delete node with only the left child node



Before delete pCurr

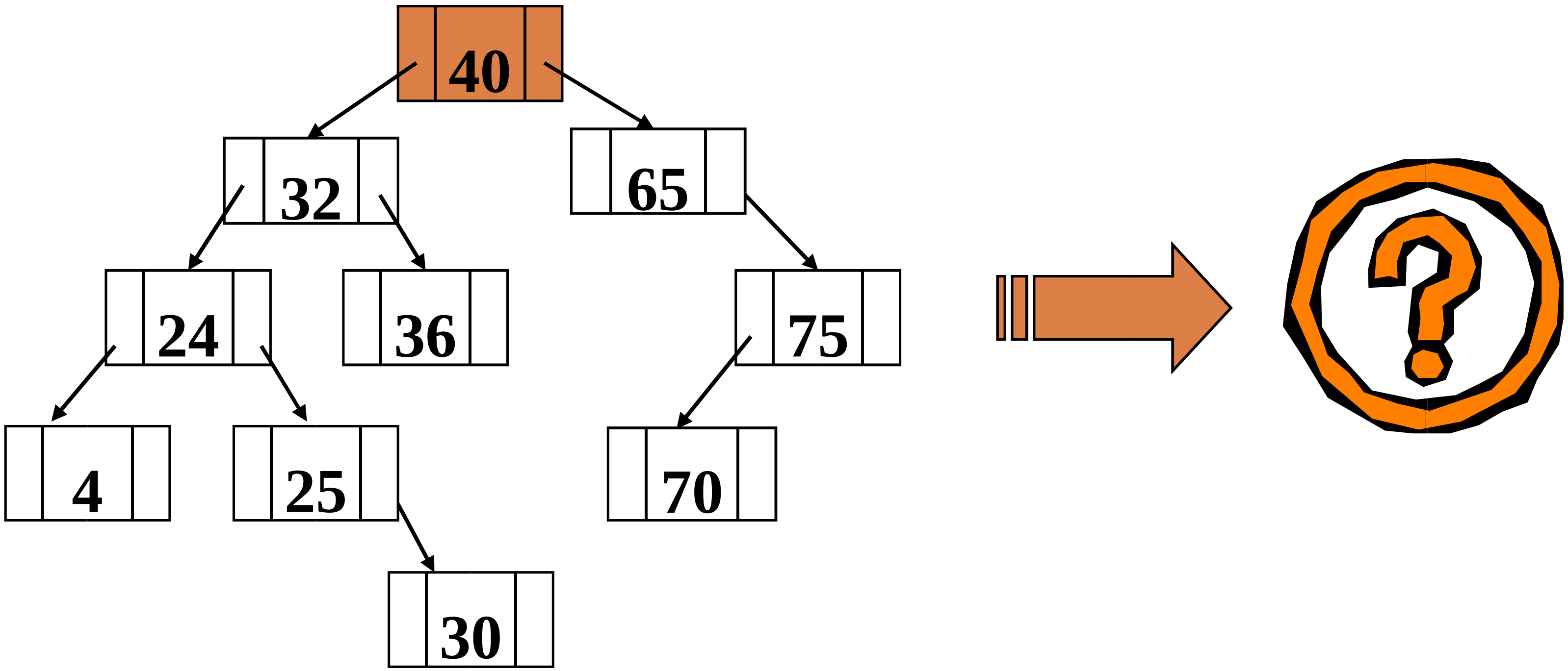


After delete pCurr

```
P->pRight = pCurr->pLeft;
delete pCurr;
```

Delete an element with two children

- Example of deleting element 40 (with 2 children)



Delete an element with two children

- Delete element pCurr with 2 child nodes:
  - Instead of deleting the pCurr node directly ...

    ... we find an element to replace p,

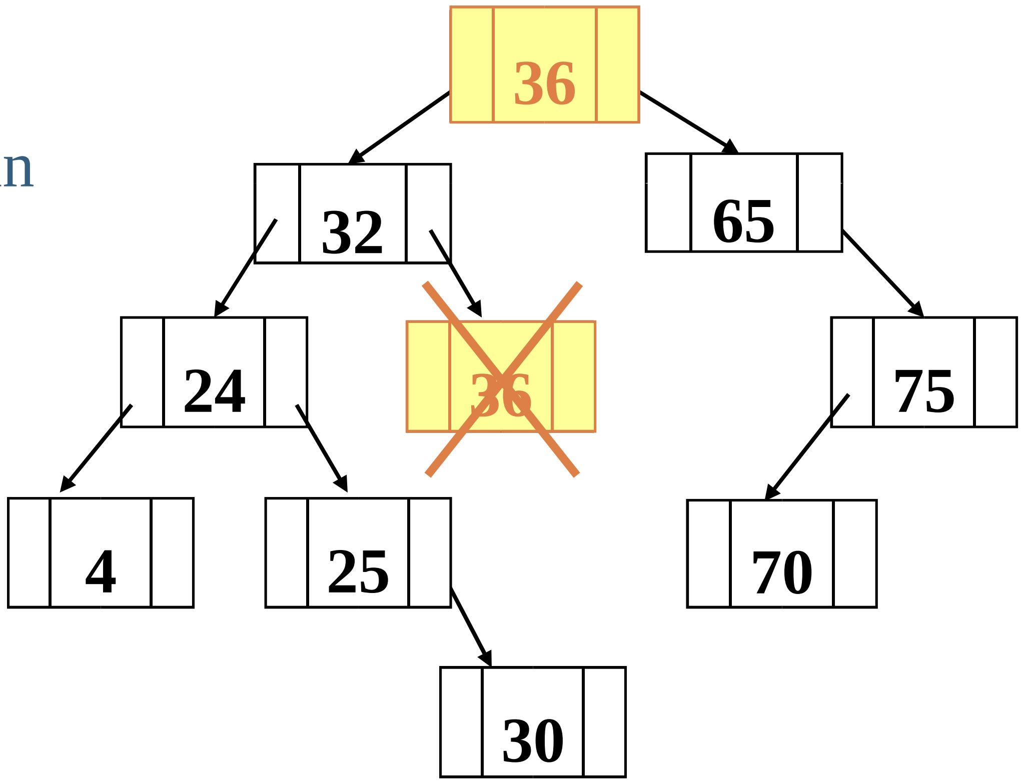
    ... copy data of p to pCurr,

    ... delete node p.
- Substitute element p:
  - is the largest element in the left subtree; or…
  - is the smallest element in the right subtree

Delete an element with two children

- Delete element 40 (with 2 children):

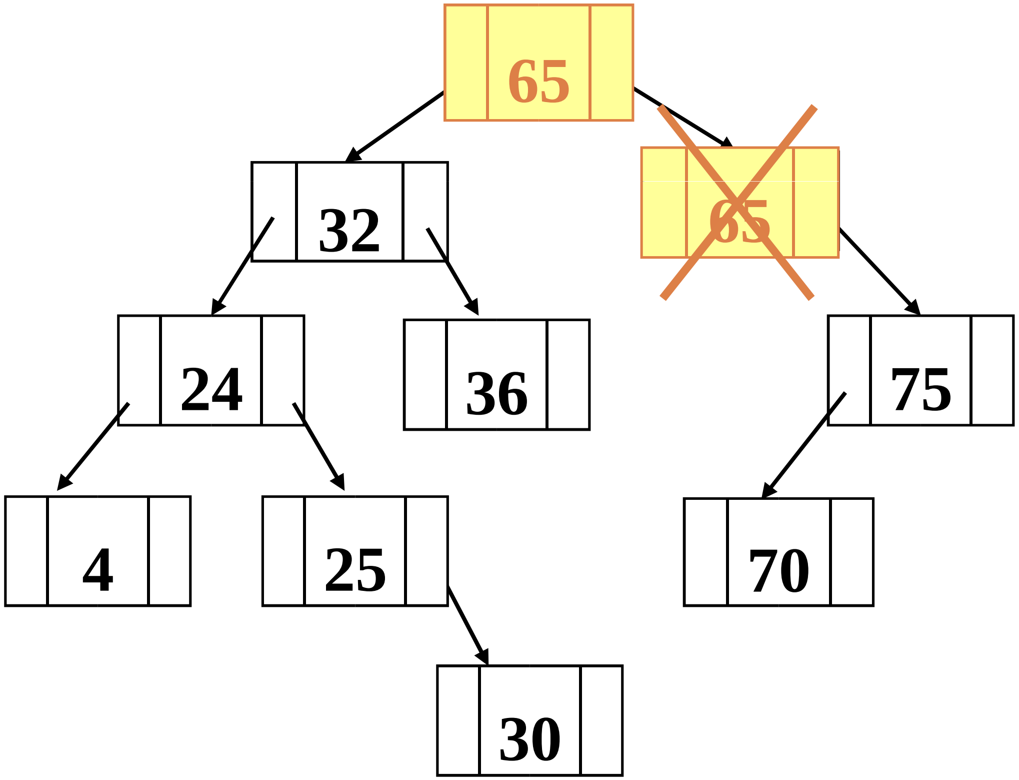
Way 1: substitute the largest element in the left subtree



Delete an element with two children

- Delete element 40 (with 2 children):

Way 2: substitute the smallest element in the right subtree



Delete an element with two children

```
int  BSTDelete(BT_NODE *&pCurr, int Key)
{
        if (pCurr==NULL)   return 0; // Not Found
        if (pCurr->Data > Key) // Find the element on left subtree
             return BSTDelete(pCurr->pLeft, Key);
        else if (pCurr->Data < Key) // Find the element on right subtree
             return BSTDelete(pCurr->pRight, Key);

        // Found node to delete (pCurr)
        _Delete(pCurr);
        return 1;
 }
```

Delete an element with two children

```
void _Delete(BT_NODE *&pCurr)
{
        BT_NODE  *pTemp = pCurr;
        if (pCurr->pRight==NULL)   // Only a left child node
              pCurr = pCurr->pLeft;
        else if (pCurr->pLeft==NULL) // Only a right child node
              pCurr = pCurr->pRight;
        else  // With 2 children
                pTemp = _SearchStandFor(pCurr->pLeft, pCurr);

        delete pTemp;
}
```

Delete an element with two children

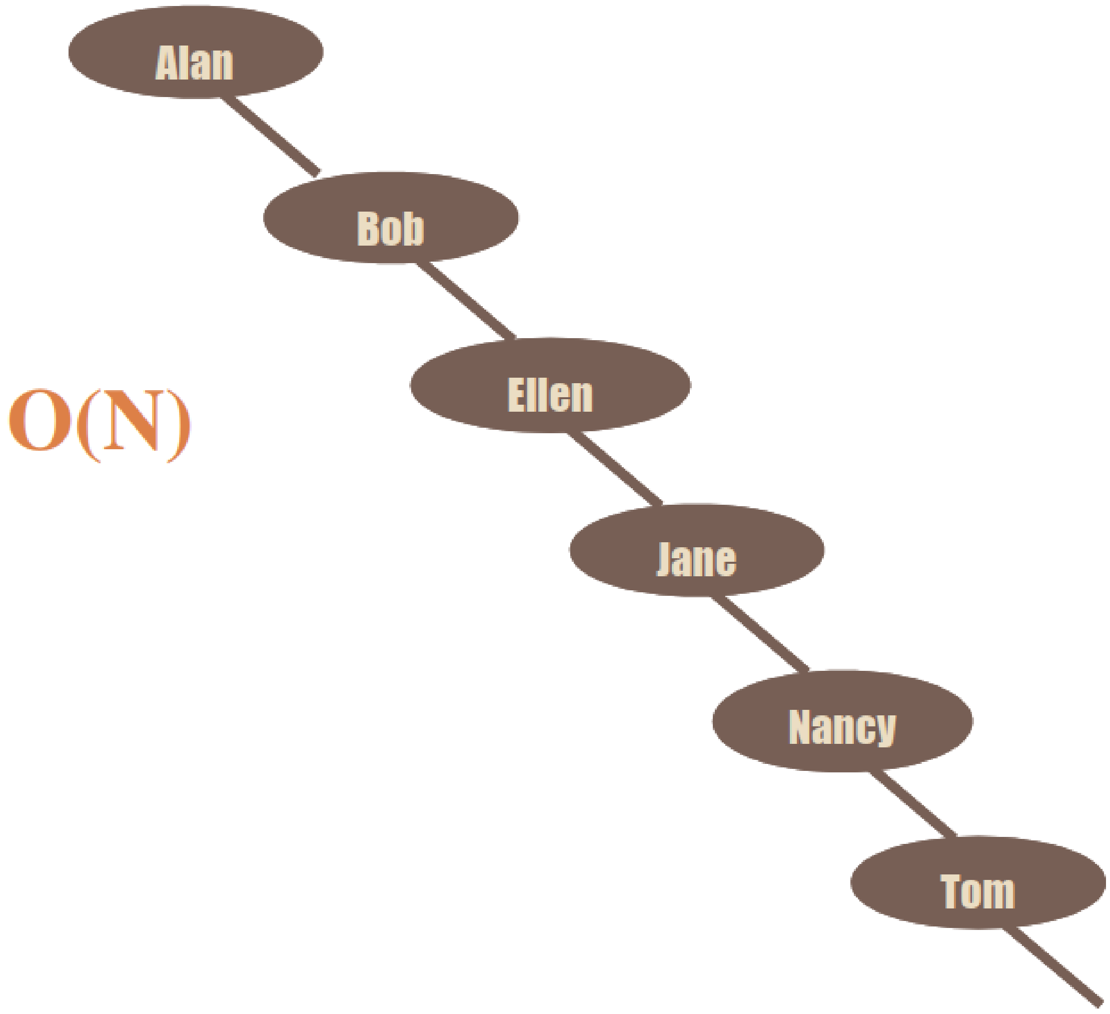
```
BT_NODE *  _SearchStandFor(BT_NODE *&p, BT_NODE *pCurr)
{
        //Find the element to substitute
        if (p->pRight != NULL)
                return _SearchStandFor(p->pRight, pCurr);

        //Substitute
        pCurr->Data = p->Data;                  // Copy data from p to pCurr
        BT_NODE *pTemp = p;
        p = p->pLeft;                           // Save the left sub-branch
        return  pTemp;                          // Delete substituted element
}
```

Outline

- Tree
- Binary Tree
- Binary Search Tree
- Balanced Binary Search Tree
  - AVL

Why need tree balance?

- The BST tree can be unbalanced



Some trees are balanced

- AVL Tree
- Red-Black Tree
- AA Tree
- Splay Tree
- …

AVL

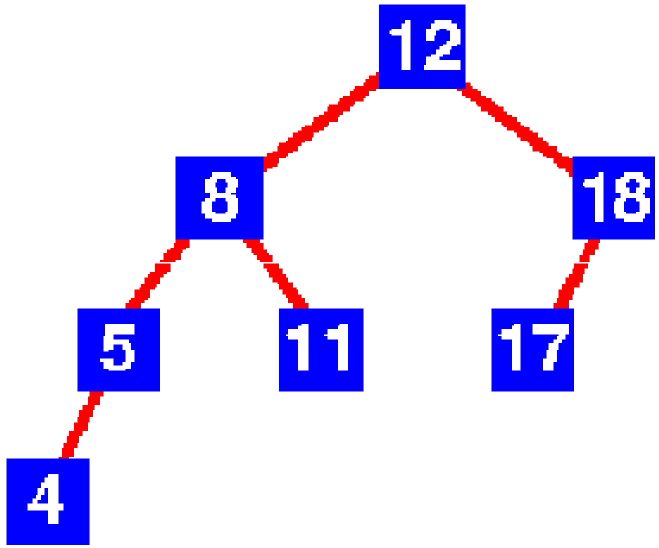
- AVL tree is a balanced BST tree
- AVL tree created by 3 authors: Adelson, Velskii, Landis proposed in 1962
- This is the first proposed dynamic balanced tree model
- The AVL tree does not have "absolute" balance, but the two child-tree never have a height difference of more than 1

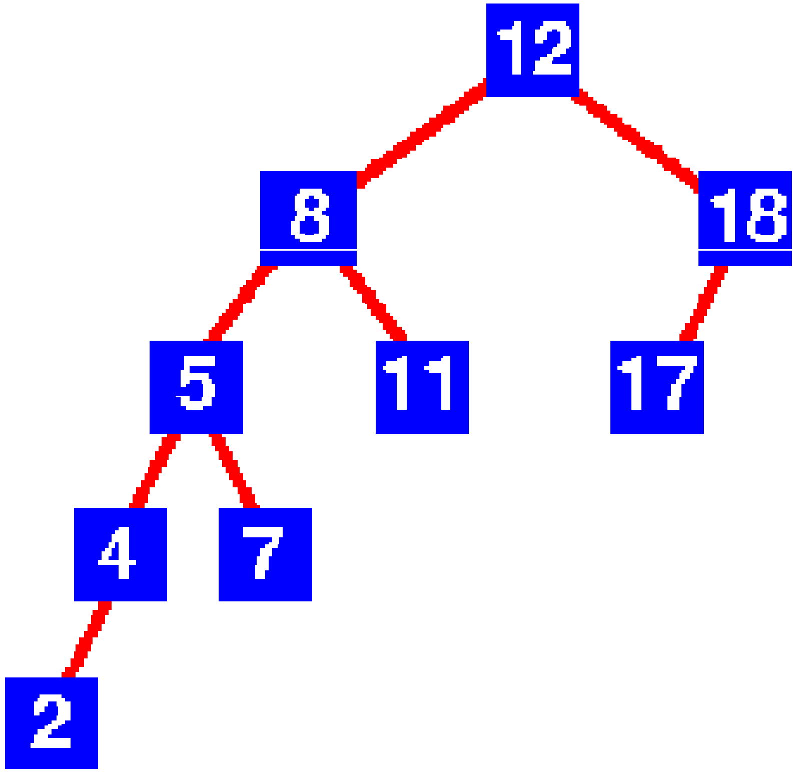
AVL

- The AVL tree is:
  - A search binary tree
  - Each node p of the tree is satisfactory:
    - the height of the left subtree (p-> pLeft) and the height of the right subtree (p-> pRight) differ by no more than 1.

$$\forall p \in TAVL\text{: } abs\ (hp -> \ pLeft \ - \ hp -> \ pRight) \ \leq \ 1$$

Example



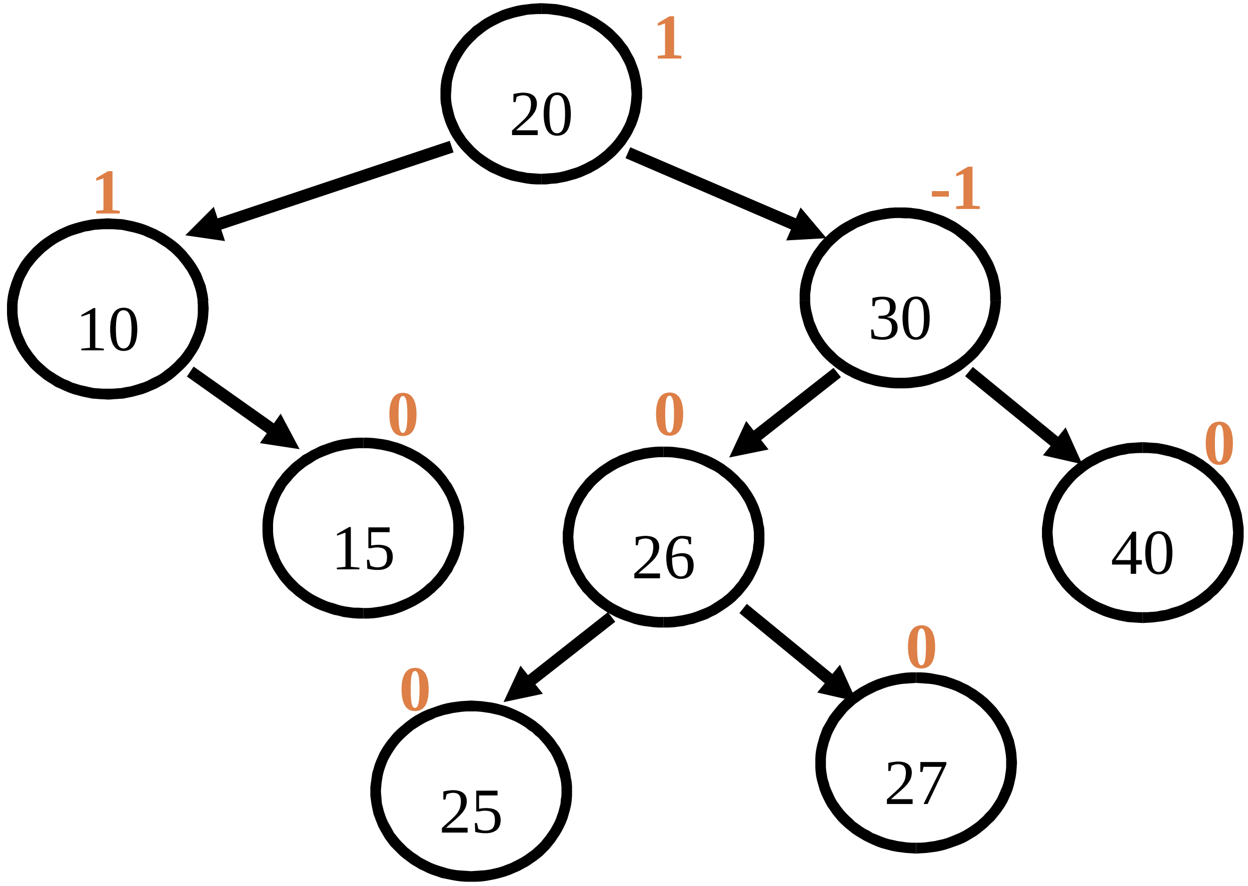


Which tree is AVL?

Balance

- Add each node in the tree a Bal field, expressing the state of that node:
  - Bal = -1: node deviated left (the left subtree is higher than the right subtree)
  - Bal = 0: balance node (the left subtree is as high as the right subtree)
  - Bal = +1: node deviates right (the right subtree is higher than the left subtree)

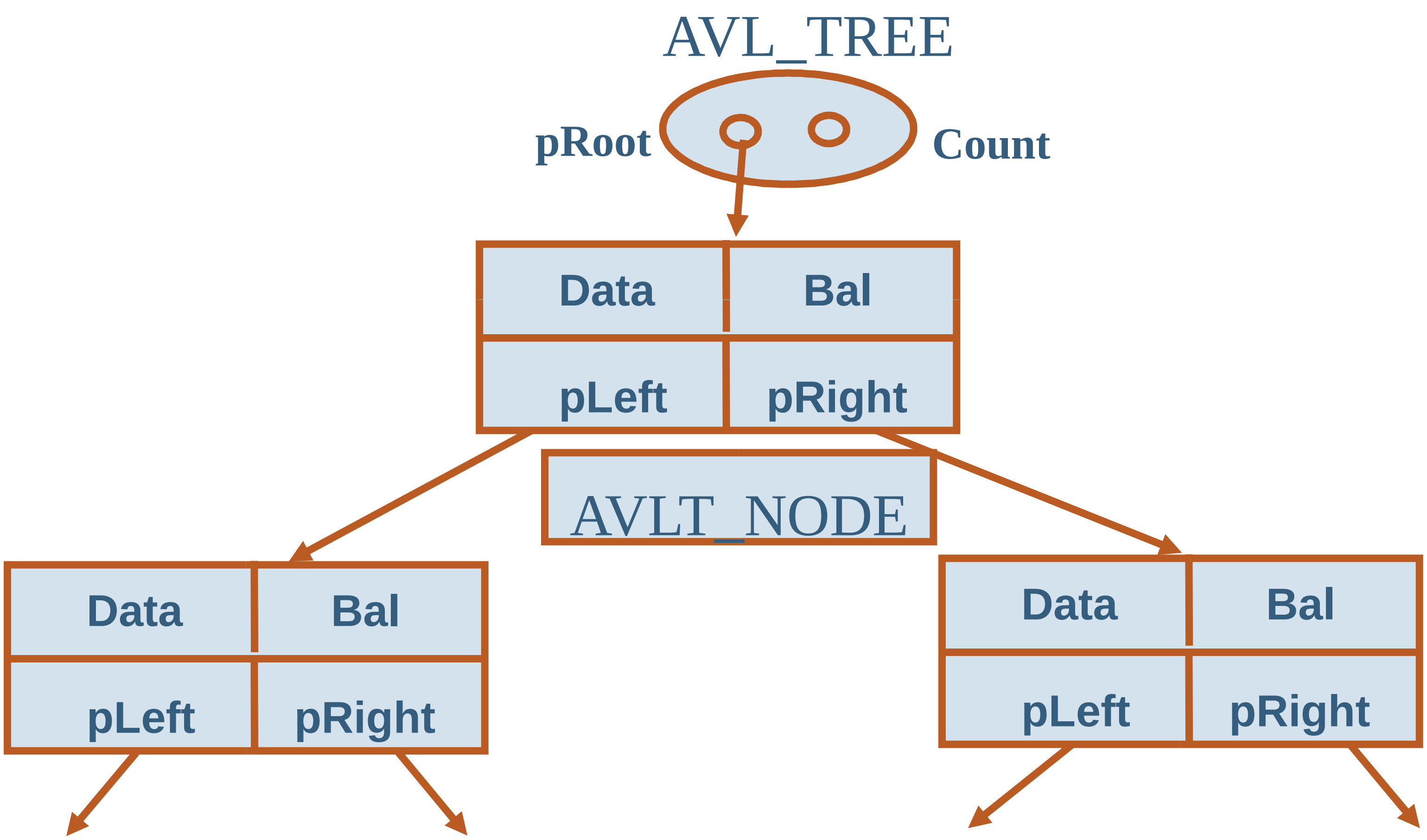
Balance



Balance

```
typedef struct tagAVLT_NODE {
    int         Data;
    int         Bal;        // Balance (-1,0,1)
    tagBT_NODE      *pLeft;
    tagBT_NODE      *pRight;
} AVLT_NODE;
```

AVL Tree


AVL_TREE
pRoot
Count
Data
Bal
pLeft
pRight
AVLT_NODE
Data
Bal
pLeft
pRight
Data
Bal
pLeft
pRight

Operations that make the tree unbalanced

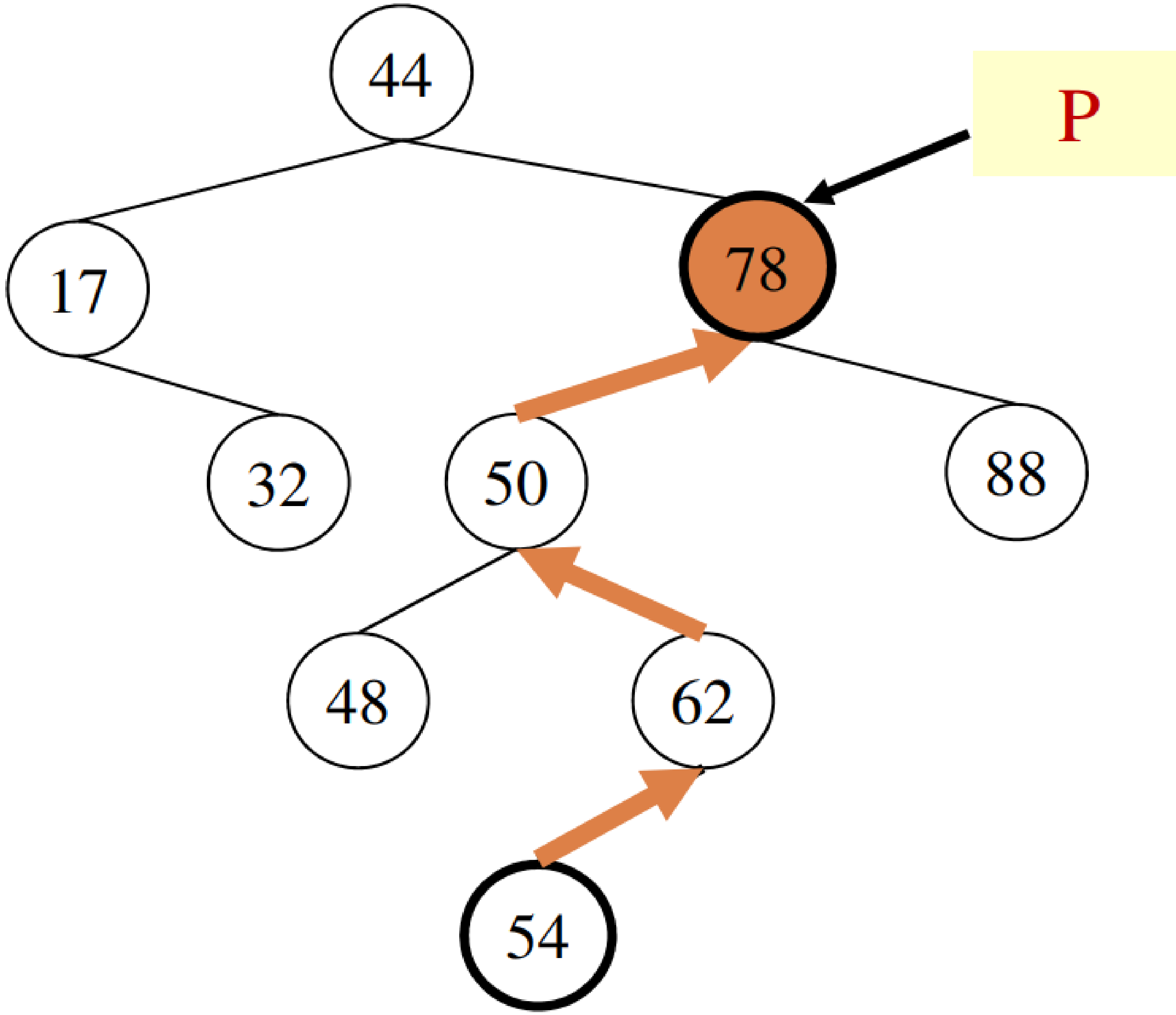
- Add an element
- Delete an element

Find the unbalance node

- Traverse from the newly added node back to the root node.
- If there is an unbalanced node, perform tree adjustment at that node.
- Adjustment can cause the nodes above to become unbalanced, so we need to adjust until no nodes are unbalanced.

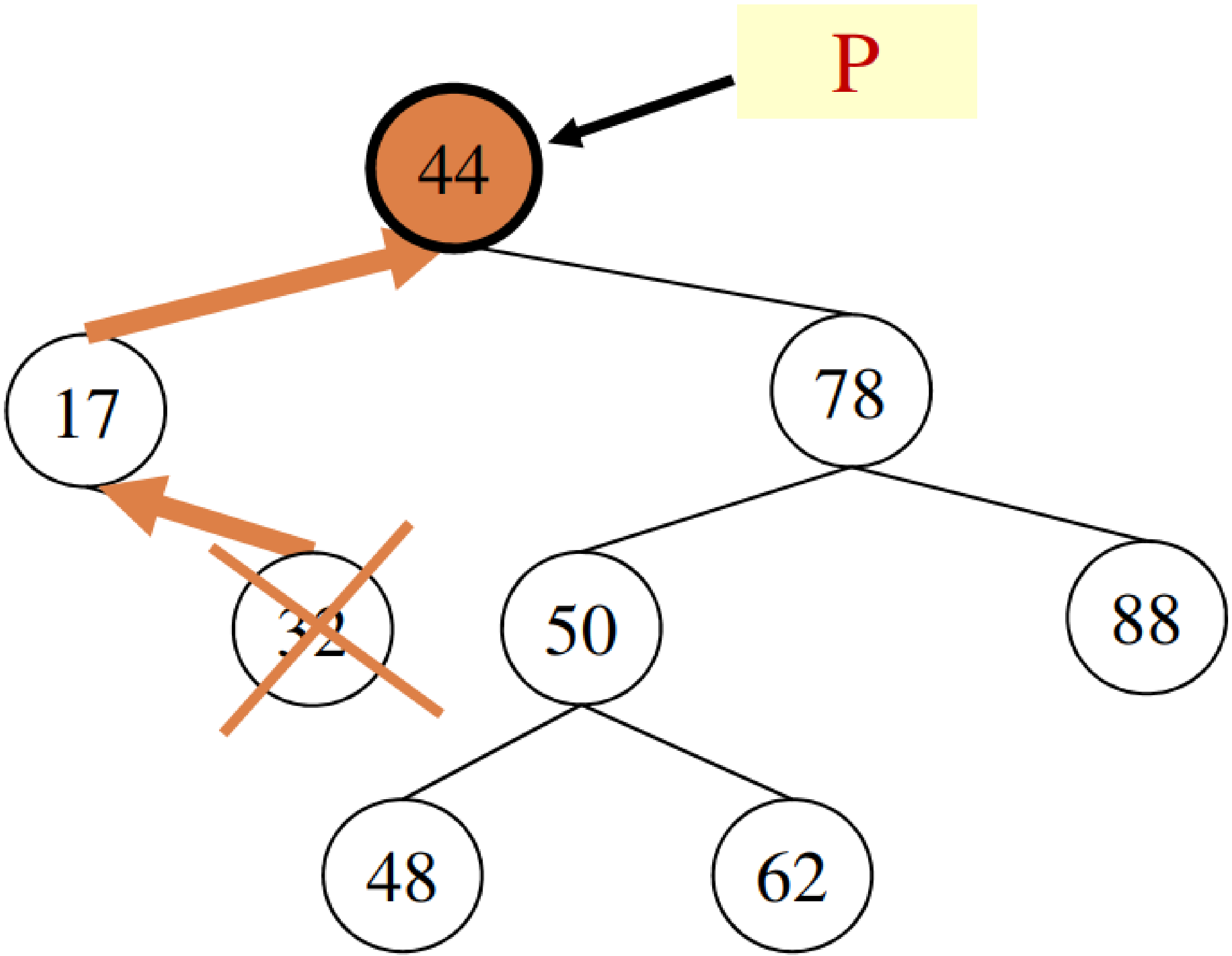
Find the unbalance node

- Adding new element make tree unbalance.

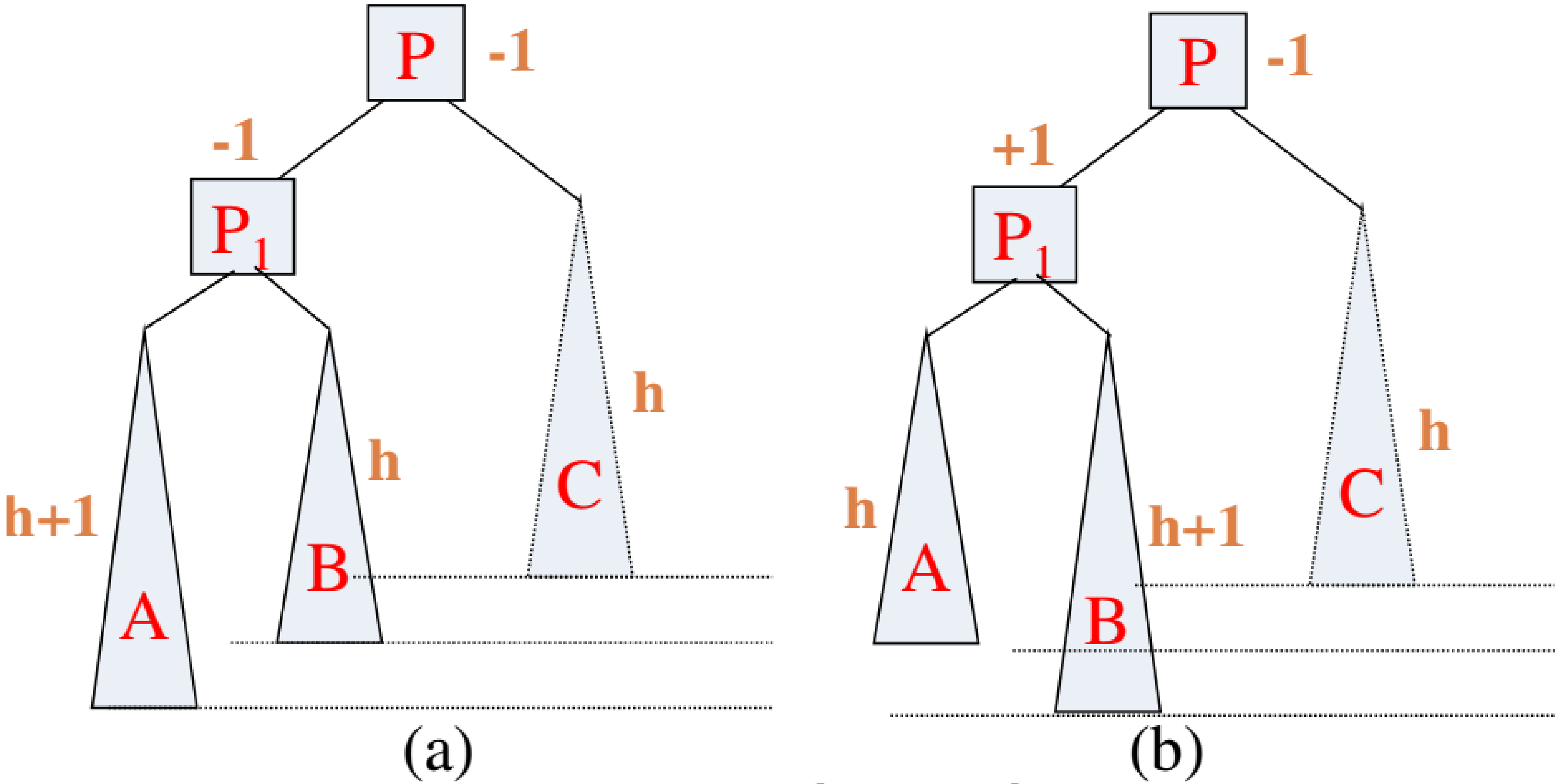


Find the unbalance node

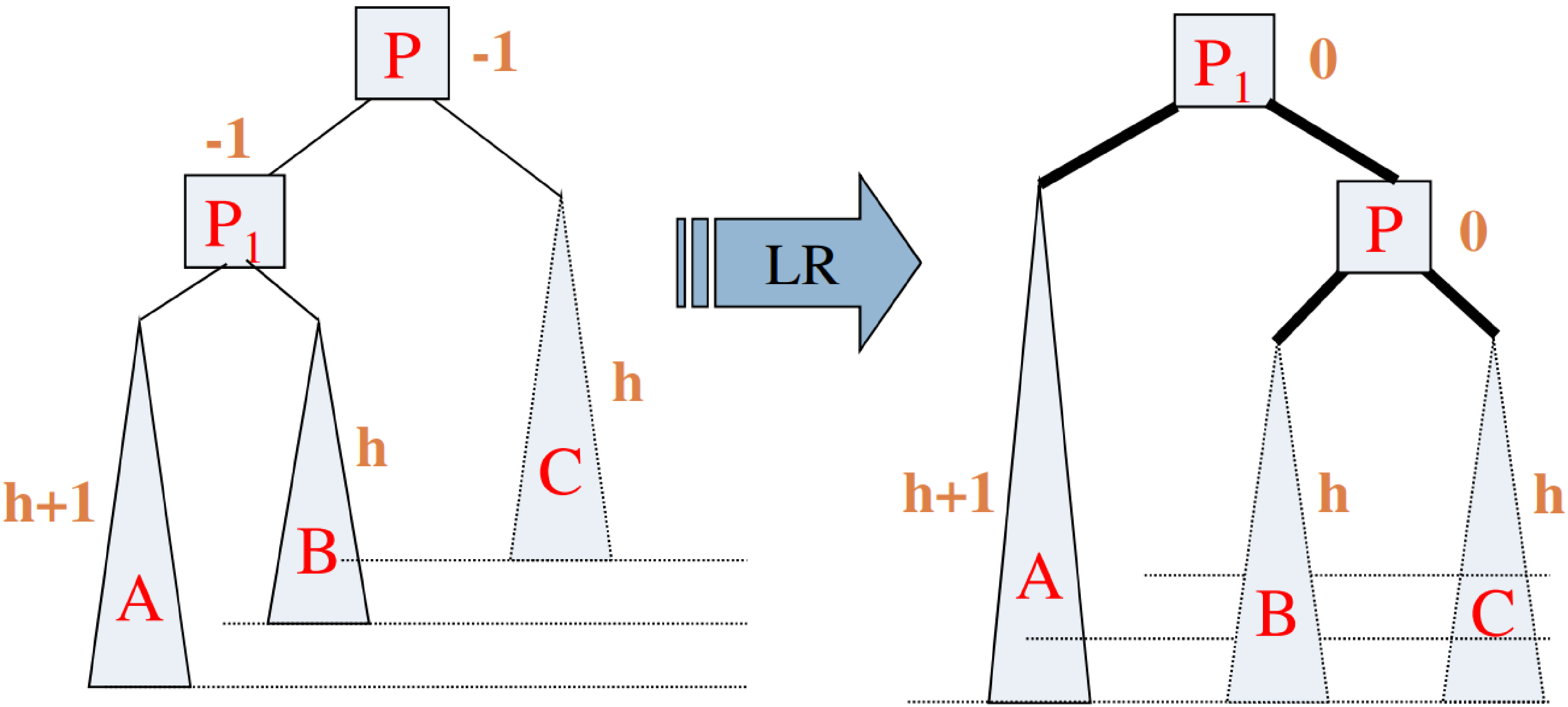
- Deleting an element make tree unbalance



Adjust the tree that is left off

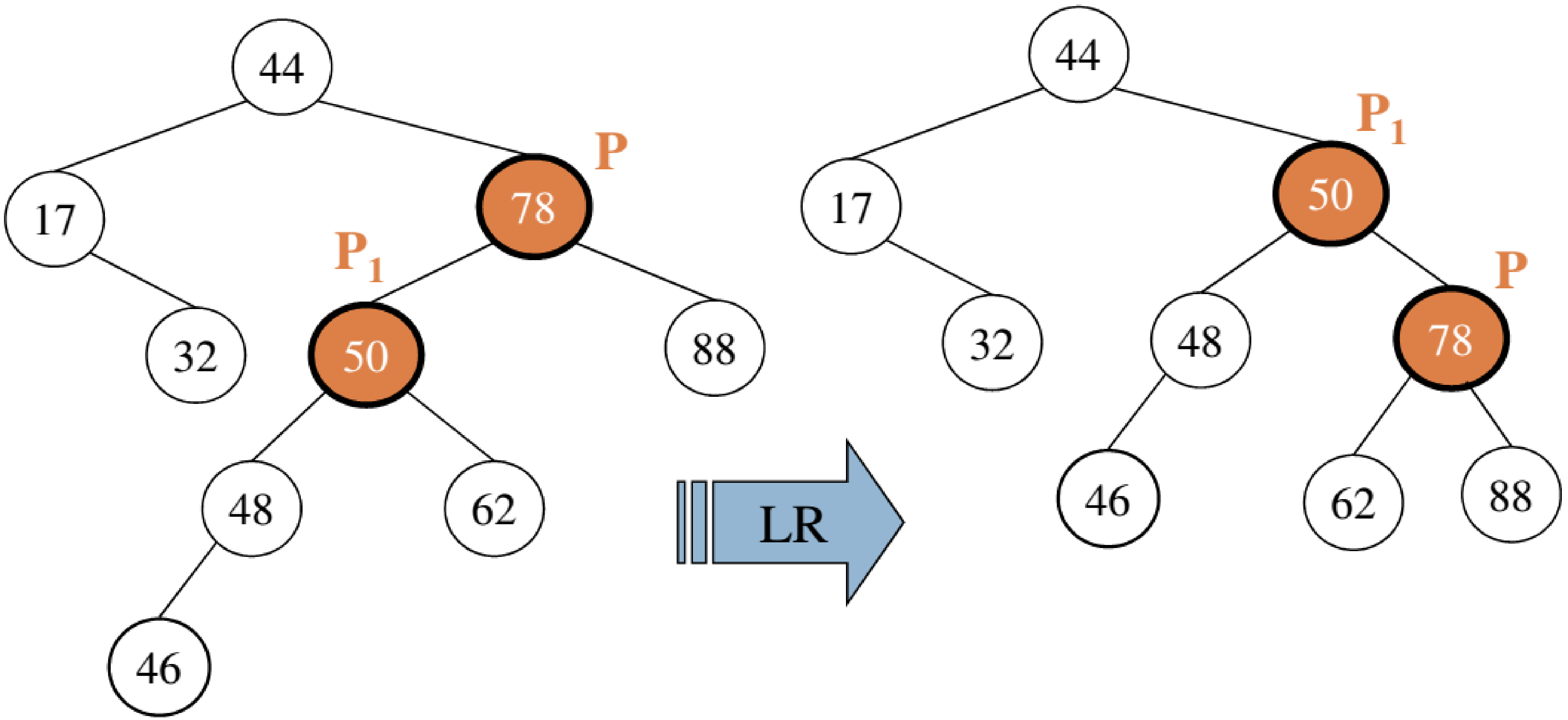


Adjust the tree that is left off – Case (a)

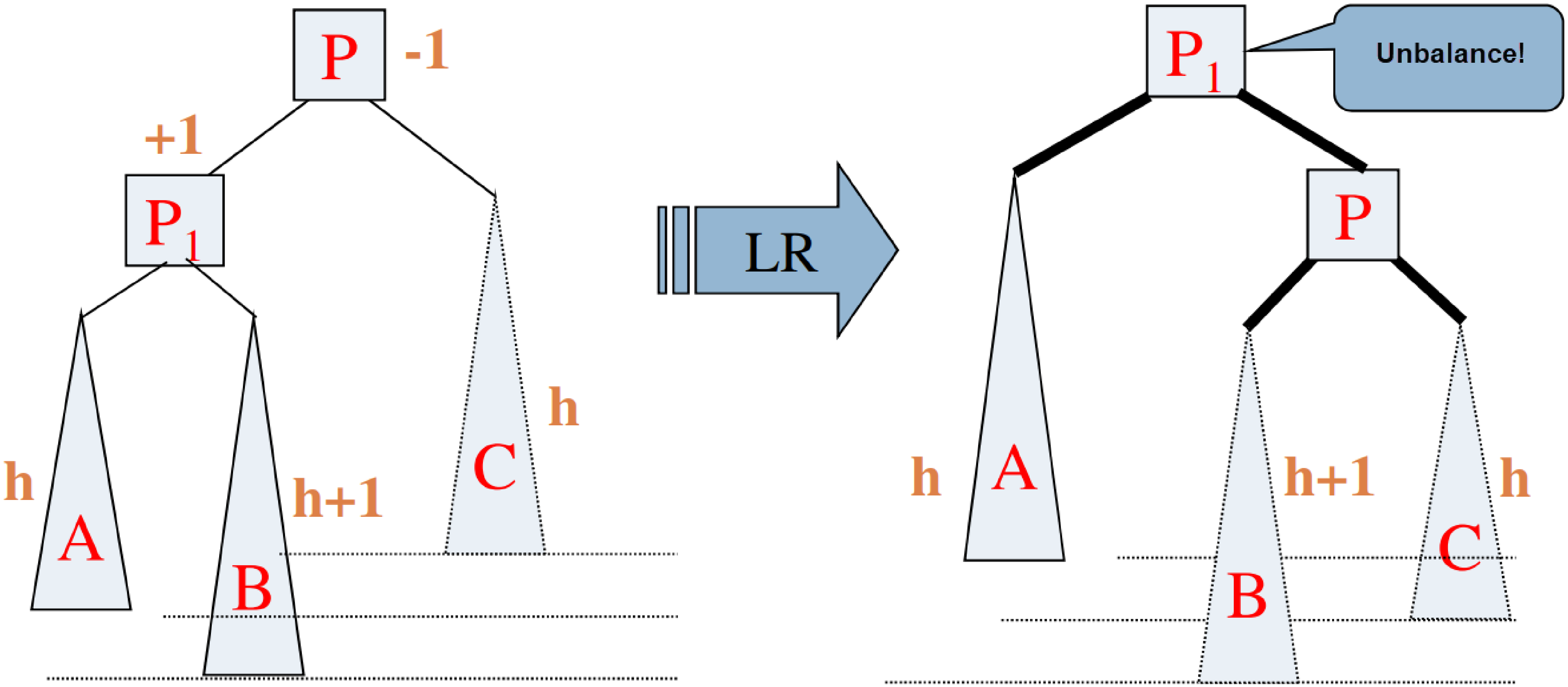


(LR – Single Left-Right)

Adjust the tree that is left off – Case (a)

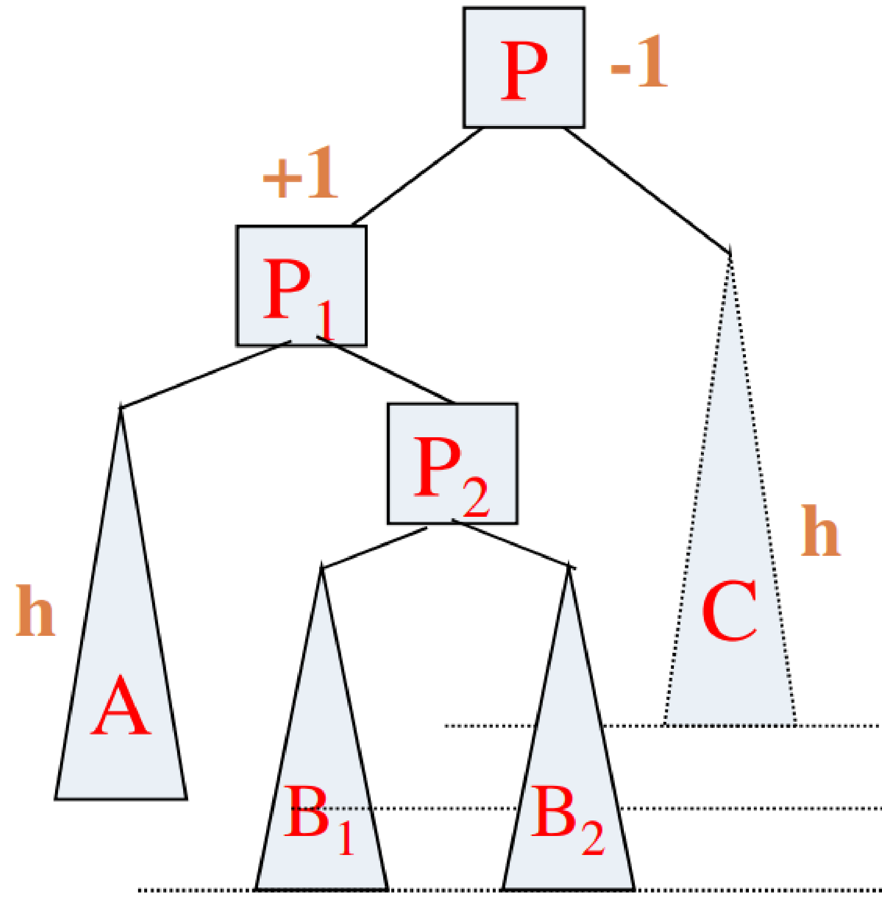


Adjust the tree that is left off – Case (b)

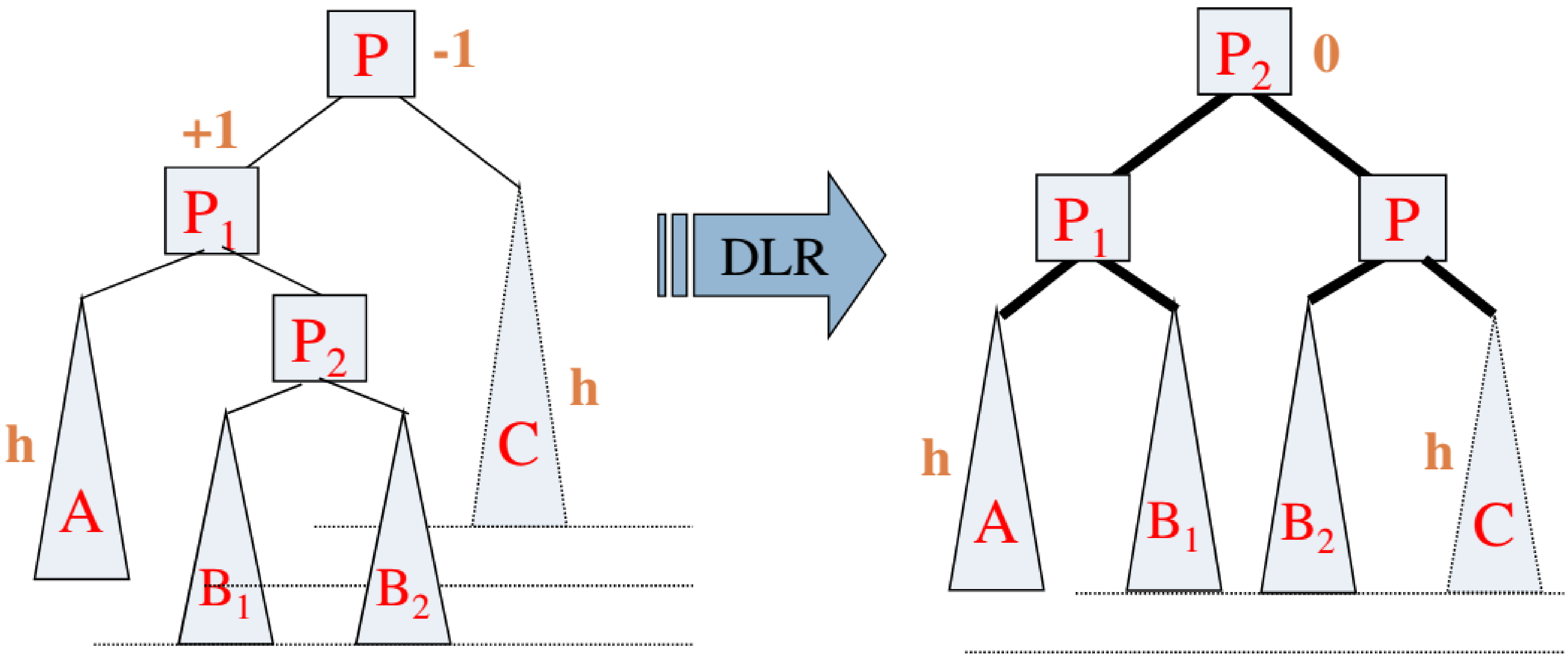


Adjust the tree that is left off – Case (b)

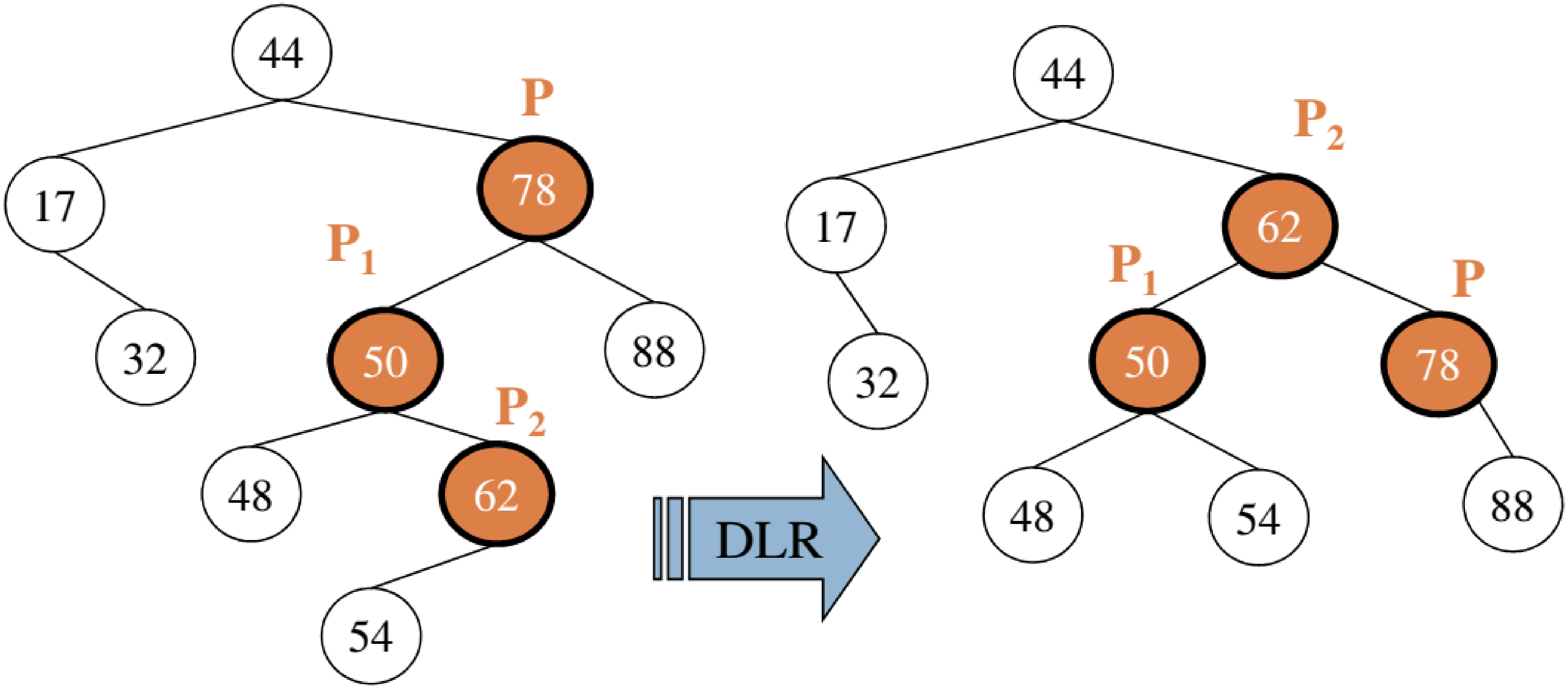
(DLR – Double Left – Right)



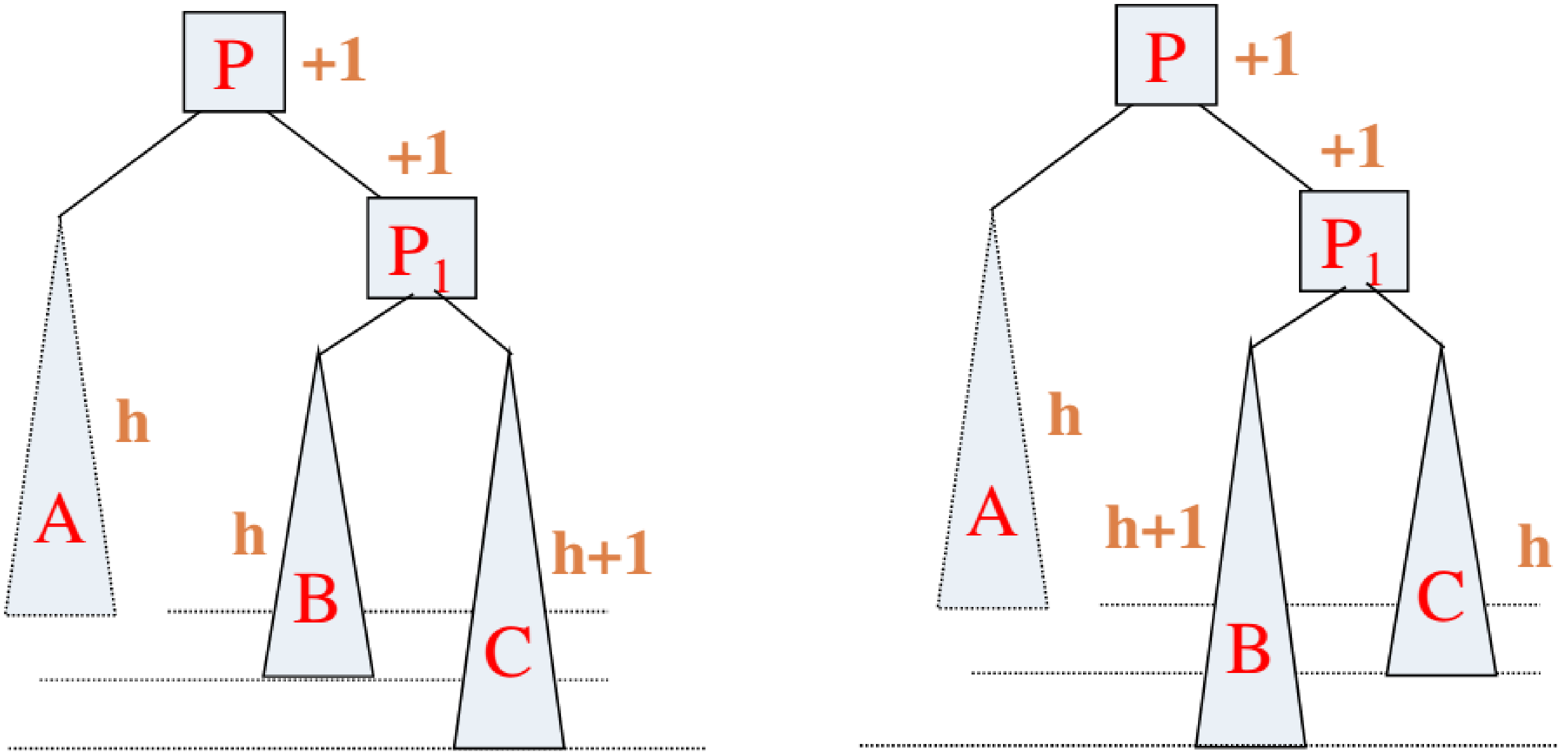
Adjust the tree that is left off – Case (b)



Adjust the tree that is left off – Case (b)



Adjust the tree that is right off

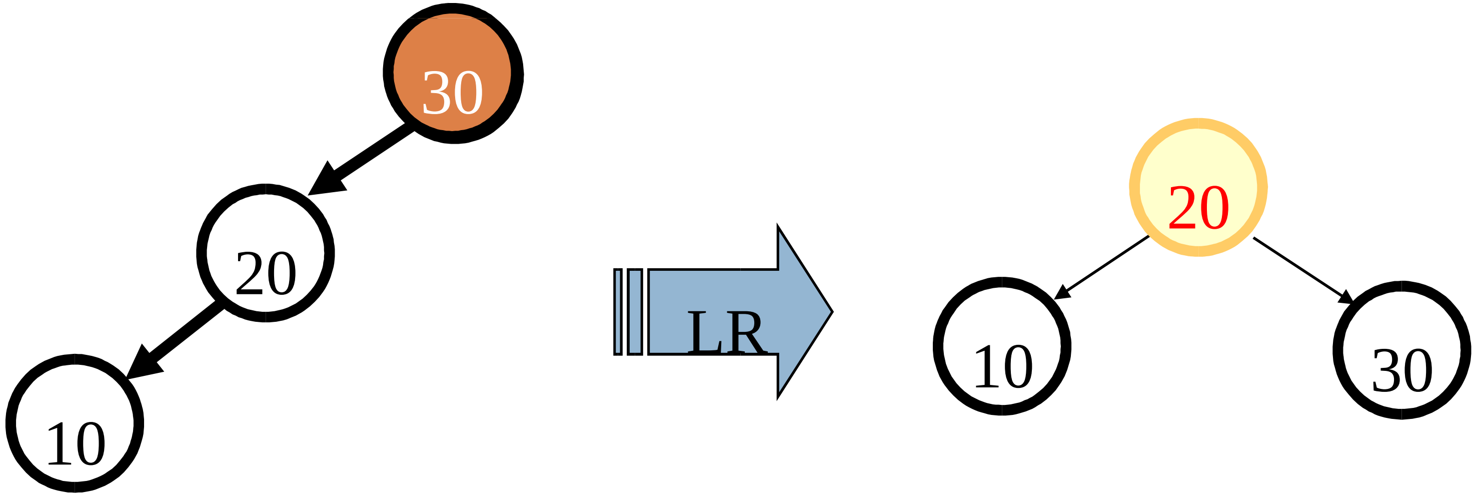


(a) (b)

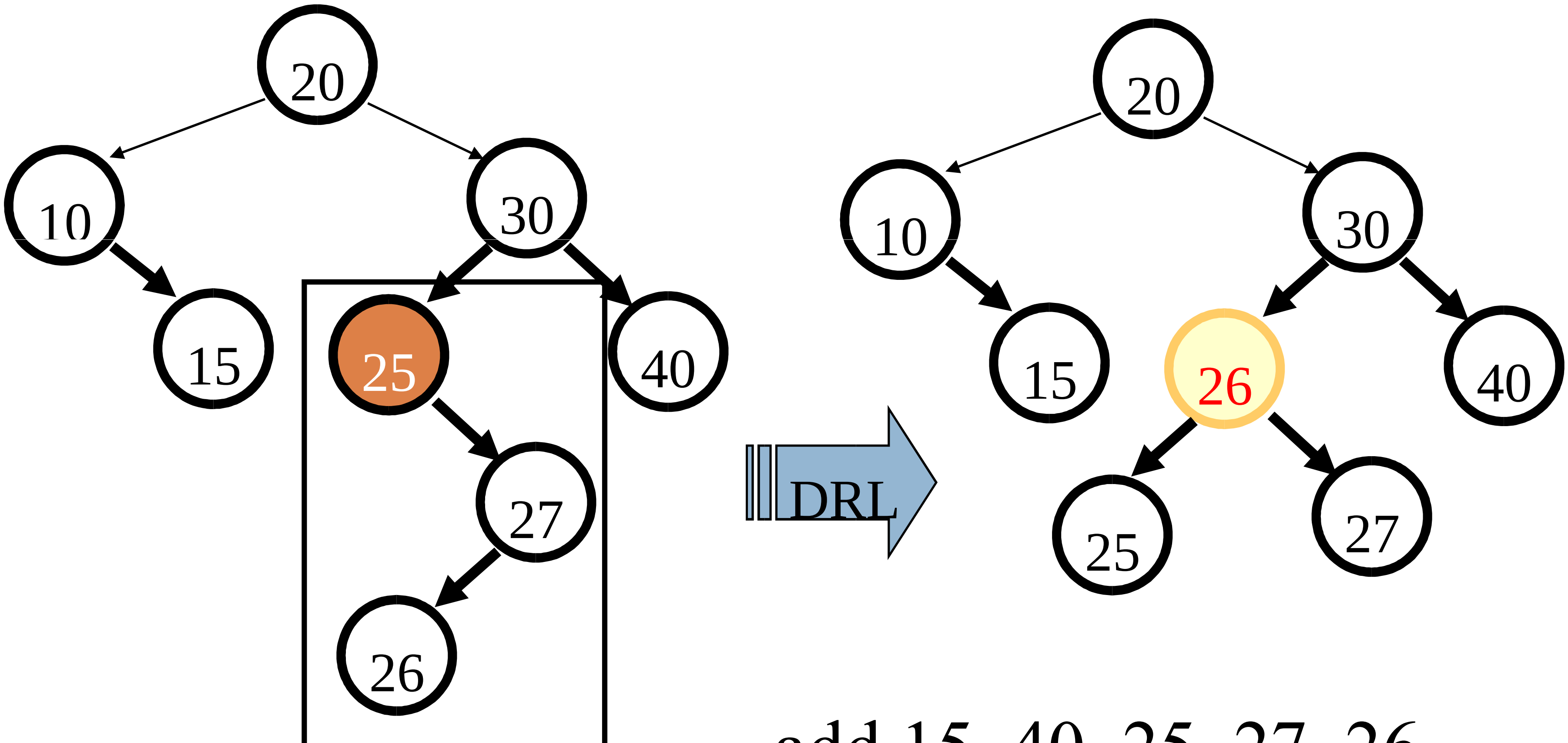
Do Similarly

Example

- Create an AVL tree with the keys respectively: 30, 20, 10,…

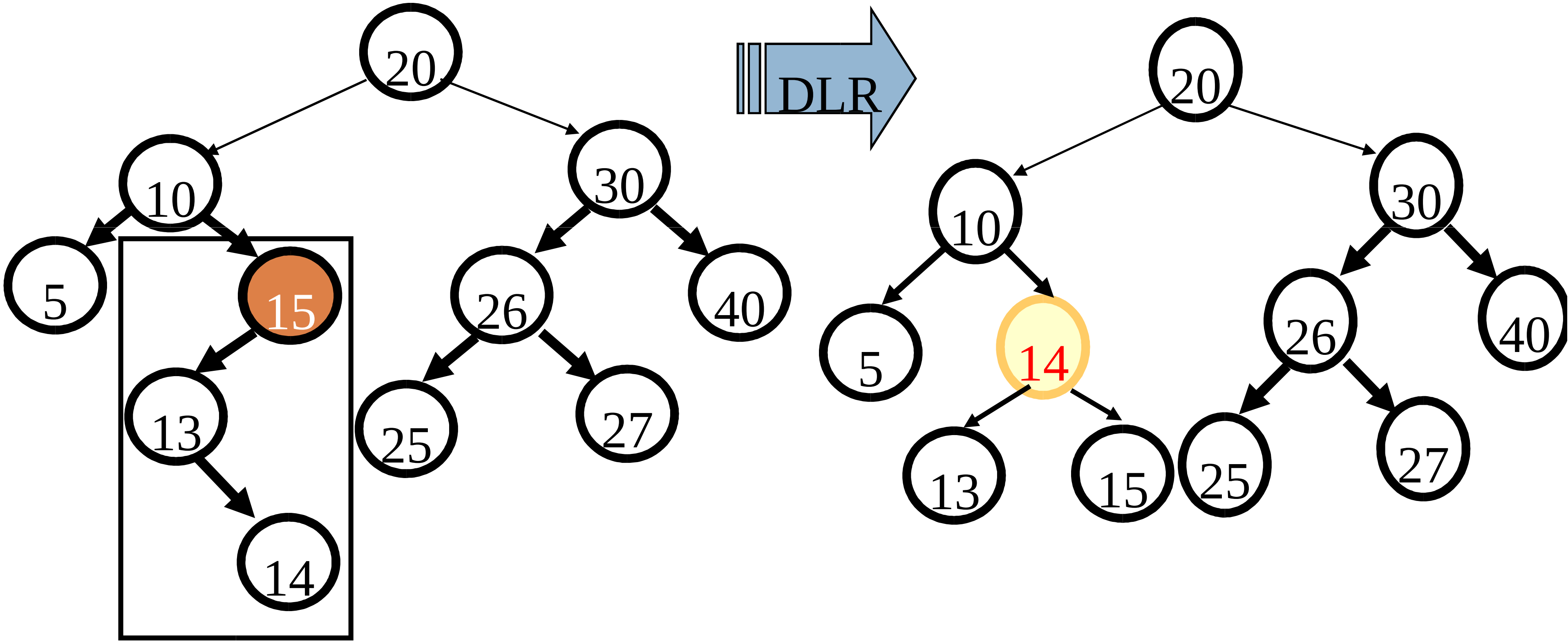


Example



…add 15, 40, 25, 27, 26

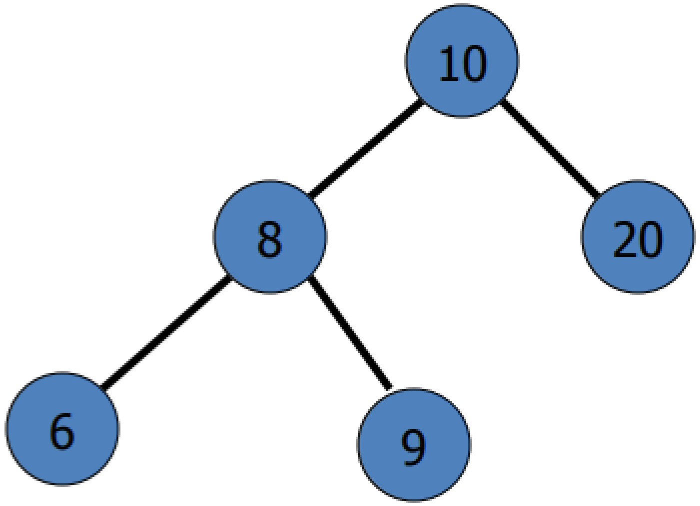
Example



…add 5, 13, 14

AVL

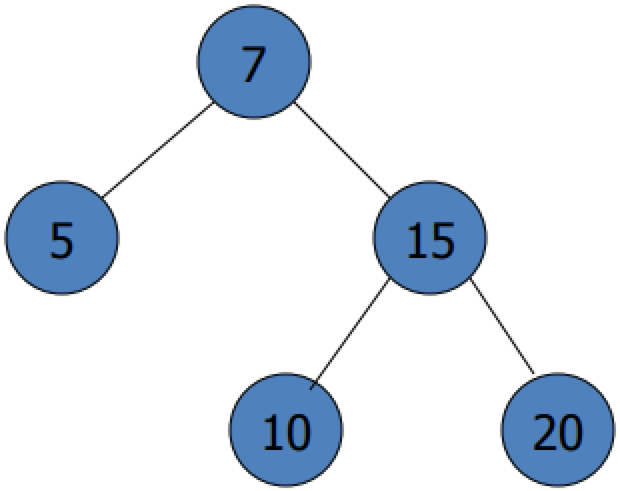
- AVL Tree



How about insert

Quiz

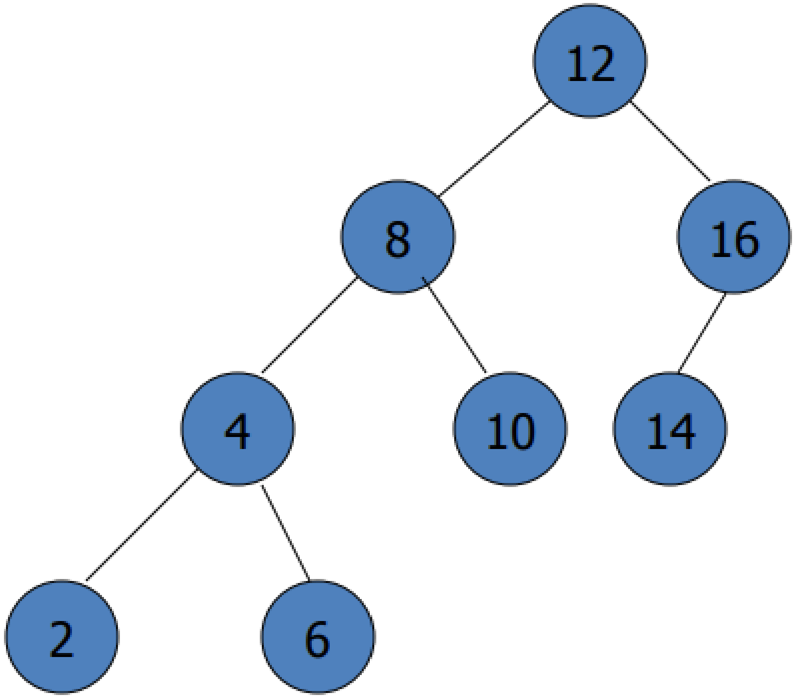
- AVL



What if insert

Quiz

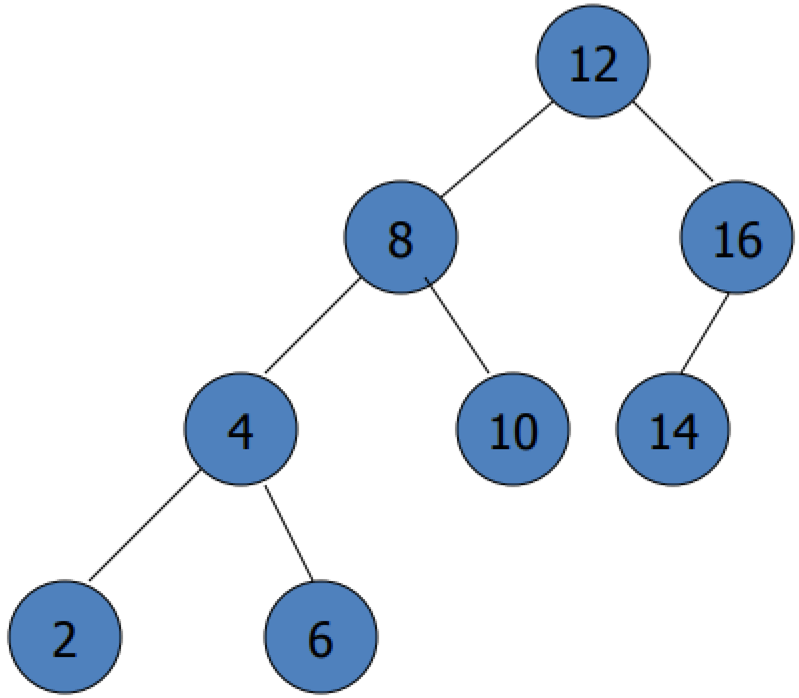
- AVL



How about insert

Quiz

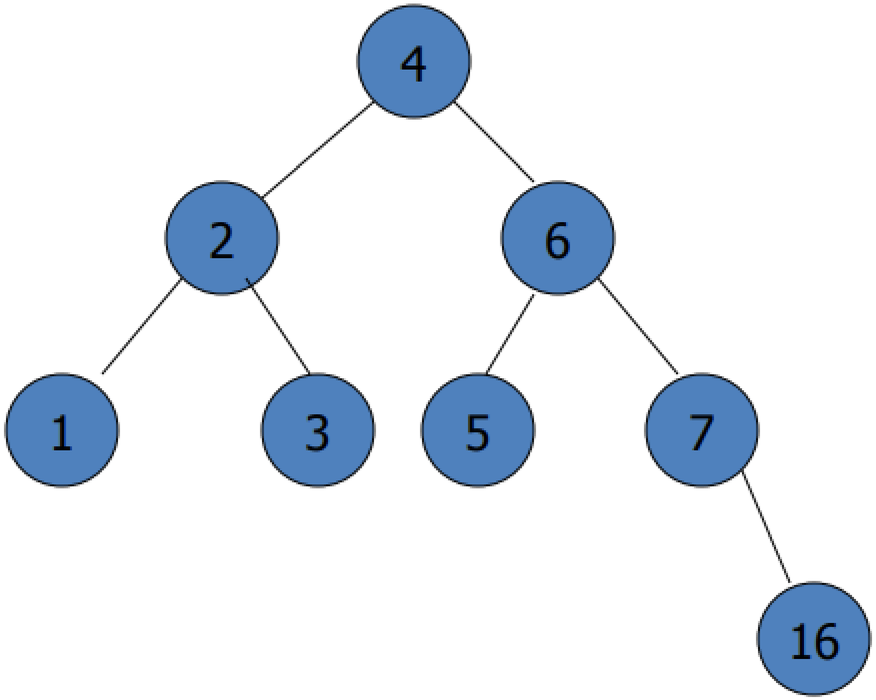
- AVL



How about insert

Quiz

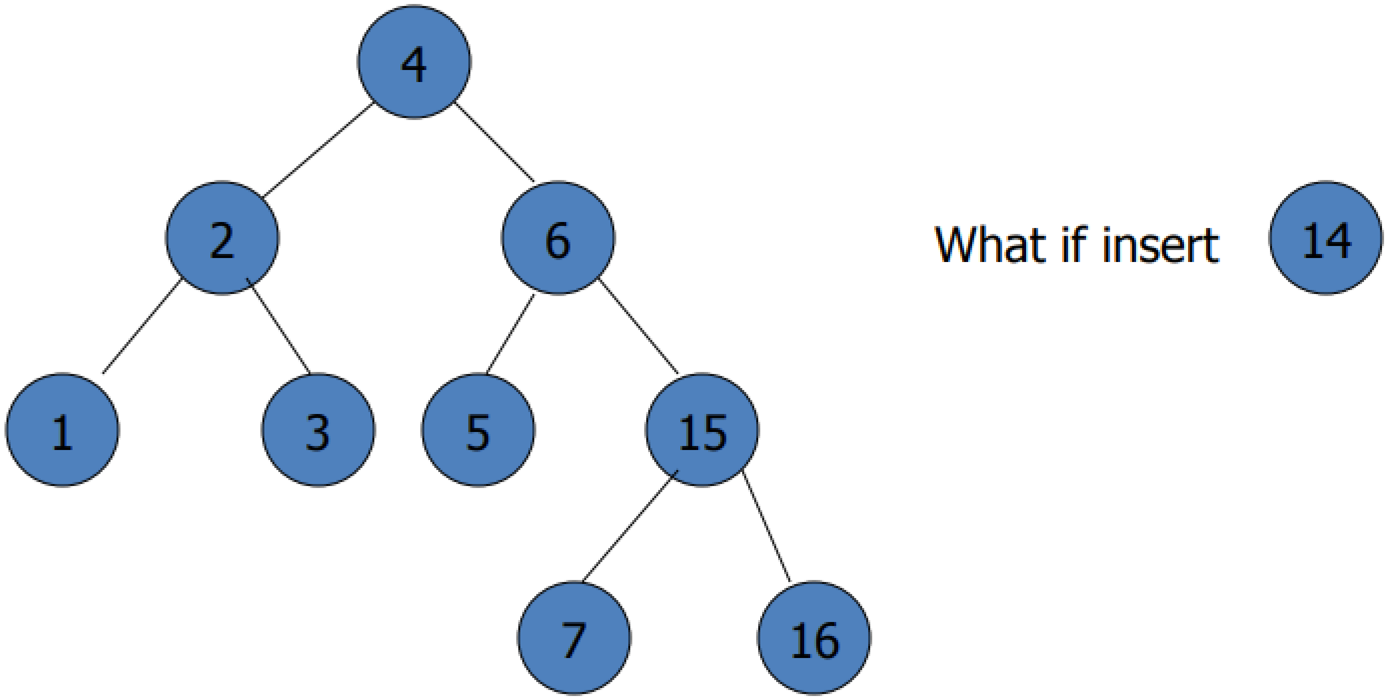
- AVL



What if insert

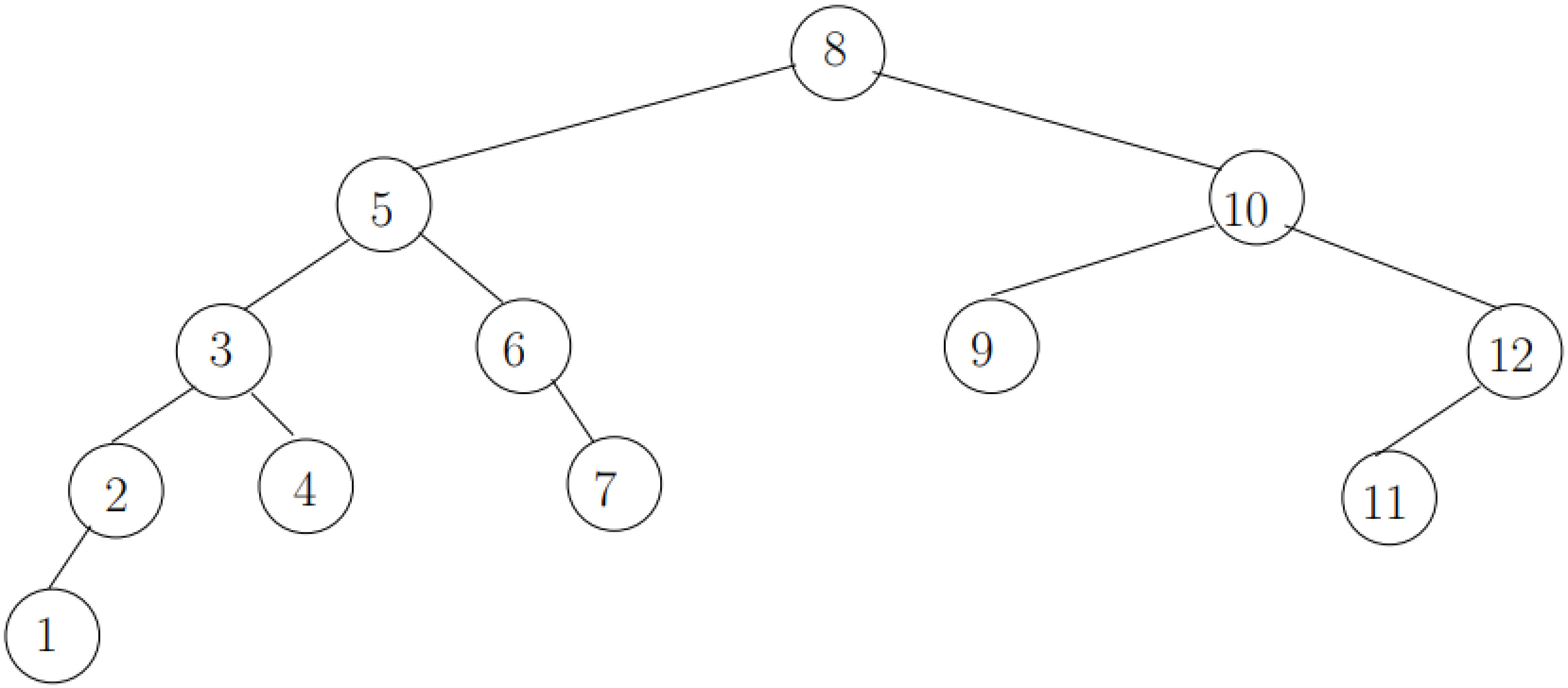
Quiz

- AVL



Quiz

- Remove the node labelled 9 from the following AVL-tree:



Quiz

Draw all the rotations that you must perform and the final AVL tree after the following elements are inserted in the given order starting from an empty tree.

1, 10, 5, 7, 3, 13, 6, 4, 8, 9

Comments

- Tree height:
  - $h_{AVL} < 1.44\log_2(N + 1)$.
  - The AVL tree was 44% higher than that of an optimal binary tree.
- Search cost: $O(\log_2 N)$
- Cost of adding an element $O(\log_2 N)$
  - Search: $O(\log_2 N)$
  - Tree adjustment: $O(\log_2 N)$
- Cost of deleting element $O(\log_2 N)$
  - Search: $O(\log_2 N)$
  - Tree adjustment: $O(\log_2 N)$

Quiz

What is the minimum number of nodes in an AVL tree of height 7?

Quiz

What is minimum possible height of AVL Tree using 8 nodes?

Quiz

What is maximum possible number of nodes in AVL tree of height-3?

Quiz

Suppose you have an AVLNode class that stores integers:

```
public class AVLNode {
    public int item;
    public AVLNode left;
    public AVLNode right;
    public AVLNode (int i, AVLNode l, AVLNode r)
      item = i; left = l; right = r; ;
}
```

Write a complete method that takes a height h, and returns a reference to the root of an AVL tree of height h that contains the minimum number of nodes. You can define helper methods and/or classes if you wish.

The End.