University of Science, VNU-HCM
Faculty of Information Technology

Data Structure and Algorithm

Binary Search Tree
Balanced Tree
AVL

Lecturer: Lê Ngọc Thành
Email: lnthanh@fit.hcmus.edu.vn

HCM City

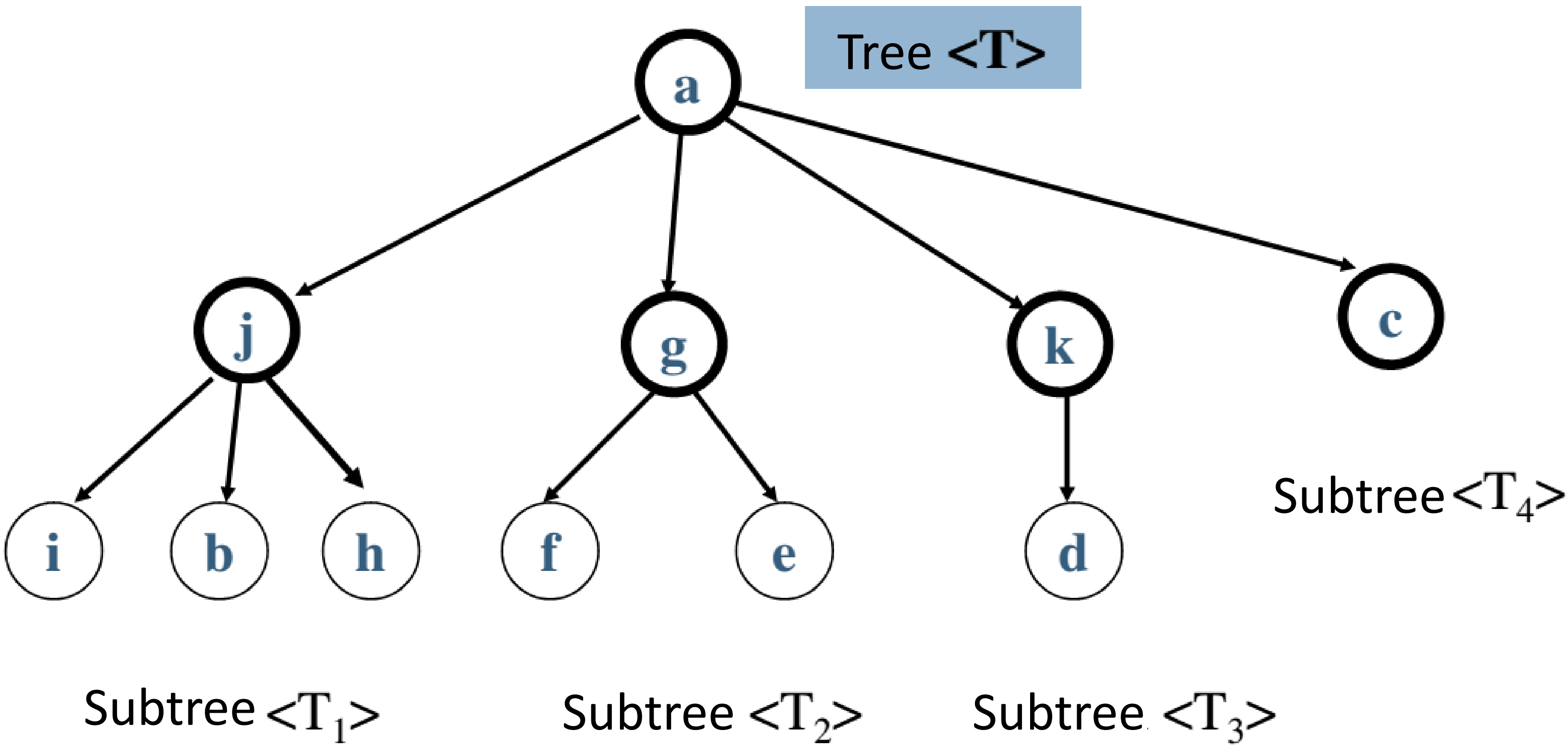
Outline

- 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, ..., 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,…, 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_1>
Subtree <T_2>
Subtree <T_3>
Subtree <T_4>

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

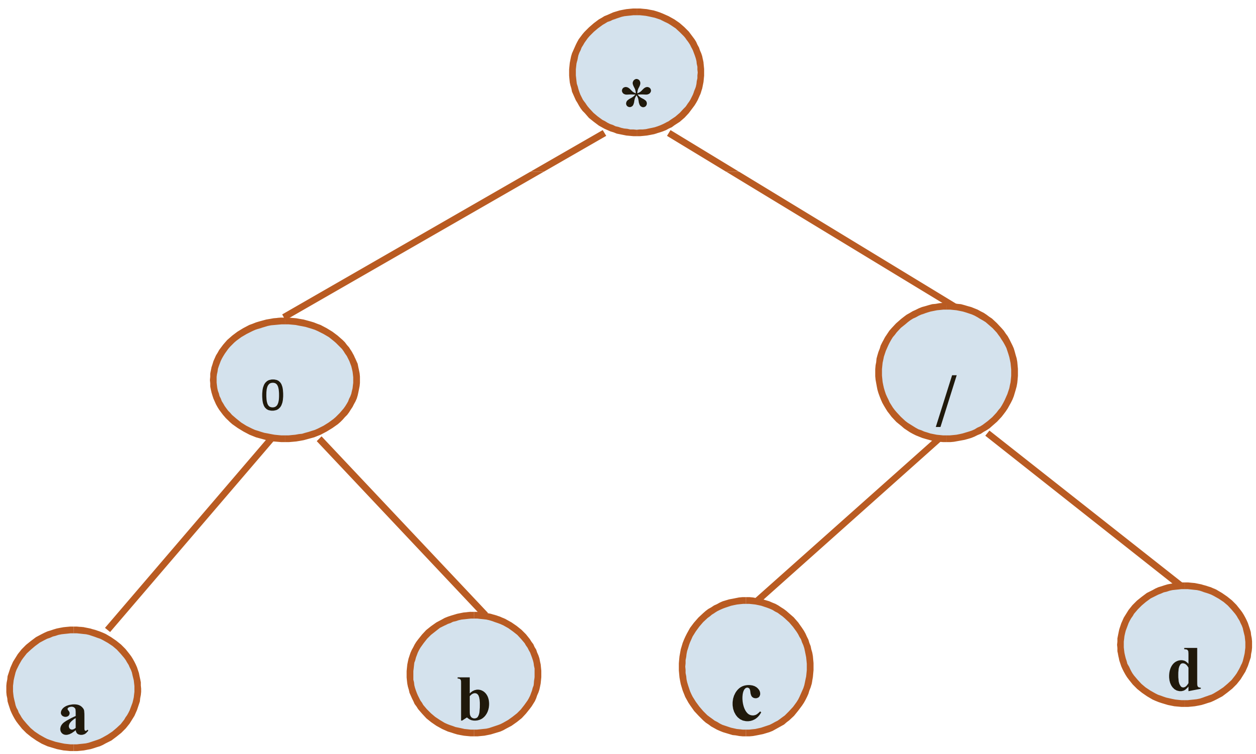
- Level:
  - Level(p) = 0 if p = root
  - Level(p) = 1 + level(parent(p)) if p! = Root
- Height of tree (h_T): the longest path from the root node to the leaf node
  - h_T = max {Path(root, p_i) | p_i is the leaf node ∈ <T>}

Outline

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

Binary Tree

- A binary tree is a tree with degree = 2



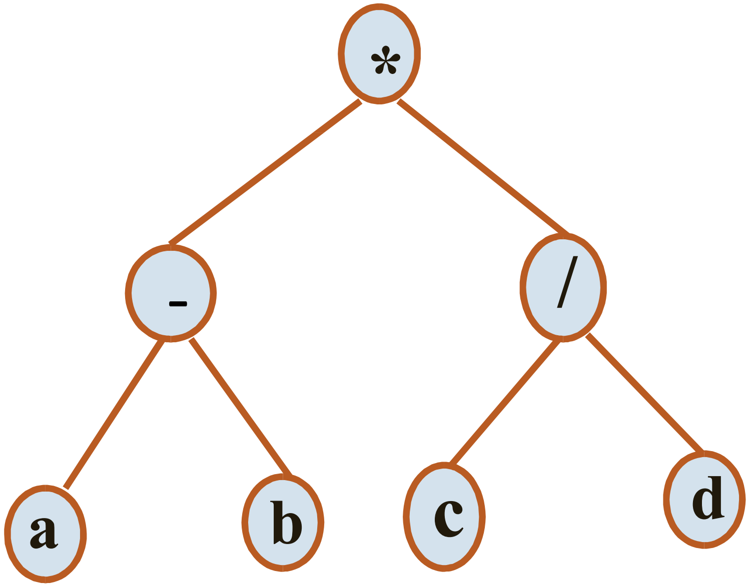
Binary Tree

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

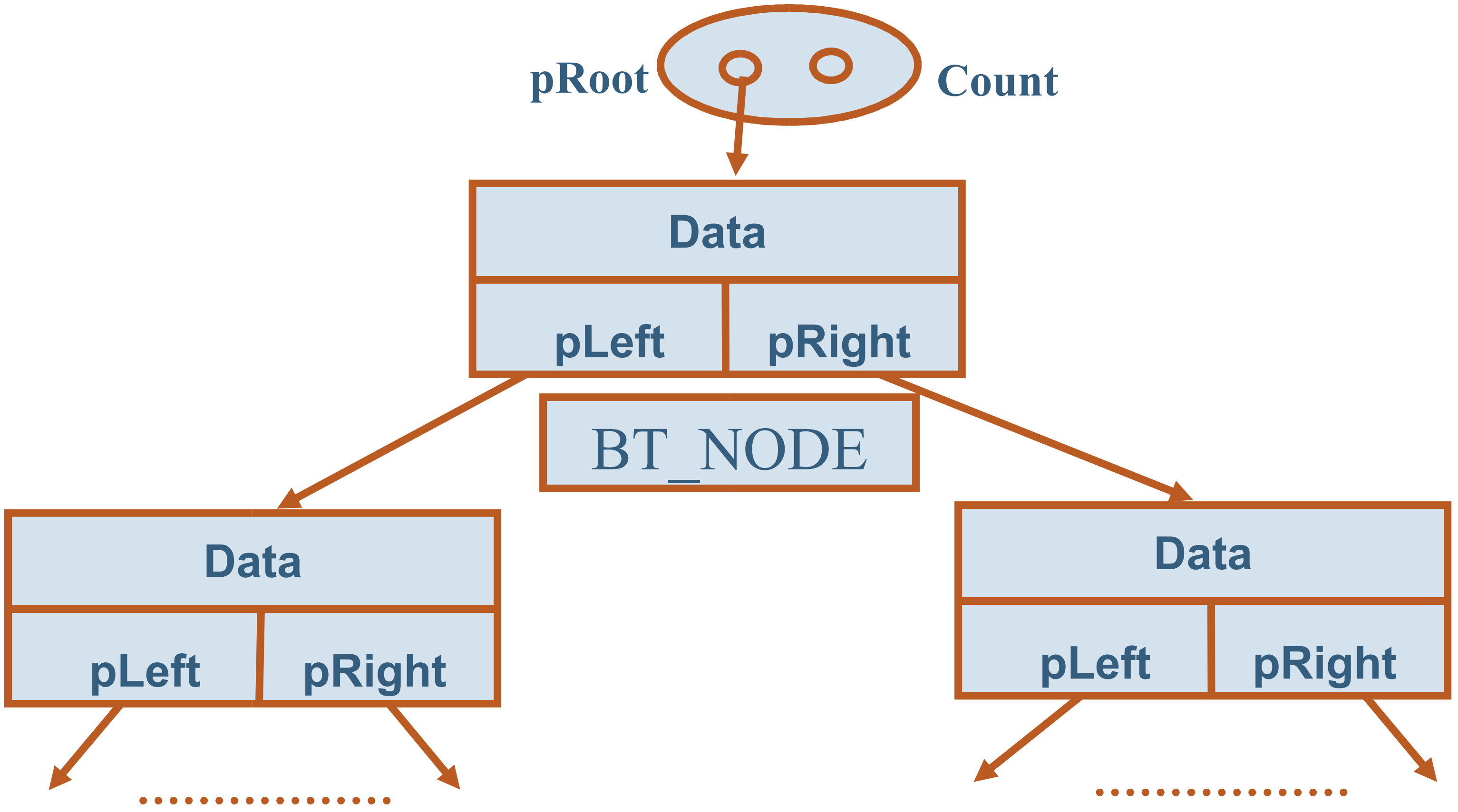
Binary Tree

- 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

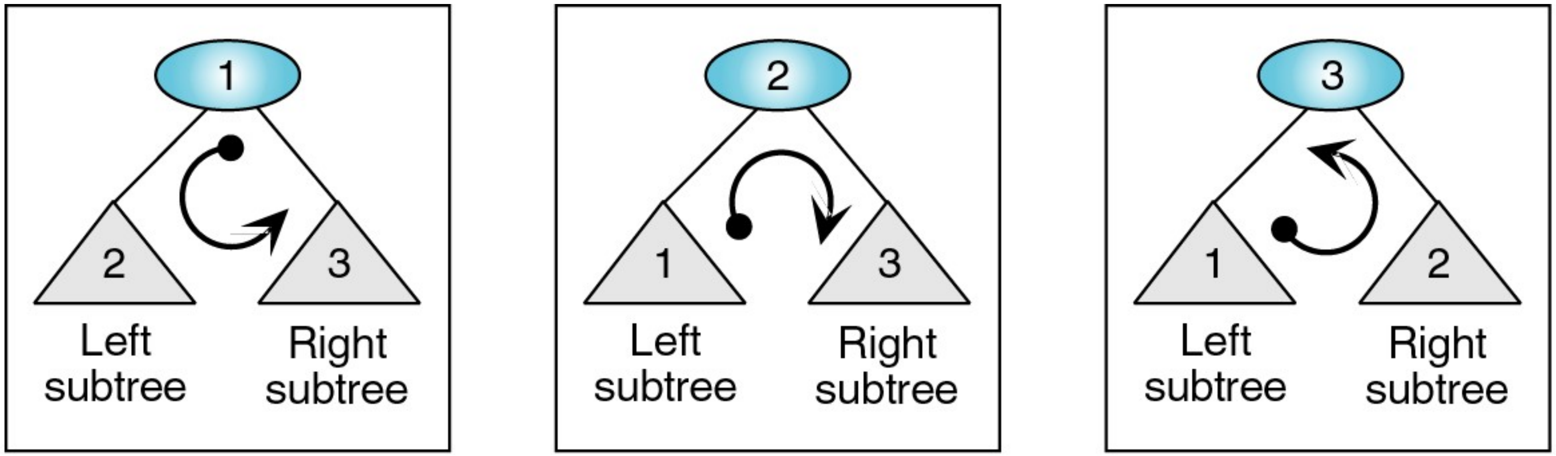

pRoot
Count
Data
pLeft
pRight
BT_NODE
Data
pLeft
pRight
Data
pLeft
pRight
....
....

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

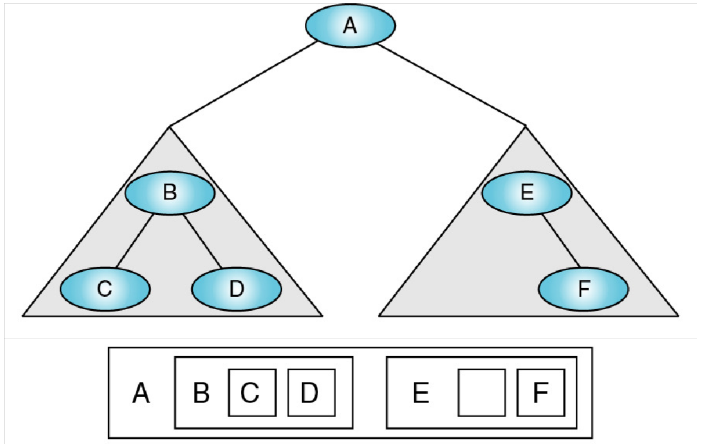
- There are 3 ways to traverse the tree:
  - Pre-Order (NLR)
  - In-Order (LNR)
  - Post-Order (LRN)



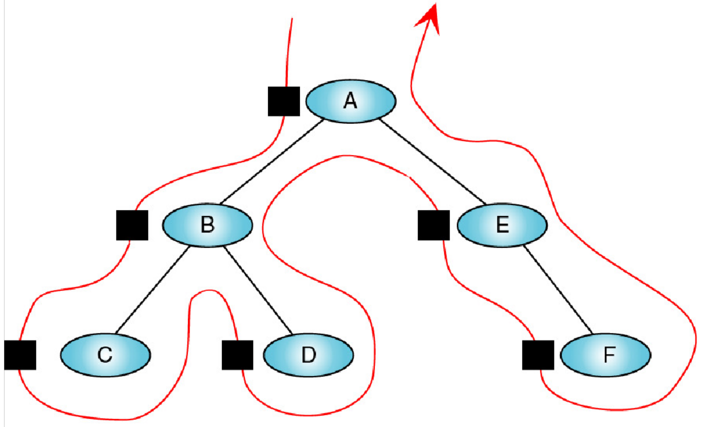
(a) Preorder traversal **(b) Inorder traversal** **(c) Postorder traversal**

Traverse in Tree - NLR

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

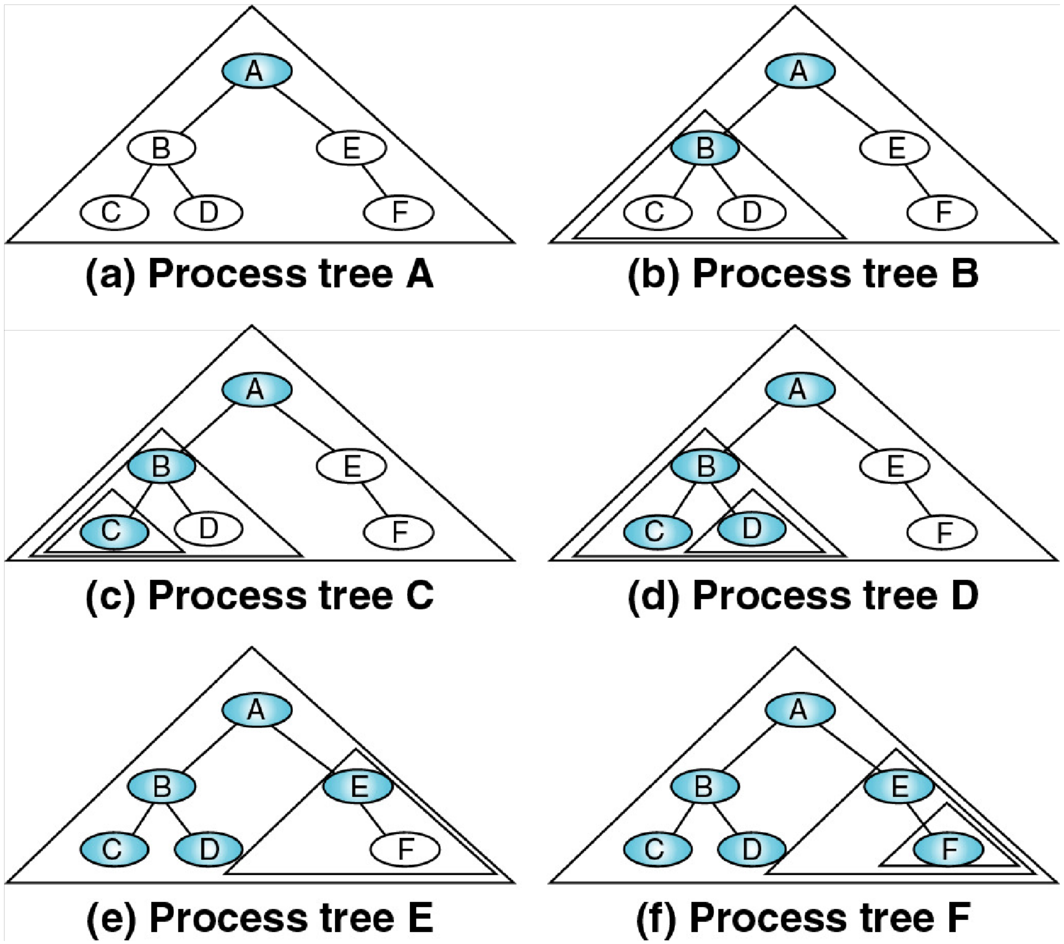


(a) Processing order



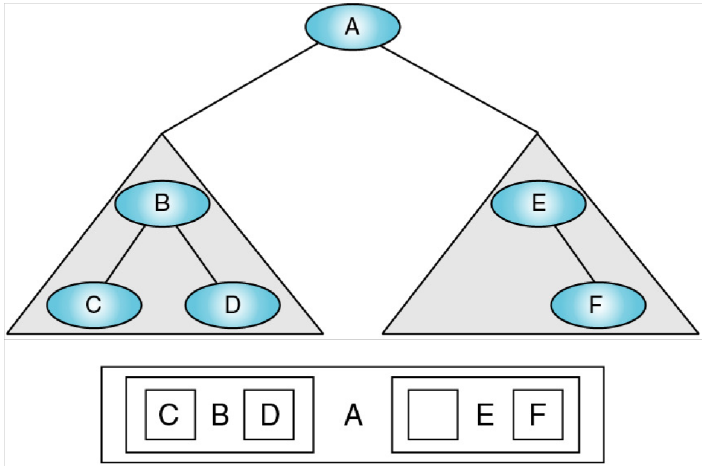
(b) "Walking" order

Traverse in Tree - NLR

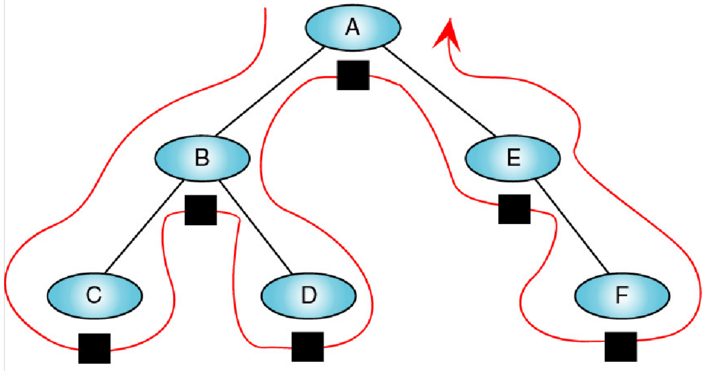


Traverse in Tree - LNR

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



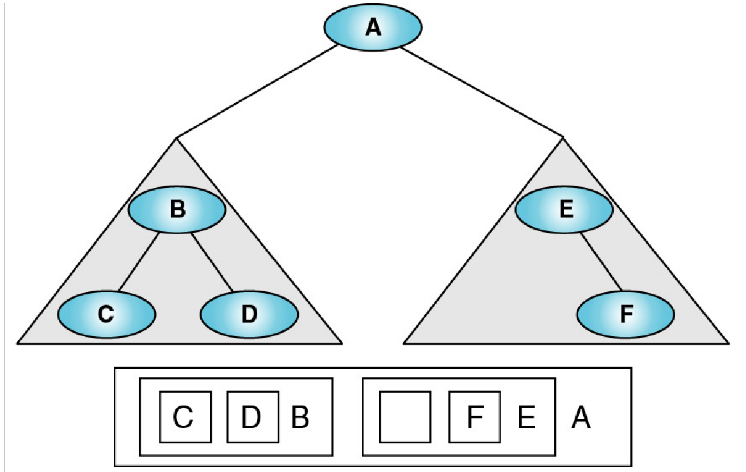
(a) Processing order



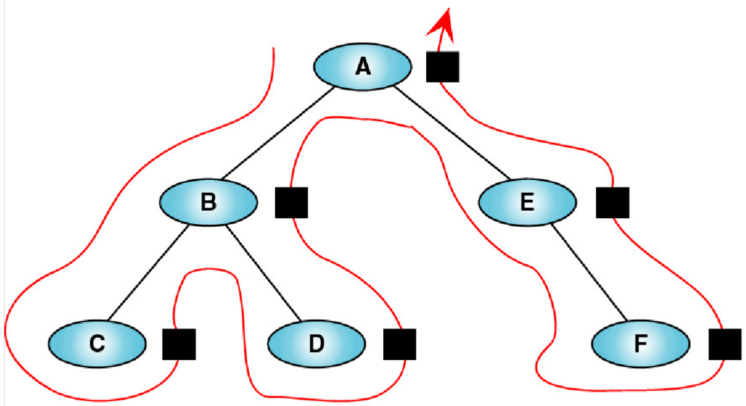
(b) "Walking" order

Traverse in Tree - LRN

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



(a) Processing order



(b) "Walking" order

Outline

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

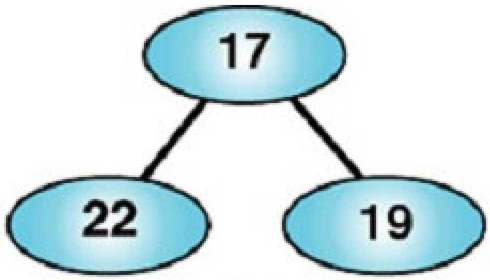
Binary Search Tree

- 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

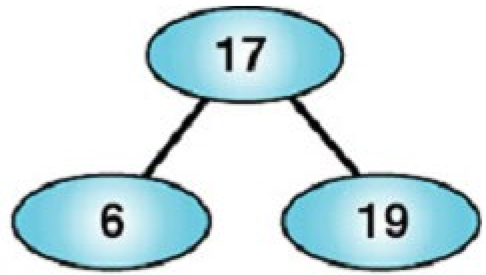
      $\forall q \in$ p-> pLeft: q-> Data <p-> Data
    - All nodes in the right subtree (p-> pRight) are greater than the value of p

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

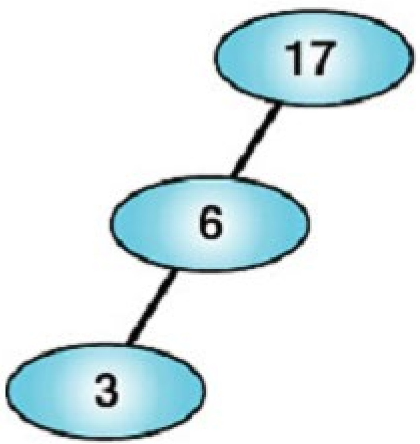
Example



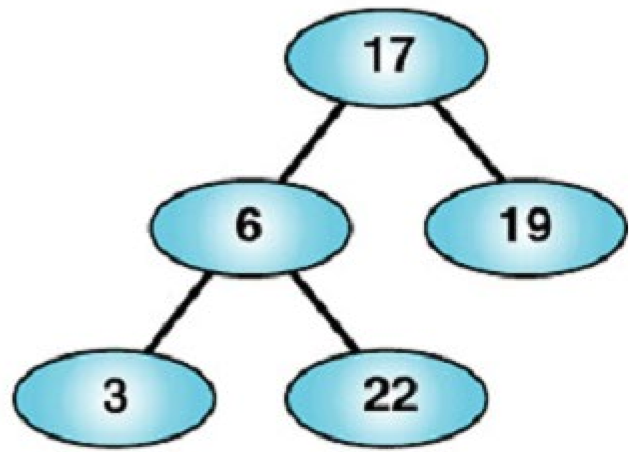
(a)



(b)



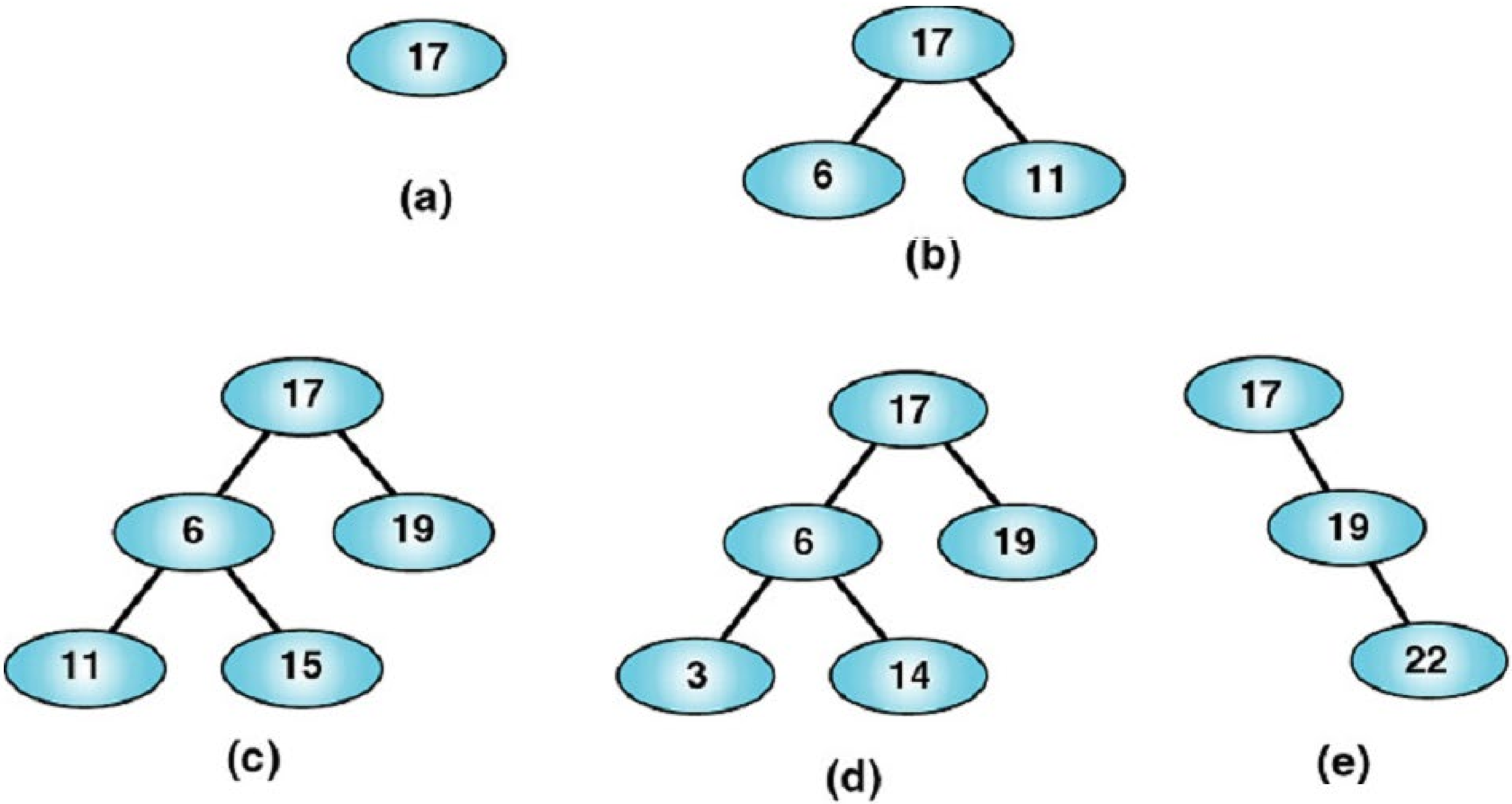
(c)



(d)

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:

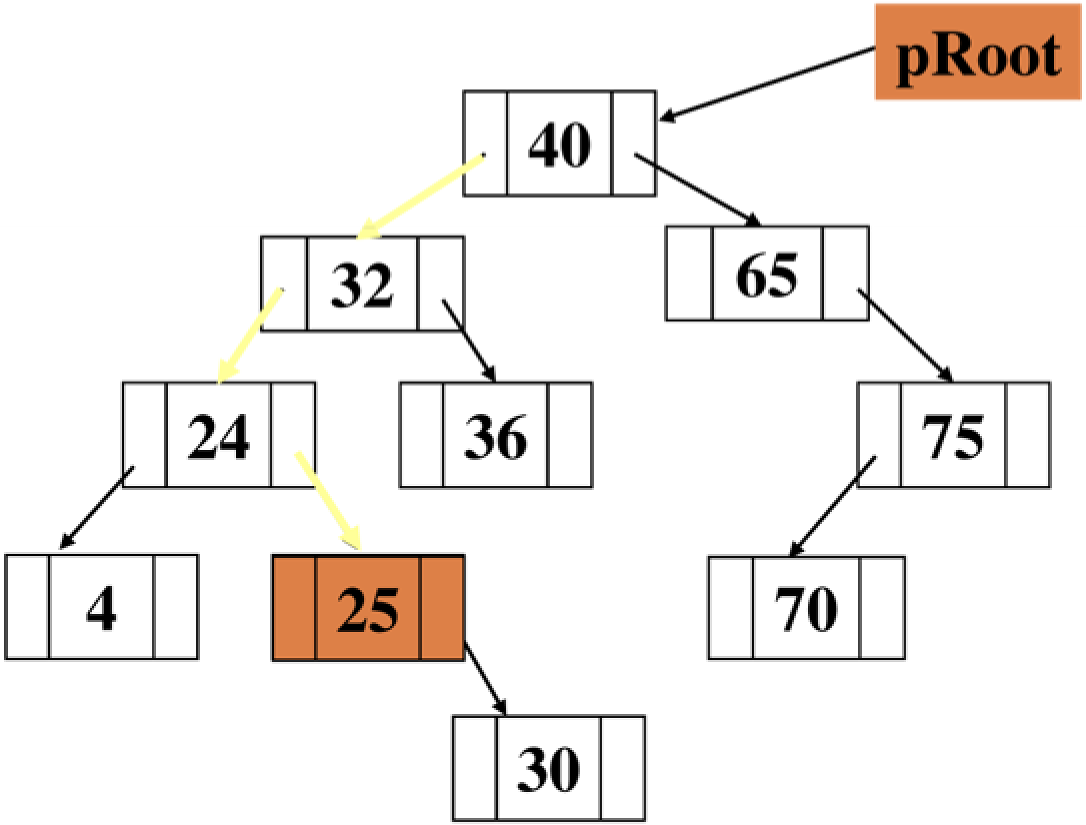
```
void BSTCreate(BIN_TREE &t)
{
    t.Count = 0;    // number of nodes in BST
    t.pRoot = NULL; // pointer of root node
}
```

- Check a empty tree:

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

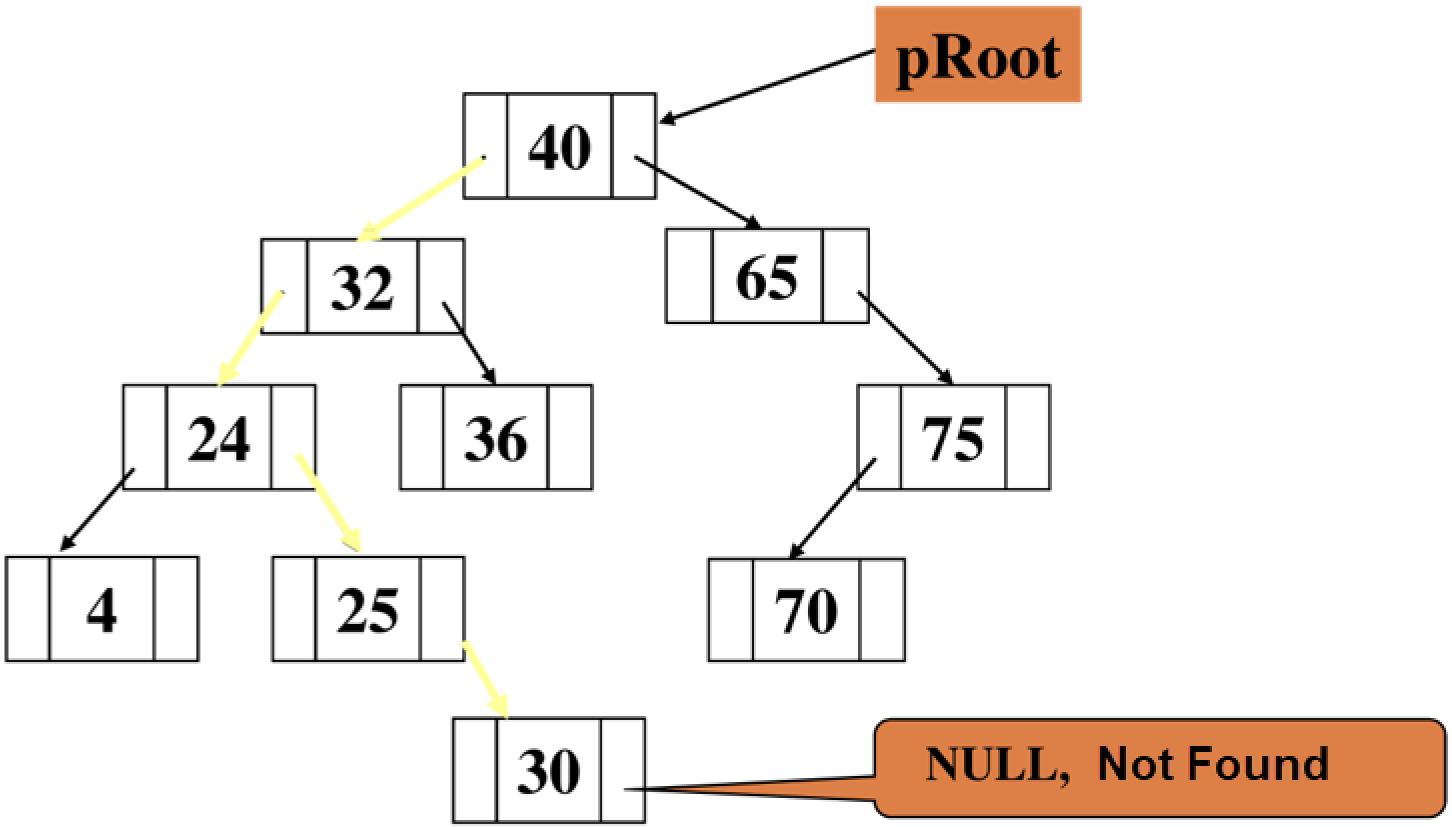
Search for an element

- Example search for element 25:



Search for an element

- Example search for element 31:

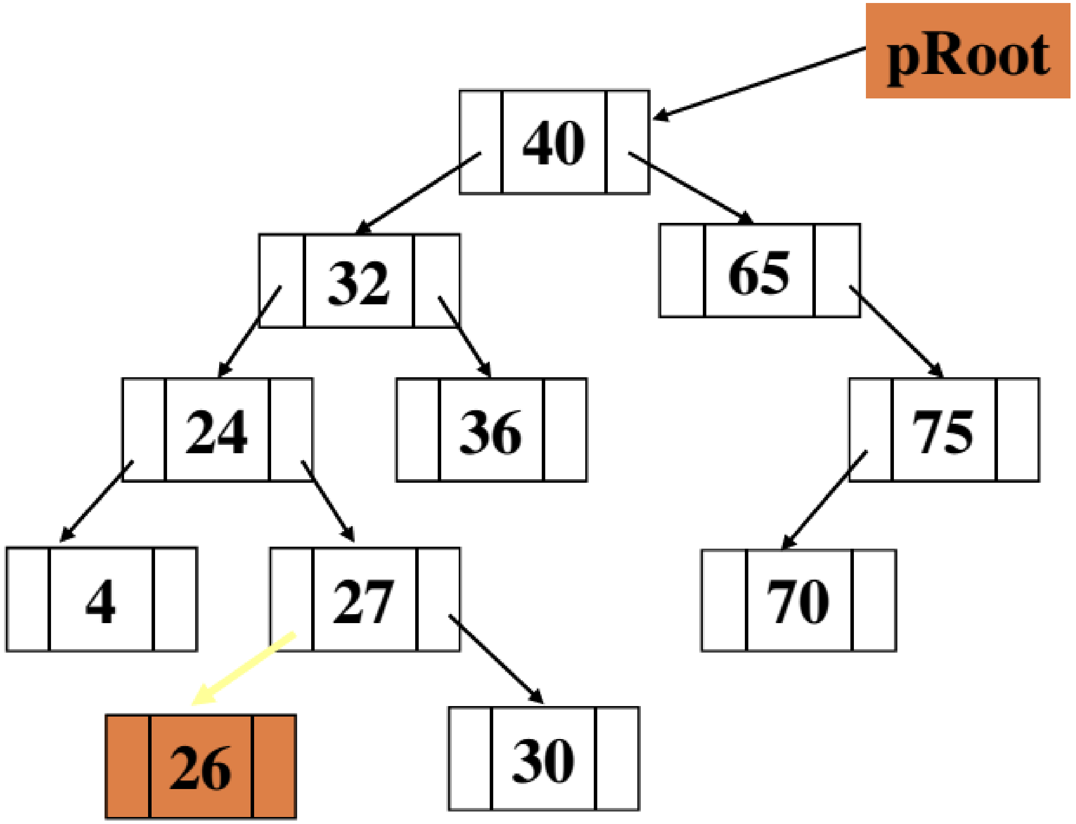


Search for an element

```
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);
}
```

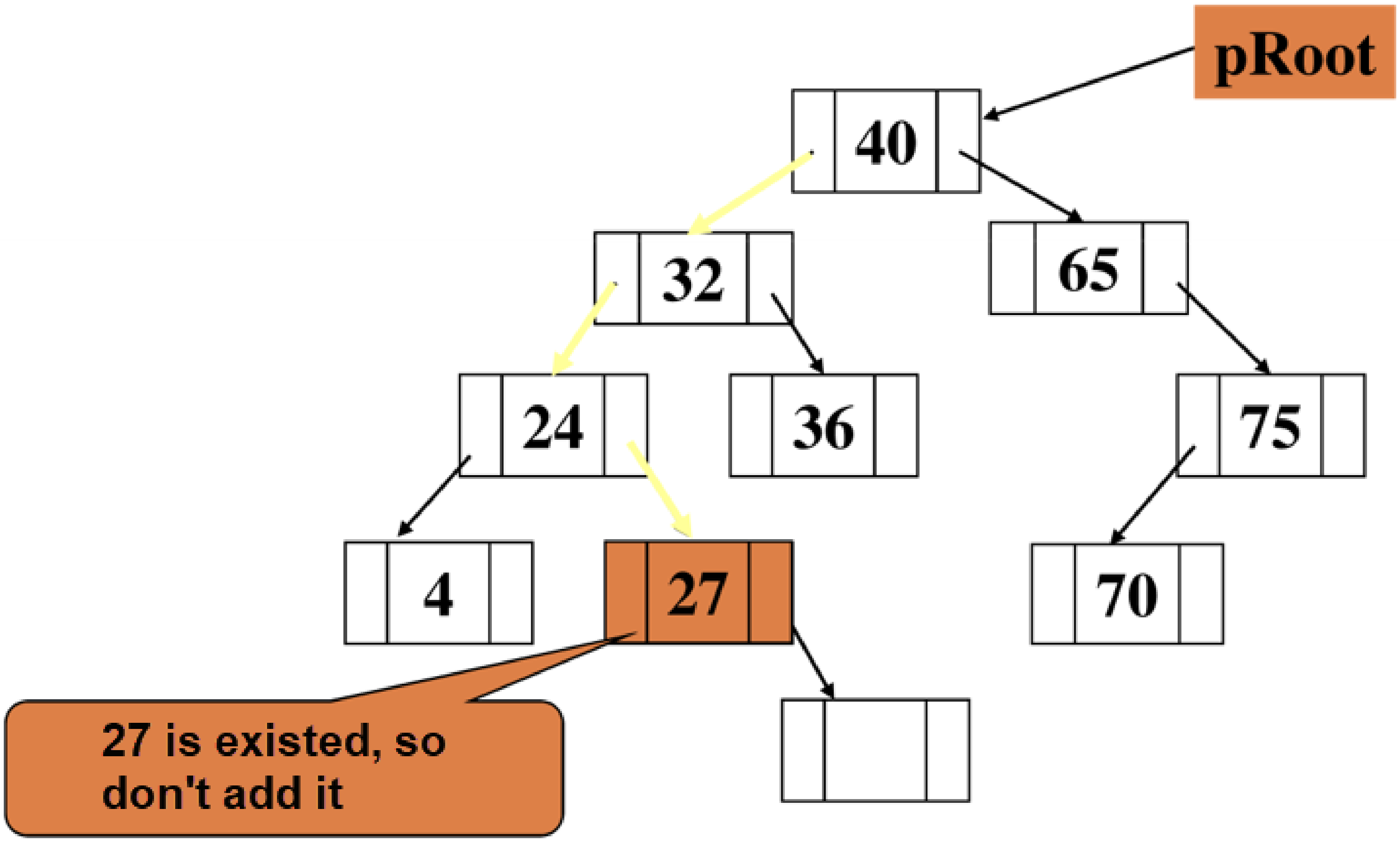
Add new element

- Example for adding element 26:



Add new element

- Example for adding element 27:



Add new element

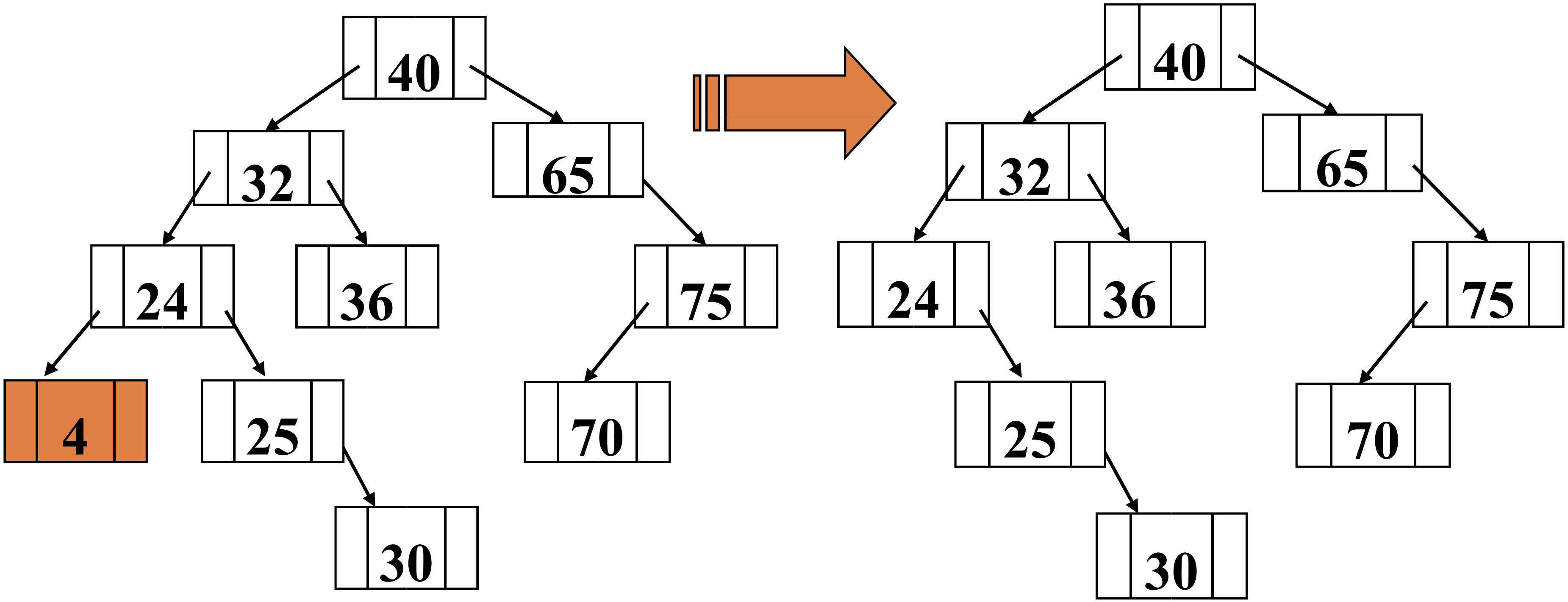
```
int  BSTInsert(BT_NODE *&pCurr, int newKey)
{
        if (pCurr==NULL) {
                pCurr = new BT_NODE;  // Create new node
                pCurr->Data = newKey;
                pCurr->pLeft = pCurr->pRight = NULL;
                 return 1; // Success to add new element
        }
        if (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
}
```

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

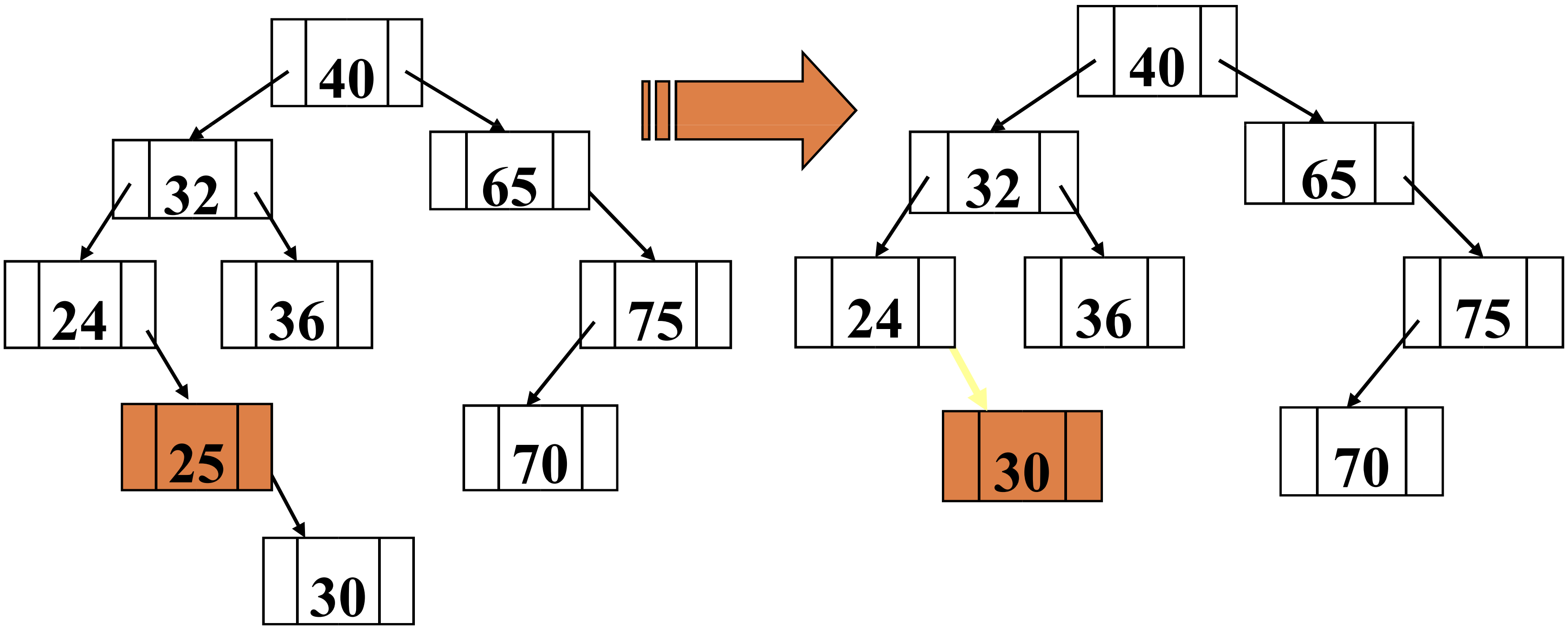
Delete an element without child

- Example of deleting element 4 (without child nodes)



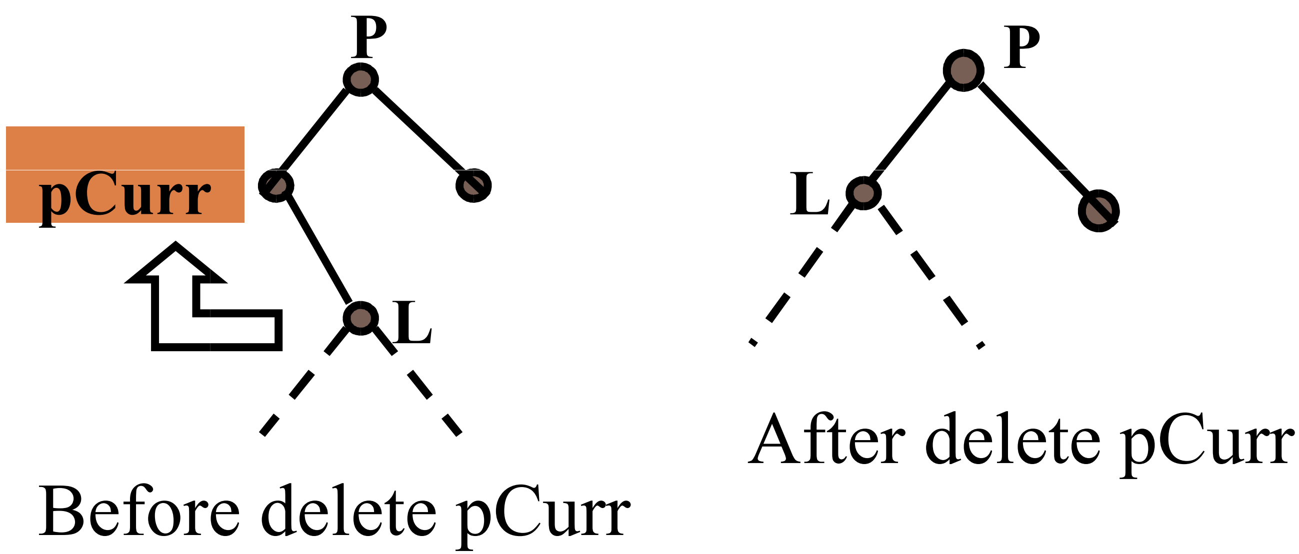
Delete an element with right child

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



Delete an element with right child

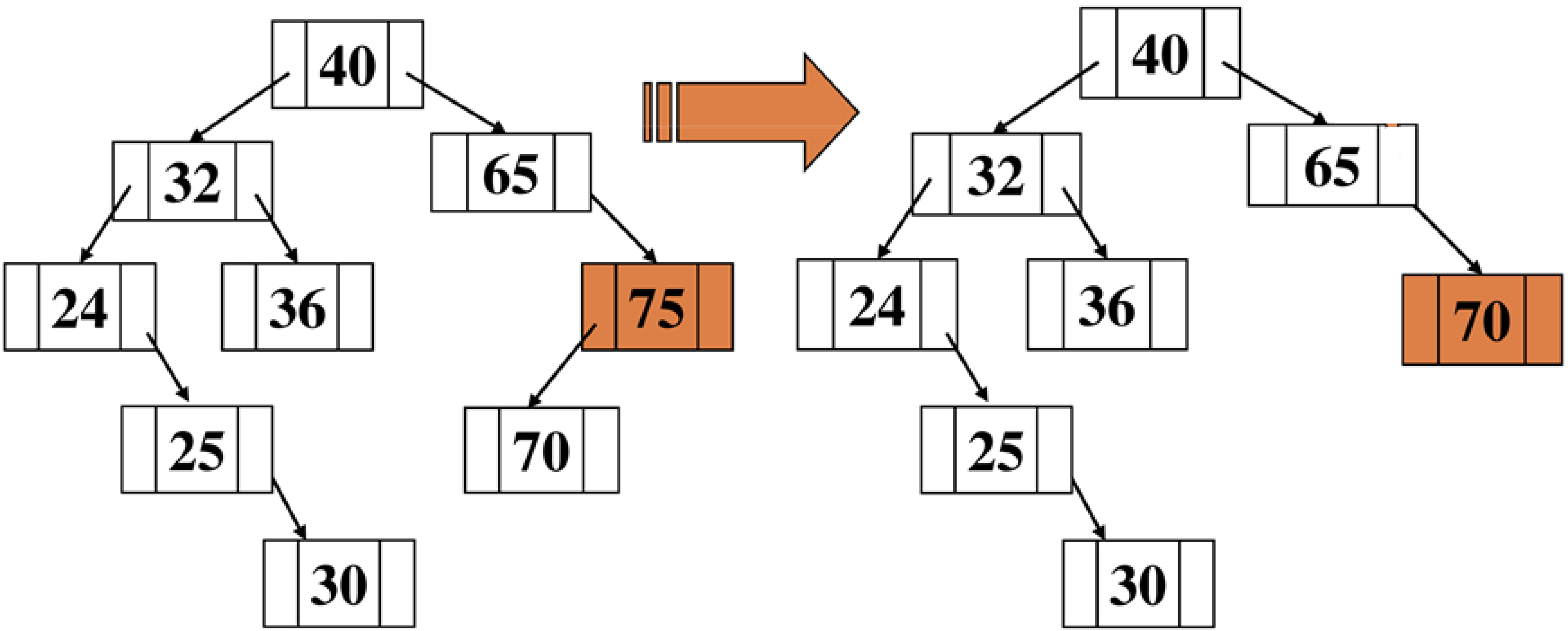
- Delete node with only the right child node



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

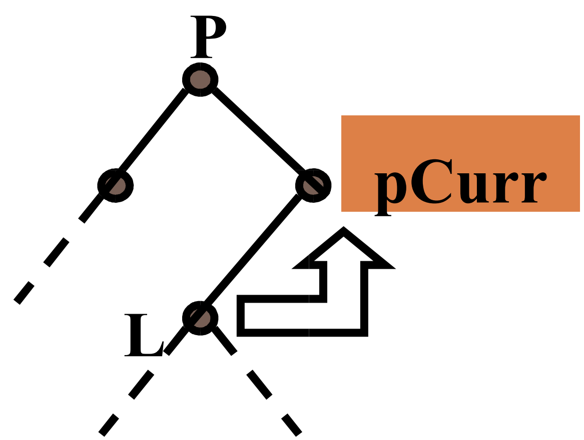
Delete an element with left child

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

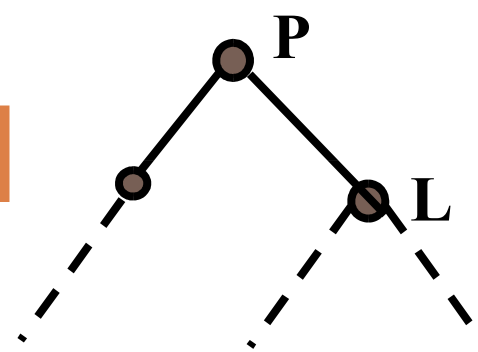


Delete an element with left child

- Delete node with only the left child node



Before delete pCurr

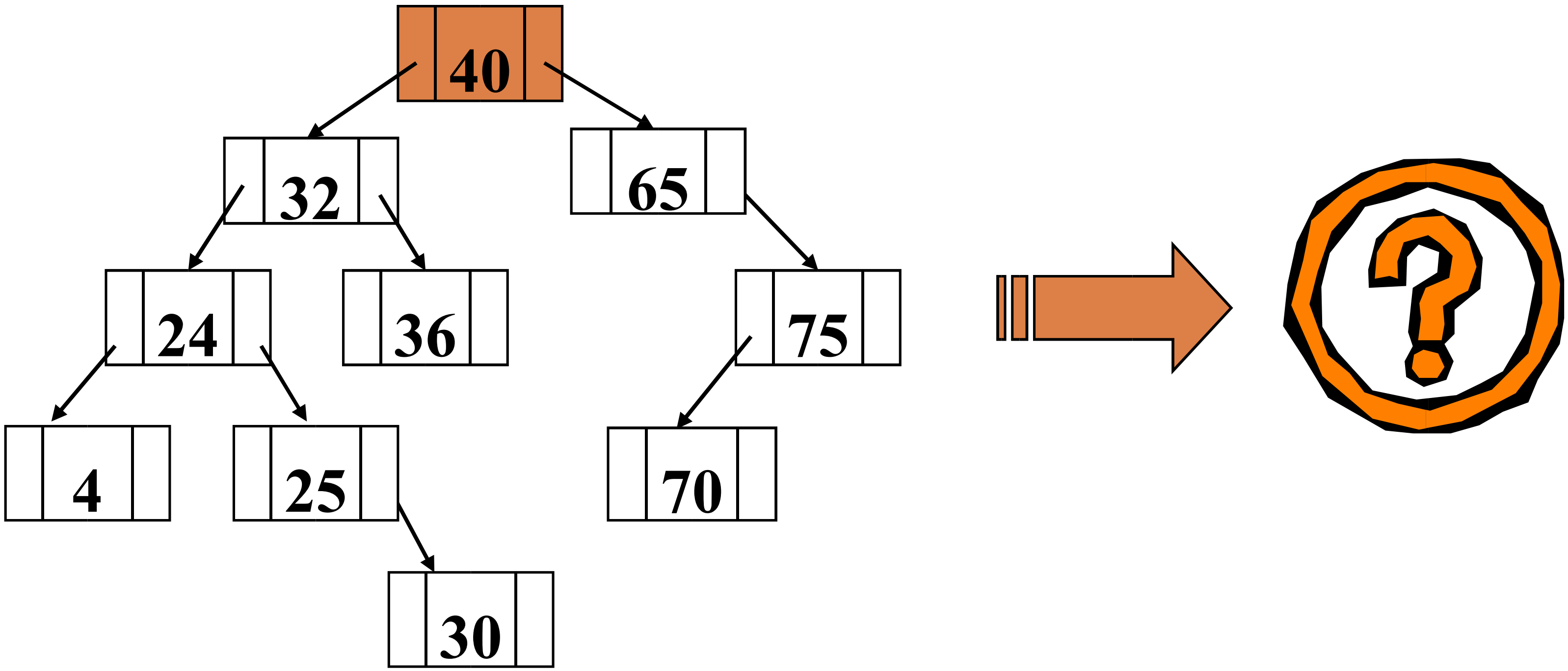


After delete pCurr

```
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,

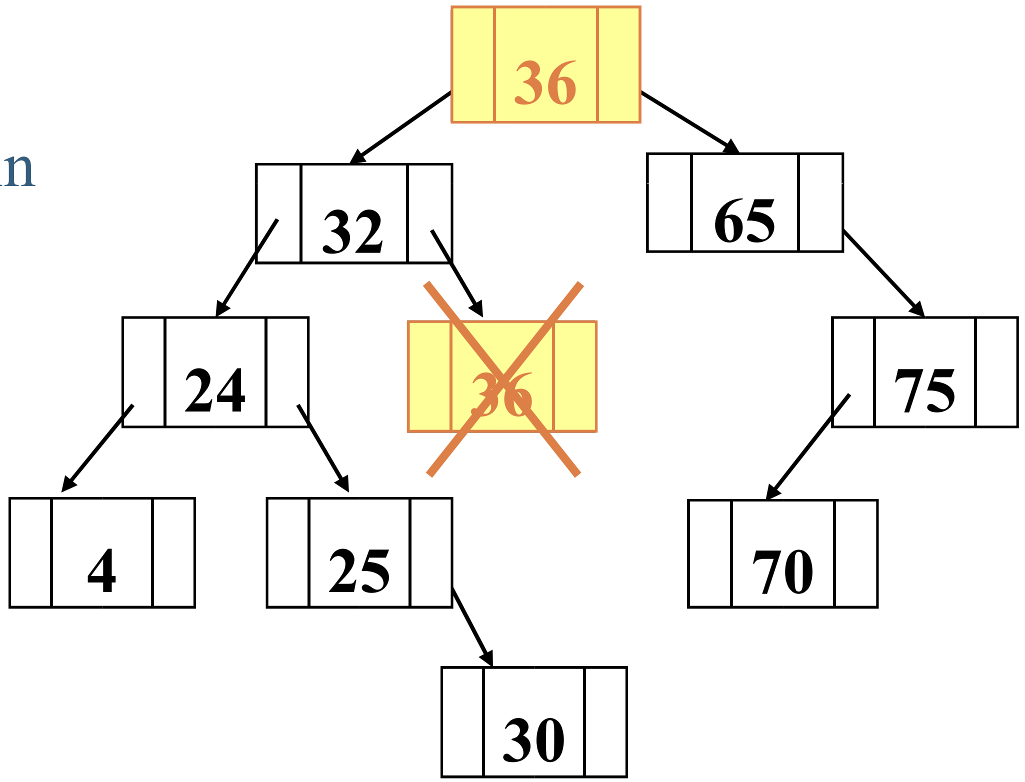
    ... copy data of p to pCurr,

    ... 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):

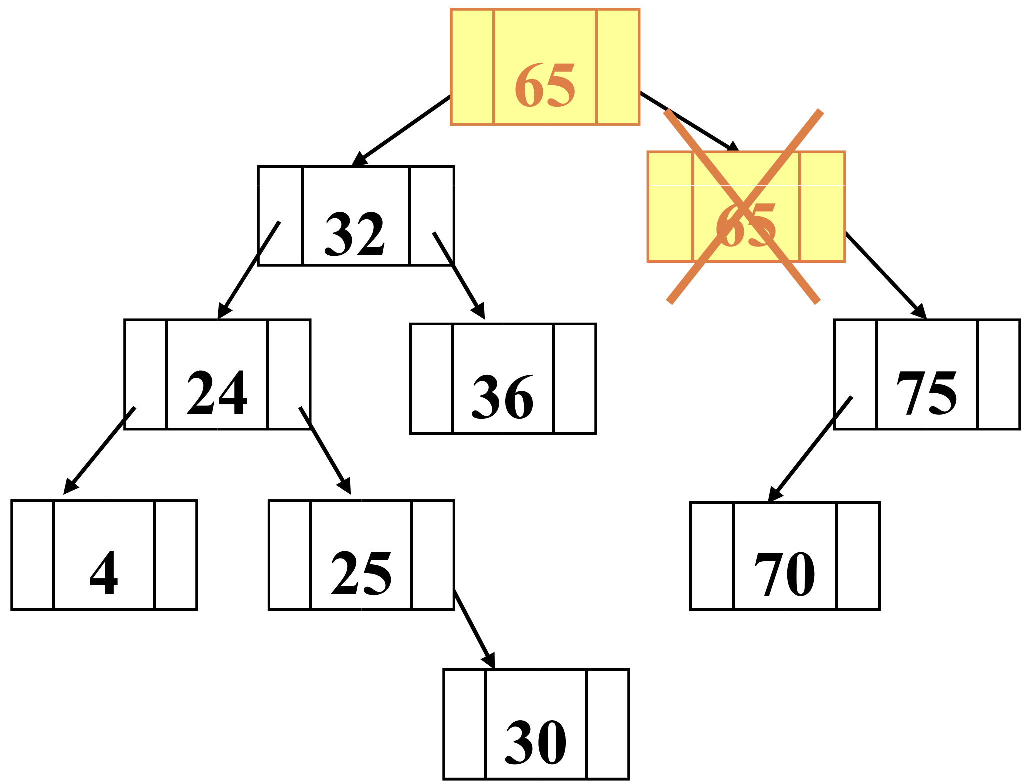
Way 1: substitute the largest element in the left subtree



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

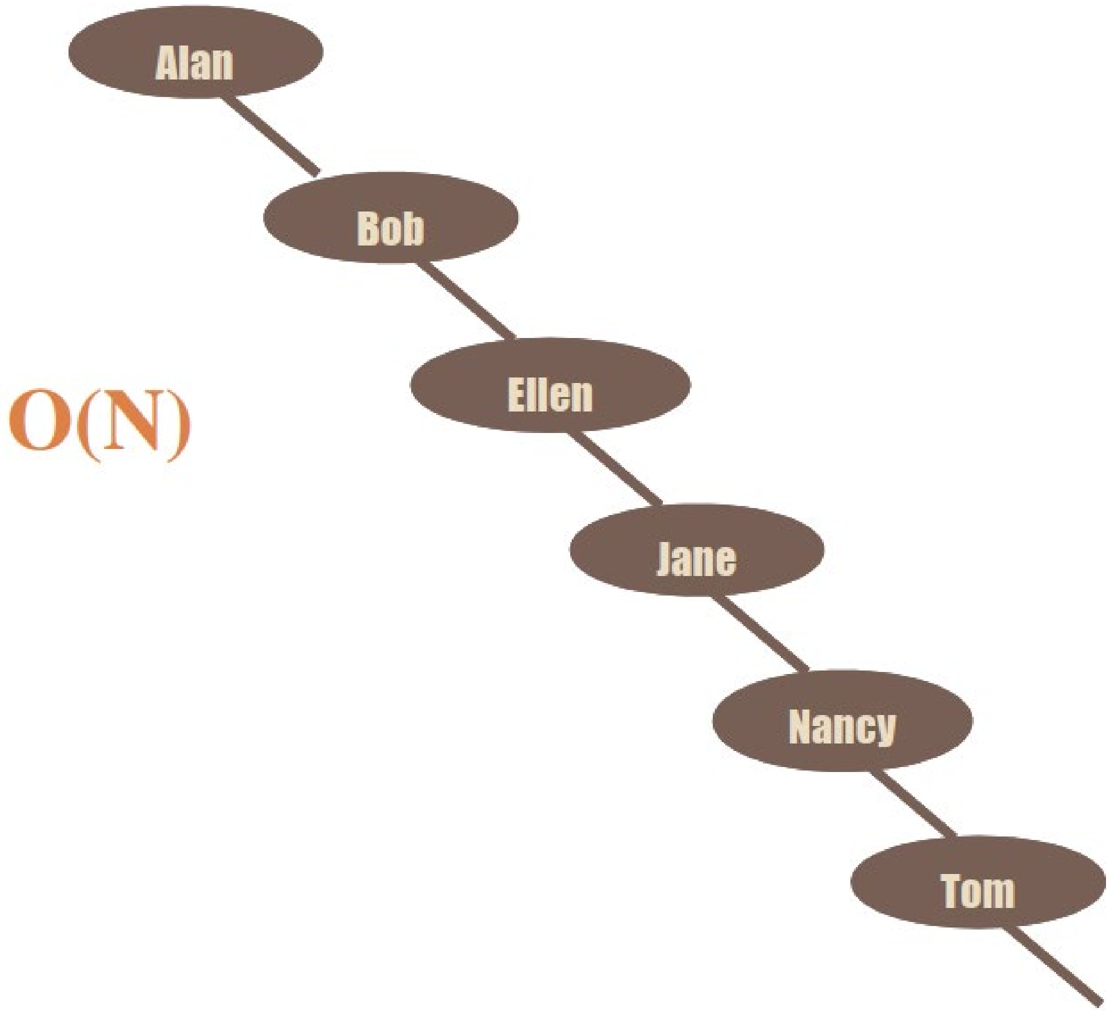
```
BT_NODE *  _SearchStandFor(BT_NODE *&p, BT_NODE *pCurr)
{
        //Find the element to substitute
        if (p->pRight != NULL)
                return _SearchStandFor(p->pRight, pCurr);

        //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

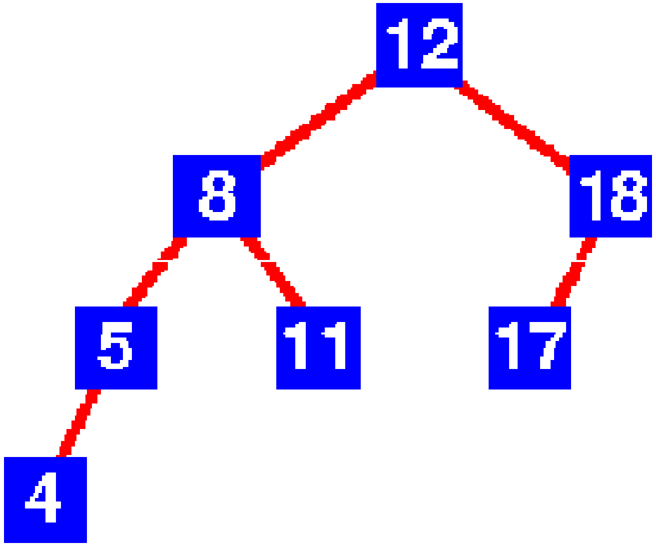
- AVL tree is a balanced BST tree
- AVL tree created by 3 authors: Adelson, Velskii, Landis proposed in 1962
- This is the first proposed dynamic balanced tree model
- The AVL tree does not have "absolute" balance, but the two child-tree never have a height difference of more than 1

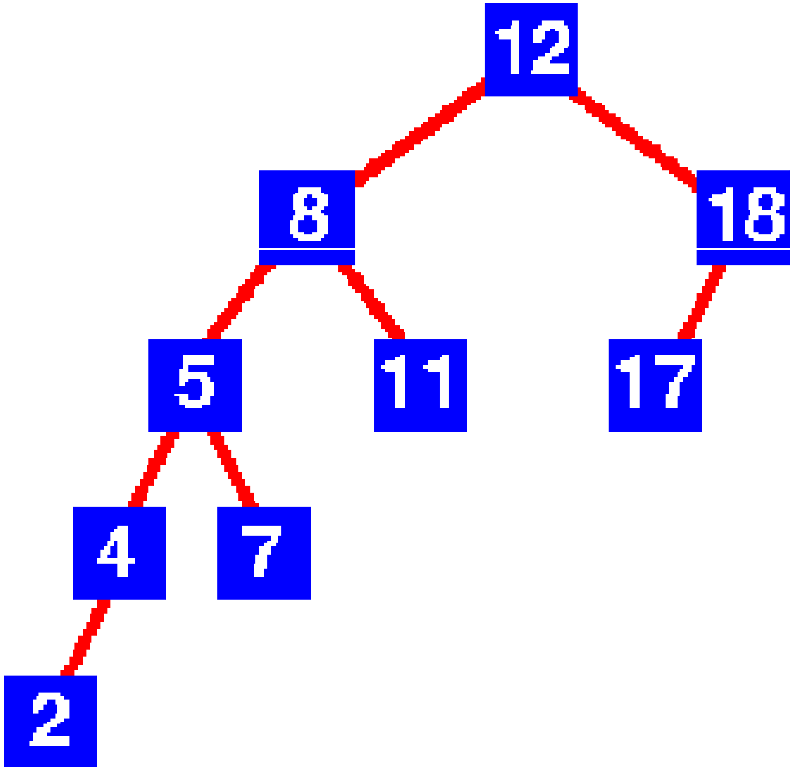
AVL

- 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.

$$\forall p \in T_{AVL}: abs\,(h_p -> pLeft - h_p -> pRight) \leq 1$$

Example



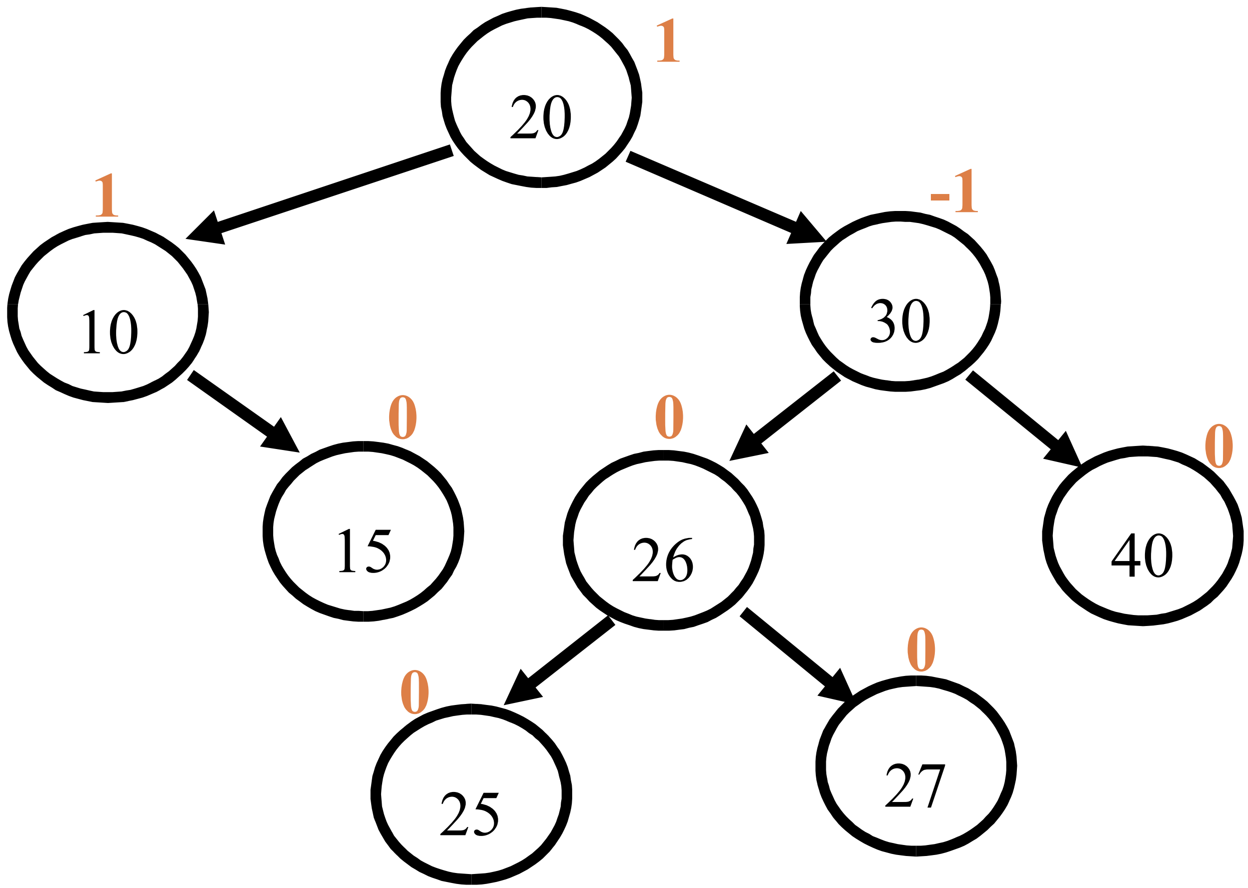


Which tree is AVL?

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)

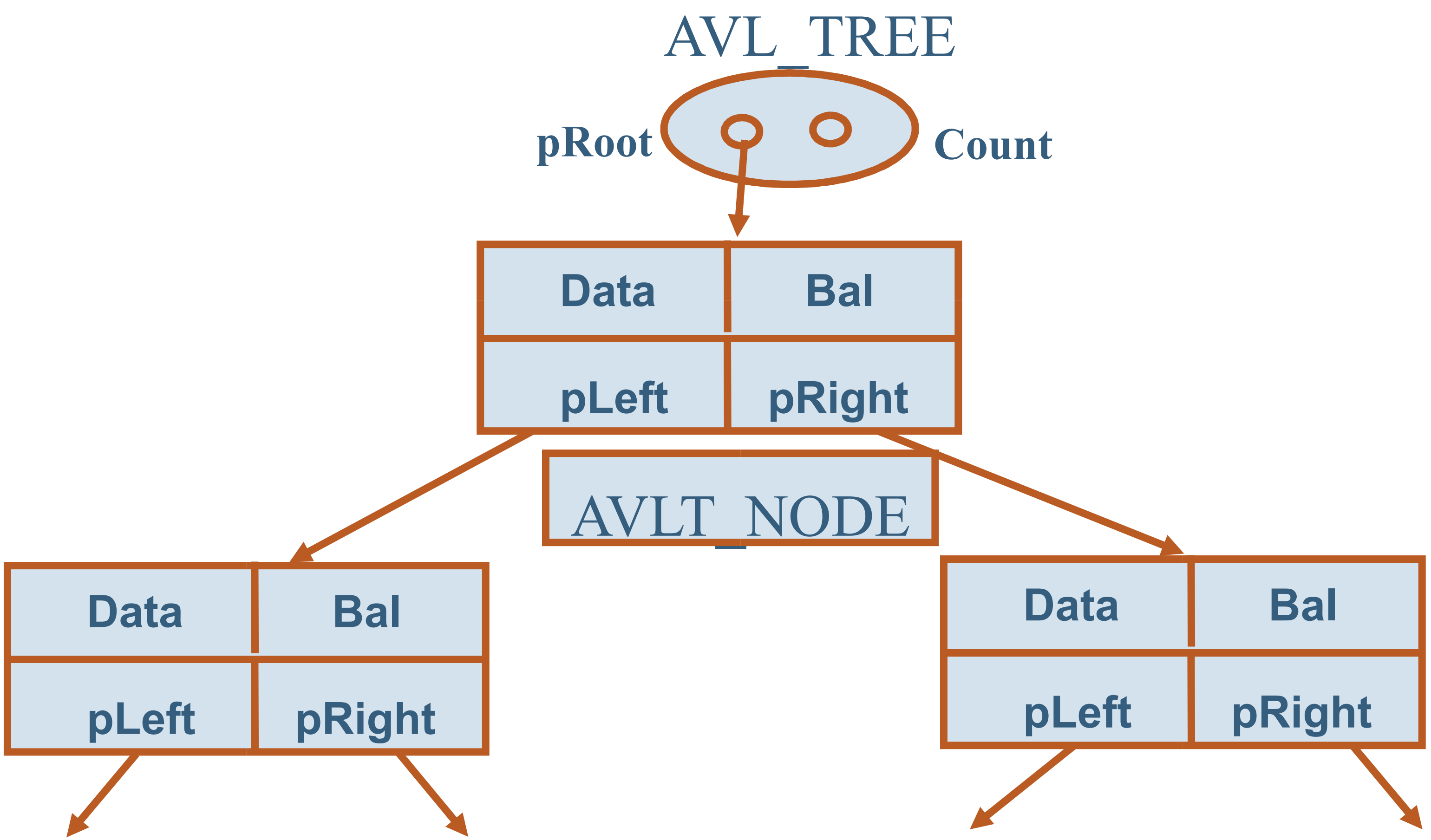
Balance



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

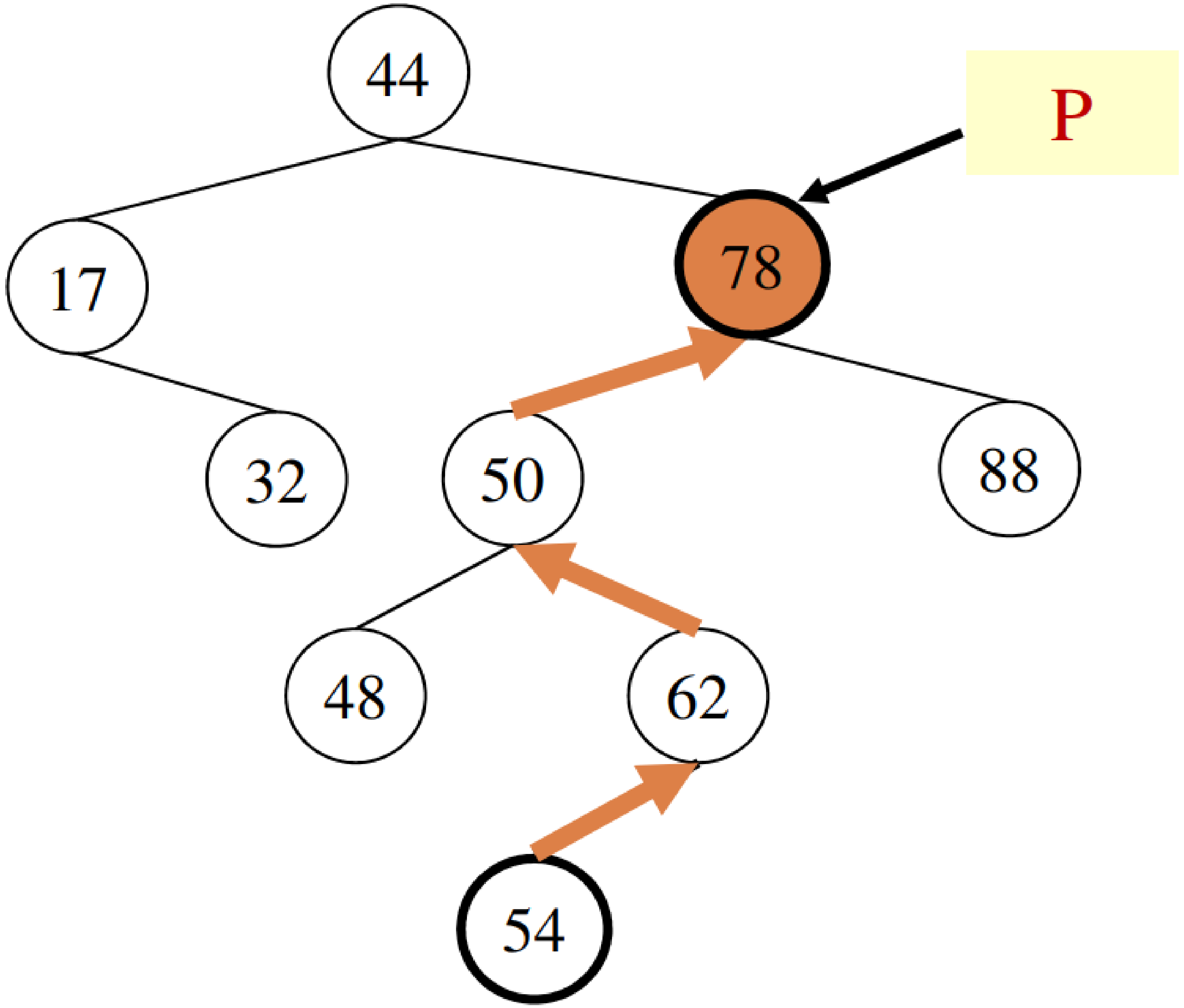
- Add an element
- Delete an element

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.

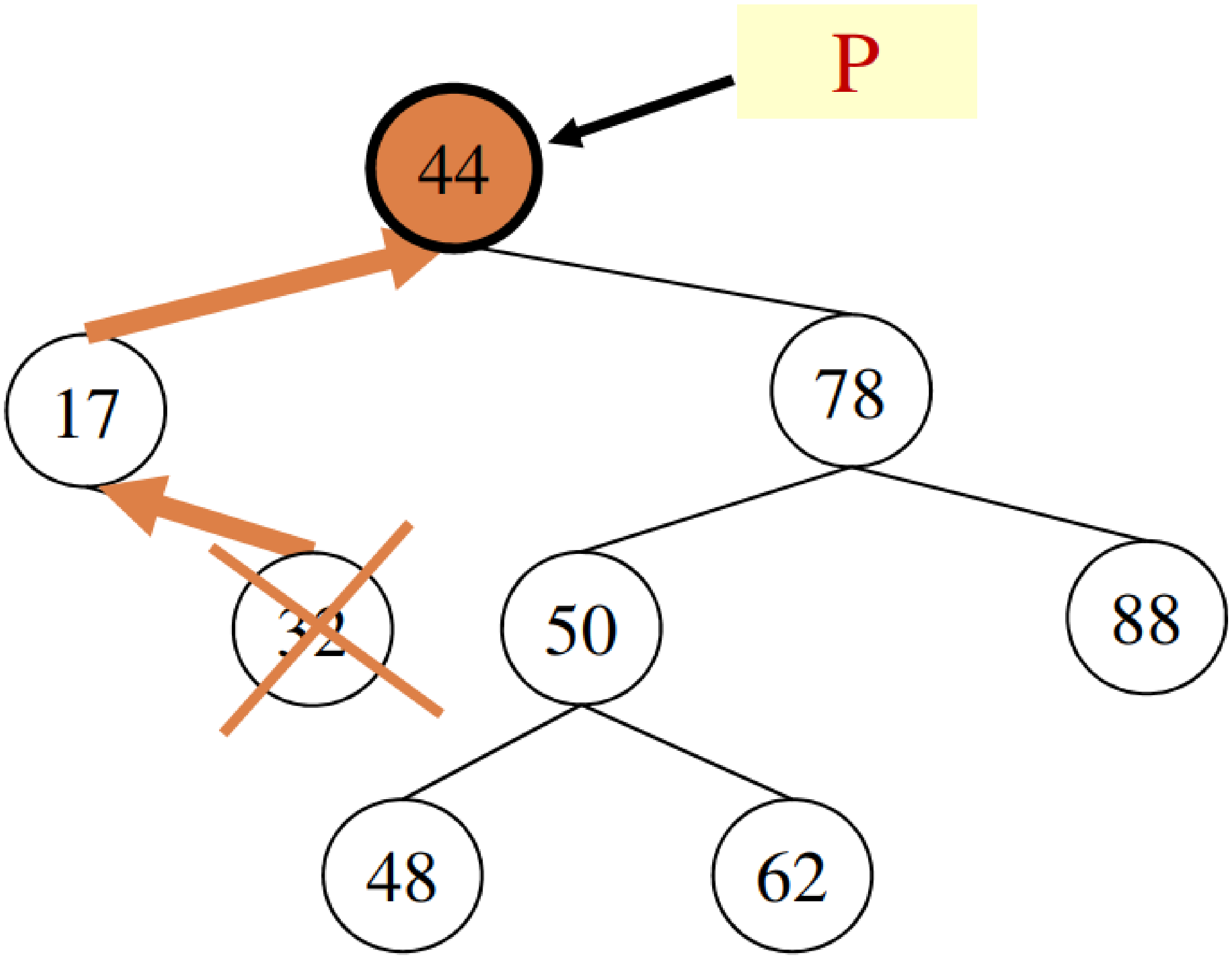
Find the unbalance node

- Adding new element make tree unbalance.

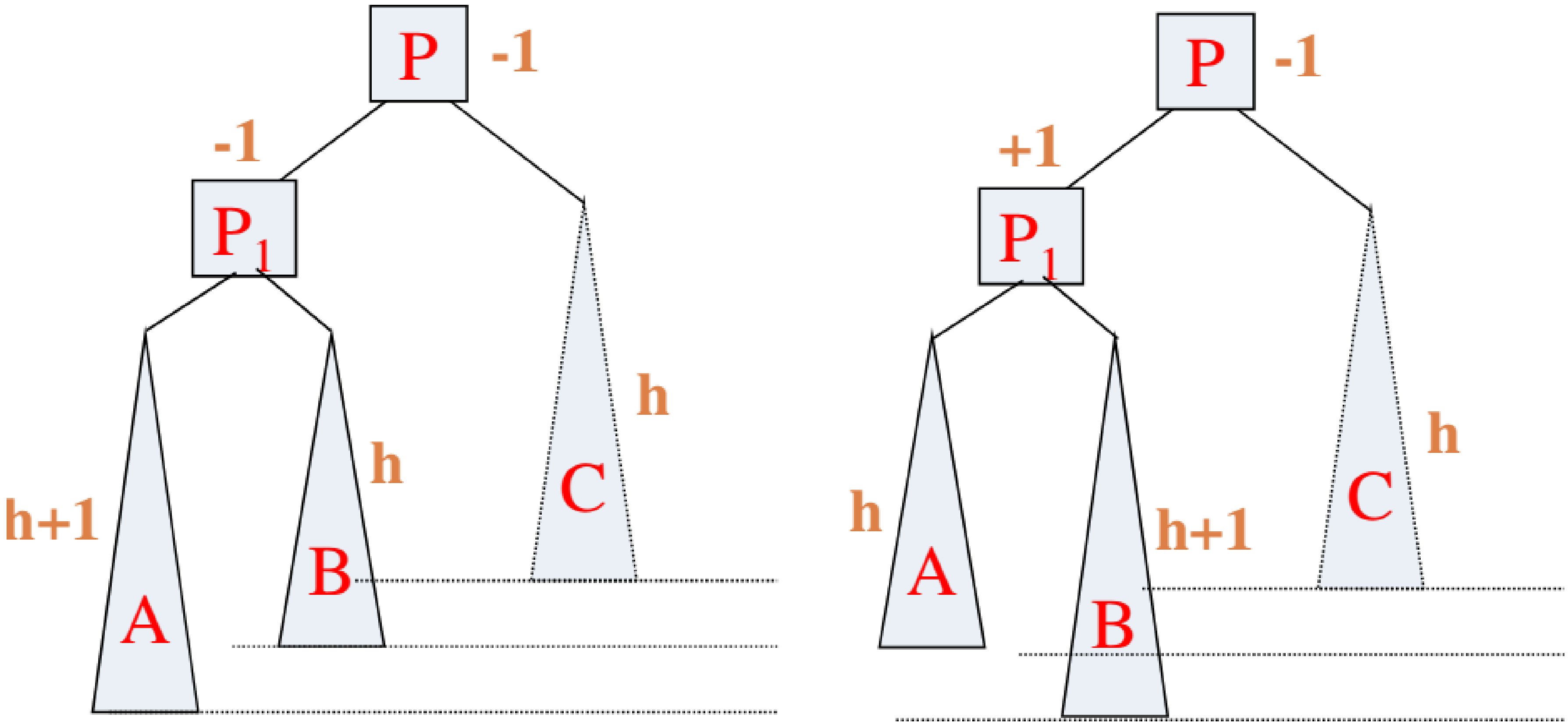


Find the unbalance node

- Deleting an element make tree unbalance

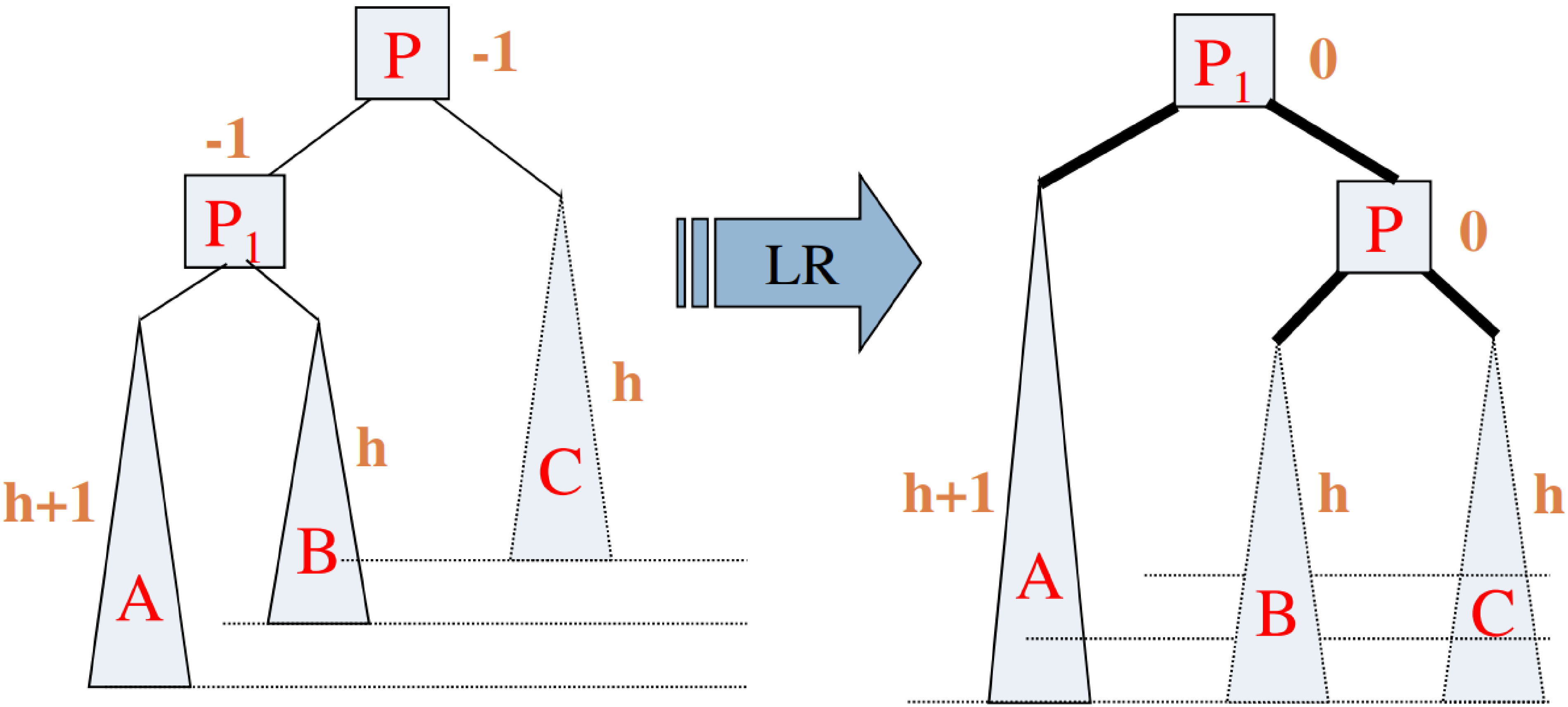


Adjust the tree that is left off



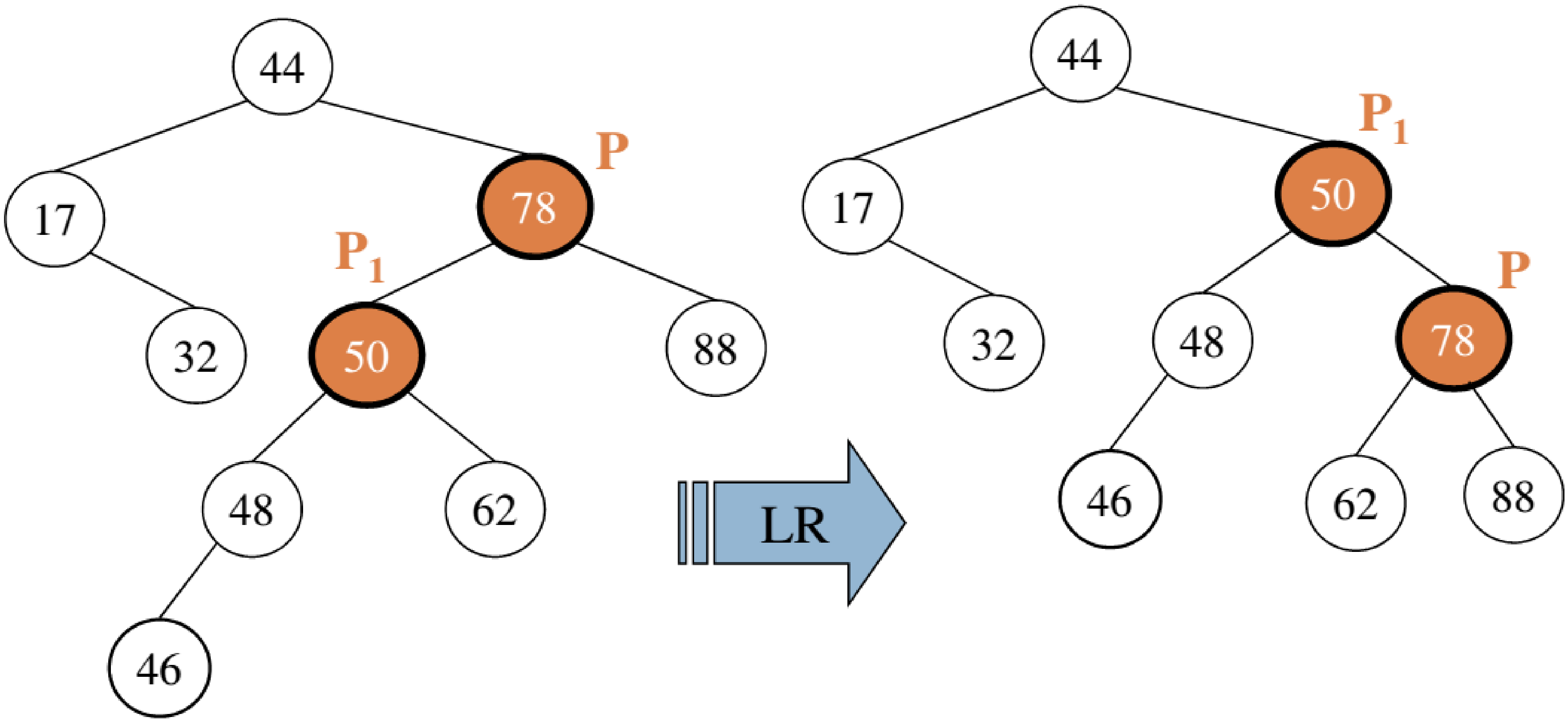
(a) (b)

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

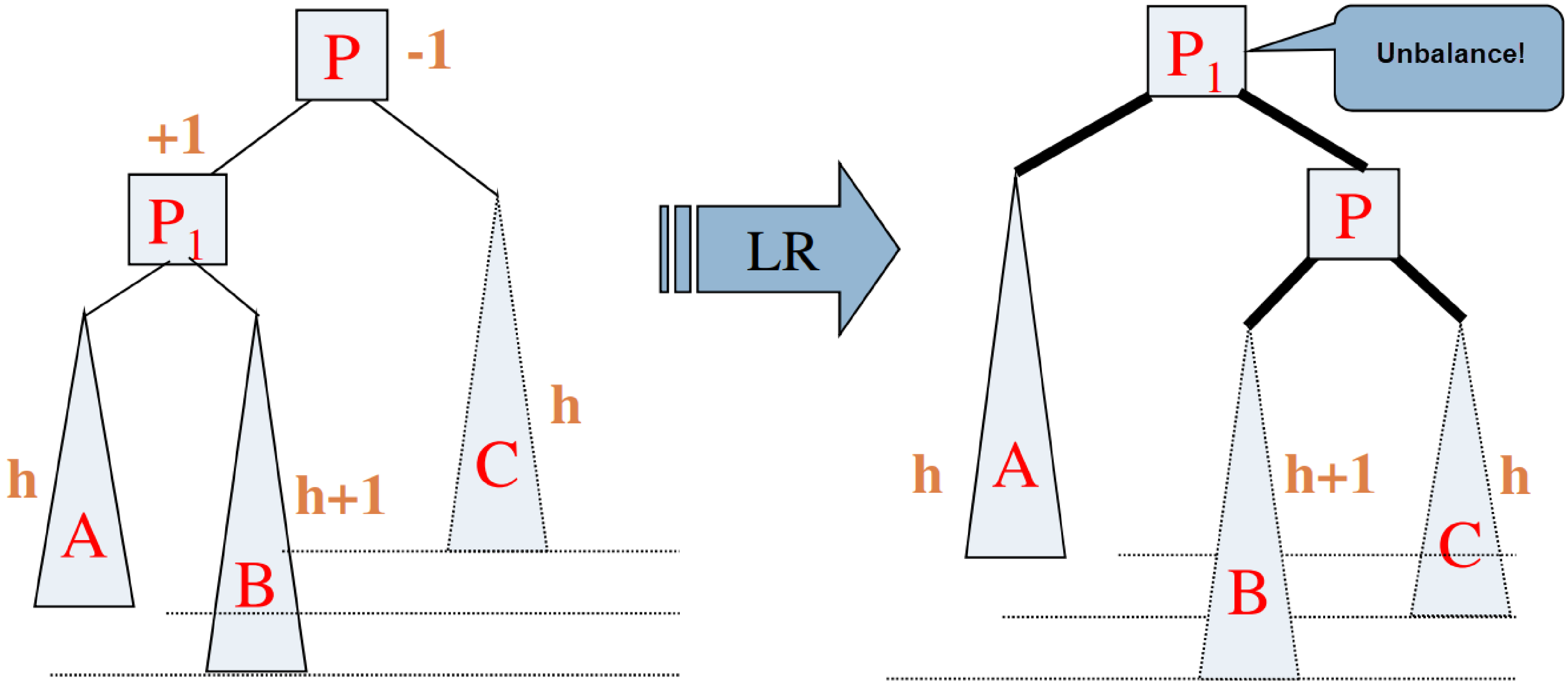


(LR – Single Left-Right)

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

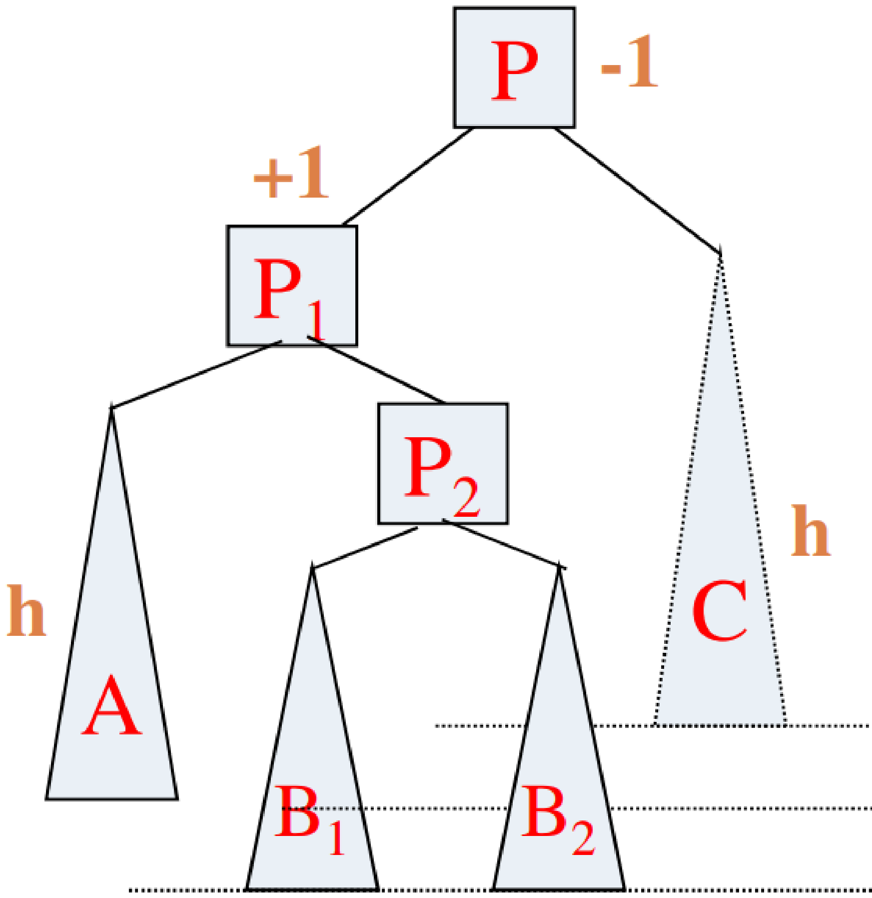


Adjust the tree that is left off – Case (b)

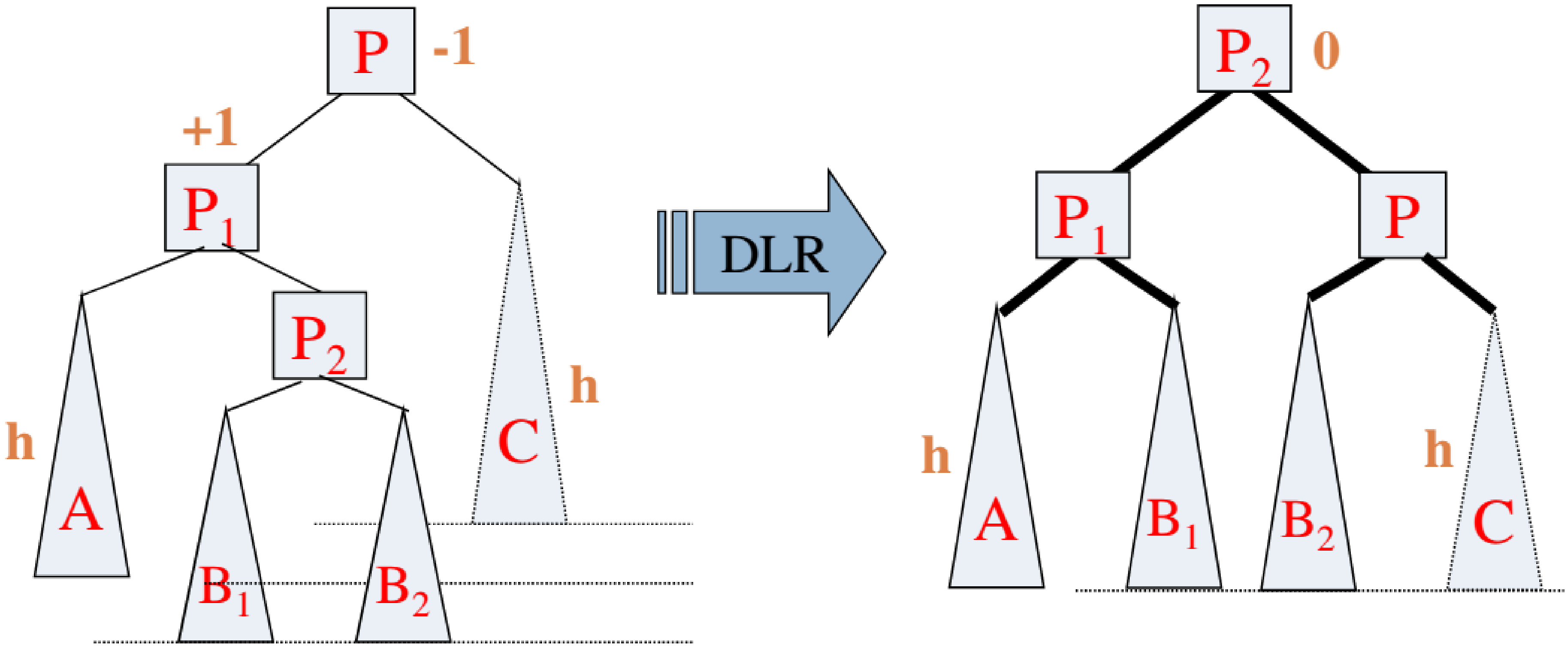


Adjust the tree that is left off – Case (b)

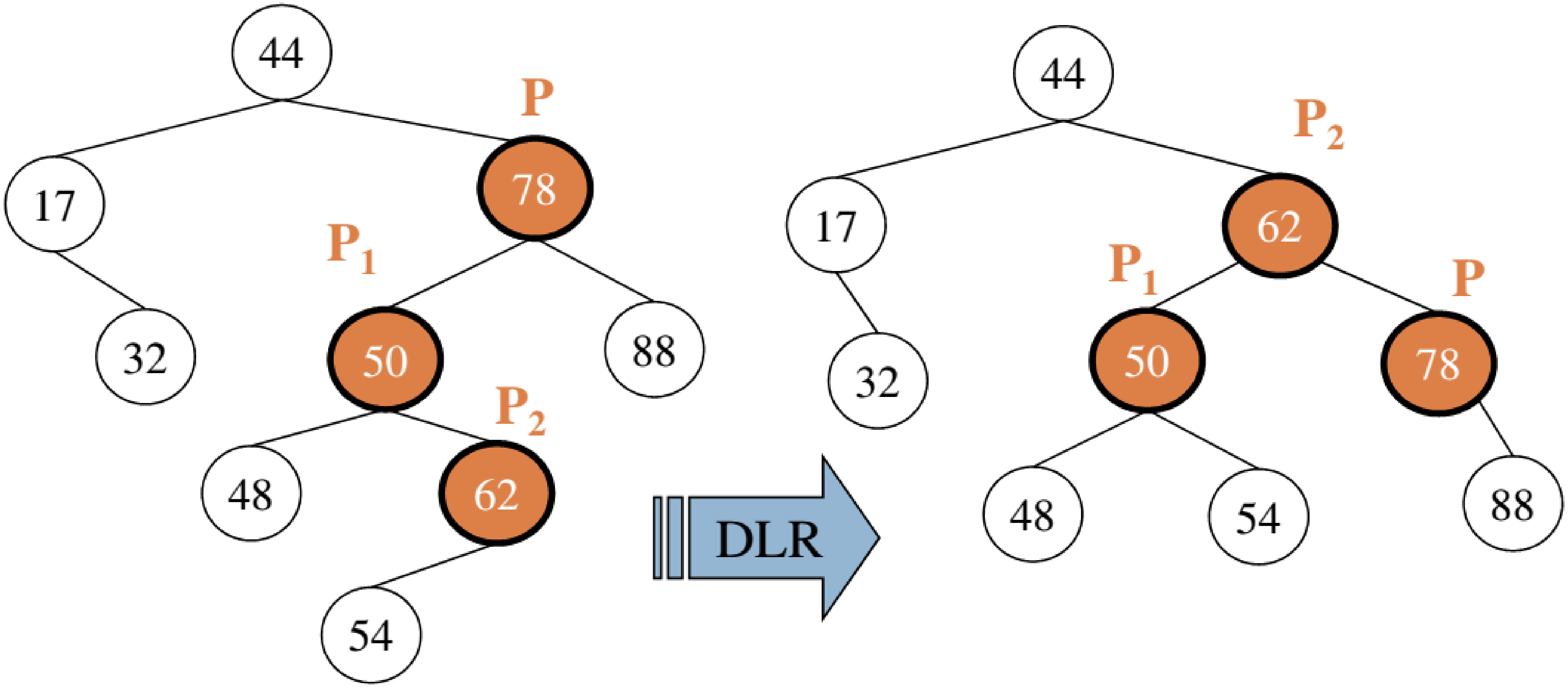
(DLR – Double Left – Right)



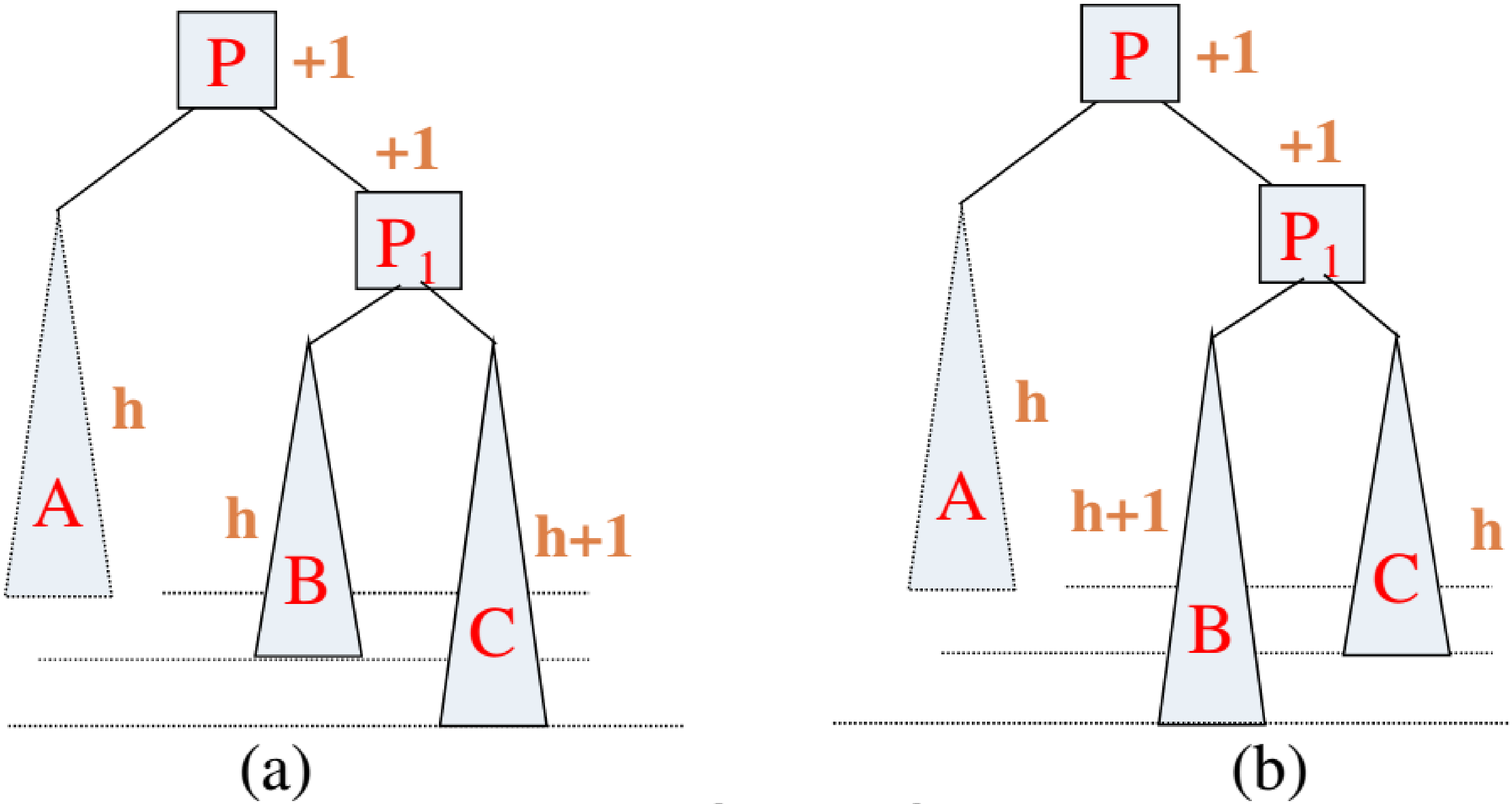
Adjust the tree that is left off – Case (b)



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



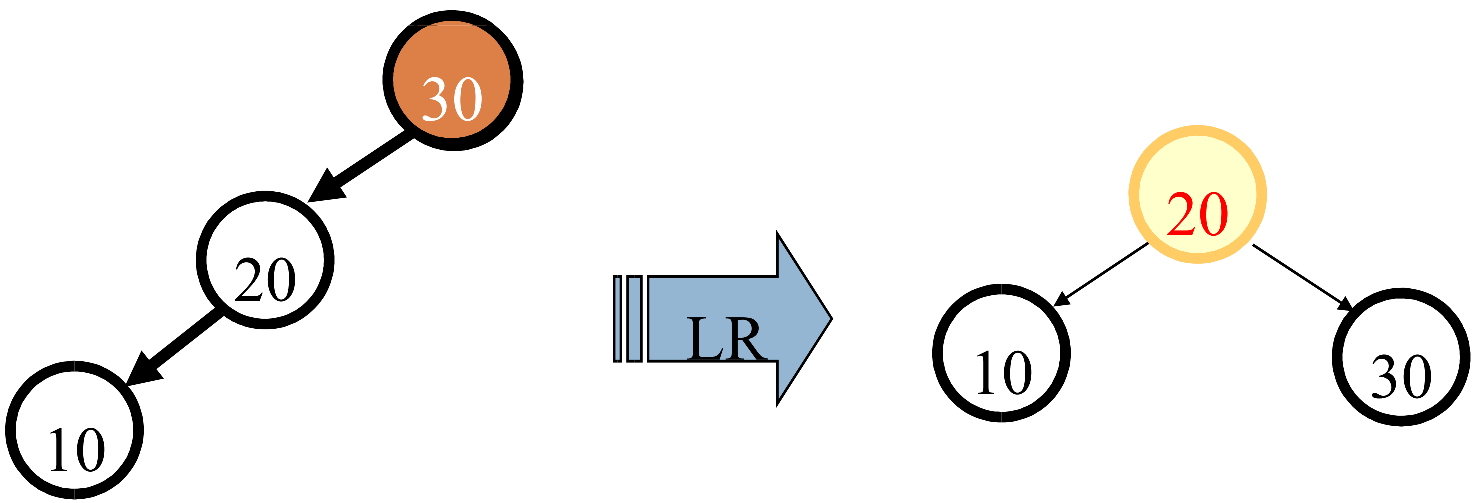
Adjust the tree that is right off



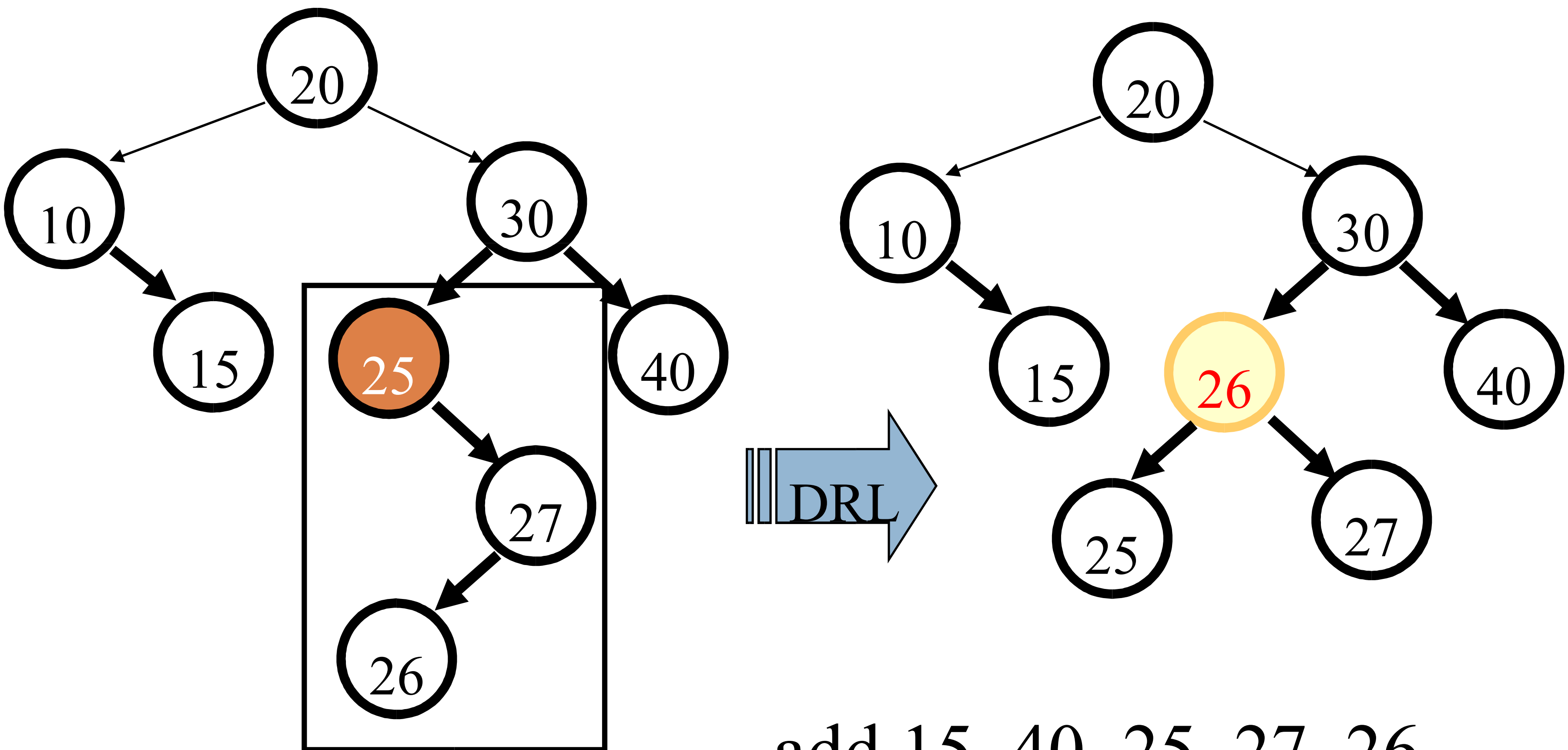
Do Similarly

Example

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

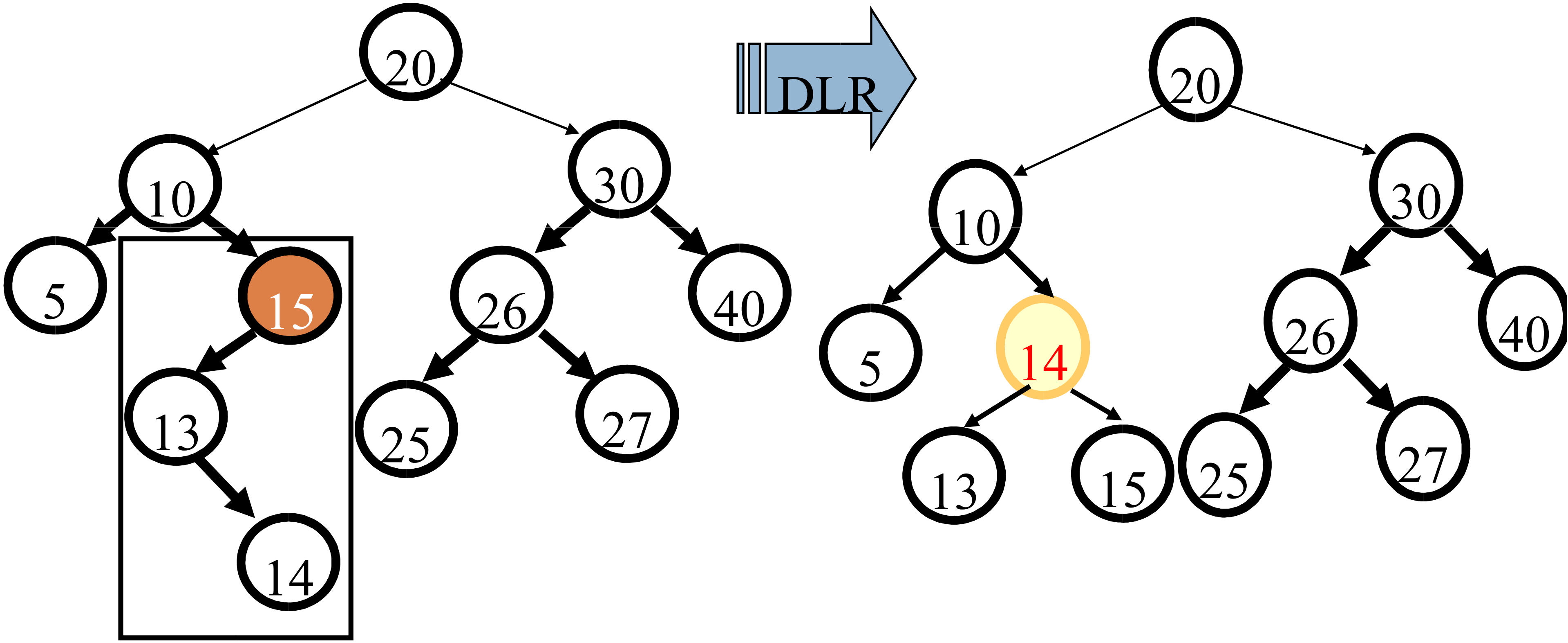


Example



…add 15, 40, 25, 27, 26

Example



…add 5, 13, 14

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: $O(\log_2 N)$

The End.