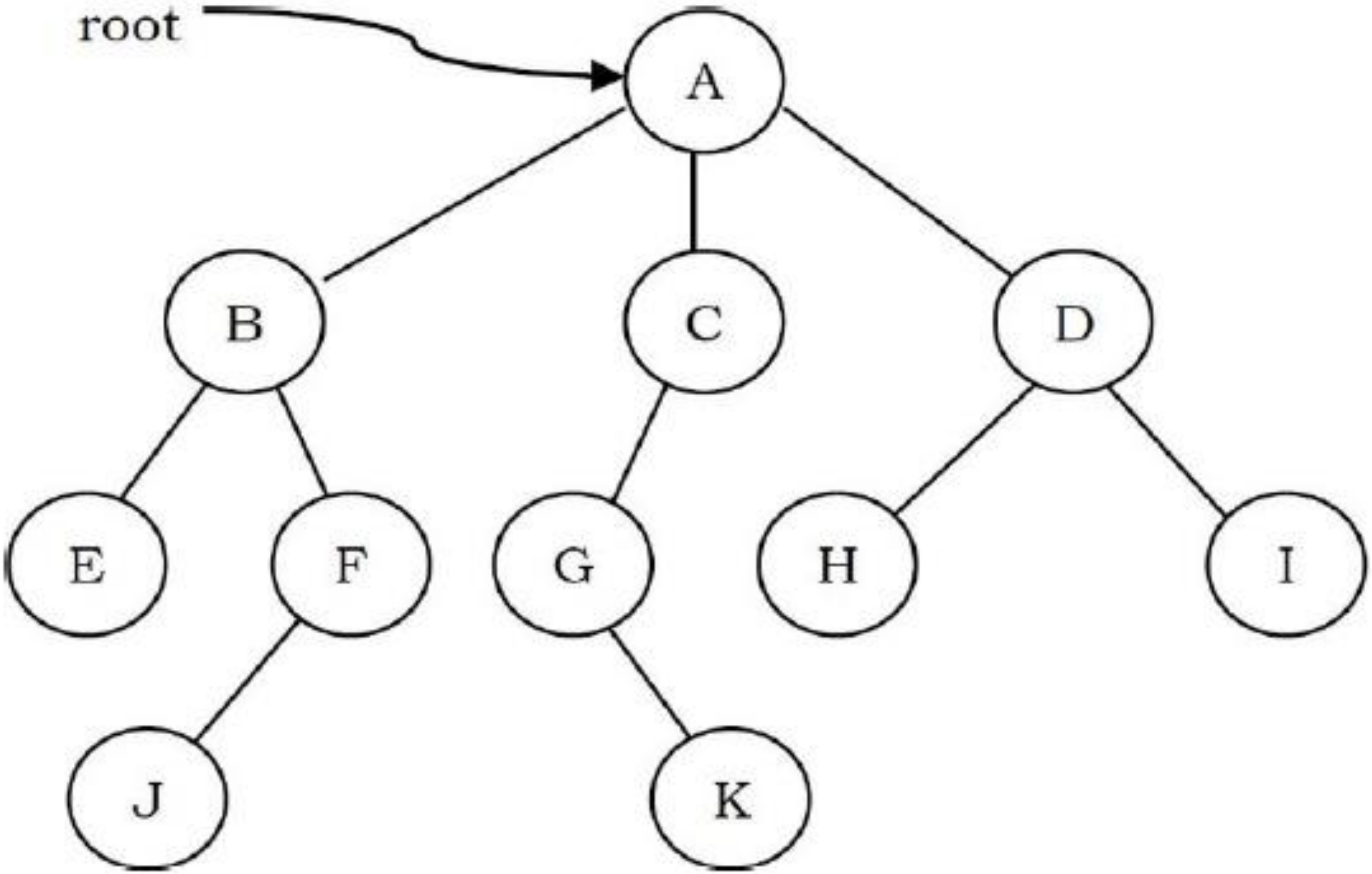
Trees

What is a Tree?

- A *tree* is a data structure similar to a linked list but instead of each node pointing simply to the next node in a linear fashion, each node points to a number of nodes.
- **Tree is an example of a nonlinear data structure.**
- A ***tree* structure** is a way of representing the **hierarchical nature of a structure in a graphical form.**
- In trees ADT (Abstract Data Type), the order of the elements is not important.

Glossary

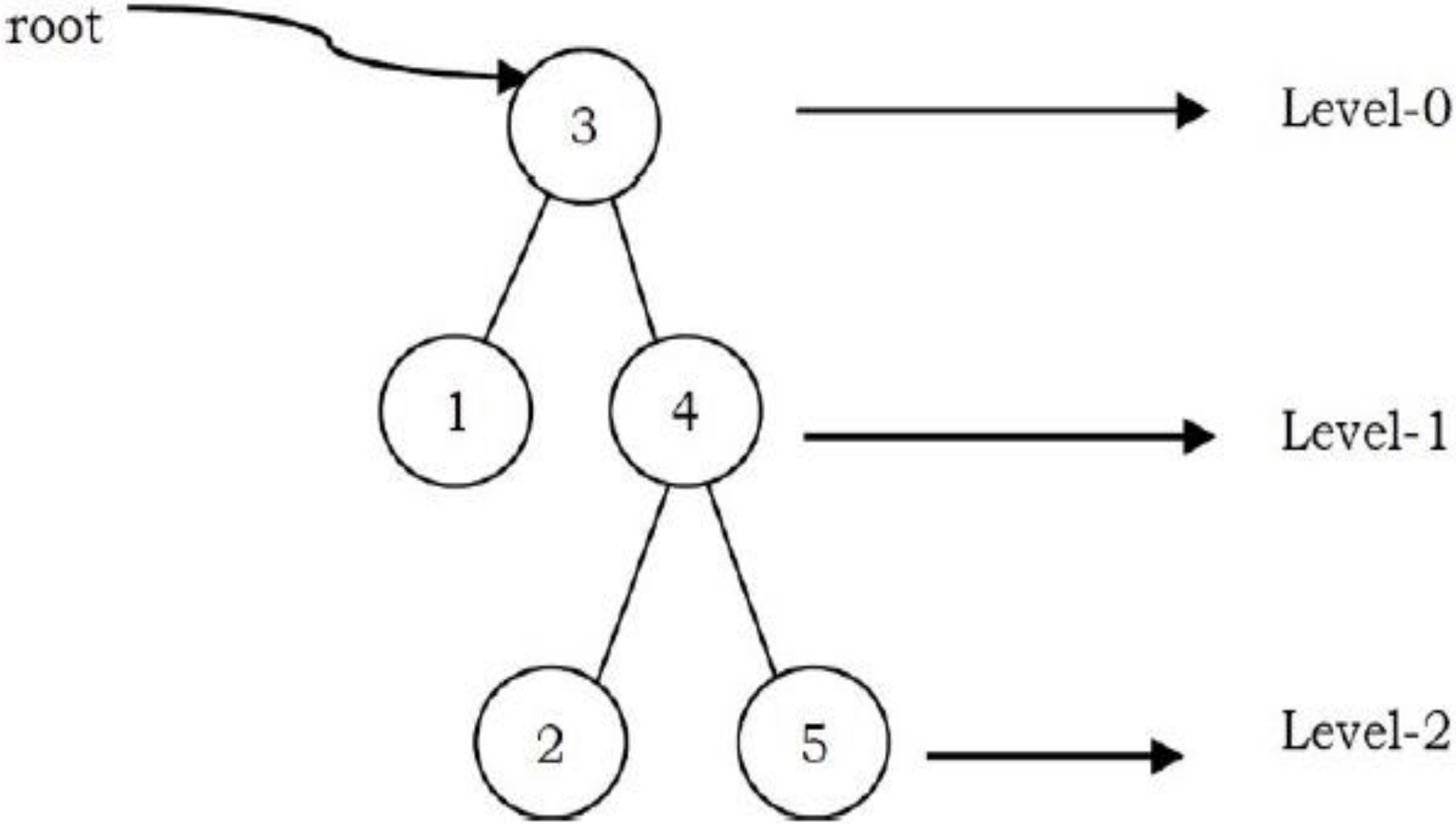


Definitions

- **Root:** The *root* of a tree is the node with **no parents**. There can be at most **one root node in a tree (*n*ode *A* in the above example)**.
- **Edge:** An ***edge* refers** to the **link** from **parent to child** (all links in the figure).

- **Leaf:** A node with **no children** is called *leaf* node (*E,J,K,H* and *I*).
- **Sibblings: Children of same parent** are called *siblings* (*B,C,D* are siblings of *A*, and *E,F* are the siblings of *B*).

- **Ancestor of a Node:** A node p is an *ancestor* of node *q* **if there exists a path from *root* to *q* and p appears on the path. The node *q* is called a *descendant* of p**.
  - For example, ***A,C* and *G*** are the **ancestors of K**.
- ***Level:*** The set of all nodes at a **given depth is called the *level* of the tree** (*B, C* and *D* are the same level). The **root node** is at **level zero**.

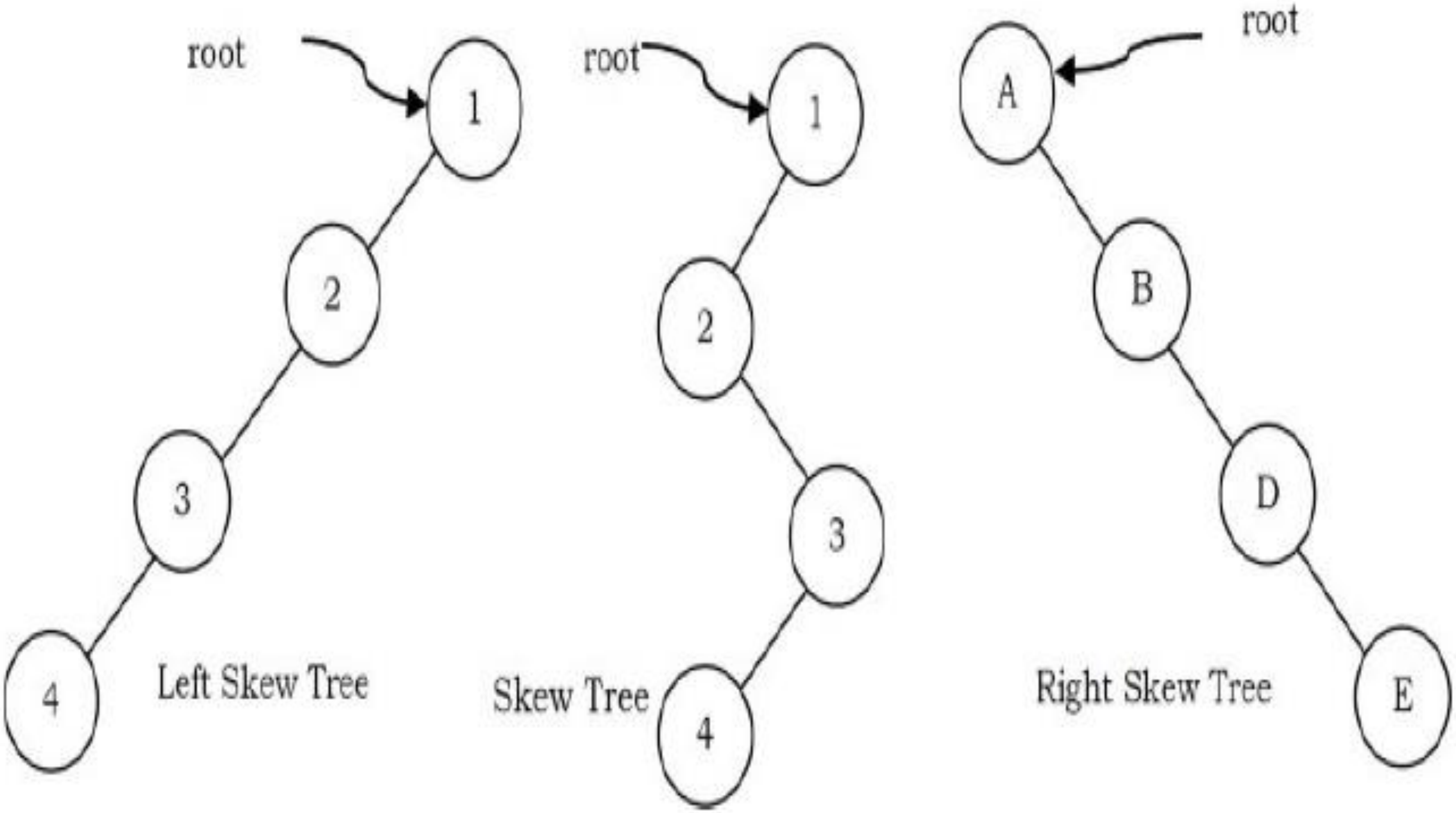

root
3
1
4
2
5
Level-0
Level-1
Level-2

- ***Depth* of a node :**The ***depth*** of a node is the **length of the path from the root to the node** (depth of *G* is 2, *A* – *C* – *G*).
- ***Height* of a node:** The ***height*** of a node is the **length of the path from that node to the deepest node.**
  - A (rooted) tree with only one node (the root) has a height of zero. In the previous example, the height of *B* is 2 (*B* – *F* – *J*).

- ***Height of the tree:*** *It* is the maximum height among all the nodes in the tree
- ***Depth of the tree:*** *Depth of the tree* is the **maximum depth among all the nodes** in the tree.
  - For a given tree, depth and height returns the same value. But for individual nodes we may get different results.
- **Size of a node:** It is the **number of descendants** it has **including itself** (the size of the subtree is 3).

- **Skew Trees:** If **every node** in a tree has **only one child (except leaf nodes)** then we call such trees ***skew trees***.
  - If every node has **only left child then we call them *left skew trees*.**
  - If every node has **only right child then we call them *right skew trees*.**

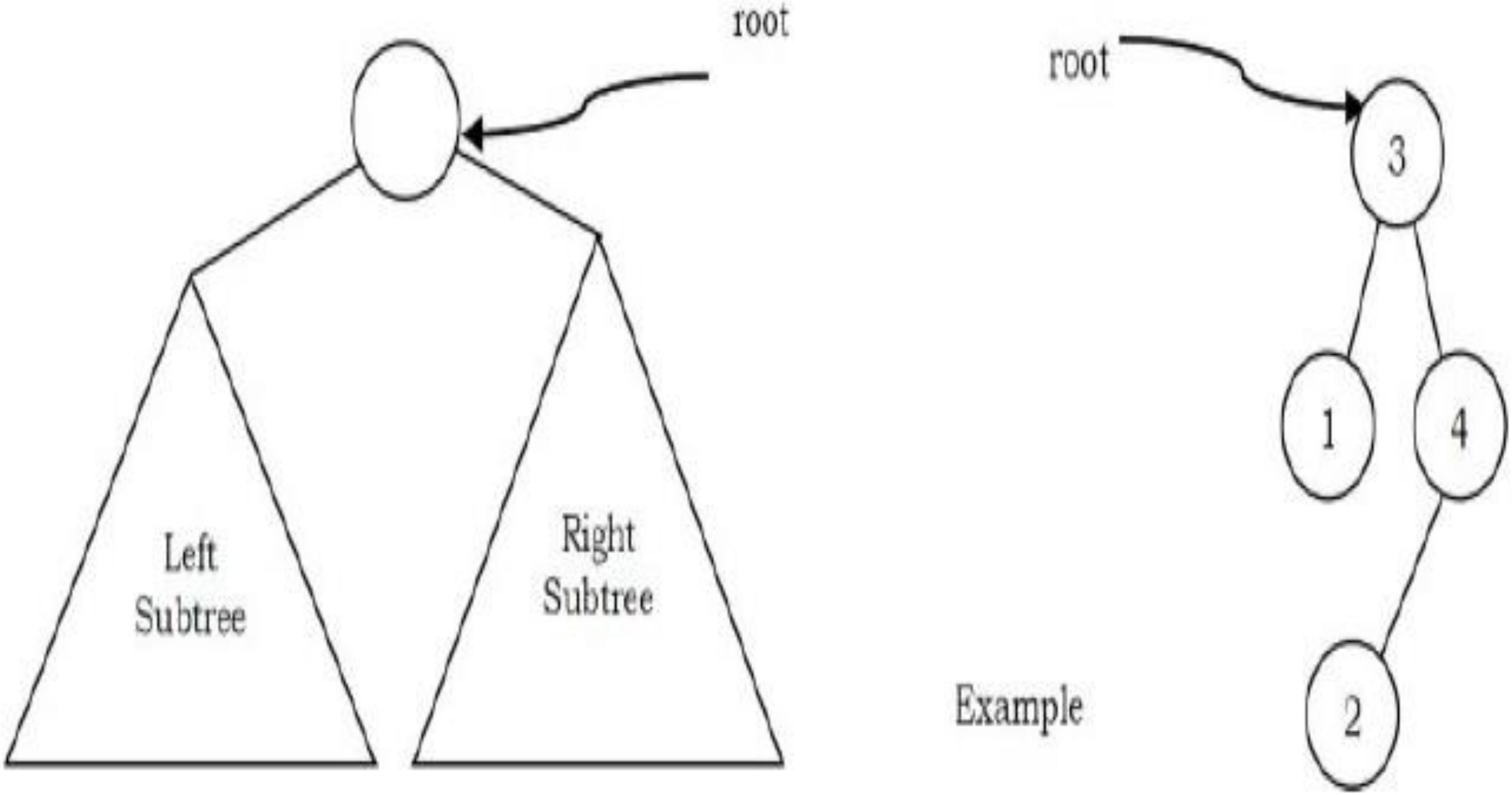
Skew Trees


root
1
2
3
4
Left Skew Tree
root
1
2
3
4
Skew Tree
root
A
B
D
E
Right Skew Tree

Binary Trees

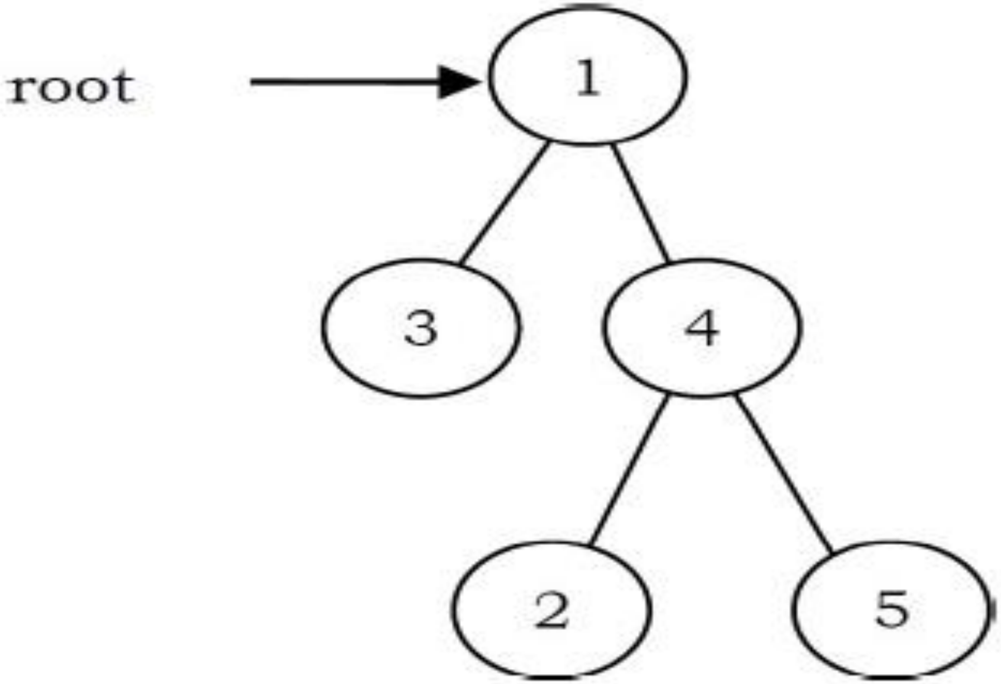
- ***Binary tree:*** A tree is called *binary tree* **if each node has zero child**, **one child or two children**.
- **Empty tree is also a valid binary tree.**
- A binary tree as consisting of a **root and two disjoint binary trees**, called **the left and right subtrees of the root**.

Generic Binary Tree



Types of Binary Trees

- **Strict Binary Tree:** A binary tree is called *strict binary tree* **if each node has exactly two children or no children.**

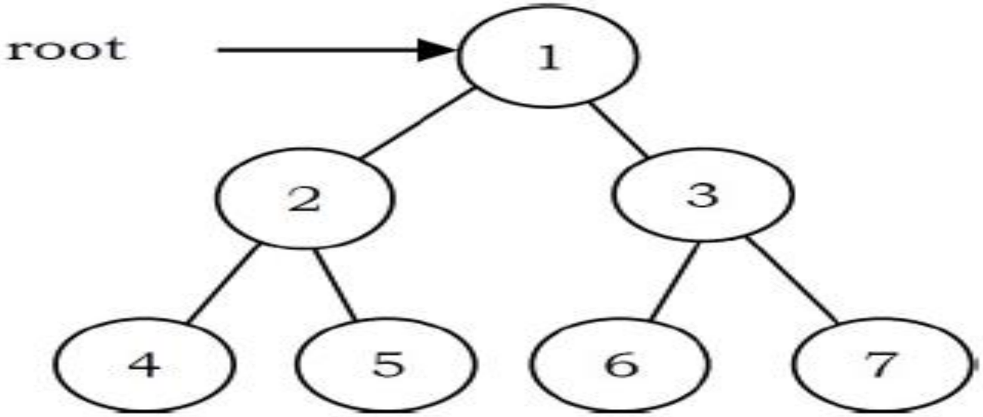


Full Binary Tree: A binary tree is called *full binary tree*

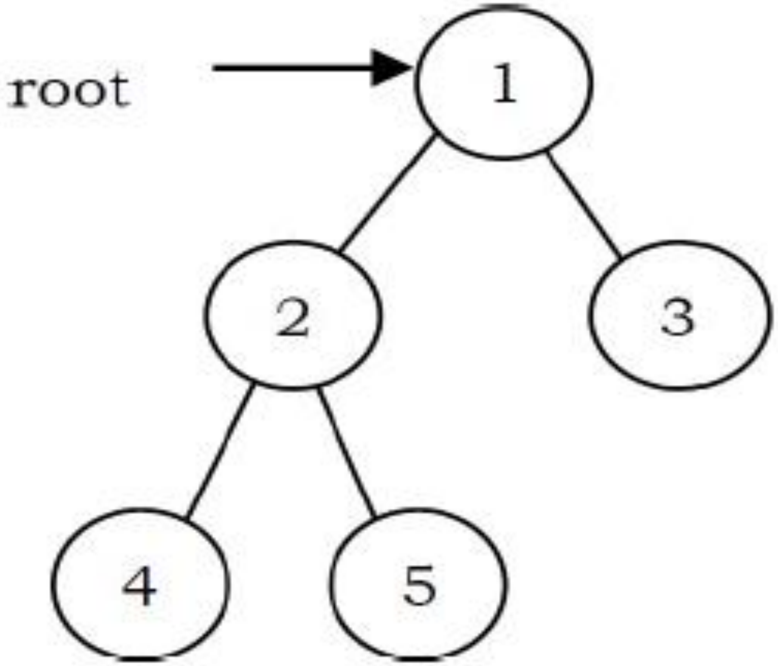
if each node has exactly two children

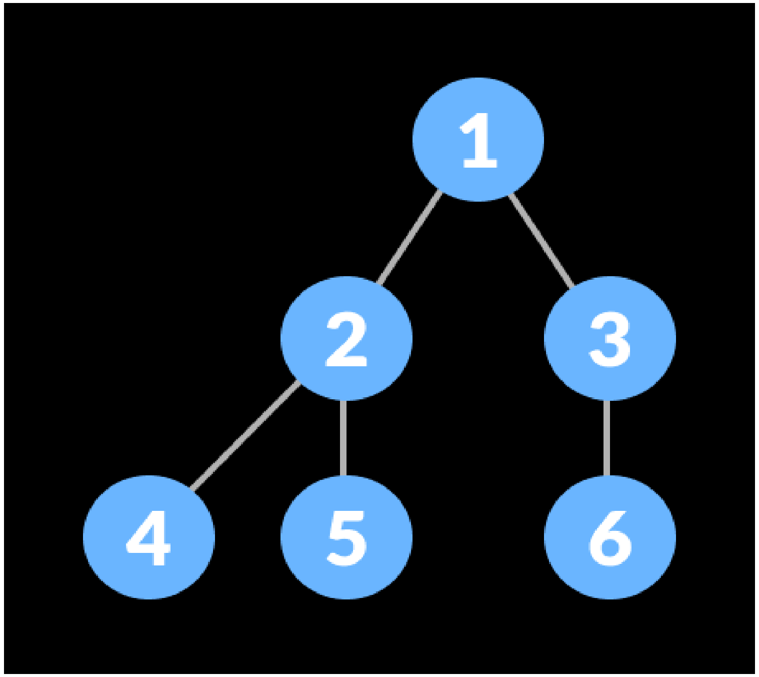
and

all leaf nodes are at the same level.

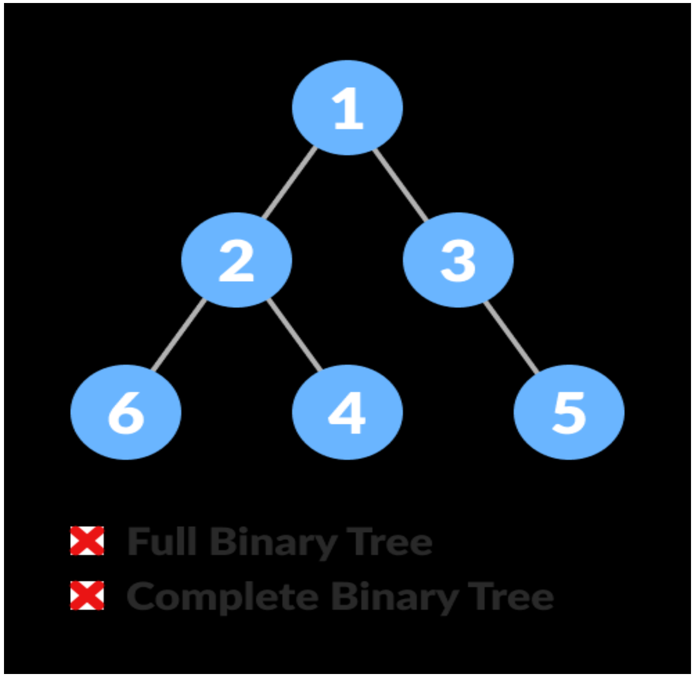


- **Complete Binary Tree:** Assume that the height of the binary tree is *h*.
  - In complete binary trees, if we give numbering for the nodes by **starting at the root** (let us say the root node has 1) then we get **a complete sequence from 1 to the number of nodes in the tree**.
  - While traversing we should give **numbering for NULL pointers also.**
- A binary tree is called ***complete binary tree***
  - **1) if all leaf nodes are at height *h* or *h* – 1** and
  - **2) without any missing number in the sequence**.

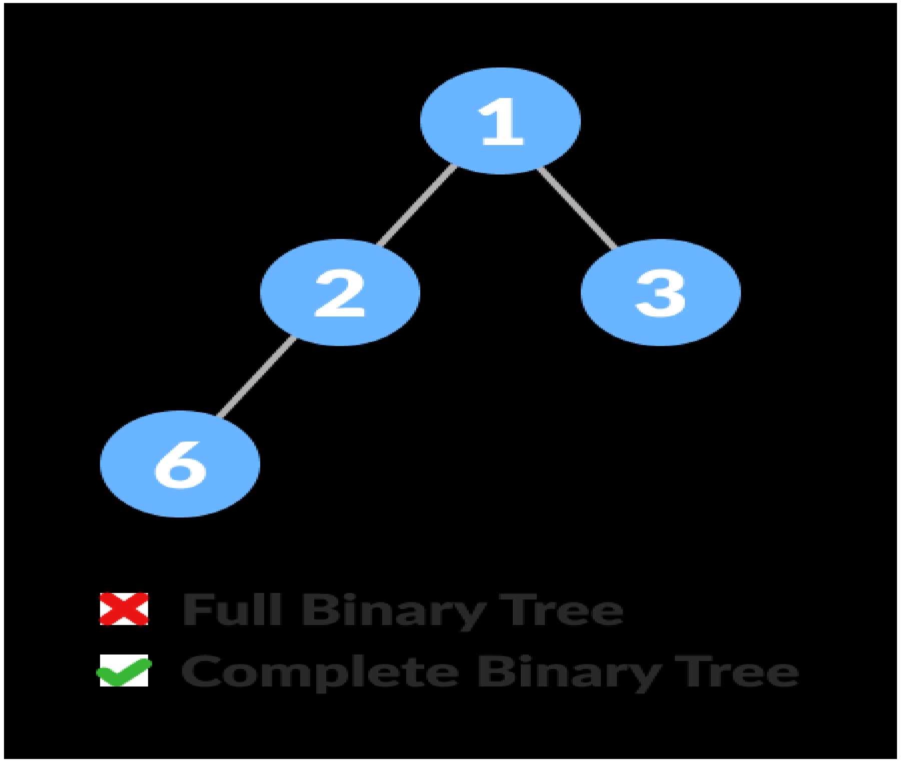

root
1
2
3
4
5



Complete Binary Tree



Not a Complete Binary Tree
Not Full Binary Tree



Complete Binary Tree
Not Full Binary Tree

```
•     Examples of Complete Binary Trees
•       1
•     /   \
•    2     3
•
•          1
•        /    \
•       2       3
•     /
•    4

•          1
•        /    \
•       2      3
•     /  \    /
•    4    5  6
```

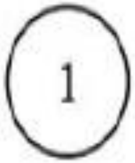
- The following trees are examples of Non-Complete Binary Trees

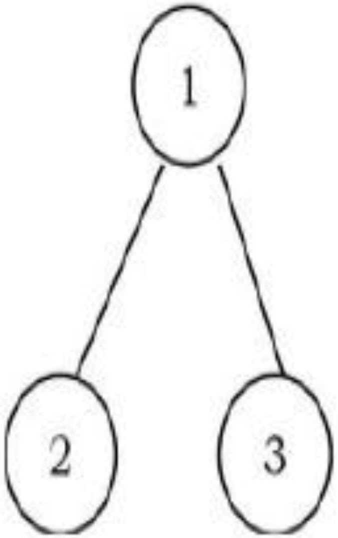
```
•    1
•      \
•       3
•
•       1
•    /    \
•   2         3
•    \      /   \
•     4    5     6

•       1
•    /    \
•   2        3
•          /   \
•         4     5
```

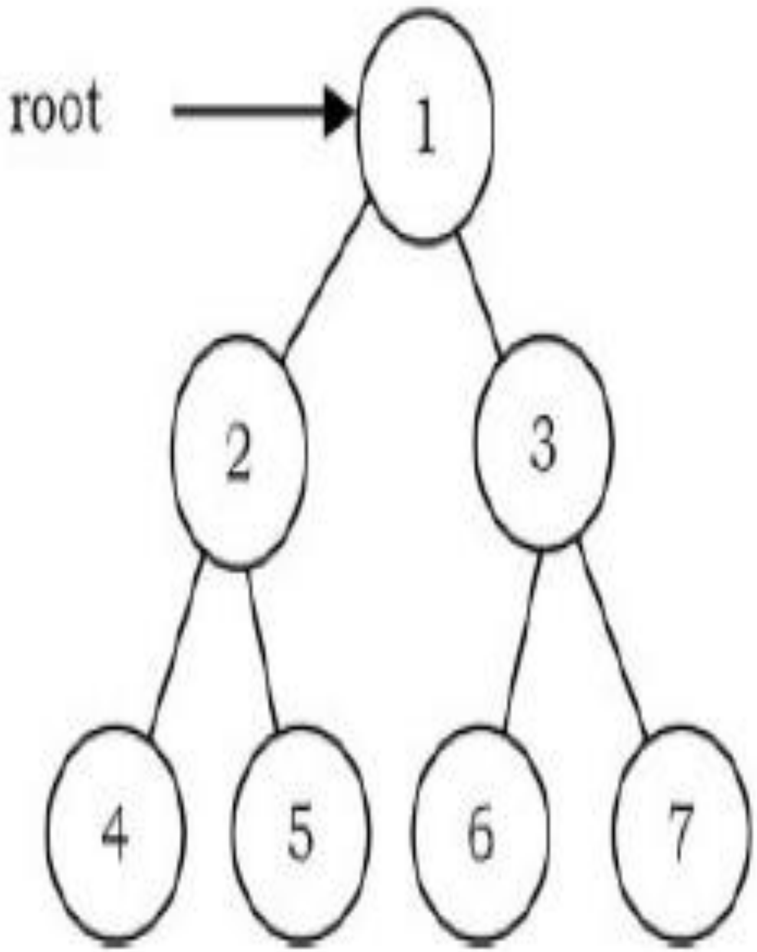
Properties of Binary Trees

- For the following properties, let us assume that the **height of the tree is *h***.
- Also, assume that **root node is at height zero**.





Height	**Number of nodes at level *h***
$h = 0$	$2^0 = 1$
$h = 1$	$2^1 = 2$



$h = 2$

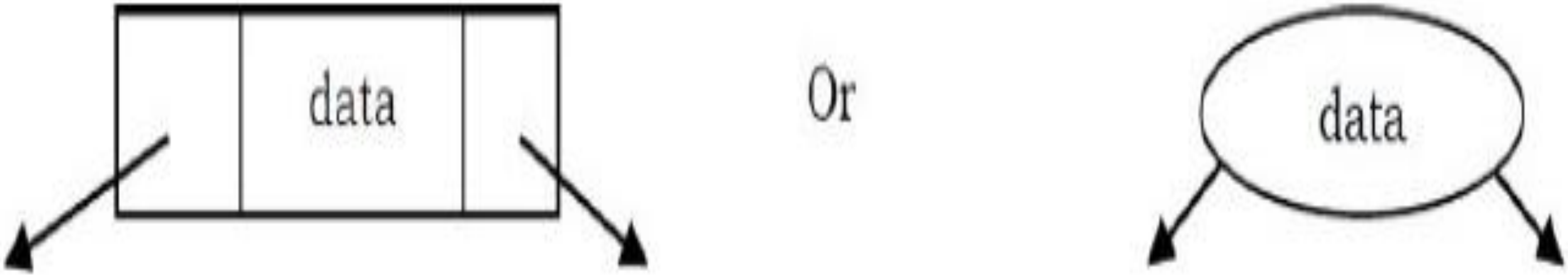
$2^2 = 4$

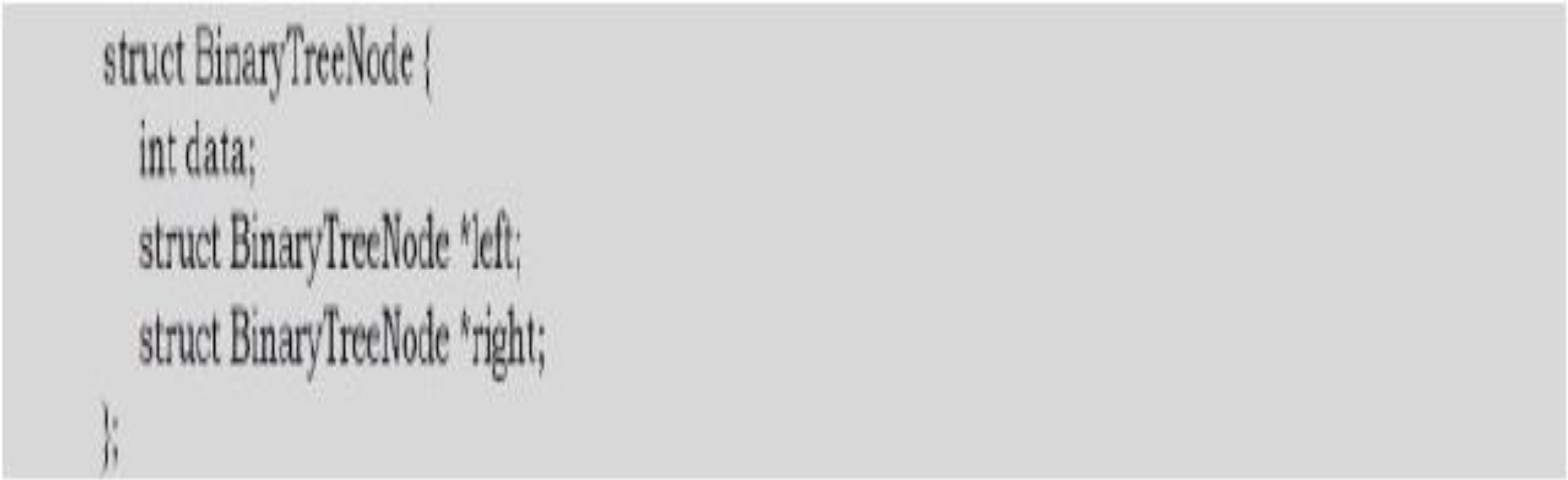
- From the diagram we can infer the following properties:
  - ❖The **number of nodes n in a full binary tree is 2^{h+1} – 1**.
    - ❖Since, there are ***h* levels** we need to add all nodes at each level [$2^0 + 2^1 + 2^2 + \cdots + 2^h$ = **2^{h+1} – 1**].
  - ❖The **number of nodes *n*** in a complete binary tree is between **2^h** (minimum) and **2^{h+1} – 1 (maximum)**.
  - ❖The **number of leaf nodes** in a full binary tree is **2*h***.
  - ❖The **number of NULL links (wasted pointers)** in a complete binary tree of **n nodes is *n* + 1**.

Structure of Binary Trees

- Assume that the data of the nodes are integers.
  - One way to represent a node (which contains data) is to have two links which point to left and right children along with data fields as shown below:

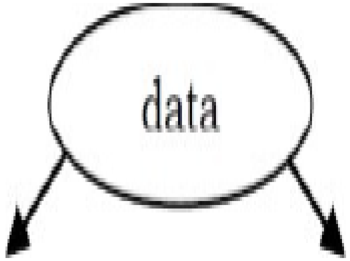
Diagrammatic representation of Binary Tree

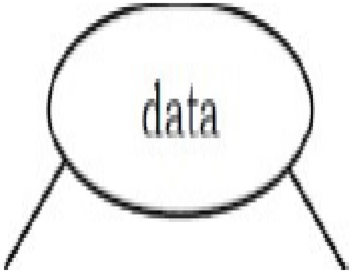




```
struct BinaryTreeNode {
   int data;
   struct BinaryTreeNode *left;
   struct BinaryTreeNode *right;
};
```

- In trees, the **default flow** is from **parent to children** and it is **not mandatory to show directed branches.**
- Both the representations shown below are the same.





Operations on Binary Trees

- **Basic Operations**
  - **Inserting** an element into a tree
  - **Deleting** an element from a tree
  - **Searching** for an element
  - **Traversing** the tree

- **Auxiliary Operations**
  - Finding the **size** of the tree
  - Finding the **height** of the tree
  - Finding the **level which has maximum sum**
  - Finding the **least common ancestor (LCA) for a given pair of nodes.**

Applications of Binary Trees

- Following are the some of the applications where *binary trees* play an important role:
  - **Expression trees** are **used in compilers**.
  - **Huffman coding trees** that are used in data **compression algorithms**.
  - **Binary Search Tree (BST)**, which supports search, insertion and deletion on a collection of items in O(*logn*) (average).
  - **Priority Queue (PQ)**, which supports search and deletion of minimum (or maximum) on a collection of items in logarithmic time (in worst case).

Binary Tree Traversals

- The process of **visiting all nodes of a tree is called *tree traversal***.
  - **Each node is processed only once** but **it may be visited more than once.**
- In tree structures there are many different ways.
- Tree traversal is like searching the tree, except that in traversal the goal is **to move through the tree in a particular order.**
- All nodes are processed in the *traversal but* ***searching* stops when the required node is found**.

- **Traversal Possibilities:**
- Starting at the root of a binary tree, there are three main steps that can be performed and the order in which they are performed defines the traversal type.
  - These steps are: performing an **action on the current node** (referred to as “**visiting” the node** and **denoted with *“D”***)
  - **Traversing to the left child node** (**denoted** with **“*L*”**), and
  - **Traversing to the right child node** (**denoted** with **“*R*”**).
- This process can be easily described through recursion.

- Based on the above definition there are 6 possibilities:
  - 1. ***LDR:*** Process **left subtree**, process the **current node data** and then process **right subtree**
  - 2. ***LRD:*** Process left subtree, process right subtree and then process the current node data

- 3. ***DLR:*** Process the current node data, process left subtree and then process right subtree
- 4. ***DRL:*** Process the current node data, process right subtree and then process left subtree

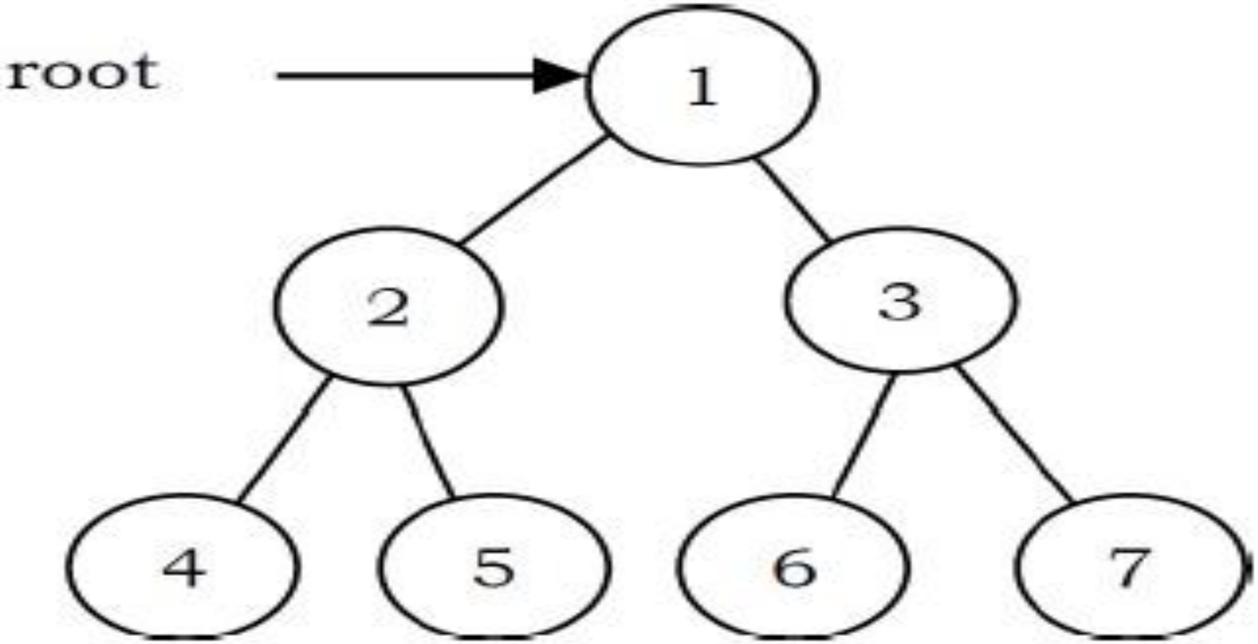
- 5. ***RDL:*** Process right subtree, process the current node data and then process left subtree
- 6. ***RLD:*** Process right subtree, process left subtree and then process the current node data

- **Classifying the Traversals**
- The sequence in which these entities (nodes) are processed defines a particular traversal method.
- The **classification is based** on the **order in which current node is processed**.
- **If we are classifying based on current node (D) and if D comes in the middle then it does not matter whether L is on left side of D or R is on left side of D.**

- It does not matter whether L is on right side of D or R is on right side of D.
- Hence,the total 6 possibilities are reduced to 3 and these are:
  - **Preorder (DLR) Traversal**
  - **Inorder (LDR) Traversal**
  - **Postorder (LRD) Traversal**

- There is another traversal method which does not depend on the above orders and it is:
  - **Level Order Traversal**: This method is inspired from Breadth First Traversal (BFS of Graph algorithms).

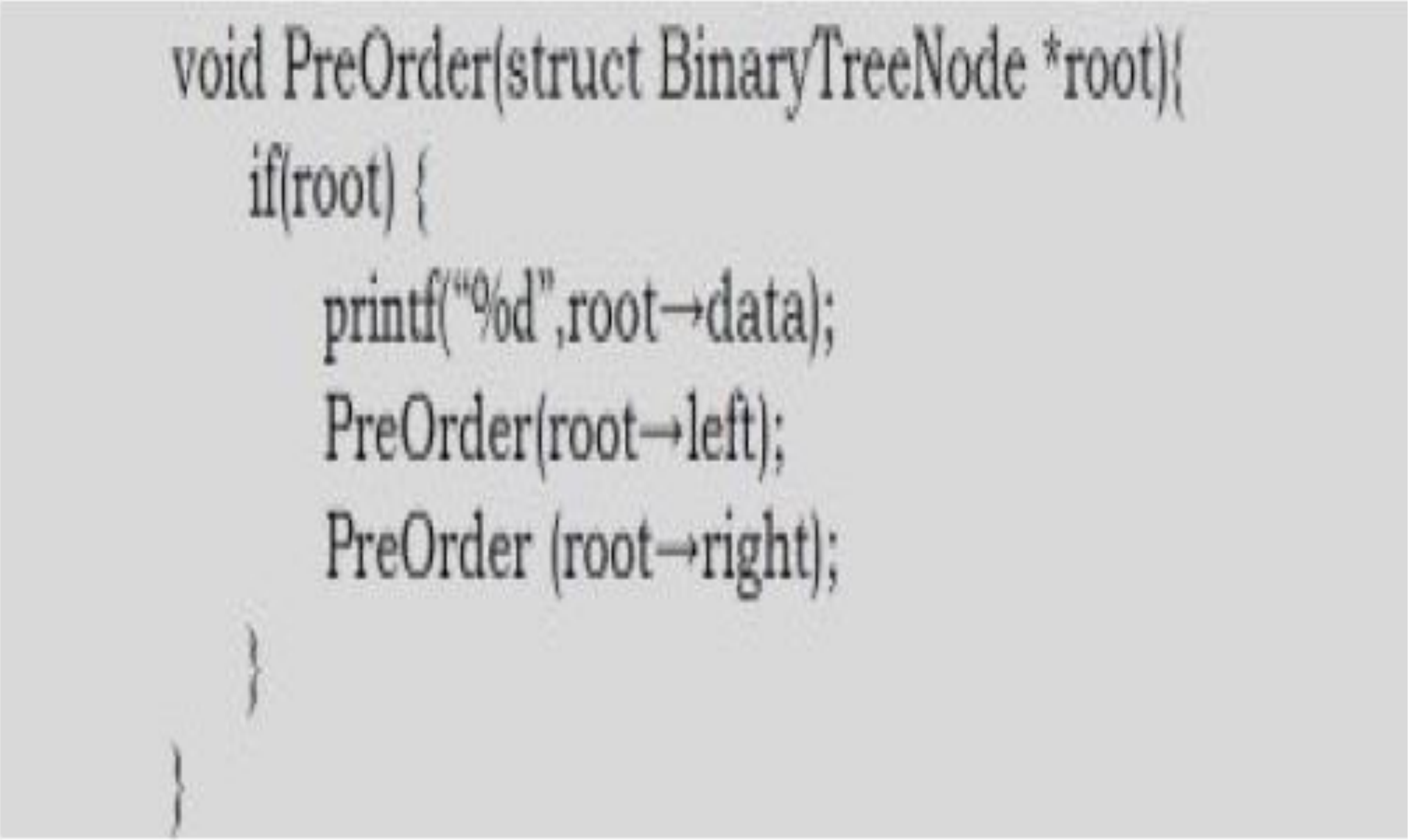
- Let us use the diagram below for the remaining discussion.



- **PreOrder Traversal**
- In preorder traversal, **each node is processed before (pre) either of its subtrees**. Even though each node is processed before the subtrees, it still requires that some information must be maintained while moving down the tree.
  - **In the example above, 1 is processed first, then the left subtree, and this is followed by the right subtree.**

- Therefore, **processing must return to the right subtree after finishing the processing of the left subtree.** So, the root information must be maintained.
  - **The obvious ADT for such information is a stack.**
  - Because of its LIFO structure, the information about the right subtrees back in the reverse order can be retrieved.

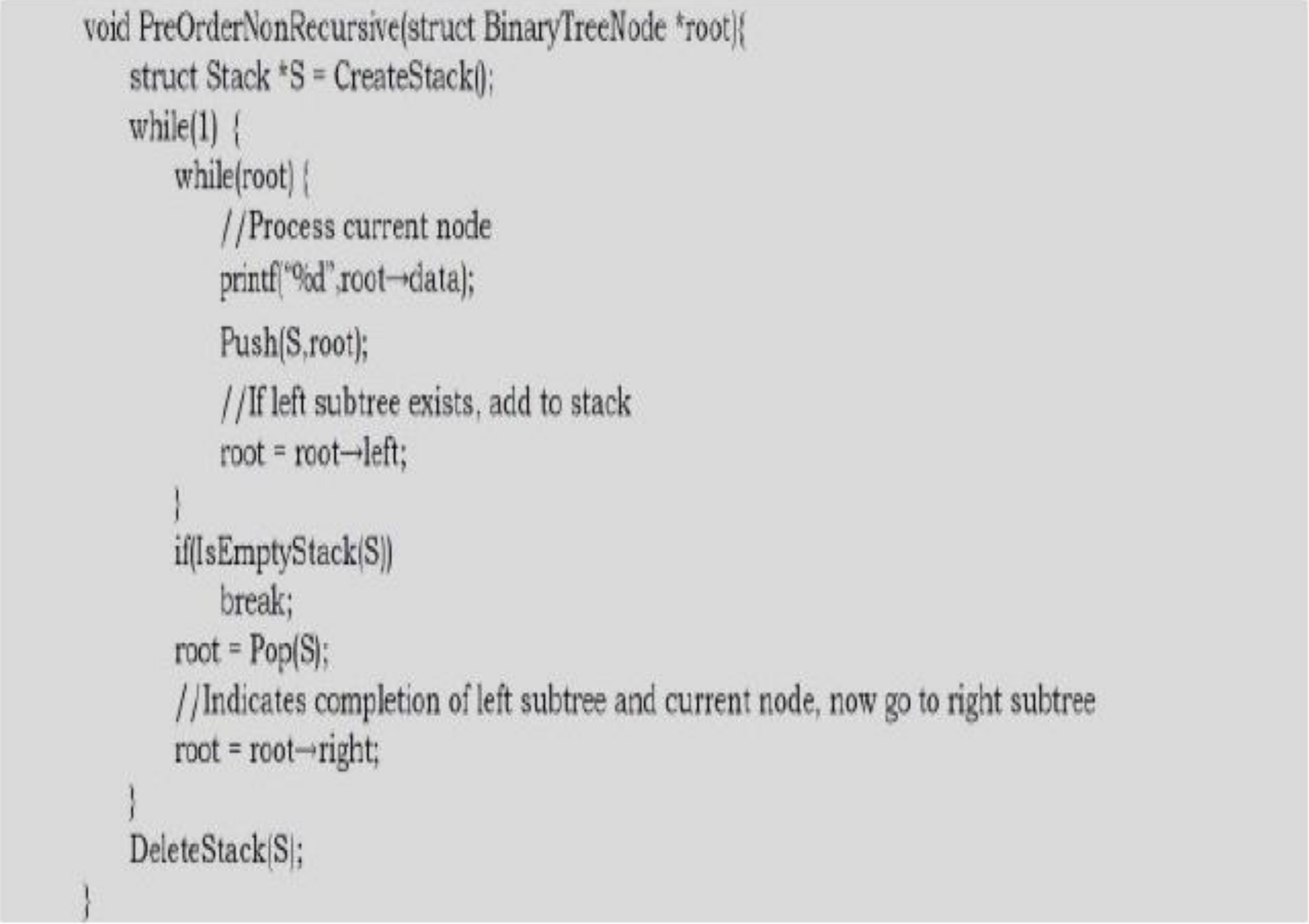
- Preorder traversal is defined as follows:
  - **Visit the root.**
  - **Traverse the left subtree in Preorder.**
  - **Traverse the right subtree in Preorder.**
- The nodes of tree would be visited in the order: 1 2 4 5 3 6 7



```
void PreOrder(struct BinaryTreeNode *root){
    if(root) {
        printf("%d",root→data);
        PreOrder(root→left);
        PreOrder (root→right);
    }
}
```

Time Complexity: O(*n*). Space Complexity: O(*n*).

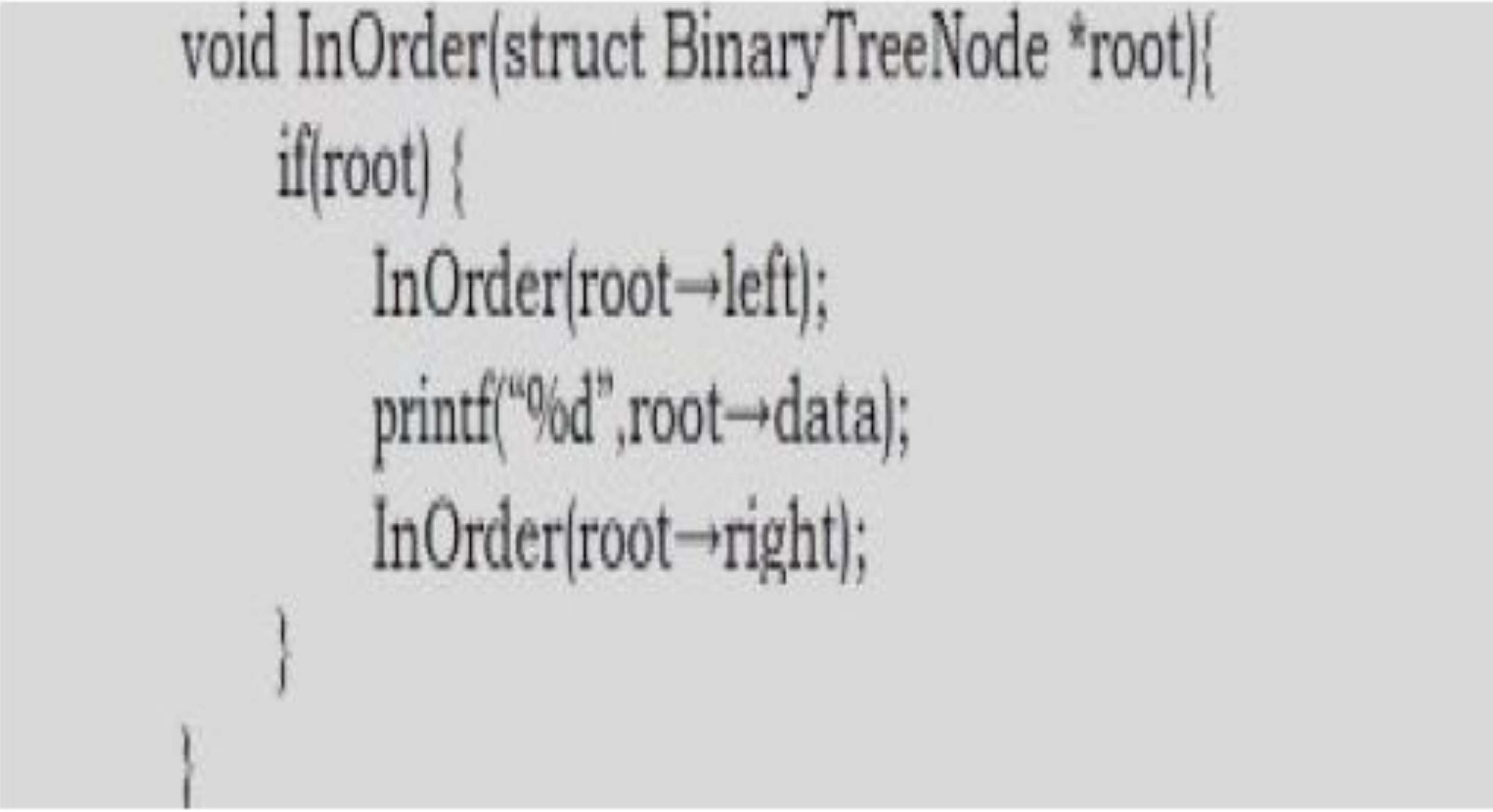
- **Non-Recursive Preorder Traversal**
- In the recursive version, a stack is required as we need to remember the current node so that after completing the left subtree we can go to the right subtree.
- To simulate the same, first we process the current node and before going to the left subtree, we store the current node on stack.
- After completing the left subtree processing, *pop* the element and go to its right subtree. Continue this process until stack is nonempty.



```
void PreOrderNonRecursive(struct BinaryTreeNode *root){
    struct Stack *S = CreateStack();
    while(1) {
        while(root) {
            //Process current node
            printf("%d",root→data);
            Push(S,root);
            //If left subtree exists, add to stack
            root = root→left;
        }
        if(IsEmptyStack(S))
            break;
        root = Pop(S);
        //Indicates completion of left subtree and current node, now go to right subtree
        root = root→right;
    }
    DeleteStack(S);
}
```

Time Complexity: O(*n*). Space Complexity: O(*n*).

- **InOrder Traversal**
- In Inorder Traversal the root is visited between the subtrees. Inorder traversal is defined as follows:
  - **Traverse the left subtree in Inorder.**
  - **Visit the root.**
  - **Traverse the right subtree in Inorder.**
- The nodes of tree would be visited in the order: 4 2 5 1 6 3 7



```
void InOrder(struct BinaryTreeNode *root){
    if(root) {
        InOrder(root→left);
        printf("%d",root→data);
        InOrder(root→right);
    }
}
```

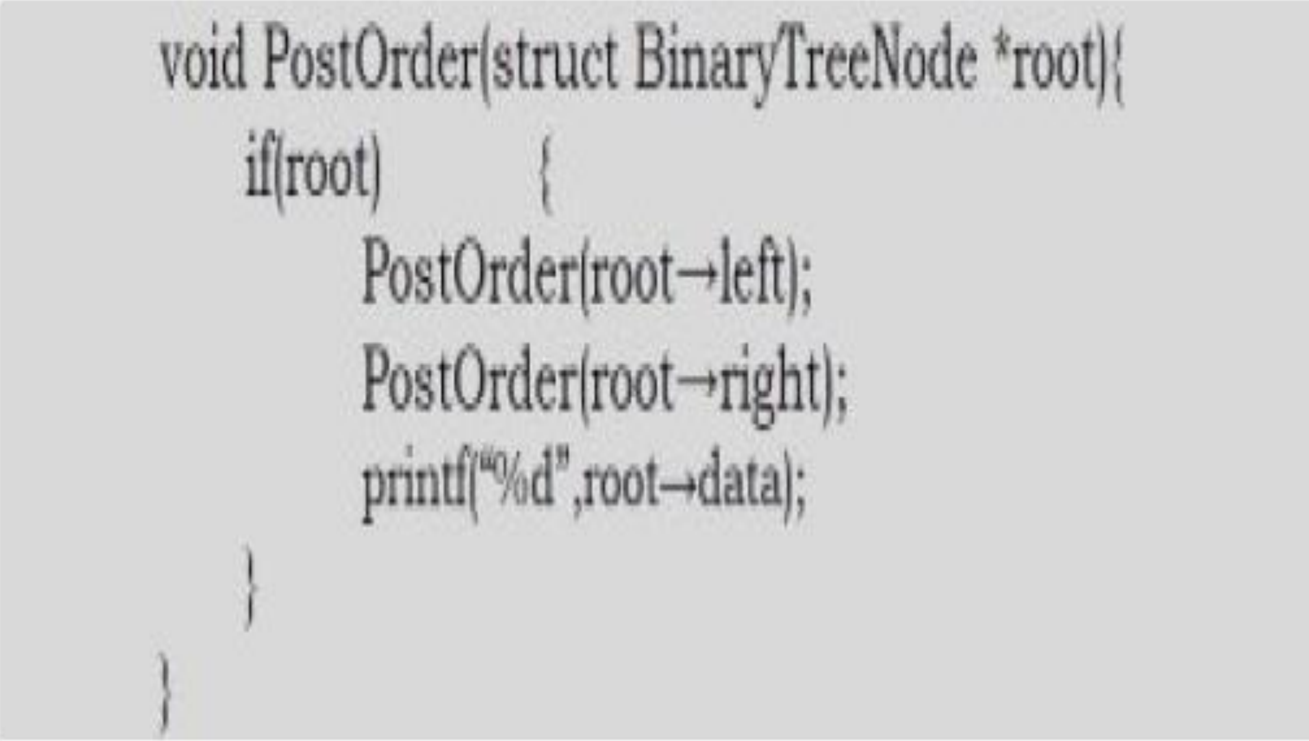
Time Complexity: O(n). Space Complexity: O(n).

- **Non-Recursive Inorder Traversal**
- The Non-recursive version of Inorder traversal is similar to Preorder. The only change is, instead of processing the node before going to left subtree, process it after popping (which is indicated after completion of left subtree processing).

```
void InOrderNonRecursive(struct BinaryTreeNode *root){
    struct Stack *S = CreateStack();
    while(1) {
        while(root) {
            Push(S,root);
            //Got left subtree and keep on adding to stack
            root = root→left;
        }
        if(IsEmptyStack(S))
            break;
        root = Pop(S);
        printf(“%d”, root→data);   //After popping, process the current node
        //Indicates completion of left subtree and current node, now go to right subtree
        root = root→right;
    }
    DeleteStack(S);
}
```

Time Complexity: O(*n*). Space Complexity: O(*n*).

- **PostOrder Traversal**
- In postorder traversal, the root is visited after both subtrees. Postorder traversal is defined as follows:
  - **Traverse the left subtree in Postorder.**
  - **Traverse the right subtree in Postorder.**
  - **Visit the root.**
- The nodes of the tree would be visited in the order: 4 5 2 6 7 3 1



```
void PostOrder(struct BinaryTreeNode *root){
    if(root)        {
        PostOrder(root→left);
        PostOrder(root→right);
        printf(“%d”,root→data);
    }
}
```

Time Complexity: O(*n*). Space Complexity: O(*n*).

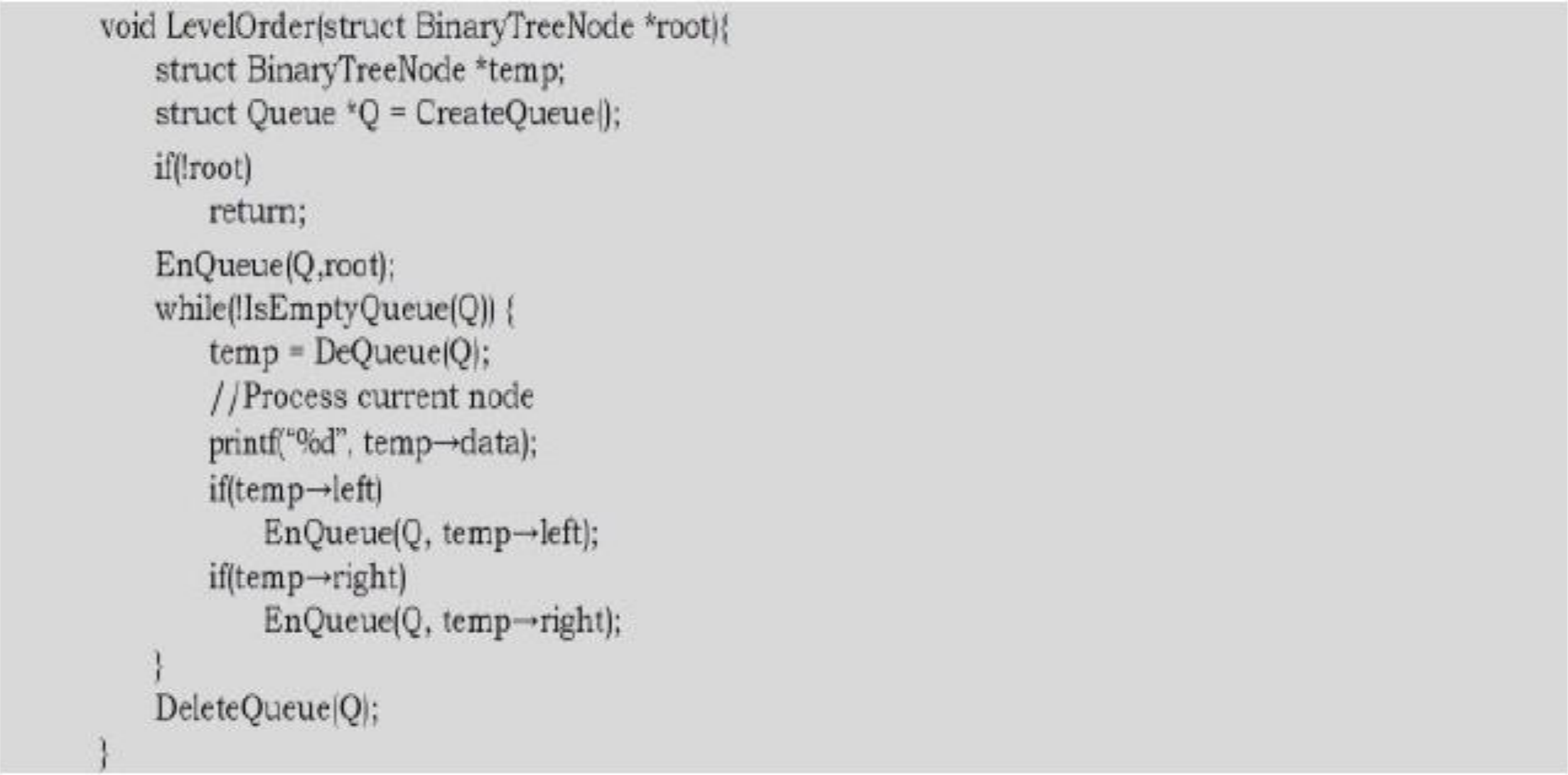
- **Non-Recursive Postorder Traversal**
- In preorder and inorder traversals, after popping the stack element we do not need to visit the same vertex again. But in postorder traversal, each node is visited twice. That means, after processing the left subtree we will visit the current node and after processing the right subtree we will visit the same current node. But we should be processing the node during the second visit.
- Here the problem is how to differentiate whether we are returning from the left subtree or the right subtree.

- We use a *previous* variable to keep track of the earlier traversed node. Let’s assume *current* is the current node that is on top of the stack. When *previous* is *current’s* parent, we are traversing down the tree. In this case, we try to traverse to *current’s* left child if available (i.e., push left child to the stack). If it is not available, we look at *current’s* right child. If both left and right child do not exist (ie, *current* is a leaf node), we print *current’s* value and pop it off the stack.
- If prev is *current’s* left child, we are traversing up the tree from the left. We look at *current’s* right child. If it is available, then traverse down the right child (i.e., push right child to the stack); otherwise print *current’s* value and pop it off the stack. If *previous* is *current’s* right child, we are traversing up the tree from the right. In this case, we print *current’s* value and pop it off the stack.

```
void PostOrderNonRecursive(struct BinaryTreeNode *root) {
   struct SimpleArrayStack *S = CreateStack();
   struct BinaryTreeNode *previous = NULL;
   do{
      while (root!=NULL){
         Push(S, root);
         root = root->left;
      }
      while(root == NULL && !IsEmptyStack(S)){
         root = Top(S);
         if(root->right == NULL || root->right == previous){
            printf("%d ", root->data);
            Pop(S);
            previous = root;
            root = NULL;
         }
         else
            root = root->right;
      }
   }while(!IsEmptyStack(S));
}
```

Time Complexity: O(*n*). Space Complexity: O(*n*).

- **Level Order Traversal**
- Level order traversal is defined as follows:
  - Visit the root.
  - While traversing level (, keep all the elements at level (+ 1 in queue.
  - Go to the next level and visit all the nodes at that level.
  - Repeat this until all levels are completed.
- The nodes of the tree are visited in the order: 1 2 3 4 5 6 7



```
void LevelOrder(struct BinaryTreeNode *root){
    struct BinaryTreeNode *temp;
    struct Queue *Q = CreateQueue();
    if(!root)
        return;
    EnQueue(Q,root);
    while(!IsEmptyQueue(Q)) {
        temp = DeQueue(Q);
        //Process current node
        printf(“%d”, temp→data);
        if(temp→left)
            EnQueue(Q, temp→left);
        if(temp→right)
            EnQueue(Q, temp→right);
    }
    DeleteQueue(Q);
}
```

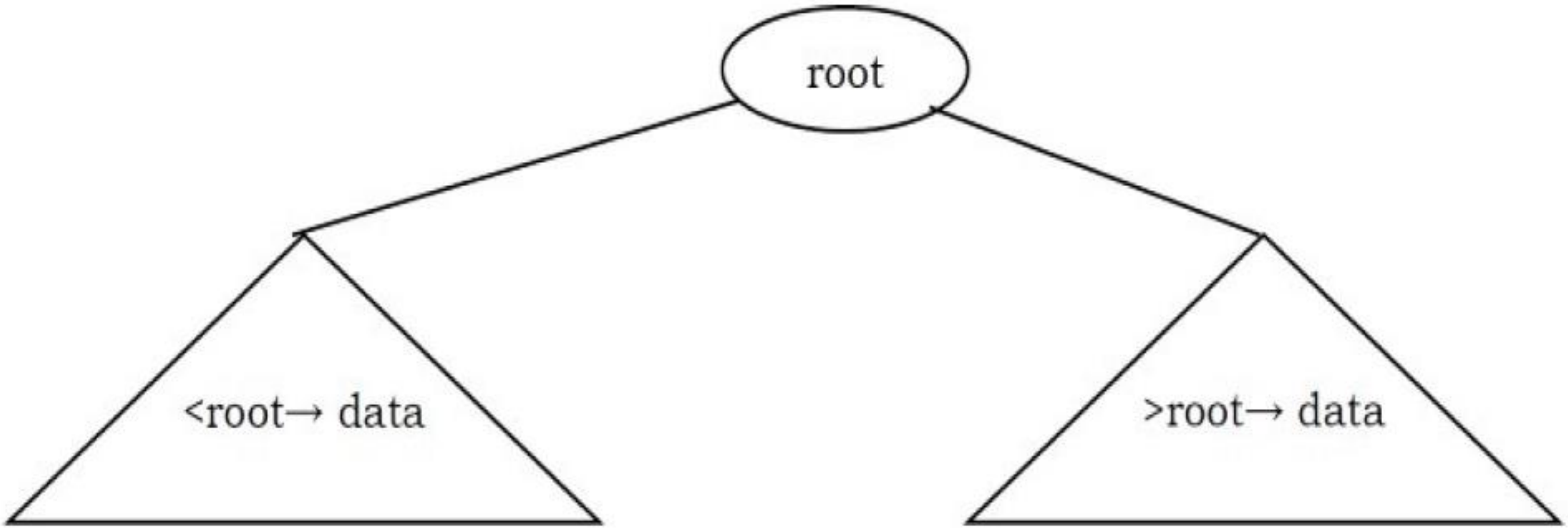
Time Complexity: O(*n*). Space Complexity: O(*n*). Since, in the worst case, all the nodes on the entire last level could be in the queue simultaneously.

Binary Search Trees (BSTs)

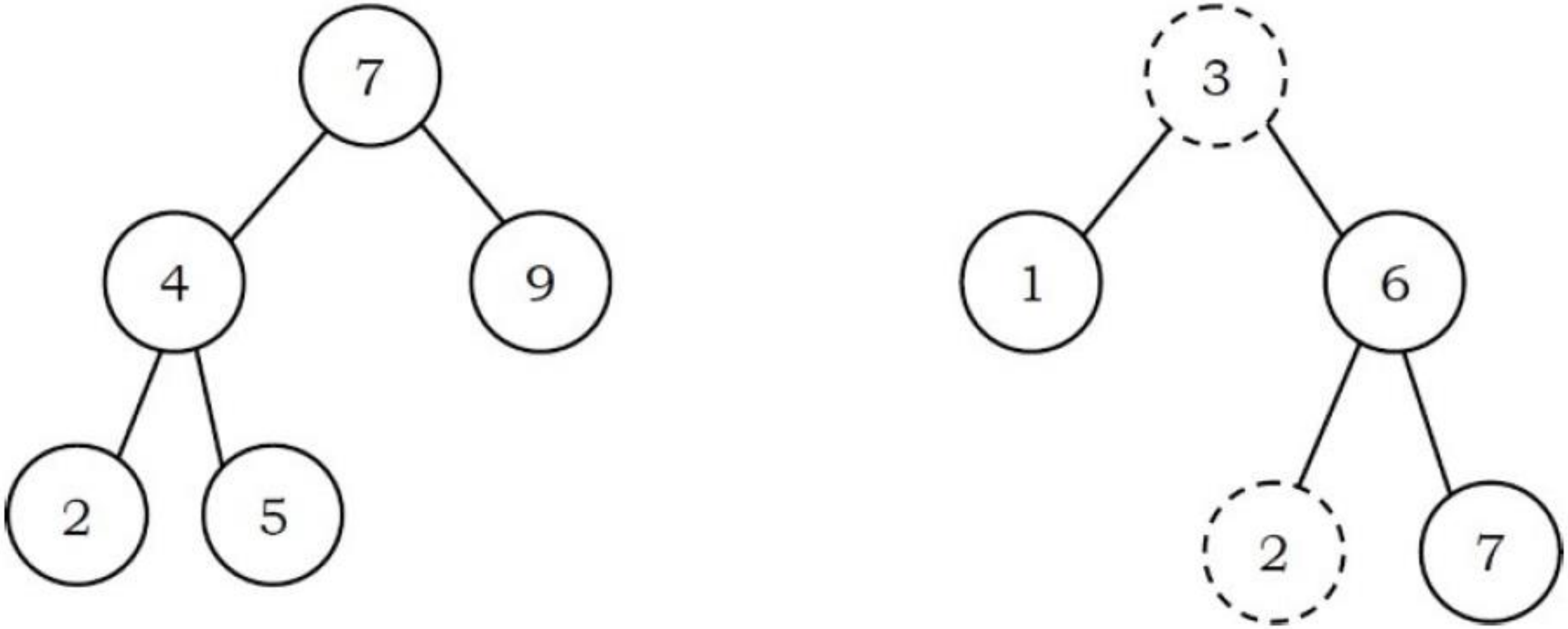
- Why Binary Search Trees?
- A variant of binary trees is Binary Search Trees (BSTs).
- As the name suggests, the main use of this representation is for searching.
- In this representation we impose restriction on the kind of data a node can contain. As a result, it reduces the worst case average search operation to O(logn).

Binary Search Tree Property

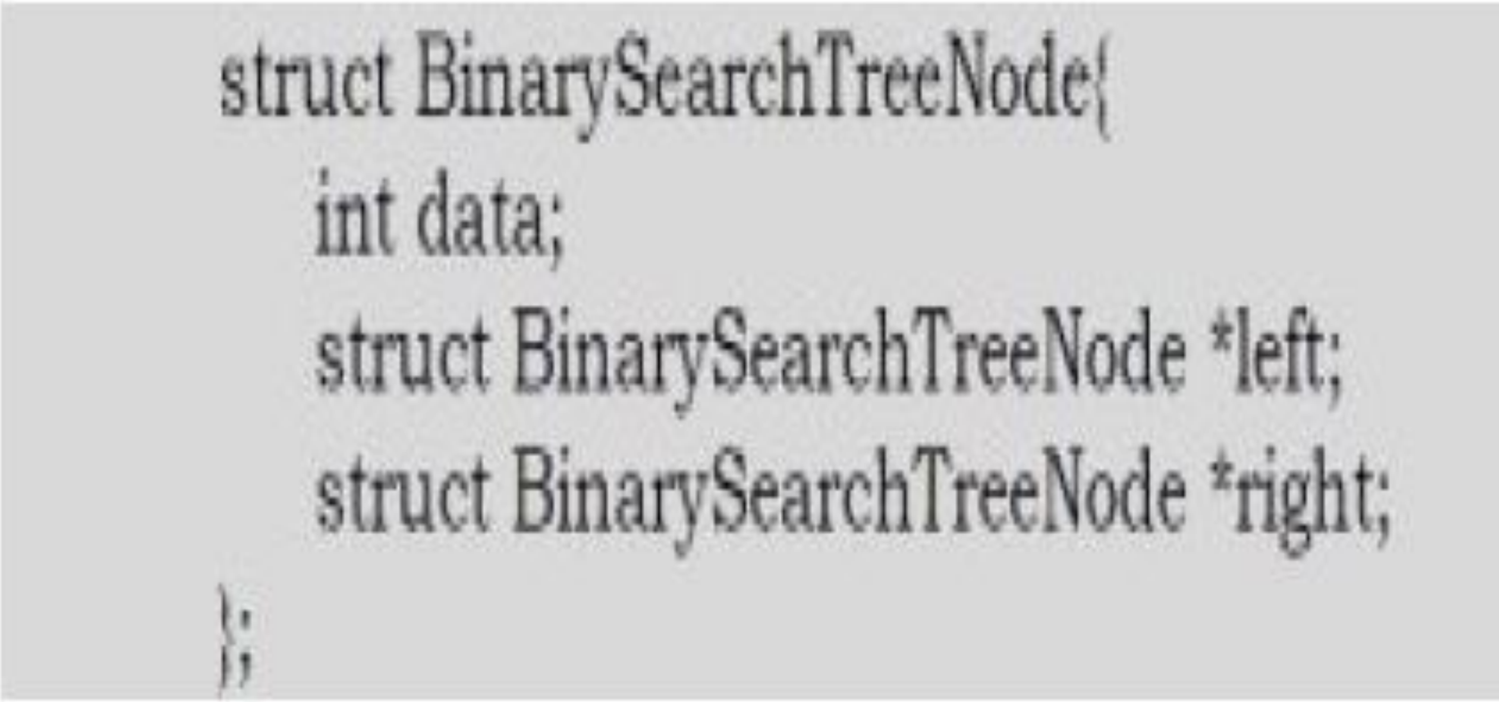
- In binary search trees, all the left subtree elements should be less than root data and all the right subtree elements should be greater than root data.
- This is called binary search tree property.
- This property should be satisfied at every node in the tree.
    - The left subtree of a node contains only nodes with keys less than the nodes key.
    - The right subtree of a node contains only nodes with keys greater than the nodes key.
    - Both the left and right subtrees must also be binary search trees.


root
<root→ data
>root→ data

- **Example:** The left tree is a binary search tree and the right tree is not a binary search tree (at node 6 it’s not satisfying the binary search tree property).


7
4
9
2
5
3
1
6
2
7

- **Binary Search Tree Declaration**



```
struct BinarySearchTreeNode{
    int data;
    struct BinarySearchTreeNode *left;
    struct BinarySearchTreeNode *right;
};
```

- **Operations on Binary Search Trees**
- **Main operations:** Following are the main operations that are supported by binary search trees:
  - Find/ Find Minimum / Find Maximum element in binary search trees
  - Inserting an element in binary search trees
  - Deleting an element from binary search trees

- **Operations on Binary Search Trees**
- **Main operations:** Following are the main operations that are supported by binary search trees:
  - Find/ Find Minimum / Find Maximum element in binary search trees
  - Inserting an element in binary search trees
  - Deleting an element from binary search trees
  - **Auxiliary operations:** Checking whether the given tree is a binary search tree or not
  - Finding *kth*-smallest element in tree
  - Sorting the elements of binary search tree and many more

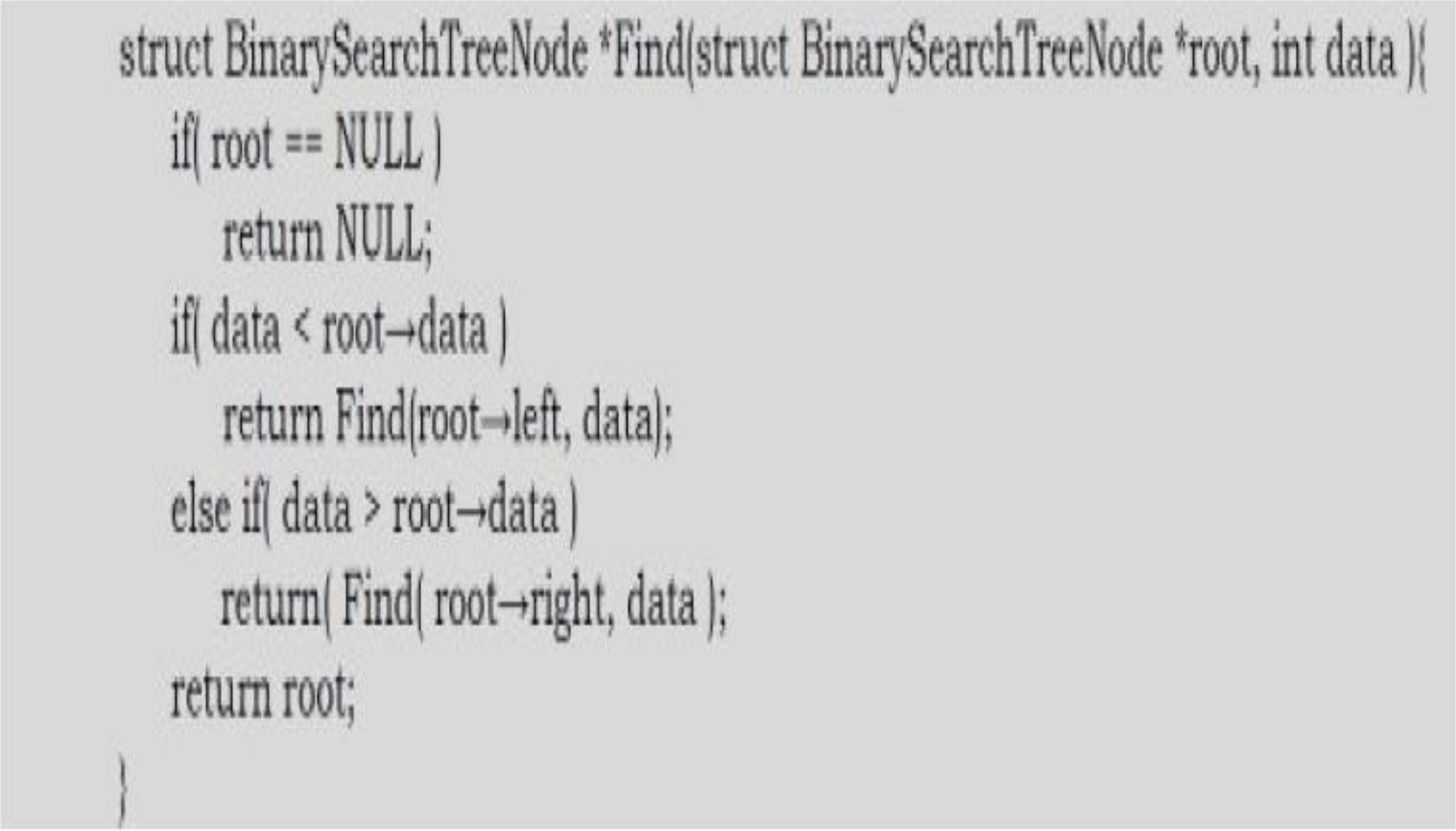
Important Notes on Binary Search Trees

- Since root data is always between left subtree data and right subtree data, performing in order traversal on binary search tree produces a sorted list.
- While solving problems on binary search trees, first we process left subtree, then root data, and finally we process right subtree. This means, depending on the problem, only the intermediate step (processing root data) changes and we do not touch the first and third steps.
- If we are searching for an element and if the left subtree root data is less than the element we want to search, then skip it. The same is the case with the right subtree.. Because of this, binary search trees take less time for searching an element than regular binary trees. In other words, the binary search trees consider either left or right subtrees for searching an element but not both.

- The basic operations that can be performed on binary search tree (BST) are insertion of element, deletion of element, and searching for an element. While performing these operations on BST the height of the tree gets changed each time. Hence there exists variations in time complexities of best case, average case, and worst case.
- The basic operations on a binary search tree take time proportional to the height of the tree. For a complete binary tree with node n, such operations runs in O(lgn) worst-case time. If the tree is a linear chain of n nodes (skew-tree), however, the same operations takes O(*n*) worst-case time.

Finding an Element in Binary Search Trees

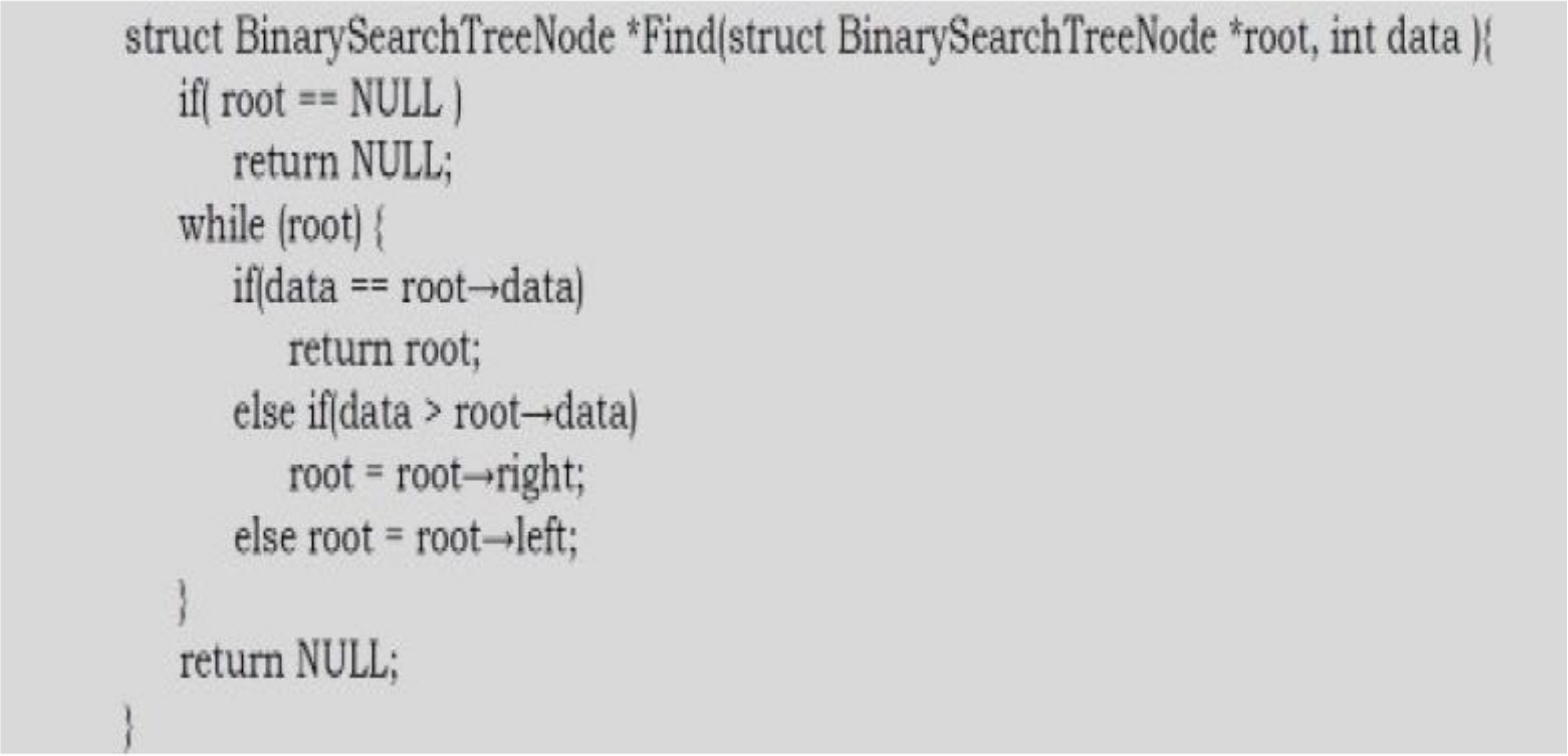
- Start with the root and keep moving left or right using the BST property.
- If the data we are searching is same as nodes data then we return current node.
  - ❖If the data we are searching is **less than nodes data** then **search left subtree** of current node;
  - ❖**otherwise search right subtree** of current node.
  - ❖If the **data is not present**, we end up in a **NULL link**.



```
struct BinarySearchTreeNode *Find(struct BinarySearchTreeNode *root, int data ){
    if( root == NULL )
        return NULL;
    if( data < root→data )
        return Find(root→left, data);
    else if( data > root→data )
        return( Find( root→right, data );
    return root;
}
```

Time Complexity: O(*n*), in worst case (when BST is a skew tree). Space Complexity: O(*n*), for recursive stack.

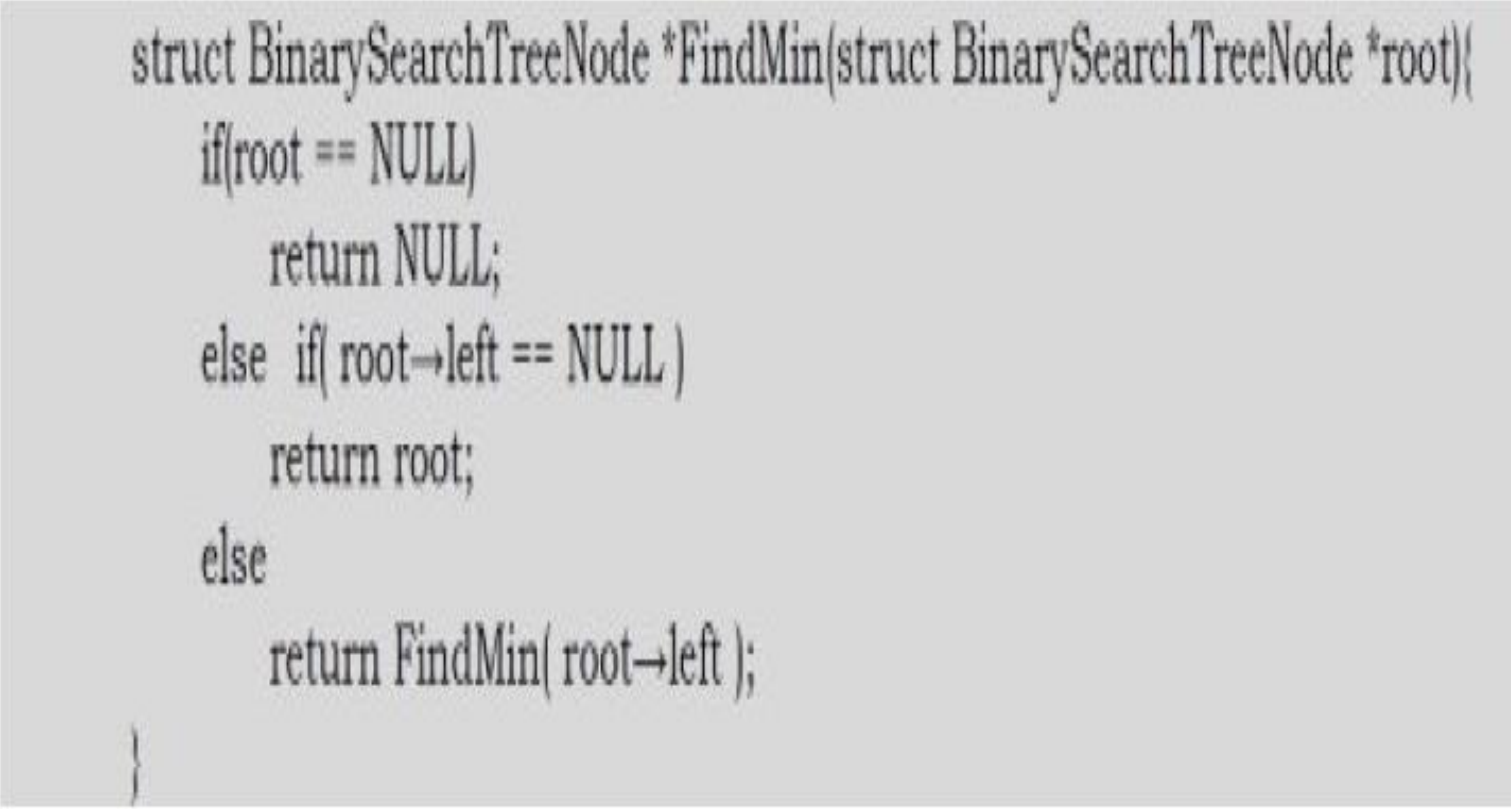
- *Non recursive* version of the above algorithm can be given as:



```
struct BinarySearchTreeNode *Find(struct BinarySearchTreeNode *root, int data ){
    if( root == NULL )
        return NULL;
    while (root) {
        if(data == root→data)
            return root;
        else if(data > root→data)
            root = root→right;
        else root = root→left;
    }
    return NULL;
}
```

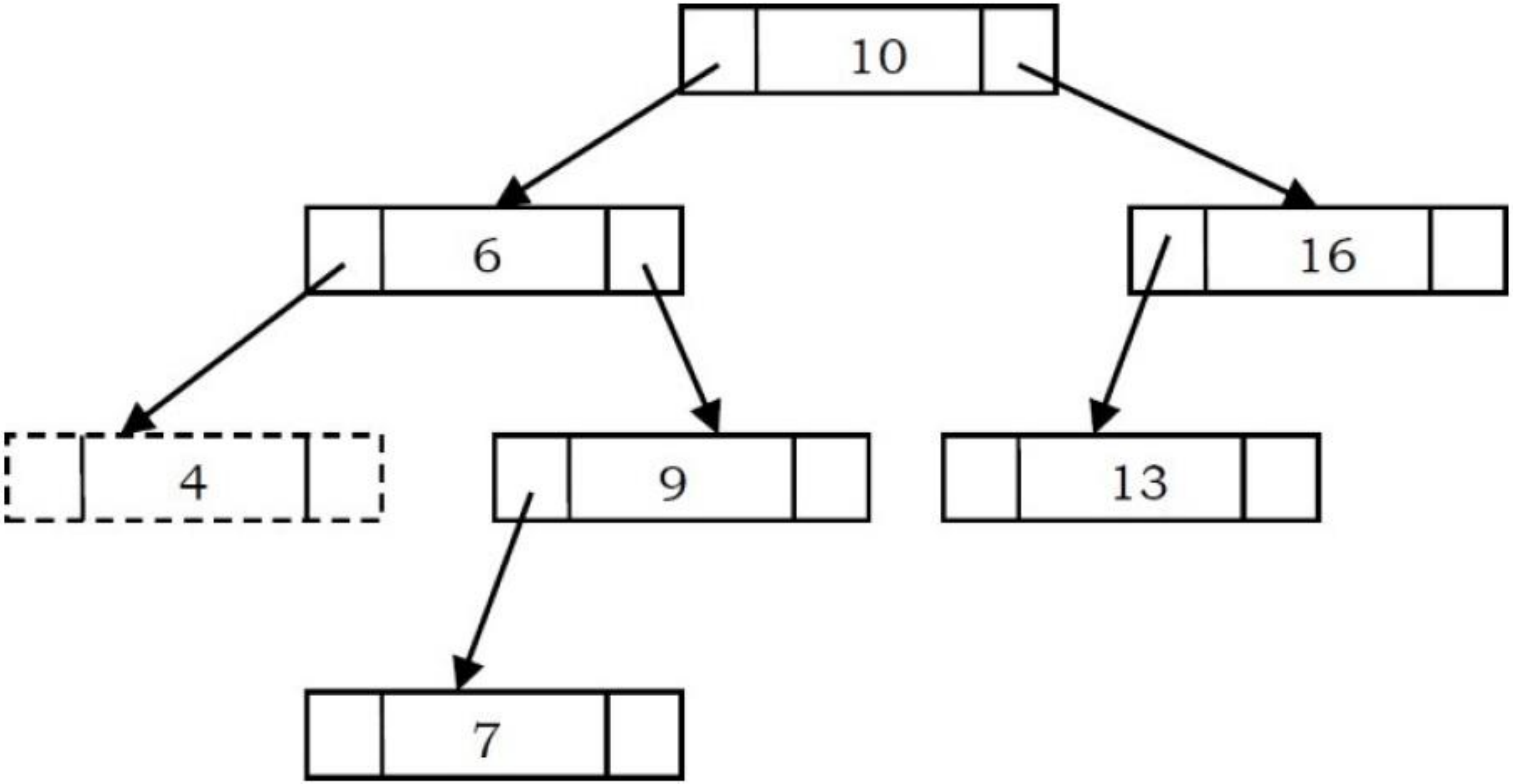
Time Complexity: O(n). Space Complexity: O(1).

- **Finding Minimum Element in Binary Search Trees**
- In BSTs, the minimum element is the left-most node, which does not has left child. In the BST below, the minimum element is **4**.



```
struct BinarySearchTreeNode *FindMin(struct BinarySearchTreeNode *root){
    if(root == NULL)
        return NULL;
    else  if( root→left == NULL )
        return root;
    else
        return FindMin( root→left );
}
```

Time Complexity: O(*n*), in worst case (when BST is a *left skew* tree).
Space Complexity: O(*n*), for recursive stack.


10
6
16
4
9
13
7

Non recursive version of the above algorithm can be given as:

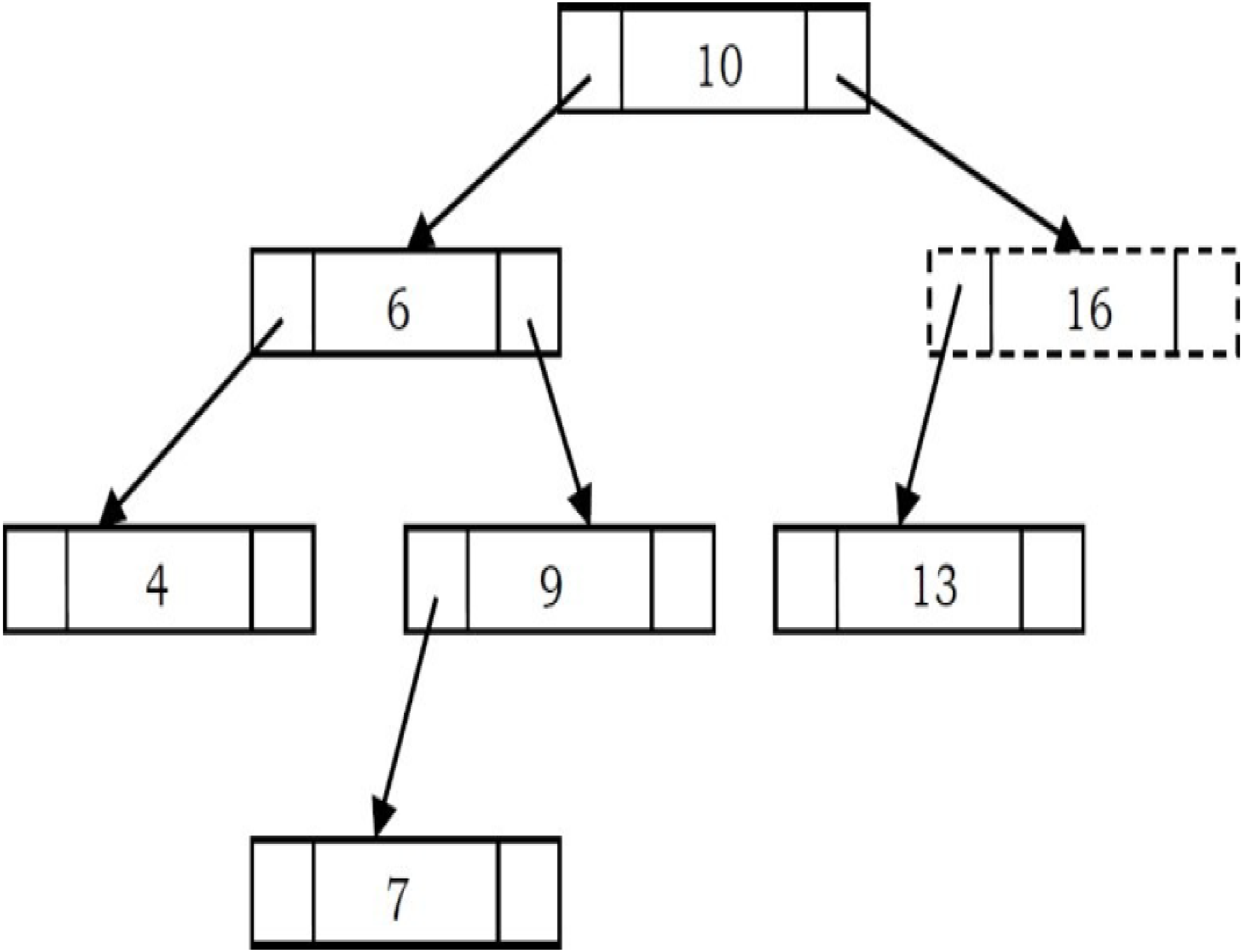
```
struct BinarySearchTreeNode *FindMin(struct BinarySearchTreeNode * root ) {
    if( root == NULL )
        return NULL;
    while( root→left != NULL )
        root = root→left;
    return root;
}
```

Time Complexity: O(*n*). Space Complexity: O(1).

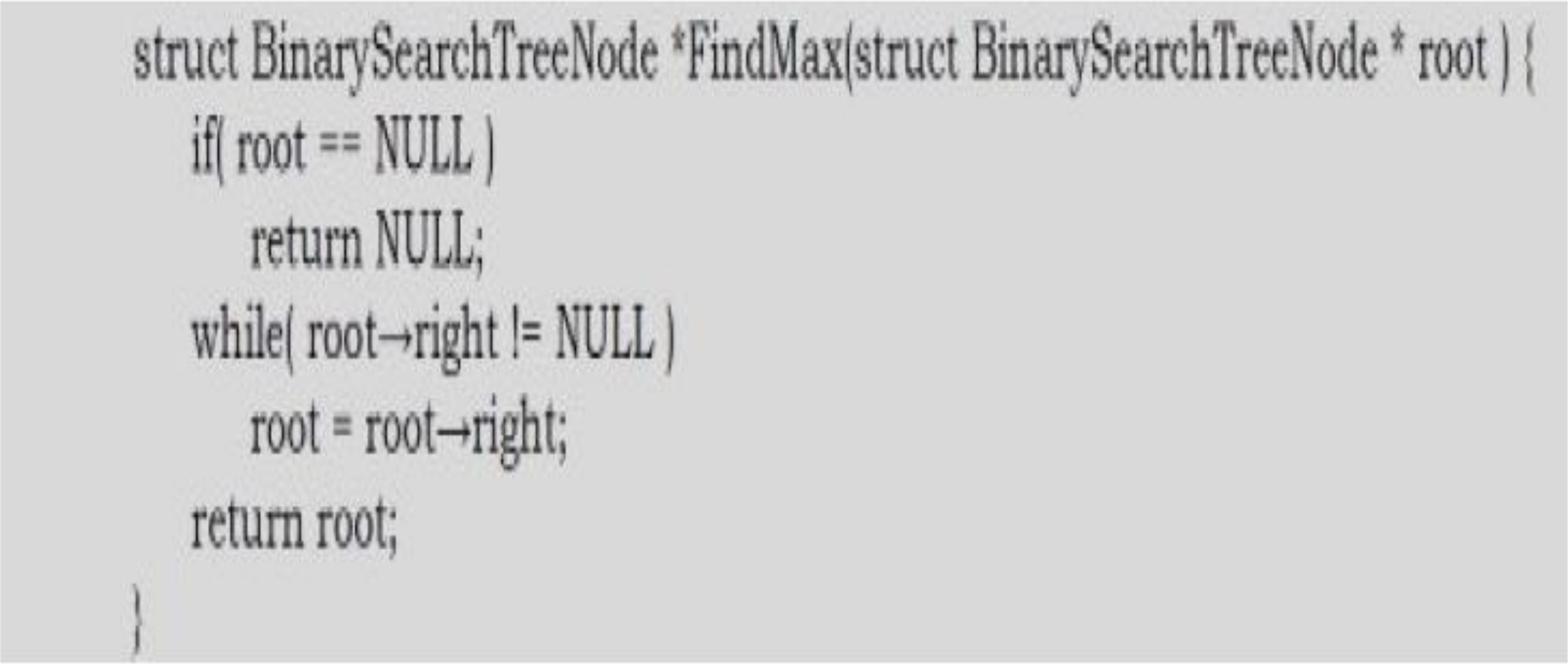
- **Finding Maximum Element in Binary Search Trees**
- In BSTs, the maximum element is the right-most node, which does not have right child. In the BST below, the maximum element is **16**.

```
struct BinarySearchTreeNode *FindMax(struct BinarySearchTreeNode *root) {
    if(root == NULL)
        return NULL;
    else if( root→right == NULL )
        return root;
    else return FindMax( root→right );
}
```

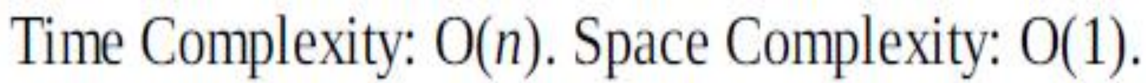
Time Complexity: O(n), in worst case (when BST is a *right skew* tree).
Space Complexity: O(n), for recursive stack.


10
6
16
4
9
13
7

- *Non recursive* version of the above algorithm can be given as:



```
struct BinarySearchTreeNode *FindMax(struct BinarySearchTreeNode * root ) {
    if( root == NULL )
        return NULL;
    while( root→right != NULL )
        root = root→right;
    return root;
}
```



Time Complexity: O(n). Space Complexity: O(1).