COM 201 – Data Structures and Algorithms

Abstract Data Types – Trees

Assist. Prof. Özge ÖZTİMUR KARADAĞ

Department of Computer Engineering – ALKÜ

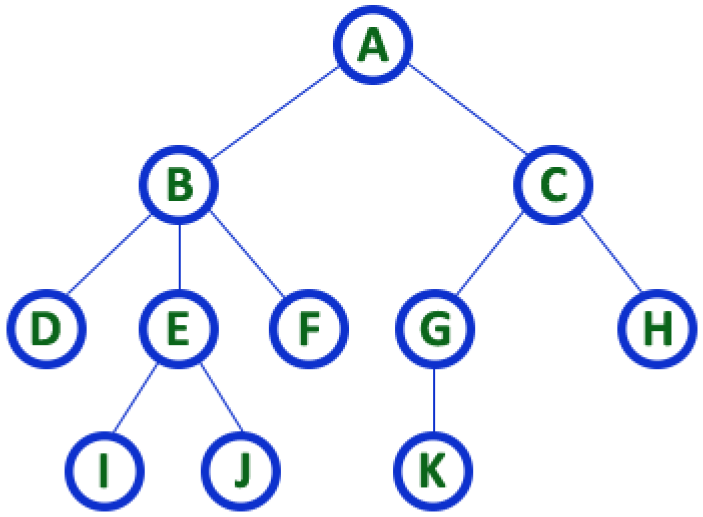
Alanya

Previously

- Linear Data Structures & Abstract Data Types
  - Array
  - Linked List
  - Stack
  - Queue

Today

- Nonlinear Data Structure
  - Tree



Outline

- Preliminaries
  - What is Tree?
  - Implementation of Trees using C++
  - Tree traversals and applications
- Binary Trees
- Binary Search Trees
  - Structure and operations
  - Analysis

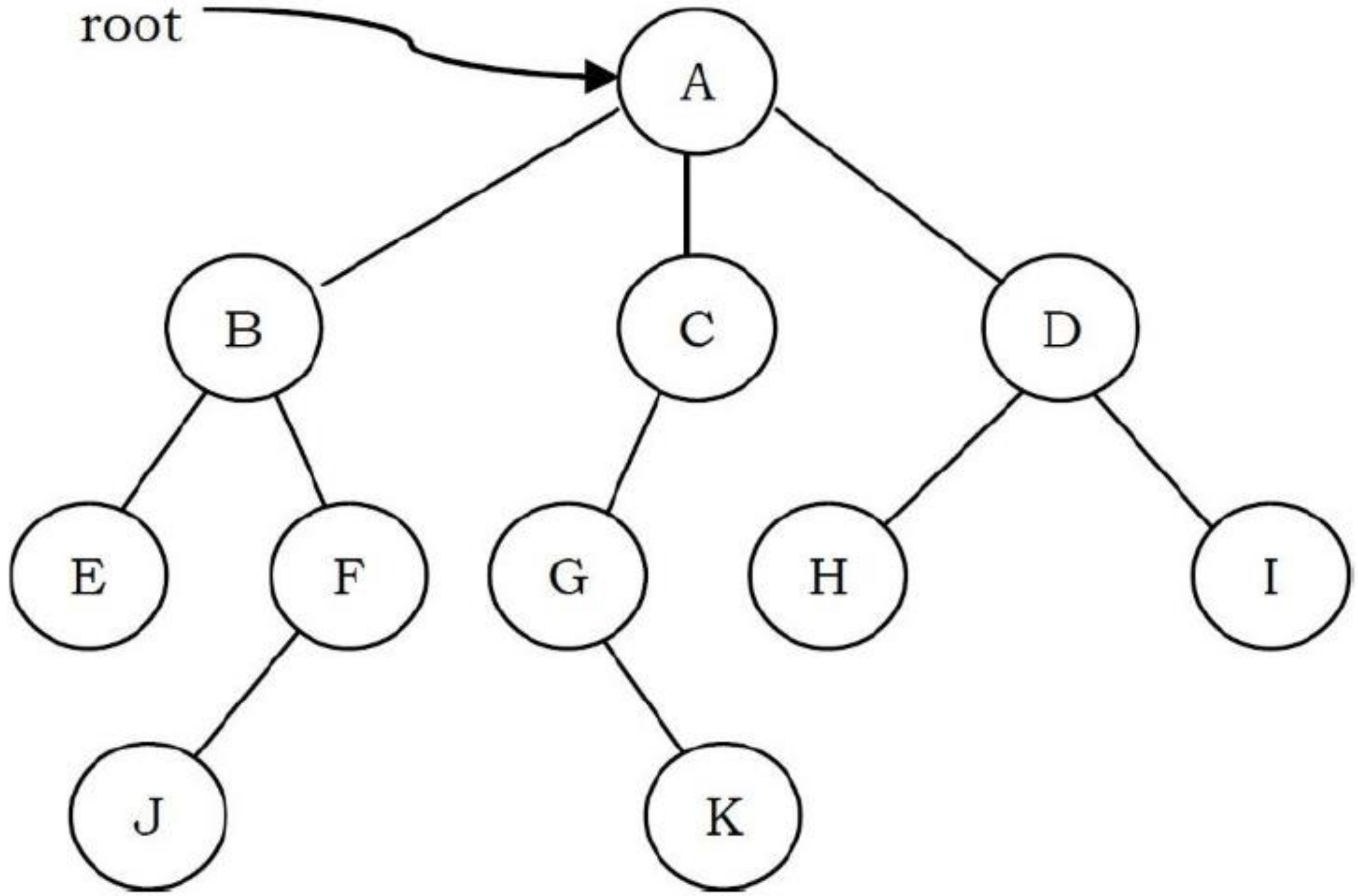
Tree

Tree

- A *tree* structure is a way of representing the hierarchical nature of a structure in a graphical form.

- 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.

Terminology



Terminology (Continued)

- The *root* of a tree is the node with no parents. There can be at most one root node in a tree (*node A* in the above example).
- An *edge* refers to the link from parent to child (all links in the figure).
- A node with no children is called *leaf* node (*E,J,K,H* and *I*).
- Children of same parent are called *siblings* (*B,C,D* are siblings of *A*, and *E,F* are the siblings of *B*).
- 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.
- 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.

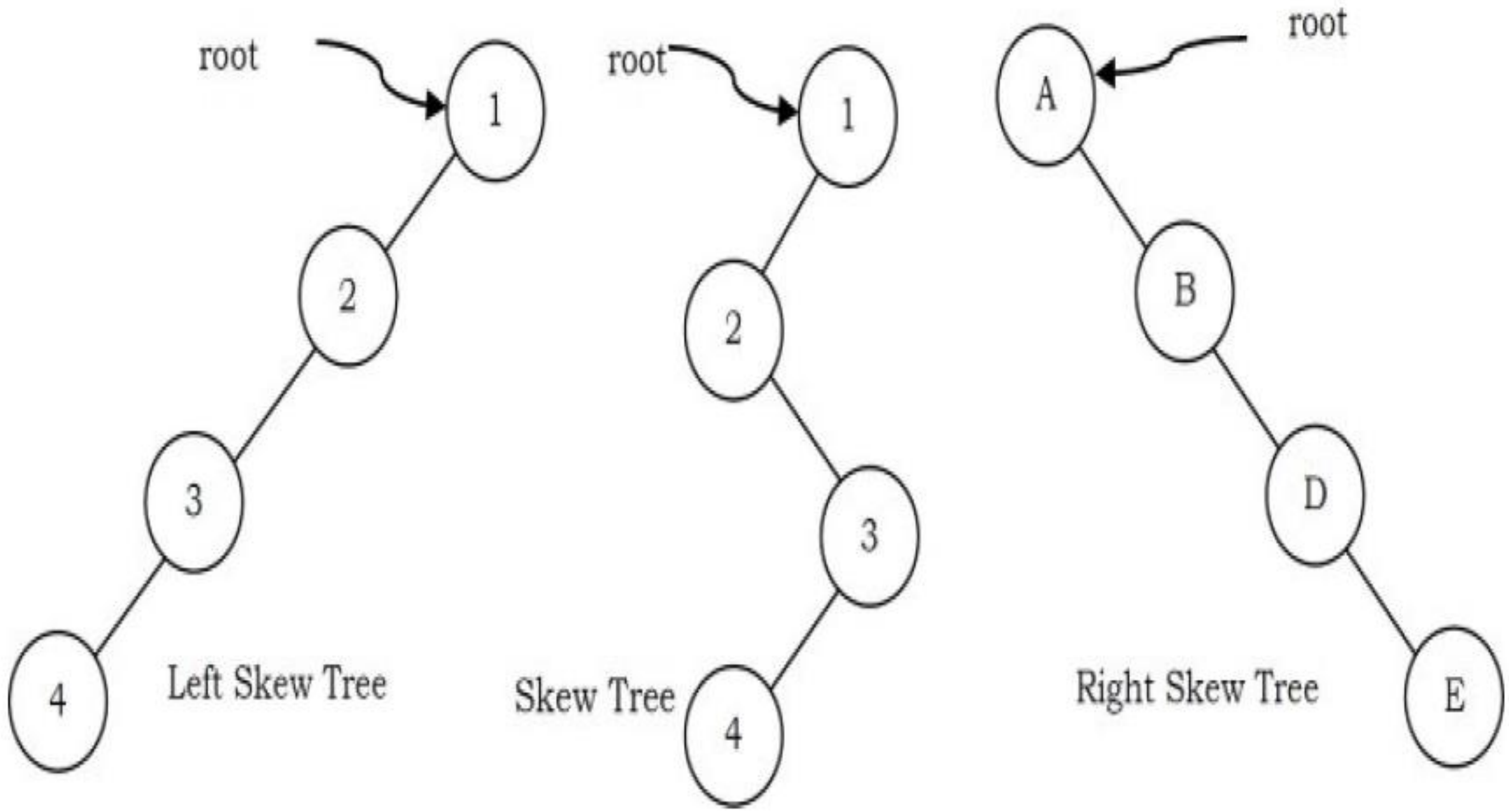
Terminology (Continued)

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

Terminology (Continued)

- *Height of the tree* is the maximum height among all the nodes in the tree and *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.
- The *size* of a node is the number of descendants it has including itself (the size of the subtree *C* is 3).
- 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*. Similarly, if every node has only right child then we call them *right skew trees.*

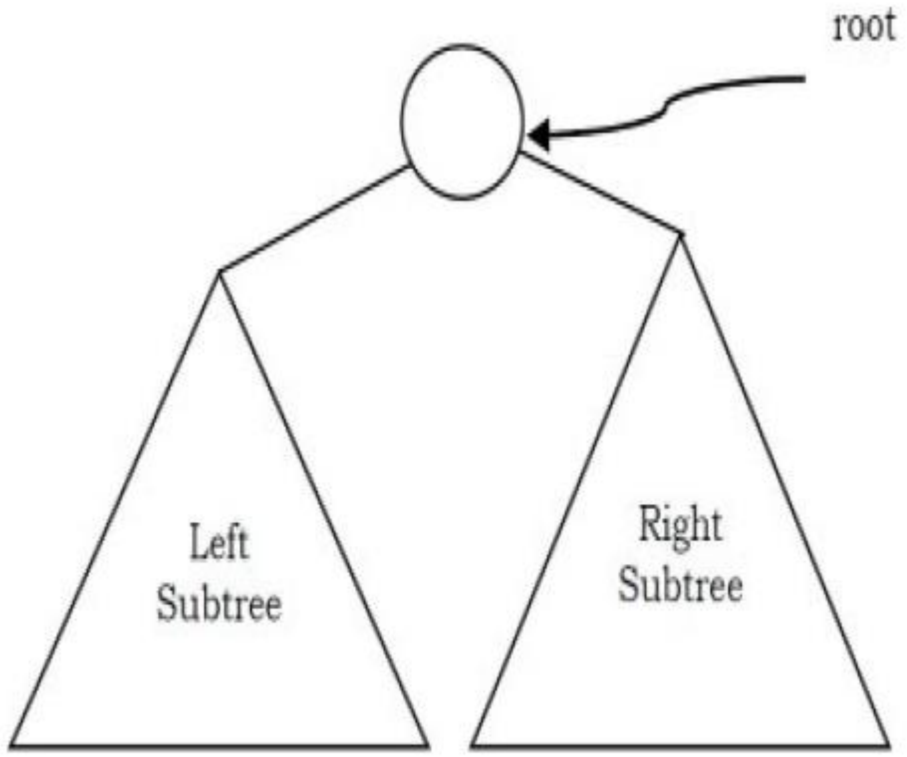
Terminology (Continued)

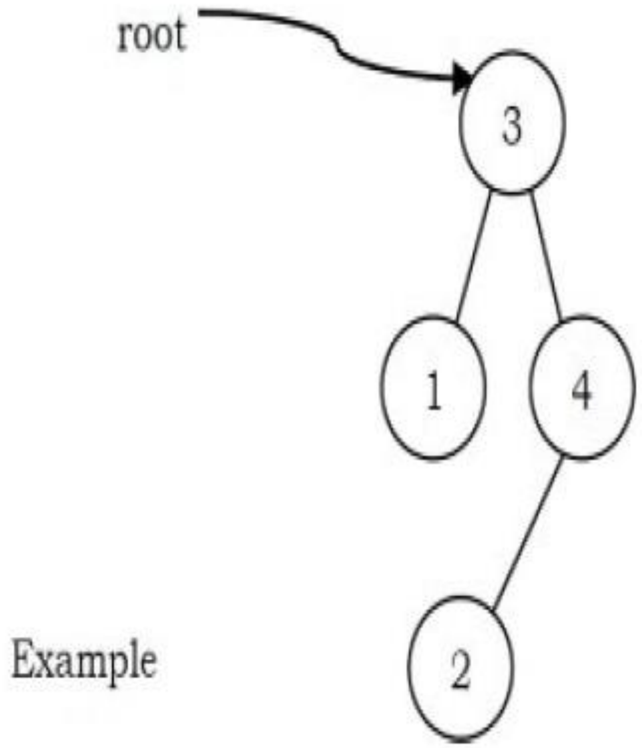


Binary Trees

- 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. We can visualize a binary tree as consisting of a root and two disjoint binary trees, called the left and right subtrees of the root.

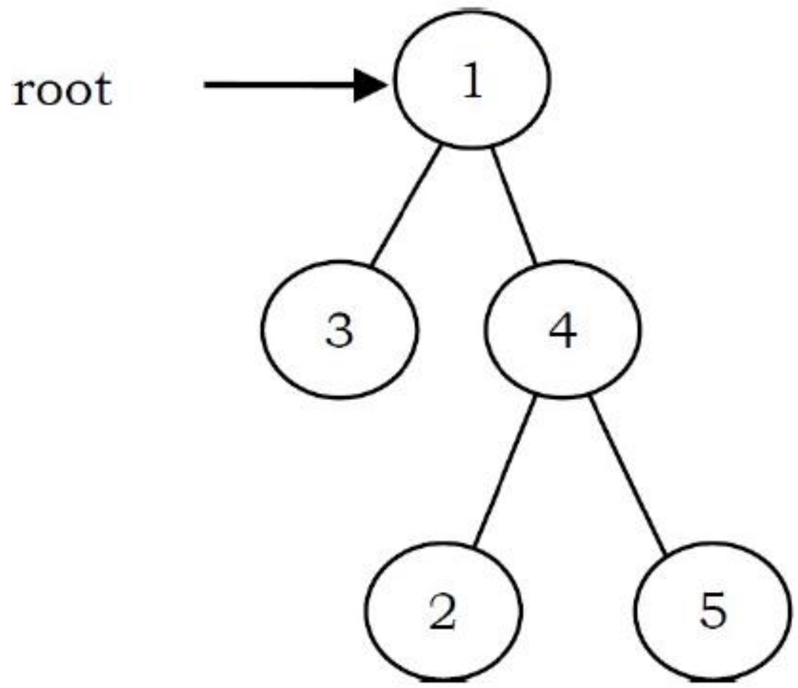
Generic Binary Tree





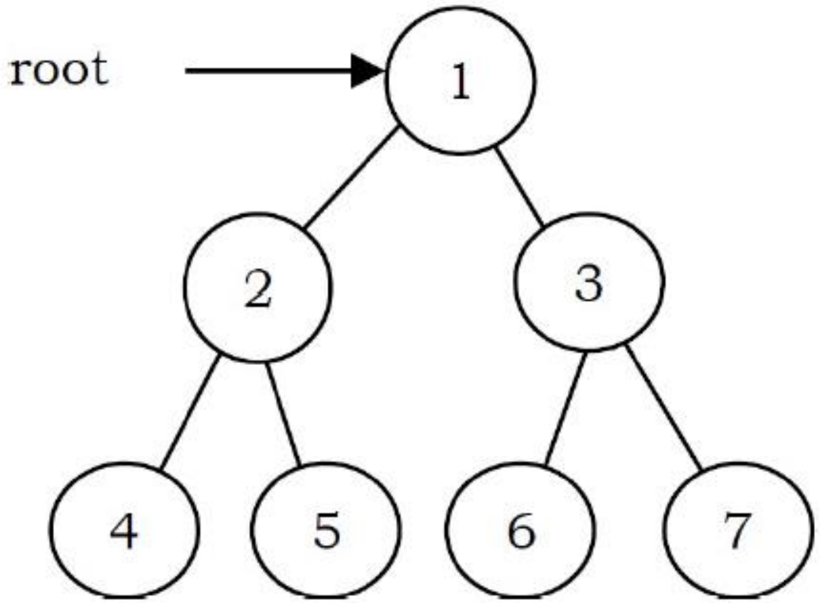
Types of Binary Trees

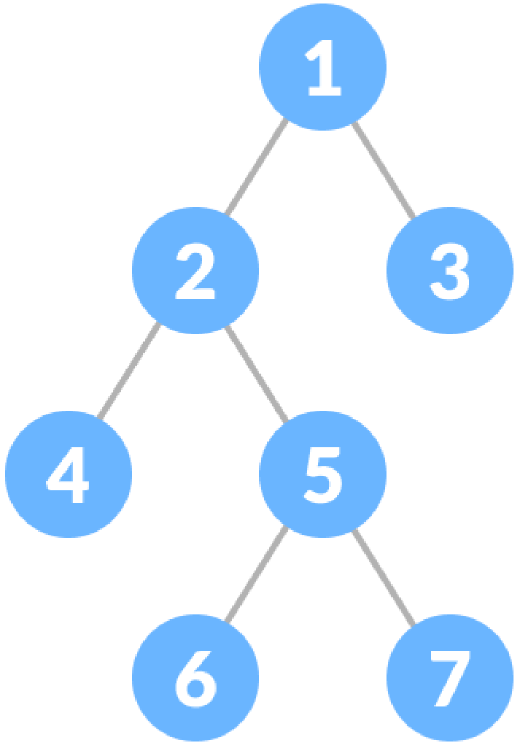
Strict Binary Tree: A binary tree is called *strict binary tree* if each node has exactly two children or no children.



Types of Binary Trees

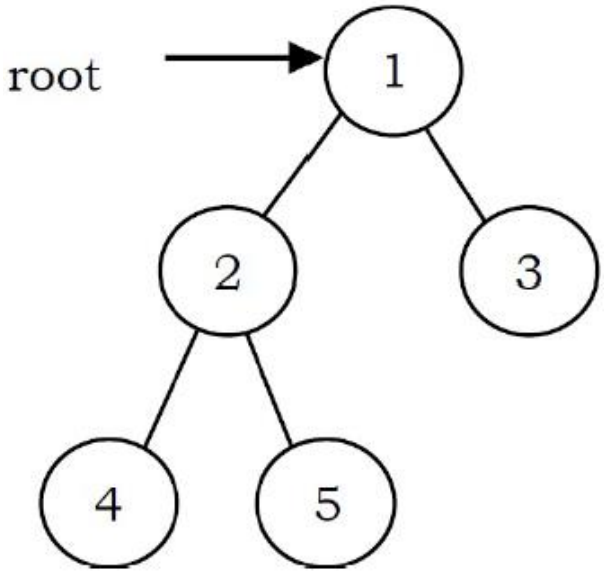
Full Binary Tree: A binary tree is called *full binary tree* if every parent/internal node has two or no children.





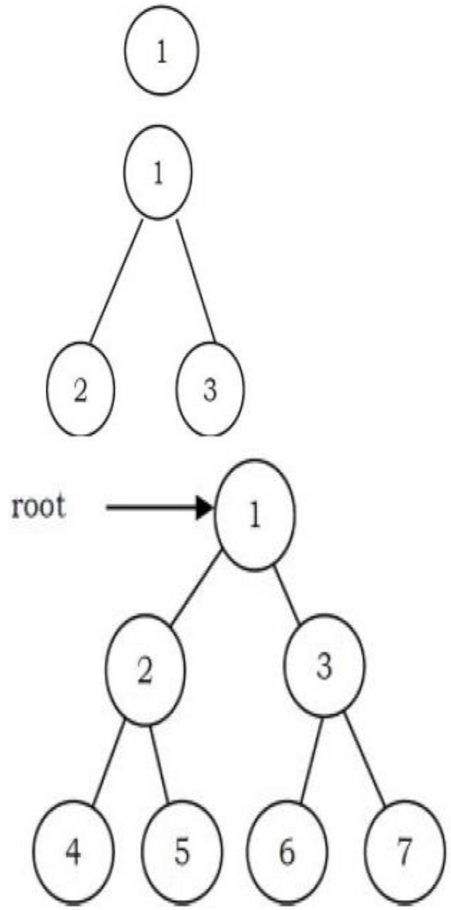
Types of Binary Trees

Complete Binary Tree: In complete binary trees, if we number 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. A binary tree is called *complete binary tree* if all leaf nodes are at height h or $h - 1$ and also without any missing number in the sequence. A complete binary tree is also a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.



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.



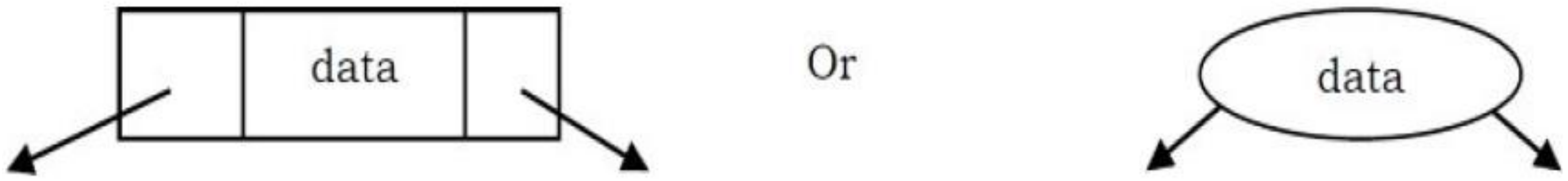
Height	**Number of nodes at level *h***
$h = 0$	$2^0 = 1$
$h = 1$	$2^1 = 2$
$h = 2$	$2^2 = 4$

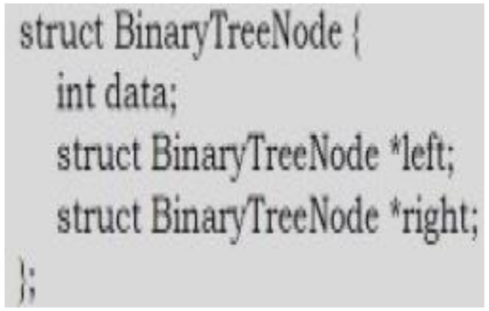
Properties of Binary Trees (Continued)

- 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

- Now let us define structure of the binary tree. For simplicity, 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:





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

Operations on Binary Trees

Main Tree Operations

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

Auxiliary Tree Operations

- Finding the size of the tree
- Finding the height of the tree

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).

Binary Tree Traversals

- In order to process trees, we need a mechanism for traversing them. 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. As we have already seen in linear data structures (like linked lists, stacks, queues, etc.), the elements are visited in sequential order. But, 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. In addition, all nodes are processed in the *traversal but searching* stops when the required node is found.

Traversal Possibilities

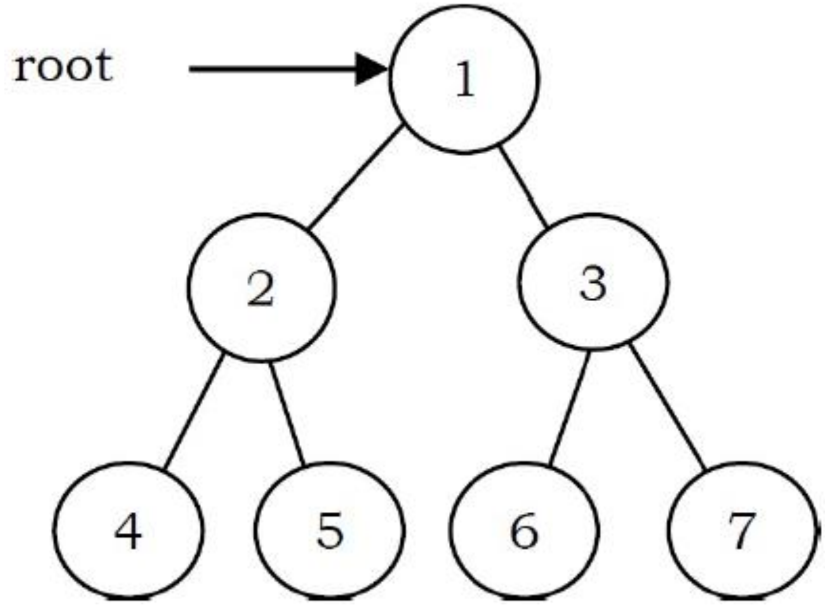
- Starting at the root of a binary tree, there are three main steps that can be performed and the order in which they are performed defines the traversal type. These steps are: performing an action on the current node (referred to as “visiting” the node and denoted with *“D”*), traversing to the left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

Classifying the Traversals

- The sequence in which these entities (*n*odes) are processed defines a particular traversal method. The classification is based on the order in which current node is processed. That means, 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.*

Classifying the Traversals (Continued)

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

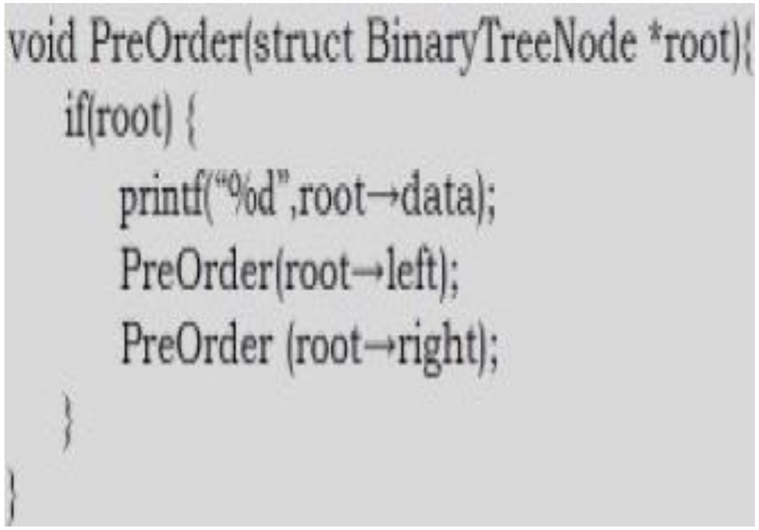


Preorder (DLR) Traversal

- In preorder traversal, each node is processed before (pre) either of its subtrees. This is the simplest traversal to understand. However, 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. To move to the right subtree after processing the left subtree, we must maintain the root information.

Preorder (DLR) Traversal (Continued)

- 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



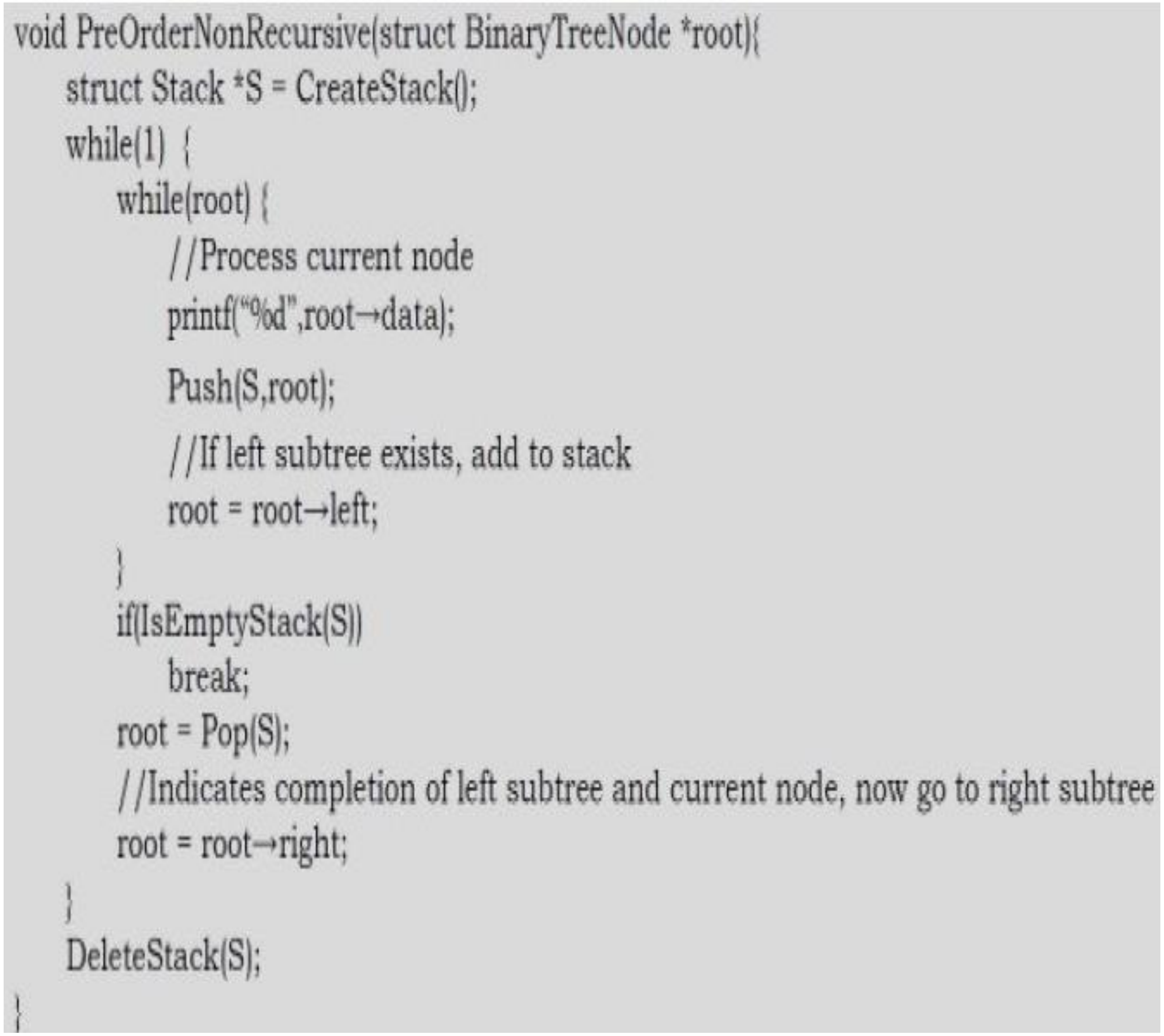
```
void PreOrder(struct BinaryTreeNode *root){
    if(root) {
        printf("%d",root→data);
        PreOrder(root→left);
        PreOrder (root→right);
    }
}
```

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.

Non-Recursive Preorder Traversal (Continued)

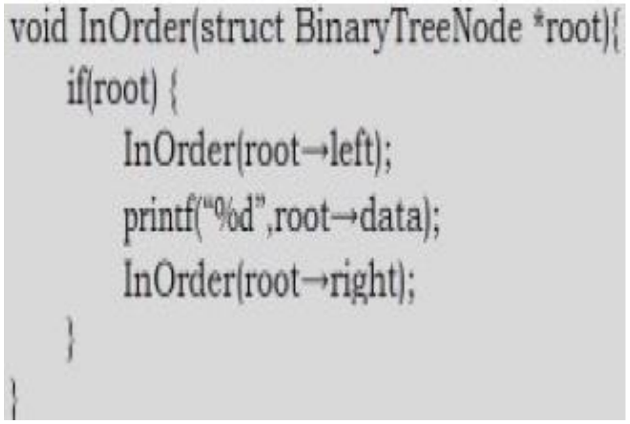


```
void PreOrderNonRecursive(struct BinaryTreeNode *root){
    struct Stack *S = CreateStack();
    while(1) {
        while(root) {
            //Process current node
            printf("%d",root→data);
            Push(S,root);
            //If left subtree exists, add to stack
            root = 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 (LDR) 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



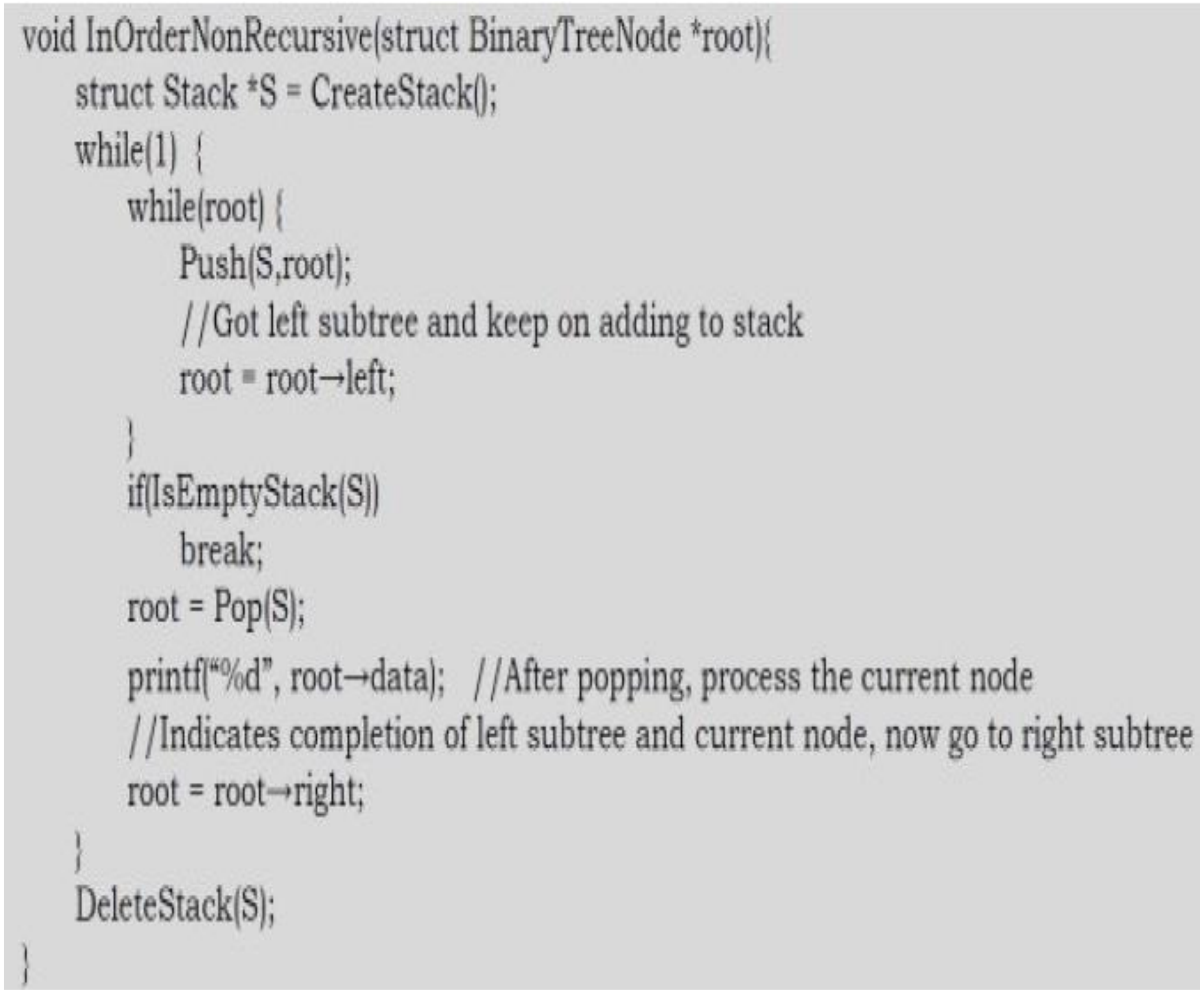
```
void InOrder(struct BinaryTreeNode *root){
    if(root) {
        InOrder(root→left);
        printf("%d",root→data);
        InOrder(root→right);
    }
}
```

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

Non-Recursive Inorder (LDR) 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).

Non-Recursive Inorder (LDR) Traversal (Continued)

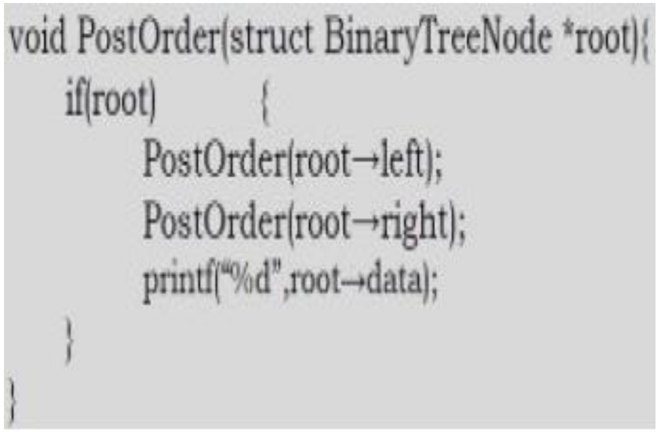


```
void InOrderNonRecursive(struct BinaryTreeNode *root){
    struct Stack *S = CreateStack();
    while(1) {
        while(root) {
            Push(S,root);
            //Got left subtree and keep on adding to stack
            root = root→left;
        }
        if(IsEmptyStack(S))
            break;
        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 (LRD) 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 tree would be visited in the order: 4 5 2 6 7 3 1



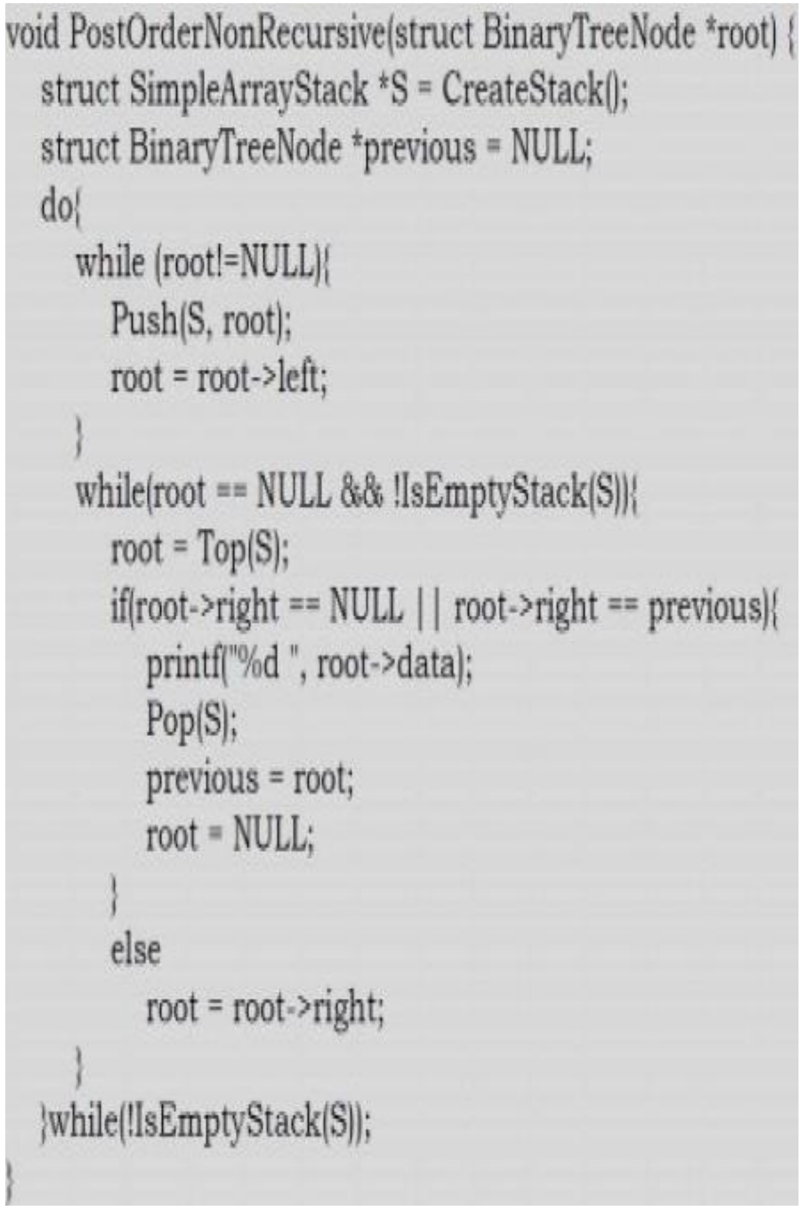
```
void PostOrder(struct BinaryTreeNode *root){
    if(root)        {
        PostOrder(root→left);
        PostOrder(root→right);
        printf("%d",root→data);
    }
}
```

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

Non-Recursive Postorder (LRD) 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 (i.e., *current* is a leaf node), we print *current's* value and pop it off the stack.

Non-Recursive Postorder (LRD) Traversal (Continued)



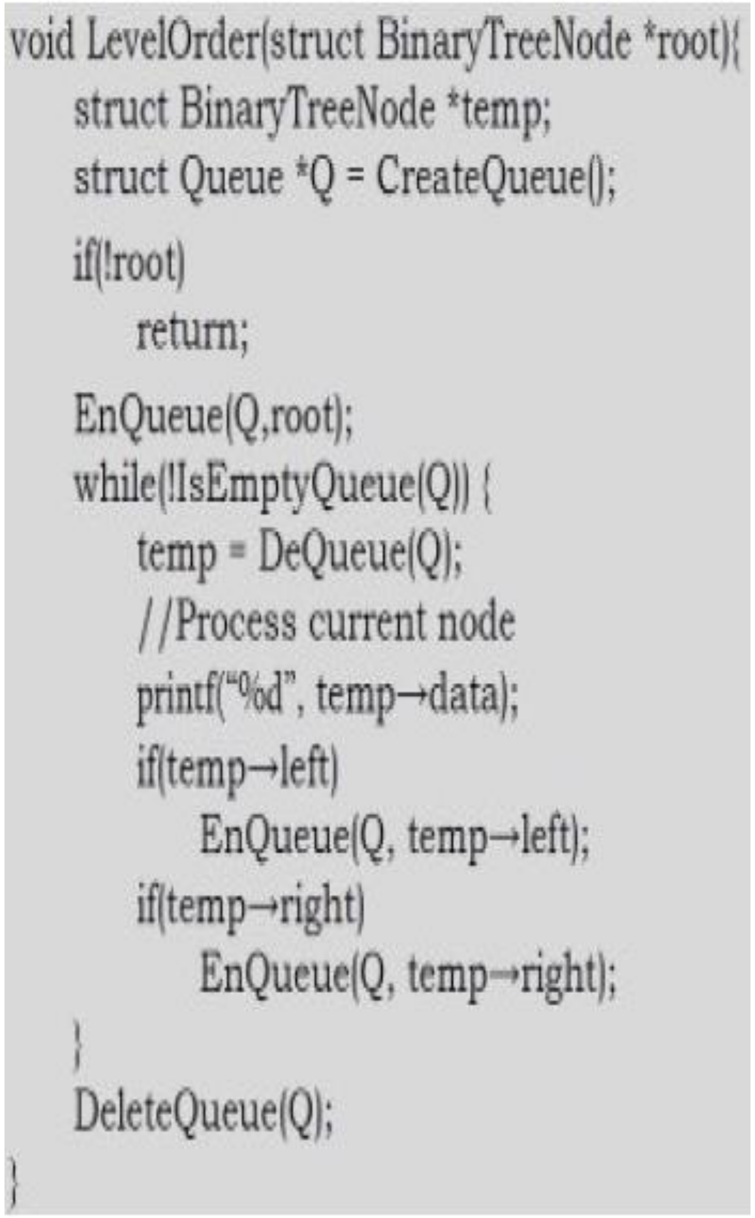
```
void PostOrderNonRecursive(struct BinaryTreeNode *root) {
  struct SimpleArrayStack *S = CreateStack();
  struct BinaryTreeNode *previous = NULL;
  do{
    while (root!=NULL){
      Push(S, root);
      root = root->left;
    }
    while(root == NULL && !IsEmptyStack(S)){
      root = Top(S);
  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

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 tree would be visited in the order: 1 2 3 4 5 6 7

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

Level Order Traversal (Continued)



```
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);
}
```

Infix, Prefix and Postfix Expressions

- When you write an arithmetic expression such as B * C, the form of the expression provides you with information so that you can interpret it correctly. In this case we know that the variable B is being multiplied by the variable C since the multiplication operator * appears between them in the expression. This type of notation is referred to as **infix** since the operator is *in between* the two operands that it is working on.
- Each operator has a **precedence** level. Operators of higher precedence are used before operators of lower precedence. The only thing that can change that order is the presence of parentheses. The precedence order for arithmetic operators places multiplication and division above addition and subtraction. If two operators of equal precedence appear, then a left-to-right ordering or associativity is used.

Infix, Prefix and Postfix Expressions (Continued)

- Prefix expression notation requires that all operators precede the two operands that they work on. Postfix, on the other hand, requires that its operators come after the corresponding operands.
- A + B * C would be written as + A * B C in prefix. The multiplication operator comes immediately before the operands B and C, denoting that * has precedence over +. The addition operator then appears before the A and the result of the multiplication.
- In postfix, the expression would be A B C * +. Again, the order of operations is preserved since the * appears immediately after the B and the C, denoting that * has precedence, with + coming after. Although the operators moved and now appear either before or after their respective operands, the order of the operands stayed exactly the same relative to one another.

Infix, Prefix and Postfix Expressions (Continued)

Infix Expression	Prefix Expression	Postfix Expression
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +

- Now consider the infix expression (A + B) * C. Recall that in this case, infix requires the parentheses to force the performance of the addition before the multiplication. However, when A + B was written in prefix, the addition operator was simply moved before the operands, + A B. The result of this operation becomes the first operand for the multiplication. The multiplication operator is moved in front of the entire expression, giving us * + A B C. Likewise, in postfix A B + forces the addition to happen first. The multiplication can be done to that result and the remaining operand C. The proper postfix expression is then A B + C *.

Infix, Prefix and Postfix Expressions (Continued)

- The operators are no longer ambiguous with respect to the operands that they work on. Only infix notation requires the additional symbols. The order of operations within prefix and postfix expressions is completely determined by the position of the operator and nothing else. In many ways, this makes infix the least desirable notation to use.

Infix Expression	**Prefix Expression**	**Postfix Expression**
(A + B) * C	* + A B C	A B + C *

Infix, Prefix and Postfix Expressions (Continued)

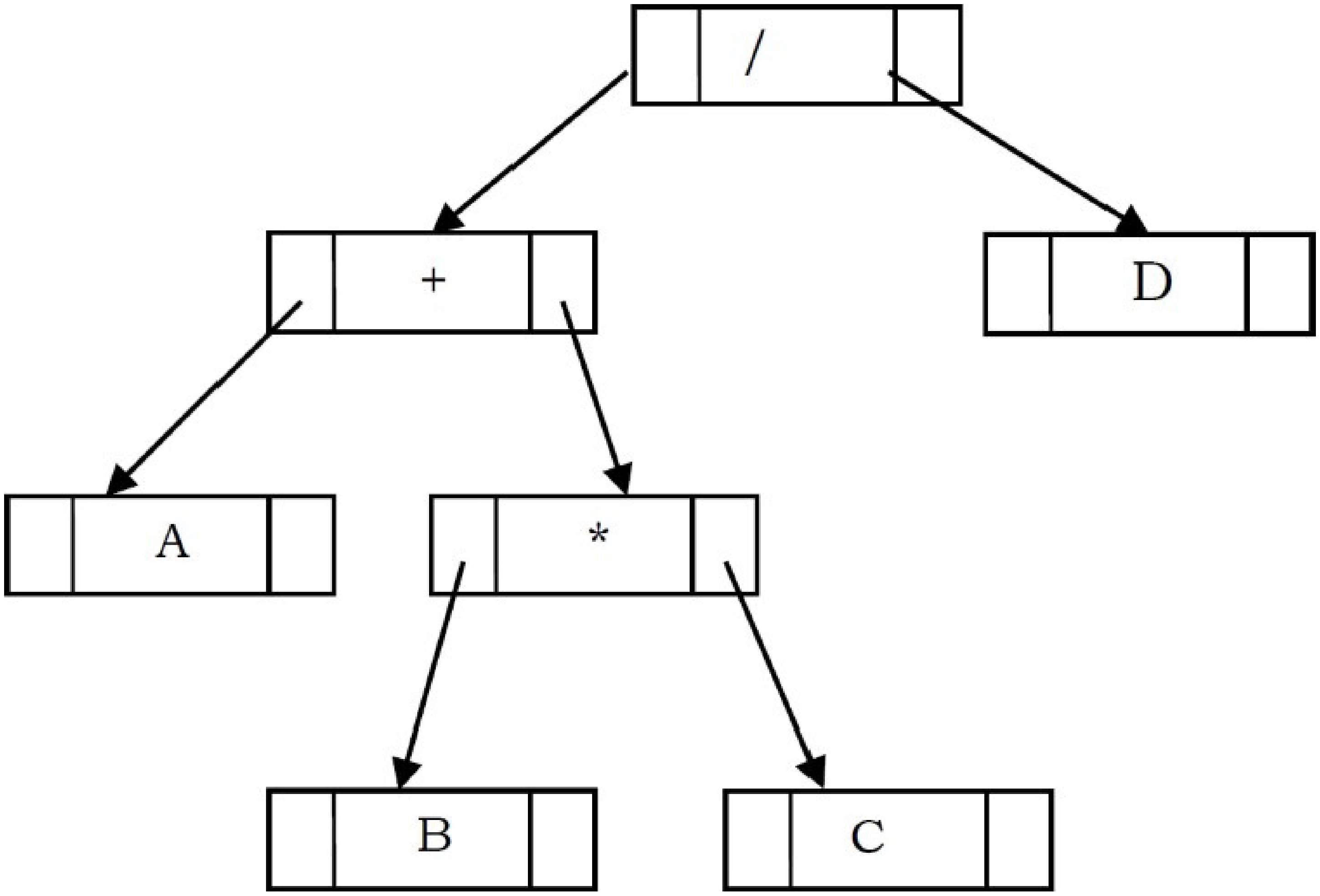
Infix Expression	**Prefix Expression**	**Postfix Expression**
A + B * C + D	+ + A * B C D	A B C * + D +
(A + B) * (C + D)	* + A B + C D	A B + C D + *
A * B + C * D	+ * A B * C D	A B * C D * +
A + B + C + D	+ + + A B C D	A B + C + D +

Expression Trees

- A tree representing an expression is called an *expression tree*. In expression trees, leaf nodes are operands and non-leaf nodes are operators. That means, an expression tree is a binary tree where internal nodes are operators and leaves are operands. An expression tree consists of binary expression. The figure below shows a simple expression tree for (A + B * C) / D.

Expression Trees (Continued)

- The figure below shows a simple expression tree for (A + B * C) / D.

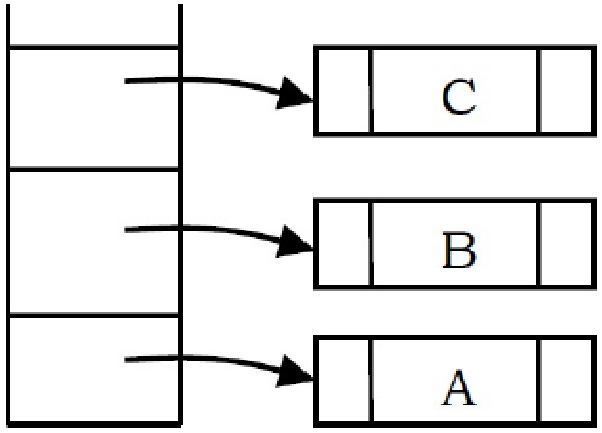


Algorithm for Building Expression Tree from Postfix Expression

- Assume that one symbol is read at a time. If the symbol is an operand, we create a tree node and push a pointer to it onto a stack. If the symbol is an operator, pop pointers to two trees T_1 and T_2 from the stack (T_1 is popped first) and form a new tree whose root is the operator and whose left and right children point to T_2 and T_1 respectively. A pointer to this new tree is then pushed onto the stack.

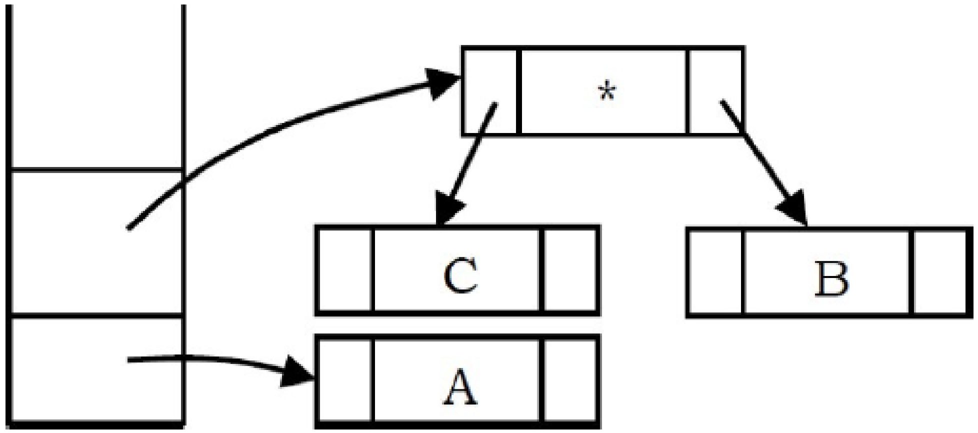
Algorithm for Building Expression Tree from Postfix Expression (Continued)

- As an example, assume the input is A B C * + D /. The first three symbols are operands, so create tree nodes and push pointers to them onto a stack as shown below.



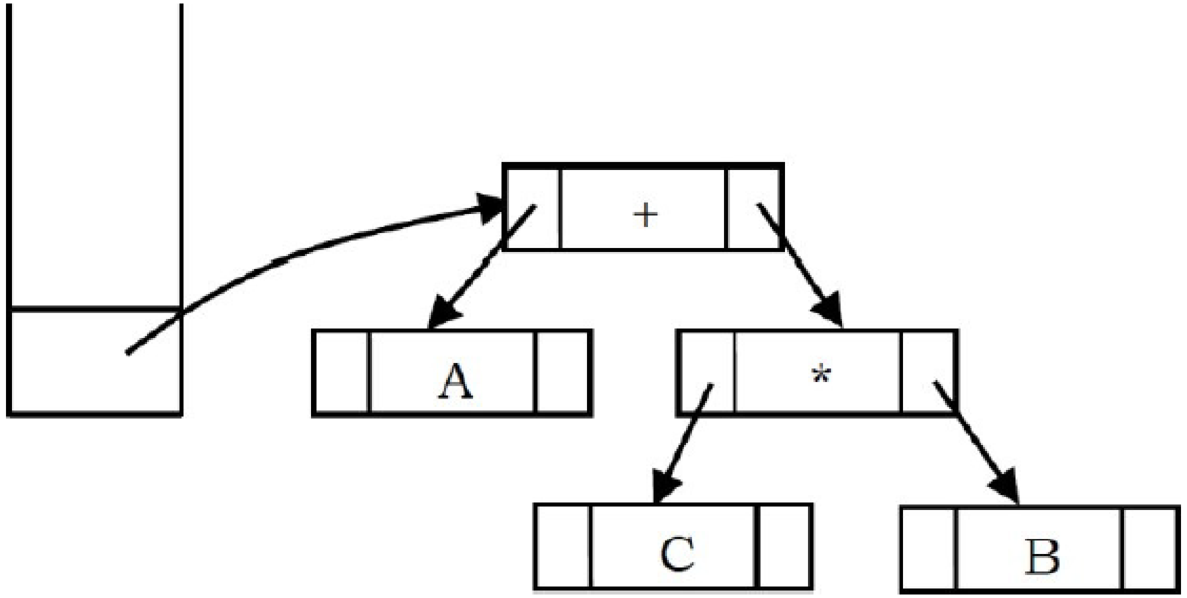
Algorithm for Building Expression Tree from Postfix Expression (Continued)

- Next, an operator ‘*’ is read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack.



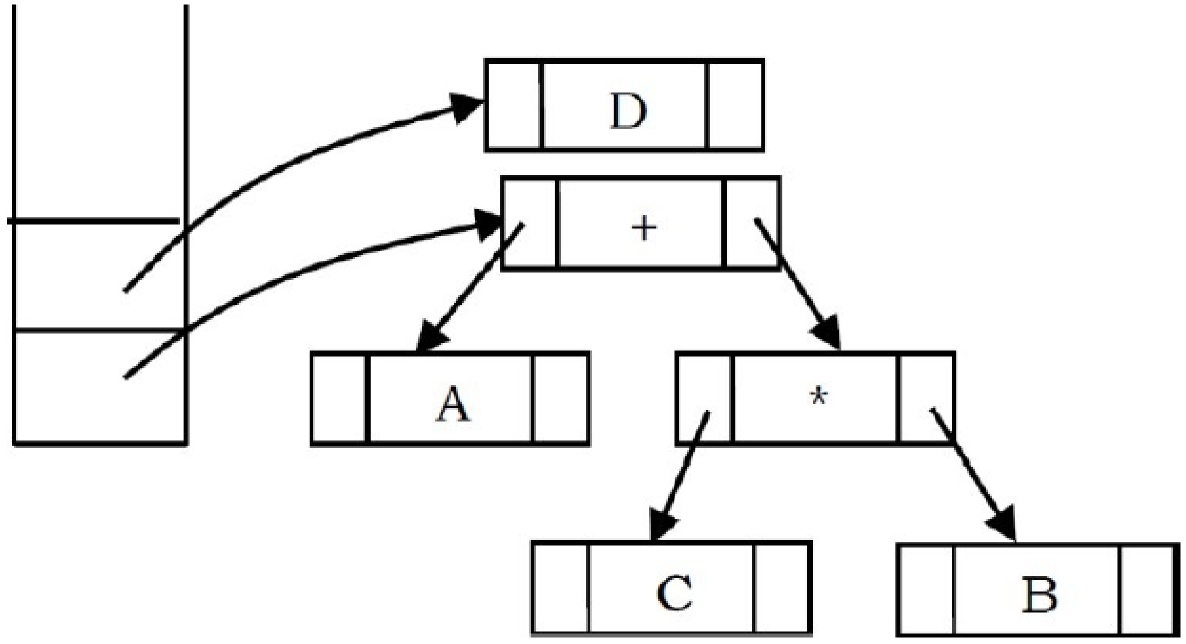
Algorithm for Building Expression Tree from Postfix Expression (Continued)

- Next, an operator ‘+’ is read, so two pointers to trees are popped, a new tree is formed and a pointer to it is pushed onto the stack.



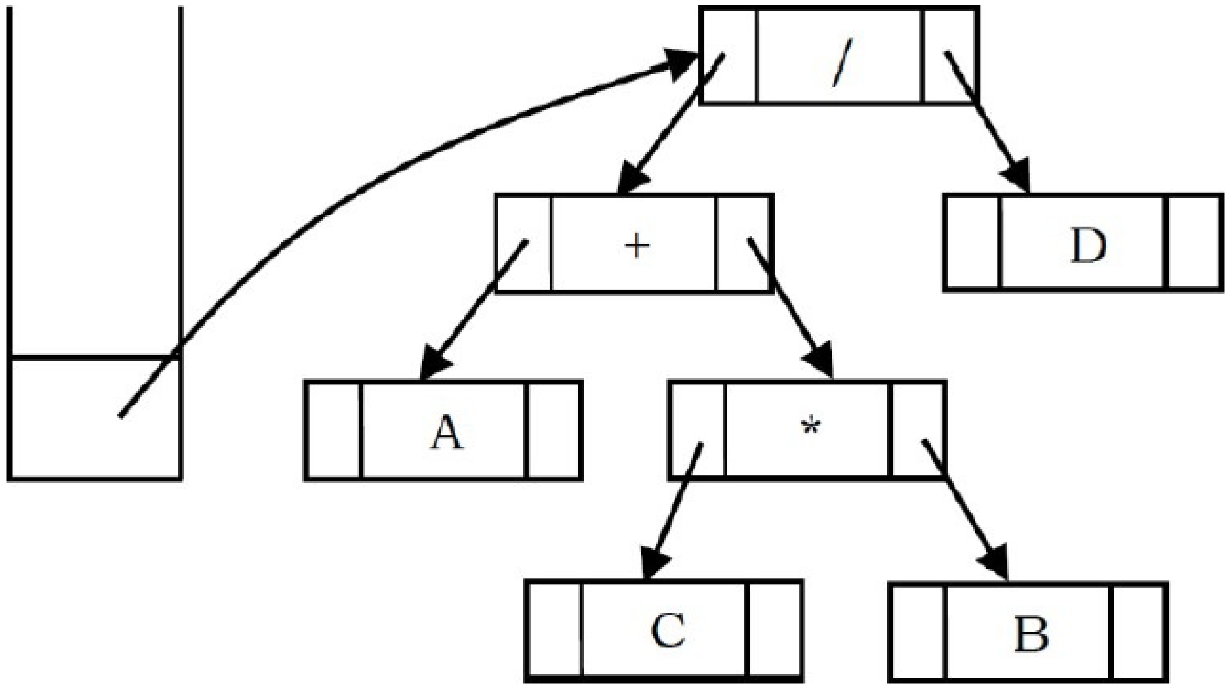
Algorithm for Building Expression Tree from Postfix Expression (Continued)

- Next, an operand ‘D’ is read, a one-node tree is created and a pointer to the corresponding tree is pushed onto the stack.



Algorithm for Building Expression Tree from Postfix Expression (Continued)

- Finally, the last symbol ('/') is read, two trees are merged and a pointer to the final tree is left on the stack.



References

- Assist. Prof. Dr. Fatih Tekbacak, Lecture Notes, Adnan Menderes University, Turkey.
- Narasimha Karumanchi; “Data Structures and Algorithms Made Easy: Data Structure and Algorithmic Puzzles”, 5th Ed., CareerMonk Publications, 2016.
- Bradley N. Miller, David L. Ranum; “Problem Solving with Algorithms and Data Structures Using Python”, 2nd Ed., Franklin, Beedle & Associates, 2011.