Unit 4 -Trees

Recursion

The process in which a function calls itself is known as recursion and the corresponding function is called the recursive function. The popular example to understand the recursion is factorial function.

Factorial function: factorial(n) = n*factorial(n-1), base condition: if n<=1 then f(n) = 1.

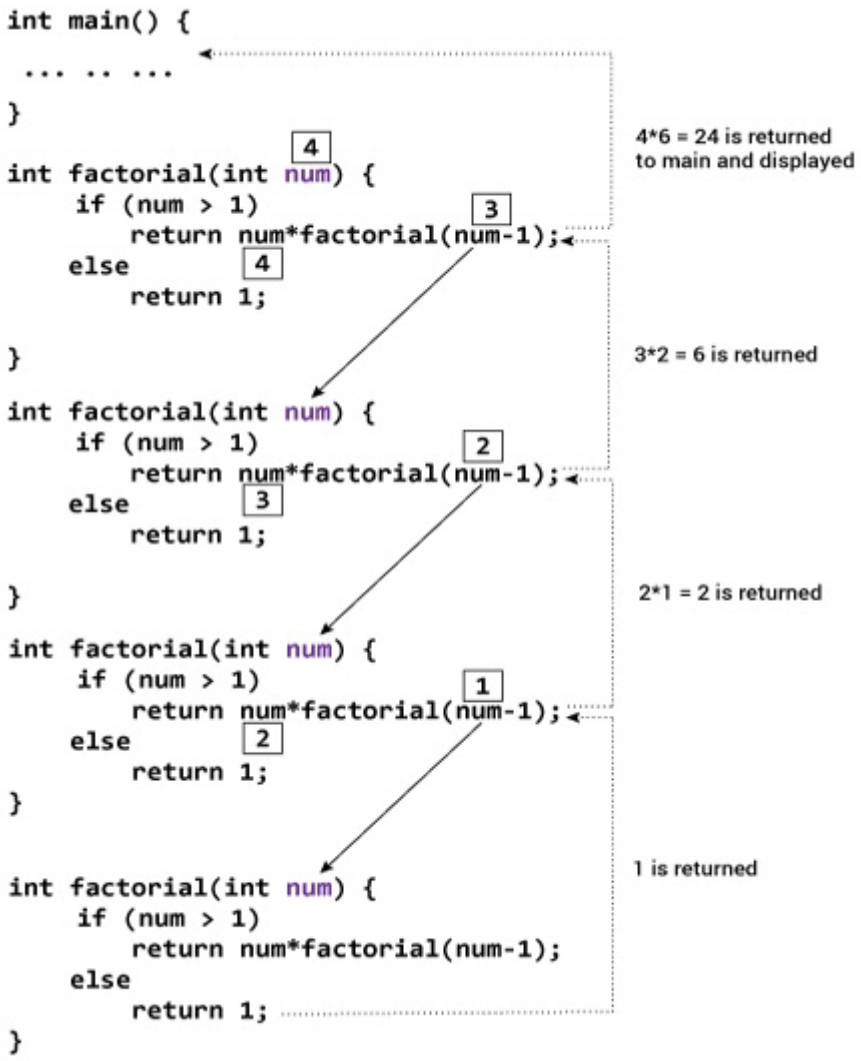
Example: Factorial

```
#include <iostream>
//Factorial function
int factorial(int num)
{
  /* This is called the base condition, it is
   * very important to specify the base condition
   * in recursion, otherwise your program will throw
   * stack overflow error.
   */
  if (num > 1)
      return num*factorial(num-1);
else
     return(1);
}
int main()
{
  int num;
  cout<<"Enter a number: ";
  cin>>num;
  cout<<"Factorial of entered number: "<<factorial(num);
  return 0;
}
```

Properties of recursive function

1. The function definition should contain a statement that call the same function.
2. The function should contain a conditional statement that should terminate the execution of the function.

Working of Factorial Program



Tree Data Structure

It can extend the concept of linked data structure (linked list, stack, queue) to a structure that may have multiple relations among its nodes. Such a structure is called a tree. A tree is a collection of nodes connected by directed (or undirected) edges. A tree is a nonlinear data structure, compared to arrays, linked lists, stacks and queues which are linear data structures.

A tree can be empty with no nodes or a tree is a structure consisting of one node called the root and zero or one or more sub trees. A tree has following general properties:

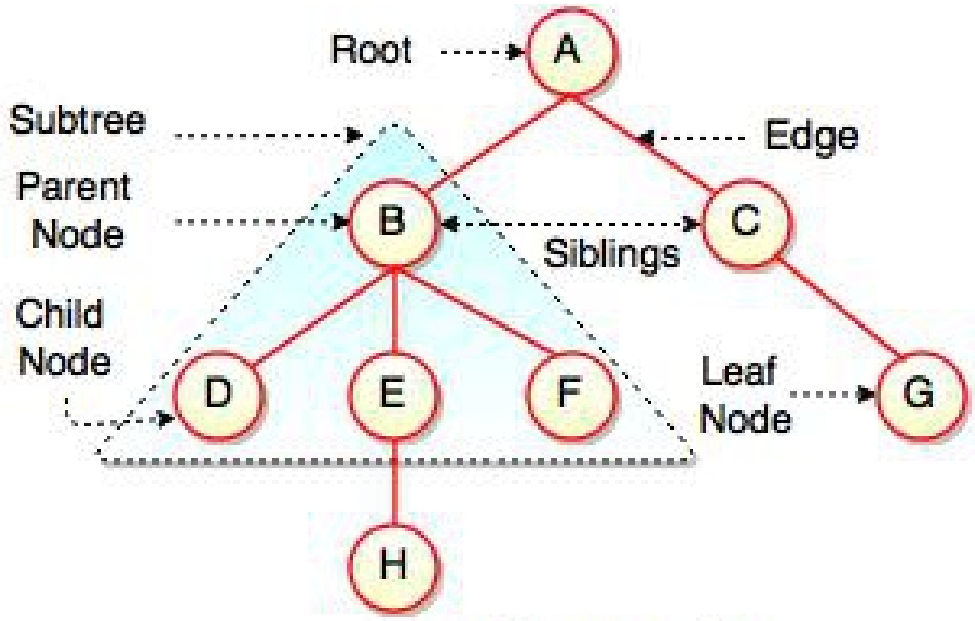


Fig. Structure of Tree

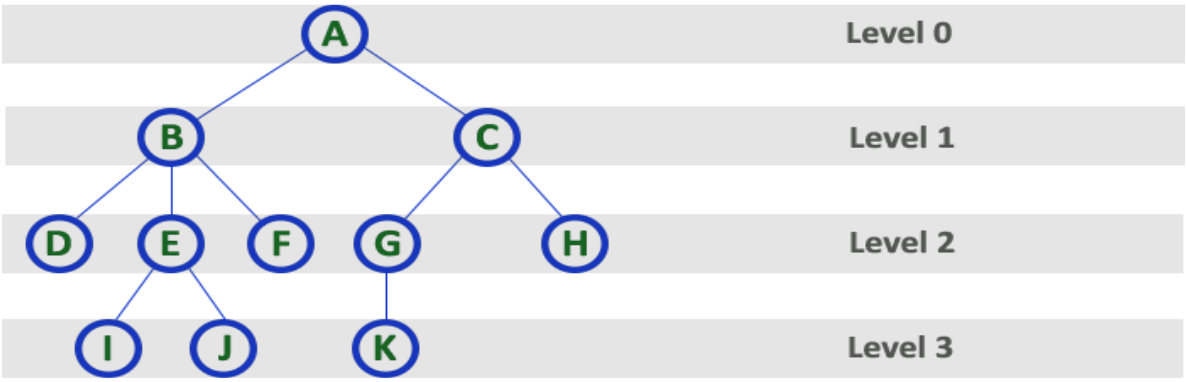
The above figure represents structure of a tree. Tree has 2 subtrees.
A is a parent of B and C.
B is called a child of A and also parent of D, E, F.

Tree is a collection of elements called Nodes, where each node can have arbitrary number of children.

Field	Description
Root	Root is a special node in a tree. The ent through it. It does not have a parent.
Parent Node	Parent node is an immediate predeces
Child Node	All immediate successors of a node are
Siblings	Nodes with the same parent are called
Path	Path is a number of successive edges fr destination node.
Height of Node	Height of a node represents the numbe path between that node and a leaf.
Height of Tree	Height of tree represents the height of
Depth of Node	Depth of a node represents the numbe root node to the node.
Degree of Node	Degree of a node represents a number
Edge	Edge is a connection between one node between two nodes or a node and a leaf.

Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



Depth of a Tree/Node

In a tree data structure, the total number of edges from root node to a particular node is called as DEPTH of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, depth of the root node is '0'.

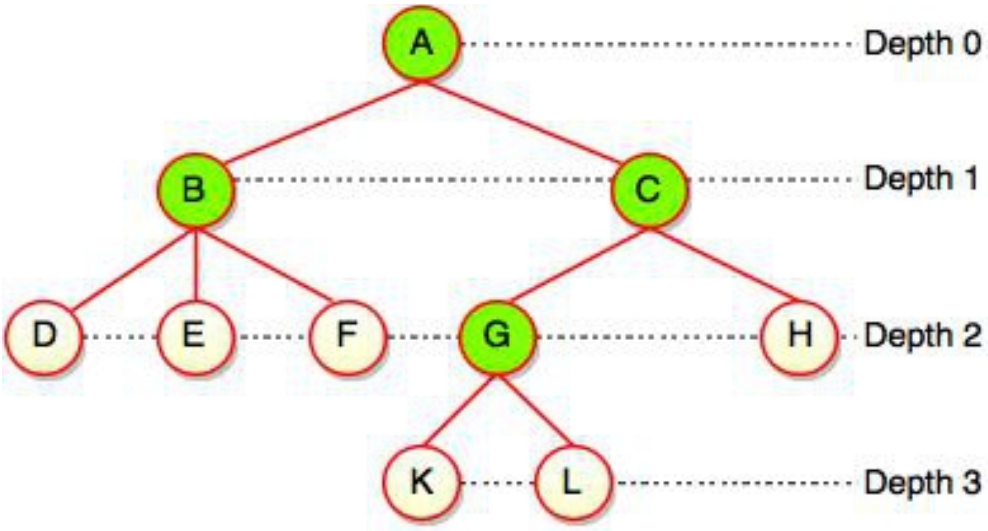
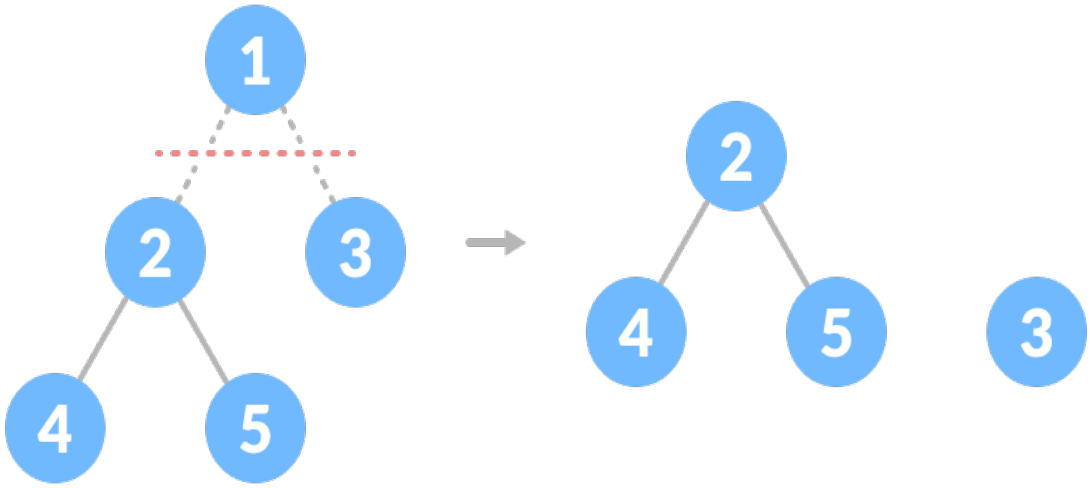


Fig. Depth of a Node

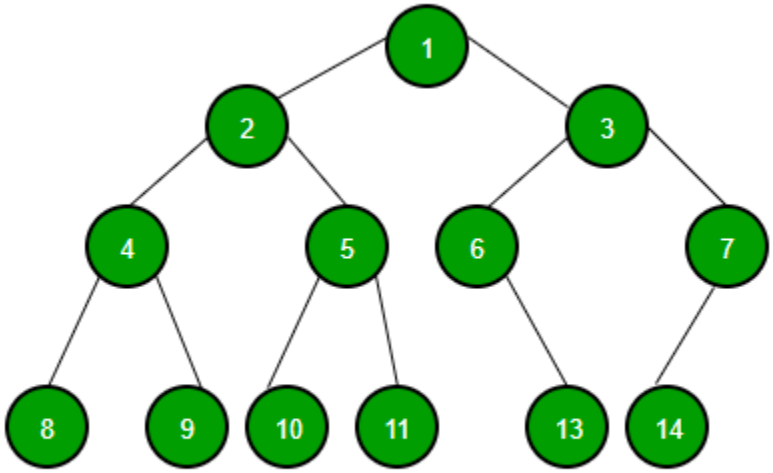
Forest

A collection of disjoint trees is called a forest.



Binary Trees

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



So, a binary tree is a hierarchical data structure in which each node has at most two children generally referred as left child and right child.

Each node contains three components:

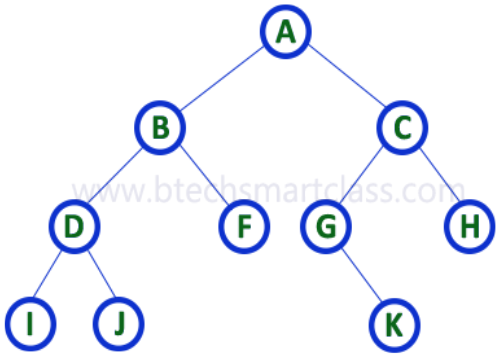
Pointer to left subtree
Pointer to right subtree
Data element
The topmost node in the tree is called the root. An empty tree is represented by NULL pointer.

Binary Tree Representations

A binary tree data structure is represented using two methods. Those methods are as follows...

1. Array Representation
2. Linked List Representation

Consider the following binary tree...



Array Representation of Binary Tree

In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.

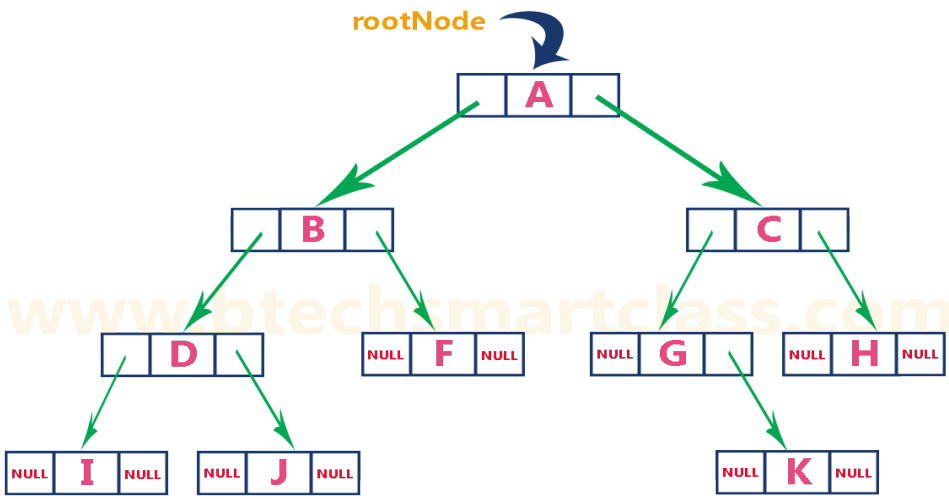
Consider the above example of a binary tree and it is represented as follows...

Linked List Representation of Binary Tree

We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

In this linked list representation, a node has the following structure...

The above example of the binary tree represented using Linked list representation is shown as follows...



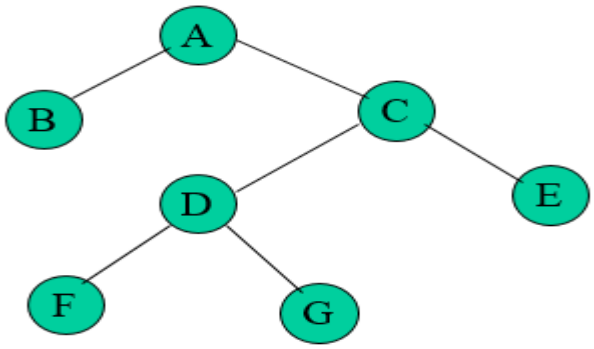
Binary trees types

1. **Strictly Binary tree**

   If every non leaf node in a binary tree has nonempty left and right sub trees, the tree is called a strictly binary tree.

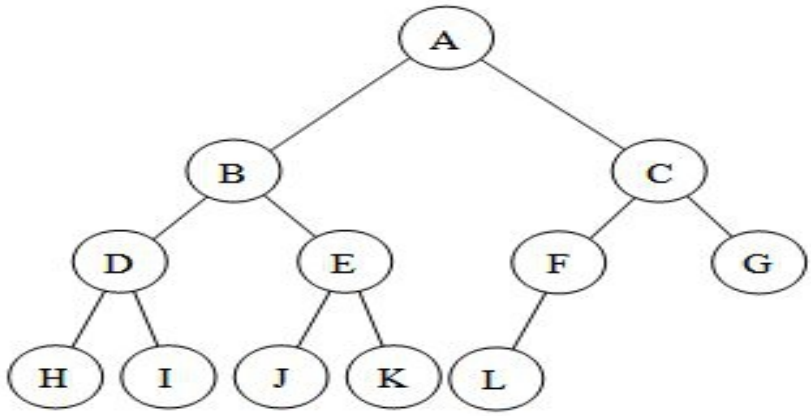
   A strictly binary tree with n leaves always contains 2n -1 nodes.

   Example:



2. **Complete Binary Tree**

   A Binary Tree is a complete Binary Tree if all the levels are completely filled except possibly the last level and the last level has all keys as left as possible.



3. **Fully Binary Trees**

   A full binary tree (sometimes proper binary tree or 2-tree) is a tree in which every node other than the leaves has two children.

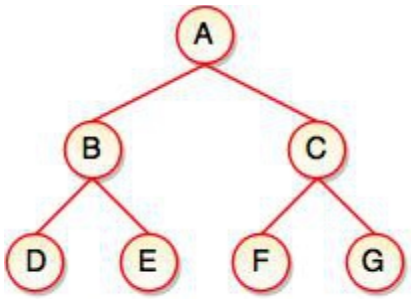
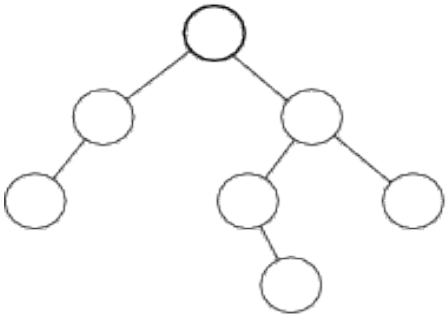
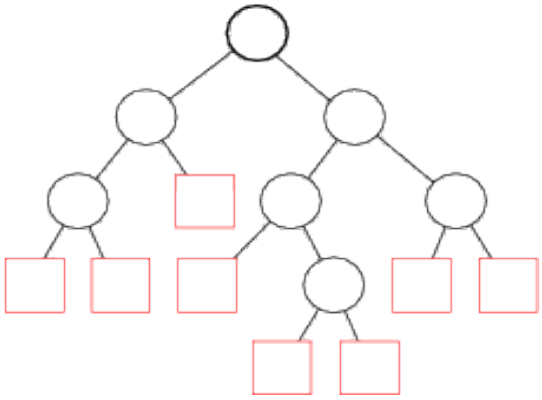


Fig. Full Binary Tree

4. **Extended Binary Trees**

   Extended binary tree is a type of binary tree in which all the null sub tree of the original tree are replaced with special nodes called external nodes whereas other nodes are called internal nodes





5. **Skewed Binary Trees**

A skewed binary tree is a type of binary tree in which all the nodes have only either one child or no child.

Types of Skewed Binary trees

There are 2 special types of skewed tree:

Left Skewed Binary Tree:

These are those skewed binary trees in which all the nodes are having a left child or no child at all. It is a left side dominated tree. All the right children remain as null.

Right Skewed Binary Tree:

These are those skewed binary trees in which all the nodes are having a right child or no child at all. It is a right side dominated tree. All the left children remain as null.

Example:

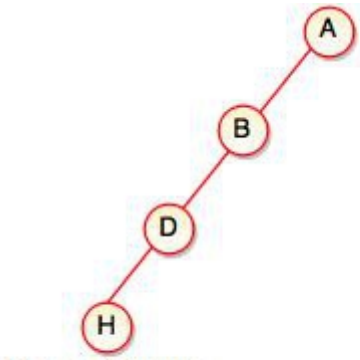


Fig. Left Skewed Binary Tree

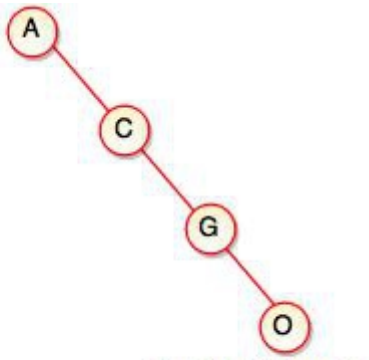


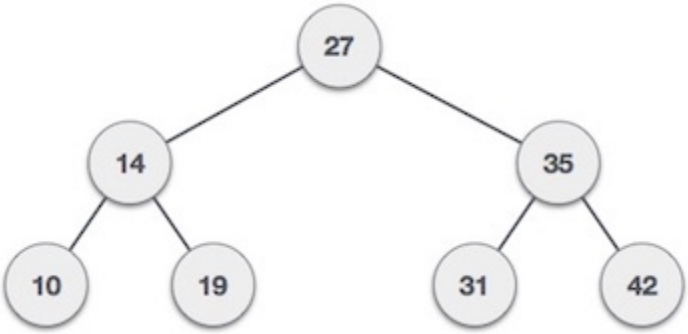
Fig. Right Skewed Binary Tree

6. Binary Search Trees

Binary Search Tree is a node-based binary tree data structure which has the following properties:

The left sub tree of a node contains only nodes with keys lesser than the node's key.

The right sub tree of a node contains only nodes with keys greater than the node's key.



Creation/node insertion of Binary Search Tree

To insert data into a binary tree involves a function searching for an unused node in the proper position in the tree in which to insert the key value. The insert function is generally a recursive function that continues moving down the levels of a binary tree until there is an unused leaf in a position which follows the following rules of placing nodes.

Algorithm

Step 1: Compare data of the root node and element to be inserted.

Step 2: If the data of the root node is greater, and if a left sub tree exists

then repeat step 1 with root = root of left sub tree

Else

Insert element as left child of current root.

return

Step 3: If the data of the root node is less and if a right sub tree exists

then repeat step 1 with root = root of right sub tree.

Else

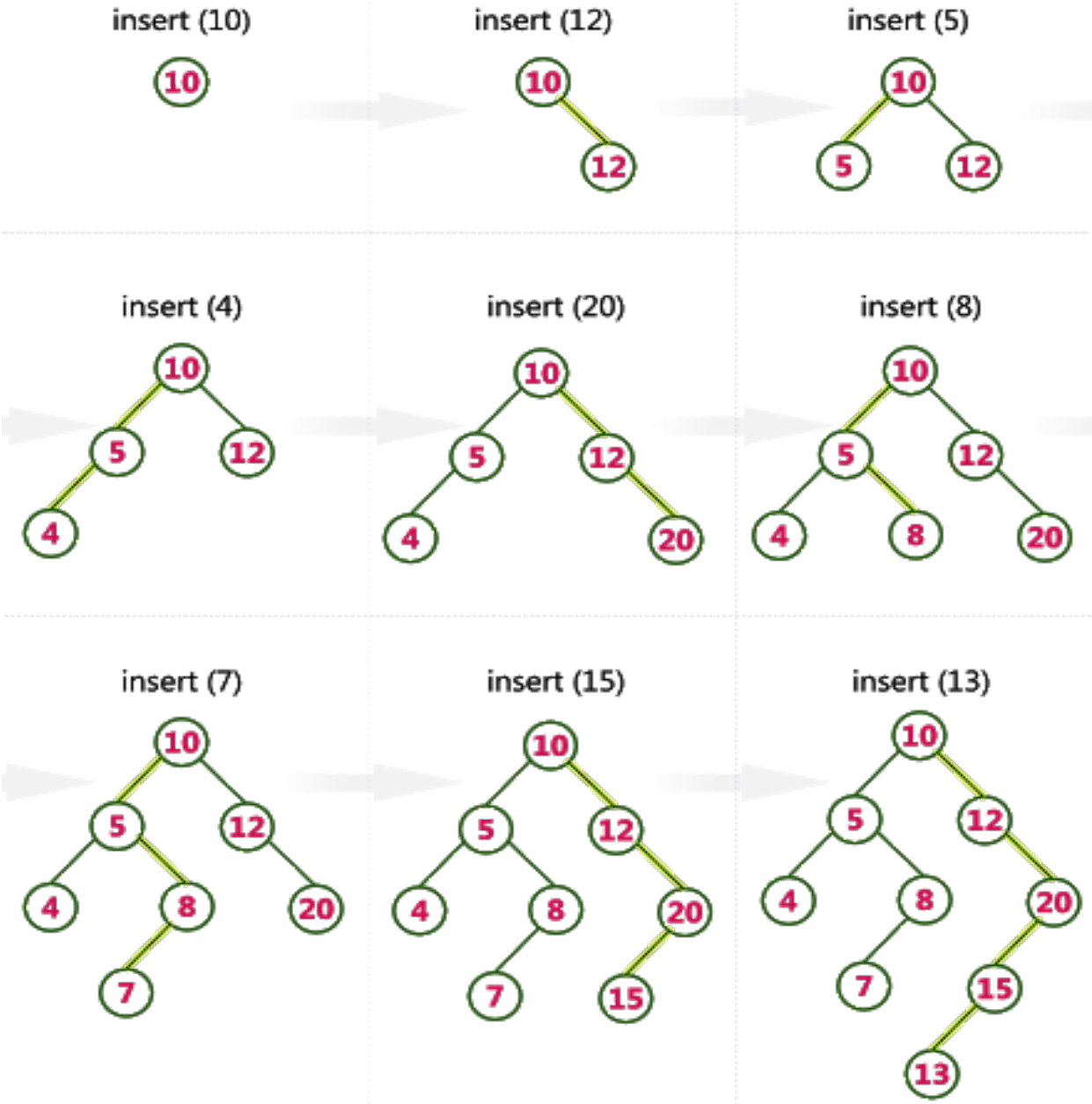
Insert element as right child of current root

return

Step 4: End

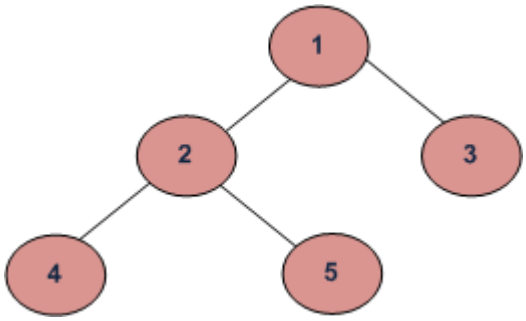
Example: Create a binary search tree using the following

10,12,5,4,20,8,7,15,13


insert (10)
10
insert (12)
10
12
insert (5)
10
5
12
insert (4)
10
5
12
4
insert (20)
10
5
12
4
20
insert (8)
10
5
12
4
8
20
insert (7)
10
5
12
4
8
20
7
insert (15)
10
5
12
4
8
20
7
15
insert (13)
10
5
12
4
8
20
7
15
13

Tree Traversal

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



1. **Preorder Traversal**

Algorithm Preorder(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

```
Void preorder(root)
{
  if (root!=NULL)
   {
     Cout<<root->info;
     preorder(root->lchild);
     preorder(root->rchild)
}
```

Preorder (Root, Left, Right) result : 1 2 4 5 3

2. **Inorder Traversal**

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Inorder (Left, Root, Right) result : 4 2 5 1 3

3. **Postorder Traversal**

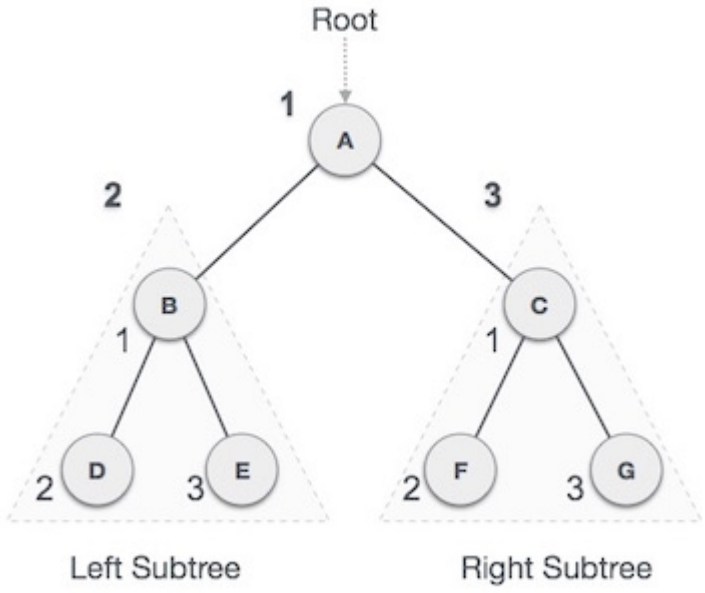
Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root;

Postorder (Left, Right, Root) result : 4 5 2 3 1

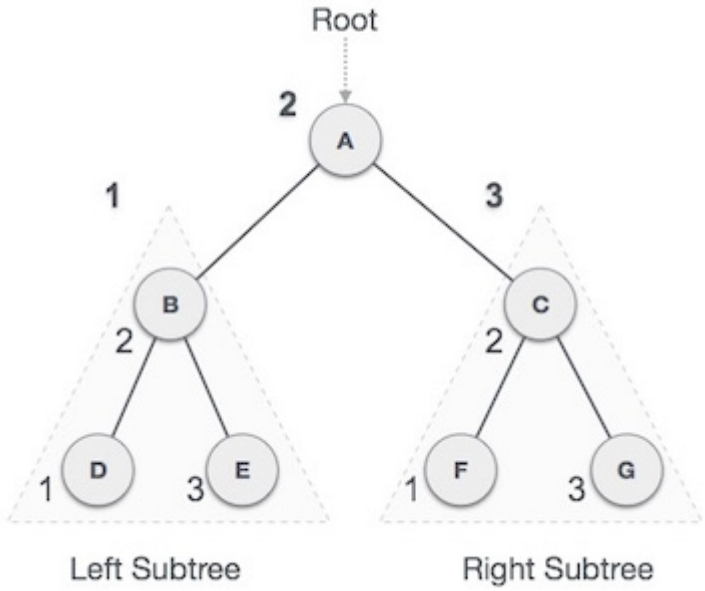
Example:

Preorder Traversal



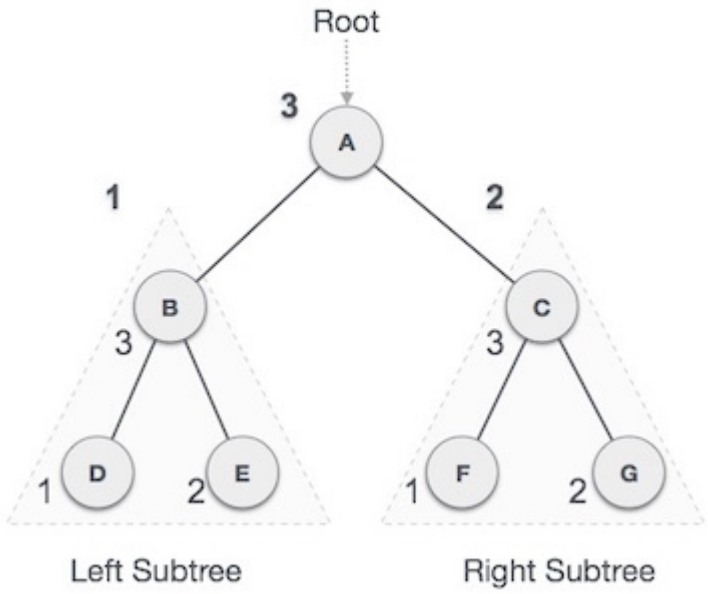
Result : A → B → D → E → C → F → G

Inorder Traversal



Result : D → B → E → A → F → C → G

Post Order Traversal



Result: D → E → B → F → G → C → A

Deletion Operation in BST

In a binary search tree, the deletion operation is performed with O(log n) time complexity.

Deleting a node from Binary search tree includes following three cases...

Case 1: Deleting a Leaf node (A node with no children)

Case 2: Deleting a node with one child

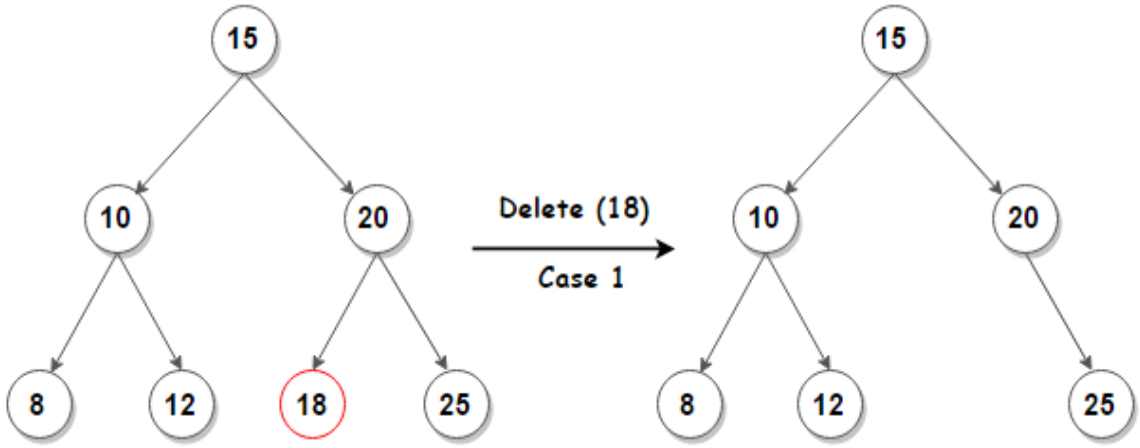
Case 3: Deleting a node with two children

Case 1: Deleting a leaf node

We use the following steps to delete a leaf node from BST...

Step 1 - Find the node to be deleted using search operation

Step 2 - Delete the node using free function (If it is a leaf) and terminate the function.



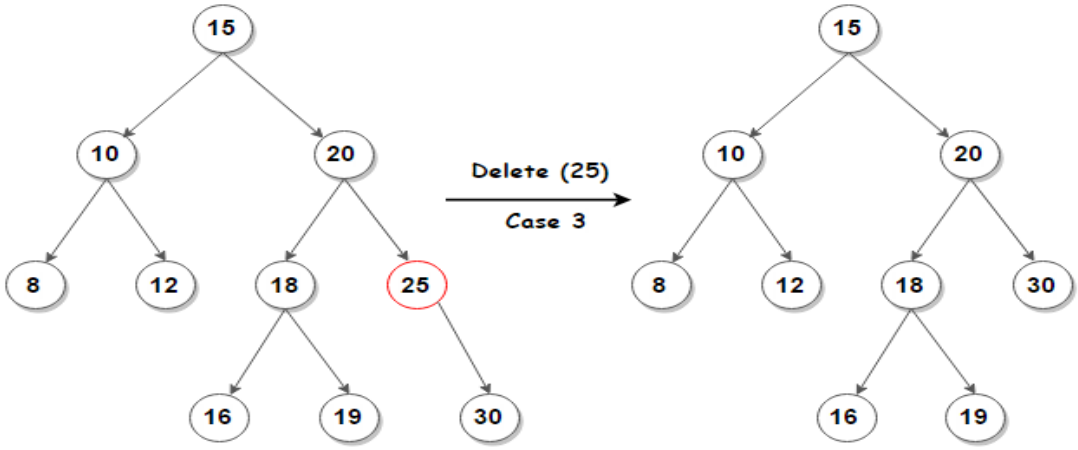
Case 2: Deleting a node with one child

We use the following steps to delete a node with one child from BST...

Step 1 - Find the node to be deleted using search operation

Step 2 - If it has only one child then create a link between its parent node and child node.

Step 3 - Delete the node using free function and terminate the function.



Case 3: Deleting a node with two children

We use the following steps to delete a node with two children from BST...

Step 1 - Find the node to be deleted using search operation

Step 2 - If it has two children, then find the largest node in its left sub tree (OR) the smallest node in its right sub tree.

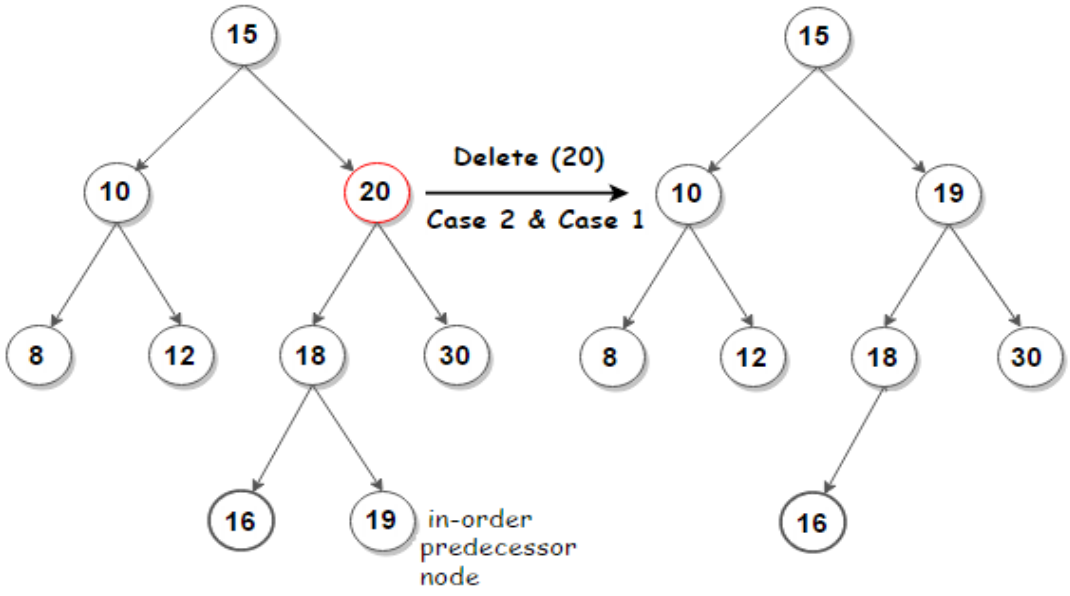
Step 3 - Swap both deleting node and node which is found in the above step.

Step 4 - Then check whether deleting node came to case 1 or case 2 or else go to step 2

Step 5 - If it comes to case 1, then delete using case 1 logic.

Step 6- If it comes to case 2, then delete using case 2 logic.

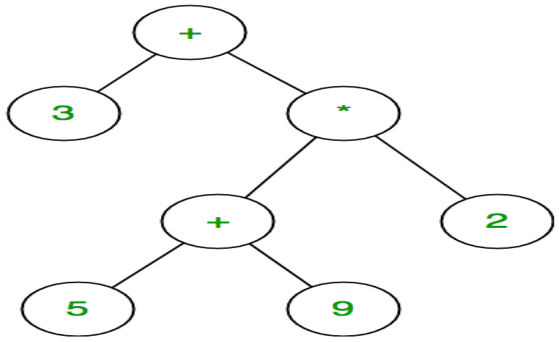
Step 7 - Repeat the same process until the node is deleted from the tree.



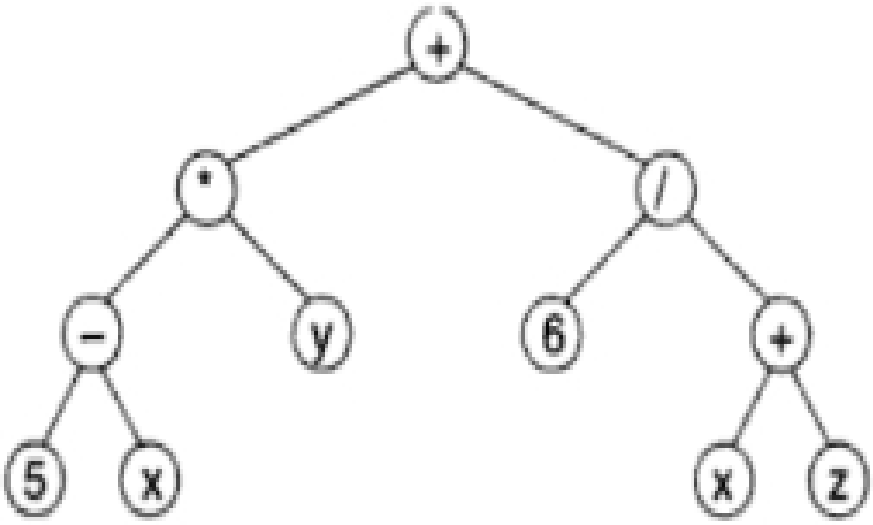
Expression Tree

Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand.

For example, expression tree for **3 + ((5+9) *2)** would be:



Example of an expression tree: (5-x)*y+6/(x + z)



Traversal

An algebraic expression can be produced from a binary expression.

The three standard depth-first traversals (Pre-order, In-order, Post-order) are representations of the three different expression formats: infix, postfix, and prefix.

An infix expression is produced by the in-order traversal.

A postfix expression is produced by the post-order traversal.

A prefix expression is produced by the pre-order traversal.

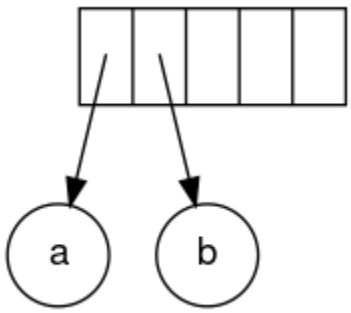
Construction of an expression tree (Postfix)

The construction of the tree takes place by reading the postfix expression one symbol at a time. If the symbol is an operand, a one-node tree is created and its pointer is pushed onto a stack. If the symbol is an operator, the pointers to two trees T1 and T2 are popped from the stack and a new tree whose root is the operator and whose left and right children point to T2 and T1 respectively is formed. A pointer to this new tree is then pushed to the Stack.

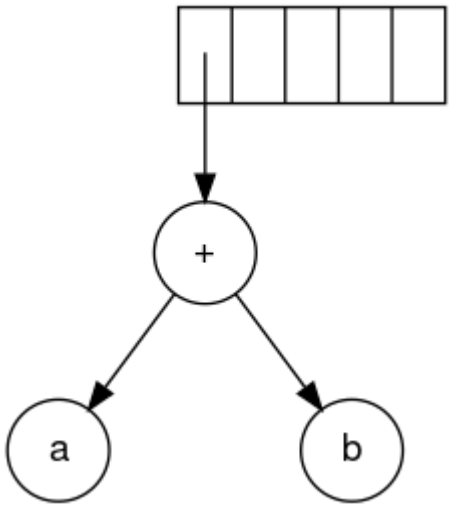
Example

The input in postfix notation is: a b + c d e + * *

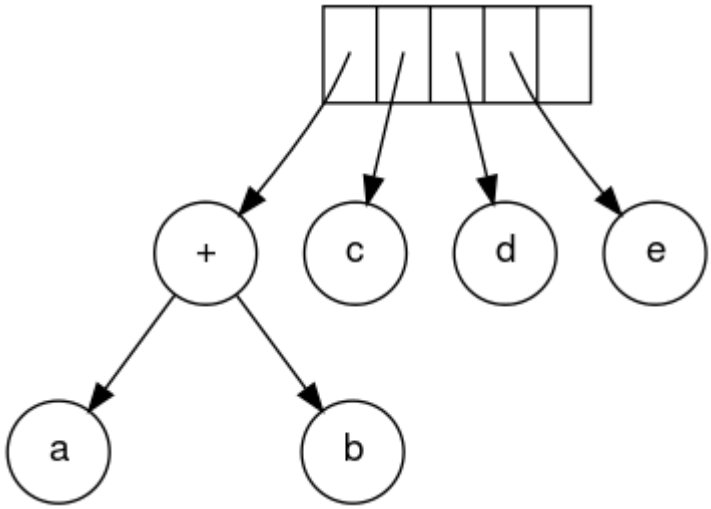
Since the first two symbols are operands, one-node trees are created and pointers to them are pushed onto a stack. For convenience the stack will grow from left to right.



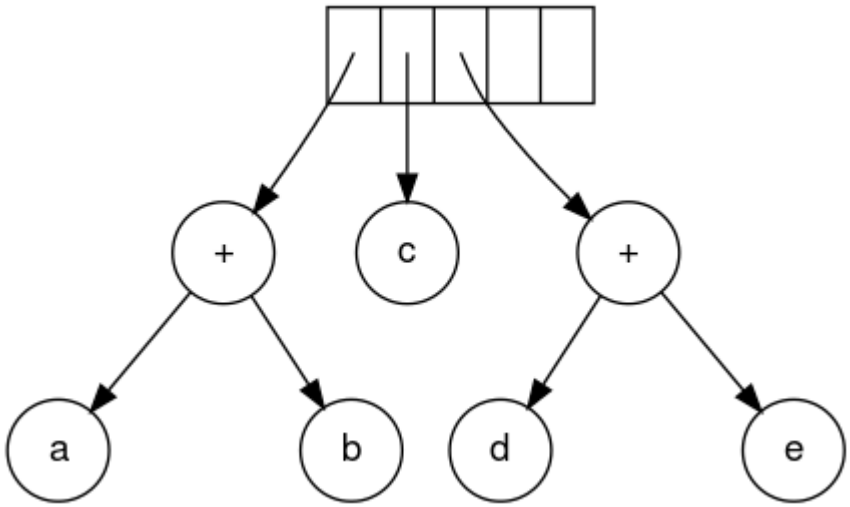
The next symbol is a '+'. It pops the two pointers to the trees, a new tree is formed, and a pointer to it is pushed onto the stack.



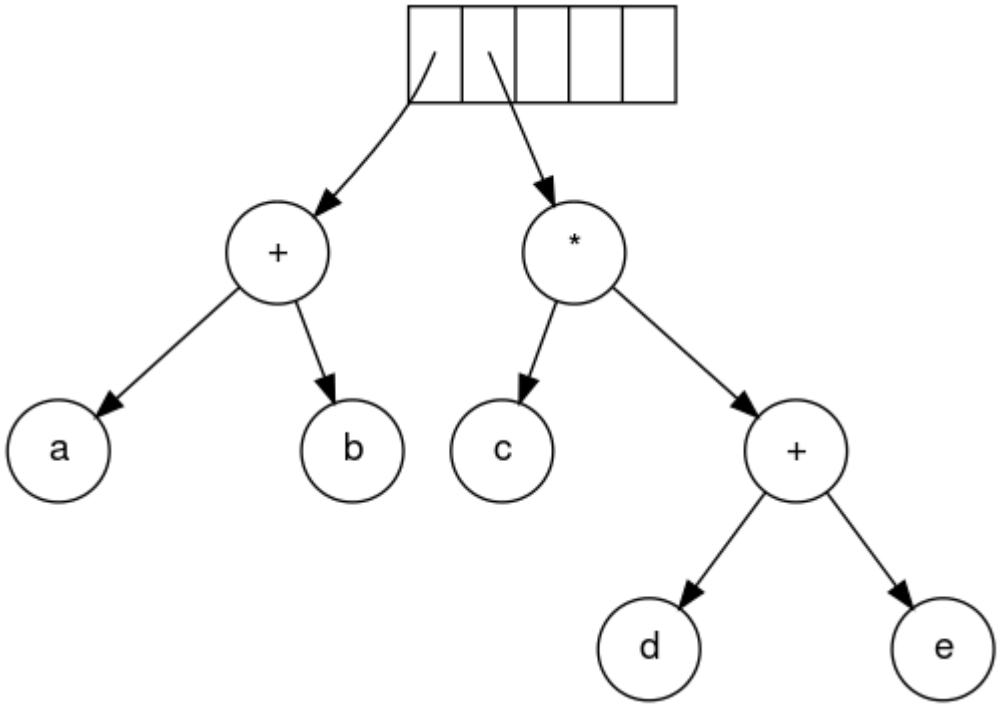
Next, c, d, and e are read. A one-node tree is created for each and a pointer to the corresponding tree is pushed onto the stack.



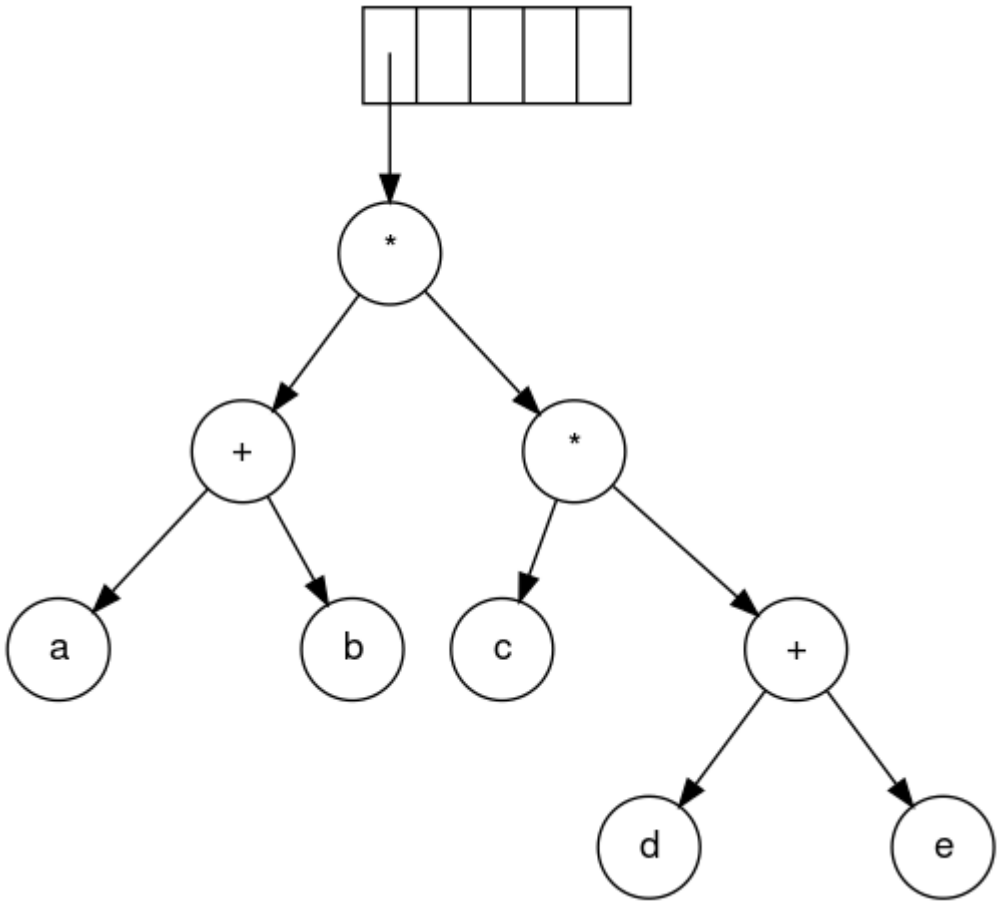
Continuing, a '+' is read, and it merges the last two trees.



Now, a '*' is read. The last two tree pointers are popped and a new tree is formed with a '*' as the root.



Finally, the last symbol “*“. The two trees are merged and a pointer to the final tree remains on the stack.


*
+
*
a
b
c
+
d
e