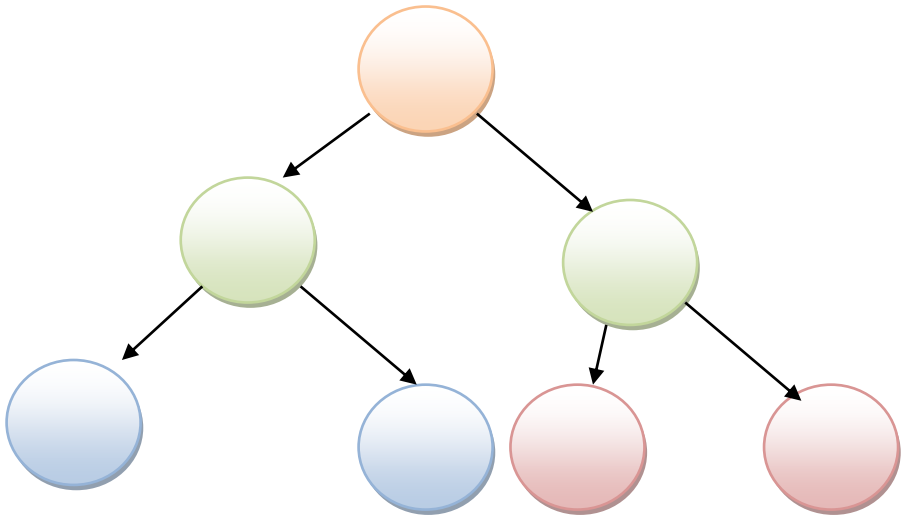
Trees

- One way to define tree is recursively. *"Tree is a non-linear data structure and it is a collection of nodes. The collection may be empty ; otherwise, a tree consists of a distinguished node called root,* ***r****, and zero or more non empty (sub) trees* T_1, T_2, ….T_k, *each of whose roots are connected by a directed edge from r".*
- In tree data structure, every individual element is called as ***Node***.
- Node in a tree data structure stores the actual data and a link to next element in hierarchical structure.



Root: Top most node in the tree.

If Root is NULL means that tree is empty.

Tree contains only one Root node.

Edge: The connecting link between any two nodes in a tree is called as **EDGE**.In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.

Parent: The node which is a *predecessor* of any node is called as PARENT NODE.

- The node which has child / children
- The node which has a branch from it to any other node is called a parent node.

The root node does not have any parent

Child:

- The node which has a link from its parent node is called as **child node**
- The node which is descendant of any node is called as **CHILD Node**
- Parent node can have any number of child nodes.
- All the nodes except root are child nodes.

Siblings:

- The nodes which belong to same Parent are called as **SIBLINGS**.
- In simple words, the nodes with the same parent are called Sibling nodes.

Leaf:

- The node which does not have a child is called as **LEAF Node**.
- In simple words, a leaf is a node with no child.

Internal Node:

- The node which has at least one child is called as **INTERNAL Nodes**.
- In simple words, an internal node is a node with at least one child.

Degree:

- The Degree of a node is the total number of children it has.

- The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'

Level:

- 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...

Height:

- The total number of edges from leaf node to a *particular node* in the longest path is called as **HEIGHT** of that *node*.
- Height of the root node is said to be **height of the tree**.
- **height of all leaf nodes is '0'.**

Depth:

- The total number of edges from root node to a *particular node* is called as **DEPTH** of that *node*
- The total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**.
- Depth of root node is 0.

Sub-Tree:

- Each child from a node forms a sub-tree recursively.
- Every child node will form a sub-tree on its parent node.

In-degree:

- It is the number of edges arriving at a node.
- The root node is the only node that has an in-degree equal to zero.

Out-degree:

- Similarly, *out-degree* of a node is the number of edges leaving that node.
- Usually, ***leaf*** nodes out-degree is ***zero***.

Binary tree:

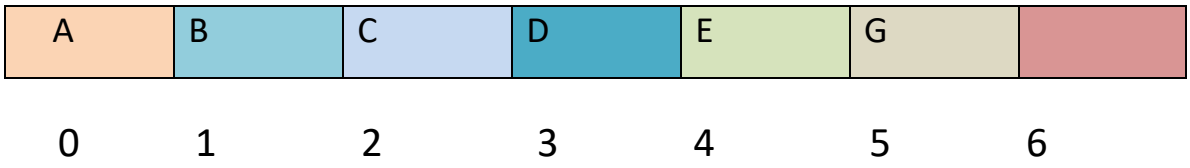
Any tree can be called as Binary Tree if every node in the tree has either zero or one or two children only (A Tree where no node has more than two children).

Types of Binary tree:

1. Strict Binary tree – It is a binary tree where each node has exactly two children or no children. A strictly binary tree with n leaves, will have (2n - 1) nodes.
2. Full Binary Tree - It is a binary tree where each non leaf node has exactly two children and all the leaf nodes are at the same level.
3. Complete Binary tree – In a complete binary tree, all the levels of a tree are filled entirely except the last level. In the last level, nodes might or might not be filled fully and also all the nodes should be filled from the left**.** The total number of nodes in a complete binary tree with depth d is 2^{d+1}-1 where leaf nodes are 2^{d} while non-leaf nodes are 2^{d}-1.

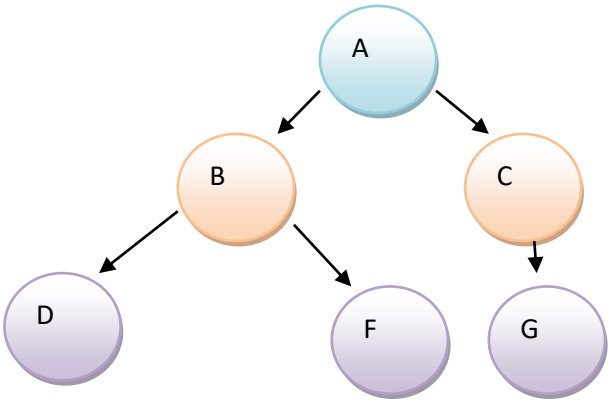
Implementation of Binary tree:

1. We can store binary trees in arrays, and specially if the tree is a complete binary tree. In this method, if a node has an index *i*, its children are found at indices 2**i*+1 and 2**i*+2, while its parent (if any) is found at index *floor((i-1)/2)* (assuming the root of the tree stored in the array at an index zero).

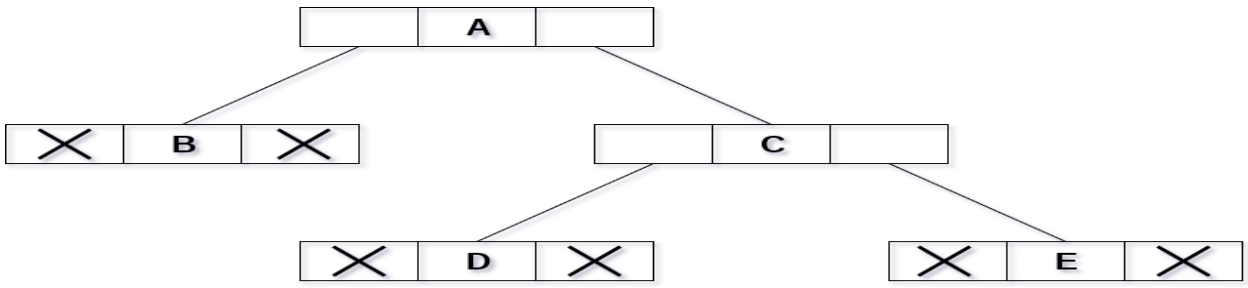


The node C is stored in location 2 and its children G is stored in location 5 (2 * 2 + 1)

The node A is stored in location 0 and its children are stored in location 1 and 2 (2 * 0 + 1 and 2 * 0 + 2)



2) In this representation, the binary tree is stored in the memory, in the form of a linked list where the number of nodes are stored at non-contiguous memory locations and linked together by inheriting parent child relationship like a tree. every node contains three parts : pointer to the left node, data element and pointer to the right node. Each binary tree has a root pointer which points to the root node of the binary tree. In an empty binary tree, the root pointer will point to null.



Binary Tree Traversal Techniques:

The problem of visiting the nodes of a tree is called *tree traversal.* There are mainly three different traversal techniques,

1. **Inorder traversal**
2. **Preorder Traversal**
3. **Postorder Traversal**

One more traversal technique called ***level order traversal*** is also used.

Inorder tree traversal:

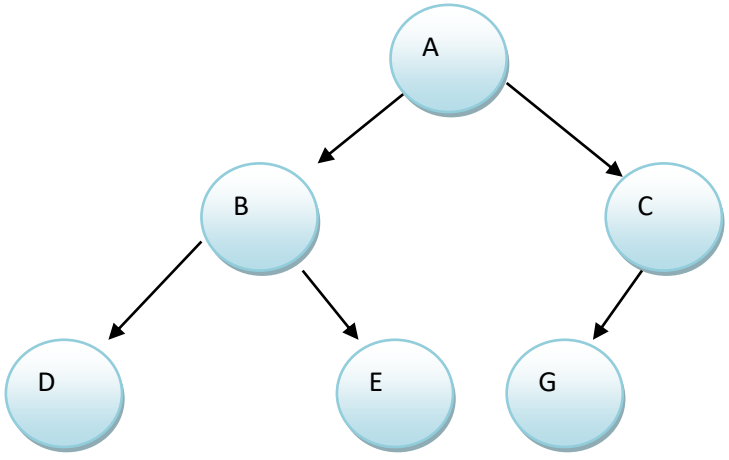
In *inorder traversal* the vising of the nodes are defined as follows

1. Traverse the Left subtree in Inorder
2. Process the root
3. Traverse the Right subtree in Inorder

The visiting order in Inorder traversal is <Left Root Right>

For example if we apply the Inorder traversal in the tree shown here results the following order:

D -> B ->E ->A ->G -> C



```
void Inorder(node *r)
{ if(r)
   {Inorder(r->left);
    output  r -> data;
    Inorder(r -> right); }
}
```

Preorder tree traversal:

In *preorder traversal* the vising of the nodes are defined as follows

1. Process the root
2. Traverse the Left subtree in preorder
3. Traverse the Right subtree in preorder

The visiting order in Inorder traversal is < Root Left Right>

For example if we apply the preorder traversal in the tree shown here results the following order:

A-> B ->D-> E -> C -> G

Postorder tree traversal:

In p*ost order traversal* the vising of the nodes are defined as follows

1. Traverse the Left subtree in postorder
2. Traverse the Right subtree in postorder
3. Process the root

The visiting order in Inorder traversal is < Left Right Root >

For example if we apply the post order traversal in the tree shown here results the following order:

D -> E -> B -> G ->C -> A

Expression Tree

A binary tree is called expression tree if the operands of the expression are stored at leaves and the operators of the expression are stored at internal nodes (non-leaf nodes).

For example for the expression ***(a + b) * (c – d)*** the equivalent expression is as follows:

An expression tree can be constructed from a postfix expression or prefix expression

The following is the procedure for constructing the expression tree from the given postfix expression:

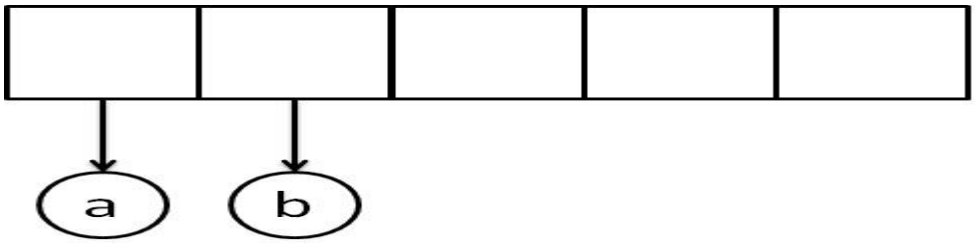
(Create a stack to store the nodes)

1. Read the each symbol from the postfix expression from left to right
2. If the symbol read is an operand then create a new node and initialize it's both left and right pointers with null. Store the operand into the node. Now push this node into the stack.

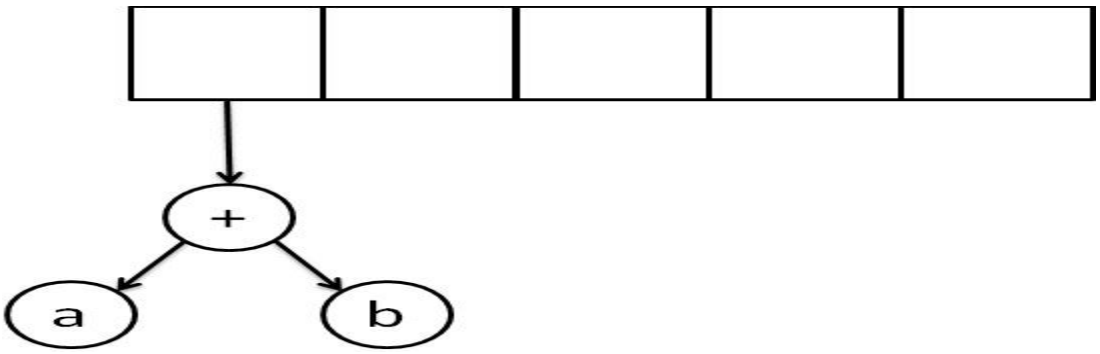
If the symbol read is an operator (assume binary operator) then pop the top two nodes from the stack, assume these are T_1 (T_1 at the top of the stack) and T_2. Now create a new node, let's call it as T_3 and store the operator in it. Store T_2 in T_3's left pointer and store T_1 in T_3's right pointer.

Input is: `a b + c *`

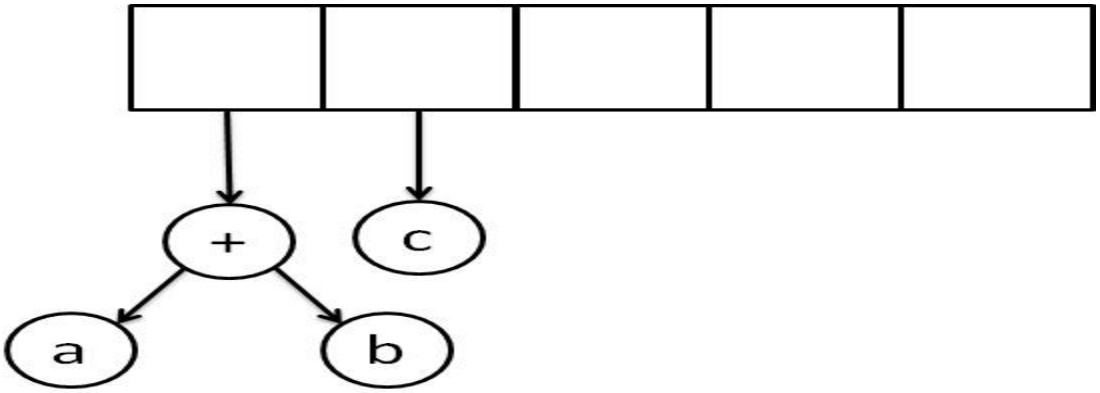
The first two symbols are operands, we create one-node tree and push a pointer to them onto the stack.



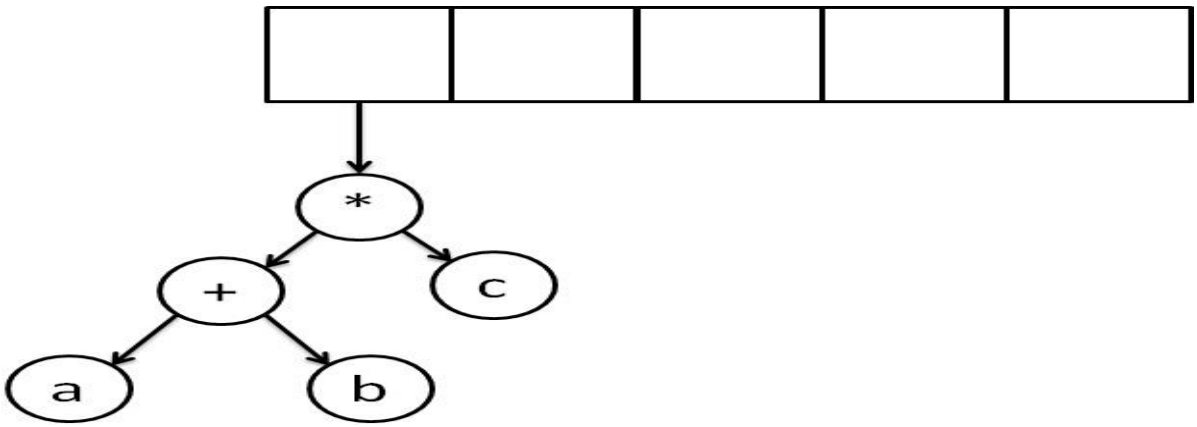
Next, read a'+' symbol, so two pointers to tree are popped. A new tree is formed and pushes a pointer to it onto the stack.



Next, '**c**' is read, we create one node tree and push a pointer to it onto the stack.



Finally, the last symbol is read ' ***** ', we pop two tree pointers and form a new tree with a, ' ***** ' as root, and a pointer to the final tree remains on the stack.

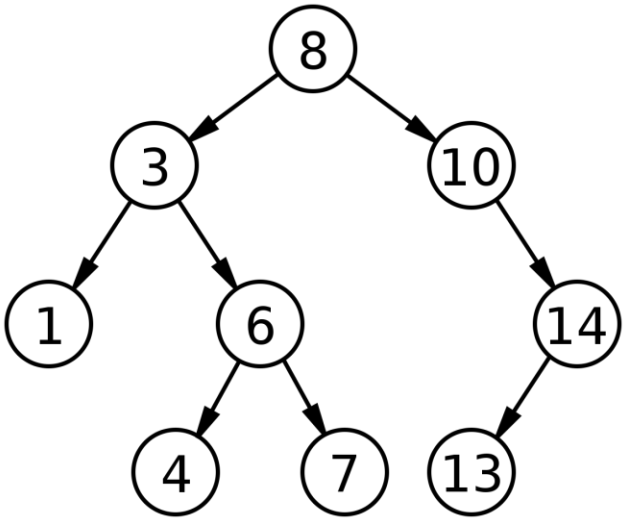


Binary Search Tree

The most widely used binary type of tree is Binary Search Tree

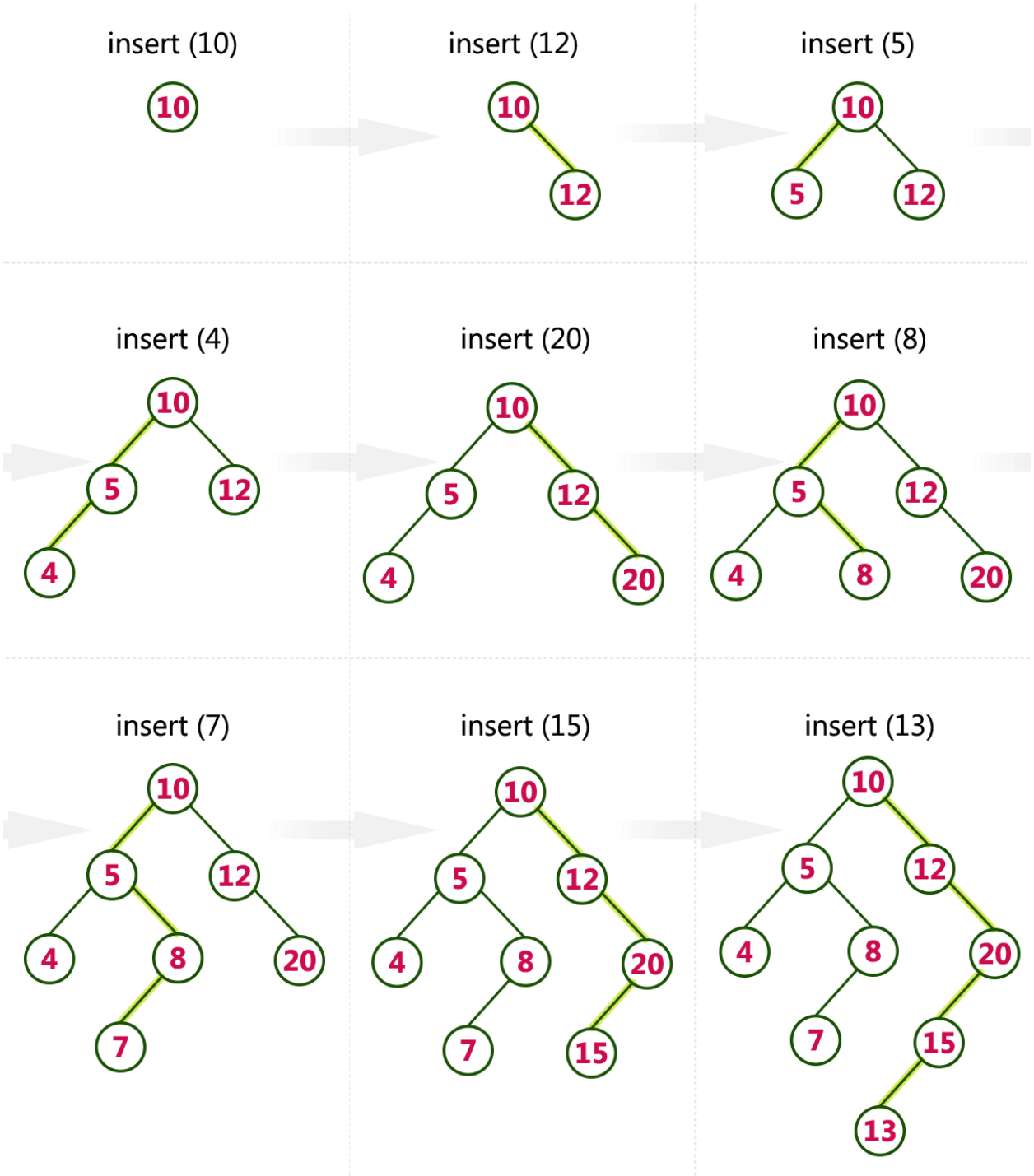
A binary tree is called binary search tree if it follows the following properties:

1. Every element has a key and no two elements have the same key'
2. The keys in the left sub tree are smaller than the key value of its root
3. The keys in the right subtree are greater than the key value of its root
4. The left and right sub trees are also binary search trees



Constructing a Binary Search Tree by inserting the following sequence of numbers.

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



Searching an element in Binary search tree:

Searching means finding or locating some specific element or node within a data structure. However, searching for some specific node in binary search tree is pretty easy due to the fact that, element in BST are stored in a particular order.

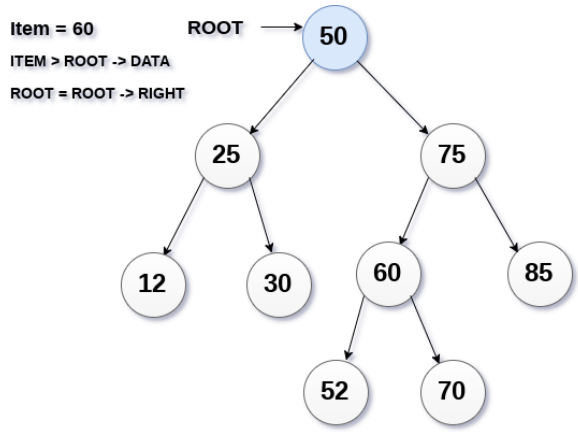
1. Compare the element with the root of the tree.
2. If the item is matched then return the location of the node.
3. Otherwise check if item is less than the element present on root, if so then move to the left sub-tree.
4. If not, then move to the right sub-tree.
5. Repeat this procedure recursively until match found.
6. If element is not found then return NULL.

Algorithm:

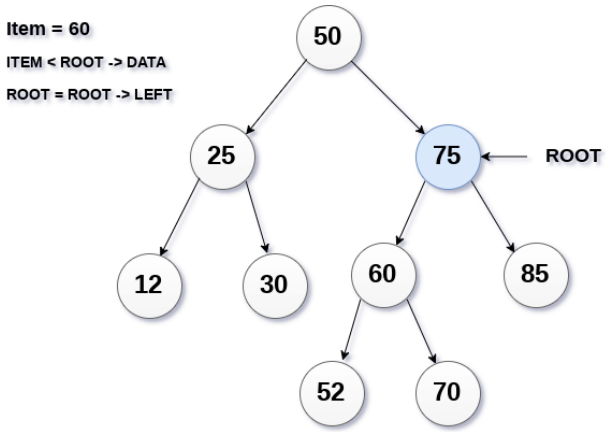
Search (ROOT, ITEM)

- **Step 1:** IF ROOT -> DATA = ITEM OR ROOT = NULL
  Return ROOT
  ELSE
  IF ROOT < ROOT -> DATA
  Return search(ROOT -> LEFT, ITEM)
  ELSE
  Return search(ROOT -> RIGHT,ITEM)
  [END OF IF]
  [END OF IF]
- **Step 2:** END
- Time Complexity (in worst case / when the tree is skewed one (All the nodes are in one side)) = O(n)
- Time Complexity (in average case) = O(log n)

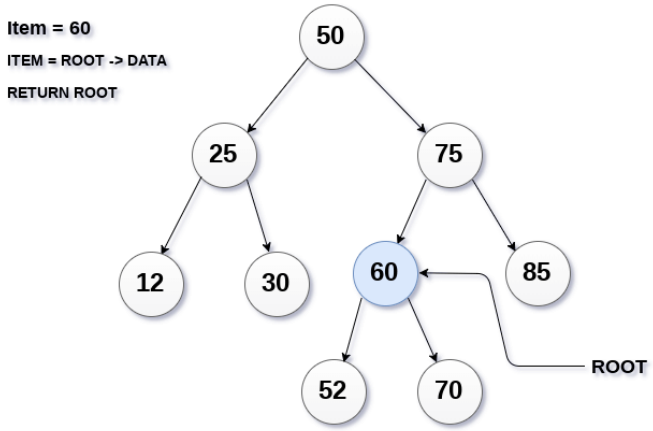
- Space complexity = O(1)



STEP 1



STEP 2



STEP 3

Inserting an element in a Binary Search Tree:

To insert a data into the binary search tree, find the location by using mechanism used in find() operations, if the element is already there then simply neglect and exit from function. Otherwise insert the data at the last location on the path traversed.

- **Step 1:** IF TREE = NULL
  Allocate memory for TREE
  SET TREE -> DATA = ITEM

```
SET TREE -> LEFT = TREE -> RIGHT = NULL
ELSE
 IF ITEM < TREE -> DATA
  Insert(TREE -> LEFT, ITEM)
ELSE
 Insert(TREE -> RIGHT, ITEM)
[END OF IF]
[END OF IF]
```

- **Step 2:** END

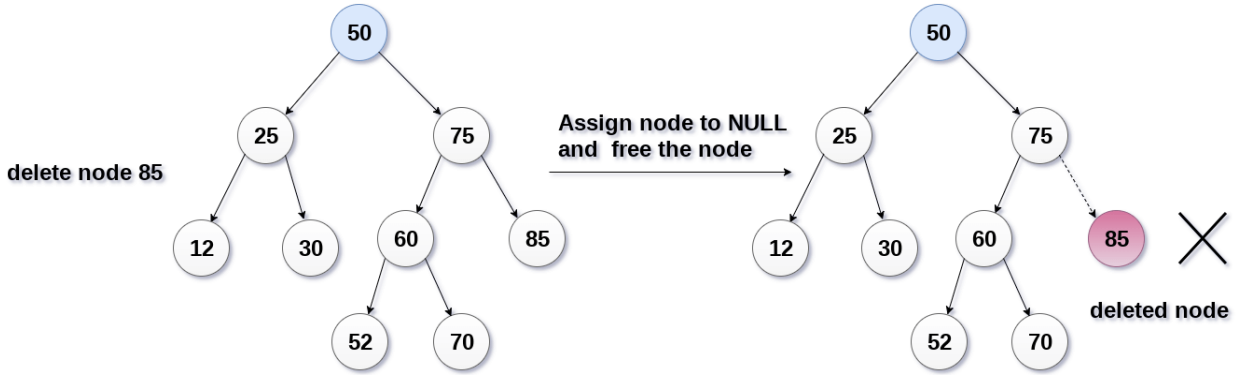
Time complexity (worst case) – O (n)

Time complexity (average case) – O (log n)

Deleting an element in a Binary Search Tree

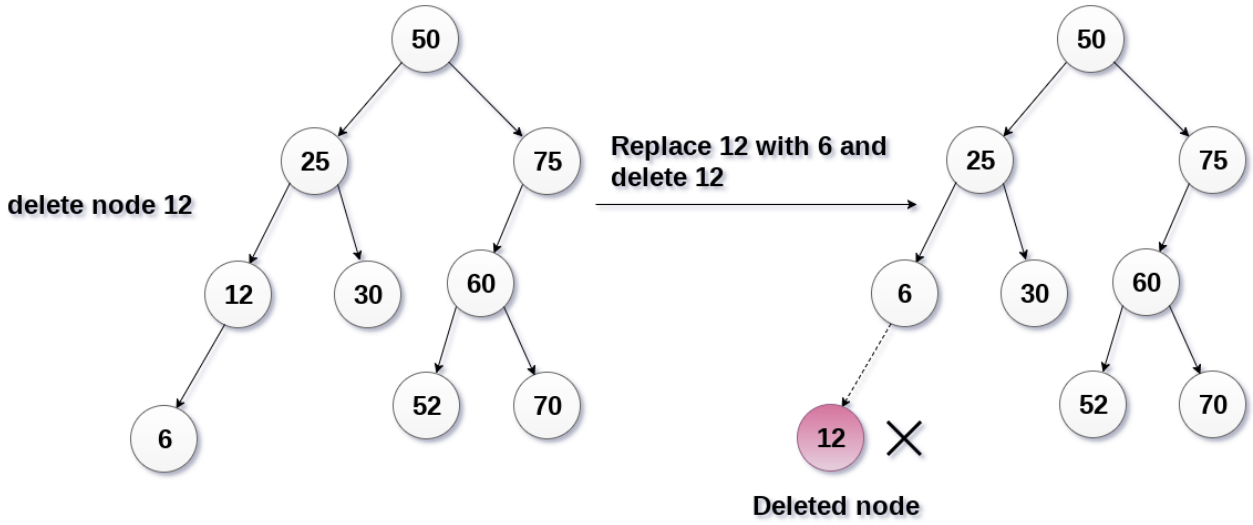
Case – 1

The deleted element is a leaf node then return NULL to its parent node.



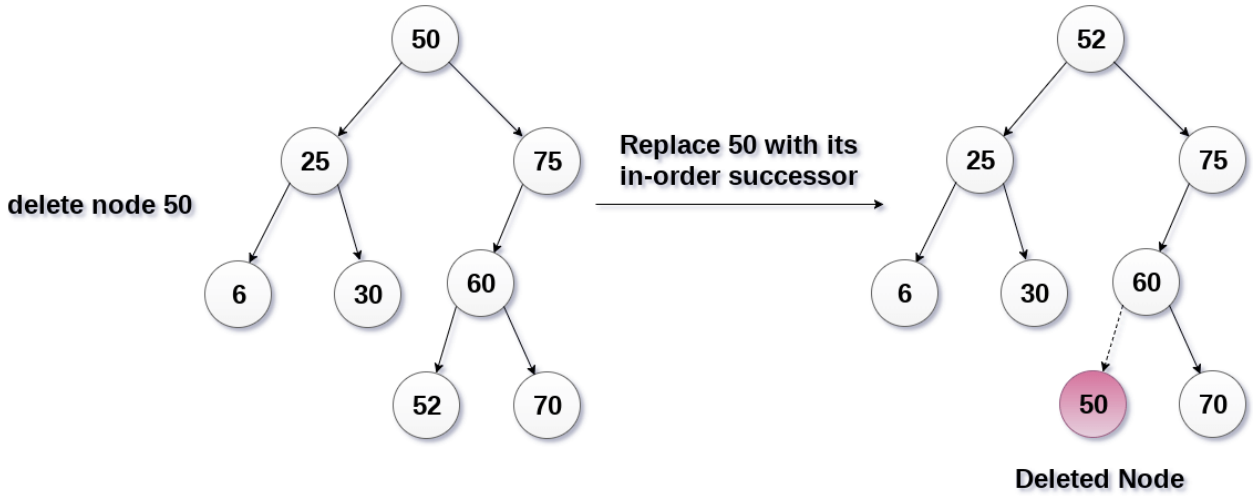
Case 2

The deleted element has one child, then send the child node to the parent of the deleted node.



Case 3

The deleted node has both left and right child, then the preferred strategy is to replace key of this node with the largest element of the left subtree and recursively delete that node.



Time complexity (worst case – if binary search tree is a skewed one) =

O (n)

Time Complexity (average case) = O (log n)

Space complexity = O(n) [stack] *// if we implement non recursive one then the space complexity will be O(1)*

AVL TREES

AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

- We have seen many data structures whose worst case time complexity for searching an element is O(n), where n is the number of elements.
- Even in the binary search trees it happens when the tree is skew one.
- We can improve the complexity of searching an element in the worst case to O(log n) using the special data structure Height Balanced trees.

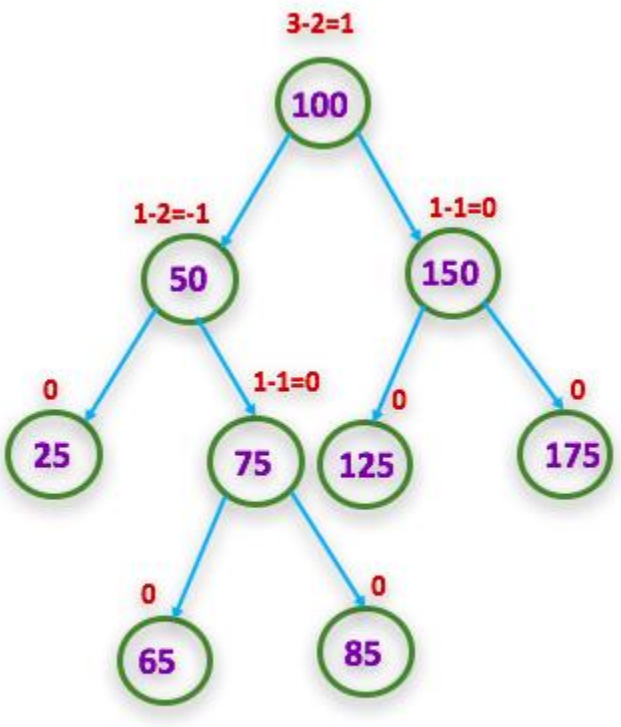
Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

Balance Factor (k) = height (left(k)) - height (right(k))

If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.

If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.

If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.



AVL Tree

AVL Rotations

We perform rotation in AVL tree only in case if Balance Factor is other than **-1, 0, and 1**. There are basically four types of rotations which are as follows:

1. L- L rotation: Inserted node is in the left subtree of left subtree of A
2. R- R rotation : Inserted node is in the right subtree of right subtree of A
3. L- R rotation : Inserted node is in the right subtree of left subtree of A

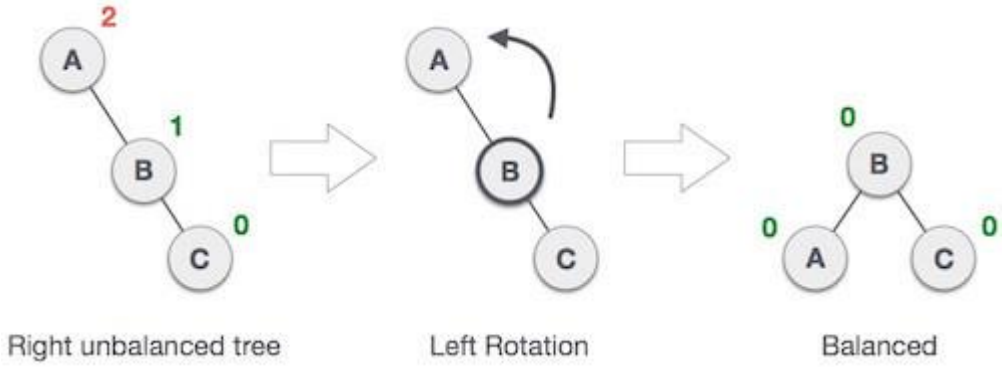
4. R- L rotation : Inserted node is in the left subtree of right subtree of A

Where node A is the node whose balance Factor is other than -1, 0, 1.

The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

1. R-R Rotation

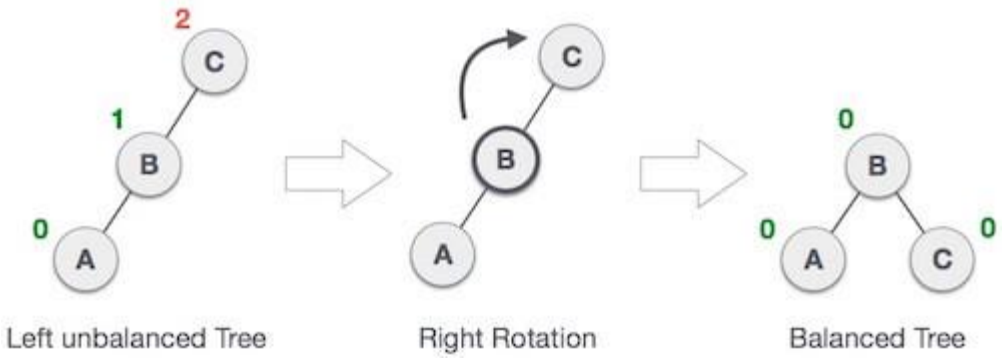
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2



In above example, node A has balance factor -2 because a node C is inserted in the right subtree of A right subtree. We perform the RR rotation on the edge below A.

2. LL Rotation

When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.



In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

3. LR Rotation

Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

4. RL Rotation

As already discussed, that double rotations are bit tougher than single rotation which has already explained above. R L rotation = LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.

Time Complexity - O(1) and Space Complexity – O(1)

Btree

B-Trees are the search trees which are not binary (BST and AVL are binary trees).

A B-Tree of order M is a tree with the following structural properties:

1. The root is either a leaf or has between 2 and M children.
2. Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
3. All leaves are at the same depth.

All the internal nodes are pointers P_1, P_2, …., P_M to the children, and values k_1, k_2, k_3, …. $k_{M-1,}$ representing the smallest key found in the subtrees P_1, P_2, ….., $P_{M,}$ respectively. Some of these pointers are NULL. All the key values in the subtree P_1 are smaller than the keys in the sub tree P_2, and so on.