# Unit 5 (The Tree)

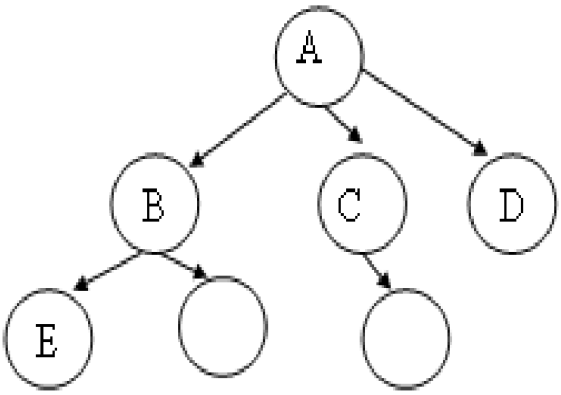
## Tree data structure:

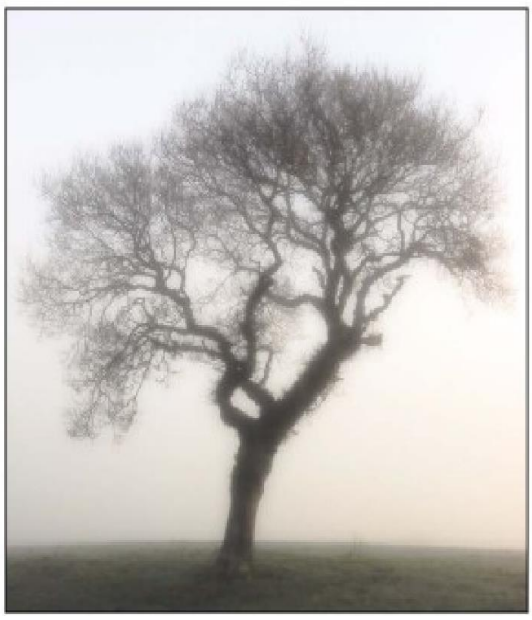
- Concept
- Operation in Binary tree
- Tree search, insertion/deletions
- Tree traversals (pre-order, post-order and in-order)
- Height, level and depth of a tree
- AVL balanced trees and Balancing algorithm
- The Huffman algorithm
- B-Tree
- Red Black Tree

## Tree:

A tree is an abstract model of a hierarchical structure that consists of nodes with a parent-child relationship.

- Tree is a sequence of nodes.
- There is a starting node known as root node.
- Every node other than the root has a parent node.
- Nodes may have any number of children.



A has 3 children, B, C, D. A is parent of B.

**Characteristics of trees:**

- Non-linear data structure
- combines advantages of an ordered array
- searching as fast as in ordered array
- insertion and deletion as fast as in linked list

**Applications of trees**

The following are the applications of trees:

- **Storing naturally hierarchical data:** Trees are used to store the data in the hierarchical structure. For example, the file system. The file system stored on the disc drive, the file and folder are in the form of the naturally hierarchical data and stored in the form of trees.
- **Organize data:** It is used to organize data for efficient insertion, deletion and searching. For example, a binary tree has a logN time for searching an element.
- **Trie:** It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.
- **Heap:** It is also a tree data structure implemented using arrays. It is used to implement priority queues.
- **B-Tree and B+Tree:** B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
- **Routing table:** The tree data structure is also used to store the data in routing tables in the routers.

## Some key terms:

**Root:** The root node is the topmost node in the tree hierarchy. In other words, the root node is the one that doesn't have any parent. In the above structure, node numbered 1 is **the root node of the tree.** If a node is directly linked to some other node, it would be called a parent-child relationship.

**Child node:** If the node is a descendant of any node, then the node is known as a child node.

**Parent:** If the node contains any sub-node, then that node is said to be the parent of that sub-node.

**Sibling:** The nodes that have the same parent are known as siblings.

**Leaf Node:-** The node of the tree, which doesn't have any child node, is called a leaf node. A leaf node is the bottom-most node of the tree. There can be any number of leaf nodes present in a general tree. Leaf nodes can also be called external nodes.

**Internal nodes:** A node has at least one child node known as an **internal**

**Ancestor node:-** An ancestor of a node is any predecessor node on a path from the root to that node. The root node doesn't have any ancestors. In the tree shown in the above image, nodes 1, 2, and 5 are the ancestors of node 10.

**Descendant:** The immediate successor of the given node is known as a descendant of a node. In the above figure, 10 is the descendant of node 5.

**Degree of a node:**

The degree of a node is the number of children of that node. In above tree the degree of node A is 3.

**Degree of a Tree:**

The degree of a tree is the maximum degree of nodes in a given tree. In the above tree the node A has maximum degree, thus the degree of the tree is 3.

**Path:**

It is the sequence of consecutive edges from source node to destination node. There is a single unique path from the root to any node.

**Height of a node:**

The height of a node is the maximum path length from that node to a leaf node. A leaf node has a height of 0.

**Height of a tree:**

The height of a tree is the height of the root.

**Depth of a node:**

Depth of a node is the path length from the root to that node. The root node has a depth of 0.

**Depth of a tree:**

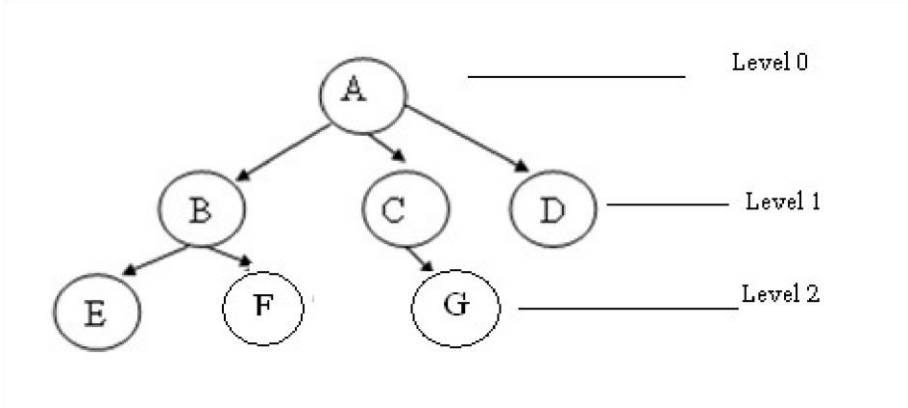
Depth of a tree is the maximum level of any leaf in the tree. This is equal to the longest path from the root to any leaf.

**Compiled by: Er. Sandesh S Poudel**

**Level of a node:**

The level of a node is 0, if it is root; otherwise it is one more then its parent.

**Illustration:**

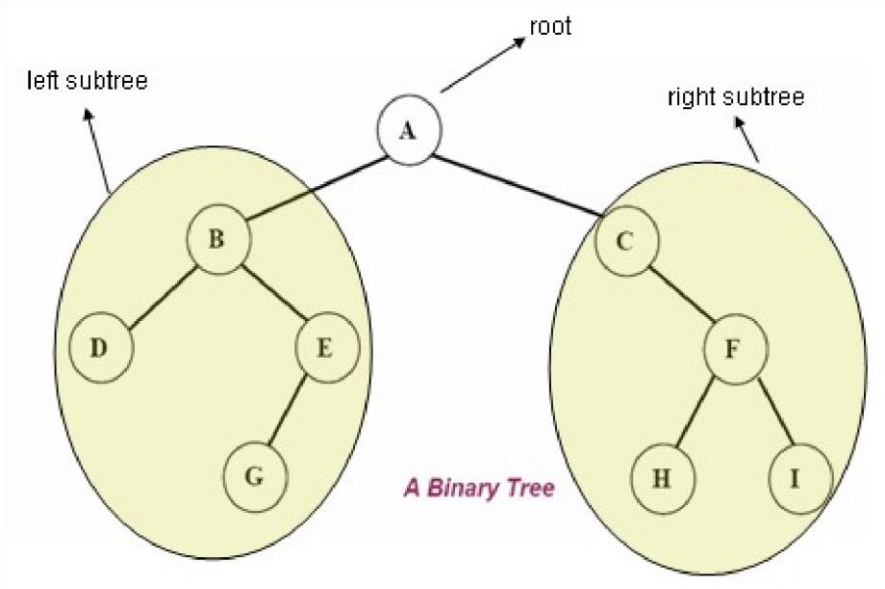



- A is the root node
- B is the parent of E and F
- D is the sibling of B and C
- E and F are children of B
- E, F, G, D are external nodes or leaves
- A, B, C are internal nodes
- Depth of F is 2
- the height of tree is 2
- the degree of node A is 3
- The degree of tree is 3

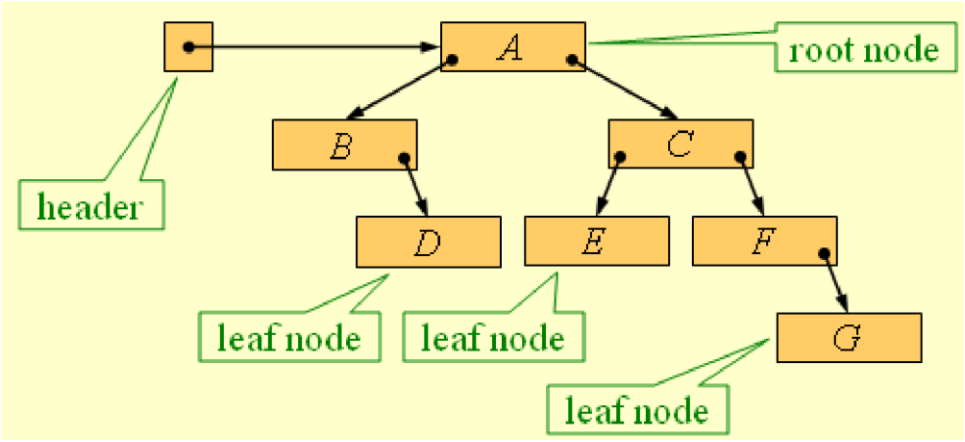
# Binary Trees:

A binary tree is a finite set of elements that are either empty or is partitioned into three disjoint subsets. The first subset contains a single element called the **root** of the tree. The other two subsets are themselves binary trees called the **left** and **right sub-trees** of the original tree. A left or right sub tree can be empty.

Each element of a binary tree is called a **node** of the tree. The following figure shows a binary tree with 9 nodes where A is the root.




•A **binary tree** consists of a **header**, plus a number of **nodes** connected by **links** in a hierarchical data structure:




## Binary tree properties:

- If a binary tree contains m nodes at level l, it contains at most **2m** nodes at level **l+1**.
- Since a binary tree can contain at most 1 node at level 0 (the rot), it contains at most $\mathbf{2^l}$ nodes at level **l.**

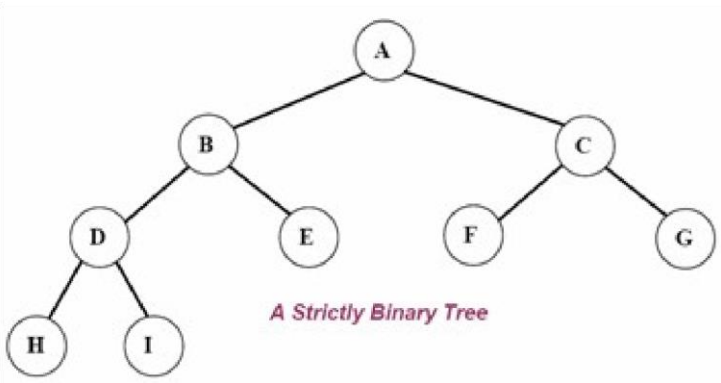
## Types of binary tree

- Complete binary tree
- Strictly binary tree
- Almost complete binary tree

**Compiled by: Er. Sandesh S Poudel**

## Strictly binary tree:

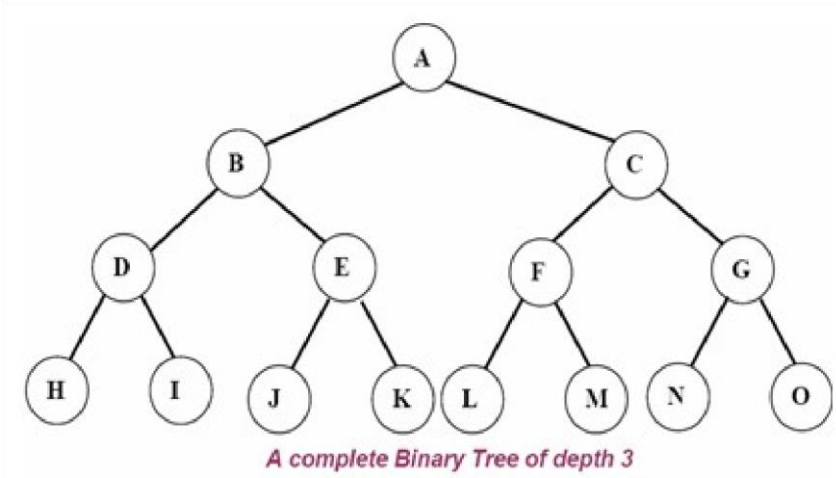
If every non-leaf node in a binary tree has nonempty left and right sub-trees, then such a tree is called a **strictly binary tree**.



A Strictly Binary Tree

## Complete binary tree:

A **complete binary tree** of depth d is called strictly binary tree if all of whose leaves are at level **d**. A complete binary tree with depth **d** has $2^{d}$ leaves and $2^{d}$ **-1** non-leaf nodes(internal)



A complete Binary Tree of depth 3

## Almost complete binary tree:

A binary tree of depth **d** is an almost complete binary tree if:

- Any node **nd** at level less than **d-1** has two sons.
- For any node **nd** in the tree with a right descendant at level **d, nd** must have a left son and every left descendant of **nd** is either a leaf at level **d** or has two sons.

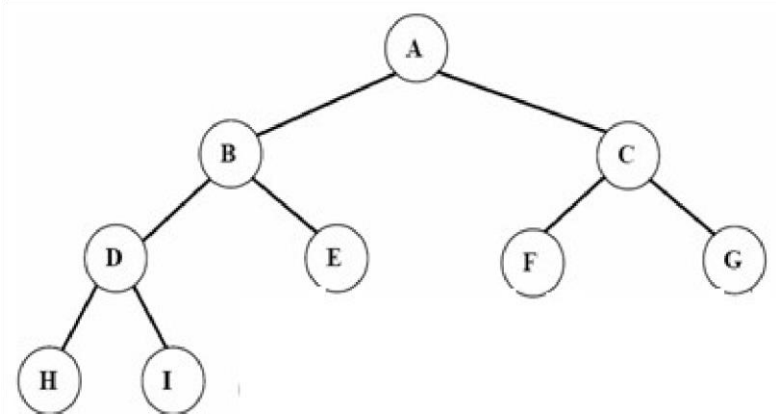



Fig Almost complete binary tree.

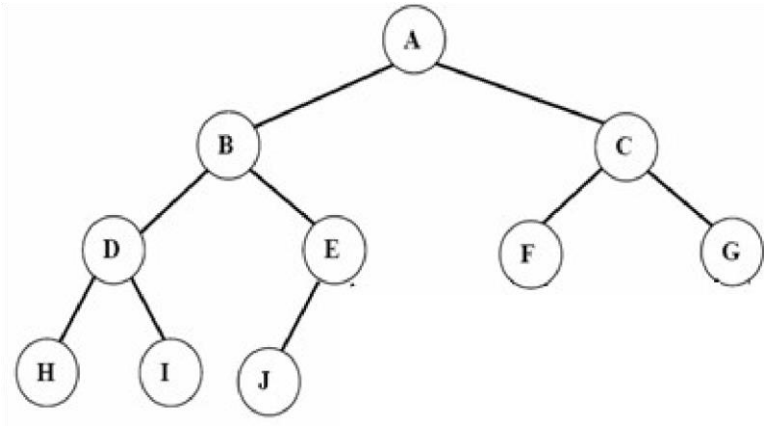



Fig Almost complete binary tree but not strictly binary tree.

Since node E has a left son but not a right son.

## Operations on Binary tree:

- **father(n,T):**Return the parent node of the node n in tree T. If n is the root, NULL is returned.
- **LeftChild(n,T):**Return the left child of node n in tree T. Return NULL if n does not have a left child.
- **RightChild(n,T)**:Return the right child of node n in tree T. Return NULL if n does not have a right child.
- **Info(n,T):** Return information stored in node n of tree T (ie. Content of a node).
- **Sibling(n,T):** return the sibling node of node **n** in tree T. Return NULL if n has no sibling.
- **Root(T):** Return root node of a tree if and only if the tree is nonempty.
- **Size(T):** Return the number of nodes in tree T
- **MakeEmpty(T):** Create an empty tree T
- **SetLeft(S,T):** Attach the tree S as the left sub-tree of tree T
- **SetRight(S,T):** Attach the tree S as the right sub-tree of tree T.
- **Preorder(T):** Traverses all the nodes of tree T in pre-order.

- **postorder(T):** Traverses all the nodes of tree T in post-order
- **Inorder(T):** Traverses all the nodes of tree T in in-order.

## C representation for Binary tree:

```
struct bnode
    {
        int info;
        struct bnode  *left;
        struct bnode  *right;
    };
    struct bnode *root=NULL;
```

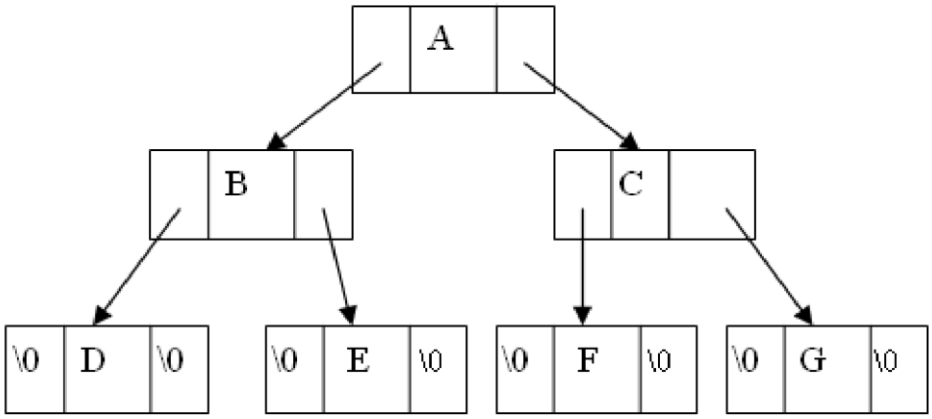



Fig: Structure of Binary tree

## Tree traversal:

The tree traversal is a way in which each node in the tree is visited exactly once in a symmetric manner.

There are three popular methods of traversal

- Pre-order traversal
- In-order traversal
- Post-order traversal

## Pre-order traversal:

The preorder traversal of a nonempty binary tree is defined as follows:

- Visit the root node
- Traverse the left sub-tree in preorder

- Traverse the right sub-tree in preorder

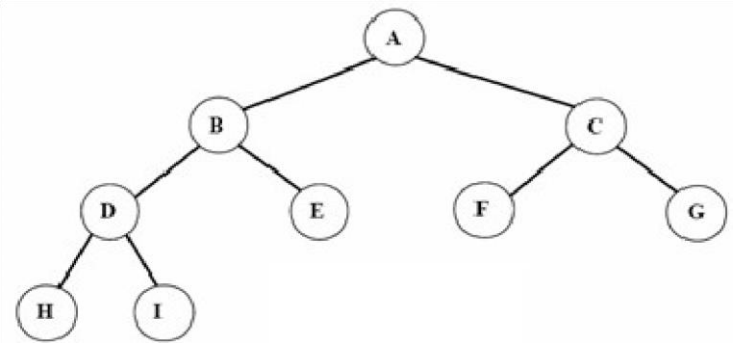



Fig: Binary tree

The preorder traversal output of the given tree is: A B D H I E C F G

The preorder is also known as depth first order.

### C function for preorder traversing:

```
void preorder(struct bnode *root)
{
        if(root!=NULL)
        {
            printf("%c", root->info);
            preorder(root->left);
            preorder(root->right);
        }
}
```

## In-order traversal:

The inorder traversal of a nonempty binary tree is defined as follows:

- Traverse the left sub-tree in inorder
- Visit the root node
- Traverse the right sub-tree in inorder

The inorder traversal output of the given tree is: H D I B E A F C G

### C function for inorder traversing:

```
        void inorder(struct bnode *root)
        {
```

```
        if(root!=NULL)
        {
            inorder(root->left);
            printf("%c", root->info);
            inorder(root->right);
        }
    }
```

## Post-order traversal:

The post-order traversal of a nonempty binary tree is defined as follows:

- Traverse the left sub-tree in post-order
- Traverse the right sub-tree in post-order
- Visit the root node

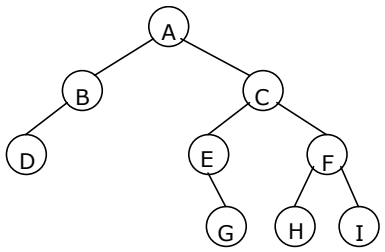
The post-order traversal output of the given tree is: H I D E B F G C A

### C function for post-order traversing:

```
    void post-order(struct bnode *root)
{
        if(root!=NULL)
        {
            post-order(root->left);
            post-order(root->right);
            printf("%c", root->info);
        }
}
```

**Example 1:**

**Traverse the following binary tree in pre, post, inorder.**

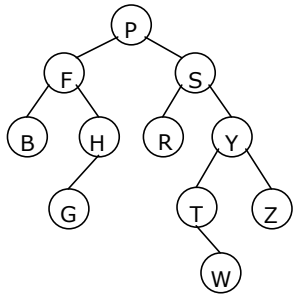



- Preorder traversal yields:
  A, B, D, C, E, G, F, H, I
- Postorder traversal yields:
  D, B, G, E, H, I, F, C, A
- Inorder traversal yields:
  D, B, A, E, G, C, H, F, I

**Binary Tree Pre, Post, Inorder Traversing**

**Example 2:**

Traverse the following binary tree in pre, post, inorder.




- Preorder traversal yields:
  P, F, B, H, G, S, R, Y, T, W, Z
- Postorder traversal yields:
  B, G, H, F, R, W, T, Z, Y, S, P
- Inorder traversal yields:
  B, F, G, H, P, R, S, T, W, Y, Z

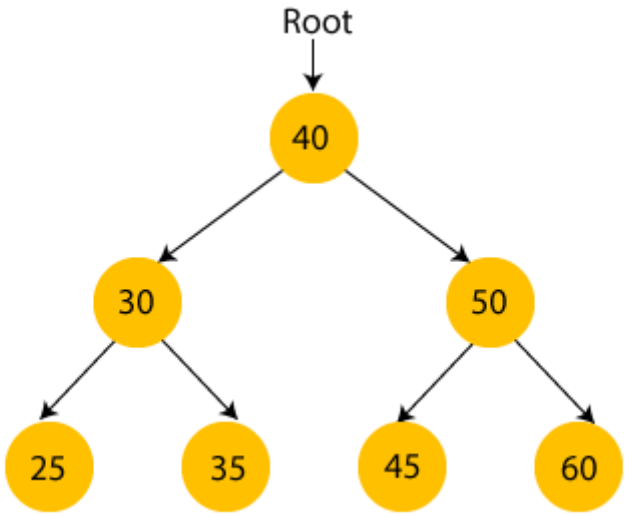
Binary Tree Pre, Post, Inorder and level order Traversing

## Binary search tree(BST):

A binary search tree (BST) is a binary tree that is either empty or in which every node contains a key (value) ans satisfies the following conditions:

- All keys in the left sub-tree o the root are smaller than the key in the root node

- All keys in the right sub-tree of the root are greater than the key in the root node
- The left and right sub-trees of the root are again binary search trees




**Example of creating a binary search tree**

Now, let's see the creation of binary search tree using an example.

Suppose the data elements are **- 45, 15, 79, 90, 10, 55, 12, 20, 50**

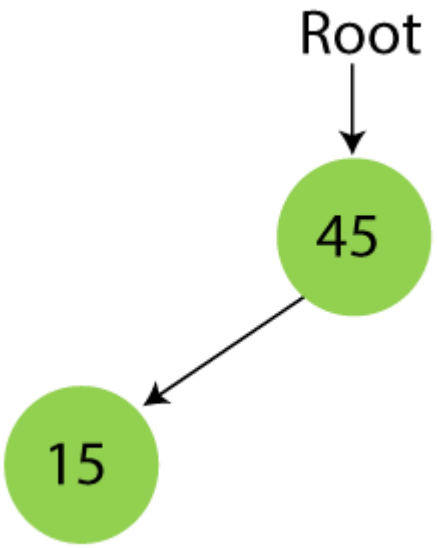
- First, we have to insert **45** into the tree as the root of the tree.
- Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below -

**Step 1 - Insert 45.**

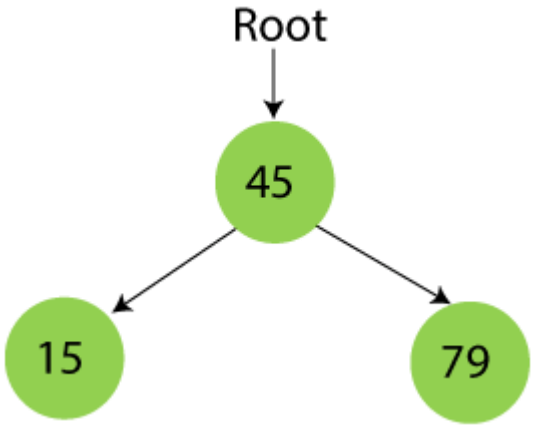
**Step 2 - Insert 15.**

As 15 is smaller than 45, so insert it as the root node of the left subtree.



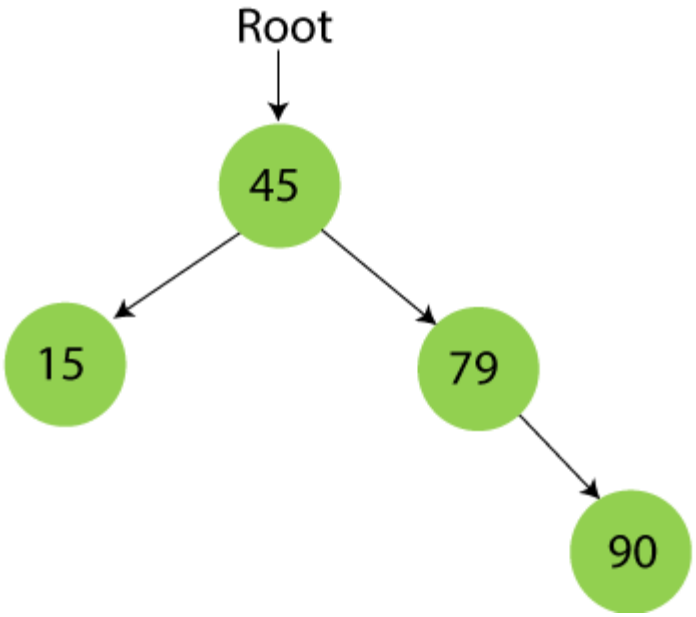

**Step 3 - Insert 79.**

As 79 is greater than 45, so insert it as the root node of the right subtree.



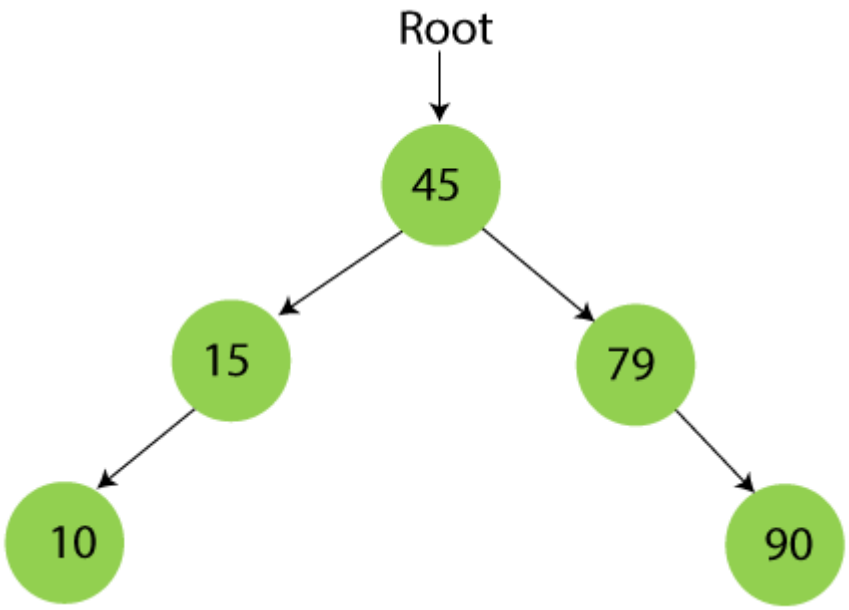

**Step 4 - Insert 90.**

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



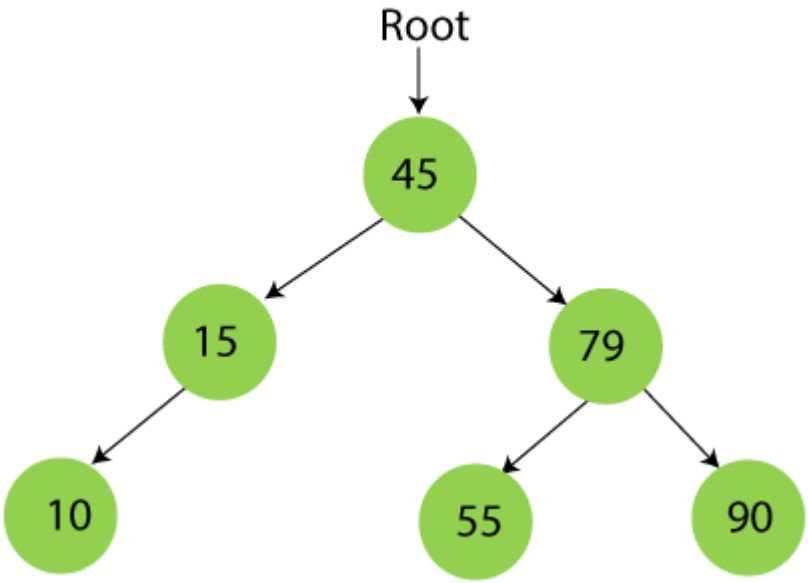

**Step 5 - Insert 10.**

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



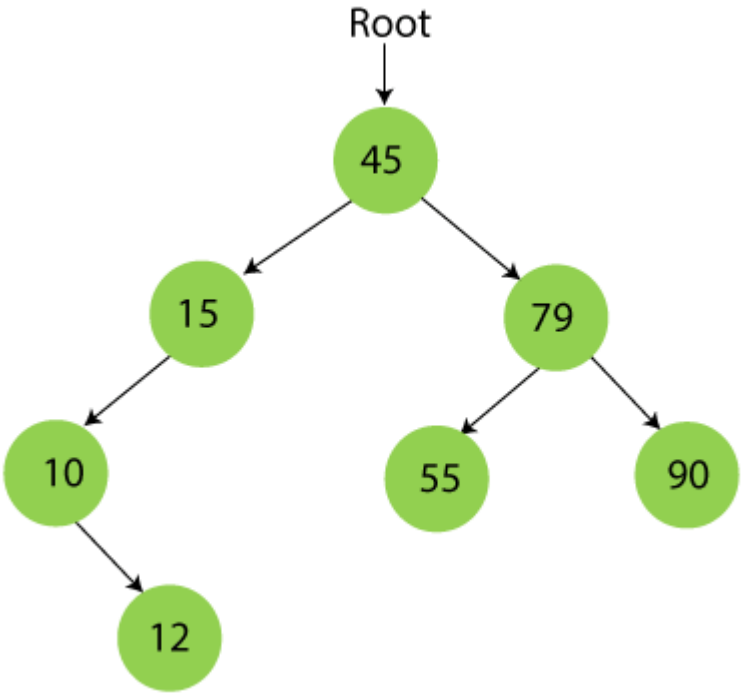

**Step 6 - Insert 55.**

55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



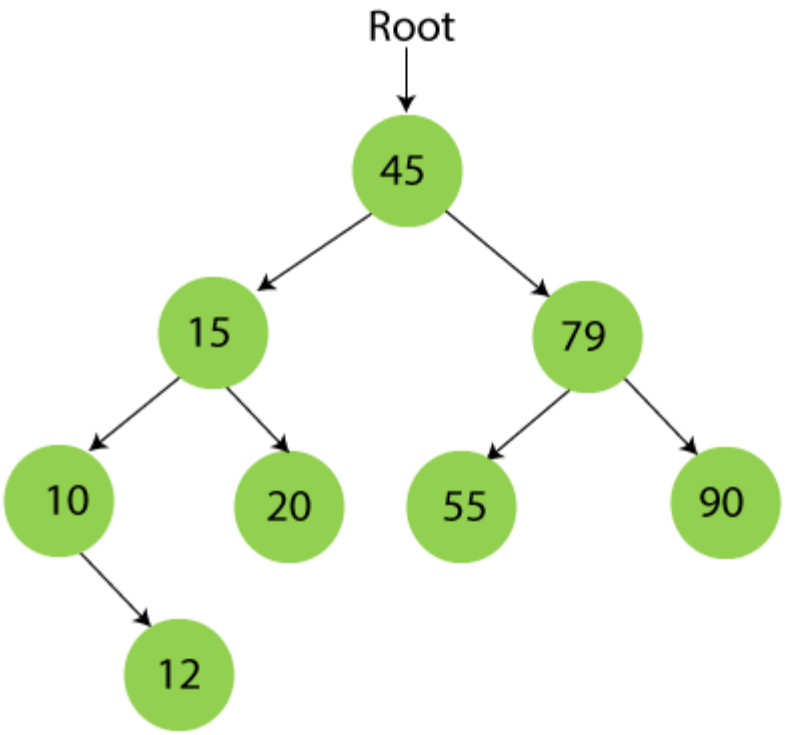

**Step 7 - Insert 12.**

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



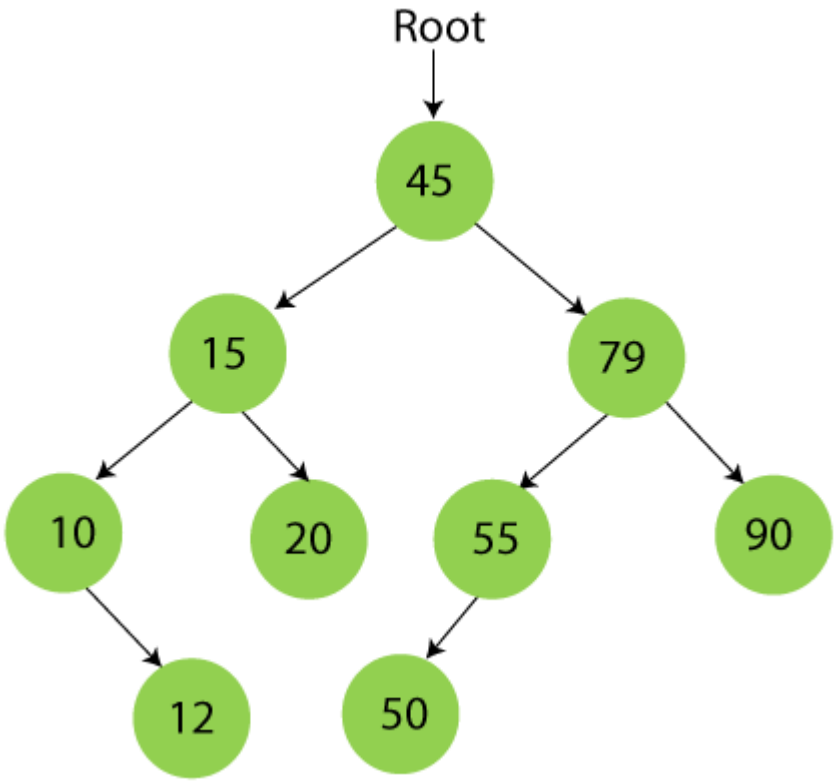

**Step 8 - Insert 20.**

20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.




**Step 9 - Insert 50.**

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.



## Operations on Binary search tree(BST):

Following operations can be done in BST:

- **Search(k, T):** Search for key k in the tree T. If k is found in some node of tree then return true otherwise return false.
- **Insert(k, T):** Insert a new node with value k in the info field in the tree T such that the property of BST is maintained.
- **Delete(k, T):**Delete a node with value k in the info field from the tree T such that the property of BST is maintained.
- **FindMin(T), FindMax(T):** Find minimum and maximum element from the given nonempty BST.

## Searching through the BST:

- **Problem: Search for a given target value in a BST.**
- **Idea: Compare the target value with the element in the root node.**
  - If the target value is **equal**, the search is successful.
  - If target value is **less**, search the left subtree.
  - If target value is **greater**, search the right subtree.
  - If the subtree is **empty**, the search is unsuccessful.

## BST search algorithm:

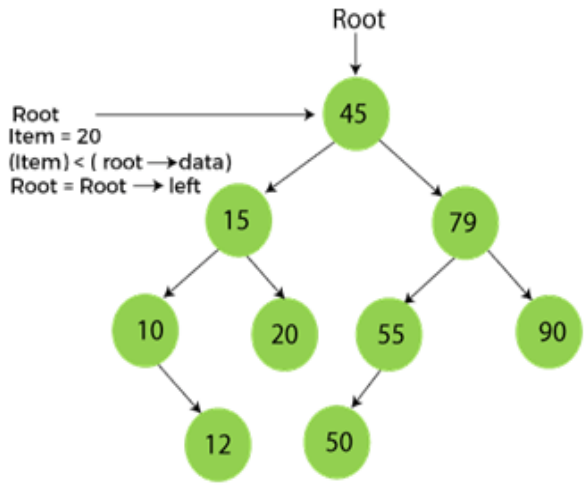
To find which if any node of a BST contains an element equal to target:

1. Set curr (current pointer -root) to the BST's root.
2. Repeat:
    2.1. If curr is null:
        2.1.1. Terminate with answer none.
    2.2. Otherwise, if target is equal to curr's element:
        2.2.1. Terminate with answer curr.
    2.3. Otherwise, if target is less than curr's element:
        2.3.1. Set curr to curr's left child.
    2.4. Otherwise, if target is greater than curr's element:
        2.4.1. Set curr to curr's right child.

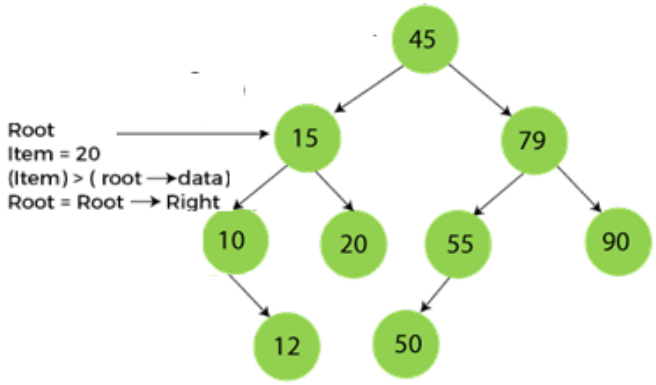
2. end

Now, let's understand the searching in binary tree using an example. We are taking the binary search tree formed above. Suppose we have to find node 20 from the below tree.

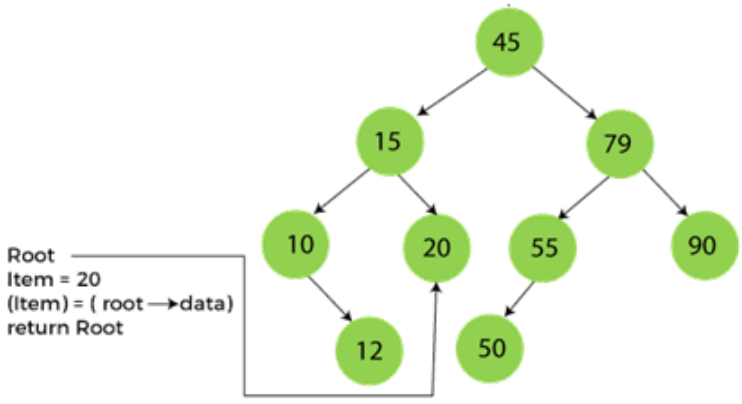
**Step1:**




**Step2:**




**Step3:**



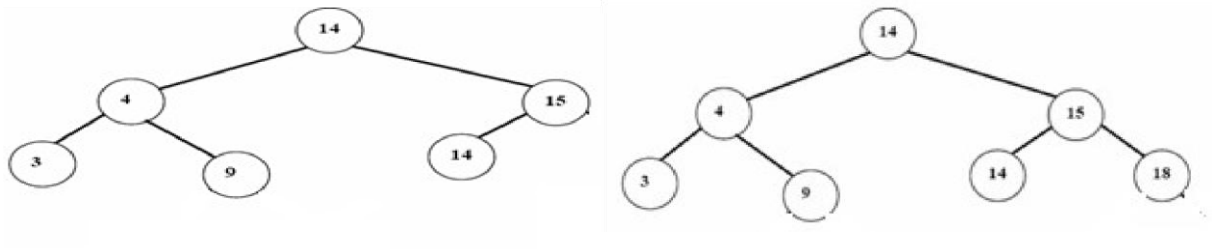
## C function for BST searching:

```
void BinSearch(struct bnode *root , int key)
    {
        if(root == NULL)
          {
              printf("The number does not exist"); exit(1);
          }
        else  if (key == root->info)
          {
              printf("The searched item is found"):
          }
        else if(key < root->info)

              return BinSearch(root->left, key);

        else

              return BinSearch(root->right, key);

    }
```

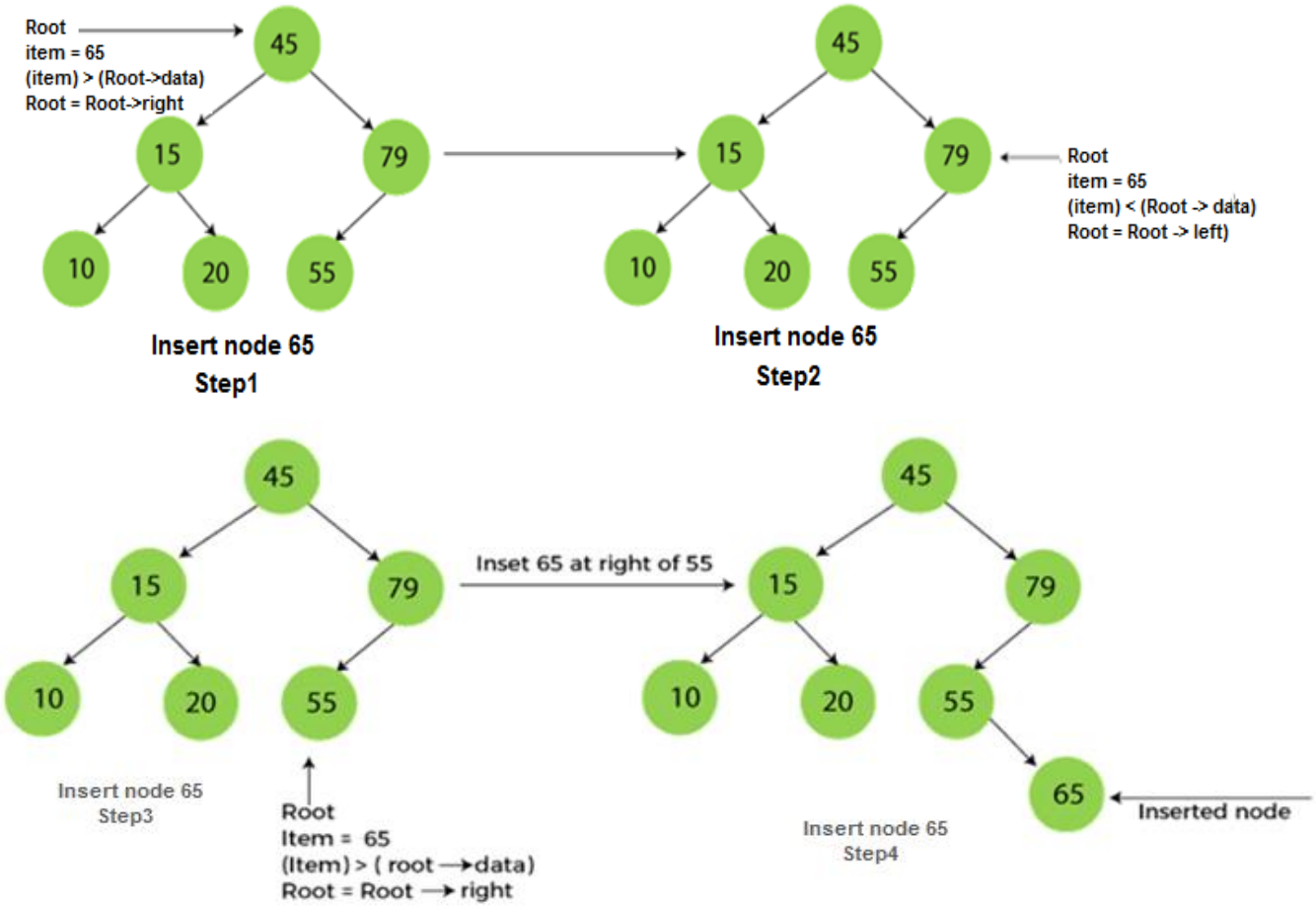
## Insertion of a node in BST:

To insert a new item in a tree, we must first verify that its key is different from those of existing elements. To do this a search is carried out. If the search is unsuccessful, then item is inserted.

•**Idea**: To insert a new element into a BST, proceed as if searching for that element. If the element is not already present, the search will lead to a null link. Replace that null link by a link to a leaf node containing the new element.




insert(18)




## BST insertion algorithm:

To insert the element elem into a BST:

1. Set parent to null, and set curr(current node-root) to the BST's root.
2. Repeat:
   - 2.1. If curr is null:
     - 2.1.1. Replace the null link from which curr was taken (either the BST's root or parent's left child or parent's right child) by a link to a newly-created leaf node with element elem.
     - 2.1.2. Terminate.
   - 2.2. Otherwise, if elem is equal to curr's element:
     - 2.2.1. Terminate.
   - 2.3. Otherwise, if elem is less than curr's element:
     - 2.3.1. Set parent to curr, and set curr to curr's left child.
   - 2.4. Otherwise, if elem is greater than curr's element:

2.4.1. Set parent to curr, and set curr to curr's right child.

3. End

### C function for BST insertion:

```
void insert(struct bnode *root, int item)
{
        if(root=NULL)
        {
                root=(struct bnode*)malloc (sizeof(struct bnode));
                root->left=root->right=NULL;
                root->info=item;
        }
        else
        {
          if(item<root->info)
                root->left=insert(root->left, item);
          else
                root->right=insert(root->right, item);
        }
}
```

## Deletion in Binary Search tree

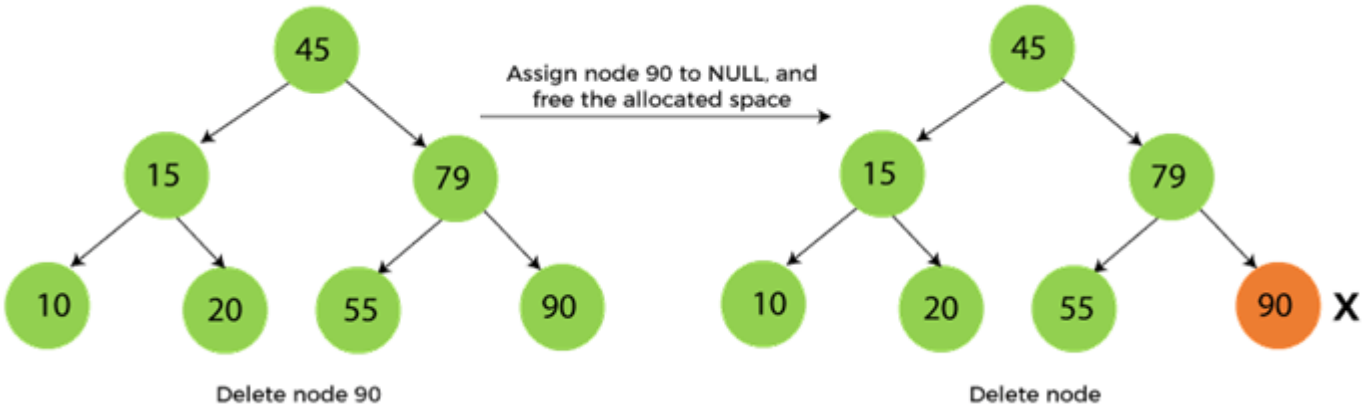
In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated. To delete a node from BST, there are three possible situations occur -

- The node to be deleted is the leaf node, or,
- The node to be deleted has only one child, and,
- The node to be deleted has two children

**When the node to be deleted is the leaf node**

It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.

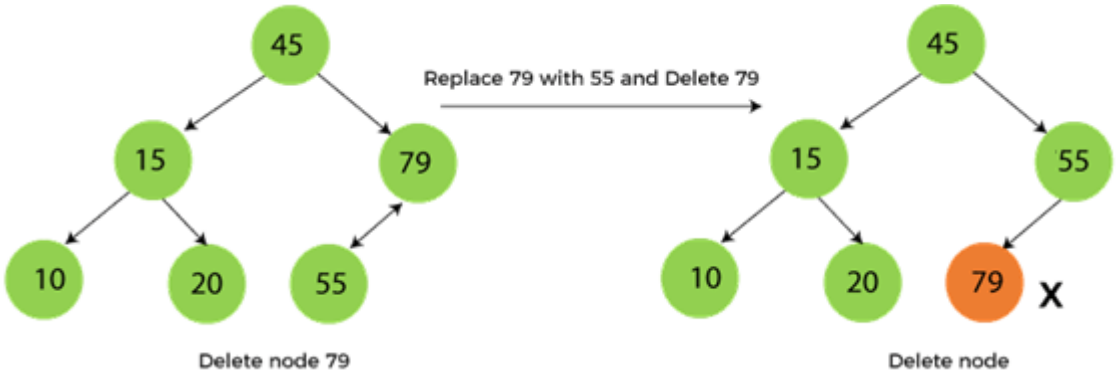
We can see the process to delete a leaf node from BST in the below image. In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.




**When the node to be deleted has only one child**

In this case, we have to replace the target node with its child, and then delete the child node. It means that after replacing the target node with its child node, the child node will now contain the value to be deleted. So, we simply have to replace the child node with NULL and free up the allocated space.

We can see the process of deleting a node with one child from BST in the below image. In the below image, suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.
So, the replaced node 79 will now be a leaf node that can be easily deleted.



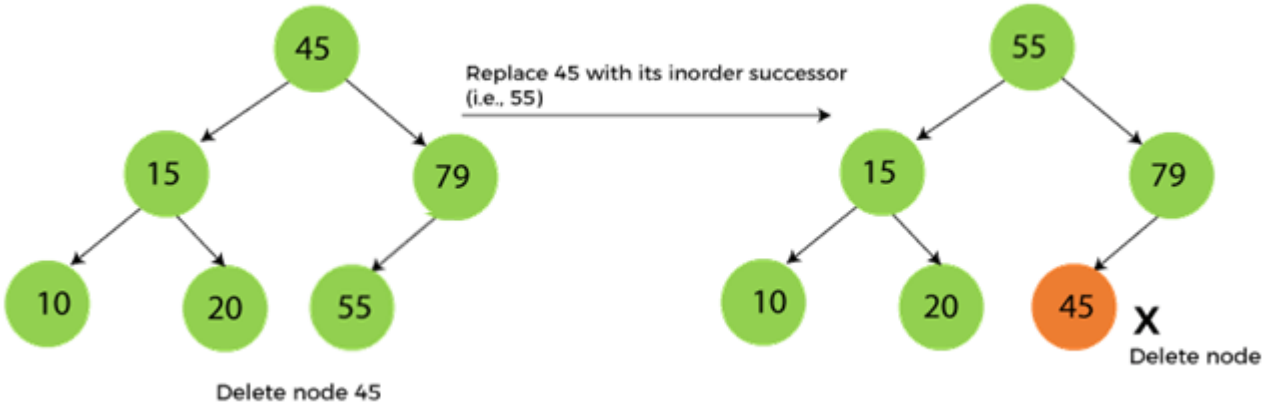

**When the node to be deleted has two children**

This case of deleting a node in BST is a bit complex among other two cases. In such a case, the steps to be followed are listed as follows -

- First, find the inorder successor of the node to be deleted.
- After that, replace that node with the inorder successor until the target node is placed at the leaf of tree.
- And at last, replace the node with NULL and free up the allocated space.

The inorder successor is required when the right child of the node is not empty. We can obtain the inorder successor by finding the minimum element in the right child of the node.

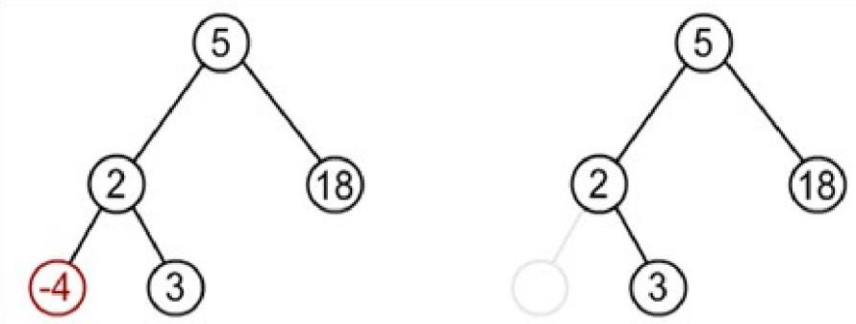
We can see the process of deleting a node with two children from BST in the below image. In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.




**Example:**

1. **The node to be deleted may be a leaf node:**

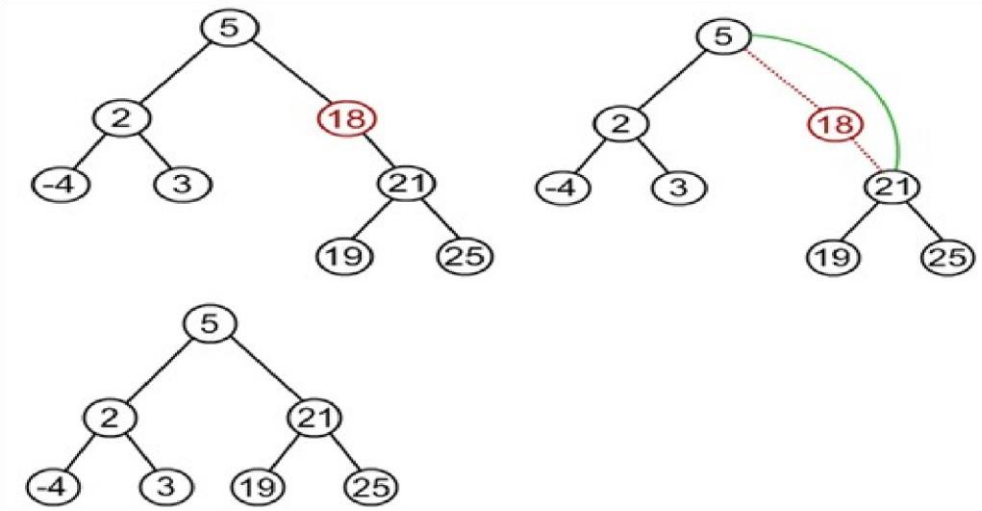
In this case simply delete a node and set null pointer to its parents those side at which this deleted node exist.




Suppose node to be deleted is -4

2. **The node to be deleted has one child:**

In this case the child of the node to be deleted is appended to its parent node. Suppose node to be deleted is 18

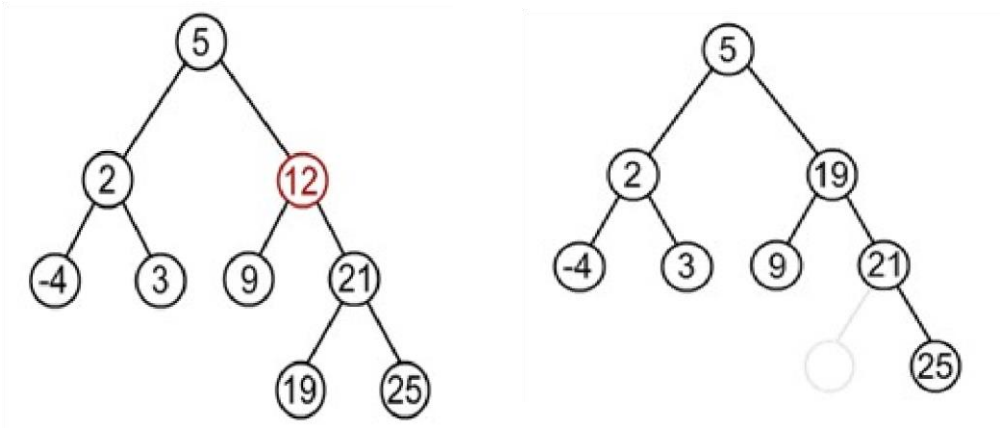



*3.* **the node to be deleted has two children:**

In this case node to be deleted is replaced by its in-order successor node.

OR

If the node to be deleted is either replaced by its right sub-trees leftmost node or its left sub-trees rightmost node.




Suppose node to deleted is 12

Find minimum element in the right sub-tree of the node to be removed. In current example it is 19.

## General algorithm to delete a node from a BST:

1. start
2. if a node to be deleted is a leaf node at left side then simply delete and set null pointer to it's parent's left pointer.
3. If a node to be deleted is a leaf node at right side then simply delete and set null pointer to it's parent's right pointer

4. if a node to be deleted has on child then connect it's child pointer with it's parent pointer and delete it from the tree
5. if a node to be deleted has two children then replace the node being deleted either by
   a. right most node of it's left sub-tree or
   b. left most node of it's right sub-tree.
6. End

## The deleteBST function:

```
struct bnode *delete(struct bnode *root, int item)
{
        struct bnode *temp;
        if(root==NULL)
        {
            printf("Empty tree");
            return;
        }
        else if(item<root->info)
                root->left=delete(root->left, item);
        else if(item>root->info)
                root->right=delete(root->right, item);
        else if(root->left!=NULL &&root->right!=NULL)  //node has two child
        {
                temp=find_min(root->right);
                root->info=temp->info;
                root->right=delete(root->right, root->info);
        }
        else
        {
                temp=root;
                 if(root->left==NULL)
                        root=root->right;
            else if(root->right==NULL)
```

```
                root=root->left;
        free(temp);
    }
    return(temp);
}
/**********find minimum element function**********/
struct bnode *find_min(struct bnode *root)
{
    if(root==NULL)
            return0;
    else if(root->left==NULL)
            return root;
        else
            return(find_min(root->left));
}
```

## AVL Tree (Balanced Tree):

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.

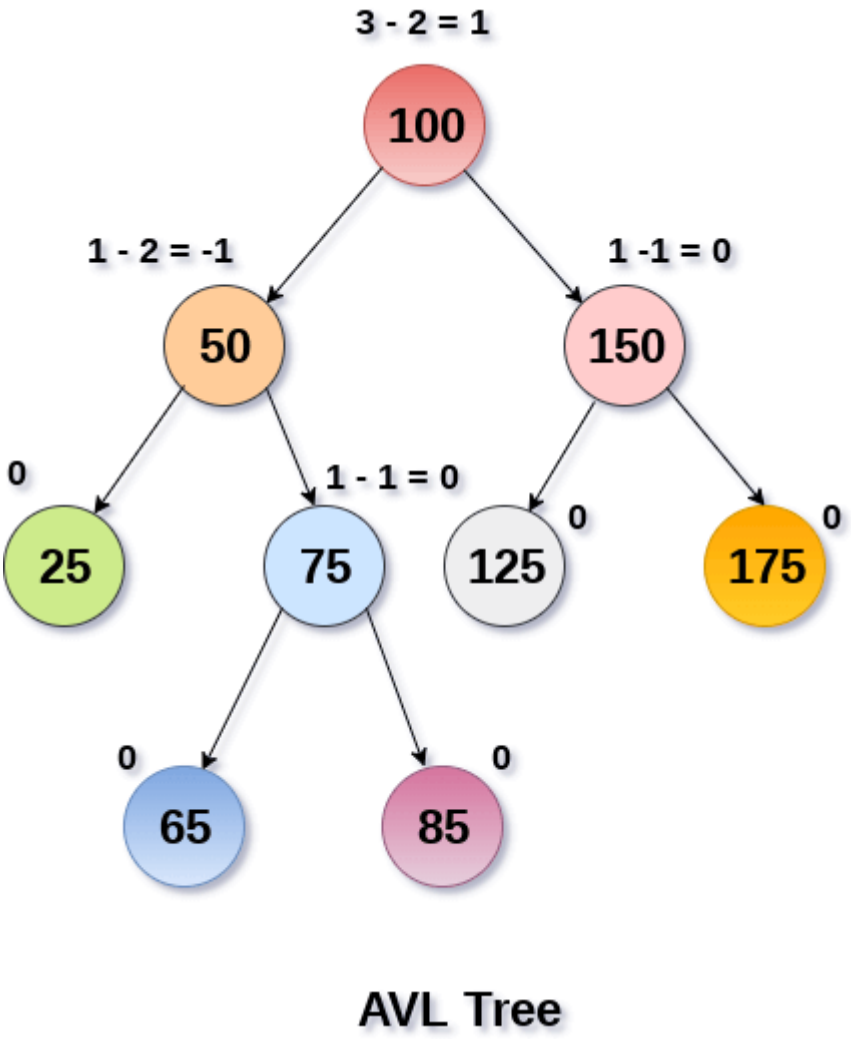
AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

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.

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. therefore, it is an example of AVL tree.




AVL Tree

## Operations on AVL tree

Due to the fact that, AVL tree is also a binary search tree therefore, all the operations are performed in the same way as they are performed in a binary search tree. Searching and traversing do not lead to the violation in property of AVL tree. However, insertion and deletion are the operations which can violate this property and therefore, they need to be revisited.

| SN | Operation | Description |
|---|---|---|
| 1 | **Insertion** | Insertion in AVL tree is performed in the same way as it is performed in a binary search tree. However, it may lead to violation in the AVL tree property and therefore the tree may need balancing. The tree can be balanced by applying rotations. |
| 2 | **Deletion** | Deletion can also be performed in the same way as it is performed in a binary search tree. Deletion may also disturb the balance of the tree therefore, various types of rotations are used to rebalance the tree. |

**Compiled by: Er. Sandesh S Poudel**

# AVL Balancing Algorithm

**Step 1:** Insert the new node using the same algorithm as for an ordinary binary tree.
**Step 2:** Beginning with the new node, calculate the height of the left or the right sub tree of each node on the path leading from the new node back up the tree towards the root.
**Step 3:** Continue these checks until either the root node is encountered and all nodes along the path have differences not greater than one or until the first difference greater than one is found.
**Step 4:** If an imbalance is found i.e. height difference is not equal to +1, -1, or 0, perform a rotation of the node to correct the imbalance by applying the following rule:

1) When the first node is out of balance according to AVL algorithm restrict the attention at that node and the two nodes in the two layers immediately below it.
2) If these three nodes lie in a straight line a single rotation is needed to restore the balance.
3) If these three nodes lies in dog legged pattern, meaning bend in the path then use the doble rotation to restore the balance.

# 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

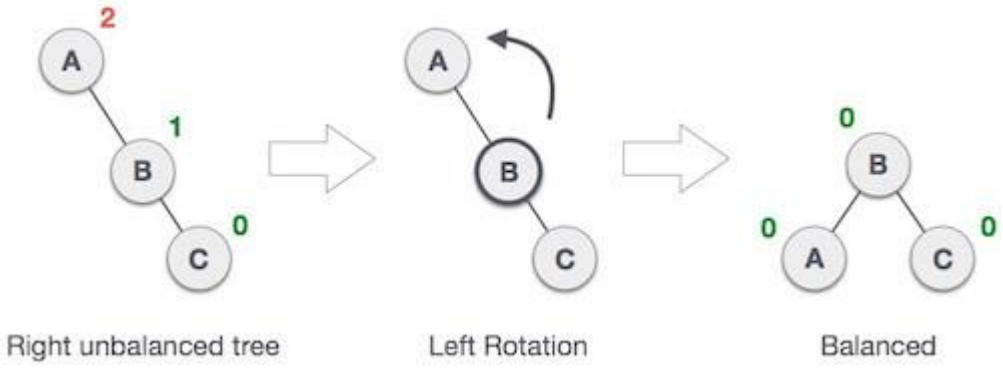
Where node A is the node whose balance Factor is other than -1, 0, 1.

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

# AVL rotations:-

## 1. RR Rotation

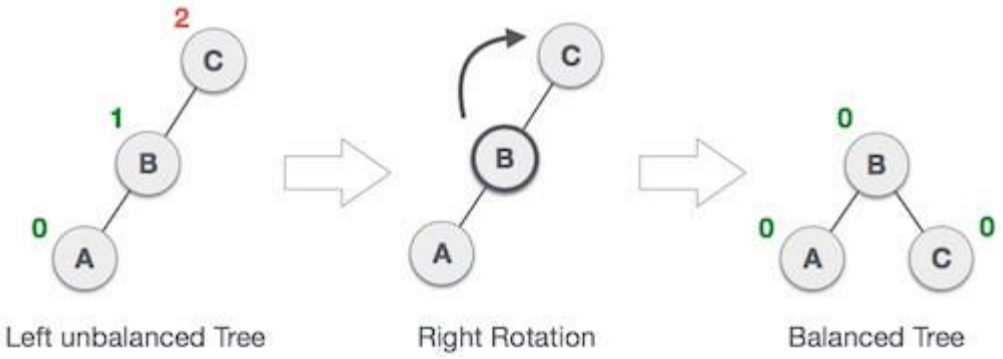
When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor -2




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.



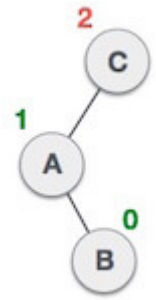
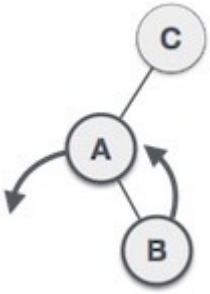
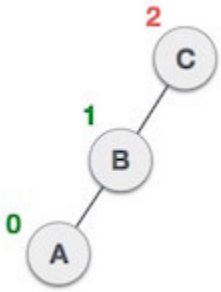
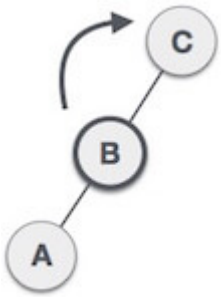

In above example, node C has balance factor 2 because a node A is inserted in the left subtree of C left subtree. We perform the LL rotation on the edge below A.

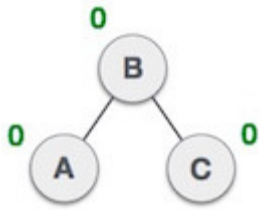
**Compiled by: Er. Sandesh S Poudel**

## 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.

**Let us understand each and every step very clearly:**

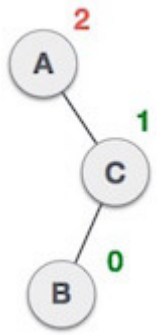
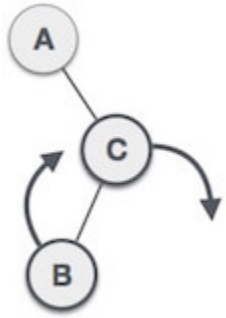
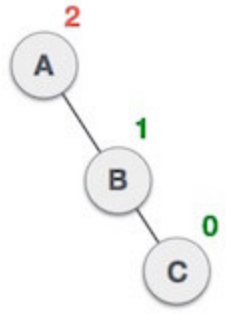
| State | Action |
|---|---|
|  | A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C |
|  | As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node **A**, has become the left subtree of **B**. |
|  | After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of **C** |
|  | Now we perform LL clockwise rotation on full tree, i.e. on node C. node **C** has now become the right subtree of node B, A is left subtree of B |

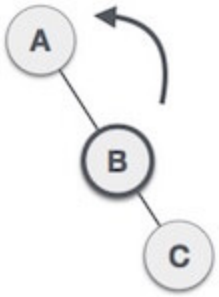



Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.

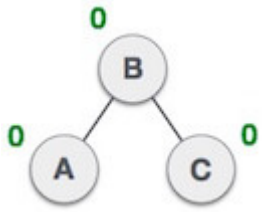
## 4. RL Rotation

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.

| State | Action |
|---|---|
|  | A node **B** has been inserted into the left subtree of **C** the right subtree of **A**, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A |
|  | As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at **C** is performed first. By doing RR rotation, node **C** has become the right subtree of **B**. |
|  | After performing LL rotation, node **A** is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A. |



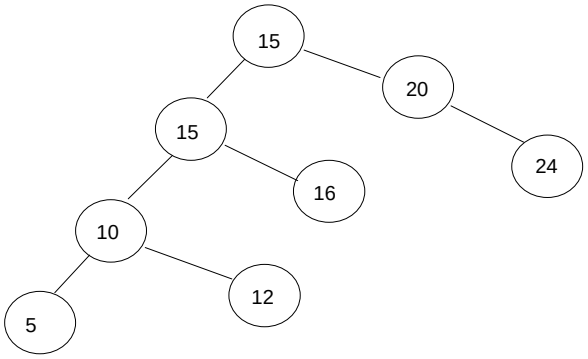

Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node **C** has now become the right subtree of node B, and node A has become the left subtree of B.




Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.

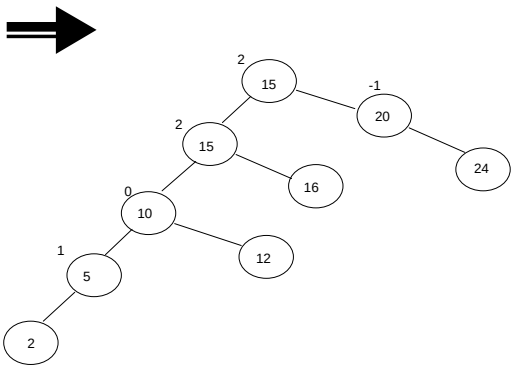
## Example:

## Left to left rotation:-

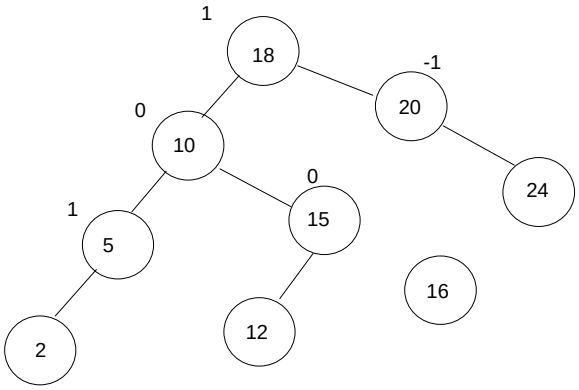



**Insert 2.**

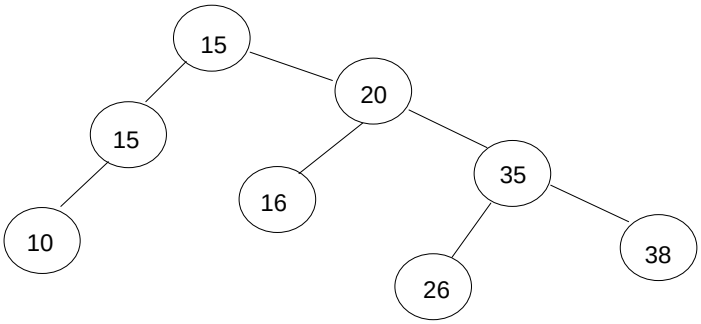
Pivot node



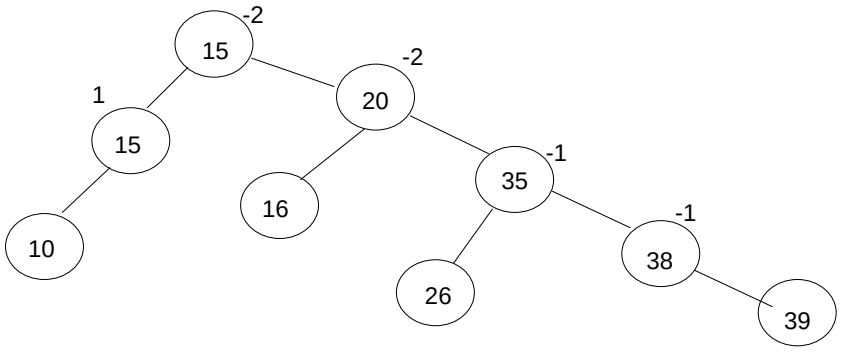
If the node is inserted on the left side of left  subtree than we perform left to left rotation.



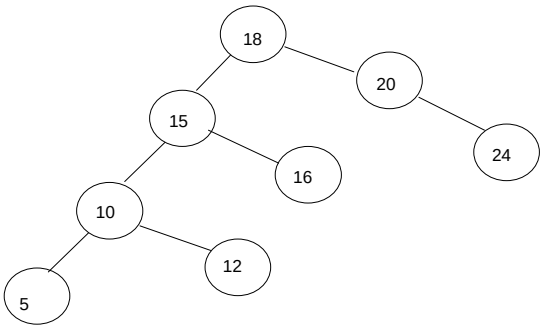

## Right to right rotation:-

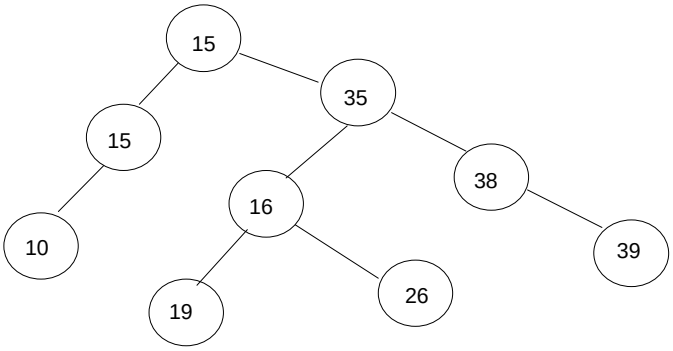



**Insert 39**




If the node is inserted on the right side of right sub tree than we perform right to right rotation.

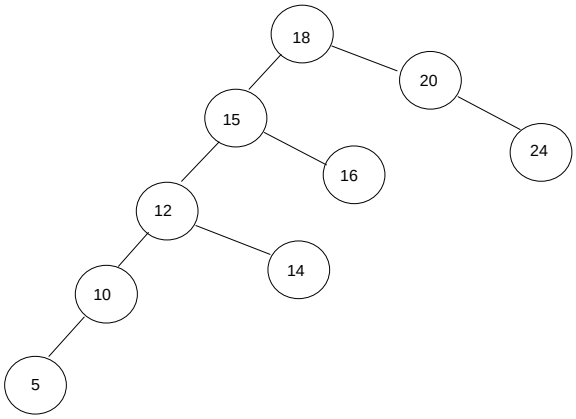


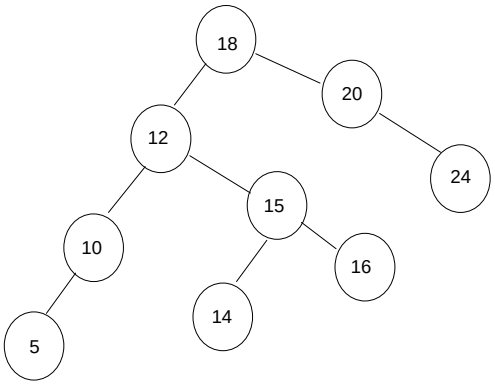




## Left to right rotation:-

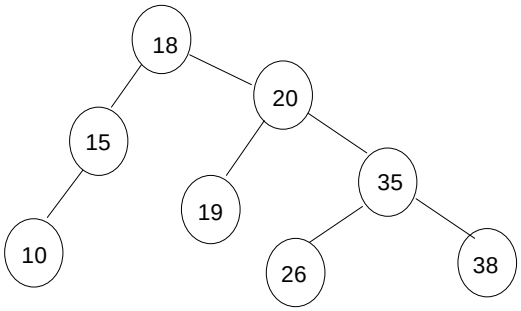
If the node is inserted on the right side of left sub tree then we perform left to right rotation.

In this left to right rotation, the operation involves two steps 1st we perform right to right from next to pivot node then perform left to left.

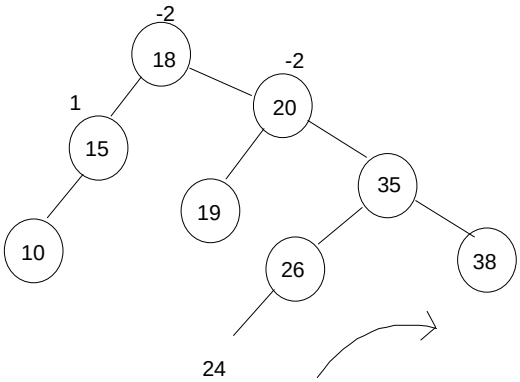


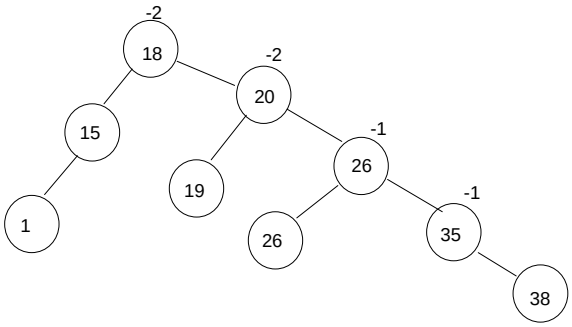




## Right to left rotation:-

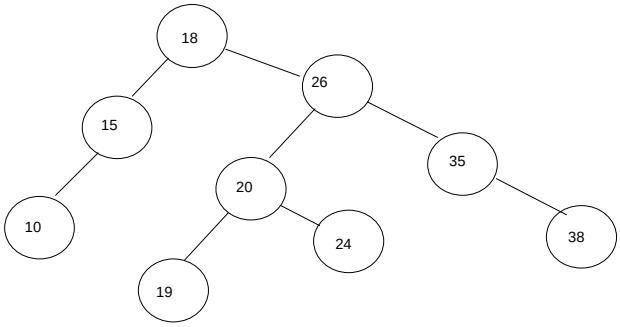



insert 24



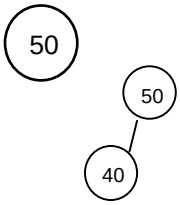

## Rotate right to Right:-




## Construct an AVL tree from given data.

I/Ps : 50, 40, 35, 58, 48, 42, 60, 30, 33, 32

**Insert 50**

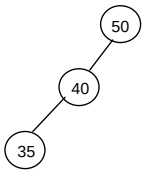



**Insert 40**

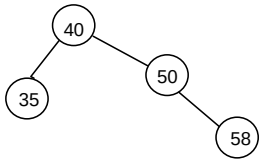



**Compiled by: Er. Sandesh S Poudel**

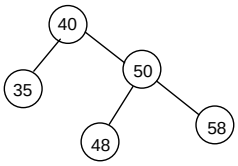
**Insert 35**



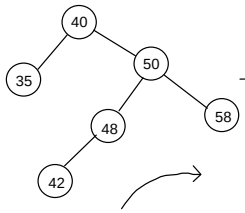

**Insert 58**




**Insert 48**

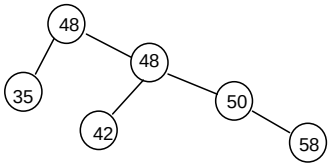



**Insert 42**

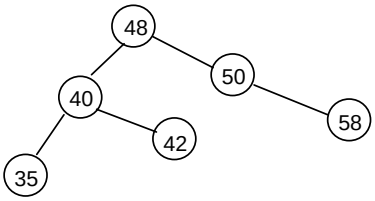



Right to left rotation

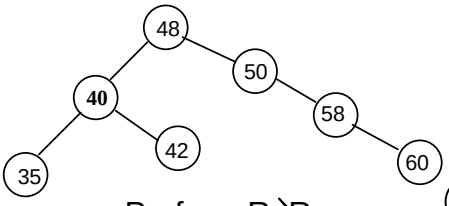
Step – I



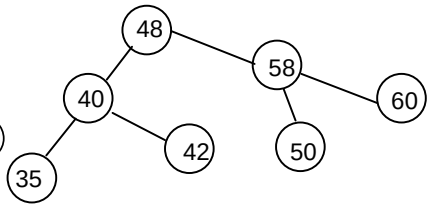

Step –ii




**Insert 60**

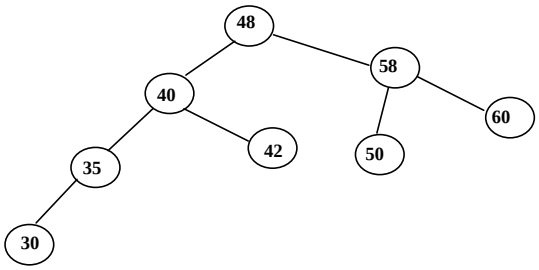



Perform R→R

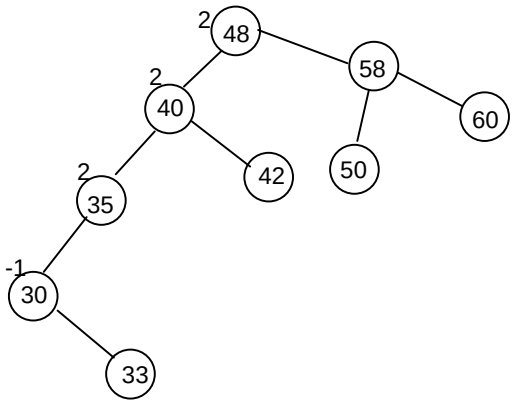



**Insert 30**

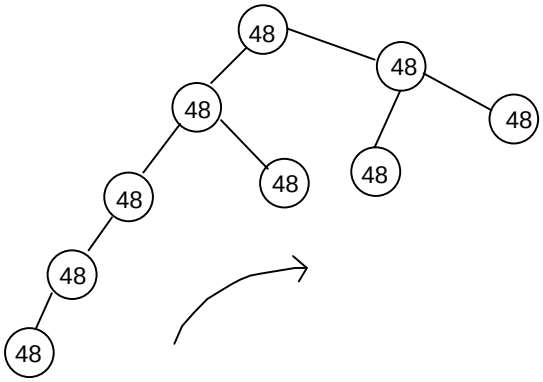
**Compiled by: Er. Sandesh S Poudel**



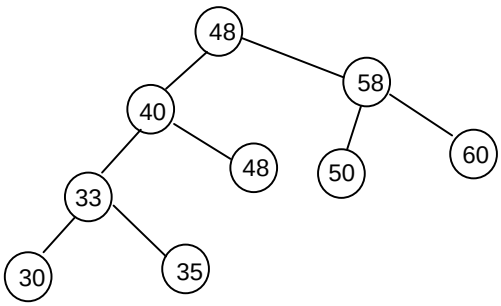

**Insert 33**



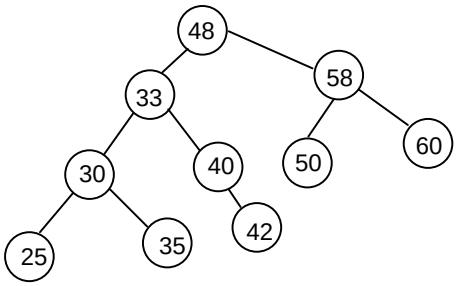

**Step I**



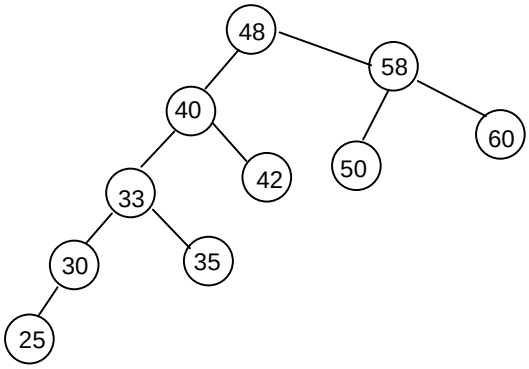

Step II




**Compiled by: Er. Sandesh S Poudel**




**Insert 25**

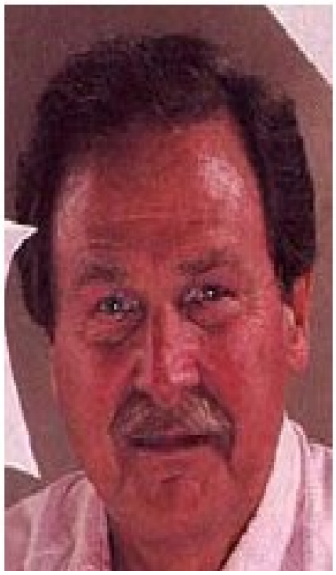



# Huffman algorithm:

*Huffman algorithm is a method for building an extended binary tree with a minimum weighted path length from a set of given weights.*

-This is a method for the construction of minimum redundancy codes .

·Applicable to many forms of data transmission



## Huffman Coding Algorithm

- In Huffman tree each node contains a symbol and its frequency. Each node of the tree represents a symbol and each leaf represents a symbol of original alphabets i.e. distinct symbols.
- In binary tree created by this method for symbol and its frequency table of alphabets is called Huffman trees. (After the discovery of this method by Huffman).
- In Huffman tree, generating the tree of the distinct symbols and their heights are according to their frequencies.
- Most frequent symbols are kept at left side of the tree.
- After constructing the binary tree, coding is done by assigning 0 to left side and 1 to right side child. This process continues up to leaf.
- Huffman tree is strictly binary tree.
- For n leaves in SBT, the total numbers of nodes are (2n - 1). So Huffman tree can be stored in an array of size 2n – 1. So advance allocation of memory is needed.

## Algorithm

**Compiled by: Er. Sandesh S Poudel**

1) Take input as string.
2) Count each character and assign its frequency.
3) Make the node for each symbol character and its frequency.
4) Put the symbol into highest priority queue according to the frequency.
5) Create new node;

   New node data = last data + second last data

   New frequency = last data frequency + second last data frequency.

6) Assign new node left = last pointer of an array

- New node right = second last pointer of an array.

7) Replace second last pointer y new node pointer and assign second last frequency = new frequency
8) Remove last data and frequency.
9) Repeat process from step 4 to step 8 until we get a single tree.
10) Stop

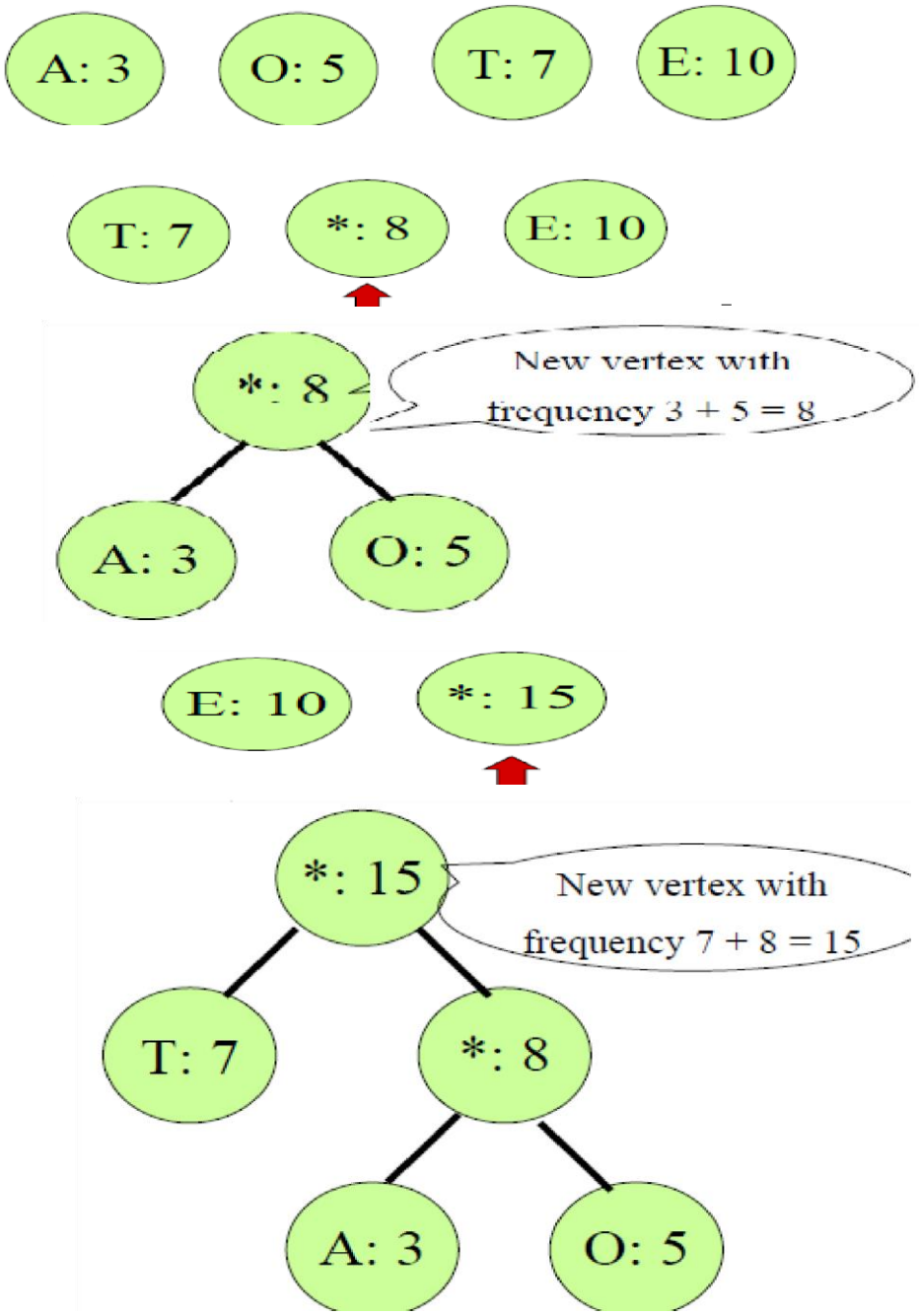
## Example 1:

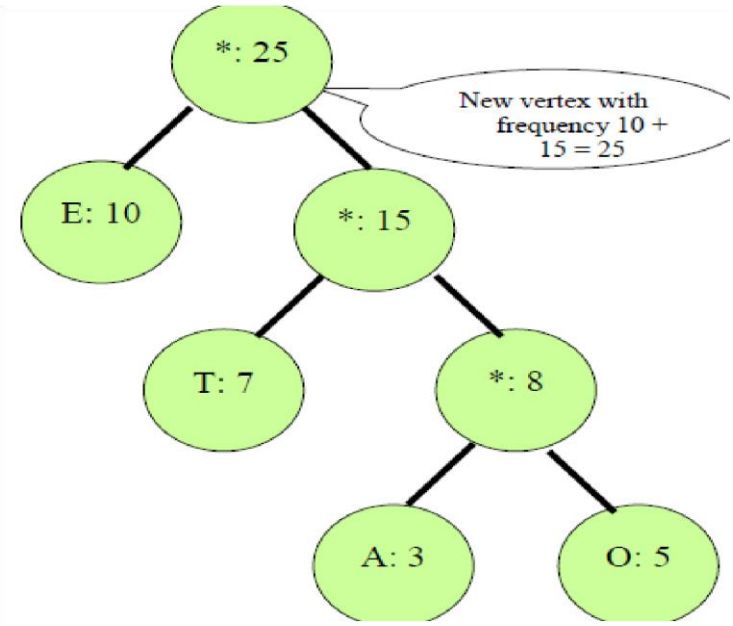
| character | frequencies |
|---|---|
| E: | 10 |
| T: | 07 |
| O: | 05 |
| A: | 03 |

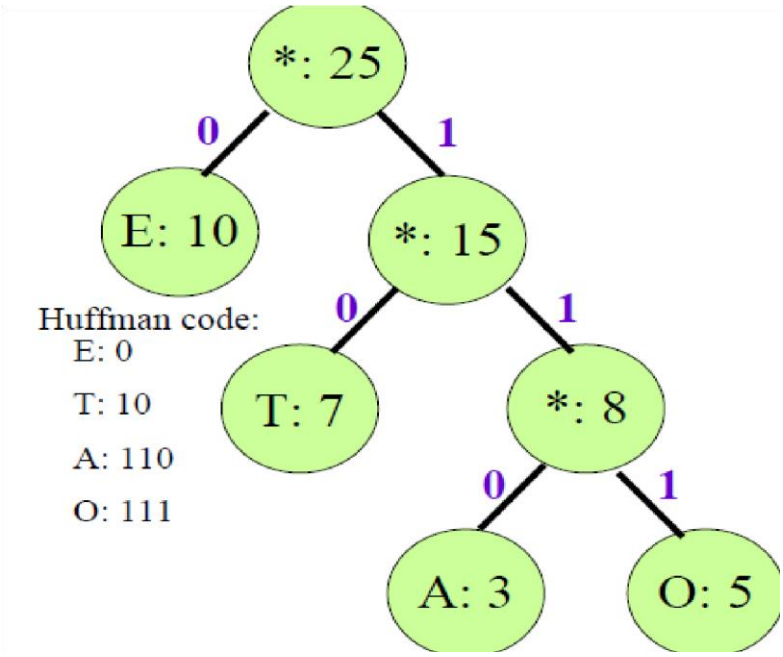
Now sort these characters according to their frequencies in non-decreasing order.

| character | frequencies | code |
|---|---|---|
| A: | 03 | 00 |
| O: | 05 | 01 |
| T: | 07 | 10 |
| E: | 10 | 11 |

Here before using Huffman algorithm the total number of bits required is nb=3*2+5*2+7*2+10*2=06+10+14+20=50bits






**Left branch is 0**

**Right branch is 1**

Now from variable length code we get following code sequence.

| **character** | **frequencies** | **code** |
|---|---|---|
| A: | 03 | 110 |
| O: | 05 | 111 |
| T: | 07 | 10 |
| E: | 10 | 0 |

Thus after using Huffman algorithm the total number of bits required is nb=3*3+5*3+7*2+10*1=09+15+14+10=48bits

(50-48)/50*100%=4%

Since in this small example we save about 4% space by using Huffman algorithm. If we take large example with a lot of characters and their frequencies we can save a lot of space.

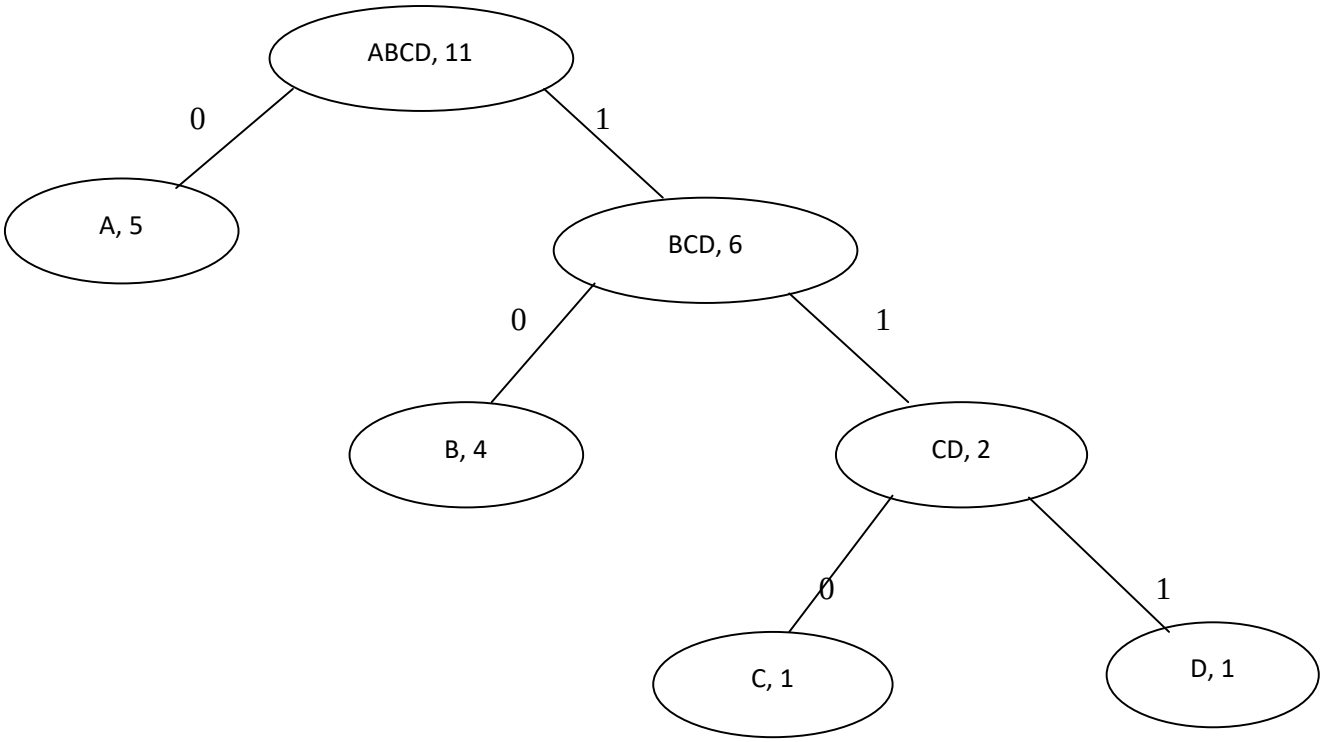
**Example 2**: Encode AABBABACADB USING Huffman Algorithm

Solution:

Here,

| SYMBOL | FREQUENCY |
|---|---|
| A | 5 |
| B | 4 |
| C | 1 |
| D | 1 |
| TOTAL FREQUENCY 11 | |

Now making node of input string and using algorithm,




The encoding is shown as below:

A 0

B 01

C 011

D 111

Analysis:

We have,

A A B B A B A C A D B

Binary representation before Huffman coding and after Huffman coding is shown as below:

| Symbol | Without H.C. | With H.C. |
|---|---|---|
| A | 00 | 0 |
| B | 01 | 01 |
| C | 10 | 011 |
| D | 11 | 111 |

Now number of bits required to represent the code before coding is 22 bits

Number of bits after coding is 19 bits.

Hence Huffman coding approach gives efficient coding in terms of space and transition of data.

### Application of Huffman coding

- Compression, encoding, transition of data

### Advantages

- Save storage and develop coding efficiency.

## B Tree:

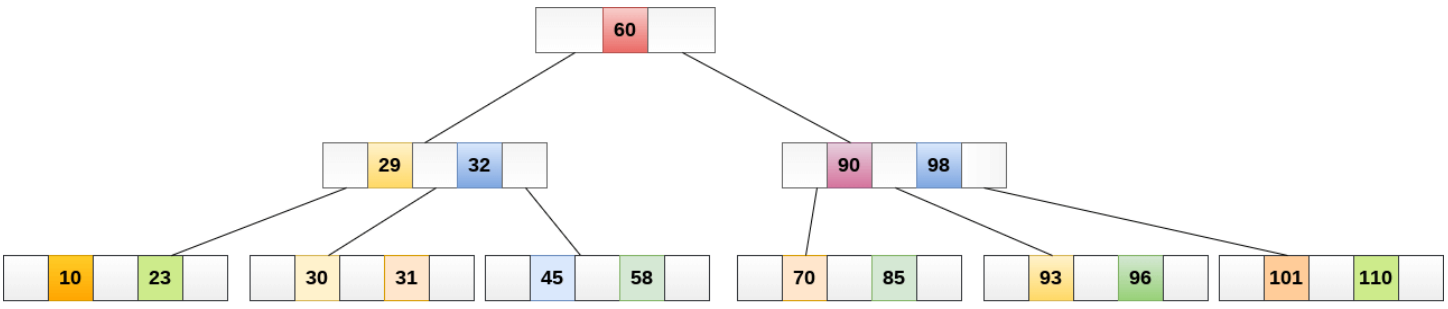
B Tree is a specialized m-way tree that can be widely used for disk access. A B-Tree of order m can have at most m-1 keys and m children. One of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least m/2 children.
3. The root nodes must have at least 2 nodes.
4. All leaf nodes must be at the same level.

It is not necessary that, all the nodes contain the same number of children but, each node must have m/2 number of nodes.

A B tree of order 4 is shown in the following image.



While performing some operations on B Tree, any property of B Tree may violate such as number of minimum children a node can have. To maintain the properties of B Tree, the tree may split or join.

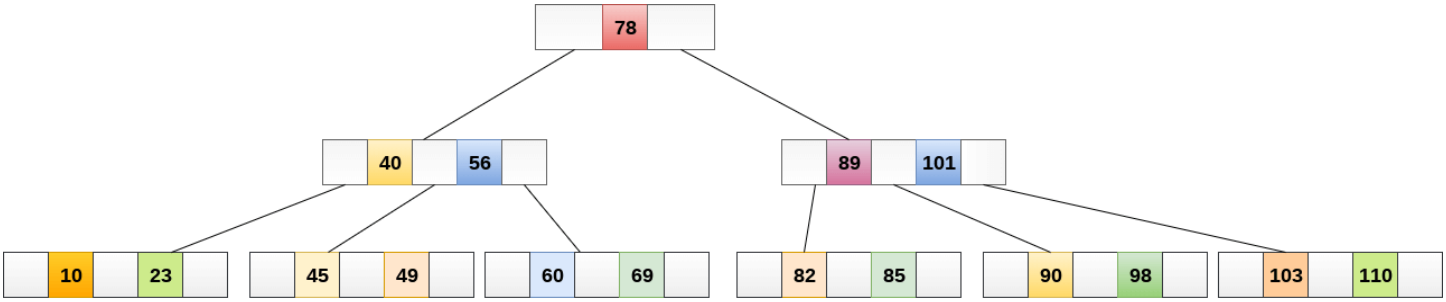
# Operations

## Searching :

Searching in B Trees is similar to that in Binary search tree. For example, if we search for an item 49 in the following B Tree. The process will something like following :

1. Compare item 49 with root node 78. since 49 < 78 hence, move to its left sub-tree.
2. Since, 40<49<56, traverse right sub-tree of 40.
3. 49>45, move to right. Compare 49.
4. match found, return.

Searching in a B tree depends upon the height of the tree. The search algorithm takes O(log n) time to search any element in a B tree.




## Inserting

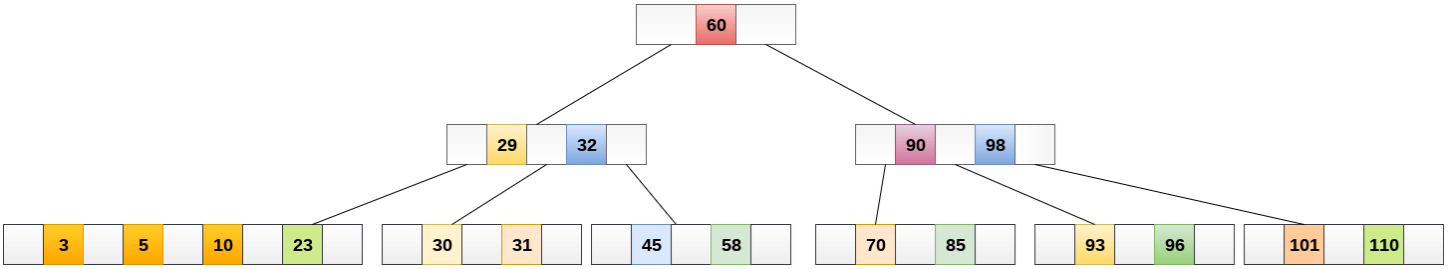
Insertions are done at the leaf node level. The following algorithm needs to be followed in order to insert an item into B Tree.

1. Traverse the B Tree in order to find the appropriate leaf node at which the node can be inserted.
2. If the leaf node contain less than m-1 keys then insert the element in the increasing order.
3. Else, if the leaf node contains m-1 keys, then follow the following steps.
    - Insert the new element in the increasing order of elements.
    - Split the node into the two nodes at the median.
    - Push the median element up to its parent node.

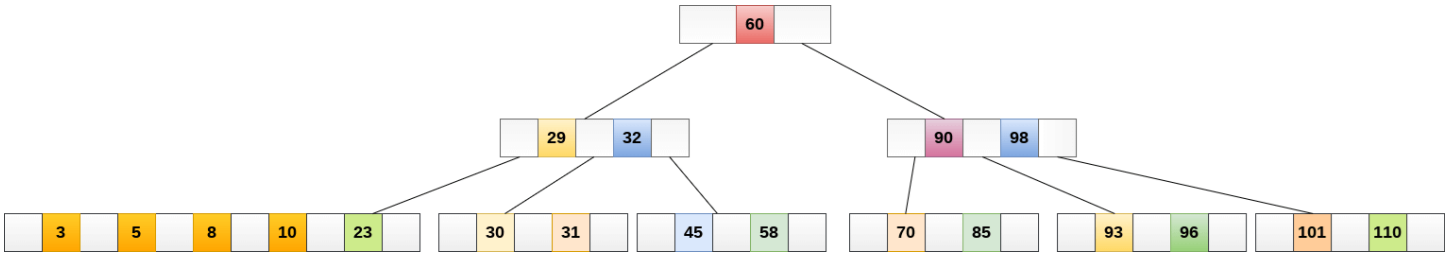
- If the parent node also contain m-1 number of keys, then split it too by following the same steps.

**Example:**

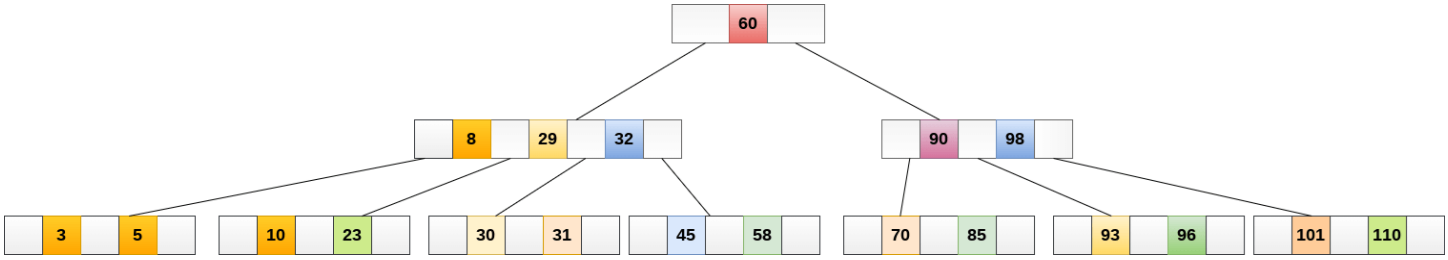
Insert the node 8 into the B Tree of order 5 shown in the following image.




8 will be inserted to the right of 5, therefore insert 8.




The node, now contain 5 keys which is greater than (5 -1 = 4 ) keys. Therefore split the node from the median i.e. 8 and push it up to its parent node shown as follows.




## Deletion

Deletion is also performed at the leaf nodes. The node which is to be deleted can either be a leaf node or an internal node. Following algorithm needs to be followed in order to delete a node from a B tree.

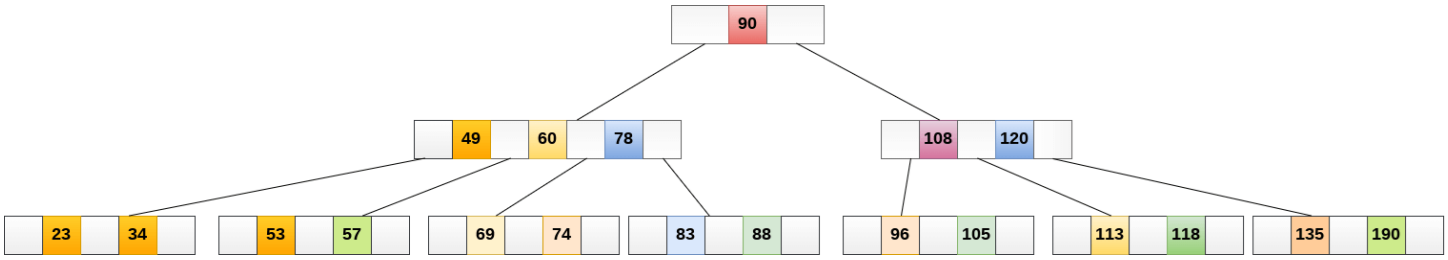
1. Locate the leaf node.

2. If there are more than m/2 keys in the leaf node then delete the desired key from the node.
3. If the leaf node doesn't contain m/2 keys then complete the keys by taking the element from eight or left sibling.
    - If the left sibling contains more than m/2 elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.
    - If the right sibling contains more than m/2 elements then push its smallest element up to the parent and move intervening element down to the node where the key is deleted.
4. If neither of the sibling contain more than m/2 elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.
5. If parent is left with less than m/2 nodes then, apply the above process on the parent too.

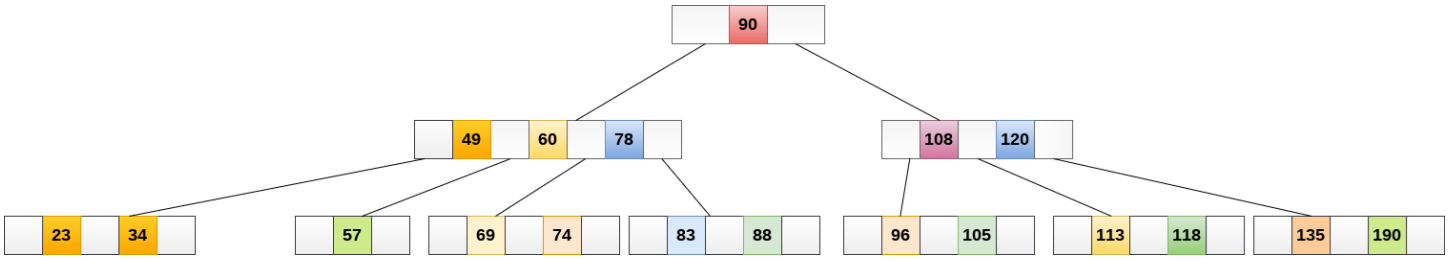
If the the node which is to be deleted is an internal node, then replace the node with its in-order successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

**Example 1**

Delete the node 53 from the B Tree of order 5 shown in the following figure.

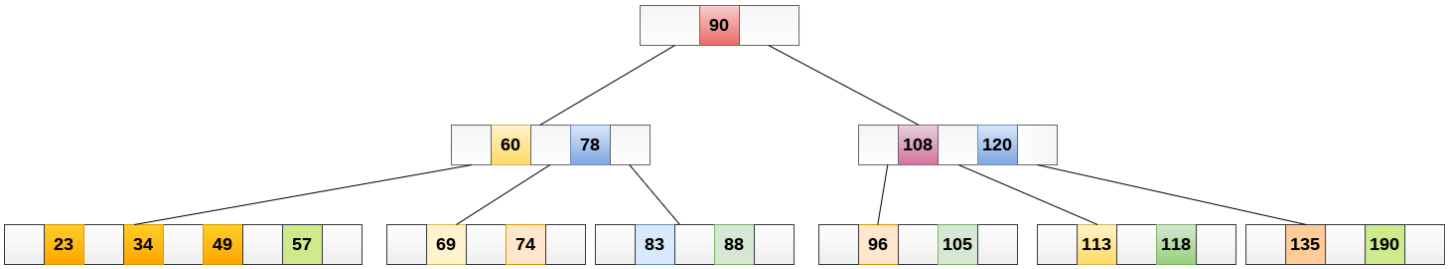



53 is present in the right child of element 49. Delete it.



Now, 57 is the only element which is left in the node, the minimum number of elements that must be present in a B tree of order 5, is 2. it is less than that, the elements in its left and right sub-tree are also not sufficient therefore, merge it with the left sibling and intervening element of parent i.e. 49.

The final B tree is shown as follows.




## Application of B tree

B tree is used to index the data and provides fast access to the actual data stored on the disks since, the access to value stored in a large database that is stored on a disk is a very time consuming process.

Searching an un-indexed and unsorted database containing n key values needs O(n) running time in worst case. However, if we use B Tree to index this database, it will be searched in O(log n) time in worst case.

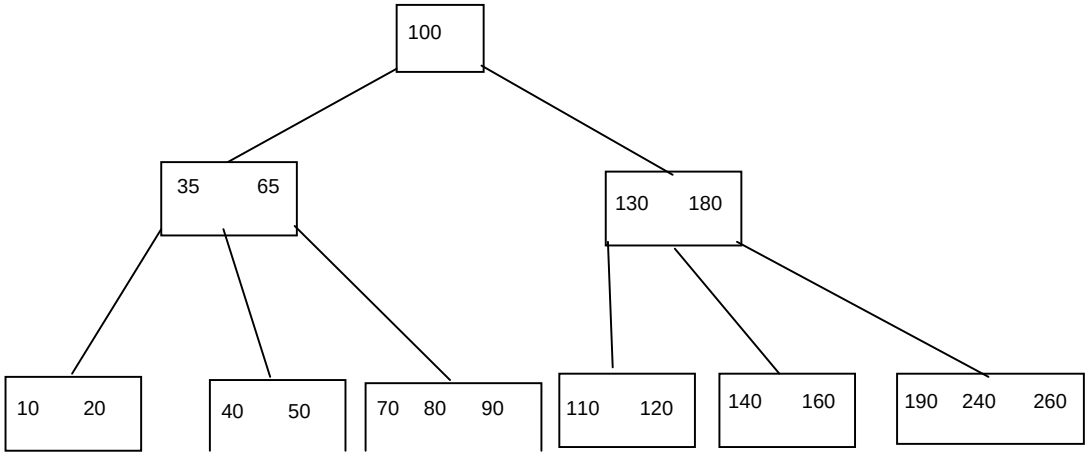
## B – Tree "Balanced Tree"

- It is also known as balanced sort tree.
- The height of the tree must be kept to a minimum.
- There must be no empty sub trees. Above the leaves of the tree.
- The leaves of the tree must all be the same level.
- All nodes except the leaves must have at least some minimum no of children.

## B – Tree of order n can be defined as:-

- Each node has at least n+2 & maximum 'n' non empty children.
- All leaves nodes will be at the same level.
- All the leaf nodes contain minimum n-1 keys.

- Keys are arranged in a defined order with in the node. All keys in sub tree to the left of the key are the procedure of the key and that on the right are successors of the key.
- When a new key is to be inserted in to a full node, then split the nodes with the median value is inserted in the parent node. In case the parent node is root, a new node is created.

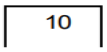


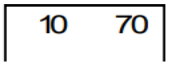

- Each node at same leaves.
- All non- leaf nodes have no empty subtree.
- Keys 1 less than no. of their children.

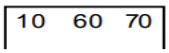
## Q. Construct a B – tree of order 5 inserting the keys:-

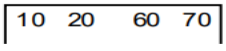
10, 70, 60, 20, 110, 40, 80, 130, 100, 50, 190, 90, 180, 140, 280.

Sol$^{n}$:-

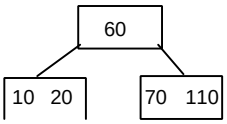
Insert 10 


Insert 70 

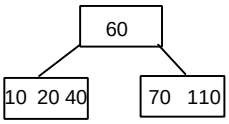

Insert 60 


Insert 20 

Insert 110

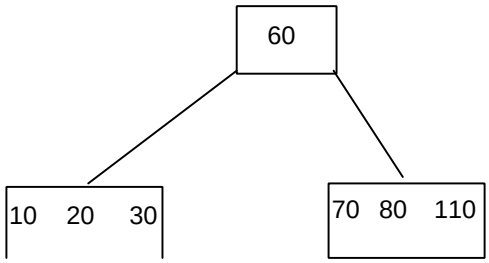



Insert 40

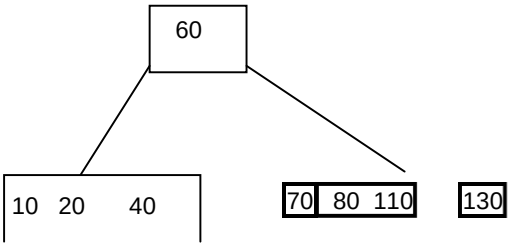



80, 130, 100, 150, 90, 180, 240, 30, 120, 140, 160

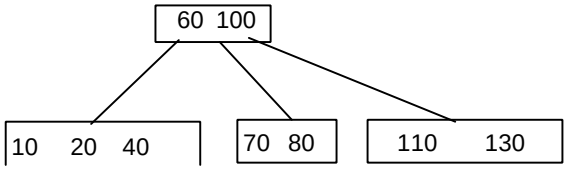
Insert 80



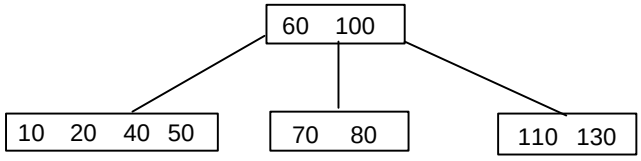

Insert 130



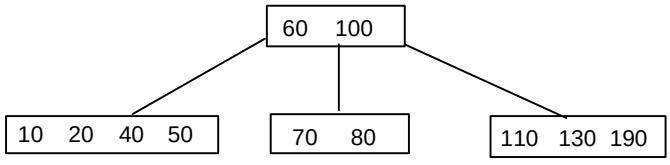

Insert 100



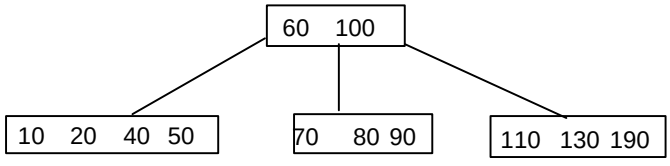

Insert 50



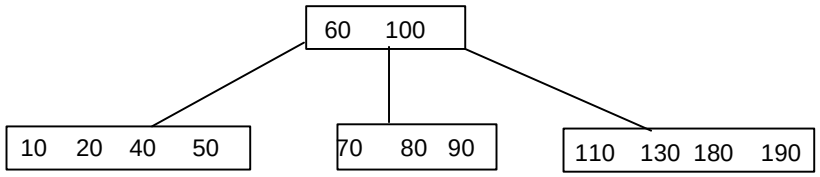

Insert 190



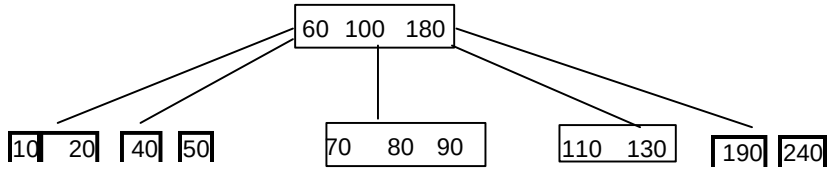

Insert 90



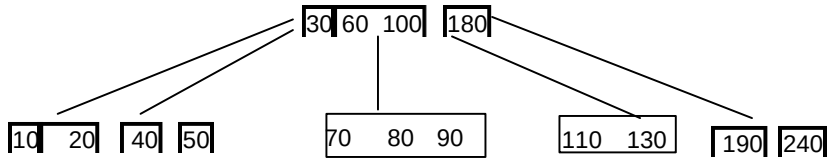
Insert 180



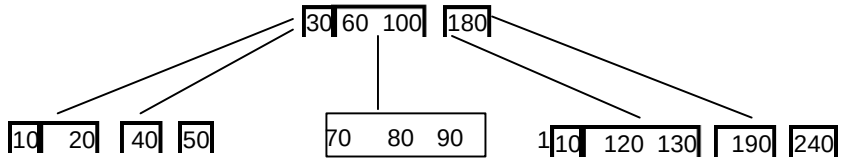

Insert 240



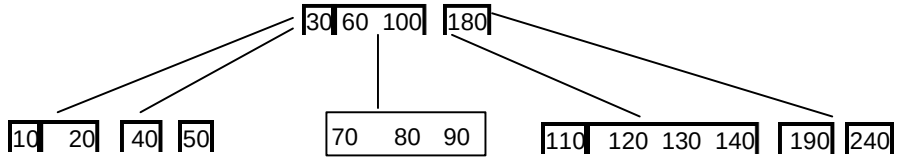

Insert 30



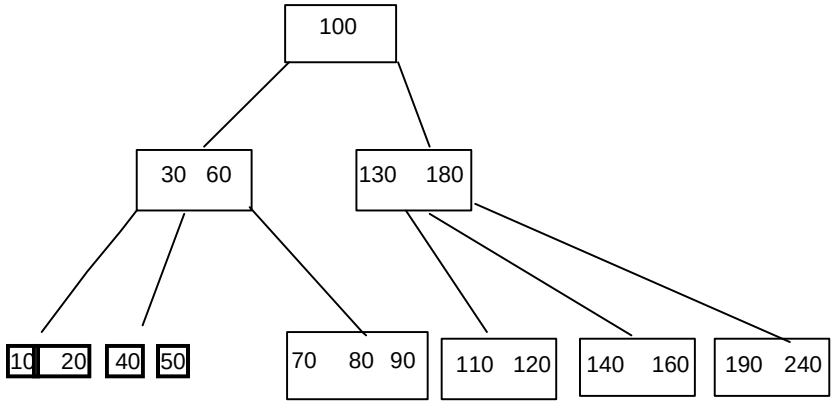

Insert 120




Insert 140




Insert 160




## Deletion in B- tree:

node is leaf node.

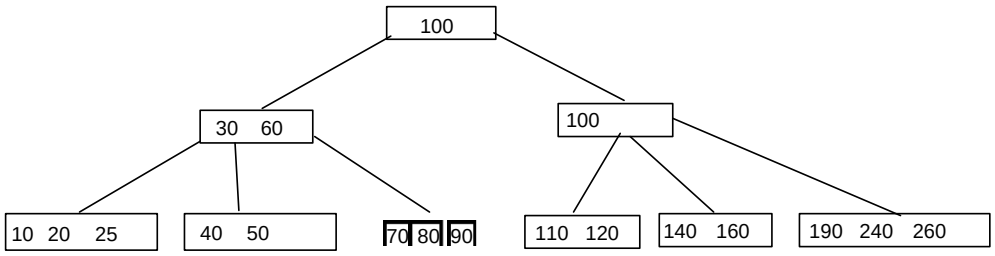
node is non leaf

- if node has more than minimum no. of keys than it can be easily deleted.

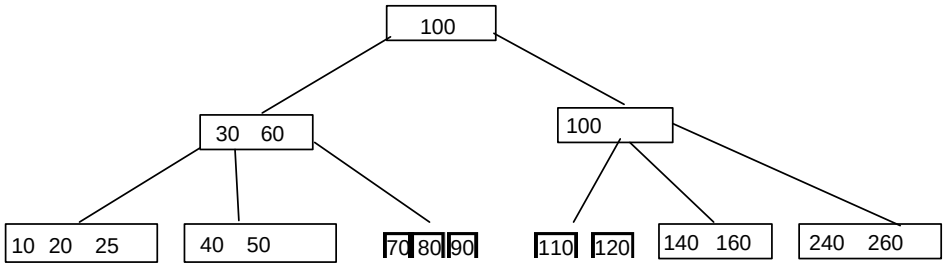
- if it has only minimum no. of keys, than first we see the no. of keys in adjacent leaf node.
- If it has more than minimum no. of keys then first key of the of the adjacent node will go to the parent node & key in parent node will be combined together in one node.
- If now parent has also less than minimum no. of keys then the same thing will be repeated until it will gate the node which has more than the minimum no. of keys.

## Node is non leaf:-

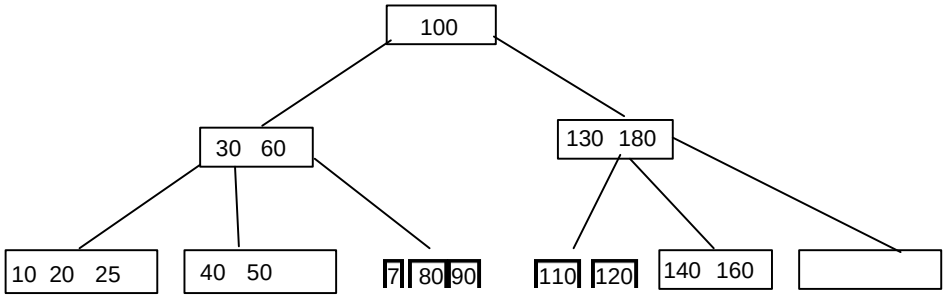
- In this case key will be deleted & it's predecessor or successor key will condition it's place.
- If both nodes of predecessor or successor key have minimum no. of keys then the rates of predecessor & successor keys will be combine.



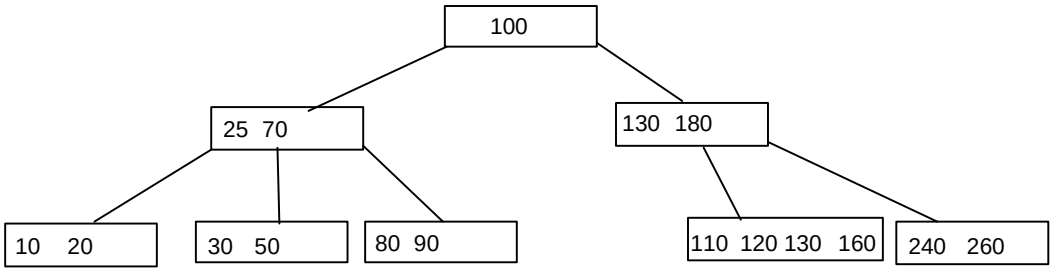

Delete 190



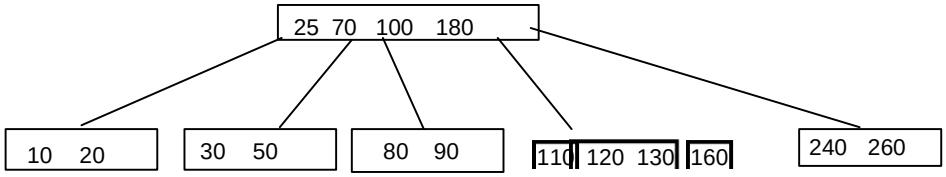

Delete 60




Delete 40




Delete 140




## Question:-

Construct a B- tree of order 5 I/P the element when keys are

659, 767, 702, 157, 728, 102, 461, 899, 920, 44, 744, 264, 384, 344, 973, 905, 999

Perform delete operation for 44, 344, 920

## Red Black Tree

A Red Black Tree is a category of the self-balancing binary search tree. It was created in 1972 by Rudolf Bayer who termed them **"symmetric binary B-trees**."

A red-black tree is a Binary tree where a particular node has color as an extra attribute, either red or black. By check the node colors on any simple path from the root to a leaf, red-black trees secure that no such path is higher than twice as long as any other so that the tree is generally balanced.

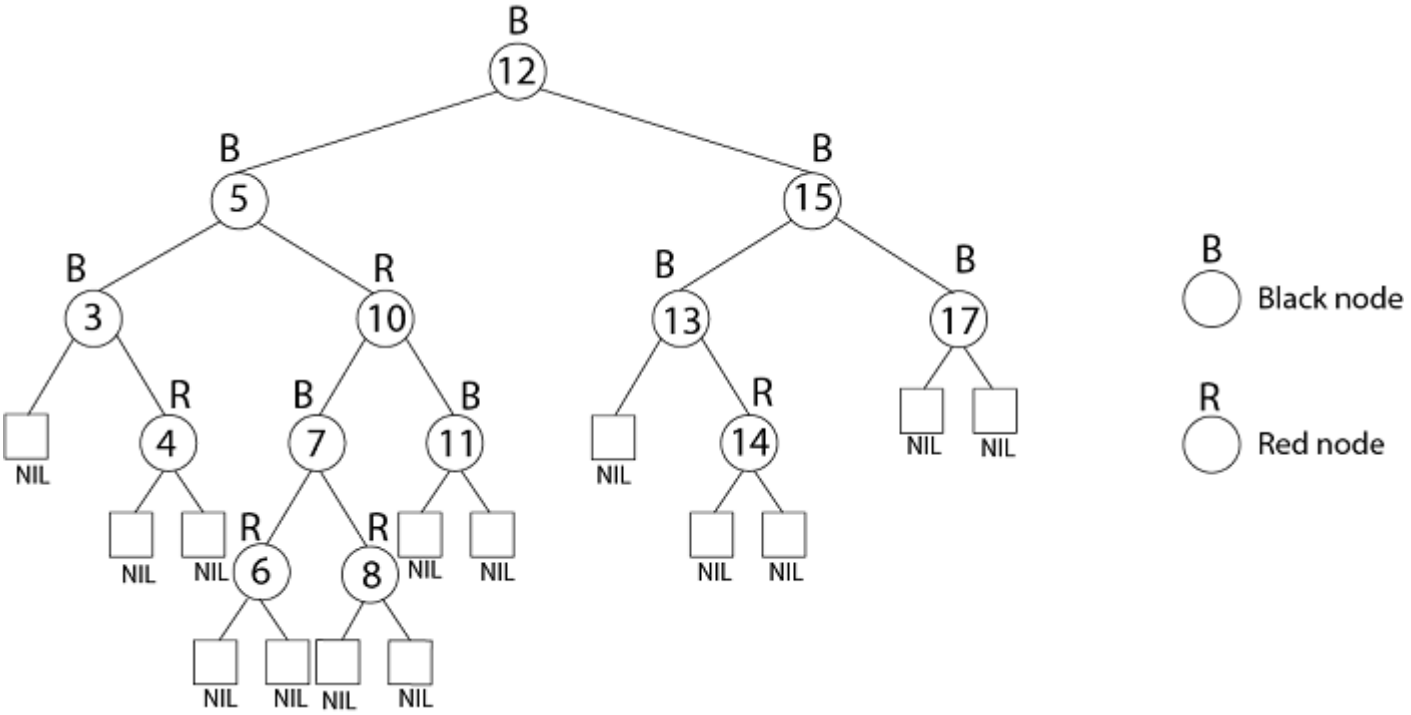
## Properties of Red-Black Trees

A red-black tree must satisfy these properties:

1. The root is always black.
2. A nil is recognized to be black. This factor that every non-NIL node has two children.
3. **Black Children Rule:** The children of any red node are black.

4. **Black Height Rule:** For particular node v, there exists an integer bh (v) such that specific downward path from v to a nil has correctly bh (v) black real (i.e. non-nil) nodes. Call this portion the black height of v. We determine the black height of an RB tree to be the black height of its root.

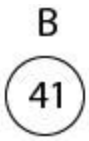
A tree T is an almost red-black tree (ARB tree) if the root is red, but other conditions above hold.



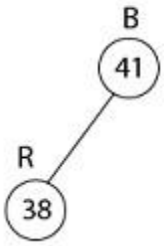

**Example:** Show the red-black trees that result after successively inserting the keys 41,38,31,12,19,8 into an initially empty red-black tree.

**Solution:**

Insert 41

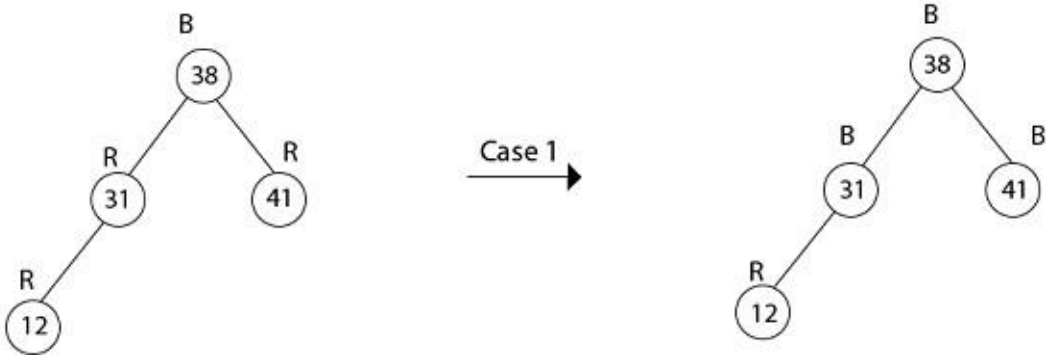



- Insert 38

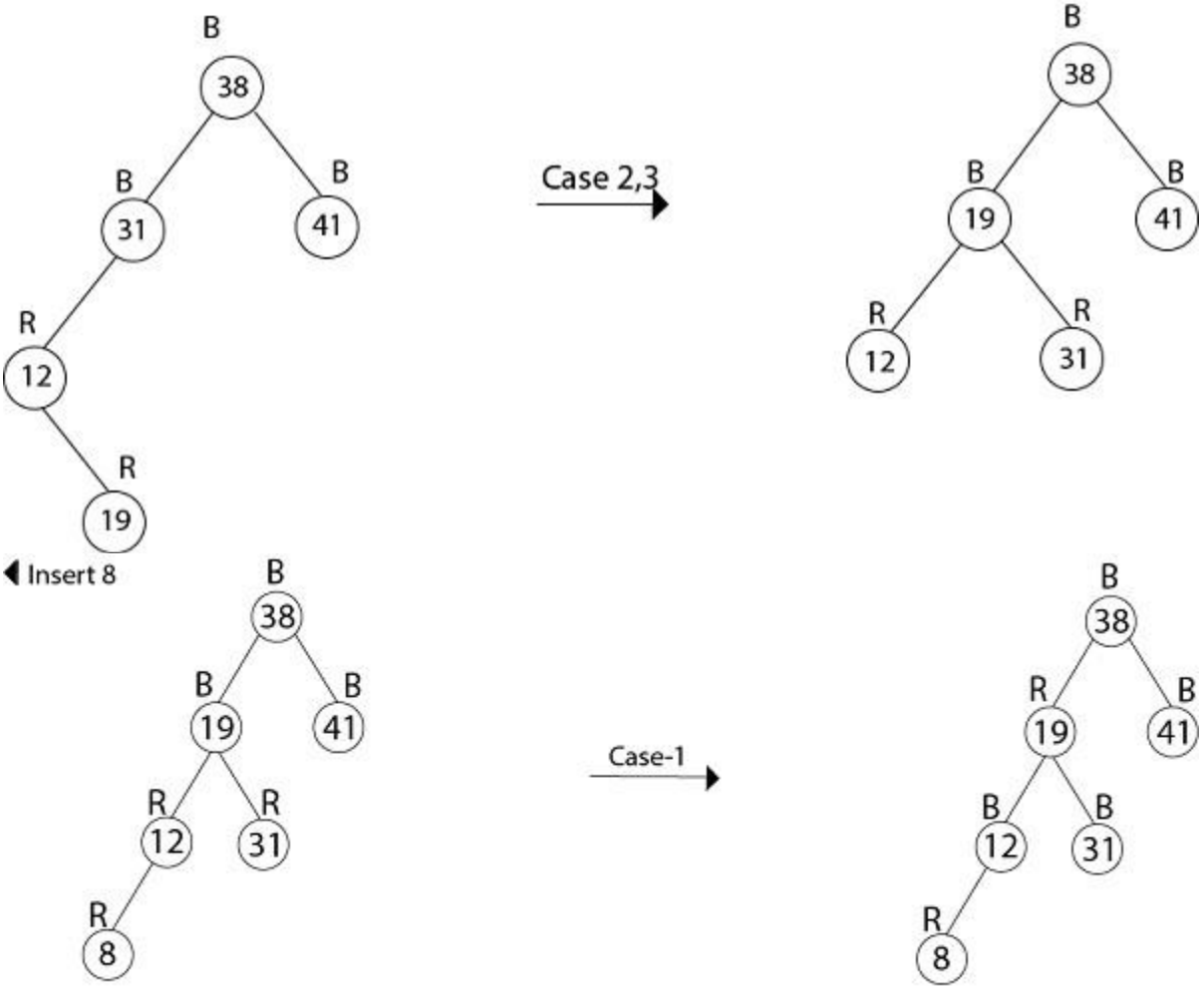



- Insert 31


- Insert 12




**Insert 19**




Thus the final tree is