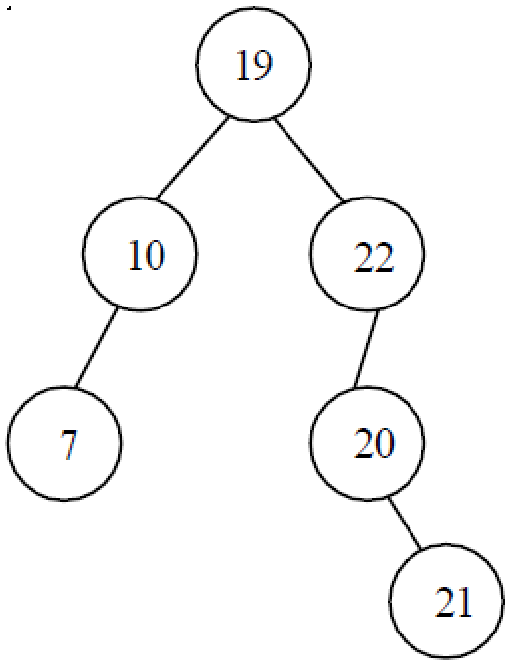
3.4 BINARY SEARCH TREE ADT

In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.



- Every BST is a Binary Tree
- But Not every binary tree is a BST

Declaration Routine for Binary Search Tree class TreeNode:

```
Node* createNode(int data)

{

    Node* newNode = (Node*)malloc(sizeof(Node));

    newNode->data = data;

    newNode->left = newNode->right = NULL;

    return newNode;

}
```

Insert : -

To insert the element X into the tree,

* Check with the root node T

* If it is less than the root,

Traverse the left subtree recursively until it reaches the T ->left equals to NULL. Then X is placed in T -> left.

* If X is greater than the root.

Traverse the right subtree recursively until it reaches the T -> right equals to NULL. Then X is placed in T-> Right.

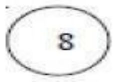
Routine to Insert into a Binary Search Tree

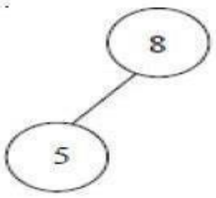
```
Node* insert(Node* root, int data)
{
    if (root == NULL)
   {
        return createNode(data);
    }
    if (data < root->data)
   {
        root->left = insert(root->left, data);
    }
 else if (data > root->data)
   {
        root->right = insert(root->right, data);
    }
    return root;
}
```

Example : -

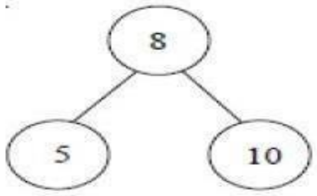
To insert 8, 5, 10, 15, 20, 18, 3

* First element 8 is considered as Root.

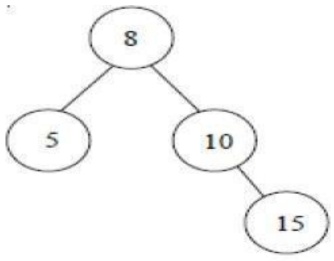




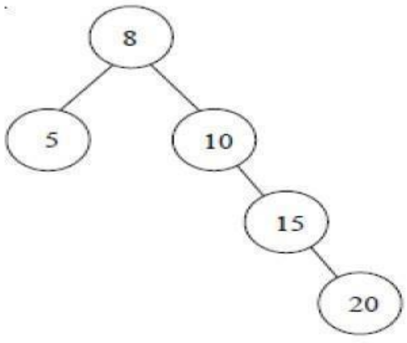
As 5 < 8, Traverse towards left



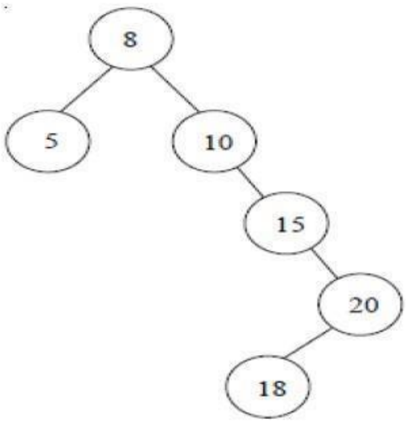
10 > 8, Traverse towards Right.



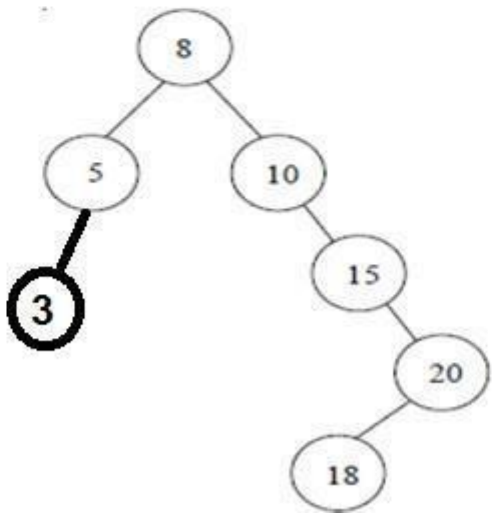
Similarly the rest of the elements are traversed.



After 20



After 18



AFTER 3

Find : -

- Check whether the root is NULL if so then return NULL.
- Otherwise, Check the value X with the root node value (i.e. T -> data)

  (1) If X is equal to T -> data, return T.

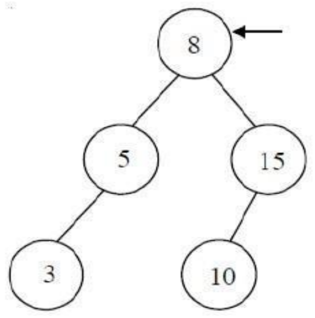
  (2) If X is less than T -> data, Traverse the left of T recursively.

  (3) If X is greater than T -> data, traverse the right of T recursively.

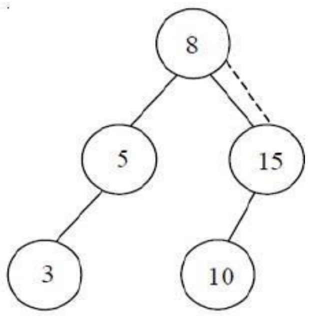
Routine for find Operation

```
Node* search(Node* root, int key)
{
    if (root == NULL || root->data == key)
        return root;

    if (key < root->data)
        return search(root->left, key);

    return search(root->right, key);
}
```

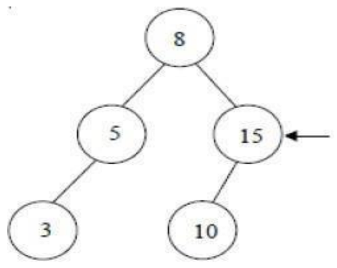
Example : - To Find an element 10 (consider, X = 10)



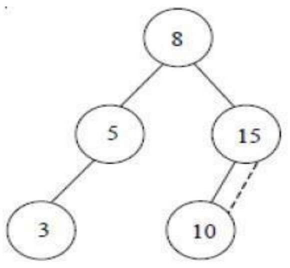
10 is checked with the Root



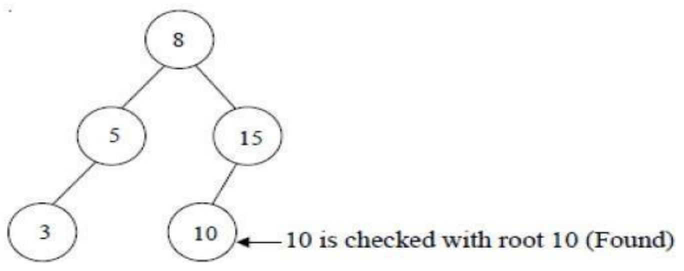
10 > 8, Go to the right child of 8



10 is checked with Root 15



10 < 15, Go to the left child of 15.



Find Min :

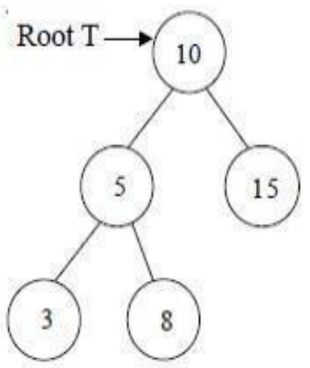
- This operation returns the position of the smallest element in the tree.
- To perform FindMin, start at the root and go left as long as there is a left child. The stopping point is the smallest element.

Recursive routine to find minimum

```
Node* findMin(Node* root)
{
   while (root && root->left != NULL)
```

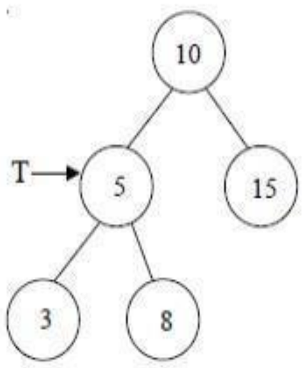
```
        root = root->left;
    return root;
}
```

Example : -



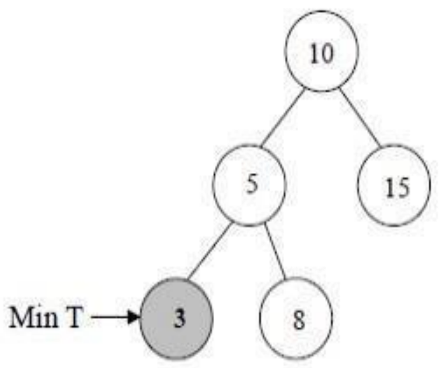
(a) T! = NULL and T→left!=NULL,

Traverse left



(b) T! = NULL and T→left!=NULL,

Traverse left



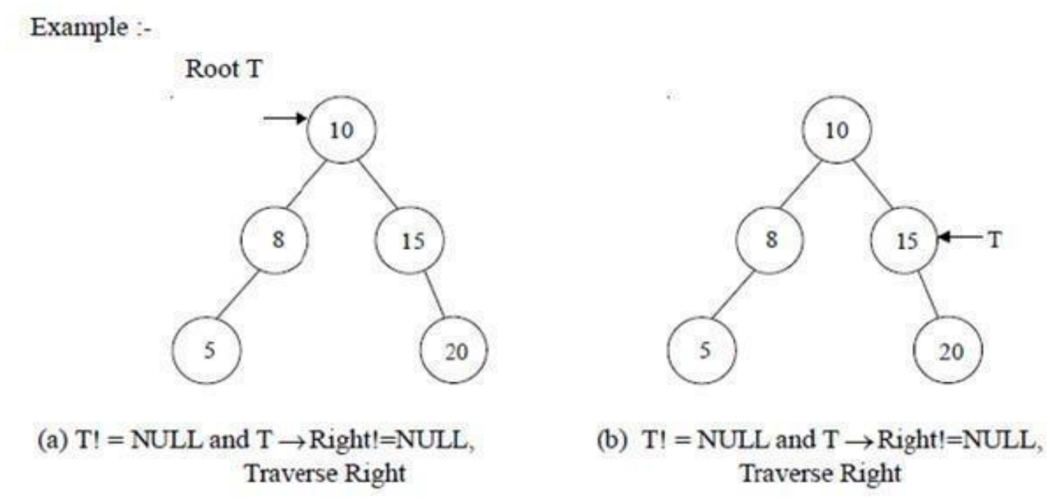
(c) Since T→left is Null, return T as a minimum element.

FindMax

FindMax routine return the position of largest elements in the tree. To perform a FindMax, start at the root and go right as long as there is a right child. The stopping point is the largest element.

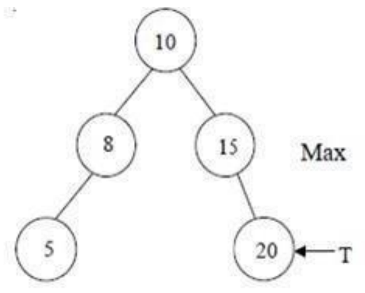
Recursive routine to find maximum

```
   Node* findMax(Node* root)
   {
        while (root && root->right != NULL)
            root = root->right;
        return root;
    }
```



(a) T! = NULL and T→Right!=NULL, Traverse Right

(b) T! = NULL and T→Right!=NULL, Traverse Right



(c) Since T→Right is NULL, return T as a Maximum element.

Delete :

Deletion operation is the complex operation in the Binary search tree. To delete an element, consider the following three possibilities.

CASE 1: Node to be deleted is a leaf node (i.e) No children.

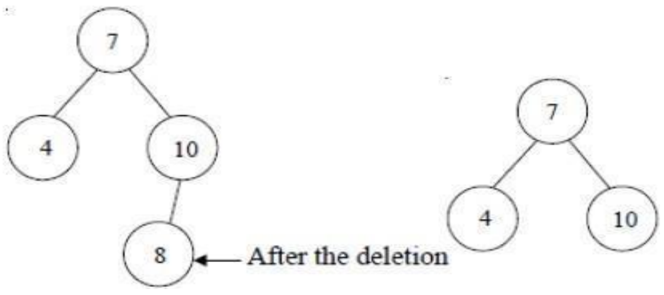
CASE 2: Node with one child.

CASE 3: Node with two children.

CASE 1 Node with no children (Leaf node)

If the node is a leaf node, it can be deleted immediately.

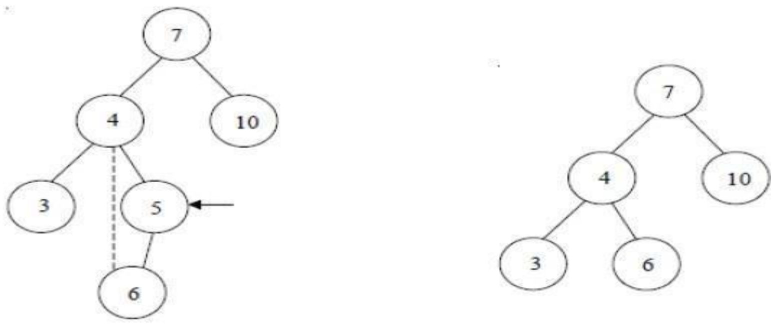
Delete : 8



CASE 2 : - Node with one child

If the node has one child, it can be deleted by adjusting its parent pointer that points to its child node.

To Delete 5



before deletion After deletion

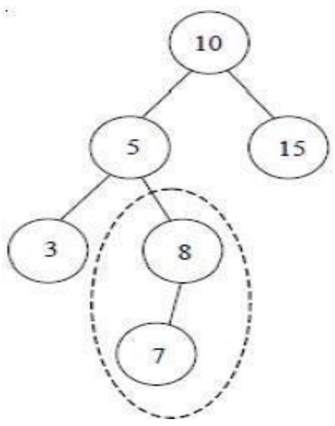
To delete 5, the pointer currently pointing the node 5 is now made to its child node 6.

Case 3: Node with two children

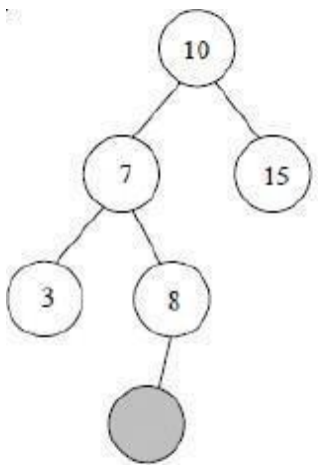
It is difficult to delete a node which has two children. The general strategy is to replace the data of the node to be deleted with its smallest data of the right subtree and recursively delete that node.

Example 1 :

To Delete 5 :



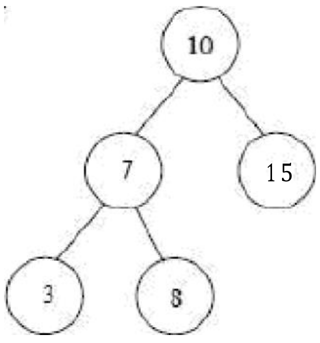
* The minimum element at the right subtree is 7.



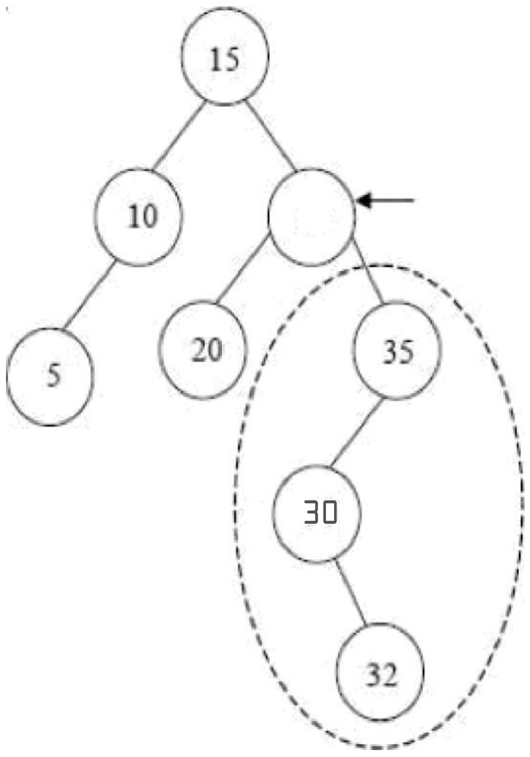
* Now the value 7 is replaced in the position of 5.

* Since the position of 7 is the leaf node delete immediately.

After deleting the node

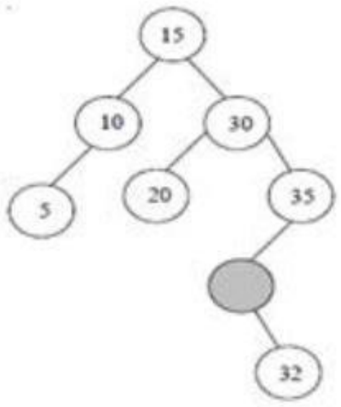


Example 2 – To Delete 25

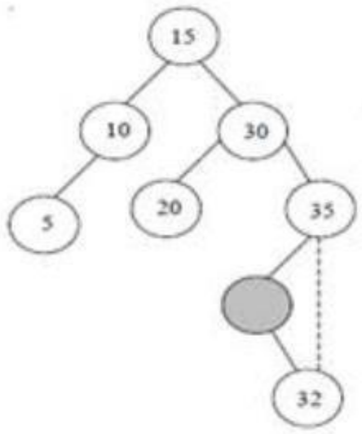


* The minimum element

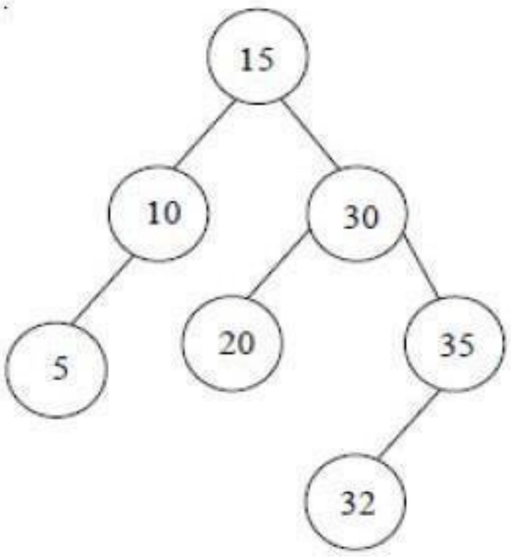
at the right subtree of 25 is 30



* The minimum value 30 is replaced in the position of 25



* Since this node has one child, the pointer currently pointing to this node is made to points to its child node 32



Binary Search Tree after deleting 25

Routine for deletion from BST Node*

```
deleteNode(Node* root, int key)
{
    if (root == NULL)
        return root;

    if (key < root->data)
        root->left = deleteNode(root->left, key);

    else if (key > root->data)
        root->right = deleteNode(root->right, key);

    else
    {
        // Node with one or no child
        if (root->left == NULL)
     {
            Node* temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL)
     {
            Node* temp = root->left;
            free(root);
            return temp;
        }
```