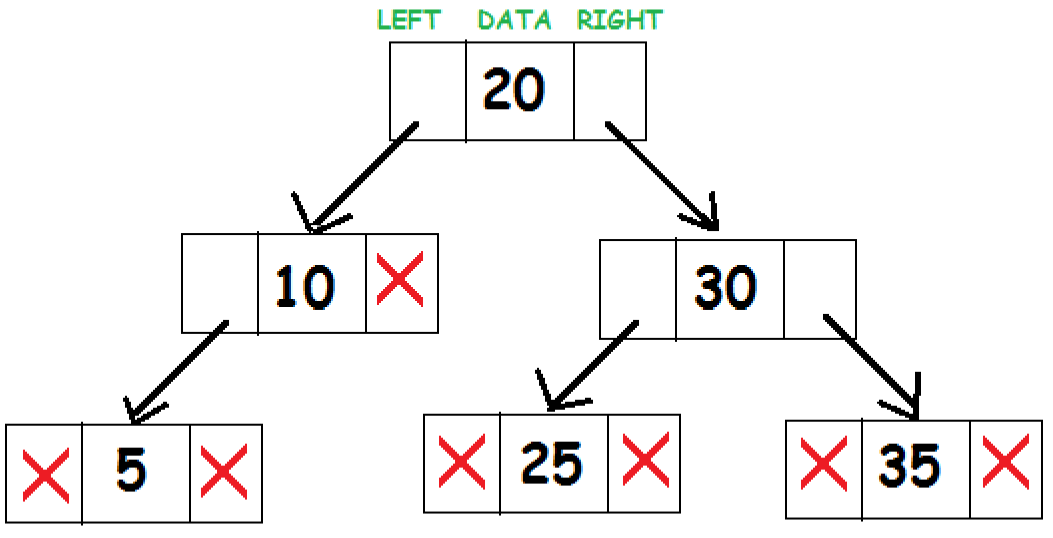
Binary Search Tree

A binary search tree is a useful data structure for fast addition and removal of data.

It is composed of nodes, which stores data and also links to upto two other child nodes. It is the relationship between the leaves linked to and the linking leaf, also known as the parent node, which makes the binary tree such an efficient data structure.

For a binary tree to be a binary search tree, the data of all the nodes in the left sub-tree of the root node should be less than the data of the root. The data of all the nodes in the right subtree of the root node should be greater than equal to the data of the root. As a result, the leaves on the farthest left of the tree have the lowest values, whereas the leaves on the right of the tree have the greatest values.

A representation of binary search tree looks like the following:



Consider the root node 20. All elements to the left of subtree(10, 5) are less than 20 and all elements to the right of subtree(25, 30, 35) are greater than 20.

Implementation of BST

First, define a struct as `tree_node`. It will store the data and pointers to left and right subtree.

```
struct tree_node

{

     int data;

     tree_node *left,  *right;

};
```
Copy

The node itself is very similar to the node in a linked list. A basic knowledge of the code for a linked list will be very helpful in understanding the techniques of binary trees.

It is most logical to create a binary search tree class to encapsulate the workings of the tree into a single area, and also making it reusable. The class will contain functions to insert data into the tree, search if the data is present and methods for traversing the tree.

```
class BST

{

     tree_node *root;

     void insert(tree_node* , int );

     bool search(int , tree_node* );

     void inorder(tree_node* );

     void preorder(tree_node* );

     void postorder(tree_node* );


     public:

     BST()

     {

          root = NULL;

     }

     void insert(int );

     bool search(int key);

    void inorder();

    void preorder();

    void postorder();

};
```

Copy

It is necessary to initialize root to NULL for the later functions to be able to recognize that it does not exist.

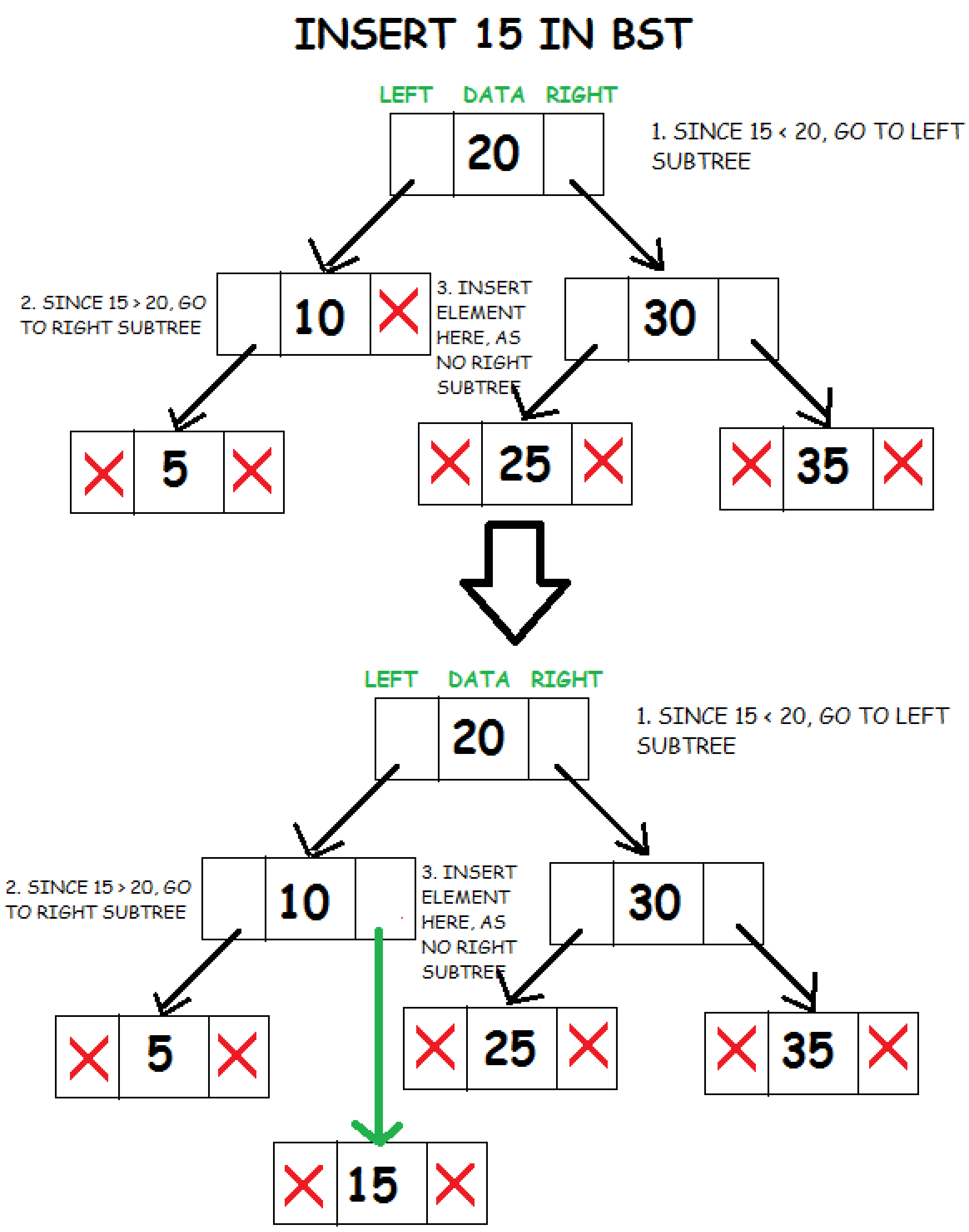
All the public members of the class are designed to allow the user of the class to use the class without dealing with the underlying design. The functions which will be called recursively are the ones which are private, allowing them to travel down the tree.

Insertion in a BST

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

- Compare data of the root node and element to be inserted.

- If the data of the root node is greater, and if a left subtree exists, then repeat step 1 with **root = root of left subtree**. Else,
- Insert element as left child of current root.
- If the data of the root node is greater, and if a right subtree exists, then repeat step 1 with **root = root of right subtree**.
- Else, insert element as right child of current root.




```
void BST :: insert(tree_node *node, int d)

{

    // element to be inserted is lesser than node’s data

    if(d < node->data)

    {

        // if left subtree is present

        if(node->left != NULL)

            insert(node->left, d);
```

```
        // create new node
        else
        {
            node->left = new tree_node;
            node->left->data = d;
            node->left->left = NULL;
            node->left->right = NULL;
        }
    }

    // element to be inserted is greater than node's data
    else if(d >= node->data)
    {
        // if left subtree is present
        if(node->right != NULL)
            insert(node->right, d);

        // create new node
        else
        {
            node->right = new tree_node;
            node->right->data = d;
            node->right->left = NULL;
            node->right->right = NULL;
        }
    }

}
```
Copy

Since the root node is a private member, we also write a public member function which is available to non-members of the class. It calls the private recursive function to insert an element and also takes care of the case when root node is NULL.

```
void BST::insert(int d)

{

     if(root!=NULL)

                  insert(root, d);

     else

    {

        root = new tree_node;

        root->data = d;

        root->left = NULL;

        root->right = NULL;

    }

}
```
Copy

Searching in a BST

The search function works in a similar fashion as insert. It will check if the key value of the current node is the value to be searched. If not, it should check to see if the value to be searched for is less than the value of the node, in which case it should be recursively called on the left child node, or if it is greater than the value of the node, it should be recursively called on the right child node.

- Compare data of the root node and the value to be searched.
- If the data of the root node is greater, and if a left subtree exists, then repeat step 1 with **root = root of left subtree**. Else,
- If the data of the root node is greater, and if a right subtree exists, then repeat step 1 with **root = root of right subtree**. Else,
- If the value to be searched is equal to the data of root node, return true.
- Else, return false.

```
bool BST::search(int d, tree_node *node)

{

     bool ans = false;



     // node is not present
```

```
     if(node == NULL)

          return false;


     // Node's data is equal to value searched

    if(d == node->data)

          return true;


     // Node's data is greater than value searched

    else if(d < node->data)

          ans = search(d, node->left);


     // Node's data is lesser than value searched

    else

          ans = search(d, node->right);


    return ans;

}
```

Copy

Since the root node is a private member, we also write a public member function which is available to non-members of the class. It calls the private recursive function to search an element and also takes care of the case when root node is NULL.

```
bool BST::search(int d)

{

     if(root ==  NULL)

          return false;

     else

          return  search(d, root);

}
```

Copy

Traversing in a BST

There are mainly three types of tree traversals:

1. Pre-order Traversal:

In this technique, we do the following :

- Process data of root node.
- First, traverse left subtree completely.
- Then, traverse right subtree.

```
void BST :: preorder(tree_node *node)

{

    if(node !=  NULL)

    {

        cout<<node->data<<endl;

        preorder(node->left);

        preorder(node->right);

    }

}
```

Copy

2. Post-order Traversal

In this traversal technique we do the following:

- First traverse left subtree completely.
- Then, traverse right subtree completely.
- Then, process data of node.

```
void BST :: postorder(tree_node *node)

{

    if(node !=  NULL)

    {

      postorder(node->left);

      postorder(node->right);

       cout<<node->data<<endl;

    }

}
```

Copy

3. In-order Traversal

In in-order traversal, we do the following:

- First process left subtree.
- Then, process current root node.
- Then, process right subtree.

```
void BST :: inorder(tree_node *node)

{

     if(node !=  NULL)

     {

        inorder(node->left);

         cout<<node->data<<endl;

        inorder(node->right);

     }

}
```

Copy

The in-order traversal of a binary search tree gives a sorted ordering of the data elements that are present in the binary search tree. This is an important property of a binary search tree.

Since the root node is a private member, we also write public member functions which is available to non-members of the class. It calls the private recursive function to traverse the tree and also takes care of the case when root node is NULL.

```
void BST :: preorder()

{

     if(root ==  NULL)

          cout<<"TREE IS  EMPTY\n";

     else

          preorder(root);

}



void BST :: postorder()

{

     if(root ==  NULL)

          cout<<"TREE IS  EMPTY\n";
```

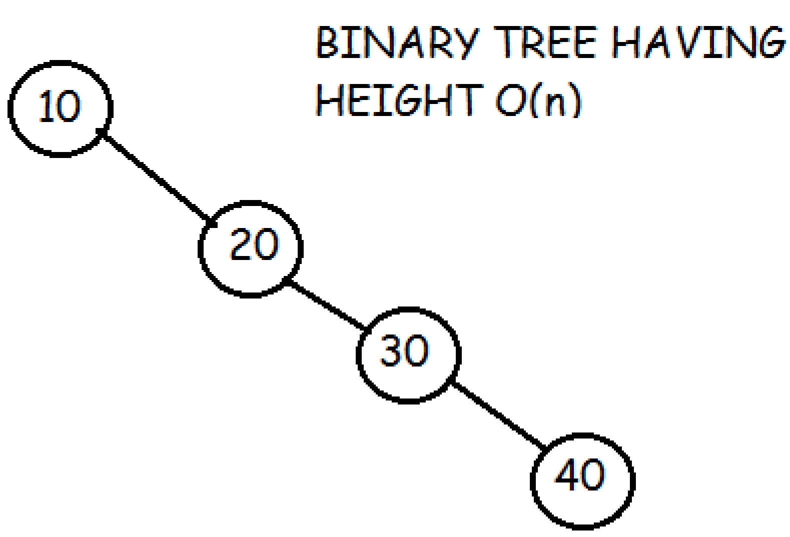
```
    else

        postorder(root);

}


void BST :: inorder()

{

    if(root == NULL)

        cout<<"TREE IS EMPTY\n";

    else

        inorder(root);

}
```

Copy

Complexity Analysis

ALGORITHM	AVERAGE CASE	WORST CASE
Space	O(n)	O(n)
Insert	O(log n)	O(n)
Search	O(log n)	O(n)
Traverse	O(n)	O(n)

The time complexity of search and insert rely on the height of the tree. On average, binary search trees with n nodes have **O(log n)** height. However in the worst case the tree can have a height of **O(n)** when the unbalanced tree resembles a linked list. For example in this case :



Traversal requires **O(n)** time, since every node must be visited.