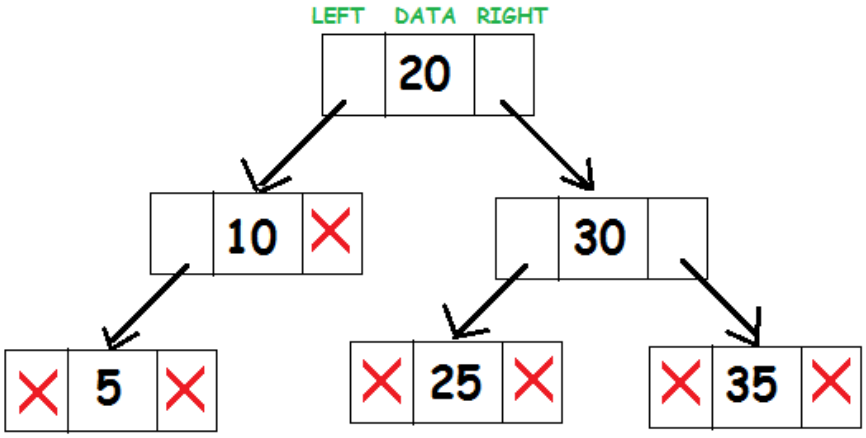
Binary Search Tree

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 up to 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.

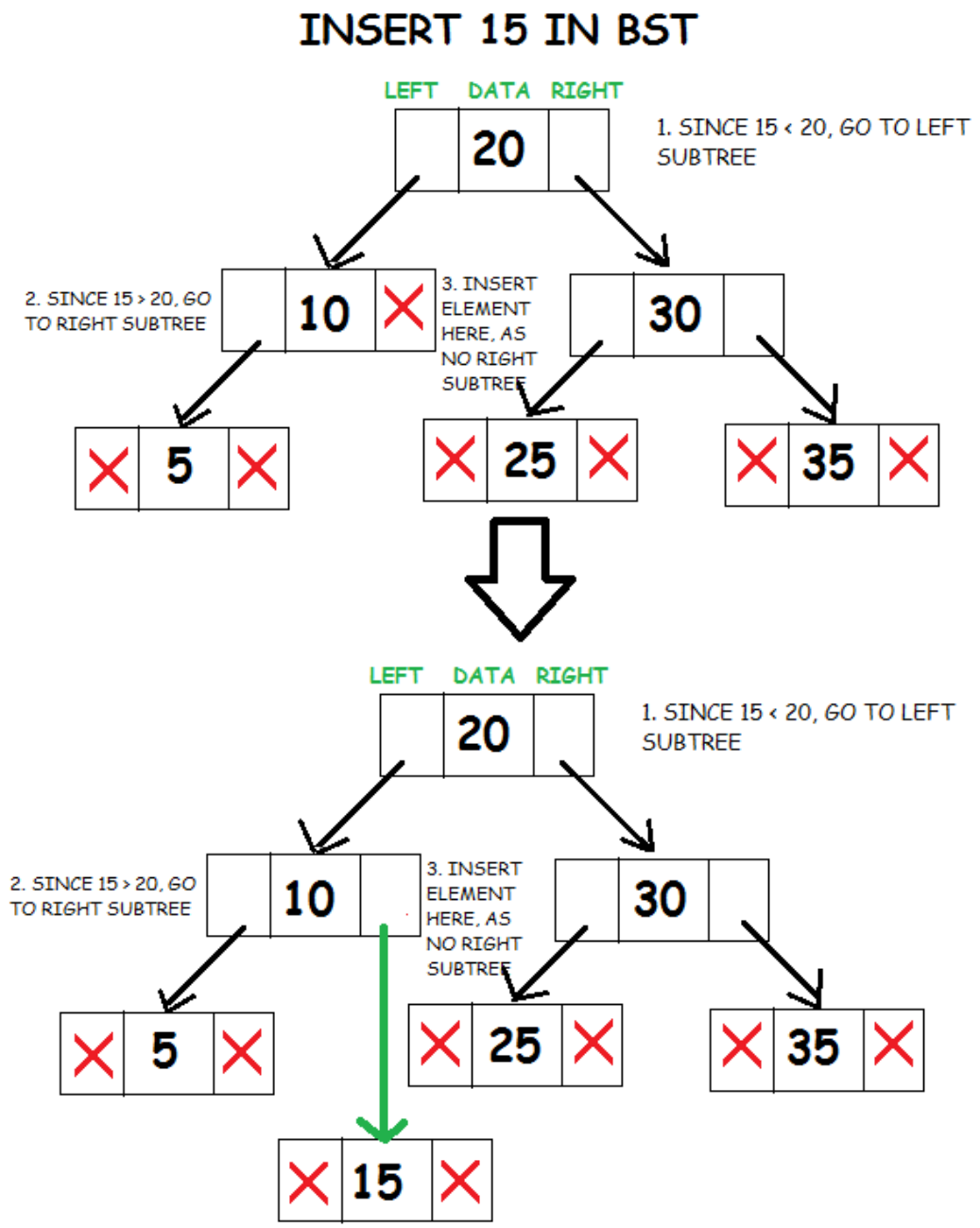
```
Struct tree_node
{
      int data;
      tree_node *left, *right;
};
```

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.

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.



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.

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 preorder(tree_node *nod)
{
      if(nod !=  NULL)
     {
            printf("%d",nod->data);
            preorder(nod->left);
            preorder(nod->right);
     }
}
```

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 postorder(tree_node *nod)
{
      if(nod !=  NULL)
      {
         postorder(nod->left);
         postorder(nod>right);
         printf("%d",nod->data);
      }
}
```

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.

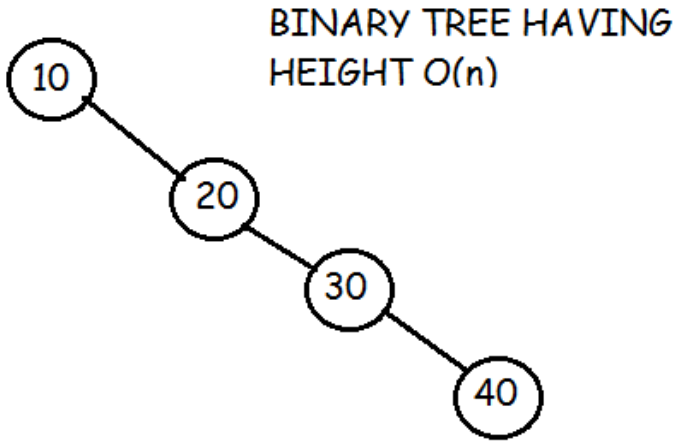
```
void inorder(tree_node *nod)
{
      if(noe !=  NULL)
      {
         inorder(nod->left);
         printf("%d",nod->data);
         inorder(nod->right);
      }
}
```

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.

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.