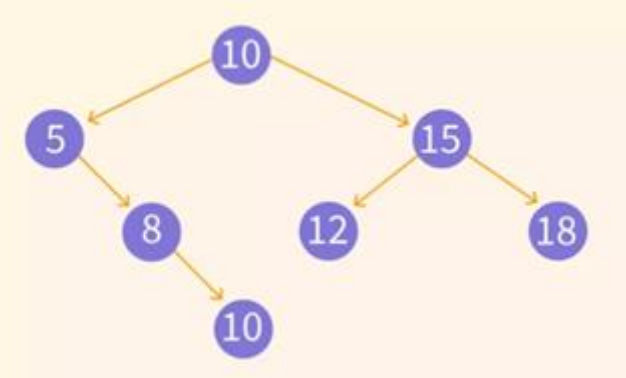
**Name – Prerna Sunil Jadhav** **SAP ID - 60004220127**

**Experiment No – 08**

## AIM: Implementation of BST




## Theory:

- Binary Search Tree is a node-based binary tree data structure which has the following properties:
  - The left subtree of a node contains only nodes with keys lesser than the node's key.
  - The right subtree of a node contains only nodes with keys greater than the node's key.
  - This means everything to the left of the root is less than the value of the root and everything to the right of the root is greater than the value of the root. Due to this performing, a binary search is very easy.
  - The left and right subtree each must also be a binary search tree.
  - There must be no duplicate nodes(BST may have duplicate values with different handling approaches)

## Algorithm:

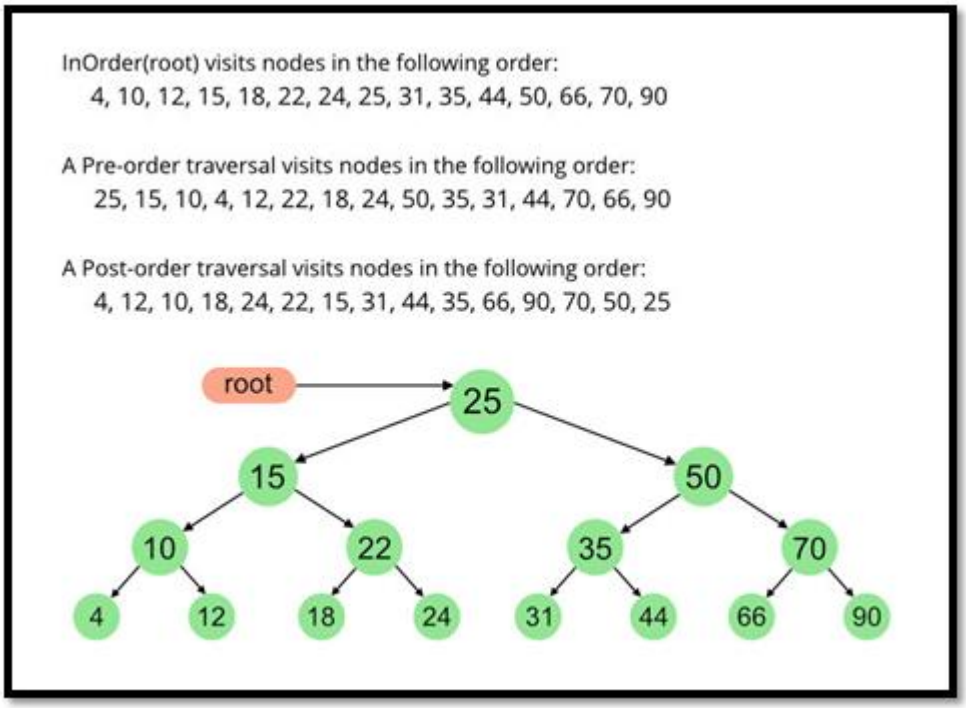
```
There are three ways which we use to traverse a tree –
In-order Traversal
Pre-order Traversal
Post-order Traversal

In-order Traversal:
Until all nodes are traversed –
Step 1 – Recursively traverse left subtree.
Step 2 – Visit root node.
Step 3 – Recursively traverse right subtree.

Pre-order Traversal:
Until all nodes are traversed –
Step 1 – Visit root node.
Step 2 – Recursively traverse left subtree.
Step 3 – Recursively traverse right subtree.

Post-order Traversal:
Until all nodes are traversed –
Step 1 – Recursively traverse left subtree.
Step 2 – Recursively traverse right subtree.
Step 3 – Visit root node.
```

## Example:




## Program:

```c
#include <stdio.h>
#include <stdlib.h>
typedef struct node
{
    int data;
    struct node *left, *right;
} node;
void btree(int a[], int n);
node *create_tree(int a[], int n);
void inorder(struct node *root);

void main()
{
    printf("Prerna Sunil Jadhav - 60004220127\n");

    int n, i;
    int a[100];
    printf("Enter number of nodes: \n");
    scanf("%d", &n);
    printf("Enter elements: \n");
    for (i = 0; i < n; i++)
    {
        scanf("%d", &a[i]);
    }
```

```
    btree(a, n);
}
void btree(int a[], int n)
{
    node *root;
    root = create_tree(a, n);
    printf("\nInorder: ");
    inorder(root);
    printf("\nPostorder: ");
    posorder(root);
    printf("\nPreorder: ");
    preorder(root);
}
node *create_tree(int a[], int n)
{
    node *p, *prev, *ptr, *root;
    int i, flag;
    root = (node *)malloc(sizeof(node));
    root->data = a[0];
    root->left = NULL;
    root->right = NULL;
    for (i = 1; i < n; i++)
    {
        ptr = (node *)malloc(sizeof(node));
        ptr->data = a[i];
        ptr->left = NULL;
        ptr->right = NULL;
        p = root;
        while (p != NULL)
        {
            prev = p;
            if (a[i] < p->data)
            {
                p = p->left;
                flag = 1;
            }
            else
            {
                p = p->right;
                flag = 0;
            }
        }
        if (flag == 1)
            prev->left = ptr;
        else
            prev->right = ptr;
```

```
    }
    return (root);
}
void inorder(node *root)
{
    if (root != NULL)
    {
        inorder(root->left);
        printf("%d\t", root->data);
        inorder(root->right);
    }
}
void preorder(node *root)
{
    if (root != NULL)
    {
        printf("%d\t", root->data);
        preorder(root->left);
        preorder(root->right);
    }
}
void posorder(node *root)
{
    if (root != NULL)
    {
        posorder(root->left);
        posorder(root->right);
        printf("%d\t", root->data);
    }
}
```

**OUTPUT:**

```
Prerna Sunil Jadhav - 60004220127
Enter number of nodes:
5
Enter elements:
12
54
9
1
56

Inorder: 1      9       12      54      56
Postorder: 1    9       56      54      12
Preorder: 12    9       1       54      56
```

**Conclusion**:

- Unlike Binary Search, which works only for a fixed data set, Binary Search Trees can be used for dynamic data sets as well.
- It can be used to find the sorted order of a dynamic data set.
- If elements are inserted/deleted randomly, it is actually faster to do it in a binary search tree than in linked lists and arrays.
- The overall shape of the tree depends on the order of insertion. If data is inserted in a sorted order or the tree becomes heavier(skewed) in one direction, the insert/search operations become expensive.
- A lot of space is required as we need to store the left and right child of each node.
- The plain implementation of Binary Search Tree is not much used since it cannot guarantee logarithmic complexity. Hence, we have to use self balancing implementations which are a bit more complex to understand and implement.