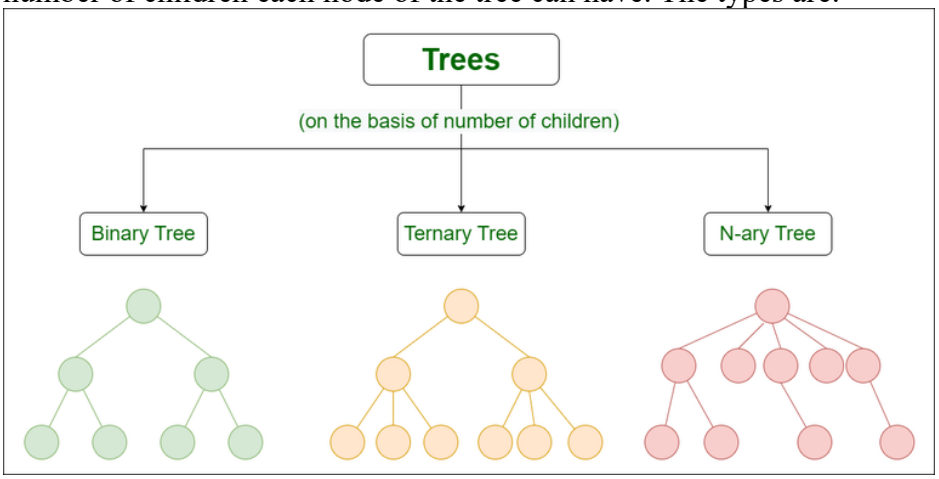
Bharatiya Vidya Bhavan's
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Autonomous Institute Affiliated to University of Mumbai)
Munshi Nagar, Andheri (W), Mumbai – 400 058.
COMPS Department

Experiment	5
Aim	**Tree data structures: Write a program to construct a binary search tree, insert an element in BST, delete an element from BST and traverse it.**
Name	David Daniels
UID	2023300038
Class	Div -A
Batch	C
Date of Submission	2-10-24

Theory

Tree data structure is a specialized data structure to store data in hierarchical manner. It is used to organize and store data in the computer to be used more effectively. It consists of a central node, structural nodes, and sub-nodes, which are connected via edges. We can also say that tree data structure has roots, branches, and leaves connected. It is a non-linear Abstract data type (ADT). It has various functions such as insert, delete, search and traverse.

Tree data structure can be classified into three types based upon the number of children each node of the tree can have. The types are:



- **Binary tree:** In a binary tree, each node can have a maximum of two children linked to it. Some common types of binary trees include full binary trees, complete binary trees, balanced binary trees, and degenerate or pathological binary trees.
- **Ternary Tree:** A Ternary Tree is a tree data structure in which each node has at most three child nodes, usually distinguished as "left", "mid" and "right".
- **N-ary Tree or Generic Tree:** Generic trees are a collection of nodes where each node is a data structure that consists of records

and a list of references to its children(duplicate references are not allowed). Unlike the linked list, each node stores the address of multiple nodes.

We shall discuss more about Binary trees. Binary tree has many types , depending on number of children of each node and the completion of levels.

However, There are also many special cases of a binary tree, such as ,

1. Binary Search Tree
2. AVL Tree
3. Red Black Tree
4. B Tree
5. B+ Tree
6. Segment Tree

We shall discuss about Binary search tree (BST)
A **Binary Search Tree** is a data structure derived from binary trees used in computer science for organizing and storing data in a sorted manner. Each node in a **Binary Search Tree** has at most two children, a **left** child and a **right** child, with the **left** child containing values less than the parent node and the **right** child containing values greater than the parent node. This hierarchical structure allows for efficient **searching**, **insertion**, and **deletion** operations on the data stored in the tree.

Basic Operations Of Binary Search Tree Data Structure:

- **Insert** − Inserts data in a tree.
- **Search** − Searches specific data in a tree to check whether it is present or not.
- **Delete** – Deletes a node in a tree
- **Traversal**: - Includes Inorder, Preorder and Postorder Traversal

1. Searching a node in BST:

Searching in BST means to locate a specific node in the data structure. In Binary search tree, searching a node is easy because of its a specific order

2. Insert a node into a BST:

A new key is always inserted at the leaf. Start searching a key from the root till a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

3. Delete a Node of BST:

It is used to delete a node with specific key from the BST and return the new BST.

4. Traversals in BST:

1) Inorder Traversal:

At first traverse **left subtree** then visit the **root** and then traverse the **right subtree**.

2) **Preorder Traversal:**

At first visit the **root** then traverse **left subtree** and then traverse the **right subtree**.

3) **Postorder Traversal:**

At first traverse **left subtree** then traverse the **right subtree** and then visit the **root**.

Advantages of Binary Search Tree (BST):

- **Efficient searching:** O(log n) time complexity for searching with a balanced BST
- **Ordered structure:** Elements are stored in sorted order, making it easy to find the next or previous element
- **Dynamic insertion and deletion:** Elements can be added or removed efficiently
- **Balanced structure:** Balanced BSTs maintain a logarithmic height, ensuring efficient operations
- **Doubly Ended Priority Queue**: In BSTs, we can maintain both maximum and minimum efficiently
- **In-order traversal:** BSTs can be traversed in-order, which visits the left subtree, the root, and the right subtree. This can be used to sort a dataset.

Disadvantages of Binary Search Tree (BST):

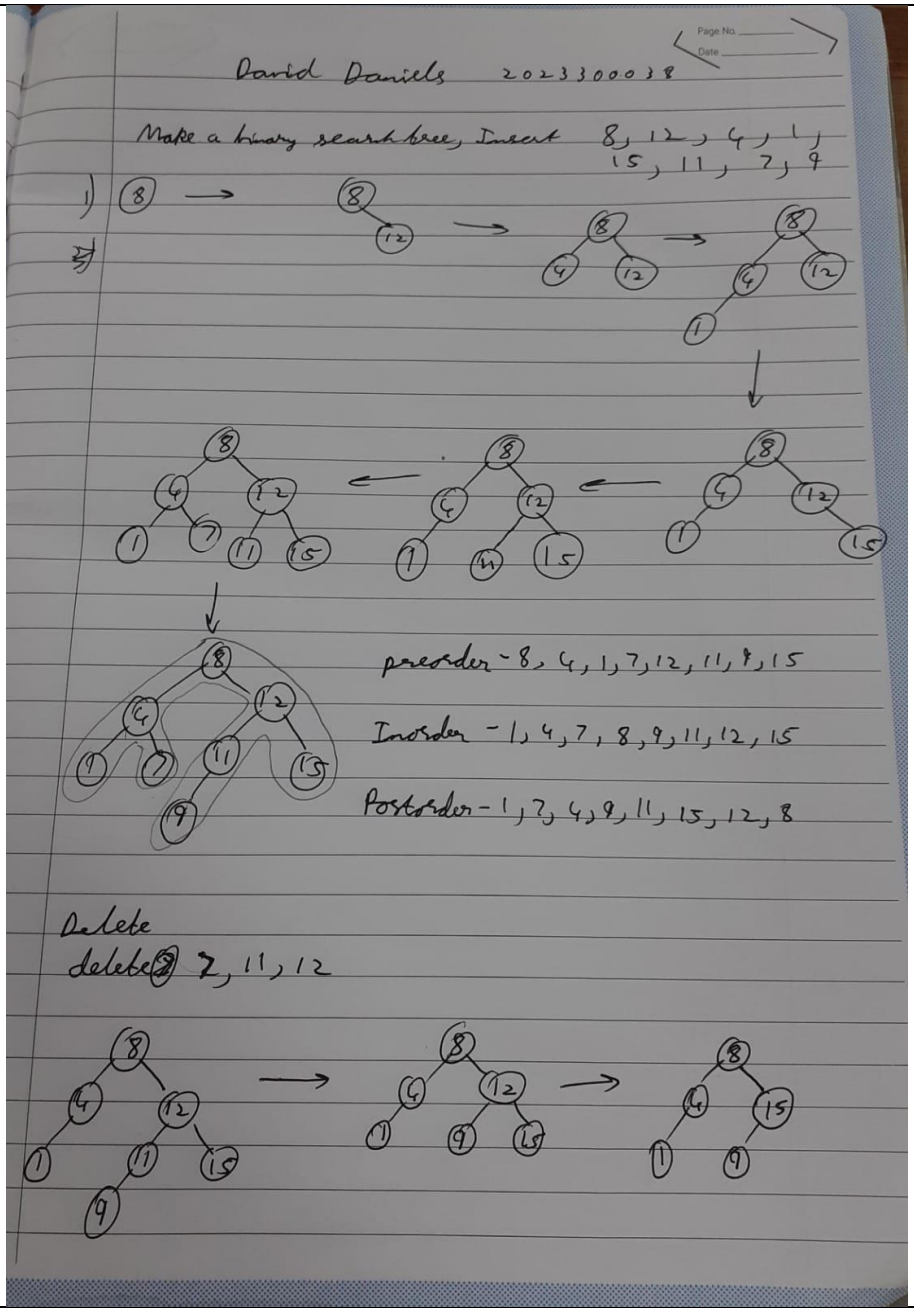
- **Skewed trees:** If a tree becomes skewed, the time complexity of search, insertion, and deletion operations will be O(n) instead of O(log n), which can make the tree inefficient
- **Worst-case time complexity:** In the worst case, BSTs can have a linear time complexity for searching and insertion if all elements are sorted.
- **Memory overhead:** BSTs require additional memory to store pointers to child nodes
- The implementation and manipulation of trees can be complex

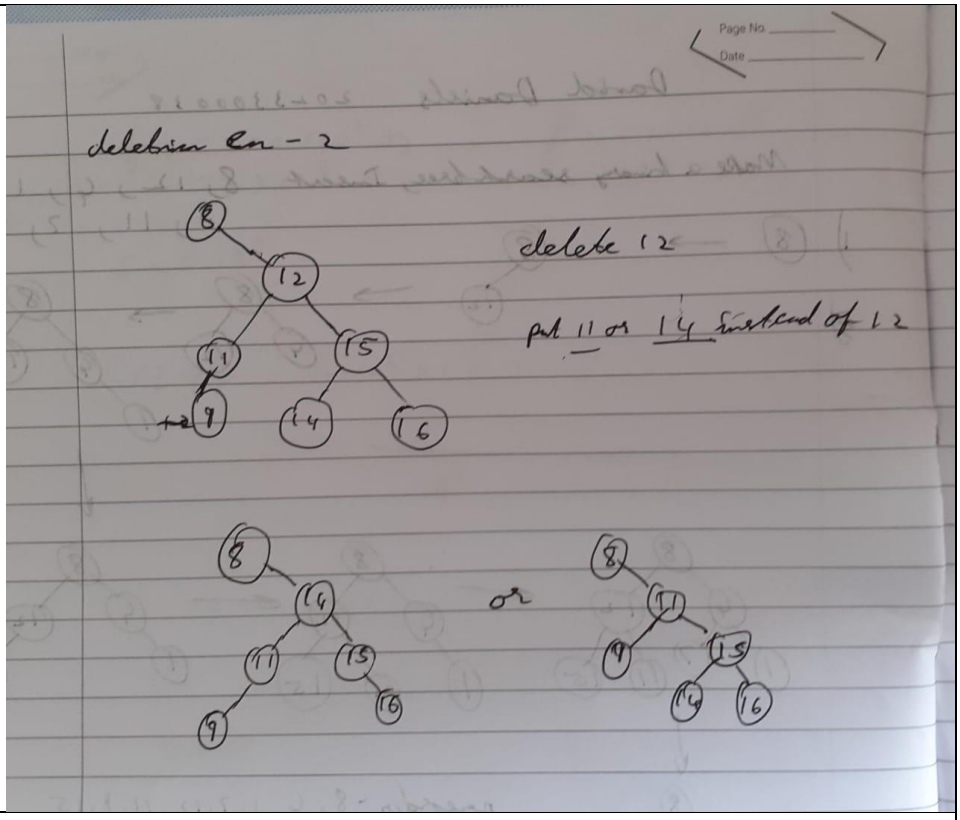
Applications of BST

1. A Self-Balancing Binary Search Tree is used to maintain sorted stream of data. For example, suppose we are getting online orders placed and we want to maintain the live data (in RAM) in sorted order of prices. For example, we wish to know number of items purchased at cost below a given cost at any moment. Or we wish to know number of items purchased at higher cost than given cost.
2. A Self-Balancing Binary Search Tree is used to implement doubly ended priority queue. With a Binary Heap, we can either implement a priority queue with support of extractMin() or with extractMax(). If we wish to support both the operations, we use a Self-Balancing Binary Search Tree to do both in O(Log n)

	3. There are many more algorithm problems where a Self-Balancing BST is the best suited data structure, like count smaller elements on right, Smallest Greater Element on Right Side, etc. 4. A BST can be used to sort a large dataset. By inserting the elements of the dataset into a BST and then performing an in-order traversal, the elements will be returned in sorted order. When compared to normal sorting algorithms, the advantage here is, we can later insert / delete items in O(Log n) time.
Algorithm	**1. Searching a node in BST:** Searching in BST means to locate a specific node in the data structure. In Binary search tree, searching a node is easy because of its a specific order. The steps of searching a node in Binary Search tree are listed as follows – 1. First, compare the element to be searched with the root element of the tree. o If root is matched with the target element, then return the node's location. o If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree. o If it is larger than the root element, then move to the right subtree. 2. Repeat the above procedure recursively until the match is found. 3. If the element is not found or not present in the tree, then return NULL. **2. Inserting a node in BST:** A new key is always inserted at the leaf by maintaining the property of the binary search tree. We start searching for a key from the root until we hit a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node. The below steps are followed while we try to insert a node into a binary search tree: • Initialize the current node with root node • Compare the **key** with the current node. • **Move left** if the **key** is less than or equal to the current node value. • **Move right** if the **key** is greater than current node value. • Repeat steps 2 and 3 until you reach a leaf node. • Attach the **new key** as a left or right child based on the comparison with the leaf node's value. **3. Deleting a node in BST:** We must take 3 cases to understand deletion ,

	Case 1. Delete a Leaf Node in BST • Simply assign node to null. Now the parent node will point to null **Case 2. Delete a Node with Single Child in BST** • Copy the child node to parent node and delete it **Case 3. Delete a Node with Both Children in BST** Deleting a node with both children is more complex. Here we have to delete the node is such a way, that the resulting tree follows the properties of a BST. We have to inorder successor or predessor of the node. Copy contents of it to the node, and delete it. **4. Traversals in a BST** **1) Inorder** • Traverse left subtree • Visit the root and print the data. • Traverse the right subtree **2) Preorder** • Visit the root and print the data. • Traverse left subtree • Traverse the right subtree **3) Postorder** • Traverse left subtree • Traverse the right subtree • Visit the root and print the data.

Problem Solving	



Program(Code)

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
typedef struct node
{
   int data;
   struct node * left;
   struct node * right;
}node;

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

}

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

```
    }

}

void inorder_print_children(node * root)
{
   if (root!=NULL)
   {
      if (root->left!=NULL)
      {
         printf("Left child of %d is %d \n",root->data ,root->left->data);
      }
      inorder(root->left);
      printf("%d \n",root->data);
      if (root->right!=NULL)
      {
         printf("Right child of %d is %d \n",root->data ,root->right-
>data);
      }
      inorder(root->right);
   }

}

void postorder(node * root)
{
   if (root!=NULL)
   {
      postorder(root->left);
      postorder(root->right);
      printf("%d ",root->data);
   }

}

node * create_node(int d)
{
   node * ptr = (node * ) malloc(sizeof(node));
   ptr->left=NULL;
   ptr->right=NULL;
   ptr->data=d;
   return ptr;
}

void insert(node * root , int data)
{
   node * prev=NULL;
   node * n= create_node(data);
   while (root!=NULL)
   {
```

```
        prev=root;
        if (root->data==data)
        {
           //duplicate
           return;
        }
        else if (data < root->data)
        {
           root=root->left;
        }
        else
        {
           root=root->right;
        }
     }

     if (prev->data > data)
     {
        //insert left , root is now leaf
        printf("The left node of %d is %d \n", prev->data , data);
        prev->left=n;
     }
     else
     {
        printf("The Right node of %d is %d \n", prev->data , data);
        prev->right=n;
     }
}

node * in_order_predecessor(node * root)
{
   // give rightmost value of left subtree
   root=root->left;
   while (root->right!=NULL)
   {
      root=root->right;
   }
   return root;
}

node * delete(node * root , int value)
{
   node * ipre=NULL;
   if (root==NULL)
   {
      return NULL;
   }

   // no children
   if (root->left==NULL && root->right==NULL)
```

```
    {
        root=NULL;
        free(root);
        return NULL;
    }

    //recursive left and right
    if (root->data > value)
    {
        root->left = delete(root->left,value);
    }
    else if (root->data < value)
    {
        root->right=delete(root->right,value);
    }
    else
    {
        // we are at node
        // find the inorder predeccsor or successor

        //assume inorder predecessor
        ipre=in_order_predecessor(root);
        root->data=ipre->data;
        root->left= delete(root->left , ipre->data);
    }
    printf("Deleting %d \n",root->data);
    return root;
}

void choice(node * root)
{
    int choice=0,num=0;
    printf(" Enter 1 for Preorder \n Enter 2 for Inorder \n Enter 3 for
Postorder \n Enter 4 to Insert \n Enter 5 to delete \n Enter 6 to print
Inorder with Children \n Enter 0 to Exit \n");

    while (true)
    {

    printf("Enter choice \n");
        scanf("%d",&choice);
        if (choice==0)
        {
            printf("\nGoodbye\n");
            break;
        }
    switch (choice)
    {
    case 1:
        {
```

```
            printf("Preorder Is: \n");
            preorder(root);
            printf("\n\n");
            break;
        }
    case 2:
        {
            printf("Inorder Is: \n");
            inorder(root);
            printf("\n\n");
            break;
        }
    case 3:
        {
            printf("Postorder Is: \n");
            postorder(root);
            printf("\n\n");
            break;
        }
    case 4:
        {
            printf("Enter number to Insert \n");
            scanf("%d", &num);
            insert(root,num);
            printf("\n\n");
            break;
        }
    case 5:
        {
            printf("Enter number to Delete \n");
            scanf("%d", &num);
            delete(root,num);
            printf("\n\n");
            break;
        }
    case 6:
        {
            printf("Inorder with children is: \n");
            inorder_print_children(root);
            printf("\n\n");
            break;
        }
    default:
        {
            printf("Error");
            break;
        }
    }
}
}
```

```
int main(int argc, char const *argv[])
{
    int num=0;
    printf("Enter root node: \n");
    scanf("%d", &num);
    node * root = create_node(num);
    choice(root);
    return 0;
}
```

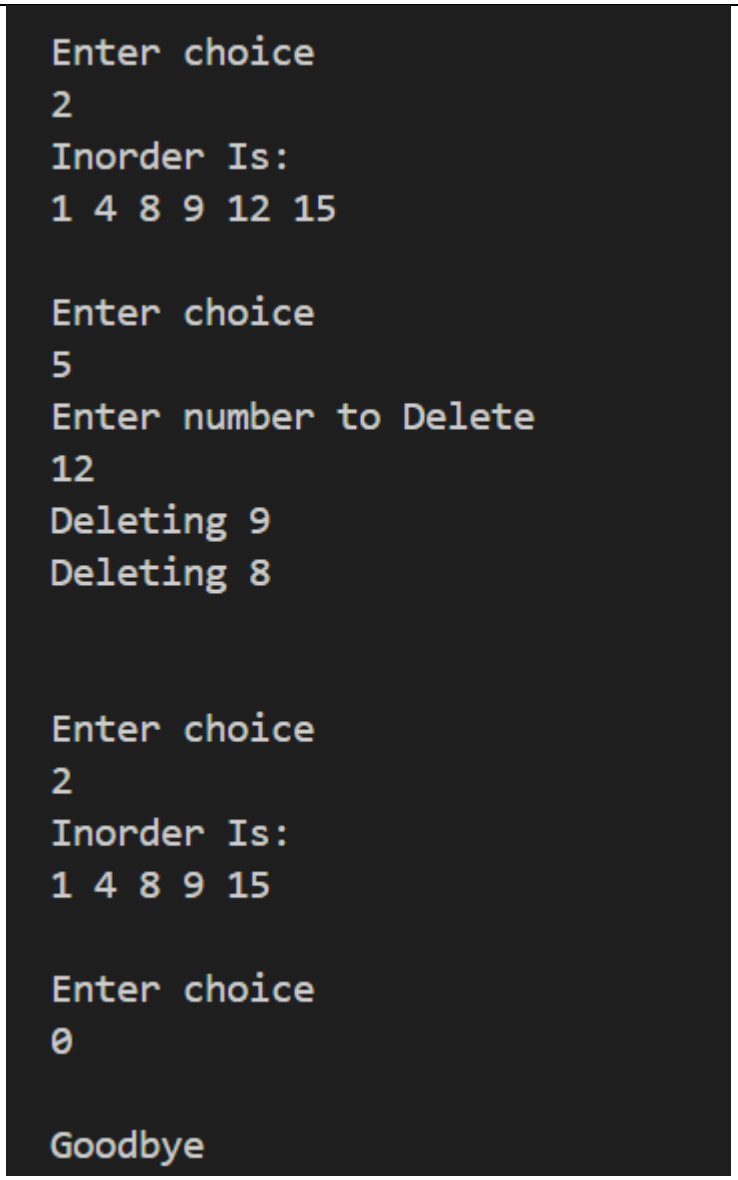
Output

```
Enter root node:
8
 Enter 1 for Preorder
 Enter 2 for Inorder
 Enter 3 for Postorder
 Enter 4 to Insert
 Enter 5 to delete
 Enter 6 to print Inorder with Children
 Enter 0 to Exit
Enter choice
4
Enter number to Insert
12
The Right node of 8 is 12


Enter choice
4
Enter number to Insert
4
The left node of 8 is 4


Enter choice
4
Enter number to Insert
1
The left node of 4 is 1
```

```
Enter choice
4
Enter number to Insert
15
The Right node of 12 is 15


Enter choice
4
Enter number to Insert
11
The left node of 12 is 11


Enter choice
4
Enter number to Insert
7
The Right node of 4 is 7


Enter choice
4
Enter number to Insert
9
The left node of 11 is 9
```

```
Enter choice
1
Preorder Is:
8 4 1 7 12 11 9 15

Enter choice
2
Inorder Is:
1 4 7 8 9 11 12 15

Enter choice
3
Postorder Is:
1 7 4 9 11 15 12 8

Enter choice
6
Inorder with children is:
Left child of 8 is 4
1 4 7 8
Right child of 8 is 12
9 11 12 15
```

```
Enter choice
5
Enter number to Delete
7
Deleting 4
Deleting 8


Enter choice
2
Inorder Is:
1 4 8 9 11 12 15

Enter choice
5
Enter number to Delete
11
Deleting 9
Deleting 12
Deleting 8
```

	
Conclusion	Thus we have implemented a binary search tree (BST) that allows for insertion, deletion, and traversal of elements. By adhering to the BST properties, our program ensures that each node maintains its left and right children according to their values. The traversal methods, including in-order, pre-order, and post-order, allows us to utilise the structure.