Aryan Nanda **221070003**

Experiment 7

AIM :- The main objective of the experiment is implementation of a Binary Search Tree (BST) with deletion operations.The program allows users to insert several values into the BST , prints the inorder traversal and then deletes a node with a specific key .

THEORY:-

BST (BINARY SEARCH TREE):- A Binary Search Tree (BST) is a binary tree data structure that follows the principles of binary search. It is designed to maintain an efficient method for searching, inserting, and deleting values in a sorted collection.

STRUCTURE OF BST :-

- Each node in the tree has at most two child nodes: a left child and a right child.
- The left subtree of a node contains only nodes with values less than the node's value.
- The right subtree of a node contains only nodes with values greater than the node's value.
- The left and right subtrees are also binary search trees.

Because of these properties, searching for a specific value in a BST can be done efficiently through a process similar to binary search. This makes BSTs useful for operations that involve maintaining a sorted set of elements.

Aryan Nanda **221070003**

BASIC OPERATIONS IN A BST :-

Insertion:

- To insert a new value, start at the root and compare the value to be inserted with the current node's value.
- If the value is less, go to the left subtree; if greater, go to the right subtree.
- Repeat this process until an empty spot is found, and insert the new node.

Deletion:

- To delete a node with a specific value, locate the node in the tree.
- Handle different cases: node with no children, node with one child, and node with two children.
- Replace the node with its successor (or predecessor) if it has two children.
- Adjust the tree structure accordingly.

Search:

- To search for a value, start at the root and compare the target value with the current node's value.
- If equal, the value is found. If less, search in the left subtree; if greater, search in the right subtree.
- Repeat the process until the value is found or a null (empty) subtree is reached.

Traversal:

- Inorder, preorder, and postorder traversals can be performed to visit all nodes in a specific order.
- Inorder traversal of a BST visits nodes in ascending order.

Binary Search Trees provide an efficient way to organize and search for data. However, the efficiency relies on the balanced nature of the tree, and unbalanced trees can lead to performance issues. Balanced variants, such as AVL trees or Red-Black trees, aim to maintain balance automatically during insertions and deletions.

INORDER SUCCESSOR:- Is the node with the smallest key greater than the key of the given node. It is the next node that would be visited in an inorder traversal of the BST.

ALGORITHM:-

1. INSERT BST OPERATION :-

1. Create a new node with the given value.

2. If the root is NULL, set the root to the new node and return.

3. Initialize current node (curr) as the root and a variable to keep track of the previous node (prev) as NULL.

4. Traverse the tree:

a. If the new node's value is less than the current node's value, move to the left subtree.

b. If the new node's value is greater than or equal to the current node's value, move to the right subtree.

c. Update prev to the current node.

5. Insert the new node as the left or right child of the previous node based on the comparison in step 4.

2. SEARCH OPERATION :-

1. Initialize a temporary node (temp) as the root and a variable to keep track of the parent node as NULL.

2. While temp is not NULL and its value is not equal to the target key:

a. Update the parent to temp.

b. If the target key is less than temp's value, move to the left subtree.

c. If the target key is greater than temp's value, move to the right subtree.

3. If temp is NULL, the key is not found.

4. Return temp.

3. INORDER TRAVERSAL OPERATION :-

1. If the current node is not NULL:

a. Recursively perform inorder traversal on the left subtree.

b. Print the value of the current node.

c. Recursively perform inorder traversal on the right subtree.

4. INORDER-SUCCESSOR OPERATION :-

1. If right of given node is not NULL:

   a. Initialize temporary pointer (temp) with right of given node.
   b. While the right of temp is not NULL traverse in the right subtree.

   Return temp

2. Else:

   a. Initialise pointer p with root and q with nullptr.
   b. Traverse the tree in the following manner until you reach the given node:

      i) If the value of the current node p is greater than the value of the given node, traverse the left subtree and update q with p. Else traverse the right subtree.

      Return q

5. DELETION OPERATION :-

1. Search for the node with the specified key and keep track of its parent.

2. If the node is not found, print "Key not found" and return.

3. Handle three cases based on the number of children of the node:

   a. Node with no children: Remove the node from its parent.

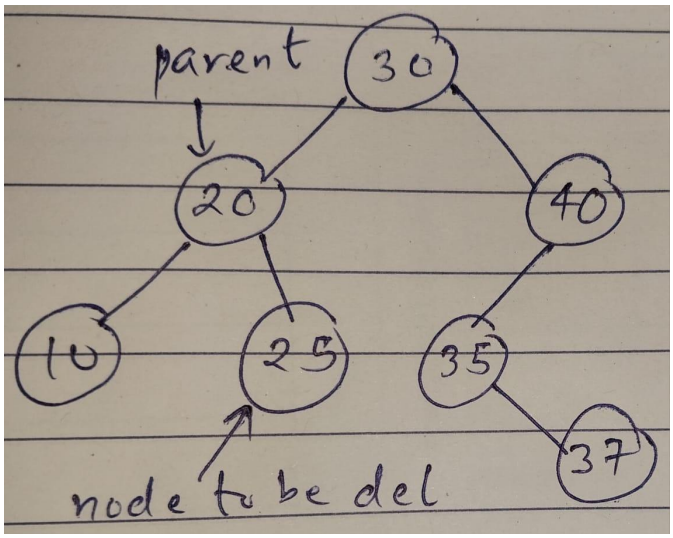
b. Node with one child: Replace the node with its child.

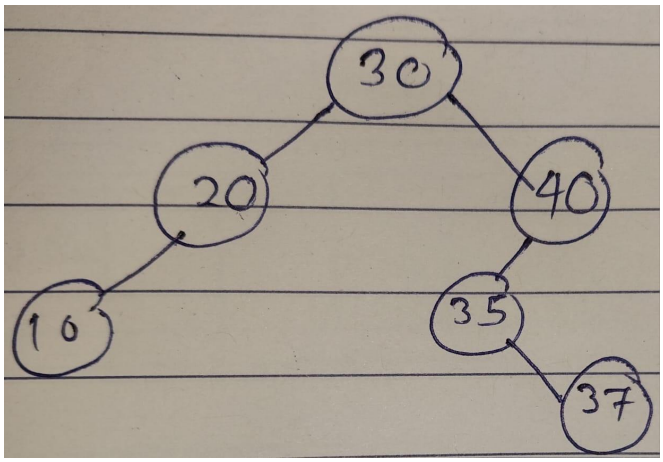
c. Node with two children: Find the inorder successor, replace the node with it, and delete the successor's original position.

EXAMPLE :-

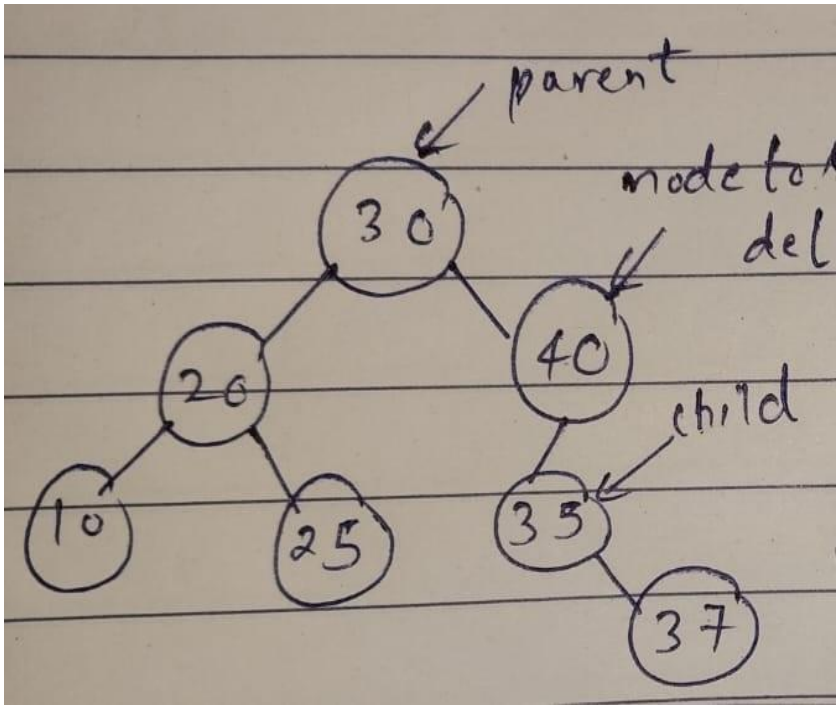
Consider this graph.

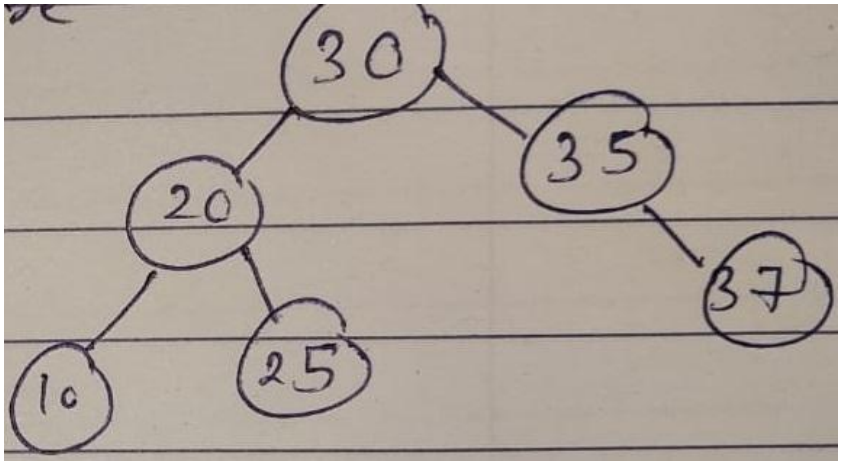
Case 1 : Node to be deleted is 25 (No children). Since it is a leaf node with no children, delete it directly. Set the right pointer of 20 (parent) as NULL.



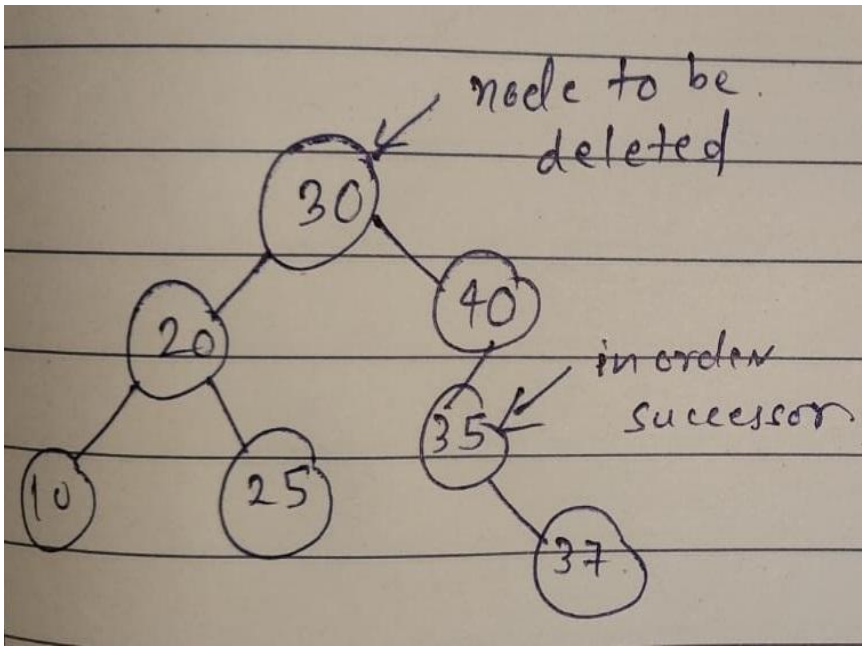


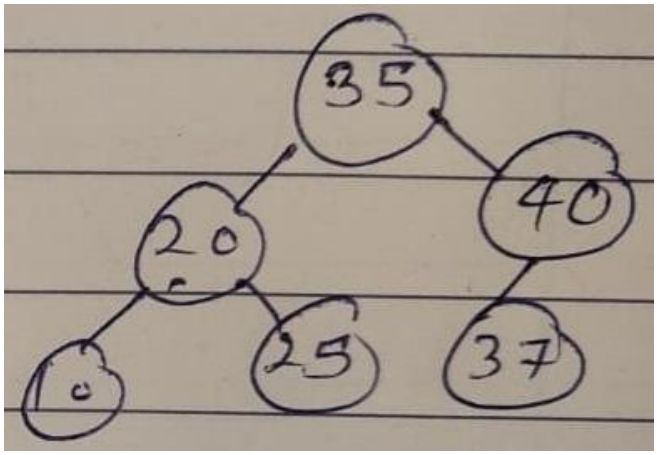
Case 2 : Node to be deleted is 40 (one child). Parent = 30 and left child = 35. Since 40 is the right child of 30, set the right pointer of 30 (parent) with 35 (child). Now delete 40.





Case 3 : Node to be deleted : 30 (two children). Find it's inorder successor = 35. Delete the inorder successor as per above two cases (inorder successor will always have at most one child) and replace this node's value with the value of inorder successor.





Code

```
#include <iostream>
#include <queue>
using namespace std;

class BinaryTreeNode {
  public:
   int data;
   BinaryTreeNode *left;
   BinaryTreeNode *right;
   BinaryTreeNode(int data) {
       this->data = data;
       left = NULL;
       right = NULL;
   }
   ~BinaryTreeNode() {
       if (left) delete left;
       if (right) delete right;
   }
};

BinaryTreeNode *takeInput() {
   int rootData;
   cin >> rootData;
   if (rootData == -1) {
       return NULL;
   }
   BinaryTreeNode *root = new BinaryTreeNode(rootData);
   queue<BinaryTreeNode*> q;
   q.push(root);
   while (!q.empty()) {
       BinaryTreeNode *currentNode = q.front();
       q.pop();
       int leftChild, rightChild;
       cin >> leftChild;
       if (leftChild != -1) {
           BinaryTreeNode *leftNode = new BinaryTreeNode(leftChild);
           currentNode->left = leftNode;
           q.push(leftNode);
       }
       cin >> rightChild;
       if (rightChild != -1) {
```

```
            BinaryTreeNode *rightNode = new BinaryTreeNode(rightChild);
            currentNode->right = rightNode;
            q.push(rightNode);
        }
    }
    return root;
}

void InOrderPrintBST(BinaryTreeNode *root){
    if(root == NULL)return;
    InOrderPrintBST(root -> left);
    cout<<root -> data<<" ";
    InOrderPrintBST(root -> right);
}

class checkBST{
    public:
    int max;
    int min;
    bool isbst;
};

checkBST helperfun(BinaryTreeNode *root){
    if(root == NULL){
        checkBST output;
        output.max=INT_MIN;
        output.min=INT_MAX;
        output.isbst=true;
        return output;
    }
    checkBST leftans=helperfun(root ->left);
    checkBST rightans=helperfun(root -> right);
    int maxdata=max(root -> data,max(leftans.max,rightans.max));
    int mindata=min(root -> data, min(leftans.min,rightans.min));
    bool finalans=root -> data>leftans.max && root -> data<=rightans.min &&
leftans.isbst && rightans.isbst;
    checkBST output;
    output.max=maxdata;
    output.min=mindata;
    output.isbst=finalans;
    return output;
}
```

```
bool isBST(BinaryTreeNode *root) {
   return helperfun(root).isbst;
}

BinaryTreeNode* minValueNode(BinaryTreeNode* node) {
   BinaryTreeNode* current = node;
   while (current && current->left != NULL) {
       current = current->left;
   }
   return current;
}

BinaryTreeNode* inorderSuccessor(BinaryTreeNode* root, int value) {
   BinaryTreeNode* current = root;
   BinaryTreeNode* successor = nullptr;
   while (current != nullptr && current->data != value) {
       if (value < current->data) {
           successor = current;
           current = current->left;
       }
       else {
           current = current->right;
       }
   }
   if (current == nullptr) {
       return nullptr;
   }
   if (current->right != nullptr) {
       return minValueNode(current->right);
   }
   else {
       return successor;
   }
}

BinaryTreeNode* deleteData(int data, BinaryTreeNode* node) {
       if (node == NULL) {
           return NULL;
       }
       if (data > node->data) {
           node->right = deleteData(data, node->right);
```

```
            return node;
        } else if (data < node->data) {
            node->left = deleteData(data, node->left);
            return node;
        } else {
            if (node->left == NULL && node->right == NULL) {
                delete node;
                return NULL;
            } else if (node->left == NULL) {
                BinaryTreeNode* temp = node->right;
                node->right = NULL;
                delete node;
                return temp;
            } else if (node->right == NULL) {
                BinaryTreeNode* temp = node->left;
                node->left = NULL;
                delete node;
                return temp;
            } else {
                BinaryTreeNode* minNode = node->right;
                while (minNode->left != NULL) {
                    minNode = minNode->left;
                }
                int rightMin = minNode->data;
                node->data = rightMin;
                node->right = deleteData(rightMin, node->right);
                return node;
            }
        }
}

bool searchInBST(BinaryTreeNode *root , int k) {
   if(root == NULL){
       return false;
   }
   bool ans = false;
   if(k==root -> data){
       return true;
   }
   else if(k<root -> data){
       ans = searchInBST(root ->left,k);
   }
```

```
    else if(k> root -> data){
        ans = searchInBST(root -> right,k);
    }
    return ans;
}

int main() {
    BinaryTreeNode *root = takeInput();
    InOrderPrintBST(root);
    cout<<"Check if the binary tree is binary Search Tree Or Not"<<endl;
    cout << (isBST(root) ? "true" : "false");
    if(isBST(root)==0){
        return 0;
    }
    int k;
    cout<<endl;
    cout<<"Enter the number you want to search for "<<endl;
    cin >> k;
    cout << ((searchInBST(root, k)) ? "true" : "false")<<endl;
    int a;
    cout<<"Enter the number whose Inorder Sucessor you want "<<endl;
    cin>>a;
    cout<<inorderSuccessor(root,a)->data<<endl;
    int n;
    cout<<"Enter the number you want to delete "<<endl;
    cin>>n;
    root = deleteData(n,root);
    InOrderPrintBST(root);
    cout<<"Check if the binary tree is binary Search Tree Or Not"<<endl;
    cout << (isBST(root) ? "true" : "false");
    delete root;
}

// ------------------------------ Sample Test Case--------------------------------
/*
BST

22 15 57
4 18
35 83
-1 -1
-1 19
```

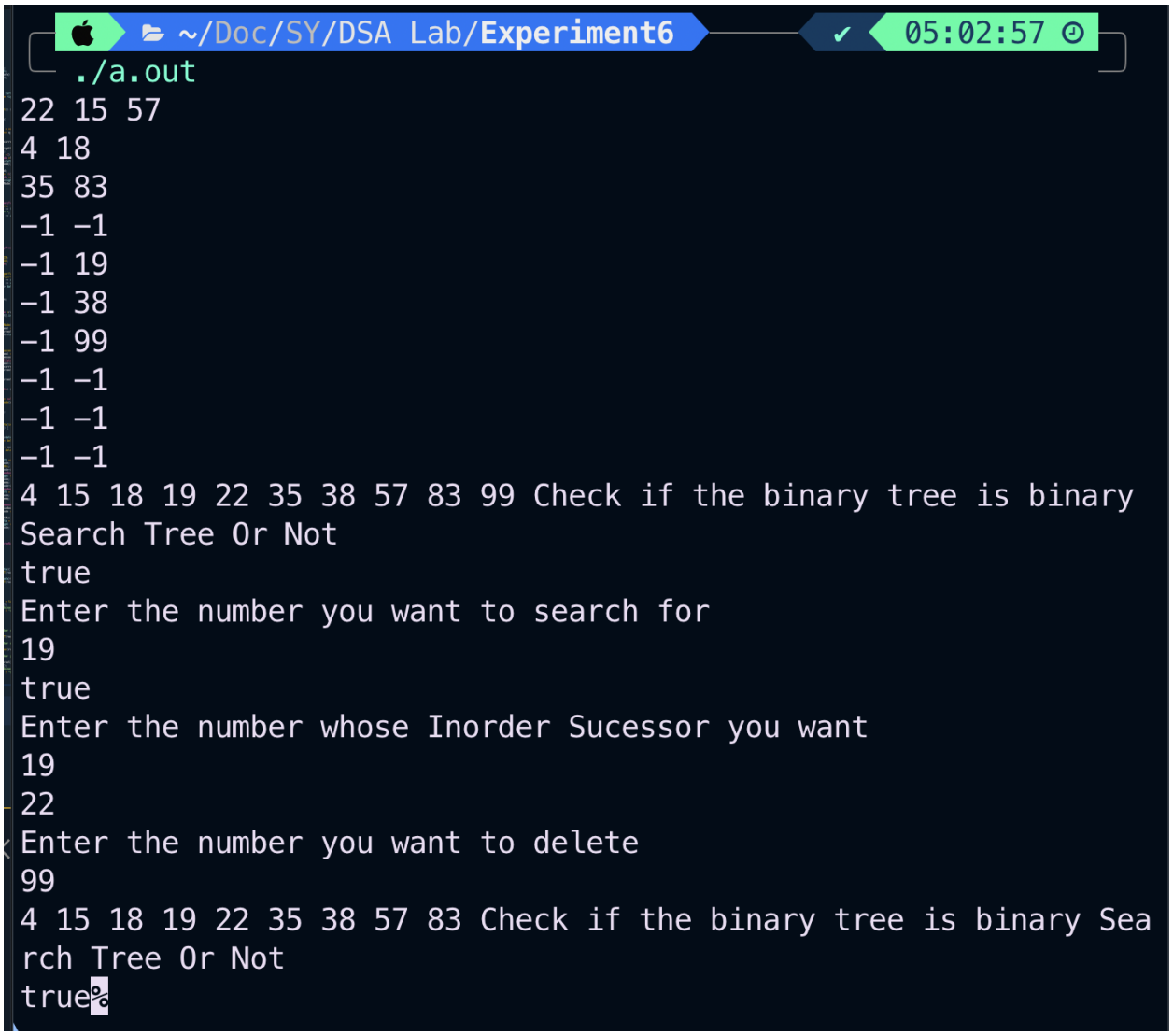
```
-1 38
-1 99
-1 -1
-1 -1
-1 -1

Not BST

5 2 10
0 1
-1 15
-1 -1
-1 -1
-1 -1

*/
```

OUTPUT :-



CONCLUSION :-

Thus in this experiment we learnt how to delete a node in BST by using the concept of inorder successor and traversals.