

## EXPERIMENT NO. – 6

### AIM :

- Perform Basic operations on Binary Search Tree.
- Find inorder successor for a given node in BST.

### THEORY :

#### *Binary Search Tree :*

It is non-linear Data structure .It 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.
- The left and right subtree each must also be a binary search tree.

#### *Operations on BST:*

- Insertion
- Searching
- Deletion
- Traversals : (Preorder, Inorder, Postorder)

#### *Inorder Traversal:*

At first traverse the left subtree then visit the root and then traverse the right subtree. The Inorder Traversal of the BST gives the values of the nodes in sorted order.

Steps to implement the idea:

1. Traverse left subtree.
2. Visit the root and print the data.
3. Traverse the right subtree.

#### *Inorder Successor :*

The inorder successor of a node is the next node in Inorder traversal of the Binary Tree. Inorder Successor is NULL for the last node in Inorder traversal.

- If a node has a right child, its successor is the node with the least value present in its right sub-tree.
- If the given node does not have a right child, its in-order successor is located above it in the tree. The successor of the given node is the parent of the node's youngest ancestor which is a left child.

**ALGORITHM:****For Insertion :**

1. Create a new node and set its info to x (value to be inserted ).
2. Set new Node's left and right pointers to null.
3. If the BST is empty, set root to new Node and exit.
4. Otherwise, initialize two pointers: current (t1) as root and parent(t2) as null.
5. While t1 is not null, do the following:
  - Set parent to current.(t2=t1)
  - If x is less than the current's info, move to the left child (t1 = t1->left).
  - Otherwise, move to the right child (t1 = t1->right).
6. After exiting the loop, check whether x is less than parent's(t2) info.
  - a. If true, set the parent's left child to new Node. (t2->left=p)
  - b. Otherwise, set parent's right child to new Node.(t2->right=p)

**For Searching:**

1. Set current (t1)as the root of the BST and parent(t2) as null.
2. While current (t1) is not null, do
  - If the t1->info > value ,
    - Set parent to current.(t2=t1)
    - Move to the left child (t1 = t1->left).
  - Else,
    - Set parent to current.(t2=t1)
    - Move to the right child (t1 = t1->right).
  - Else, the value is found, and return the value.
3. If the loop exits, the value is not found in the BST.
4. Return -1 , if value is not found .

**For Inorder Traversal :**

1. Check BST is empty or not.
2. If BST is not Empty
  - Traverse the left subtree
  - Print the info of root
  - Then traverse the right subtree.

**For Inorder successor :**

1. Set a variable successor to NULL.
2. Initialize a temporary pointer to the root of the tree .
3. While temp is not NULL and temp->info is not equal to key:

- a. if(temp->info > key)
  - 1. update the successor to temp.
  - 2. Move temp to the left child.
- b. if(temp->info < key)
  - 1. move temp to the right child.
  - 4. If temp is NULL, return NULL.
  - 5. If temp has a right child:
    - a. Set temp1 to the right child of temp.
  - a.           b. Find the leftmost node in the right subtree of temp1.
  - b. Return the leftmost node as the inorder successor.
  - 1. If temp does not have a right child:
    - a. Return the last updated successor.

**EXAMPLE:**

Insert 45:

Create a new node new\_node with info 45.

Set root to new\_node.

Insert 15:

Start with p pointing to root (45).

Move towards left of 45.

Set q to 45 and p to 15 .

Search(20)

Start with (p) pointing to root (45).

Move left to 45.

Move left to 15.

Move right to 20.

Value 20 found.

Inorder Successor(50)

Search for 50.

The node 50 has no right child .

And nearest ancestor of 50 is 55 . So The inorder successor of 50 is 55.

**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

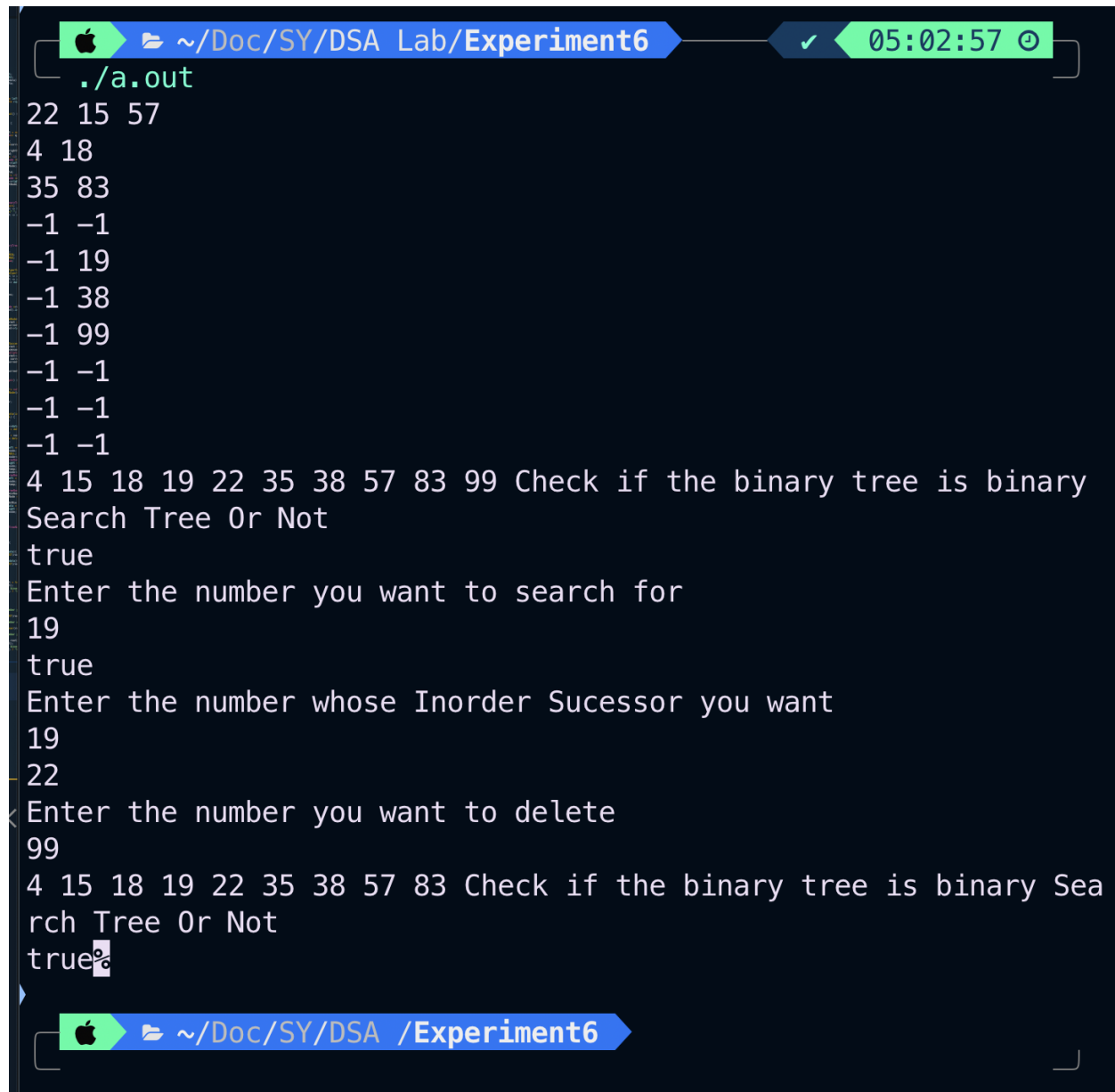
-1 19

-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:**

```
~/Doc/SY/DSA Lab/Experiment6 05:02:57
./a.out
22 15 57
4 18
35 83
-1 -1
-1 19
-1 38
-1 99
-1 -1
-1 -1
-1 -1
4 15 18 19 22 35 38 57 83 99 Check if the binary tree is binary
Search Tree Or Not
true
Enter the number you want to search for
19
true
Enter the number whose Inorder Sucessor you want
19
22
Enter the number you want to delete
99
4 15 18 19 22 35 38 57 83 Check if the binary tree is binary Sea
rch Tree Or Not
true%
```

**CONCLUSION:**

Hence we have successfully implemented operations on Binary Search Tree and found Inorder Successor of the given node.