Assignment – 5

23. Implement an binary search tree and write functions for insertion & deletion.

Source Code: main()

```
#include <iostream>
using namespace std;
typedef struct Node
{
    Node *left, *right;
    int data;
    Node(int data)
    {
        this->left = NULL;
        this->data = data;
        this->right = NULL;
    }
} Node;
Node *root = NULL;
void createBST();
Node *createNode(int);
void freeTree(Node *root);
void printTree(Node *, int);
void insertBST(Node *&, int);
Node *deleteNodeBST(Node *, int);
Node *inOrderPredecessor(Node *ptr);
int main(){
    createBST(); // Initialize the BST with some predefined values.
    while (1)    {
        cout << "\nMenu:\n";
        cout << "1. Insert Node in the tree\n";
        cout << "2. Delete Node from tree\n";
        cout << "0. Exit\n";
        printTree(root, 0);
        cout << "\nEnter your choice: ";
        int choice, value;
        cin >> choice;
        switch (choice)       {
        case 1:
            cout << "Enter value to insert: ";
            cin >> value;
            insertBST(root, value);
            break;
        case 2:
            cout << "Enter value to delete: ";
            cin >> value;
            root = deleteNodeBST(root, value);
            break;
        case 0:
            freeTree(root);
            cout << "\nExiting...\n\n";
            exit(0);
        default:
            cout << "Invalid choice. Try again." << endl;
        }
    }
```

Source Code: insertBST()

```
void insertBST(Node *&ptr, int value)
{
    if (ptr == NULL)
    {
        ptr = createNode(value);
        return;
    }

    if (value < ptr->data)
        insertBST(ptr->left, value);

    else if (value > ptr->data)
        insertBST(ptr->right, value);

    else
        cout << "\nDuplication is not allowed! " << value << " already exists.\n";
}
```

Source Code: createBST()

```
void createBST()
{
    root = NULL; // Reset root for initialization

    insertBST(root, 8);
    insertBST(root, 3);
    insertBST(root, 10);
    insertBST(root, 1);
    insertBST(root, 6);
    insertBST(root, 14);
    insertBST(root, 4);
    insertBST(root, 7);
    insertBST(root, 13);
}
```

Source Code: freeTree()

```
void freeTree(Node *root)
{
    if (root == nullptr)
        return;

    freeTree(root->left);
    freeTree(root->right);

    delete root;
}
```

Source Code: printTree()

```
void printTree(Node *root, int space = 0)
{
    if (root == nullptr)
        return;

    space += 5;

    printTree(root->right, space);

    cout << '\n';
    for (int i = 5; i < space; i++)
    {
        cout << ' ';
    }
   out << root->data;

    printTree(root->left, space);
}
```

Source Code: inOrderPredecessor ()

```
Node *inOrderPredecessor(Node *ptr)
{
    ptr = ptr->left;
    while (ptr->right != NULL)
        ptr = ptr->right;
    return ptr;
}
```

Source Code: createNode()

```
Node *createNode(int data)
{
    return new Node(data);
}
```

Source Code: deleteNodeBST()

```
Node *deleteNodeBST(Node *ptr, int value)
{
    if (ptr == NULL)
    {
        cout << "Value " << value << " not found in the tree.\n";
        return NULL;
    }

    if (value < ptr->data)
    {
        ptr->left = deleteNodeBST(ptr->left, value);
    }
    else if (value > ptr->data)
    {
        ptr->right = deleteNodeBST(ptr->right, value);
    }
    else
    {
        // Node with one child or no child
        if (ptr->left == NULL)
        {
            Node *temp = ptr->right;
            delete ptr;
            return temp;
        }
        else if (ptr->right == NULL)
        {
            Node *temp = ptr->left;
            delete ptr;
            return temp;
        }

        // Node with two children
        Node *inPre = inOrderPredecessor(ptr);
        ptr->data = inPre->data;
        ptr->left = deleteNodeBST(ptr->left, inPre->data);
    }
    return ptr;
}
```

Output

```
Menu:
1. Insert Node in the tree
2. Delete Node from tree
0. Exit
        14
            13
    10
8
            7
        6
            4
    3
        1
Enter your choice: 1
Enter value to insert: 5

Menu:
1. Insert Node in the tree
2. Delete Node from tree
0. Exit
        14
            13
    10
8
            7
        6
                5
            4
    3
        1
Enter your choice: 2
Enter value to delete: 6

Menu:
1. Insert Node in the tree
2. Delete Node from tree
0. Exit
        14
            13
    10
8
            7
        5
            4
    3
        1
Enter your choice: 0
Exiting...
```

24. Write a program to find the height of a binary tree.

Source Code: main()

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

typedef struct Node
{
    Node *left, *right;
    int data;

    Node(int data)
    {
        this->left = NULL;
        this->data = data;
        this->right = NULL;
    }
} Node;

Node *root = NULL;

void createBinaryTree();
Node *createNode(int);
void freeTree(Node *root);
void printTree(Node *, int);
int heightCalculate(Node *);

int main()
{
    createBinaryTree(); // Initialize the BinaryTree with some predefined
values.

    printTree(root, 0);

    cout << "\n\n"
         << "The height of the current tree is: " << heightCalculate(root)
         << "\n\n"
         << endl;

    return 0;
}
```

Source Code: insertBST()

```
void insertBST(Node *&ptr, int value)
{
    if (ptr == NULL)
    {
        ptr = createNode(value);
        return;
    }

    if (value < ptr->data)
        insertBST(ptr->left, value);

    else if (value > ptr->data)
        insertBST(ptr->right, value);

    else
        cout << "\nDuplication is not allowed! " << value << " already exists.\n";
}
```

Source Code: createBST()

```
void createBST()
{
    root = NULL; // Reset root for initialization

    insertBST(root, 8);
    insertBST(root, 3);
    insertBST(root, 10);
    insertBST(root, 1);
    insertBST(root, 6);
    insertBST(root, 14);
    insertBST(root, 4);
    insertBST(root, 7);
    insertBST(root, 13);
}
```

Source Code: freeTree()

```
void freeTree(Node *root)
{
    if (root == nullptr)
        return;

    freeTree(root->left);
    freeTree(root->right);

    delete root;
}
```

Source Code: printTree()

```
void printTree(Node *root, int space = 0)
{
    if (root == nullptr)
        return;

    space += 5;

    printTree(root->right, space);

    cout << '\n';
    for (int i = 5; i < space; i++)
    {
        cout << ' ';
    }
    out << root->data;

    printTree(root->left, space);
}
```

Source Code: heightCalculate()

```
int heightCalculate(Node *root)
{
    if (root == NULL)
        return 0;
    int left_height = heightCalculate(root->left);
    int right_height = heightCalculate(root->right);

    return max(left_height, right_height) + 1;
}
```

Source Code: createNode()

```
Node *createNode(int data)
{
    return new Node(data);
}
```

Output

```
        14
            13
    10
8
            7
        6
            4
    3
        1

The height of the current tree is: 4
```

25.Implement an algorithm to check if a binary tree is a binary search tree (BST).

Source Code: main()

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

typedef struct Node
{
    Node *left, *right;
    int data;

    Node(int data)
    {
        this->left = NULL;
        this->data = data;
        this->right = NULL;
    }
} Node;

Node *root = NULL;

void createBinaryTree();
Node *createNode(int);
void freeTree(Node *root);
void printTree(Node *, int);
int heightCalculate(Node *);

int main()
{
    createBinaryTree(); // Initialize the BinaryTree with some predefined
values.

    printTree(root, 0);

    cout << "\n\n"
         << "The height of the current tree is: " << heightCalculate(root)
         << "\n\n"
         << endl;

    return 0;
}
```

Source Code: createTree()

```
void createTree()
{
    root = NULL; // Reset root for initialization

    root = createNode(1);
    Node *node1 = createNode(2);
    Node *node2 = createNode(3);
    Node *node3 = createNode(4);
    Node *node4 = createNode(5);
    Node *node5 = createNode(6);
    Node *node6 = createNode(7);

    root->left = node1;
    root->right = node2;
    node1->left = node3;
    node1->right = node4;
    node2->left = node5;
    node2->right = node6;
}
```

Source Code: printTree()

```
void printTree(Node *root, int space = 0)
{
    if (root == nullptr)
        return;

    space += 5;

    printTree(root->right, space);

    cout << '\n';
    for (int i = 5; i < space; i++)
    {
        cout << ' ';
    }
    cout << root->data;

    printTree(root->left, space);
}
```

Source Code: createNode()

```
Node *createNode(int data)
{
    return new Node(data);
}
```

Source Code: freeTree()

```
void freeTree(Node *root)
{
    if (root == nullptr)
        return;

    freeTree(root->left);
    freeTree(root->right);

    delete root;
}
```

Source Code: isBST()

```
bool isBST(Node *root)
{
    if (root == NULL)
        return true;

    if (sizeof(root) > 3)
        return false;

    if (root->left != NULL && root->left->data > root->data)
        return false;

    if (root->right != NULL && root->right->data < root->data)
        return false;

    if (!isBST(root->left) || !isBST(root->right))
        return false;

    return true;
}
```

Output

```
        7
    3
        6
1
        5
    2
        4

The tree is not a Binary Search Tree.
```

26. Create a function to find the lowest common ancestor (LCA) Of two nodes in a binary tree.

Source Code: main()

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

// Definition for a binary tree node.
typedef struct Node
{
    int data;
    Node *left;
    Node *right;
    Node(int data, Node *left = NULL, Node *right = NULL)
    {
        this->data = data;
        this->left = left;
        this->right = right;
    }
} Node;

void printTree(Node *, int);
Node *createNode(int val);
Node *createTree();
Node *lowestCommonAncestor(Node *root, Node *p, Node *q);
Node *root = createTree();

int main()
{
    printTree(root, 0);
    Node *p = root->left->left;
    Node *q = root->left->right->right;
    Node *lca = lowestCommonAncestor(root, p, q);
    cout << "\n\nThe lowest common ancestor of " << p->data << " and
"
         << q->data << " is " << lca->data
         << endl;
    return 0;
}
```

Source Code: createTree()

```
Node *createTree()
{
    Node *root = createNode(5);
    root->left = createNode(3);
    root->right = createNode(1);
    root->left->left = createNode(6);
    root->left->right = createNode(2);
    root->right->left = createNode(0);
    root->right->right = createNode(8);
    root->left->right->left = createNode(7);
    root->left->right->right = createNode(4);
    return root;
}
```

Source Code: printTree()

```
void printTree(Node *root, int space = 0)
{
    if (root == nullptr)
        return;

    space += 5;

    printTree(root->right, space);

    cout << '\n';
    for (int i = 5; i < space; i++)
    {
        cout << ' ';
    }
    cout << root->data;

    printTree(root->left, space);
}
```

Source Code: createNode()

```
Node *createNode(int data)
{
    return new Node(data);
}
```

Source Code: freeTree()

```
void freeTree(Node *root)
{
    if (root == nullptr)
        return;

    freeTree(root->left);
    freeTree(root->right);

    delete root;
}
```

Source Code: lowestCommonAncestor()

```
Node *lowestCommonAncestor(Node *root, Node *p, Node *q)
{
    if (root == NULL || root == p || root == q)
        return root;

    Node *left = lowestCommonAncestor(root->left, p, q);
    Node *right = lowestCommonAncestor(root->right, p, q);

    if (left == NULL)
        return right;
    else if (right == NULL)
        return left;
    else
        return root;
}
```

Output

```
        8
    1
        0
5
            4
        2
            7
    3
        6

The lowest common ancestor of 6 and 4 is 3
```