Name: Gavali Deshabhakt Nagnath
Roll No. 202CD005

Data Structures and Algorithms

Assignment-2

Index

Q1. Design a tree data structure with operations such as: inserting an element, deleting an element, finding the depth of the tree, finding the number of nodes at each level, finding the total number of nodes, traversing the tree (in order, pre-order, post-order).

Program:

```
// implementation of 2-ary tree

#include <iostream>

using namespace std;
template <class T>
class treeNode
{
public:
   T data;
   treeNode *left;
   treeNode *right;
   treeNode<T> *next;
   treeNode()
   {
      data = 0;
      left = right = NULL;
   }
   treeNode(T d)
   {
      data = d;
      left = NULL;
      right = NULL;
   }
};
template <class T>
class queue
{
public:
   treeNode<T> *front;
   treeNode<T> *rear;

   queue()
   {
      front = NULL;
      rear = NULL;
   }
   bool isEmpty()
   {
      treeNode<T> *tempf = front;
      treeNode<T> *tempr = rear;
```

```
    if (tempf == NULL)
    {
        return true;
    }
    else
    {
        return false;
    }
}
void enqueue(treeNode<T> *n)
{
    if (front == NULL)
    {
        front = n;
        rear = n;
    }
    else
    {
        rear->next = n;
        rear = n;
    }
}
treeNode<T> *dequeue()
{
    treeNode<T> *temp = NULL;
    if (isEmpty())
    {
        return NULL;
    }
    else
    {
        if (front == rear)
        {
            temp = front;
            front = NULL;
            rear = NULL;
            return temp;
        }
        else
        {
            temp = front;
            front = front->next;
```

```
            return temp;
          }
        }
    }
};
template <class T>
class Tree
{
public:
    treeNode<T> *root;
    T ele;
    Tree() { root = NULL; }
    void CreateTree();
    bool isEmpty(treeNode<T> *root)
    {
        if (root == NULL)
            return true;
        else
        {
            return false;
        }
    }
    void PreOrder(treeNode<T> *p);
    void PostOrder(treeNode<T> *p);
    void InOrder(treeNode<T> *p);
    void LevelOrder(treeNode<T> *p);
    int DepthOfTree(treeNode<T> *root);
    int count(treeNode<T> *root);
    void Search(treeNode<T> *root);
    treeNode<T> *checkifelementpresentintree(T n);
    int NodesAtEachLevel(treeNode<T> *root, int curr, int desired);
    void insert(T pdata);
    treeNode<T>* deleteNode(treeNode<T> *n, treeNode<T> *delNode);
};
template <class T>
void Tree<T>::CreateTree()
{
    treeNode<T> *pop, *next;
    T x;
    queue<T> q;
    cout << "\n* NOTE: Please enter -1 if there's no node. *" << endl
         << endl;
    cout << "Enter Root element : ";
    cin >> x;
```

```
    cout << endl;
    root = new treeNode<T>();
    root->data = x;
    root->left = root->right = NULL;
    q.enqueue(root);
    while (!q.isEmpty())
    {
        pop = q.dequeue();
        cout << "Enter left child of " << pop->data << ": ";
        cin >> x;
        cout << endl;
        if (x != -1)
        {
            next = new treeNode<T>();
            next->data = x;
            next->left = next->right = NULL;
            pop->left = next;
            q.enqueue(next);
            next = NULL;
        }
        cout << "Enter right child of " << pop->data << ": ";
        cin >> x;
        cout << endl;
        if (x != -1)
        {
            next = new treeNode<T>();
            next->data = x;
            next->left = next->right = NULL;
            pop->right = next;
            q.enqueue(next);
            next = NULL;
        }
    }
}
template <class T>
void Tree<T>::PreOrder(treeNode<T> *p)
{
    if (p != NULL)
    {
        cout << p->data << " ";
        PreOrder(p->left);
        PreOrder(p->right);
    }
```

```
}
template <class T>
void Tree<T>::InOrder(treeNode<T> *p)
{
   if (p != NULL)
   {
      InOrder(p->left);
      cout << p->data << " ";
      InOrder(p->right);
   }
}
template <class T>
void Tree<T>::PostOrder(treeNode<T> *p)
{
   if (p != NULL)
   {
      PostOrder(p->left);
      PostOrder(p->right);
      cout << p->data << " ";
   }
}
template <class T>
void Tree<T>::LevelOrder(treeNode<T> *root)
{
   queue<T> q;
   q.enqueue(root);
   cout << root->data << " ";
   while (!q.isEmpty())
   {
      root = q.dequeue();
      if (root->left)
      {
         cout << root->left->data << " ";
         q.enqueue(root->left);
      }
      if (root->right)
      {
         cout << root->right->data << " ";
         q.enqueue(root->right);
      }
   }
}
template <class T>
int Tree<T>::DepthOfTree(treeNode<T> *root)
{
```

```
    int x = 0, y = 0;
    if (root != NULL)
    {
        x = DepthOfTree(root->left);
        y = DepthOfTree(root->right);
        if (x > y)
            return (x + 1);
        else
            return (y + 1);
    }
    else
    {
        return 0;
    }
}
template <class T>
int Tree<T>::count(treeNode<T> *root)
{
    treeNode<T> *temp = root;
    int x, y;
    if (temp != NULL)
    {
        x = count(temp->left);
        y = count(temp->right);
        return x + y + 1;
    }
    return 0;
}
template <class T>
int Tree<T>::NodesAtEachLevel(treeNode<T> *node, int curr, int desired)
{
    if (node == NULL)
        return 0;
    if (curr == desired)
        return 1;
    return NodesAtEachLevel(node->left, curr + 1, desired) +
           NodesAtEachLevel(node->right, curr + 1, desired);
}
template <class T>
treeNode<T> *Tree<T>::checkifelementpresentintree(T n)
{
    queue<T> q;
    q.enqueue(root);
    treeNode<T> *temp = root;
```

```
    while (temp->data != n && !q.isEmpty())
    {
        temp = q.dequeue();
        if (temp->left)
        {
            q.enqueue(temp->left);
        }
        if (temp->right)
        {
            q.enqueue(temp->right);
        }
    }
    if (temp->data == n)
    {
        return temp;
    }
    else
    {
        return NULL;
    }
}
template <class T>
void Tree<T>::insert(T pdata)
{
    treeNode<T> *temp = checkifelementpresentintree(pdata);
    if (temp != NULL)
    {
        int in;
        T val;
        do
        {
            cout << "\n1.Attach to left of " << temp->data << endl;
            cout << "2.Attach to right of " << temp->data << endl;
            cout << "3.Go To previous Menu\n>" << endl;
            cin >> in;
            switch (in)
            {
            case 1:
            {
                if (temp->left == NULL)
                {
                    cout << "Enter element to be inserted : ";
                    cin >> val;
```

```
            if (checkifelementpresentintree(val) != NULL)
            {
                cout << "Element already present in Tree." << endl;
            }
            else
            {
                treeNode<T> *newNode = new treeNode<T>();
                newNode->data = val;
                temp->left = newNode;
                cout << "Element Added to tree successfully." << endl;
                in = 3;
            }
        }
        else
        {
            cout << "Node Already Occupied" << endl;
        }
        break;
    }
    case 2:
    {
        if (temp->right == NULL)
        {
            cout << "Enter element to be inserted : ";
            cin >> val;
            if (checkifelementpresentintree(val) != NULL)
            {
                cout << "Element already present in Tree." << endl;
            }
            else
            {
                treeNode<T> *newNode = new treeNode<T>();
                newNode->data = val;
                temp->right = newNode;
                cout << "Element Added to tree successfully." << endl;
                in = 3;
            }
        }
        else
        {
            cout << "Node Already Occupied" << endl;
        }
```

```
            break;
         }
         case 3:
         {
            break;
         }
         default:
         {
            cout << "Invalid Input\nTry Again" << endl;
         }
         }
      } while (in != 3);
   }
   else
   {
      cout << "Parent Unavailable" << endl;
   }
}
template <class T>
treeNode<T>* Tree<T>::deleteNode(treeNode<T> *r, treeNode<T> *delNode)
{
   treeNode<T> *temp = checkifelementpresentintree(delNode->data);
   if (r == NULL)
   {
      return NULL;
   }
   else
   {
      if (r->left == NULL)
      {
         treeNode<T> *temp = r->right;
         delete r;
         return temp;
      }
      else if (r->right == NULL)
      {
         treeNode<T> *temp = r->left;
         delete r;
         return temp;
      }
      else if(r->left!=NULL && r->right!=NULL)
      {
         treeNode<T> *temp = r->right;
         temp->left=r->left;
         delete r;
         return temp;
```

```
        }
        return r;
    }
}
int main()
{
    Tree<int> t;
    int input;
    do
    {
        cout << "\n************************************************" << endl;
        cout << "***************  Tree Operations  ***************" << endl;
        cout << "************************************************" << endl
             << endl;
        cout << "1.Create Tree\n2.Insert element in tree\n3.Delete element form tree\n4.Tree Traversal\
n5.Depth of tree\n6.Search Node \n7.Number of nodes at each level\n8.Total Number of Nodes\
n9.Clear Screen\n10.Exit" << endl;
        cout << ">";
        cin >> input;
        switch (input)
        {
        case 1:
        {
            cout << "Tree Creation" << endl;
            t.CreateTree();
            break;
        }
        case 2:
        {
            cout << "Insert Element in Tree" << endl;
            cout << "Enter Parent: ";
            cin >> t.ele;
            t.insert(t.ele);
            break;
        }
        case 3:
        {
            cout << "Delete Element from Tree" << endl;
            cout << "Enter Element To be deleted: ";
            treeNode<int> *delNode = new treeNode<int>();
            cin >> delNode->data;
            treeNode<int> *temp = new treeNode<int>();
            temp = t.checkifelementpresentintree(delNode->data);
```

```
    if (temp != NULL)
    {
        t.deleteNode(t.root, delNode);
        cout << "Deletion successfull" << endl;
    }
    else
    {
        cout << "Element NOT present in List." << endl;
    }
    break;
}
case 4:
{
    cout << "Tree Traversal" << endl;
    if (t.isEmpty(t.root))
    {
        cout << "Tree is Empty" << endl;
    }
    else
    {
        cout << "Pre-Order Traversal  :";
        t.PreOrder(t.root);
        cout << endl;
        cout << "In-Order Traversal   :";
        t.InOrder(t.root);
        cout << endl;
        cout << "Post-Order Traversal :";
        t.PostOrder(t.root);
        cout << endl;
        cout << "Level-Order Traversal:";
        t.LevelOrder(t.root);
        cout << endl;
    }
    break;
}
case 5:
{
    cout << "Depth of Tree is " << t.DepthOfTree(t.root) - 1 << "." << endl;
    break;
}
case 6:
{
```

```
    cout << "Search Node " << endl;
    cout << "Enter Element: ";
    cin >> t.ele;
    if (t.checkifelementpresentintree(t.ele) != NULL)
    {
        cout << "Element in Tree" << endl;
    }
    else
    {
        cout << "Element Not in Tree" << endl;
    }
    break;
}
case 7:
{
    cout << "Number of nodes at each level" << endl;
    for (int i = 0; i < t.DepthOfTree(t.root); i++)
    {
        int count = t.NodesAtEachLevel(t.root, 0, i);
        cout << "Level " << i + 1 << " has " << count << " nodes" << endl;
    }
    break;
}
case 8:
{
    cout << "Total Number of Nodes in Tree are " << t.count(t.root) << "." << endl;
    break;
}
case 9:
{
    system("clear");
    break;
}
case 10:
{
    cout << "\n\nProgram Ended\n"
         << endl;
    break;
}
default:
{
    cout << "Invalid input\nTry Again" << endl;
```

```
        }
        }
    } while (input != 10);
    return 0;
}
```

Output:

```
************************************************
*****************  Tree Operations  *****************
************************************************

1.Create Tree
2.Insert element in tree
3.Delete element form tree
4.Tree Traversal
5.Depth of tree
6.Search Node
7.Number of nodes at each level
8.Total Number of Nodes
9.Clear Screen
10.Exit
>1
Tree Creation

* NOTE: Please enter -1 if there's no node. *

Enter Root element : 1

Enter left child of 1: 2

Enter right child of 1: 3

Enter left child of 2: 4

Enter right child of 2: 5

Enter left child of 3: 6

Enter right child of 3: 7

Enter left child of 4: -1

Enter right child of 4: -1

Enter left child of 5: -1
```

```
Enter right child of 5: -1

Enter left child of 6: -1

Enter right child of 6: -1

Enter left child of 7: -1

Enter right child of 7: -1


************************************************
**********  Tree Operations  **********
************************************************

1.Create Tree
2.Insert element in tree
3.Delete element form tree
4.Tree Traversal
5.Depth of tree
6.Search Node
7.Number of nodes at each level
8.Total Number of Nodes
9.Clear Screen
10.Exit
>4
Tree Traversal
Pre-Order Traversal  :1 2 4 5 3 6 7
In-Order Traversal   :4 2 5 1 6 3 7
Post-Order Traversal :4 5 2 6 7 3 1
Level-Order Traversal:1 2 3 4 5 6 7

************************************************
**********  Tree Operations  **********
************************************************

1.Create Tree
2.Insert element in tree
3.Delete element form tree
4.Tree Traversal
5.Depth of tree
6.Search Node
7.Number of nodes at each level
8.Total Number of Nodes
9.Clear Screen
10.Exit
>2
Insert Element in Tree
```

```
Enter Parent: 2

1.Attach to left of 2
2.Attach to right of 2
3.Go To previous Menu
>1
Node Already Occupied

1.Attach to left of 2
2.Attach to right of 2
3.Go To previous Menu
>2
Node Already Occupied

1.Attach to left of 2
2.Attach to right of 2
3.Go To previous Menu
>3

************************************************
**********  Tree Operations  **********
************************************************

1.Create Tree
2.Insert element in tree
3.Delete element form tree
4.Tree Traversal
5.Depth of tree
6.Search Node
7.Number of nodes at each level
8.Total Number of Nodes
9.Clear Screen
10.Exit
>2
Insert Element in Tree
Enter Parent: 4

1.Attach to left of 4
2.Attach to right of 4
3.Go To previous Menu
>1
Enter element to be inserted : 5
Element already present in Tree.

1.Attach to left of 4
2.Attach to right of 4
3.Go To previous Menu
>1
Enter element to be inserted : 6
```

```
Element already present in Tree.

1.Attach to left of 4
2.Attach to right of 4
3.Go To previous Menu
>1
Enter element to be inserted : 9
Element Added to tree successfully.

************************************************
**********  Tree Operations  **********
************************************************

1.Create Tree
2.Insert element in tree
3.Delete element form tree
4.Tree Traversal
5.Depth of tree
6.Search Node
7.Number of nodes at each level
8.Total Number of Nodes
9.Clear Screen
10.Exit
>4
Tree Traversal
Pre-Order Traversal  :1 2 4 9 5 3 6 7
In-Order Traversal   :9 4 2 5 1 6 3 7
Post-Order Traversal :9 4 5 2 6 7 3 1
Level-Order Traversal:1 2 3 4 5 6 7 9

************************************************
**********  Tree Operations  **********
************************************************

1.Create Tree
2.Insert element in tree
3.Delete element form tree
4.Tree Traversal
5.Depth of tree
6.Search Node
7.Number of nodes at each level
8.Total Number of Nodes
9.Clear Screen
10.Exit
>5
Depth of Tree is 3.

************************************************
**********  Tree Operations  **********
```

```
************************************************

1.Create Tree
2.Insert element in tree
3.Delete element form tree
4.Tree Traversal
5.Depth of tree
6.Search Node
7.Number of nodes at each level
8.Total Number of Nodes
9.Clear Screen
10.Exit
>6
Search Node
Enter Element: 5
Element in Tree

************************************************
**********  Tree Operations  **********
************************************************

1.Create Tree
2.Insert element in tree
3.Delete element form tree
4.Tree Traversal
5.Depth of tree
6.Search Node
7.Number of nodes at each level
8.Total Number of Nodes
9.Clear Screen
10.Exit
>6
Search Node
Enter Element: 10
Element Not in Tree

************************************************
**********  Tree Operations  **********
************************************************

1.Create Tree
2.Insert element in tree
3.Delete element form tree
4.Tree Traversal
5.Depth of tree
6.Search Node
7.Number of nodes at each level
8.Total Number of Nodes
9.Clear Screen
```

```
10.Exit
>7
Number of nodes at each level
Level 1 has 1 nodes
Level 2 has 2 nodes
Level 3 has 4 nodes
Level 4 has 1 nodes
Level 5 has 0 nodes

************************************************
**********  Tree Operations  **********
************************************************

1.Create Tree
2.Insert element in tree
3.Delete element form tree
4.Tree Traversal
5.Depth of tree
6.Search Node
7.Number of nodes at each level
8.Total Number of Nodes
9.Clear Screen
10.Exit
>8
Total Number of Nodes in Tree are 8.

************************************************
**********  Tree Operations  **********
************************************************

1.Create Tree
2.Insert element in tree
3.Delete element form tree
4.Tree Traversal
5.Depth of tree
6.Search Node
7.Number of nodes at each level
8.Total Number of Nodes
9.Clear Screen
10.Exit
>3

************************************************
**********  Tree Operations  **********
************************************************

1.Create Tree
2.Insert element in tree
3.Delete element form tree
```

4.Tree Traversal
5.Depth of tree
6.Search Node
7.Number of nodes at each level
8.Total Number of Nodes
9.Clear Screen
10.Exit
> 10

Q2. Create a BST data structure, with all operations mentioned above for a tree.

Program:

```
// implementation of Binary search tree

#include <iostream>
#define SPACE 10

using namespace std;
template <class T>
class treeNode
{
public:
   T data;
   treeNode *left;
   treeNode *right;
   treeNode<T> *next;
   treeNode()
   {
      data = 0;
      left = right = NULL;
   }
   treeNode(T d)
   {
      data = d;
      left = NULL;
      right = NULL;
   }
};
template <class T>
class queue
{
public:
   treeNode<T> *front;
   treeNode<T> *rear;

   queue()
   {
      front = NULL;
      rear = NULL;
   }
   bool isEmpty()
   {
      treeNode<T> *tempf = front;
      treeNode<T> *tempr = rear;
```

```
    if (tempf == NULL)
    {
        return true;
    }
    else
    {
        return false;
    }
}
void enqueue(treeNode<T> *n)
{
    if (front == NULL)
    {
        front = n;
        rear = n;
    }
    else
    {
        rear->next = n;
        rear = n;
    }
}
treeNode<T> *dequeue()
{
    treeNode<T> *temp = NULL;
    if (isEmpty())
    {
        return NULL;
    }
    else
    {
        if (front == rear)
        {
            temp = front;
            front = NULL;
            rear = NULL;
            return temp;
        }
        else
        {
            temp = front;
            front = front->next;
```

```
            return temp;
        }
    }
  }
};
template <class T>
class BST
{
public:
  treeNode<T> *root;
  T element;
  BST()
  {
    root = NULL;
  }
  bool isEmpty()
  {
    if (root == NULL)
    {
      return true;
    }
    else
    {
      return false;
    }
  }
  void insertNode(treeNode<T> *n)
  {
    if (root == NULL)
    {
      root = n;
      cout << "Value inserted at root";
    }
    else
    {
      treeNode<T> *temp = root;
      while (temp != NULL)
      {
        if (n->data == temp->data)
        {
          cout << "No duplicates Allowed in BST" << endl;
          break;
        }
```

```
            else if ((n->data < temp->data) && (temp->left == NULL))
            {
                temp->left = n;
                cout << "Value inserted to left" << endl;
                break;
            }
            else if (n->data < temp->data)
            {
                temp = temp->left;
            }
            else if ((n->data > temp->data) && (temp->right == NULL))
            {
                temp->right = n;
                cout << "Value inserted to right" << endl;
                break;
            }
            else
            {
                temp = temp->right;
            }
        }
    }
}
void printBST(treeNode<int> *r, int space)
{
    if (r == NULL)
    {
        return;
    }
    space += SPACE;
    printBST(r->right, space);
    cout << endl;
    for (int i = SPACE; i < space; i++)
        cout << " ";
    cout << r->data << endl;
    printBST(r->left, space);
}
void printPreOrder(treeNode<T> *n)
{
    if (n == NULL)
    {
        return;
```

```
    }
    cout << n->data << " ";
    printPreOrder(n->left);
    printPreOrder(n->right);
}
void printInOrder(treeNode<T> *n)
{
    if (n == NULL)
    {
        return;
    }
    printInOrder(n->left);
    cout << n->data << " ";
    printInOrder(n->right);
}
void printPostOrder(treeNode<T> *n)
{
    if (n == NULL)
    {
        return;
    }
    printPostOrder(n->left);
    printPostOrder(n->right);
    cout << n->data << " ";
}
void printGivenLevel(treeNode<T> *n, int level) //Used for LevelOrderTraversal
{
    if (n == NULL)
    {
        return;
    }
    else if (level == 0)
    {
        cout << n->data << " ";
    }
    else
    {
        printGivenLevel(n->left, level - 1);
        printGivenLevel(n->right, level - 1);
    }
}
void printLevelOrder(treeNode<T> *n)
```

```
{
   int h = DepthOfTree(n);
   for (int i = 0; i <= h; i++)
      printGivenLevel(n, i);
}
int DepthOfTree(treeNode<T> *root)
{
   if (root == NULL)
   {
      return 0;
   }
   else
   {
      int x = DepthOfTree(root->left);
      int y = DepthOfTree(root->right);
      if (x > y)
      {
         return x + 1;
      }
      else
      {
         return y + 1;
      }
   }
}
int TotalNumberOfNodesInTree(treeNode<T> *root)
{
   if (root == NULL)
   {
      return 0;
   }
   else
   {
      int x = TotalNumberOfNodesInTree(root->left);
      int y = TotalNumberOfNodesInTree(root->right);
      return x + y + 1;
   }
}
int CalcNodesAtEachLevel(treeNode<T> *n, int curr, int desired)
{
   if (n == NULL)
      return 0;
```

```
        if (curr == desired)
            return 1;
        else
        {
            return CalcNodesAtEachLevel(n->left, curr + 1, desired) + CalcNodesAtEachLevel(n->right,
curr + 1, desired);
        }
    }
    void printNumberOfNodesAtEachLevel(treeNode<T> *root)
    {
        int h = DepthOfTree(root);
        for (int i = 0; i < h; i++)
        {
            int count = CalcNodesAtEachLevel(root, 0, i);
            cout << "At Level " << i + 1 << " ,Number of Nodes are " << count << "." << endl;
        }
    }
    treeNode<T> *minValueNode(treeNode<T> *node)
    {
        treeNode<T> *current = node;
        /* loop down to find the leftmost leaf */
        while (current->left != NULL)
        {
            current = current->left;
        }
        return current;
    }
    treeNode<T> *deleteNode(treeNode<T> *r, int v)
    {
        // base case
        if (r == NULL)
        {
            return NULL;
        }
        // If the key to be deleted is smaller than the root's key,
        // then it lies in left subtree
        else if (v < r->data)
        {
            r->left = deleteNode(r->left, v);
        }
        // If the key to be deleted is greater than the root's key,
        // then it lies in right subtree
```

```
    else if (v > r->data)
    {
        r->right = deleteNode(r->right, v);
    }
    // if key is same as root's key, then This is the node to be deleted
    else
    {
        // node with only one child or no child
        if (r->left == NULL)
        {
            treeNode<T> *temp = r->right;
            delete r;
            return temp;
        }
        else if (r->right == NULL)
        {
            treeNode<T> *temp = r->left;
            delete r;
            return temp;
        }
        else
        {
            // node with two children: Get the inorder successor (smallest
            // in the right subtree)
            treeNode<T> *temp = minValueNode(r->right);
            // Copy the inorder successor's content to this node
            r->data = temp->data;
            // Delete the inorder successor
            r->right = deleteNode(r->right, temp->data);
            //deleteNode(r->right, temp->data);
        }
    }
    return r;
}
T CheckIfNodeExists(T n)
{
    queue<T> q;
    q.enqueue(root);
    treeNode<T> *temp = root;
    while (temp->data != n && !q.isEmpty())
    {
        temp = q.dequeue();
```

```
            if (temp->left)
            {
                q.enqueue(temp->left);
            }
            if (temp->right)
            {
                q.enqueue(temp->right);
            }
        }
        if (temp->data == n)
        {
            return n;
        }
        else
        {
            return 0;
        }
    }
};
int main()
{
    int user_input;
    BST<int> obj;
    cout <<"\n*************************************************"
cout <<"\n********** Operations On Binary Search Tree **********"
cout <<"\n*************************************************";
    do
    {
        treeNode<int> *temp = new treeNode<int>();
        cout << "\nSelect Operation: " << endl;
        cout << "1.Insert Node\n2.Search Node\n3.Delete Node\n4.Print BST values \n5.Tree Traversal\
n6.Number of Nodes at each level\n7.Total Number of Nodes is Tree\n8.Depth Of Tree\n9.Clear\
n10.Exit" << endl;
        cout << ">";
        cin >> user_input;
        switch (user_input)
        {
        case 1:
        {
            cout << "Insert" << endl;
            cout << ">";
            cin >> obj.element;
            temp->data = obj.element;
```

```
    if (obj.root == NULL)
    {
        obj.insertNode(temp);
    }
    else if (obj.element == obj.CheckIfNodeExists(obj.element))
    {
        cout << "No Duplicates are Allowed" << endl;
    }
    else
    {
        obj.insertNode(temp);
    }
    break;
}
case 2:
{
    cout << "Search Node" << endl;
    cout << "Enter Value to be searched : ";
    cin >> obj.element;
    temp->data = obj.element;
    if (obj.element == obj.CheckIfNodeExists(obj.element))
    {
        cout << "Element is Present in Tree." << endl;
    }
    else
    {
        cout << "Element is Not Present in Tree." << endl;
    }
    break;
}
case 3:
{
    cout << "Delete Node" << endl;
    cout << "Enter VALUE of TREE NODE to DELETE in BST: ";
    cin >> obj.element;
    temp->data = obj.element;
    if (obj.element == obj.CheckIfNodeExists(obj.element))
    {
        temp = obj.deleteNode(obj.root, obj.element);
        cout << "Node " << temp->data << " Deleted" << endl;
    }
    else
```

```
    {
        cout << "Node NOT found" << endl;
    }
    break;
}
case 4:
{
    cout << "Print BST" << endl;
    obj.printBST(obj.root, 5);
    break;
}
case 5:
{
    obj.printBST(obj.root, 5);
    cout << "\nPre-Order Traversal of BST   : ";
    obj.printPreOrder(obj.root);
    cout << "\nIn-Order Traversal of BST    : ";
    obj.printInOrder(obj.root);
    cout << "\nPost-Order Traversal of BST  : ";
    obj.printPostOrder(obj.root);
    cout << "\nLevel order Traversal of BST : ";
    obj.printLevelOrder(obj.root);
    break;
}
case 6:
{
    cout << "Number of Nodes at each Level are :" << endl;
    obj.printNumberOfNodesAtEachLevel(obj.root);
    break;
}
case 7:
{
    cout << "Total Number of Nodes in Tree are : " << obj.TotalNumberOfNodesInTree(obj.root);
    break;
}
case 8:
{
    cout << "Depth of Tree is " << obj.DepthOfTree(obj.root) - 1;
    break;
}
case 9:
    system("clear");
```

```
            break;
        case 10:
            cout << "\n********Program Ended********\n";
            break;
        default:
            cout << "Invalid Input" << endl;
        }
    } while (user_input != 10);
    return 0;
}
```

Output:

```
**************************************************
********** Operations On Binary Search Tree **********
**************************************************
Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print BST values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>1
Insert
>55
Value inserted at root
Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print BST values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>1
Insert
>22
Value inserted to left

Select Operation:
```

1.Insert Node
2.Search Node
3.Delete Node
4.Print BST values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>1
Insert
>11
Value inserted to left

Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print BST values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>1
Insert
>33
Value inserted to right

Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print BST values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>1
Insert
>77
Value inserted to right

Select Operation:
1.Insert Node

2.Search Node
3.Delete Node
4.Print BST values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>1
Insert
>66
Value inserted to left

Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print BST values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>1
Insert
>88
Value inserted to right

Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print BST values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>1
Insert
>55
No Duplicates are Allowed

Select Operation:
1.Insert Node
2.Search Node

3.Delete Node
4.Print BST values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>1
Insert
>33
No Duplicates are Allowed

Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print BST values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>2
Search Node
Enter Value to be searched : 22
Element is Present in Tree.

Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print BST values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>2
Search Node
Enter Value to be searched : 23
Element is Not Present in Tree.

Select Operation:
1.Insert Node
2.Search Node
3.Delete Node

4.Print BST values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>4
Print BST

```
                88

        77

                66

    55

                33

        22

                11
```

Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print BST values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>5

```
                88

        77

                66

    55

                33
```

22

11

Pre-Order Traversal of BST : 55 22 11 33 77 66 88
In-Order Traversal of BST : 11 22 33 55 66 77 88
Post-Order Traversal of BST : 11 33 22 66 88 77 55
Level order Traversal of BST : 55 22 77 11 33 66 88
Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print BST values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>6
Number of Nodes at each Level are :
At Level 1 ,Number of Nodes are 1.
At Level 2 ,Number of Nodes are 2.
At Level 3 ,Number of Nodes are 4.

Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print BST values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>7
Total Number of Nodes in Tree are : 7
Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print BST values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit

```
>8
Depth of Tree is 2
Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print BST values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>3
Delete Node
Enter VALUE of TREE NODE to DELETE in BST: 11
Node 55 Deleted

Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print BST values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>4
Print BST

                88

        77

                66

    55

                33

        22

Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
```

4.Print BST values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>5

```
                88

          77

                66

    55

                33

          22
```

Pre-Order Traversal of BST : 55 22 33 77 66 88
In-Order Traversal of BST : 22 33 55 66 77 88
Post-Order Traversal of BST : 33 22 66 88 77 55
Level order Traversal of BST : 55 22 77 33 66 88
Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print BST values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>3
Delete Node
Enter VALUE of TREE NODE to DELETE in BST: 66
Node 55 Deleted

Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print BST values
5.Tree Traversal
6.Number of Nodes at each level

7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>4
Print BST

```
            88

      77

  55

            33

      22
```

Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print BST values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>5

```
            88

      77

  55

            33

      22
```

Pre-Order Traversal of BST : 55 22 33 77 88
In-Order Traversal of BST : 22 33 55 77 88
Post-Order Traversal of BST : 33 22 88 77 55
Level order Traversal of BST : 55 22 77 33 88
Select Operation:
1.Insert Node
2.Search Node
3.Delete Node

4.Print BST values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>10

********Program Ended********

Q3. Design a AVL tree, with all valid operations above

Program:

```
#include <iostream>
# define SPACE 10
using namespace std;
template <class T>
class treeNode
{
public:
   treeNode<T> *left;
   T data;
   int height;
   treeNode<T> *right;
   treeNode<T> *next;
   treeNode()
   {
      left = right = NULL;
      data = height = 0;
   }
   treeNode(T d)
   {
      left = right = NULL;
      data = d;
      height = 0;
   }
};
template <class T>
class queue
{
public:
   treeNode<T> *front;
   treeNode<T> *rear;
```

```
queue()
{
   front = NULL;
   rear = NULL;
}
bool isEmpty()
{
   treeNode<T> *tempf = front;
   treeNode<T> *tempr = rear;
   if (tempf == NULL)
   {
      return true;
   }
   else
   {
      return false;
   }
}
void enqueue(treeNode<T> *n)
{
   if (front == NULL)
   {
      front = n;
      rear = n;
   }
   else
   {
      rear->next = n;
      rear = n;
   }
}
treeNode<T> *dequeue()
{
   treeNode<T> *temp = NULL;
   if (isEmpty())
   {
      return NULL;
   }
   else
   {
      if (front == rear)
      {
```

```
            temp = front;
            front = NULL;
            rear = NULL;
            return temp;
         }
         else
         {
            temp = front;
            front = front->next;
            return temp;
         }
      }
   }
};
template <class T>
class AVLT
{
public:
   treeNode<T> *root;
   T element;
   AVLT()
   {
      root = NULL;
   }
   int BalanceFactor(treeNode<T> *p)
   {
      int hl, hr;
      hl = p && p->left?p->left->height:0;
      hr = p && p->right? p->right->height:0;
      return hl - hr;
   }
   int NodeHeight(treeNode<T> *p)
   {
      int hl, hr;
      hl = p && p->left ? p->left->height : 0;
      hr = p && p->right ? p->right->height : 0;
      return hl > hr ? (hl + 1) : (hr + 1);
   }
   treeNode<T> *LLRotation(treeNode<T> *p)
   {
      treeNode<T> *pl = p->left;
      treeNode<T> *plr = pl->right;
      p->left = plr;
```

```
    pl->right = p;
    p->height = NodeHeight(p);
    pl->height = NodeHeight(pl);
    if (root == p)
        root = pl;
    return pl;
}
treeNode<T> *RRRotation(treeNode<T> *p)
{
    treeNode<T> *pr = p->right;
    treeNode<T> *prl = pr->left;
    p->right = prl;
    pr->left = p;
    p->height = NodeHeight(p);
    pr->height = NodeHeight(pr);
    if (root = p)
        root = pr;
    return pr;
}
treeNode<T> *LRRotation(treeNode<T> *p)
{
    treeNode<T> *pl = p->left;
    treeNode<T> *plr = pl->right;
    p->left = plr->right;
    pl->right = plr->left;
    plr->left = pl;
    plr->right = p;
    p->height = NodeHeight(p);
    pl->height = NodeHeight(pl);
    plr->height = NodeHeight(plr);
    if (root = p)
        root = plr;
    return plr;
}
treeNode<T> *RLRotation(treeNode<T> *p)
{
    treeNode<T> *pr = p->right;
    treeNode<T> *prl = pr->left;
    p->right = prl->left;
    prl->left = p;
    prl->right = pr;
    pr->left = prl->right;
```

```
    p->height = NodeHeight(p);
    pr->height = NodeHeight(pr);
    prl->height = NodeHeight(prl);
    if (root = p)
        root = prl;
    return prl;
}
treeNode<T>* insert(treeNode<T> *p,T nd)
{
    if(p==NULL)
    {
        treeNode<T>* temp=new treeNode<int>();
        temp->data=nd;
        temp->height=1;
        return temp;
    }
    if(nd<p->data)
    {
        p->left=insert(p->left,nd);
    }
    else if(nd>p->data)
    {
        p->right=insert(p->right,nd);
    }
    p->height=NodeHeight(p);
    if (BalanceFactor(p) == 2 && BalanceFactor(p -> left) == 1)
         return LLRotation(p);
    else if (BalanceFactor(p) == 2 && BalanceFactor(p -> left) == -1)
         return LRRotation(p);
    else if (BalanceFactor(p) == -2 && BalanceFactor(p -> right) == -1)
         return RRRotation(p);
    else if (BalanceFactor(p) == -2 && BalanceFactor(p -> right) == 1)
         return RLRotation(p);
    return p;
}
int DepthOfTree(treeNode<T>* root)
{
    if(root==NULL)
    {
        return 0;
    }
    int x=DepthOfTree(root->left);
```

```
    int y=DepthOfTree(root->right);
    return x>y?(x+1):(y+1);
  }
  int CalcNodesAtEachLevel(treeNode<T>* n,int current,int desired)
  {
    if(n==NULL)
    {
      return 0;
    }
    if(current=desired)
    {
      return 1;
    }
    else
    {
      return CalcNodesAtEachLevel(n->left,current+1,desired) + CalcNodesAtEachLevel(n-
>right,current+1,desired);
    }

  }
  void PrintNumberOfNodesAtEachLevel(treeNode<T>* root)
  {
    int h=DepthOfTree(root);
    for(int i=0;i<h;i++)
    {
      int count=CalcNodesAtEachLevel(root,0,i);
      cout <<"At Level " << i+1 << ", Number of Nodes are " << count <<"."<<endl;
    }
  }
  void printAVLT(treeNode<int> *r, int space)
  {
    if (r == NULL)
    {
      return;
    }
    space += SPACE;
    printAVLT(r->right, space);
    cout << endl;
    for (int i = SPACE; i < space; i++)
      cout << " ";
    cout << r->data << endl;
    printAVLT(r->left, space);
```

```
}
void PreOrder(treeNode<T>* root)
{
   if(root==NULL)
   {
      return;
   }
   else
   {
      cout << root->data <<" ";
      PreOrder(root->left);
      PreOrder(root->right);
   }
}
void InOrder(treeNode<T>* root)
{
   if(root==NULL)
   {
      return;
   }
   else
   {
      InOrder(root->left);
      cout << root->data <<" ";
      InOrder(root->right);
   }
}
void PostOrder(treeNode<T>* root)
{
   if(root==NULL)
   {
      return;
   }
   else
   {
      PostOrder(root->left);
      PostOrder(root->right);
      cout << root->data <<" ";
   }
}
void printGivenLevel(treeNode<T>* n,int level) //Used for LevelOrderTraversal
{
```

```
    if(n==NULL)
    {
        return;
    }
    else if(level==0)
    {
        cout << n->data << " ";
    }
    else
    {
        printGivenLevel(n->left,level-1);
        printGivenLevel(n->right,level-1);
    }
}
void LevelOrder(treeNode<T> *n)
{
    int h = DepthOfTree(n);
    for (int i = 0; i <= h; i++)
         printGivenLevel(n, i);
}
int TotalNumberOfNodesInTree(treeNode<T>* root)
{
    if(root==NULL)
    {
        return 0;
    }
    else
    {
        int x=TotalNumberOfNodesInTree(root->left);
        int y=TotalNumberOfNodesInTree(root->right);
        return x+y+1;
    }

}
treeNode<T> *minValueNode(treeNode<T> *node)
{
    treeNode<T> *current = node;
    /* loop down to find the leftmost leaf */
    while (current->left != NULL)
    {
        current = current->left;
    }
```

```
    return current;
}
treeNode<T> *deleteNode(treeNode<T> *p, int v)
{
    // base case
    if (p == NULL)
    {
        return NULL;
    }
    // If the key to be deleted is smaller than the root's key,
    // then it lies in left subtree
    else if (v < p->data)
    {
        p->left = deleteNode(p->left, v);
    }
    // If the key to be deleted is greater than the root's key,
    // then it lies in right subtree
    else if (v > p->data)
    {
        p->right = deleteNode(p->right, v);
    }
    // if key is same as root's key, then This is the node to be deleted
    else
    {
        // node with only one child or no child
        if (p->left == NULL)
        {
            treeNode<T> *temp = p->right;
            delete p;
            return temp;
        }
        else if (p->right == NULL)
        {
            treeNode<T> *temp = p->left;
            delete p;
            return temp;
        }
        else
        {
            // node with two children: Get the inorder successor (smallest
            // in the right subtree)
            treeNode<T> *temp = minValueNode(p->right);
```

```
            // Copy the inorder successor's content to this node
            p->data = temp->data;
            // Delete the inorder successor
            p->right = deleteNode(p->right, temp->data);
            //deleteNode(p->right, temp->data);
        }
    return p;
    }
    p->height=NodeHeight(p);
    if (BalanceFactor(p) == 2 && BalanceFactor(p -> left) == 1)
        return LLRotation(p);
    else if (BalanceFactor(p) == 2 && BalanceFactor(p -> left) == -1)
        return LRRotation(p);
    else if (BalanceFactor(p) == -2 && BalanceFactor(p -> right) == -1)
        return RRRotation(p);
    else if (BalanceFactor(p) == -2 && BalanceFactor(p -> right) == 1)
        return RLRotation(p);
    return p;
}
T CheckIfNodeExists(T n)
{
    queue<T> q;
    q.enqueue(root);
    treeNode<T> *temp = root;
    while (temp->data != n && !q.isEmpty())
    {
        temp = q.dequeue();
        if (temp->left)
        {
            q.enqueue(temp->left);
        }
        if (temp->right)
        {
            q.enqueue(temp->right);
        }
    }
    if (temp->data == n)
    {
        return n;
    }
    else
    {
```

```
            return 0;
        }
    }
};
int main()
{
    AVLT<int> obj;
   int user_input;
    do
    {
        treeNode<int> *temp = new treeNode<int>();
        cout << "\nSelect Operation: " << endl;
        cout << "1.Insert Node\n2.Search Node\n3.Delete Node\n4.Print AVL Tree values \n5.Tree
Traversal\n6.Number of Nodes at each level\n7.Total Number of Nodes is Tree\n8.Depth Of Tree\
n9.Clear\n10.Exit" << endl;
        cout << ">";
        cin >> user_input;
        switch (user_input)
        {
        case 1:
        {
            cout << "Insert" << endl;
            cout << ">";
            cin >> obj.element;
            temp->data = obj.element;
            if (obj.root == NULL)
            {
                obj.root=obj.insert(obj.root,obj.element);
            }
            else if (obj.element == obj.CheckIfNodeExists(obj.element))
            {
                cout << "No Duplicates are Allowed" << endl;
            }
            else
            {
                obj.insert(obj.root,obj.element);
            }
            break;
        }
        case 2:
        {
            cout << "Search Node" << endl;
            cout << "Enter Value to be searched : ";
```

```
        cin >> obj.element;
        temp->data = obj.element;
        if (obj.element == obj.CheckIfNodeExists(obj.element))
        {
            cout << "Element is Present in Tree." << endl;
        }
        else
        {
            cout << "Element is Not Present in Tree." << endl;
        }
        break;
    }
    case 3:
    {
        cout << "Delete Node" << endl;
        cout << "Enter VALUE of TREE NODE to DELETE in AVL Tree: ";
        cin >> obj.element;
        temp->data = obj.element;
        if (obj.element == obj.CheckIfNodeExists(obj.element))
        {
            temp = obj.deleteNode(obj.root, obj.element);
            cout << "Node " << obj.element << " Deleted" << endl;
        }
        else
        {
            cout << "Node NOT found" << endl;
        }
        break;
    }
    case 4:
    {
        cout << "Print AVL Tree" << endl;
        obj.printAVLT(obj.root, 5);
        break;
    }
    case 5:
    {
        obj.printAVLT(obj.root, 5);
        cout << "\nPre-Order Traversal of AVL Tree   : ";
        obj.PreOrder(obj.root);
        cout << "\nIn-Order Traversal of AVL Tree    : ";
        obj.InOrder(obj.root);
```

```
            cout << "\nPost-Order Traversal of AVL Tree  : ";
            obj.PostOrder(obj.root);
            cout << "\nLevel order Traversal of AVL Tree : ";
            obj.LevelOrder(obj.root);
            break;
        }
        case 6:
        {
            cout << "Number of Nodes at each Level are :" << endl;
            obj.PrintNumberOfNodesAtEachLevel(obj.root);
            break;
        }
        case 7:
        {
            cout << "Total Number of Nodes in Tree are : " << obj.TotalNumberOfNodesInTree(obj.root);
            break;
        }
        case 8:
        {
            cout << "Depth of Tree is " << obj.DepthOfTree(obj.root) - 1;
            break;
        }
        case 9:
            system("clear");
            break;
        case 10:
            cout << "\n********Program Ended********\n";
            break;
        default:
            cout << "Invalid Input" << endl;
        }
    } while (user_input != 10);
    return 0;
}
```

Output:

```
Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print AVL Tree values
5.Tree Traversal
6.Number of Nodes at each level
```

7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>1
Insert
>55

Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print AVL Tree values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>1
Insert
>66

Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print AVL Tree values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>1
Insert
>77

Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print AVL Tree values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit

```
>1
Insert
>88

Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print AVL Tree values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>4
Print AVL Tree

                88

        77

    66

        55

Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print AVL Tree values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>5

                88

        77

    66

        55

Pre-Order Traversal of AVL Tree   : 66 55 77 88
```

```
In-Order Traversal of AVL Tree    : 55 66 77 88
Post-Order Traversal of AVL Tree  : 55 88 77 66
Level order Traversal of AVL Tree : 66 55 77 88
Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print AVL Tree values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>6
Number of Nodes at each Level are :
At Level 1, Number of Nodes are 1.
At Level 2, Number of Nodes are 2.
At Level 3, Number of Nodes are 1.

Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print AVL Tree values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>7
Total Number of Nodes in Tree are : 4
Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print AVL Tree values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>8
Depth of Tree is 2

Select Operation:
1.Insert Node
```

2.Search Node
3.Delete Node
4.Print AVL Tree values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>2
Search Node
Enter Value to be searched : 55
Element is Present in Tree.

Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print AVL Tree values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>2
Search Node
Enter Value to be searched : 56
Element is Not Present in Tree.

Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print AVL Tree values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>3
Delete Node
Enter VALUE of TREE NODE to DELETE in AVL Tree: 55
Node 55 Deleted

Select Operation:
1.Insert Node
2.Search Node

```
3.Delete Node
4.Print AVL Tree values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>4
Print AVL Tree

            88

    77

            66

Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print AVL Tree values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
8.Depth Of Tree
9.Clear
10.Exit
>5

            88

    77

            66

Pre-Order Traversal of AVL Tree   : 77 66 88
In-Order Traversal of AVL Tree    : 66 77 88
Post-Order Traversal of AVL Tree  : 66 88 77
Level order Traversal of AVL Tree : 77 66 88
Select Operation:
1.Insert Node
2.Search Node
3.Delete Node
4.Print AVL Tree values
5.Tree Traversal
6.Number of Nodes at each level
7.Total Number of Nodes is Tree
```

8.Depth Of Tree
9.Clear
10.Exit
>10

********Program Ended********

Q4. Design a tree for a given infix expression. Perform the conversion to pre and postfix expressions using the traversal.

Program:

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

typedef struct node
{
    char data;
    struct node *left, *right;
} * nptr;

nptr newNode(char c)
{
    nptr n = new node;
    n->data = c;
    n->left = n->right = nullptr;
    return n;
}

nptr build(string& s)
{

    // Stack to hold nodes
    stack<nptr> stN;

    // Stack to hold chars
    stack<char> stC;
    nptr t, t1, t2;

    // Prioritising the operators
    int p[123] = { 0 };
    p['+'] = p['-'] = 1, p['/'] = p['*'] = 2, p['^'] = 3,
    p[')'] = 0;

    for (int i = 0; i < s.length(); i++)
```

```
{
    if (s[i] == '(') {

        // Push '(' in char stack
        stC.push(s[i]);
    }

    // Push the operands in node stack
    else if (isalpha(s[i]))
    {
        t = newNode(s[i]);
        stN.push(t);
    }
    else if (p[s[i]] > 0)
    {
        // If an operator with lower or
        // same associativity appears
        while (
            !stC.empty() && stC.top() != '('
            && ((s[i] != '^' && p[stC.top()] >= p[s[i]])
                || (s[i] == '^'
                    && p[stC.top()] > p[s[i]])))
        {

            // Get and remove the top element
            // from the character stack
            t = newNode(stC.top());
            stC.pop();

            // Get and remove the top element
            // from the node stack
            t1 = stN.top();
            stN.pop();

            // Get and remove the currently top
            // element from the node stack
            t2 = stN.top();
            stN.pop();

            // Update the tree
            t->left = t2;
            t->right = t1;
```

```
            // Push the node to the node stack
            stN.push(t);
        }

        // Push s[i] to char stack
        stC.push(s[i]);
    }
    else if (s[i] == ')') {
        while (!stC.empty() && stC.top() != '(')
        {
            t = newNode(stC.top());
            stC.pop();
            t1 = stN.top();
            stN.pop();
            t2 = stN.top();
            stN.pop();
            t->left = t2;
            t->right = t1;
            stN.push(t);
        }
        stC.pop();
    }
  }
  t = stN.top();
  return t;
}


void PostOrder(nptr root)
{
  if (root)
  {
    PostOrder(root->left);
    PostOrder(root->right);
    cout << root->data;
  }
}
void PreOrder(nptr root)
{
  if (root)
  {
    cout << root->data;
    PreOrder(root->left);
```

```
        PreOrder(root->right);
    }
}

int main()
{

    string s;
    cout <<"\nConverting InFix Expression to PreFix and PostFix using expression Tree\n"<<endl;
    cout <<"Enter Infix Expression: ";
    cin >> s;
    s = "(" + s;
    s += ")";
    nptr root = build(s);
    cout <<"PreFix Expression using PreOrder Traversal: ";
    PreOrder(root);
    cout <<endl;
    cout <<"PostFix Expression using PostOrder Traversal: ";
    PostOrder(root);
    cout <<endl;

    return 0;
}
```

Output:

```
Converting InFix Expression to PreFix and PostFix using expression Tree

Enter Infix Expression: (a+b)*(c+d)*(e+f)/g
PreFix Expression using PreOrder Traversal: /**+ab+cd+efg
PostFix Expression using PostOrder Traversal: ab+cd+*ef+*g/
```

Q5. Define a hash function with a linear probing strategy. Handle collisions using a list at each node.

Program:

```
#include<iostream>
#include<limits>

using namespace std;
#define SIZE 10

int hashfun(int key)
{
   return key%SIZE;
}
int probe(int H[],int key)
{
   int index=hashfun(key);
   int i=0;
   while(H[(index+i)%SIZE]!=0)
   {
      i++;
      if(i>=10)
      {
         break;
      }
   }
   return (index+i)%SIZE;
}
void insert(int H[],int key)
{
   int index=hashfun(key);
   if(H[index]!=0)
      index=probe(H,key);
   H[index]=key;
   cout << "Element inserted successfully at index " << index << "."<<endl;
}
int search(int H[],int key)
{
   int index=hashfun(key);
   int i=0;
   while(H[(index+i)%SIZE]!=key)
      i++;
   return (index+i)%SIZE;

}
```

```
int main()
{
   int HT[10]={};
   int index,input;
   do
   {
      cout << "\n\nHashing with Linear Probing"<<endl;
      cout << "\n0.Info about Hashing with linear probing\n1.Insert Element\n2.Search\n3.Print Hash
Table\n4.Clear Screen\n5.Quit\n>";
      cin >> input;
      std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
      std::cin.clear();
      switch(input)
      {
         case 0:
            {
               cout <<"\n********** Hash Table With linear probing
**********"<<endl;
               cout << "\n1.Load factor (which is ratio of Total number of elements in hash" <<      "\n
table and hash table size) should be <= 0.5";
               cout << "\n2.Hence half of the hash table remains empty.";
               cout << "\n3.It is not suggested to use hash table with deletion operation."<<
                     "\n  Since when Smashing occurs with shift element to next index,"<<
                     "\n  so during deletion if we delete a element then we have to shift all" <<"\n  the
elements by (-1) index.";
               cout <<"\n4.During Searching if we find an empty space in Hash Table then we stop "<<"\
n  the search at that index because it is certain that the element is "<<"\n  not present in the table";
            }
            break;
         case 1:
            {
            cout << "Insert Operation"<<endl;
            cout << "Enter element which you want to insert: ";
            cin >> index;
            int count=1;
            for(int i=0;i<SIZE;i++)
            {
               if(HT[i]==0)
               {
                  continue;
               }
               else
               {
```

```
            count++;
          }
       }
       if(count>5)
       {
          cout << "Exceeding Load factor."<<endl;
          cout << "(Load factor should be less than or equal to  0.5)"<<endl;
          cout << "Cannot add more entries to hash table."<<endl;
       }
       else
       {
          insert(HT,index);
       }
       count=1;
       break;
       }
    case 2:
       {
       cout << "Search Operation"<<endl;
       cout << "Enter Element which you want to search: ";
       cin >> index;
       int temp=search(HT,index);
       if(temp>=0&&temp<SIZE)
       {
          cout << "Element found at index "<< temp<<endl;
       }
       else
       {
          cout << "Element not found in hash table"<<endl;
       }
       break;
       }
    case 3:
       {
       cout << "The Hash Table is "<<endl;
       int sz=sizeof(HT)/sizeof(int);
       for(int i=0;i<sz;i++)
       {
          cout << HT[i]<<" ";
       }
       cout <<endl;
       break;
```

```
            }
        case 4:
            {
            system("clear");
            break;
            }
        case 5:
            cout << “\nProgram Ended" <<endl;
            break;
        default:
            cout << "Invalid input\nTry again"<<endl;
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
            std::cin.clear();
      }
   } while (input!=5);
   return 0;
}
```

Output:

Hashing with Linear Probing

0.Info about Hashing with linear probing
1.Insert Element
2.Search
3.Print Hash Table
4.Clear Screen
5.Quit
>0

********** Hash Table With linear probing **********

1.Load factor (which is ratio of Total number of elements in hash
table and hash table size) should be <= 0.5
2.Hence half of the hash table remains empty.
3.It is not suggested to use hash table with deletion operation.
Since when Smashing occurs with shift element to next index,
so during deletion if we delete a element then we have to shift all
the elements by (-1) index.
4.During Searching if we find an empty space in Hash Table then we stop
the search at that index because it is certain that the element is
not present in the table

Hashing with Linear Probing

0.Info about Hashing with linear probing

1.Insert Element
2.Search
3.Print Hash Table
4.Clear Screen
5.Quit
>1
Insert Operation
Enter element which you want to insert: 1
Element inserted successfully at index 1.

Hashing with Linear Probing

0.Info about Hashing with linear probing
1.Insert Element
2.Search
3.Print Hash Table
4.Clear Screen
5.Quit
>1
Insert Operation
Enter element which you want to insert: 22
Element inserted successfully at index 2.

Hashing with Linear Probing

0.Info about Hashing with linear probing
1.Insert Element
2.Search
3.Print Hash Table
4.Clear Screen
5.Quit
>1
Insert Operation
Enter element which you want to insert: 2
Element inserted successfully at index 3.

Hashing with Linear Probing

0.Info about Hashing with linear probing
1.Insert Element
2.Search
3.Print Hash Table
4.Clear Screen
5.Quit
>1
Insert Operation
Enter element which you want to insert: 5
Element inserted successfully at index 5.

Hashing with Linear Probing

0.Info about Hashing with linear probing
1.Insert Element
2.Search
3.Print Hash Table
4.Clear Screen
5.Quit
>1
Insert Operation
Enter element which you want to insert: 8
Element inserted successfully at index 8.

Hashing with Linear Probing

0.Info about Hashing with linear probing
1.Insert Element
2.Search
3.Print Hash Table
4.Clear Screen
5.Quit
>1
Insert Operation
Enter element which you want to insert: 3
Exceeding Load factor.
(Load factor should be less than or equal to 0.5)
Cannot add more entries to hash table.

Hashing with Linear Probing

0.Info about Hashing with linear probing
1.Insert Element
2.Search
3.Print Hash Table
4.Clear Screen
5.Quit
>3
The Hash Table is
0 1 22 2 0 5 0 0 8 0

Hashing with Linear Probing

0.Info about Hashing with linear probing
1.Insert Element
2.Search
3.Print Hash Table
4.Clear Screen
5.Quit
>2

Search Operation
Enter Element which you want to search: 22
Element found at index 2

Hashing with Linear Probing

0.Info about Hashing with linear probing
1.Insert Element
2.Search
3.Print Hash Table
4.Clear Screen
5.Quit
>2
Search Operation
Enter Element which you want to search: 2
Element found at index 3

Hashing with Linear Probing

0.Info about Hashing with linear probing
1.Insert Element
2.Search
3.Print Hash Table
4.Clear Screen
5.Quit
>2
Search Operation
Enter Element which you want to search: 45
Element not found in hash table

Hashing with Linear Probing

0.Info about Hashing with linear probing
1.Insert Element
2.Search
3.Print Hash Table
4.Clear Screen
5.Quit
>5

Program Ended