# Department of Computing

# School of Electrical Engineering and Computer Science

**CS - 250: Data Structure and Algorithms**

**Class: BSCS 10AB**

**Lab 10 : Binary Search Tree – Part B**

**Date: 14th December, 2021**

**Time: 10:00 am – 12:50 pm   
&  
 02:00 pm – 4:50 pm**

# Instructor: Dr. Yasir Faheem

# Lab Engineer: Aftab Farooq

# 

# Lab 10 : Implementation of Binary Search Trees – Part B

**Introduction**

This lab is based on the implementation of Binary Search tree and its functions.

**Objectives**

The objectives of this lab are the following:

* Become familiar with implementation of binary search trees
* Study some statistics of binary search trees
* Write simple applications using binary search tree

**Tools/Software Requirement**

Visual Studio 2012 or gcc or g++

**Description**

In computer science, a binary search tree (BST), which may sometimes also be called an ordered or sorted binary tree, is a node-based binary tree data structure which has the following properties:

* The left sub-tree of a node contains only nodes with keys less than the node's key.
* The right sub-tree of a node contains only nodes with keys greater than the node's key.
* Both the left and right sub-trees must also be binary search trees.
* There must be no duplicate nodes.

In this lab, you will expand implement binary search tree, study some statistical properties of BST and write a simple application using the BST.

Here is a template of how your class/structure looks like.

class BST\_Node{

Template data;

BST\_Node \*LeftChild;

BST\_Node \*RightChild;

};

**Lab Task**

**Tasks**

**Previous Week’s Tasks:** In the previous lab you implemented the following operation of a binary search tree:

* bool IsEmpty();
* void Search(template value)
* Void InsertWithoutDuplication(template value)
* Void InsertWithDuplication(template value)
* Traversing a binary tree in pre-order, in-order and post-order.
* Implement a function that prints the smallest value of a BST.
* Implement a function that prints the largest value of a BST.
* Implement a function to calculate the height of a BST.
* Implement a function that calculates the depth of a BST.

**In today’s lab,** your task is to implement the following operations of a binary search tree:

Deleting a node. You may call your search(value) function. You should implement all the three cases:

Tree:

Main Function:

tre->Insert(20);

tre->Insert(30);

tre->Insert(10);

tre->Insert(15);

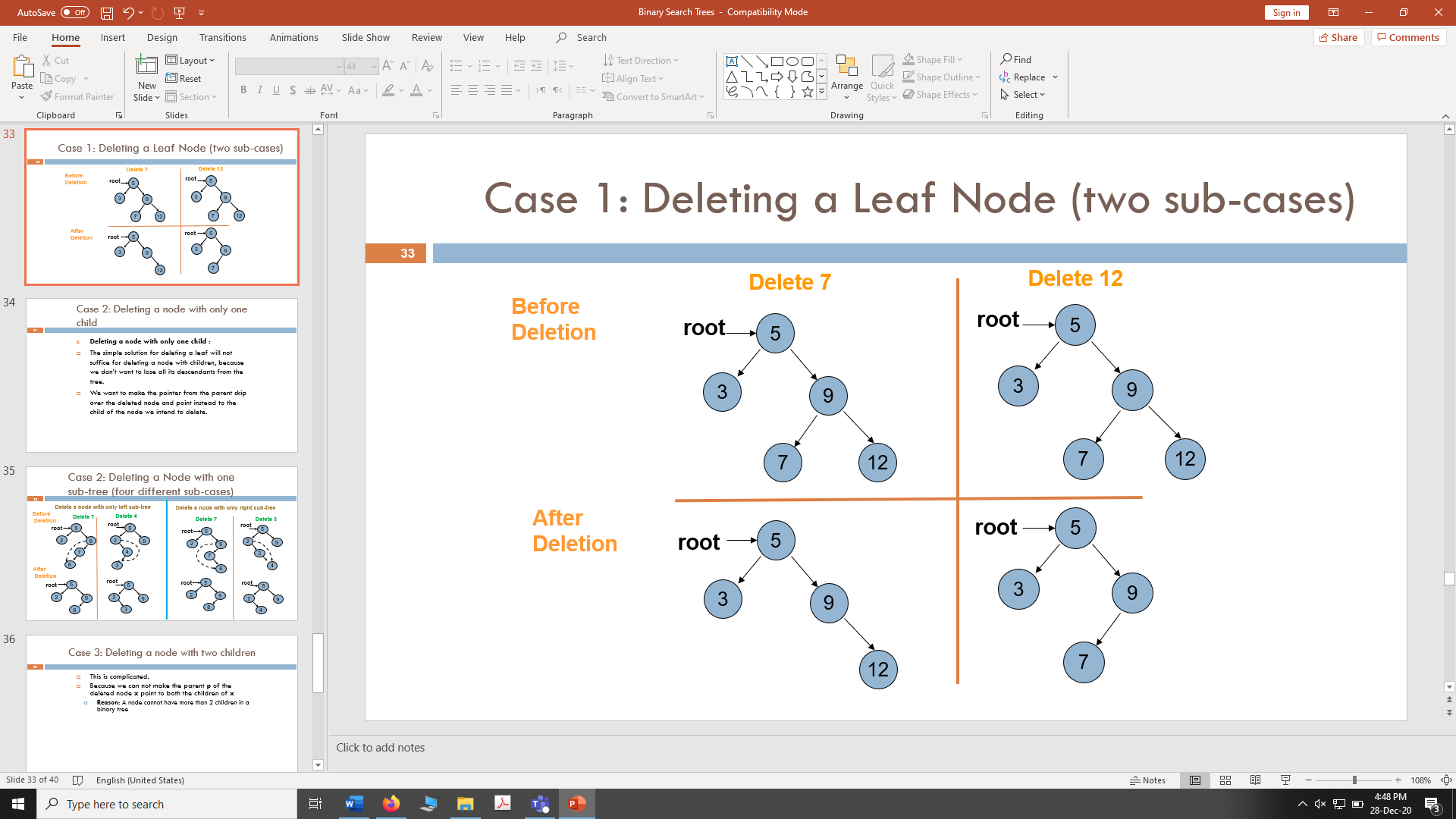
tre->Insert(5);

tre->Insert(25);

tre->Insert(27);

**a).** **Deleting a leaf node.**

Include the case in which the node being deleted is a root node.



**Code:**

void DeleteNode(tree\* node,int value)

{

int flag = Search(root, value); //searching the node

if (flag == 1)

{

if (ploc->left->data == value) //if value is found

{

if (loc->left == l && loc->right == r) //if node both left and right are null

{

ploc->left = l; //assign left of ploc is null

delete loc; // delete the node

}

}

else if (ploc->right->data == value )//if value is found

{

if (loc->left == l && loc->right == r)

{

ploc->right = r;//assign right of ploc is null

delete loc; // delete the node

}

}

}

}

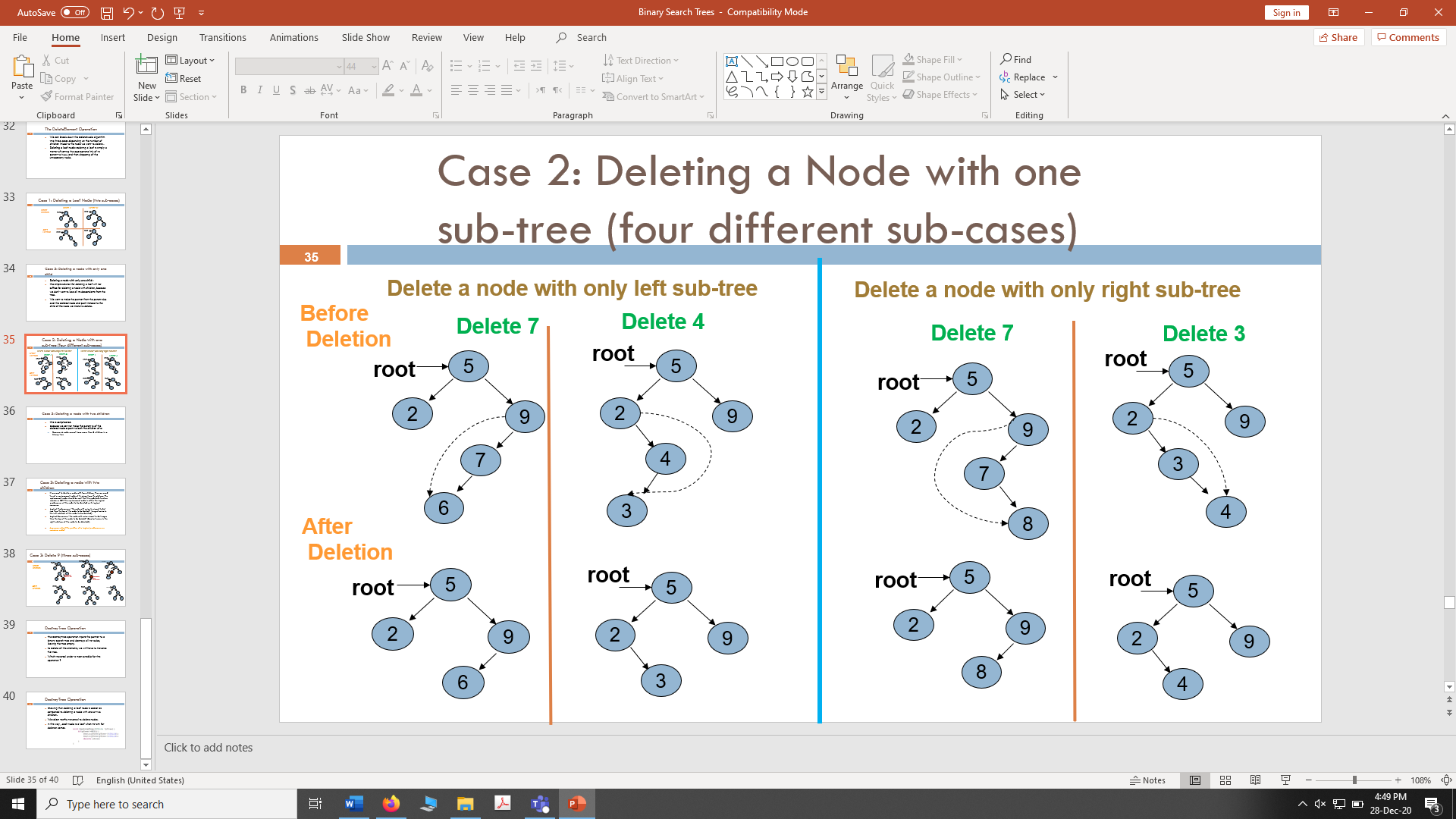
**Screenshot:**

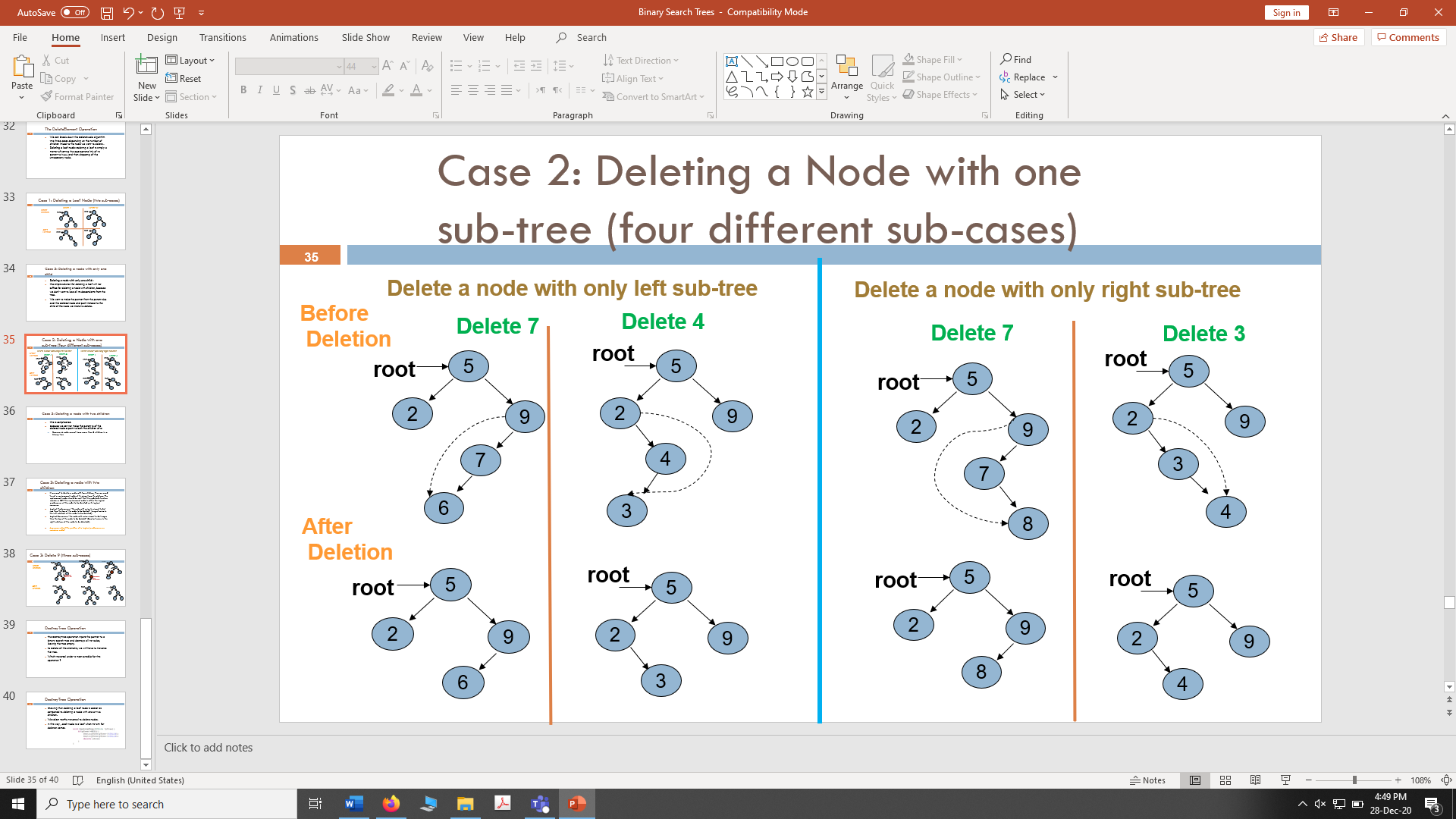
Text

Description automatically generated

**b). Deleting a node with only one sub-tree.**

Include the case in which the node being deleted is a root node.





**Code:** void DeleteNodeSub\_Tree(tree \* node, int value)

{

int flag = Search(root, value); //searching the node

if (flag == 1)

{

if (ploc->left != l && ploc->right == r)//if left node is not null and right is null

{

if (loc->right == r && loc->left != l) //right of loc is null and left of loc is not null

ploc->left = loc->left; //loc is deleted, ploc left has data of loc left

else

ploc->left = loc->right;//loc is deleted, ploc right has data of loc right

}

else

{

if (loc->right == r && loc->left != l)//right of loc is not null and left of loc is null

ploc->right = loc->left;//loc is deleted, ploc left has data of loc left

else

ploc->right = loc->right;//loc is deleted, ploc right has data of loc right

}

//delete loc;

}

}

**ScreenShot:**

Text

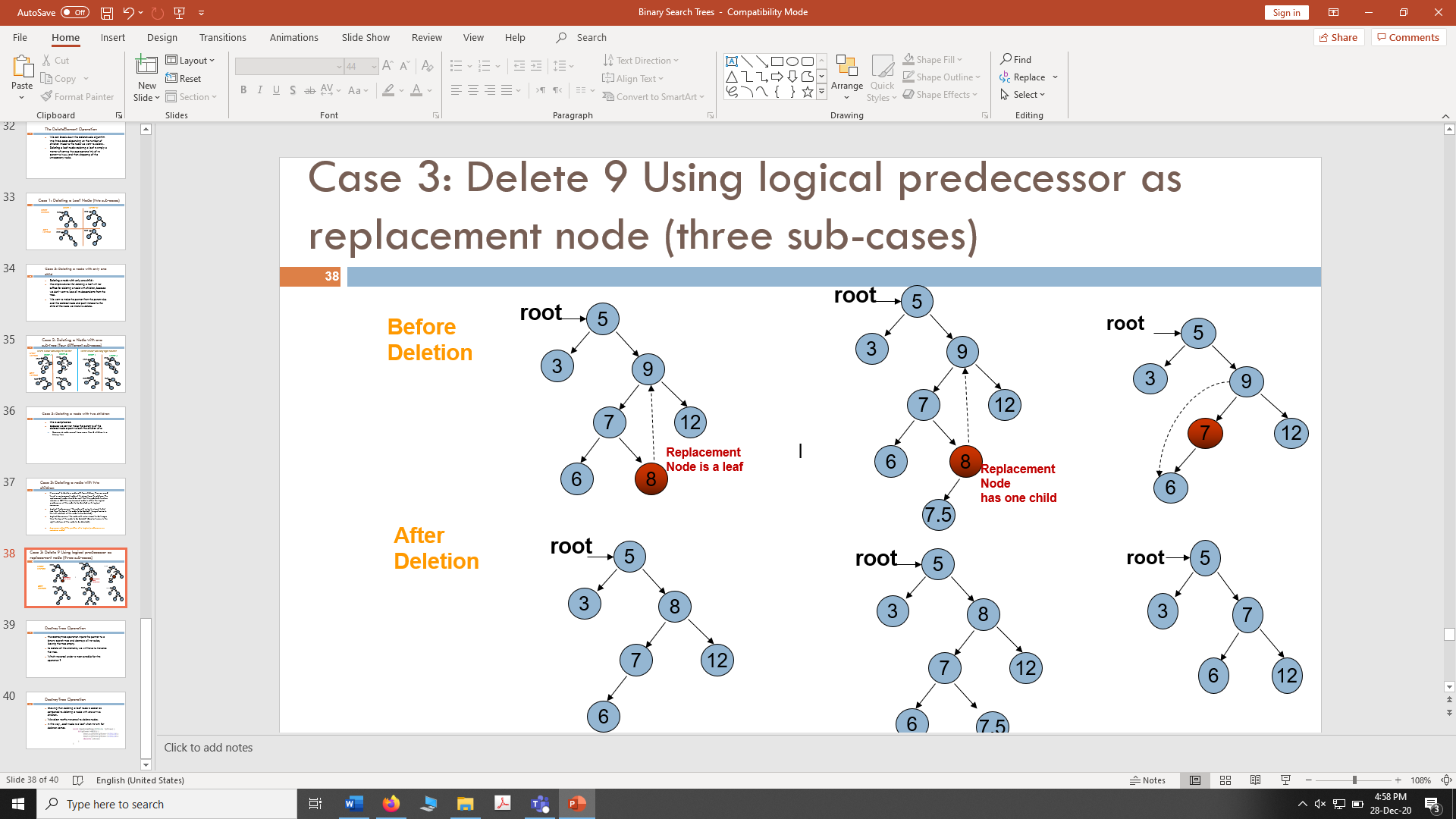
Description automatically generated

**c).** **Deleting a node with two children:** If we want to delete a node with two children, then we need to put a replacement node at its place from its sub-tree. The replacement node should be such that the updated structure remains a BST. The replacement node is either the logical predecessor of the node to be deleted or its logical successor.

**Logical Predecessor:** ”*the node with value to closest to but less than the key of the node to be deleted*” (largest value in the left sub-tree of the node to be deleted).

**Logical Successor:** ”*the node with value closest to but larger than the key of the node to be deleted*” (Smallest value in the right sub-tree of the node to be deleted).

In the below example, the node containing value 9 is deleted using its logical predecessor as replacement node. Include the case in which the node being deleted is a root node.



**Code:** void DeleteNodeWithTwoNodes(tree\* node, int value)

{

int flag = Search(root, value);

if (flag == 1)

{

if (loc->left != l && loc->right != r) //if left node is not null and right is not null

{

RightNode(loc->left);

if (ploc->right == loc && rloc->left == l)//right loc's left is null

{

ploc->right = rloc;

prloc->right = r;

delete loc;

}

else if (ploc->right == loc && rloc->left != l)//right loc's left is not null

{

ploc->right = rloc;

prloc->right = rloc->left;

delete loc;

}

else if (ploc->right == loc && loc->left->right == r)//current loc->left->right is null

{

ploc->right = loc->left;

loc->left->right = loc->right;

delete loc;

}

}

}

}

**ScreenShot:**

Text

Description automatically generated

1. **Destroy tree:** Implement a function that destroys all nodes of a binary tree leaving the tree empty.

void DeleteTree(tree\* node)

{

if (!(node == l || node == r))//checking if node is node of not null

{

DeleteTree(node->left); //treversing through the tree

DeleteTree(node->right);//treversing through the tree

delete node;// deleting node one by one

length--; // as node get deleted decrement length

}

}

Screenshot:

Text

Description automatically generated

1. Implement a function that traverses a BST and **prints** only its **leaf nodes.**

void PrintLeafs(tree\* node)

{

if (node->left == l && node->right == r) //the node which has no leaves, then output node

cout << " Node: " << node->data << endl;

if (!(node == l || node == r))

{

PrintLeafs(node->left); //treversing through the tree

PrintLeafs(node->right);//treversing through the tree

}

}

ScreenShot:

Text

Description automatically generated

1. Implement a function that traverses all nodes of a BST once, and counts the number of leaves, nodes with only left child, nodes with only right child and nodes with two children in it. You should maintain a separate counter variable for each of the four types of nodes.

void Child\_Counter(tree\* node)

{

if ((node->left == l && node->right == r))//the node that has no leaves, then increment leaf counter

leaf++;

if (!(node == l || node == r))

{

if (node->left != l && node->right != r)//if node has no leaves

lrchild\_counter++; //;lrchild counter is incremented

else if (node->left != l && node->right == r)//if node has only right leaf null

lchild\_counter++;

else if (node->left == l && node->right != r)//if node has only left leaf null

rchild\_counter++;

Child\_Counter(node->left);

Child\_Counter(node->right);

}

}

ScreenShot:

Text

Description automatically generated

1. Implement a function that **deletes** all **leaf** nodes of a **given BST only.**

void DeleteLeaf(tree\* node)//function deletes the leafs

{

if (node->left == l && node->right == r)//deletes nodes that have left and right node as null

{

if (node->data < ploc->data)

ploc->left = l;

else

ploc->right = r;

//delete node;

length--;

}

if (!(node == l || node == r))

{

ploc = node;

DeleteLeaf(node->left);

DeleteLeaf(node->right);

}

}

Main()

tre->Insert(5);

tre->Insert(3);

tre->Insert(7);

tre->Insert(6);

tre->Insert(8);

tre->Insert(12);

cout << " Leaf Nodes: " << endl;

tre->PrintLeafs(tre->root);

cout << " Updated Tree: " << endl;

tre->DeleteLeaf(tre->root);

tre->PreOrder(tre->root); cout << endl;

ScreenShot:

Text

Description automatically generated

1. Implement a function that **deletes** only those nodes from a **given** **BST** that have only **left sub-tree.** Only those nodes should be deleted that have left branch in the original tree.

Code:

void DeleteLeftTree(tree\* node)

{

if (node->left != l) //if the left tree contains any nodes

{

root->left = l; // left node should be made null

}

}

Main Function

tre->DeleteLeftTree(tre->root);

tre->PreOrder(tre->root);

Screenshot:

Text

Description automatically generated