

**Codes for Insert + Search:**

#pragma once

#include "BinaryTreeNode.h"

#include "binaryTreeType.h"

template <class T>

class binarySearchTree:public binaryTreeType<T>

{

public:

void insert(T item)

{

BinaryTreeNode<T> \*current, \*trailCurrent, \*newNode;

newNode = new BinaryTreeNode<T>;

newNode->info = item;

newNode->llink = NULL;

newNode->rlink = NULL;

if (root == NULL)

root = newNode;

else

{

current = root;

while (current != NULL)

{

trailCurrent = current;

if (current->info == item)

{

cout << "Item already in list" << endl;

return;

}

else if (current->info > item)

current = current->llink;

else

current = current->rlink;

}

if (trailCurrent->info > item)

trailCurrent->llink = newNode;

else

trailCurrent->rlink = newNode;

}

}

bool search(T item)

{

BinaryTreeNode<T>\*p = root;

bool found = false;

BinaryTreeNode<T> \*current;

if (p == NULL)

cout << "Tree is empty" << endl;

else

{

current = p;

while (current != NULL && !found)

{

if (current->info == item)

found = true;

else if (current->info > item)

current = current->llink;

else

current = current->rlink;

}

}

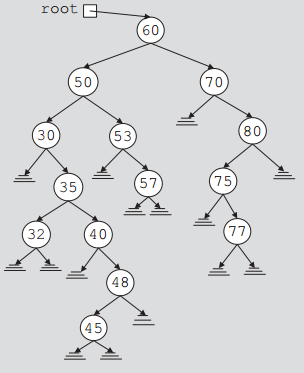
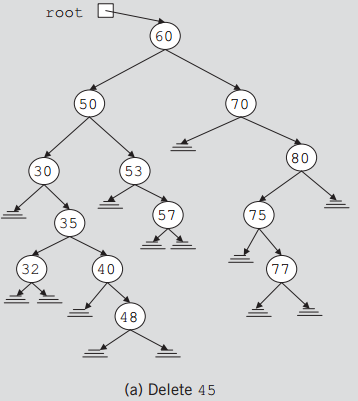
return found;

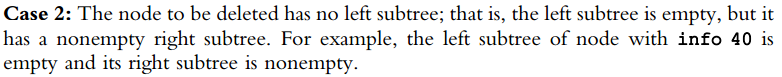
}

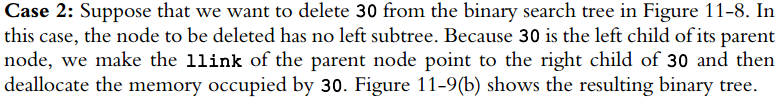
};

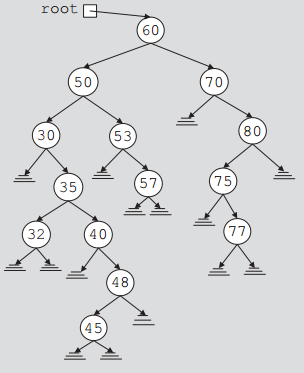
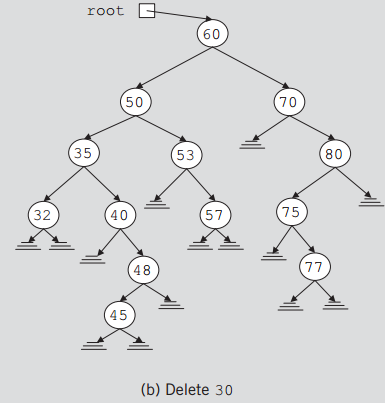
**Delete:**

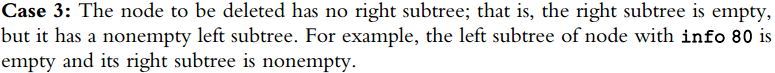


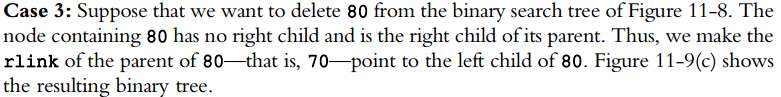
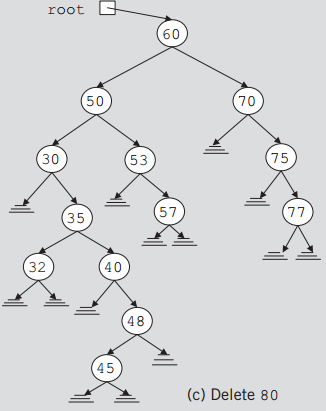


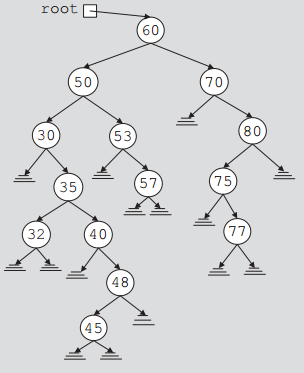




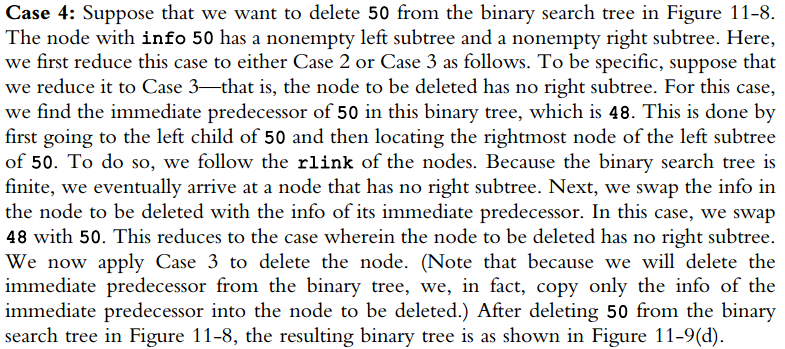
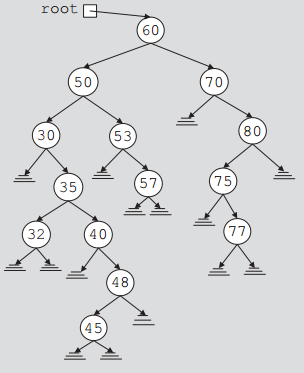
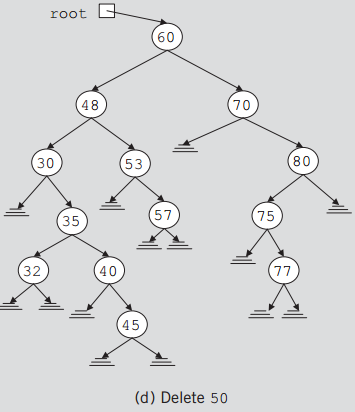












void deleteFromTree(BinaryTreeNode<T> \*&p)

{

BinaryTreeNode<T> \*current, \*trailCurrent, \*temp;

if (p == NULL)

cout << "Node to be deleted is NULL" << endl;

else if (p->llink == NULL && p->rlink == NULL)

{

temp = p;

p = NULL;

delete temp;

}

else if (p->llink == NULL)

{

temp = p;

p = temp->rlink;

delete temp;

}

else if (p->rlink == NULL)

{

temp = p;

p = temp->rlink;

delete temp;

}

else

{

current = p->llink;

trailCurrent = NULL;

while (current->rlink != NULL)

{

trailCurrent = current;

current = current->rlink;

}

p->info = current->info;

if (trailCurrent == NULL)

p->llink = current->llink;

else

trailCurrent->rlink = current->llink;

delete current;

}

}

void deleteNode(const T &deleteItem)

{

BinaryTreeNode<T>\*current;//pointer to traverse the tree

BinaryTreeNode<T>\*trailCurrent;//pointer behind current

bool found = false;

if (root == NULL)

cout << "Cannot delete from the empty tree." << endl;

else

{

current = root;

trailCurrent = root;

while (current != NULL && !found)

{

if (current->info == deleteItem)

found = true;

else

{

trailCurrent = current;

if (current->info>deleteItem)

current = current->llink;

else

current = current->rlink;

}

}//endwhile

if (current == NULL)

cout << "The delete item is not in the tree." << endl;

else if(found)

{

if (current == root)

deleteFromTree(root);

else if(trailCurrent->info>deleteItem)

deleteFromTree(trailCurrent->llink);

else

deleteFromTree(trailCurrent->rlink);

}//endif

}

}

**PRATICE QUESTIONS**

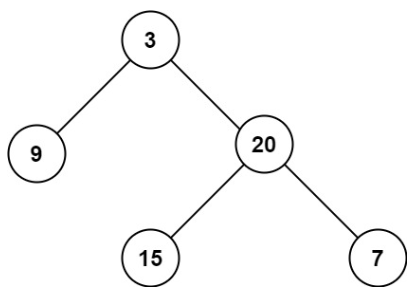
**QNO1:** Given the root of a binary tree, return its maximum depth.

A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Example:

Input: [3,9,20,null,null,15,7]

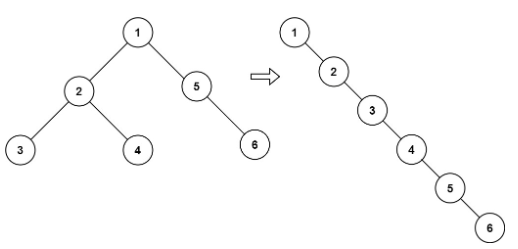
Output: 3

****

**QNO2:** Given the root of a binary tree, flatten the tree into a "linked list":

* The "linked list" should use the same TreeNode class where the right child pointer points to the next node in the list and the left child pointer is always null.
* The "linked list" should be in the same order as a [pre-order traversal](https://en.wikipedia.org/wiki/Tree_traversal#Pre-order,_NLR) of the binary tree.

**Example:**

****

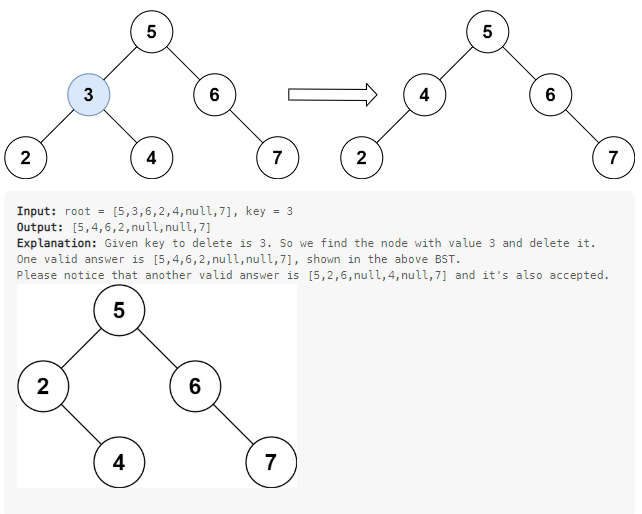
**Input: root:** [1,2,5,3,4,null,6]

**Output:** [1,null,2,null,3,null,4,null,5,null,6]

**QNO3:** Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return the root node reference (possibly updated) of the BST.

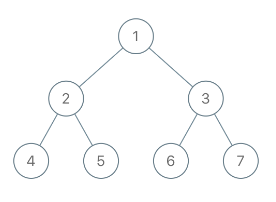
* Basically, the deletion can be divided into two stages:
* Search for a node to remove.
* If the node is found, delete the node.

Example:



**QNO4:** Given the root of a binary tree, each node in the tree has a distinct value.

* After deleting all nodes with a value in to\_delete, we are left with a forest (a disjoint union of trees).
* Return the roots of the trees in the remaining forest. You may return the result in any order.



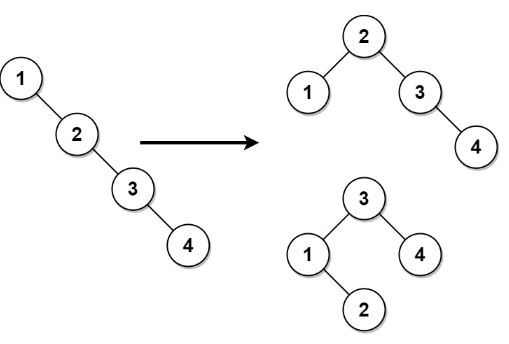
**Input:** [1,2,3,4,5,6,7], **to\_delete** = [3,5]

**Output:** [[1,2,null,4],[6],[7]]

**QNO5:** Given the root of a binary search tree, return a balanced binary search tree with the same node values. If there is more than one answer, return any of them.

A binary search tree is balanced if the depth of the two subtrees of every node never differs by more than 1.

Example:

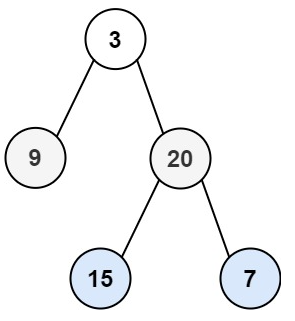


**Input:**  [1,null,2,null,3,null,4,null,null]

**Output:** [2,1,3,null,null,null,4]

**Note:** This is not the only correct answer, [3,1,4,null,2] is also correct.

**QNO6:** Given the root of a binary tree, return the level order traversal of its nodes' values. (i.e., from left to right, level by level).

****

**Input**: root = [3,9,20,null,null,15,7]

**Output**: [[3],[9,20],[15,7]]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*