```
//=======BinaryTree.h ===============
// Binary Tree Template
#ifndef BINARYTREE_H
#define BINARYTREE_H
#include <iostream>
using namespace std;

template <class T>
class BinaryTree
{
public:
  struct TreeNode   {
    T value;
    TreeNode *left;
    TreeNode *right;
  };
  TreeNode *root;
  void insert(TreeNode *&, TreeNode *&);
  void destroySubTree(TreeNode *);
  void deleteNode(T, TreeNode *&);
  void makeDeletion(TreeNode *&);
  void displayInOrder(TreeNode *);
  void displayPreOrder(TreeNode *);
  void displayPostOrder(TreeNode *);
public:
  BinaryTree()   { root = nullptr; }   // Constructor
  ~BinaryTree() { destroySubTree(root); }  // Destructor
  void insertNode(T);
  bool searchNode(T);
  void remove(T);
  void displayInOrder()      { displayInOrder(root); }
  void displayPreOrder()    { displayPreOrder(root); }
  void displayPostOrder()  { displayPostOrder(root); }
};

//************************************************
// insert accepts a TreeNode pointer and a pointer to a node. *
// The function inserts the node into the tree pointed to by  *
// the TreeNode pointer. This function is called recursively. *
//************************************************
template <class T>
void BinaryTree<T>::insert(TreeNode *&nodePtr, TreeNode
*&newNode)
{
     if (nodePtr == nullptr) {          // Insert the node.
        nodePtr = newNode; }
     else if (newNode->value < nodePtr->value) {
        insert(nodePtr->left, newNode); }// Search the left branch
     else {
        insert(nodePtr->right, newNode);// Search the right branch
     }
}
//************************************************
// insertNode creates a new node to hold num as its value, *
// and passes it to the insert function.                   *
//************************************************
template <class T>
void BinaryTree<T>::insertNode(T num)
{
  TreeNode *newNode = nullptr;    // Pointer to a new node.
  // Create a new node and store num in it.
  newNode = new TreeNode;
  newNode->value = num;
  newNode->left = newNode->right = nullptr;
// Insert the node.
  insert(root, newNode);
}

//************************************************
// destroySubTree is called by the destructor. It   *
// deletes all nodes in the tree.                   *
//************************************************
template <class T>
void BinaryTree<T>::destroySubTree(TreeNode *nodePtr)
{
     if (nodePtr->left)
     {    destroySubTree(nodePtr->left);      }
     if (nodePtr->right)
     {    destroySubTree(nodePtr->right);    }
  delete nodePtr;
}

//************************************************
// searchNode determines if a value is present in   *
// the tree. If so, the function returns true.      *
// Otherwise, it returns false.                     *
//************************************************
template <class T>
bool BinaryTree<T>::searchNode(T num)
{
     bool status = false;
     TreeNode *nodePtr = root;

     while (nodePtr)   {
          if (nodePtr->value == num) {
               status = true;             }
          else if (num < nodePtr->value) {
               nodePtr = nodePtr->left;          }
          else{
               nodePtr = nodePtr->right;        }
     }
     return status;
}

//************************************************
// remove calls deleteNode to delete the       *
// node whose value member is the same as num. *
//************************************************
template <class T>
void BinaryTree<T>::remove(T num)
{   deleteNode(num, root); }
//************************************************
// deleteNode deletes the node whose value   *
// member is the same as num.                *
//************************************************
template <class T>
void BinaryTree<T>::deleteNode(T num, TreeNode *&nodePtr)
{
     if (num < nodePtr->value)
     {    deleteNode(num, nodePtr->left);    }
     else if (num > nodePtr->value)
     {    deleteNode(num, nodePtr->right);   }
```

```
      else
      {     makeDeletion(nodePtr);    }
}

//************************************************
// makeDeletion takes a reference to a pointer to the node  *
// that is to be deleted. The node is removed and the       *
// branches of the tree below the node are reattached.      *
//************************************************
template <class T>
void BinaryTree<T>::makeDeletion(TreeNode *&nodePtr)
{
      // Temporary pointer, used in reattaching the left subtree.
   TreeNode *tempNodePtr = nullptr;

   if (nodePtr == nullptr)
   {     cout << "Cannot delete empty node.\n";   }
   else if (nodePtr->right == nullptr)   {
     tempNodePtr = nodePtr;
     nodePtr = nodePtr->left; // Reattach the left child
     delete tempNodePtr;   }
   else if (nodePtr->left == nullptr)   {
     tempNodePtr = nodePtr;
     nodePtr = nodePtr->right; // Reattach the right child
     delete tempNodePtr;
   }
   // If the node has two children.
   else
   {
     // Move one node the right.
     tempNodePtr = nodePtr->right;
     // Go to the end left node.
        while (tempNodePtr->left)  {
              tempNodePtr = tempNodePtr->left;
        }
     // Reattach the left subtree.
     tempNodePtr->left = nodePtr->left;
     tempNodePtr = nodePtr;
     // Reattach the right subtree.
     nodePtr = nodePtr->right;
     delete tempNodePtr;
   }
}

//************************************************
// The displayInOrder member function displays the values   *
// in the subtree pointed to by nodePtr, via inorder traversal.  *
//************************************************
template <class T>
void BinaryTree<T>::displayInOrder(TreeNode *nodePtr)
{
   if (nodePtr)
   {
     displayInOrder(nodePtr->left);
     cout << nodePtr->value << endl;
     displayInOrder(nodePtr->right);
   }
}

//************************************************
// The displayPreOrder member function displays the values  *
// in the subtree pointed to by nodePtr, via preorder traversal. *
//************************************************
template <class T>
void BinaryTree<T>::displayPreOrder(TreeNode *nodePtr)
{
   if (nodePtr)
   {
     cout << nodePtr->value << endl;
     displayPreOrder(nodePtr->left);
     displayPreOrder(nodePtr->right);
   }
}

//************************************************
// The displayPostOrder member function displays the values *
// in the subtree pointed to by nodePtr, via postorder traversal.*
//************************************************
template <class T>
void BinaryTree<T>::displayPostOrder(TreeNode *nodePtr)
{
   if (nodePtr)
   {
     displayPostOrder(nodePtr->left);
     displayPostOrder(nodePtr->right);
     cout << nodePtr->value << endl;
   }
}

#endif
//===================================
// Chapter 20, Programming Challenge 1: Binary Tree Template
// spc 20-1
#include <iostream>
#include "BinaryTree.h"
using namespace std;

int main(){
   // Create an instance of BinaryTree
   BinaryTree<int> tree;
   // Insert some values into the tree.
   cout << "Inserting nodes.\n";
   tree.insertNode(5);
   tree.insertNode(8);
   tree.insertNode(3);
   tree.insertNode(12);
   tree.insertNode(9);
   // Display the nodes.
   cout << "Here are the values in the tree:\n";
   tree.displayInOrder();
   cout << endl;
   cout << "Deleting 8...\n";   // Delete the 8 node.
   tree.remove(8);
   cout << "Deleting 12...\n";   // Delete the 12 node.
   tree.remove(12);
   // Display the nodes again.
   cout << "Now, here are the nodes:\n";
   tree.displayInOrder();
   cout << endl;
   return 0;
}
```

InClassActivity#4 **Binary Tree** Name:_____Row/Seat_____
10/23/19

a) Trace the program spc 20-1 and show the output of the program.
b) Modify the spc 20-1 code with postorder and then display the output