Lab 13
Implementation and Applications of Binary Trees, Binary Search Trees

Section 1: Guess program outputs.

1.
```
#include <iostream>
#include <string>

using namespace std;

class IntBinaryTree
{
private:
   struct TreeNode
   {
      int value;
      TreeNode* left;
      TreeNode* right;
      TreeNode(int value1,
               TreeNode* left1 = nullptr,
               TreeNode* right1 = nullptr)
      {
         value = value1;
         left = left1;
         right = right1;
      }
   };

   TreeNode* root;

   // helper member functions of private interface
   void insert(TreeNode*& tree, int num);
   void destroySubtree(TreeNode* tree);
   void remove(TreeNode*& tree, int num);
   void makeDeletion(TreeNode*& tree);

   void displayInOrder(TreeNode* tree) const;
   void displayPreOrder(TreeNode* tree) const;
   void displayPostOrder(TreeNode* tree) const;

public:
   // these member functions are the public interface
   IntBinaryTree()
   {
      root = nullptr;
   }
   ~IntBinaryTree()
   {
      destroySubtree(root);
   }

   void insert(int num)
   {
      insert(root, num);
   }
```

```
    bool search(int num) const;

    void remove(int num)
    {
       remove(root, num);
    }

    void showInOrder() const
    {
       displayInOrder(root);
    }
    void showPreOrder() const
    {
       displayPreOrder(root);
    }
    void showPostOrder() const
    {
       displayPostOrder(root);
    }
};


void IntBinaryTree::insert(TreeNode*& tree, int num)
{
    // If the tree is empty, make a new node and
    // make it the root of the tree.
    if (!tree)
    {
       tree = new TreeNode(num);
       return;
    }

    // If num is already in tree: return.
    if (tree->value == num)
       return;

    // The tree is not empty:
    // recursive insert() the new node into the
    // left or right subtree.
    if (num < tree->value)
       insert(tree->left, num);
    else
       insert(tree->right, num);
}

void IntBinaryTree::destroySubtree(TreeNode* tree)
{
    if (!tree) return;

    // recursive destroySubtree()
    destroySubtree(tree->left);
    destroySubtree(tree->right);

    // Delete the node at the root.
    delete tree;
}
```

```
bool IntBinaryTree::search(int num) const
{
   TreeNode* tree = root;

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

   return false;
}

void IntBinaryTree::remove(TreeNode*& tree, int num)
{
   if (tree == nullptr) return;


   if (num < tree->value)
      remove(tree->left, num);
   else if (num > tree->value)
      remove(tree->right, num);
   else
      // We have found the node to delete.
      makeDeletion(tree);
}


void IntBinaryTree::makeDeletion(TreeNode*& tree)
{
   // Used to hold node that will be deleted.
   TreeNode* nodeToDelete = tree;

   // Used to locate the  point where the
   // left subtree is attached.
   TreeNode* attachPoint;

   if (tree->right == nullptr)
   {

      tree = tree->left;
   }
   else if (tree->left == nullptr)
   {

      tree = tree->right;
   }
   else
   {
      // Move to right subtree.
      attachPoint = tree->right;

      // Locate the smallest node in the right subtree
      // by moving as far to the left as possible.
      while (attachPoint->left != nullptr)
         attachPoint = attachPoint->left;
```

```
      // Attach the left subtree of the original tree
      // as the left subtree of the smallest node
      // in the right subtree.
      attachPoint->left = tree->left;


      tree = tree->right;
   }

   // Delete the tree node
   delete nodeToDelete;
}

void IntBinaryTree::displayInOrder(TreeNode* tree) const
{
   if (tree)
   {
      displayInOrder(tree->left);
      cout << tree->value << "  ";
      displayInOrder(tree->right);
   }
}

void IntBinaryTree::displayPreOrder(TreeNode* tree) const
{
   if (tree)
   {
      cout << tree->value << "  ";
      displayPreOrder(tree->left);
      displayPreOrder(tree->right);
   }
}

void IntBinaryTree::displayPostOrder(TreeNode* tree) const
{
   if (tree)
   {
      displayPostOrder(tree->left);
      displayPostOrder(tree->right);
      cout << tree->value << "  ";
   }
}

int main()
{
    IntBinaryTree tree;

    cout << "Inserting the numbers 5 8 3 12 9\n";
    tree.insert(5);
    tree.insert(8);
    tree.insert(3);
    tree.insert(12);
    tree.insert(9);

    if (tree.search(3))
      cout << "3 is found in the tree\n";
    else
      cout << "3 was not found in the tree\n";

    cout << "Inorder traversal:  ";
    tree.showInOrder();
```

```
    cout << endl;

    cout << "Preorder traversal:  ";
    tree.showPreOrder();
    cout << endl;

    cout << "Postorder traversal:  ";
    tree.showPostOrder();
    cout << endl;

    cout << "Deleting 8\n";
    tree.remove(8);
    cout << "Deleting 12\n";
    tree.remove(12);

    cout << "Inorder traversal again:  ";
    tree.showInOrder();
    cout << endl;

    return 0;
}
```

Section 2: Review Questions and Exercises

1. In what ways is a binary tree similar to a linked list?

2. A ternary tree is like a binary tree, except each node in a ternary tree may have three children: a left child, a middle child, and a right child. Write an analog of the TreeNode declaration that can be used to represent the nodes of a ternary tree.

3. Imagine a tree in which each node can have up to a hundred children. Write an analog of the TreeNode declaration that can be used to represent the nodes of such a tree. A declaration such as

```
struct TreeNode
{
  int value;
  TreeNode* child1;
  TreeNode* child2;
  TreeNode* child3;
  . . .
  . . .
  . . .
};
```

that simply lists all the pointers to the hundred children is not acceptable.

Section 3: Programming Challenges

1. Simple Binary Search Tree Class
Write a class for implementing a simple Binary Search Tree (BST) capable of storing numbers.
The class should have member functions
void insert(double num)
bool search(double num) const
void inorder(vector<double>& v)
The inorder function is passed an initially empty vector v: it fills v with the inorder list of numbers stored in the binary search tree.
Demonstrate the operation of the class using a suitable driver program.

2. Tree Size
Modify the binary search tree created in the previous programming challenge to add a member function
int size()
that returns the number of items (nodes) stored in the tree.
Demonstrate the correctness of the new member function with a suitable driver program.