# BINARY SEARCH TREES AND THEIR VARIETY OF ORDERINGS

Fordham University

Department of Computer and Information Science

CISC 2200: Data Structures

Instructor: Prof. H. M. Ammari

Student: Destin Piagentini

Fall 2018



Table of Contents

# Introduction

This program allows the users to create and edit a binary search tree data structure. In a binary search tree, all nodes have two other pointers that point left and right respectively. On the left-hand side, nodes with values less than the node in question are stored. On the right, nodes with values greater than the node in question are stored. This data structure allows searching algorithms to reduce their search area by half each pass through, for a time complexity of O(log n).

The program also allows users to print in any of three orderings: Preorder, Inorder, and Postorder. In “preorder,” each node is examined. First, the node itself is printed, then the function is recursively called to explore that node’s left pointer (left branch), and finally the function is recursively called to explore that node’s right pointer (right branch). In “inorder,” the node’s left pointer (left branch) is first recursively explored with the same function, then the node itself is printed, then the node’s right pointer (right branch) is recursively explored with the same function. In the end, the outcome should be a sorted list of all elements in the binary search tree. In “postorder,” the element’s left pointer (left branch) is recursively explored with the same function, then the node’s right pointer (right branch) is recursively explored with the same function, and finally the node itself is printed. The user may print any of these orders at any time from the main menu.

# Design and Solution

### Approach and Evolution of the Design

When first approaching this program, the implementation of a binary search tree seemed like a large problem. However, by breaking the task down into smaller problems, I was able to arrive at the solution. First I outlined the BinarySearchTree class and the node struct. For the node struct, all was needed was a “Left” pointer, a “Right” pointer, and a data member to hold a value. I knew I needed a constructor and a destructor, a function for searching through the tree (searching for a value), functions to add and delete a node, functions for printing the three different orders, functions to check if the tree is empty or if the tree is full, and, finally a function to deallocate the entire tree (called by the destructor). I also knew I needed a pointer to point to the root, and a data member to keep track of the number of elements in the search tree (along with it’s accessor).

Of the functions I listed out, I first defined the constructor (setting the Root pointer to null, and setting the NumberOfElements data member to 0). Next, for the destructor, I simply made it call MakeEmpty. After that, I quickly defined MakeEmpty to delete the nodes in postorder (the most efficient order for this task).. Finally, I created the NumberOfElements accessor, the IsFull function, and the IsEmpty function since these functions are all relatively easy implementations. I also defined the three “order” functions. I did, however, notice that these functions would be impossible to call from the main function as the main function does not have access to the Root pointer, which should be the start of the recursive calls. I then created three separate public functions that would simply call the true function starting with Root. These new

functions could be safely called form main, and I moved the functions that could not be called from main under the private of the class.

Next, I defined the function that would search for a node based on the value given. I quickly realized I encountered the same problem as with the ordering functions in the sense that I could not start the search from Root in main. Like the ordering functions, I then created a new function that could be called from main that would simply call the true function with Root as the recursive onset. In this function, the algorithm is simple. If we find the value, or if we reach where we expect it to be but we find null, we return that pointer location. We can check if the value is in the tree or not from main by checking whether the function returns null or not. If not any of those cases, however, we recursively decide whether to call again in the left or right branch based on the comparison to the current node.

With this searching function done, I was able to move on to the AddNode function. The AddNode function would search for the spot where the value should be, and create a new node in that position, and load that new node with the new value. After the AddNode function was working, I implemented the DeleteNode function. The DeleteNode function worked a bit different as it accepted the node to be deleted as a variable. The delete function then determined whether the node had two children, one child, or no children. In each case, the Delete function would appropriately handle itself. If two children, the delete function would find the right-most node of the left branch, overwrite the value needed to be deleted with the right-most ‘s value, then call itself to delete the right-most node (since that node must either contain one child or no children). If one child, a temp variable would delete the node after the node pointer reorients itself to the child after it. If no children, the node is simply deleted and the pointer is set to null.

The next step for me was to implement the main function. This main function had a menu system, where one could edit the tree or print any order. This was accomplished via use of a switch statement. When editing the tree, the main function will ensure that the element does not already exist in the tree, or, if deleting, that the element does exist in the tree. This is accomplished by the FindNodeByValue function, which will return the location where the value is or is expected to be. If the function returns null, it is an indicator that the element is not in the tree. If all conditions are met, the addition/deletion of eleemnts in the tree takes place.

While bug testing after setting up main, however, I encountered that my delete function did not work as expected, but my AddNode function did. I noticed that for the AddNode, I passed in the reference to the pointer, while in the DeleteNode function I did not. After changing the DeleteNode function to accept a pointer by reference (so that we truly get the pointer we are editing, and not a copy of it), the function still was not working. I noticed that the function was unable to realign itself to maintain tree structure as the program kept crashing. I then noticed that my searching function also needed to return a reference, so that it returns the actual pointer to the node so that we can realign the tree. Still, the program did not work. I noticed that I first created a temporary node object, then set it to the result of the searching function, then plugged the temporary variable in to the delete function. By using a temp variable, I was still using a copy. Instead, I called the searching function directly into the parameters when calling the DeleteNode function, ensuring the DeleteNode function is called with the actual pointer, so that it may be correctly adjusted in the binary tree.

After I made this realization, I edited the AddNode function to read in the spot where the node is to be added by calling the searching function in its parameter list instead of searching for the spot in the AddNode function itself. Since the searching function will return the pointer even

if its null, we are being returned the branch where the node is to be added. With this, I was able to simplify the AddNode function greatly, and reduce it to a time complexity of O(1). After this, I made adjustments to the main system to ensure the menu printed correctly and to add a brief introduction to the user.

## Brief Summary of Each Function

BinarySearchTree()

This is the constructor for the BinarySearchTree class. This constructor is called when an object for the BinarySearchTree class is created in main. This constructor simply sets the "Root" pointer to null, and initializes the NumberOfElements data member to 0. This function runs at a time complexity of O(1).

~BinarySearchTree()

This is the destructor for the BinarySearchTree class. The precondition for this function is that the tree object exists. The postcondition is that all memory will be deallocated in the binary search tree. This function is called when the object goes out of scope. This function simply calls the MakeEmpty() function starting at "Root". This function runs at a time complexity of O(1).

BinaryNode *& FindNodeByValue(int SearchingValue)

This function serves as a way for main to invoke the "FindNode" function with the value that is being searched for. The precondition of this function is that the tree object exists. The postcondition is that the function will return the results of the FindNode function, with the binary search tree unchanged. The function simply calls the FindNode function starting at "Root", since main can not use Root. This function runs at a time complexity of O(1).

BinaryNode *& BinarySearchTree::FindNode(int SearchingValue, BinaryNode *& SearchingNode)

This function is a recursive function that searches for a given value, halving the search area through each iteration (because of the nature of a binary search tree). The preconsition is that SearchingNode is an ancestor of the node containg the value to be searched, or is an ancestor of the spot where the node would be if it existed. The tree object must also be initialized. The post condition is that either the spot where the node with the given value should exist or does exist is returned, or the function recursively calls itself searching either the left branch or the right branch if it has not yet reached a conclusion. The function itself runs at a time complexity of O(1), with the number of recursive calls being O(log n) due to the nature of a binary search tree.

void AddNode(int ValueToAdd, BinaryNode* & NodeToAdd)

The Addnode function creates a node with the given value at the spot dictated by NodeToAdd. The precondition is that the tree object is initialized and the value to add does not already exist in the tree. The postcondition is that a new node will be created at the spot where it belongs in the binary search tree structure, with the value loaded in, and the Left/Right pointers initialized to null. This function simply dynamically creates a new node, loads it with the given value, and sets the node's two pointers to null. The function will also update the NumberOfElements variable. This function runs at a time complexity of O(1).

void DeleteNode(BinaryNode *& NodeToDelete)

The DeleteNode function will delete the node at the pointer of NodeToDelete, and ensure that all adjustments are made so that the binary search tree structure is maintained. The precondition of this function is that tree object is initialized, and that the value being deleted

exists in the array. The postcondition is that the value is deleted, one less node is in the array, and the binary search tree structure is maintained. This function will determine whether the node to delete has two children, one child, or no children. If no children, the node will simply be deleted and the pointer will be set to null. NumberOfEements will also be decremented. If one child, a temp pointer will point to the node being deleted, while the actual pointer will point to the child to "skip" over the node to be deleted. Then, the temp pointer deletes the original node. NumberOfElements will also be decremented.

If there is two children, a temporary node variable will find the right-most node of the left branch. A temporary variable will contain the value of the right-most node. We can not overwrite the original node yet, since that would result in the same value existing in two different nodes in the array. Since we will call the Delete function again based on the right-most value, we must wait to overwrite the original node until after the function is called again. Directly afterwards, the function will call itself to delete the right-most node. Since it is the right-most node, it can only have one child to the left or no children, so the function calling itself runs no risk of an infinite loop (since the function only calls itself in the case of two children). Finally, the temporary variable holding the right-most node's value will be used to overwrite the node containing the value to be deleted. This function runs at a worse case scenario of O(n), where n is the number of elements that exist to the right of the top of the left branch of the NodeToDelete when we need to find the right-most node in the case of the NodeToDelete having two children.

void Preorder() const, void Inorder() const, void Postorder() const

These functions are needed for main to invoke the start of an ordering print. However, since main does not have access to the Root variable to start the recursive functions, these functions will do it for main. The precondition for each function is that a tree object exists. The

postcondition for each function is that the binary search tree will be printed in the respective order. Each function simply calls their respective order's recursive function, starting with the "Root" pointer. The time complexity for each of these functions is O(1).

void PrintPreorder(BinaryNode * Current) const, void PrintInorder(BinaryNode * Current) const, void PrintPostorder(BinaryNode * Current) const

These functions recursively print the binary search tree in their respective orders. The precondition for each function is that the "Current" pointer is a node in the tree (or a nodes pointer that equals null), and that a tree object is initialized. The postcondition is that the function will recursively print the correct order based on which order it is if the "Current" pointer is not null. For preorder, this means "Current" is printed first, then we recursively call the function with the node to "Current's" left, and the node to "Current's" right.. For inorder, this means we first recursively call the function with the node to "Current's" left, then we print "Current", then we recursively call the function with the node to "Current's" right. For postorder, this means we first recursively call the function with the node to "Current's" left, then we recursively call the function with the node to "Current's" right, and then we print Current last. Note that in all functions, this order only operates if "Current" is not null. The Time complexity of each function is O(1), with the overall time complexity of the recursive calling being O(n) where n is the number of elements in the binary search tree.

bool IsFull() const

This function checks whether the binary search tree is full or not. The precondition for this function is that the tree object is initialized, and the postcondition is that the function will return true or false depending on whether the tree is full or not, but the tree will remain unchanged. The function tries to dynamically allocate a new binary node. If there is an error, the function catches it, and returns true, indicating that the binary search tree is full. If there is no

error, the test node is deleted and the function returns false, indicating that there is still room to add more nodes. This function runs at a time complexity of O(1).

bool IsEmpty() const

This function checks whether the binary search tree is empty or not. The precondition for this function is that the tree object is initialized, and the postcondition is that the function will return true or false depending on whether the tree is empty or not, but the tree will remain unchanged. The function will return whether the Root pointer is equal to null or not. If the Root pointer is null, there are no nodes in the tree, and the tree is empty. This function runs at a time complexity of O(1).

int GetNumberOfElements() const

This function returns the NumberOfElements data member for use in the main function. The precondition is that a tree object is initialized, and the postcondition is that NumberOfElements is returned, and the tree is unchanged. This function runs at a time complexity of O(1).

void MakeEmpty(BinaryNode *& Current)

This function will recursively delete nodes in postorder until all nodes are deleted. Postorder is used as it is the most efficient way to delete nodes. The precondition for this function is that the tree object is initialized, and “Current” is an existing node in the tree (or one of the pointers of a node in the tree that is null). The postcondition is that all nodes in the tree will be deallocated, and the tree will be empty. If the “Current” node is not null, the function will recursively delete the nodes in postorder. The MakeEmpty function will be recursively called first for the node to the left of “Current”, then the MakeEmpty function will be recursively called

for the node to the right of “Current”, then “Current” will be deleted. This function runs at a time complexity of O(1), and the recursive time complexity is O(n) where n is the number of nodes in the binary search tree.

## A Brief Overview of the General Program Loop

The program begins with a brief introduction to the user explaining what a binary search tree data structure, and of the ordering methods. Next the main menu is printed. From the menu, the user can edit the tree, or print in any of the three orders. In addition, the user can quit the program. If the user chooses to edit the function, the edit menu will be brought up where the user can choose to add or delete values from the tree. If the value a user wants to add is already in the tree, the program will print an error message, list the values in the tree, and allow the user to enter in a new number. If the value a user wants to delete is not in the tree, the program will print an error message, list the values in the tree, and allow the user to enter in a new number. If the user exits, the user will be brought back to the main menu. If the user chooses to print any of the orderings, they will be brought to a separate screen where that ordering is printed. If the user chooses to exit from the main menu, they will be asked if they want to start with a new binary tree. If they answer yes, they will be brought to the main menu again with a new, clear tree. If not, the program will exit.

## How I Know the Program is Implemented Correctly

I know my program sorts all nodes correctly based on the orderings. Because of the orderings, I can text expected behavior based on actual behavior. Because all nodes are in their proper locations, each ordering is printed correctly.

# Implementation

Note: some comments or code may be edited from its original format to be more readable in a word processor (namely, Microsoft Word).

**BinarySearchTreeOrdering.cpp**

```
// BinarySearchTreeOrdering.cpp : This file contains the 'main' function. Program
execution begins and ends there.
//

#include <iostream>
#include "BinarySearchTree.h"


int main()
{

	char Input;

	//Intro

	std::cout << "Welcome to Binary Search Tree Creator!\n\n";

	std::cout << "In this program, you will be able to create and edit a binary search
tree, and print its contents in three different 'orders': preorder, inorder, and
postorder.\n\n";

	std::cout << "A binary search tree is a data structure in which each element can
point\nto every element lower than it, and every element higher than it.\n\n";
	std::cout << "As we search for an element down the tree, we can reduce our search
area by half each time by comparing what we arelooking for to a certain point on the
tree.\n\n";
	std::cout << "If we are looking for an item of lesser value, we explore that
point's left branch. If we are looking for an element of a higher value, we explore that
trees right branch.\n\n";

	std::cout << "We can also display the cotents of the tree in different types of
orders:\n\n";
	std::cout << "\tPreorder: As we move along points on the tree, the element being
looked\n\tat is printed first, then we look at the element to its left,then\n\tto its
right. We repeat this process until all elements are printed.\n\n";
	std::cout << "\tInorder: As we move along points on the tree, the element to the
left\n\tof the element in question is looked at first, then the element\n\titself, then
the element to the right. Repeat this process\n\tfor all elements, and the result will be
printed in a sorted order.\n\n";
	std::cout << "\tPostorder: As we move along points on the tree, the element to
the\n\tleft of the element in question is looked at first, then the element\n\tto the
right is looked at, and finally the element itself is printed.\n\tWe repeat this process
until all elements are printed.\n\n";

	std::cout << "Press any key to continue to the main menu: ";
```

```
std::cin >> Input;

do {

	BinarySearchTree Tree;

	system("CLS");

	do {

		std::cout << "Main Menu (Please type a letter based on which queue
		structure you'd like):\n\n\t[1] Edit Tree\n\t[2] Print
		Preorder\n\t[3] Print Inorder\n\t[4] Print Postorder\n\t[5]
		Quit\n\n";
		std::cout << "Enter here: ";
		std::cin >> Input;

		switch (Input) {

			default: {

				system("CLS");
				std::cout << "Invalid Input! Try again!\n\n";
				break;

			}

			case '1': {

				system("CLS");

				do {

					std::cout << "There are " <<
	Tree.GetNumberOfElements() << " element(s) in the binary search tree.\n\n";

					std::cout << "Options:\n\n[A/a] Add
				Number\n[D/d] Delete Number\n[Q/q] Quit to Main menu\n\n";
					std::cin >> Input;

					if (Input == 'A' || Input == 'a') {

						if (Tree.IsFull()) {

							system("CLS");
							std::cout << "Error! Tree is full!
							Try deleting a value first!\n\n";

						} else {

							//To check if the value is already
							in the tree.
							BinaryNode * TempNode;

							do {

								int InputValue;
```

```
                        std::cout << "Please enter
                        a number to add: ";
                        std::cin >> InputValue;

                        //Create temp object so
                only have to run FindNodeByValue() once.
                        TempNode =
                Tree.FindNodeByValue(InputValue);

                        if (TempNode != 0) {

                                //If value already
                                exists in tree.

                                system("CLS");
                                std::cout << "Error!
                Value already in tree! Try again!\n\n";
                                std::cout << "These
                        are the values in the tree: ";
                                Tree.Inorder();
                                std::cout << "\n\n";


                        } else {

                                //If value does not
                        already exist in tree.

Tree.AddNode(InputValue, Tree.FindNodeByValue(InputValue));

                        }

                } while (TempNode != 0);

                system("CLS");

        }

    } else if (Input =='D' || Input == 'd') {

        if (Tree.IsEmpty()) {

                system("CLS");
                std::cout << "Error! Tree is
                empty! Try adding a value!\n\n";

        } else {

                //To check if the value is not in
                tree.
                BinaryNode * TempNode;

                bool Error;

                do {

                        Error = false;
```

```
                                        int InputValue;

                                        std::cout << "Please enter
                                a number to delete from the tree: ";
                                        std::cin >> InputValue;

                                        TempNode =
                                Tree.FindNodeByValue(InputValue);

                                        if (TempNode == 0) {

                                                system("CLS");
                                                std::cout << "Error!
                                Value not in tree! Try again!\n\n";
                                                std::cout << "These
                                are the values in the tree: ";

                                                Tree.Inorder();
                                                std::cout << "\n\n";

                                                //We must run loop
                                                again.
                                                Error = true;

                                        } else {

                                                //Start at node to
                                                delete.

                        Tree.DeleteNode(Tree.FindNodeByValue(InputValue));

                                        }

                                } while (Error);

                                system("CLS");

                        }


                }
                else if (Input != 'Q' && Input != 'q') {

                        system("CLS");
                        std::cout << "Invalid Input! Try
                        again!\n\n";

                }

        } while (Input != 'Q' && Input != 'q');

        system("CLS");

        break;

}

case '2': {
```

```
            system("CLS");

            if (Tree.IsEmpty()) {

                  std::cout << "The Binary seatch tree is empty.";

            } else {

                  std::cout << "Preorder: ";
                  Tree.Preorder();

            }

            std::cout << "\n\nPress any key to return to the main
            menu: ";
            std::cin >> Input;
            system("CLS");

            break;

      }

      case '3': {

            system("CLS");

            if (Tree.IsEmpty()) {

                  std::cout << "The Binary seatch tree is empty.";

            } else {

                  std::cout << "Inorder: ";
                  Tree.Inorder();


            }

            std::cout << "\n\nPress any key to return to the main
            menu: ";
            std::cin >> Input;
            system("CLS");

            break;

      }

      case '4': {

            system("CLS");

            if (Tree.IsEmpty()) {

                  std::cout << "The Binary seatch tree is empty.";

            } else {
```

```
                                        std::cout << "Postorder: ";
                                        Tree.Postorder();


                                }

                                std::cout << "\n\nPress any key to return to the main
                                menu: ";
                                std::cin >> Input;
                                system("CLS");

                                break;

                        }

                        case '5': {

                                system("CLS");
                                break;
                                //Needed so default case doesn't run when 5 is hit.

                        }

                }

        } while (Input != '5');

        //Ask to start over with new Tree.

        std::cout << "Would you like to start again with a new tree? (Type 'Y' or
        'y' for yes): ";
        std::cin >> Input;

        std::cout << "\n\n";

} while (Input == 'Y' || Input == 'y');

}
```

## BinarySearchTree.h

```
#pragma once

struct BinaryNode {

        int Value;

        BinaryNode * Left;
        BinaryNode * Right;

};

class BinarySearchTree {
        public:

        //Constructor/Destructor
```

```
        BinarySearchTree();
        ~BinarySearchTree();

        //Functions

        BinaryNode *& FindNodeByValue(int SearchingValue);

        void AddNode(int ValueToAdd, BinaryNode* & NodeToAdd);
        void DeleteNode(BinaryNode *& NodeToDelete);

        void Preorder() const;
        void Inorder() const;
        void Postorder() const;

        bool IsFull() const;
        bool IsEmpty() const;

        int GetNumberOfElements() const;

        private:

        //Data Members

        BinaryNode * Root;
        int NumberOfElements;

        //Helper functions

        BinaryNode *& FindNode(int SearchingValue, BinaryNode *& SearchingNode);

        void PrintPreorder(BinaryNode * Current) const;
        void PrintInorder(BinaryNode * Current) const;
        void PrintPostorder(BinaryNode * Current) const;

        void MakeEmpty(BinaryNode *& Current);

};
```

**BinarySearchTree.cpp**

```
#include <iostream>
#include "BinarySearchTree.h"


//Precondition: New BinrySearchTree object created in main.
//Postcondition: BinarySearchTree object initialized.
BinarySearchTree::BinarySearchTree() {

        //Initialize root to null.
        Root = 0;

        NumberOfElements = 0;

}
```

```
//Precondition:: BinarySearchTree object goes out of scope, object was previously
initialized.
//Postcondition: All dynamically allocated memory is deallocated safely.
BinarySearchTree::~BinarySearchTree() {

	MakeEmpty(Root);

}

//Precondition: Tree object is initialized.
//Postcondition: Node with value is returned (for use in calling the delete function) or
null if not found. Tree is unchanged.
BinaryNode *& BinarySearchTree::FindNodeByValue(int SearchingValue) {

	return FindNode(SearchingValue, Root);

}

//Precondition: SearchingNode is an ancestor of the node we're looking for (or would be,
if the node does not currently exist).
//Postcondition: If found, node the value is in is returned. If not found, returns null.
Tree is unchanged.
BinaryNode *& BinarySearchTree::FindNode(int SearchingValue, BinaryNode *& SearchingNode)
{

	if (SearchingNode == 0 || SearchingNode->Value == SearchingValue) {

		//If found or not found where expected, we return what we have.
		//Returns where number should be if not found.
		return SearchingNode;

	}

	//If we can not determine a final answer yet, keep recursivly looking.

	if (SearchingValue < SearchingNode->Value) {

		//If value we're looking for is greater than what we have, go right.
		return FindNode(SearchingValue, SearchingNode->Left);

	} else if (SearchingValue > SearchingNode->Value) {

		//If value we're looking for is less than what we have, go left.
		return FindNode(SearchingValue, SearchingNode->Right);

	}

}

//Precondition: Value does not already exist in binary search tree and tree object is
initialized.
//Postcondition: Value added correctly to binary search tree.
void BinarySearchTree::AddNode(int ValueToAdd, BinaryNode* & NodeToAdd) {

	//Create new node at location, load it with values.

	NodeToAdd = new BinaryNode;
```

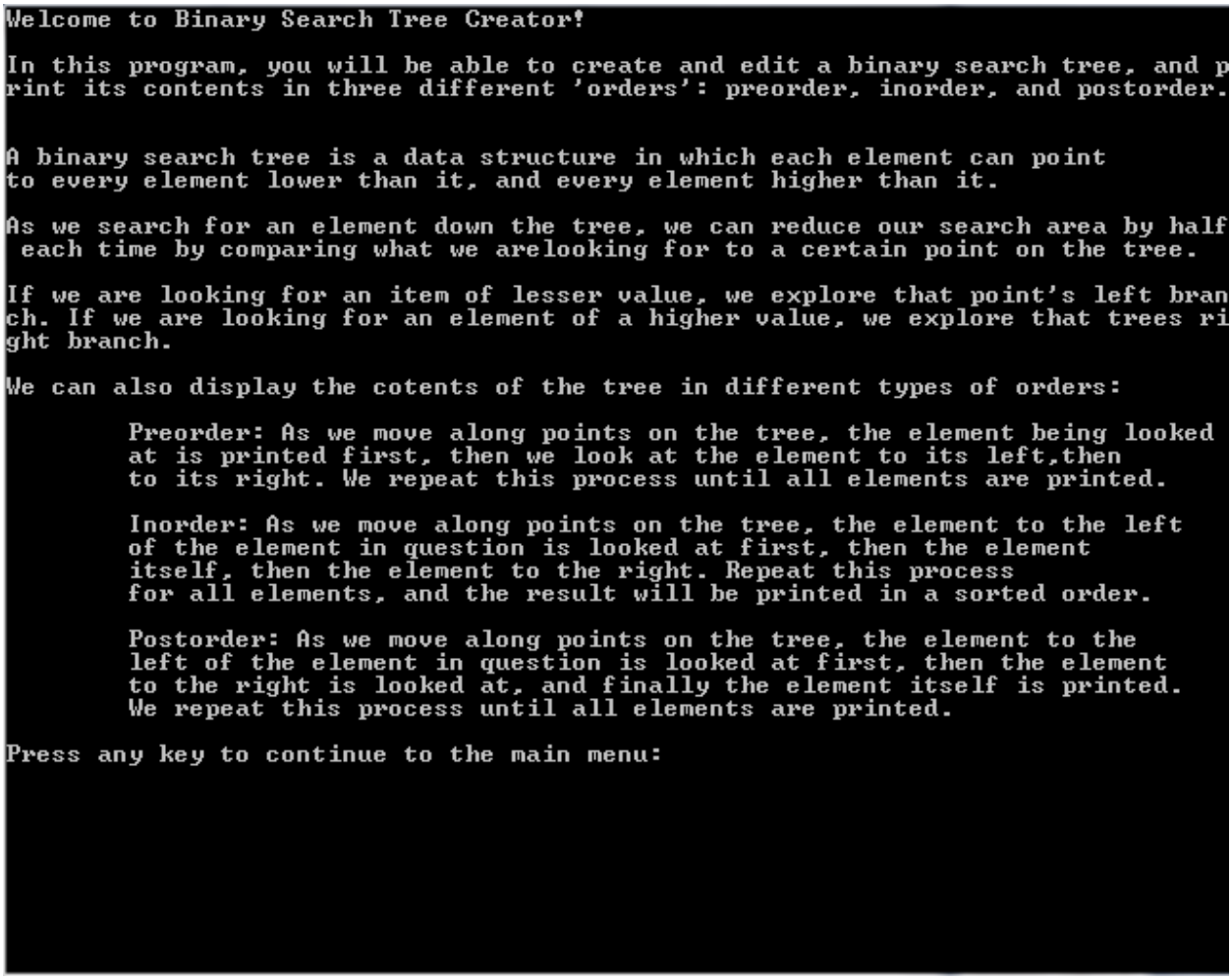
```
        NodeToAdd->Value = ValueToAdd;

        NodeToAdd->Left = 0;

        NodeToAdd->Right = 0;

        //Update the NumberOfElements
        NumberOfElements++;

}

//Precondition: Value already exists in binary search tree and tree object is
initialized.
//Postcondition: Value deleted. If two children, right-most node of left branch where
value was is deallocated after that value is overwritten over the deleted value's.
//If one children, node is deleted and pointer points to child. If no children, node is
simply deleted.
void BinarySearchTree::DeleteNode(BinaryNode *& NodeToDelete) {

        if (NodeToDelete->Left == 0 && NodeToDelete->Right == 0) {

                //If node is a leaf node.

                delete NodeToDelete;

                //Make NodeToDelete null.
                NodeToDelete = 0;

                //Update NumberOfElements to reflect changes.
                NumberOfElements--;

        } else if (NodeToDelete->Left == 0) {

                //If only the left node has no descendent.

                //Create temp node so we can safely delete
                BinaryNode * TempNode = NodeToDelete;

                NodeToDelete = NodeToDelete->Right;

                delete TempNode;

                //Update NumberOfElements to reflect changes.
                NumberOfElements--;

        } else if (NodeToDelete->Right == 0) {

                //If only the right node has no descendent.

                //Create temp node so we can safely delete
                BinaryNode * TempNode = NodeToDelete;

                NodeToDelete = NodeToDelete->Left;

                delete TempNode;

                //Update NumberOfElements to reflect changes.
                NumberOfElements--;
```

```
	} else {

		//If Two childs

		//Use TempNode to remain at the location of the value to delete.
		BinaryNode * TempNode = NodeToDelete;

		//Find Right-most child of the left branch.

		TempNode = TempNode->Left;

		while (TempNode->Right != 0) {

			TempNode = TempNode->Right;

		}

		int TempInt = TempNode->Value; //Copy value over to temp int.

		//Can not copy over to NodeToDelete yet to avoid duplicate values
		//(We are calling DeleteValue again)

		//Since the right-most node of the elft branch will
		//Either have one child or no children, we can call function
		//and the above case will handle and restructure.

		DeleteNode(FindNodeByValue(TempNode->Value));

		NodeToDelete->Value = TempInt;

		//Do not decrement because the function handling deletion of right-most
descendent of left branch will already decrement.

	}

}

//Precondition: Tree object is initialized.
//Postcondition: Tree object's values are printed in "pre"order.
void BinarySearchTree::Preorder() const {

	PrintPreorder(Root);

}

//Precondition: "Current" is a node in tree.
//Postcondition: "Current" is printed, left branch of "Current" is printed, and right
branch of "Current" is printed.
void BinarySearchTree::PrintPreorder(BinaryNode * Current) const {

	//Preorder: Print "Current" first.

	//If "Current" exists.
	if (Current != 0) {

		std::cout << Current->Value << " | ";
		PrintPreorder(Current->Left);
```

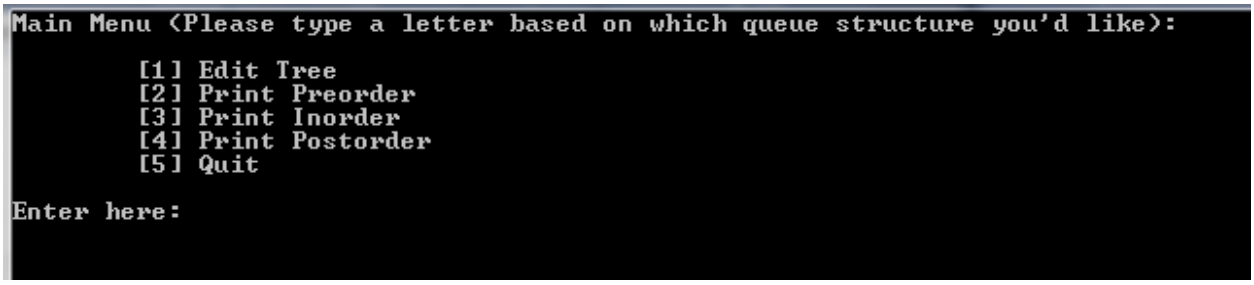
```
		PrintPreorder(Current->Right);

	}

}

//Precondition: Tree object initialized.
//Postcondition: Tree object's values printed in "in"order.
void BinarySearchTree::Inorder() const {

	PrintInorder(Root);

}

//Precondition: "Current" is a node in tree.
//Postcondition: Left branch of "Current" is printed, "Current" is printed, and right
branch of "Current" is printed.
void BinarySearchTree::PrintInorder(BinaryNode * Current) const {

	//Inorder: Print "Current" in between two branches.

	//If "Current" exists.
	if (Current != 0) {

		PrintInorder(Current->Left);
		std::cout << Current->Value << " | ";
		PrintInorder(Current->Right);

	}

}

//Precondition: Tree object initialized.
//Postcondition: Tree object's values printed in "post"order.
void BinarySearchTree::Postorder() const {

	PrintPostorder(Root);

}

//Precondition: "Current" is a node in tree.
//Postcondition: Left branch of "Current" is printed, right branch of "Current" is
printed, and "Current" is printed.
void BinarySearchTree::PrintPostorder(BinaryNode * Current) const {

	//Postorder: Print "Current" last.

	//If "Current" exists.
	if (Current != 0) {

		PrintPostorder(Current->Left);
		PrintPostorder(Current->Right);
		std::cout << Current->Value << " | ";

	}

}
```

```
//Precondition: Tree object is initialized.
//Postcondition: Determiend whether to be full or not.
bool BinarySearchTree::IsFull() const {

	BinaryNode * TempNode;

	try {

		//Create a temproary node to see if tis possible.
		//Delete after if no problems.

		TempNode = new BinaryNode;
		delete TempNode;
		return false;

	}
	catch (std::bad_alloc exception) {

		//If an error, tree is full.
		return true;

	}

}

//Precondition: Tree object is initialized.
//Postcondition: Determiend whether to be empty or not.
bool BinarySearchTree::IsEmpty() const {

	return (Root == 0);

}

//Precondition: Tree object is initialized. Current is an existing node in the tree.
First "Current" is root.
//Postcondition: Both branches of "Current" are first deleted, then "Current" is deleted.
//Everything under and including "Current" is deleted.
void BinarySearchTree::MakeEmpty(BinaryNode * & Current) {

	//Delete in postorder. Most efficient.

	if (Current != 0) {

		MakeEmpty(Current->Left);
		MakeEmpty(Current->Right);
		delete Current;

	}

}

//Precondition: Tree object is initialized.
//Postcondition: Returns the data member "NumberOfElements".
int BinarySearchTree::GetNumberOfElements() const {

	return NumberOfElements;

}
```

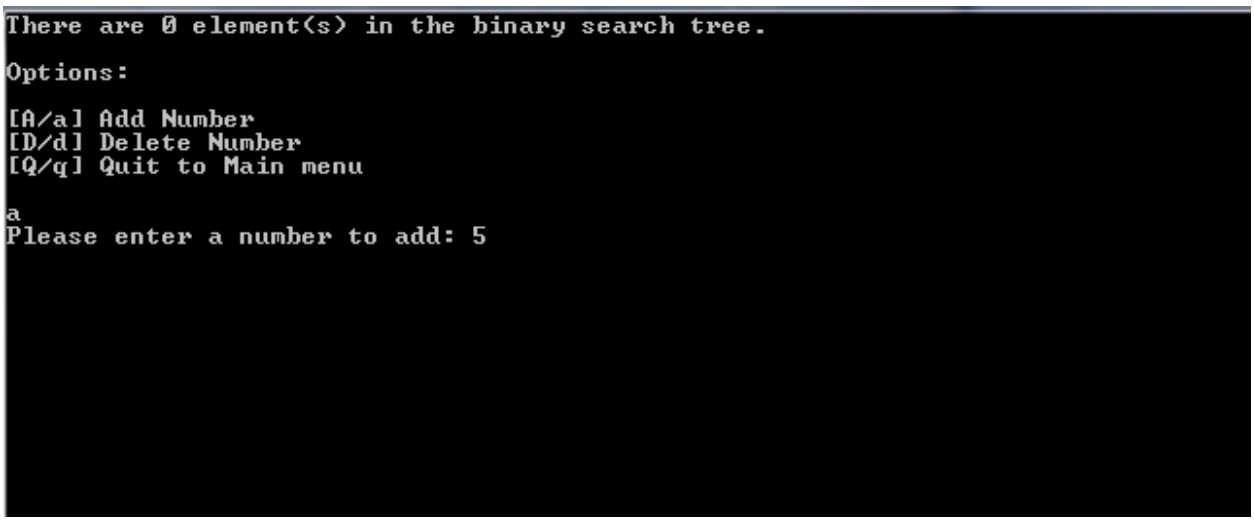
# Tests



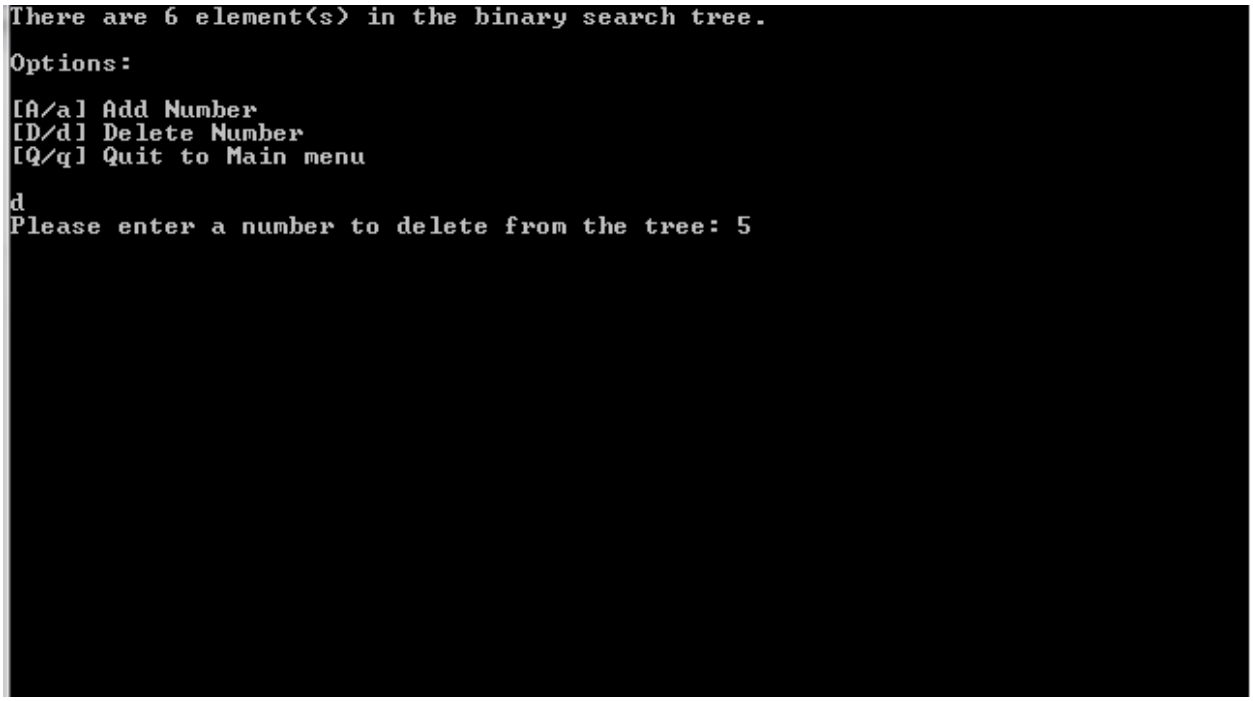

The introduction menu when the program first starts.




The main menu.

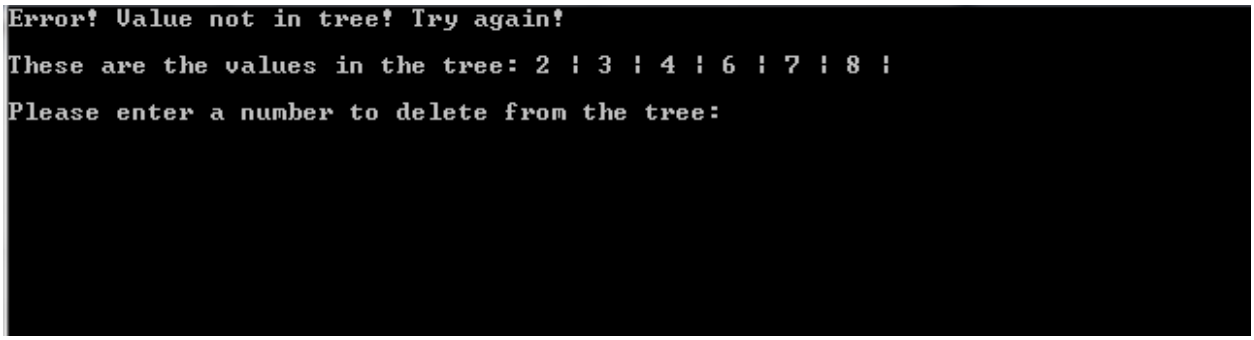



An example of adding numbers to the tree. Note the counter at the top, displaying how many elements are in the tree.

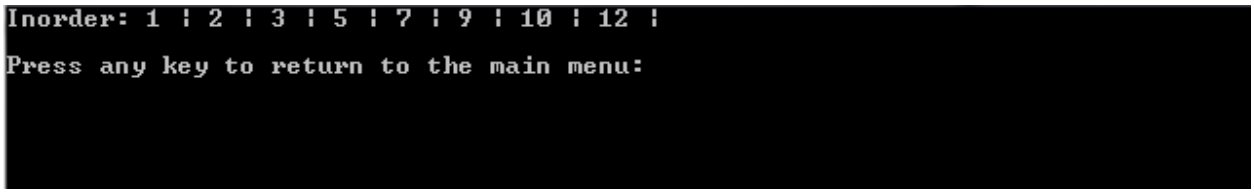



An example of the user deleting values from the tree.


Error message when adding a number that already exists in the tree.

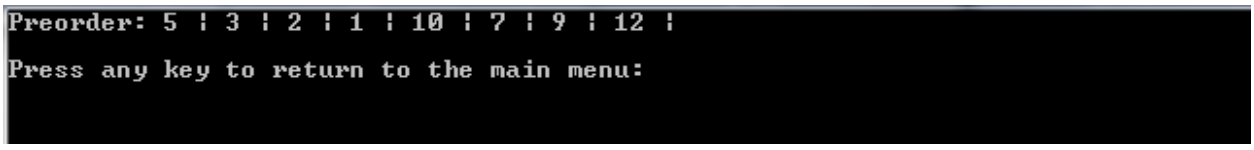



Error message when adding a number that already exists in the tree.




Inorder printing.

Elements added in order of 5 → 10 → 7 → 9 →3 → 2 →1 → 12



```
Postorder: 1 ¦ 2 ¦ 3 ¦ 9 ¦ 7 ¦ 12 ¦ 10 ¦ 5 ¦

Press any key to return to the main menu:
```

Preorder and Postorder printing. As shown by these printings, the nodes are correctly in the binary search tree structure based on how the preorder and postorder print.

```
There are 8 element(s) in the binary search tree.

Options:

[A/a] Add Number
[D/d] Delete Number
[Q/q] Quit to Main menu

d
Please enter a number to delete from the tree: 1
```

Deleting node with no children.

```
There are 7 element(s) in the binary search tree.

Options:

[A/a] Add Number
[D/d] Delete Number
[Q/q] Quit to Main menu

d
Please enter a number to delete from the tree: 7
```

Deleting node with one child.

```
Preorder: 5 ¦ 3 ¦ 2 ¦ 10 ¦ 9 ¦ 12 ¦

Press any key to return to the main menu:
```

```
Inorder: 2 | 3 | 5 | 9 | 10 | 12 |

Press any key to return to the main menu:
```

```
Postorder: 2 | 3 | 9 | 12 | 10 | 5 |

Press any key to return to the main menu:
```

Binary search tree structure maintained and adjusted based on deletions.

```
There are 6 element(s) in the binary search tree.

Options:

[A/a] Add Number
[D/d] Delete Number
[Q/q] Quit to Main menu

d
Please enter a number to delete from the tree: 5
```

Deleting node with two children, also root node.

```
Preorder: 3 | 2 | 10 | 9 | 12 |

Press any key to return to the main menu:
```

```
Inorder: 2 | 3 | 9 | 10 | 12 |

Press any key to return to the main menu:
```

```
Postorder: 2 | 9 | 12 | 10 | 3 |

Press any key to return to the main menu:
```

Binary search tree structure still maintained.

```
Invalid Input! Try again!

Main Menu (Please type a letter based on which queue structure you'd like):

        [1] Edit Tree
        [2] Print Preorder
        [3] Print Inorder
        [4] Print Postorder
        [5] Quit

Enter here:
```

```
Invalid Input! Try again!

There are 5 element(s) in the binary search tree.

Options:

[A/a] Add Number
[D/d] Delete Number
[Q/q] Quit to Main menu
```

Error message if invalid input entered in any menu.

```
Would you like to start again with a new tree? (Type 'Y' or 'y' for yes):
```

Asking to start with a new tree after exiting from main menu.

# Lessons Learned

In this project, I learned how to efficiently implement a binary search tree data structure. I learned a lot by implementing each function of the data structure, and why this data structure is more efficient than others when searching. (A O(log n) time complexity always beats out O(n).) I also learned in greater depth how the different orderings worked. I got hands on experience with pointers and what scenarios their references need to be utilized. I also got more experience designing a program that is easy for a user to use.

# References

I used no outside references for this project other than class material.