Introduction

This assignment had us implement a binary search tree. It is programmed in C++ with Microsoft Visual Studio 2019 Community Edition using the version ISO C++14 Standard. It does NOT utilize generic class templates to have a Tree<int> or a Tree<string> as there is an issue with Microsoft Visual Studio 2019 Community Edition implementing templated classes with generic types. We randomly selected 100 numbers with a range from 1-1000 for our usage in this assignment.

Class Diagram

Graphical user interface, text, application

Description automatically generated

struct TreeNode

Defined in header [<tree>](https://en.cppreference.com/w/cpp/header/set)

struct TreeNode

{

};

TreeNode is a stucture that contains an integer variable named data, a TreeNode pointer for left, right, and parent. This structure allows us to implement a binary search tree.

**Member types**

|  |  |
| --- | --- |
| **Member type** | **Definition** |
| int | data holds the current nodes value |
| TreeNode\* | pLeft is a pointer to a struct TreeNode. |
| TreeNode\* | pRight is a pointer to a struct TreeNode. |
| TreeNode\* | pParent is a pointer to a struct TreeNode. |

class Tree

Defined in header [<tree>](https://en.cppreference.com/w/cpp/header/set)

class Tree

{

};

Tree is a class that performs the role of a binary search tree. It has private data member public methods, and private methods.

**Member types**

|  |  |
| --- | --- |
| **Member type** | **Definition** |
| TreeNode\* | mpRoot is a pointer to a struct TreeNode. |
| int | mnDebug holds a value (0=false, 1=true) to output information at each method call/ |
| std::ofstream | mFile holds the output of the tree in text format to a file named “output.txt” in the current run time path. |
| std::ofstream | mFile2 holds the output of the tree in a CSV format to a file named “output.csv” in the current run time path. |

**Public Member functions**

|  |  |
| --- | --- |
| Tree() | default constructor. Calls Initialize() |
| [Tree](https://en.cppreference.com/w/cpp/container/set/~set)(int debug) | Tree(int debug) construct which calls Initialize and then sets debug mode (0=false, 1=true) |
| ~Tree() | Destructor for the Tree class. It will call DeInitialize(mpRoot) to delete the contents of the tree. |
| Initialize(int debug) | Initialize sets member variables mpRoot and mnDebug. It DOES NOT initialize the output file streams to null or empty. |
| DeInitialize() | This method calls the Empty method passing it the value for mpRoot. Empty is a recursive post-order method. |
| Add(int value) | Adds an integer into the tree if Find return 0 (false) for the value you are inserting. It calls the recursive method Add passing in mpRoot and the value you are inserting. |
| Remove(int value) | Removes an integer from the tree if Find returns 1 (true) for the value you want to remove. It calls the recursive method Remove passing in mpRoot and the value you want to remove. |
| Maximum() | Returns the trees maximum value. This method calls the recursive method Maximum passing in mpRoot. |
| Minimum() | Returns the trees minimum value. This method calls the recursive method Minimum passing in mpRoot. |
| Find(int value) | Finds a value in the tree. Returns 1 (found) or 0 (not found). This method calls the recursive method Find passing in mpRoot and the value you are looking for. |
| InorderTraverse(int full) | This method displays the tree in order by calling the recursive method InorderTraverse passing in mpRoot and the value for full. Full = 1 outputs all of the node details to the screen and Full = 0 outputs just the data and a space to the screen. |
| PreorderTraverse(int full) | This method displays the tree in order by calling the recursive method PreorderTraverse passing in mpRoot and the value for full. Full = 1 outputs all of the node details to the screen and Full = 0 outputs just the data and a space to the screen. |
| PostorderTraverse(int full) | This method displays the tree in order by calling the recursive method PostorderTraverse passing in mpRoot and the value for full. Full = 1 outputs all of the node details to the screen and Full = 0 outputs just the data and a space to the screen. |
| OutputTreeToFile(const char\* path, const char\* filename, int append) | This method will output the tree contents to a file. Path = “.\\” to output in the current runtime directory. Filename = “outout.txt”. flag append (0=false, 1 = true) is used to indicate ios::out (append = 1) and ios::out | ios::trunc (append =0). |
| OutputTreeToCSVFile(const char\* path, const char\* filename, int append) | This method will output the tree contents to a file. Path = “.\\” to output in the current runtime directory. Filename = “outout.csv”. flag append (0=false, 1 = true) is used to indicate ios::out (append = 1) and ios::out | ios::trunc (append =0). The tree’s contents are output in this order: NODE,DATA, PARENT, LEFT, RIGHT <<newline>> |
| OutputHorizontalTree() | This method outputs a horizontal view of the tree’s contents to the console. |

**Private Member functions**

|  |  |
| --- | --- |
| Add(TreeNode\* pNode, int value) | This recursive method will add a value to the tree using standard binary tree logic (if current node->data < value) go right, else go left. |
| Empty(TreeNode\* pNode) | This recursive method performs a post order traversal to remove all nodes from the tree. |
| Successor(int value) | This method calls Find passing in mpRoot and value. If the point returned by Find is NULL it returns -1, else the successor to the node found. |
| Successor(TreeNode\* pNode) | This method will find a nodes successor. If pNode->right <> NULL return minimum(pNode->pRight) Else walk the pNode->pParent until parent = NULL and current node == parent->pRight. Then it returns the parent node = NULL as -1 or parent node data as the successor value. |
| Predecessor(int value) | This method calls Find passing in mpRoot and value. If the point returned by Find is NULL it returns -1, else the predecessor to the node found. |
| Predecessor (TreeNode\* pNode) | This method will find a nodes successor. If pNode->pLeft <> NULL return Maximum(pNode->pRight) Else walk the pNode->pParent until parent = NULL and current node == parent->pLeft. Then it returns the parent node = NULL as -1 or parent node data as the predecessor value. |
| Remove(TreeNode\* pNode, int value) | This recursive method will remove from the tree the value specified. |
| Maximum(TreeNode\* pNode) | This recursive method will check if pNode == NULL and returns -1 if it does. Then it checks pNode->pRight == NULL and returns pNode->data if pNode->pRight = NULL. Otherwise it calls Maximum passing in the value for pNode->pRight. |
| Minimum(TreeNode\* pNode) | This recursive method will check if pNode == NULL and returns -1 if it does. Then it checks pNode->pLeft == NULL and returns pNode->data if pNode->pLeft = NULL. Otherwise it calls Minimum passing in the value for pNode->pLeft. |
| Find(TreeNode\* pNode, int value) | This recursive method checks to see if pNode->data == value if it matches it returns pNode Else it checks pNode->data < value then go right else go left. |
| InorderTraverse(TreeNode\* pNode, int full) | This recursive method checks pNode != NULL and calls InorderTraverse(pNode->pLeft), then prints out to the console all the nodes information if FULL=1 else prints out to the console the pNode->data and a “ “. Lastly it calls InorderTraverse(pNode->pRight); |
| PreorderTraverse(TreeNode\* pNode, int full) | This recursive method checks pNode != NULL and prints out to the console all the nodes information if FULL=1 else prints out to the console the pNode->data and a “ “. Then it calls PreorderTraverse (pNode->pLeft), then it calls PreorderTraverse (pNode->pRight); |
| PostorderTraverse(TreeNode\* pNode, int full) | This recursive method checks pNode != NULL and calls PostorderTraverse (pNode->pLeft), then it calls PostorderTraverse (pNode->pRight); Then it prints out to the console all the nodes information if FULL=1 else prints out to the console the pNode->data and a “ “. |
| EndsWith(const char\* str, const char\* suffix) | This is a helper method to determine if a path ends with “\”. Its use if helpful in formatting path and file into pathFile that is used to open a file with. |
| OutputTreeToFile(TreeNode\* pNode) | This recursive method outputs InOrder the trees contents to the console by writing out all of the nodes member values with a CR/LF between each nodes member values.. |
| OutputTreeToCSVFile(TreeNode\* pNode) | This recursive method outputs PreOrder the trees contents to the console by writing out all of the nodes member values with CR/LF between each nodes member values. |
| OutputHorizontalTree(const std::string prefix, TreeNode\* pNode, int isLeft) | If pNode is not NULL then it outs the current pNode member variables separated by commas to the console. It then recursively calls itself by going left then right. |
| OutputHorizontalTree(TreeNode\* pNode) | This method is called by OutputHorizontalTree(mpRoot).  This method then calls the recursive OutputHorizontalTree method passing “”, pNode,false) to kick start the output of the binary tree |

**Example Output**

**Console:**

**Debug Mode = 1**

Tree::Tree(1)

Tree::Initialize

Tree::Add(int 3 )

Tree::Find(value 3 )

Tree::Find(TreeNode\* 00000000, value 3 )

Tree::Add(TreeNode\* 00000000, int 3 )

Tree::Add(int 1 )

Tree::Find(value 1 )

Tree::Find(TreeNode\* 00993AF0, value 1 )

Tree::Find(TreeNode\* 00000000, value 1 )

Tree::Add(TreeNode\* 00993AF0, int 1 )

Tree::Add(TreeNode\* 00000000, int 1 )

Tree::Add(int 5 )

Tree::Find(value 5 )

Tree::Find(TreeNode\* 00993AF0, value 5 )

Tree::Find(TreeNode\* 00000000, value 5 )

Tree::Add(TreeNode\* 00993AF0, int 5 )

Tree::Add(TreeNode\* 00000000, int 5 )

Tree::Add(int 2 )

Tree::Find(value 2 )

Tree::Find(TreeNode\* 00993AF0, value 2 )

Tree::Find(TreeNode\* 00993BB0, value 2 )

Tree::Find(TreeNode\* 00000000, value 2 )

Tree::Add(TreeNode\* 00993AF0, int 2 )

Tree::Add(TreeNode\* 00993BB0, int 2 )

Tree::Add(TreeNode\* 00000000, int 2 )

Tree::Add(int 4 )

Tree::Find(value 4 )

Tree::Find(TreeNode\* 00993AF0, value 4 )

Tree::Find(TreeNode\* 00995440, value 4 )

Tree::Find(TreeNode\* 00000000, value 4 )

Tree::Add(TreeNode\* 00993AF0, int 4 )

Tree::Add(TreeNode\* 00995440, int 4 )

Tree::Add(TreeNode\* 00000000, int 4 )

Post Order Traverse BEFORE

t--3

|--1

| t--2

t--5

|--4

Removing 2

Tree::Remove(int 2 )

Tree::Find(value 2 )

Tree::Find(TreeNode\* 00993AF0, value 2 )

Tree::Find(TreeNode\* 00993BB0, value 2 )

Tree::Find(TreeNode\* 00994D00, value 2 )

Tree::Remove(TreeNode\* 00993AF0, value 2 )

Tree::Remove(TreeNode\* 00993BB0, value 2 )

Tree::Remove(TreeNode\* 00994D00, value 2 )

2 == 2

Removing pNode: 00994D00

Post Order Traverse AFTER remove node (node has no nodes)

t--3

|--1

t--5

|--4

Post Order Traverse BEFORE

t--3

|--1

| t--2

t--5

|--4

Removing 5

Post Order Traverse AFTER remove node (node has 1 nodes)

t--3

|--1

| t--2

t--4

Post Order Traverse BEFORE

t--3

|--1

| t--2

t--5

|--4

Removing 3

Post Order Traverse AFTER remove node (node has 1 nodes)

t--4

|--1

| t--2

t--5

Tree::~Tree

Tree::DeInitialize

Tree::Empty(TreeNode\* 00993AF0 )

Tree::Empty(TreeNode\* 00993BB0 )

Tree::Empty(TreeNode\* 00000000 )

Tree::Empty(TreeNode\* 00000000 )

Tree::Empty(TreeNode\* 00995440 )

Tree::Empty(TreeNode\* 00995380 )

Tree::Empty(TreeNode\* 00000000 )

Tree::Empty(TreeNode\* 00000000 )

Tree::Empty(TreeNode\* 00000000 )

Debug Mode = 0

You won’t see output related to the function being called with what value.

Outputting a nodes contents FULL = 1

pNode: 009935F0

pNode->data: 7

pNode->pParent: 00990878

pNode->pLeft: 00000000

pNode->pRight: 00000000

---------------pNode: 00990878

pNode->data: 13

pNode->pParent: 00986490

pNode->pLeft: 009935F0

pNode->pRight: 009930A0

---------------Outputting a nodes contents FULL = 0

7 13

Using an array of 100 random numbers whose values range from 1-1000

numbers array: 195 781 611 382 967 769 567 960 13 761 563 956 361 849 747 352 190 744 546 742 937 134 538 140 136 922 920 350 325 915 519 472 710 709 905 985 703 897 304 499 743 101 691 886 24 96 735 882 93 289 91 286 876 316 489 478 683 204 275 273 907 879 663 38 598 658 854 652 650 451 646 645 37 838 623 35 258 237 629 628 429 821 426 225 618 221 23 219 21 414 412 871 804 604 799 444 7 202 123 788

In Order Traverse

root :00986490

7 13 21 23 24 35 37 38 91 93 96 101 123 134 136 140 190 195 202 204 219 221 225 237 258 273 275 286 289 304 316 325 350 352 361 382 412 414 426 429 444 451 472 478 489 499 519 538 546 563 567 598 604 611 618 623 628 629 645 646 650 652 658 663 683 691 703 709 710 735 742 743 744 747 761 769 781 788 799 804 821 838 849 854 871 876 879 882 886 897 905 907 915 920 922 937 956 960 967 985

Pre Order Traverse

root :00986490

195 13 7 190 134 101 24 23 21 96 93 91 38 37 35 123 140 136 781 611 382 361 352 350 325 304 289 286 204 202 275 273 258 237 225 221 219 316 567 563 546 538 519 472 451 429 426 414 412 444 499 489 478 598 604 769 761 747 744 742 710 709 703 691 683 663 658 652 650 646 645 623 618 629 628 735 743 967 960 956 849 838 821 804 799 788 937 922 920 915 905 897 886 882 876 854 871 879 907 985

Post Order Traverse

root :00986490

7 21 23 35 37 38 91 93 96 24 123 101 136 140 134 190 13 202 219 221 225 237 258 273 275 204 286 289 316 304 325 350 352 361 412 414 426 444 429 451 478 489 499 472 519 538 546 563 604 598 567 382 618 628 629 623 645 646 650 652 658 663 683 691 703 709 735 710 743 742 744 747 761 769 611 788 799 804 821 838 871 854 879 876 882 886 897 907 905 915 920 922 937 849 956 960 985 967 781 195

Tree Minimum = 7

Tree Maximum = 985