CS5103

DJ Yuhn

Assignment 2

Binary Search Tree

A binary search tree is a data structure that begins with a root node and has at most two children for each node within the tree. The value stored at the root node is greater than the values stored to the left and is less than the values stored to the right. The time complexity for searching within a binary tree is $O(\log_2 N)$. A binary tree has three different ways to traverse the elements: preorder, inorder, and postorder. The following is a procedural algorithm for each traversal:

- Preorder:
  - Print out the root's value, regardless if it is the actual root or a subtree's root.
  - Traverse the left subtree of this node by recursively calling on preorder function.
  - Traverse the right subtree of this node by recursively calling on preorder function.
- Inorder:
  - Traverse the left subtree of the root node by recursively calling on inorder function.
  - Print out the root's value, regardless if it is the actual root or a subtree's root or leaf.
  - Traverse the right subtree of the root node by recursively calling on inorder function.
- Postorder:
  - Traverse the left subtree of the root node by recursively calling on postorder function.
  - Traverse the right subtree of the root node by recursively calling on postorder function.
  - Print out the root's value, regardless if it is the actual root or a subtree's root or leaf.

In my implementation of a binary search tree, I used the Shunting Yard algorithm to alter the following expression for the assignment:

$$(a + b * c) + ((d * e + f) * g)$$

The Shunting Yard algorithm is an algorithm for parsing mathematical expressions that are in infix notation, producing a postfix notation string. This allowed me to parse the above expression to insert each operator and operand into the expression tree. The procedural algorithm for Shunting Yard is as follows from Wikipedia:

- While there are tokens to be read:
  - Read a token:
    - If the token is an operand, push it to the output queue.
    - If the token is an operator then:
      - While there is an operator at the top of the operator stack with greater or equal precedence, pop operators from the stack into the output queue.
      - Push the read operator onto the operator stack.
    - If the token is a left parenthesis, '('.
      - Push it onto operator stack.
    - If the token is a right parenthesis, ')'.

            - While the operator at the top of the operator stack is not a left parenthesis, '(':
                - Pop operators from the operator stack onto the output queue.
            - Pop the left parenthesis, '(', from the stack.
- If there are no more tokens to be read:
    - While there are still operator tokens in the stack:
        - Pop the operator onto the output queue.

After producing the postfix notation string using the above algorithm, I proceeded to insert the string into the tree. The functions isOperator() and createTree() were used to create the expression tree. The following is the procedural algorithm for each function.

- isOperator(char op)
    - If op is a '+' or a '-' or a '*' or a '/', return true.
    - Else return false.
- createTree(stack<node*>, string postfixString)
    - Utilize a stack to hold the soon to be created nodes for each operator and operand.
    - Create a for loop to go through the length of the postfixString.
    - For each character encountered in the string, check if it is an operator.
        - If the character is not an operator.
            - Create a new node and insert the character into the new node.
            - Push the new node to the stack.
        - If the character is an operator.
            - Create a new node and insert the operator into the node.
            - Have the right pointer of the new node point to the node at the top of the stack.
            - Pop the top of the stack.
            - Have the left pointer of the new node point to the node at the top of the stack.
            - Pop the top of the stack.
            - Push the new node.
    - Return the top of the stack.

Once the expression tree was created, I utilized the procedural algorithms for preorder, inorder, and postorder defined earlier to traverse the tree and output the expression in different forms.

Main_ExpTree.cpp Code below for the expression tree.

```
/**
CS5103 - Assignment 2 Expression Tree

@author DJ Yuhn
*/

#include <iostream>
#include <string>
#include <vector>
#include <map>
#include <list>
#include <stack>
#include <algorithm>

struct node {
	char data;
	node* left = nullptr;
	node* right = nullptr;
};

bool isOperator(char op);
void preorder(node * leaf);
void inorder(node * leaf);
void postorder(node * leaf);
std::string shuntingYard(std::string rpnExpr);
node* createTree(std::stack<node*> stack, std::string rpnExpr);
void destroyTree(node* leaf);

int main() {
	std::string expr = "(a + b * c) + ((d * e + f) * g)";

	std::stack<node*> treeStack;

	std::string rpnExpr = shuntingYard(expr);

	node* root = createTree(treeStack, rpnExpr);

	std::cout << "Original Expression:\n" << expr
		<< "\nParentheses are removed in the output below.\n\n";
	std::cout << "Preorder:" << std::endl;
	preorder(root);
	std::cout << "\n\nInorder:" << std::endl;
	inorder(root);
	std::cout << "\n\nPostorder:" << std::endl;
	postorder(root);
	std::cout << "\n\n";

	destroyTree(root);

	return 0;
}

bool isOperator(char op) {
	if (op == '+' || op == '-' ||
		op == '*' || op == '/')
```

```
                return true;
        else
                return false;
}

void preorder(node * leaf) {
        if (leaf != nullptr) {
                std::cout << leaf->data << " ";
                preorder(leaf->left);
                preorder(leaf->right);
        }
};

void inorder(node * leaf) {
        if (leaf != nullptr) {
                inorder(leaf->left);
                std::cout << leaf->data << " ";
                inorder(leaf->right);
        }
};

void postorder(node * leaf) {
        if (leaf != nullptr) {
                postorder(leaf->left);
                postorder(leaf->right);
                std::cout << leaf->data << " ";
        }
};

std::string shuntingYard(std::string expr) {
        std::vector<char> output;
        std::stack<char> stack;
        std::string outputString;

        std::map<char, int> opPriority;
        opPriority['+'] = 10;
        opPriority['-'] = 10;
        opPriority['*'] = 20;
        opPriority['/'] = 20;

        // Remove whitespace from the expression
        expr.erase(std::remove_if(expr.begin(), expr.end(), isspace),
expr.end());

        // Gather the characters
        for (int i = 0; i < expr.size(); i++) {

                // If character is operator
                if (isOperator(expr[i])) {
                        char o1 = expr[i];

                        if (!stack.empty()) {
                                char o2 = stack.top();

                                while (isOperator(o2) && (opPriority[o1] <=
opPriority[o2])) {
                                        // Pop o2 and on to output
```

```
                                        stack.pop();
                                        output.push_back(o2);

                                        if (!stack.empty())
                                                o2 = stack.top();

                                        else
                                                break;
                                }
                        }

                        stack.push(o1);
                }

                // If character is '('
                else if (expr[i] == '(') {
                        stack.push(expr[i]);
                }

                // If character is ')'
                else if (expr[i] == ')') {
                        char topChar = stack.top();
                        while (stack.top() != '(') {
                                output.push_back(topChar);
                                stack.pop();
                                if (stack.empty())
                                        break;
                                topChar = stack.top();
                        }

                        if (!stack.empty())
                                stack.pop();
                }

                // If character is number or letter
                else
                        output.push_back(expr[i]);


        }

        while (!stack.empty()) {
                char restChar = stack.top();

                output.push_back(restChar);

                stack.pop();
        }

        for (int i = 0; i < output.size(); i++)
                outputString += output[i];

        return outputString;

}

node * createTree(std::stack<node*> stack, std::string rpnExpr) {
```

```
        for (int i = 0; i < rpnExpr.size(); i++) {
                if (!isOperator(rpnExpr[i])) {
                        node *newNode = new node;
                        newNode->data = rpnExpr[i];
                        stack.push(newNode);
                }

                else if (isOperator(rpnExpr[i])) {
                        node *newNode = new node;
                        newNode->data = rpnExpr[i];
                        newNode->right = stack.top();
                        stack.pop();
                        newNode->left = stack.top();
                        stack.pop();
                        stack.push(newNode);
                }
        }

        return stack.top();
}

void destroyTree(node* leaf) {
        if (leaf != nullptr) {
                destroyTree(leaf->left);
                destroyTree(leaf->right);
                delete leaf;
        }
};
```

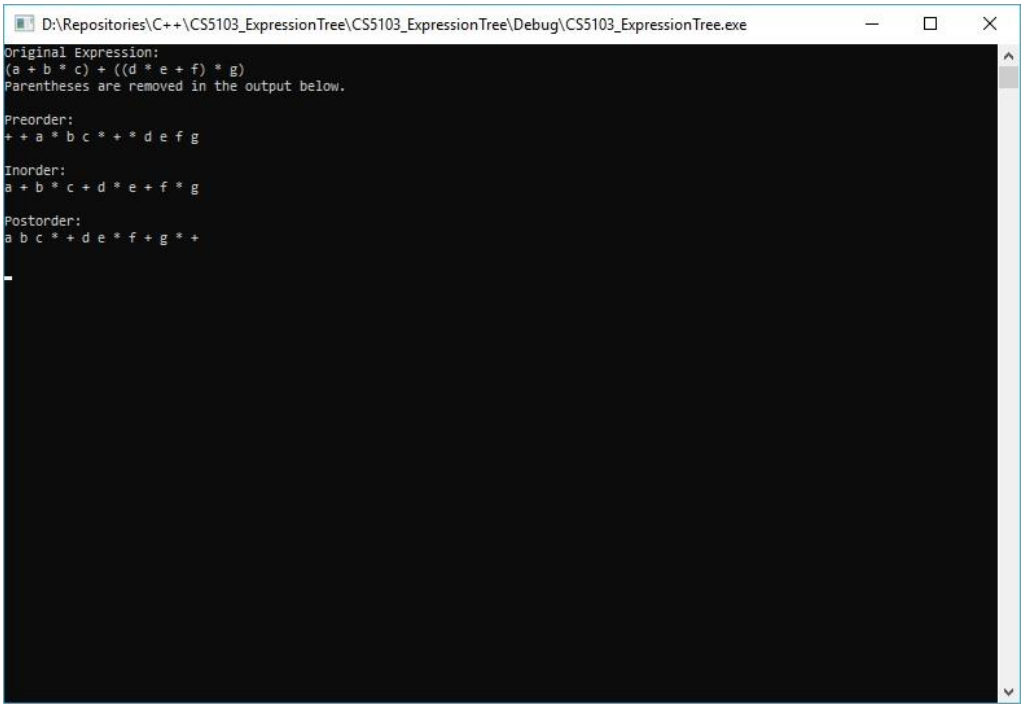

Figure 1 - Output for Expression Tree

Adelson-Velskii and Landis (AVL) Tree

An AVL Tree is a data structure that is a self balancing binary search tree. It is an approximation of an ideal binary tree (completely balanced), by maintaining a height close to minimum. Much of the functionality of an AVL tree is the same as the binary search tree, the difference being functions necessary to conduct rotations to keep itself balanced. In my implementation of the AVL tree, the node structure was given a variable of type integer to track its height as the tree becomes larger. The procedural algorithm for insert(), getDiff(), updateHeight(), rebalance(), rotateLeft(), rotateRight(), rotateRightLeft(), and rotateLeftRight() are as follows:

- insert(T data, node*& leaf)
    - If the node pointer is null
        - Create a new node inserting the data into the node.
        - Update the height of the node by calling updateHeight().
    - Else
        - If the data to be inserted is less than the value of the node.
            - Recursively call insert on the left node.
            - Call the rebalance function on the current node.
        - If the data to be inserted is greater than the value of the node.
            - Recursively call insert on the right node.
            - Call the rebalance function on the current node.
    - Update the height of the node.
- getDiff(node *& leaf)
    - Create two integer variables, leftH and rightH, and initialize both to 0.
    - If leaf's left node is not a null pointer, assign the height from this left node to leftH.
    - If leaf's right node is not a null pointer, assign the height from this right node to rightH.
    - Return leftH – rightH.
- updateHeight() // Used to update the node's height when adding and deleting nodes
    - Create three integer variables, leftH, rightH, and max, and initialize each to 0.
    - If leaf's left node is not a null pointer, assign the height from this left node to leftH.
    - If leaf's right node is not a null pointer, assign the height from this right node to rightH.
    - If leftH is greater than rightH, assign max the value of leftH.
        - Else assign max the value of rightH.
    - Assign height to be max + 1.
- rebalance(node *& leaf)
    - Create an integer variable, heightDiff, and assign it the difference from the leaf.
    - If heightDiff is greater than 1.
        - If the difference of the leaf's left node is greater than 0, assign leaf to be the nodes after rotating them right.
        - Else
            - Assign leaf to be the nodes after rotating them left then right.
    - Else if heightDiff is less than -1.
        - If the difference of the leaf's right node is less than 0, assign leaf to be the nodes after rotating them left.
        - Else

        - Assign leaf to be the nodes after rotating them right then left.
- rotateLeft(node *& leaf)
    - Create a temp pointer of type node and have it point to the right node of leaf.
    - Have leaf's right pointer point to the temp pointer's node's left node.
    - Have temp pointer's node's left pointer point to leaf.
    - Call updateHeight on leaf.
    - Return the temp pointer.
- rotateRight(node *& leaf)
    - Create a temp pointer of type node and have it point to the left node of leaf.
    - Have leaf's left pointer point totemp pointer's node's right node.
    - Have temp pointer's node's right pointer point to leaf.
    - Call updateHeight on leaf.
    - Return the temp pointer.
- rotateRightLeft(node *& leaf)
    - Create a temp pointer of type node and have it point to leaf's right node.
    - Have leaf's right pointer be assigned the pointer after having called rotateRight(temp).
    - Return the pointer returned from rotateLeft(leaf).
- rotateLeftRight(node *& leaf)
    - Create a temp pointer of type node and have it point to leaf's left node.
    - Have leaf's left pointer be assigned the pointer after having called rotateLeft(temp).
    - Return the pointer returned from rotateRight(leaf).

The following sequences of numbers were given for the assignment:

9, 27, 50, 15, 2, 21, 18, 32, 44, 28, 36

Using the above methods, as each item from the sequence was inserted into the tree, the tree would balance itself if necessary.

AVLTree_Main.cpp Code below for the AVL Tree.

```
/**
CS5103 - Assignment 2 AVL Tree
AVLTree_Main.cpp

@author DJ Yuhn
*/

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

int main() {
	AVL<int> tree;
	int arr[] = { 9,27,50,15,2,21,18,32,44,28,36 }; //Sequence of numbers
given


	for (int i = 0; i < 11; i++) {
		if (i == 0) {
			std::cout << arr[i] << " was inserted into the tree."
<< std::endl;
			tree.insert(arr[i]);
			std::cout << "The following is the tree's data
inorder:"
				<< std::endl;

			tree.inorder();
			std::cout << "\nThe height of the tree is: "
				<< tree.getTreeHeight() << std::endl;
		}
		else {
			std::cout << "\n\n" << arr[i] << " was inserted into
the tree.";
			tree.insert(arr[i]);
			std::cout << "\nThe following is the tree's data
inorder after any necessary rotations:"
				<< std::endl;

			tree.inorder();
			std::cout << "\nThe height of the tree is: "
				<< tree.getTreeHeight() << std::endl;
		}
	}

	return 0;
};
```

AVL.h Code below for the AVL Tree.

```
/**
CS5103 - Assignment 2 AVL Tree
AVL.h

@author DJ Yuhn
*/

#ifndef AVL_H
#define AVL_H

template <class T>
class AVL {
public:
	AVL() { root = nullptr; }
	~AVL() { destroyTree(root); }

	void insert(T value) { insert(value, root); }
	void inorder() { inorder(root); }
	int getTreeHeight() const { return root->height - 1; }

private:
	struct node {
		T data;
		node* left;
		node* right;
		int height;

		node(T val) {
			data = val;
			left = nullptr;
			right = nullptr;
			height = 0;
		}

		void updateHeight();
	};

	node* root;

	void insert(T data, node*& leaf);
	void destroyTree(node*& leaf);

	void rebalance(node*& leaf);
	int getDiff(node*& leaf);
	node* rotateLeft(node*& leaf);
	node* rotateRight(node*& leaf);
	node* rotateRightLeft(node*& leaf);
	node* rotateLeftRight(node*& leaf);

	void inorder(node * leaf);
};

/**
Updates the node height when called. Use when inserting or deleting nodes.
```

```
*/
template<class T>
void AVL<T>::node::updateHeight() {
	int leftH = 0;
	int rightH = 0;
	int max = 0;

	if (left != nullptr)
		leftH = left->height;
	if (right != nullptr)
		rightH = right->height;

	if (leftH > rightH)
		max = leftH;
	else
		max = rightH;

	height = max + 1;
};

/**
Inserts the value into the AVL. Update the height of the node and rebalance.

@param value The value of the data in the node.
@param leaf Pointer of type node to a leaf.
*/
template<class T>
void AVL<T>::insert(T value, node *& leaf) {
	if (leaf == nullptr) {
		leaf = new node(value);
		leaf->updateHeight();
	}

	else {
		if (value < leaf->data) {
			insert(value, leaf->left);
			leaf->updateHeight();
			rebalance(leaf);
		}
		else {
			insert(value, leaf->right);
			leaf->updateHeight();
			rebalance(leaf);
		}
	}

	leaf->updateHeight();
};


/**
Recursively deletes the nodes of the AVL tree

@param leaf Pointer of type node to a leaf.
*/
template<class T>
inline void AVL<T>::destroyTree(node *& leaf) {
```

```
        if (leaf != nullptr) {
                destroyTree(leaf->left);
                destroyTree(leaf->right);
                delete leaf;
        }
};

/**
Checks each node's left and right branches. Determines if they are
unbalanced.
If the branches are unbalanced, they are rotated upwards until the heights
are equal.

@paramm leaf Pointer of type node to a leaf.
*/
template<class T>
inline void AVL<T>::rebalance(node *& leaf) {
        int heightDiff = getDiff(leaf);
        if (heightDiff > 1) {
                if (getDiff(leaf->left) > 0)
                        leaf = rotateRight(leaf);
                else
                        leaf = rotateLeftRight(leaf);
        }
        else if (heightDiff < -1) {
                if (getDiff(leaf->right) < 0)
                        leaf = rotateLeft(leaf);
                else
                        leaf = rotateRightLeft(leaf);
        }
}

/**
Get the difference of the node's left and right branch heights.
If it is positive, the left side is greater. Else the right side is greater.

@param leaf Pointer of type node to a leaf.
*/
template<class T>
inline int AVL<T>::getDiff(node *& leaf) {
        int leftH = 0;
        int rightH = 0;

        if (leaf->left != nullptr)
                leftH = leaf->left->height;
        if (leaf->right != nullptr)
                rightH = leaf->right->height;

        return leftH - rightH;
};

/**
Rotates a branch to the left to balance and updates the height of the node.

@param leaf Pointer of type node to a leaf.
@return temp Pointer to the elevated right node that is now above the
original leaf
```

```
*/
template<class T>
typename AVL<T>:: node * AVL<T>::rotateLeft(node *& leaf) {
	node* temp = leaf->right;
	leaf->right = temp->left;
	temp->left = leaf;

	leaf->updateHeight();

	std::cout << "\n\tThe node containing " << temp->data
		<< " and its right branch were rotated left. \n\tThe node
containing "
		<< leaf->data << " and its left branch are now " << temp->data
		<< " node's left branch.";

	return temp;
};

/**
Rotates a branch to the right to balance and updates the height of the node.

@param leaf Pointer of type node to a leaf.
@return temp Pointer to the elevated left node that is now above the original
leaf
*/
template<class T>
typename AVL<T>:: node * AVL<T>::rotateRight(node *& leaf) {
	node* temp = leaf->left;
	leaf->left = temp->right;
	temp->right = leaf;

	std::cout << "\n\tThe node containing " << temp->data
		<< " and its left branch were rotated right. \n\tThe node
containing "
		<< leaf->data << " and its right branch are now " << temp-
>data
		<< " node's right branch.";

	leaf->updateHeight();

	return temp;
};

/**
Rotates a branch to the right and then the left.

@param leaf Pointer of type node to a leaf.
@return Pointer to the elevated leaf's right node's left branch.
*/
template<class T>
typename AVL<T>:: node * AVL<T>::rotateRightLeft(node *& leaf) {
	node* temp = leaf->right;
	leaf->right = rotateRight(temp);

	return rotateLeft(leaf);
};
```

```
/**
Rotates a branch to the left and then the right.

@param leaf Pointer of type node to a leaf.
@return Pointer to the elevated leaf's left node's right branch.
*/
template<class T>
typename AVL<T>:: node * AVL<T>::rotateLeftRight(node *& leaf) {
	node* temp = leaf->left;
	leaf->left = rotateLeft(temp);

	return rotateRight(leaf);
}

/**
Traverses the tree inorder, outputting contents of each node.

@param leaf Pointer of type node to a leaf.
*/
template<class T>
void AVL<T>::inorder(node * leaf) {
	if (root == nullptr)
		std::cout << "Tree is empty" << std::endl;

	else if (leaf != nullptr) {
		inorder(leaf->left);
		std::cout << leaf->data << " ";
		inorder(leaf->right);
	}
};

#endif
```

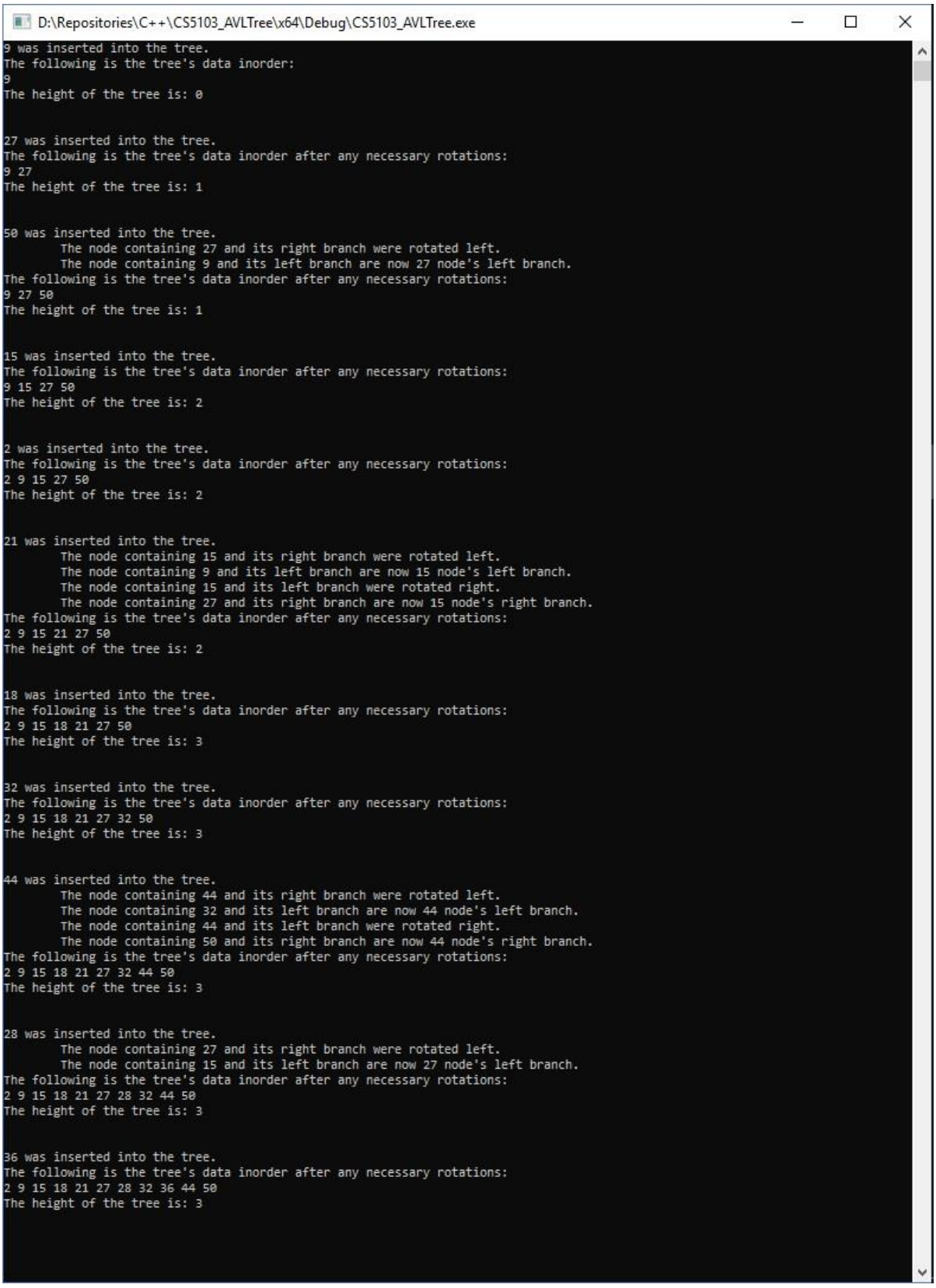

Figure 2 - Output for AVL Tree

References

https://en.wikipedia.org/wiki/Shunting-yard_algorithm

https://stackoverflow.com/questions/43615177/is-it-possible-to-construct-a-tree-of-postfix-or-prefix-form

https://stackoverflow.com/questions/12381210/create-binary-tree-from-mathematical-expression#12381550

https://stackoverflow.com/questions/423898/postfix-notation-to-expression-tree

https://github.com/djyuhn/CS5103_BinarySearchTree

https://kukuruku.co/post/avl-trees/

http://www.sanfoundry.com/cpp-program-implement-avl-trees/