Gabriel Brehm

CS260

Dr, Breeann Flesch

3 May 2018

Lab 4 Report

For this lab I wrote two new classes to create and use an AVL tree. The AVL tree class inherited from my BST class, so it can make use of all of BST's functions and methods. The AVL tree node class inherited from my tree node class. I also updated a few of the functions from the BST class so they could be properly overwritten and utilized in the AVL tree class. However, this report will only cover the functions in the AVL tree class.

The first function is a constructor that simply calls the BST constructor.

```
template<typename T>
AVLTree<T>::AVLTree()
{
    BSTree<T>();
}
```

The second function is the insert function. This function takes in a T type element as a parameter and returns a Boolean. This function starts by calling the BST insert with the element parameter. Since the BST insert returns a Boolean, if its result is true, then the balance path function is called. Then the AVL insert itself will return a Boolean.

```
template<typename T>
bool AVLTree<T>::insert(T theElement)
{
    bool result = BSTree<T>::insert(theElement);
    if(result == true)
    {
        balancePath(theElement);
    }
    return result;
}
```

The third function is a simple function that returns the height of the given AVL tree node. It takes in a pointer to an AVL tree node and returns the height as an int value.

```
template<typename T>
int AVLTree<T>::getHeight(AVLTreeNode<T>* theNode)
```

```
{
    return theNode->height;
}
```

The fourth function is a create new node function that takes in a T type element and returns a pointer to an AVL tree node on the heap. This function is not actually called anywhere in the class because of a last-minute implementation change. It could be implemented rather easily though.

```
template<typename T>
AVLTreeNode<T>* AVLTree<T>::createNewNode(T theElement)
{
    AVLTreeNode<T>* newNode = new AVLTreeNode<T>(theElement);
    return newNode;
}
```

The fifth function is the path function, that takes in a T type element and returns a vector of pointers to AVL tree nodes. The function creates a pointer to an AVL tree node called current, and sets it equal to the root, and creates a vector of pointers to AVL tree nodes. It then goes through a while loop until current is equal to NULL. Inside the loop, current is added to the vector, then if the element parameter is less than the element that current is pointing to, current is set equal to it's left child. Otherwise, if opposite is true, then current will be set to its right child. Once neither of those cases is true, the loop breaks, and the vector is returned.

```
template<typename T>
vector<AVLTreeNode<T>*> AVLTree<T>::path(T theElement)
{
    AVLTreeNode<T>* current = static_cast<AVLTreeNode<T>*>(this->root);
    vector<AVLTreeNode<T>*> treeVector;
    while(current != NULL)
    {
        treeVector.push_back(current);
        if(theElement < current->element)
        {
            current = static_cast<AVLTreeNode<T>*>(current->left);
        }
        else if(theElement > current->element)
        {
            current = static_cast<AVLTreeNode<T>*>(current->right);
        }
        else
            break;
    }
    return treeVector;
}
```

The sixth function is the balance path function. This function takes in a T type element as a parameter and does not return anything. First the function creates a vector of pointers to AVL tree nodes and sets it equal to the result of the path function called with the element. Then it creates a pointer to an AVL tree node called temp, and an iterator that points to the end of the vector we just created. It then makes a loop that goes from the end of the path to the beginning.

Inside the loop, the temp variable is set equal to the last pointer in the array. Then there are four possible cases. The first is that the balance factor of the temp variable is less than or equal to -2, and its left child's balance factor is 0 or -1, in which case it will call the balance left left function. The second case is that the balance factor of the temp variable is greater than or equal to 2, and its right child's balance factor is equal to 0 or 1, in which case it will call the balance right right function. The third case is when the balance factor of the temp variable is less than or equal to -2, and its left child's balance factor is equal to 0 or 1, in which case it will call the balance left right function. The fourth and final case is that the balance factor of the temp node is greater than or equal to 2, and its right child's balance factor is equal to 0 or -1, in which case it will call the balance right left function.

```
template<typename T>
void AVLTree<T>::balancePath(T theElement)
{
    vector<AVLTreeNode<T>*> treePath = path(theElement);
    AVLTreeNode<T>* temp;
    typename vector<AVLTreeNode<T>*>::iterator it = treePath.end() - 1;
    for(; it >= treePath.begin(); it--)
    {
        temp = *it;
        if(balanceFactor(temp) <= -2 &&
          (balanceFactor(static_cast<AVLTreeNode<T>*>(temp->left)) == 0 ||
           balanceFactor(static_cast<AVLTreeNode<T>*>(temp->left)) == -1))
        {
            balanceLL(temp, *(it - 1));
        }
        else if(balanceFactor(temp) >= 2 &&
                (balanceFactor(static_cast<AVLTreeNode<T>*>(temp->right)) == 0 ||
                 balanceFactor(static_cast<AVLTreeNode<T>*>(temp->right)) == 1))
        {
            balanceRR(temp, *(it - 1));
        }
        else if(balanceFactor(temp) <= -2 &&
                (balanceFactor(static_cast<AVLTreeNode<T>*>(temp->left)) == 0 ||
                 balanceFactor(static_cast<AVLTreeNode<T>*>(temp->left)) == 1))
        {
```

```
                balanceLR(temp, *(it - 1));
            }
            else if(balanceFactor(temp) >= 2 &&
                   (balanceFactor(static_cast<AVLTreeNode<T>*>(temp->right)) == 0 ||
                    balanceFactor(static_cast<AVLTreeNode<T>*>(temp->right)) == -1))
            {
                balanceRL(temp, *(it - 1));
            }
        }
    }
```

The seventh function is the balance factor function. This function takes in a pointer to an AVL tree node and returns an int value that represents he balance factor. It starts by making an int called bf, and then proceeds to check four different cases. In the first case, the right and left children of the node are not NULL, in which case it sets the bf variable to the height of the right node minus the height of the left node. In the second case, the left node is NULL, so bf is set to the height of the right node plus 1. In the third case, the right node is NULL, so bf is set to the negative height of the left node minus 1. The last case is that both children are NULL, in which case bf is set to 0. The variable bf is then returned.

```
template<typename T>
int AVLTree<T>::balanceFactor(AVLTreeNode<T>* theNode)
{
    int bf;
    if(theNode->right != NULL && theNode->left != NULL)
    {
        bf = (theNode->right->height) - (theNode->left->height);
    }
    else if(theNode->right != NULL)
    {
        bf = (theNode->right->height) + 1;
    }
    else if(theNode->left != NULL)
    {
        bf = ((theNode->left->height) * -1) - 1;
    }
    else
    {
        bf = 0;
    }
    return bf;
}
```

For the lase four functions, I will only explain two of them because the remaining two are literally mirror images of them. For example, to go from the balance left left function to the balance right right function, one must simply switch all instances of left to right and all instances of right to left.

The eighth function is the balance left left function, that takes in two pointers to AVL tree nodes; a node, called A, and its parent node. The function creates a pointer to an AVL tree node called B and sets it equal to A's left child. Then it checks if A is the root, in which case the root is set to B. Otherwise it checks if A is a left child, in which case A's parent's left child is set to B. Otherwise it sets A's parent's right to B. Then A's left is set to B's right, and B's right is set to A. Then BST's fix height function is called on both A and B.

```
template<typename T>
void AVLTree<T>::balanceLL(AVLTreeNode<T> *A, AVLTreeNode<T> *parentOfA)
{
    AVLTreeNode<T>* B = static_cast<AVLTreeNode<T>*>(A->left);
    if(A == this->root)
    {
        this->root = B;
    }
    else if(parentOfA->left == A)
    {
        parentOfA->left = B;
    }
    else
    {
        parentOfA->right = B;
    }
    A->left = B->right;
    B->right = A;
    BSTree<T>::fixHeight(A);
    BSTree<T>::fixHeight(B);
}
```

The ninth function is the balance left right function that takes in two pointers to AVL tree nodes; a node, called A, and its parent node. The function creates a pointer to an AVL tree node called B and sets it to A's left. Then it checks if A's left's right's left child is not NULL, in which case it makes two more pointers to AVL tree nodes called C and D where C is set to B's right, and D is set to C's left. Then, B's right is set to D, C's left is set to B, and A's left is set to C. Otherwise, A's left is set to B's right, B's right is set to NULL, and A's left's left is set to B. Finally, the function calls the balance left left function using A and A's parent as parameters.

```
template<typename T>
void AVLTree<T>::balanceLR(AVLTreeNode<T>* A, AVLTreeNode<T>* parentOfA)
{
    AVLTreeNode<T>* B = static_cast<AVLTreeNode<T>*>(A->left);
    if(A->left->right->left != NULL)
    {
        AVLTreeNode<T>* C = static_cast<AVLTreeNode<T>*>(B->right);
        AVLTreeNode<T>* D = static_cast<AVLTreeNode<T>*>(C->left);
        B->right = D;
```

```
            C->left = B;
            A->left = C;
        }
        else
        {
            A->left = B->right;
            B->right = NULL;
            A->left->left = B;
        }
        balanceLL(A, parentOfA);
}
```

Work Cited

Peer Collaboration:

I worked with Jacob Malmstadt, Megan T, Walker M, Mike D on this project.

Websites:

http://www.cplusplus.com/

https://stackoverflow.com

http://en.cppreference.com/w/