Gabriel Brehm

CS260

Dr. Breeann Flesch

20 May 2018

Lab 3 Report

For this lab I wrote three different headers to create and use a Binary Search Tree. This report will cover the functions written in the BSTree.h header only, while the other two header files can be viewed with the actual code.

The first two functions are the no argument constructor and the destructor. The constructor sets the root of the tree to NULL and the size to 0. The destructor uses the clear function (seen later) to delete all of the nodes off the heap for memory management.

```
template<class T>
BSTree<T>::BSTree()
{
    root = NULL;
    size = 0;
}

template<class T>
BSTree<T>::~BSTree()
{
    clear();
}
```

These next two functions are the search and search helper functions. The search function takes in an element that the user wants to look for in the tree. It then calls on the search helper function which recursively walks through the tree, starting at the root, looking for the node that contains the element. If the tree is empty, the search helper function will return the node, and the search function will then return false. If the search helper function determines that the root holds the element the user is looking for, then the root is returned, and the search function returns true. Otherwise, it checks the size of element in the current node, and determines whether it should

look at its left child or right child, and call itself on that node. This process is repeated until the whole tree is searched, or the node containing the element is found.

```
template<class T>
bool BSTree<T>::search(T theElement)
{
    TreeNode<T>* tempNode = searchHelper(theElement, root);
    if(tempNode == NULL)
    {
        return false;
    }
    else
    {
        return true;
    }
}

template<class T>
TreeNode<T>* BSTree<T>::searchHelper(T theElement, TreeNode<T>* theNode)
{
    if(theNode == NULL)
    {
        return theNode;
    }
    else if(theElement == theNode->element)
    {
        return theNode;
    }
    else if(theElement < theNode->element)
    {
        return searchHelper(theElement, theNode->left);
    }
    else
    {
        return searchHelper(theElement, theNode->right);
    }
}
```

The next two functions are the insert and insert helper functions. The insert function takes in an element that the user want to insert and sets the root equal to the result of the insert helper function. The insert helper function takes in that same element, and the root itself. If the root is NULL, then the helper function simply returns a pointer to a new node on the heap containing the desired element. Otherwise, it compares the element of the root to the element the user wants to insert. If the element to be inserted is smaller than the root's element, the function is called again with the root's left child as the parameter. Likewise, if it is larger, the function is

recursively called on the root's right child. This process is repeated until the appropriate location to place the new node is found.

```
template<class T>
bool BSTree<T>::insert(T theElement)
{
    root = insertHelper(theElement, root);
    size++;
    return true;
}

template<class T>
TreeNode<T>* BSTree<T>::insertHelper(T theElement, TreeNode<T>* theNode)
{
    if(theNode == NULL)
    {
        return new TreeNode<T>(theElement);
    }
    else if(theElement < theNode->element)
    {
        theNode->left = insertHelper(theElement, theNode->left);
    }
    else
    {
        theNode->right = insertHelper(theElement, theNode->right);
    }
    return theNode;
}
```

These next two functions are the is empty function and the get size function. The is empty function checks to see if the size is 0 and if the root is NULL. If both of those conditions are met, then it is empty, and it returns true, otherwise it returns false. The get size function simply returns the size. I do not think there has ever been a more elegant function than the get size function.

```
template<class T>
bool BSTree<T>::isEmpty() const
{
    if(size == 0 && root == NULL)
        return true;
    else
        return false;
}

template<class T>
int BSTree<T>::getSize() const
{
    return size;
}
```

Next are the clear and clear helper functions. The clear function calls the clear helper function on the root, and then sets the root to NULL and the size to 0. The clear helper function takes in a pointer to a node as a parameter, and then walks through the tree using a post order traversal deleting the nodes off the heap.

```
template<class T>
void BSTree<T>::clear()
{
    clearHelper(root);
    root = NULL;
    size = 0;
}

template<class T>
void BSTree<T>::clearHelper(TreeNode<T>* theNode)
{
    if(theNode != NULL)
    {
        clearHelper(theNode->left);
        clearHelper(theNode->right);
        delete theNode;
    }
}
```

Finally, the last two functions are the begin function and the fix height function. The begin function returns an iterator to the root of the tree. This is so we can easily traverse the tree using the ++ operator if we were to choose to. The fix height function fixes the height of each node, based on its position in the tree. If the node is NULL, its height is 0. If both the left and right children of the node are NULL, then the height is also 0. Otherwise, if the just one of the children nodes is NULL, then the height is equal to the height of the other node plus 1. Then if neither of the children nods are NULL, then the height is equal to the maximum height of the two children plus 1. The function then recursively calls itself on the two children, left then right.

```
template<class T>
Iterator<T> BSTree<T>::begin()
{
    return Iterator<T>(this->root);
}

template<class T>
void BSTree<T>::fixHeight(TreeNode<T> *theNode)
{
    if(theNode == NULL)
    {
        theNode->height = 0;
    }
    else if(theNode->left == NULL && theNode->right == NULL)
    {
        theNode->height = 0;
    }
    else if(theNode->left == NULL)
    {
        theNode->height = theNode->right->height + 1;
    }
    else if(theNode->right == NULL)
    {
        theNode->height = theNode->left->height + 1;
    }
    else if(theNode->right->height > theNode->left->height)
    {
        theNode->height = theNode->right->height + 1;
    }
    else if(theNode->left->height > theNode->right->height)
    {
        theNode->height = theNode->left->height + 1;
    }
    else
    {
        fixHeight(theNode->left);
        fixHeight(theNode->right);
    }
}
```

Work Cited

Peer Collaboration:

I worked with Jacob Malmstadt, Megan T, Walker M, Mike D on this project. The iterator.h header file was written by Stacia Fry.

Websites:

http://www.cplusplus.com/

https://stackoverflow.com

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

Textbooks:

Introduction to Programming with C++