(100)

Name: Clay Sarafin & Taylor Heilman

CS 271 - proj1000

1. Let a, b and c be arbitrary nodes in subtrees in α, β, γ where α is a left subtree of node x, node y is the right child of x, β is a left subtree of y and γ is a right subtree of y. Let a have a depth of d, b have a depth of e and c have a depth of f. After a left rotation has been performed on node x the depths of a and c change while the depth of b stays the same. α remains the left the subtree of x, but node x is now the left child of y, therefore the depth of a increases by one, $d + 1$. γ remains the right subtree of y but y is now the parent of x instead of being the child of x, so the depth of c decreases by one, $f - 1$. β changes from the left subtree of y to the right subtree of x, hence the depth of b doesn't change, e.
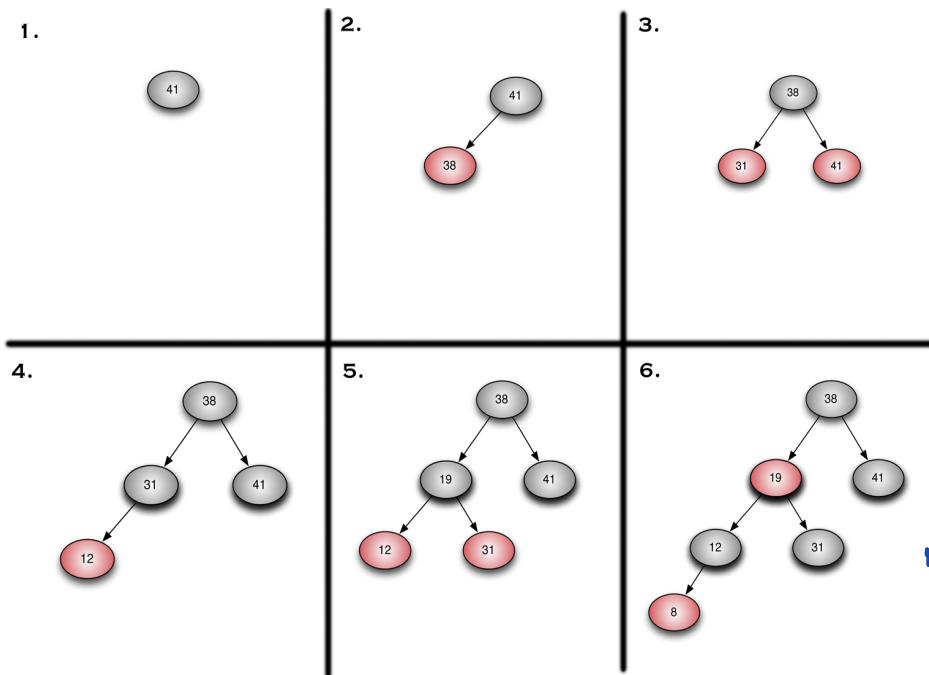
2.

3. Suppose we have a non empty Red Black Tree. By the fifth property of a Red Black Tree we know for every node x, every path to a descendent leaf contains $bh(x)$, the black height of x, black nodes. Therefore the shortest path from x to a leaf would be at least of length $bh(x)$, where the path contains at least zero red nodes. According to the fourth property of a Red Black Tree we know that if a red node has children both must be black nodes, hence red nodes can not be children of another red node.

Therefore the longest path from x to a leaf would also contain $bh(x)$ black nodes. At most, every other node in the longest path is red because the path alternates between red and black nodes, so the length of the longest path is at most $2(bh(x))$. Hence the longest path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest path from x to a descendant leaf.

4. See Attached

5. See Attached

6. We compared the insert time of our Binary Search tree, Hash Table and Red Black Tree. On average the Red Black Tree inserted the entire movie library in 0.2182 seconds. The Hash Table was a close second with an average time of 0.2236 and the Binary Search Tree took an average 3.076 seconds. The insert time for the Binary Search Tree was much longer than the Red Black Tree and Hash Table due to the ordering of the movie files. Since the movies in the movie library were in alphabetical order, ever subsequent movie added into the BST was added to the right child due to how the Binary Search tree determined the placement of strings (the movie titles). Since every node is connected to one other node the Binary Search Tree was essentially a linked list, therefore the insert time was close to $O(n^2)$. The running time to insert into the Hash Table is largely affected by the hash function. A poor hash function may cause a large amount of movies to be chained together, making the running time of inserting a movie close to $O(n)$. Our hash function was fairly effecient which allowed for a running time slightly larger than $O(1)$ The Red Black Tree had the fastest running time due to its characteristics which helps make the tree more complete and doesn't allow the problem that the Binary Search Tree encountered with constanly inserting to the right. Therefore the insert time for our Red Black Tree was $O(logn)$.

	Red Black Tree	Hash Table	Binary Search Tree
	0.174	0.192	3.087
	0.224	0.225	3.029
	0.253	0.249	3.114
	0.242	0.238	3.132
	0.198	0.214	3.018
Average	0.2182	0.2236	3.076

```cpp
// rb.h
// Red/Black Tree header file
// Clay Sarafin & Taylor Heilman


#ifndef RB_H
#define RB_H


template <class T>
class RBNode{
public:
        RBNode();
        RBNode(T* initValue);
        RBNode(char c);

        T* value;
        char color;

        RBNode<T>* parent;
        RBNode<T>* left;
        RBNode<T>* right;
};

template <class T>
class RBTree{
protected:
        int count;
        RBNode<T> *root;
        static RBNode<T> *nil;

        void createNode(RBNode<T>* node);

        RBNode<T>* copy(RBNode<T> *node, RBNode<T> * newP);
        void dealloc(RBNode<T> *ptr);

        RBNode<T> *getNode(RBNode<T> *ptr, const T& z);

        T *maximum_private(RBNode<T>* ptr);
        T *minimum_private(RBNode<T>* ptr);

        std::string inOrder_private(RBNode<T> *ptr, std::string str);
        std::string preOrder_private(RBNode<T> *ptr, std::string str);
        std::string postOrder_private(RBNode<T> *ptr, std::string str);


public:
        RBTree();
        RBTree(const RBTree<T>& rb);
        ˜RBTree();

        RBTree<T>& operator=(const RBTree<T>& rb);

        bool empty();
        T* get(const T& z);

        void insert(T* z);
        void insertFixUp(RBNode<T>* z);
        void leftRotate(RBNode<T>* z);
        void rightRotate(RBNode<T>* z);


        T* maximum();
```

```
        T* minimum();

        T* successor(const T& z);
        T* predecessor(const T& z);

        std::string inOrder();          // return string of elements from an inorder traversal
        std::string preOrder();         // return string of elements from a preorder traversal
        std::string postOrder();        // return string of elements from a postorder traversa
l

        int getCount();
};

class EmptyError {};
class ExistError {};

#endif

#include "rb.cpp"
```

```cpp
// rb.cpp
// Red/Black Tree implementation
// Clay Sarafin & Taylor Heilman


#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sstream>

using namespace std;
template<class T>
RBNode<T> *RBTree<T>::nil = new RBNode<T>('b');

/*code is essentially a c/p from the BST class,
except with the appropiate names replaced (such as BSTNode -> RBNode)
as well as the appropriate NULL pointers replaced to nil*/


/*=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
                      Node - (DE)CONSTRUCTORS
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+*/
/*--------------------------------------------
 * Default Constructor
 * PreConditions:        n/a
 * PostConditions:       empty Node class
                         all pointers (value, left, right, parent) point nil
--------------------------------------------*/
template<class T>
RBNode<T>::RBNode(){
        value = NULL;
        left = NULL;
        right = NULL;
        parent = NULL;

        color = 'r';
}

/*--------------------------------------------
 * Construct with Item Pointer
 * PreConditions:        pointer to an value of type T
 * PostConditions:       Node class where the value pointer points to the value
                         all other pointers point to nil
--------------------------------------------*/
template<class T>
RBNode<T>::RBNode(T *initValue){
        value = initValue;
        left = NULL;
        right = NULL;
        parent = NULL;

        color = 'r';
}

/*--------------------------------------------
 * Construct with color
 * PreConditions:        char 'r' for red, 'b' for black
 * PostConditions:       Node class where the node has color c
--------------------------------------------*/
template<class T>
RBNode<T>::RBNode(char c){
        value = NULL;
        left = NULL;
```

```
        right = NULL;
        parent = NULL;

        color = c;
}

/*=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
                        RBTree - (DE)CONSTRUCTORS
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+*/
/*---------------------------------------------
 * Default Constructor
 * PreConditions:      n/a
 * PostConditions:     empty RBTree class created, where root points to nil
---------------------------------------------*/
template <class T>
RBTree<T>::RBTree(){
        count = 0;
        root = nil;
}


/*---------------------------------------------
 * Copy Constructor, operator=
 * PreConditions:      a RBTree
 * PostConditions:     RBTree is copied over to a new RBTree, and is in the exact same form
---------------------------------------------*/
template <class T>
RBTree<T>::RBTree(const RBTree<T>& rb){
        root = nil;
        root = copy(rb.root, nil);
}
template <class T>
RBTree<T>& RBTree<T>::operator=(const RBTree<T>& rb){
        dealloc();

        root = copy(rb.root, nil);

        return *this;
}

template <class T>
RBNode<T>* RBTree<T>::copy(RBNode<T> *node, RBNode<T> *newP){
        //will traverse through the RBTree to be copied w/ preOrder
        //instead of printing, the new node will be created and will
        //be assigned to the parent in the parameter
        if (node == nil)
                return nil;
        RBNode<T> *newNode = new RBNode<T>(node->value);
        newNode->color = node->color;
        newNode->parent = newP;
        newNode->left = copy(node->left, newNode);
        newNode->right = copy(node->right, newNode);

        return newNode;
}


/*---------------------------------------------
 * Deconstructor
 * PreConditions:      n/a
 * PostConditions:     all nodes that exists in the RBTree are deleted
---------------------------------------------*/
template <class T>
```

```cpp
RBTree<T>::~RBTree(){
        dealloc(root);
}

template <class T>
void RBTree<T>::dealloc(RBNode<T> *ptr){
        if (ptr == nil)
                return;
        dealloc(ptr->left);
        dealloc(ptr->right);
        delete ptr;

        return;
}




/*=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
                        RBTree - FUNCTIONS
=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+*/
/*---------------------------------------------
 * createNode()
 * PreConditions:      a preexisting node
 * PostConditions:     pointers in the NULL that need to assigned to nil will be assigned to
nil
---------------------------------------------*/
template <class T>
void RBTree<T>::createNode(RBNode<T>* node){
        node->parent = nil;
        node->left = nil;
        node->right = nil;
}



/*---------------------------------------------
 * empty()
 * PreConditions:      n/a
 * PostConditions:     returns true if empty, false if not
---------------------------------------------*/
template <class T>
bool    RBTree<T>::empty(){
        return (count == 0);
        //alt approach: return (root == nil);
}

/*---------------------------------------------
 * get()
 * PreConditions:        tree is not empty
                        value of type T has the operator '==' and '<' overloaded
                        user has never modified what the value pointer points to in insert()
 * PostConditions:      returns a pointer to the value if it exists
                        returns nil if it does not
---------------------------------------------*/
template <class T>
T*      RBTree<T>::get(const T& z){
        if (empty())
                throw EmptyError();
        RBNode<T> *node = getNode(root, z);
        //node = nil when RBTree is empty, or if value doesn't exist
        //already covered the empty part, so logically the node doesn't exist
        if (node == nil)
                return NULL;            //does this matter now?
```

```
        return node->value;
}

template <class T>
RBNode<T>*      RBTree<T>::getNode(RBNode<T> *ptr, const T& z){
        if (ptr == nil)                         //BC: value not found, or tree is empty
                return nil;
        if (*(ptr->value) == z)                 //found value
                return ptr;
        if(z < *(ptr->value))                   //traverse tree to find value
                return getNode(ptr->left,z);
        else
                return getNode(ptr->right,z);

}

/*---------------------------------------------
 * insert()
 * PreConditions:       value of type T has the operator '<' overloaded
 * PostConditions:      value z is inserted into the tree
                        tree is still a valid RBTree
---------------------------------------------*/
template <class T>
void    RBTree<T>::insert(T *z){
        RBNode<T>* y = nil;
        RBNode<T>* x = root;

        RBNode<T>* newNode = new RBNode<T>(z);
        createNode(newNode);

        while (x != nil){ //traverse through tree to find appropriate place to insert node
                y = x;
                if (*(newNode->value) < *(x->value))
                        x = x->left;
                else
                        x = x->right;
        }

        newNode->parent = y;
        if (y == nil)                                   //tree is empty, make root point to ne
w node
                root = newNode;
        else if (*(newNode->value) < *(y->value))       //make appropriate subchild point to i
nserted node
                y->left = newNode;
        else
                y->right = newNode;

        insertFixUp(newNode);                           //call function to make the RBTree val
id again
        count++;
}

/*---------------------------------------------
 * insertFixUp()
 * PreConditions:       pointer to node inserted into the tree
 * PostConditions:      tree will be fixed to be a valid RBTree
---------------------------------------------*/
template <class T>
void RBTree<T>::insertFixUp(RBNode<T>* z){
        RBNode<T>* y;

        while (z->parent->color == 'r'){
```

```cpp
                if (z->parent == z->parent->parent->left){
                        y = z->parent->parent->right;
                        if (y->color == 'r'){                      //only need to change colors
                                z->parent->color = 'b';
                                y->color = 'b';
                                z->parent->parent->color = 'r';
                                z = z->parent->parent;
                        }
                        else{
                                if (z == z->parent->right){      //some rotations necessary
                                        z = z->parent;
                                        leftRotate(z);
                                }
                                z->parent->color = 'b';
                                z->parent->parent->color = 'r';
                                rightRotate(z->parent->parent);
                        }
                }
                else{
                        y = z->parent->parent->left;
                        if (y->color == 'r'){                      //only have to change colors
                                z->parent->color = 'b';
                                y->color = 'b';
                                z->parent->parent->color = 'r';
                                z = z->parent->parent;
                        }
                        else{
                                if (z == z->parent->left){      //some rotations necessary
                                        z = z->parent;
                                        rightRotate(z);
                                }
                                z->parent->color = 'b';
                                z->parent->parent->color = 'r';
                                leftRotate(z->parent->parent);
                        }
                }
        }

        root->color = 'b';                                         //make root black in case it w
as changed to red
}

/*---------------------------------------------
 * leftRotate()
 * PreConditions:      pointer to node in the RBTree
 * PostConditions:     nodes in the tree will be rotated to the left such that
                       (with y being the right child of z):
                               left child of z is not modified
                               right child of y is not modified
                               z's right child is the left subtree of y
                               y's left child is z
---------------------------------------------*/
template <class T>
void RBTree<T>::leftRotate(RBNode<T>* z){
        RBNode<T>* child = z->right;

        z->right = child->left;
        if(child->left != nil)
                child->left->parent = z;
        child->parent = z->parent;
        if(z->parent == nil)
                root = child;
        else if (z == z->parent->left)
```

```
                z->parent->left = child;
        else
                z->parent->right = child;
        child->left = z;
        z->parent = child;
}

/*---------------------------------------------
 * rightRotate()
 * PreConditions:        pointer to node in RBTree
 * PostConditions:       nodes in the tree will be rotated to the right such that
                         (with y being the left child of z):
                                 right child is not modified
                                 left child of y is not modified
                                 z's left child is the right subtree of y
                                 y's right child is z
---------------------------------------------*/
template <class T>
void RBTree<T>::rightRotate(RBNode<T>* z){
        //code from leftRotate(), except left <-> right
        RBNode<T>* child = z->left;
        z->left = child->right;
        if(child->right != nil)
                child->right->parent = z;
        child->parent = z->parent;
        if(z->parent == nil)
                root = child;
        else if (z == z->parent->right)
                z->parent->right = child;
        else
                z->parent->left = child;
        child->right = z;
        z->parent = child;
}


/*---------------------------------------------
 * maximum()
 * PreConditions:        tree is not empty
                         user has never modified what the value pointer points to in insert()
 * PostConditions:       returns pointer to the maximum value in the tree
---------------------------------------------*/
template <class T>
T* RBTree<T>::maximum(){
        return maximum_private(root);
}

template <class T>
T* RBTree<T>::maximum_private(RBNode<T>* ptr){
        if (ptr == nil)//nothing exists in RBTree
                throw EmptyError();

        if (ptr->right == nil)
                return ptr->value;
        else
                return maximum_private(ptr->right);
}

/*---------------------------------------------
 * minimum()
 * PreConditions:        tree is not empty
                         user has never modified what the value pointer points to in insert()
 * PostConditions:       returns pointer to the minimum value in the tree
```

```
-------------------------------------------*/
template <class T>
T* RBTree<T>::minimum(){
        return minimum_private(root);
}

template <class T>
T* RBTree<T>::minimum_private(RBNode<T>* ptr){
        if (ptr == nil)//nothing exists in RBTree
                throw EmptyError();

        if (ptr->left == nil)
                return ptr->value;
        else
                return minimum_private(ptr->left);
}

/*-------------------------------------------
 * sucessor()
 * PreConditions:        T z exists in RBTree
                         successor of z exists in the RBTree
                         user has never modified what the value pointer points to in insert()
 * PostConditions:       pointer to successor of z is returned
-------------------------------------------*/
template <class T>
T* RBTree<T>::successor(const T& z){
        RBNode<T> *ptr0 = getNode(root,z);
        if (ptr0 == nil)
                throw ExistError();
        RBNode<T> *ptr1;
        if (ptr0->right != nil)
                return minimum_private(ptr0->right);
        ptr1 = ptr0->parent;
        while((ptr1 != nil) && (ptr0 == ptr1->right)){
                ptr0 = ptr1;
                ptr1 = ptr1->parent;
        }

        if (ptr1 == nil)                    //successor does not exist
                throw ExistError();

        return ptr1->value;
}

/*-------------------------------------------
 * predecessor()
 * PreConditions:        T z exists in RBTree
                         precedessor of z exists in the RBTree
                         user has never modified what the value pointer points to in insert()
 * PostConditions:       pointer to predecessor of z is returned
-------------------------------------------*/
template <class T>
T* RBTree<T>::predecessor(const T& z){
        RBNode<T> *ptr0 = getNode(root,z);
        if (ptr0 == nil)
                throw ExistError();
        RBNode<T> *ptr1;
        if (ptr0->left != nil)
                return maximum_private(ptr0->left);
        ptr1 = ptr0->parent;
        while((ptr1 != nil) && (ptr0 == ptr1->left)){
                ptr0 = ptr1;
                ptr1 = ptr1->parent;
```

```cpp
        }

        if (ptr1 == nil)                      //predecessor does not exist
                throw ExistError();

        return ptr1->value;
}

/*----------------------------------------------
 * inOrder(), preOrder(), and postOrder()
 * PreConditions:        n/a
 * PostConditions:       inOrder  : string representation of the values in the RBTree are in or
der
                         preOrder : string represenations of the values in RBTree are shown
                                    in pre order (starts with root, ends with max value)
                         postOrder: string represenations of the values in RBTree are shown
                                    in post order (starts with min value, ends with root)
                         all      : each value is followed by ", "
                                    : user has never modified what the value pointer points to in
 insert()
----------------------------------------------*/
template <class T>
std::string RBTree<T>::inOrder(){
        std::string str;
        str = inOrder_private(root,str);
        if (str == "")          //want to avoid using pop_back() on an empty string
                return str;
        else{
                str.pop_back();
                str.pop_back();
                return str;
        }
}
template <class T>
std::string RBTree<T>::inOrder_private(RBNode<T> *ptr, std::string str){
        if (ptr == nil){
                return str;
        }
        std::string str_null;   //null dummy string to help with recursion;
                                //if you pass in str as the parameter again, then
                                //it will append to that string in the funcion, then return th
at string
                                //it will then append it to the string in the initial call, wh
ich is something
                                //not desired

        //left, print, right
        str += inOrder_private(ptr->left, str_null);
        str += std::to_string(*(ptr->value));
        str += "-";
        str += ptr->color;
        str += ", ";
        str += inOrder_private(ptr->right, str_null);

        return str;
}


template <class T>
std::string RBTree<T>::preOrder(){
        std::string str;
        str = preOrder_private(root,str);
        if (str == "")
```

```cpp
                return str;
        else{
                str.pop_back();
                str.pop_back();
                return str;
        }
}
template <class T>
std::string RBTree<T>::preOrder_private(RBNode<T> *ptr, std::string str){
        if (ptr == nil){
                return str;
        }
        std::string str_null;

        //print, left, right
        str += std::to_string(*(ptr->value));
        str += "-";
        str += ptr->color;
        str += ", ";
        str += preOrder_private(ptr->left, str_null);
        str += preOrder_private(ptr->right, str_null);

        return str;
}

template <class T>
std::string RBTree<T>::postOrder(){
        std::string str;
        str = postOrder_private(root,str);
        if (str == "")
                return str;
        else{
                str.pop_back();
                str.pop_back();
                return str;
        }
}
template <class T>
std::string RBTree<T>::postOrder_private(RBNode<T> *ptr, std::string str){
        if (ptr == nil){
                return str;
        }
        std::string str_null;

        //left, right, print
        str += postOrder_private(ptr->left, str_null);
        str += postOrder_private(ptr->right, str_null);
        str += std::to_string(*(ptr->value));
        str += "-";
        str += ptr->color;
        str += ", ";

        return str;
}


/*---------------------------------------------
 * getCount()
 * PreConditions:        n/a
 * PostConditions:       number of nodes that exists in the RBTree is returned
---------------------------------------------*/
template <class T>
```

```cpp
int RBTree<T>::getCount(){
       return count;
}
```

```cpp
// test_rb.cpp
// Red/Black Tree tests
// Clay Sarafin & Taylor Heilman

#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

#include "rb.h"

using namespace std;

int *zero  = new int(0);
int *one   = new int(1);
int *two   = new int(2);
int *three = new int(3);
int *four  = new int(4);
int *five  = new int(5);
int *six   = new int(6);
int *seven = new int(7);
int *eight = new int(8);
int *nine  = new int(9);

void insertTest(){
        RBTree<int> tree;

        tree.insert(five);
        tree.insert(zero);
        assert(tree.getCount() == 2);
        assert(tree.preOrder() == "5-b, 0-r");  //tests for empty tree, left subtree
        tree.insert(nine);
        assert(tree.preOrder() == "5-b, 0-r, 9-r");     //test for right subtree;
        //add everything else
        tree.insert(one);
        tree.insert(two);
        tree.insert(three);
        tree.insert(four);
        tree.insert(six);
        tree.insert(seven);
        tree.insert(eight);
        assert(tree.preOrder() == "5-b, 1-r, 0-b, 3-b, 2-r, 4-r, 7-r, 6-b, 9-b, 8-r");
}

void getTest(){
        RBTree<int> tree;

        try{
                tree.get(*one);
                assert(false);
        }
        catch(EmptyError){
        }

        tree.insert(one);
        tree.insert(two);
        tree.insert(three);
        tree.insert(four);
        tree.insert(six);
        tree.insert(seven);
        tree.insert(eight);
```

```
        //test for root
        assert(tree.get(*one) == one);
        //test for non-leaf
        assert(tree.get(*two) == two);
        //test for leaf
        assert(tree.get(*eight) == eight);        //not sure what's going on here

        //test for nonexisting node in tree;
        assert(tree.get(*five) == NULL);
}

void maxMinTest(){
        RBTree<int> tree;

        //test for EmptyError exceptions
        try{
                tree.minimum();
                assert(false);
        }
        catch(EmptyError exception){
        }
        try{
                tree.maximum();
                assert(false);
        }
        catch(EmptyError exception){
        }

        tree.insert(zero);
        tree.insert(one);
        tree.insert(two);

        //test min & max
        assert(tree.maximum() == two);
        assert(tree.minimum() == zero);
}

void predSuccTest(){
        RBTree<int> tree;

        //test ExistError exceptions
        try{
                tree.predecessor(0);
                assert(false);
        }
        catch(ExistError exception){
        }
        try{
                tree.successor(0);
                assert(false);
        }
        catch(ExistError exception){
        }

        tree.insert(four);
        tree.insert(two);
        tree.insert(three);
        tree.insert(one);
        tree.insert(seven);
        tree.insert(five);
        tree.insert(nine);
```

```cpp
        assert(tree.predecessor(2) == one);
        assert(tree.successor(2) == three);

        assert(tree.predecessor(4) == three);
        assert(tree.successor(4) == five);

        assert(tree.predecessor(9) == seven);
        //test EmptyError exception for an item that doesn't exist
        try{
                tree.successor(9);;
                assert(false);
        }
        catch(ExistError exception){
        }

        assert(tree.predecessor(3) == two);
        assert(tree.successor(3) == four);

}

void printTest(){
        RBTree<int> tree;
        assert(tree.inOrder() == "");
        assert(tree.preOrder() == "");
        assert(tree.postOrder() == "");

        tree.insert(zero);
        tree.insert(one);
        tree.insert(two);

        assert(tree.inOrder() == "0-r, 1-b, 2-r");
        assert(tree.preOrder() == "1-b, 0-r, 2-r");
        assert(tree.postOrder() == "0-r, 2-r, 1-b");

        RBTree<int> tree2;

        tree2.insert(four);
        tree2.insert(two);
        tree2.insert(three);
        tree2.insert(one);
        tree2.insert(seven);
        tree2.insert(five);
        tree2.insert(nine);
        /*
                        3-b
                      /       \
            2-b               5-r
           /                 /       \
        1-r               4-b         7-b
                                          \
                                           9-r
        */
        assert(tree2.inOrder() == "1-r, 2-b, 3-b, 4-b, 5-r, 7-b, 9-r");
        assert(tree2.preOrder() == "3-b, 2-b, 1-r, 5-r, 4-b, 7-b, 9-r");
        assert(tree2.postOrder() == "1-r, 2-b, 4-b, 9-r, 7-b, 5-r, 3-b");

}

void copyTest(){
        RBTree<int> tree0;

        tree0.insert(four);
        tree0.insert(two);
```

```cpp
        tree0.insert(three);
        tree0.insert(one);
        tree0.insert(seven);
        tree0.insert(five);
        tree0.insert(nine);

        RBTree<int> tree1(tree0);

        assert(tree1.inOrder() == "1-r, 2-b, 3-b, 4-b, 5-r, 7-b, 9-r");
        assert(tree1.preOrder() == "3-b, 2-b, 1-r, 5-r, 4-b, 7-b, 9-r");
        assert(tree1.postOrder() == "1-r, 2-b, 4-b, 9-r, 7-b, 5-r, 3-b");
}

int main(){
        insertTest();
        getTest();
        maxMinTest();
        predSuccTest();
        printTest();
        copyTest();

        delete zero;
        delete one;
        delete two;
        delete three;
        delete four;
        delete five;
        delete six;
        delete seven;
        delete eight;
        delete nine;
        return 0;
}
```

```cpp
// dict.h
// header file for dictoinary class
// Clay Sarafin & Taylor Heilman

#ifndef DICT_H
#define DICT_H

#include <iostream>

#include "rb.h"

using namespace std;

template <class KeyType>
class Dictionary : public RBTree<KeyType>{
        public:
                Dictionary() : RBTree<KeyType>() {};

                using RBTree<KeyType>::get;
                using RBTree<KeyType>::insert;
                using RBTree<KeyType>::count;
                using RBTree<KeyType>::inOrder;
                using RBTree<KeyType>::empty;


                std::string toString();
};

#include "dict.cpp"

#endif
```

```cpp
// dict.cpp
//
// Clay Sarafin & Taylor Heilman

#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <string>


using namespace std;


/*-------------------------------------------
 * toString()
 * PreConditions:        inOrder from BST class does not have brackets, every entry is followed
 by ", "
                            and does not end with std::endl
                    type KeyType's operator<< is overloaded, and is formatted as "key:valu
e"
 * PostConditions:       string representation of the entire dictionary is printed out
-------------------------------------------*/

template <class KeyType>
std::string Dictionary<KeyType>::toString(){
        std::string str;
        str += "{";
        str += inOrder();
        str += "}";

        return str;
}
```