makefile

```
###############################################################
# Program:
#     Lesson 09, Map
#     Brother Helfrich, CS265
# Author:
#     David Lambertson and Derek Calkins
# Summary:
#     This program allows users to create Maps and use them to
#     save keys with values and access those values later through
#     the square bracket operator.
# Time:
#     David: 50% & Derek 50%
###############################################################

##############################################################
# The main rule
##############################################################
a.out: lesson09.o wordCount.o
	g++ -g -o a.out lesson09.o wordCount.o
	tar -cf lesson09.tar *.h *.cpp makefile

##############################################################
# The individual components
#      lesson09.o     : the driver program
#      wordCount.o    : the wordCount() function
##############################################################
lesson09.o: bnode.h bst.h pair.h map.h lesson09.cpp
	g++ -g -c lesson09.cpp

wordCount.o: map.h wordCount.h wordCount.cpp bnode.h bst.h
	g++ -g -c wordCount.cpp
```

map.h

```
/***********************************************************************
* Program:
*    Lesson 09, Map and balanced BSTs
*    Brother Helfrich, CS 235
* Author:
*    David Lambertson and Derek Calkins
* Summary:
*    this is a Abstract Data Type : Map
*    It allows a user to create their own maps and use the square
*    bracket operator to save new keys or access old keys
************************************************************************/
#ifndef MAP_H
#define MAP_H

#include "bst.h"
#include "pair.h"

template <class T, class U>
class MapIterator;

/*******************************************
 * All the private and public member variables and
 * methods for which a Map needs to work how we desire.
 ******************************************/
template <class T, class U>
class Map
{
  public:
   //default constructor
  Map(): tree(), numItems(0) {}
   //copy constructor
  Map(Map<T, U> & rhs) : tree(), numItems() { this = rhs; }
   //destructor
   ~Map() { clear(); }
```

Commented [HJ1]: What dues ‘U’ stand for. The most common names for the templates are ‘T1’ and ‘T2’ when there are two data-types used. For the Map, however, we commonly use ‘K’ for key and ‘V’ for value.

Commented [HJ2]: Should be const.

```
   //Our simple empty, size and clear functions.
   bool empty() {return (numItems == 0); }
   int size() { return numItems; }
   void clear()
   {
      tree.clear();
      numItems = 0;
   }

   /*********************************
    * assignment operator to copy a Map.
    ********************************/
   Map<T, U> & operator =(Map<T, U> & oldMap)
   {
      this->tree = oldMap.tree;
      this->numItems = oldMap.numItems;
      return *this;
   }

   //square bracket operator, see below.
   U & operator [](const T & key);

   //begin, end, rbegin, rend functions returning Iterators
   MapIterator<T, U> begin() { return MapIterator<T, U>(tree.begin());  }
   MapIterator<T, U> end()   { return MapIterator<T, U>(tree.end());    }
   MapIterator<T, U> rbegin(){ return MapIterator<T, U>(tree.rbegin()); }
   MapIterator<T, U> rend()  { return MapIterator<T, U>(tree.rend());   }

   //find function which calls our BST find function
   MapIterator<T, U> find(const T & value)
   {
      Pair<T , U> temp;
      temp.first = value;
      return MapIterator<T, U>(tree.find(temp));
   }

   //allows us to access which ever node we desire to find.
   BSTIterator<Pair<T, U> > getNode(T value)
   {
      Pair<T, U> temp;
      temp.first = value;
      return tree.find(temp);
   }


  private:
   BST<Pair<T, U> > tree;
   int numItems;
};

/**********************************************
 * Square Bracket Operator
 * allows users to either save a new key within
 * the map or display our value for the key given to us.
 *********************************************/
template <class T, class U>
U & Map<T, U> :: operator [](const T & key)
{
   Pair<T, U> temp;
   temp.first = key;
   if (tree.find(temp) == NULL)
   {
      tree.insert(temp);
      numItems++;
   }
   BSTIterator<Pair<T ,U> > it = tree.find(temp);
   return (*it).second;
}



/************************************************
 * This is the class definition for the Binary
 * Search Tree Iterator.
 *********************************************/
template <class T, class U>
class MapIterator
{
```

Commented [HJ3]: Should be const.

Commented [HJ4]: Good.

Commented [HJ5]: perfect

Commented [HJ6]: well done.

Commented [HJ7]: Find is called twice on the common case when we found it the first time. Can you think of a design that only calls it once?

```
  public:
   //default constructor
  MapIterator() : it(NULL) {}

   //non-default constructor
   MapIterator(BSTIterator<Pair<T, U> > p) { it = p; }

   //assignment operator
   MapIterator<T, U> operator =(const MapIterator<T, U> & rhs)
   {
      this->it = rhs.it;
      return *this;
   }

   //are they equal?
   bool operator ==(const MapIterator<T, U> & rhs)
   { return (this->it == rhs.it); }

   //are they not equal?
   bool operator !=(const MapIterator<T, U> & rhs)
   { return (this->it != rhs.it); }

   //dereference operator
   U & operator *() { return (*it).second; }

   //overloaded operators, ++ and --
   MapIterator<T, U> operator ++() { ++it; return *this; }
   MapIterator<T, U> operator --() { --it; return *this; }

  private:
   //so we have access to the BST
   BSTIterator<Pair<T, U> > it;

};

#endif // MAP_H
```

Commented [HJ8]: perfect

wordCount.h

```
/***********************************************************************
* Header:
*    WORD COUNT
* Summary:
*    This will contain just the prototype for the wordCount()
*    function
* Author
*    Derek Calkins and David Lambertson
************************************************************************/

#ifndef WORD_COUNT_H
#define WORD_COUNT_H

/*******************************************
 * this is a dummy int class so that we can
 * use a default constructor of 0.
 ******************************************/
class Count
{
  public:
   //default constructor
  Count(): count(0) {}
   //non-default constructor
  Count(const int & count) : count(count) {}
   //returns the number for which we want
   int getCount() const { return count; }
   //++ overloaded to increment
   Count operator ++ ()
   {
      ++count;
      return *this;
   }

   //assignment operator overloaded to let us save a new number
   Count operator = (const int & rhs)
   {
      count = rhs;
      return *this;
   }
```

Commented [HJ9]: well done.

```
  private:
   int count;

};

/****************************************************
 * WORD COUNT
 * Prompt the user for a file to read, then prompt the
 * user for words to get the count from
 ****************************************************/
void wordCount();

#endif // WORD_COUNT_H
```

bst.h

```
/*******************************
 * Module:
 *    Lesson 08, BST
 *    Brother Helfrich, CS 235
 * Author:
 *    David Lambertson and Derek Calkins
 * Summary:
 *    This program contains the necessary
 *    methods for creating a Binary Search
 *    Tree and an iterator for it.
 ****************************/

#ifndef BST_H
#define BST_H

#include "bnode.h"

template<class T>
class BSTIterator;

/***********************************************
 * This is the class definition for our Binary
 * Search Tree.
 ************************************************/
template <class T>
class BST
{
  public:
  BST() : myRoot()  {}

  BST(BinaryNode<T> * root) : myRoot(root) {}

   ~BST() { clear(); }

   //adds the new data in the appropriate place
   void insert(const T & data);

   //assignment operator using copyBinaryTree from BinaryNode class
   BST<T> operator =(BST<T> & rhs)
   {
      BinaryNode<T> * pSrc = rhs.myRoot;
      myRoot->copyBinaryTree(pSrc, myRoot);
   }

   //also a friend of the iterator so we can use it's variable
   void remove(BSTIterator<T> spot);

   //checks to see if we have anything in the tree
   bool empty() const { return (myRoot == NULL); }

   //clears all of the nodes in the tree
   void clear()
   {
      if (myRoot == NULL)
         return;
      deleteBinaryTree(myRoot);
   }

   //allows the user to be able to get the root node
   BinaryNode<T> * getRoot()
   {
```

Commented [HJ10]: Unchanged.

```
      return myRoot;
   }

   //return iterator to data if found
   BSTIterator<T> find(const T & data);

   //iterator to the lowest value node
   BSTIterator<T> begin();
   //iterator to NULL
   BSTIterator<T> end(){ return BSTIterator<T>(NULL); }
   //iterator to the highest value node
   BSTIterator<T> rbegin();
   //iterator to NULL
   BSTIterator<T> rend(){ return BSTIterator<T>(NULL); }

  private:
   BinaryNode<T> * myRoot;
   BinaryNode<T> * findForInsert(BinaryNode<T> * & p, const T & data);

};

/************************************************
 * This is the definition for our insert function
 ***********************************************/
template<class T>
void BST<T> :: insert(const T & data)
{
   BinaryNode<T> * pNew = findForInsert(myRoot, data);
   if (myRoot == NULL)
   {
      myRoot = new BinaryNode<T>(data);
   }
   else if (data > pNew->data)
   {
      pNew->addRight(data);
   }
   else
   {
      pNew->addLeft(data);
   }
}

/**************************************
 * This finds the parent Node of which we should add.
 *************************************/
template <class T>
BinaryNode<T> * BST<T> :: findForInsert(BinaryNode<T> * & p, const T & data)
{
   if (p == NULL)
      return p;
   if (p->data > data)
   {
      if (p->pLeft == NULL)
         return p;
      findForInsert(p->pLeft, data);
   }
   else
   {
      if (p->pRight == NULL)
         return p;
      findForInsert(p->pRight, data);
   }
}

/*****************************************
 * Begin (returns a pointer to lowest)
 ****************************************/
template <class T>
BSTIterator<T> BST<T> :: begin()
{
   BinaryNode<T> * tmp = myRoot;
   if (!tmp)
      return tmp;
   while (tmp->pLeft)
   {
      tmp = tmp->pLeft;
   }
   BSTIterator<T> it = tmp;
   return it;
```

```
}

/****************************************
 * Reverse Begin (returns a pointer to highest)
 ****************************************/
template <class T>
BSTIterator<T> BST<T> :: rbegin()
{
   BinaryNode<T> * tmp = myRoot;
   if (!tmp)
      return tmp;
   while (tmp->pRight)
   {
      tmp = tmp->pRight;
   }
   BSTIterator<T> it = tmp;
   return it;
}

/****************************************
 * FIND
 * Finds if we have the data and returns
 * a pointer to that node
 ****************************************/
template <class T>
BSTIterator<T> BST<T> :: find(const T & data)
{
   BinaryNode<T> * tmp = myRoot;
   while (tmp != NULL)
   if (tmp->data == data)
      return BSTIterator<T>(tmp);
   else if (tmp->data > data)
      tmp = tmp->pLeft;
   else
      tmp = tmp->pRight;

   return end();
}

/***************************************
 * REMOVE
 * removes node at location given.
 **************************************/
template <class T>
void BST<T> :: remove(BSTIterator<T> p)
{
   //if the node is not within the BST
   if (p == end())
      return;

   //if I don't have any children
   if(!p.spot->pLeft && !p.spot->pRight)
   {
      if(p.spot->amIRight())
         p.spot->pParent->pRight = NULL;
      else
         p.spot->pParent->pLeft = NULL;
      //after setting parent to NULL, delete node
      delete p.spot;
   }
   //if I have two children
   else if(p.spot->pLeft && p.spot->pRight)
   {
      //create iterator to point to successor
      BSTIterator<T> it = p;
      //move new iterator to point to successor
      ++it;
      //copy data from successor to node to overwrite
      p.spot->data = it.spot->data;
      //since we know that the successor will have one
      //child or no children, pass back successor node
      //to be able to delete
      remove(it);
   }
   //I have a child
   else
   {
      //do I have a left child?
      if(p.spot->pLeft)
```

```
      {
         p.spot->pLeft->pParent = p.spot->pParent;
         if(p.spot->amIRight())
            p.spot->pParent->pRight = p.spot->pLeft;
         else
            p.spot->pParent->pLeft = p.spot->pLeft;
      }
      //I must have a right child
      else
      {
         p.spot->pRight->pParent = p.spot->pParent;
         if(p.spot->amIRight())
            p.spot->pParent->pRight = p.spot->pRight;
         else
            p.spot->pParent->pLeft = p.spot->pRight;
      }
      //after changing the pointers delete node
      delete p.spot;
   }
}

/************************************************
 * This is the class definition for the Binary
 * Search Tree Iterator.
 ***********************************************/
template <class T>
class BSTIterator
{
  public:
   //default constructor
  BSTIterator() : spot() {}

   //non-default constructor
   BSTIterator(BinaryNode<T> * p)
   {
      if (p == NULL)
         spot = NULL;
      else
         spot = p;
   }

   //assignment operator
   BSTIterator<T> operator =(const BSTIterator<T> & rhs)
   {
      this->spot = rhs.spot;
      return *this;
   }

   //are they equal?
   bool operator ==(const BSTIterator<T> & rhs)
   { return (this->spot == rhs.spot); }

   //are they not equal?
   bool operator !=(const BSTIterator<T> & rhs)
   { return (this->spot != rhs.spot); }

   //dereference operator
   T & operator *() { return spot->data; }

   //overloaded operators
   BSTIterator<T> operator ++();
   BSTIterator<T> operator --();

  private:
   BinaryNode<T> * spot;

   //friend so we can access the iterator
   template <class U>
   friend void BST<U> :: remove(BSTIterator<U> p);
};

/**********************************************
 * Increment Operator
 * Goes to the successor
 *********************************************/
template <class T>
BSTIterator<T> BSTIterator<T> :: operator ++()
{
   // has right child
```

```
   if (spot->pRight != NULL)
   {
      spot = spot->pRight;
      while (spot->pLeft)
         spot = spot->pLeft;
   }

   // has no right child
   else
   {
      while (spot->amIRight())
         spot = spot->pParent;
      if (spot->pParent != NULL)
         spot = spot->pParent;
      else
         spot = NULL;
   }
   return *this;
}

/**********************************************
 * Decrement Operator
 * Goes to the predecessor
 **********************************************/
template <class T>
BSTIterator<T> BSTIterator<T> :: operator --()
{
   // has left child
   if (spot->pLeft != NULL)
   {
      spot = spot->pLeft;
      while (spot->pRight)
         spot = spot->pRight;
   }

   // has no left child
   else
   {
      while (spot->amILeft())
         spot = spot->pParent;
      if (spot->pParent != NULL)
         spot = spot->pParent;
      else
         spot = NULL;
   }
   return *this;
}


#endif // BST_H
```

pair.h

```
/***********************************************************************
 * Module:
 *    Lesson 07, Pair
 *    Brother Helfrich, CS 235
 * Author:
 *    Br. Helfrich
 * Summary:
 *    This program will implement a pair: two values
 ************************************************************************/

#ifndef PAIR_H
#define PAIR_H

#include <iostream>  // for ISTREAM and OSTREAM

/**********************************************
 * PAIR
 * This class couples together a pair of values, which may be of
 * different types (T1 and T2). The individual values can be
 * accessed through its public members first and second.
 *
 * Additionally, when compairing two pairs, only T1 is compared. This
 * is a key in a name-value pair.
 ***********************************************/
template <class T1, class T2>
```

Commented [HJ11]: Unchanged.

```
class Pair
{
public:
   // constructors
   Pair() {}
   Pair(const T1 & first, const T2 & second) : first(first), second(second) {}
   Pair(const Pair <T1, T2> & rhs) : first(rhs.first), second(rhs.second) {}

   // copy the values
   Pair <T1, T2> & operator = (const Pair <T1, T2> & rhs)
   {
      first  = rhs.first;
      second = rhs.second;
      return *this;
   }

   // constant fetchers
   const T1 & getFirst()  const { return first;  }
   const T2 & getSecond() const { return second; }

   // compare Pairs.  Only first will be compared!
   bool operator >  (const Pair & rhs) const { return first >  rhs.first; }
   bool operator >= (const Pair & rhs) const { return first >= rhs.first; }
   bool operator <  (const Pair & rhs) const { return first <  rhs.first; }
   bool operator <= (const Pair & rhs) const { return first <= rhs.first; }
   bool operator == (const Pair & rhs) const { return first == rhs.first; }
   bool operator != (const Pair & rhs) const { return first != rhs.first; }

   // these are public.  We cannot validate!
   T1 first;
   T2 second;
};

/*****************************************************
 * PAIR INSERTION
 * Display a pair for debug purposes
 ****************************************************/
template <class T1, class T2>
inline std::ostream & operator << (std::ostream & out, const Pair <T1, T2> & rhs)
{
   out << '(' << rhs.first << ", " << rhs.second << ')';
   return out;
}

/*****************************************************
 * PAIR EXTRACTION
 * input a pair
 ****************************************************/
template <class T1, class T2>
inline std::istream & operator >> (std::istream & in, Pair <T1, T2> & rhs)
{
   in >> rhs.first >> rhs.second;
   return in;
}

#endif // PAIR_H
```

bnode2.h

Commented [HJ12]: Not used??

```
// you might want to put these methods into your BinaryNode class
// to help you debug your red-black balancing code

/****************************************************
 * BINARY NODE :: FIND DEPTH
 * Find the depth of the black nodes. This is useful for
 * verifying that a given red-black tree is valid
 ****************************************************/
template <class T>
int BinaryNode <T> :: findDepth() const
{
   // if there are no children, the depth is ourselves
   if (pRight == NULL && pLeft == NULL)
      return (isRed ? 0 : 1);

   // if there is a right child, go that way
   if (pRight != NULL)
      return (isRed ? 0 : 1) + pRight->findDepth();
   else
```

```
      return (isRed ? 0 : 1) + pLeft->findDepth();
}

/****************************************************
 * BINARY NODE :: VERIFY RED BLACK
 * Do all four red-black rules work here?
 ***************************************************/
template <class T>
void BinaryNode <T> :: verifyRedBlack(int depth) const
{
   depth -= (isRed == false) ? 1 : 0;

   // Rule a) Every node is either red or black
   assert(isRed == true || isRed == false); // this feels silly

   // Rule b) The root is black
   if (pParent == NULL)
      assert(isRed == false);

   // Rule c) Red nodes have black children
   if (isRed == true)
   {
      if (pLeft != NULL)
         assert(pLeft->isRed == false);
      if (pRight != NULL)
         assert(pRight->isRed == false);
   }

   // Rule d) Every path from a leaf to the root has the same # of black nodes
   if (pLeft == NULL && pRight && NULL)
      assert(depth == 0);
   if (pLeft != NULL)
      pLeft->verifyRedBlack(depth);
   if (pRight != NULL)
      pRight->verifyRedBlack(depth);
}

/******************************************************
 * VERIFY B TREE
 * Verify that the tree is correctly formed
 *****************************************************/
template <class T>
void BinaryNode <T> :: verifyBTree() const
{
   // check parent
   if (pParent)
      assert(pParent->pLeft == this || pParent->pRight == this);

   // check left
   if (pLeft)
   {
      assert(pLeft->data <= data);
      assert(pLeft->pParent == this);
      pLeft->verifyBTree();
   }

   // check right
   if (pRight)
   {
      assert(pRight->data >= data);
      assert(pRight->pParent == this);
      pRight->verifyBTree();
   }
}
```

bnode.h

```
/***********************************************
 * Program:
 *     Lesson 07, Binary Tree
 *     Brother Helfrich, CS265
 * Author:
 *     David Lambertson
 * Summary:
 *     This file holds the definition of the binary node
 *     used to create a binary tree.
 * Time:
 *    this part of the program took me around 5 hours.
```

```
 *********************************************/
#ifndef BNODE_H
#define BNODE_H

#include <iostream>
#include <cassert>

/****************************************************
 * This is the class that holds our Binary Node Definition.
 * It allows us to create Binary Nodes which are used for the tree.
 ************************************************/
template <class T>
class BinaryNode
{
  public:
   T data;
   BinaryNode<T> * pLeft;
   BinaryNode<T> * pRight;
   BinaryNode<T> * pParent;
   bool isRed;


   //Default Constructor
  BinaryNode() :pLeft(NULL), pRight(NULL), pParent(NULL), isRed(true) {}

   //Non-Default Constructor
  BinaryNode(T data) : data(data), pLeft(NULL), pRight(NULL),pParent(NULL),
      isRed(true) { case1(); }

   /********************************
    * These are our two add Left functions.
    * One takes data and the other takes a Node
    *********************************/
   void addLeft(const T & data);
   void addLeft(BinaryNode<T> * pNew);

   /********************************
    * Similar to our add Lefts, just for right.
    ******************************/
   void addRight(const T & data);
   void addRight(BinaryNode<T> * pNew);

   /****************************************
    *This checks to see if I am the Right child.
    ************************************/
   bool amIRight() const
   { return ((this->pParent) && (this->pParent->pRight == this)); }

   /*************************************
    * This checks if I am the Left child.
    ************************************/
   bool amILeft() const
   { return ((this->pParent) && (this->pParent->pLeft == this)); }

   //Prototype for copying a binary tree
   void copyBinaryTree(const BinaryNode<T> * pSrc, BinaryNode<T> * & pDest)
      throw (const char *);


  private:
   void case1(); //user doesn't need to know that I implemented
   void case2(); //a red-black tree along with my binary node.
   void case3();
   void case4();
   void balance();
};

/**************************************
 * case 1 for Black-Red Tree
 *************************************/
template<class T>
void BinaryNode<T> :: case1()
{
   //if I don't have a parent I must be the root
   //so I need to be black
   if(this->pParent == NULL)
      this->isRed = false;
}
```

Commented [HJ13]: Perfect function name.

Commented [HJ14]: cool

```
/************************************
 * case 2 for Black-Red Tree
 ************************************/
template<class T>
void BinaryNode<T> :: case2()
{
   //makes the parent black
   pParent->isRed == false;
}

/****************************************************
 * case 3 for Black-Red Tree
 ****************************************************/
template<class T>
void BinaryNode<T> :: case3()
{
   //these are the nodes we need to change
   BinaryNode<T> * pGran = this->pParent->pParent;
   BinaryNode<T> * pAunt = ((pGran->pRight == this->pParent) ?
                          pGran->pLeft : pGran->pRight);

   //recolor the node appropriately
   pGran->isRed = true;
   pAunt->isRed = false;
   pParent->isRed = false;

   //balance or check to see that we are all good
   pGran->balance();
}

/******************************
 * Case 4 for Black Red Tree
 ******************************/
template<class T>
void BinaryNode<T> :: case4()
{
   //these are so we don't lose this data in the four functions
   //while rearranging the pointers for rotating
   BinaryNode<T> * pGran = pParent->pParent;
   BinaryNode<T> * pAunt = ((pGran->pRight == pParent) ?
                          pGran->pLeft : pGran->pRight);
   BinaryNode<T> * pSibling = ((pParent->pRight == this) ?
                            pParent->pLeft : pParent->pRight);

   //case 4a
   //if I am the left child and my parent is the left child
   if(this->amILeft() && this->pParent->amILeft())
   {
      //change the colors of parent and grandparent
      pParent->isRed = false;
      pGran->isRed = true;

      //rearrange pointers for rotation
      pParent->pRight = pGran;

      //these are for seeing if we have a great-grandparent
      //and if I do, set the appropriate pointer to new child
      if(pGran->amIRight())
         pGran->pParent->pRight = pGran->pLeft;
      if(pGran->amILeft())
         pGran->pParent->pLeft = pGran->pLeft;

      //set pointers of parent and grandparent
      pParent->pParent = pGran->pParent;
      pGran->pParent = pParent;

      //bring back sibling if we have one
      pGran->pLeft = pSibling;
      if(pSibling)
         pSibling->pParent = pGran;

      //to break out of case 4 function
      return;
   }

   //case 4b
   //if I am the right child and my parent is the left child
   if(this->amIRight() && this->pParent->amILeft())
   {
```

```
      //change the colors of grandparent and I
      this->isRed = false;
      pGran->isRed = true;

      //rearrange pointers for rotation
      this->pParent->pRight = this->pLeft;

      //check if I have children
      //if I do, change pointers appropriately
      if(this->pLeft != NULL)
         this->pLeft->pParent = this->pParent;
      pGran->pLeft = this->pRight;
      if(this->pRight != NULL)
         this->pRight->pParent = pGran;

      //change parents and grandparents pointers
      this->pLeft = pParent;
      this->pRight = pGran;

      //these are for seeing if we have a great-grandparent
      //and if I do, set the appropriate pointer to new child
      if(pGran->amIRight())
         pGran->pParent->pRight = this;
      if(pGran->amILeft())
         pGran->pParent->pLeft = this;

      //finish changing pointers
      this->pParent = pGran->pParent;
      this->pRight->pParent = this;
      this->pLeft->pParent = this;

      //to break out of case 4 function
      return;
   }

   //case 4c
   //if I am the right child and my parent is the right child
   if(this->amIRight() && this->pParent->amIRight())
   {
      //change the colors of parent and grandparent
      pParent->isRed = false;
      pGran->isRed = true;

      //rearrange pointers for rotation
      pParent->pLeft = pGran;

      //these are for seeing if we have a great-grandparent
      //and if I do, set the appropriate pointer to new child
      if(pGran->amILeft())
         pGran->pParent->pLeft = pGran->pRight;
      if(pGran->amIRight())
         pGran->pParent->pRight = pGran->pRight;

      //set pointers of parent and grandparent
      pParent->pParent = pGran->pParent;
      pGran->pParent = pParent;

      //bring back sibling if we have one
      pGran->pRight = pSibling;
      if(pSibling)
         pSibling->pParent = pGran;

      //to break out of case 4 function
      return;
   }

   //case 4d
   //if I am the left child and my parent is the right child
   if(this->amILeft() && this->pParent->amIRight())
   {
      //change the colors of grandparent and I
      this->isRed = false;
      pGran->isRed = true;

      //rearrange pointers for rotation
      this->pParent->pLeft = this->pRight;

      //check if I have children
      //if I do, change pointers appropriately
```

Commented [HJ15]: not call case1()?

```cpp
      if(this->pRight != NULL)
         this->pRight->pParent = this->pParent;
      pGran->pRight = this->pLeft;
      if(this->pLeft != NULL)
         this->pLeft->pParent = pGran;

      //change parents and grandparents pointers
      this->pRight = pParent;
      this->pLeft = pGran;

      //these are for seeing if we have a great-grandparent
      //and if I do, set the appropriate pointer to new child
      if(pGran->amIRight())
         pGran->pParent->pRight = this;
      if(pGran->amILeft())
         pGran->pParent->pLeft = this;

      //finish changing pointers
      this->pParent = pGran->pParent;
      this->pLeft->pParent = this;
      this->pRight->pParent = this;

      //to break out of case 4 function
      return;
   }
}

/***************************
 * this is the overall function
 * that controls the balancing
 * it calls the different cases.
 **************************/
template<class T>
void BinaryNode<T> :: balance()
{
   //if I am the root
   if(pParent == NULL)//case 1
   {
      case1();
      return;
   }

   //if my parent is not the right color
   if(pParent->isRed == false)//case 2
   {
      case2();
      return;
   }

   //create these to check between case 3 and case 4
   BinaryNode<T> * pGran = this->pParent->pParent;
   BinaryNode<T> * pAunt = ((pGran->pRight == this->pParent) ?
                             pGran->pLeft : pGran->pRight);

   //if I have an aunt
   if(pAunt != NULL) //case 3
   {
      case3();
      return;
   }
   //if I don't have an aunt
   else //case 4
   {
      case4();
      return;
   }
}

/********************************
 * overloaded insertion operator allows us to display.
 *******************************/
template <class T>
std::ostream& operator <<(std::ostream& out,  const BinaryNode<T> * tmp)
{
   if (tmp == NULL)
      return out;
   return out << tmp->pLeft << tmp->data  << ' ' << ((tmp->isRed)? 'R' : 'B')
              << ' ' << tmp->pRight;
}
```

```
/**************************************
 * Function definition of our first addLeft
 *************************************/
template <class T>
void BinaryNode<T> :: addLeft(const T & data)
{
   //if I don't already have a left child
   if (this->pLeft == NULL)
   {
      BinaryNode<T> * left = new  BinaryNode<T>;
      left->data = data;
      this->pLeft = left;
      left->pParent = this;
   }
   //go down to that next left node
   else
      this->pLeft->addLeft(data);
   //balance that new node after we have inserted
   pLeft->balance();
}

/****************************************
 * second definition of addLeft
 ***************************************/
template <class T>
void BinaryNode<T> :: addLeft(BinaryNode<T> * left)
{
   //if I don't already have a left child
   if (this->pLeft == NULL)
   {
      this->pLeft = left;
      left->pParent = this;
   }
   //go down to that next left node
   else
      this->pLeft->addLeft(left);
}

/**********************************
 * first definition of addRight
 ********************************/
template <class T>
void BinaryNode<T> :: addRight(const T & data)
{
   //if I don't already have a right child
   if (pRight == NULL)
   {
      BinaryNode<T> * right = new BinaryNode<T>;
      right->data = data;
      this->pRight = right;
      right->pParent = this;

   }
   //go down to that next right node
   else
      this->pRight->addRight(data);
   //balance that new node after we have inserted
   pRight->balance();
}

/****************************************
 * Second definition of addRight
 ************************************/
template <class T>
void BinaryNode<T> :: addRight(BinaryNode<T> * right)
{
   //if I don't already have a right child
   if (pRight == NULL)
   {
      this->pRight = right;
      right->pParent = this;
   }
   //go down to that next right node
   else
      this->pRight->addRight(right);
}

/*********************************************
```

```
 * function definition of deleteBinaryTree allowing us
 * to delete a binary tree we have created.
 ******************************************/
template <class T>
void deleteBinaryTree(BinaryNode<T> * & root)
{
   if (root == NULL)
      return;
   deleteBinaryTree(root->pLeft);
   deleteBinaryTree(root->pRight);
   delete root;
   root = NULL; //needed this to get rid of last node
}


/********************************************
 * starting at the root, deep copies the tree.
 ********************************************/
template<class T>
void BinaryNode<T> :: copyBinaryTree(const BinaryNode<T> * pSrc,
                                     BinaryNode<T> * & pDest)
throw (const char *)
{
   //create node to be able to iterate through source
   BinaryNode<T> * p = NULL;

   try
   {
      //if I am the root
      if (pSrc->pParent == NULL)
      {
         p = new BinaryNode<T>(pSrc->data);
         p->isRed = pSrc->isRed;
         pDest = p;
      }
      //if I have a right child, copy data to destination
      if (pSrc->pRight)
      {
         p = new BinaryNode<T>(pSrc->pRight->data);
         p->isRed = pSrc->pRight->isRed;
         pDest->addRight(p);
         copyBinaryTree(pSrc->pRight, pDest->pRight);
      }
      //if I have a left child, copy data to destination
      if (pSrc->pLeft)
      {
         p = new BinaryNode<T>(pSrc->pLeft->data);
         p->isRed = pSrc->pLeft->isRed;
         pDest->addLeft(p);
         copyBinaryTree(pSrc->pLeft, pDest->pLeft);
      }
   }
   catch(...)
   {
      throw "ERROR!!!!";
   }
}

#endif //BNODE_H
```

wordCount.cpp

```
/***********************************************************************
 * Module:
 *    Lesson 09, WORD COUNT
 *    Brother Helfrich, CS 235
 * Author:
 *    Derek Calkins and David Lambertson
 * Summary:
 *    This program will implement the wordCount() function
 ************************************************************************/
#include "map.h"       // for MAP
#include "wordCount.h" // for wordCount() prototype
#include <fstream>
using namespace std;
void readFile(Map <string, Count> & counts, const string & fileName);

/*****************************************************
```

```
 * WORD COUNT
 * Prompt the user for a file to read, then prompt the
 * user for words to get the count from
 ****************************************************/
void wordCount()
{
   string fileName;
   string word;
   Map<string, Count> occur;

   cout << "What is the filename to be counted? ";
   cin  >> fileName;

   cout << "What word whose frequency is to be found. Type ! when done\n";

   readFile(occur, fileName);

   //to get a word for word frequency from user
   do
   {
      cout << "> ";
      cin  >> word;

      cout << "\t" << word << " : " << (occur[word]).getCount() << endl;
   }
   while(word != "!");

}

/*****************************************
 * this function reads in the words of the file
 * and if finds a repeat, just increments Count
 ****************************************/
void readFile(Map <string, Count> & counts, const string & fileName)
{
   //open the file to be read
   ifstream fin(fileName.c_str());
   if(fin.fail())
      cout << "FAIL";

   Count num;   //to hold the amount of times a word is found
   string word; //to read in each word
   fin >> word;
   while(!fin.eof())
   {
      MapIterator<string, Count> it; //NULL check

      //if we have a new word
      if(it == counts.find(word))
      {
         num = 1;                     //initial count of word
         counts[string(word)] = num; //set Count to word
      }
      //add to Count of the word we already have
      else
      {
         num = *(counts.find(word)); //Count becomes count of word
         ++num;                      //increase Count
         counts[word] = num;         //set Count to word
      }
      //get next word
      fin >> word;
   }

   fin.close();
}
```

lesson09.cpp

```
/***********************************************************************
* Program:
*    Lesson 09, Map and balanced BSTs
*    Brother Helfrich, CS 235
* Author:
*    Br. Helfrich
* Summary:
*    This is a driver program to exercise the Map class.  When you
```

```
*    submit your program, this should not be changed in any way.  That being
*    said, you may need to modify this once or twice to get it to work.
***********************************************************************/

#include <iostream>      // for CIN and COUT
#include <string>        // for STRING
//#include "bst.h"
#include "map.h"         // for BST class which should be in bst.h
#include "wordCount.h"   // for the wordCount() function
using namespace std;


// prototypes for our four test functions
void testSimple();
void testAdd();
void testIterate();
void testQuery();
void testSort();
void testBalance();

// To get your program to compile, you might need to comment out a few
// of these. The idea is to help you avoid too many compile errors at once.
// I suggest first commenting out all of these tests, then try to use only
// TEST1.  Then, when TEST1 works, try TEST2 and so on.
#define TEST1   // for testSimple()
#define TEST2   // for testAdd()
#define TEST3   // for testIterate()
#define TEST4   // for testQuery()
#define TESTB   // for testBalance()

/**********************************************************************
 * MAIN
 * This is just a simple menu to launch a collection of tests
 ***********************************************************************/
int main()
{
   // menu
   cout << "Select the test you want to run:\n";
   cout << "\t1. Just create and destroy a Map\n";
   cout << "\t2. The above plus add a few entries\n";
   cout << "\t3. The above plus display the contents of a Map\n";
   cout << "\t4. The above plus retrieve entries from the Map\n";
   cout << "\ta. Count word frequency\n";
   cout << "\tb. Test tree balancing\n";

   // select
   char choice;
   cout << "> ";
   cin  >> choice;
   switch (choice)
   {
      case 'a':
         wordCount();
         break;
      case 'b':
         testBalance();
         cout << "Test Balance complete\n";
         break;
      case '1':
         testSimple();
         cout << "Test 1 complete\n";
         break;
      case '2':
         testAdd();
         cout << "Test 2 complete\n";
         break;
      case '3':
         testIterate();
         cout << "Test 3 complete\n";
         break;
      case '4':
         testQuery();
         cout << "Test 4 complete\n";
         break;
      default:
         cout << "Unrecognized command, exiting...\n";
   }

   return 0;
```

```
}

/*******************************************
 * TEST SIMPLE
 * Very simple test for a Map: create and destroy
 ******************************************/
void testSimple()
{
#ifdef TEST1
   // Test1: a bool-int Map
   cout << "Create a bool-int Map\n";
   Map <bool, int> m;

   // Test2: double-bool Map
   cout << "Create a double-bool Map\n";
   Map <double, bool> * pM = new Map <double, bool>;
   delete pM;
#endif //TEST1
}

/*******************************************
 * TEST ADD
 * Add a few nodes to the Map then
 * destroy it when done
 *****************************************/
void testAdd()
{
#ifdef TEST2
   // create
   cout << "Create an integer-string Map\n";
   Map <int, string> m1;
   Map <int, string> m2;
   cout << "\tEmpty? " << (m1.empty() ? "yes" : "no") << endl;
   cout << "\tCount: " << m1.size() << endl;

   // fill
   cout << "Fill with 10 values\n";
   m1[8]  = string("eight");    //               8
   m1[4]  = string("four");     //          +----+----+
   m1[12] = string("twelve");   //          4         12
   m1[2]  = string("two");      //       +--+--+   +--+--+
   m1[6]  = string("six");      //       2     6   9     13
   m1[9]  = string("nine");     //     +-+   +-+   +-+
   m1[13] = string("thirteen"); //     0     5       11
   m1[0]  = string("zero");
   m1[5]  = string("five");
   m1[11] = string("eleven");

   m2 = m1;
   m1.clear();
   cout << "\tEmpty? " << (m2.empty() ? "yes" : "no") << endl;
   cout << "\tCount: " << m2.size() << endl;

   // clear
   cout << "Empty the contents\n";
   cout << "\tEmpty? " << (m1.empty() ? "yes" : "no") << endl;
   cout << "\tCount: " << m1.size() << endl;

#endif // TEST2
}

/*******************************************
 * TEST ITERATE
 * We will build a Map and display the
 * contents on the screen
 ******************************************/
void testIterate()
{
#ifdef TEST3
   cout.setf(ios::fixed | ios::showpoint);
   cout.precision(1);

   //
   // An empty map
   //
   try
   {
      cout << "Create an empty bool-bool Map\n";
      Map <bool, bool> m;
```

```
      MapIterator <bool, bool> it;
      cout << "\tEmpty? " << (m.empty() ? "yes" : "no") << endl;
      cout << "\tCount: " << m.size() << endl;

      // display the contents
      cout << "\tContents: ";
      for (it = m.begin(); it != m.end(); ++it)
         cout << (*it) << " ";
      cout << endl;

      // map deleted
      cout << "\tMap deleted\n";
   }
   catch (const char * s)
   {
      cout << "Thrown exception: " << s << endl;
   }

   //
   // a non-trivial map
   //
   try
   {
      cout << "Create a string-integer Map\n";
      Map <string, int> m1;
      Map <string, int> m2;
      MapIterator <string, int> it;
      cout << "\tEmpty? " << (m1.empty() ? "yes" : "no") << endl;
      cout << "\tCount: " << m1.size() << endl;

      // fill the tree
      cout << "\tFill the Map with: f c i b e g j a d h\n";
      m1[string("f")] = 6;
      m1[string("c")] = 3;   //               f
      m1[string("i")] = 9;   //          +----+----+
      m1[string("b")] = 2;   //          c         i
      m1[string("e")] = 5;   //       +--+--+   +--+--+
      m1[string("g")] = 7;   //       b     e   g     j
      m1[string("j")] = 10;  //     +-+   +-+   +-+
      m1[string("a")] = 1;   //     a     d       h
      m1[string("d")] = 4;
      m1[string("h")] = 8;
      //cout << m1["f"] << m1["d"] << m1["h"] << endl;
      m2 = m1;
      m1.clear();
      cout << "\tCount: " << m2.size() << endl;


      //cout << m2.begin()
      // display the contents forward
      cout << "\tContents forward:  ";
      for (it = m2.begin(); it != m2.end(); ++it)
         cout << *it << "  ";
      cout << endl;

      // display the contents backwards
      cout << "\tContents backward: ";
      for (it = m2.rbegin(); it != m2.rend(); --it)
         cout << *it << "  ";
      cout << endl;

      // tree deleted
      cout << "\tMap deleted\n";
   }
   catch (const char * s)
   {
      cout << "Thrown exception: " << s << endl;
   }

#endif // TEST3
}

/*******************************************
 * TEST QUERY
 * Prompt the user for items to put in the map
 * and then allow the user to query for items
 *******************************************/
void testQuery()
{
```

```
#ifdef TEST4
   try
   {
      // create the map
      cout << "Create a char-string Map\n";
      Map <char, string> m;
      char letter;
      string word;

      // fill the map
      cout << "Please enter a letter word pair. "
           << "Enter ! for the letter when finished.\n";
      cout << "> ";
      cin  >> letter;
      while (letter != '!')
      {
         cin >> word;
         m[letter] = word;
         cout << "> ";
         cin  >> letter;
      }
      cout << "\tThere are " << m.size() << " items in the map\n";

      // prompt for the values in the map
      cout << "Please enter the letter to be found. Enter ! when finished.\n";
      cout << "> ";
      cin  >> letter;
      while (letter != '!')
      {
         cout << '\t' << m[letter] << endl;
         cout << "> ";
         cin  >> letter;
      }
   }
   catch (const char * s)
   {
      cout << "Thrown exception: " << s << endl;
   }
#endif // TEST4
}

/**************************************************
 * TEST BALANCE
 * Test if a given tree is balanced
 *************************************************/
void testBalance()
{
#ifdef TESTB
   cout << "Create a simple Binary Search Tree\n";
   BST <int> tree;
   BinaryNode <int> * root;

   // Case 1: Add a black root
   //cout << "\tCase 1\n";
   tree.insert(60);                           //      60b
   root = tree.getRoot();
   assert(root->isRed == false);
   cout << "\tPass Case 1\n";

   // Case 2: Add two children which will be red
   //cout << "\tcase 2\n";
   tree.insert(50);                           //       60b
   tree.insert(70);                           //   +----+-----+
   assert(root->pRight->isRed == true);       //   50r       70r
   assert(root->pLeft->isRed == true);
   cout << "\tPass Case 2\n";

   // Case 3: Add a child which should case 50 and 70 to turn black
   //cout << "\tCase 3\n";
   tree.insert(20);                            //           60b
   assert(root->isRed == false);               //      +-----+-----+
   assert(root->data  == 60);                  //     50b         70b
   assert(root->pRight->isRed == false);       //   +--+
   assert(root->pRight->data  == 70);          // 20r
   assert(root->pLeft->isRed == false);
   assert(root->pLeft->data  == 50);
   assert(root->pLeft->pLeft->isRed == true);
   assert(root->pLeft->pLeft->data  == 20);
   cout << "\tPass Case 3\n";
```

```
   // Case 4a: Add a child to 20 which should cause a right rotation on 50
   //cout << "\tCase 4a\n";
   tree.insert(10);                           //           60b
   assert(root->isRed == false);              //      +-----+-----+
   assert(root->data  == 60);                 //     20b         70b
   assert(root->pRight->isRed == false);      //   +--+--+
   assert(root->pRight->data  == 70);         // 10r     50r
   assert(root->pLeft->isRed == false);
   assert(root->pLeft->data  == 20);
   assert(root->pLeft->pLeft->isRed == true);
   assert(root->pLeft->pLeft->data  == 10);
   assert(root->pLeft->pRight->isRed == true);
   assert(root->pLeft->pRight->data  == 50);
   cout << "\tPass Case 4a\n";

   // Case 4b: Add 30 (Case 3 then 2) followed by 40 (Case 4b)
   //cout << "\tcase 4b\n";
   tree.insert(30); // cause 3, followed by 2
   tree.insert(40); // cause 4b
   assert(root->isRed == false);              //              60b
   assert(root->data  == 60);                 //       +-------+-------+
   assert(root->pRight->isRed == false);      //      20r              70b
   assert(root->pRight->data  == 70);         //  +----+-----+
   assert(root->pLeft->isRed == true);        // 10b        40b
   assert(root->pLeft->data  == 20);          //         +---+---+
   assert(root->pLeft->pLeft->isRed == false);//        30r      50r
   assert(root->pLeft->pLeft->data  == 10);
   assert(root->pLeft->pRight->isRed == false);
   assert(root->pLeft->pRight->data  == 40);
   assert(root->pLeft->pRight->pRight->isRed == true);
   assert(root->pLeft->pRight->pRight->data  == 50);
   assert(root->pLeft->pRight->pLeft->isRed == true);
   assert(root->pLeft->pRight->pLeft->data  == 30);
   cout << "\tPass Case 4b\n";

   // Case 4c: Add 100 (Case 2) followed by 110 (Case 4c) rotate left
   //cout << "\tCase 4c\n";
   tree.insert(100);   // case 2
   tree.insert(110);   // case 4c
   assert(root->isRed == false);              //         60b
   assert(root->data  == 60);                 //  +-------+-------+
   assert(root->pRight->isRed == false);      // ...             100b
   assert(root->pRight->data  == 100);        //              +----+-----+
   assert(root->pRight->pRight->isRed == true);//             70r       110r
   assert(root->pRight->pRight->data  == 110); //
   assert(root->pRight->pLeft->isRed == true);
   assert(root->pRight->pLeft->data  == 70);
   cout << "\tPass Case 4c\n";
   //cout << tree.getRoot() << endl;

   // Case 4d: Add 90 (Case 3 then 2) followed by 80 (Case 4d)
   //cout << "\tCase 4d\n";
   tree.insert(90);    // case 3 followed by 2
   tree.insert(80);    // case 4d
   assert(root->isRed == false);               //         60b
   assert(root->data  == 60);                  //  +-------+-------+
   assert(root->pRight->isRed == true);        // ...             100r
   assert(root->pRight->data  == 100);         //              +----+-----+
   assert(root->pRight->pRight->isRed == false);//             80b       110b
   assert(root->pRight->pRight->data  == 110);  //          +---+---+
   assert(root->pRight->pLeft->isRed == false); //         70r    90r
   assert(root->pRight->pLeft->data  == 80);
   assert(root->pRight->pLeft->pLeft->isRed == true);
   assert(root->pRight->pLeft->pLeft->data  == 70);
   assert(root->pRight->pLeft->pRight->isRed == true);
   assert(root->pRight->pLeft->pRight->data  == 90);
   cout << "\tPass Case 4d\n";

   // make sure it all works as we expect
   cout << "Final tree:";
   for (BSTIterator <int> it = tree.begin(); it != tree.end(); ++it)
      cout << ' ' << *it;
   cout << endl;

#endif // TESTB
}
```

Test Bed Results

```
cs235d.out:

Started program
   > Select the test you want to run:
   >    1. Just create and destroy a Map
   >    2. The above plus add a few entries
   >    3. The above plus display the contents of a Map
   >    4. The above plus retrieve entries from the Map
   >    a. Count word frequency
   >    b. Test tree balancing
   > > 1
   > Create a bool-int Map
   > Create a double-bool Map
   > Test 1 complete
Program terminated successfully

Started program
   > Select the test you want to run:
   >    1. Just create and destroy a Map
   >    2. The above plus add a few entries
   >    3. The above plus display the contents of a Map
   >    4. The above plus retrieve entries from the Map
   >    a. Count word frequency
   >    b. Test tree balancing
   > > 2
   > Create an integer-string Map
   >    Empty? yes
   >    Count: 0
   > Fill with 10 values
   >    Empty? no
   >    Count: 10
   > Empty the contents
   >    Empty? yes
   >    Count: 0
   > Test 2 complete
Program terminated successfully

Started program
   > Select the test you want to run:
   >    1. Just create and destroy a Map
   >    2. The above plus add a few entries
   >    3. The above plus display the contents of a Map
   >    4. The above plus retrieve entries from the Map
   >    a. Count word frequency
   >    b. Test tree balancing
   > > 3
   > Create an empty bool-bool Map
   >    Empty? yes
   >    Count: 0
   >    Contents:
   >    Map deleted
   > Create a string-integer Map
   >    Empty? yes
   >    Count: 0
   >    Fill the Map with: f c i b e g j a d h
   >    Count: 10
   >    Contents forward:  1  2  3  4  5  6  7  8  9  10
   >    Contents backward: 10  9  8  7  6  5  4  3  2  1
   >    Map deleted
   > Test 3 complete
Program terminated successfully

Started program
   > Select the test you want to run:
   >    1. Just create and destroy a Map
   >    2. The above plus add a few entries
   >    3. The above plus display the contents of a Map
   >    4. The above plus retrieve entries from the Map
   >    a. Count word frequency
   >    b. Test tree balancing
   > > a
   > What is the filename to be counted? /home/cs235/lesson09/D_C_121.txt
   > What word whose frequency is to be found. Type ! when done
   > > Nephi
   >    Nephi : 0
```

```
   > > Lord
   >    Lord : 6
   > > Christ
   >    Christ : 1
   > > I
   >    I : 2
   > > the
   >    the : 79
   > > C++
   >    C++ : 0
   > > !
Program terminated successfully

Started program
   > Select the test you want to run:
   >    1. Just create and destroy a Map
   >    2. The above plus add a few entries
   >    3. The above plus display the contents of a Map
   >    4. The above plus retrieve entries from the Map
   >    a. Count word frequency
   >    b. Test tree balancing
   > > b
   > Create a simple Binary Search Tree
   >    Pass Case 1
   >    Pass Case 2
   >    Pass Case 3
   >    Pass Case 4a
   >    Pass Case 4b
   >    Pass Case 4c
   >    Pass Case 4d
   > Final tree: 10 20 30 40 50 60 70 80 90 100 110
   > Test Balance complete
Program terminated successfully

No Errors
```

Grading Criteria

Criteria	Exceptional 100%	Good 90%	Acceptable 70%	Developing 50%	Missing 0%	Weight	Score
Map interface	The interfaces are perfectly specified with respect to const, pass-by-reference, etc.	lesson09.cpp compiles without modification	All the methods in Map match the problem definition	Map has many of the same interfaces as the problem definition	The public methods and variables in the Map class do not resemble the problem definition	10	
Map Implementation	Passes all four Map testBed tests	Passes three testBed tests	Passes two testBed tests	Passes one testBed test	Program fails to compile or does not pass any testBed tests	20	
MapIterator	Solution works, is elegant, and efficient	Both forward and reverse iterators work	Works in some limited cases	Elements of the solution are present	No attempt was made to iterate through the Map	10	
Word Count	Code is elegant and efficient	Passes the Word Count testBed tests	The code essentially works but with minor defects	Elements of the solution are present	The Word Count problem was not attempted	10	
Red-Black Tree	Passes ass the Red-Black tests	Passes Case 4a	Passes Case 3	Passes Case 1 and Case 2	No Red-Black tree tests pass test-bed	40	
Code Quality	There is no obvious room for improvement	All the principles of encapsulation and modularization are honored	One function is written in a "backwards" way or could be improved	Two or more functions appears "thrown together."	The code appears to be written without any obvious forethought	20	
Style	Great variable names, no errors, great comments	No obvious style errors	A few minor style errors: non-standard spacing, poor variable names, missing comments, etc.	Overly generic variable names, misleading comments, or other gross style errors	No knowledge of the BYU-I code style guidelines were demonstrated	10	
Total							120

Commented [HJ16]: You guys absolutely nailed it!