makefile

```
###############################################################
# Program:
#     Lesson 10, Sorts
#     Brother Helfrich, CS265
# Author:
#     David Lambertson and Derek Calkins
# Summary:
#     This program will let the user choose an amount of data
#     and sort it using seven different sorts.
# Time:
#     Overall, this assignment took us 8 hours of coding and discussion.
#     Derek: 50% David: 50%
###############################################################

##############################################################
# The main rule
##############################################################
a.out: lesson10.o
	g++ -g -o a.out lesson10.o
	tar -cf lesson10.tar *.h *.cpp makefile

##############################################################
# The individual components
#      lesson10.o     : the driver program
##############################################################
lesson10.o: bnode.h bst.h lesson10.cpp sortValue.h \
              sortBinary.h sortInsertion.h sortHeap.h sortBubble.h \
              sortSelection.h sortMerge.h sortQuick.h
	g++ -g -c lesson10.cpp
```

Commented [HJ1]: Well do ne!

Commented [HJ2]: Good.

sortSelection.h

```
/***********************************************************************
 * Module:
 *    Lesson 10, Sort Select
 *    Brother Helfrich, CS 235
 * Author:
 *    David Lambertson
 * Summary:
 *    This program will implement the Selection Sort
 ************************************************************************/

#ifndef SORT_SELECTION_H
#define SORT_SELECTION_H

/*****************************************************
 * SORT SELECTION
 * Perform the selection sort
 ****************************************************/
template <class T>
void sortSelection(T array[], int num)
{
   //for the entire array
   for (int i = 0; i < num-1; i++)
   {
      int smallPost = i; //position of the smallest item
      T smallest = array[i];// the actual smallest item
      for (int j = i + 1; j < num; j++)
      {
         if (!(array[j] > smallest)) //if I find something smaller, save it
         {
            smallPost = j;
            smallest = array[j];
         }
      }

      //switch the position i, with the smallest
```

Commented [HJ3]: I and J are not the best names here.

Commented [HJ4]: Good variable name.

```
      T temp = array[i];
      array[i] = smallest;
      array[smallPost] = temp;

   }
}


#endif // SORT_SELECTION_H
```

sortBubble.h

```
/***********************************************************************
 * Module:
 *    Lesson 10, Sort Bubble
 *    Brother Helfrich, CS 235
 * Author:
 *    Derek Calkins
 * Summary:
 *    This program will implement the Bubble Sort
 ************************************************************************/

#ifndef SORT_BUBBLE_H
#define SORT_BUBBLE_H

/*****************************************************
 * SORT BUBBLE
 * Perform the bubble sort
 ****************************************************/
template <class T>
void sortBubble(T array[], int num)
{
   T data;
   int numComp = num - 1; //to know how many times we have swapped
   //go until i have moved my last item or we don't need to swap
   while(numComp)
   {
      int last = 0; //to check if we have swapped
      //loop until we reach the end of the unsorted items
      for(int i = 0; i < numComp; ++i)
      {
         //if the current spot's item is greater than
         //the next spot's item, switch the two.
         if(array[i] > array[i + 1])
         {
            //swap items
            data = array[i];
            array[i] = array[i + 1];
            array[i + 1] = data;
            //if we swapped we change last to i
            last = i;
         }
      }
      //we go until we have reached the last swapped item
      numComp = last;
   }
}


#endif // SORT_BUBBLE_H
```

Commented [HJ5]: This is quite odd.

sortQuick.h

```
/***********************************************************************
 * Module:
 *    Lesson 10, Sort Quick
 *    Brother Helfrich, CS 235
 * Author:
 *    Derek Calkins (using code from textbook)
 * Summary:
 *    This program will implement the Quick Sort
 ************************************************************************/

#ifndef SORT_QUICK_H
#define SORT_QUICK_H

/*****************************************************
 * SORT QUICK
```

```
 * Perform the quick sort
 ****************************************************/
template <class T>
void sortQuick(T array[], int num)
{
   quickSort(array, 0, num - 1);
}

/****************************************************
 * Split Function
 * This function separates and swaps the values based
 * on the pivot.
 ****************************************************/
template <class T>
int split(T array[], const int & iLeft, const int & iRight)
{
   //set pivot to value of iLeft
   T pivot = array[iLeft];
   //start at index of left
   int left = iLeft;
   //start at index of right
   int right = iRight;

   //loop while the indexes are not equal
   while(right > left)
   {
      //until we find a value less than or equal to pivot
      while(array[right] > pivot)
         right--;

      //until we find a value more than pivot
      while((right > left) &&
            (pivot > array[left] || pivot == array[left]))
         left++;

      //if the indexes haven't meet yet
      if(right > left)
      {
         //swap left and right values compared to pivot
         T data = array[left];
         array[left] = array[right];
         array[right] = data;
      }
   }
   int position = right;           //creates position that is right index
   array[iLeft] = array[position]; //assigns data in left index to
                                   //data in position
   array[position] = pivot;        //assigns data in position to pivot
   return position;                //return position index
}

/*******************************************
 * Recursive Quick Sort
 * This function sets the position as the new pivot
 * and changes the range of values we compare by one
 * for each recursive function call.
 *******************************************/
template <class T>
void quickSort(T array[], const int & iLeft, const int & iRight)
{
   int position;
   //while index of right is less than index of left
   if(iRight > iLeft)
   {
      //set position as new pivot
      position = split(array, iLeft, iRight);
      //moves iRight to position minus one
      quickSort(array, iLeft, position - 1);
      //moves iLeft to position plus one
      quickSort(array, position + 1, iRight);
   }
}
#endif // SORT_QUICK_H
```

Commented [HJ6]: Good.

Commented [HJ7]: Not by-reference; this is just an integer. It is cheap to copy.

Commented [HJ8]: Great names.

sortInsertion.h

```
/************************************************************************
 * Module:
```

```
 *    Lesson 10, Sort Insertion
 *    Brother Helfrich, CS 235
 * Author:
 *    Derek Calkins
 * Summary:
 *    This program will implement the Insertion Sort
 ************************************************************************/

#ifndef SORT_INSERTION_H
#define SORT_INSERTION_H

#include <cassert>

/*****************************************************
 * SORT INSERTION
 * Perform the insertion sort
 ****************************************************/
template <class T>
void sortInsertion(T array[], int num)
{

   int iWall = 1; //start at one because the first item
                  //in the array is already sorted
   //shift iWall after we insert
   for(iWall; iWall < num; iWall++)
   {
      T data = array[iWall]; //save data in item we want to insert
      int spot = binarySearch(array, iWall, data);
      int i;
      //loop through to shift the data until we want to insert
      for(i = iWall; i > spot; --i)
         array[i] = array[i - 1];
      array[i] = data;  //shift the data
   }
}

/*****************************************************
 * Binary Search
 * This finds the correct position where the data
 * should be inserted.
 *****************************************************/
template <class T>
int binarySearch(const T array[], int size, const T & data)
{
   int iFirst = 0;
   int iLast = size - 1;
   int iMiddle;

   //while the last index is not less than the first index
   while(iLast >= iFirst)
   {
      //for binary search to start at middle for
      //log n find rather than linear find
      iMiddle = (iLast + iFirst) / 2;

      //if the data of iMiddle is greater than data
      //then our position must be before the iMiddle
      if(array[iMiddle] > data)
         iLast = iMiddle - 1;
      //else our position needs to be after iMiddle
      else
         iFirst = iMiddle + 1;
   }
   //return iFirst because it will be where the position
   //needs to be
   return iFirst;
}


#endif // SORT_INSERTION_H
```

Commented [HJ9]: Good, This should be declared here. Just putting the variable here in the loop does nothing.

sortValue.h

```
/************************************************************************
 * Component:
 *    Lesson 10, Sort Value
 *    Brother Helfrich, CS 235
 * Author:
```

```
 *    Br. Helfrich
 * Summary:
 *    The purpose of this data-type is to test sorting algorithms.
 *    This data-type has several properties:
 *     1.   It only defines the less-than operator, the only thing necessary
 *          for a comparison sort
 *     2.   It counts the number of times that a value is copied
 *     3.   It counts the number of times the < operator was called
 *     4.   The insertion operator is defined so you can display the results
 ************************************************************************/

#ifndef SORT_VALUE_H
#define SORT_VALUE_H

#include <iostream>  // for CIN
#include <stdlib.h>  // for rand()

/*********************************************************
 * SORT VALUE
 *********************************************************/
class SortValue
{
public:
   // constructors
   SortValue() : value(0) {}
   SortValue(const SortValue & rhs) { *this = rhs; }

   // comparision
   bool operator > (const SortValue & rhs) const
   {
      compares++;
      return value > rhs.value;
   }
   bool operator == (const SortValue & rhs) const
   {
      return value == rhs.value;
   }

   // assignment
   SortValue & operator = (const SortValue & rhs)
   {
      assign++;
      value = rhs.value;
      return *this;
   }
   SortValue & operator = (int rhs)
   {
      assign++;
      value = rhs;
      return *this;
   }

   // fill with a random number
   void random()
   {
      value = rand() % 1000000;
   }

   // reset the counters
   void reset()
   {
      assign   = 0;
      compares = 0;
   }

   // display
   friend std::ostream & operator << (std::ostream & out, const SortValue &rhs)
   {
      out << rhs.value;
      return out;
   }

   // get the statistics
   unsigned long getAssigns()  const { return assign;   }
   unsigned long getCompares() const { return compares; }

private:
   int value;                     // the value we will be sorting with
   static unsigned long assign;   // # times the assigment operator was called
```

```
   static unsigned long compares; // # times the < operator was called
};

unsigned long SortValue :: assign = 0;
unsigned long SortValue :: compares = 0;

#endif // SORT_VALUE_H
```

sortHeap.h

```
/***********************************************************************
 * Module:
 *    Lesson 10, Sort Heap
 *    Brother Helfrich, CS 235
 * Author:
 *    David Lambertson and Derek Calkins
 * Summary:
 *    This program will implement the Heap Sort
 ************************************************************************/

#ifndef SORT_HEAP_H
#define SORT_HEAP_H

#include <iostream>

template <class T>
class Heap
{
  public:
   //default constructor
  Heap() :num(0) {}

   //non-default constructor
  Heap(T array[], int num) : array(array), num(num) {}

   //destructor
   ~Heap() {}

   /**********************************************
    *heapifies the array and turns it into a proper heap
    *starting at the last parent node
    ********************************************/
   void heapify()
   {
      array--;
      for (int i = num/2; i > 0 ; --i)
          percolateDown(i);
   }

   //prototype for percolateDown function
   void percolateDown(int spot);

   //returns what our max number is
   T getMax() { return array[1]; }

   /*********************************************
    *swaps the max(first item) with the item last in the array
    * getting the max in its proper place, then places a wall
    * after the last item.
    *****************************************/
   void deleteMax()
   {
      swap(1, num);
      num--;
   }

   /****************************************
    *calls the deleteMax function, then percolates Down
    * on the root to make a proper heap again.
    * Repeats this until we have finished the whole heap.
    *************************************/
   void sort()
   {
      while (num > 0)
      {
         deleteMax();
         percolateDown(1);
      }
```

Commented [HJ10]: Need a comment block just before the class definition.

Commented [HJ11]: Great work.

```
   }

  private:
   T * array;
   int num;

   //swaps the two items at spots passed in.
   void swap(int spot1, int spot2)
   {
      T temp = array[spot2];
      array[spot2] = array[spot1];
      array[spot1] = temp;
   }


};

/****************************************
 *takes a spot as a parameter and from that given spot,
 *heapifies the subtree.
 ************************************/
template <class T>
void Heap<T>:: percolateDown(int spot)
{
   int iLeft = spot * 2;
   int iRight = iLeft + 1;

   //if our iLeft is on the right side of the wall, just finish.
   if (iLeft > (num-1))
      return;

   //if spot is less than either child
   if (array[iLeft] > array[spot] || array[iRight] > array[spot])
   {
      //if its left is greater than right
      if (array[iLeft] > array[iRight])
      {
         swap(spot, iLeft);
         percolateDown(iLeft);
      }
      else
      {
         swap(spot, iRight);
         percolateDown(iRight);
      }
   }


}



/*****************************************************
 * SORT HEAP
 * Perform the heap sort
 ****************************************************/
template <class T>
void sortHeap(T array[], int num)
{
   Heap<T> heap(array, num); //create the heap
   heap.heapify();           //heapify the heap
   heap.sort();              //sort the heap

}


#endif // SORT_HEAP_H
```

Commented [HJ12]: Sorta like thing1 and thing2

Commented [HJ13]: Good.

bst.h

```
/*******************************
 * Module:
 *    Lesson 08, BST
 *    Brother Helfrich, CS 235
 * Author:
 *    David Lambertson and Derek Calkins
 * Summary:
```

```cpp
 *    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()
   {
      if (myRoot->pParent != NULL)
         myRoot = myRoot->pParent;
      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
```

sortBinary.h

```
/***********************************************************************
 * Module:
 *    Lesson 10, Sort Binary
 *    Brother Helfrich, CS 235
 * Author:
 *    David Lambertson
 * Summary:
 *    This program will implement the Binary Tree Sort
 ************************************************************************/

#ifndef SORT_BINARY_H
#define SORT_BINARY_H

#include "bst.h"
#include <cassert>
#include <iostream>

/*****************************************************
 * SORT BINARY
 * Perform the binary tree sort
 ****************************************************/
template <class T>
void sortBinary(T array[], int num)
{
   //create the binary search tree
   BST<T> tree;

   //insert everything into the tree
   for (int i = 0; i < num; i++)
   {
      tree.insert(array[i]);
   }

   int i = 0;

   //insert the sorted tree back into the array using infix order.
   for (BSTIterator<T> it = tree.begin(); it != tree.end(); ++it, i++)
   {
      array[i] = *it;
   }
}


#endif // SORT_BINARY_H
```

Commented [HJ14]: So easy.

sortMerge.h

```
/***********************************************************************
 * Module:
 *    Lesson 10, Sort Merge
 *    Brother Helfrich, CS 235
 * Author:
 *    Derek Calkins and David Lambertson
 * Summary:
 *    This program will implement the Merge Sort
 ************************************************************************/

#ifndef SORT_MERGE_H
#define SORT_MERGE_H

/*****************************************************
 * SORT MERGE
 * Perform the merge sort
 ****************************************************/
template <class T>
void sortMerge(T array[], int num)
{
   int arrayNum = 0, array1Num = 0;
   T array1[num];
   int i = 0;
   int numMerges = 1;

   //while we keep merging execute this code
   while(numMerges)
   {
      numMerges = 0;
      merge(array, array1, arrayNum, array1Num, num, numMerges);

      arrayNum = 0;
      array1Num = 0;
      //save into the actual array
      for (int m = 0; m < num; ++m)
         array[m] = array1[m];
   }

}

/***********************************************************
 * This takes us through the array, finding the subarrays
 * and merges two subarrays and continues until the end of
 * the full array.
 **********************************************************/
template <class T>
void merge(T array[], T array1[], int arrayNum, int array1Num, const int & num, int & numMerges)
{
   int iStart = 0, iEnd=0, jStart=0, jEnd=0;

   //find the first sub array
   iStart = arrayNum;
   for(; arrayNum < num; ++arrayNum)
   {
      //when we found the end of the first sub array
      if(array[arrayNum] > array[arrayNum + 1])
      {
         iEnd = arrayNum;
         break;
      }
   }

   //when we reached the end of the array
   if (arrayNum == num)
      iEnd = arrayNum-1;
   ++arrayNum;

   //find the second sub array
   jStart = arrayNum;
   for(; arrayNum < num; ++arrayNum)
   {
      //when we found the end of the second sub array
      if(array[arrayNum] > array[arrayNum + 1])
      {
         jEnd = arrayNum;
         break;
```

Commented [HJ15]: Technically, this is illegal. We cannot have a variable in the size part of an array declaration.

Commented [HJ16]: This spot is expensive.

```
      }
   }

   //when we reached the end of the array
   if (arrayNum == num)
      jEnd = arrayNum-1;

   ++arrayNum;

   //if there only one more sub array rather than two
   if(iEnd == num-1)
      for (; iStart < num; ++iStart)
         array1[iStart] = array[iStart];

   //since we have reached the end, kick out
   if (arrayNum > num)
      return;

   for (; array1Num <= jEnd; ++array1Num)
   {
      if (iEnd < iStart) //end of i sublist
      {
         array1[array1Num] = array[jStart];
         jStart++;
      }
      else if (jEnd < jStart) //end of array1Num sublist
      {
         array1[array1Num] = array[iStart];
         iStart++;
      }
      // if item in array1Num sublist is less than arrayNum
      else if (array[iStart] > array[jStart])
      {
         array1[array1Num]= array[jStart];
         jStart++;
      }
      //if item in array1Num sublist is more than arrayNum
      else
      {
         array1[array1Num] = array[iStart];
         iStart++;
      }
   }

   merge(array, array1, arrayNum, array1Num, num, numMerges);
   numMerges++;

}

#endif // SORT_MERGE_H
```

Commented [HJ17]: Whoa! This is quite complex.

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;
}

/************************************
 * 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())
   {
      //std:: cout << 'b';

      //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
   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
```

lesson10.cpp

```
/***********************************************************************
* Program:
*    Lesson 10, Sorting
*    Brother Helfrich, CS 235
* Author:
*    Br. Helfrich
* Summary:
*    This is a driver program to exercise various Sort algorithms. 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 <iomanip>         // for SETW
#include <ctime>           // for time(), part of the random process
#include <stdlib.h>        // for rand() and srand()
#include "sortValue.h"     // for SortValue to instrument the sort algorithms
#include "sortBubble.h"    // for sortBubble()
#include "sortSelection.h" // for sortSelection()
#include "sortInsertion.h" // for sortInsertion()
#include "sortBinary.h"    // for sortBinary()
#include "sortHeap.h"      // for sortHeap()
#include "sortMerge.h"     // for sortMerge()
#include "sortQuick.h"     // for sortQuick()
using namespace std;

// prototypes for our test functions
void compareSorts();
```

```
void testIndividualSorts(int choice);

/******************************************
 * SORT NAME AND FUNCTION
 * This facilitates testing a number
 * of sorts
 ******************************************/
struct SortNameAndFunction
{
   const char * name;
   void (* sortInteger)(int       array[], int num);
   void (* sortValue  )(SortValue array[], int num);
};
const SortNameAndFunction sorts[] =
{
   { NULL,             NULL,          NULL          },
   { "Bubble Sort",    sortBubble,    sortBubble    },
   { "Selection Sort", sortSelection, sortSelection },
   { "Insertion Sort", sortInsertion, sortInsertion },
   { "Binary Sort",    sortBinary,    sortBinary    },
   { "Heap Sort",      sortHeap,      sortHeap      },
   { "Merge Sort",     sortMerge,     sortMerge     },
   { "Quick Sort",     sortQuick,     sortQuick     }
};

/**********************************************************************
 * MAIN
 * This is just a simple menu to launch a collection of tests
 **********************************************************************/
int main()
{
   // menu, built from the sortValues list above
   cout << "Select the test you want to run:\n";
   cout << "\t0. To compare all the sorting algoritms\n";
   for (int i = 1; i <= 7; i++)
      cout << '\t' << i << ". "
           << sorts[i].name << endl;

   // user specifies his choice
   int choice;
   cout << "> ";
   cin  >> choice;

   // execute the user's choice
   if (choice == 0)
      compareSorts();
   else if (choice >= 1 && choice <= 7)
      testIndividualSorts(choice);
   else
      cout << "Unrecognized command, exiting...\n";

   return 0;
}

/*******************************************
 * CREATE TEST ARRAYS
 * Generate test arrays for the purpose of
 * comparing sorts.  This function has one
 * client: compareSort()
 ******************************************/
void createTestArrays(SortValue * & arrayStart,
                      SortValue * & arraySort,
                      int & num)
{
   // prompt for size
   cout << "How many items in the test (10000 - 40000 are good numbers)? ";
   cin  >> num;

   // allocate the array
   arrayStart = new(nothrow) SortValue[num];
   arraySort  = new(nothrow) SortValue[num];
   if (arrayStart == NULL || arraySort == NULL)
   {
      cout << "Unable to allocate that much memory";
      return;
   }

   // fill the array with random values
   cout << "What type of test would you like to run?\n";
```

```
   cout << "   1. random numbers\n";
   cout << "   2. already sorted in ascending order\n";
   cout << "   3. already sorted in decending order\n";
   cout << "   4. almost sorted in ascending order\n";
   cout << "   5. random but with a small number of possible values\n";
   cout << "> ";
   int option;
   cin >> option;

   switch (option)
   {
      case 5:  // random but with a small number of possible values
         for (int i = 0; i < num; i++)
            arrayStart[i] = rand() % 10;
         break;
      case 4: // almost sorted in ascending order
         for (int i = 0; i < num; i++)
            arrayStart[i] = i + rand() % 10;
         break;
      case 3: // already sorted in decending order
         for (int i = 0; i < num; i++)
            arrayStart[i] = num - i;
         break;
      case 2: // already sorted in ascending order
         for (int i = 0; i < num; i++)
            arrayStart[i] = i;
         break;
      case 1: // random numbers
      default:
         for (int i = 0; i < num; i++)
            arrayStart[i].random();
   }
}


/*******************************************
 * COMPARE SORTS
 * Compare the relative speed of the various sorts
 ******************************************/
void compareSorts()
{
   // allocate the array
   SortValue * arrayStart;
   SortValue * arraySort;
   int num;
   createTestArrays(arrayStart, arraySort, num);
   if (arrayStart == NULL || arraySort == NULL)
      return;

   // get ready with the header to the table
   srand(time(NULL));
   cout.setf(ios::fixed);
   cout.precision(2);
   cout << "      Sort Name    Time       Assigns      Compares\n";
   cout << " ---------------+-------+-------------+-------------\n";

   for (int iSort = 1; iSort <= 7; iSort++)
   {
      // get ready by copying the un-sorted numbers to the array
      for (int iValue = 0; iValue < num; iValue++)
         arraySort[iValue] = arrayStart[iValue];
      arraySort[0].reset();

      // perform the sort
      int msBegin = clock();
      sorts[iSort].sortValue(arraySort, num);
      int msEnd = clock();

      // report the results
      cout << setw(15) << sorts[iSort].name                    << " |"
           << setw(6)  << (float)(msEnd - msBegin) / 1000000.0 << " |"
           << setw(12) << arraySort[0].getAssigns()            << " |"
           << setw(12) << arraySort[0].getCompares()           << endl;
   }

   // all done
   delete [] arrayStart;
   delete [] arraySort;
}
```

```
/*******************************************
 * TEST INDIVIDUAL SORTS
 * For a given sort selected by "choice",
 * feed it 100 random 3-digit integers and
 * display the results.
 *
 * To test with a smaller number of items,
 * for debugging purposes, just set the size variable
 * to a smaller value such as "size = 10;"
 *******************************************/
void testIndividualSorts(int choice)
{
   assert(choice >= 1 && choice <= 7);

   // prepare the array
   int array[] =
   {
      889, 192, 528, 675, 154, 746, 562, 482, 448, 842, 929, 330, 615, 225,
      785, 577, 606, 426, 311, 867, 773, 775, 190, 414, 155, 771, 499, 337,
      298, 242, 656, 188, 334, 184, 815, 388, 831, 429, 823, 331, 323, 752,
      613, 838, 877, 398, 415, 535, 776, 679, 455, 602, 454, 545, 916, 561,
      369, 467, 851, 567, 609, 507, 707, 844, 643, 522, 284, 526, 903, 107,
      809, 227, 759, 474, 965, 689, 825, 433, 224, 601, 112, 631, 255, 518,
      177, 224, 131, 446, 591, 882, 913, 201, 441, 673, 997, 137, 195, 281,
      563, 151
   };
   int size = sizeof(array) / sizeof(array[0]);

   // display the list before they are sorted
   cout << sorts[choice].name << endl;
   cout << "\tBefore:\t" << array[0];
   for (int i = 1; i < size; i++)
      cout << (i % 10 == 0 ? ",\n\t\t" : ", ")
           << array[i];
   cout << endl << endl;

   // perform the sort
   sorts[choice].sortInteger(array, size);

   // report the results
   bool sorted = true;
   cout << "\tAfter:\t" << array[0];
   for (int i = 1; i < size; i++)
   {
      cout << (i % 10 == 0 ? ",\n\t\t" : ", ")
           << array[i];
      if (array[i - 1] > array[i])
         sorted = false;
   }
   cout << endl;
   cout << "The array is "
        << (sorted ? "" : "NOT ")
        << "sorted\n";
}
```

Test Bed Results

```
cs235d.out:

Started program
   > Select the test you want to run:
   >     0. To compare all the sorting algoritms
   >     1. Bubble Sort
   >     2. Selection Sort
   >     3. Insertion Sort
   >     4. Binary Sort
   >     5. Heap Sort
   >     6. Merge Sort
   >     7. Quick Sort
   > > 1
   > Bubble Sort
   >     Before:   889, 192, 528, 675, 154, 746, 562, 482, 448, 842,
   >        929, 330, 615, 225, 785, 577, 606, 426, 311, 867,
   >        773, 775, 190, 414, 155, 771, 499, 337, 298, 242,
   >        656, 188, 334, 184, 815, 388, 831, 429, 823, 331,
   >        323, 752, 613, 838, 877, 398, 415, 535, 776, 679,
```

```
   >       455, 602, 454, 545, 916, 561, 369, 467, 851, 567,
   >       609, 507, 707, 844, 643, 522, 284, 526, 903, 107,
   >       809, 227, 759, 474, 965, 689, 825, 433, 224, 601,
   >       112, 631, 255, 518, 177, 224, 131, 446, 591, 882,
   >       913, 201, 441, 673, 997, 137, 195, 281, 563, 151
   >
   >    After:   107, 112, 131, 137, 151, 154, 155, 177, 184, 188,
   >       190, 192, 195, 201, 224, 224, 225, 227, 242, 255,
   >       281, 284, 298, 311, 323, 330, 331, 334, 337, 369,
   >       388, 398, 414, 415, 426, 429, 433, 441, 446, 448,
   >       454, 455, 467, 474, 482, 499, 507, 518, 522, 526,
   >       528, 535, 545, 561, 562, 563, 567, 577, 591, 601,
   >       602, 606, 609, 613, 615, 631, 643, 656, 673, 675,
   >       679, 689, 707, 746, 752, 759, 771, 773, 775, 776,
   >       785, 809, 815, 823, 825, 831, 838, 842, 844, 851,
   >       867, 877, 882, 889, 903, 913, 916, 929, 965, 997
   > The array is sorted
Program terminated successfully

Started program
   > Select the test you want to run:
   >    0. To compare all the sorting algoritms
   >    1. Bubble Sort
   >    2. Selection Sort
   >    3. Insertion Sort
   >    4. Binary Sort
   >    5. Heap Sort
   >    6. Merge Sort
   >    7. Quick Sort
   > > 2
   > Selection Sort
   >    Before:   889, 192, 528, 675, 154, 746, 562, 482, 448, 842,
   >       929, 330, 615, 225, 785, 577, 606, 426, 311, 867,
   >       773, 775, 190, 414, 155, 771, 499, 337, 298, 242,
   >       656, 188, 334, 184, 815, 388, 831, 429, 823, 331,
   >       323, 752, 613, 838, 877, 398, 415, 535, 776, 679,
   >       455, 602, 454, 545, 916, 561, 369, 467, 851, 567,
   >       609, 507, 707, 844, 643, 522, 284, 526, 903, 107,
   >       809, 227, 759, 474, 965, 689, 825, 433, 224, 601,
   >       112, 631, 255, 518, 177, 224, 131, 446, 591, 882,
   >       913, 201, 441, 673, 997, 137, 195, 281, 563, 151
   >
   >    After:   107, 112, 131, 137, 151, 154, 155, 177, 184, 188,
   >       190, 192, 195, 201, 224, 224, 225, 227, 242, 255,
   >       281, 284, 298, 311, 323, 330, 331, 334, 337, 369,
   >       388, 398, 414, 415, 426, 429, 433, 441, 446, 448,
   >       454, 455, 467, 474, 482, 499, 507, 518, 522, 526,
   >       528, 535, 545, 561, 562, 563, 567, 577, 591, 601,
   >       602, 606, 609, 613, 615, 631, 643, 656, 673, 675,
   >       679, 689, 707, 746, 752, 759, 771, 773, 775, 776,
   >       785, 809, 815, 823, 825, 831, 838, 842, 844, 851,
   >       867, 877, 882, 889, 903, 913, 916, 929, 965, 997
   > The array is sorted
Program terminated successfully

Started program
   > Select the test you want to run:
   >    0. To compare all the sorting algoritms
   >    1. Bubble Sort
   >    2. Selection Sort
   >    3. Insertion Sort
   >    4. Binary Sort
   >    5. Heap Sort
   >    6. Merge Sort
   >    7. Quick Sort
   > > 3
   > Insertion Sort
   >    Before:   889, 192, 528, 675, 154, 746, 562, 482, 448, 842,
   >       929, 330, 615, 225, 785, 577, 606, 426, 311, 867,
   >       773, 775, 190, 414, 155, 771, 499, 337, 298, 242,
   >       656, 188, 334, 184, 815, 388, 831, 429, 823, 331,
   >       323, 752, 613, 838, 877, 398, 415, 535, 776, 679,
   >       455, 602, 454, 545, 916, 561, 369, 467, 851, 567,
   >       609, 507, 707, 844, 643, 522, 284, 526, 903, 107,
   >       809, 227, 759, 474, 965, 689, 825, 433, 224, 601,
   >       112, 631, 255, 518, 177, 224, 131, 446, 591, 882,
   >       913, 201, 441, 673, 997, 137, 195, 281, 563, 151
   >
   >    After:   107, 112, 131, 137, 151, 154, 155, 177, 184, 188,
```

```
   >       190, 192, 195, 201, 224, 224, 225, 227, 242, 255,
   >       281, 284, 298, 311, 323, 330, 331, 334, 337, 369,
   >       388, 398, 414, 415, 426, 429, 433, 441, 446, 448,
   >       454, 455, 467, 474, 482, 499, 507, 518, 522, 526,
   >       528, 535, 545, 561, 562, 563, 567, 577, 591, 601,
   >       602, 606, 609, 613, 615, 631, 643, 656, 673, 675,
   >       679, 689, 707, 746, 752, 759, 771, 773, 775, 776,
   >       785, 809, 815, 823, 825, 831, 838, 842, 844, 851,
   >       867, 877, 882, 889, 903, 913, 916, 929, 965, 997
   > The array is sorted
Program terminated successfully

Started program
   > Select the test you want to run:
   >    0. To compare all the sorting algoritms
   >    1. Bubble Sort
   >    2. Selection Sort
   >    3. Insertion Sort
   >    4. Binary Sort
   >    5. Heap Sort
   >    6. Merge Sort
   >    7. Quick Sort
   > > 4
   > Binary Sort
   >    Before:   889, 192, 528, 675, 154, 746, 562, 482, 448, 842,
   >       929, 330, 615, 225, 785, 577, 606, 426, 311, 867,
   >       773, 775, 190, 414, 155, 771, 499, 337, 298, 242,
   >       656, 188, 334, 184, 815, 388, 831, 429, 823, 331,
   >       323, 752, 613, 838, 877, 398, 415, 535, 776, 679,
   >       455, 602, 454, 545, 916, 561, 369, 467, 851, 567,
   >       609, 507, 707, 844, 643, 522, 284, 526, 903, 107,
   >       809, 227, 759, 474, 965, 689, 825, 433, 224, 601,
   >       112, 631, 255, 518, 177, 224, 131, 446, 591, 882,
   >       913, 201, 441, 673, 997, 137, 195, 281, 563, 151
   >
   >    After:   107, 112, 131, 137, 151, 154, 155, 177, 184, 188,
   >       190, 192, 195, 201, 224, 224, 225, 227, 242, 255,
   >       281, 284, 298, 311, 323, 330, 331, 334, 337, 369,
   >       388, 398, 414, 415, 426, 429, 433, 441, 446, 448,
   >       454, 455, 467, 474, 482, 499, 507, 518, 522, 526,
   >       528, 535, 545, 561, 562, 563, 567, 577, 591, 601,
   >       602, 606, 609, 613, 615, 631, 643, 656, 673, 675,
   >       679, 689, 707, 746, 752, 759, 771, 773, 775, 776,
   >       785, 809, 815, 823, 825, 831, 838, 842, 844, 851,
   >       867, 877, 882, 889, 903, 913, 916, 929, 965, 997
   > The array is sorted
Program terminated successfully

Started program
   > Select the test you want to run:
   >    0. To compare all the sorting algoritms
   >    1. Bubble Sort
   >    2. Selection Sort
   >    3. Insertion Sort
   >    4. Binary Sort
   >    5. Heap Sort
   >    6. Merge Sort
   >    7. Quick Sort
   > > 5
   > Heap Sort
   >    Before:   889, 192, 528, 675, 154, 746, 562, 482, 448, 842,
   >       929, 330, 615, 225, 785, 577, 606, 426, 311, 867,
   >       773, 775, 190, 414, 155, 771, 499, 337, 298, 242,
   >       656, 188, 334, 184, 815, 388, 831, 429, 823, 331,
   >       323, 752, 613, 838, 877, 398, 415, 535, 776, 679,
   >       455, 602, 454, 545, 916, 561, 369, 467, 851, 567,
   >       609, 507, 707, 844, 643, 522, 284, 526, 903, 107,
   >       809, 227, 759, 474, 965, 689, 825, 433, 224, 601,
   >       112, 631, 255, 518, 177, 224, 131, 446, 591, 882,
   >       913, 201, 441, 673, 997, 137, 195, 281, 563, 151
   >
   >    After:   107, 112, 131, 137, 151, 154, 155, 177, 184, 188,
   >       190, 192, 195, 201, 224, 224, 225, 227, 242, 255,
   >       281, 284, 298, 311, 323, 330, 331, 334, 337, 369,
   >       388, 398, 414, 415, 426, 429, 433, 441, 446, 448,
   >       454, 455, 467, 474, 482, 499, 507, 518, 522, 526,
   >       528, 535, 545, 561, 562, 563, 567, 577, 591, 601,
   >       602, 606, 609, 613, 615, 631, 643, 656, 673, 675,
   >       679, 689, 707, 746, 752, 759, 771, 773, 775, 776,
```

```
   >       785, 809, 815, 823, 825, 831, 838, 842, 844, 851,
   >       867, 877, 882, 889, 903, 913, 916, 929, 965, 997
   > The array is sorted
Program terminated successfully

Started program
   > Select the test you want to run:
   >    0. To compare all the sorting algoritms
   >    1. Bubble Sort
   >    2. Selection Sort
   >    3. Insertion Sort
   >    4. Binary Sort
   >    5. Heap Sort
   >    6. Merge Sort
   >    7. Quick Sort
   > > 6
   > Merge Sort
   >    Before:   889, 192, 528, 675, 154, 746, 562, 482, 448, 842,
   >       929, 330, 615, 225, 785, 577, 606, 426, 311, 867,
   >       773, 775, 190, 414, 155, 771, 499, 337, 298, 242,
   >       656, 188, 334, 184, 815, 388, 831, 429, 823, 331,
   >       323, 752, 613, 838, 877, 398, 415, 535, 776, 679,
   >       455, 602, 454, 545, 916, 561, 369, 467, 851, 567,
   >       609, 507, 707, 844, 643, 522, 284, 526, 903, 107,
   >       809, 227, 759, 474, 965, 689, 825, 433, 224, 601,
   >       112, 631, 255, 518, 177, 224, 131, 446, 591, 882,
   >       913, 201, 441, 673, 997, 137, 195, 281, 563, 151
   >
   >    After:   107, 112, 131, 137, 151, 154, 155, 177, 184, 188,
   >       190, 192, 195, 201, 224, 224, 225, 227, 242, 255,
   >       281, 284, 298, 311, 323, 330, 331, 334, 337, 369,
   >       388, 398, 414, 415, 426, 429, 433, 441, 446, 448,
   >       454, 455, 467, 474, 482, 499, 507, 518, 522, 526,
   >       528, 535, 545, 561, 562, 563, 567, 577, 591, 601,
   >       602, 606, 609, 613, 615, 631, 643, 656, 673, 675,
   >       679, 689, 707, 746, 752, 759, 771, 773, 775, 776,
   >       785, 809, 815, 823, 825, 831, 838, 842, 844, 851,
   >       867, 877, 882, 889, 903, 913, 916, 929, 965, 997
   > The array is sorted
Program terminated successfully

Started program
   > Select the test you want to run:
   >    0. To compare all the sorting algoritms
   >    1. Bubble Sort
   >    2. Selection Sort
   >    3. Insertion Sort
   >    4. Binary Sort
   >    5. Heap Sort
   >    6. Merge Sort
   >    7. Quick Sort
   > > 7
   > Quick Sort
   >    Before:   889, 192, 528, 675, 154, 746, 562, 482, 448, 842,
   >       929, 330, 615, 225, 785, 577, 606, 426, 311, 867,
   >       773, 775, 190, 414, 155, 771, 499, 337, 298, 242,
   >       656, 188, 334, 184, 815, 388, 831, 429, 823, 331,
   >       323, 752, 613, 838, 877, 398, 415, 535, 776, 679,
   >       455, 602, 454, 545, 916, 561, 369, 467, 851, 567,
   >       609, 507, 707, 844, 643, 522, 284, 526, 903, 107,
   >       809, 227, 759, 474, 965, 689, 825, 433, 224, 601,
   >       112, 631, 255, 518, 177, 224, 131, 446, 591, 882,
   >       913, 201, 441, 673, 997, 137, 195, 281, 563, 151
   >
   >    After:   107, 112, 131, 137, 151, 154, 155, 177, 184, 188,
   >       190, 192, 195, 201, 224, 224, 225, 227, 242, 255,
   >       281, 284, 298, 311, 323, 330, 331, 334, 337, 369,
   >       388, 398, 414, 415, 426, 429, 433, 441, 446, 448,
   >       454, 455, 467, 474, 482, 499, 507, 518, 522, 526,
   >       528, 535, 545, 561, 562, 563, 567, 577, 591, 601,
   >       602, 606, 609, 613, 615, 631, 643, 656, 673, 675,
   >       679, 689, 707, 746, 752, 759, 771, 773, 775, 776,
   >       785, 809, 815, 823, 825, 831, 838, 842, 844, 851,
   >       867, 877, 882, 889, 903, 913, 916, 929, 965, 997
   > The array is sorted
Program terminated successfully

No Errors
```

Grading Criteria

Criteria	Exceptional 100%	Good 90%	Acceptable 70%	Developing 50%	Missing 0%	Weight	Score
Bubble Sort	The sort is perfectly implemented according to the design	Testbed for this sort runs without error	A minor bug exists but style and code quality are excellent	The essense of the algorithm is properly represented	No attempt was made	5	4.5
Selection Sort	The sort is perfectly implemented according to the design	Testbed for this sort runs without error	A minor bug exists but style and code quality are excellent	The essense of the algorithm is properly represented	No attempt was made	10	10
Insertion Sort	The sort is perfectly implemented according to the design	Testbed for this sort runs without error	A minor bug exists but style and code quality are excellent	The essense of the algorithm is properly represented	No attempt was made	10	10
Binary Sort	The sort is perfectly implemented according to the design	Testbed for this sort runs without error	A minor bug exists but style and code quality are excellent	The essense of the algorithm is properly represented	No attempt was made	5	5
Heap Sort	The sort is perfectly implemented according to the design	Testbed for this sort runs without error	A minor bug exists but style and code quality are excellent	The essense of the algorithm is properly represented	No attempt was made	40	40
Merge Sort	The sort is perfectly implemented according to the design	Testbed for this sort runs without error	A minor bug exists but style and code quality are excellent	The essense of the algorithm is properly represented	No attempt was made	40	36
Quick Sort	The sort is perfectly implemented according to the design	Testbed for this sort runs without error	A minor bug exists but style and code quality are excellent	The essense of the algorithm is properly represented	No attempt was made	30	30
Total							135.5

Commented [HJ18]: Well done.