**Project Description: Binary Search Tree Letter Processor**

In this program, you will implement a binary search tree. Your implementation will be very general, such that the tree can store **any**class that appropriately inherits from the class stored in the node. This improves the object-orientation of the class and you will likely reuse your binary search tree later this quarter.

While the implementation will be very general, in this assignment it will be used to store characters (in ASCII order) from a document (read from a file by my driver below). The binary search tree will maintain a count on the number of times each character appears in the document and produce a table of the occurrences. Note that capitalization matters - do not change the case of any character. Each different character that appears in the document should be stored exactly once in a single binary search tree, along with the count on the number of appearances. You may ignore white space, if you want, but not punctuation. (This makes file input easier.)

**Comparable**

You will be responsible for writing two classes, but not the driver. The first class should very simple. It will just encapsulate a character. We will call this class Comparable in homage to the Java Comparable interface (although Comparable is not a keyword in C++; it merely serves as a descriptive name). Your Comparable class should provide the following methods (mostly by reusing char methods):

* operator==, operator!=, operator<, operator>: these are straightforward and just reuse the same char operators. This will result in the characters being sorted into ASCII order.
* operator<<: this should use char::operator<< - make sure to use syntax that allows chaining such as: cout << c1 << " " << c2 << endl;
* operator>>: this should read in a char from an istream. It can use char::operator<< or the istream::get() method.

Note that no constructors, destructor, or operator= should be necessary. You can rely on the defaults provided by the compiler, since you will not use dynamic memory within this class. Also note that these are the **only**methods that your class should require. This class should be small and simple (but still documented appropriately).

**Search Tree**

We will call the binary search tree class SearchTree. The nodes in your SearchTree class **must** have the following form:

struct Node { // The name is not important, but don't use charNode

Comparable \*item; // You MUST use a pointer to the Comparable data

... }; // Count, pointers to left and right children

The mandatory use of a pointer to the data is so that this data structure can later support an abstract class (and a hierarchy of concrete classes that inherit from it). You must also implement the following methods. In addition, you may use as many private methods (e.g., for recursion) as you wish.

* Constructors and destructor
  + default constructor
  + copy constructor (deep copy)
  + destructor
* Overloaded operators
  + operator= : assignment of one tree to another (deep copy)
  + operator== : two trees are equal only if they have the same data, the same structure, and the same number of occurrences for each comparable
  + operator!=
  + operator<< : output the frequency table in sorted order. This should be an inorder traversal of your tree. Your output should look similar to:

. 1  
a 3  
e 1  
f 1  
h 7  
...

* Mutator functions
  + bool insert(Comparable \*) :  inserts a Comparable into the tree **or** increments the count if it is already in the tree. This method should return false if the Comparable is found (and, thus, not inserted). Ownership of the memory is transferred to the tree only if the object is not found in the tree (in which case it must later be deallocated by the tree).
  + bool remove(const Comparable &) :  removes one occurrence of a Comparable from the tree. If it is the last occurrence, remove the node. Return false if the Comparable is not found.
  + void makeEmpty() :  removes and deallocates all of the data from the tree.
* Accessor functions
  + const Comparable \*retrieve(const Comparable &) const :  finds a Comparable in the tree using a key stored in the parameter. This is useful for cases where the Comparable stores more than the sorting key. Return nullptr if not found.
  + int height(const Comparable &) const :  returns the height of the node storing the Comparable in the tree. A leaf has height 0. Return -1 if the Comparable is not found.