CS 2510 Exam 3 – Fall 2012

Name: ____________________

Student Id (last 4 digits): ____________________

• Write down the answers in the space provided.
• You may use all parts of the Java language we have learned. If you need a method and you don't know whether it is provided, define it. You do not need to include the curly braces for every `if` or every `else`, as long as the statements you write are correct in standard Java.
• For tests you only need to provide the expression that computes the actual value, connecting it with an arrow to the expected value. For example `s.method() -> true` is sufficient.
• Remember that the phrase "develop a class" or "develop a method" means more than just providing a definition. It means to design them according to the **design recipe**. You are *not* required to provide a method template unless the problem specifically asks for one. However, be prepared to struggle if you choose to skip the template step.
• We will not answer *any* questions during the exam.

Good luck.

Problem	Points	/
1A		/ 8
1B		/ 4
1C		/ 7
1D		/ 7
1E		/ 4
1F		/ 4
1G		/11
1H		/ 5
1I		/10
1J EXTRA		/ 8
Total		/60

Try to solve all problems other than G, I, and J, then work on those, if you have the time.

30 Points

Problem 1

In decoding secret messages it often helps to know the frequency whith which each letter of alphabet occurs in a typical text.

A. 8 POINTS

We start by recording the text as an `ArrayList` of `String`s, where each `String` is just one character.

Design the method `allSingles` that consumes an `ArrayList` of `String`s and mutates this list so that each item is a `String` of length 1. If the item was `null`, it is replaced by a `String` that represents one blank space. It the item was a `String` longer than one, it will be replaced by a `String` that is the first letter of the original `String`.

B. 4 POINTS

We now want to compute the frequency of all letters in the given `ArrayList` of `String`s of length one.

These frequencies are typically shown as a histogram that records how often each character (including the space) occurs in the given text.

We decided to use the following the data representation for the histogram data, so that you can easily tell how often each letter appears in the text. The histogram is then a simple `ArrayList` of item of the type `LFreq`. The number of entries in the `ArrayList` is equal to the number of distinct letters in the text we are working with. For English language, this would be `27`, representing all letter of alphabet and the blank space — if we ignore all other special characters and all digits.

Class LFreq (Letter Frequency):

```
+----------+
| LFreq    |
+----------+
| int freq |
| String s |
+----------+
```

Make an example of the `ArrayList` that represents the histogram you would get from the following text:

`go on and do great things`

You can include only the letters that occur in this text - and the order does not matter. Make sure you include the blank space character.

Hint: You need to build the `ArrayList` inside a `void init()...` method.

C. 7 POINTS

Design the (*override*) method `equals` for the class `LFreq` that returns `true` if the two `String`s match, even if the `freq` are different.

Make sure you do everything that is required to properly override the Java `equals` method.

This page is for your work

D. 7 POINTS

Design the method (in some `Algorithms` class) `computeHisto` that computes the histogram for a given text. It consumes the `ArrayList` of `String`s of length one and produces an `ArrayList` of `LFreq` items.

Alternately, instead of an `ArrayList` of `LFreq` items, you may produce a `HashMap` of entries where the `String` will be the key and the frequency will be the `value` (see the hint 2).

Hint 1: You may assume that all letters of alphabet will be represented, as will be the space. Alternately, you may build the `ArrayList` that would contain only the letters in the text you are now analyzing (such as our example text `go on and do great things`).

Hint 2: You may choose to built first a `HashMap` of entries where the `String` will be the key and the frequency will be the `value`. If you do this, you can stop there, you do not need to produce the `ArrayList`.

E. 4 POINTS

We need to find out which letter has the highest frequency. To do so, we need a `Comparator`.

Design the class that implements the `Comparator` interface by comparing items of the type `LFreq`, the ordering based on the letter frequencies.

F. 4 POINTS

The encoding for each letter in our `String` is given by a binary tree we can build, once we know the frequencies of all letters:

```
    +----------+
    | AHT      |<---------+-+
    +----------+          | |
    | int freq |          | |
    +----------+          | |
        / \               | |
        ---               | |
         |                | |
    --------------        | |
    |             |       | |
+----------+ +----------+ | |
| LFreq    | | Node     | | |
+----------+ +----------+ | |
| String s | | HT left  |-+ |
+----------+ | HT right |---+
             +----------+
```

(We explain later how this kind of trees are built.)

Show the representation of the following two trees as an instance of `AHT`, as they would appear in the `Examples` class.

(The first tree represents an optimal encoding for the `String` *cat at bat*, the second one represents an optimal encoding for the `String` *here there*)

```
      10                  10
     /  \                /  \
    6    4              4    6
   / \  / \            / \  / \
 t3 a3 2  sp2        r2 h2 e4  2
       / \                    / \
      c1 b1                  t1 sp1
```

Each leaf node includes the frequency of that letter in the examined text.

Each path from the root of the tree to a leaf can be described as a sequence of choices: *left* or *right*.

A path from the root to the leaf is then a secret encoding of the letter in the leaf.

So, for the first sample tree shown, the path *'left right'* leads to the letter *a*, the path *'right left right'* leads to the letter *b*, and the path *'right right'* leads to *sp*, i.e. the space (that would be represented as `String s = " "`

G. 11 POINTS

We now describe such path from the root to a leaf as an `ArrayList` of `Boolean` values, with `true` representing the *left* choice and `false` representing the choice of moving to the *right*.

Design the method `whereTo` for the classes that implement the interface `AHT` that consumes an `ArrayList` of `Boolean` values that represent the path from the root of the tree to a leaf, and returns the `String` value of the `Leaf` reached by this path.

If the path does not lead to a `Leaf` throw an `RuntimeException` with a message `"Invalid path"`.

... This page is intentionally left blank ...

H. 5 POINTS

The tree we have built (called the *Huffman tree*) has the property that the value of each node is the sum of the values of its left and right children.

Design the method `validTree` for the `AHT` class hierarchy that verifies that this tree is a valid *Huffman tree*.

I. 10 POINTS

If we use `0` for `false` amd `1` for `true` then a `String s = "0010''` represents the letter `t` and the `String s = "0010''` represents the letter `t`.

Design the method `findPath` for the `AHT` classes that consumes a `String` of length one and produces a `String` that represents this encoding and ends with the given `String`.

So, for the input `"t''` your first tree would produce the `String "01t"`, your second tree would produce the `String "110t"`.

J. 8 POINTS EXTRA CREDIT

To design our encoding, David Huffman tells us that we should save the histogram data in a priority queue where the highest priority item is the one with the lowest frequency of occurrence, and then convert it into a *Huffman* tree in a clever way that then defines the encoding for every character in our original `String`.

The only thing you need to know at this point is that the priority queue built from the histogram will contain as data instances of `AHT` and will use the `freq` field to determine the priority (defined in some class `FreqComp` that extends `Comparator<AHT>`). The lowest `freq` == the highest priority.

Design the method `buildTree` that consumes the priority queue we have built and produces the `AHT` that represents the encoding. It works as follows:

- if the size of the priority queue is empty, no tree can be built and it throws an exception.
- if the size of the priority queue is 1, the priority queue contains just one `AHT`, and the method removes it from the priority queue and returns it.
- otherwise, the method removes the two items with the highest priority and adds to the priority queue a new `AHT` that has the two removed `AHT`s as its left and right subtrees.

The documentation for the needed methods for the Java`PriorityQueue` is attached.

The method `buildPQ` shown below has been used to build the initial priority queue.

```
// insert the letter frequency data into a priority queue
// produce an priority queue of LFreq data
public PriorityQueue<AHT> buildPQ(ArrayList<LFreq> lfreqList){

  PriorityQueue<AHT> pq = new PriorityQueue<AHT>(20, new FreqComp());

  for (LFreq lf: lfreqList){
    pq.offer(lf));
  }
  return pq;
}
```

... This page is intentionally left blank ...

```
*** class ArrayList<T> ***

//remove all elements from this list
void clear()

// add the given element at the end of this list
boolean add(T t)

// add the given element at the given index, shifting all items after
// to the right
void add(int index, T t)

// Returns the element at the specified position in this list
T get(int index)

// Returns true if this list contains no elements
boolean isEmpty()

// Removes the element at the specified position in this list
T remove(int index)

// Returns the number of elements in this collection
int size()


*** class String ***

// Returns the length of this string
int length()

// Tests if this string ends with the specified suffix
boolean endsWith(String suffix)

// Returns a new string that is a substring of this string
// The substring begins at the specified beginIndex
// and extends to the character at index endIndex - 1
String substring(int beginIndex, int endIndex)
```

```
*** class HashMap<K, V> ***

// Returns the value to which the specified key is mapped,
// or null if this map contains no mapping for the key.
V get(Object key)

// Associates the specified value with the specified key in this map.
// Returns the previous value associated with key,
// or null if there was no mapping for key or the key mapped to null
V put(K key, V value)

//Returns true if this map contains a mapping for the specified key.
boolean containsKey(Object key)

*** class PriorityQueue<T> ***

// the constructor that reserves the given capacity
// that orders the elements according to the specified comparator
PriorityQueue(int initalCapacity, Comparator<T> comparator)

// Inserts the specified element into this priority queue
boolean add(T t)

// Inserts the specified element into this priority queue
boolean offer(T t)

// Retrieves, but does not remove, the head of this queue,
// or returns null if the queue is empty
T peek()

// retrieves and removed the head of this queue
// or returns null if the queue is empty
T poll()

// retrieves and removed the head of this queue
// or returns null if the queue is empty
T remove()

// Returns the number of elements in this collection
int size()
```