CS2133: Computer Science II
Assignment 6

Prof. Christopher Crick

1 Class Design (20 points)

Design a class hierarchy (classes and/or interfaces) to support a program dealing with geographic objects. Support the following classes (at least):

- Countries
- States/Provinces
- Cities
- Boundary Segments
- Rivers

Support the following operations, where applicable, plus any others that seem useful (arguments have been omitted, but you will have to include them):

- area()
- capital()
- getCities()
- getCountry()
- distance() – between cities
- boundaryLength() – total length of boundary
- neighbors() – objects sharing boundaries
- borderOf() – the countries/states this separates

Write out the class definitions, instance variables and method definitions. Some of the instance variables should probably be various kinds of Collections. *You do not need to implement the methods*, but you should include comments inside each method that describe the approach you would take (alternately, you can actually implement them – that might be simpler for some methods). Use interfaces and superclasses where appropriate. Supply javadoc comments for all classes, interfaces, and methods. The system you write should compile, even if it doesn't actually work (because the methods are just stubs with comments). Identify all of the classes as belonging to the package

"geography", and put the .java files in a directory called geography, so that javadoc functions properly. Note that this means that the compiler must be run from outside the geography directory, and you should type "javac geography/*.java", say, to compile your code. This is annoying, and one of the reasons I dislike Java's package system. (You would do the same thing to run a program in a package, but in this particular case you are not writing a program, so the issue won't come up.)

Note: This problem is deliberately openended. Don't panic! Be creative!

Extra credit: Be *especially* creative!

2 Permutations (30 points)

Implement a class that works like an iterator and generates all possible permutations of a list. It cannot actually *be* an Iterator, because the official Java Iterator interface looks like the following:

```
public interface Iterator<E> {
    public boolean hasNext();
    public E next();
    public void remove();
}
```

The next() method of a Java iterator returns an element of a collection, but our permutation generator must return a whole list. Nevertheless, we will adopt the mechanics of iterators. Create a Permutations object that implements the following three methods (at least):

```
public class Permutations<E> {
    public Permutations(List<E> list);
    public boolean hasNext();
    public List<E> next();
}
```

Notice the difference? We should be able to call the constructor with an arbitrary list. Then, as long as hasNext() returns true, we should be able to call next() and get a new permutation of our list.

The easiest way to generate permutations is (not surprisingly) recursively. The algorithm for *creating* a Permutations object is as follows:

- (Base case) If I am a Permutations object of list length 0, do nothing, except to note that I should always return false when hasNext() is called.
- (Recursive case) Remove and remember the first element (**c**) from the list.
- Create and remember a new Permutations object (**P**) with the leftover list.
- Obtain and remember the first permutation (**L**) from this new object, or an empty list if it has none (because it is size 0).
- Initialize an index counter (**i**) to 0.

Each time the next() method is called on a Permutations object, it should do the following:

- Return a copy of **L** with **c** inserted at position **i**. Increment **i**.
- Once **i** becomes too large, set **L** to **P**.next() and reset **i** to 0.
- If **P** has no next permutation, then this object is finished as well. hasNext() should return false from here on out.

Here's how it works for the list [0, 1, 2]:

- For each permutation of [1, 2],
- Insert 0 into each position in the list.

Thus successive calls to next() return [0, 1, 2], [1, 0, 2], [1, 2, 0], [0, 2, 1], [2, 0, 1] and [2, 1, 0]. After this last list is returned, hasNext() should return false.

Hint: The list you return should be a newly-created one, either a LinkedList or an ArrayList (your choice), with elements copied over. Don't try to use the same list that you were given in the constructor, because you'd be disassembling it and reassembling it at multiple levels of recursion, and it's almost impossible to keep it straight and do it right.

Extra Credit: You should really think about doing this one, because it's a great way to test your Permutations object. Use your Permutations object to implement the exponential NP sort that we talked about in class. Generate all of the permutations of a list in turn, checking each one to see if it's sorted. Stop when you find the sorted list. How long a list can you sort using this algorithm, before the time it takes becomes intolerable?

3 Huffman encoding and decoding (50 points)

Your final major program for CSII will be to create a pair of command-line programs to compress and decompress arbitrary files. This will require some bit-level operations, creating a Comparable binary tree class that will be used to create bit encodings, and using various Collections classes for counting, sorting and mapping.

Huffman codes have been around since the early fifties, and are still used for data compression. Modern compression methods use Huffman codes as part of the algorithm, and then layer other compression techniques on top. Huffman codes are general-purpose – they can be used to compress any arbitrary byte stream – though they are generally less efficient then special-purpose approaches that apply only to specific kinds of data.

The basic idea behind Huffman codes is to represent bytes as variable-length bit strings, rather than having every byte be represented by eight bits. If commonly-occurring bytes require only two or three bits instead of eight, while very uncommon bytes are represented with ten or fifteen, it adds up to a win in file size.

Coding and decoding Huffman codes is based on building a binary tree structure known as a Huffman tree. The leaves of the tree represent the bytes of the original file. The path through a tree from leaf to root (for encoding) or from root to leaf (for decoding) yields a sequence of bits that code for that particular byte. The tree is built in such a way that common bytes are located near the root, while rare ones are located far down the tree. The easiest way to illustrate the process is with an example.

Suppose the byte sequence we wish to encode is [-35, 41, -35, 41, 116, -35, 22, -35, 116] (bytes in Java are always signed, so they range in value from -128 to +127). The first step is to count the number of occurrences of each byte value: [-35:4, 22:1, 41:2, 116:2]. Now create Huffman leaves for each byte, putting them into a priority queue sorted according to their frequency count. Then repeatedly pull pairs of nodes out of the priority queue and connect them via a new parent node. This node's value should be the total of the values of its children. Put the node back into the priority queue, and continue the process (illustrated in Figure 1) until you have a single node. You now have a Huffman tree built according to byte frequency.

To encode a byte, start at the leaf of the tree represented by that byte and traverse up the tree until you reach the root. Every branching where you came from the left is represented by a '0', while every branching where you came from the right is a '1' (make sure you build the codes up in the correct order, from right to left). Thus the bit strings for our example are: [-35:0, 116:10, 22:110, 41:111]. Notice how common values are represented with short strings and uncommon ones with long, just as we intended. The bit sequence of our original byte string (9 bytes of 8 bits each) was 72 bits long; our encoded string is 17 bits: 01110111100110010.

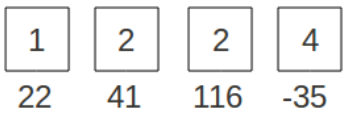
Decoding a string of bits is simply reversing the process. Starting from the root of the tree and taking each bit in turn from the bit string, follow the corresponding branch through the tree until you end at a leaf. The byte represented by that leaf is the one you want. In our example, the first bit of the code is 0. We take the left branch from the root and end up at the -35 leaf. There's our first byte. The next bit is a 1; we take the right branch at the root and we are not yet at a leaf. The next bit is 1; we take the right branch again. Still no leaf. The next bit is 1 again, and now we've arrived at the 41 leaf, our second byte. Starting from the root again, the next bit is 0, which leads us to -35, our third byte. And so on.

The whole idea for the encoder, then, is to read in a file, construct a Huffman tree based on the frequency of bytes in a file, run through each byte in the file and figure out its Huffman bit sequence, concatenate all the bits together, and save the tree and the bit sequence to disk in a new, smaller file. Decoding is basically the reverse: read in the Huffman tree and bit sequence, then use the tree to decode the sequence into an array of bytes. Save the bytes to disk and you have reconstructed the original file.

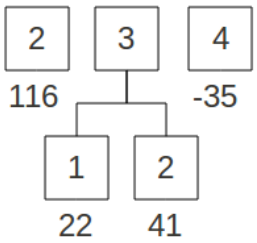
One more thing to worry about: how to go from a sequence of bits to a byte, and vice versa? We will have a lecture about bit twiddling and bitwise logical operations Individual bits can be manipulated and extracted by ANDing and ORing with a single bit in a particular position. I suggest that you take a look at this code and figure out how it works, but in any case, I'm providing it for your use (Figure 2). Notice that we're using integers to represent bits – 32 times more space than we need! But it's the easiest alternative.

The first step is to read in all of the bytes of a file (specified as a command line argument), using a FileInputStream. You can call the stream's available() method to find out how big your byte array must be, and then simply call the read method with your byte array as an argument.

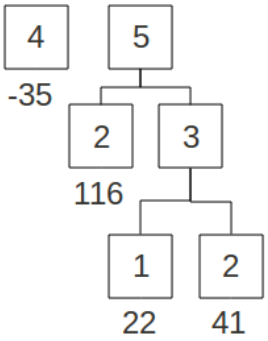
Now construct your Huffman leaf nodes. This class should contain a pointer to a parent and two children (labeled "zero" and "one"). It should also have a variable to represent the node's frequency count, and one to represent a particular byte. Leaf nodes will have null children and a filled-in byte variable, internal nodes will have null byte variables, and the root of the tree will have a null parent. Remember that this class must implement Comparable, so that it can be used inside a PriorityQueue. I suggest creating 256 Huffman leaves, one set to each possible byte, and with initial counts of zero. These can be stored in a HashMap, keyed by bytes.



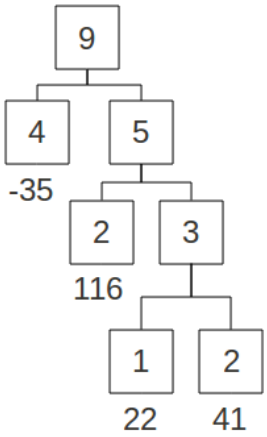
Step 1: Leaves sorted in queue



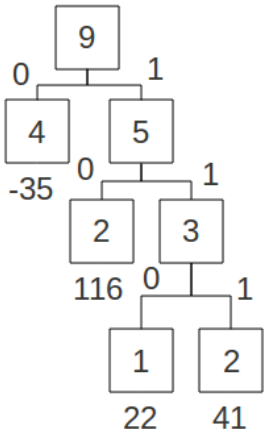
Step 2: Lowest values combined, totalled and reinserted into queue



Step 3: Lowest values combined, totalled and reinserted into queue



Step 4: Queue has one single element -- the whole Huffman tree



Step 5: Tree can be used for encoding and decoding.
Paths left and right through the tree correspond to 1s and 0s.

Figure 1: The process of building a Huffman tree.

```
import java.util.*;

public class Twiddle {

  public static byte[] bitsToBytes(List<Integer> l) {

    byte[] toReturn = new byte[l.size() / 8];
    int index = 0;
    for (int i = 0; i < l.size(); i += 8) {
      for (int j = 0; j < 8; j++) {
        toReturn[index] = (byte)(toReturn[index] | (l.get(i+j) << (7-j)));
      }
      index ++;
    }
    return toReturn;
  }

  public static List<Integer> bytesToBits(byte[] b) {

    ArrayList<Integer> toReturn = new ArrayList<Integer>();
    for (int i = 0; i < b.length; i++) {
      for (int j = 7; j >= 0; j--) {
        toReturn.add((b[i] & (1 << j)) >> j);
      }
    }
    return toReturn;
  }
}
```

Figure 2: The Twiddle class.

Proceed byte by byte through your file data and increment the count of the corresponding Huffman leaf each time you encounter a particular byte. After you have done so, each Huffman leaf knows its frequency within the file. Add all of the leaves to a PriorityQueue (a slight optimization is to add only those leaves with a count greater than zero).

Construct the tree by removing two nodes from the PriorityQueue, combining them with a new parent node, and adding the parent node back into the queue, as illustrated in Figure 1. Once only one node is left, you have the fully-built Huffman tree.

For each byte in your file in turn, retrieve the corresponding Huffman leaf from the HashMap, and obtain the bit string by traversing up the tree. I suggest writing a recursive method within your Huffman node class to do so. These bit sequences, when attached one after the other in a list, can then be turned back into an array of bytes. **Warning:** Think about what happens if your bit string is not evenly divisible by 8. You will need to pad your bit string so that it is. You will also have to devise a mechanism so that this bit padding does not confuse the decoder.

Now you can write your compressed file to disk using an ObjectOutputStream. Write your Huffman tree, any other information you need (like perhaps the length of the file), and then the byte array. Call it the same name as the original source file, but with the suffix ".huff" attached.

Decompression works much the same, in reverse. Create an ObjectInputStream, read in the Huffman tree and any additional information you saved, then the array of bytes. You can use the available() method (after you've read in the objects) to see how large the array needs to be, and the readFully() method to get a hold of the bytes. Turn the bytes into a bit stream, and then use the Huffman tree, starting from the root, to figure out which byte is represented by the bit sequence. Again, a recursive decoding method is recommended. Create a byte array to hold each new byte as you decode it, until you have decoded the entire bit stream (and handled the padding appropriately). Write this file to disk.

Turning in

You know the drill. You should have a number of .java files from Problem 1, and they should be in a package directory so that javadoc can be run on them. Call the package (and the directory) "geography" – please don't follow the Java standard and call it something absurd like "edu.okstate.cs.cs2133.assn6.geography"! Problem 2 will have a file called Permutations.java, at the least. Problem 3 will have Huff.java (the encoding program), Puff.java (the decoding program), your Huffman tree class (perhaps called HuffmanNode.java or something like that), and my Twiddle.java file (or one of your own), as well as any other classes you write. Zip all of your .java files into a zip file called assignment_6_your_name.zip and upload it to the Dropbox at oc.okstate.edu. Ensure that everything can be compiled and run from the command line. This assignment is due Wednesday, November 16, at noon.