COSC-211: DATA STRUCTURES
HW5: HUFFMAN CODING

Due Thursday, March 8, 11:59pm

Reminder regarding intellectual responsibility: This is an individual assignment, and the work you submit should be your own. Do not look at anyone else's code, and do not show anyone your code (except for me and the course TAs).

1 Introduction

Internally, all data is stored in a computer in binary—that is, using a combination of 0's and 1's. When you write a text file, each character is represented using a distinct binary string. There are many different ways of mapping a printable character (for example, the letter `A`) to a binary string. In the standard ASCII format, characters are represented as 8-bit (a bit is a "binary digit") string. The letter `A` is `01000001`, `B` is `01000010`, and so on. There are ASCII representations for symbols other than letters; for example, the symbol `]` is `01011011`. Using this representation, we can express the text `amherstmammoths` as the binary string `01100001 01101101 01101000 01100101 01110010 01110011 01110100 01101101 01100001 01101101 01101101 01101111 01110100 01101000 01110011` (we wouldn't actually write the spaces, they are just here to delimit the individual characters from our original text).

This seems an awfully long representation, given that there are only 8 distinct characters in the string `amherstmammoths`. We could have done better if we had used a 3-bit code for each character: `000` for `a`, `001` for `e`, and so on. Then we would have ended up with the binary string `000 011 010 001 101 110 111 011 000 011 011 100 111 010 110`, which is much shorter than the original version!

In 1951, David Huffman (then a doctoral student at MIT) came up with a better idea than using 8-bit strings for all characters: what if we used the *frequency* with which each character appears in our text to choose the binary representation and its length? That way, we could use shorter binary strings to represent characters that appear more frequently (like `m`, in our example) and longer binary strings to represent characters that appear less frequently. Huffman's idea, known as *Huffman coding*, has become an important tool in data compression (that is, using fewer bits to represent the same amount of data).

In this assignment, you will write a program that compresses and decompresses text files using Huffman's algorithm. The goals are for you to get practice working with linked objects and trees and for you to see an application of some of the data structures we've discussed in class.

2 Huffman Coding

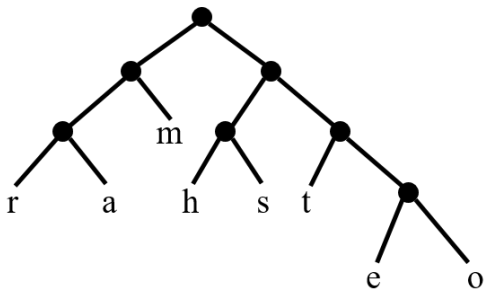
The idea behind Huffman coding is that characters that appear more frequently in a text should be represented using fewer bits than characters that appear less frequently. To that end, the first step

in producing a Huffman code is counting how many times each character appears in the text. For example, in our example string `amherstmammoths`, we would produce the following frequency table:

Character	Frequency
a	2
e	1
h	2
m	4
o	1
r	1
s	2
t	2

Based on this, we know that we want the binary encoding for the letter `m`, which appears 4 times, to be shorter than the binary encoding for the letter `r`, which appears only once.

The way that we'll accomplish this is by building a binary tree. Each leaf will correspond to a different character, and the path from the root to the leaf will give us the binary encoding for the character at that leaf. Some leaves will be "higher up" (i.e., closer to the root) than others, and these will have shorter encodings than leaves that are farther from the root. Our goal will to be to build the tree in such a way that leaves corresponding to more frequent letters are closer to the root than leaves corresponding to less frequent letters. For example, a tree based on the character frequencies in `amherstmammoths` might look like:



We read an encoding for a particular character off this tree by tracing the path from the root to a leaf. Every time we take a left branch, we add a `0` to the end of our encoding, and every time we take a right branch, we add a `1` to the end of our encoding. For example, to reach the letter `h` from the root we go right (`1`), then left (`0`), then left again (`0`), so our encoding for `h` is `100`. Notice that in this tree the letter `m`, which has the highest frequency in our original text string, also has the shortest path back to the root. There are many internal nodes in our tree, each of which has exactly two children, and none of which stores its own letter.

To create this tree, we start off by creating a new tree node for each of our letters, and adding all of these nodes to a priority queue. Here a node's priority level is the frequency of its corresponding letter in the original text. Unlike the priority queues we discussed in class, in this priority queue

a *lower* frequency will correspond to a *higher* priority. We then repeat the following until there's only one node left in the priority queue:

1. Remove the two highest-priority (i.e., lowest frequency) nodes from the priority queue.
2. Create a new internal node whose children are the two nodes that were just removed, and whose priority level is the sum of its children's priority levels.
3. Add this new node back into the priority queue.

The one node remaining in the priority queue at the end of this process is the root of the Huffman tree. I encourage you to run through this by hand to convince yourself of what it does and to confirm that the string `amherstmammoths` does indeed give you the tree I've drawn above (or something similar; your result might not be identical depending on the order in which you insert elements with equal priority).

Once we have created this tree, we can use it to encode new text strings that use the same set of characters. We can also use it to decode compressed binary strings that have been encoded using the same tree.

3 Your Tasks

Begin by downloading some starter code:

```
$ wget -nv -i https://goo.gl/nrQjWK
```

You should now have two Java files:

- `Node.java` contains the starter code to build a Huffman tree. There are fields to store the node's `symbol` (if it is a leaf), the `count` of the number of times that symbol appeared in the original text, and the `left` and `right` children (if it is an internal node). I have also included a constructor that takes a `char` and an `int` as inputs and sets the `symbol` and `count` accordingly.
- `HuffmanCode.java` contains the scaffolding code in which you will fill in methods to complete this assignment.

3.1 Implementation

Your job is to use this starter code to write a program that encodes and decodes files using Huffman trees. To start off, you'll need a priority queue:

1. Copy your `PQHeap.java` class from HW4. Update this class so that it is now a heap of `Node` objects, and so that it is a min-heap (meaning that the highest-priority element is the one with the *lowest* `count`). You also no longer need to implement the `PriorityQueue` interface.

You will also need to fill in the following methods in `HuffmanCode.java`:

2. `getFrequencies()`: the input should be the name of a text file, and the output should be a length-127 array containing the frequency of each character. (Note: every `char` in Java is really an `int`. Capital letters `A` to `Z` are `ints` 65-90 and lowercase letters `a` to `z` are `ints` 97-122. You can get a `char`'s `int` representation by casting the `char` to an `int`.) I recommend using Java's `FileReader` class to read the input file; you can look at the `readMessage()` method in `HuffmanCode.java` for an example of how to use this.

3. `makeTree()`: the input should be an array of character frequencies, and the output should be a `Node` that is the root of the tree constructed as described in Section 2. You should include all 127 characters in the tree, even if their frequency was 0 in the original text file.

4. `createCodes()`: the input should be an array (initially empty) that is to be filled in with each character's corresponding binary encoding and the tree from which to construct these encodings. (Hint: use recursion!)

5. `encode()`: the input should be a `String` containing the message you wish to encode and an array containing the mappings from character to binary string. The output should be a `String` containing the Huffman-encoded representation of the input.

6. `decode()`: the input should be a binary `String` containing an encoded message and a `Node` that is the root of a Huffman tree. The output should be a `String` containing the decoded text version of the input message.

You can use the `readMessage()` method, which I have provided for you, to read a text file character-by-character and store it in a `String`.

3.2 Encoding and Decoding

Once you have written your code, you're ready to use it to encode and decode some messages. I have provided you with several long text files:

- `telltaleheart_poe.txt` contains the short story "The Telltale Heart," by Edgar Allan Poe.
- `amodestproposal_swift.txt` contains the essay "A Modest Proposal," by Jonathan Swift.
- `uglyduckling_andersen.txt` contains the story "The Ugly Duckling," by Hans Christian Anderson.

I obtained the three above files from Project Gutenberg (https://www.gutenberg.org/), a database of works that are in the public domain. This is a great resource for finding books and stories that were written before 1923, and all three of the selections I've included with this assignment are (in my opinion) entertaining reads. These files are good examples to use as the input to *create* a Huffman tree: they are sufficiently long that you get a pretty good distribution of letters.

7. For each of the three above text files, use the code you wrote above to build a Huffman tree and encode the text contained in the file. In a comment at the top of your `HuffmanCode.java` file, answer the following question:

   (a) How much space does the Huffman-coded version of the file require relative to the original uncompressed file? (That is, for each file, what is `(num bits with Huffman coding)/(num bits without Huffman coding)`?)

I have also included three *encoded* text files: `codedmessage1.txt`, `codedmessage2.txt`, and `codedmessage3.txt`. Each of these three messages was produced using the Huffman tree generated by one of the long text files (it is up to you to figure out which one, if that matters). Do the following:

8. Decode each of the messages. Create a new text file called `decodedmessages.txt` (in **exactly** that format), and put the three decoded messages into that text file.

9. In a comment at the top of your `HuffmanCode.java` file, answer the following question:

   (b) Do all longish English text files produce the same Huffman tree? How do you know?

4 Submit your work

Make sure you **test your code thoroughly** before submitting it. Code that does not compile will not receive credit.

Submit your `HuffmanCode.java`, `PQHeap.java`, and `decodedmessages.txt` using either the submission web site or (from `remus/romulus`) the command line:

```
$ cssubmit HuffmanCode.java PQHeap.java decodedmessages.txt
```

This assignment is due on Thursday, March 8, 11:59pm.