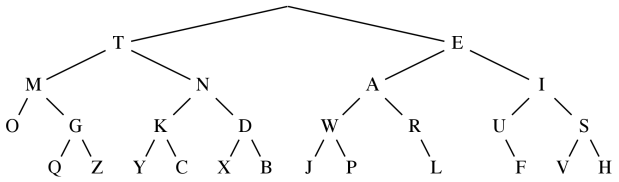
[P10] Huffman Coding

Due: 12:01am, Monday December 9th.

Description

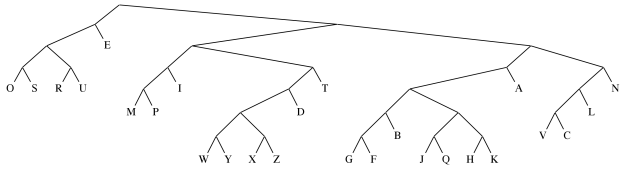
Consider the following example of Morse code, wherein symbols are represented as a series of dots and dashes:



Characters in the English language which occur more frequently have shorter representations than those which occur rarely.

As the representation of some symbols are prefixes for the representation of other symbols, Morse code requires a symbol delineator, naturally a short pause, to delineate one symbol from another. When a human is encoding and decoding a message Morse code functions well, but from a computational perspective it is generally unsuitable.

Instead we turn to prefix codes. In a prefix code, no bitwise representation of a symbol is the prefix of another bitwise representation. The following is one such code, again optimized for English character frequency:



Symbols may only occupy leaves, as any symbol occupying an internal node would have a representation that is a prefix of their subtrees. A prefix code does not require a delineator at the cost of longer symbols.

While there are many algorithms for constructing prefix codes, we are going to focus on Huffman coding in this program.

Huffman Coding

Huffman coding is a method for constructing a prefix code using the frequency or probability of the input symbols. The more frequent a symbol occurs, the shorter the code should be. Huffman coding is optimal for symbol-by-symbol coding, where every symbol has a known occurrence probability and the appearance of a symbol is independent of the occurrence of any other symbol. If these assumptions are broken the theory of Markov processes will lead to a more optimal code.

The prefix trie is constructed from the bottom-up. At each step, the two symbols with the lowest frequency are chosen and connected with the symbols forming the left and right subtrees. The root connecting them is a symbol corresponding to the sum of the frequency of its children. This new 'virtual' symbol is then added to the remaining pool. Repeating this process will result in a single symbol as the root of the coding trie.

By convention, the lowest frequency will always form the left subtree (0 bit) and the second lowest frequency will always form the right subtree (1 bit).

If all the symbols have the same frequency the generated Huffman code will be equal to a block code where every symbol is enumerated by the codes of equal length.

File Formats

The input file format, also known as the plaintext format, consists of string of uppercase alphabetic characters (A–Z). No other characters will exist in the file. Consider the following example which specifies the result of a repeated coin toss:

```
HTHHHTHTTHHHHTHHTHHHTHHTTHHTHTHHHTTTHHTHHHHTHTHHTHTTTHTHTHHHHTHTTTTHTHT
TTTHHHHTHHTHHTHHTHHTHHHHHHTTHHTTHHHHHHTHHTHHHHTHHTTTTTHHTHHTTTHHTHHHTHHH
```

A close examination of these trials would suggest that our coin is not quite fair. This is a discussion for another time.

The output file format consists of a series of packets. Each packet consists of a byte indicating the number of characters in the packet, followed by a string of binary digits representing the encoded message. If the Huffman coding tree for

the coin toss example was $H \to 0, T \to 1$ then the binary representation of the encoded message would be:

```
01111111
01000101 10000100 10001001 10010100 01111001 00001010 01011101 01000010
11110101 11100001 00100100 10010000 00110011 00000010 01000010 01111100

00010001
01001110 01000100 00000000
```

(whitespace added for clarity). As Java's bytes are signed, the maximum packet length is 127 characters. When the encoded message length is not a multiple of eight, there is zero padding at the end of the message. As there is no direct method of differentiating padding from content, a length specification is required.

While this is the binary result, it is not the plaintext result (the result of opening the file in a text editor). In order to investigate the result you should use the unix utility `hexdump`. The hexadecimal representation of the encoded message as viewed by the `hexdump` utility would be:

```
0000000 6f 45 84 89 94 79 0a 5d 42 f5 e1 24 90 33 02 42
0000010 7c 11 4e 44 00
```

The lefthand column tracks the number of bytes, and the remaining columns give the hexadecimal representation of each byte, space-delineated.

For a slightly more complicated example, lets assume that our unfair coin sometimes bounces away under a table. When this happens we record an 'invalid' result. As heads is still more common than tails, we still want to give heads a shorter representation. Assume the Huffman coding tree is now $H \to 0, T \to 10, I \to 11$. For the message:

```
HTHHITHTTHHIHIHHTHHHTHHTIIHTHTHHITTTTHHIHHHHTHIHHTHTTTHTHIHHHITHTTTTHTHI
```

the binary representation would be:

```
01001000
01000111 00101000 11011001 00001000 10111101 00100011 10101010 00110000
10011001 00101010 01001100 01110010 10101001 00110000
```

Note once again the padding at the end of the message.

Finally, the hexadecimal representation would be:

```
0000000 48 47 28 d9 08 bd 23 aa 30 99 2a 4c 72 a9 30
```

Requirements

You are not allowed to import any classes from the `java.util` *package with the exception of* `Scanner`*. You are encouraged to utilize classes from* `java.io` *and to build any abstractions you deem necessary. You may utilize data structures written as part of earlier assignments.*

A *PriorityQueue* class is given with stubbed out methods. You must implement the stubbed out methods to provide the appropriate functionality. Your priority queue implementation must be backed by a heap to receive full points.

You must implement a *Huffman* class with the following methods. The encoder assumes that all messages contain only uppercase alphabetic characters (A–Z). An encoder is constructed with an array of length 26 containing the probabilities of each character A–Z in turn. You may assume that the probabilities sum to one, although the absence of this condition should not cause any problems.

```
/**
 * Constructs the Huffman coding tree.
 *
 * @param the symbol probabilities (weights), length 26,
 * corresponding to the symbols A--Z.
 */
public Huffman(double[] probabilities)
```

Any zero probabilities should be ignored. You may assume that this class will never build a tree with a zero probability character and then later use that character in either the encoding or decoding phase.

```
/**
 * Encodes the character stream given by the input filename
 * using this Huffman code, outputting the resulting bit stream
 * to the output filename.
 *
 * @throws IOException
 * @param input the input filename; the file is guaranteed to
 * be a character stream
 * @param output the output filename; the generated data should
 * be a bit stream
 */
public void encode(String input, String output)
throws IOException
```

```
/**
 * Decodes the bit stream given by the input filename using
 * this Huffman code, outputting the resulting character stream
 * to the output filename.
 *
 * @throws IOException
 * @param input the input filename; the file is guaranteed to
 * be a bit stream
 * @param output the output filename; the generated data should
 * be a character stream
 */
public void decode(String input, String output)
throws IOException
```

Both encoding and decoding read and write data from files. While a `Scanner` and `PrintWriter` may suit some of your needs, you must also interact with `FileInputStream` and `FileOutputStream`. How these classes are used to serve the purpose of this assignment is an implementation detail left to the student. It is recommended that the student build tools which encapsulate behavior and concerns.

```
/**
 * Reads a plaintext format file and calculates the probabilities
 * for each character A--Z.
 *
 * @param input the input filename
 * @return the probabilities for each possible character
 */
public static double[] probabilities(String input)
throws IOException
```

Grading

The point breakdown is as follows.

50 Heap correctly implemented.

50 Huffman coding correctly implemented.

Your code should conform to the courses coding style especially those concerning variable names, whitespace, and class, (main) method, and inline comments. You can lose up to 30 points for bad style or commenting.

Reminder — a program which fails to compile on the lab machines (under Java 1.6) will receive 0 points.

Submission

1. Create a zip file of your program using the utility `zip`.

```
> zip -r Program10.zip ...
```

2. Run the submit program.

```
> submit
```

   A text-based interface will appear in the terminal.

3. Use the arrow keys and enter to select a class (cs1131), section (sec1), and assignment (Program10).

4. Use the arrow keys to navigate to the file you would like to submit. Use the spacebar to select the file Program10.zip (an astrisk will appear next to the filename). If the filename is incorrect or you attempt to submit a different file, submit will not accept it.

5. Hit return to submit the selected file(s) and confirm the submission.

6. Verify that the file appears in the "Already Submitted" window pane.

You can use submit at any time to verify that the files have been turned in. The date/time of each submission is also listed.