CSCI 340
Data Structures and Algorithms
Spring 2018
Programming Project 2 – Huffman Codes

Due Date: Tuesday, February 27
Points: 50

Objective: Write a program to encode an ASCII file using a Huffman code.

Background
Huffman codes are a method of lossless data compression that uses variable length codes. The method was invented by David Huffman in the 1950s. As we shall see, it requires processing the file twice, once to create the code and once to compress the file.

There are many ways to represent a character. ASCII is an example of a *fixed length code*. Every character uses 8 bits as its representation. Unicode is a standard, where every character uses 16 (or more) bits. Huffman codes, by contrast, are *variable length codes*. Some characters might be represented by only 2 bits, while others might require 10 or 12 bits.

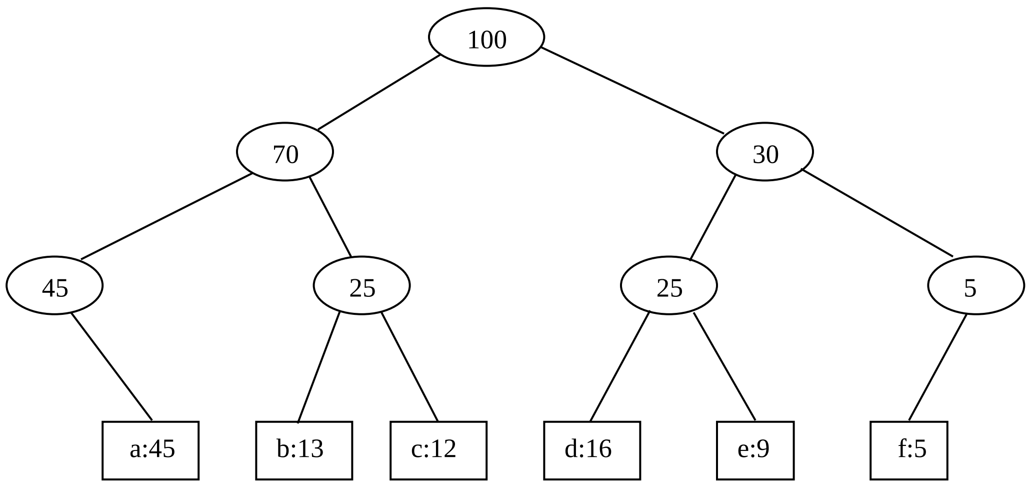
Huffman codes are *prefix codes*. That is, no codeword is also a prefix of some other codeword. Prefix codes are desirable because they simplify encoding and decoding. Encoding is just the concatenation of the codewords. Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous. For decoding we can simply identify the initial codeword, translate it back to the original character, and repeat the process on the remainder of the encoded file.

For example, consider the following table from a document of 100,000 characters including just a-f.

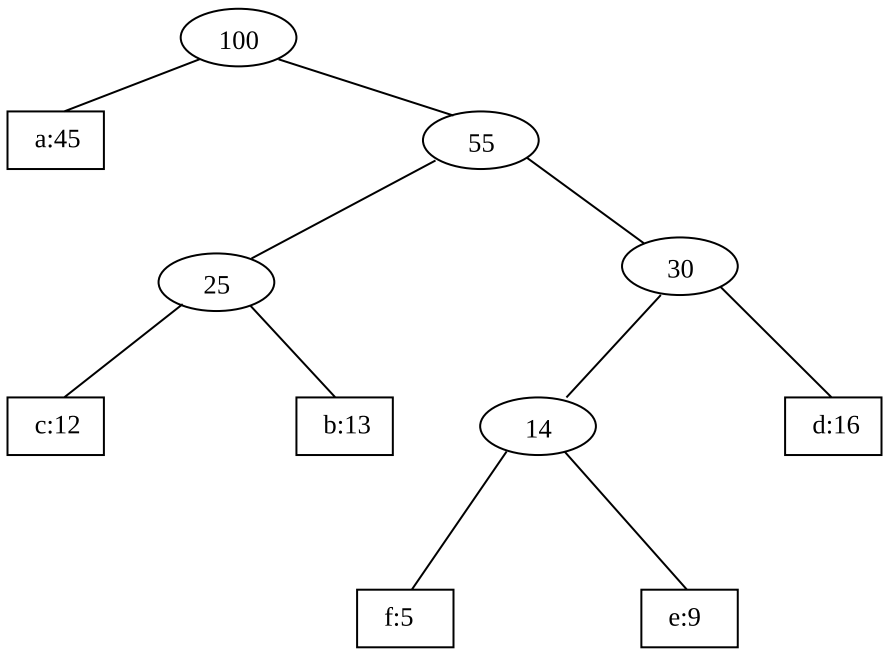
	a	b	c	d	e	f
Frequency(in thousands)	45	13	12	16	9	5
last 3 bits of ASCII	001	010	011	100	101	110
Prefix Code	0	101	100	111	1101	1100

Even if we just use the last 3 bits of the ASCII code, the file would encode to be 300,000 bytes long.
The Prefix code would be 45,000*1 + 13,000*3 + 12,000*3 + 16,000*3 + 9,000*4 + 5,000* 4 = 176,000 bytes. This is a significant savings.

Decoding needs a convenient representation for the prefix code so that the initial codeword can be easily picked off. A binary tree whose leaves are the given characters provides one such representation. We interpret the binary codeword for a character as the path from the root to that character, where 0 means "go to the left child" and 1 means "go to the right child." The following figures represent the binary trees for the two codes above. Note that these are not binary search trees, since the leaves need not appear in sorted order and internal nodes do not contain character keys.



Binary tree representing encoding using last three bits of ASCII code



Tree representing the prefix code in the table on page 1

Given a tree T corresponding to a prefix code, it is a simple matter to compute the number of bits required to encode a file. For each character c in the alphabet, let f (c) denote the frequency of c in the file and let dT(c) denote the depth of c's leaf in the tree. Note that dT(c) is also the length of the codeword for character c. The number of bits required to encode a file is

$$B(T) = \sum_{c \in C} f(c) * dT(c)$$

Huffman invented a greedy algorithm that constructs an optimal prefix code called a Huffman code. The algorithm builds the tree T corresponding to the optimal code in a bottom-up manner. It begins with the set of leaves, C, and performs a sequence of merge operations to create the final tree. Here is the pseudocode

```
TreeNode Huffman ( C )
{
   priorityqueue pq;
   insert all the leaves into pq;
   for (i=1; i<n; i++)
   {
      z = new TreeNode
      x = pq.extract_min( )
      y = pq.extract_min( )
      z.left = x
      z.right = y
      z.freq = x.freq + y.freq
      pq.insert( z )
   }
   return pq.extract_min( )
}
```

n is the number of letters in your alphabet.

Q is a priority queue of TreeNodes. Recall that you insert elements into a priority queue in any order, but they are always taken out minimum first.

Your Assignment:
You are to write a java class named Huffman, that contains two static methods, `encode()` and `decode()`. Obviously, you may use many other auxiliary methods, as you choose.

`public static void encode (String originalFilename, String codeFilename, String compressedFilename)`
`encode()` processes a file and creates two output files, a file containing the characters and their codes, and a file containing the Huffman encoded version of the original file.

Step 1. Create a table of frequencies. Simply count the occurrences of each byte/character in the file. A `TreeMap` works well for this. The key is the letter, the value is an object that contains the letters frequency and the Huffman code. The value class is easily coded as an internal class. That is, the value class is only used within Huffman, so it can be defined there. Note that in step 1 we only fill in the key and the frequency.

Step 2. Create the binary tree using the Huffman encoding algorithm above.

Step 3. Traverse the encoding tree and record the Huffman code for each letter in the value objects step 1. Write the table of letters, frequencies and codes in the code file. Note that it is best to use the ASCII values (ints) for the letters, as not all ASCII characters are easy to distinguish in a file, e.g. carriage return and linefeed.

Step 4. Process the original data file again, this time creating a Huffman compressed file.

`public static void decode (String compressedFilename, String codeFilename, String decompressedFilename )`

`decode()` reads in the two files created by `huffmanEncode()`, decompresses the file and writes the output to the decompressed file.

Help along the way:
Your instructor has created two classes that may be of use.

BitInputStream

`BitInputStream` reads one bit at a time from a file. It is useful for decoding. It contains the following constructors and methods:
`BitInputStream (String filename)`
constructs a BitInputStream for reading. If the file doesn't exist, an error is printed and the program exits.
`public int readBit()`
returns the next 0 or 1 from the file. It returns -1 when the end of file is reached.
`public int nextBit()`
an alias for `readBit()`

BitOutputStream

`BitOutputStream` writes one bit at a time to a file. It also will write a String containing 0s and 1s to the file as if they were bits. It contains the following constructors and methods:
`BitOutputStream (String filename)`
constructs a `BitOutputStream` for writing. Exits the program is there is an error.
`public void writeBit(int bit)`
writes the bit to the file.
`public void writeString(String bits)`
Write the bits contained in a `String` to the file. The `String` is interpreted as "1" is a 1 bit. Any other character is a 0 bit.
`public void close()`
closes the file. Since a file must contain an even number of bytes, the final byte is padded with 0s.

He has also posted a example of how to read a file byte at a time using a `RandomAccessFile`.

Test Data:
Your instructor has placed a test file on the course website and D2L, `mcgee.txt`. This should give fairly good compression, as it is almost all letters.

What to turn in:
Only electronic submission is required. Submit `Huffman.java` Do NOT use a package, or more precisely, do use the default package. Your instructor will provide his own test main. Good software engineering is expected. Use lots of comments and appropriate indentation.