Assignment 10: Huffman Codes

Implementing a Huffman coding scheme

Overview

In this assignment you will implement the Huffman coding procedure in order to generate an optimal prefix code for a given text file. Your program will read a specified text file and generate a binary codeword for each character appearing in the text. Assuming the procedure is correctly implemented, the resulting encoding of the character values will result in the smallest possible character-by-character encoding of the text file.

Background

A binary encoding of a document associates a *binary codeword*—i.e., sequence of `0`s and `1`s—to each symbol or characer appearing in the document. One standard encoding, the ASCII encoding, represents each character using 1 Byte (= 8 bits), so there are $2^8 = 256$ possible characters. For example, in ASCII the codeword for `A` is `01000001`, while `B` is encoded as `01000010`, etc. While ASCII encoding ensures that all possible characters are represented using 8 bits, the encoding is potentially inefficient:

1. ASCII assigns a codeword for every possible `char` value. However, most (English) text documents use considerably fewer distinct characters. For example, the complete works of Shakespeare (encoded in ASCII) use fewer than 120 distinct `char`s.

2. In a given text, some characters may be signficantly more frequent than others. For example, in `shakespeare.txt`, the single `` ` `` (space) character accounts for about 17% of all characters in the text, and the 8 most frequent character account for more than half of the characers in the document.

For many texts, using a different character encoding can result in a smaller overall file size. In particular, this can be the case when shorter codewords are assigned to more frequent characters. The size of the resulting encoding of a file (in bits) can be computed according to the following pseudo-code:

```
size = 0
for each char ch do:
    f = # of occurances of ch in text
    length = length of encoding of ch
    size += f * length
return size
```

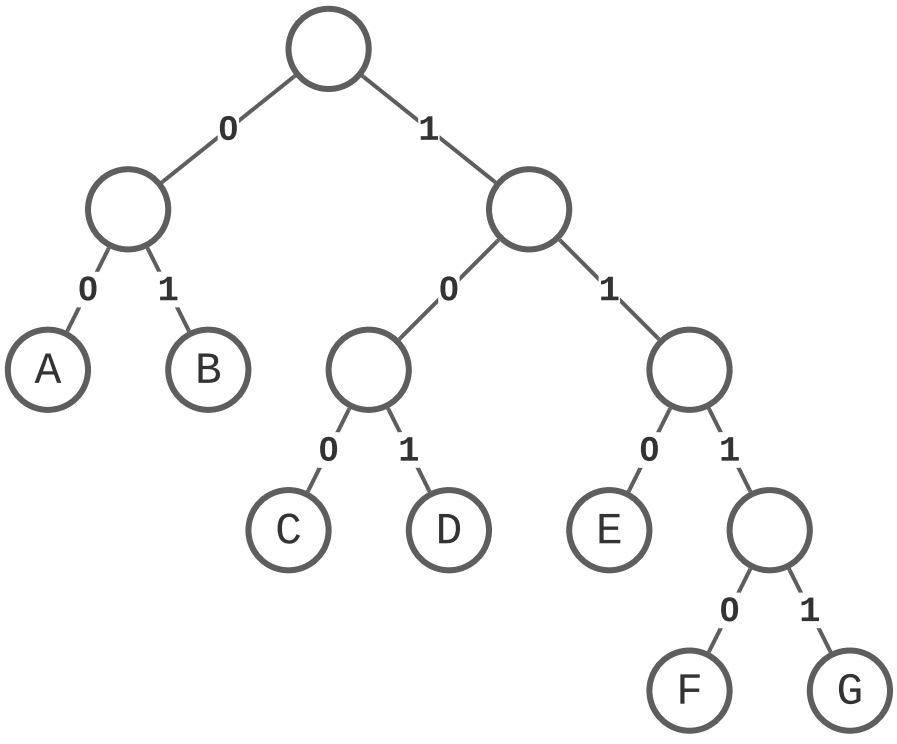
Prefix Codes

Given two binary sequences, $B = b_1 b_2 \cdots b_k$ and $C = c_1 c_2 \cdots c_\ell$, we say that B is a *prefix* of C if $b_1 = c_1, b_2 = c_2, \ldots, b_k = c_k$. For example, `101` is a prefix of `10110`. A *(binary) prefix code* is a code in which no codeword of any character is the prefix of the codeword of another character. For example, in a prefix code, we could not associate `A` with `101` and `B` with `10110`, because the codeword for `A` is a prefix of the codeword of `B`.

A prefix code can be represented as a binary tree. Each character is associated with a leaf in the tree, and each edge from parent to child has an associated label, `0` or `1`. The codeword for a character `ch` is then the sequence of `0`s and `1`s encountered along the path from the tree's root to the leaf storing `ch`. For example, consider the following prefix code.

char	codeword
`A`	`00`
`B`	`01`
`C`	`100`
`D`	`101`
`E`	`110`
`F`	`1110`
`G`	`1111`

We can associate this code with the following tree:



Observe that the edges along the path from the tree's root to the leaf labeled `A` are labeled `00`, corresponding to `A`'s codeword, and so on.

Huffman Encoding

The Huffman encoding procedure generates a prefix code for a given text that will result in the shortest possible encoding (among all prefix codes for that text). In particular, the Huffman procedure generates a tree as above that represents the optimal encoding. Huffman's procedure builds the tree from the leaves up. In order to implement the procedure, each node in the tree must additionally store a weight. The weight of a leaf is the frequency (i.e., number of occurances) of the corresponding character in the next. The weight of each internal node is the sum of the weights of its children. Thus, the root's weight is the total number of characters in the text.

The procedure proceeds as follows. Initially, the "tree" consists of a collection of nodes corresponding to the distinct characters in the document and the weight of each node is its character's frequency in the text. Then the following action is repeated until the collection of nodes has size 1:

- removed the two lightest (minimum weight) nodes from the collection, and set them as children of a new node
- set the new node's weight to be the sum of the childrens' weights
- add the new node back into the collection

The following pseudo-code describes the procedure in more detail.

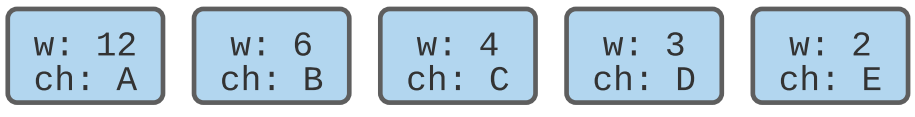
```
Initialize:
    C = empty collection of nodes

    for each character ch in text do:
        if C contains a node nd with value ch:
            increment nd.weight
        else:
            create a new node nd storing value ch with weight 1
            add ch to C
    endfor

BuildTree:
    Q = empty priority queue

    for each node nd in C:
        add nd to Q with priority nd.weight
    endfor

    while Q.size > 1:
        nd0 = Q.removeMin()
        nd1 = Q.removeMin()
        nd2 = new node with nd0 and nd1 as children
        nd2.weight = nd0.weight + nd1.weight
        add nd2 to Q with priority nd2.weight
    end while

    root = Q.removeMin()
```

As a concrete example, consider the following text:

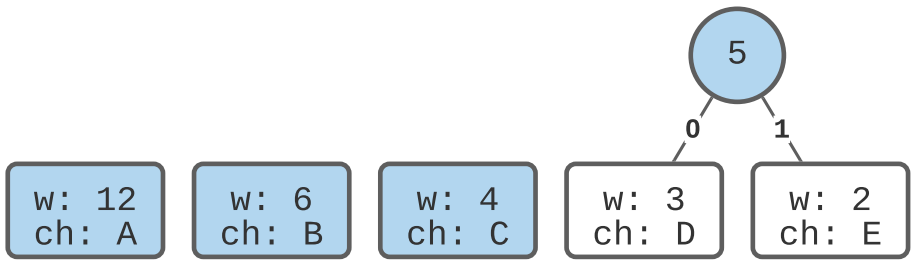
`AAAAAAAAAAAABBBBBBCCCCDDDEE`

char	frequency
A	12
B	6
C	4
D	3
E	2

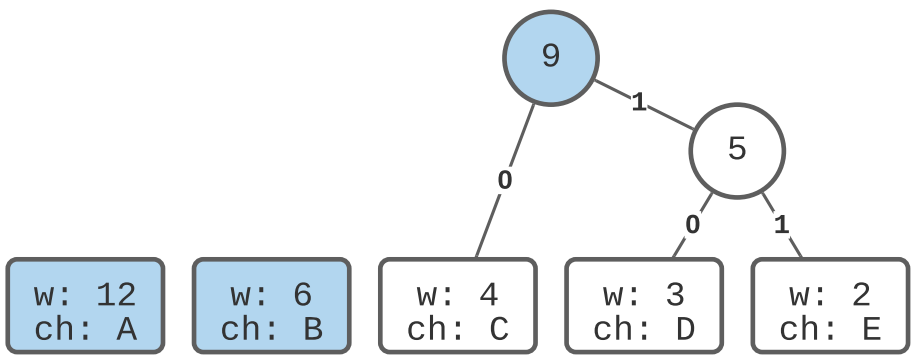
Thus, after initialzation, we would store the following collection of nodes:



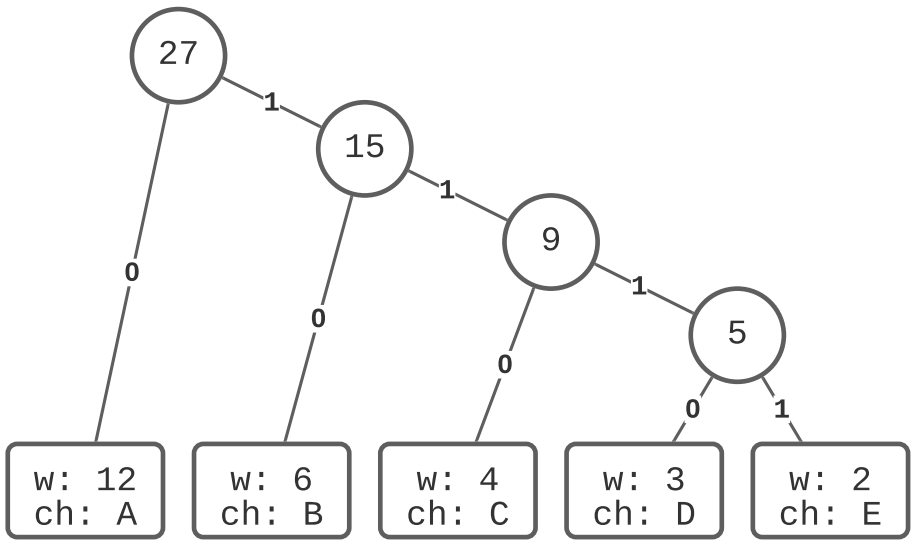
Here, `w` stores the node's weight, and `ch` its value. After the first iteration of the `Build_Tree`, the two lightest nodes (storing `D` and `E`) would be replaced with a parent node of weight 5:



Here, the shaded nodes are stored in the collection (represented by a priority queue), while the un-shaded nodes have been removed. The next iteration gives the following collections:



Continuing as above, we arrive at the final tree:



This tree corresponds to the following (optimal) prefix code:

char	codeword	length	frequency
A	0	1	12
B	10	2	6
C	110	3	4
D	1110	4	3
E	1111	4	2

From this chart, we can compute the total size of the encoded text to be 56 bits.

Note

There is some ambiguity in the description of the Huffman coding scheme above. In particular, when removing two nodes from the collection, it does not specify which nodes should be removed in the case of a 3-(or more)-way tie, and it does not specify which node should be th `0`-child and `1`-child of the new parent node. Thus, there are many distinct Huffman codes for a given text. However, they are

Your Task

For this assignment, you will write a program that implements the Huffman coding procedure described above. Specifically, you will implement a class `Huffman` that implements the `PrefixCode` interface (see code below). Your program must generate a Huffman code using an `InputStream` containing the text to be encoded. Your `Huffman` class will provide the following methods, described in the `PrefixCode` interface:

- `void generateCode(InputStream in)` generates the Huffman code based on an `InputStream`
- `String getCodeword(char ch)` returns a (binary) codeword associated with a character
- `int getChar(String codeword)` returns the character associated with a (binary) codeword
- `String encode(String str)` encodes a string of characters as a binary string
- `String decode(String str)` decodes a binary string to give the original string
- `int originalSize()` returns the original size of the `InputStream` in Bytes
- `int compressedSize()` returns the size of the resulting encoded string in Bytes

Suggestions & Resources

In order to implement a `PrefixCode`, you will need to use a an `InputStream`. You can read the `InputStream` documentation here. In particular, the `int read()` method will allow you to sequentially read the successive characters appearing in the `InputStream`. *This is the only `InputStream` method that your program must use.*

You will need to *use* (at least) four (not necessarily distinct) data structures to implement the Huffman encoding procedure:

1. A data structure to store the frequency counts of each character appearing in the `InputStream` from which the code is generated.
2. A data structure (suggested: priority queue) to perform the “merging” operation that constructs the Huffman tree described above.
3. A data structure that allows you encode characters efficiently to their binary string representations (for the `getCodeword` and `encode` methods).
4. A data structure that allows you to decode codewords efficiently (for the `getChar` and `encode` methods).

For all of these operations, you can use any code presented in class or that you’ve written for previous assignments (or of you may, of course, implement everything from scratch for this assignment). However, it may be more convenient and/or efficient to use built-in Java implementations of data structures. Specifically, you can look into the `PriorityQueue` and `HashMap` classes. Note that these implementations offer slightly different (and strictly more!) functionality to implementations we wrote in class.

Code

The file `hw10-huffman-codes.zip` contains the following files:

- `PrefixCode.java` An interface specifying the behavior or a prefix code.
- `HuffmanTester.java` A tester for your `Huffman` class implementing the `PrefixCode` interface.
- `RunTimer.java` A utility to test the running times of procedures.
- `metamorphoses.txt` A mid-length text to test your Huffman encoding program on.
- `shakepseare.txt` A long text to test your Huffman encoding program on.

Questions

1. Suppose Huffman coding is used to compress a text file containing 8 million characters, so that the original file size is 8 MB (= 8 million bytes). (Each character is encoded using 1 Byte = 8 bits.) What is the smallest possible size of a Huffman encoding that could result from the file? What orignal file(s) would generate an encoding of minimal size?

2. Will the Huffman encoding always generate a smaller document than the original document? Could the Huffman encoding result in a larger encoded file? For what docuemnts would you expect a Huffman encoding to result in the largest encoded file?

3. The Huffman code for `shakespeare.txt` results in an encoding that is approximately 60% of the size of the original file. Suppose the encoding for `shakespeare.txt` is used to encode other document, `some-text.txt`. Would you expect a similar compression ratio (60%) for the resulting encoding? Why or why not? Under what conditions would you expect to see a similar compression ratio for `some-text.txt`?

What to Submit

Sumbit your `Huffman.java` file, as well as any other files needed for your program to complile and run. Additionally, include your responses to the questions above in `responses.md` or `responses.pdf`.

Evaluation

Your submission will receive a score out of 10 points, broken down as follows:

- 5 points for `Huffman.java`
  - Code compiles, runs, and passes tests in `HuffmanTester.java`.
  - Code is sensibly organized with helpful comments.
- 5 points for your responses to the questions above.