CS 10 PS-3
Huffman coding

In this assignment, you will use Huffman encoding to compress and decompress files. This brings together trees, maps, priority queues, and file i/o, all to help save the bits! While Huffman coding has been around for a while, it made a splashier appearance more recently when scientists used it to store images in DNA (popular press; article).

Background

File compression compacts the contents of a file (or bunch of files, as when you turn in a zip file of this homework) to save disk space and transfer time. Some compression schemes (like jpeg for images) are *lossy*, in that they throw away information. Other compression schemes (like zip) are *lossless* — you get back exactly what you put in, but it's just stored in a more compact manner.

One of the earliest schemes for lossless file compression was invented by Huffman (as a term project!). Instead of using 7 bits to encode each character, as ASCII does, it uses a variable-length encoding of characters. Frequently occurring characters get shorter code words than infrequently occurring ones. (A code word is the sequence of 0's and 1's used to encode the character.)

Huffman encoding gives the smallest possible fixed encoding of a file. A fixed encoding means that a given letter is represented by the same code wherever it appears in the file. Some more recent schemes that do better than Huffman's are adaptive, which means that how a letter is encoded changes as the file is processed.

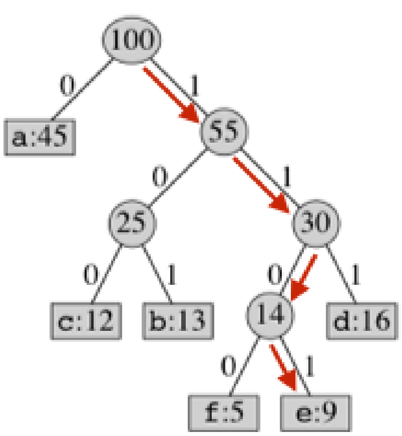
Pages 595-597 of the textbook contain a description of how Huffman encoding works and even gives you psuedocode for how to build the Huffman code tree.

Huffman Encoding

A problem with variable-length encodings is figuring out where a code word ends. For instance, if we used '1' for 'E', '10' for 'F' and '00' for 'X' we would be in trouble. When decoding the file we would not know whether to interpret '1' as 'E', or as the start of the code for 'F'. Huffman Encoding removes this ambiguity by producing *prefix-free* codes. This means that for any given code word, adding bits to the end cannot produce another code word. Hence no code word is a prefix of another. When the computer observes a series of bits that corresponds to a code word, it knows that it cannot be part of a larger code word, and can safely interpret the series of bits as its corresponding character.

At this point the purpose and value of Huffman Encoding should be fairly clear. So how do we do it? The task is to generate a set of prefix-free codes whose lengths are inversely correlated with the frequency of the encoded character. There are two clever parts of the algorithm, the use of a binary tree to generate the codes, and the construction of the binary tree using a priority queue. Specifically we will construct a tree such that each character is a leaf and the path from the root to that character gives that character's code word, where a left child is interpreted as a 0 and a right child as a 1.

For example, in this tree, the codeword for e is 1101:



The number to the right of a colon in a leaf of this figure gives the frequency of the character. The number in an internal node gives the sum of the frequencies of the leaves in its subtree. We'll soon see why we need this information.

Generate a Frequency Table

The first step in creating the code for a given file is to learn what the character frequencies are. This is fairly simple. First we create a `Map` with characters as keys and a frequencies as values. Recall that we instantiate the types with classes such as `Character` and `Integer` instead of primitives such as `char` and `int`.Then we read the file one character at a time. We add the character to the map if it is not there and increment the corresponding map value if it is there. At the end we will have a map that maps each character to the number of times that it appears in the file. (Note: in Java we have to use appropriate wrapper classes to store the character and the frequency, because we cannot use primitives in a `Map`.)

Put Initial Trees in Priority Queue

We now create an initial tree for each character. The data consists of two values, a character and its frequency. We then add all these initial single-character trees to a priority queue. The priority queue is set up to return the tree with the lowest frequency when asked to remove.

You can use the binary tree code from class. Because you need to store two values and a tree node has only one `data` you will have to create a class that stores a character and a frequency. This new class will be the data type for the tree. You should provide accessor methods as needed.

You can use whatever priority queue implementation you like. I found it easiest to use Java's `PriorityQueue` class, which implements a heap-based priority queue. Note that the things that you are comparing are the frequency counts in the root nodes of two trees. It is simplest to write a separate `TreeComparator` class that implements the Comparator interface. The only method that you need to implement is `compare`, which takes two tree nodes and returns -1, 0, or 1 depending on whether the first has a smaller frequency count, the counts are equal, or the second has the smaller frequency count. You pass a `TreeComparator` object to the `PriorityQueue` constructor. As we discussed, you could also use an anonymous function if you like that style.

Tree Creation

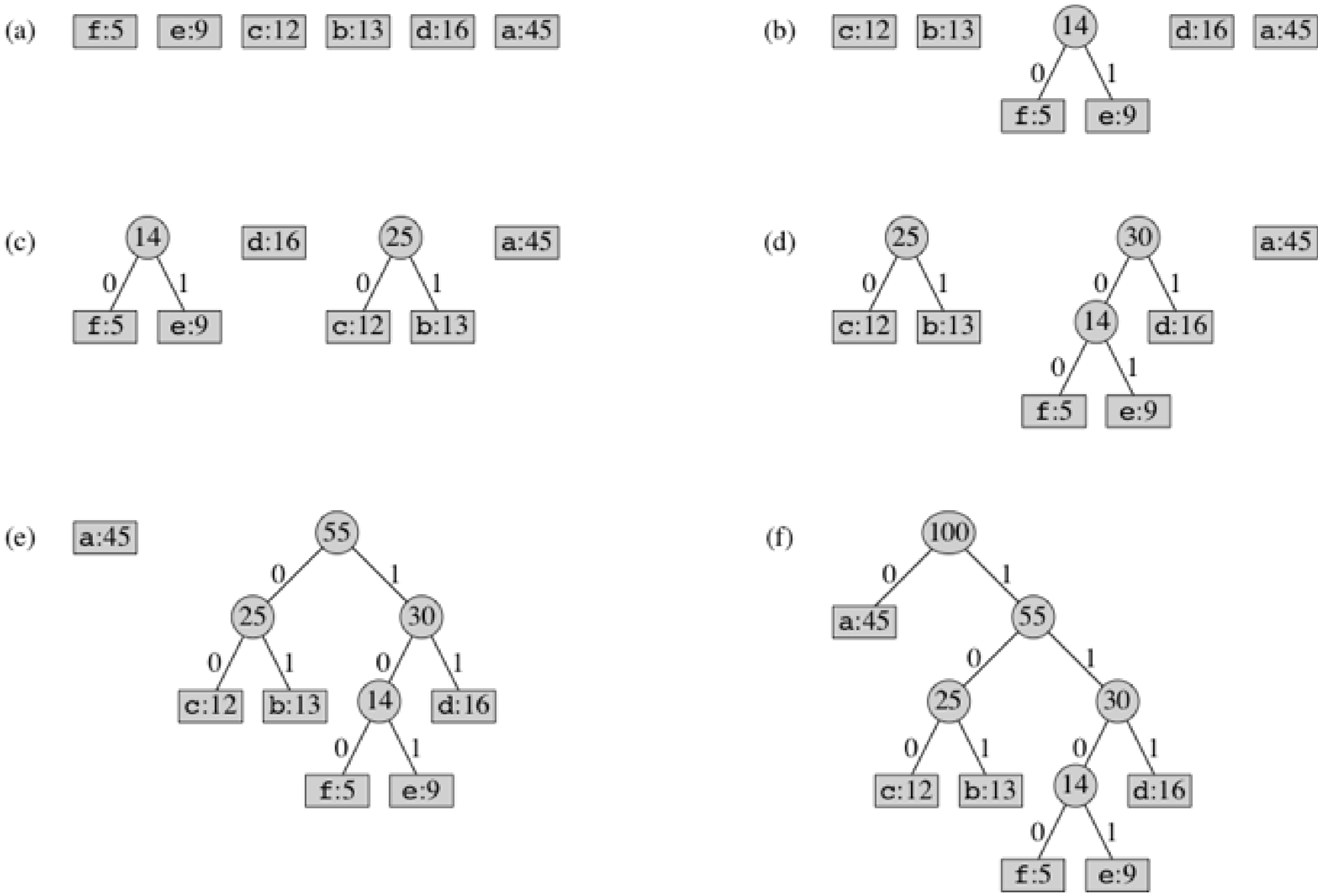
Notice that all leaf paths in a tree are unique. We represent the paths as a sequence of bit values, '0' for following the left path to the left child and '1' for following the right path. This way we create a variable length bit code for each leaf-node character. We also see that a path to a leaf L cannot contain the bit code for a path to any other leaf L', because if it did then then L' would not be a leaf.

In creating the tree we would like to have the lowest frequency characters be deepest in the tree, and hence have the longest bit codes, and the highest frequency characters be near the top of the tree. We use our priority queue to achieve this.

1. Extract the two lowest-frequency trees T1 and T2 from the priority queue.
2. Create a new tree T by creating a new root node r, attaching T1 as r's left subtree, and attaching T2 as r's right subtree. (Which of T1 and T2 is the left or right subtree does not matter.)
3. Assign to the new tree T a frequency that equals the sum of the frequencies of T1 and T2.
4. The character value at an inner node doesn't matter; that's only used for a leaf.
5. Insert the new tree T into the priority queue (which will base its priority on the frequency value it holds).

Each time steps 1–4 execute, the number of trees in the priority queue reduces by one. We keep repeating the above four steps until only one tree remains in the priority queue. This tree is our Huffman *code tree*.

Here is an example that shows the steps. In each part, the tree roots appear in order of increasing frequency. Part (a) shows the initial single-character trees for the first six characters of the alphabet. Parts (b)–(f) show the result of executing steps 1–4 above, with part (f) showing the final Huffman code tree.



Code Retrieval

We now have a fully constructed tree. The exact proof for why the tree is so efficient can be found online or in most algorithms textbooks, so we will not go through it. However, it should be intuitive that since we built up from the lowest frequency nodes, the lower a character's frequency, the deeper its leaf node will be in the tree.

The tree encodes all of the information about the code for each character, but given a character it is a bother to search through the tree to find it in a leaf and trace its path from the root. To make encoding fast we want a `Map` that pairs characters with their code words. That is, we want to pair each character with a string of '0's and '1's that describes the path from the root to that character.

We can construct the entire map during a single traversal of the tree. We just have to keep track of a "path so far" parameter as we do the traversal. I will let you work out the details. There are less efficient ways to solve the problem, but for full credit you need to produce the code map during a single traversal of the code tree.

Compression

To compress, we repeatedly read the next character in your text file, look up its code word in the code map, and then write the sequence of 0's and 1's in that code word as bits to an output file. Reading and writing files is not hard, but is a bit detailed. This is described below.

Decompression

Decompression is similar to compression, in that we read from one file and write to another. To decode, run down the code tree until we reach a leaf. Start at the root and read the first bit from the compressed file. If it is a '0' go left and if it is a '1' go right. Repeatedly read a bit and go left or right until you reach a leaf. We have now decoded your first character. Get the character out of the current tree node's data and write it to the output file. Then go back to the root and repeat the process. When there are no more bits to read, the file is decompressed. (Hopefully you just wrote out a character and returned to the root at this point. If not something is wrong.) You can compare your input file and decompressed file to verify that they are identical.

You may have noticed that we have cheated a bit: we kept the code tree that we used for compression around and later used it for decompression. In a practical compression application (e.g. zip) there is one compress method and an independent decompress method. The code tree has to be somehow included in the file during compression so that it can be read and used for decompression. Doing this is extra credit (see below).

Reading and Writing Files

For compression we need to first read characters from a plain text file and then write bits to a different, compressed, file. For decompression we read bits from a compressed file and write characters to a different, plain text, file. Fortunately the Java library has classes that make it easy to read characters from a file and write characters to a file. As discussed in class:

```
BufferedReader input = new BufferedReader(new FileReader(pathName));
```

where "pathName" is the name of the file on your computer. This will open a file and save a buffer and information about the current position in the file within a `BufferedReader` object referenced by `input`. To read a character, call `input.read`. The `read` method returns an `int` that holds the Unicode encoding of the character. You just have to cast it to be a `char`. When the file is empty `read` returns the value -1. This gives us a different loop structure for reading characters than the line-by-line thing that we've been using:

```
int cInt = input.read(); // Read next character's integer representation
while (cInt != -1) {
  char c = (char)cInt;
  // ... Do something with the character
  cInt = input.read(); // Read next character's integer representation
}
```

Similarly for output:

```
BufferedWriter output = new BufferedWriter(new FileWriter(decompressedPathName));
```

where `decompressedPathName` is the name of the output file that you want to create. To write a character `c`, call `output.write(c)`.

When reading or writing is finished, the file should be close by calling `close()` (e.g. `input.close()` or `output.close()`). This frees up resources and cleans things up. If you don't close the output file the last buffer may not get written and the file can be left in an inconsistent state!

Reading and Writing Bits

Unfortunately the Java library does not have classes to read and write bits. To remedy this situation, we provided classes to read and write bits for you. You don't have to read the code, although you might find it interesting to do so. The tricky part is that we can only read and write bytes to files, so if we have a number of bits that is not a multiple of 8 the last byte will be only partially full. How can we tell how may of those bits are useful and how many are garbage?

BufferedBitReader and BufferedBitWriter are used in a manner very similar to the classes above. The way to open a reader and a writer is:

```
BufferedBitReader bitInput = new BufferedBitReader(compressedPathName);
BufferedBitWriter bitOutput = new BufferedBitWriter(compressedPathName);
```

To read, we call `bitInput.readBit()` and get back an `boolean`. But we can only do that if there is still a bit to get. The `bitInput.hasNext` method indicates if there is indeed a bit left to read. So the input can be wrapped up in a loop that's a bit more iterator-like than the input methods we've used so far:

```
while (bitInput.hasNext()) {
  boolean bit = bitInput.readBit();
  // do something with bit
}
```

To write, call `bitOutput.writeBit(bit)`, where `bit` is again a `boolean`.

Note that the bits are *boolean* values `false` and `true`, and these correspond to the code *characters* '0' and '1'.

For your safety, I've limited the number of bytes that can be written to a file, with a static variable. If you want a bigger file, you can increase that size. But one possible bug results in an infinite loop that writes bits until your disk is full, so this enables the program to bail out earlier than that.

There is also a `close` method for each of these classes. Call it when you are done reading or writing the file. If you do not close the output file I can guarantee that you will not be able to correctly read it later.

Handling Exceptions

All of the methods above may potentially throw an `IOException`. Because this is a checked exception you need to include `try` and `catch` blocks. Think carefully about where you want to catch errors. It may be easier to handle them in the main method than in a method that it calls.

You should also include make sure files are properly closed, even if an exception occurs during reading/writing (either catch that exception, or include a "finally" clause, as illustrated in the notes).

Filenames

You will be dealing with three files: the original text file, the compressed text file, and the decompressed text file. I suggest that you hardcode the original file path name in the `main`, or as a `static final`. **Use relative pathnames, with files in the "inputs" folder**, e.g., `String pathName = "inputs/WarAndPeace.txt"` so that your TA can directly run your program.

Generate the names of the other two, e.g., by putting "_compressed" and "_decompressed" after the file name. So if the original file were "WarAndPeace.txt" the compressed file would be "WarAndPeace_compressed.txt" and the decompressed file would be "WarAndPeace_decompressed.txt". This makes it easier to keep track of the relationships between your various test files. (Look up the `String` class's substring method and use it to construct the new names.)

Note that you don't need to create the output files manually — your Java code will do that. You might have to refresh the IntelliJ file browser, though.

You can look at the text files in IntelliJ IDEA. Looking at the compressed files will be uninormative, as it tries to interpret your bits in terms of characters, which they aren't representing. And if there are fewer bits than a byte, it won't show up correctly. But you can open these in various other editors if you want to; post on Canvas to share what binary editors you like.

Exercises

For this assignment, you may work either alone, or with one other partner. Same discussion as always about partners, and a new form is provided.

Provided in Canvas files for PS-3 are the bit reader and writer and two documents for testing (the US Constitution and *War & Peace*). No scaffold code — you're on your own!

1. You may choose to have an instance of an object hold all the various state, and provide methods for it to compress and decompress. Or you may think of this as a compression library, and write static methods that pass all the relevant information in parameters. Either is fine here; it's mostly just a stylistic choice.
2. Implement the compression method (and any helper methods and classes)
3. Implement the decompression method (and any helper methods and classes)
4. Consider *boundary cases* that your program might encounter — an empty file, one with only a single character, with a single character repeated a bunch of times, with a few different tree structures, etc. Make sure your approach handles them all.
5. Create small test files to help debugging, including boundary cases. When those results look good, try the US Constitution and *War & Peace*

This is ***not*** the kind of program where you can write out all the code, click the run button, select WarAndPeace.txt, and expect to be able to tell from the files printed where the bugs are in your program.

- Start with a short file (a couple of dozen characters), preferably one with a range of character frequencies. Even just "hello" is a fine first test case. Use a `System.out.println` to print out the frequency map. (The Java `Map` implementations have a `toString` that gives reasonable output.)
- Then print the code tree. (You did override `toString` in your class that holds the character and its frequency, didn't you?)
- Also print out the code map.
- The sample solution currently has all of these print statements in it, within the bodies of if-statements that check whether a boolean debugging flag is set. It could be interesting to see the frequency numbers for WarAndPeace.txt and to see how much the codeword length varies.
- You can also put print statements to print out every character or bit read from files and written to files. Again, you should either comment them out before you compress WarAndPeace.txt, or put the print statements under the control of a *debugging flag*.

Extra Credit

You are now able to compress and decompress files using Huffman encoding. However, you used the same tree to both to encode to decode. Normally the tree that you used to encode the file will not be around for you to use it to decode the file. For a practical system (similar to zip) the code tree has to be saved and reconstructed. Note that when you are decoding the frequencies do not matter, only the tree shape and the order of the characters at the leaves.

For extra credit implement a way to write out the code tree and then read it back in and regenerate it. You should write out the tree when you compress a file and read it in when you decompress a file. For the basic extra credit write a separate file to store the information needed to reconstruct the tree. For substantial additional extra credit write this information at the front of the file that contains the encoded characters, so that the file contains all of the information needed to decompress it.

When I say "write out the tree" I mean write out enough information to be able to regenerate the tree. One option is to use some sort of parenthesized notation that you then parse to reconstruct the tree. The trick of writing the tree in preorder and inorder could work, but you would need to generate unique names at the internal nodes that do not conflict with each other or with characters at the leaves. You could reconstruct the tree from frequency data, but you may have to be careful about the way that you deal with equal frequencies. I am sure that there are many other possibilities. You would like your representation to be compact, because the goal is to compress the file.

Substantial extra credit will be given for encoding and decoding in actual DNA :).

Submission Instructions

Turn in a single zipfile containing your code, thoroughly documented; your test files (original, compressed, and uncompressed); and the compressed and uncompressed versions of the Constitution. In the "Comments" field give the compressed size of WarAndPeace.txt, but *do not* include the output.

Acknowledgement

Some of the text for this assignment is modified from a writeup by Delaney Granizo-Mackenzie.

Grading rubric

Total of 100 points.

Correctness (70 points)

- **10** Generates frequency map
- **5** Creates a priority queue of initial single-character trees
- **15** Builds the code tree
- **10** Traverses the code tree to generate a `Map` from characters to code words
- **10** Reads the input file, compresses it, and writes the compressed file.
- **15** Reads the compressed file, decompresses it, and writes the decompressed file.
- **5** Handles boundary cases correctly.

Structure (10 points)

- **4** Good decomposition into objects and methods
- **3** Proper use of instance and local variables
- **3** Proper use of parameters

Style (10 points)

- **3** Comments for classes and methods
- **4** Good names for methods, variables, parameters
- **3** Layout (blank lines, indentation, no line wraps, etc.)

Testing (10 points)

- **5** Output from your small test case and from your boundary case(s)
- **5** Output from USConstitution.txt