Graduate-Credit Project

Due by 11:59 p.m. on December 11

Important: *This project can only be submitted by grad-credit students, and there will be no partial credit for submissions by others. For grad-credit students, the project will be worth the same as an ordinary homework assignment.*

Overview

For this project, you will:

- develop the data structures associated with Huffman encoding
- use these data structures and the associated algorithms to implement programs that compress and decompress files.

Reviewing the provided code

In the starter code for this project, we've given you the implementations of three types of objects that will facilitate your work:

- `BitReader` objects, which allow you to read one bit at a time from a file
- `BitWriter` objects, which allow you to write one bit at a time from a file
- `Code` objects, which represent the variable-length encoding (the *code*) of a given character.

You should review these classes and try to understand at least the basics of how they work. Here are some questions that you should answer for yourself:

- How do `BitReader` and `BitWriter` objects manage to read and write one bit at a time from to a file? You don't need to fully understand the bitwise operations that these objects perform, but you should understand the way that they use a one-byte instance variable as a buffer for the bits that are read/written.
- Where in your programs could you use `BitReader` and `BitWriter` objects? Which of their methods would you call?
- How does a `Code` object store a variable-length character encoding?
- How are `Code` objects used in the `BitReader` and/or `BitWriter` classes? Which method(s) from these classes expect a `Code` object as an argument?
- Which `Code` method allows you to gradually build up a character encoding, one bit at a time? Which `Code` method allows you to remove the rightmost bit from a character encoding? Where might you use these methods?

We have also given you starter code for the two programs that you will implement:

- `Huff`, which compresses a file using Huffman encoding
- `Puff`, which decompresses a file that has been compressed using Huff.

Designing your implementation

We strongly encourage you to construct a detailed design before doing any coding. Here is one possible set of steps that you could take in formulating your design:

1. Review the lecture notes on Huffman encoding. You may also want to work through one or two concrete examples of forming a Huffman tree and using it and the associated data structures and algorithms to compress and decompress a piece of text.

2. Make a list of all the methods you will need to create and manipulate Huffman trees. For the moment, you don't need to associate these methods with a particular class. That will come later. If you have worked through a concrete example, you might find it helpful to write the names of the methods next to the appropriate steps in creating the Huffman tree. For example, you might write "combineTwoNodes()" next to the steps in which you combine nodes.

3. Make a list of any other methods that you may need for performing the compression or decompression.

4. Specify the data structures that you will use in this assignment – including Huffman trees themselves. Begin with an English description of the data structures and their purpose, and then write an outline of how you will implement them in Java.

5. Divide your methods from steps 2 and 3 into three lists:
   - those unique to Huff
   - those unique to Puff
   - those used by both programs (possibly because they are methods that are associated with one of the data structures)

   Some of the methods should be part of the classes that represent the data structures. For other methods, it may be better to make them (possibly static) methods of the Huff and Puff classes. You should decide which methods belong to each class and whether or not they should be static.

6. Consider what code you will need to add to the `main()` methods in the `Huff` and `Puff` classes. We have given you the beginning of each `main()` method; the code we have provided opens the I/O streams that you will need in order to read and write the appropriate files. You should take it from there.

   <u>Here are some guidelines for performing the file I/O:</u>

   - In Huff, we have given you code that creates a `FileReader` object named `in` for the original text file. You may assume that the input file uses an 8-bit encoding scheme (i.e., ASCII). Use this object's `read()` method to read in the original text file one character at a time. See the

Java documentation for the `InputStreamReader` class (of which `FileReader` is a subclass) for a description of the `read()` method. In order to read in the file a second time, you will need to create a new `FileReader` object for the file and use it instead of the original one.

- In Huff, we have also given you code that creates an `ObjectOutputStream` object named `out` for the encoded/compressed file. You should use the appropriate `ObjectOutputStream` method(s) to output a header of the file. In particular, the `writeInt` method of that class may be useful for this purpose. The actual format of the header is up to you; the lecture notes give a rough idea of is needed.
- Depending on how you decide to structure the header, you may end up writing only frequencies, or both chars and frequencies. If you take the latter approach, it's worth noting that a `char` is really a number, so you can use the `writeInt` method of the provided `ObjectOutputStream` object to write both the `char`s and frequencies in the header. Also, it's worth noting that you do **not** need a delimiter (e.g., a comma) between values in the header, because the file you are creating is a binary file, and thus a value of a given type always has the same length. For example, if you use the `writeInt` method, each integer that you write to the header will have exactly four bytes. Just make sure that the reads that Puff performs when reading the header correspond to the writes that Huff performs when writing the header.
- Another issue to consider when designing the header is whether or not the header will have a consistent length. If it does not have a consistent length, then you will need to include any additional value in the header that allows Puff to figure out how large the header is. If it does have a consistent length, then this extra value is unnecessary.
- Also in Huff, we have given you code that creates a `BitWriter` object named `writer` for the encoded/compressed file. Use the appropriate method(s) of that object to write the variable-length character encodings to the new file.
- In Puff, we have given you code that creates an `ObjectInputStream` object named `in` for the encoded/compressed file. You should use the appropriate `ObjectInputStream` method(s) to read in the header of the file. In particular, the `readInt` method may be useful for this purpose. Make sure that the reads that you perform correspond to the writes that you performed in creating the header.
- In Puff, we have also given you code that creates a `BitReader` object named `reader` for the encoded/compressed file. Use the appropriate method(s) of that object to read the variable-length character encodings from the compressed file.

- Also in Puff, we have given you code that creates a `FileWriter` object named `out` for the decompressed file. Use this object's `write(char c)` method to write the decoded file one character at a time. See the Java documentation for the `OutputStreamWriter` class (of which `FileWriter` is a subclass) for a description of the `write(char c)` method.

7. Write pseudocode for each of the methods. Remember, pseudocode is not Java!

8. Evaluate your design by answering the following questions:
   - Do the classes and their methods work well together?
   - Is there duplication of code between methods?
   - Does each method have *one* easily stated purpose?
   - Are you satisfied with your design? How could you improve it?

Implementing your design

Transform the pseudocode you have written into Java code. Think about the order in which you will do this before starting. For a given class, you should finish the smaller, more easily tested methods first and test them before going on to the higher-level methods. Then, you can be more confident that your helper methods are correct when debugging your larger methods.

Please do NOT impose a package structure on your files. Keep everything in the default package.

Make sure to follow the guidelines for file I/O given earlier in this document.

In addition, make sure that your code is well-commented. Each new class that you create should begin with detailed comments that explain the purpose of the class. Each method should be preceded by comments that describe what it does. Additional comments should be included as needed to explain code within a given method.

In addition to your code files, please submit a text file called README.txt that includes a brief description of each class in your submission. This will make it easier for us to navigate through your files when grading. As part of this file, please include a discussion of the efficiency of your implementation. In particular, you should use big-O notation to give the time efficiency of the key operations that you perform as part of your implementation.

Testing
Once you have implemented everything, make sure it all works! One key test: when you use Puff to decompress a file that Huff has compressed, you should get back a copy of the original file! In other words, if you do the following from the command line (or, if you're using DrJava, from the Interactions Pane):

```
java Huff original.txt compressed.bin
java Puff compressed.bin copy.txt
```

then `original.txt` and `copy.txt` should be identical. (If you prefer not to use the command line, you can simply run each program and enter the appropriate filenames in response to the prompts that the program gives.)

We have given you a file named `romeo.txt` that you can use for testing.

Submitting Your Work

You should use Canvas to submit all of your work on the final project as follows:

- Go to the page for submitting assignments (logging in as needed using the Login link in the upper-right corner, and entering your Harvard ID and PIN).
- Click on the **Grad-credit Project** link.
- Click on the *Submit Assignment* link near the upper-right corner of the screen. (If you have already submitted something for the project, click on *Re-submit Assignment* instead.)
- Use the *Choose File* button to select a file to be submitted. Continue selecting files by clicking *Add Another File*, and repeat the process for each file. ***Important:*** You must submit all of the files at the same time. If you need to resubmit a file for some reason, you should resubmit everything.
- Once you have chosen all of the files that you need to submit, click on the *Submit Assignment* button.
- After submitting the assignment, you should check your submission carefully. In particular, you should:
  - Check to make sure that you have a green checkmark symbol labeled *Turned In!* in the upper-right corner of the submission page (where the Submit Assignment link used to be), along with the names of all of the files that you are submitting.
  - Click on the link for each file to download it, and view the downloaded file so that you can ensure that you submitted the correct file.

  We will not accept any files after the fact, so please check your submission carefully!

Note: If you encounter problems submitting your files, close your browser and start again, or try again later if you still have time. If you are unable to submit and it is close to the deadline, email your homework before the deadline to `cscie22@fas.harvard.edu`