Programming Assignment 9: Huffman Coding

Due: Thursday, April 17th at 11 P.M. CT

Learning Goals:

- To learn about, implement, test, and debug the compression and decompression algorithms for Huffman Encoding
- To practice implementing data structures such as a priority queue
- To expose students to bit manipulation, bit tracking, and bit usage to store data

Assignment Guidelines and Specifications

This is a **pair programming assignment**, which means you are welcome to work with a partner to complete this assignment. You are not required to work with a partner and are welcome to complete the assignment individually if you wish. If you choose to work with a partner, there are some important rules you must follow:

- You should follow these best practices when pair programming.
- Your partner must **be in the same section as you** and have the **same TA**.
- You will write multiple program files, and both of your names and EIDs must be included in the headers of these files. **Only one member of the pair will submit the assignment through Gradescope**. After submitting the assignment, on the submission page, click "View or edit group" on the upper righthand corner of the screen. Enter the name of the other partner in order to add them to the submission. After doing this, the other partner should get an email confirming that they were added to the submission. **If you do not add your partner to the Gradescope submission, the other member of the pair will not get any credit for completing this assignment.** After adding the other partner, check that both partners are able to see the submission on their Gradescope accounts. This will mean that you have both been correctly linked to the submission.
- If you choose to use late days, **both partners must have the required amount** of late days. If either partner does not, **both partners will receive a 0 on this assignment**

Introduction

In this assignment, you will be exposed to and develop your own form of the famous Huffman coding compression algorithm. You will write a program that performs two related tasks: **compressing** files and **decompressing** files (that are compressed by the first part of the program). The program should allow a user to utilize the Huffman coding algorithm to compress a file and undo the compression upon request. The program will also inform the user of the bits they save and write upon performing the compression. When writing this program, you should implement additional classes to break the problem up into smaller, more manageable pieces and prevent repeated code. You should have a class that models a Huffman code tree and a priority queue. You may also find it useful to have separate classes to manage the details of compression (`Compressor`) and decompression (`Decompressor`).

Getting Started

Here are the necessary files for this assignment:

- **`HuffmanSource.zip`**: This is a zip file with the given classes to use in the assignment. These files are necessary for testing and running your solution.
    - Within this zip file is **`SimpleHuffmanPrcoessor.java`**. This file contains the skeleton of the methods you will be expected to implement for this assignment. It is highly recommended that you **create new classes as necessary** rather than writing all of your solution code in `SimpleHuffProcessor`. You will be expected to follow (and graded on) good object-oriented programming design.
    - The `Huff.java` program creates an interface that you will use to interact with the `SimpleHuffmanProcessor` that you create. By default, the program creates a graphical interface. However, this can easily be changed to a textual interface by commenting out one Line 22 and commenting in Line 23 in `Huff.java`.
    - **All other files in this folder must remain unchanged.**
- **`Documentation`**: This is documentation for the provided classes in `HuffmanSource` that are not from the standard Java library. You should refer to this extensively.
- **`smallFileTester.zip`**: This is a zip file that contains a small sample test file. You should use this to help you incrementally develop your program.
    - **`smallTxt.txt`**: This is an example small file for you to compress. It contains the string "Eerie eyes seen near lake."
    - **`smallTextsFreqsAndCodes.txt`**: This file contains the expected frequencies and Huffman codes for the intermediate step of the encoding process.
    - **`Eyes_Count_Format_explicit.txt`**: This is the file (using standard count format) with 0s and 1s shown as chars (not stored as bits). This may be helpful when debugging.
    - **`Eyes_Tree_Format_explicit.txt`**: This is the file (using standard count format) with 0s and 1s shown as chars (not stored as bits). This may be helpful when debugging.
    - There is more information about the expected return values from this file later in the testing section.
- **`largeFileTester.zip`**: This is a zip file that contains a large sample test file.
    - **`ciaFactBook2008.txt`**: This is an example large file for you to compress.
    - **`ciaFactBook2008FreqsAndCodes.txt`**: This file contains the expected frequencies and Huffman codes for the intermediate step of the encoding process.
    - **`CIA2008_Factbook_SCF_Header_ExplicitBits.txt`**: This is the file (using standard count format) with 0s and 1s shown as chars (not stored as bits). This may be helpful when debugging.
    - **`CIA2008_Factbook_STF_Header_ExplicitBits.txt`**: This is the file (using standard count format) with 0s and 1s shown as chars (not stored as bits). This may be helpful when debugging.

    - There is more information about the expected return values from this file later in the testing section.
- **`ExplicitBitOutputWriter.java`**: This file contains a class that converts any file to a file of ASCII 0's and a's. This can be useful for the small files to find minor differences in the output vs. expected text and .hf files.
- **`DecompressionTester.zip`:** These files are provided so that you may test your decompression algorithm. We will not provide the original files.
- **`TestingHarness.zip`**: Provided in this zip file is another way to test your program. This is a similar test harness to what we will use when grading your submission. The `DoNothingHuffViewer` is used instead of the one provided in the previous zip files. It is recommended that **you make a separate instance of the project when testing with this harness** to avoid potential complications.
    - To use the harness, you will need the included bevotest.jar file (created by John Thywiseen). See this page for info on how to include a jar in your Eclipse project.
- **`calgary.zip`**, **`waterloo.zip`**, **`BooksAndHTML.zip`**: These are zip files with multiple large files to test the correctness of your program. It is recommended that you use the provided `HuffMark` program to test these.
- **`AdditionalTests.zip`**: This zip file contains even more tests that you may want to use!

`SimpleHuffProcessor`

The `SimpleHuffProcessor` class will be broken down into 3 methods (2 of which we will group together due to the latter's dependence on the former) to perform the responsibilities required for compression and decompression of target files.

This section will be structured by covering the specific responsibilities of each method through a high-level description and explanation of parameters, followed by a limited but insightful HowTo for the section. During the overviews, we will refer to an external "How-To" guide that can be found here. Whenever these sections refer to a how-to guide, note that they all refer to the same guide (the one linked above), just different parts of it. **It is highly recommended that you thoroughly read through the how-to guide prior to beginning any coding.**

Part 1: Compression

PreprocessCompress

```
public int preprocessCompress(InputStream in, int headerFormat)
```

Preconditions: None

Parameters:

- `InputStream in`: An `InputStream`, similar to a `Scanner`, is an object linked to a file that is used to read in information from the file. (During testing, you will select which file this will be). For the purposes of our assignment, you must make this a `BitInputStream`. For more information, refer to documentation linked above and the relevant section of the how-to guide.
- `int headerFormat`: This is a numerical value that represents whether we will be using the Standard Count header format or the Standard Tree header format when storing information about the encoding scheme itself.

This method should return the number of bits in the original file minus the number of bits written to the newly compressed `.hf` file. This is known as the number of bits 'saved' by performing a compression. When calculating the number of bits written to the file, you should include the Huffman Magic Number, the Header Format Number, the Header itself, and the actual data written to the file. Do **not** include any padding added from the `BitInputStream`.

The expected behavior of the `preprocessCompress()` method is outlined in the compression section of the how-to guide. In this method you should also prepare all the values and information you will need to perform the compression (including the actual Huffman codes and the relevant header information). This is because in order to count the number of bits you are 'saving' using compression, you will need to count the number of bits you will be writing during the compression. In other words, you have to determine everything about the compression **without writing any information to the file**.

Again, note that this method does NOT write anything to a file, but rather just sets up the user to call the compress method by preparing all the necessary data, codes, and variables.

Compress

```
public int compress(InputStream in, OutputStream out, boolean force)
```

Preconditions: The `preprocessCompress()` method has already been called

Parameters:

- `InputStream in`: This parameter is the same `InputStream` that is passed into the `preprocessCompress()` method. They both refer to the same file and must both be converted to a `BitInputStream`. They're both used for reading information from the file.
- `OutputStream out`: This parameter is an object linked to the new `.hf` file (the compressed file) you are creating. Similar to the `InputStream`, it must be converted to a `BitOutputStream`. This is because you need explicit control over how many bits to write to the output file. You will use this to write to the new compressed file.
- `Boolean force`: This parameter is used to determine whether or not to perform the compression. When compressing a file, the testing / execution environment we provide for this assignment automatically calls `preprocessCompress()` prior to `compress()`. This means all values calculated in `preprocessCompress()`, including the return value, will be available to the `compress()` method since they are run in the same instance. If the number of bits we 'save' from compression, which is the return value of `preprocessCompress()`, is negative (meaning we don't save space by compressing the file), then the compression will only be performed if the `force` parameter is `true`.

This method will return the number of bits written to the file (excluding padding). After running this method, you should have a resulting `.hf` file that is the compressed version of whatever file was passed in during execution. You can expect that `preprocessCompress()` has run prior to the execution of the `compress()` method. Therefore in this method, you can use the data created in `preprocessCompress()` without issue (including the Huffman Tree, the Huffman Codes, and any other relevant information).

This method will not do much counting (as that work was already performed in `preprocessCompress`), but rather, it will read in the file 8 bits at a time and write the corresponding encoding to the compressed file. This method will also write the magic number, the header format, and the header itself as mentioned in **this separate structure guide**. By the end of the execution, you should have a newly compressed file whose structure follows the layout of the first image in the guide linked above.

If the `force` compression is off and the number of bits 'saved' is negative, `compress()` should return an error message and not produce an output file.

This method's sole purpose is to write out the codes to the target file through the `OutputStream` to create the user's requested compressed file.

Tips for PreprocessCompress and Compress

In this section, we will provide an additional high-level overview of the `preprocessCompress()` and `compress()` methods and provide some advice. **Before reading this section, you should read the how-to guide's compression section.** Some notes on this section of the guide:

- Steps 1 through 5 of the compression section should be implemented in the `preprocessCompress()` method, **not** the `compress()` method.
- The `preprocessCompress()` method should calculate the frequency array, the Huffman Tree, the Huffman code, the number of bits 'saved', and the number of bits written to the file.

Here is a brief overview of the execution order, behavior, and outline of the two methods:

1. You should initially read in the entire file in `preprocessCompress()` in order to create the frequency array. Then, create nodes using the given `TreeNode` class using the characters and frequencies. It is important to keep this in mind when reading from the file.
2. In `preprocessCompress()`, you will then populate a priority queue (as mentioned in steps 2, 3 and 4 of the guide) using the nodes. **Note that you will need to write your own fair priority queue class.** Additional specifications, restrictions, and clarifications for the behavior of this class are provided below. Some brief information can be found here.
3. In `preprocessCompress()`, use this priority queue to create a Huffman Coding tree. You can find more information about the structure of the tree in the slides from lecture, this part of the how-to guide, or the second half of this document covering header formats.
4. In `preprocessCompress()`, generate the codes from the Huffman Coding tree as is covered here in the how-to guide.
5. Although not specified in the how-to guide, in `preprocessCompress()`, you will also have to calculate the size of the header of the file to return the correct value. To help you do so, read the second half of this guide. This is the final work that needs to be completed in the `preprocessCompress()` method.
6. You can now begin the `compress()` method. This method simply uses the information and encodings created by the `preprocessCompress()` method. You can assume that `preprocessCompress()` will always be called prior to `compress()`.
7. The expectation of the behavior of the `compress()` method is listed in the paragraph **after** Steps 1-5 in the compression section of the guide. The method should write the magic number, the header (again shown here), and the encoding of the original file.
8. For an in-depth explanation of how the file structure and how to write each part, read the section on writing compressed files in the how-to guide. It should be referenced thoroughly when writing your code for the `compress()` method.

Part 2: Decompress

```
public int uncompress(InputStream in, OutputStream out)
```

Preconditions: None

Parameters:

- `InputStream in`: This parameter is an `InputStream` that links to a `.hf` file, or previously compressed file. Similar to your other `InputStreams`, it must also be converted into a `BitInputStream` since you will be examining the bits one at a time.
- `OutputStream out`: This parameter is an object linked to the target destination of the decompression. Similar to the previous `OutputStream`, it must also be converted to a `BitOutputStream`.

This method will decompress a file and return the number of bits written to the new, decompressed file. At the end of this method, the output file should be a replica of the original file before it went through your compression algorithm.

This method will always run separately from `preprocessCompress()` and `compress()`. This means that any instance variables you use between `preprocessCompress()` and `compress()` **cannot** be used in this method. It is expected that this method runs on its own and calculates all the necessary information (such as the codes) using only the information given from the compressed file. This method therefore, will reimplement many operations you completed in the compression section. You will see significant repeated functionality, but there are still key differences you should watch out for.

Tips for Decompress

In this section, we will provide an additional high-level overview of the `uncompress()` method and provide some advice. **Before reading this section, you should read the how-to guide's decompression section. Additionally, read the writing compressed files section.**

Here is a brief overview of the execution order, behavior, and outline of the method:

1. First, you'll need to ensure that the file is a `.hf` file by checking its magic number.
2. Check the header format number to determine whether the encoding is using the tree format or the count format.
3. Read in the header to recreate either the frequency array (with count format), or the coding tree (with tree format). The best way to understand how the headers are represented in code is to reference this section in the how-to guide.
4. If the header uses the count format, recreate the frequency array and then follow the instructions of `preprocessCompress()` to recreate the Huffman Coding tree.
5. If the header uses the tree format, directly recreate and unflatten the tree.

6. Once you have recreated the tree, decode the compressed file and output the original values to the output file.
7. At this point, you have hopefully recreated the original file. Finally, return the number of bits written to the output file.

Additional Assignment Specifications

- You may not use the Java `PriorityQueue` class in any way on this assignment. **You are expected to implement your own fair priority queue class without consulting any external resources.** You may, but are not required to, use the Java `LinkedList` or Java `ArrayList` as your internal storage containers for the priority queue.
- When initially adding the `TreeNodes` containing the value-frequency pairs to the priority queue, you **must** add them in ascending order based on the value– i.e add the value 0 and its frequency, then the value 1 and its frequency, and so forth.
- Your priority queue must handle ties in a fair way. This means that if you add a value to the queue that has equal priority to an existing value, it must be enqueued after the existing values. For your assignment, this will occur in the form of two nodes having the same frequency. The node added to the queue last should be behind all other nodes of the same or lower frequencies.
- When dequeuing elements from the priority queue to build your Huffman Code Tree, the first element dequeued must be the left child of the new connecting node and the second element dequeued must be the right child of the new connecting node.
- **Do not rely on the Java `InputStream` `reset()` method.** The test harness we use has an `InputStream` that does not support that method and simply causes an exception. You will fail tests if you rely on or use this method.
- After each method returns, you should close the input and output streams with the `close()` method. This will ensure that the stream closes and adds the necessary padding to the end of the files.

General Assignment Advice

Due to the complexity of this assignment, we are providing you with tips and common pitfalls to avoid for this assignment.

- The best advice we can give is read the assignment handout and all the guides before writing any code. It is very important that you read the entire how-to and the secondary how-to on structure before progressing.
- Store the state / instance variables of `preprocessCompress()` for later use in `compress()`. However, do not utilize these values with `uncompress()` as it does assume that any other methods have been previously run.
- Many students struggle to understand the stream objects. We recommend reading this section of the how-to guide that explicitly covers how the bit reading/writing works. Additionally, many questions can be answered by reading the linked documentation of the start code.

- Remember to include the pseudo end-of-file (PEOF) constant as an additional node in your priority queue. More information about the PEOF value can be found here and in the lecture slides.
- **The most important thing you can do on this assignment is test your code incrementally.** We know that we say this for every assignment, but it is of utmost important here. You have so many working parts (like your PriorityQueue and HuffmanTree) that any error in these could cause further issues down the link. **Instead of writing the entire `preprocessCompress()` method and then starting to test, we highly recommend you test your data structures first, and then proceed.**
- Your code should use multiple different methods to break up the responsibilities of the assignment. A lot of the work needs to be handled by outside data structures and can be repeated.
- While not required, some students find it beneficial to put the compression and decompression work in respective `Compressor` and `Decompressor` classes. This can make the `SimpleHuffProcessor` class very light and very organized.
- Although it was mentioned prior, please be aware of this section on exceptions. Students typically opt for the first approach, but it is a matter of preference. It does need to be handled so you don't accidentally ignore the PEOF.
- For the Store Custom Format in the header format options, if this option is selected, the program should return an error message to the user and close the streams. This is not meant to be handled in any alternative manner.
- In the code you turn in, only make reference to the data type `IHuffViewer`. Do not refer to any potential implementations of `IHuffViewer` such as `GUIHuffViewer`. There is no guarantee any particular implementation of `IHuffViewer` will be used when we grade.

Testing SimpleHuffProcessor

We have provided you with the `Huff.java`, `HuffMark.java`, and `Diff.java` files to make executing, testing, and debugging your code easier.

In order to perform the counting, compression, or decompression, you must use the `Huff.java` program, which will create a pop-up graphical interface for you to use to execute the SimpleHuffProcessor methods. The `Huff.java` program will handle much of the trivial parts of the compression, such as setting up the input and output streams, and making the new files. Through this graphical interface, you will be able to make and save new compressed and decompressed files.

The how-to guide depicts several different windows and details the expected behaviors of the graphical interface. Most students find using it to be fairly intuitive. On the upper left hand side, there is a 'Header Format' tab where you can switch between the count format and the tree format. In the 'Options' tab, you can force compression. Under the 'File' tab, you have the

option to either count, compress, or decompress a file. Selecting one of these options will call the associated methods you wrote in `SimpleHuffProcessor`.

Note that the mystery files are meant for you to test solely your decompression. There are many students whose implementations work because of a consistent misunderstanding in their compression and decompression. These files should hopefully expose any such issues.

For the `smallTxt.txt` file and the `ciaFactbook2008.txt` file, we have provided you with the correct, expected return values for the `preprocessCompress()` and `compress()` methods:

Expected return values for `smallTxt.txt` are:

Standard Count Format:	Standard Tree Format:
preprocessCompress returns -8138	preprocessCompress returns -120
compress returns 8346	compress returns 328

Expected return values for `ciaFactbook2008.txt` are:

Standard Count Format:	Standard Tree Format:
preprocessCompress returns 28867432	preprocessCompress returns 28874526
compress returns 48230392	compress returns 48223298

Do not use `System.out.println()` in this assignment. The `HuffViewer` class provides multiple methods to print debugging information to your graphical interface. The viewer uses the `showError()`, `showMessage()`, and `update()` methods. These are covered in the associated documentation.

Note that it is **not** possible to display the information inside of `.hf` files. These are not standard file formats and will not output readable information if you attempt to read them. However, by using the `Diff.java` class we have provided to you, you can check if the `.hf` files your program creates match the sample `.hf` files we provide to you. This class will tell you the first occurrence of a bit difference, allowing for exact comparison of these normally unreadable files.

Finally, you can use the `HuffMark.java` class to test large directories of files. This class is intended to be used on directories and will convert all non-`.hf` files into `.hf` files. The `HuffMark` program does not provide a viewer for your `SimpleHuffProcessor`, so you have to comment out any calls to `showString()` or `viewer.showMessage()`. You can leave in the calls to `showError()`, assuming you won't suffer any errors. The method calls may be in other classes if you created separate `Compressor` and `Decompressor` classes.

The expected return values of the file compression for the BooksAndHTML folder are below for your comparison (note the timing data does **not** have to match):

```
compressing to: C:\Users\scottm\Documents\EclipseProjects\A11_CS314_Huffman\BooksAndHTML\A7_Recursion.html.hf
A7_Recursion.html from   41163 to        26189 in        0.040
compressing to: C:\Users\scottm\Documents\EclipseProjects\A11_CS314_Huffman\BooksAndHTML\CiaFactBook2000.txt.hf
CiaFactBook2000.txt from         3497369 to      2260664 in      0.280
compressing to: C:\Users\scottm\Documents\EclipseProjects\A11_CS314_Huffman\BooksAndHTML\jnglb10.txt.hf
jnglb10.txt from         292059 to       168618 in       0.030
compressing to: C:\Users\scottm\Documents\EclipseProjects\A11_CS314_Huffman\BooksAndHTML\kjv10.txt.hf
kjv10.txt from   4345020 to      2489768 in      0.330
compressing to: C:\Users\scottm\Documents\EclipseProjects\A11_CS314_Huffman\BooksAndHTML\melville.txt.hf
melville.txt from        82140 to        47364 in        0.020
compressing to: C:\Users\scottm\Documents\EclipseProjects\A11_CS314_Huffman\BooksAndHTML\quotes.htm.hf
quotes.htm from  61563 to        38423 in        0.010
compressing to: C:\Users\scottm\Documents\EclipseProjects\A11_CS314_Huffman\BooksAndHTML\rawMovieGross.txt.hf
rawMovieGross.txt from   117272 to       53833 in        0.010
compressing to: C:\Users\scottm\Documents\EclipseProjects\A11_CS314_Huffman\BooksAndHTML\revDictionary.txt.hf
revDictionary.txt from   1130523 to      611618 in       0.090
compressing to: C:\Users\scottm\Documents\EclipseProjects\A11_CS314_Huffman\BooksAndHTML\syllabus.htm.hf
syllabus.htm from        33273 to        21342 in        0.010
compressing to: C:\Users\scottm\Documents\EclipseProjects\A11_CS314_Huffman\BooksAndHTML\ThroughTheLookingGlass.txt.hf
ThroughTheLookingGlass.txt from  188199 to       110293 in       0.020
--------
total bytes read: 9788581
total compressed bytes 5828112
total percent compression 40.460
compression time: 0.840
```

Empirical Analysis

It's time now to further analyze our Huffman Coding algorithm! This analysis will largely require the use of the **`HuffMark.java`** class mentioned above in the Testing section. You will provide the results of your analysis in a README.txt file (that you will create).

Run the `HuffMark` program on the files in `calgary.zip` (which represents the Calgary Corpus, a standard compression suite of files for empirical analysis), `waterloo.zip` (a collection of .tiff images used in some compression benchmarking), and `BooksAndHTML.zip` (contains a number of text files and html documents). Provide your benchmark results from the files in these folders in your README. You may want to modify our benchmarking program to print more data than it currently does, and include those results as well.

Additionally, in your README, answer the following questions.

1. What kinds of files lead to lots of compression?
2. What kinds of files saw little to no compression?
3. The benchmarking program skips files with `.hf` suffixes (as in, it does not compress already compressed files). Edit the code to remove this restriction. What happens when you try to compress a .hf file that has already been compressed?

Submission Checklist

Before you submit, run through our submission checklist to make sure you didn't forget anything! Did you remember to:

1. Follow the program hygiene guide?
2. Review and follow the assignment specifications in the syllabus (pg 6 - 7)?
3. Check the preconditions of code and throw exceptions as necessary?
4. Complete the `SimpleHuffProcessor` class and all of its methods?
5. Create and complete your own fair priority queue class?
6. Create and complete your own additional classes as necessary?
7. Avoid sharing instance variables between `compress()` and `uncompress()`?
8. Only make reference to the data type `IHuffViewer`, and avoid referring to any implementations of it?
9. Pass all of the included tests determined by matching return values and checking that your compressed and decompressed files are the same as the compressed and decompressed files we provide (with `Diff.java`)?
10. Conduct and record your analysis/benchmark results?
11. Include your results and analysis in a separate `README.txt` file?
12. Fill in the header comment of `SimpleHuffProcessor` for the file?
13. Include a header comment in all other classes that you created?

If you have done all these things, then you're ready to submit `SimpleHuffProcessor.java`, `README.txt`, and any additional classes that you created to Gradescope!

Thank you to Professor Mike Scott for sharing this assignment!
Based on an assignment created by Owen Astrachan of Duke University