Christian Znidarsic
EN.605.202 Data Structures
4/19/22

# Lab 3 Analysis

## Data Structure Description

For this lab, two main data structures were used.

### Min Heap

A min heap MinHeap() was used as a priority queue. The min heap was written to accept TreeNode objects. This heap was special in that, unlike a basic heap, it factored in several different characteristics of it's TreeNode contents (frequency, complexity, and alphabetical order) to determine priority. In other words, "min" was defined differently than it would in a regular min heap. It was not solely decided by which node had the smallest integer frequency value, but it was also decided by which nodes had complex keys, and which nodes had keys with the lowest letter in the alphabet.

### Queue

A queue was also used in this lab. The use of the queue was less fundamental to the project scope than the min heap was, but it served an important purpose. It was used mainly for error checking of input files. An input file would be read, one character at a time, while simultaneously being checked for errors. As each character is read, it is also pushed into a queue. The queue is then passed to external methods if the input is determined to be valid. The FIFO nature of the queue, unlike a stack, allowed the contents to be read out in the same order as they were read in. This characteristic lends itself particularly well to error checking, as the input file can effectively be read twice without creating multiple BufferedReader objects.

## Design

My program works using 11 total classes: Main, MinHeap, HuffmanTree, TreeNode, Queue, QueueNode, Validation, and four custom exception classes: InvalidChoiceException, InvalidClearTextException, InvalidEncodedException, and InvalidFrequencyTableException. The most significant classes are described in-depth below.

### Main

The main class operates primarily within one main while-loop. This while loop iterates until the program is complete. Within the while loop are several if(boolean) statements, and their arguments dictate the flow of the program. The function of the while loop is to allow erroneous file inputs from the user. If an exception, such as IOException is thrown, the loop just iterates again and the user gets another chance to enter a correct input. This allows smooth use and minimizes the consequence of mistakenly entering an invalid file path. It also allows for invalid files to be passed to the program. When these invalid files are passed, they are passed to methods in the Validate class. If the validate class returns that the input is false, an exception is thrown within main and the user is notified, and also given another chance to enter a valid file.

The user is first prompted to enter the name of a file containing a valid frequency table. If the table is determined to be valid, main calls HuffmanTree to convert the frequency table into a Huffman tree. At this point, the user is given two options. Enter E to encode a file, or enter D to decode a file. Depending on which option the user chooses, they are prompted to enter either a Clear Text file, or an Encoded text file. Main then calls Validate again to determine if the file is valid, and if it is, then main calls HuffmanTree to either decode or encode the file.

Main was designed to gracefully handle user error, while also providing enough flexibility to make encoding and decoding simple.

**Queue**

The Queue class specifies Queue objects. It includes two QueueNode fields, one is called front, and one is called rear. The constructor method sets two properties this.front and this.rear to both be null. This is the case when the queue is empty. The Queue class implements a linked list to store nodes. This allows items to be added and removed in O(1) time, and without indices. This also means that there are virtually no size limitations to the queue.

The method enqueue() takes the key value of a new node as input, and adds it to the rear of the queue. Because this method

The dequeue() method removes the node at the head of the queue.

The isEmpty method returns true when the queue has no values in it, and false when there is at least 1 node in the queue.

**QueueNode**

The QueueNode class is the smallest class in the program. It simply specifies a queue node. Each node has two properties: key and next. Key holds the data of the node as a char, and next is a pointer to the next node.

**MinHeap**

The MinHeap class defines a minheap. Its primary public methods are insertNode() and removeRoot(). As mentioned earlier, the MinHeap is particular about how it defines “min”. Nodes are compared using the criteria (in order of importance) frequency, key complexity, and key alphabetization. The key is stored in an array of chars to allow the array to be sorted. This allows access to the earliest alphabetical letter in each key array for each node.

**HuffmanTree**

The HuffmanTree() class does a lot of the heavy lifting in this program. It contains the two primary methods encode() and decode().

The method getHuffTree ()takes a queue of chars as input, and sets the root field to the resulting HuffmanTree. The queue of chars that is passed to it is the frequency table input by the user. It uses that frequency table to build TreeNodes, and one-by-one pushes them into the MinHeap. Then, it takes that MinHeap, pops two nodes, determines priority, combines them and pushes them back into the MinHeap

until the heap contains one node. It then sets the root field equal to that node, which holds the final HuffmanTree.

The getCodes() method takes the HuffmanTree created earlier, and outputs a hashmap that holds the key-value pairs of letter and code. The field called codes is set to equal this hashmap so it can be accessed at any time using HuffmanTree.codes.

**Justification**

To me this ordering made sense because main only performs user I/O and very general operations like calling other more complex methods.

Queue and QueueNode are both completely self-contained classes that have all of the fields and methods needed to implement a Queue data structure together. Queue objects can be created and manipulated cleanly. This is also the case with the HuffmanTree class, and the MinHeap class. These two classes, however, also contain methods that external methods can call to alter the contents of the respective classes.

## Errors Handled

**User I/O**

The user I/O is handled swiftly. If the user enters a file name that does not exist, they are given a message, and allowed to try again.

**Frequency Table files**

If the user enters a file that contains an invalid frequency table, they are given a message and allowed to try again. It is especially important that a valid frequency table be used, or else the rest of the program will not function properly.

The only valid format of a frequency table is:

A … 123 … \n
B … 456 … \n
…etc.

Where the ellipses represent any non-letter chars. If the input does not meet these criteria, the user is prompted to enter a different frequency table.

**Clear Text files**

The rules of a valid clear text file are less stringent than those of a frequency table. The only item that Validation.validateClearText() checks for is if the input clear text file contains any chars that are not either letters, spaces or newlines. If there is 1 or more char that does not fit any of these three categories, then the user is given a warning that the file contains non-letters, but the file is still accepted. It is

accepted because many common inputs may contain commas, periods or other punctuation, even though these chars are not encoded. This validation step is intentionally forgiving to allow flexibility while also warning the user that non-letters will not be encoded.

### Encoded Text files

A valid encoded text file must only contain 1s, 0s and newlines. If the input file contains characters that are neither of these three, then the user is given an error, and allowed to try entering a different encoded text file.

## Test Input Files Key

### ErrorFreqTable.txt

This file contains an invalid frequency table that will not be accepted by the program. It is the same frequency table as we were provided for this project, except it contains one extra letter. When this file is passed to Validation.validateFreqTable(), an InvalidFrequencyTableException will be thrown and the user will informed and prompted to enter a valid frequency table file.

### ErrorEncoded.txt

This file contains the same encoded text as Encoded.txt, except there is one erroneous letter in it. When this file is entered, an InvalidEncodedException is thrown and the user is informed and prompted to enter a different encoded text file.

### SampleClearText1.txt

This file contains the same clear text as ClearText.txt, except with all the punctuation removed. The purpose of this file is to demonstrate how, when an input clear text file has no punctuation, the warning to the user is not given.

### SampleClearText2.txt

This file contains two custom clear text lines to be encoded. They both contain punctuation, so the user *will* be given the warning regarding non-letters in the file, but it will still be accepted and encoded.

Christian Znidarsic
EN.605.202 Data Structures
4/19/22

# Compression

In this program I was able to achieve compression. If a typical char contains 8 bits, then we can say that, because this algorithm is able to assign codes between 3 and 8 bits to every char (in a 26 letter alphabet), the program achieved compression. The fine details regarding how much space the actual Strings holding the encoded data require could possibly mean that compression was not technically achieved, but I am assuming here that because the codes are fewer than 8 chars long each, that we achieved at least a representation of compression.

### Compared to conventional encoding

A conventional encoding scheme may simply assign codes in ascending binary order to each letter. For example:

A: 00
B: 01
C: 10
D: 11
E: 100
etc.

This simple encoding method may technically work, but it does not prioritize letters that are used more frequently. Using Huffman encoding optimizes the length of codes for more frequently used letters, and reserves longer codes for less frequently used letters.

### Different tie-breakers

One interesting way to tweak my program would be to change tie-breakers when building the min heap and also when constructing the Huffman Tree supernodes. Currently, simple nodes are given priority over complex nodes. If this were reversed, I do not think it would make a big difference. It would change the encoding, however. But with respect to efficieny, it shouldn't make a difference. The same goes for switching whether earlier letters in the alphabet have priority over later letters.

# Efficiency

### Time Complexity

This program has some O(n) elements in encode() and decode(), as well as a few other methods. However, the percolateDown() method in the MinHeap class is O(nlogn). Thus, the time complexity of the program is O(1) + O(n) + O(nlogn) = **O(nlogn).**

Christian Znidarsic
EN.605.202 Data Structures
4/19/22

### Space Complexity

Each character read from the input file is stored in a queue, and then moved from the queue into nodes in a heap. Other variables are all O(1). Thus, in worst case scenario, all items are stored twice resulting in O(2n). Thus, the program has a space complexity of O(1) + O(2n) = O(n).

## Reflection

### Biggest Strength

The part of this lab that I am most proud of is how the program both functions correctly, and also handles a wide variety of user errors gracefully. The user is allowed infinite attempts to enter correct file names thanks to a loop in the main method. The program also first validates *all* input files before allowing them to be used by the rest of the program. This allows for significant user error and recovers from these errors swiftly, not interrupting the flow of the program.

### Biggest Difficulty

The biggest difficulty by far was not realizing that the min heap needed to be created not only based on node frequency, as one would expect a typical min heap to, but also on key complexity and alphabetization. I actually tried implementing this approach early on and spent a long time writing conditions to ensure that the min heap would always return the highest priority node. However, I did this in an inefficient way and ended up thinking in circles for a long time. I eventually ditched that approach and simply sorted the min heap based only on frequency. When I completed the other parts of the program, my results were not coming out correctly. They were extremely close to correct, but there were maybe one or two letters that had incorrect codes. After a long time I tried once again to implement more detailed priority comparisons within the min heap class, and after a while I figured it out. After solving that problem, the program functioned as expected and my results were in line with what was shown in the project outline.

### Conclusion

I am very proud of my work on this lab. I spent by far the most time on this lab compared to the previous two labs, both due to the difficulty of this lab as well as the time I wanted to spend making it better. I got started ahead of time and had enough time to implement all the features that I wanted in the program. I am proud of how I did especially on my last lab, and I feel that this lab is on par with my last one in terms of code functionality, error handling, and detailed analysis.