CSCI 3901 Assignment 2

Due date: 4pm (Halifax time) Wednesday, October 13, 2021 in Brightspace

Problem 1

Goal

Get practice in writing test cases.

Problem

Write test cases for the following code.

We want a system that schedules Masters thesis defenses. The system will gather thesis information from the student, including

- the student's name,
- the defense date and time,
- the thesis title,
- the thesis abstract,
- the name of the supervisor (or co-supervisors),
- the two thesis readers, and
- the defense chair.

All this information is mandatory.

The system then confirms the schedule with the supervisor and then the readers and the defense chair. Once everyone has agreed to the defense, the system posts an announcement 7 days before the defense. Agreement is done electronically, where the system e-mails the individual with a web link that allows them to approve or reject the defence. The readers and the defense chair cannot approve the defense until the supervisor or co-supervisors approves the defense.

The whole approval process is meant to be completed 2 weeks before the defense. However, some students submit the information in less time and some approvers, like the supervisors, readers, and the defense chair, may do their approvals late.

In this problem, assume that the system will receive its input that initiates the defence request from keyboard text data entry.

Group your unit tests as input validation, boundary tests, control flow tests, or data flow tests.

Notes

- You are not asked to write code to solve this problem.

- Only write test cases for this problem.
- Do not provide test data for your test cases. Do provide what the method is expected to return.
- I am looking for distinct test cases. Do not duplicate conditions across cases.
- Ensure that your test case description is short but also clear on what is being tested. Cases where we can't tell what is being tested will be discarded.

Marking scheme

- List of test cases, assessed based on completeness of coverage for the problem and distinctness of the cases – 5 marks

Problem 2

Goal

Implement a data structure from basic objects.

Problem

We will work on a program to compress text files. The compression will use a Huffman code and will be adaptive. More information on Huffman codes follows the problem description.

Huffman Code

A Huffman code replaces each character of a file with a bit sequence (a sequence of bit-values of 0 or 1). A regular ASCII character (standard text file) uses 8 bits for each letter. In the Huffman code, frequently-used letters will use fewer than 8 bits and infrequently-used letters will use more than 8 bits. On the whole, the final file uses fewer total bits and so is compressed.

Suppose that we are building a Huffman code for a set of characters $\{c_i\}$. We eventually create a binary tree where each leaf contains one of the letters and every letter is in exactly one leaf. The path from the root / top of the binary tree to the leaf tells us how to encode that letter. When we follow to a left child, we output a bit of 0. When we follow to a right child, we output a bit of 1. Each letter then has a unique sequence of 0's and 1's.

We must begin with the frequency of use f_i for each character c_i. The idea is to build the binary tree from the bottom to the top by combining the least frequent characters or subtrees into one tree until we have a single tree at the end.

More specifically:

1. Sort the characters by increasing frequency, breaking ties by the alphabetic order of letters. Call this sorted list S.
2. While S has more than 1 entry

   a. Remove the smallest and second smallest entries from the list. Call them X and Y. These could be characters or the roots of some small trees.
   b. Create a new node N with X as the left child and Y as the right child
   c. Create a frequency f_N for N as the combined frequency of X and Y.
   d. Insert N back into S, sorting by the frequency f_N and, if frequencies are tied, with N at the end of all items in S with frequency f_N
3. The final element in S is now the root of the tree that gives you the Huffman code.

The codebook of the tree is then the set of all letters at the leaves of the tree and their corresponding 0 and 1 (left and right) paths in the tree from the root to the leaf.

To encode a file, you output the codeword of each character in the file.

To decode a file, you must first have the binary tree; basic encodings usually store the letter frequencies at the start of the file so that you can rebuild the binary tree. Given the binary tree, you read the file to decode one bit at a time and follow the sequence of bits as a left and right child path from the root of the binary tree to a leaf. When you reach a leaf, you print the character at the leaf and then restart the next path from the root of the binary tree again.

Adaptation to the Huffman Code

The standard Huffman code requires that you do two passes through the file. The first pass counts the character frequencies from which to build the Huffman code. The second pass then encodes the file. We want to have a single pass of the file for our encoding. Consequently, we will tally the frequency of characters as we read the file and will update our code periodically.

Your Huffman code will begin with two special characters: “new-character” and “end-of-file”. New-character will be encoded as 0 and end-of-file will be encoded as 1 in the Huffman code (when sorting characters, have new-character and end-of-file be at the end of the sorted list, in that order). When we encounter an input character c that is not in our Huffman code, we do the following:

- Output the code for new-character
- Output the single character c
- Add c to the Huffman code by finding the rightmost character in the Huffman tree, making two children for that rightmost node in the Huffman tree, making the existing character the left child, and making new character c the right child.

This first adaptation means that our rightmost side of the tree will get long quickly.

The second adaptation that we do is that we periodically rebuild the Huffman code using all of the frequencies currently in the tree. Rebuilding the tree will shorten these long codes. The rebuilding is triggered by us processing a number of characters from the input file. We start by rebuilding after 2 characters. Next, we rebuild after 4 characters (from the last rebuild time). The next rebuild is after 8 more characters, then 16, then 32, following increasing powers of 2.

These powers of 2 stop growing at a maximum number. That maximum number is supplied when we start the encoding process.

We include a third adaptation to the tree. When the Huffman code is rebuilt, we can either start counting the characters used at 0 again or we can carry forward the character counts that already exist. The choice of behavior is also supplied when we start the encoding process and doesn't change through the whole encoding process.

Your task

Write a class that will encode and decode a file's content using the adaptive Huffman code. Your class should implement the following interface class:

```
public interface FileCompressor {
    boolean encode ( String input_filename, int level, boolean reset,
                                    String output_filename );
    boolean decode ( String input_filename, String output_filename  );
    Map<Character, String> codebook ( );
}
```

The functions have the following semantics:

encode	Encodes the content if input_filename using the adaptive Huffman code and stores the output into outpue_filename. Re-calculation of the Huffman code, after the adaptive growing phase, happens after every 2^{level} characters are coded. If "reset" is true then the re-calculated Hufffman code restarts the frequency counts at 0 for all characters (after building the code); otherwise, the frequency counts continue as they were before and represent the frequency of data seen in the whole file so far. The level is an integer between 0 and 9. That digit will be stored as the first character of the output file The reset Boolean value will be stored as the second character of the output file: 0 for false and 1 for true. Both the level and the reset value will be the ASCII character of the integer.
decode	Decodes the context of input_filename using the adaptive Huffman code (and the specification of level and reset encoded in the file) and stores the decoded file as output_filename.
codebook	Return a Map of all of the letters that appear in the Huffman code and a string of 0 and 1 characters that represent the character when coded by the Huffman code.

Your program won't do actual compression. To make debugging (and testing) easier, we will print the characters 0 and 1 into the "compressed" output file rather than print 0 and 1 as individual bits. Consequently, your output file will be about 8x larger than the actual compressed file would be.

Assumptions

- All characters in the text files to be encoded will be printable characters.

Constraints

- You may not use data structures or abstract data types from the Java Collection Framework for your data structure without prior permission other than Arrays, HashMaps, and TreeSets. You must build any other data structures that you want to use.
- Write your solution in Java.
- If in doubt for testing, I will be running your program on timberlea.cs.dal.ca. Correct operation of your program shouldn't rely on any packages that aren't available on that system.

Notes

- Think through the design of your data structure before you start to implement it.
- Although some marks are reserved for the efficiency of your solution, begin with getting a solution that works.
- Look at where the bulk of the marks are in the marking scheme to help focus your efforts.

Marking scheme

- Documentation (internal and external) – 3 marks
- Program organization and clarity – 5 marks
- Creating a basic Huffman code – 10 marks
- Adaptive Huffman code – 5 marks
- Ability to encode a file – 5 marks
- Ability to decode a file – 5 marks
- Return of a codebook – 3 marks
- Efficiency, based on one of the following two criteria – 4 marks
    - The efficiency of the data structure that you implemented or
    - If you chose a simple data structure to ensure that you can complete the work then the mark is on a very short description of why you chose this data structure plus a description of the data structure that you would have chosen (given more time / experience) and why.

Test cases

Standard "using a file" tests for both encode and decode (both file parameters)

- Null passed as file name
- Empty string passed as file name
- File not available (doesn't exist / permissions don't let you create the file)

Input file for encode

- File has one character on one line
- File has many characters on one line

- File has two lines
- File has many lines

Level tests for encode and for decode
- Set a level of 0
- Set a level of 9
- Set a mid-range level

Reset tests
- Look to a case that had no adaptation steps and so no invocation of possible resets
- Ask for no reset across two adaptation steps in the algorithm
- Ask for no reset once the adaptations are using the maximum window size
- Ask for reset across two adaptation steps in the algorithm
- Ask for reset once the adaptations are using the maximum window size

Huffman code
- Encode/decode an empty file
- Encode/decode a file with 1 character
- Encode/decode a file with character counts at the following sizes (level of 4) to check around adaptation boundaries and reaching a stabilized level for adaptations
    - 1 (no adaptation steps)
    - 2 (no adaptation steps)
    - 3 (one adaptation step)
    - 6 (one adaptation step)
    - 7 (two adaptation steps)
    - 14 (two adaptation steps)
    - 30 (three adaptation steps)
    - 50 (five adaptation steps)
- Encode/decode a file with all the same characters
- Encode/decode a file where each adaptation stage has its own character alone
    - With resets
    - Without resets
- Encode/decode a file that uses different characters exactly once
- Encode/decode a file that uses different characters in different combinations (regular text)
- Decode a file that has a non-number as the level
- Decode a file that has something other than 0 or 1 for the reset value
- Apply the decode method to a corrupted file (so won't decode) and ensure that we don't crash and don't create an inordinately large output file

Codebook
- Ask for codebook of empty encoding
- Ask for codebook of ending with 1 character added
- Ask for codebook after an adaptation step has had to happen

- Ask for a codebook after we've inserted many nodes on a right path and just before a rebuilding of the Huffman tree happens (to get really long codewords)