TCSS 342 - Data Structures
Assignment 4 - Compressed Literature

Version 1.1.4 (April 23rd, 2022)

Time Estimate: 10-15 hours

Learning Outcomes

The purpose of this assignment is to:

- Use your priority queue data structure for a specific task.
- Build and traverse a Huffman tree.
- Implement the Huffman compression algorithm.
- Write a binary file.

Overview

Standard encoding schemes like ASCII are convenient and relatively efficient. However we often need to use data compression methods to store data as efficiently as possible. I have a large collection of raw text files of famous literature, including Leo Tolstoy's *War and Peace* consisting of over 3 million characters, and I'd like to store these works more efficiently. David Huffman developed an efficient method for compressing data based on character frequency in a message.

In this assignment you will implement Huffman's coding algorithm by:

- counting the frequency of characters in a text file.
- creating one tree for each character with a non-zero count.
    - the tree has one node in it and a weight equal to the character's count.
- repeating the following step until there is only a single tree:
    - merge the two trees with minimum weight into a single tree with weight equal to the sum of the two tree weights by creating a new root and adding the two trees as left and right subtrees.
- reading (recursively) the code for the characters stored in leaf nodes from the path from root to leaf.
- using the code for each character to create a compressed encoding of the message.

To complete this you will upgrade one public class and implement one public class and three private class:

- MyOrderedList<Type>
- HuffmanEncoder
    - CodeNode
    - FrequencyNode
    - HuffmanNode

Formal Specifications

MyOrderedList<Type extends Comparable<Type>>
…
… +binarySearch(item : Type) : Type -binarySearch(item : Type, int start, int finish) : Type +get(index : int) : Type

Method summary

- `binarySearch` - Upgrade this method to return the item searched for. This is common when that item contains other data that we want. Our HuffmanEncoder will need this functionality to find nodes based on the character stored in them.
- `get` - Returns the element stored at `index` and `null` if the `index` is out of bounds. This method should run in O(1) time.

HuffmanEncoder
-inputFileName : String -outputFileName : String -codesFileName : String -book : BookReader -frequencies : MyOrderedList<FrequencyNode> -huffmanTree : HuffmanNode -codes : MyOrderedList<CodeNode> -encodedText : byte[]
+HuffmanEncoder() -countFrequency() -buildTree() -extractCodes(root : HuffmanNode, code : String) -encode() -writeFiles()

Field summary

- `inputFileName` - filename for the uncompressed input file
    - For testing default to “./WarAndPeace.txt”.
- `outputFileName` - filename for the compressed output file

    - For testing should be set to “./WarAndPeace-compressed.bin”.
- `codesFileName` - filename for the codes output file
    - For testing should be set to “./WarAndPeace-codes.txt”.
- `book` - A book reader initialized with the inputFileName.
- `frequencies` - A list that stores the frequency of each character in the input file.
- `huffmanTree` - The root of the Huffman tree.
- `codes` - A list that stores the codes assigned to each character by the Huffman algorithm.
- `encodedText` - The encoded binary string stored as an array of bytes.

Method summary

- `HuffmanEncoder` - The constructor should call the helper methods in the correct order to carry out Huffman’s algorithm.
- `countFrequency` - This method counts the frequency of each character in the book and stores it in `frequencies`.
    - Iterate through the text character by character maintaining counts in `frequencies`.
    - The counts are stored in FrequencyNodes.
    - It should output the time it takes to count the frequencies.
- `buildTree` - This method builds the Huffman tree and extracts the codes from it, storing them in `codes`. It does so by carrying out these steps:
    - Create a single Huffman node for each character weighted by its count.
    - Add all the nodes to a priority queue.
    - Merge Huffman nodes until only a single tree remains.
    - Store the root of the remaining tree in huffmanTree.
    - Extract the codes from the tree and store them in `codes` using the recursive helper function like this:

      extractCodes(huffmanTree,"");
    - It should output the time it takes to build the tree and extract the codes.
- `extractCodes` - A recursive method that traverses the Huffman tree to extract the codes stored in it.
    - This method will conduct a recursive depth-first traversal of the Huffman tree.
    - The path of left and right moves is stored in the code parameter by adding “0” for left traversals and “1” for right traversals.
    - When a leaf is reached the code is stored in the `codes` list.
- `encode` - Uses the `book` and `codes` to create `encodedText`.
    - For each character in text, append the code to an intermediate string.
    - Convert the string of character into a list of bytes and store it in encodedText.

    - You can convert a string of ‘0’s and ‘1’s to a byte with this line:

      byte b = (byte)Integer.parseInt(str,2);
    - It should output the time it takes to encode the text.
- `writeFiles` - Writes the contents of `encodedText` to the `outputFileName` and the contents of `codes` to `codesFileName`.
    - It should output the time it takes to write the files.

```
FrequencyNode implements Comparable<FrequencyNode>
--------------------------------------------------
+character : Character
+count : Integer
--------------------------------------------------
+compareTo(other : FrequencyNode) : int
+toString() : String
```

Field summary

- `character` - The character that this count is for.
- `count` - The count for this character.

Method summary

- `compareTo` - This method compares the nodes based on the `character` stored in them.
- `toString` - This method returns the contents of the node in this format:

  `character:count`

```
HuffmanNode implements Comparable<HuffmanNode>
--------------------------------------------------
+character : Character
+weight : Integer
+left : HuffmanNode
+right : HuffmanNode
--------------------------------------------------
+HuffmanNode(ch : Character, wt : Integer)
+HuffmanNode(left : HuffmanNode, right : HuffmanNode)
+compareTo(other : HuffmanNode) : int
+toString() : String
```

Field summary

- `character` - The character that this node stores. Only leaves store characters.

Internal nodes have no characters.

- `weight` - The weight of the Huffman tree rooted at this node.
- `left` - The root of the left sub-tree.
- `right` - The root of the right sub-tree.

Method summary

- `HuffmanNode` - There are two constructors for Huffman nodes. The first is to create the initial leaf nodes. The second is used when merging two nodes.
- `compareTo` - This method compares the nodes based on the `weight` stored in them.
- `toString` - This method returns the contents of the node in this format:

```
character:weight
```

CodeNode implements Comparable<CodeNode>
+character : Character +code : String
+compareTo(other : CodeNode) : int +toString() : String

Field summary

- `character` - The character that this node stores the code for.
- `code` - The code assigned to this character.

Method summary

- `compareTo` - This method compares the nodes based on the `character` stored in them.
- `toString` - This method returns the contents of the node in this format:

```
character:code
```

Testing

It is important that you test your code to ensure it works on all potential inputs. Please do this in a separate Main class, and without using additional tools (like JUnit). You will not submit your test code. Instead, your data structures will be tested by code developed by the instructor and/or grader. This code will not be provided to you, as you should test your code before submitting it. If your code fails our tests we will let you know which tests it fails so you can find and correct your errors.

Here is the output from my solution:

```
Reading input file "./WarAndPeace.txt"... 3291642 characters in 66 milliseconds.

Finding words and adding them to a linked list...  in 333 milliseconds.
The linked list has a length of 570240.

Counting frequencies of characters... 86 unique characters found in 229
milliseconds.

Building a Huffman tree and reading codes...  in 2 milliseconds.

Encoding message...  in 280 milliseconds.

Writing compressed file... 1875165 bytes written in 2 milliseconds.
```

Submission

You will submit a .zip file containing:

- MyArrayList.java
- MyLinkedList.java
- MyOrderedList.java
- MyPriorityQueue.java
- BookReader.java
- HuffmanEncoder.java

Grading Rubric

In order to count as complete your submission must:

1. Implement the Huffman algorithm without errors.
2. Get the numbers right:
    a. You should read in 3291642 characters from the input file.
        - 3291623 characters are detected if you use UTF-8 instead of ASCII.
    b. You should detect 86 unique characters in the input file.
        - 90 characters are detected if you use UTF-8 instead of ASCII.
    c. You should output 1875165 bytes to the compressed file.
        - 1875128 bytes are written if you use UTF-8 instead of ASCII.
3. Be as efficient as possible.
    a. You don’t have to be as fast as my demo, but taking much longer for any step suggests a bug or inefficiency.

Reminder: Incomplete assignments can always be corrected and resubmitted. If they are completed within 7 days of the due date they will count as late and after that period they will count as missed. Please review the grading matrix for the number of permitted late and missed assignments.