TCSS 342 - Data Structures
Assignment 9 - Compressed Literature 3 (Word Codes)

Version 1.2.4 (May 26, 2022)

Time Estimate: 6-10 hours

Learning Outcomes

The purpose of this assignment is to:

- Use your hash table data structure for a specific task it is well suited for.
- Build and traverse a Huffman tree.
- Implement the Huffman compression algorithm.
- Write a binary file.

Overview

Huffman's algorithm works on strings of symbols. We already ran Huffman's algorithm using each character in the text as a symbol, but we can run it again using the words (and separators) as symbols. To do this we will make a minor upgrade to the BookReader so that it stores the separators as well as the words when it parses the book for the words. Then we can run the Huffman algorithm treating these as the symbols.

We will also upgrade our Huffman algorithm so that it works with our new MyHashMap instead of the OrderedLists we previously used to count the frequencies and store the codes. This will also mean that we no longer need the FrequencyNodes or CodeNodes, which is another benefit of working with maps.

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

- BookReader
- HuffmanEncoder
    - HuffmanNode

Formal Specifications

BookReader
… +wordsAndSeparators : MyLinkedList<String>
…

+parseWords()

Field summary

- `wordsAndSeparators` - A list that stores both the words and separators in the order that they appear in the book.

Method summary

- `parseWords` - This method should be upgraded to fill `wordsAndSeparators` as well.

HuffmanEncoder
… -wordCodes : boolean -frequenciesHash : MyHashTable<String, Integer> -codesHash : MyHashTable<String, String>
… -countFrequency() -buildTree() -extractCodes(root : HuffmanNode, code : String) -encode() -writeFiles()

Field summary

- `wordCodes` - A boolean that determines if the HuffmanEncoder uses character codes or word and separator codes.
    - For this assignment hard code this boolean to true.
- `frequenciesHash` - A hash table that stores the frequency of each word or separator in the input file.
- `codesHash` - A hash table that stores the codes assigned to each word and separator by the Huffman algorithm.

Method summary

- `countFrequency` - This method should be upgraded to use the hash table `frequenciesHash`.
- `buildTree` - This method should be upgraded to use the hash table `frequenciesHash`.

- `extractCodes` - This method should be upgraded to use the hash table `codesHash`.
- `encode` - This method should be upgraded to use the hash table `codesHash`.
- `writeFiles` - This method should be upgraded to use the hash table `codesHash`.

HuffmanNode implements Comparable<HuffmanNode>
… +word : String
… +HuffmanNode(word : String, wt : Integer) +toString() : String

Field summary

- `word` - The word or separator that this node stores. Only leaves store words or separators. Internal nodes have no characters.

Method summary

- `HuffmanNode` - Upgrade the constructor to take a string.
- `toString` - This method returns the contents of the node in this format:

`word:weight`

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 75 milliseconds.

Finding words and adding them to a linked list...  in 583 milliseconds.
The linked list has a length of 570240.

Counting frequencies of words and separators... 20251 unique words and
separators found in 139 milliseconds.

Building a Huffman tree and reading codes...  in 56 milliseconds.
```

Encoding message... in 254 milliseconds.

Writing compressed file... 1035074 bytes written in 96 milliseconds.

Submission

You will submit a .zip file containing:

- MyArrayList.java
- MyLinkedList.java
- MyHashTable.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 20251 unique words and separators in the input file.
        - 20255 characters are detected if you use UTF-8 instead of ASCII.
    c. You should output 1035074 bytes to the compressed file.
        - 1035040 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.