TCSS 342 - Data Structures
Assignment 4 - Compressed Literature 2

Guidelines

This assignment consists of programming work with written work as extra credit. Solutions should be a complete working Java program including both your work and any files provided by the instructor used in your solution. These files should be compressed in a zip file for submission through the Canvas link.

This assignment is to be completed on your own or in a group of two. If you choose to work in a group of two this must be clear in your submission. Please see the course syllabus or the course instructor for clarification on what is acceptable and unacceptable academic behavior regarding collaboration outside a group of two.

Assignment

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 about 22000 different words (counting differences in capitalization), and I'd like to store these works more efficiently. David Huffman developed a very efficient method for compressing data based on character frequency in a message. We applied this method in a straightforward way in Assignment 3 to compress the literature but I have a better idea on how to apply Huffman's encoding.

Since all the literature in my collection is in the English language it is reasonable to suppose they will not just have commonly used characters but commonly used words as well. My idea is that if we can treat every distinct word as a symbol and every non-word character (white space, punctuation, etc.) also as a symbol then we can apply Huffman's encoding based on the frequency of words in the text instead of characters. I propose this will help us compress the text much smaller.

To carry this out we need an efficient way to store each word as we encounter it and the associated count it accumulates as we scan the entire document. While a binary search tree may work, the constant access time of a hash table is more attractive. You will implement a hash table to help store the word counts and eventual codes for each word. You can then modify a solution to Assignment 4 to count words instead of characters

In this assignment you will implement a hash table by:

- creating a table with 32768 (2^15) buckets.
- creating a hash function that for any key will produce an integer in the range [0...32767]
- implementing get and put methods (no need for remove) that hash a key and then retrieve or store a value from the proper bucket.
- uses linear probing to handle collisions.

- **(Optional)** uses another collision handling strategy other that linear probing *except* chaining.

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

- counting the frequency of words and separators in a text file.
    - for the purposes of this assignment a word will count as string of characters from the set {0,...,9,A,...,Z,a,...,z,',-}. Notice the ' and capital letters may appear in the string for words like "wouldn't've" and "d'Eckmuhl". Every other character is a separator and will be handled as a string of length one (i.e. " ", "\n", "!", etc).
    - use a hashtable of your creation (see above) to store each word and separator in the hash table with its count.
- creating a tree with a single node for each word or separator with a non-zero count weighted by that 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.
- labelling the single tree's left branches with a 0 and right branches with a 1 and reading the code for the strings stored in leaf nodes from the path from root to leaf.
- using the code for each string to create a compressed encoding of the message.
- **(Optional)** provide a method to decode the compressed message.

You are also responsible for implementing a Main controller that uses the CodingTree class to compress a file. The main must:

- Read the contents of a text file into a String.
- Pass the String into the CodingTree in order to initiate Huffman's encoding procedure and generate a map of codes.
- Output the codes to a text file.
- Output the compressed message to a binary file.
- Display compression and run time statistics.

Formal Specifications

You are responsible for implementing the MyHashTable<K, V> class that must function according to the following interface:

- MyHashTable<K, V>(int capacity)
    - creates a hash table with capacity number of buckets (for this assignment you will use capacity 2^15 = 32768)
    - K is the type of the keys
    - V is the type of the values
- void put(K searchKey, V newValue)
    - update or add the newValue to the bucket hash(searchKey)
    - if hash(key) is full use linear probing to find the next available bucket

- V get(K searchKey)
    - return a value for the specified key from the bucket hash(searchKey)
    - if hash(searchKey) doesn't contain the value use linear probing to find the appropriate value
- boolean containsKey(K searchKey)
    - return true if there is a value stored for searchKey
- void stats()
    - a function that displays the following stat block for the data in your hash table:

```
Hash Table Stats
================
Number of Entries: 22690
Number of Buckets: 32768
Histogram of Probes: [14591, 3419, 1510, 859, 479, 337, 238, 169, 166, 100, 90,
78, 53, 42, 51, 54, 29, 28, 18, 17, 21, 20, 17, 15, 12, 10, 12, 11, 6, 4, 12, 4, 9, 13,
5, 4, 7, 0, 0, 3, 2, 3, 2, 3, 5, 1, 3, 2, 1, 2, 6, 2, 1, 3, 1, 1, 2, 3, 3, 0, 2, 2, 1, 1, 1, 1,
2, 4, 3, 2, 1, 0, 2, 1, 3, 0, 0, 2, 2, 1, 2, 3, 2, 1, 3, 0, 0, 0, 0, 0, 0, 3, 2, 0, 0, 0, 1, 1,
0, 0, 1, 0, 0, 0, 0, 1, 0, 1, 2, 0, 0, 1, 0, 0, 0, 1, 2, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1,
0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 1,
0, 2, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0,
0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0,
0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0,
1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
1, 0, 1, 0, 2, 2, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1,
0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 1,
1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1]
Fill Percentage: 69.244385%
Max Linear Prob: 533
Average Linear Prob: 3.434773
```

- *private* int hash(K key)
    - a *private* method that takes a key and returns an int in the range [0…capacity].
- String toString()
    - a method that converts the hash table contents to a String.

You are responsible for implementing the CodingTree class that must function according to the following interface:

- CodingTree(String fulltext)

    - a constructor that takes the text of an English message to be compressed.
    - The constructor is responsible for calling all methods that carry out the Huffman coding algorithm and ensuring that the following property has the correct value.
- MyHashTable<String, String> codes
    - a **public** hash table of words or separators used as keys to retrieve strings of 1s and 0s as values.
- String bits
    - a **public** data member that is the message encoded using the Huffman codes.
- **(Optional)** String decode(String bits, Map<String, String> codes) - this method will take the output of Huffman's encoding and produce the original text.

You will also create a Main class that is capable of compressing a number of files and includes methods used to test components of your program.

- void main(String[] args) - this method will carry out compression of a file using the CodingTree class.
    - Read in from a textfile. You may hardcode the filename into your program but make sure you test with more than one file.
    - Output to two files. Again feel free to hardcode the names of these files into your program. These are the codes text file and the compressed binary file.
    - Display statistics. You must output the original size (in bits), the compressed size (in bits), the compression ratio (as a percentage), the elapsed time for compression, and the hash table statistics.
    - Component testing. Include all methods used for component testing and debugging.
- Test files - include at least one test file that is a piece of literature of considerable size.
    - Check out Project Gutenberg an online database of literature in text format.

To implement your CodingTree all other design choices are left to you. It is strongly encouraged that you use additional classes and methods and try to use the built in data structure whenever possible. For example, in my sample solution I make use of a private class to count the frequency of each string, a private node class to implement my tree, a recursive function to read the codes out of the finished tree, and a priority queue to handle selecting the minimum weight tree.

Empirical Analysis **(Optional)**

In order to get extra credit for implementing another collision handling strategy you must provide a short analysis of the performance of your strategy to linear probing. Include in this analysis which of the two methods you think worked better with this data (if either) and provide evidence in the form of statistics to back up your analysis.

Submission

The following files are provided for you:

- WarAndPeace.txt - the plain text of Leo Tolstoy's novel War and Peace.
- codes.txt - An appropriate set of codes produced by my sample solution. (Note: This is not a unique solution. Other proper encodings do exist.)
- compressed.txt - The compressed text of the novel. It's size is the most relevant feature.
- trace.txt - Sample console output from my solution.

You will submit a .zip file containing:

- Main.java - the simulation controller.
- CodingTree.java - the completed and functional data structure.
- MyHashTable.java - the completed and functional data structure.
- <sample>.txt - a novel or work of art in pure text for that you have tested your program on.
- compressed.txt - the compressed version of your selected text.
- codes.txt - the codes produced on your selected text.
- **(Optional)** Analysis.pdf - the written analysis of your better collision handling strategy.

Grading Rubric

Each of the following will be awarded **one or more** points toward your assignment grade. Not all points need to be achieved to receive a perfect grade. Excess points contribute to the total points gathered for the quarter. Excess points at the end of the quarter will convert into a bonus assignment grade.

CodingTree

- Correct implementation of each of the steps of Huffman's algorithm.
- Proper selection and use of data structures.
- Efficient implementation of all steps.
- Proper definition of private data structures.
- The steps of the algorithm that will be independently graded are:
  - Counting characters.
  - Initializing one tree per character.
  - Building Huffman tree.
  - Extracting codes.
  - Encoding text.

MyHashTable

- Proper selection and use of data structures.
- Correct and efficient implementation of hash(key).
- Correct and efficient implementation of put(key, value), get(key), containsKey(key).
- Correct and efficient implementation of toString(key).
- Correct implementation of linear probing.
- Correct calculation and display of Hash Table Statistics.

Main

- Read text from input file.
- Output codes to text file.
- Output encoded text to binary file.
- Component testing or debugging code.
- Display run statistics.
- Extra test files.

Decode **(Optional)**

- Decoder creates uncompressed character string from a string of bits and the codes map.
- Decoder reads the compressed binary file.
- Decoder reads the codes text file.
- Decoder writes the uncompressed message to a text file.

Analysis **(Optional)**

- Correct implementation of an alternative collision handling strategy.
- Draw a conclusion on the comparative success of the two strategies on this data.
- Support your conclusion with empirical evidence from your program.

Extra Points

- Compiles first try.
- All and only the files requested are submitted.
- First graded submission. (That is, not a resubmission after a grade awarded.)
- On time.
- No debugging help.

Tips for maximizing your grade:

- Make sure your classes match the interface structure exactly. I will use my own controller (Main.java) and test files on your code and it should work without changes. Do not change the method name (even capitalization), return type, argument number, type, and order. Make sure all parts of the interface are implemented.
- Only zip up the .java files. If you use eclipse these are found in the “src” directory, not the “bin” directory. I can do nothing with the .class files.
- All program components should be in the default package. If you use a different package it increases the difficulty of grading and thus affects your grade.
- Place your name in the comments at the top of every file. If you are a group of two make sure both names appear clearly in these comments.