File Compressor

File Compressor is a program for losslessly compressing/decompressing files.

Description

File Compressor is a program that is capable of losslessly compressing text files into binary .hcz files and then decompressing those back to the original text file without any differences. It does this using 3 different flags, -b for build codebook for text file using Huffman coding, -c for compress text file into .hcz file using codebook, and -d for decompress .hcz file into text file using codebook. Building a codebook uses a hashmap to measure token frequencies, and then a min heap to build a Huffman tree that contains binary codes for each token. Additionally, these commands can be applied recursively to a directory by appending the -R flag: then the command is applied to every text file found in the directory as well as the text files within directories found in the directory. Note that recursively building a codebook produces one codebook that includes the tokens for all the text files and text files in directories within the directory.

Usage

```
./fileCompressor <flag(s)> <path or file> |codebook|
./fileCompressor -b someFile
./fileCompressor -c someFile someCodebook
./fileCompressor -d someFile.hcz someCodebook
./fileCompressor -R -b someDirectory
./fileCompressor -R -c someDirectory someCodebook
./fileCompressor -R -d someDirectory someCodebook
```

Implementation

Hash table We chose to use an open-addressing scheme with linear program for our hash table. Open addressing generally performs better than closed addrsesing and chaining with the advantage of locality of reference and additionally, we do not have to allocate memory during insertions unless we are rehashing the table, lowering chances of a failure immeasurably. Finally, we do not have to implement delete functions in this project, which simplifies this open addressing scheme greatly. We chose a load factor of 0.5 as a tradeoff between speed and space as generally the user isn't compressing gigabyte-size files, and many prefer a faster speed over a lower temporary memory usage.

Build Codebook To build the codebook, we first read in the input file. Then, for each token, we insert it into the hash table. If we see the same token, we increase the frequency of that inserted token by 1. Once we have read in every token and delimiter, we have all the frequencies of the words in the hash table. Then, we iterate through the hash table and we insert each token, frequency pair into a heap. We sort the heap, then create a Huffman tree using the algorithm described in the assignment. Once we have the Huffman tree, we generate an escape code. With this, we traverse through the Huffman tree and print out each token and it's corresponding 'code' that is associated with it. After going through the entire tree, we are done.

Compress To implement compress, we read in the codebook and input file. Then we build the hash table from the codebook. For each token, delimiter pair in the input file, we would look them up in the hash table after processing the escape character and write those to the output file. Repeat for the entire file, and then we are done.

Decompress To implement decompress, we read the codebook and compressed file. We built the hash table once again from the codeobok. For each character in the compressed file, we checked it and an increasing substring after it in the hash table. Upon a match, we would print out the coressponding decoded text from the hash table to the decompressed file. We repeat this until the entire file is processed.

Time and Space Complexity

Let n be the number of tokens found. Let m be the number of unique tokens found. We will analyze the time and space complexities for each of the 3 basic commands. Let us first compute the complexities of the underlying data structures we use.

Hash table Time: To insert into the hash table, it takes us O(1) time, average and best case. In the worst case, it could take up to O(n) time. Retrieving an element is the same: O(1) time to get in average and best case, and O(n) in worst case. This is simply because we just have to compute the hash, which takes constant time, and look in that index of the hash table. In the worst case, we would have to check all the spots if all the words mapped to the same index; however this is very unlikely to happen. As for the space complexity, we rehash the table every time the load factor is exceeded. However, with a load factor of 0.5, we will only use 2n number of spaces. To surmise:

Space: O(n)

Insert: O(1) average, O(n) worst

Get: O(1) average, O(n) worst

Heap To insert a node, we put it at the bottom and sift up; the height is approximately log n, so there are log n comparisons and inserting takes O(lgn). The same logic applies to shift down. As for space, it takes O(n) space, as it is represented as an array and the items fill up the array perfectly to a power of 2. So, we have

Space: O(n)

Sift up: O(lgn) worst, average

Sift down: O(lgn) worst, average

IO Operations With IO operations, let k be the number of bytes in the file. Then, reading in a file will take O(k) time, since at most we read only a constant number of bytes into the program. The same applies for writing.

Read: O(k), best, worst, average

Write: O(k), best, worst, average

Build codebook Time: First, we must read in the file, taking O(k1) time. Then, we create the hash table. We iterate through each token, of which there are O(n), and insert O(m) of them into the hash table, taking O(1) time each. Average: O(n) Worst: O(nm). Then. we must add each entry into the heap and sort it. We insert m entries and sift up each of them, taking O(mlgm) time on average and worst. Then, we create the Huffman tree by removing two nodes, creating a subtree, and inserting it back. This is two sift down and one sift up operations, making it O(lgm) as well. We do this m times at most, creating the tree in O(mlgm) time. Finally, we need to print out this information into the codebook. To generate the escape code, we add another ! every time we see an entry with n !'s. This takes O(m) time to do. To print out the codebook, we recursively traverse the BST that is the huffman tree. This takes O(m) time, and for each node, we simply print the 'code' value generated and also the token, taking O(1) time each. So, printing the tree is O(m) time. In total we have:

Average case: O(n+mlgm+k1+k2)

Worst case: O(nm mlgm+k1+k2)

Where k1 is the # bytes in input file and k2 is the # bytes in the output codebook.

Space: We used a hash table, which is O(m) space, and the heap, which is O(m) space. Together with the files we read, the space is Average and worst: O(m+k1).

Compress Time: To compress a file, we first read in the codebook and insert each pair of (code, token) into a hash table. Reading takes O(k1) time, insertion of each m unique tokens will take O(1) * m time on average. Thus we have O(k1+m) on average and O(k1+m^2) in the worst case. Next, we will read in the file we are compressing, this will take O(k2) time, where k2 is the number of bytes in this file. After, we take each token from the input, get it's corresponding code from hash table, and write it out. Getting all n tokens, each O(1) time, leads to a O(n) average or O(nm) worst case. Finally, we must write it out: assuming the corresponding code is a constant size, we must write out k3 bytes. In total, we have:

Average case: O(n + m + k1+k2+k3)

Worst case: O(nm + m^2 + k1+k2+k3)

where n is the number of tokens and m is the number of unique tokens, and k1, k2, k3 are the number of bytes in the codebook, inputfile, and output file respectively.

Space: We have the hash map which we read into which is O(m) space. Additionally, we hold the input files, k1 and k2.

Average and worse case: O(m + k+1+k2)

Decompress Time: Decompress is similar to compress. Let k1 be the bytes in the codebook, k2 be the bytes in the compressed file, and k3 be the bytes in the output file. For IO alone, we will take O(k1+k2+k3) time. Once we have read in the compressed file, we can start decoding it. We iterate through the string, and attempt to match one character, then 2, etc. until we find a match. When we find a match, we repeat the progress beginning at the end of the last match. Thus, for each character we perform a get on the hashmap, which takes O(1) time average case. The length of the huffman code is at most m and log(m) on average, and there are at most n tokens, so we perform nm gets worst case and nlgm on average. Thus, the runtime is:

Average case: O(nlogm + k1+k2+k3)

Worst case: O(nm + k1+k2+k3)

Space: O(m + k+1 + k2)