C: FILECOMPRESSION MANUAL

WRITTEN BY ARYAN SHAH AND ALLEN ZHANG

NAME

fileCompressor- compresses and decompresses file(s) encoded by a Huffman Codebook

SYNOPSIS

./**fileCompressor** <flag> <path or file> |codebook|

DESCRIPTIO N

Creates a Huffman codebook for a file or files in a directory and its subdirectories, compressing the files following the encoding in the codebook, and decompresses compressed files in the format .hcz

If <flag> or <path or file> or |codebook| flags are missing or incorrect, return fatal error.

Options:

-b: build codebook

-will create a Huffman Codebook in the same directory as ./fileCompressor from <file>

-c: compress

-will create a compressed file with extension .hcz from <file> compressed by |codebook| called ./HuffmanCodebook

-d: decompress

-decompresses <file> with .hcz extension into original file with |codebook| called ./HuffmanCodebook

-R: recursive

-followed by another flag -b,-c,-d and applies operation to all files and subfolders of <path>

Design and Implementation

Building Codebook:

Building the codebook first requires reading the file for tokens and inserting into an AVL Tree. If the token is in the AVL Tree, we update its occurrence counter, otherwise we create a new node. Since inserting into an AVL Tree has a time complexity of O(log(n)), inserting n tokens will have a time complexity of O(nlog(n)). The space complexity is O(n) in the worst case, when every token is unique, since we need a node for every token. Then we traverse through the AVL Tree and insert into a minheap, which has a space complexity of O(n). Traversing the AVL tree once has a time complexity of O(n), since we are just going to the left and right of each node and creating a heap node.

Heapifying the heap into a minheap has a time complexity of O(n). We then iterate through the heap and assign bitcode values for every node that has a token in it, which has a time complexity of O(n); Then we search for each token in the AVL Tree and set its bitCode pointer to the bitcode value calculated by the minheap. Since we search in an AVL n times, we have a time complexity of O(nlogn). Then we iterate through AVL tree and call the write function to write output into a file called HuffmanCodebook, which has a time complexity of O(n).

Compressing Codebook:

For compressing a file, first we read the codebook provided and create an alphabetically sorted AVL Tree with nodes containing tokens and their respective bitcodes. The worst case time complexity is O(nlogn) and space complexity is O(n), for n unique tokens in the codebook. Then we read the file we want to compress, and search for the token in the AVL Tree. Once the token is found, we get the respective bitcode and write it to the compressed file. If there are m tokens in the file, then the time complexity for searching for all of them and writing to the compressed file is O(mlogn);

Decompressing Codebook:

For decompressing, we read the codebook and create an AVL Tree sorted by bitcodes containing tokens and their respective bitcodes. The worst case time complexity is O(nlogn) and space complexity is O(n), for n unique tokens in the codebook. We read the compressed file and search for the bitcode in the codebook. Assuming there are m tokens in the .hcz file to decompress, the time complexity of searching for each token m times in an AVL tree of size n is O(mlogn). The token returned by the search is then written to a compressed file.

Recursive Flag:

The recursive flag simply requires us to recursively iterate through all the files in a directory and its subdirectories. We implemented a recursive version of build,compress,and decompress by have a function recursively traverse through a folder and call the build,compress,or decompress functions depending on the flag.