Salman Hashmi (sah285)

Usama Sajid (us71)

CS214 Systems Programming fileCompressor ReadMe

fileCompressor is a program developed in C meant to implement the Huffman Coding in order to compress and decompress files without losing data. Below is a description of the algorithm and runtime of the program.

Building the Huffman CodeBook

The first step, before encoding or decoding, was to build a codebook based on the Huffman algorithm. In order to do so, a couple of data structures were used, including a linked list and a minheap. Two structs were also used, each serving as a node for either the Huffman Tree or the linked list. The words were read from each file with a run time of O(n) for a single file, or O(m * n) for recursive, with m being the number of files to be read and n being the size of each file.

Linked List

First, a Linked List (LL) was created in order to store unique words and their frequencies. This was done by iterating through each file and pulling from it, word by word. Evertime a word was read, it would be inserted into the LL. In order to insert a word into the LL, the program would iterate through the existing values, if they exist. If the word read matched a word that has already been come across, then its frequency would be updated by one. If this was the first time that that word was come across, then it would be added to the end of the LL with a frequency of 1. Each LL node was comprised of a struct, known as tableNode, which was comprised of a char* to represent a word, an int to represent the frequency, and a tableNode pointer which points to the next node. An LL has a search runtime of O(n), and space complexity of O(n) as well.

MinHeap

After the frequencies were read from every file, they were entered into a MinHeap (MH). The MH always keeps the smallest value in the heap, and was imperative in building the Huffman Tree. The MH was sorted by the frequencies of the words. The MH nodes were comprised of a char* to store the word, an int to store its frequency, and two node pointers to point to a node's children. The child pointers were initialized to null initially when in the MH,

but would have values when building the Huffman Tree. The MH time complexity for sorting and searching is O(n * log(n)), and the space complexity is O(1).

Huffman Tree

The Huffman Tree (HT) was built using the MH. This was done by creating a new node, the same struct as the MH, with the word initialized to an empty string, the left and right node pointers set to the two smallest nodes which were then removed from the MH, and frequency set to the sum of the frequencies of its two children. That node was then re-entered into the MH, and continued until MH only had one node left. This had a time complexity of O(n * log(n)) and space complexity of O(1).

Huffman CodeBook

The Huffman CodeBook (HC) was built using the HT that was created out of the MH. It was written in the same directory as the fileCompressor.c file and used to perform the compression and decompression functionalities. The program iterated through the HT, assigning a value of 0 every time it went down the left branch and a value of 1 every time it went down the right. Once the program reached a word, it would print the code of that word to a file, followed by a tab, followed by the word itself, followed by a new line character. The time complexity for this is O(n * log(n)), and the space complexity is O(1).

Compressing

Files were compressed using the written HC. The program would retrieve each code-word combination from the HC and add it into an array. The program would then retrieve each word from the desired file and search for the respective word in the array, then print the code associated with it to a new file. This function had a time complexity of O(n * m) where n is the size of each file, and m is the size of the array. This function has a space complexity of O(1).

Decompressing

Files were decompressed using the written HC. The program would retrieve each code-word combination from the HC and add it into an array. The program would then iterate through the file and match each substring and search for the respective code in the array, then print the word associated with it to a new file. Once the code is matched, the buffer storing the code is reset to an empty string and continues the algorithm for the rest of the file. This

function had a time complexity of O(n * m) where n is the size of each file, and m is the size of the array. This function has a space complexity of O(1).

Recursive

The recursive flag is used to apply any of the above instructions to an entire directory, including all of its direct files as well as files in its subdirectories. This method runs with a worst case time complexity of O(n * m) where n is the number of files to be read and m is the length of the file.

Overall

The overall time and space complexity of the program is O(n) and O(1) respectively.