fileCompressor - ReadMe

Overview

This instance of fileCompressor was a project conducted by Advith Chegu and Savan Patel. As you may know, Huffman code is supposed to be a data efficient representation of storing information. In this scenario, all files in a directory(which also includes its subdirectory) has its respective data accumulated and accounted for. So elements that appear frequently would have a short bitwise representation while elements that are rare will have a larger bitwise representation to account for its presence. In this program, a user is allowed to do one of three things: build a Huffman Codebook from a file or directory, compress a given text file or every text file in a directory and its subdirectories, or decompress a txt.hcz file back into its .txt representation as well as every txt.hcz file in a given directory and its subdirectories. The latter component of each option is a recursive function call which allows for our program to traverse the contents of a given directory and apply the given flag for each file.

The Code

Our entire program relies on the use of helper methods to complete the task of the program. We will list the function prototypes and tell what each function does.

int direcTraverse (DIR *, int, int, char *) - This specific function will recursively go through a given directory and will go through all of its files and all of the files within its subdirectories. It will then store the path within a files array and go through said files array to compile the occurrences of each specific token.

int buildCodebook (int) - This function will through each and every file from the files array and tokenize them.

int fileReader (int) - This function will take in a file descriptor, read everything from said file and store it in a local buffer.

int tableInsert (char *) - This function will take in a token and find the hashcode of said token. Once found it will by mod with the hashtable size to get its index. If it does not exist, it will be added to the hashtable. If it does exist, then the token's frequency will be incremented.

int tokenizer (char *, int) - This function will take in a buffer and the size of said buffer. It will go through each character in the buffer and split them into tokens while also accounting for specific control codes.

int buildSubTrees() - This method will pop two tokens at a time and combine them to create a subtree. It will run until there is 1 item left in the heap and that 1 item will be the Huffman tree.

struct node * pop() - This function will pop the node off the top of the min heap.

int siftDown(struct node *, int) - This function will help preserve the structure of the heap by maintaining order and pushing things down the tree where they belong.
int heapTransfer() - This method will go through the hashtable node by bode and call heapInsert. It will go through every index and traverse said indexes.
int increaseHeapSize() - This function will increase the size of the heap - which is represented by an array.
int siftUp(struct node *, int) - This function will preserve the heap structure by moving things up to its proper place.
int heapInsert(struct node *) - This function will find the node you need to insert into the heap and insert it into the end of the heap - which is represented by an array.
void printHeap() - This function will print out the contents of the heap.
int printTable() - This function will print the table to check frequencies.
void freeFiles(int) - This function will release the files once they have been dealt with.
int decompressFiles(int) - This method will traverse the files array and decompress them.
void buildHuffTree(char *, int) - This method will traverse the contents of the Huffman Codebook and recreate the Huffman tree.
int codebookReader (int) - This function will take in a file descriptor and write each item to the Huffman table.
int huffInsert(char *, char *, struct node *) - This method will insert the node in the correct part of the huffman tree.
int readHcz(int, int) - This function reads in a .hcz file.
int huffmanCodebookReader (int) - This function will also take in a file descriptor and write items to the Huffman table.
int huffTokenizer (char *, int) - This function will take in the Huffman Codebook buffer and send each token and its bitcode representation into hInsert.
int hInsert (char *, char *) - This function will take a token and its bitcode representation and insert it into a hash table.
int compressionFileReader (int, int) - This function will read a given file and store it into a local buffer, calling savTokenizer afterwards.
int compressionWriter(char *, int) - This function will take in a token and a file descriptor and will write the token’s bitcode representation into the file through the file descriptor.

int savTokenizer (char *, int, int) - This function will read in the file you want to compress and separate it into the respective tokens.

int compressFiles(int) - This method will traverse all the files in the file array and compress all of them.

int freeHuffmanTable() - This method will clear memory by freeing the Huffman Table.

int depthFirstSearch(struct node *, int, char *) - This function will mainly take a path and a file descriptor to write Huffman code to.

Time Complexity

In terms of time complexity, our code would have a time complexity of O(n*logn) because of the fact that we have a Huffman tree and insertion could possibly take that long given a worst case scenario. Sadly, the Huffman tree is a necessity within our program since it represents the bulk of build, compress, and decompress. While making the Huffman tree, tokens also need to be placed into their proper location within the tree and that also contributes to the time complexity. The time complexity is O(n*logn) because insertion into a tree is O(logn) time but we have to do it for n tokens. Luckily, our program does not reach n^2 time as there are no nested loops or anything of that ilk.

Usage

To compile our code, one can simply type into the directory's terminal:

```
gcc fileCompressor.c -o fileCompressor
```

Our code can be used by inserting a singular flag (-b, -c, -d) or the recursive flag and any one of the previous three flags stated before.

Contributing

Please adhere to Rutgers policy on Academic Integrity when viewing the code and do have a good day!