FILECOMPRESSOR

NAME

fileCompressor - builds a codebook from a file(s), compresses a file(s) using a given codebook, or decompresses a file(s) using a given codebook

SYNOPSIS

```
fileCompressor -R | -b | -c | -d FILENAME CODEBOOK
```

DESCRIPTION

Build mode: Will read a given file and output a HuffmanCodebook file in the directory that fileCompressor is called from.

Compress mode: Will read a given file and encode it using a given HuffmanCodebook file.

> Encoded file is written as <FILENAME>.hcz in the same directory as given file.

Decompress mode: Will read a given <FILENAME>.hcz file and decode it using a given HuffmanCodebook file.

> Decoded file is written as <FILENAME> in the same directory as given file. Will overwrite <FILENAME> if it already exists in that directory.

Recursive mode: When called in recursive mode along with another mode, fileCompressor will run that operation on all files in a given directory and sub-directory.

- HuffmanCodebook will be written in the directory fileCompressor is called from
- Build mode takes only a FILENAME argument.
- Compress mode takes a FILENAME and a CODEBOOK argument.
- Decompress mode takes a FILENAME.hcz and a CODEBOOK argument.
- Recursive mode must be run along with another mode. It cannot be used by itself.
- Recursive build will output one HuffmanCodebook for all files in a given directory and subdirectory. This codebook can be used to encode/decode those same files.
    - Recursive compress will encode each file in a given directory and subdirectories using the given HuffmanCodebook. A <FILENAME>.hcz file will be generated alongside the original file.
    - Recursive decompress will decode each .hcz file in a given directory and subdirectories using the given HuffmanCodebook. <FILENAME> will be generated alongside the corresponding .hcz file. If <FILENAME> already exists, it will be overwritten.

OPTIONS

-R Enable recursive mode for another flag.

-b Build a codebook from a file.

-c Compress a file using a HuffmanCodebook

-d Decompress a .hcz file using a HuffmanCodebook

EXIT STATUS

Returns EXIT_FAILURE(1) on fatal exceptions.
Fatal exceptions include:

- Incorrect number of arguments
- Invalid flags
- File read error including missing file, no access, etc
- File write errors including no access, etc
- Failure to open a directory
- Out of memory errors

IMPLEMENTATION

- Build mode utilizes a greedy Huffman Algorithm to build a Huffman Tree from a binary Minheap.
    - I create a list of words to count their frequencies, and insert each node in the list into the minheap and heapify each time.
    - From the minheap, I build a Huffman Tree by taking the two minimum nodes from the minheap, linking them with an empty node that has the sum of the frequencies, put it back into the minheap, then reheap the tree to maintain the minheap structure. This repeats until there is only one node left in the minheap, which is the head of the Huffman Tree.

- Compress and Decompress mode both use a shared funtion to read a codebook and construct a list of words and codes.
    - Compress uses this list to encode a file. When it finds a word in the file, it finds that word in the list and writes the code to an encoded file.
    - Decompress uses the list to reconstruct the Huffman tree using the Huffman code of each word. Then it reads the 1s and 0s from the .hcz file walks down the Huffman tree until it reaches a leaf node, which contains the corresponding word, and writes it to a decoded file.

- Recursive mode reuses all the existing modules used by single-file mode. There is only one additional function that loops over the directories and calls the function as necessary. The recursiveMode function takes a function pointer and a flag to determine which function to run on a file.

- There are three seperate file-reading functions, one for each different mode. I wrote a MACRO that contains the initialization of shared variables and error checking, then call that macro at the top of each file-reading function.

COMPLEXITY ANALYSIS

- The greedy algorithm fileCompressor uses to build a Huffman Tree takes O(n logn) time.
    - It takes O(logn) to insert a node into the minheap, O(1) time to access the minimum nodes in the minheap, and repeats O(n) times.

- The space complexity of a Huffman Tree is O(n) where n is the number of symbols.

- The list that data is stored into takes O(n) time to search.
    - Due to time constraints, I used a list. Given more time, I would've implemented a Red/Black Tree or AVL Tree for O(logn) time, or a Hash Table for O(1) time.

- The list that data is stored into takes O(n) space in memory.
    - This list uses the same nodes as the Huffman Tree and minheap, so the nodes are reorganized into those structures instead of allocating more memory to build them seperately which saves memory.