Design And Implementation

Authors: Sam Olagun and Maanya Tandon

brew cask reinstall basictex

Organization

For organization of our code, we separated all of our helper functions into different .c files, and put them into a src folder. We have heap.c which functions as a heap class. flags.c parses through the input to the command line and calls the correct functions (compress, decompress, etc). Tokens.c tokenizes the file into the linked list and gets the frequencies. Huffman.c creates the huffman codebook using a heap. Warnings.c outputs the correct statements, and free.c contains the helper functions used to free memory. Outside of the src folder we have the main program functions in compress,c, decompress.c, and build_codebook.c.

Tokenizing the file

This project is intended to be used as a file compression system. In the terminal we are given a 2 flags (whether to build the huffman codebook recursively or not), and then a path or a file. We tokenize the file in tokens.c by using whitespaces as a delimiter. Tokens.c has 2 functions: Token_create and Token_create_frequency which assigned the word and the frequency to a token. Token_read_file reads the file in general, but token_read_file_distinct read the file and incremented the frequencies. All the tokens were stored in a linked list.

Flags

We handle the flags by having a flags struct that stores the flag type for build codebook, compress, decompress, and recursive in booleans in the struct. For the flags, we handled testcases such as the number of flags inputted, and the types of flags inputted. If more than 2 of the accepted flags are inputted, we output a warning that says more than 2 flags can not be inputted. If more than 2 flags are inputted and one of the flags does not exist, it outputs a warning that you can not have more than 2 flags, but it still runs because 2 of the flags were valid. When flags are mixed (ex. -Rb instead of -R -b), a warning is given to the user that flags can not be mixed.

Tokenizing the file has an O(n) runtime complexity because it is just traversing through a linked list.

Build Codebook

Build codebook came in three parts:

1. Counting frequency
2. Creating the Huffman Tree

3. Printing the Huffman Tree

Counting Freqeuncy

We created a function called `Token_read_file_distinct` which read a file into a linked list containing words, delimeters and their respective frequencies.

1. First, it read all of the words and delimeters into a linked list.

   Each linked list insertion is O(*1*) and we do *n* of these. **The runtime of this operation is $\mathbf{O(\boldsymbol{n})}$**.

2. Next, it removed all of the duplicates by inserting every linked list item into a new list only if it was not already containined in the list. q > Each linked list insertion for a distinct linked list is O(n) and we do n of these. **The runtime of this operation is $\mathbf{O(\boldsymbol{n2})}$**.

The total run time is: $\boldsymbol{n + n2} = \mathbf{O(\boldsymbol{n2})}$.

Creating Huffman Tree

We created the equivalent of a heap class with functions that created nodes for the heap, inserted nodes into the heap, sifted an element up to its proper place, removed the minimum, etc. The heap was used to build the huffman codebook. This was done by heapifying the tokens from the linked list, and then removing and merging the two smallest elements of the minheap and inserting it back in. Ultimately we are left with a tree which results in the codebook.

Since each node is visited once, the total run time is: $\mathbf{O(\boldsymbol{n})}$.

Printing the Huffman Tree

Compression

Compression is started by parsing through the file created by huffman codebook, and tokenizing each word and it's code into a tree node. We do this by creating a linked list called tokens from the 'tokens_read_file' function. We traverse the linked list and add each huffman code with it's associated word to a node. **Traversing through a linked list is O(n) time complexity.** We create each node using the 'compress_create_node' function in our heap class, and insert each element into the tree using the 'Tree_insert' function from binary_tree.c. Eventually, we have a tree which has each word and it's associated code. We created a function called 'compress_helper' which takes the pathname and a void pointer to a piece of data. The compress helper mallocs enough data for the .hcz file and outputs the results of the compression to the file by searching for each token in the tree. > **Searching and inserting into a binary tree is O(logn) runtime on average, with it's worstcase runtime being O(n).**

Decompression

Decompress came in three parts:

1. Reading in `HuffmanCodebook`
2. Recreating the Huffman Tree
3. Decompressing the file

Reading in `HuffmanCodebook`

To read in `HuffmanCodebook` we used our `Token_read_file` method which reads a file into a linked list.

Since every character n is visited once and linked list insertion is $\mathbf{O(\mathit{1})}$ this operation is $\mathbf{O(\mathit{n})}$.

Each of the n lines is stored only once.

Total Runtime	Total Space Complexity
$\mathbf{O(\mathit{n})}$	$\mathbf{O(\mathit{n})}$

Recreating the Huffman Tree

To recreate the Huffman tree, every one of the n lines in `HuffmanCodebook` is inserted into a tree.

Since tree insertions are $\mathbf{O(\mathit{n})}$ worst case in our Huffman tree (Huffman tree's are unbalanced), this operation is $\mathbf{O(\mathit{n2})}$.

Every of the n lines is stored once.

Total Runtime	Total Space Complexity
$\mathbf{O(\mathit{n2})}$	$\mathbf{O(\mathit{n})}$

Decompressing the file

To decompress the file, every one of the k characters in our compessed file was visited once. On each character visit, the corresponding Huffman tree made one move, either to the left or the right $\mathbf{O(\mathit{1})}$. The only space allocated in this step is each character in the compressed file.

Total Runtime	Total Space Complexity
$\mathbf{O(\mathit{k})}$	$\mathbf{O(\mathit{k})}$