Assignment 1: Code Design & Efficiency Analysis

Code Design

Organization

We separated our helper functions into files, and put them into `src/`. `heap.c` contains min heap functions. `flags.c` contains command line flag parsing functions. `tokens.c` contains functions that tokenize the file into the linked list with frequencies. `huffman.c` contains methods to create a huffman tree. `warning.c` contains helper methdos to print warnings. `free.c` contains helper functions used to free memory. Outside of `src/` we have a file for every subcommand—`compress.c`, `decompress.c`, and `build_codebook.c`.

Tokenization

In the terminal, we are given a 2 flags, the subcommand and the recursive flag, and additionally a path or a file. We tokenize the file in `tokens.c` using whitespace as a delimiter. `tokens.c` has a function called `Token_read_file` that reads files into a linked list. We also wrote a function called `Token_read_file_distinct` that reads files into a linked list and removes duplicates.

Flags

We have a `Flags` struct that stores flags and pathnames. We handled flag warnings in this file. 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, we output a warning that you can not have more than 2 flags. When flags are mixed (ex. -Rb instead of -R -b), a warning is given to the user that flags can not be mixed.

Efficiency Analysis

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.
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. Each linked list insertion for a distinct linked list is $O(n)$ and we do n of these.

The total run time is: $n + n^2 = O(n^2)$ where n is the number of words.

Time Complexity	Total Space Complexity
$O(n^2)$	$O(n)$

Creating The Huffman Tree

We created a min-heap to create the Huffamn Tree. 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 min-heap and inserting it back in. Ultimately we are left with one elmenet, a tree, which is used to build the codebook.

Since each node is visited once, the total run time is: $O(n)$ where n is the number of words.

Time Complexity	Total Space Complexity
$O(n)$	$O(n)$

Printing The Huffman Tree

Printing a tree is $O(n)$ worst case, where n is the number of nodes.

Time Complexity	Total Space Complexity
$O(n)$	$O(1)$

Compression

Compression came in three parts:

1. Reading in `HuffmanCodebook`
2. Creating a Binary Tree
3. Compressing the file

Reading In `HuffmanCodebook`

Compression is started by parsing through `HuffmanCodebook` 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.

Time Complexity	Total Space Complexity
$O(n)$	$O(n)$

Where n is the size of the linked list.

Creating A Binary Tree

Using `TreeNode_create_encoding`, we 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 in a file name and a `void*` pointer to a piece of data. The compress helper 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 $O(log(n))$ runtime on average, with it's worstcase runtime being $O(n)$.

Time Complexity	Total Space Complexity
$O(n)$	$O(n)$

Compressing The File

Compression was a matter of traversing the already existing binary tree for every of the n words and lookuping up their encodings.

Time Complexity	Total Space Complexity
$O(nlog(n))$	$O(1)$

Decompression

Decompression 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 $O(1)$ this operation is $O(n)$.

Each of the n lines is stored only once.

Time Complexity	Total Space Complexity
$O(n)$	$O(n)$

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 $O(n)$ worst case in our Huffman tree (Huffman tree's are unbalanced), this operation is $O(n^2)$.

Every of the n lines is stored once.

Time Complexity	Total Space Complexity
$O(n^2)$	$O(n)$

Time Complexity	Total Space Complexity

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 $O(1)$. The only space allocated in this step is each character in the compressed file.

Time Complexity	Total Space Complexity
$O(k)$	$O(k)$