Assignment2: File Compression

Colin Walsh
Christian Mitton

The Design

Our project can be split into the following two phases:

- (1.) Data Structures/ Huffman encoding phase
  - (a.) HashTable Creation
  - (b.) TokenizeFile
  - (c.) Huffman Encoding functions
    - Huffman Tree creation
    - Get code
  - (d.) Codebook operations
    - Codebook creation
    - Compress
    - Decompress
- (2.) File traversal phase
  - (a.) build codeBook: -b
  - (b.) compress: -c
  - (c.) decompress: -d
  - (d.) recursive: R
  - (e.) Makefile

(1.) The Data Structures/ Huffman encoding phase

(a.) HashTable Creation

A **Hash table** with linked list chaining serves as the core of the project, so it was designed to be efficient and flexible. The Hash table is made of HashNodes. A HashNode has a variety of fields, and this allows a single struct to be used through out many aspects of the program. For ex. Here are some fields in the HashNode struct:

```
char *word;   // used for inserting to hashTable
int freq;     // used for inserting to hashTable
struct HashNode *next;     // used for linked lists
struct HashNode *left;     // used for huffmanTree
struct HashNode *right;    // used for huffmanTree
etc.
```

Standard insert into the table is O(1), and if chaining was needed, all nodes would be added to the front of a linked list, which is also an O(1) operation. Each node in the hash table has the format ==> (token : frequency)

(b.) TokenizeFile

TokenizeFile takes a file as an input. The tokenizer would iterate through a file character by character using read(), and input tokens and their frequencies to a hash table. It would then return this hash-table to whoever called the tokenizeFile function. If it encounters a non-space/ non-tab character, it would add that character to a character linked list, where each node contained a single character. If it encounters a space/tab/ new line character it converts the character linked list into a single word node. For example, if this was the input:

```
"Some Text Input\n"
```

"Some" would be turned into this:

```
S —> o —> m —> e —>
```

And then each node in this linked list would be combined into one node to form a node that contains a single word:

Some —>

The same process would apply to the words “Text” and “Input”, and by the end of the tokenizer function, it would return a HashTable with the following key value pairs:

(“Some” : 1) —> NULL

(“Text” : 1) —> NULL

(“SPACE” : 2) —> NULL

(“Input” : 1) —> NULL

(“NEW-LINE” : 1) —> NULL

If the need for chaining arose, the table returned may look something like this:

(“Some” : 1) —> NULL

(“Input” : 1) —> (“Text” : 1) —> NULL

(“SPACE” : 2) —> NULL

(“NEW-LINE” : 1) —> NULL

(c.) Huffman Encoding Functions

We used a BST to create the **Huffman encoding**. One argument is given to this function: the name of a file. This file is tokenized using tokenizeFile, and tokenzeFile returns a hash table with all the tokens and their frequencies. This table is used to construct the Huffman tree. By the end of the Huffman Encoding function, A Huffman BST is created/returned.

To **get the Huffman code** of a given token, (with a provided Huffman tree) the program would traverse all root to leaf paths in the tree, and during each root to leaf traversal, the path is tracked using a linked list. While traversing, if the leaf is the target token, the path is returned as a linked list. This pathList is used to generate the Huffman code. The pathList and the Huffman tree will be compared/traversed simultaneously, and while traversing both, a code is generated.

We didn't have enough time to optimize the Huffman tree, but if we did we would use an array instead of a BST, and this would remove the need to traverse every path in order to get the code for one token.

(d.) Codebook operations

CreateCodebook accepts one argument: a file name. This file is passed into tokenizeFile, which returns a hash table with tokens and their frequencies. This hash table is then passed into huffmanEncoding. This returns a Huffman BST. Using this BST, we get the code for each token from a file, and write it to a codebook file.

For **compress** two arguments need to be given, a file name and a codebook. The file would be traversed character by character. Each character would be added to a linked list. If a space/ new-line/ tab was encountered, the linked list of characters would be converted into a single word node, similar to how tokenizeFile works. The program then searches the codebook for this word, and retrieves its corresponding code. This code is then written to a new file with the .hcz extension. Then, after writing the code for the word to the compressed file, the code for the space/ new-line/ tab is also searched for, and then their corresponding code is written to the file. (For our gulliver's travels test case it is a large text file, so compress takes an about of 40 seconds to complete)

For **decompress** two arguments which is the hcz file and the codebook. The codebook is parsed into a working huffman coding tree which is then used to calculate huffman codes and their corresponding words. The codebook is then fed through the tree in order to write the proper words to the output file. The output file is created with a _out extension.

(2.) File traversal phase

(a.) -b <path>

The **build codebook** flag specifies a target file in order to construct a huffman tree for output. When this flag is invoked, createCodeBook is called from fileCompressor in order to output to the correct location. The outputted codebook is simply titled "HuffmanCodebook"

(b.) -c <path> <codebook path>

The **compress** flag specifies a target file to compress using a pre-constructed codebook from the build path. This flag will error if a codebook is not found, or if a target file does not exist. Compress calls the compress function in huffman.h in order to output the file. The output will be titled <path>.hcz.

(c.) -d <hcz path> <codebook path>

The **decompress** flag specifies a target hcz file to decompress using a codebook. This flag will error if a codebook is not found, or if the target file either doesn't exist or is not a .hcz file. The flag calls decompress from huffman.h in order to output the corresponding output file. The output file will always be titled <path>_out.<extension>.

(d.) -R <flag> <path> <optional_codebook>

Recursion works on simple directories but not if there are directories nested in one another. The reason for this is because our memory gets corrupted (it goes out of available memory bounds).

(e.) Makefile

For the **Makefile** we have a number of directives for a variety of purposes. All compiles everything you need to run fileCompressor with it's corresponding flags (a.k.a the main program). However, for testing purposes, we have included directives to test our various data structures. Huffman outputs a huffmanCodeTest which can be used to

test a huffmanTree, and the same goes for hash table and tokenizer. These directives were crucial for debugging.