Read Me

Kevin Shah (kas665) and Manav Patel (mpp124)

Design/Implementation
The file compression was made with a combination of many data structures. The ones used were: hash table, a min heap, a qausi-tree, and a Huffman tree. All of these will be explained in further detail below.

The first stage in the program was to generate a Huffman Codebook from a file. To do this, the each unique token and its frequency had to be stored. For this, we utilized a hash table. Even though hash tables have their issues, it was perfect for this implementation. Our hash table was built to a size of five thousand with a fixed capacity and linked list capabilities. To store a token, it was hashed to the sum of the acsii values of each individual char inside the token. This allowed for each token to be stored to a relatively unique place, and allowed searching for the token at best case O(1) time. Of course, in larger files there will be some tokens that end up hashing to the same point and at that point, one would have to traverse the linked list, to search for the token. But these linked lists will be in fact very small due to the uniqueness of the keys. It turns out, that the hash table is the most time-efficient way to store and search for tokens carrying out the functions at an average time of between O(1) and O(log n).

After this, all the tokens were placed in a min heap which was aptly converted into a Huffman Tree in the standard Huffman tree generating algorithm.

After that, a an ingenious function was utilized to assign the bitcode (the binary code generated based on the the Huffman Tree) and store it in an array called book. It kept track of its depth (index) and everytime it recursed left or right, it added a '0' or '1' respectively. Then this book was printed out and that was the end of that.

For compress, it tokenized the HuffmanCodebook and storing both the bitcode and token inside another hash table. Next, it had to tokenize the file again and search for each token inside the hash table. Once it found the specific token from the hash table, it would write the bitcode into a .hcz file.

Decompress was done in a slightly different manner. For each bitcode matching the token, it created a path to the token. For example, the code "1010", would create from the root of the tree, one node in the right, which from there would create a node to the left, then another one to the right, and finishing with adding the node to the left. By doing this, the actual act of decompressing the file would make only make comparisons as how long the compressed file is. This makes it very efficient, rather than having to search bitcodes of variable lengths inside an array.

A large portion of the code is geared towards reading large quantities of bytes and tokenizing it based on spaces and control character. While tokenize functions were created to reduce the literal code itself, there were many instances where the code had to be tokenized in different manners, which effectively cluttered up the code again.

To print the control characters without using the escape “\” we first detected those, and when printing the HuffmanCodebook, we used the following translations:
A tab printed as <\t>
A space printed as <\s>
A newline printed as <\n>
A vertical tab printed as <\v>

The read() and write() functions were used to read from files and create and write into files using file descriptors. The program was made in a way so that it recursively checked each directory using DIR pointers and be able to use the functions opendir() and readdir() as well as be able to error check. Additionally, the build, compress, and decompress functions were able to read the name of the files as well as the file types using d_type and d_name in order to function properly. The recursion part of these functions was primarily to concatenating the paths with the recursive paths and so on.

Time/Space Complexity

When analyzing time and space complexity there many factors that have to be analyzed. Of course the main one is the size of file being compressed. Other factors to be considered is the ratio of distinct tokens to total tokens. For this purpose, we will refer to n as the number of tokens and m as the ratio of unique tokens : total tokens. Thus $m * n$ is the number of unique tokens.

The major time/space consuming parts of the code is storing each unique token in a data structure and keeping track of its frequency. An even greater time consuming portion of the program was to comb through each token and compress it by finding it, and then writing the bitcode.

For this purpose we used a hashtable of fixed size of 5000. To hash, as explained above, each token’s total sum of acsii is generated, and then its modded against the table size which is 5000. Another thing to consider is when more than 5000 distinct words are loaded into the tree. Of course, it accounts for this by creating a linked list. However, this overload has to be considered. When considering an overload, an even distribution is expected due the extreme randomness of the hashing. This overload factor can be represented as k.

As m approaches one (that is every token is distinct), as long as $m * n$ is less than 5000, we can expect a O(1) search and insert time.

As *m* approaches one (that is every token is distinct), and *m * n* > 5000 and as *k* increases, we can expect a O(k) search and insert time. But, *k* will never exceed *n* and so it is guaranteed that the times is less than O(n) in an impossible scenario.

In the case if we used a binary search tree the search time unreliably ranges from O(log n) to O(n). And while an AVL would have been probably the best for extremely large files at the instance when *k > log n*, it was not used because of time constraints. While hash tables are generally frowned against, this way of hashing using the acsii sum of the token and limiting the expansion capabilities of the hash table, this implementation makes it better than Binary Search Trees and slightly less capable than AVLs.

When looking at space complexity, we can expect around the same as other data structures. Even though, a big number in 5000 is seen, at the end it is a constant number and is insignificant.

Both, compress and build utilize hash tables to store the tokens in some way. However, decompress uses a different mechanism. It is probably the fastest way to decompress that is possible. Its details are discussed above.

Testplan.txt
This is how we tested all the functions. To test build, we used test.txt files of variable components to generate the HuffmanCodebook. We tested: only one token, many tokens, and no tokens in combination to make sure the build function was working properly. We also made use of the many HuffmanTree Generators online to perfect our code. We compared the HuffmanCodebook along with the a printed version of the Huffman Tree to the online trees to make sure the algorithm of the code was functioning correctly.

To test compress was very simple. We first built the codebook, which at this point was functioning close to 100%. Then we again called it on files that had one token, many tokens, and no tokens. Using the HuffmanCodebook, we traced the chain of "bits" written into the file to make sure it was actually compressing correctly. We compared the original text file to the compressed version using the HuffmanCodebook.

To test decompress was very simple. We simply built and compressed. We then renamed the original file as something else. And then we ran decompress. This decompressed file was then compared with the original file using a compare function we built for testing, but did not include in the actually implementation. This compare function went from byte to byte or in better terms from char to char and compared each to each other, before moving on. This made sure the decompress function was working to its full capabilities. We also checked to compress compressed files, and then also decompress decompressed files.

To test the recursive calls on build, compress, decompress, we went through a similar approach as the one above. However, we created directories with many subdirectories, with many more directories. Inside each of these directories we had normal files (within the ascii restraints) and ran build, compress, and decompress. Then we used testing methods described above to verify if it ran correctly or not. The amount of directories and files inside were held variable and each time, our program came out on top.

We also tested to build, compress, and decompress as symbolic file/link.

We also had many error checking mechanisms which include but are not limited to: NULL files, incorrect parameters, incorrect format of parameters, empty files. By which we induced the error and checked for the various error print statements we have in effect. These were also successful.