Vivian Trieu
Kyle Frick

Codebook:

Codebook has many moving parts to it. Firstly, an AVL tree is used to store the tokens of a file. This maximizes efficiency when storing the tokens. The AVL tree organizes the tokens in alphabetical order, and if it is a control character, it will store it in accordance to ASCII value at that point. When a duplicate is stumbled upon, the frequency is increased. While this is occuring, a linked list of the tokens is being built so that when the min heap is made, we traverse through the linked list and find the frequency quickly in the AVL tree. The min heap is used to store the tokens so that huffman coding can be utilized. As a min heap priority queue, it follows huffman coding the least frequent tokens are always at the top of the heap. This makes it easy to dequeue the smallest node. The T queue, aka tree queue, will hold small trees that has been made when combining individual nodes. Then we would dequeue the smallest and combine until we are left with one tree in the T queue. Then, after, we would traverse the huffman tree so that we can write into the Huffman codebook.

The time complexity of codebook would be nlogn because it has to go through each token, which is n, and inserting into the AVL tree is logn. In addition, the min heap would be nlogn because it needs to insert into the min heap n times, and insertion into a min heap is logn. This maximizes the time complexity of huffman coding to be nlogn as well because of the use of the priority queue.

Compress & Decompress:

- Compress & Decompress uses the following to run.

```
- /* Compression and Decompression */
- //typedefs
- typedef struct node
- {
-    char *code;
-    char *rep;
-    struct node *next;
- } node;
-
- //methods
- static node *createNode(char *code, char *rep);
- void freeLL(node *root);
- static node *insertEndLL(node *head, char *code, char *rep);
- void printLL(node *head);
- char *findCode(node *root, char *token);
- char *compareCode(char *code_in, node *root);
```

- It utilizes a LinkedList as its storage data structure since when reading in and tokenizing the HuffmanCodeBook, it is already in DFS order (0 → 1 order), which is exactly how the compareCode() method determines if the code runs.
- findCode() goes through the LinkedList in a linear fashion to search for the given token and then writes that token's code to the file after returning. This is done over and over until all tokens within the read file have been converted to their binary code.

Issues:

- The methods that do work are all the methods other than the creation of the HuffmanCodeBook. All the methods above the main method within the fileCompressor.c work except for writeHuffmanCode().
- To run compress and decompress, the compress.c file is the one that is needed. The ./a.out HuffmanCodeBook <file> <file.hcz>
- Also since the compress and decompress are using a LinkedList instead of a tree to work, the time that it would need to go through the linear search would be O(n)(N) where n is the number of unique tokens while N is the number of tokens within the file read in.