void checkFile(char *basePath, int flag1, int flag2)

- checkFile takes in the pathname of a file and two integers that indicate if there are two flags or one, and if they are flags what they represent.
    - 1 == build
    - 2 == recurse
    - 3 == compress
    - 4 == decompress
    - 0 == does not represent flag
- checkFile first checks if it is a file
    - If Yes -> we check the flags to indicate what to do for file
        - If we pass in 0 0 we write the contents of file to where all the other files are written to
        - If one of the flags is 3, we compress
        - If one of the flags is 4, we decompress
    - If No the function just returns to the recursive function

void recursive(char *basePath, int checkFlag1, int checkFlag2)

- recursive takes in the pathname of a file and two integers that indicate if there are two flags or one, and if they are flags what they represent.
- Checks if it is a directory
    - If it is not directory we pass basePath into checkFile
    - If it is a directory we recurse through the directory

Struct node* newNode(struct node* head, char* var)

- newNode takes in the pointer of a node and a char variable
- Create nodes and Insert into List

void frequency(struct node *head)

- Removes duplicate nodes
- Everytime it removes a duplicate node it increments the frequency of the original node

int delimeters(char token)

- Takes in a character
- checks if delimeter is a space, tab, or newline and returns int corresponding to each
    - 1 == Space
    - 2 == Newline
    - 3 == Tab

Int checkFlag(char *flag)

- Takes in a string
- Compares it to each flag
- The string is:
    - 1 == build
    - 2 == recursive
    - 3 == compress
    - 4 == decompress
    - 0 == not representing a flag

struct node* buildList(int flagb)

- Builds linked list with all the tokens from all files in directory
- It reads in tokens one by one from writeFile, which is a file that contains all tokens from all the files

void insert(struct node* head, struct node** array, int capacity)

- Inserts nodes into a minHeap array
- void swap(struct node* a, struct node* b)
    - Swaps two nodes in minHeap when necessary

void huffmanTreeBuilder(struct node** array, int capacity)

- Builds the huffman codebook
- struct node* huffmanHelper(struct node** array, int capacity)
    - Helper function for building the tree, takes two nodes from the minHeap
    - struct node* heapMin(struct node** array, int capacity)
        - Takes the minimum value in the minHeap
        - void heapify(struct node** array, int index, int capacity)
            - Recursive function that rebalances the minHeap
            - void swap(struct node* a, struct node* b)
    - struct node* mallocNewNode(int frequency)
        - Creates a new node for the minHeap by mallocing and new node and placing the new frequency into the node.
    - void insertNewNode(struct node** array, struct node*hi, int capacity)
        - Inserts the newly made node into the minHeap
- void printCodeArray(struct node* head, int* codes, int hi)
    - Creates the HuffmanCodebook text file and inserts the byte values of the tokens by recursively traversing the Huffman tree

void compress(char* file)

- Compresses the given file by placing the bytes and their tokens into a linked list from the HuffmanCodebook text file and writing the found bitstrings into a txt.hcz file.
- int compressCompare(int sz, int fd3, struct press* ptr, char* string, int check)
    - Compares each token in the file with the tokens in the linked list and returns 1 if a token is found.

void decompress(char* fileH)

- Decompresses the given file by placing the bytes and their tokens into a linked list from the HuffmanCodebook text file and writing the found tokens into a .txt file.
    - int compressCompare(int sz, int fd3, struct press* ptr, char* string, int check)
        - Compares each bitstring in the file with the bitstrings in the linked list and returns 1 if the bitstring is found.

Total heap usage:

- 196 allocs, 415126 bytes allocated

Runtime:

build:

O(b^d + n + nlogn)

- b represents the branches of a directory

- d represents the depth of the directory

Compress & Decompress:

O(n^t)

Design and Implementation

Build:

Our design for build involves:

- Recursing through directory given and checking if it is a file
    - If it is a file we write its content to a writeFile
- Next the program writes everything to write file we build a linked list with all the tokens
- Then we remove the duplicates and increase frequencies of the original
- After, we make the min heap and create the Huffman Tree, which generates the HuffmanCodebook.txt

Compress:

If the flags do not indicate recursion we pass the argument straight to compress.

- The file is directly passed into the function and it outputs a txt.hcz file that is compressed.

If flags do indicate recursion we pass it to the function as we check and recurse through each file

- The file is directly passed into function and it outputs a txt.hcz file that is compressed as we recurse through the files.

Decompress:

If the flags do not indicate recursion we pass the argument straight to decompress.

- The file is directly passed into the function and it outputs a txt.hcz file that is decompressed.

If flags do indicate recursion we pass it to the function as we check and recurse through each file

- The file is directly passed into function and it outputs a txt.hcz file that is decompressed as we recurse through the files.