Asst1 Readme

Example Usage

Supported Format

```
# Building a Codebook
./fileCompressor -b /path/to/file
./fileCompressor -R -b /path/to/dir

# Compressing Files
./fileCompressor -c /path/to/file ./HuffmanCodebook
./fileCompressor -R -c /path/to/dir ./HuffmanCodebook

# Decompressing Files
./fileCompressor -d /path/to/file ./HuffmanCodebook
./fileCompressor -R -d /path/to/dir ./HuffmanCodebook
```

- Note: If you try to give junk paths, junk flag combinations, files that don't exist, dirs that don't exist, etc. then it will print the error and exit

Example:

```
# Build Executable
$ make

# Let us assume there exists a directory files/, containing 2 files inside of it, file1 a

.
..
fileCompressor
main.c
files/
  - file1
  - file2


# Let's go ahead and Build our HuffmanCodebook

$ ./fileCompressor -R -b files/

  -> [ Starting to Build Codebook]
  -> [ Successfully created Codebook ]

# Now that we have our HuffmanCodebook, we can compress files

$ ./fileCompressor -R -c files/ ./HuffmanCodebook

  -> [ Started Compression ]
```

```
  -> Compressing files//file1
  -> Compressing files//file2
  -> [ Successfully Finished Compression ]

# Our dir structure should look like the following now

.
..
fileCompressor
main.c
files/
  - file1
  - file1.hcz
  - file2
  - file2.hcz

# Let's go ahead and delete file1 and file2, and decompress to get them back

$ ./fileCompressor -R -d files/ ./HuffmanCodebook

# Tada! The files are back just as before
```

Implementation

The way we implemented our `fileCompressor` was by taking advantage of `MinHeaps` and `BinaryTrees` in order to construct a Huffman Coding Tree. After that, we traversed down the tree tracking the paths we have taken to generate the compression codes, making sure to write them out to `./HuffmanCodebook` if the `-b` flag is passed in. This gives us a list of bit strings of all strings contained over the file/files, and we can create an encoded file based off of `./HuffmanCodebook`

If the user passed in `-c`, we have to compress the file or files in the directory they want according to the `./HuffmanCodebook`. We do this by reading over the contents of the `./HuffmanCodebook` and building a dynamic `LinkedList` datastructure storing `Key/Value` pairs. This allows us to search for the key or value and get the other returned. Then, we read over all files and build up a temporary buffer until we reach a string that is a valid `Value` property in our `LinkedList`, and write the corresponding `Key` property to the corresponding `.hcz` compressed file.

If the user passed in `-d`, we almost do exactly the same as we did for `-c`, but we keep loading the buffer with the contents of the corresponding `.hcz` compressed file(s) until we hit a sequence of bits that are a valid `Key` property in our `LinkedList`.

The user can also pass in `-R` in combination with any of the other flags, which will have the program recursively do the command for all files in the directory the user passes in as well.

Runtime

The heaviest and most expensive operation that we do throughout our code is the following for each flag:

- Compression `-c`:
    - Compression requires us to build up the `CodebookNode* LinkedList`, which is O(n) where `m` is the number of lines in the `./HuffmanCodebook`. Also, whenever we build up the buffer of the uncompressed file, we have to search the `LinkedList` to find the bitstring that will replace it. So, worst case the file contains `m` unique 1 character long strings, leading to `O(m * m)` complexity
- Decompression `-d`:
    - Decompression requires us to also build up the `CodebookNode* LinkedList`, which is O(n) where `m` is the number of lines in the `./HuffmanCodebook`. Also, whenever we read 1 byte of the compressed file, we have to check if the current sequence corresponds to a valid bitstring sequence. So, worst case the compressed file contains `n` digits, each digit resulting in a full search of a `LinkedList`, so the complexity is `O(m * n)`
- Building `-b`:
    - Building the codebook requires reading `m` total characters across the file(s) it has to compress to build up all of the `HeapNode` elements for the `MinHeap`. Everytime we pick 2 off the top of the `MinHeap` requires `2 * O(logn)`, which is less than `O(m)`, so we can ignore it. But, we also have to traverse the complete tree using `PostOrder` to write out the bitstrings, which is `O(n)` where n is the # of characters. Therefore building the `./HuffmanCodebook` is `O(m * n)`