Design and implementation of FileCompressor

To build a huffman codebook given a file:
Command line:

```
./fileCompressor -b <filename>
```

Design and implementation:

When the command line is not formatted in the expected way, the program will print the type of error and swiftly return. In correct usage, in order to build a huffman code book for the file flag, the execution of this task requires the initiation of 5 important functions (not including freeing memory or helper functions). The first main function used is:

- buildCodeBookFile(char * fileName)

The function opens the file, and upon success (failure to open results in a return of the program) of opening, the file will be read in 100 bytes at a time, stored into "bufferNodes". The bufferNodes are structures which have a stack allocated char array of 100, when the array is filled with characters from a file the bufferNodes create a linked list of content read in from the file. Once the read is completed, the function returns. Next:

- createHashTable(char flag)

The function first allocates a hashtable of size 10 * the number of bufferNodes created (as to create a larger hashtable for larger inputs). It takes a flag which indicates if it should use the directory global linked list or the file linked list and proceeds accordingly. The function's job is to parse the file contents that was read into the linked list. The file is parsed based on tokens, and the individual tokens are stored as hashnodes into the hashtable ,once parsing is complete and the bufferNodes are feed, it returns: Next:

- createMinHeap()

This function removes nodes from the hashtable and inserts them into the minHeap, once complete it frees the hashtable and returns. Next:

- storeHuffmanCode()

This function removes two nodes at a time away from the minHeap and conjoins them in a loop until only 1 node is left, this is when the final huffman tree is created. The minheap is freed then the function invokes the next method:

- printTree(struct hashnode * root, int * encoding, int index)

The printTree function descends down the root of the huffman tree and once it reaches leaf nodes it writes the correctly formatted data to the HuffmanCodeBook. Once the whole tree is correctly written, the nodes are freed and the function returns and the program is complete.

To build a huffman codebook given a directory:
Command line:

```
./fileCompressor -R -b <directory name>
```

The only part that differs between the file type build and directory build is the command line and one function. Since the program will have to create a huffman code book for all the files in the directory (and subdirectories), the following function is invoked:

- listFiles(char * startPath)

The purpose of this function is similar to the "buildCodeBookFile" function but instead of just parsing the singular file given, the listFiles function descends down, reading in 100 bytes at a time and stores the file contents into a chain of bufferNodes. Once the recursive function returns (implying the parsing of all found files is complete), the same functions are invoked as in the file case. createHashTable to build the hash table, createMinHeap to create the min heap, storeHuffmanCode to build the Huffman tree, and finally printTree to insert the formatted pieces into the HuffmanCodeBook file.

Time complexity:

For a file and a directory, the build method has the following complexity. Define first the input "n" as in the number of bytes needed to be processed.

Given n, n/100 is the amount of bufferNodes that will be created. The size of the hashtable is then defined as 10* (the number of bufferNodes) so 10*(n/100). Then once this operation is done. The content is parsed, 1 by 1. The parsing stage is O(n). The insertion into a hashtable is O(1), this done n times is O(n). Once the hash table is filled, the minHeap is constructed by inserting the nodes from the hashtable into the minHeap. The removal of nodes from the hashtable and inserting into the minheap takes O(nlogn) *since insert into min heap is logn and for n inserts, nlogn). Then removal from the min heap to construct the huffman tree is also O(nlogn) by the same logic as insertion. The last step is traversing the tree for the write operations. At worst the huffman tree is skewed, where the traversal would take n/2 = O(n) time. In the end all the pieces result in: O(n) + O(n) + O(nlogn) + O(nlogn) + O(n) = O(nlogn).

To compress given a file:
Command line:

```
./fileCompressor -c <filename> HuffmanCodeBook
```

Design and Implementation:

When the command line is not formatted in the expected manner, an error is printed and the program returns. Upon correct usage, the program will create a compressed hcz version of the original file filled with solely 1's and 0's.

- compressFile(char * fileName, char * huffmanCodeBook)

This function is used in many places, one of which is in the compression stage of the program. Given the huffmanCodeBook the program parses the code book content into bufferNodes 100 bytes at a time and stores it into a global linked list for later processing.

- cParseForHash(char * flagger)

This function is responsible for parsing the content of the huffman code book from the global linked list, depending on if it is for compression or decompression the flag is used to be able to decide to hash for compression (index by literal word not encoded word). Once the content is parsed and stored into "decompNodes" the function returns

- cMakeFile(char * fileName)

The function is responsible for reading in the content of the target file into bufferNodes, once this is complete the new hcz file is opened/created and as the parsing of the original file from the bufferNode commences, the writing to the new file happens side by side. As soon as a token or control code is detected the hash table is searched for the content and returns the encoded replacement. Once the bufferNodes are completely parsed, the function returns and the program is complete.

To compress given a directory:
Command line:

```
./fileCompressor -R -c <directory name> HuffmanCodeBook
```

The only difference between the file and directory version of compression is in the last step. The parsing of the hash table is done the exact same way, the process of compression for the file is instead done recursively and every time a new file is come across the proper function for the singular file is called upon. CompressFile and cParseForHash are invoked the same way. The only difference is in the usage of cMakeFile:

- dirCompressFiles(char * startPath, char flag)

This function recursively descends the starting path, upon finding a non-hcz file the function will call cMakeFile (description from before) and create the compressed version of every file found in the directory and all subdirectories found. Once complete the function returns and the program is finished.

Time complexity:

I will define the size (in bytes) of the Huffmancode book with “n” and the number of files in subdirectories as “m” (in case of recursive descent)

At first the entirety of the huffman code book is read in 100 bytes at a time into bufferNodes, which then allocated n/100 * 10 size space for the holding hash table. The insertion into the hash table takes O(n) time for the worst case (if every byte in the Huffman code book is unique). Once the insertion is complete, the hashtable is then used to search for specific tokens. Because the hash function and creation of the hash table is optimized, the searches require about O(1) time, for n searches this becomes O(n) in complexity. The content is written to the file about n times so again O(n) in the worst case. At worst, the overall complexity of compression for a singular file is: O(n) + O(n) + O(n) = O(n). Then for the recursive flag with m files, O(m) x O(n) = O(mn) time.

To decompress given a file:
Command line:

```
./fileCompressor -d <filename> HuffmanCodeBook
```

The simple thing about decompression once compression is explained is the functions used in compression and decompression are the same, only the flags are set differently. compressFile and cParseForHash are both used in the exact same manner, the only difference is the last function:

- reconstructFiles(char * fileName)

This function is similar to cMakeFile as in the way it searches for the corresponding word, given the encoded version and writes the found content to the file. If it comes across control code, it will replace it with the corresponding white space instead. Once all the content is stored the function closes the opened files and the program is complete.

To decompress given a directory
Command line:

```
./fileCompressor -R -d <directory name> HuffmanCodeBook
```

As in the compression instructions, the only difference between the file and recursive methods for decompression is that when given a directory, the program uses dirCompressFiles (as described before) to recursively find hcz files to decompress, the files found are passed to the reconstructFiles function for individual reconstruction.

Time Complexity:

Again, I will define the size (in bytes) of the Huffmancode book with “n” and the number of files in subdirectories as “m” (in case of recursive descent)

At first the entirety of the huffman code book is read in 100 bytes at a time into bufferNodes, which then allocated n/100 * 10 size space for the holding hash table. The insertion into the hash table takes O(n) time for the worst case (if every byte in the Huffman code book is unique). Once the insertion is complete, the hashtable is then used to search for specific tokens. Because the hash function and creation of the hash table is optimized, the searches require about O(1) time, for n searches this becomes O(n) in complexity. The content is written to the file about n times so again O(n) in the worst case. At worst, the overall complexity of compression for a singular file is: O(n) + O(n) + O(n) = O(n). Then for the recursive flag with m files, O(m) x O(n) = O(mn) time.