Amy Wang, Qingyang Wang

Assignment 1 - README

Overall:

The *fileCompressor.c* application takes in a sort flag for build, compression, decompression, and recursion, a file or directory/path, and a given Huffman Codebook if necessary. If build is called a min heap and BST is built and used to find the most efficient bit sequence for the corresponding token and a Huffman Codebook is created. Spaces, tabs, and new lines are used as delimiters and also count as tokens on their own. If compression is called, the given codebook is read, stored as a linked list, and the tokens from the original file are paired with corresponding bit sequences and the compressed version of the file is created. If decompression is called, the given codebook is read and stored as a BST, the bit sequences from the original file are paired with its corresponding tokens, and the decompressed version of the file is created. Recursion for build uses all the files in a given directory/path to create a single Huffman codebook. Recursion for compression compresses all the files in a given directory/path, and recursion for decompression decompresses all the files. Errors throughout the project are printed accordingly.

Analysis of Time, Space, and Complexity

- **Build:** A min heap and a queue is used in this part. For the min heap, the worst case time complexity for search, insertion, and deletion is O(logn), and the space complexity is O(n). For the queue, the worst case time complexity for search is O(n), for insertion and deletion is O(1), and the space complexity is O(n).
- **Compress:** A linked list is used in this part. The worst case time complexity for search and deletion is O(n), for insertion is O(1). The space complexity is O(n).
- **Decompress:** A binary search tree is used in this part. The worst case time complexity for search, insertion, and deletion is O(n) when the tree is extremely skewed to one side.

The average time complexity for the binary search tree for search, insertion, and deletion is O(logn). The space complexity is O(n).

Makefile

- **all:** compile the code and name the executable fileCompressor
- **clean:** remove the executable and the created Huffmancodebook

Implementation of Functions:

1. **main: int main(int argc, char* argv[])**
    - The main function takes an array of parameters and checks parameter's order, flag, and type. If the user gives correct parameters, according to the flag(s), the corresponding function(s) will be called to build a codebook, compress, or decompress, either recursively or not.
2. **readCodebook: int readCodebook (const char* codebookName)**
    - The readCodebook function takes in a given Huffman Codebook and creates a linked list that stores each token given and its respective bit sequence. If the Codebook contains an escape character, it is stored as the head of the linked list. Then, it proceeds to read the rest of the given codebook, reading bit by bit until it reads either a '\t' or '\n'. A node is malloced when a tab is read. If there exists an escape character, the bits read before tab is stored as the value and bits read before newline is stored as the token. If there is no escape character, the code keeps count and acts accordingly. This function also checks if the given bit sequence only contains 0's and 1's. If not, an error message is thrown, the corresponding token is ignored, and the code continues to read the rest of the file.
        - Returns 1: if there is an escape character
        - Returns 0: if there is no escape character
3. **writecp: void writecp(const char* cpRename, int fd2, cbNode* comptemp)**
    - The writecp function takes the corresponding bit sequence of a token and writes it into the compressed file.

4. **freecbNode: void freecbNode(cbNode* head)**
    - The freeNode function takes in the head of the linked list created by the readCodebook function and frees the malloced nodes.
5. **compress: int compress(const char* compressFile, int escape)**
    - The compress function takes in the file to be compressed and an integer that tells it whether or not there is an escape character. It first creates a file that adds .hcz to the end of the file to be compressed. It then reads the original file bit by bit until it reaches a delimiter of space, tab, or newline. The bits read before are compared to the tokens stored in the linked list formed in readCodebook. If found, the writecp function is called. the corresponding bit sequence stored in the same linked list is written into the .hcz file. The code then repeats the same process with the delimiter token and continues until it reaches the end of the file. If a corresponding bit sequence cannot be found, an error is printed.
6. **rebuildTree: int rebuildTree (const char* codebookName)**
    - The rebuildTree function takes in the given Huffman Codebook and then builds a binary search tree based on the bit sequences of the tokens. If the codebook contains an escape character, the character will be stored separately in an array. This function reads the codebook bit by bit, going left if a 0 is read and right if a 1 is read. If a bit is read, the end of the bit sequence is not reached, and the node is null after going left or right, a placeholder node is created to act as a place holder. If the end of the bit sequence is reached, a new node is created and the corresponding token is stored. The process is repeated until every token and their bit sequence has been combined to make a tree. This function also checks if the given bit sequence only contains 0's and 1's. If not, an error message is thrown, the corresponding token is ignored, and the code continues to read the rest of the file.
        - Returns 1: if there is an escape character
        - Returns 0: if there is no escape character
7. **writedcp: void writedcp(const char* dcpRename, int fd2, Huffnode* htemp)**

- The writedcp function takes the token of a corresponding bit sequence and writes it into the decompressed file.

8. **freedcptree: void freedcptree(Huffnode* node)**
   - The freedcptree function takes the head of the tree created by the rebuildTree function and uses recursion to free all the malloced nodes.
9. **decompress:**
   int decompress(const char* decompressFile, Huffnode* hhead, char* escapechar, int escchar)
   - The decompress function takes in a file to be decompressed, the head of the BST made in the rebuildTree function, an array that stores the escape character, and an int that states whether or not there is an escape character. It first takes the file and creates another file removing the .hcz at the end. The function then reads the file to be decompressed bit by bit. From the head of the tree, a pointer goes left if a 0 is read and right if a 1 is read. If there is no corresponding token after reading a sequence or if the bit sequence to be decompressed contains anything besides 0's or 1's, an error is printed. If a token is found, the pointer goes back to the head and repeats until it reaches the end of the file. If an escape character exists, tokens of the same length as escape character plus one are compared to the escape character stored in the array and if proper format, the corresponding delimiter is printed.
10. **searchHeap: int searchHeap(hNode* h, char* target)**
    - The searchHeap function takes in the head of the current min heap and a token as a target. If the target is found, the function returns the index of the target token and if the target is not found, -1 is returned.
11. **swap: void swap(hNode* heap, int a, int b)**
    - The swap function takes in the root of the min heap and the index of two hNodes which need to be swapped. It swaps all the fields of the two hNodes.
12. **updateHeap: int updateHeap(heap* h)**

- The updateHeap function takes in a pointer to the min heap structure, and checks if any node's left or right child has a smaller value than the node itself. If found one, then it will swap the node with larger value with the node with smaller value.

13. **inserthNode: int inserthNode(heap* h, hNode* newNode, int isHuffTree)**
    - The inserthNode function takes in a pointer to the min heap structure, a pointer to a new hNode that is going to be inserted into the current min heap, and a integer which checks if the new node is obtained from tokenizing a given file or from building Huffman tree (0 means is from tokenizing a given file and 1 means is from building the Huffman tree). It will insert the hNode into the proper position of the min heap according to its value.
14. **traverseMinHeap: int traverseMinHeap(heap* h)**
    - The traverseMinHeap function takes in a pointer which points to the root of a min heap. It will traverse through the min heap and print out the token and frequency associated with each hNode. This function is just for testing the code, and it is not used in the actual code.
15. **removeRoot: hNode* removeRoot(heap* h)**
    - The removeRoot function takes in a pointer to a min heap. It removes the root of that heap and adjusts the position of other hNodes in the min heap. It will return the pointer to the root hNode which is associated with a token that has the smallest frequency.
16. **createHuffnode: Huffnode* createHuffnode(int num, char* token)**
    - The createHuffnode function takes in an integer which is the frequency of a token and the actual token. It will create a hNode and return the pointer to that hNode.
17. **searchHuffQueue: Huffnode* searchHuffQueue(HuffQueueNode* hq, char* target)**
    - The searchHuffQues function takes in a pointer to the start of a queue which stores Huffman trees. It will search and return the Huffnode pointer whose token is the same as the target or return NULL if not found.
18. **freeHuffQueue: int freeHuffQueue(HuffQueueNode* head)**

- The freeHuffQueue function takes in a pointer to the start of the Huffman queue and free each HuffQueueNode in that queue.

19. **buildHuffmanTree: Huffnode* buildHuffmanTree(heap* h)**
    - The buildHuffmanTree function takes in a pointer to the root of the min heap. Based on this min heap and Huffman's algorithm, it will build a Huffman tree and return the root of the Huffman tree,
20. **HuffmanHeight: int HuffmanHeight(Huffnode* node)**
    - The HuffmanHeight function takes in a pointer to the root of the min heap and returns the height of that min heap.
21. **traverseHuffman: int traverseHuffman(Huffnode* top)**
    - The traverseHuffman function takes in a pointer to the root of a Huffman tree. It will traverse the tree and print out the token and value associated with each Huffnode. This function is just for testing the code, and it is not used in the actual code.
22. **encodeHuffman:**
    int encodeHuffman(Huffnode* node, char* arr, int idx, int fd, int isFirst, char* escaping)
    - The encodeHuffman function takes in a huffnode(which is the root of the Huffman tree), an array to store a token's Huffman code, an integer which make sure the number 0 or 1 gets put into the correct position of the Huffman code array, an integer that is the file descriptor, and the escaping character(s) which is being used in the Huffman codebook. The function will generate the Huffman code for each token and write the escaping character(s), Huffman code, and token into a file called HuffmanCodebook.
23. **tokenizeFile: int tokenizeFile (char* fileName, heap* h)**
    - The tokenizeFile function takes in a file's name and a min heap pointer which points to the root of the min heap. The method will separate all distinct words and delimiters in the given file and build a min heap based on frequencies of each token.

24. buildCodebook: int buildCodebook(heap* h)

- The buildCodebook function takes in a pointer to the root of the min heap. It determines the escaping character(s), then calls buildHuffmanTree and encodeHuffman to generate the Huffman codebook.

25. recursionAllFiles: int recursionAllFiles(char* dirName, char flag, heap* h, int escapeC)

- The recursionAllFiles function takes in the name of a directory, a flag that was passed in to main, a pointer to the min heap, and an integer indicating whether the first line of HuffmanCodebook is the escaping character(s)(1 indicates yes and 0 indicates 0). It will recursively descend into all subdirectories and build a codebook, or compress, or decompress all files in that directory and subdirectories according to the flag.