Jonathan Morlen

ECE 368 – Purdue University

10/10/2019

Project 2 Milestone 1

huff.c:

1. *char* readFile(FILE* fp):* File reader to parse data line by line from the given file to the program.
2. *void getFrequency(char* file, int* asciiFrequency):* function to determine the frequency of characters in the file. Edits the appropriate index of the 256-long asciiFrequency array to update the amount of times that character appears in the file.
3. *struct Node { }:* Binary tree node struct to hold necessary information. Functions as a doubly-linked list.
4. *Node** frequencySort(int* asciiFrequency):* A sorting algorithm to sort the frequencies in ascending order. Has a local array of all the ASCII characters. When sorting, in addition to the swapping of the int's, it also swaps the corresponding characters from the ASCII character array, sorting them. May be able to optimize a bit by removing unused ascii characters and shortening the arrays. Once sorted, creates the Node** linked list by traversing the frequency and character arrays simultaneously, starting at the first nonzero character. This will create a linked list of Node*'s with frequencies' in ascending order.
5. *void createTree(Node** frequencyNodes):* Tree forming function that takes the frequency-sorted Node** list and creates a tree from them. First the pseudo-EOF character is added to the tree with a frequency of one and a value of 256. Then the function traverses the list from lowest to highest, grabbing the two nodes with the smallest frequencies and creating a tree from them as described. When getting these nodes, it removes them from the list. After pairing them and treeing them, it then adds the root node of the tree back into the list in the first sorted position it can find, again traversing low to high frequency-wise. This is done until there is only one node in the list, it being the root node of the Huffman encoding tree.
6. *struct Dictionary { }:* A dictionary-like struct that holds each of the characters as the key with their binary representation as the value. This will be the table of bit encodings as described in the project document. Probably composed of two arrays with the characters in one and the frequencies in the other.
7. *Dictionary: getBitEncdoing(Node* encodingTree):* A function that traverses the tree through all possible paths creating the bit encoding per character. Probably will be recursive that counts the number of times the function moves down the tree, creates a char array of that size and fills it with the 1's and 0's used to get to that character. Using the dictionary-like struct described in number 7, it will add these character encoding pairs to the data structure, returning it.
8. *char* createCompressed(char* originalFile, Dictionary bitEncodings):* A function that takes in all of the bit encodings created in the function in number 7 and uses it to create the compressed file. This function will read from the originalFile variable and then translate from the character to the bit encoding described in the dictionary passed in, adding it to the compressed file char*. Once done a pseudo-EOF bit string will be added in from the tree, ending the file.

9. *char* createHeader(Node* encodingTree):* This function will create a pre-order traversal for the unhuffing function to use to rebuild the tree. A pseudo-EOF character will be added at the end to differentiate between header and the compressed bits.
10. *void saveCompressed(char* compressed, char* filename):* Now a new file will be created and the compressed bit encodings will be saved to it. It will be named appropriately.

unhuff.c:

1. *Node* getHeader(FILE* fp, int* position):* This function will read until the first pseudo-EOF character is found and will rebuild the binary tree based off of the pre-order traversal it finds. It will return this rebuilt tree as well as modifying the position variable to contain the position of the first bit in the actual data to be decompressed in order for the second pseudo-EOF to function as intended.
2. *void decompress(Node* encodingTree, FILE* fp, int postion):* This function will fseek() to the position passed in and start reading bits, following the tree and writing characters to the new, unhuff'd file.

Addendum:

In number 3, It will be easier to sort a fixed size (255) array than a linked list of Node*'s so the sorting of the frequencies will occur before the asciiFrequency conversion to Node**.

In the Node struct, the character will be held in an int instead of a char to allow the possibility of a pseudo-EOF character to be added at the end.