File Compressor

Mignot Mesele

Purpose

The purpose of the program is to compress the data in a file to reduce the amount of storage needed to download files. The program uses the huffman code algorithm, which uses priority queues to organize frequencies of characters and compress them into parent nodes, shortening the amount of data to write back.

How to Use the Program

To use the program you'd simply type in the command argument, ./huff along with a command option and certain parameters. This would specify whether to read a file, write a file, or print the help message. Options:

- -i: reads from input file. Requires a filename as an argument.
- -o: writes to output file. Requires a filename as an argument.
- -h: displays help message

You need to use the Makefile, which enables you to run the commands, make filename and make clean. The make command converts the c files into executable files to be able to use huff.c, pqtest.c, bwtest.c, and nodetest.c. The make clean command removes all files created by make, clearing any issues that the object and executable files could cause.

Program Design

Data Structures

- arrays-This is used as the buffer to store data.
- heap-This is being used to access data in other files.
- BitWriter-This made to create the output buffer and write bits of data.
- Node-This is made to keep track of the contents in nodes, made in the priority queues for the program.
- Priority Queue-The Priority Queue structure is created to store all List Elements, which contains all the nodes of the huffman tree in order, starting with the head containing the node that has the least frequent character in the file.
- List Element- The List Element structure is created to store each element in the priority queue with a node and a pointer to the next list element in the queue.
- Code-The code structure is created to store the code and code length of each leaf node into the code table.

Algorithms

```
huffman code algoritm
    open the input file
    create the histogram and fill it
    close the input file
    create the huffman tree
    create the code table and fill it
    reopen the input file
    open the output file
    compress the input file and write it to the ouput file
    close the output file and the input file
    free the histogram, code table, and the code tree
```

```
huffman compress file algorithm
    write H into buffer
    write c into buffer
    write the filesize into buffer
    write num_leaves into buffer
    write the code tree
    for every byte b from input buffer
        code = code_table[b].code
        code_length = code_table[b].code_length
        for 0 to code_length-1
            write the rightmost bit of code
            prepare the next bit to be written
```

Function Descriptions

- BitWriter *bit_write_open(const char *filename)-This function takes an output filename and opens it, returning a pointer to the output buffer that is then written to later in the program.
- void bit_write_close(BitWriter **pbuf)-This function takes a double pointer, closes the file, and frees memory allocated for the buffer.
- Buffer *bit_write_bit(BitWriter *buf, uint8_t x)-This function takes a buffer and a byte, writing each bit and clearing the byte for next time.
- void bit_write_uint8(BitWriter *buf, uint8_t x)-This function takes a buffer and a byte, calling bit_write_bit() 8 times to write a byte.
- void bit_write_uint16(BitWriter *buf, uint16_t x)-This function takes a buffer and 2 bytes, calling bit_write_bit() 16 times to write the 2 bytes.
- void bit_write_uint32(BitWriter *buf, uint32_t x)-This function takes a buffer and 4 bytes, calling bit_write_bit() 32 times to write the 4 bytes.
- Node *node_create(uint8_t symbol, double weight)-This function creates a node struct, containing the symbol and weight of the node, and returns it.
- void *node_free(Node **node)-This function frees the node that the given double pointer points to.
- void *node_print_tree(Node *tree, char ch, int indentation)-This function takes a tree pointer, a character, and an indentation to print the tree.
- PriorityQueue *pq_create(void)-This function allocates memory for a Priority Queue struct and returns a pointer to it.

- void pq_free(PriorityQueue **q)-This function takes a double pointer to a priority queue and free it.
- bool pq_is_empty(PriorityQueue *q)-This function takes a priority queue object and checks if the queue's list is NULL and returns true or false otherwise.
- bool pq_size_is_1(PriorityQueue *q)-This function takes a priority object and checks if the queue list contains a node but the next list element is null, returning true or false if it has other list elements. void enqueue(PriorityQueue *q, Node *tree)-This function takes a priority queue object and a the given node, tree, inserting the node into the priority queue. bool dequeue(PriorityQueue *q, Node **tree)-This function takes a priority queue object and a double pointer to the given node, tree, removing an element in the queue with the lowest weight, assigning the node into tree, freeing the element, and returning both true and the removed node. If the queue is empty, it returns false.
- void pq_print(PriorityQueue *q)-This function takes a priority queue object and prints a diagnostic of the trees in the queue.
- bool pq_les_than(Node *n1, Node *n2)-This function takes 2 nodes and returns true if the weight of n2 is greater than the weight of n1, false if the weight of n1 was greater than n2's weight, and returns the outcome of checking if n1's symbol is less than n2's symbol, which only happens if their weights are the same.
- uint64_t fill_histogram(Buffer*inbuf, double *histogram)-This function takes an input buffer and a given histogram, filling in the data from the buffer into the histogram, returning the total size of the file.
- Node *create_tree(double *histogram, uint16_t *num_leaves)-This function takes in a histogram and a node counter, creating a Huffman Tree by creating and filling a priority queue and running the huffman code algorithm. This algorithm compress the nodes into one parent node. Then returns the tree and the node counter.
- Code *fill_code_table(Code *code_table, Node *node, uint64_t code, uint8_t code_length)-This function takes in the given code table, node, code, and code length, filling the code table, with recursion, by traversing the tree until the leaf nodes and stores the code and code length of the node, using the node's symbol as the index.
- void huff_compress_file(BitWriter *outbuf, Buffer *inbuf, uint32_t filesize, uint16_t num_leaves, Node *code_tree, Code *code_table)-This function takes in buffers for the input and output file, the input file's size, number of leaf nodes, the huffman tree, and the code table. It then writes all of these inputs into the output file.
- void huff_write_tree(BitWriter *outbuf, Node *node)-This function takes in a buffer for the output file and the huffman tree, recursively writing the tree into the output file.

Results

My program ran as expected. I used the given command lines to compress the given files and after running dehuff, returned with no difference in dehuff and text files. Below I have a picture of my terminal running all the commands and returning the expected output with the last line showing what would happen if I compared the wrong file, which returned the print statement if there's a difference. I learned a lot making this assignment, especially the concept of Priority Queues, trees, and how they are used to compress the data into one node in order from the least frequent being at the head of the tree to the most frequent being at the leaf nodes.

```
o such file or directory

$ ./huff -i files/zero.txt -o files/zero.huff
$ ./huff -i files/one.txt -o files/one.huff
$ ./huff -i files/two.txt -o files/two.huff
$ ./huff -i files/test1.txt -o files/test1.huff
$ ./huff -i files/test2.txt -o files/test2.huff
$ ./huff -i files/color-chooser-orig.txt -o files/color-chooser-orig.huff
$ ./dehuff-x86 -i files/zero.huff -o files/zero.dehuff
$ ./dehuff-x86 -i files/one.huff -o files/one.dehuff
$ ./dehuff-x86 -i files/two.huff -o files/two.dehuff
$ ./dehuff-x86 -i files/test1.huff -o files/test1.dehuff
$ ./dehuff-x86 -i files/test2.huff -o files/test2.dehuff
$ ./dehuff-x86 -i files/color-chooser-orig.huff -o files/color-chooser-orig.dehuff
$ diff files/zero.txt files/zero.dehuff
$ diff files/one.txt files/one.dehuff
$ diff files/two.txt files/two.dehuff
$ diff files/test1.txt files/test1.dehuff
$ diff files/test2.txt files/test2.dehuff
$ diff files/color-chooserig.txt files/color-chooser-orig.dehuff
 such file or directory
$ diff files/color-chooser-orig.txt files/color-chooser-orig.dehuff
$ diff files/color-chooser-orig.txt files/test2.dehuff
ig.txt and files/test2.dehuff differ
```

Figure 1: Screenshot of the commands being inputted and returning the correct output in the terminal.

Error Handling

Errors that I've ran into consists of infinite loops, segmentation faults, invalid reads and writes, and leaks. With the infinite loops, my error was that I left a lot of varaibles in main, uninitialized. The invalid reads and writes were caused by using pointers that pointed to NULL, causing segfaults and memory leaks. My issue derived from me allocating memory in enqueue for two pointers that pointed to the previous and current list element when traversing the priority queue. I realized that I didn't even need to to do that so I just deleted the two lines that did the allocating and the program fixed itself. After that I didn't have an errors besides memory leaks that were caused because I forgot to free memory that was allocated in main for the histogram, tree, and code table.