Assignment 7 Design

Purpose

The purpose of this program is to compress and decompress a data file without losing any data (lossless data compression). We do not have an infinite amount of space in our computers so it is often nice to be able to make files smaller to take up less space. We also need a way to bring those files back to their original form when they need to be used. That is what this project does.

Questions:

Describe the goal of compression. The goal of compression is to be able to store some data in less space than before. When the data does not have a lot of entropy, such as 'aaaaaaaaaa', we can see that we don't need to use 8 bits to represent each letter a, but rather we can use much fewer bits and convert back to its original form later.

What is the difference between lossless and lossy compression? What type of compression is huffman coding? What about JPEG? Can you lossily compress a data file?

Lossless compression means that you don't lose any data and can get the file back to its original form. In lossy compression, a little bit of data gets lost. Huffman coding uses lossless compression. On the other hand, JPEG typically loses a little bit of data. You can lossily compress a text file but typically that's not what you want.

Can your program accept any file as an input? Will compressing a file using Huffman coding make it smaller in every case? This program specifically works for text files. Huffman coding will not always make it smaller. If there is a lot of characters and they all appear about the same number of times, it becomes much more difficult to compress it and may result in a larger file.

How big are the patterns in huffman coding? What kind of files does this lend itself to? The less random the files are, the bigger the patterns are. This is usually the case for text files.

Take a picture on your phone. What resolution is the picture? How much space does it take up in your storage? The resolution of the picture is 12mp and takes up about 2mb.

Why do you think that the picture you took is smaller than you would expect? My phone uses data compression behind the scenes by figuring out which colors are most frequent and storing them in less bits.

What is the compression ratio? It is about 3.5.

Would you expect your program to compress the image you just took with your Huffman program? I would say no because each pixel takes 3 bytes, and that makes it difficult because we wouldn't be able to treat each pixel as one, but rather a combination of 3 parts.

Are there multiple ways to achieve the same smallest file size? Explain why this might be. This is possible because some characters may show up the same number of times as other characters, and so your tree might look a little bit different if you swapped the characters with the same weight. In the case of my program, if the weights are the same, then we compare the symbols and the smaller symbol is further up in the queue, but it does not have to be this way.

When traversing the code tree, is it possible for an internal node to have a symbol? No. A node with a symbol must have no child node because we need a way to know when one compressed symbol ends and another begins.

Why do we bother creating a histogram instead of randomly assigning a tree? Because some characters are more common than others, and this can vary from file to file. We would like to know how often each character comes up in a file to optimize the amount of times we can represent common letters with short codes.

Compare huffman coding to morse code. Why does Morse code not work for our purposes? Huffman coding is very similar to Morse code, except that no code is the prefix for another code, which gives a distinct way of knowing when one symbol ends and another begins. This is the problem with Morse code because sometimes we do not know if a letter is fully finished, or if it is just the start of another letter.

Using the example binary, calculate the compression ratio of a large text of your choosing. I choose to compress the declaration of independence and it brought it down from 14757 bytes to 8979 bytes.

Testing

To test my code, I will have to test the helper files, then the huff.c and dehuff.c files. Luckily this assignment already comes with tests for the helper files so I will use those to test my code, and also use valgrind to make sure that there are no memory leaks. As for the huff file, I will use shell scripts to compare it with the huff-x86 reference. I will test a variety of different files and make sure that their compressed files are the same. I will then dehuff all of these files and check to make sure it is the same as the original text.

How to Use the Program

To use this program, make sure that you have the required files on your computer, then type “make” to build it. You can then type ./huff -i <infile> -o <outfile>. <infile> is the file that you want to compress and <outfile> is where you want it to go. You can also do a -h flag to view the help menu. This is not required, but -i and -o are. Once you have done this, the program will write the compressed data into the outfile. To view the size of a file, type ls -l <file> and it will tell you the number of bytes the file takes up. To then bring a file back to its original form, type ./dehuff -i <infile> -o <outfile>. <infile> is the file that you want to decompress and <outfile> is where you want it to go. The outfile should be the same as the original file before it got compressed and decompressed. Again you can use ls -l <file> to see how many bytes the file takes up. Just like in huff, you can use -h to view the help menu for dehuff.

Program Design

The program is split up into a number of different files. There is a bitwriter.c file which contains functions to create new bitwriters, close them, write individual bits, and write 8, 16, and 32 bit integers. A bitwriter struct contains a pointer to a file, an 8-bit buffer called byte, and a uint8_t called bit_position to keep track of where we are in the buffer. The bitreader struct contains the exact same thing. In bitreader.c, there are functions to open a bitreader, close it, and read a bit from the file inside the bitreader. There are also functions to read 8, 16, and 32 bits, which just call bit_read_bit many times.

The Node struct contains a symbol, a weight, a code, a code_length, and pointers to two more nodes known as left and right. It contains functions to create nodes with a given symbol and weight. There is also a function to free the node, and one more to print out a tree for diagnostic purposes. The PriorityQueue struct, as defined in pq.c, consists of just a pointer to a ListElement struct called list. The ListElement struct contains a pointer to a node and a pointer to another ListElement. This is our priority queue, and is sorted by weight, from least to greatest. There are functions to create, free, enqueue, dequeue, check if it is empty or has only one element, and lastly a function to print it out.

Huff.c is where it gets put together, and keeps track of the number of each element in a file using a histogram. There are functions to fill the histogram and create the tree based on that histogram. There will also be a Code table which keeps track of each element's new code (representation) and the length of that code. Finally, there is a function that puts it all together and dumps the compressed file into the given outfile.

Dehuff.c keeps track of the code_table and will use it to bring all the new character codes back to their original forms.

Function Descriptions/Pseudocode

bit_write_open() takes in a filename, opens it, allocates a new bitwriter, and saves the opened file into it. It sets the byte and bit_position fields to 0 and returns the newly created bitwriter. bit_write_close () takes in a double pointer to a bitwriter, and first checks if there is any data in the buffer that has not been written yet, and if there is, it writes that last bit of data to the stream. Then it closes the stream, frees the pointer to a bitwriter, and sets the pointer to NULL. bit_write_bit() first checks if the bit_position is greater than 7, meaning the buffer is full. If it is, it writes the bit to the file using fputc and sets the byte and bit_position fields to 0. Then, using bit shifting and masking, it updates the bit at bit_position in byte to the inputted bit, either 1 or 0.

```
if (bit == 0){
    buf->byte = buf->byte & ~(0x01 << buf->bit_position);
    }
    if (bit == 1){
        buf->byte = buf->byte | (uint8_t) (0x01 << buf->bit_position);
```

Finally update bit_position by 1.

Bit_write_uint8 takes in a pointer to a bitwriter and an 8 bit integer. It declares an 8 bit local integer called res, and iterates 8 times, 0 to 7, setting res to x & (0x01 <<i). This will extract whatever value is in bit i. It then shifts res by i bits back to the right and passes that into bit_write_bit(). Bit_write_uint16 and bit_write_uint32 do this 16 and 32 times, respectively.

For the bitreader file, bit_read_open creates a new bitreader, opens the given file in read mode, stores that file in the bitreader, initializes the byte field to 0, and the bit_position to 8. The reason we start the bit_position off at 8 is because we want to immediately start reading from the file stream. The function then returns the pointer to the new bitreader. bit_read_close() takes in a double pointer to a bitreader closes the file stream, frees the pointer to a bitreader and sets the pointer to NULL. bit_read_bit() takes in a pointer to a bitreader and first checks if the bit_position is 8. If it is, meaning it has already read all the bytes in the buffer, it will read a new byte from the stream into the buffer and set the bit_position back to 0. It will then save the bit at the given bit_position of the byte, increase bit_position by 1, and return the saved bit. Bit_read_uint8 takes in a pointer to a bitreader, and declares a local buffer. It iterates from i=0 to 7 and reads a bit from the stream and puts it into the i'th bit in the local buffer, again using bit shifting and masking. After the loop, it returns the bit. Bit_read_uint16 and 32 work the same except they iterate 16 and 32 times, respectively.

The node part is relatively simple. Node_create() will dynamically allocate a node and set its weight and symbol to the given weight and symbol. It then returns the created node. node_free() is a little bit more complicated just because it is a recursive function. It takes in a double pointer to a node, and if the pointer to the node is not null, it calls node free on its left node, then node free on its right node, then frees the current node and sets the pointer to NULL. This will free all the nodes in the tree. Lastly, node_print_node is just for diagnostic purposes.

The priorityqueue part is a little bit more complicated but still not too bad. pq_create() doesn't take in any parameters, and simply dynamically allocates a new pointer to a priority queue and returns it. pq_free() takes in a double pointer to a priority queue, frees the pointer to the pq, and sets the pointer to NULL. pq_is_empty() takes in a pointer to a queue and returns true if the list field of the queue is NULL, otherwise it returns false. Pq_size_of_1 takes in a pointer to a priority queue and checks if the priority queue's list field exists, and if it does, then the list's next field is NULL. If so, return true, otherwise, return false. pq_less_than() takes in two ListElements and first compares their weights. If the first elements weight is less than the seconds weight, return true. Otherwise, if they are equal, then if the first elements symbol is less than the seconds symbol, return true. Otherwise, return false. Enqueue() takes in a pointer to a priority queue and a pointer to a node. It creates a new ListElement and sets its tree field to the given node, and sets its next field to NULL to begin with. If the queue is empty, then set the priority queue's list field to the newly created ListElement and return nothing. Otherwise, iterate through the priority queue until the given node is less than the current node pointer. Once it is, set the previous node's next field to the given node, and the given node's next field to the current node from the loop. Dequeue takes in a priority queue pointer and returns the element with the lowest weight (the first node). It creates a local ListElement pointer called tracker which is set to the head of the pq. The head is then set to the tracker's next field. The node in tracker gets saved, the tracker gets freed, and the node gets returned. Finally pq_print prints the pq for diagnostic purposes.

Now we move on to huff.c. First comes fill_histogram() which takes in a pointer to a file and a pointer to a 256 element array of 32 bit ints. It iterates through the array and clears all the values to 0. It then sets two elements, indexes 0x00 and 0xff, to 1, so that we can be sure there are at least two elements that have a non-zero value. It then reads one character at a time using fgetc from the given file, and increments the array at the index of the read character from the file by 1, and increments the total number of bytes read by 1. This function returns the total number of bytes read. create_tree() takes in a histogram and a pointer to a 16 bit int, and creates a new priority queue by calling pq_create. It then goes through the histogram, and each time we come across a value that is not zero, we make a new node with that symbol and weight, and add it to the queue via enqueue. Then, while the size of the queue is greater than 1, we will dequeue two elements, create a new node with their given weight (and a symbol of a 0 because internal nodes do not have a symbol), and set its left and right fields to the two elements that we dequeued. We will then enqueue that new node. Once we do this enough times and the size of the queue is 1, dequeue it one more time, and return that node pointer.

fill_code_table() takes in a pointer to a code_table array of 256 values, a pointer to a node, a 64 bit code and an 8 bit code_length. If the node is internal, meaning it has at least one child, then call fill_code_table() with the same parameters except for code_length becomes incremented by 1. Then right after this recursive call, set the bit at code_length to 1 and call fill_code_table() again with code_length + 1. If the node is not internal, then set the code_table at the node's symbol's index's code field to the given code, and the code_length to the current code length. huff_compress_file() takes in a pointer to a bitwriter, a pointer to a file, a 32 bit int for the filesize, a 16 bit int for the number of leaves, a pointer to a node which is the code_tree, and a pointer to the code table. It writes the letters H and C, then the filesize and num_leaves. It then calls another function, huffwrite tree, which iterates through an entire tree starting with the head node. If the node is a leaf, then write the bit to the bitwriter pointer, as well as the node's symbol. If the node is internal, recursively call the function for its left node, then its right, and write bit 0 to the bitwriter pointer. Then, in an infinite loop, get a char from the infile, find its code and length based on the code table, and write the code to the outbuf.

For the decompression algorithm, it will take a pointer to a file and a pointer to a bitreader. It first checks to make sure that the first two letters are H and C, and the next are the filesize and num of leaves. The number of nodes is 2 * numleaves - 1. From 0 to the number of nodes, it reads a bit from the inbuf and if the bit read is a 1, it creates a symbol with the next byte from the inbuf, and creates a new node with that symbol and a weight of 0. If the bit is 0, it will pop two elements from the stack and these will be the child nodes of a newly created node with weight and symbol of 0. Once the new node is created, it will be pushed onto the stack. After this is done from 0 to the num of leaves, we pop from the stack one more time into a head node. Then for each bit in the file, we read a bit from the inbuf, and if it a 1, move to the left node, otherwise, move to the right node, and if there is no child node, exit the while loop. Then, still inside the for loop, we must put the node's symbol inside into the output file, and that is all.