CSE 3100 Systems Programming
Homework #4

For instructions on how to work on the assignment on Mimir, please review materials in Lab 0. Do not forget to submit your work before the deadline.

1 Introduction

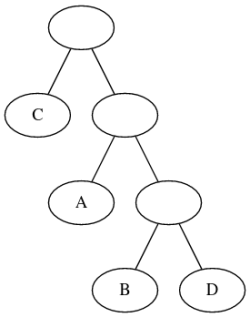
We will be creating a file compression system in this assignment. It can read and compress ASCII text files, and then decompress them.

2 Huffman Coding

In a standard text file, a byte is used to store each character. For example, in ASCII, the 'A' would be represented by the binary number: 10000001. Thus, each character takes up 8 bits. However, most text files don't contain all 128 characters in ASCII, and character frequency is generally non-uniform. So, we would like a code to represent each character using a variable number of bits, where the most common characters are represented using the fewest number of bits. More specifically, we would like a character coding that minimizes the number of bits needed to represent a given string of characters.
Suppose we have a string S containing n unique characters $A = \{a_1, a_2, \cdots, a_n\}$, and character a_k occurs w_k times in the string. Character a_k can be represented by a code c_k, where $|c_k|$ is the number of bits in the code. For example, consider the string $S = XYZZ$. Then, $A = \{X, Y, Z\}$, with $w_1 = 1$, $w_2 = 1$ and $w_3 = 2$. Suppose our code is $c_1 = 01$, $c_2 = 00$ and $c_3 = 1$. S can be encoded as: $XYZZ = 010011$, requiring 6 bits. Of course, the number of bits required to represent a string can be computed as:

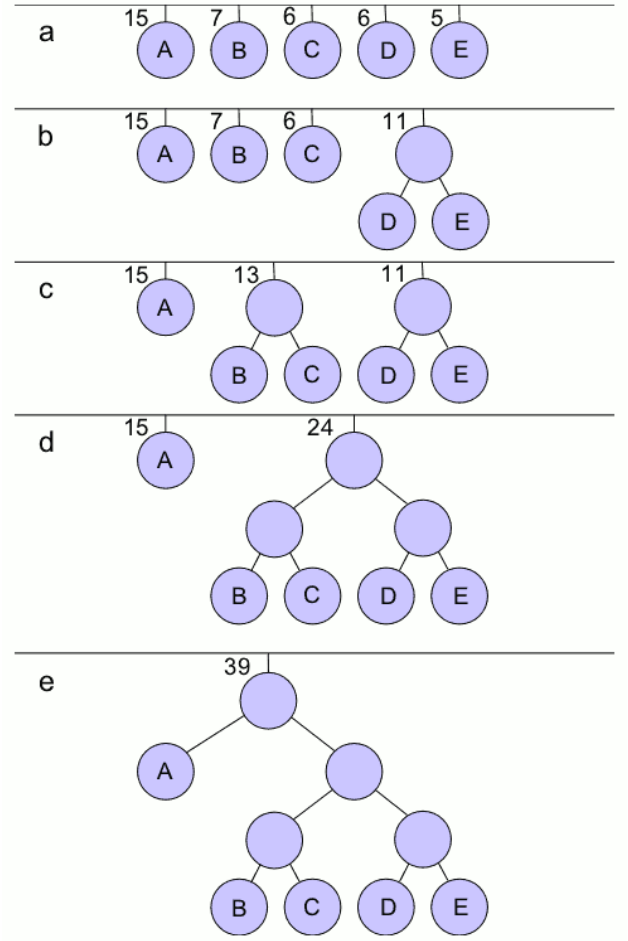
$$L(W, C) = \sum_{k=1}^{n} w_k |c_k| \tag{1}$$

But, we have to be careful. Not all codes will be be reasonable. For example, suppose $c_1 = 10$, $c_2 = 1$ and $c_3 = 0$. When we decode the encoded string, it would be impossible to differentiate between X and YZ. This is where *Huffman Coding* is useful . This method, invented by David Huffman (while he was a graduate student), allows us to construct a code that minimizes equation 1, while being unambiguously decoded. The basis for this method is that we construct a binary tree, where each leaf vertex corresponds to a character, and then the codeword for the character is the path from the root to the leaf node. A 0 tells us to go to the left child, and a 1 tells us to go to the right child. For example, consider the following tree:



A `C` would be represented by 0, while an `A` would be represented by 10. `B` would be encoded by 110, while `D` would be encoded by 111. How can we construct such a tree for a string S with characters $A = \{a_1, \cdots, a_n\}$ and frequency $W = \{w_1, \cdots, w_n\}$? We begin with a min priority-queue. Recall that a PQ holds a collection

of objects, and allows you to extract the lowest (or highest) priority object in the queue. For each letter $a_k \in A$, we create a node containing a_k, with priority w_k. These nodes go into the queue. Then, we repeatedly extract the two lowest priority nodes u and v in the queue, and create a new internal node t, with u and v as its children, and priority that is the sum of the priorities of u and v. Here is an example of this in action:



More precisely, these are the steps of the algorithm for alphabet A with weights W.

1. Create a node for each letter $a_k \in A$, with priority w_k.
2. Add the nodes to the Priority Queue PQ.
3. While there is more than 1 node in PQ
   (a) $u \leftarrow extractMin(PQ)$
   (b) $v \leftarrow extractMin(PQ)$
   (c) Create a node t with u as its left child, v as the right child, and priority that is the sum of u and v.
   (d) Add node t to PQ
4. The remaining node is the root node of the tree

When we want to compress the string S, we construct a Huffman Tree for it, and extract the code for each character in the string's alphabet A. Then, to compress, we simply concatenate the codes for each character in the string.

We've provided you with a priority queue implementation, based on the heap you saw during class. It can be found in `heap.c` and `heap.h`. You need to implement the `createHuffmanTree`, `freeHuffmanTree`, `extractCodes`, `extractCodesAux` and `codeToString` functions in `huffman.c`. I will go over each function here:

2.1 createHuffmanTree

```
Node* createHuffmanTree(unsigned long* freq);
```

This function takes in an `unsigned long* freq`, and returns a `Node*`. The `freq` parameter should be an array of `unsigned long`, with length equal to `NUM_SYMBOLS`, which is 128. Basically, standard ASCII contains 128 symbols, and each of these has an element in the `freq` array, indexed according to the ASCII code for that symbol. So, for example, the frequency of 'A' could be accessed by either `freq['A']` or `freq[65]` (A has ASCII code 65). Note that you only need to create nodes for symbols with frequency greater than zero (i.e. those that actually appear in the text).
If you open `huffman.h`, you'll notice that a `Node` has the following fields:

- `unsigned long freq`
- `char c`
- `struct Node* left`
- `struct Node* right`

Each leaf node of the Huffman Tree corresponds to a character, which is stored in the `c` field, as well as a frequency. Since a leaf node, by definition, does not have any children, the `left` and `right` fields are NULL. However, for internal nodes, set `c` to 0, and `freq` to the sum of `freq` for its left and right child.

2.2 freeHuffmanTree

```
void freeHuffmanTree(Node* root);
```

This function should free each node of the tree. You'll find recursion helpful here.

2.3 extractCodes and extractCodesAux

```
void extractCodesAux(Node* node, Code* codes,unsigned long long path, char nbits);
Code* extractCodes(Node* root);
```

The purpose of these two functions is to navigate the Huffman Tree, and extract the codes for each symbol. This will give us an array that maps each symbol in the text to a `Code`. Recall that a symbol is encoded by its path through the tree, and that each symbol is stored on a leaf node of the tree. In `huffman.h`, you'll notice that a `Code` contains an `unsigned long long bits`, and `char nbits`. For example, in the above huffman tree (the one with purple nodes), the letter 'C' would be encoded by `101`; the `bits` field would be set to `0000000000000000000000000000000000000000000000000000000000000101`, and the `nbits` field would be set to 3. 'A' would be encoded by `0`, and the `bits` field would be set to `0000000000000000000000000000000000000000000000000000000000000000`, and `nbits` would be set to 1. The `extractCodes` function should allocate memory for `NUM_SYMBOLS Code`, and call the `extractCodesAux` function. The `extractCodesAux` function should traverse the tree, keeping track of its path using the `path` and `nbits` parameters. When it reaches a leaf node, it should add the code for that symbol to the `codes` array. Recursion is your friend here.

2.4 codeToString

```
char* codeToString(Code code);
```

This takes a `Code`, and converts it to a string. This is useful because we'd like our compression program to output the code for each symbol to the user.

2.5 Huffman Mimir Tests

I've given you two Mimir tests that test these functions. Basically, they call `createHuffmanTree` to create a huffman tree, and use `extractCodes` and `codeToString` to extract the symbol codes. The "Extreme Huffman Tree" builds a huffman tree containing for the string `bbbbbbbbbbbbbbbbbbccrrss`. What makes it "extreme" is that the four characters (b, c, r and s) could potentially be represented using only two bits per character; however, the huffman tree encodes r and s using three bits each. This is because "b" occurs very frequently in the string.

3 Compress

In this part, you'll need to implement three functions: `countBits` and `compress` in `compression.c`, and `packBits` in `file.c`.

3.1 countBits

```
unsigned long countBits(char* text, unsigned long sz, Code* codes);
```

You're given a string `text` of length `sz`, and an array of `Code` (of length `NUM_SYMBOLS`). Each character `c` in `text` corresponds to some `Code` consisting of `nbits` bits. Loop over the the string, and count the total number of bits needed to store the encoded string.

3.2 compress

```
BitStream compress(char* text, unsigned long sz, Code* codes);
```

Now that we have the codes for the characters in `text`, we can actually do the compression. But, this begs the question: how do we store the compressed string? C doesn't have a bit type, where we can simply create an array of bits, and manipulate it like any other array. Suppose the compressed string consists of m bits. To make things simple, we'll use an array of `char`, where each element stores a bit (as either 0 or 1). You may notice that this wastes an incredible amount of memory: for each bit in the compressed string, we're using an entire `char`, which takes up a whole 8 bits. But, storing it this way in memory makes the code more modular, so its an acceptable tradeoff for a homework assignment. This array (and its length) get put into a `BitStream` struct, which is defined in `compression.h` file. The `countBits` function returns the number of `char` you need to allocate for this array.

3.3 packBits

```
ByteArray packBits(char* bits, unsigned long numBits);
```

Now that we have a `BitStream` of the compressed string, we need to pack the bits into bytes, so that they can be efficiently stored on disk. Suppose we have an array of "bits" (each being stored by a `char`) as follows: $\{0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 1, 1\}$, with length 12. This can be stored using 2 bytes. The first byte would consist of the first 8 bits: `00010111`, or 23 in decimal. To store the remaining four bits, we need to add *padding*; we append `0000` so that the second byte would be: `01110000` or 112 in decimal. The `ByteArray` struct would be filled in as follows:

- `bytes` would be an array $\{23, 112\}$.
- `numBytes` would be 2.
- `padding` would be 4, since we had to append 4 bits of padding

3.4 Mimir Tests

I created a Mimir test for each of these functions: `countBits`, `compress`, and `packBits`.

4 File Operations

```
FileData* readFile(char* fName);
```

Open `compress.c`. You'll notice a struct called `FileData`. You need to implement the `readFile` function, to initialize a `FileData` struct, and fill in the following fields: `size`, `contents` and `freq`. Open the file, compute its length (the `fseek` and `ftell` functions are helpful here), and read its contents. Then, count the character frequencies. I explained how the character frequencies can be stored in an array of length `NUM_SYMBOLS` in the section on the `createHuffmanTree` function.

5 Wrapping Up

5.1 `huffmanSetup`

```
void huffmanSetup(FileData* fd);
```

This function needs to call the necessary functions to build the Huffman Tree, and extract the codes. The `root` and `codeWords` fields need to be filled in.

5.2 `huffmanCompress`

```
HuffData* huffmanCompress(FileData* fd);
```

Call the `compress` and `packBits` functions to compress the file contents.

5.3 `releaseFile`

```
void releaseFile(FileData* fd);
```

This function should free the huffman tree, and the `codeWords`, `contents`, and the `freq` fields of `fd`, as well as `fd` itself.

5.4 Mimir Test

I've provided two Mimir tests that test the `compress` program. One calls it to compress a file, and then compares the output to a reference. The other uses Valgrind to check for any memory errors and/or leaks.

6 Running Compression and Decompression

You can use `make` to compile the compression program. We can create a file called `example.txt` containing the word "hello", and compress it as follows:

```
$ printf "hello" > example.txt
$ ./compress example.txt example.jzip
Uncompressed size: 5
Compressed size: 2
Number of pad bits: 6
Symbols:
e: 1, 111
h: 1, 10
l: 2, 0
o: 1, 110
```

We've given you the binary for the `decompress` program. Run it as follows:

```
$ ./decompress example.jzip example_output.txt
```