Computer Systems Principles

Huffman

Contents

Huffman Project Assignment

Overview

This project assignment will exercise your understanding of bit-level manipulation in C, allocations and manipulation of array and pointer-linked data structures in C, and working with larger C programs. You must complete all the exercises below using a text editor of your choice. Make sure you follow the instructions carefully. The actual code you write is fairly short. However, the details are quite precise. Programming errors often result from slight differences that are hard to detect – so be careful and understand exactly what the exercises are asking you to do.

The goal of this project is to complete the implementation of two programs using the Huffman coding algorithm for file compression and decompression. In particular, the first program `huffc` will take as input a text file in ASCII format and generate a compressed version of that file as output:

```
$ ./huffc books/holmes.txt holmes.he
```

The program will not produce any output to the console, but it will create a new file called `holmes.he` that is a compressed form of the original input file `books/holmes.txt`. In our implementation this results in the following file sizes:

```
$ ls -lh books/holmes.txt
-rw-r--r-- 1 vagrant vagrant 14M Oct 10 17:45 books/holmes.txt
$ ls -lh holmes.he
-rw-r--r-- 1 vagrant vagrant 8.2M Oct 14 18:51 holmes.he
```

As you can see, the `huffc` program was able to compress the original input text file by almost 42%. Since compression is not really interesting unless we can decompress a compressed file, we can then run the decompression program `huffd` to convert the file back to its original form:

```
$ ./huffd holmes.he holmes.txt
```

Note, that we did not overwrite the original text file in the books directory! Be careful that you do not do this! We can then see if our decompressed file is the same size as the original text file:

```
$ ls -lh books/holmes.txt
-rw-r--r-- 1 vagrant vagrant 14M Oct 10 17:45 books/holmes.txt
$ ls -lh holmes.txt
-rw-r--r-- 1 vagrant vagrant 14M Oct 14 18:55 holmes.txt
```

Although this compares the size, it is even more important that they contain the **exact same** bytes. This is an easy test using the `diff` tool:

```
$ diff books/holmes.txt holmes.txt
```

The `diff` tool should return without printing anything to the console. If it shows differences, you know something went wrong! (Note: if you ever want to compare two *binary* files, take a look at the `cmp` program.)

In this project you will complete the implementation of these two programs to compress and decompress ASCII text files using Huffman coding.

Suggested Reading

Although the goal of this assignment is not really to learn about Huffman coding in particular, you will need to read up on how it works in order to understand the provided starter code and how to implement the parts indicated below. A good starting point is the description of Huffman coding on Wikipedia.

You should also watch the video that we provide that covers the Huffman algorithm and highlights the important files in the implementation.

Note, you should read the documentation for this assignment *multiple times* to get a better understanding of what you must do, how Huffman coding works, and how the implementation is constructed, so that you will be successful. Each reading will provide deeper understanding as you come to grasp all the parts and how they work together.

Part 0: Project Startup

Please download the project startup zip file, as for previous assignments. Once you have the zip file in your home directory you can execute the following command:

```
$ unzip huffman-proj-student.zip
```

This will *unarchive* the contents of the zip file and you will see the `huffman-proj-student` directory. You can then go into that directory from the terminal using `cd` (change directory):

```
$ cd huffman-proj-student
```

Part 1: Understand The Code

We provide you with starter code for this assignment. Your first task is to **read** through each of the provided source files in detail, so that you understand the structure of the code. This is particularly important for this assignment because of the number of source files involved. The `huffc` and `huffd` programs are themselves quite simple, but they depend on a number of "modules" for their implementation. The following is a description of each of these modules and their corresponding source files.

BitsIO Module

The BitsIO module consists of the following source files:

- `bits-io.h`
- `bits-io.c`

The BitsIO module provides support for writing to and reading from a compressed file. A Huffman compressed file is stored in the following format:

```
[SERIALIZED HUFFMAN CODING TREE] [HUFFMAN ENCODED (COMPRESSED) DATA]
0                              s h                                n
```

The `SERIALIZED HUFFMAN CODING TREE` part of the file starts at byte *0* and ends at byte *s*. It contains a string-based representation of the Huffman coding tree that was used to encode the input ASCII text file. We need to include the tree as part of the compressed file so we know how to decompress the encoded ASCII in the `HUFFMAN ENCODED DATA` part of the file. This part of the file starts at byte *h* (one after *s*) and continues to the end of the file, with byte *n*. It contains the bit encoding of each of the characters found in the original input text file.

This module includes functions for opening and closing a compressed file, writing and reading bits, and writing and reading the Huffman tree. The two most important functions are `bits_io_write_bit` and `bits_io_read_bit`. Both of these functions depend on special formatting of bytes that are written to and read from a compressed file. The details of the formatting are elaborated in the `bits-io.c` file – which you should read completely! This module relies on the stdio `fputc` and `fgetc` functions for writing and reading a byte respectively. It is not possible to write or read individual *bits* to and from a file, so we must buffer in an individual byte the bits that we write and read. When we are writing bits to a file, we write bits to that one-byte buffer until it is full before writing the byte to the actual file. Likewise, when we read in individual bits we must first read a byte from the compressed file into the byte buffer, and then read individual bits from the byte buffer.

To do this correctly we must have a well defined format for the bytes we are writing and reading. In the case of writing bits we initialize the byte buffer `B` with the following bit-level representation:

```
B = 11111110
```

We provide a `#define` for this value, `NO_BITS_WRITTEN`. Thus, in C we would initialize B like this:

```
B = NO_BITS_WRITTEN;
```

The 0 bit is used as a delimiter between the unwritten bits and the written bits. Each time we write a bit to the byte buffer we shift the byte buffer to the left by 1 (`B << 1`) and write the bit `w` to the byte buffer using the bitwise OR (`|`) operator. Thus, the way we write a bit to the byte buffer is `B = (B << 1) | w`. If `w` is the bit `1` then the result of a write to this buffer would be:

```
B = 11111101
```

We continue to write bits to the byte buffer until the most significant bit is `0`. For example, here is a byte buffer that is full:

```
B = 01001101
```

When the byte buffer is full we write the byte to the output file and re-initialize with `NO_BITS_WRITTEN`. The main structure we use for representing a bit stream connected to a file (with a byte buffer) is the `BitsIOFile` struct found in `bits-io.c`. You will notice the definition of the byte buffer as a field of this structure called `byte`. You can also look at the implementation of the `bits_io_write_bit` function to see how it is used. Notice that this format allows us to write only 7 bits to a byte - not 8 bits - since we are using one bit as the delimiter. This introduces a 1 bit overhead for each byte that is written to the compressed file. (If we applied the same concept to a larger unit, say a 32-bit or 64-bit word, the fractional space loss would be much lower, but this one-byte buffer serves for the assignment.)

Reading bits from the byte buffer is a little more complicated since we must handle the special case of the last byte in the compressed file, which may not have been fully filled with bits written. In general we simply shift the byte buffer left by 1 to start reading the next bit available. Starting from the full byte buffer above a left shift by 1 would remove the 0 bit in the most significant bit position. We would then shift left again to read the next bit.

However, we also desire a clear stopping condition, so we apply somewhat the same delimiter concept as we shift, shifting particular bit values in at the low end as we shift the other bits out the high end. Specifically, the first time we shift, we shift in a 1, and after that always shift in a 0. When the value becomes 10000000 (for which we `#define` the name `ALL_BITS_READ`), we know we are done reading from that byte. (Notice that, given our shifting rules, and the way we encoded when writing, this value cannot arise other than as the result of having read all the useful bits out of a byte.)

The comments in `bits-io.c` include more detailed illustrations of the scheme in action.

Concerning EOF, normally it would be problematic to read using `fgetc` into an unsigned byte and then try to check for EOF. The reason is that EOF is the int value -1. When this is stored into an unsigned byte, the value stored will be 11111111 in binary, but since the number is unsigned, it will be 255, which can never equal -1. Therefore, it is usually safer to put the result of `fgetc` into an int and compare against that, since it can distinguish the legimitate byte value 255 from the EOF value -1. In this case, though, we luck out: our encoding does not allow 11111111 to occur as a byte in the file, because we always have the 0 delimiter bit. Thus, we *can*, in this instance, read into the unsigned byte and look at the value to see whether we have EOF. But comparing against EOF itself will still give the wrong answer. Therefore we `#define` a symbol, `EOF_VALUE` that *is* suitable for comparing against an unsigned byte to see if we read EOF from `fgetc`.

Tree Module

The Tree module consists of the following source files:

- `tree.h`
- `tree.c`

The Tree module provides support for creating a Huffman binary tree. A Huffman binary tree consists of *internal* and *leaf* tree nodes. Both node types are represented by the structure `TreeNode`; they are distinguished by a `type` field that can either be `INTERNAL` or `LEAF` (values of an enumeration type). An internal tree node is a node that has at least one child node. Its frequency field is always the sum of the frequencies of its children (initially 0 when it has no children yet), and its character is always the null character `'\0'` (that field is not really used for internal nodes). A leaf tree node is a node that has no children and its frequency field has a value that is greater than 0 and a character encountered from the input text file. A `TreeNode` also has a unique id that is used to identify it when it is serialized in the compressed file (as mentioned in the BitsIO module section). In addition, a `TreeNode` has an additional pointer called `next` that points to another `TreeNode`. This field is used during the deserialization process to form a linked list of tree nodes.

The module provides functions for creating new `TreeNode` objects, freeing them, getting the size of the tree, and printing a tree. You may consider using the `tree_print` function during testing and debugging to see what your tree looks like, to make sure it is something sensible. In addition, this module provides the `tree_serialize` and `tree_deserialize` functions that convert a tree of `TreeNode` objects into a string that is written to a file, and conversely converts a string representation of a tree into `TreeNode` objects. This is used by the BitsIO module to write the `SERIALIZED TREE` part of its formatted file. You do not need to implement anything in this file. However, it is important that you review the implementation and read the comments in the code to understand how it works.

Priority Queue Module

The Priority Queue module consists of the following source files:

- `pqueue.h`
- `pqueue.c`

The Priority Queue module provides functionality for creating and using a priority queue, to assist in building the Huffman tree. This implementation is simple in that it uses an array and a `sort` function rather than a more efficient implementation such as a heap data structure. The module provides functions for creating a priority queue, freeing a priority queue, enqueue and dequeue, size, and printing.

The implementation consists of a `PriorityQueue` structure containing an array of `TreeNode` pointers and an integer field `count` indicating the number of used slots in the priority queue. The array has a max size of 256, which will never be exceeded because we will only need it large enough to hold the ASCII character set. In fact, we could easily reduce this to 128 as the ASCII characters range from 0 to 127. Note that the implementation is customized for this application and thus does not provide a generic implementation of a priority queue data structure.

We include a utility function called `sort` that uses the `qsort` function defined in `stdlib.h`. You can use `man` to read up more details on the `qsort` function. In short, it will sort an array given a *comparator* function. We have not discussed function pointers in C. However, this is an example of one. The comparator function is used to sort the priority queue by priority in ascending order (lowest priority first). The `sort` function is called after a `TreeNode` object is enqueued in the queue.

You will complete the implementation of the `pqueue_enqueue` and `pqueue_dequeue` functions. You will see that these are labeled with a `TODO` and additional instructions on how to complete the implementation.

Table Module

The Table module consists of the following source files:

- `table.h`
- `table.c`

The table module is used to create an *encoding table* that maps characters to their bit encoding. The table is constructed using a Huffman tree built from a specific input text file. The module provides the `table_build` function that takes as its parameter a `TreeNode*` and returns an `EncodeTable*`. The returned encoding table can be used with the `table_bit_encode` function to return a character string (`char *`) of '1' and '0' digits representing the encoding. The encoding string is terminated by the usual null character.

Although it is possible to use the original Huffman tree to determine the encoding for a character it would require a search over the entire tree to find the corresponding encoding. For this reason we create a simple lookup table that "remembers" what the encoding is. This table is built from visiting all the nodes and paths in the Huffman tree and recording each path to each leaf node (each character).

This module also provides a `table_print` function that can be used during testing and debugging to ensure that your implementation is working properly. You do not need to make any changes to this module.

Huffman Module

The Huffman module consists of the following source files:

- `huffman.h`
- `huffman.c`

The Huffman module provides the core functionality for the Huffman coding algorithm. The Huffman coding algorithm consists of three phases. The first phase computes the frequencies of characters found in the input text file. After we have found the frequencies of the characters, the second phase creates a new `TreeNode` for each of the characters that were found (frequency greater than 0) and adds them to a priority queue. The priority queue will arrange the `TreeNode` objects in ascending order based on their frequency. The third phase iterates over the priority while it has more than one item, building the Huffman tree. The details of each of these phases are elaborate in `huffman.c`. You must implement each of the phases in order to successfully construct a Huffman tree.

Encoder and Decoder Module

The Encoder and Decoder modules consists of the following source files:

- `encoder.h`
- `encoder.c`
- `decoder.h`
- `decoder.c`

The Encoder module uses the modules described above to carefully encode an ASCII text file and generate a Huffman compressed output file. The Decoder module uses the modules described above to decode a Huffman compressed file and generate the ASCII text file. The API to both of these modules are simple. For encoding, you create a new `Encoder` object, encode the file, and free the `Encoder`. The Decoder works in a similar fashion. You do not need to modify these modules.

The huffc and huffd Programs

The `huffc` and `huffd` programs are built from the `huffc.c` and `huffd.c` source files respectively. You should read the code contained in these files - it is self-explanatory.

Compiling the Project

To compile the project you need to use the following command:

```
$ make
```

This will produce several *object files* and four *executable files*. The four executable files are:

- `huffc`
- `huffd`
- `treeg`
- `tableg`

The `huffc` and `huffd` executables perform Huffman compression and decompression respectively. The `treeg` program will read in an ASCII text file and print a text representation of the Huffman tree. You can use this as part of your testing and debugging to see if you have done this properly. You can use our provided solution executable `treegs` to see what our solution prints out. To see if your output is the same as ours you can redirect the output of both programs to files and compare them using the `diff` command. (*Note:* We are *not* talking about comparing the *programs* - your versions will almost certainly be different from ours because your source code is different. It is the *output* of the programs that you should compare.)

Likewise, the `tableg` executable will print out the table created from the Huffman tree. We have provided our solution executable `tablegs` so that you can compare the output against our solution.

You can remove all the generated binary files using make:

```
$ make clean
```

Part 2: Testing

We have provided tests that you can run to see if you are on the right track toward a solution. The `test/public-test.c` file contains tests using the check C unit testing framework. You do not need to know every detail of the check framework to use it. Each test is defined by using special macros that auto-generate test functions. If you want to know more about C macros you can read this article. You can run the tests with the following command:

```
$ make test
```

You are welcome and encouraged to introduce additional tests. To add a new test you should copy one of the existing tests and modify it. After you add your own test you need to add it to the test suite. To do this you simply extend the `tester_suite` function to include an additional `tcase_add_test(tc_inc, <your test name>);`.

Part 3: Complete Priority Queue

Your first task is to implement the functions:

- `pqueue_enqueue`
- `pqueue_dequeue`

in the `pqueue.c` file. The description of what you need to do is outlined in the functions themselves. You should run the tests to make sure that your priority queue implementation is working properly before you move on to the next part.

Part 4: Complete Huffman

Your second task is to implement the functions:

- `compute_freq`
- `create_tree_nodes`
- `build_tree`

in the `huffman.c` file. The description of what you need to do is outlined in the functions themselves. You should run the tests to make sure that your Huffman coding implementation is working properly before you move on to the next part. In addition, you can use the `treeg` and `tableg` executables to see what the output is and compare against our solution executables. If you are getting the exact output as our implementation you are in great shape.

Part 5: Complete BitsIO

Your third task is to implement the function:

- `bits_io_read_bit`

in the `bits-io.c` file. The description of what you need to do is outlined in the function itself. You should run the tests to make sure that your implementation is working properly before you move on to the next part.

Part 6: Compression and Decompression Running and Testing

Once you reach this part of the assignment and the tests are passing you are in a good position to start running the actual `huffc` and `huffd` executables. We have provided the solution binaries as `huffcs` and `huffds` that you can use to see if your implementation is working properly. For example, you can run our `huffcs` to generate a compressed file and then use your `huffd` to decompress. Likewise, use your `huffc` to generate a compressed file and then use our `huffds` to decompress it. You should also compare the size of the compressed files generated by our solution and yours to see if they match. You can do this using the following command:

```
$ ./huffc books/holmes.txt holmes.he
$ ./huffcs books/holmes.txt holmess.he
$ ls -lh holmes.he holmess.he
```

Another test you can perform is to decompress a compressed file generated by your programs and compare the decompressed text file against the original to make sure they are identical:

```
$ ./huffc books/holmes.txt holmes.he
$ ./huffd holmes.he holmes.txt
$ diff holmes.txt books/holmes.txt
```

You can also use the `cmp` program to compare binary files, byte for byte.

Debugging Hints

C programs involving pointers can be tricky to debug. You should use the `valgrind` tool to help identify problems in your code and any memory leaks (allocating memory without freeing it). To run `valgrind` from the command line you do this:

```
$ valgrind ./huffc books/holmes.txt holmes.he
$ valgrind ./huffd holmes.he holmes.txt
```

This will report any invalid access to memory and any memory that had been allocated and not freed before the program terminated. If your program tries to access memory in a way it is not allowed to do you will likely see `segmentation violation (core dump)` as the only output after running your program. Make sure you use `valgrind` to help better understand where things went wrong. Here is a list of how to go about debugging your C code:

1. Use `printf` to output debugging information. Do not underestimate the usefulness of this simple method of debugging!
2. Use `valgrind` to narrow the scope of where your problem is (which function), then use the `printf` method.
3. Use `gdb` if you are really stuck and need to step through your program one line at a time.

We will check your programs to make sure that you do not have any memory leaks.

Submission Instructions

You must submit your assignment as a zip file. After you complete the assignment you need to run the following command from your `huffman-proj-student` directory:

```
$ make zip
```

This will create the file `huffman-submit.zip` which you need to upload to the assignment activity in Moodle. **Make sure you add the submission zip, not the original one!** Please submit your assignment to Moodle by the assigned due date. Please make sure you have followed all the instructions described in this assignment. Failure to follow these instructions precisely will likely lead to considerable point deductions and possibly failure for the assignment.