ECE26400 Programming Assignment #5

This assignment is a continuation of PE08 and PE09. The main learning goals are:

1. How to create and manipulate linked lists and trees.
2. How to free the memory associated with a linked list and a tree.
3. How to perform bit-wise operations

1 Getting started

You should unzip on eceprog.ecn.purdue.edu the zip file `pa05_files.zip` using the following command:

```
unzip pa05_files.zip
```

The zip file `pa05_files.zip` contains a folder, named `PA05`, and a subfolder `examples` containing a few examples:

1. Please see Section 4 about the files in the subfolder.

In this assignment, you have to decide what `.h` file(s) to create and what `.c` file(s) to create. You must also create a `Makefile` that we will use to compile your source code into an executable called `pa05` using the command "`make pa05`".

2 Huffman coding

We assume that you have read the file `PE09.pdf` and successfully completed PE09. At this point, you should be able to construct a Huffman coding tree for any valid input file.

In PE09, one of the output files produced is the character-based representation of the Huffman coding tree constructed for a valid input file. In this assignment, you will deal with the bit-based representation. In particular, you will demonstrate that you are able to output to a file a bit-based representation of the Huffman coding tree constructed in PE09. This is one of the tasks in performing Huffman coding of the input file. **This task accounts for the full credit of the assignment.**

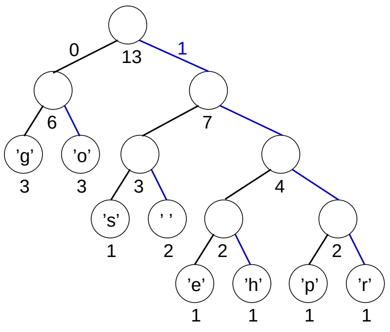
You will also demonstrate that given a file that contains a bit-based representation of a Huffman coding tree, you are able to re-construct the Huffman coding tree. This is one of the tasks in performing Huffman decoding of a compressed file. **This task allows you to earn bonus points.**

2.1 An encoding step

The following description is adapted from Section 2.5.1 of `PE09.pdf`.

The compression program must store some information in the compressed file that will be used by the decompression (decoding) program.

In PE09, to store the tree in a file, we use a pre-order traversal, writing each node visited as follows: when you encounter a leaf node, you write a 1 followed by the ASCII character of the leaf node. When you encounter a non-leaf node, you write a 0.



For the "go go gophers" example, the topology information is stored as `"001g1o001s1 001e1h01p1r"`, based on pre-order traversal. In this example, we use characters `'0'` and `'1'` to distinguish between non-leaf and leaf nodes. As there are eight leaf nodes, there are eight `'1'` characters and seven `'0'` characters for non-leaf nodes. This approach used a total of 23 bytes. We refer to this as the character-based representation.

The representation of a Huffman coding tree can be made more economical if we use bits 0 and 1 to distinguish between non-leaf and leaf nodes. In this example, there will be a total of 79 bits (64 bits for the ASCII codes of the eight leaf nodes, 8 1-bits for the leaf nodes, and 7 0-bits for the non-leaf nodes).

For example, in the bit-based approach, the first 16 bits (or the first 2 bytes **in the convention that left to right is MSB to LSB**) describing the topology of the Huffman tree constructed for encoding the string `"go go gophers"` are `00111100 11111011`.

The least significant two bits of the first bytes are 0-bits (for non-leaf nodes). The next bit is a 1-bit (for a leaf node). It is followed by the ASCII code for `'g'`, which has a bit pattern of `01100111`. The 5 least significant bits (`00111`) are in this byte. The 3 most significant bits (`011`) are in the least significant position of the next byte. This is followed by another 1-bit, followed by `1111`, the 4 least significant bits of the ASCII code for `'o'`, which has a bit pattern of `01101111`.

Note that ASCII code for 'g' straddles two bytes. Similarly, the ASCII code for 'o' straddles two bytes.

In the bit-based representation of the Huffman coding tree, the last byte may not contain 8 bits. In this case, the most significant unused bits of the last byte should be assigned 0.

In PE09, you focus on the construction of a Huffman coding tree and the storage of a Huffman coding tree in a file using the character-based representation. In PA05, you deal with the storage of a Huffman coding tree in a file using the bit-based representation.

2.2 A decoding step

The following description is adapted from Section 2.6 of `PE09.pdf`.

Given a compressed (or coded) file, which contains, among other things, the topology information, and a bit stream corresponding to the encoding of the original file, the decompression program must first re-build a Huffman coding tree based on the topology information.

In PA05, you will work on the (re-)building of a Huffman coding tree based on the bit-based representation of the pre-order traversal of the original Huffman coding tree stored in the file. To demonstrate that you have successfully re-constructed the original Huffman coding tree, we require you to store the topology information in a file using the bit-based representation under a **post-order traversal** of the re-constructed Huffman coding tree.

For the "go go gophers" example, the character-based representation of the post-order traversal of the Huffman coding tree is `"1g1o01s1 01e1h01p1r0000"`.

In the bit-based representation, the first 16 bits (or the first 2 bytes **in the convention that left to right is MSB to LSB**) describing the topology are are `11001111 10111110`.

The LSB of the first bytes is a 1-bit. It is followed by the ASCII code for `'g'`, which has a bit pattern of `01100111`. The 7 least significant bits (`1100111`) are in this byte. The most significant bit (`0`) are in the least significant position of the next byte. This is followed by another 1-bit, followed by `101111`, the 6 least significant bits of the ASCII code for `'o'`, which has a bit pattern of `01101111`.

3 Byte to bits and bits to byte

A challenge in PA05 is in the reading of a bit from or writing a bit to a file. Recall that the smallest unit that you can read from/write to a file is a byte. It is important that we use the same order to read a bit from/write a bit to a byte for a pair of compression/decompression programs to work correctly. **The order in which you read a bit from a byte or write a bit to a byte should be from the LSB to the MSB.**

A strategy to read a bit or to write a bit to a file is to always read a byte from or write a byte to the file. You should maintain an index to indicate where you are in that byte. For example, when you first read in a byte, the index should be pointing to the least significant position. After you have read in one bit, the index should be updated to point to the next more significant position. When the index is beyond the most significant position, you have exhausted the entire byte, and the next bit should be from the least significant position of the next byte read.

Similarly, for writing, you should have an byte that contains numeric value 0 to begin with, with the index initially pointing to the least significant position. After you have written in one bit, the index should be updated to point to the next more significant position. When the index is beyond the most significant position, you have filled in a byte, and you should write the filled byte to the file and use an new byte (with an initial numeric value of 0) for the next bit.

You may also have to write an 8-bit ASCII character that straddles two bytes to a file to represent the Huffman coding tree. You have to split the 8-bit ASCII character into two parts. The lower significant part has to complete the partially filled byte into a fully filled byte, which is then written into the file. The higher significant part will form the next partially filled byte (at the lower significant bit positions). For decompression, you have to read an 8-bit ASCII character that straddles two bytes in the compressed file when you want to re-construct the Huffman coding tree.

I assume that you are already familiar with bit-wise operations. If you have questions about bit-wise operations, I refer you to the descriptions of PE07 and PA04.

3.1 Suggested implementation

This is an implementation that you may want to consider. We define a user-defined type as follows:

```
typedef struct _bFILE {
   FILE *fp;
   unsigned char buffer;
   unsigned char bit_index;
   unsigned char mode;
} bFILE;
```

In this structure, the `fp` field stores the stream from/to which you read or write. The `bit_index` field tells you how many valid bits are stored in the field `buffer`. These valid bits occupy bits 0 through `bit_index` -1 if `bit_index` is positive. The `mode` field indicates whether the `fp` is an input stream or an output stream.

The idea is that if you want to read a bit from file, you read an (unsigned char) into the `buffer` field, and use that to supply a bit at a time. You use the `bit_index` to keep track of the position index of the bits so that you can supply the correct bit. Note that this is similar to the file position pointer in the structure `FILE`.

Similarly, if you want to write a bit to a file, you store the new bit in the `buffer` field. You use the `bit_index` field to keep track of the next position in `buffer` at which you save the new bit. When the `buffer` field has eight valid bits, you write it to the file.

These are some suggested functions:

```
bFILE *b_fopen(char *filename, char *mode);

int fgetbit(bFILE *bfp);

int fputbit(int bit, bFILE *bfp);

int b_fclose(bFILE *bfp);
```

b_fopen should be similar to fopen, so that you can allocate space for a bFILE, and use the fp field to store the corresponding stream (for reading or writing). You should also initialize bit_index and buffer to appropriate values. You should also store the mode (read or write) of the file stream.

fgetbit should be similar to fgetc. If there are no valid bits in buffer, fgetbit should read a byte from bfp->fp and update the relevant fields. If there are valid bits in bfp->buffer, the function should return the the least significant valid bit, occupying the least significant bit position of the returned int. Just like fgetc, the function has to update the relevant fields.

fputbit should be similar to fputc. bfp->bit_index provides the bit position at which to fill in the bit in bfp->buffer, and update the relevant fields. If there are sufficient valid bits, the byte should be written to bfp->fp, and the relevant fields should be updated.

b_fclose should be similar to fclose, so that you clean up the necessary stream and free the associated memory. Note that if the file stream has been opened for writing, any partially filled buffer should be flushed (written) to the file stream.

You may want to define more functions. For example, you may want to define functions that read/write 8 bits to/from a file.

4 Program you have to write

For this assignment, you are given the flexibility of designing your .h and .c files. They should be compiled into an executable called pa05. The executable takes in an option, an input file, and produces an output file:

```
./pa05 -e input_file output_file
```

The option "-e" implies that you want to run the encoding option of pa05. It should build a Huffman coding tree based on the input_file, and produce an output_file that contains a bit-based representation of the Huffman coding tree (under pre-order traversal). It is important that the Huffman coding tree is constructed based on the priority queue implementation as specified in PE09.

In the examples subfolder, some sample input files are given: gophers, stone, empty, and binary1 (as in PE09). The corresponding output files are gophers.etree, stone.etree, empty.etree, binary1.etree.

We can also run pa05 with the "-d" option:

```
./pa05 -d input_file output_file
```

The option "-d" implies that you want to run the decoding option of pa05. It should re-build a Huffman coding tree based on an input_file that contains the bit-based representation of the original Huffman coding tree obtained obtained using pre-order traversal. It then produces an output_file that contains a bit-based representation of the Huffman coding tree under **post-order** traversal of the re-constructed tree.

(As the "-d" option is meant for you to earn bonus points, you should simply return EXIT_FAILURE if you do not want to implement functions associated with this option.)

The output files of the encoding option are the input files of the decoding options. The corresponding output files for the decoding option are gophers.dtree, stone.dtree, empty.dtree, and binary1.dtree.

If an invalid option is provided, the program should return EXIT_FAILURE. If the input file cannot be opened, the program should not produce any output files and it should return EXIT_FAILURE. If the input file can be opened for reading, and the corresponding output file can be produced, the program should return EXIT_SUCCESS. Otherwise, the program should return EXIT_FAILURE.

You may assume that when an input file for the "-d" option can be opened for reading, it contains a valid bit-based representation of a Huffman coding tree obtained under pre-order traversal.

Regardless of the exit status of the program, the program should not have any memory issues (as reported by valgrind).

5 Compiling your program

As you have advanced to this stage, we believe that you have the necessary experience to decide what gcc command should be used in your Makefile. We will use the command "make pa05" to create an executable named pa05.

6 Submission

You must submit a zip file called PA05.zip, which may contain any number of files. You must also include in the zip file a Makefile that allows us to generate an executable called pa05 by using the command "make pa05."

Assuming that you are in the folder PA05, which contains all your source files (.c and .h) and the Makefile, use the following command to zip your files:

```
zip PA05.zip *.[ch] Makefile
```

Make sure that you name the zip file as PA05.zip. Moreover, the zip file should not contain any folders. Submit PA05.zip through Brightspace. You may submit as many versions as you like. Brightspace will keep only the most recent submission, and we will grade only the final submission.

7 Grading

You should submit only .c files, .h files, and a Makefile in the zip file PA05.zip. We will use the Makefile to compile and we expect an executable called pa05. The command make pa05 should generate pa05. If we fail to obtain pa05, you do not receive any credit for this assignment.

As mentioned earlier, the executable pa05 expects 3 arguments: 1 option, 1 input filename, and 1 output filename.

You earn points by producing the correct output files. The "-e" option accounts for 100 points, i.e., the full credit of the assignment. Up to 5 points will be deducted if your main function does not behave as decribed in Section 4.

The `"-d"` option accounts for **50 bonus points. You may assume that when an input file for the `"-d"` option can be opened for reading, it contains a valid bit-based representation of a Huffman coding tree obtained under pre-order traversal.**

Even if you choose not to implement functions associated with the `"-d"` option because you are not interested in the bonus points, you should still write the `main` function assuming that the `"-d"` option is available. You should assume that the input file can be read properly and that the output file can be produced properly.

Be aware that we set a time-limit for each test case based on the size of the test case. If your program does not complete its execution before the time limit for a test case, it is deemed to have failed the test case. We will not announce the time-limits that we will use. They will be very reasonable.

The occurrence of any memory issues (memory errors or memory leaks flagged in a `valgrind` report) will result in 50-point penalty.

8 A few points to remember

All debugging messages should be printed to `stderr`. If the program creates an output file when it should not, or the output file is correct, you do not receive credit for that output file.

Grading of programming exercises and assignments is performed on machines with similar setup as eceprog.ecn.purdue.edu. You should perform testing of your work on eceprog.ecn.purdue.edu before submission. Correct output on your computer does not translate into correct output on eceprog.ecn.purdue.edu.