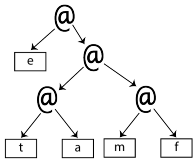
15-122: Principles of Imperative Computation, Spring 2023

Programming Homework 9: Huffman Compression



Due: Thursday 6th April, 2023 by 9pm

In this programming assignment, we will progressively develop a C program to compress (and uncompress) files — something similar to applications like `pkzip`/`pkunzip` or `gzip`/`gunzip` which you may have used. Like these applications, your program will be able to handle inputs as short as a tweet and as long as the collected works of William Shakespeare, and it won't be limited to just text files. Along the way, we will see how trees can play a role in data compression and why they need to be supplemented with other data structures to achieve good performance. We will also take a quick peek at file input/output in C.

Download the assignment handout from the course website or Autolab. The file `README.txt` in the code handout goes over the contents of the handout and explains how to hand the assignment in. There is a TEN (10) PENALTY-FREE HANDIN LIMIT. Every additional handin will incur a small (5%) penalty (even if using a late day). Your score for this assignment will be the score of your last Autolab submission.

Notes:

- Differently from previous assignments, the starter code contains *object-code files* (ending in a `.o` extension) instead of C source files. To compile and run your code, you **must** be logged in on a `unix.andrew.cmu.edu` machine.
- This assignment will not be graded for style. However you will find it helpful to apply the good style habits you have developed: reasonable contracts, at most 80-character lines, and comments that make it clear to a reader how your algorithm works and what invariants you expect to hold. You should use the libraries provided for you to make your code simple and clear. We expect you to write your own helper functions when appropriate. Bad style will have no direct effect on your grade but will make your life harder.
- Be sure to include appropriate `REQUIRES`, `ENSURES`, and `ASSERT` annotations in your code. If you write any auxiliary functions, include precise and appropriate pre- and post-conditions. You should write these as you are writing the code rather than after you're done: documenting your code as you go along will help you reason about what it should be doing, and thus help you write code that is both clearer and more correct.
- This assignment makes use of many types defined in various header files. We suggest that, as you encounter them, you write them down somewhere convenient so that you can quickly refer to their definition.
- Be careful that all the memory you allocated is freed by the end of a normal execution (you do not need to do so for abnormal exits, like when encountering an error condition

or a contract violations). We will use `valgrind` to check for memory leaks and safety violations ... and so should you.

Data Compression: Overview

Whenever we represent data in a computer, we have to choose some sort of *encoding* with which to express it in binary. When representing strings in C0, for instance, we use ASCII codes to represent the individual characters. Other encodings are possible as well. For example, the UNICODE standard defines several character encodings with a variety of different properties. The simplest, UTF-32, uses 32 bits per character.

Under the extended ASCII encoding, each character is represented using 8 bits, so a string of length n requires $8n$ bits of storage. For example, consider the string `"free coffee"`. It is represented as the following $11 \times 8 = 88$ bits in ASCII:

```
01100110  01110010  01100101  01100101  00100000  01100011  01101111
f         r         e         e                   c         o

01100110  01100110  01100101  01100101
f         f         e         e
```

(The spaces around the ASCII code of each letter are shown to ease readability: they are not part of the encoding of this string. Also, `00100000` is the ASCII code of the space character.)

This encoding of the string is rather wasteful, though. In fact, since there are only 6 distinct characters in the string (including the space character), we should be able to represent it using a custom encoding that uses only $\lceil \log 6 \rceil = 3$ bits to represent each character. If we were to use the following custom encoding:

Character	Code
`'c'`	`000`
`'e'`	`001`
`'f'`	`010`
`'o'`	`011`
`'r'`	`100`
`' '`	`101`

the string would be represented with only $11 \times 3 = 33$ bits:

```
010  100  001  001  101  000  011  010  010  001  001
f    r    e    e         c    o    f    f    e    e
```

By using this custom encoding, we have saved $(88 - 33)/88 = 62\%$ of the space used by the standard ASCII representation of the above string.[1]

Can we do even better? Our custom encoding uses the same number of bits for each character — it is a *fixed-length encoding*. But in our input string, the letter `e` occurs much

[1] Well, almost: we would also need to store the encoding itself so that we can recover the string — more on this later.

more frequently than `c` for instance. It may be worthwhile to use a smaller bit pattern to encode the character `e` even at the expense of having to use longer bit patterns to encode `c`. Our next encoding — a *variable-length encoding* — embraces this idea:

Character	Code
`'c'`	`1101`
`'e'`	`0`
`'f'`	`10`
`'o'`	`1110`
`'r'`	`1111`
`' '`	`1100`

Using it, the string `"free coffee"` is represented as follows:

```
10  1111  0  0  1100  1101  1110  10  10  0  0
f   r     e  e        c     o     f   f   e  e
```

That's just 26 bits, for a $(88 - 26)/88 = 70\%$ space saving.

For a variable-length encoding to be viable, there needs to be a simple way to go from a bit string like the above to the original text. One such way is for the encoding to be *prefix-free*: no code word is a prefix of any other code word. Both our encodings are prefix-free: in the last one, for example, if a bit string starts with `10`, the first character of the corresponding text must be `f` because no other character encoding starts with `10`.

It can be proven that this encoding is optimal for this particular string: no other prefix-free encoding can represent the string using fewer than 26 bits. Moreover, the encoding is optimal for *any* string that has the same distribution of characters as the sample text. In this assignment, you will implement a method for constructing such optimal encodings developed by David Huffman.

Huffman Coding: A Brief History

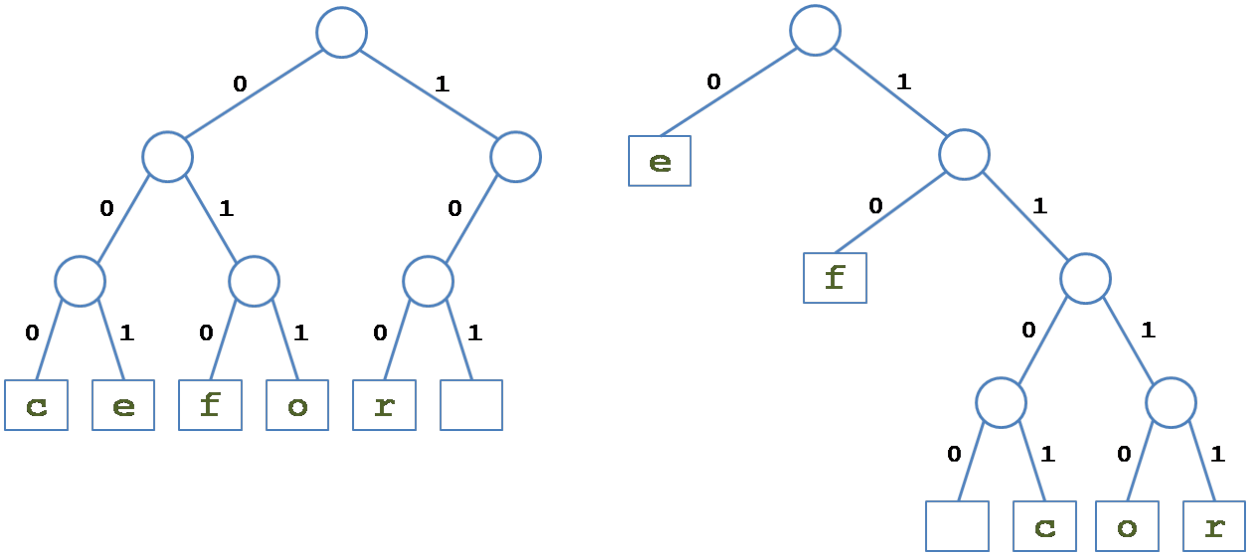
Huffman coding is an algorithm for constructing optimal prefix-free encodings given a frequency distribution over characters. It was developed in 1951 by David Huffman when he was a Ph.D student at MIT taking a course on information theory taught by Robert Fano. It was towards the end of the semester, and Fano had given his students a choice: they could either take a final exam to demonstrate mastery of the material, or they could write a term paper on something pertinent to information theory. Fano suggested a number of possible topics, one of which was efficient binary encodings: while Fano himself had worked on the subject with his colleague Claude Shannon, it was not known at the time how to efficiently construct optimal encodings.

Huffman struggled for some time to make headway on the problem and was about to give up and start studying for the final when he hit upon a key insight and invented the algorithm that bears his name, thus outdoing his professor, making history, and attaining an "A" for the course. Today, Huffman coding enjoys a variety of applications: it is used as part of the DEFLATE algorithm for producing ZIP files and as part of several multimedia codecs like JPEG and MP3.

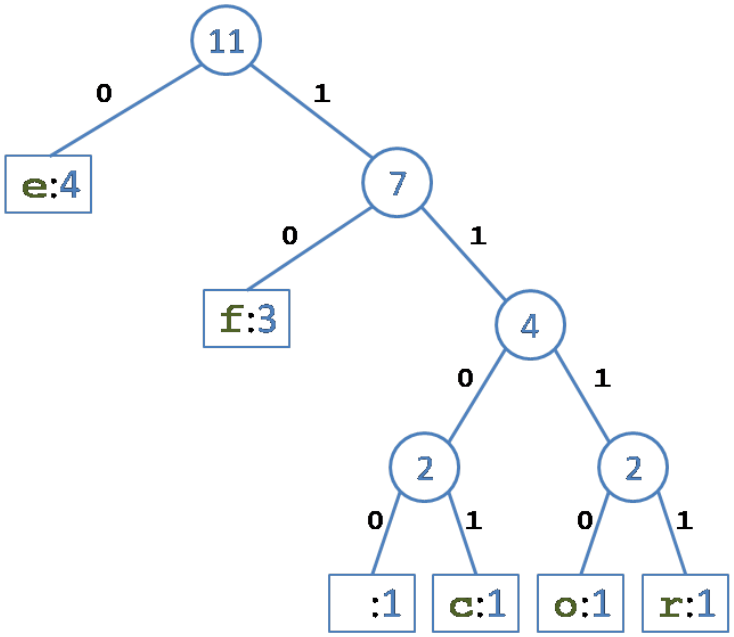
1 Huffman Trees

Prefix-free encodings — where no code word is a prefix of any other code word — can be represented as binary trees with characters stored at the leaves: a branch to the left corresponds to a 0 bit and a branch to the right corresponds to a 1 bit, so that the path from the root to a leaf gives the code word for the character stored at that leaf.

For instance, the fixed-length and variable-length encodings shown earlier — both being prefix-free — are represented by the following two binary trees, respectively.



In the tree on the right, which corresponds to our variable-length encoding, frequently-occurring characters have shorter paths from the root. We can see this property clearly if we label each subtree with the total frequency of the characters occurring at its leaves. The frequencies in the following tree are based on the sample string `"free coffee"`. A frequency-annotated tree is called a *Huffman tree*.



Huffman trees have a recursive structure: a Huffman tree is either a leaf containing a character and its frequency, or an interior node containing the combined frequency of two child Huffman trees. We draw the leaves, which contain character data, as rectangles to distinguish them from the interior nodes, which we draw as circles.

We represent both kinds of Huffman tree nodes in C using a **struct** htree_node, abbreviated as the type htree:

```
typedef struct htree_node htree;
struct htree_node {
  symbol_t value;
  unsigned int frequency;
  htree *left;
  htree *right;
};
```

The value field of a leaf contains a character and is irrelevant for interior nodes. Interior nodes should have exactly two children. In view of generalizing our encoding from strings of printable characters to arbitrary data, we draw value from the type symbol_t of *symbols*. symbol_t is an unsigned integer type, making it suitable to index arrays by symbols. Within this type, the symbols we may want to represent are in the range $[0, \texttt{NUM\_SYMBOLS})$.

The well-formedness criteria of Huffman trees give rise to the following recursive definitions:

- An htree is a *valid htree* if it is non-NULL and it is either a *valid htree leaf* or a *valid htree interior node*.
- An htree is a *valid htree leaf* if its frequency is strictly positive, and left and right children are NULL.
- An htree is a *valid htree interior node* if its left and right children are *valid htrees*, and its frequency is the sum of the frequency of its children.

Task 1 (2 points) In file huffman.c, implement the following functions that formalize the Huffman tree data structure invariants:

Function:	**Returns true iff...**
`bool is_htree_leaf(htree *H);`	the node is a leaf
`bool is_htree_interior(htree *H);`	the node is an interior node
`bool is_htree(htree *H);`	the tree is a Huffman tree

You may test your code by hand-building various htree's in file test-htree.c. Compile your work with

```
% make htree
```

and then run your tests with

```
% ./test-htree
```

The next tasks will further exercise your specification functions.

2 Constructing Huffman Trees

Huffman's key insight was to use the frequencies of symbols to build an optimal encoding tree from the bottom up. Given a set of symbols and their associated frequencies, we can build an optimal Huffman tree as follows:

1. Construct leaf Huffman trees for each symbol/frequency pair.
2. Repeatedly choose two minimum-frequency Huffman trees and join them together into a new Huffman tree whose frequency is the sum of their frequencies.
3. When only one Huffman tree remains, it represents an optimal encoding.

This is an example of a *greedy algorithm* since it makes locally optimal choices that nevertheless yield a globally optimal result at the end of the day. Selection of a minimum-frequency tree in step 2 can be accomplished using a *priority queue*.

A *priority queue* is a queue-like data structure where elements are retrieved based on their *priority*: higher priority elements are removed first. The priority of an element is determined by a *priority function*, of type `has_higher_priority_fn` defined as follows:

```
// f(x,y) returns true if e1 is STRICTLY higher priority than e2
typedef bool has_higher_priority_fn(elem e1, elem e2)
        /*@requires e1 != NULL && e2 != NULL; @*/ ;
```

When creating a priority queue, a priority function is given which determines whether an element has higher priority than another. An interface to generic priority queues can be found in `lib/pq.h`.

A sample run of the algorithm is shown on page 7. Note that this isn't the only Huffman tree we could've constructed from these frequencies — whenever two frequencies were the same we broke the tie arbitrarily. Likewise, which is the left child and which is the right can be chosen arbitrarily.

Task 2 (5 points) In file `huffman.c`, write a function

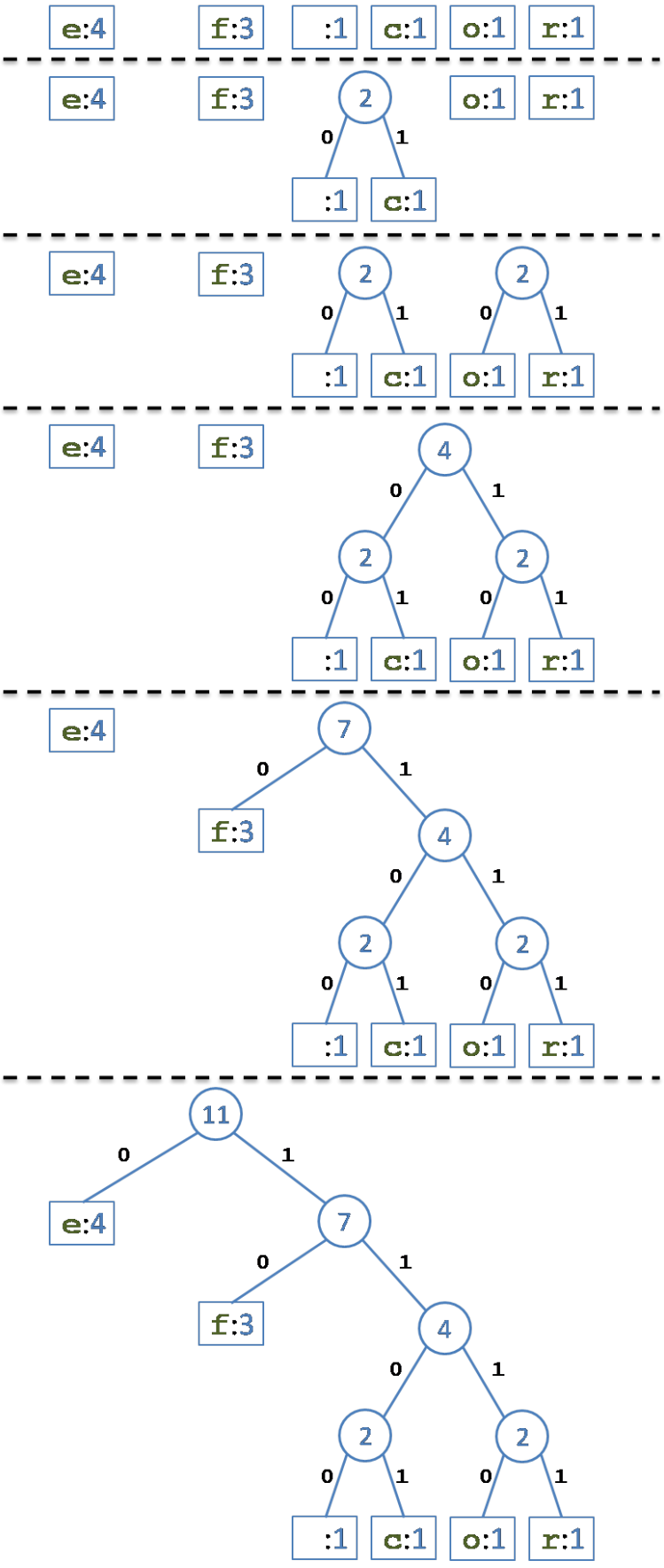
```
htree* build_htree(freqtable_t table);
```

that constructs an optimal encoding for an alphabet with `NUM_SYMBOLS` symbols using Huffman's algorithm. The *frequency table* `table`, of type `freqtable_t`, is an unsigned integer array of length `NUM_SYMBOLS`. The entry `table[c]` contains the reported frequency of symbol `c`. Recall that, in C, a **`char`** is a one-byte integer type that represents the ASCII value of the character it corresponds to. This means that we can use characters as indices in an array (for example, `table['a']` is the same as `table[97]` since `a` has ASCII value `97`). The entry `table[c]` may be 0 if `c` is not expected to occur in a text (for example if `c` is a non-printable ASCII character and we are only interested in encoding text files). See file `freqtable.h`.

Use the code in the included `lib/pq.h` as your implementation of priority queues.

Two observations:

- Huffman trees for a frequency table with just one symbol are not particularly useful, because the optimal encoding of a single symbol would be the empty bit string. But



the source strings, all of which must just repeat the same symbol, would be mapped to the same empty string. Therefore, `build_htree` must signal an error (by calling the **`error`** function) if there are fewer than two symbols with non-zero frequency, and otherwise return an interior node as a result.

- Huffman trees are not unique due to symmetries, additionally complicated by the fact that multiple symbols or subtrees might have identical frequencies. All possible valid Huffman codes for a given text will have the same length (due to its optimality guarantee) but may otherwise be different. So the particular codes produced in your implementation may look different from the ones shown in this writeup.

The directory `data/freq` contains a number of frequency files, which have a textual representation format

```
<symbol 1>:<frequency 1>
<symbol 2>:<frequency 2>
...
```

where the symbol is separated from the frequency by a colon `':'`. Each `<symbol n>` is either a printable ASCII character (like `A`) or an hexadecimal number of the form `0x`*nn* (like `0x41`). By convention, frequency files have a `.frq` extension. Take a look the frequency file for our ongoing example at `data/freq/free_coffee.frq`.

You can test your implementation by compiling your code with

```
% make
```

and then running

```
% ./huff-safe -f <freq_file> -R
```

The command-line flag `-R` instructs `huff-safe` to print the `htree` it has built from frequency file `<freq_file>`. Use the additional flag `-Q` to also display the frequency table in `<freq_file>`.

For instance, to test your code with our ongoing example, you would type:

```
% ./huff-safe -f data/freq/free_coffee.frq -R
```

or, if you want to display the frequency table,

```
% ./huff-safe -f data/freq/free_coffee.frq -R -Q
```

3 Decoding Bit Strings

The Huffman encoding of a text using a Huffman tree that accounts for all of the symbols in it is a bit string that replaces each symbol with its Huffman code. We will see how to efficiently carry out such encoding shortly. One thing we can do right away is decode a bit string using the very Huffman tree that was used to encode it.

Given an encoded bit string and the Huffman tree that was used to encode it, we decode the bit string as follows:

1. Initialize a pointer to the root of the Huffman tree.
2. Repeatedly read bits from the bit string and update the pointer: when you read a 0, follow the left branch, and when you read a 1, follow the right branch.
3. Whenever you reach a leaf, output the symbol at that leaf and reset the pointer to the root of the Huffman tree.

If the bit string was properly encoded, then when all of the bits are exhausted, the pointer should once again point to the root of the Huffman tree. If this is not the case, then the decoding fails for the given input.

As an example, we can use our on-going encoding to decode the following message:

```
110111101010001100101110111111001011110 0
   c    o f fee      f    o    r      f    ree
```

Task 3 (8 points) In file `huffman.c`, implement the decode function

```
symbol_t* decode_src(htree *H, bit_t *code, size_t *src_len);
```

This function takes in a bit string `code` (details below) and the Huffman tree `H` to decode it with. It returns an array of symbols decoded from `code` using `H`. The last argument, the pointer `src_len`, is used by the function to communicate to the caller the length of the returned array. Of the several ways to approach this function, we suggest you do two passes on the input, one pass to determine the length of the array to return, and a second pass to populate it.

The function should return the decoded bit string if the string can be decoded and should signal an error otherwise, using the **`error`** function.

For your convenience, the bit string `code` is a (`NUL`-terminated) C string of the ASCII characters `'0'` and `'1'` — not a great choice for file compression but good enough for the time being. We call this representation of bit strings "binascii".

After compiling your code using `make`, you can test it by running

```
% ./huff-safe -D -a <binascii_file> -r <htree_file>
```

This asks `huff-safe` to use your `src_decode` function to decode the binascii string in file `<binascii_file>` using the Huffman tree in file `<htree_file>`. The decoded text will be printed on the terminal, together with some statistics. The handout directory `data/binascii` contains sample binascii files with extension `.01`. Huffman tree files have the format

```
<symbol 1>:<Huffman code 1>
<symbol 2>:<Huffman code 2>
...
```

You can find a few Huffman tree files, with extension `.htr`, in directory `data/htree` (and you can make your own — but make sure that they correspond to valid Huffman trees!). As usual, you can use the flags `-Q` and `-R` to display the frequency table and the Huffman tree if you wish. Additionally, the `-V` flag will display the encoded and decoded texts in lockstep for ease of debugging.

Thus, to test your `src_decode` on our ongoing example, you would run

```
% ./huff-safe -D -a data/binascii/free_coffee.01 -r data/htree/free_coffee.htr
```

which should result in the string `"free coffee"` being printed on the terminal.

4 Encoding Strings

To encode a text using a Huffman tree appropriate for it, we need to replace each symbol in it with its Huffman code. One way to do this is to traverse the entire tree for each symbol in the text: for a tree containing n symbols, that's $O(n)$ for each traversal — check it out — so that encoding a text of length m would take $O(nm)$.

A smarter way to proceed is to build an auxiliary data structure that allows us to retrieve the Huffman code of each symbol in constant time — like the table on page 3. We use an array with `NUM_SYMBOLS` entries, each containing the `NUL`-terminated bit string code of one of the symbols (or `NULL` if this symbol is not in our Huffman tree). A single traversal of the Huffman tree is sufficient to populate this *code table*, and from then on a constant-time array access gives us the Huffman encoding of each input symbol. The price we pay for this gain in efficiency is extra memory to store the code table — typically a worthy trade-off.

Task 4 (3 points) In file `huffman.c`, write the function

```
codetable_t htree_to_codetable(htree *H);
```

mapping Huffman trees to their code tables. *We recommend a recursive approach with heap-allocated strings. Feel free to use the built-in* ***<string.h>*** *library — google it.* The type `codetable_t`, defined in file `htree.h`, is an array of (`NUL`-terminated) `bit_t*`. This array has size `NUM_SYMBOLS`.

You can test your code by running

```
% ./huff-safe -f <freq_file> -T
```

where the flag `-T` prints the code table of the Huffman tree constructed from frequency file `<freq_file>`. Doing so on our ongoing example takes the form

```
% ./huff-safe -f data/freq/free_coffee.frq -T
```

Finally, we are ready to write the encoding function.

Task 5 (3 points) In the same file, write a function

```
bit_t* encode_src(codetable_t table, symbol_t *src, size_t src_len);
```

that efficiently encodes `src` of length `src_len` using code table `table`. This function returns the resulting binascii bit string (recall that these are `NULL`-terminated). Again, you may want to consider a two-pass implementation.

The function should return the encoded bit string if the string can be encoded and should signal an error otherwise, using the **`error`** function.

You can test your code by running

```
% ./huff-safe -E -s <source_file> -f <freq_file>
```

This will use your `encode_src` to return the binascii encoding the file `<source_file>` using frequency file `<freq_file>`, together with some statistics. The frequency file corresponding to sample texts in directory `data/source` have a similar name with a `.frq` extension in directory `data/freq`. Supplying the usual `-Q`, `-R`, `-T`, and/or `-V` flags will provide additional information were you to need it. Doing this on our ongoing example takes the form

```
% ./huff-safe -E -s data/source/free_coffee.txt -f data/freq/free_coffee.frq
```

You can save the encoded text by supplying a `-a <binascii_file>`. If you do so, the invocation

```
% ./huff-safe -D -a <binascii_file> -f <freq_file>
```

should give you back the contents of `<source_file>`. Try that!

5 Basic Input/Output in C

Up to now, symbol frequencies are independent from the source files they are used to encode. To compress a file the way `pkzip` does it, we need to draw the symbol frequencies from this very file. This task gives us an opportunity to learn a bit about file input/output in C. In truth, we will be just scratching the surface of this far-ranging topic, but we need to start somewhere.

In a C program, we read and write to a file via a *file descriptor*, a pointer of type `FILE*`. We obtain a file descriptor by calling the function `fopen(fname, mode)` where `fname` is the path of the file we want to work with, and `mode` is a string with which we tell `fopen` whether we want to read from (`"r"`) or write to (`"w"`) this file. Once we are done using the file, we need to close its descriptor using the function `fclose(descr)`.

The function `fgetc(descr)` reads the next character from the file with descriptor `descr`. But there may be no next character if we have reached the end of the file! In that case it will return the special value `EOF`.

As a summary, here are the (slightly simplified) prototypes of the above functions:

```
FILE *fopen(char *fname, char *mode);
void fclose(FILE *desc);
int fgetc(FILE *descr);
```

You can read more about file I/O in C by researching the library `<stdio.h>`.

Task 6 (1 point) In file `huffman.c`, write the function

```
freqtable_t build_freqtable(char *fname);
```

which returns the frequency table of the symbols in file `fname`.

You can test this function by running

```
% ./huff-safe -F -s <source_file>
```

For example,

```
% ./huff-safe -F -s data/source/free_coffee.txt
```

In fact, you can now test your `encode_src` by running

```
% ./huff-safe -E -s <source_file>
```

It will use your `build_freqtable` to compute the frequency table of `<source_file>` rather than reading it from a frequency file. This is done as follows for our ongoing example:

```
% ./huff-safe -E -s data/source/free_coffee.txt
```

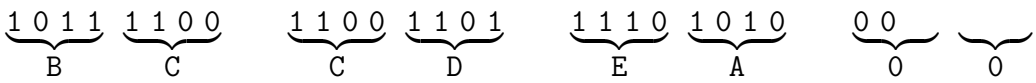
6 Packing Bits into Bytes

Our goal for this assignment is to perform actual file compression, but binascii uses a whole byte to represent each bit. Your next task will be to compress a binascii bit string into actual binary.

Task 7 (2 points) In file `huffman.c`, implement the functions

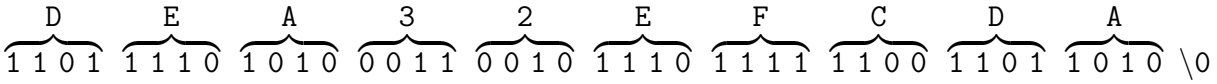
```
uint8_t* pack(bit_t *bits);
bit_t* unpack(uint8_t *c, size_t len);
```

`pack(bits)` returns an array of bytes (type `uint8_t`) where each `'0'` in `bits` is shrunk into a 0-bit and each `'1'` into a 1-bit. The first digit in `bits` will be the most significant bit of the first byte of the result. If the size of `bits` is not divisible by 8, the last byte is padded with 0-bits. For example, `pack` returns the 4-byte array `BCCDEA00` (here in hexadecimal) on our binascii encoding of `"free coffee"`:



where the last six bits have been padded with zeros.

Conversely, `unpack(bytes, len)` converts each bit in the byte array `bytes` of length `len` into a `NUL`-terminated binascii string. For example, calling `unpack` on the byte array `DEA32EFCDA` (written in hex) of length 5 yields:



where the rightmost symbol in the output is the `NUL` terminator.

You may find it useful to write helper functions that pack/unpack a single byte. Also, the function `num_padded_bytes`, described in file `bitpacking.h`, may come handy.

We encourage you to write standalone tests for `pack` and `unpack`, similarly to what you did for `is_htree` in task 1. The starter file is `test-pack.c` and the compilation command is `make pack`.

You can test your code thoroughly by performing actual compression and uncompression on files:

```
% ./huff-safe -C -s <source_file> -h <compressed_file>
```

uses the code you have developed in this assignment to compress `<source_file>` writing the result into `<compressed_file>`. This file contains both a code table extracted from `<source_file>` and the packed Huffman encoding of its contents. The compression ratio, i.e., the space savings, will be printed to terminal. You can supply the flags we have encountered so far to display various information. Thus, to compress our ongoing example into the file `my_first_compression.hip`, you would type

```
% ./huff-safe -C -s data/source/free_coffee.txt -h my_first_compression.hip
```

You can use any file you want as the source file, but if you try it out on very large files, you will want to compile your code with `make fast` and use `huff-fast` instead of `huff-safe`. Do so only after you are sure your code works properly as it disables all annotations!

You can compress binary files, for example images, but doing so may not save all that much space — you may even get a negative compression ratio since the compressed file embeds the code table of your source file!

Now the real test. You can uncompress a compressed file by running

```
% ./huff-safe -U -h <compressed_file>
```

to display the corresponding source to terminal. So, the call

```
% ./huff-safe -U -h my_first_compression.hip
```

should print `"free coffee"` together with some statistics.

Add the argument `-s <file>` to dump the result to `<file>` instead, which will be helpful if the original source was very large or binary. The usual flags are available for your perusal.

For testing purposes, we provide a number of compressed files (with extension `.hip`) in directory `data/compressed`. Uncompress them and check that they are identical to the corresponding source files in directory `data/source` by using the Unix `diff` utility.

7 The Ultimate Test

How much confidence do you have in your implementation? Are you willing to submit your work in *compressed* form? That's exactly what you will need to do to get the last point of this assignment.

Task 8 (1 point) Compress your submission, file `huffman.c`, into a file named `huffman.c.hip` and submit this file to Autolab.

We will use our code to decompress it and then grade what comes out. If decompression fails, or the result are not a valid C file, you will get a 0 for the whole assignment. Don't panic though: if you tested your code vigorously for all prior tasks and it worked as expected, the odds of this happening are *very low*. Moreover, you can always look at the Autolab output and make a regular submission if it reports a failed decompression.