CSC 230 Project 5

Huffman Coding

For this Project, you will write a pair of programs that work together, one called `encode` and the other called `decode`. The first program uses Huffman coding to "encode" a file, and in so doing compress it. "Decoding" the encoded file converts it back into the original file. The Huffman coding algorithm creates a bit string to represent each unique character in a document based on its frequency, with shorter strings used to encode characters that occur more frequently. These bit strings are "prefix codes" in that the bit string used to encode a given character is not used as the first part of the bit string for any other character.

For example, if the letters, `a`, `c`, and `t`, are represented by the bit strings, `0000`, `00101`, and `1111`, respectively, the word `act` can be encoded as `0000001011111`. This encoding would only require 2 bytes instead of the 3 bytes required when using ASCII characters. Decoding the string involves matching the sequences of bits in the string to the codes for the letters. Starting at the beginning of the bit string, the sequences, `0`, `00`, and `000` do not match any letter, but `0000` matches an `a`. Likewise, the following bit sequences `0`, `00`, `001`, and `0010` do not match any letter, but `00101` matches a `c`. Finally, the last sequence `1111` matches a `t`.

You will use the programs as follows, where codes-1.txt is a list of prefix codes for each letter of the alphabet as well as space, newline, and eof, input-1.txt is a file to be encoded, and encoded-1.bin is the encoded (binary) version of the file.

```
# encode input-1.txt using codes-1.txt
$ ./encode codes-1.txt input-1.txt encoded-1.bin
# decode, to get back a copy of the original.
$ ./decode codes-1.txt encoded-1.bin output.txt
# See if it looks exactly like the original
$ diff output.txt input-1.txt
```

As with recent assignments, you'll be developing this project using git for revision control. You should be able to just unpack the starter into the p5 directory of your cloned repo to get started. See the Getting Started section for instructions.

This Project supports a number of our course objectives. See the Learning Outcomes section for a list.

Rules for Project 5

You get to complete this project individually. If you're unsure what's permitted, you can have a look at the academic integrity guidelines in the course syllabus.

In the design section, you'll see some instructions for how your implementation is expected to work. Be sure you follow these rules. It's not enough to just turn in a working program; your program has to follow the design constraints we've asked you to follow. For this project, we're putting some constraints on the functions you'll need to define, and on one data structure you'll need to use. Still, you will have lots of opportunities to

design parts of the project for yourself.

Requirements

This section says what your programs are supposed to be able to do, and what they should do when something goes wrong.

Running the encode / decode programs

The `encode` program and the `decode` program both take three command-line arguments, a file with prefix codes, an input file, and an output file.

Running `encode`, as follows will get it to read input from the file `input.txt` and use codes.txt to create and output an encoded representation to the file `encoded.bin`.

```
./encode codes.txt input.txt encoded.bin
```

If you give it the wrong number of command-line arguments, it should print this usage message to standard error and exit with a status of 1:

```
usage: encode <codes-file> <infile> <outfile>
```

If the codes-file contains any symbols other than the lowercase letters, "space", "newline", and "eof", or codes that are longer that 12 characters or contain characters other than 0 and 1, or does not contain a code for each lowercase letter, "space", "newline", and "eof", it should print this error message to standard error and exit with a status of 1.

```
Invalid code file
```

If the input file contains any characters other than lowercase letters, spaces, and newlines, it should print this error message to standard error and exit with a status of 1.

```
Invalid input file
```

Running `decode`, as follows will get it to read an encoded representation from the file `encoded.bin`, decode it using codes.txt, and output the result to `output.txt`.

```
./decode codes.txt encoded.bin output.txt
```

If it's successful, running the two programs like this should produce an output file, `output.txt` that exactly matches the original input, `input.txt`.

If you run `decode` with the wrong number of command-line arguments, it will print the following usage message to standard error and exit with a status of 1:

```
usage: decode <codes-file> <infile> <outfile>
```

If the codes-file contains any symbols other than the lowercase letters, "space", "newline", and "eof", or codes that are longer that 12 characters or contain characters other than 0 and 1, or does not contain a code for each lowercase letter, "space", "newline", and "eof", it should print this error message to standard error and exit with a status of 1.

If the encoded input file contains anything other than a sequence of valid prefix codes, as described in the next section, followed by zero or more 0 bits, it should print this error message to standard error and exit with a status of 1.

```
Invalid input file
```

If either program can't open one of the given files (e.g., `file.txt`), it will report an error message like the following to standard error. Here, this is the error message reported by `perror()`, given the filename that couldn't be opened, so it may vary depending on what went wrong. After printing the error message, the program should exit with a status of 1:

```
file.txt: No such file or directory
```

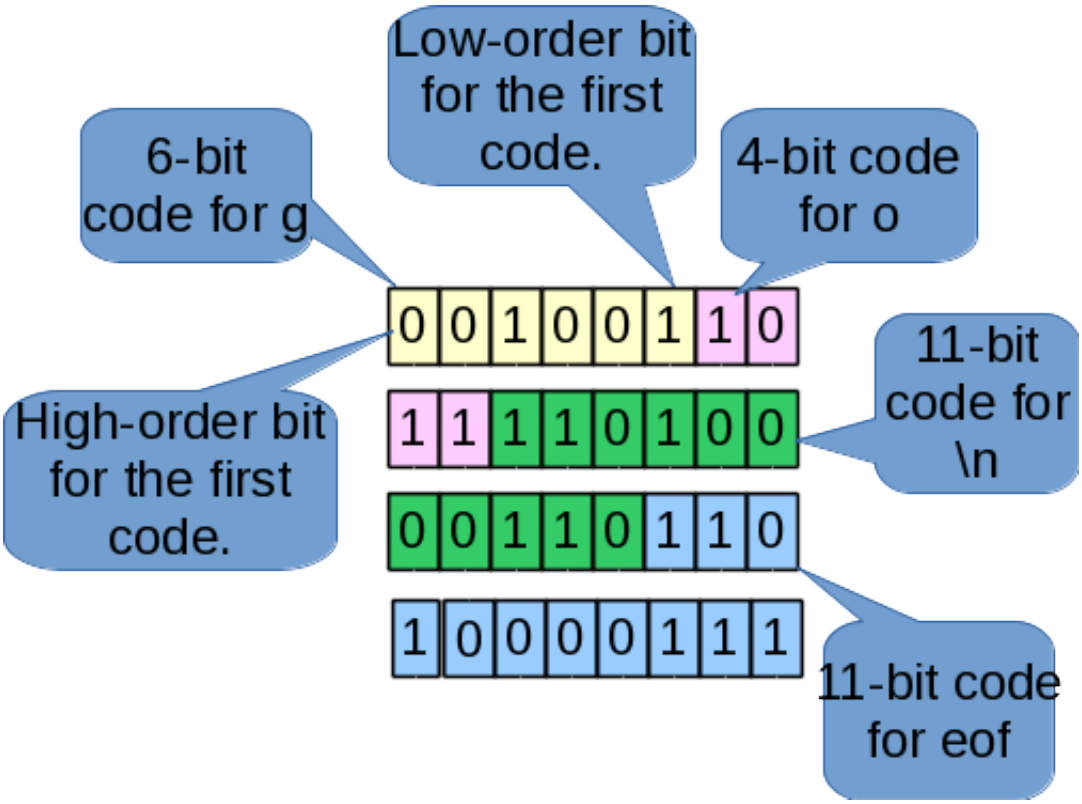
Symbol / Byte Encoding

The `encode` program will encode a file consisting of lowercase letters, spaces, and newlines by representing each symbol (character) with a binary prefix code based on Huffman coding. Two different files containing prefix codes for each letter of the alphabet as well as space, newline, and eof are provided for you, codes-1.txt and codes-2.txt. It may be interesting to see which set of codes creates a smaller encoded version of a given file. These files are based on the tables found at http://slideplayer.com/slide/6184217/18/images/52/Huffman+code+for+English+alphabet.jpg and http://www.yorku.ca/mack/uist2011-f2.jpg, respectively.

Encoded File Representation

An encoded file should consist of codes for each character in the input file with a special code to mark the end of the file. If the last byte of the encoded file is not full, the unused bits should be filled with 0's.

The following figure shows what test input, `input-1.txt`, would look like when encoded. The `input-1.txt` file contains the symbol `g`, followed by `o`, followed by a newline. Each row in this figure shows all 8 bits in a byte of the encoded file, with the high-order bit on the left (like you'd normally expect) and the low-order one on the right. The encoded file contains the code 6-bit code, `001001`, for `g`, followed by the 4-bit code, `1011`, for `o`, followed by the 11-bit code, `11010000110`, for the newline, followed by the 11-bit code, `11010000111`, for eof. This encoding fits perfectly into 4 bytes. If there had been any unused bits, they would be filled with 0s.



Encoded representation for input-1.txt

In this case, all of these codes fit in a 4-byte output file. The code for `g` is stored in the 6 high-order bits of the first byte, leaving 2 bits to store the start of the code for the `o`. The remaining 2 bits of the code for `o` are stored in the 2 high-order bits of the next byte. Then, the remaining 6 bits are used to store the first part of the 11-bit code for the newline. The remaining 5 bits of the code for the newline go in the 5 high-order bits of the next byte which leave 3 bits for the first part of the 11-bit code for eof. The remaining 8 bits of the code for eof fill the next byte.

Like this example shows, bits in the output file will be used starting with the high-order bits and going toward the low-order bits. The highest-order bits of a code will be written first, going in the next available bit in the current byte of the file. Once all the bits of a byte are used, we'll start using bits of the next byte, starting with the high-order ones. This may leave some lower-order bits in the last byte unused. You'll just fill these with zeros.

Design

Binary I/O

The encode program will read a text file and create a binary file while the decode program will read a binary file and create a text file. Therefore, be sure you open the encode output stream and the decode input stream in binary mode.

Program Organization

Your source code will consist of four implementation files and two headers.

- `codes.c` / `codes.h`
  This component will contain a data structure for representing the contents of the code file, along with functions to convert between symbols and the variable-length binary codes used to represent them.

- `bits.c` / `bits.h`
  This component will define functions for writing out bit sequences to and from a file. With the help of these functions, it will be easy for the main program to read and write the bit codes needed for encoding/decoding. Some of the header for this component has already been written for you in the starter. You can add more functions to this component if you want to.

- `encode.c`
  This component will implement the main function of the encode program. It will be responsible for handling the command-line arguments, reading characters from the input file, using the `codes` component to convert them to codes, and using the `bits` component to write them out to the output file.

- `decode.c`
  This component will implement the main function of the decode program. It will be responsible for handling the command-line arguments, using the `bits` component to read codes from the input file, using the `codes` component to convert them back to characters and writing those to the output file.

As part of your implementation, you will need to define and use certain functions we're expecting. You'll proably want to define some additional functions, to help simplify and organize your implementation.

- `const char * symToCode( int ch );`
  Given the ASCII code for a character or EOF (-1), this function returns a string containing the code used to represent it. For example, if you are using the `codes-1.txt` code file, `symToCode( 'a' )` should return "0000" and `symToCode(EOF)` should return "11010000111". The string returned by this function can't be changed by the caller, and it doesn't need to be freed by the caller (see Code Representation below). If there's no code to represent the given character, this function returns NULL.

- `int codeToSym( const char *code );`
  Given a string containing a code, this function returns the ASCII character it represents or EOF (-1). For example, if you are using `codes-1.txt`, `codeToSym( "0000" )` should return 'a' and codeToSym("11010000111") should return EOF. If the code does not represent a character, this function returns `-2`.

The bits.h header is already written for you, but you can add more functions if you want. There's some additional description of it in the next section. It will implement at least the following three functions.

- `void writeBits( const char *code, BitBuffer *buffer, FILE *fp );`
  Write the code stored in the code parameter. Temporarily store bits in the given buffer until we have 8 of them to write, then write the resulting byte to the given file.

- `void flushBits( BitBuffer *buffer, FILE *fp );`
  If there are any bits buffered in buffer, write them out to the given file in the high-order bit positions of the next byte, leaving zeros in the low-order bits. When you're done writing out codes, you'll need to call this function to make sure any bits in the last, partially full byte get written out.

- `int readBit( BitBuffer *buffer, FILE *fp );`
  Read and return the next bit (0 or 1) from the given file or -1 if the end of file has been reached. The given buffer may contain some bits left over from the last read, and if this read has any left-over bits, it should leave them in that buffer.

Code Representation

In your codes component, you will need to define how you want to represent the contents of the code file. You'll need to store all these codes in some kind of data structure in memory, so you can quickly find the binary representation for a character, or the character that corresponds to a given sequence of bits. You get to choose how to represent this data structure. You can either store it statically, or dynamically allocate it.

Your codes component will need a function to read the contents of the code file at program start-up, and if you dynamically allocate the data structure you use to represent codes, you'll need a function to free this data structure at program termination.

In your data structure, you'll need to store a string of 1 and 0 characters for each code. Whenver the `symToCode()` function is called, you'll return a pointer to the string for that code. This is an efficient way to give the code back to the caller, letting the caller use part of the data structure instead of making a copy every time we need a code. Since you're letting the caller see part of your data structure, we marked the return value as const, so client code won't try to change what's in the string. Also, since you're just returning a pointer to an existing string, you don't have to allocate space or copy the string before you return it, or worry about freeing the string when the caller is done with it.

Bit Buffering

File I/O operations will let you read one or more bytes at a time, but they won't let you read or write individual bits one at a time. To write out bit sequences that don't start and end on byte boundaries, we'll need to temporarily buffer some bits until we have 8 of them (a byte) that we can write out. The same will be true for reading. We have to read a whole byte at a time, but if we only need some bits from that byte, we'll keep the rest buffered until we're ready for them.

In the `bits.h` header, the starter includes a definition of a simple structure you'll use for buffering bits during reading and writing (along with prototypes for the functions that use it). It's called BitBuffer and it has just two fields, one for temporarily storing up to 8 bits and another for keeping up with how many bits are buffered.

```
typedef struct {
  /** Storage for up to 8 bits left over from an earlier read or waiting
      to be written in a subsequent write. */
  unsigned char bits;

  /** Number of bits currently buffered. */
  int bcount;
} BitBuffer;
```

You'll need to use a BitBuffer while reading or writing an encoded file, to temporarily store a number of bits that doesn't constitute a whole byte. For example, if the first code requires 5 bits, you'll have 3 bits that you

don't need yet. You will store the first part (or all) of the next code in the remaining 3 bits, and then write the byte to the binary output file.

You have some freedom in how you use the BitBuffer during reading and writing. For example, you could keep buffered bits in the high-order bits of the `bits` field, shifting everything to the left as you remove bits from the buffer. Or, you could keep every bit in the position it started in when you read it, and use a movable mask to look at individual bits when you need them. You could even manage the buffer differently for reading and writing; whatever is easiest for you to understand and implement.

Build Automation

You get to implement your own Makefile for this project (called `Makefile` with a capital 'M', no filename extension). Its default target should build both your encode and decode programs, compiling each source file to an object file and then linking the objects together into an executable.

As usual, your Makefile should correctly describe the project's dependencies, so targets can be rebuilt selectively, based on what parts of the project have changed. It should compile source files with our usual command-line options, including `-g` for help with debugging. It should also have a clean rule, to let the user discard temporary files as needed.

Testing

The starter includes a test script, along with test input files and expected outputs. When we grade your program, we'll test it with this script, along with a few other test inputs we're not giving you. To run the automated test script, you should be able to enter the following:

```
$ chmod +x test.sh # probably just need to do this once
$ ./test.sh
```

This will automatically build your program and see how it does against all the tests.

As you develop and debug your programs, you'll want to be able to run them yourself, to see what they're doing and maybe try them out inside the debugger. As you run the test script, you'll see it reports on how it's running your program for each test. You can copy this command to run your program directly to get a better idea of how it's behaving.

Binary File Report

To help debug, I'm providing source code for a simple program that prints out the contents of a file in binary. This may help you to see if the bit sequences you're trying to write to a file are really making it there. It's called dumpbits.c. You can compile it like any C program, then redirect standard input from a file to get it to printout the contents of any file in binary. For example, if you run it as follows, it will show the contents of the encoded representation for test case 1 using codes-1.txt:

```
csc$ ./dumpbits < encoded-1.bin
```

```
0000 00100110
0001 11110100
0002 00110110
0003 10000111
```

Be careful about running this program on a windows system. I'm worried that, since it's reading from standard input, it will give you text stream behavior on the input, rather than binary. It should work fine on the common platform, since there's no difference between a text and a binary stream.

Memory Error and Leaks

When it terminates successfully, your program is expected to free all of the dynamically allocated memory it allocates and close any files it opens. Valgrind can help you find memory errors or leaked files.

To get valgrind to check for memory errors in one of your programs, you can run your program like this:

```
$ valgrind --tool=memcheck --leak-check=full ./encode codes-1.txt input-1.txt encoded.bin
-lots of valgrind output deleted-
```

To get it to look for file leaks instead, you can run it like the following. You'll get a report that file descriptors 0, 1 and 2 are still open. That's normal; those are standard input, standard output and standard error. If you see others, that's probably a file leak.

```
$  valgrind --track-fds=yes ./decode codes-1.txt encoded-1.bin output.txt
-lots of valgrind output deleted-
```

Remember, valgrind will be able to give you a more useful report if you compile with the `-g` flag, so don't forget it.

Test Inputs

Most of the test cases involve encoding a file and then decoding it to recover the original input. So, the first 8 of these test cases are tests for both programs. The rest address error cases, so they just test one of the two programs.

1. An input file containing three symbols (g, o, newline) that uses codes-1.txt and fits exactly into 4 bytes (with no need for extra 0s at the end).
2. An input file containing three symbols (a, t, newline) that uses codes-1.txt and requires 2 extra 0s at the end.
3. A file containing three letters separated by spaces that uses codes-1.txt.
4. A file containing each letter of the alphabet followed by a newline that uses codes-1.txt.
5. A file containing each letter of the alphabet followed by a newline that uses codes-2.txt.
6. A file containing words starting with each letter of the alphabet that uses codes-1.txt.
7. A file containing words starting with each letter of the alphabet that uses codes-2.txt.
8. An empty file that uses codes-1.txt.
9. A error-handling test for `decode` that uses codes-1.txt and does not contain valid codes.
10. A error-handling test for `encode` that uses codes-1.txt and an input file that contains uppercase letters.
11. A error-handling test for `encode` with only one command-line argument:

```
csc$ ./encode codes-1.txt
usage: encode <codes-file> <infile> <outfile>
```

12. A error-handling test for `decode` with a nonexistent file.
```
csc$ ./decode codes-1.txt encoded-12.bin output.txt
encoded-12.bin: No such file or directory
```
13. A error-handling test for `encode` that uses bad-codes.txt.

Grading

The grade for your programs will depend mostly on how well they function. We'll also expect them to compile cleanly, to follow the style guide and to follow the design given for each program. We'll also try your programs under valgrind, to see if it can find anything to complain about.

- Compiling cleanly on the common platform: **10 points**
- Working Makefile: **8 points**
- Behaves correctly on all tests: **60 points**
- Program follows the style guide: **20 points**
- Deductions
  - Up to **-60 percent** for not following the required design.
  - Up to **-30 percent** for failing to submit required files or submitting files with the wrong name.
  - Up to **-30 percent** penalty for file leaks, memory leaks or other memory errors.
  - **-20 percent** penalty for late submission.

Getting Started

To get started on this project, you'll need to clone your NCSU github repo and unpack the given starter into the p5 directory of your repo. You'll submit by checking files into your repo and pushing the changes back up to the NCSU github.

Clone your Repository

You should have already cloned your assigned NCSU github repo when you were working on Project 2. If you haven't already done this, go back to the assignment for Project 2 and follow the instructions for cloning your repo.

Unpack the starter into your cloned repo

Make sure you're in the `p5` directory in your cloned repo. You will need to copy and unpack the Project 5 starter. We're providing this as a compressed tar archive, starter5.tgz. After you download this file, you can unpack its contents into your `p5` directory. As with previous assignments, remember there's a `.gitignore` file that needs to be there, even though this file won't show up (by default) in a directory listing.

As usual, if you are logged in on one of the common platform systems, you can save yourself a few steps by

unpacking the starter directly from our official copy in AFS. Be sure you're in the `p5` directory of your repo and run:

```
$ tar xzvpf /afs/eos.ncsu.edu/courses/csc/csc230/common/www/proj/p5/starter5.tgz
```

Instructions for Submission

If you've set up your repository properly, pushing your changes to your assigned CSC230 repository should be all that's required for submission. When you're done, we're expecting your repo to contain the following files. You can use the web interface on github.ncsu.edu to confirm that the right versions of all your files made it.

- `encode.c` : source file, created by you.
- `decode.c` : source file, created by you.
- `codes.c` : source file, created by you.
- `codes.h` : header file, created by you.
- `bits.h` : header file, provided with the starter, maybe with some more function prototypes added by you.
- `bits.c` : source file, created by you.
- `Makefile` : the project's makefile, created by you.
- codes-*.txt : code files, provided with the starter
- bad-codes.txt : bad code file, provided with the starter
- `input-*.txt` : test inputs given to the `encode` program, provided with the starter. These are also the expected decoded results for the `decode` program.
- `encoded-*.bin` : expected encoded results for each input, provided with the starter. These are also the test inputs for the `decode` program.
- `stderr-*.txt` : expected error output for a few of the tests.
- `test.sh` : test script, provided with the starter.
- `.gitignore` : a file provided with the starter, to tell git not to track temporary files for this project.

Pushing your Changes

To submit your project, you'll need to commit your changes to your cloned repo, then push them to the NCSU github. Project 2 has more detailed instructions for doing this, but I've also summarized them here.

As you make changes to your project, you'll need to stage new or modified files in the index:

```
$ git add .
```

Then, before you commit, it's a good idea to check to make sure your index has the right files staged:

```
$ git status
```

Once you've staged a set of related changes, commit them locally:

```
$ git commit -am "<meaningful message for future self>"
```

Of course, you haven't really submitted anything until you push your changes up to the NCSU github:

```
$ unset SSH_ASKPASS # if needed
$ git push
```

Checking Jenkins Feedback

Checking jenkins feedback is similar to the previous Project. Visit our Jenkins system at http://go.ncsu.edu/jenkins-csc230 and you'll see a new build job for Project 5. This job polls your repo periodically for changes and rebuilds and tests your project automatically whenever it sees a change.

Learning Outcomes

The syllabus lists a number of learning outcomes for this course. This assignment is intended to support several of theses:

- Write small to medium C programs having several separately-compiled modules
- Explain what happens to a program during preprocessing, lexical analysis, parsing, code generation, code optimization, linking, and execution, and identify errors that occur during each phase. In particular, they will be able to describe the differences in this process between C and Java.
- Correctly identify error messages and warnings from the preprocessor, compiler, and linker, and avoid them.
- Find and eliminate runtime errors using a combination of logic, language understanding, trace printout, and gdb or a similar command-line debugger.
- Interpret and explain data types, conversions between data types, and the possibility of overflow and underflow
- Explain, inspect, and implement programs using structures such as enumerated types, unions, and constants and arithmetic, logical, relational, assignment, and bitwise operators.
- Trace and reason about variables and their scope in a single function, across multiple functions, and across multiple modules.
- Allocate and deallocate memory in C programs while avoiding memory leaks and dangling pointers. In particular, they will be able to implement dynamic arrays and singly-linked lists using allocated memory.
- Use the C preprocessor to control tracing of programs, compilation for different systems, and write simple macros.
- Write, debug, and modify programs using library utilities, including, but not limited to assert, the math library, the string library, random number generation, variable number of parameters, standard I/O, and

file I/O.

- Use simple command-line tools to design, document, debug, and maintain their programs.
- Use an automatic packaging tool, such as make or ant, to distribute and maintain software that has multiple compilation units.
- Use a version control tools, such as subversion (svn) or git, to track changes and do parallel development of software.
- Distinguish key elements of the syntax (what's legal), semantics (what does it do), and pragmatics (how is it used) of a programming language.