Project 6 – File Compression
CS 251, Fall 2020, Reckinger

Collaboration Policy: By submitting this assignment, you are acknowledging you have read the collaboration policy in the syllabus for projects. This project should be done individually. You may not receive any assistance from anyone outside of the CS 251 teaching staff. If you have been approved to have a partner, by filling out this form, you may collaborate (including code sharing) with your partner. **If you are partnering, please read the Pair Programming section before continuing).

Late Policy: You are given a total of 5 late days to use at your discretion (for all projects, not each project). You may use the late days in 24-hour chunks (either 1 day at a time or all five at once). You do not need to alert the instructor to ask permission to use late days. You can manage your use of late days on Mimir directly. **For all students late days will not be available to you in Mimir automatically, please post to Piazza and we can give you the late day manually.** If both partners have enough late days to use, you can request to use them on Piazza and we will manually apply them. Both partners will lose a late day for each late day used.

Early Bonus: If you submit a finished project early (by Wednesday, November 11^{th} at midnight), you can receive 10% extra credit. Please note, your submission needs to pass 100% of the tests cases to receive the early bonus.

Test cases/Submission Limit: Unlimited submissions.

What to submit: (1) util.h; (2) secretmessage.txt.huf (3) Partner eval (if you had one)—WARNING—if you submit other files (like compressed and uncompressed text files, you might think you are passing the test cases. However, we will re-test your code with ONLY these two files. Make sure you are doing the same!)

.pdf starter code

Project Overview

For this projet, you will build a file compression algorithm that uses binary trees and priority queues. Your program will allow the user to compress and decompress files using the standard Huffman algorithm for encoding and decoding. You will also use a custom hashmap class that is provided. Huffman encoding is an example of a lossless compression algorithm that works particularly well on text but can, in fact, be applied to any type of file. Using Huffman encoding to compress a file can reduce the storage it requires by a third, half, or even more, in some situations. You'll be impressed with the compression algorithm, and you'll be equally impressed that you're outfitted to implement the core of a tool that imitates one you're already very familiar with.

The starter code for this project is larger than what is typical. You should only edit the files you need to turn in, do not edit the remaining files. If you accidentally edit them, re-download the starter code using the link above.

Related reading:

Huffman on Wikipedia

ASCII Table

Huffman Coding – More interesting reading

Huffman Encoding

Huffman encoding is an algorithm devised by David A. Huffman of MIT in 1952 for compressing textual data to make a file occupy a smaller number of bytes. Though it is a relatively simple compression algorithm, Huffman is powerful enough that variations of it are still used today in computer networks, fax machines, modems, HDTV, and other areas.

Normally textual data is stored in a standard format of 8 bits per character, using an encoding called *ASCII* that maps each character to a binary integer value from 0-255. The idea of Huffman encoding is to abandon the rigid 8-bits-per-character requirement, and instead to use binary encodings of different lengths for different characters. The advantage of doing this is that if a character occurs frequently in the file, such as the very common letter **'e'**, it could be given a shorter encoding (i.e., fewer bits), making the overall file smaller. The tradeoff is that some characters may need to use encodings that are longer than 8 bits, but this is reserved for characters that occur infrequently, so the extra cost is worth it, on the balance.

The table below compares ASCII values of various characters to possible Huffman encodings for some English text. Frequent characters such as space and 'e' have short encodings, while rarer characters (like 'z') have longer ones.

Character	ASCII Value	ASCII Binary	Huffman Binary
`' '`	32	00100000	10
`'a'`	97	01100001	0001
`'b'`	98	01100010	0111010
`'c'`	99	01100011	001100
`'e'`	101	01100101	1100
`'z'`	122	01111010	00100011010

The steps you'll take to do perform a Huffman encoding of a given text source file into a destination compressed file are:

1. **count frequencies**: Examine a source file's contents and count the number of occurrences of each character, and store them in a map using the `hashmap` class you are provided.
2. **build encoding tree**: Build a binary tree with a particular structure, where each node represents a character and its count of occurrences in the file. A **priority** queue is used to help build the tree along the way.
3. **build encoding map**: Traverse the binary tree to discover the binary encodings of each character.
4. **encode data**: Re-examine the source file's contents, and for each character, output the encoded binary version of that character to the destination file.

The main.cpp file provided allows you to test each step as you go to make sure it is correct. If you run this along side the solution.exe, you can be sure you are doing things correctly.

Milestone 1 – Build Frequency Map - Option 1

As an example, let's use example.txt whose contents are "ab ab cab". In the original file, this text occupies 10 bytes (80 bits) of data, including spaces and special "end-of-file" (EOF) byte.

byte	1	2	3	4	5	6	7	8	9	10
char	'a'	'b'	' '	'a'	'b'	' '	'c'	'a'	'b'	EOF
ASCII	97	98	32	97	98	32	99	97	98	256
binary	01100001	01100010	00100000	01100001	01100010	00100000	01100011	01100001	01100010	N/A

In step 1 of Huffman's algorithm, a count of each character is computed. This frequency table is represented as a map:

```
98: 'b'    -->  3
99: 'c'    -->  1
97: 'a'    -->  3
32: ' '    -->  2
256:  EOF  -->  1
```

Note that **all** characters must be included in the frequency table, including spaces, any punctuation, and the EOF marker.

You will need to read the hashmap.cpp and hashmap.h files to understand how to use this abstraction. Instead of having you write this file, I am just giving it to you and it is your job to read and understand how to use it. Being able to read and understand code is almost as important as being able to write code.

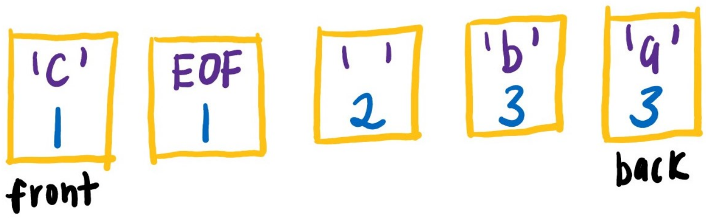
For all of the tests in the autograder, we are testing how your abstraction via i/o comparison. Therefore, insertion into the containers has to be the same. For example, with the frequency map, you must add the PSEUDO_EOF character to the map last. PSEUDO_EOF is a constant defined in the bitstream.h file and is available to use. You must manually add this to the map.

Write buildFrequencyMap function

This is Milestone 1 of the encoding process. In this function you read file passed in as parameter. You should count and return a mapping from each character (represented as `int` here) to the number of times that character appears in the file. You should also add a single occurrence of the fake character `PSEUDO_EOF` into your map. You may assume that the input file exists and can be read, though the file might be empty. An empty file would cause you to return a map containing only the 1 occurrence of `PSEUDO_EOF`. Also, make sure sure you are using the stream documentation to determine how to read from a stream a single character. Please make sure you include any header files for whatever you are using in util.h prior to submitting.

Milestone 2 – Build Encoding Tree – Option 2

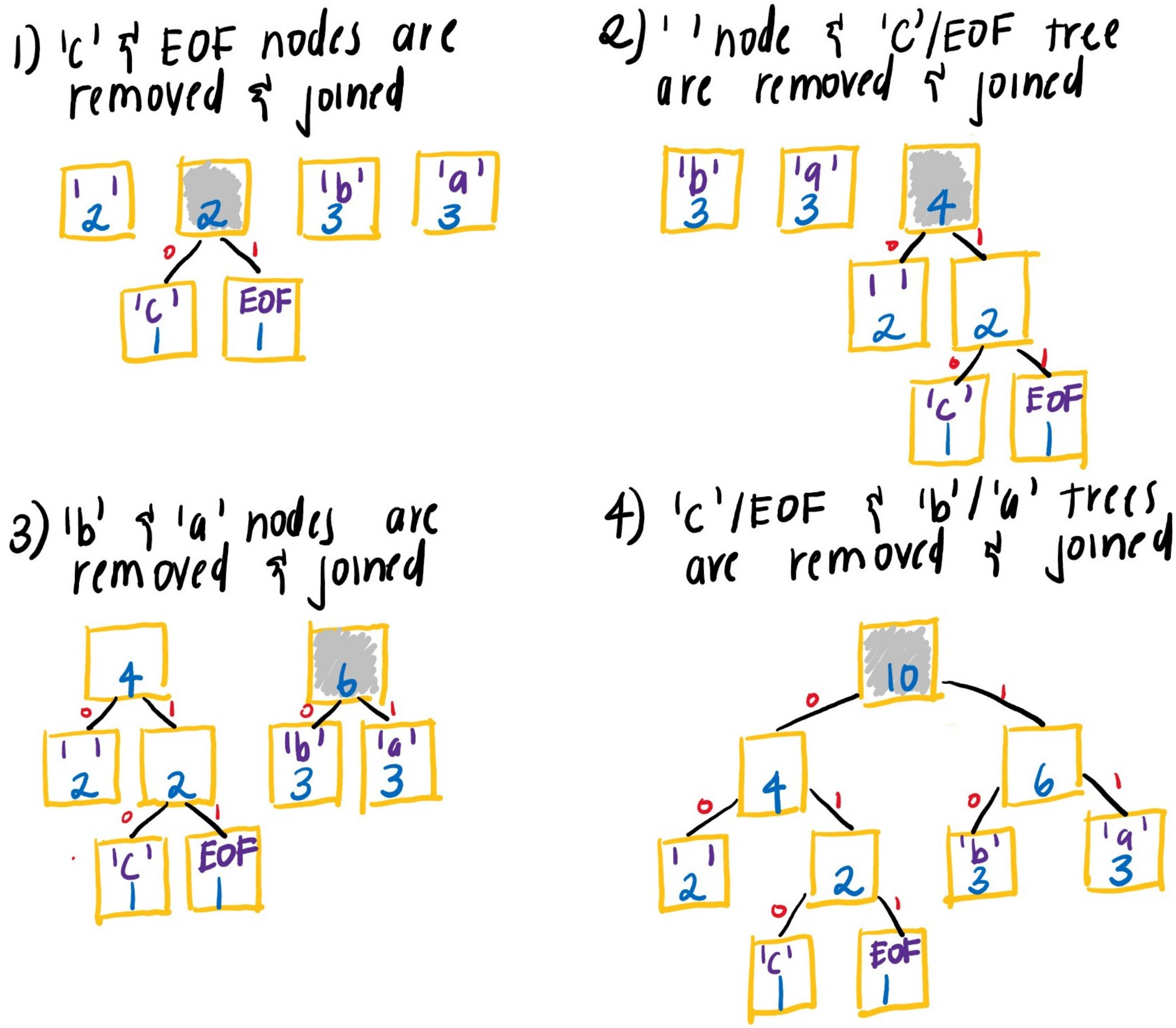
Step 2 of Huffman's algorithm builds an encoding tree as follows. First, we place our counts into node structs (out of which we will build the binary tree); each node stores a character and a count of its occurrences. Then, we put the nodes into a priority queue (using the priorityqueue you wrote, or the C++ library priority queue if you prefer), which stores them in prioritized order, where smaller counts have a higher priority. This means that characters with lower counts will come out of the queue sooner, as the figure below shows.



Now, to construct the tree, the algorithm repeatedly removes two nodes from the front of the queue (i.e., the two nodes with the smallest frequencies) and joins them into a new node whose frequency is their sum. The two nodes are positioned as children of the new node; the first node removed becomes the left child, and the second becomes the right. The new node is re-inserted into the queue in sorted order (and we can observe that its priority will now be less urgent, since its frequency is the sum of its children's frequencies). This process is repeated until the queue contains only one binary tree node with all the others as its children. This will be the root of our finished Huffman tree.

If you are using your priority queue, do NOT mix begin/next with dequeue/peek. Only use peek/dequeue for this. There is no defined behavior in the project for how begin/next behaves with dequeue/peek.

The following diagram illustrates this process Notice that the nodes with low frequencies end up far down in the tree, and nodes with high frequencies end up near the root of the tree. As we shall see, this structure can be used to create an efficient encoding in the next step.



Write buildEncodingTree function

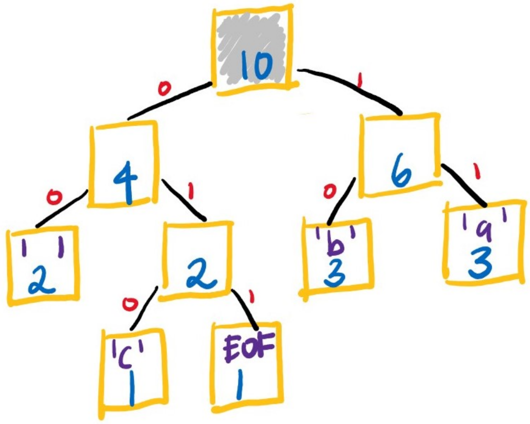
This is Milestone 2 of the encoding process. In this function you will accept a frequency map (like the one you made in `buildFrequencyMap`) and use it to create a Huffman encoding tree based on those frequencies. Return a pointer to the node representing the root of the tree.

You may assume that the frequency table is valid: that it does not contain any keys other than char values, `PSEUDO_EOF`, and `NOT_A_CHAR`; that all counts are positive integers; and that it contains at least one key/value pairing.

When building the encoding tree, you will need to use a priority queue to keep track of which nodes to process next. Use the `priorityqueue` abstraction you wrote in the previous project or the C++ standard library. This allows each element to be enqueued along with an associated priority. If you did not finish Project 5, please see section below called “Priority Queue”.

Milestone 3 – Build Encoding Map – Option 3

The Huffman code for each character is derived from your binary tree by thinking of each left branch as a bit value of 0 and each right branch as a bit value of 1, as shown in the diagram below:



The code for each character can be determined by traversing the tree. To reach ' ', we go left twice from the root, so the code for ' ' is 00. Similarly, the code for 'c' is 010, the code for EOF is 011, the code for 'a’ is 11 and the code for 'b is 10. By traversing the tree, we can produce a map from characters to their binary representations. Though the binary representations are integers, since they consist of binary digits and can have arbitrary lengths, we will store them as strings. For this tree, the encoding map would look like this:

```
98: 'b'    -->  10
256:  EOF  -->  011
99: 'c'    -->  010
97: 'a'    -->  11
32:   ' '  -->  00
```

Write the buildEncodingMap function

This is Milestone 3 of the encoding process. In this function will you accept a pointer to the root node of a Huffman tree (like the one you built in `buildEncodingTree`) and use it to create and return a Huffman encoding map based on the tree's structure. Each key in the map is a character (int in our case), and each value is the binary encoding for that character represented as a string. For example, if the character 'a' has binary value 10 and 'b' has 11, the map should store the key/value pairs 'a':"10" and 'b':"11". If the encoding tree is `nullptr`, return an empty map.

Milestone 4 – Encode text – Option 4

*(**Please double check the starter code you have to ensure that in main.cpp, near line 183, you have the encode call as:*

```
string codeStr = encode(input, encodingMap, output, size, true);
```

If the last parameter is false, please change to true. It doesn’t matter for the autograder, but it will matter for testing in your IDE. You should only write bits to file if the last parameter is true. If it is false, only return the string representation of bits.)

Using the encoding map, we can encode the file’s text into a shorter binary representation. Using the preceding encoding map, the text "`ab ab cab`" would be encoded as:

1110001110000101110011

The following table details the char-to-binary mapping in more detail. The overall encoded contents of the file require 22 bits, or a little under 3 bytes, compared to the original file size of 10 bytes.

char	‘a’	‘b’	‘ ‘	‘a’	‘b’	‘ ‘	‘c’	‘a’	‘b’	EOF
binary	11	10	00	11	10	00	010	11	10	011

Since the character encodings have different lengths, often the length of a Huffman-encoded file does not come out to an exact multiple of 8 bits. Files are stored as sequences of whole bytes, so in cases like this the remaining digits of the last bit are filled with 0s. You do not need to worry about implementing this; it is part of the underlying file system.

byte	1	2	3
char	a b a	b c a	B EOF
binary	11 10 00 11	10 00 010 1	1 10 011 *00*

It might worry you that the characters are stored without any delimiters between them, since their encodings can be different lengths and characters can cross byte boundaries, as with 'a' at the end of the second byte. But this will not cause problems in decoding the file, because Huffman encodings by definition have a useful prefix property where no character's encoding can ever occur as the start of another's encoding. (If it's not clear to you how this works, trace through the example tree above, or one produced by your own algorithm, to see for yourself.)

Write encode function

This is Milestone 4 of the encoding process. In this function you will read one character at a time from a given input file, and use the provided encoding map to encode each character to binary, then write the character's encoded binary bits to the given output bit stream. After writing the file's contents, you should write a single occurrence of the binary encoding for `PSEUDO_EOF` into the output so that you'll be able to identify the end of the data when decompressing the file later. You may assume that the parameters are valid: that the encoding map is valid and contains all needed data, that the input stream is readable, and that the output stream is writable. The streams are already opened and ready to be read/written; you do not need to prompt the user or open/close the files yourself. There is a parameter in the encode function called "makeFile". If makeFile is true, you will write the output file and return the string representation of the bits. If makeFile is false, you will not write to the output file, but you will still return the string representation of the bits.

Running into sigpipe error? Read section "SIGPIPE" below.

Milestone 5 – Decode text – Option 5

You can use a Huffman tree to decode text that was previously encoded with its binary patterns. The decoding algorithm is to read each bit from the file, one at a time, and use this bit to traverse the Huffman tree. If the bit is a 0, you move left in the tree. If the bit is 1, you move right. You do this until you hit a leaf node. Leaf nodes represent characters, so once you reach a leaf, you output that character. For example, suppose we are given the same encoding tree above, and we are asked to decode a file containing the following bits:

`11100100010010100 11`

Using the Huffman tree, we walk from the root until we find characters, then output them and go back to the root.

- We read a 1 (right), then a 1 (right). We reach 'a' and output a. Back to the root.
  `11100100010010100 11`
- We read a 1 (right), then a 0 (left). We reach 'b' and output b. Back to root.
  `11100100010010100 11`
- We read a 0 (left), then a 1 (right), then a 0 (left). We reach 'c' and output c.
  `11100100010010100 11`
- We read a 0 (left), then a 0 (left). We reach ' ' and output a space.
  `11100100010010100 11`
- We read a 1 (right), then a 0 (left). We reach 'b' and output b.
  `11100100010010100 11`
- We read a 0 (left), then a 1 (right), then a 0 (left). We reach 'c' and output c.
  `1110010001001010011`

- We read a 1 (right), then a 0 (left). We reach 'b' and output b.
  `11100100010010`<u>`10`</u>`011`
- We read a `0, 1, 1.` This is our EOF encoding pattern, so we stop. The overall decoded text is “`abc bcb`”.
- `1110010001001010`<u>`011`</u>

Write decode function

This is the “decoding a file” process described previously. In this function you should do the opposite of `encode`; you read bits from the given input file one at a time, and recursively walk through the specified decoding tree to write the original uncompressed contents of that file to the given output stream. The streams are already opened and you do not need to prompt the user or open/close the files yourself.

Milestone 6 – Free the tree – Option 6

Since we are allocated memory for our Huffman tree, we must make sure to clean it up. Implement this function and test using valgrind.

Milestone 7 – Compress – Option C

The functions above implement Huffman’s algorithm, but they have one big flaw. The decoding function requires the encoding tree to be passed in as a parameter. Without the encoding tree, you don’t know the mappings from bit patterns to characters, so you can't successfully decode the file. We will work around this by writing the encodings into the compressed file, as a header. The idea is that when opening our compressed file later, the first several bytes will store our encoding information, and then those bytes are immediately followed by the binary bits that we compressed earlier. It’s actually easier to store the character frequency table, the map from Milestone 1 of the encoding process, and we can generate the encoding tree from that. For our `ab ab cab` example, remember this is the frequency table:

```
98: 'b'    -->  3
99: 'c'    -->  1
97: 'a'    -->  3
32: ' '    -->  2
256: EOF   -->  1
```

In the header of the compressed file, we will write:

```
{98:3, 99:1, 97: 3, 32: 2, 256: 1}
```

We don't have to write the encoding header bit-by-bit; just write out normal ASCII characters for our encodings. We could come up with various ways to format the encoding text, but this would require us to carefully write code to write/read the encoding text. There's a simpler way. You already have a hashmap of character frequency counts from Step 1 of encoding. In C++, collections like maps can easily be read and written to/from streams using `<<` and `>>` operators. We have provided overriden versions of these operators for the `hashmap` class, so all you need to do for your header is write your map into the bit output stream first before you start writing bits into the compressed file, and read that same map back in first when you decompress it later. The overall file is now 34 bytes: 31 for the header and 3 for the binary compressed data.

Looking at this new rendition of the compressed file, you may be thinking, “The file isn’t compressed at all; it actually got larger than it was before! It went up from 9 bytes (“`ab ab cab`”) to 34!” That's true for this contrived example. But for a larger file, the cost of the header is not so bad relative to the overall file size. There are more compact ways of storing the header, too, but they add too much challenge to this assignment, which is meant to practice trees and data structures and problem solving more than it is meant to produce a truly tight compression.

Now, that you understand how to include the header in the compresed file, it is time to write the compress function. In this function you should compress the given input file into an output file. You should read the input file one character at a time, building an encoding of its contents, and write a compressed version of that input file, including a header, to the specified output file. This function should be built on top of the other encoding functions and syou hould call them as needed. You may assume that the file is valid, but the input file might be empty.

Milestone 8 – Decompress – Option D

This function should do the opposite of compress; it should read the bits from the given input file one at a time, including your header packed inside the start of the file, to write the original contents of that file to a new output file. You may assume that the filename is valid, but the input file might be empty.

Input/Output Streams

In the past we have wanted to read input an entire line or word at a time, but in this program it is much better to read one single character (byte) at a time. So you should use the following input/output stream functions:

ostream: `void put(char ch)` – writes a single character to the output stream.

istream: `char get()` – reads a single character from input.

In parts of this program you will need to read and write bits to files. To read or write a compressed file, even a whole byte is too much; you will want to read and write binary data one single bit at a time, which is not directly supported by the default in/output streams. Therefore the Stanford C++ library provides `obitstream` and `ibitstream` classes with `writeBit` and `readBit` members to make it easier.

obitstream (bit output stream) member	**Description**
`void writeBit(int bit)`	writes a single bit (0 or 1) to the output stream

ibitstream (bit input stream) member	**Description**
`int readBit()`	reads a single bit (0 or 1) from input; -1 at end of file

Note that the bit in/output streams also provide the same members as the original ostream and istream classes from the C++ standard library, such as `getline`, `<<`, `>>`, etc. But you usually don't want to use them, because they operate on an entire byte (8 bits) at a time, or more, whereas you want to process these streams one bit at a time.

Priority Queue

The solution for this project uses the priority queue abstraction we wrote in Project 5. The way that abstraction handles duplicate priorities is that it orders elements by insertion order. You will need to handle duplicates in whatever priority queue you use in the same way in order to pass the autograder and get credit for your work. If neither you nor your partner have a working implementation of the priority queue, here are your options:

(1) Use the C++ priority queue. Lecture from Week 11 shows you how to use it and how to set up a prioritize function. In order to handle duplicates properly, we suggest you alter the Huffman node:

```
struct HuffmanNode {
    int character;
    int count;
    int order;
    HuffmanNode* zero;
    HuffmanNode* one;
};
```

You can use the third member variable, order, to keep track of insertion order as you build the nodes. And then you should write your prioritize function to order using count first, but when count is a tie, order using order.

(2) You could use a C++ map. The key would be count, the value would be a vector of HuffmanNode*s (that are in order of insertion). I explained how you can do this to test your priority queue in Project 5. It could also be used here! So many maps!! You will need to use a map, not an unordered map, in order to keep the keys inorder.

SIGPIPE Error

If you are receiving this error that means you have an efficient code. Things you should check:

1. Eliminate unecessary loops or recursive calls. You should only loop through the data at most one time.

2. Make sure you are using the makeFile flag properly. If makeFile is false, you should NOT call writeBit anywhere in your code (it is slow!).

3. Weird thing I just learned. During string concatentation, it turns out that under the hood these two forms of string concatentation are very different:

str += add;

vs.

str = str + add;

(source: https://www.oreilly.com/library/view/optimized-c/9781491922057/ch04.html, Under this heading: "Use Mutating String Operations to Eliminate Temporaries".

When you use the += operator, it optimizes a lot to avoid making extra temporary copies of add. Therefore, you want to do str += add! It on the order of ten times faster.

Tips/Strategies

- When writing the bit patterns to the compressed file, note that you do not write the ASCII characters '0' and '1' (that wouldn't do much for compression!), instead the bits in the compressed form are written one-by-one using the `readBit` and `writeBit` member functions on the bitstream objects. Similarly, when you are trying to read bits from a compressed file, don't use `>>` or byte-based methods like `get` or `getline`; use `readBit`. The bits that are returned from `readBit` will be either 0 or 1, but not '0' or '1'.
- Work step-by-step. Get each part of the encoding program working before starting on the next one.
- Start out with small test files (two characters, ten characters, one sentence) to practice on before you start trying to compress large books of text. What sort of files do you expect Huffman to be particularly effective at compressing? On what sort of files will it less effective? Are there files that grow instead of shrink when Huffman encoded? Consider creating sample files to test out your theories.
- Your implementation should be robust enough to compress any kind of file: text, binary, image, or even one it has previously compressed. Your program probably won't be able to further squish an already compressed file (and in fact, it can get larger because of header overhead) but it should be possible to compress multiple iterations, decompress the same number of iterations, and return to the original file.
- Your program only has to decompress valid files compressed by your program. You do not need to take special precautions to protect against user error such as trying to decompress a file that isn't in the proper compressed format.
- The operations that read and write bits are somewhat inefficient and working on a large file (100K and more) will take some time. Don't be concerned if the reading/writing phase is slow for very large files.

Requirements

1. You must have a clean valgrind report when your code is run. Check out the makefile to run valgrind on your code. All memory allocated (call new) must be freed (call delete). The test cases run valgrind on your code.
2. No global or static variables.
3. You may not change any of the function headers in util.h. However, you can and should add helper functions to the file.
4. Code must be written efficiently. Complexity should be minimized. Deductions to final submission will be applied for solutions that are inefficient, in space or in time.

Pair Programming

From Wikipedia, "Pair programming is an agile software development technique in which two programmers work together at one workstation. One, the driver, writes code while the other, the observer or navigator, reviews each line of code as it is typed in. The two programmers switch roles frequently."

Given our current pandemic, you will need to adapt. Unfortunately, Mimir does not naturally allow two programmers to collaborate in a live IDE. The way it works is you will each have your own separate IDE but you will both submit to the same submission folder. Once the code is submitted, you will both have access to that code (and can "open in IDE" to bring it back into your terminal). Be aware that when you do that it is in a temporary .submission folder, so be sure to copy it somewhere more permanent if you plan to edit. Therefore, my recommendation is that the "driver" codes and the navigator is navigating via a shared screen. In Zoom, you can use remote access to trade off who is the driver and keep the code in one IDE, or you can switch back and forth if you'd like (using the submission folder as your "branch").

As soon as this assignment is released, please set up a meeting with you partner. Get together a plan and a timeline. Here is a list of discussion topics to go through with your partner during your first meeting:

-How many late days do we have available? You may only use a late day if you both have one to use.

-Do we want to submit the project early for extra credit?

-What days/times are best for us to work on this? And can we put some dates down on the calendar to get us scheduled.

-What will our roles be? How often should we rotate driver/navigator?

-Do we agree on having an open line of communication if we feel the partnership is not mutually beneficial? How will go about addressing this?

-What are the goals of each partner? To finish as fast as possible? To understand the concepts behind the code? To explore beyond the scope of the project?

Creative Component - secretmessage.txt.huf

Along with your program, turn in a file secretmessage.txt.huf that stores a compressed message from you to your TAs. Create the file by compressing a text file with your compress function. The message can be anything (appropriate) you choose. Your TA will decompress your message with your program and read it while grading.

Citations/Resources

Owen L. Astrachan, Duke University and Julie Zelenski, Stanford Unversity.