Assignment #5: Huffman coding

Thanks to Owen Astrachan at Duke University for the inspiration.

Due: Mon, May 15th in class

Your mission in this assignment is to write and use some handy-dandy ADTs along with your newly-gained knowledge of trees to write a super-snazzy compression and decompression utility based on the Huffman coding algorithm. Huffman coding is an example of a lossless compression algorithm that works particularly well on text and in fact, can be applied to any type of file, not just text files. It can reduce the storage required by a third or half or even more in some situations. Hopefully, you will be impressed with this awesome algorithm and your ability to implement such a nifty tool!

Carefully read Handout #29 for background information on compression and the specifics of the Huffman coding algorithm. This handout doesn't repeat that material, but instead just describes the structure of the program as we are asking you to implement. Your assignment is to write a program that allows the user to compress and decompress as many files as they like, using the standard Huffman algorithm for encoding and decoding.

Overview of the module structure

Like all complex programs, the development goes more smoothly if the task is divided into separate modules. We have already divided the task up into four modules:

`huffman`— This module contains the main program, which is responsible for compressing and decompressing files. It uses the encoding module to build the encoding and the binary file module to read and write encoded bit patterns into files.

`pqueue`— This module defines an abstract data type for a priority queue. A priority queue is somewhat like an ordinary queue except that it dequeues elements according to their priority. It is used by the encoding module to manage the node collection when building an encoding tree.

`binaryfile`— This module is already written for you. It defines specialized I/O operations to read and write a single bit at a time to a file.

`encoding`— This module defines an abstract data type representing a particular encoding. It has a function to build an encoding tree and manages the mapping from character to bit pattern and back. It also is responsible for reading and writing the file header containing the encoding table.

As usual, we give you compiled .lib files for each module to aid you in your development.

The binary file module — single bit I/O functions

Let's first start off with an easy module— the binary file module is already written for you and all you need to do is use it. It provides functions to open a binary file for reading or writing and functions to read or write a single bit at a time. The functions exported from the module are listed by name here, you'll find the full specification about the functions in the **`binaryfile.h`** interface file include in the starting project folder.

```
BinaryFile *OpenForReading(string filename);
BinaryFile *OpenForWriting(string filename);
void WriteBit(BinaryFile *bfile, int bit);
int ReadBit(BinaryFile *bfile);
FILE *GetUnderlyingFile(BinaryFile *bfile);
void CloseAndFree(BinaryFile *bfile);
```

You will not need to change anything or do any development on this module, you just need to learn how to correctly be a client of its exported interface.

The priority queue ADT — dequeues elements by priority

The priority queue is a variation on the standard FIFO queue described in Chapter 10 of the text. In some cases, a FIFO strategy may be too simple for the activity being modeled. A hospital emergency room, for example, needs to schedule patients according to priority. A patient who arrives with a more serious problem will pre-empt others even if they have been waiting longer. This is a *priority queue*, where elements are added to the queue by priority and when asked to dequeue one, it is the highest priority element in the queue that is removed. Such an ADT would be useful in a variety of situations; in fact, you can even sort data by inserting all the values into a priority queue and then dequeueing them one by one, which will pull them out in order.

While building an optimal encoding tree, a priority queue will be used to manage the collection of nodes being processed. At each stage you dequeue the two minimum nodes (those two have what is defined as "higher priority"), combine them into a new tree, and enqueue the new root node back onto the queue for later processing.

Here are the functions that make up the interface of the priority queue, the formal specification for each function is detailed in the **`pqueue.h`** interface file given in the starting project folder:

```
pqueueADT NewPQueue(cmpFn cmp);
void FreePQueue(pqueueADT queue);
bool PQueueIsEmpty(pqueueADT queue);
bool PQueueIsFull(pqueueADT queue);
void Enqueue(pqueueADT queue, void *newElem);
void *DequeueMax(pqueueADT queue);
```

Note the priority queue is polymorphic (i.e. it stores elements of **`void*`** type) which means the implementation assumes nothing more about the elements other than that they are some kind of pointer. But since the implementation needs to know how to order queue elements relative to each other, the client will have to supply a compare-by-priority function that can be applied to the queue elements to determine their ordering. Since that function is chosen once and must remain consistent throughout the lifetime of the ADT, it is appropriate that it be passed just once when a new priority queue is created.

There are many data structures you could choose to represent and manipulate a priority queue, with various tradeoffs between amount of memory required, speed of the Enqueue and Dequeue operations, complexity of code to get the job done, etc. You have total freedom to choose the implementation strategy with the tradeoffs that you prefer.

Hints and requirements for the pqueue module:

- The priority queue must be polymorphic and allow any type of pointer to be stored as queue elements. It will be similar to the polymorphic stack we went through together in lecture.
- We aren't requiring anything sophisticated for your implementation, the only mandate is that it correctly meet the functional specification given in the interface file. It is perfectly fine if your queue has *O(N)* performance on Enqueue or Dequeue in order to support the priority behavior.
- In the starter folder, we have provided both the linked-list and ring-buffer implementations of an ordinary FIFO queue as given in Chapter 10 of the text. You may want to adapt code from one of those implementations. You will likely only need to modify the Enqueue and/or Dequeue operations. You could also build something of your own design, such as a binary tree or the nifty heap that we showed in class.

- It makes good sense to write test code to exercise the priority queue in isolation before you integrate it into the larger program. This will allow you to find and fix its bugs without having to wade through the camouflage of the other modules.

The encoding ADT — builds and manages character-bit pattern encoding

The encoding ADT manages an encoding tree that assigns each character a unique encoded bit pattern. The module implements the Huffman algorithm for building an optimal encoding tree for an input file. It is also responsible for reading and writing the file header to a compressed file that enables the mapping to be reconstructed later when decompressing.

Here are the functions that make up the interface of the encoding ADT:

```
encodingADT NewEncoding(void);
void FreeEncoding(encodingADT e);
string EncodedBitPatternForChar(encodingADT e, int ch);
int CharForEncodedBitPattern(encodingADT e, string pattern);
void BuildEncodingForInputFile(encodingADT e, FILE *in);
void WriteEncodingTableToBinaryFile(encodingADT e, BinaryFile *bfile);
void ReadEncodingTableFromBinaryFile(encodingADT e, BinaryFile *bfile);
```

Most of the data managed by this ADT is internally stored in the form of an encoding tree as described in the data compression handout. Building this tree is done by analyzing the input file to count the characters and then constructing the optimal tree via the Huffman algorithm.

In order to facilitate quick retrieval of a character's encoding, the encoding ADT should maintain a secondary structure that allows quick access rather than having to hunt through the encoding tree to find the character of interest. The simplest means to do this is to create a 256-member array with entries from 0 to 255 (one for each character) and assign the entries by tracing out all the paths in the encoding tree. Then when you need to look up the encoding for a particular character, you have immediate access to it.

Going in the other direction, from bit pattern to character, is a matter of tracing your way down the encoding tree making left and right turns on the 0 and 1s to find the character at the end (or possibly to discover that the bit pattern isn't valid for the given encoding). Given the encoding ADT should already maintain an encoding tree, you will need no additional structure to support converting a bit pattern to its char.

The other responsibility of the encoding ADT is writing the encoding table information to the compressed file and later reading that information and recreating the encoding tree for use in decoding. Each encoding is unique to each file and we must store information about the encoding itself as the file header so we know how to interpret the compressed bit stream.

There are many options you have for reading and writing the encoding table. You could store the table at the head of the file in a long, human-readable string format using the ASCII characters '0' and '1' , one character entry per each line, like this:

```
h = 01
a = 000
p = 10
y = 1111
...
```

Reading this back in would allow you to recreate the tree path by path. You could have a line for every character in the ASCII set; characters that are unused would have an empty bit pattern. Or you could conserve space by only listing those characters that appear in the encoding. In such a case you

must record a number that tells how many entries are in the table or put some sort of sentinel or marker at the end so you know when you have processed them all.

As an alternative to storing sequences of ASCII '0' and '1' characters for the bit patterns, you could store just the character frequency counts and rebuild the tree again from those counts in order to decompress. Again we might include the counts for all characters (including those that are zero) or optimize to only record those that are non-zero. Here is how we might encode the non-zero character counts for the "happy hip hop" string (the 7 at the front says there are 7 entries to follow)

```
7 h 3 a 1 p 4 y 1  2 i 1 o 1
```

If you're feeling really inventive, you can go even further in your quest to save space. For example, it's possible to store the bit patterns by writing a sequence of single bits or you can devise a means to efficiently dehydrate the tree itself and store it.

You can use any combination of I/O routines (getc/putc, scanf/printf, single bit read/write, etc.) to save and restore the encoding table. The most critical issue is that your reading and writing functions are consistent. Whatever format you record the encoding table when writing should be understood and followed by the function that reads it to recreates the encoding later.

Hints and requirements for the encoding module:

- This is certainly the most complex of the modules, so figure that you will spend the bulk of your time developing and debugging this module.
- The encoding interface uses **`int`** instead of **`char`** because of the necessity to be able to represent values (such as **`EOF`**) that are distinct from the 256 patterns already used in the ASCII character set. Take care to respect those types. Also, for a somewhat obscure reason, it is unwise to use a **`char`** to directly index into an array, an **`int`** should be used instead. Our suggestion for avoiding problems is **to not use any variables of type char in your program**. Use **`int`** everywhere instead. It can do everything **`char`** can and more.
- It will be helpful to include error-checking in your functions to detect if they are ever used improperly as an aid for debugging. For example, if a client tries to look up a bit pattern for a char that is out of range or using an encoding that hasn't been set up yet, it would be more helpful to report that with Error than to reference outside the array or quietly return NULL.
- Any working format you devise for writing and reading the encoding table into the compressed file is acceptable. If you develop a successful space-saving technique (more compact than the list of character counts, let's say) we'll consider that an extension worthy of extra credit.
- Just FYI: the format used for the encoding table file header in our library/demo version is a list of character counts like this: **`32=45|97=10|99=8`** where the first number is the ASCII value of the char followed by = and its count. A vertical bar separates entries. Characters with non-zero counts are excluded. This is a format you might consider or you can adopt your own more space-efficient representation. When testing, keep in mind that a program is only expected to decompress files that were compressed using the same encoding header format (i.e. files compressed with your version of encoding.c will not likely be compatible with our .lib version and vice versa).

Huffman module —compression and decompression

The final module is the one that manages the compression and decompression tasks. It repeatedly offers the user the option to compress or decompress a file until the program has satisfied all their compression needs.

To compress a file, you will first create the optimal encoding for the input file, calling upon the functionality provided by the encoding ADT. Once the encoding is created, you write the file header

containing the encoding table to the output file. Then you process the input file character by character, writing the compressed form of each character to the output file bit-by-bit using the binary writing functions. Once you have processed all the characters in the input, voila, a compressed file! The program then reports the size of the original file, the size of the compressed file, and the factor by which it was able to shrink the file.

To uncompress a file, you use the encoding ADT's function to first read the encoding table from the file header so that it can recreate the same Huffman tree that was used to compress the data. Then you read through the compressed file bit-by-bit using the binary file read function and build up a bit pattern. Once that bit pattern matches a sequence for a particular character, you print that translated character to the result file and pick up reading the bits where you left off. Repeat until EOF and, presto, you have reconstituted the original file!

Hints and requirements for the huffman module:

- As humans are careless creatures, be sure to verify that their responses are valid and re-prompt them where necessary to correct errors.
- When you fopen the FILE* for the compression input or decompressed output, rather than pass the usual "r" for reading or "w" for writing mode, you must instead use "rb" or "wb". The additional of the "b" indicates the file may not be ordinary text and prevents the OS from going behind your back turning carriage returns into linefeeds or silly things like that**. This is essential if you are working in Visual C++** (CodeWarrior works either way).
- Compressing a file will require reading through the file twice: once by the encoding module to count the characters, and again in the huffman module when processing each character as part of writing the compressed output.
- 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 binary file output functions.
- Although in most cases, each encoded bit pattern will be 5-10 bits long, in the absolute, most lopsided case, a single bit pattern can be at most 256 bits long. This number may come in handy when trying to set up a temporary buffer into which to build the bitstring as you are reading from the compressed file.
- One way to get the size of a file is to use the `int ftell(FILE *f)` function which is a part of standard C. Given an opened FILE *, it will report the position of the file cursor. If you call `ftell` on a file when at EOF, it returns the length of the file.
- Our supplied huffman.lib module has one extra debugging command, a simple operation to compare two files character-by-character and report the first position at which they differ or whether they match entirely. This will be useful when trying to verify that the decompressed result exactly matches the original. **Your program will not have the match command** (i.e. you do not have to implement this operation), it is just provided to you as a debugging aid in the demo version.

General hints and suggestions

- *One module at a time.* As always, concentrate on one module in isolation. Use our compiled library modules to substitute for the parts you haven't yet written or are not currently doing development on.
- *Build test cases*. Make small test files (two characters, ten characters, one sentence) to practice on before you starting trying to compress the complete works of Shakespeare. 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? Create

sample files to test out your theories. Note that handling a file containing just one character (single or repeated multiple times) would require a special case (do you see why?) We will not expect you to do this and will not test against this case.

- *Create infrastructure to help debug*. Since the encoded binary files are impossible to decipher in a normal text editor (try opening one – they look like garbage), it is next to impossible to figure out what has gone astray by looking at the contents of a malformed file. You'll have to be a bit more inventive about coming up with ways to debug during your development. Building infrastructure (for example, debugging routines to print out the frequency counts, printing out the encoding tree/table, writing a parallel file using ASCII '0' and '1' characters instead of bits, etc.) will prove to be useful. As Owen said about his Duke students when they did a similar assigment: "most students build test and debugging functions as part of the program or eventually wish they had."
- *You are responsible for freeing memory*. Both your encoding and pqueue ADTs should have properly implemented free functions and the program should close all files and free the memory used after finishing each compression/decompression operation.
- *Your program should be able to compress any file.* Your implementation should be robust enough to compress any given file: text, binary, image, or even one it has previously compressed. You won't be able to further squish an already compressed file (and in fact, it can get larger because of the additional overhead of the encoding table) but it is 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 try to decompress a file that isn't in the proper compressed format. The binary file module does some error-checking to ensure the file is binary and was properly closed, and so it may catch some mis-uses for you, but beyond that, you are not expected to gracefully detect these errors. Also, given that your and our versions won't necessarily store the file header containing the encoded table in the same format, it is not expected that your program will be able to decompress files compressed by the demo or our version of the libraries and vice versa.
- *Writing/reading bits can be slow*. The operations that read and write bits are somewhat inefficient and working on a large file (100K and more) will take some time. It is definitely worthwhile to do some tests on large files as part of stress-testing your program, but don't be concerned if the reading/writing phase is takes a bit more time that you might expect.

A little extra challenge: extension ideas

- *Shrink the encoding table header.* Especially for small files, the encoding table header can significantly cut into the savings from the Huffman encoding. A good challenge is coming up with a very tight means of storing the encoding table in the smallest space. It will probably require reading and writing bit-by-bit to compress the storage needed.
- *Optionally use a standard Huffman encoding.* Another idea about reducing the overhead of the encoding table is to analyze a large body of English text and come up with a standard encoding tree that is not optimal for any given file, but is generally close to what is needed in most cases. Rather than building a specific encoding for a particular file, you would use this general encoding, and not bother to write out the encoding table, but instead leave a marker that tells your program to just use the general one when decoding. If you provide this, be sure it is an optional feature and your program can still use an individually optimized Huffman encoding.
- *Research other compression algorithms and perhaps implement one.* You may find it interesting to do some investigation into the techniques behind other compression algorithms such as Zip, GIF, JPEG, MPEG, Lempel-Ziv, and so on. Although most of them are fairly complicated, you

might learn some interesting things for your own edification or gather enough information and inspiration to implement a simplified version of one of these algorithm in your program.

- *Adapt the program to do encryption and decryption instead.* Encrypting and decrypting follows very much the same translation model as encoding and decoding, but in this case the encoding serves to disguise the input instead of squeezing excess space. It would be an interesting modification to develop an alternate implementation of the encodingADT model that builds a mapping to be used for encryption rather than decryption, perhaps by requiring a password to unscramble or unlock the encoding table.

Accessing Files

On the class web site, there are two folders of starter files: one for Mac CodeWarrior and one for Visual C++. Each folder contains these files:

huffman.lib	Compiled library for huffman module.
pqueue.h	Interface file for pqueue ADT.
pqueue.lib	Compiled library for pqueue ADT.
encoding.h	Interface file for encoding module.
encoding.lib	Compiled library for encoding module.
binaryfile.h	Interface file for binary file module.
binaryfile.lib	Compiled library for binary file module.
Standard queues	Folder with source for usual FIFO queue as array and linked list.
Huffman Demo	Compiled version of a working program.

To get started, create your own starter project and add the .lib files for the four modules. Selectively replace the .lib files with .c files as you progress in your development

Deliverables

At the beginning of Monday's lecture, turn a manila envelope containing a printout of your code for all three modules and a floppy disk containing your entire project. Everything should be clearly marked with your name, CS106B and your section leader's name! Once again, remember to keep a backup in a safe place. Never turn in the only electronic copy of a program!

"I love deadlines. I love the whooshing sound they make as they fly by." — Douglas Adams
Don't let this deadline whoosh by you! After this, just two programs to go!