EECS 2510 – Non-Linear Data Structures and Programming in C++

Lab 2 – Due: Tuesday, 03/1/2016 at 08:30 AM

For this programming lab, you are to create a complete Huffman encoding system, and use it to encode and decode a large text file.

As we discussed in class, the first step is to create a table containing the alphabet frequency counts. For this assignment, your "alphabet" will be the entire 256-symbol ASCII set (o.k., technically, ASCII is only a 128-symbol set, but your program may encounter any of the possible 256 values that can be put into a byte).

Create a class (`Huffman`) that exposes (`public:`) three `void` methods:

`InitializeFromFile(string FileName)` - takes supplied file name and builds a Huffman tree, table of encoding strings, etc. Should print number of bytes read. You may also want to print the character frequency counts, encoding strings, etc., for your own debugging purposes (but not in the version you turn in). This method "sets up" the whole Huffman mechanism (i.e., after you `InitializeFromFile`, *then* you can encode or decode)

To build your tree, you will use the file `Shakespeare.txt`, which I have provided. This file contains the complete text of Shakespeare's plays (~5 MB). You will read this file one character at a time, and count how many times each character appears.

From the character frequency count table, build a Huffman tree as described in class

Once your tree is built, you will need to traverse the tree and build a table of encoding strings, one string for each of the 256 possible characters in a byte. Note: some of your strings will probably be VERY long, because MANY of the 256 possible values are not represented in this file. Don't be surprised if you have some strings of over 100 characters, corresponding to symbols with frequency counts of zero.

When it finishes, `InitializeFromFile` should output its elapsed time (in seconds, with three decimal places).

`EncodeFile(string InFile, string OutFile)` - Takes supplied file names and encodes the data in `InFile`, placing the result in the `OutFile`. Should also check to make sure `InitializeFromFile` has been run. Should print input / output byte counts, and compute the size of the ENCODED file as a percentage of the size of the ORIGINAL file (level of compression). Also display the elapsed time (in seconds, with three decimal places)

`DecodeFile(string InFile, string OutFile)` - Takes supplied file names and decodes the `InFile`, placing the result in the `OutFile`. Should also check to make sure that `InitializeFromFile` has been run. Should print bytes in / bytes out count, as well as the elapsed time (as above).

Your `main()` function, therefore, should:

1. Call `InitializeFromFile`
2. Encode `Shakespeare.txt`, placing the encoded data into a file called `Shakespeare.enc`.
3. Then decode `Shakespeare.enc` into `Shakespeare.dec`.

From a command prompt, use `fc/b` to verify that `Shakespeare.txt` and `Shakespeare.dec` have *exactly* the same contents.

Your `main()` function may look something like this:

```
int main()
{
     Huffman T;
     char WaitChar;
     T.InitializeFromFile("c:\\Shakespeare.txt");
     T.EncodeFile("c:\\Shakespeare.txt", "c:\\Shakespeare.enc");
     T.DecodeFile("c:\\Shakespeare.enc", "c:\\Shakespeare.dec");
     cout << "Processing finished.  Press ENTER to exit\n\n";
     cin.get(WaitChar);
     return 0;
}
```

You will need to do stream file I/O for this assignment. See the Savitch text and the Visual Studio documentation on file streams.

A VERY quick primer on stream file I/O in C++:

We can open a file as a stream of bytes (perfect for this application). For this assignment, you may want to `#include` all of the following (although they're not ALL required for stream I/O:

```
#include "stdafx.h"
#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>
#include <limits>
```

To open a file from which you can read a stream of bytes, the file needs to be an *input stream* (there's also *output stream* to which we can write). Declare the file variable as being of type `ifstream`:

```
ifstream inputFile;  // create an input stream called inputFile
```

The `ifstream.open` method requires a file name (as a string), and a file *mode*. The KIND of string it requires is a different string representation than we've been using. The kind of string it needs is called a C String, or `c_str` for short. You must convert your string argument to a `c_str` to use it here:

```
void HuffmanTree::InitializeFromFile(string FileName)
{
      …
      ifstream inputFile;
      inputFile.open(FileName.c_str, ios::binary);
```

The `ios::binary` mode tells the system to not translate any of the data it reads (CR/LF, skip nulls, etc.); without this flag, your results will be off.

The `fail()` method will tell you if the file failed to open, so you should check `inputFile.fail()` for a successful opening before you proceed.

Once your file is open, you can read it a byte at a time:

```
char symbol;
inputFile.get(symbol);
```

If you have attempted to read BEYOND the last character in the file, executing the `.get()` method again will set the end-of-file flag (at which point you want to stop, of course, and close the file, with `inputFile.close()`). **Note: the end-of-file flag is NOT set when you read the last character; it is set when you try to read another character AFTER the last one.**

```
while (!inputFile.eof())
{
      … process characters
}
```

Just as we use `get()` to read single characters from the file, we use `put()` to write a single character to a file (opened on an `ostream`, rather than an `istream`):

```
char outChar;
outputFile.put(outChar);
```

As with all of your programming assignments, all code you write for this assignment must be yours and yours alone (except for what is in the lecture slides and the Weiss text, of course). You may *not* use any code from any other source, including the Internet, even as a reference. Using code other than what you write (or are explicitly permitted to use) will be considered academic dishonesty, and will be dealt with in most severe terms.

Your code must be well-documented:

- Use block comments at the start of each method, briefly explaining what it does
- Use line comments liberally to explain what's going on
- Use internal documentation, like descriptive variable names where appropriate
- Make SURE you have a suitable block header on your source file WITH YOUR NAME!

Some helpful starting points:

- Create your program as a Win32 Console application within VS 2015
- Your program should use the `std` namespace; NOT the `System` namespace
- To get access to `cin` and `cout`, use "`#include <iostream>`"
- To get access to the string class and its supporting code, "`#include <string>`"
- To get access to `INT_MIN` and `INT_MAX`, the smallest and largest integers there are, "`#include <limits>`". One or both of those may be useful to you.

DO NOT WAIT TO GET STARTED ON THIS – This is a <u>much</u> more ambitious project than the BST lab was. You will write more code, and the code will be more intricate, and therefore, harder to debug.

Because you will have to write the encoder and the decoder "blind", it will be hard to know what the problem is if it doesn't work just right from end-to-end. If your decoded file doesn't match the original file, it will be hard to tell whether the encoder or the decoder is the problem. That's why it's a good idea to hand-encode the first few characters. If the first 30-40 bits in the encoded file are good, then the problem is probably in the decoder.

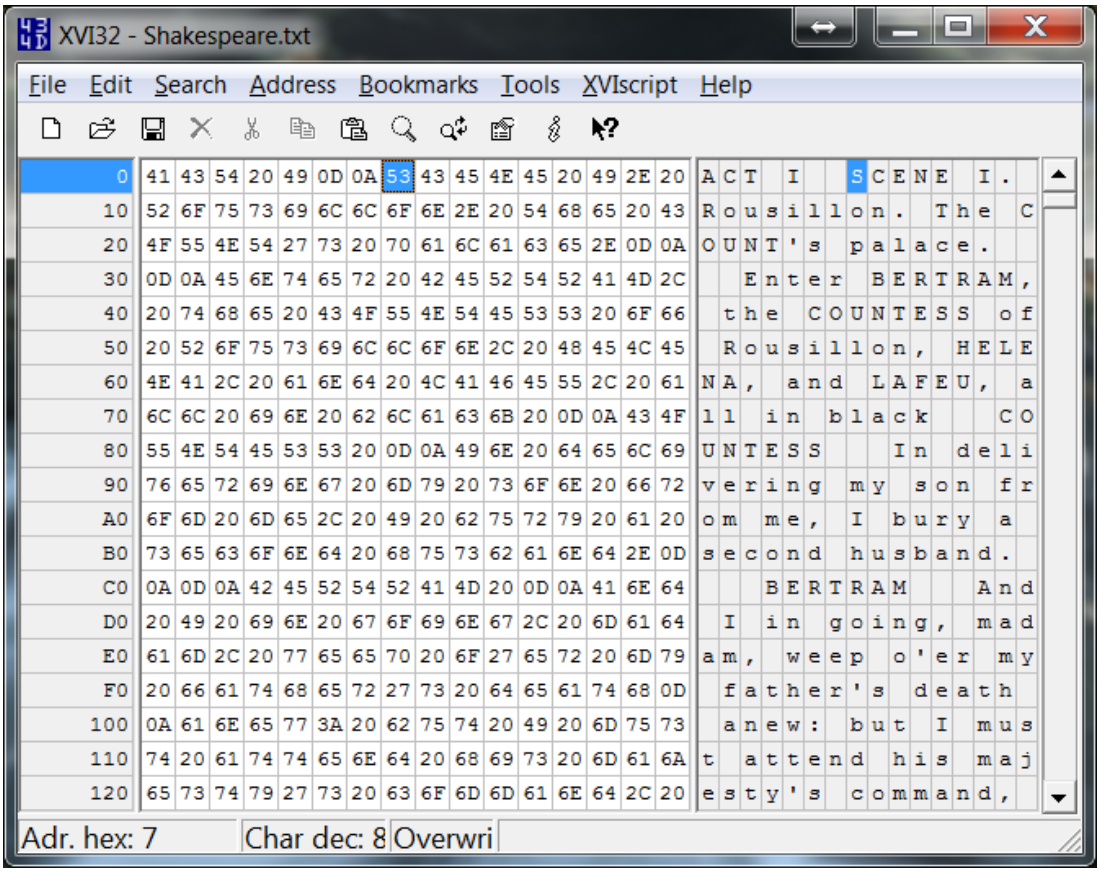
IF YOU RUN INTO PROBLEMS, AND HAVE MADE A GOOD ATTEMPT AT SOLVING THEM, E-MAIL ME, RATHER THAN LOSING TIME LOST IN THE WILDERNESS!

This program should probably be written in the order you need to process things. Start with the initializer, as you can't do ANYTHING until you have the frequency counts, use them to build a tree, and then traverse the tree to build the character string table.

Next, write the encoder, and finally, write the decoder. You should probably have some utility routines to tell you what the tree structure is, what the character encoding strings are, whether a node is a leaf, etc. As mentioned above, you should also hand-encode the first few characters ("ACT I"), so you know what to look for in your ".enc" file. This requires you to have the character encoding strings, of course.

A hex debugger might be helpful. I like XVI32 (it happens to be free). You can download it here: http://www.chmaas.handshake.de/delphi/freeware/xvi32/xvi32.htm. If you want to see what else is available, check Wikipedea for "comparison of hex editors". There are LOTS to choose from.

Besides its price ($0.00), XVI has some features that make it particularly helpful for projects like this, where we need to examine the contents of a file at the byte (and bit) level. If you open Shakespeare.txt in XVI32, you get a display like this:

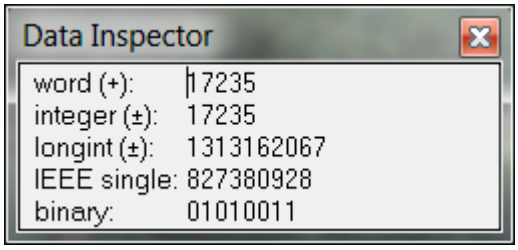


You can drag the window's left/right borders to make it wider or narrower, but I recommend using a width of 16 (hex 10).

The left side of the XVI32 window shows the file's contents in hex, and the right panel shows the contents in ASCII (when the characters are printable).

In the screen shot above, I have clicked on the 8th byte of the file (offset 7), which has a hex value of 0x53, representing the ASCII character "S".

Although you should be readily able to take the hex value 0x53 and go immediately to its binary equivalent of 0101 0011, XVI32 provides a tool to help. From the "Tools" menu, select "Data Inspector", and a small window pops up showing us the contents of this byte:



[Note: the 17235 represents the 16-bit integer stored as two consecutive bytes 0x4353, and the 1313162067 represents the 32-bit integer 0x4E454353]

If you are new to this kind of programming (manipulating individual bits and bytes), you might be tempted to "go fishing" and see what facilities the C++ library provides to help with getting this all done. DON'T. YOU ARE NOT ALLOWED TO USE BITSETS, VECTORS, OR ANY OTHER C++ TEMPLATE LIBRARY FEATURES.

You should not need anything beyond the creation of nodes (a la the Binary Search Tree project – for creating the Huffman tree), simple strings, arrays of nodes, arrays of strings (the character encoding strings), and arrays of ints (for the character frequency counts).

When it comes time to encode / decode the file, DO NOT convert the whole file into a single string of "1" and "0" characters – the 5 MB Shakespeare file would become a 40 million-character string, and your performance (not to mention your memory requirements) will be atrocious.

There are LOTS of variables that can influence run time. For this program, on my PC, using a Flash Drive, this program ran in just under 30 seconds. When I ran it with the input and output files on the hard drive, it took about 12 seconds. Enabling some particular speed optimizations in the compiler, I was able to get the run time down to about 7.5 seconds. I don't want you to worry about chasing any particular run time, but if your code is taking 5 minutes to process the Shakespeare file, that should be an indication that you have missed something in your code.