Lab 10: Huffman Coding last modified by Aaron Bloomfield on November 22, 2013 12:46:03 AM EST

Laboratory 10: Huffman Coding

Week of November 25, 2013

Objective

1. To become familiar with prefix codes
2. To implement a useful application using a variety of data structures
3. To analyze the efficiency of your implementation.

Background

In lecture we discussed Huffman coding and the construction of prefix code trees. We have also covered a variety of data structures this semester: lists, trees, hash tables, and heaps. Several of these may be useful for this lab. In addition, in this lab you are required to think about the underlying representation and efficiency of these structures.

Reading(s):

- Class notes on Huffman trees are available on Collab.
- Huffman compression and decompression is covered in Weiss section 10.1.2.
- Weiss code for binary heaps is also available on the class website. You may use/modify this code if you wish, or implement your own heaps, but you may NOT use the STL priority_queue class (you can use other STL classes – i.e. non-heap related classes). IF YOU ARE GOING TO USE THIS CODE, you should really remove the templates. It will save you a heck of a lot of time.

Lab Procedure

Pre-lab

1. The pre-lab is the compression phase of the Huffman coding.
2. You may NOT use a STL class for the heap (such as priority_queue), but you may use other STL classes (i.e. non-heap related classes).
3. Implement the Huffman encoding routine discussed in the pre-lab section.
4. Your program must compile with make!
5. Your program should only take in one command-line parameter!
6. Files to download: fileio.cpp, and the example files (in the lab10/examples directory, or as one examples.zip file)
7. Files to submit: heap.cpp, heap.h, huffmanenc.cpp, Makefile (you can submit additional .cpp/.h files, if needed, as long as it compiles with 'make')

In-lab

1. The in-lab is the decompression phase of the Huffman coding.
2. Implement the Huffman decoding routine discussed in the in-lab section.
3. Your program must compile with make!
4. Your program should only take in one command-line parameter!
5. Files to download: inlab-skeleton.cpp, and your pre-lab files
6. Files to submit: huffmandec.cpp, Makefile (you can submit additional .cpp/.h files, if needed, as long as it compiles with 'make')

Post-lab

1. Complete the post-lab report, as described in the post-lab section.
2. Read Tutorial 10: Objective C, as described in the post-lab section.
3. ~~Write the Objective C program described in the post-lab section, which implements a singly-linked list.~~
4. Files to download: ~~helloworld.m~~.

5. Files to submit: postlab10.pdf, ~~linkedlist.m (no Makefile for this lab!); if your program uses multiple .m or .h files, that's fine~~.

Huffman Encoding and Decoding

The basic steps in compression (for the pre-lab) are:

1. Read the source file and determine the frequencies of the characters in the file.
2. Store the character frequencies in a heap (priority queue).
3. Build a tree of prefix codes (a Huffman code) that determines the unique bit codes for each character.
4. Write the prefix codes to the output file, following the file format above.
5. Re-read the source file and for each character read, write its prefix code to the output, following the file format described herein.

We are also writing additional information to the file (compression ratio and tree cost), as described herein.

The basic steps in decompression (for the in-lab) are:

1. Read in the prefix code structure from the compressed file. You can NOT assume that you can re-use the tree currently in memory, as we will be testing your in-lab code on files that you have not encoded.
2. Re-create the Huffman tree from the code structure read in from the file.
3. Read in one bit at a time from the compressed file and move through the prefix code tree until a leaf node is reached.
4. Output the character stored at the leaf node.
5. Repeat the last two steps until the encoded file is finished.

Huffman compression and decompression is covered in Weiss section 10.1.2, as well as the lecture notes.

Requirements

Assume that only printable ASCII characters will occur in the source (original, uncompressed) text file. That is, your program should ignore newlines and tabs, but should not ignore spaces – thus, spaces need to be encoded, just like with other (printable) characters. Your program should be case-sensitive (count upper-case and lower-case versions of the same letter as different characters). If you look at the ASCII table in the Huffman coding lecture notes, you want to be able to handle all of the characters in the red box.

You ***must*** use a heap (priority queue) data structure to receive full credit on this pre-lab. You may use heap code from the textbook or elsewhere, but must give credit for code you use. You may NOT use the priority_queue class from the STL, but you may use other classes from the STL (i.e. non-heap related classes).

Real Huffman encoding involves writing bit codes to the compressed file. To simplify things in your implementation, you will only be reading and writing whole ASCII characters the entire time. To represent the zeroes and ones of the bit codes you will write the characters ‘0’ and ‘1’ to your output file. Yes, this means that the actual size of our compressed file will be larger than our original file (terrific, you “compressed” the character ‘t’ into the five characters “01011”), but it will simplify the lab considerably.

Hints/Common Mistakes

Implementation. You will need to select several different data structures to implement Huffman compression and decompression. Don’t get more complicated than is necessary, but do keep efficiency in mind. Consider the following questions:

- What will be the most common operations required on each data structure?
- How much time and space will be required?

Use the answers to these questions to guide your selection. Your solution will be judged slightly on how efficient it is (both in terms of time and space). Very inefficient solutions will lose a few points. Be sure you have a good explanation for your implementation choices:

- Do you predict better data locality?
- Does a purely Big-Theta analysis not tell the whole story?

Whatever your implementation, you should be able to accurately describe its worst case Big-Theta performance both in running time and space (memory) usage.

Serializing Data Structures. One thing we have not dealt with in previous labs is serializing various data structures (writing them to a file or standard output, and reading them back in). You will need to do this with your prefix code tree. The fact that you must read this data structure from a file and write to standard output may affect how you choose to represent it in your program. Keep in mind that the format for how to write it to a file is fixed, as described below (in the pre-lab section). You will need to re-create your Huffman coding tree from the first part of the file format described below.

operator<(): If you are creating a HuffmanNode (or similar) with an operator<() method, make sure that method is const (both in the .h file and in the .cpp file).

Pre-lab

For the pre-lab, you will implement the Huffman encoding algorithm using a heap. In particular, you will need to implement all of the basic compression steps described later in this document. You may use and modify code from the textbook (posted on the Collab website in /misc/), any code you generated, but you may NOT use the STL heap class (called priority_queue) – you can use other (non-heap related) classes from the STL.

Your program must take in a single command-line parameter, which is the name of the file whose contents will be encoded. We have some sample plain text and encoded text files on the Collab site – a description of these files is in the in-lab section. Keep in mind that, as described below, your program will need to ignore newlines and non-printable characters in the input file.

Your program should output to the standard output (i.e. cout) the exact file format described below, and nothing else.

As part of the file format described below, your program will need to print out the compression ratio and the Huffman tree cost. The compression ratio is defined, in bits, as: (size of the original file)/(size of compressed file). You should exclude the size of the prefix code tree in the compression ratio, and assume that the 0's and 1's you generated for the compressed file count as one bit each (rather than an 8-bit character). The Huffman tree cost is the same as described in lecture.

Your program should be in three files: heap.h, heap.cpp, and huffmanenc.cpp (but may contain more, depending on your implementation). The heap.h and heap.cpp will contain the heap implementation, and the huffmanenc.cpp will contain the main() method and the other Huffman encoding methods needed by your program. We'll be calling 'make' to compile your program, so make sure your Makefile works as well.

To read in the input file character by character, see the fileio.cpp file available in Collab.

File Format

In an effort to both ease the in-lab implementation (as we can thus provide examples) as well as ease the grading, there is a very strict file format for a Huffman encoded message for this lab. This allows the two parts to be developed, implemented, and tested separately. Although real Huffman encoding uses bits, we will write these bits to a file using the characters '0' and '1', as that will make it easier to check and debug our code. The challenging part of reading in a file (which is done during the in-lab) is re-creating the Huffman coding tree. This is discussed in detail in the in-lab section.

The file format is as follows.

The first lines of the file are the ASCII characters that are encoded, followed by their bit encoding. Only one such encoding per line, with no blank lines allowed. The format for a line is the ASCII character, a single space, and then the '1' and '0' characters that are the encoding for that character, followed by a newline. The order of the characters does not matter. However, your code to read in the file must be able to handle the characters in any order. Keep in mind that the first character on this type of line can be a space, but cannot be a newline, tab, or a non-printable character. No line in this part can be more than 256 characters. You can safely assume that the data provided will be a valid Huffman coding tree (i.e. there won't be an internal node with one child, etc.).

Following that is a separator line, and is a single line containing 40 dashes and no spaces.

The next lines are the encoded message, using the characters '0' and '1'. You may separate these any way you would like using whitespace (space, tab, return) – thus, you can have one per line, if you would like. For the pre-lab, separating the encodings by spaces or newlines will make checking and debugging your code easier: having a space between each encoded letter, and a return between each encoded word, will help considerably. However, when the file is read in, your code MUST be able to read in a Huffman encoded message regardless of the whitespace formatting.

The next line is a separator, and is also a single line containing 40 dashes and no spaces.

The last few lines of the file are English text, which displays the compression ratio and the cost of the Huffman tree. This does not need to be read in by the decompression routines. As long as you output the required information, and it is easily understandable by a human, it can be in a format similar to (but not necessarily the same as) what is shown below. You can have additional information as well, as long as we can easily find what we are looking for (compression ratio and Huffamn tree cost).

For the in-lab, you can assume that we will not provide you with invalid Huffman file formats. You can safely assume that we will provide you with a few different valid file formats (a few such examples will be available on the Collab site for the in-lab).

The following is the Huffman file format for the example given on slides 11 and 13 of the Huffman encoding lecture slide set.

```
a 0
b 100
c 101
d 11
----------------------------------------
11 100 0 101 0 0 11
----------------------------------------
There are a total of 7 symbols that are encoded.
There are 4 distinct symbols used.
There were 56 bits in the original file.
There were 13 bits in the compressed file.
This gives a compression ratio of 4.30769.
The cost of the Huffman tree is 1.85714 bits per character.
```

Below is an equivalent version of the same file. Note that the characters are not in the same order in the previous example, the whitespace for the middle part is quite different, the English explanation in the third part says the same thing but in a different format, and the particular prefix codes are different (but note that the lengths are the same). Your in-lab code will need to be able to read in both of these files (as well as others posted on the Collab site). For writing your pre-lab, you should consider having a space or a newline between the Huffman encoded characters, as that will make your code easier to check and debug.

```

d 11
c 101
b 100
a 0
----------------------------------------
11
100
0 1 0 1 0 0 1 1
----------------------------------------
Compression ratio: 4.31
The Huffman tree cost: 1.85 bits per character
```

Note that your encoding does not have to exactly match – in particular, the bits that your program uses to encode it will depend on the implementation of your heap. So while your bits can be off, the number of bits for each character should NOT be different than the examples given.

To read in the input file character by character, see the fileio.cpp file available in Collab.

Hints

A few hints from experience with this in previous semesters.

- Just copying the Weiss code verbatim will not help you, as you need to *understand* what is going on - the code does not "compile out of the box." Either make sure you understand it, or write your own code.
- It will be **far** easier to put HuffmanNode pointers in your heap, rather than HuffmanNode objects (or whatever your Huffman tree nodes are called). If you put the actual objects in, then you are going to be running into problems with scoping issues, inadvertent calls to operator=(), etc. Stay with the pointers - it will save you time in the long run.

In-lab

Make sure your pre-lab code can read in any printable ASCII character, including spaces, but not newlines or tabs. If you look at the ASCII table in the Huffman coding lecture notes, you want to be able to handle all of the characters in the red box. Keep in mind that text files will always have a newline character (‘\n’) at the end of each line. Some text files might also have the carriage return character (‘\r’) at the end of each line as well. This is from the difference in text file formats between Windows and Unix (or Cygwin).

Your program should output to the standard output (i.e. cout) the *exact* file format described above, and nothing else.

There are additional examples of encodings and decodings on the Collab site. Keep in mind that the bits that your code generates for a given character do not have to match the bits that are shown in the examples (as this will depend on the implementation of your heap), but the number of bits per character DOES have to match.

We provide a number of sample files for you to test your code with. A brief description of each is described here. The 'normal' files are the English input. The 'encoded' files are the Huffman encoded files, following the file format described above. Except where indicated, the middle part of each encoded file (the digits '0' and '1') has a space is inserted between each letter from the original file, so that you can see which letter is encoded as which bitcode.

- normal1.txt / encoded1.txt: This is the first example from the lecture slides ('dbacaad')
- normal2.txt / encoded2.txt: This is the second example from the lecture slides, in the 'Huffman Encoding' section. This is the example that we built up the Huffman tree from. Note that the example in the slides had a newline character, which was replaced by the letter 'I' in these files.
- normal3.txt / encoded3.txt: This is a paragraph from 'Gadsby' (http://en.wikipedia.org/wiki/ Gadsby_%28novel%29), which is a novel that does not ever use the letter 'e'.
- normal4.txt / encoded4.txt: The first paragraph from a front page stories in the 27 November 2007 edition of the Cavalier Daily (the story is at http://www.cavalierdaily.com/CVArticle.asp?ID=31789&pid=1656) .
    - encoded4a.txt: This is the same encoding as the previous file (encoded4.txt), but with all spaces in the middle section of the file removed, so that it's just a very long string of '0's and '1's.

During in-lab, you will implement the decompression steps for the Huffman encoding. These steps are listed at the beginning of this lab document.

First, make sure that your code can read in the file - you can download the inlab-skeleton.cpp file, which can properly read in the encoded input files.

Next, focus on creating the Huffman coding tree. There are a number of ways to go about doing this – we present one such algorithm here. As you are (recursively) creating each node in the tree, you know the bitcode code so far to get to that node (remember that following a left child pointer generates a '0', and following a right child pointer generates a '1'). We'll call this bitcode-so-far the 'prefix' (as it is the prefix for all bitcodes below this node). If that prefix is one of the listed bitcodes in the first part of the file, then we are at a leaf (remember that all characters in a Huffman tree are leaves), and the node will not have any children. Otherwise, we are dealing with an internal node – and you will have to create left and right child nodes, calling yourself recursively on each one. Keep in mind that when you call yourself recursively on the left child, you have to add '0' to the end of the prefix; likewise, you have to add '1' to the prefix for the right child. This algorithm will require you to search through the bit codes that were read in from the first part of the file. Also keep in mind that the size of the input here (the number of characters) is very small (only 80 or so) – which means that if you choose a linear time complexity data structure (vector, for example), your code will run just fine.

Not creating a Huffman tree from the file will result in zero credit for the in-lab. The whole point of this part is to create the tree!

Lastly, read in the second part of the file, transverse your Huffman tree, and output a character when you reach a leaf node. You can output as much text as you would like, such as status updates as to how the program is progressing. The only caveat is that the decoded file must be the last thing printed, and it must be clear where the other text ends and the decoded message that you are decoding begins (a separator of dashes would be fine for this). Of course, you are more than welcome to just print out the decoded message and nothing else.

You will need to get the Huffman decoder working when you submit the in-lab, as you will not be submitting it again for the post-lab. If you cannot get it working during the in-lab time, then you should request a lab extension, and submit what you have done so far.

As with the pre-lab, you should ensure that those files compile successfully with make.

Post-lab

For the post-lab, we want you to do a time and space complexity analysis of your compression and decompression code. You'll submit a written report that describes your implementation choices, and also documents your analysis of time and space complexity. See the section below for a discussion about the space/time complexity.

The deliverable for the post-lab is a PDF document named postlab10.pdf. It must be in PDF format! See How to convert a file to PDF for details.

The Huffman decoding code was submitted for the in-lab, and won't be submitted for the post-lab. The pre-lab code was be submitted for the pre-lab, and also won't be submitted for the post-lab.

A written post-lab report (a page is fine) that includes the following. You can double-space your report, but no funky stuff with the formatting (standard size fonts, standard margins, etc.). And if you double-space your report, you need to increase the number of pages appropriately.

1. A description of your implementation. Describe the data structures used in your implementation and *why* you selected them.
2. An efficiency analysis of *all steps* in Huffman encoding/decoding.
    1. For each of the steps of compression and decompression (see "Huffman Encoding and Decoding"), give the worst

case running time of your implementation.
2. In addition, give the worst case *space complexity* (i.e. how many bytes of memory are used in each data structure) of your implementation.

Time and Space Complexity

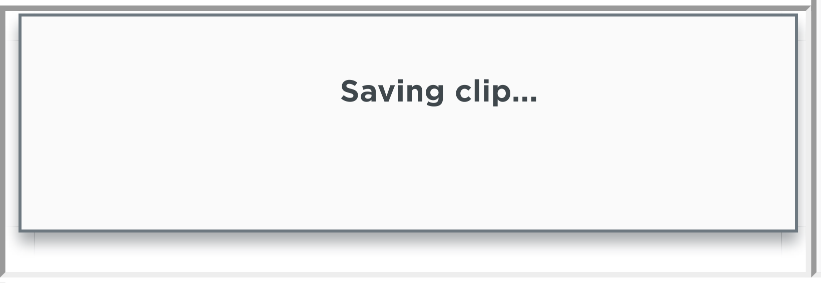
Worst case running time – for this be sure to include all steps of the compression and decompression. You can leave off the cost of calculating the compression ratio, printing the cost of the tree, and printing a listing of the bit code for each character that was asked for in the pre-lab. Refer to the list of steps given earlier in the lab.

Space complexity – for this, you should calculate the number of bytes that are used by each data structure in your implementation. The easiest way to do this is to step through your code, just as you have done for the worst case running time, and make a note each time you use a new data structure. You do not need to take into account scalar variables (loop counters, other singleton variables), focus on the data structures whose size depends on values such as the total number of characters and the total number of unique characters, and use those values in your answer.

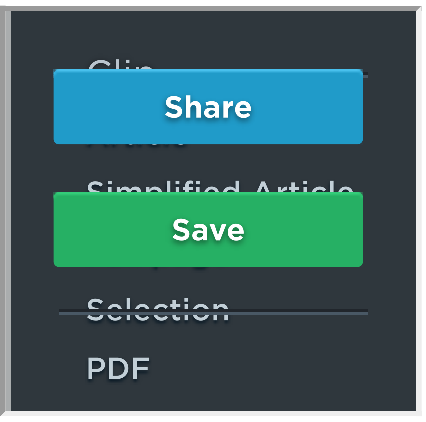
~~Objective C~~

~~Read though, and complete, Tutorial 10: Objective C; you will need to download helloworld.m.~~

Be the first to comment


Saving clip...

To:


Clip
Share
Simplified Article
Save
Selection
PDF