

CS 216

Laboratory 8:

Huffman Coding

Objective:

To 1) become familiar with prefix codes, 2) implement a useful application using a variety of data structures, and 3) 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.

Overview of this Lab

For the **pre-lab**, you will implement Huffman encoding and decoding, using a heap. When you come to lab, you'll demonstrate this to the TAs, along with some fairly simple additions to your code that print the table of characters their codes, the cost of the tree, the compression ratio, etc. We also want you to do a time and space complexity analysis of your code before you come to lab. (See details later in this document.)

For the **in-lab** portion of this lab, you'll see how the make utility is used to control compilation of multi-file C++ programs.

For the **post-lab**, you'll submit the program you did for the pre-lab along with a written report that describes your implementation choices and also documents your analysis of time and space complexity.

Note: you can work in groups of two, as long as your partner is in your lab section. This assignment lends itself to working in pairs – there are separate tasks that can be done in parallel and then combined.

Huffman Encoding and Decoding

The basic steps in compression are:

- Read the source file and determine the frequencies of the characters in the file.
- Store the character frequencies in a heap (priority queue).
- Build a tree of prefix codes (a Huffman code) that determines the unique bit codes for each character.
- Write the prefix codes or code tree to the output file.
- Re-read the source file and for each character read, write its prefix code into the output file. (You must WRITE the prefix code/tree and the encoded file to the SAME output file).

The basic steps in decompression are:

- Read in the prefix code structure (tree or array) from the compressed file. (do NOT re-use the tree currently in memory).
- Read in one bit at a time from the compressed file and move through the prefix code tree until a leaf node is reached.
- Write the character stored at the leaf node into the decompressed file.
- Repeat.

Huffman compression and decompression is covered in Weiss section 10.1.2.

Requirements

Assume that only printable ASCII characters will occur in the source (original, uncompressed) file. That is, you should NOT have any newlines or other unprintable

characters in any file you attempt to compress. Your program should be case-sensitive (count upper-case and lower-case versions of the same letter as different characters) and should handle spaces.

You must use a heap (priority queue) data structure to receive full credit on this lab. You may use heap code from the book or elsewhere, but must give credit for code you use.

Real Huffman encoding involves writing bit codes to the compressed file. To simplify things in our implementation, we 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

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. What will be the most common operations required on each data structure? How much time and space will be required? Use these 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. Just be sure you have a good explanation for your implementation choices – do you predict better data locality? does a purely Big-Oh analysis not tell the whole story? Whatever your implementation, you should be able to accurately describe its worst case big-Oh performance both in running time and space (memory) usage.

One thing we have not dealt with in previous labs is serializing various data structures (writing them to a file and reading them back). You will need to do this with your prefix code tree. The fact that you must read and write this data structure to and from a file may affect how you choose to represent it in your program. Since memory addresses (pointers) cannot easily be written to a file and then read back in, you may want to use a tree representation that does not make use of pointers and dynamic memory. There are many ways to handle this. Using array indices rather than pointers is one possibility. Another possibility is to do a recursive traversal and write out each node together with two extra pieces of information: something that identifies its parent, and whether it's a left or right child. (You'll need a special value for the root, which has no parent.) Then when reading each node back in, you find the node's parent in the tree you're building and then insert the node as the left or right child.

It is acceptable to write either the Huffman tree or the Huffman code to your output file (you should write this tree/code to the SAME file that contains your encoded message). Writing the letter frequencies and having your decode program reconstruct the prefix tree from that will NOT get full credit.

A *common* error students make (that results in losing points), is they re-use the prefix tree created for encoding (and currently sitting in memory in some data structure), when decoding. You should instead be using the prefix tree/code read in from the encoded file during the de-code step. That is, you should be able to 1) run your program and encode a file, 2) quit your program, 3) restart it, and then decode the encoded file generated in step one. You will not get full credit for re-using the code/tree in your program from step 1 to decode the file without having read the tree/code back in from the encoded file.

Space and Time Complexity

- 1) 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 on page 2 of the lab.
- 2) 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 thru 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 like - total # of characters, total # of unique characters, and use those values in your answer.

Procedure

Pre-lab

You may work in partners (2 people max, must be in same lab section):

0. Implement Huffman compression and decompression. Note you **must** use a heap in your implementation.
1. Modify your implementation to calculate and print the “compression ratio.”
 - Defined (in bits): (size of the original file)/(size of compressed file)
 - Exclude the size of the prefix code tree in the compression ratio.
 - 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).
2. Modify your implementation to print the cost of the Huffman tree as described in class. (see the course slides)
3. Modify your implementation to print a listing of the bit code for each character.
 - For example, “a 01, g 10110, t 11,” etc.
4. Calculate the worst case Big-Oh space and time complexity of all parts of your algorithm (compression and decompression) BEFORE coming to lab.

In-lab

Come to lab on with one printed copy of the checkoff sheet per group and a completely working pre-lab. If for some reason you don’t have your code ready to demo, please bring another multi-file application to use for the in-lab activities. Your hashing or AVL code would work for this.

0. Demonstrate your Pre-lab to a TA using the TA’s test files.
 1. Both partners will need to answer questions about your implementation.
1. <TO BE ANNOUNCED IN LAB>
 2. Lab Activity involves UNIX; you will need access to your BLUE.UNIX account.
2. **TURN in YOUR CHECK-OFF sheet to the TA BEFORE LEAVING Lab.**

Post-lab

Only one copy of the Post-lab – code and report needs to be turned in per group.

0. Please submit the following **electronically via the toolkit** before 11:59 PM on Friday, December 9, 2005.
 3. A complete listing of the code required for Huffman encoding and decoding.
 - a) If you used code from the book (and didn’t modify it) then you do not need to turn it in, but clearly indicate this is what you used.
 - b) Note that we will be evaluating your code based on style and correctness.
 4. A written post-lab report (2-4 pages) that includes:
 - 1) A **description** of your implementation.
 - a) In 1-2 pages, describe the data structures used in your implementation and why you selected them. (please use text, pdf, or ps format)
 - 2) An **efficiency analysis** of all steps in Huffman encoding/decoding.
 - a) For each of the steps of compression and decompression (see “Huffman Encoding and Decoding”), give the worst case running time of your implementation.

[see next page !!]

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

*If you did not get elements of the in-lab test sequence checked off during lab, you may attempt to receive partial credit for these by successfully implementing those elements after lab and submitting your corrected code at turn-in. In order to receive these points you must submit in addition to your code, a paragraph (**text file only** please) describing what difficulties you encountered getting your code working and what you did to solve them.*

CS 216: Program and Data Representation Checkoff

Date:	Circle: Your 3 Tue 12:15 – 2 PM Section: 2 Tue 7:30 – 9:15 PM
My <u>Email</u> Address: (UVa ID please, no aliases)	My Name:
Partner's <u>Email</u>:	My Partner's Name:

<u>Lab 8, Part 2: Huffman Coding</u>		
ONLY ONE CHECKOFF SHEET PER GROUP		
	<u>Partner 1</u>	<u>Partner 2</u>
Demonstrated Huffman encoding and decoding of TA's test file.		
Answered question about your Huffman implementation		
Explained space and time complexity of compression and decompression.		
UNIX Makefile activity		

Partner 1 Pledge:

Partner 2 Pledge: