Assignment 6, Huffman Encoding Review Session:

Wednesday, November 14, 2011. CS106B Autumn 2012.
Sophia Westwood (sophia@cs.stanford.edu)

Introduction:

- Due Wednesday, Nov. 28 at 5:00pm (the Wednesday after Thanksgiving)

Story:

- 1951: David Huffman was in an information theory class where the professor gave each student the option of either taking a final exam, or writing a term paper about the problem of finding the most efficient binary code. Huffman was about to give up and start studying for the exam, since he was unable to prove any codes were the most efficient. Then he came up with the idea of a binary tree sorted by frequency, the Huffman tree! And he proved it was the most efficient.

Motivation:

Priority queues, trees, pointers:

- Awesome. Also used as part of the actual algorithm used to compress ZIP files and PNG images, called DEFLATE.

Morse code:

- Another example of a variable length encoding
- The more common letters take less time to transmit:

  E . (1 unit)
  T _ (3)
  A . _ (5)
  X _ . . _ (11)
  Q _ _ . _ (13)
  Z _ _ . . (11)
- Take this idea to the extreme and you get Huffman coding.

Information theory:

- How much information does it take to represent some amount of data?
- A fascinating subfield of CS with connections to programming language design, signal processing, and cryptography, among other things

Sample run:

- How does Huffman do on The Road Not Taken? First five chapters of Pride and Prejudice? What about a png image of a puppy?
  - What would you expect to see be the most and least compressed of these?

In-depth look at compression:

(See the Huffman handout for pictures of this process, if you're reading online.)

Encoding "abracadabra": Building the Huffman tree:

a:5, b:2, r:2, c:1, d:1

- Make a node out of each letter, and stick these into a priority queue where the priority of a node is its frequency
- Now grab the smallest two from the priority queue. Create a new node whose children are a and b and whose priority is the sum of a's and b's priorities. Stick that node back into the priority queue.

c (1), d (1) -> cd (2)

- Now you have

a: 5, b: 2, r: 2, cd: 2

(c and d are now leaves for a blank node.)

- And repeat, combining binary trees as you go, until there is only one binary tree left that has all the letters.

a: 5, rcd: 4, b: 2

rcdb: 6, a: 5

rcdba: 11

- So in the end you get something like

r=000, c=0010, d=0011, b=01, a=1

  - Success! 'a' only takes one bit to encode.

Note that you could also get other valid encodings!:

- For example, r=110, c=1110, d=1111, b=10, a=0
- Why is this?
  - If there three elements with the same frequency (priority) in the pqueue, it does not matter which one we choose
  - After we dequeue two nodes, it does not matter which one is the left child and which is the right child
- What stays the same?
  - The highest frequency character requires the fewest number of bits to encode
  - The lowest frequency character requires the most number of bits to encode
  - If you take any character's bitcode, it is not the prefix of any other character's bitcode
    - This is what lets us unambiguously decode the file

Encoding "abracadabra": using the Huffman Tree:

"abracadabra" = 10100010010100111010001

- Just replace each character by its Huffman code.
  - Say going left in the tree is a "0", and going right is a "1". Then, the path from the root down to the character's leaf node gives us the character's Huffman code.
- 23 bits for what used to be 88! Not too shabby.

Writing a compressed file:

- There are two things we need during decompression in order for it to work: A way to build the huffman tree, and the encoded data. So, when writing the compressed file, we need to consider how to encode both of these into the file.
- First, let's add some extra information at the top of the file that will let us later reconstruct the tree.
  - Sadly, adding this extra header ruins our compression improvement for "abracadabra" because the original file is so small -- but, for larger files, the header information is tiny compared to the actual file. So no worries.
- So what should go in this header so that we can rebuild our tree when decoding?
  - You can either store character counts (with which you can rebuild the tree using the algorithm we just went over) or a representation of the tree itself. The first one tends to be easier.
- After writing the header to the file, write the encoding of each character one bit at a time
  - Do not put spaces between the bits, or between the encodings! We don't need them because of the way the Huffman algorithm constructs the bitcodes (try running through and decoding an example bitstring to see this for yourself)

Reading a compressed file:

- Reconstruct the Huffman tree using the header, and then simply walk through the encoded bits and convert them into their original values using the Huffman tree. As before, "0" can mean go left, "1" can mean go right. So, we process the encoded data one bit at a time, going right and left down the tree, and then whenever we reach a leaf, we write out that leaf value, go back to the root, and continue decoding the remaining bits until we decode the pseudo-EOF.

Stumbling blocks:

- ints and chars
  - As emphasized in the assignment handout, use ints instead of chars whenever possible. Ints can do everything that chars can, and avoid some nasty problems.
  - In particular, **Don't index into an array (Vector, etc.) with a char!!!**
    - details for the curious: chars are (usually) "signed" by default, which means that here they are only guaranteed to cover the range -128 to 127. So, attempting to use a character value greater than 127 will

wrap around and become negative -- and a negative array index is not good. (don't worry too much about this -- just use ints!)
    - To make things worse, Visual Studio breaks this convention, so PC users might not even catch this bug -- again, just use ints.
- More ints and chars
    - Make sure to get your file reading loop right in the compression part (ie, read into an int, NOT a char). (here it's usually Macs that are forgiving and PCs that crash)
        - EOF (the end-of-file marker) is unequal to any valid character code by definition (for good reasons!)
            - ... so what happens if you try to store EOF in a character? Don't. Use ints, so EOF is just a non-ASCII value.
        - Use chars and you could be in trouble.
- Psuedo-EOF
    - Problem: the bstream (bit stream) will always write full bytes, meaning 8 bits at a time. So, if we only write 13 bits during compression, bstream rounds up and writes 16 bits (2 bytes). So, we end up with 3 extra bits at the end that we don't want the decompressor to think are real bits.
    - Solution: invent your own end-of-file indicator that you put in during compression when you're done writing the encoded text. Then, when this indicator comes up during decompression, we know that we've decoded all the text and can ignore any remaining extra bits in the file.
    - What's a good choice for this end-of-file indicator, which we'll call our pseudo-EOF?
        - Use something that will never be a valid ASCII value -- since ASCII values range up to 255, that means that 256 is a good choice.
    - But we can't just write in the bits for "256" directly at the end of a file! After all, it is possible that our encoded text will also have some "256" bit string in it (remember that the encoded text is just a bunch of 0's and 1's that we generated with our Huffman encoding, and so it is not valid ASCII). The whole point of the pseudo-EOF in the first place was that it would be unique.
    - So, what we have to do is encode the pseudo-EOF as another entry in our Huffman tree (with frequency of 1). Then, we put the *encoded* pseudo-EOF (ie, the Huffman bitstring of the pseudo-EOF) as the end-of-file indicator. Now we're guaranteed that the pseudo-EOF bitstring is unique. During compression, we can check each of the decoded values to figure out when we've reached the pseudo-EOF (ie, when we decode a 256), at which point we can stop.
- Reconstructing the tree
    - As we saw above, multiple valid trees can correspond to the same set of character counts.
    - It's vital you recover the exact tree you used for compression -- any deviation will break the decompression.
    - If you process the same characters in a different order, you might get a different tree (bad times).
    - Be sure that you're always guaranteed that the Huffman tree during decompression will be exactly the same as the Huffman tree during compression

Final suggestions:

- Two handout suggestions that go really well together:
    - Write debugging functions that cout your structures (the private section of a class is a nice place to put these)
    - Test on small files (large files will bog down your console with debugging output)
        - But don't worry if the compressed versions of the really small files are actually bigger than the uncompressed version! Remember that we have the extra header data, and so the compression payoff will only show on the bigger files.
- As always, be sure to free any memory you allocate.
- Don't try compressing your cpp files! If your program has bugs...well. :(

- Questions?

Good luck! This is one of the more challenging assignments we assign, and most students struggle on some parts. Remember that the LaIR is available for help if you get stuck on a tricky bug and can't figure it out. It's also one of the cooler assignments, so try to get started early and have fun :)