//Robert Alexander, RMA3MZ

Post-Lab 10 Report

Implementation

For the implementation of the huffman encoding I used a combination of data structures: a heap with an underlying vector and a Huffman tree. I created a heap class, which stores Huffman nodes. Each Huffman node holds a character, the frequency that the character occurs in the file, the code that is associated with that node/character, and a left and right pointer (which are Huffman nodes). The heap then is able to insert and findmin to assemble a standard heap data structure. I then have a Huffman tree class. The Huffman tree class has a method (createHuffman) that takes in a heap and then forms a Huffman tree by forming combinations of the two smallest nodes (which are then organized into a standard Huffman tree). The tree class has two additional methods, one which prints the code to the screen using the new tree structure, and one which sets the code to each of the Huffman nodes in the tree.

In my actual Huffman encoding file I first create an array that holds frequency values in each of the positions (freq[128]). The actual positions in the array coordinate with the ascii values of each character, and the frequency that character appears in the file is stored as an int in that position in the array. The only characters that matter are between ascii values of 32 and 127, so I take that into account in my encoding file. I then create a new heap and insert a new Huffman node into the heap (the node's value is the position in the array and the frequency is the int stored in that position). I then run my Huffman tree methods to create a tree inside the heap and then set the code values for the nodes. I then use the underlying vector from the heap (which has been organized by the tree calls) and use that to match characters with the correct code.

For the decoding portion, the implementation was much simpler. I created a modified version of my pre-lab Huffman tree that creates a tree from a Huffman node, a string *code*, and a character that holds the letter for that particular node. The tree is then created and the encoded file is converted using the information from the tree.

Time and Space Analysis

For encoding we first start with a 128 integer array, which we move through linearly. This leads to a running time of $\Theta(n)$. Then while looping through this linearly we run an insert on the heap, which adds a running time of $\Theta(\log n)$. Then we move onto creating the tree. As we are constantly deleting elements from the heap, the time complexity is $\Theta(\log n)$. We also need to take into account that we are using a vector as an underlying data structure, so we could be causing a linear time if we pass 100 elements (though in practice this almost never happens). Then we call printCode, which runs linearly (since in the worst case it could iterate through *n* nodes). Also, we need to account for setCode, which also runs linearly. All in all this leads to a running time of $\Theta(n^4 \log n)$, however this number is much better when considering the average case.

Space Required encoding:

- HuffNode=13 bytes
  - Freq=int=4 bytes
  - Letter=char=1byte
  - Code size, uncertain
  - Left/right pointers=8 bytes
- Frequency array=512kb
  - 128 spots * 4 (since it holds ints)
- Heap=1300 bytes
  - Vector of size 100 that holds 13 byte elements=1300
- Since the tree actually just modifies the original heap, I do not use additional storage for the tree.
- Total required space: 1812 bytes

For the decoding phase, the space complexity is much simpler. We linearly read in the file, so this causes a running time of $\Theta(n)$. We also have a running time of $\Theta(n)$ for the creation of the Huffman tree. Thus the decoding running time is $\Theta(n)$.

Space Required Decoding:

- 1st huffman node=13 bytes
- c-string of 256 bytes
- 2nd Huffman node=13 bytes
- The tree's space is dynamically allocated depending on the situation
- Total required space: 282 bytes