Bamphiane Annie Phongphouthai

Bp8qg

Postlab9.pdf

November 22, 2017

Implementation. I decided to use the examples given in the pdr site as a guidance. Also, the vector heap given in slides as the implementation. I created a Huffman node class to act as the elements in the heap. The Huffman nodes had to contain a character, pointers, and frequency. The heap that is made from a vector implementation, I constructed a Huffman tree by popping off two elements then combining them together and creating the tree that way. I chose the vector implementation for the heap because to me it was a lot easier. Also, it provided a good idea of where the elements are in the vector, it is easier to find the terms that way. The Huffman tree works like a priority queue. Where the element with the most priority will be the next element to be taken off the queue. The vector gives me the freedom to look at what character occurred the most and encoding it by counting the steps down the tree. The direction of where the route was taken to that node had to also be taken into consideration. Left adding a 0 to the prefix and going right would add a 1 to the prefix. I used an array to keep track of the frequencies for the characters. This was easy because the total possible printable characters for the lab was 128, and it was an easy implementation to use loops.

Compression. To find out efficiency, I have to look at the encoding structures and look at the worse possibly run time case for each step. In the encoding, first I had to read the frequencies of the characters. That used an array, so the runtime would be $\Theta(n)$. That is the worst case for an array. Then, the source file had to be read in and frequencies had to be stored into the heap. This process is $\Theta(\log(n))$. Inserting into the heap is $\Theta(\log(n))$ because we have to do an insert then percolate up. Another reason is because frequency rates may be updated as a character may happen more often later in the reading. Once the frequencies are in the heap, prefix is determined for each character. The deleteMin() method has to run twice for each time the while loop is done, and then the percolate down has to also be called. Therefore, the worse case runtime is $\Theta(\log(n))$. The last step is to read the file again and print. This has to be a worse case runtime of $\Theta(n)$. It has to go through each element to print.

For worst space complexity. To do this I looked at elements stored in that data structure then multiplied it by the size of the element. For example, array holds 128 ints, each 4 bytes, so 512 bytes. Huffman node store a char that is 1 byte, frequency that is 4, a total of 2 pointers that is 4 bytes each, possibly a string of prefix that is 4 bytes (depends on number of characters in the string). So, it would be 13 bytes for each node. The heap is 13 bytes per node and vector is 100 of them, so it takes 1300 bytes. Total space would be about, 1842 bytes depending on string size.

Decoding Huffman, we need to read the prefix of the compressed file. This would just be $\Theta(n)$, because have to go through each element. Then create the Huffman tree similar to the phase of compressing the files, this would be $\Theta(n)$. It needs to go through and read in the bits, then create the nodes according the bit value. If a 0 is encountered, it either creates a node then

add a left pointer or, if the node exists, add a left pointer. Same goes for when a 1 is encountered, only instead of a left pointer, it would add a right. In turn, this would also take $\Theta(n)$.

For the space complexity, the char array contains 256 chars. A char is 1 byte. Hence the space for that is 256 bytes. The Huffman node used before contains 13 bytes each (same node used in the compression phase) and two are used. Thus, that is 26 bytes. Therefore, the total is 282 bytes.