Eza Rasheed

er6qt

04-18-19

postlab10.pdf

Implementation:

In my implementation for Huffman encoding, I created a heapNode class, a heap class, and a heapTree class. In the heapNode class, each node has a frequency of the amount of times that a character is present in the input file, a character that holds the prefix code for each letter in the input file, a left and right pointer to each node, and the prefix code associated with each letter. The heap class stores the heap nodes of the tree and percolates up and down the tree, find's the minimum value in the tree, and inserts the nodes. As for the separate heapTree class I made to create my tree and make and set the prefix table, my createHuffTree method takes in a heap as a parameter and creates a tree by combining and merging the two lowest(minimum) value nodes and my createPrefix method prints the characters with their associated prefix codes to the screen, while my serPrefix method sets the code of each of the heap nodes. The heap structure I used was a vector of heapNodes. The reason I used a vector is because you can easily add and remove nodes from it, it saves space as they are more compact than dynamic memory, and they work better with cache, which saves time. It is similar to an array, but more efficient as you can resize it and expand it to meet the need of your size requirements. As well, when creating a tree, a vector is preferable because when trying to find a specific element in the tree by mapping through it, a vector is good for finding where elements are located. Vectors allow us easy access to the most frequent characters so that we can encode the characters based on the bit values 0 and 1. This is why I chose a vector data structure on my heap so that it could store the nodes that I add or delete and form a tree from it. In my main Huffman encoding file, I used an array to store the frequencies of the characters being read in from the input file. This was set to 128, as there were 128 printable characters possible (ascii values 32-128). I then looped through each value in the array iteratively to make all the elements equal to 0 (empty array). Afterwards, I read the source file to determine the frequencies of the characters in the file and calculate the total number of letters in the input file. From there I created a heap and iteratively went through the array that holds the frequency of each element and inserted in the distinct (only occurring once) characters into the heap, and added the count for the number of distinct symbols. I then initialized a new to tree to call my createHuffTree method on, and then printed and set my prefix codes. This is what led up to building my tree of prefix codes to determine the unique bit codes for each character. Finally, in the last step for compression, when re-reading in the file, I created a vector-based heap that pointed to my getPreCode() function to successfully encode the characters in the input file and output it to the screen.

As for the implementation of Huffman decoding, it was less involved. I reused the heapNode class from the pre-lab to construct a new tree that recursively creates each node within it. I only needed to pass in a root node, a string to store the prefix codes, and a char that held each character. This enabled me to decode the file through various steps.

Time/Space Complexity Analysis

When encoding the Huffman tree, the first step in compression is to store determine the frequencies of the characters in the file. For this, I created an integer array that held the frequency values of the 128 printable characters (frequen[128]). I moved through this array iteratively, element by element, which is why the worst-case time complexity for this linear search is O(n). Then the second step is to store the character frequencies in a heap. When I looped through each element linearly in the array from 32 to 128, and inserted the characters into the heap, this has a worst-case running time of O(logn). The third step is to build the tree of prefix codes. For this implementation, I found the two minimum values in the heap and deleted elements from the heap. Because we had to perform deleteMin in this method multiple times for the implementation, this has a time complexity of O(logn) as well. The fourth step is to write the prefix codes to the output file. In creating the prefix codes and setting the prefix codes, the code runs linearly to iterate through the n number of nodes. So for both of these methods combined to be able to write the prefix codes, there is a time complexity of O(n^4logn). Lastly, to re-read the source file and write the prefix code for each character read in, it is linear O(n) running time has each character in the file is read in iteratively to be converted(encoded) and printed.

When decoding the Huffman tree, the first step in decompression is to read in the prefix code structure from the compressed file. This has a liner runtime O(n). Then, the second step is to re-create the Huffman tree from the code structure read in from the file. In order for this to be implemented, you have to iterate through each element(character), which results in a time complexity of linear O(n) as well. I did recursive calls to create my tree, which is also linear time. The third and fourth step is to read in one bit at a time from the compressed file and move through the prefix code tree until a leaf node is reached and output the character at that node. In order to do this, the tree is iterated through to decode the prefix codes and print out the characters, which is why these two steps also have a running time of linear O(n).

For the worst-case space complexity of the data structures for my Huffman encoding, I determined this by multiplying the number of elements stored in the structure by the size of the elements being stored in. So for my array that stored in the frequency of each character, it stored 128 elements that were of type integer, so 4 bytes. Therefore, 128*4 = 512 bytes. Each heapNode in a heap to create a tree has a frequency(int=4bytes), character(char=1byte), left and right node pointer(8 bytes each), and a string that hold the prefix code (string=?). This equals to 21+ bytes. The number of nodes stored in the vector-based heap data structure is varied based on what is being read in to create the tree, but assuming the worst case is a vector of size 100, that would be 100 multiplied by the size of each heapNode(21+bytes), so 2100 bytes. So, the total space for the Huffman encoding data structures would be ~2,633 bytes.

In Huffman decoding, the worst-case space complexity is determined by the one data structure I used: heapNode. As calculated before, each heapNode takes up ~21+ bytes, and since I used two heapNodes in my program to read and decode the prefix codes, the result is ~42 bytes of space.