Pat Wongwiset
nw9ca
11/21/17
postlab10.pdf

Post-lab 10 Report: Huffman Encoding/Decoding Analysis

On the encoding part, huffmanenc.cpp, the program read a “normal” file and store unique characters and their frequencies in two different vectors. Reading and collecting data from a file will be O(n) worst case runtime such that n is the total number of symbols in the file. Then, all symbols and their frequencies will be put in heapNode and then pass into a min-heap by looping m times such that m is the number of unique symbols and equals to the size of the frequency vector. Since heap is a complete binary tree, each insertion runtime to the heap is O(logm). Insertion m elements would take O(m*logm) runtime complexity. After that, I created a Huffman tree by makeTree() method from the heap. I wrote makeTree() method instead of makeTree() function because the Huffman tree holds the prioritization of min-heap properties, to create a tree. The tree popped up two prioritized heapNodes which have the smallest number of frequency, created an internal node which have a left and right heapNode as two previous node, inserted the internal node inside the heap again, continued popping two minimum heapNodes until the size of heap was one, not including heap[0] that is intentionally null. The runtime complexity in this step is O(m*logm) again since the tree was created by inserting m elements from the heap. I created readTree() method to print out a prefix code for each unique character for the first section of the pre-lab file format after the Huffman Tree was created. To encode the text for the second section, I used an unordered_map (hash table) to collect the prefix code as a ‘value’ while a ‘key’ is character. When printing out in the first section, the runtime complexity is O(m*logm) since it printed m elements and each element is printed at O(logm) runtime. When rereading, searching and encoding symbols in the file, the program improved time efficiency of encoding n elements to be O(n); each element has O(1) runtime for hash table. The implementation for the last section and each separation line are constant runtime complexity, and the program closed the file to secure unexpected adjustment of the file at the last line of the program.

Next, the program can decode an “encoded”file by the code in huffmandec.cpp. The program created a decoded Huffman Tree from prefix coded in the first section of the file by decTree() function. The runtime complexity in this step is O(m*logm) such that m is the number of the unique encoded character, and m means the same value as m in the first paragraph. Then, readTree() function, which is different from readTree() method, was created to decode each encoded symbol in the text of section two. The readTree() function passed prefix code and then printed out individual symbol until the end of the file**.** The runtime complexity of searching and printing out the decoded symbol is O(n*l) such that n is the total number of symbols and l is the longest length of the prefix code.

For worst case space complexity, the program use space = sizeof(frequency vector) + sizeof(char vector) + sizeof(min-heap) + sizeof(Huff tree) + sizeof(hash table) + sizeof(decoded Huff tree) = 4*m + 2*m + 8*m + 8*m + (8+8)*m + 8*m = O(m)

I used the Huffman tree structure to learn how file compression and decompression work. I think it is challenging when the program has to decode the code with no separator, and I am really curious how we can implement from the Huffman Tree for that. Also, this is interesting to learn more about how all data deleted to the ‘trash’ can be retrieved even after emptying the trash.