Maddie Stigler
mgs4ff
11/25/14
postlab10.pdf

In order to implement my Huffman encoding program, I used several data structures to make up the heap, tree, node, and to help unite these three in the huffmanenc.cpp file. For the heap structure, I built from Professor Bloomfield's binaryheap.h file and used a vector to store huffmanNodes. Each huffmanNode contained a frequency, character, left/right pointer, and a prefix code. From that point, each node was inserted into the vector through the insert method and percolated up, using Professor Bloomfield's percolate method. In addition, nodes could be deleted through the deleteMin() method and percolated down. The vector data structure was preferred because nodes could be added and removed. The dynamic aspect of a vector made more sense. I also used a tree class to build the tree of prefix codes. The tree took in a heap and created a tree of prefix codes by looping through the heap and merging the two min nodes. In the huffmanenc.cpp file, an array was used to store the frequencies of the characters being read in. This made sense because the number of possible characters was fixed at 128. In order to implement the decoding portion of the lab, I used my huffmanNode class again and used a tree function that takes a node, string, and char and forms a tree that is used to decode the file.

When examining the efficiency of all steps in Huffman encoding/decoding, I first go through my encoding structures to find the worst case run time of each step and the worst space complexity of all data structures used. In Huffman encoding, reading the source file and storing character's frequencies in an array results in a runtime of Θ (n). Storing the character frequencies in a heap results in a log(n) runtime because the insert function percolates up the heap data structure. After storing the frequencies in a heap, the prefix code tree is built to determine unique bit codes for each character. This also has a worst case run time of log(n) as the while loop performs the deleteMin() operation twice each iteration. This results in a downward percolation of the heap, thus resulting in a log(n) runtime. Finally, re-reading the source code, printing and setting the prefix codes have a worse case runtime of Θ (n) because they have to iterate through each value to print.

In analyzing the worst space complexity of data structures used in the encoding process, I look at the variable number of items stored in the structure and then multiply that number by the size of the item being stored. For the frequency array, the array holds 128 integers of 4 bytes each resulting in 512 bytes used. Each huffmanNode stores frequency(4 bytes), 1 char(1 byte), a left pointer(4 bytes), a right pointer(4 bytes), and a string (depends), resulting in at least 13 bytes per huffmanNode. The heap data structure is a vector of 100 huffmanNodes, each 13 bytes, resulting in at least 1300 bytes of space.

The decompression stage of Huffman decoding first begins by reading in the prefix code from the compressed file, resulting in a runtime of Θ (n). Next, it re-creates the Huffman tree from the code structure, iterating through each element to do this. This also has a runtime of Θ (n). Iterating through the prefix code tree until a leaf node is reached and outputting the character stored at that leaf has a linear runtime as well.

In decoding, the two data structures used are the huffmanNode from the encoding code and the array of 256 chars. The huffmanNode consumes at least 13 bytes and two huffmanNodes are utilized in the decode program thus resulting in 26 bytes. The array contains chars which are 1 byte each so the array consumes 256 bytes of space.