Derek Johnson, Dej3tc
11/28/19

## *Overview of Implementation*

For the encoding process we first needed to determine all the characters used and the number of times they appear. To accomplish this I used a std::map with a character mapping to an integer. For each character in the file, the map will check if it already has a mapping. If so, it will increase the integer associated by 1 and if it is a new int it will create a new entry and set the integer associated to 1. The map data type was chosen because it offers very fast lookup times and is easy to modify. In order to implement the Huffman coding process, we will need to create a min tree using the values in the map. I created a Min-Heap structure that will store and be able to order Binary Heap Nodes. The Min-Heap is implemented using a vector of node pointers. The vector data type was chosen because the list will need to grow and shrink. The Binary Heap Nodes all contain left and right pointers so that we can link them into a tree later. For each entry in the map, a Binary Heap Node was created to store its character value and frequency. The node is then pushed onto the heap using the insert method that orders nodes by frequency. The next step is to build the Huffman Tree. Because of the ordering property, this can be done by popping the first two values off the heap, making a new node to link them, and then pushing the new node back onto the heap until there is only one node left in the heap. For the final step, I used a new map with a character key and a string value. The map runs in O(log n) time, the same as searching through the tree we created but the stl::map class will store values in a balanced tree so it should run faster for lookups. I then recursed through the min-tree to fill the map with characters and there associated binary encoded value. Finally, using the map, every character in the text file was replaced with its binary value.

The decoding process was done by taking the prefix codes and characters and making a Binary Heap Node with that character value. An empty node was used as the root of a new min tree and the new nodes were then added one at a time to the tree. Node data type was the same as in the encoding portion of the lab. This eventually leaves us with a binary search tree in which every leaf node has a character value associated with it. The decoding process from here is relatively straightforward. I moved through the string of binary values, traveling left or right down the min tree. Once a node with a character value is found, the character is printed out and the process starts over from the top of the tree. This eventually yields a fully decoded message.

## *Efficiency Analysis*

**Encoding**

1) Read the source file and determine frequencies
    a. The average runtime for map access is O(1) but its worst case is O(log n). Since this needs to be repeated for each char in the file the run case is O(n log n).
    b. Space complexity is n * (size of map<char, int>)
2) Store the character frequencies in a heap

   a. An insert needs to be performed on every map element. Accessing the first member of a map runs in constant time. The insert method runs in O(log n). Since this needs to be repeated for each set in the map the time complexity is O(n log n).
   b. Binary Heap that has a vector of n pointers as well as n Nodes. n(8) + n (21)
3) Build a tree of prefix codes
   a. Two methods need to be performed every time, insert and deleteMin. Insert runs in O(log n) and deleteMin runs in O(log n). Since it needs to be performed for every element in the heap, this step runs in O(n log n)
   b. We will be using the same pointers and nodes as before but with the addition of empty nodes to fill in the tree. The number of these empty nodes will vary but has a max equal to the number of character containing nodes. The added space is then n(21).
4) Write the prefix codes to the output file
   a. The worst case run time for getting each element in the tree is O(n) as the tree could be stacked vertically. Writing each element to a <char, string> map and writing it to the output is done in constant time. Therefore the time is O(n^2).
   b. Size of each member of the map = 8(left*) + 8(right*) + 1(char) + x(string). Therefor the space = (17+x)n, n = number of leaf nodes in the tree
5) Re-read the source file and write its prefix code
   a. We are finding the encoded value of each character from the map created in the last step. This happens in O(log n) time and needs to be done for each char in the string. Run time is O(n log n).
   b. No added space

**Decoding:**

1) Read the prefix code structure and create the Huffman Tree (combination of steps 1-2)
   a. A new Node is created for each prefix code and a insertByBinaryCode method is run to push the node onto the tree. The worst case run time for insertByBinaryCode is O(n) so the total run time is O(n^2).
   b. Make 2n binHeapNodes where n=(number of distinct characters). Space = 2n(21)
2) Get the decoded value for each bit (combination of steps 3-5)
   a. Since you can just trace through the tree as you recurse over the encoded string, run time is O(n) (assuming n=encodedBits).
   b. No additional memory.