Adarsh Solanki as5nr
4/20/12
CS 2150 Post-Lab 10

Encoding

First the program iterates through the input file character by character and records the total number of distinct characters and also the frequencies of each character. I stored the count of each character into an integer array of size 128 to hold the possible ASCII letters indexed by their value. This requires an iteration through the entire file (linear time complexity of Theta(n) where n is the size of the file) and a constant array of integers (space complexity of Theta(1))

Next, I created a heap with size `count` (count being the total number of characters in the input file) iterated through the 128-integer array and for each non-zero value (meaning that the character was present in the input file at least once), I added it to the heap with a new Huffman Tree with the parameters of the character and the count. This implicitly created a root Huffman Node with the value of the character from the implementation of the constructor for Huffman Trees with a character and an integer. This is linear time/space complexity as a function of the number of distinct characters in the input file.

Then, in order to build the final Huffman tree with which to encode the document, I looped `count - 1` times deleting the minimum tree from the heap twice and calling a helper function `mergeTrees()` that created a new node with left and right pointers to the two argument trees to be merged and returned a pointer to the root node of the resulting tree.

A recursive printPrefixCode() method was used to print to console each of the distinct characters and their prefix code for the decoding step. It's body was simply a call to the encapsulated recursive subroutine `printrecurse(treeNode* root, string code)`. This routine had a base case of no left/right children (leaf) where the element(character) is printed to

the console followed by its prefix code and the element/count tuple is stored into a the getCode map stored in the Huffman tree. Otherwise, if there was a left child, the function was recursively called on the left child with a "0" appended to the code string, and a right child was the same but with a "1" appended instead.

Lastly, in order to get the encoded file, I iterated through the list and used the member map getCode to retrieve the proper prefix code string for each character. This was a linear time/space based on the length of the file.

Decoding

In order to subsequently decode the messages, the Huffman tree had to be built again and then traversed through the entire decoded file, printing out once leaf nodes are reached. More specifically:

First an empty Huffman tree is constructed with a root node that has all null data members/pointers. Next, the file is iterated through in order to scrape each character / prefix code pairing. For each pairing, a buffer is used to store the prefix code, and then a `while` loop is used to traverse the tree, going to `currNode->left` for 0's and right for 1's. If no node exists, a new one is created with null data members. When the buffer has one character left (the end of the prefix code), a new node is created with the character provided as the argument to the constructor for the `element` member. The non-terminal nodes within the reconstructed Huffman tree were left null as a check of sorts, as none of the members should be accessed under proper functionality. This was all computed in a space/time complexity linear with regard to the number of distinct characters in the source file.

Next, the compressed file is iterated through to create a string `bits` that holds the binary representation of the source. The helper function `parse(huffmanTree*, string)` is used to traverse the tree and perform most of the decoding. This function loops while the string still has characters and traverses the tree based on the leading character (left 0, right 1) and

removing the front character each time the node moves. When a leaf is reached, the `element` of the node is printed (containing the character it stands for) and the current node is switched back to the root. Once the target string is empty, the void function returns. This has time complexity of the length of the encoded file and constant space complexity.