Johnny Li

COP 3530

Project 3 Implement Huffman Algorithm

Commentary

My program firstly implements a single integer array in the ‘getFrequencies()’ in order to create a table to store the characters, as ASCII values in the index, and frequencies of those characters. This method only stores the necessary number characters since there a defined amount, 128, rather than having to use excess space in storing with a double integer array which would store the character in one column and the frequencies in the other column, allocating 256 spaces. A single integer array reduces the necessary storage memory by half and faster access to data, a more efficient method. A minimal heap was constructed in order to give quick access to the smallest frequency because that smallest frequency will always be the first element in the array list used thus the fastest search time. The benefits of using an array list were that the sized did not need to be defined prior the creation where the array list grows as more frequency values are placed in, thus keeping the memory space to the minimal. In the ‘buildTree()’, the Huffman Tree is constructed as a binary tree with the encoded characters stored at the leaves based on the frequencies of the characters. This allows for lossless data compression where variable-length codes are assigned to the input characters and lengths of the assigned codes are based on the frequencies of corresponding characters. A massive amount of information is compressed to a traceable code; therefore, it saves a large amount of space, allows for fast access to the character, and simplify of user implementation. Nodes were used to store the children position, character, and frequency and connected in the Huffman Tree. For the traversal of the Huffman Tree, a hashmap was implemented to efficiently locate a value based on a character key and inserting and deleting values based on a character key. This method has a computational complexity of O(1) when continuously accessing the codes when traversing or building strings for output thus saved time and space when used in encoding and decoding.

The computational complexity of the operations in the encoding is O(n) as every character obtained from the Illiad must be compared to the code from the index of the character and written be into a string. The opposite is true for the decoder as it is also an O(n) where every code inputted it must be searched in the hashmap to find the character and written be into a string. For the traversal, the program must follow the given path in the Huffman Tree to identify and print the character thus the computational complexity of the operations is O(n) as it might have to traverse an entire part of the tree to get a character out. The computational complexity of the overall operations of the Huffman Algorithm is O(nlogn) where there are n- character nodes and the ‘removeMin()’ would be called 2\*(n – 1) times to extract all the nodes. The ‘removeMin()’ takes O(logn) time since it calles ‘minheapify()’ which divides the nodes so, overall complexity is O(nlogn).

This project enhanced my conceptual knowledge of all the data structures learned in class, including concepts from hashmap and heap to Huffman Tree, and reinforced my applicational ability to implement this knowledge in code. I learned how to implement a Huffman algorithm, a lossless data compression occurs through a construction of a minimal heap where variable-length codes are assigned to the input characters and lengths of the assigned codes are based on the frequencies of corresponding characters. Then a Huffman tree is built to hold a coded path for each character based on the frequencies. The encoding process takes the characters and translates it to the code based on the hashmap while the decoding process takes the code and translate it to the character based on the hashmap.