605.202: Introduction to Data Structures

Claire Harelson

JHU ID: charels1

Lab 3 Analysis

Due Date: April 18, 2023

Dated Turned In: April 18, 2023

Lab 3 Analysis

For this project I utilized a heap data structure. A heap is useful in that you can structure it to be either a min or max heap, meaning that all of a node's children will either be larger or smaller than that node, respectfully. For this application, a min heap made sense to use as it allowed me to order my frequency data by putting the lower priority items at the top of the heap. This ordering would then take advantage of the recursive property of trees to return the items in preorder, resulting in the lowest priority items at the bottom of the tree. This ordering is beneficial for the use of Huffman encoding as it will set the lower frequency items to have longer codes and higher frequency items to have shorter codes, ideally resulting in compression of code.

To design my Heap class, I initiated a constructor along with values to initialize an array, the heap size (which I was able to leverage as an index), the left and right children, a parent node, and the root node. The methods of parent, left child, and right child were designed to take in the index of the current node and determine the forementioned attributes via their respective calculations and return the node of interest. The exchange method allowed for the swapping of two items at specified indices. The insert method allowed an item to be appended at the bottom of the heap (end of the array) and called to the percolate up method. This would then compare the newly added item with its parent node and swap their locations if its value was lower in order to maintain min heap properties. Similarly, I created a remove method to return the root node. Since you can't directly delete the root node, I stored this value in a temporary object and replaced the root node with the lowest priority item from the bottom of the heap. Then this would call to the percolate down method to swap this 'new' root node with its child node to maintain min heap properties. Percolate down called to a method called smaller child that was used to determine whether the current node's left or right child was smaller in value in order to accurately percolate down. Lastly, the traverse method returned the heap in preorder.

Unfortunately, I think I overcomplicated this project in my head and code, and therefore was not able to properly solve and complete the assignment. However, I did get relatively close to the solution and will describe my intended approach. My main file to run the program was set to intake three files, the frequency table data, an encoded text, and a decoded text, then write the desired output to an output file. I designed a function to parse the frequency table data in and create a min heap from it. This part was successful in that it produced a heap in which all child nodes were smaller in value than their parent node. I then employed a merging function to merge the individual nodes one by one until this resulted in one large node. This is where I had an issue as I wasn't able to merge in the proper order while incorporating the necessary tie breakers. The closest I could get was merging items in their min heap ordering. I then utilized my Huffman tree function on the merged items, which again, was wrong (pain). The intention of this was to produce a dictionary that matched each letter of the alphabet to their corresponding Huffman binary code. My approach resulted in the most frequently used letters having longer encodings, opposite of how this should have been. Additionally, I believe the range for the length of codes should have been between one and five digits, whereas mine ranged from one to twenty-six. Although these incorrect results would not yield a correct solution, I still designed encoding and decoding functions.

My encoding function was designed to iterate through the letters in a string and map them to the keys of my code encryption dictionary. If the letter and key matched, the letter would be replaced with the corresponding binary code. If my binary coding had been correct, this function actually should have worked properly! On the other hand, my decryption function is definitely not correct, regardless of the code encryption dictionary. I could only think of a method to iterate one item at a time through a binary string, where instead it should go over multiple numbers until it hits a corresponding binary code. Going though one by one will simply result in returning only the letters that have a code of 0 or 1. I’m assuming I needed to somehow use the Huffman tree function to complete this conversion of binary items to alphabetical letters, but obviously couldn’t figure out how.

Due to my inability to complete the base requirements of this lab, I wasn’t able to spend much time adding enhancements. I do include some error checking in terms of file IO in my main function. I also include a runtime metrics file designed to track the amount of time it took for the program to carry out its encoding and decoding functions.

I know due to my errors that I was not able to achieve any useful data compression as the incorrect mapping of characters to binary strings had the opposite effect of increasing my data size (oops). However, I think if this project were finished correctly that some data compression would have been achieved. I believe that on a small scale such as this that it likely does not make much of a difference in terms of time and space efficiency. Although on a much larger scale I do think that this method of encryption would result in quite useful data compression. I think that if a different approach was used for breaking ties using alphabetical ordering over the number of letters in a key that this would be slightly more efficient. In this case, only two items need to be compared. For the number of items to be compared, an additional step is needed to count the length of each letter grouping.

From this project I learned how to properly create and implement a heap object! Although I hope to be able to use the built in heap object moving forward, I do think this was a useful assignment to achieve a better understanding of the heap data structure. I didn’t quite grasp the Huffman encoding portion of this assignment but did get a little extra practice on creating binary trees. For my next project, my number one goal is to successfully complete it (ha ha)! In particular though I need to work on not overcomplicating my code. I think my brain gets pretty confused and sometimes this results in me trying to solve problems by adding more code, rather than trying to simplify existing code. I definitely need to spend some more time trying to understand searching and sorting for the next lab.

In terms of time efficiency with this program, I tried to measure the observed efficiency in nanoseconds of how long it took to convert each statement. Because my output wasn’t correct, I’m unable to draw a conclusion based on these observed time statistics. In terms of my min heap that I implemented, both the insertion and deletion operations have a time complexity of O(logn)O(logn). The percolate up and down functions have a complexity of O(n)O(n). The Huffman algorithm has a time complexity of O(nlogk) and space complexity of O(k), where k represents the nodes.