**Huffman Encoding Tree Analysis**

**Brandon L Morrow**

**Project 3 Analysis**

Huffman trees can be a very useful tool when working to compress files or encode messages. This program demonstrates the power of Huffman trees, and of implementing multiple data structures. This also shows the power of sorting and how it can greatly affect the outcome when decoding or encoding. By simply changing the sorting algorithm of the priority queue you can change the conversion by creating an entirely different Huffman tree. The same can be said of frequency tables, and this program truly demonstrates the ability to manipulate the output by manipulating and fine-tuning the data structures that create it.

This program reads the frequency table and places the characters and their frequencies into Huffman nodes as it reads the file. It then builds the binary tree utilizing our custom priority queue which implements an insertion sort to ensure the characters are in frequency order with lowest being the priority. The tree is then displayed using a preorder traversal. Once the tree is created the program reads the tree to display a key showing which clear text characters have what encoded value. After displaying the key it then decodes the encoded file and displays the clear text form. Finally, it encodes a cleartext file and displays the results. I checked that both methods are working properly by making sure that the encode and decode methods can read each other’s outputs and produce the same results.

I was able to build the binary tree using nodes relatively quickly, but had difficulty with the priority queue class due to the character precedence changing how the tree was building. I quickly realized that how you break ties can become a huge issue and could result in an entirely different Huffman tree should you even slightly modify how the Huffman/binary tree nodes are analyzed. It took a lot of time to fine tune the queue to ensure that the priority is correct. There was more trial and error involved than I would have liked because I tried to build a savant level priority queue instead of simply using an array and insertion sort like I ended up doing. I ended up doing what is normally best and going back to create a simpler version of the priority queue I had built.

Error handling is also more difficult when utilizing many data structures so I had to be sure to test one method at a time. I noticed that the tie breakers in the sort matters to ensure that the queue stays in priority order or else it builds an incorrect tree making the output unreadable.

Even a slight change to the sorting method or the buildTree method would change the output. This made debugging extremely difficult at times and forced me to check each individual method. My debugger also had a bug during the process which actually started changing lines of code when I was cycling through line breaks. This caused a nightmare until I figured out what was happening. The Huffman encoding trees are easy to break and require solid debugging since the sort in your priority queue class must work to be able to read the encoded message.

In order to confirm my findings for big-O, I built a test into the program for the time it takes for execution. I started the timer at the beginning of the program and stopped it: after the tree was built, after the preorder traversal was completed, after decoding the encoded file, and after encoding the clear text file to measure the speed of each method. This confirmed my finding that the complexity of the program is O(n) which can be clearly seen at higher lines of input. This was accomplished by utilizing an insertion sort for the priority queue (O(n)), a binary tree search for the decode algorithm in the decode() method(O(n)), and a linear search for the encode algorithm in the encodeString() method (O(n)). We can clearly see that these return a result of O(n) complexity in the graph below.

preT

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Runtime O(N) Analysis** | | | | | | | |
| **Lines Decoded** | **Lines Encoded** | **Decode**  **Time (ms)** | **Encode**  **Time (ms)** | **Total Time(ms)** | **Decode Ratio** | **Encode**  **Ratio** | **Total Ratio** |
| **100,000** | **100,000** | **533** | **1,349** | **1,882** | **0.00533** | **0.0135** | **0.0188** |
| **200,000** | **200,000** | **954** | **2,516** | **3,470** | **0.00477** | **0.0126** | **0.0174** |
| **300,000** | **300,000** | **1,454** | **3,515** | **4,969** | **0.00485** | **0.0117** | **0.0166** |
| **400,000** | **400,000** | **1,826** | **4,642** | **6,468** | **0.00457** | **0.0162** | **0.0162** |
| **500,000** | **500,000** | **2,197** | **5,685** | **7,882** | **0.00440** | **0.0114** | **0.0158** |

What I Learned

I have a much better understanding of how Huffman coding works, and also have much more practice putting together applications that utilize multiple data structures. This lab required me to build a priority queue as well as a binary tree data structure. I did a great job of taking the time to write out on paper what I needed to program to complete the project. On previous labs I was way too brute force and was not stopping to think out what I was programming until after an excessive amount of trial and error.

I learned that using custom data structures can be better than using built in functions because it can be easier to manipulate the data structure to fit your specific problem. Built in libraries are great, but I tested my code with a built-in priority queue and it did not perform the same as the priority queue that I built. This will be extremely helpful in the future since I now have a great understanding of how to implement two data structures together.

What I Might do differently Next Time

I need to do a better job of taking a practical approach during the developmental portion. To quote Greek mythology I was trying to create a program that was a work of art and flew too close to the sun. I built a great priority queue class using a singly linked list that had head and tail pointers and could be used as a generic data structure. The only problem was that I could not get the sort to work like it should have been due to it being confusing where all the pointers were pointing. After days of trying to get the sort to work correctly I realized I should have taken a much simpler approach and built the priority queue with an array-based implementation. It only took me 30 minutes to do this approach and I was mad at myself for wasting days of my life for something I could have done in 30 minutes had I not shot too high trying to make an amazing priority queue class. All I needed was functional and should have identified that I did not have the time I needed to build what I was building within a week or two span.

I would start with the data structures and then work my way to the final product. I would start programming bottom up instead of top down like I did. I became frustrated with the priority queue so I ended up then switching over to building the tree class using the built in priority queue method. This worked decently well, but that class did not work like my priority queue so it caused me extra work to then redesign certain aspects in the main to match my priority queue. It also became hard to test my priority queue since my main was clogged up by that point reading the file. This was not a massive issue since I had everything compartmentalized, but I think that I should have finished the priority queue before moving on.

Justification for Design Decisions

There are four classes in this program and they are Main, HPriorityQueue, HuffNode and HuffTree. This program is designed to be compartmentalized for easy adjustment to any required elements.

This program is designed so that the data is read into the file and stored in the binary tree nodes (HuffNodes). These nodes are loaded into a priority queue(HPriorityQueue). The lowest frequency items are then taken from the queue and assigned as the left and right nodes of a new parent node. This parent node is then passed to the queue. This process continues until only the root node remains in the queue. The root node is then returned to the main as the Huffman tree.

Once the tree is built the HuffTree class displays the tree utilizing a preorder traversal to print the tree. This class also has a method that creates a Huffman key array that contains letters, and their Huffman encoding, and then displays that key (displayHuffKey() method).

The program then converts the Huffman code from a file to clear text using the decode() method. The decode method moves down the tree based on if the file has a 0 or a 1 until it hits a leaf node. Once a leaf is reached it adds the character to a string and returns the string at the end of the line.

The program then converts clear text to Huffman code using the encodeString() method. This utilized the Huffman key we built earlier and searches the key for the character. Once the search finds the character it is looking for it returns the encoded version of that character.

Issues of Efficiency

The data is processed as it is read from the file so the access time of the filereader is O(1) and the different methods are O(n) complexity. This program utilizes an insertion sort, binary tree search, and linear search to create the tree, read the tree, and read the key. All three of these methods are O(n) complexity and the runtime directly scales with the size of the file.

Generally, O(n) appears to be the best-case scenario for this program when processing the prefix data from a file. The runtime scales proportionally to the amount of data in the file and would only go to a worst cast O(n^2) for a small time should the input frequency table be read from highest frequency to lowest frequency. Even in this case the input frequency table will be constant 26 characters (the total number in the alphabet) so this methods runtime does not scale proportionally with the data like the encode and decode methods.