Johnathan Tripp
COSC 310
Prof. Kennedy
12/11/16

Huffman Encoding Tree Project – Post-Mortem

To implement the Huffman encoding tree, I first began with implementing the various Tree ADTs given by the textbook in order to finally implement a functional Binary Tree. I found this process extremely tedious, as in order to implement the binary tree according to the text, there are several layers of interfaces and abstract classes that must also be implemented. For the purposes of this project, I made minor modifications to the binary tree and node classes I implemented in order to suit the needs of the Huffman encoding project. For example, in addition to the element stored at the node, in this case a Character, I also added a field for storing that character's frequency which could be accessed by the various methods used to generate the tree.

Once I completed implementation of the binary tree I turned my attention to the implementation of the Huffman encoding algorithm itself. Ultimately I did not find this too difficult to do, though I realized late into working on this project that I had been approaching the problem in a manner perhaps more difficult than it needed to be. The Driver class for the project begins by reading in an input text file which is then read to store the characters used throughout the text. Characters are stored without duplication in an ArrayList. After this step, the file is processed again, this time counting the occurrences of each character in the file. The results of this count are placed in a LinkedList, which stores the characters and their corresponding frequencies as Binary Tree nodes. I used a simple loop to count the occurrences, though in

retrospect using something such as a regular expression may have provided a cleaner and faster solution.

Once frequencies have been counted and the Huffman tree nodes are created, the list that stores the nodes is sorted. This simplifies the process of accessing characters with the lowest frequencies as those are in the front of the list. The least occurring two nodes are removed and joined to form a new tree with the combined frequencies at its root. This tree is re-inserted into the list. Because I used a LinkedList, to avoid extra implementation of data structures, I simply called the *Collections.sort* method on the list to sort it after each insertion. While my solution does work and performs its functions quickly, I realized during implementation that I had forgotten details about implementation from the lecture slides that mentioned using a minheap priority queue to store the trees. Due to my less than stellar understanding of heaps as they apply to priority queues, I opted not to pursue that option, especially as I had already implemented much of my current solution.

After generating the Huffman tree, the rest of the project became easy. I implemented a simple recursive method detailed in the lecture slides to generate the encoded values for the characters, and printed all the values out, as well as the encoded text from the input file. This was by far the easiest part of the project to implement. Once I became familiar with the concepts behind Huffman encoding trees, the implementation of this project became much easier. However, I faced a lot of difficulty in implementing the data structures detailed by the book for the Binary Tree, due to the sheer number of different classes and interfaces that have to be made and kept track of. I am also sure that I made this project more difficult than necessary due to the way I implemented it in terms of generating and storing the tree, but while working on this I felt as though I just hit a wall and couldn’t see alternative ways to accomplish the goal for this

project. I am, however, satisfied with the results of this project as it does run very quickly and, in my testing, without errors.