Lab 3

Jordan Carson
Section EN.605.202
Summer 2022
August 9th, 2022

Summary

Notes on the approach and implementation are discuss below. Below are my top-level observations.

1. Using a frequency table from the text will provide a more efficient encoding on the text
2. The method for ties is arbitrary if the actual frequency from the encoded text matches the expected frequency
3. All characters must be encrypted for space savings

Approach and Notes

I tried to first create the Huffman Tree by hand. This is not easy or straightforward! The examples in the lectures and in the lab, handout did not deal with the possibility that merging two letters may not be the lowest; this could cause an infinite loop if using weights and not properly handling collisions. Thus, after figuring out how to handle these collisions via trial and error I began coding my solution.

Implementation

Created several helper functions for reading the input variables – found in helpers.py. Next, created two classes to assist with building the trees. First, a HuffmanNode holding the character, frequency and left and right pointers. Second, the HuffmanTree. This tree can cover some or all of the 26 letters in the English Dictionary. There are a few constructors and methods to assist with creating the tree, printing, and traversal.

1. Take a key (char) and a frequency to create a single node
2. Same as 1, but allows us to identify a tie-breaker method. This could change the results based on which method is selected.
3. Takes the Trees and creates the parent but placing the sub-trees and nodes in the correct order to derive the new parent's frequency and key.

Encoding and Decoding

I created a memorized table of codes for each letter, instead of needing to traverse the tree each time. This may save time, but with really large inputs we may not be able to use this implementation.

For decoding, its very easy to traverse the tree. Given that the codes are not all the same length, it may be less efficient to try and use the memorized table to decode the messages.

Overall, I decided to ignore punctuation, whitespace and lowercased all characters (uppercase works too). Newlines and carriage returns need to be investigated separately.

Results

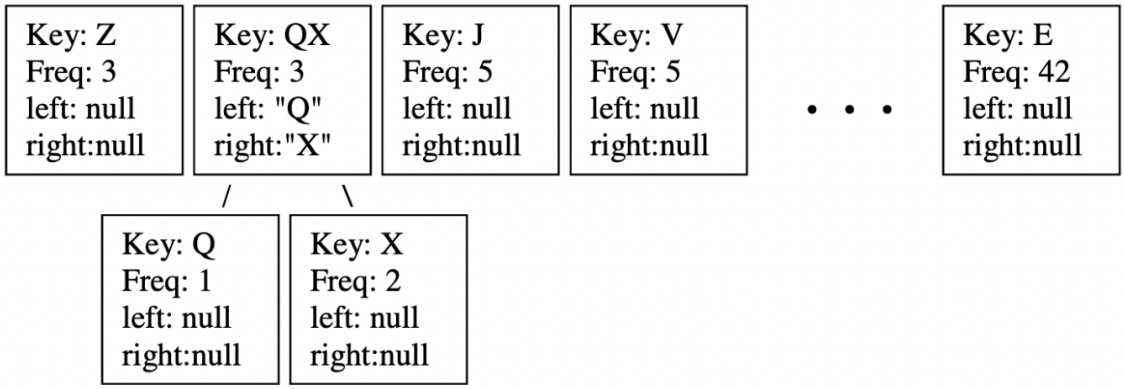
Below are the results of printing the code dictionary after loading the given input text into PriorityQueue and HuffmanTree as the initial step before merge.

{'A': '19', 'B': '16', 'C': '17', 'D': '11', 'E': '42', 'F': '12', 'G': '14', 'H': '17', 'I': '16', 'J': '5', 'K': '10', 'L': '20', 'M': '19', 'N': '24', 'O': '18', 'P': '13', 'Q': '1', 'R': '25', 'S': '35', 'T': '25', 'U': '15', 'V': '5', 'W': '21', 'X': '2', 'Y': '8', 'Z': '3'}

[<HuffmanNode: char_list='Q', freq=1, left=None, right=None>, <HuffmanNode: char_list='K', freq=10, left=None, right=None>, <HuffmanNode: char_list='D', freq=11, left=None, right=None>, <HuffmanNode: char_list='F', freq=12, left=None, right=None>, <HuffmanNode: char_list='P', freq=13, left=None, right=None>, <HuffmanNode: char_list='G', freq=14, left=None, right=None>, <HuffmanNode: char_list='U', freq=15, left=None, right=None>, <HuffmanNode: char_list='I', freq=16, left=None, right=None>, <HuffmanNode: char_list='B', freq=16, left=None, right=None>, <HuffmanNode: char_list='H', freq=17, left=None, right=None>, <HuffmanNode: char_list='C', freq=17, left=None, right=None>, <HuffmanNode: char_list='O', freq=18, left=None, right=None>, <HuffmanNode: char_list='M', freq=19, left=None, right=None>, <HuffmanNode: char_list='A', freq=19, left=None, right=None>, <HuffmanNode: char_list='X', freq=2, left=None, right=None>, <HuffmanNode: char_list='L', freq=20, left=None, right=None>, <HuffmanNode: char_list='W', freq=21, left=None, right=None>, <HuffmanNode: char_list='N', freq=24, left=None, right=None>, <HuffmanNode: char_list='T', freq=25, left=None, right=None>, <HuffmanNode: char_list='R', freq=25, left=None, right=None>, <HuffmanNode: char_list='Z', freq=3, left=None, right=None>, <HuffmanNode: char_list='S', freq=35, left=None, right=None>, <HuffmanNode: char_list='E', freq=42, left=None, right=None>, <HuffmanNode: char_list='V', freq=5, left=None, right=None>, <HuffmanNode: char_list='J', freq=5, left=None, right=None>, <HuffmanNode: char_list='Y', freq=8, left=None, right=None>]

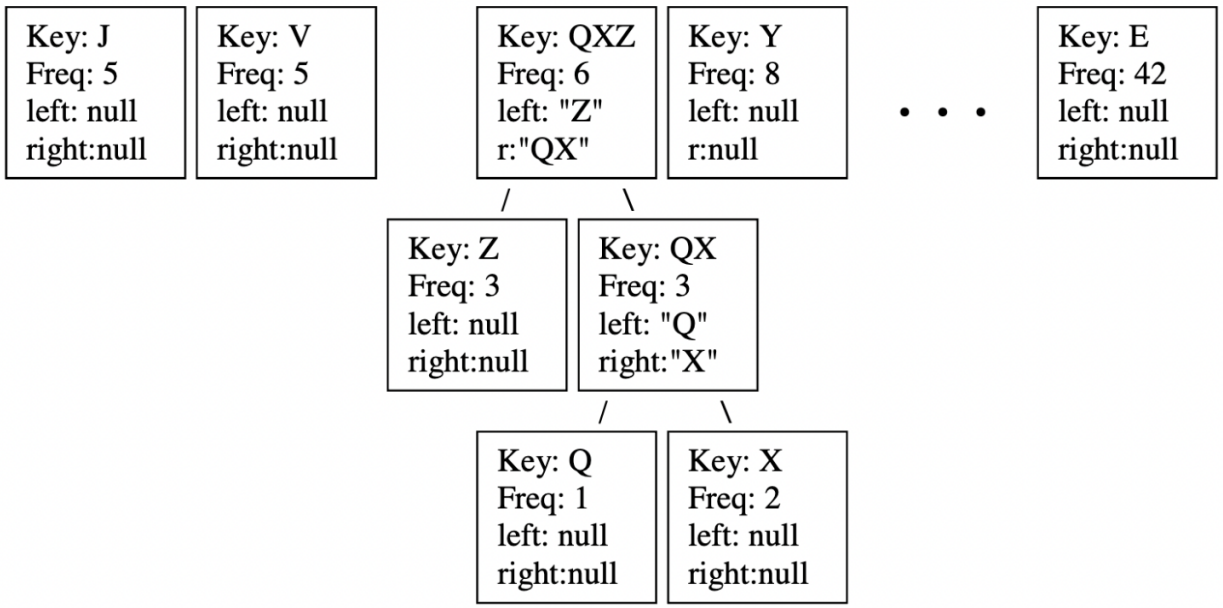
Post Merge Results.

After first merge.



The idea is that when nodes merge, they retain the pointers to its children, so that after 25 or so merges the remaining nodes will be a proper Huffman Tree.

After second merge.



As we can see Key QXZ has a frequency of 6, 1 + 2 + 3, right points to QX while left points to Z.

Next Steps

Create a python library via Poetry or another tool (wheel) to package the code. This is a great way to compress data as the space does reduce in size while retaining the same information after encoding.

Tie Breaker Analysis

Conventional encoding such as gzip uses Huffman Encoding trees under the hood! So of course, the results achieved useful data compression.

- For breaking ties, I used a default and mirror approach.
- The largest bit count for the least frequent letters is 8. Why? Well ANSI uses 8 bits per character, so the encoding must be as good as ANSI! When the encoding text has the same frequency as the table, the number of bits ranges around 4.5. As mentioned below.

Different Tie-Breaker Methods

key	default	mirror	freq	default length	mirror length
A	11111	0000	19	5	4
B	11000	11001	16	5	5
C	11010	11011	17	5	5
D	01100	01100	11	5	5
E	010	010	42	3	3
F	01101	01101	12	5	5
G	10101	10110	14	5	5
H	11011	11010	17	5	5
I	11001	11000	16	5	5
J	001010	000101	5	6	6
K	00100	00011	10	5	5
L	0001	0010	20	4	4
M	0000	11111	19	4	5
N	0111	0111	24	4	4
O	11110	11110	18	5	5
P	10100	10100	13	5	5
Q	10110010	10101000	1	8	8
R	1000	1001	25	4	4
S	1110	1110	35	4	4
T	1001	1000	25	4	4
U	10111	10111	15	5	5
V	001011	000100	5	6	6
W	0011	0011	21	4	4
X	10110011	10101001	2	8	8
Y	101101	101011	8	6	6
Z	1011000	1010101	3	7	7
AVERAGE				**4.467312349**	**4.467312349**