Lab 03
Adam Rich
Section EN.605.202.87
Spring 2018
April 24, 2018

Summary

Notes on the approach and implementation and results are discussed below. I will summarize here some key observations:

1. Using a frequency table from a text will provide more efficient encoding on that text
2. The method for handling ties doesn't really matter if the actual frequency matches the expected frequency of the text being encoded
3. Any frequency table that is built on real English text gives space savings on encryption

Approach and Notes

Before coding, I attempted to create the Huffman Tree for the given frequency table and "tie breaking scheme" by hand. Creating such a table by hand is not easy! Additionally, all of the examples in the lectures and in the lab handout did not deal with the possibility that after merging two letters that the newly merged node would not be one of the two lowest still. These hurdles were easily overcome using trial and error and the example provided in the assignment sheet.

Once I was able to do most of the tree by hand I began writing the code.

Implementation

I created two classes to assist with building the trees. The obvious one is HuffmanTree. These trees can be complete, i.e. covering all 26 letters, incomplete, or just single nodes. There are ~~two~~ three constructors in this class.

1. takes just a key and a frequency and creates a single node
2. same as the first, but also allows the user to specify a "tie-breaking" method other than default
3. takes two HuffmanTrees and creates the parent, putting the sub-trees in the correct order and deriving the new parent's frequency and key

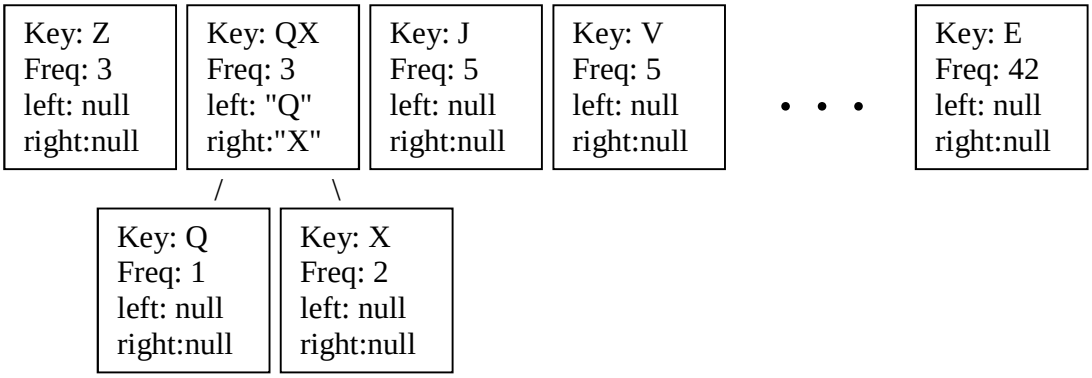
The second class I created was a helper class for handling the building of the trees from the 26 individual starting nodes. I called this class the HuffmanLeafPile, since each single letter is a leaf in the final tree. The method "rakeUp" creates the tree by doing successive "merges". It is probably best to illustrate how this works graphically.

Initially the HuffmanLeafPile is an array of 26 single-node HuffmanTrees:

Key: Q Freq: 1 left: null right:null	Key: X Freq: 2 left: null right:null	Key: Z Freq: 3 left: null right:null	Key: J Freq: 5 left: null right:null	Key: V Freq: 5 left: null right:null	• • •	Key: E Freq: 42 left: null right:null

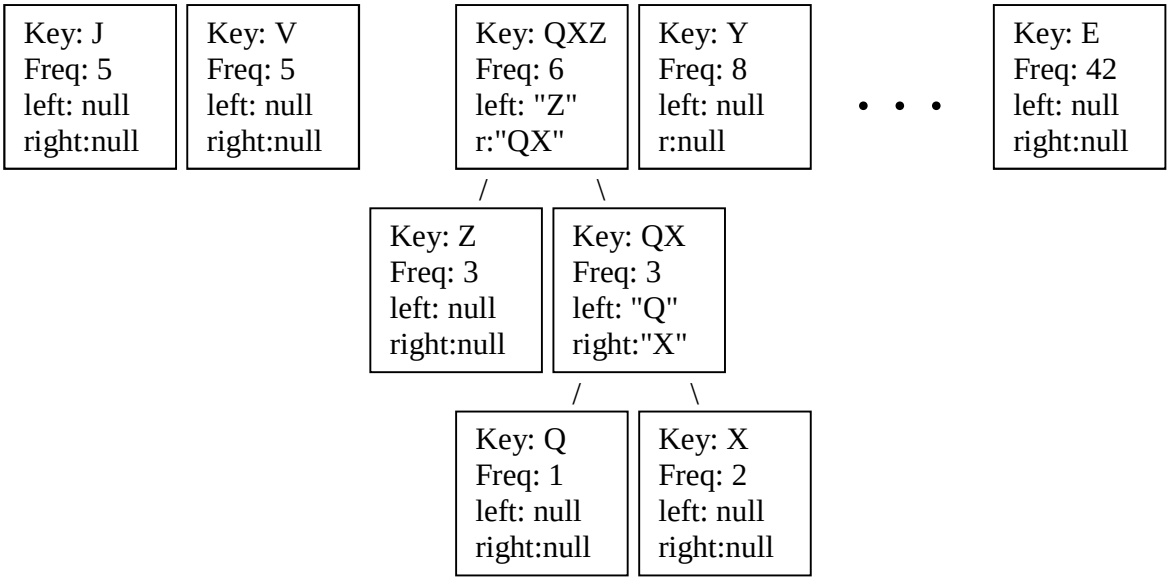
In my implementation these were actually sorted in descending order, but for illustration (since all the activity happens at the lower end!) I show them sorted this way. After creating the array of nodes and sorting, the last two nodes are removed, merged and the new parent is added back to the array which is then sorted again.

After first merge.

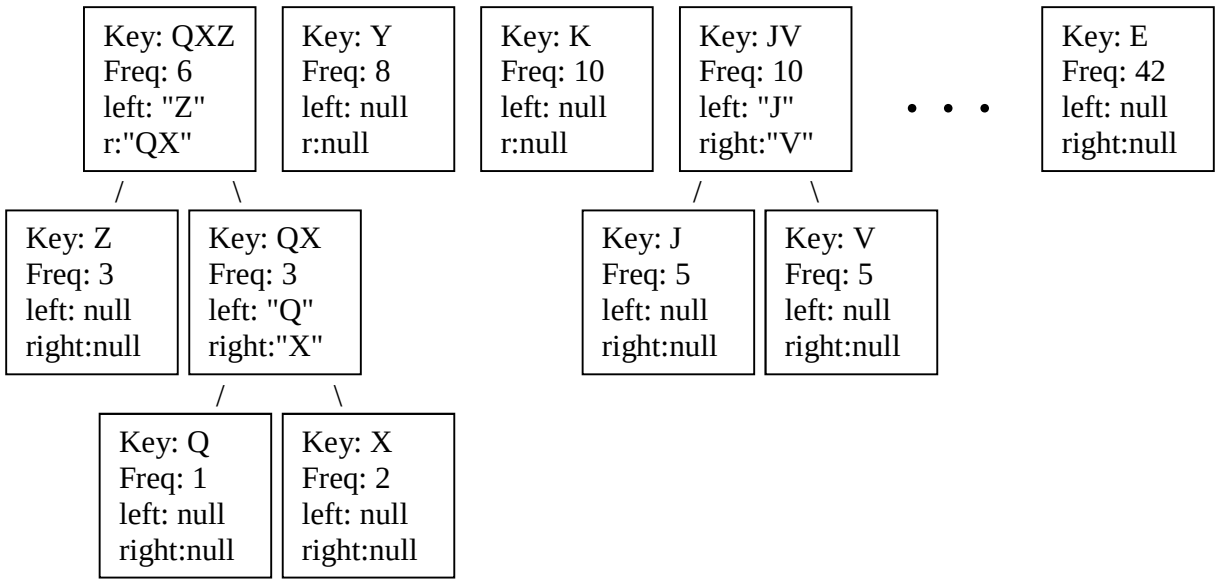


The new key QX maintains pointers to its children, so that after 25 merges the remaining node will be a proper HuffmanTree.

After second merge.



And for a final example, because it does not use the QXZ node, which is an example not covered in class:



The final tree, printed using pre-order traversal follows. Along with the an array of the final encodings one would achieve for each letter (0 goes left, 1 goes right):

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ:413
  DEFJKLMNVW:169
    JKLMVW:80
      LM:39
        M:19
        L:20
      JKVW:41
        JKV:20
          K:10
          JV:10
            J:5
            V:5
        W:21
    DEFN:89
      E:42
      DFN:47
        DF:23
          D:11
          F:12
        N:24
  ABCGHIOPQRSTUXYZ:244
    GPQRTUXYZ:106
      RT:50
        R:25
        T:25
      GPQUXYZ:56
        GP:27
          P:13
          G:14
        QUXYZ:29
          QXYZ:14
            QXZ:6
              Z:3
              QX:3
                Q:1
                X:2
            Y:8
          U:15
    ABCHIOS:138
      BCHI:66
        BI:32
          B:16
          I:16
        CH:34
          C:17
          H:17
      AOS:72
        S:35
        AO:37
          O:18
          A:19
```

```
Table of codes:

A:11111
B:11000
C:11010
D:01100
E:010
F:01101
G:10101
H:11011
I:11001
J:001010
K:00100
L:0001
M:0000
N:0111
O:11110
P:10100
Q:10110010
R:1000
S:1110
T:1001
U:10111
V:001011
W:0011
X:10110011
Y:101101
Z:1011000
```

One interesting observation with this "default" tree is that the biggest bit count, for the least frequent letters, is 8. ANSI encoding uses 8 bits per character, so the encoding will automatically be as good as ANSI. When encoding text that has roughly the same frequency as the table, the number of bits per letter will be about 4.5.

Encoding and Decoding
To encode strings I created a memoized table of codes for each letter, like in the table above, instead of traversing the tree each time.

To decode, I found it easiest to traverse the tree. Because the codes are not all the same length it would be less efficient to try and use the code table to decode messages.

In both decoding and encoding I ignored case, punctuation and whitespace. Except, I did retain newline and carriage return characters.

Processing Strings
I did not use many String functions. The ones I did use are

- toUpperCase – for convenience in converting lower case characters
- charAt – to be able to treat strings as index-able character arrays
- concatenation = "+"
- equals – since Java, for some reason, doesn't support "==" for String comparison

Reading and Writing Files
To speed up the reading and writing of larger files, I used classes from the ***java.nio*** namespace.

Sorts Used
Sorts were used in two places. In the HuffmanLeafPile after each of the 25 merges the new parent node needs to be inserted in the correct place in the array. Because the array is relatively small, I just put the parent node on the end, decremented an integer signifying the true end of the data, and then re-sorted. The sort used was a quadratic sort.

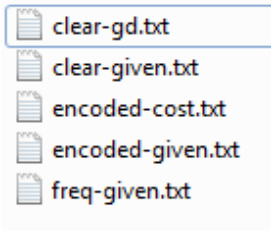
The other area where a sort is needed, again a quadratic sort, is when a composite key is created for a parent node. I sorted the character array so that the letters all appear in alphabetical order. This has these advantages:

- makes everything easier to read
- the top node of a complete tree will always be "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
- when ordering nodes with ties, it is easy to alphabetize based on the first letter in the key

Frequency Table Reports
When the app is run, whether for decoding or encoding, a "frequency analysis report" is created. This shows the original frequency tree, the pre-order printed representation of the tree, and the resulting code table. Examples can be seen in the "output" folder.

Test Cases

given



The lab gives a frequency table, sample encoded message, and sample clear message. These all include the stem "given". The file "encoded-cost.txt" was the encoded message Dr. Cost sent in his reminder email that the lab is due April 24. The file "clear-gd.txt" is a copy of the decoding of "encoded-given.txt". Encoding that file gives the same bit-message as the original.

montana

A few years ago, my then 12-year old daughter and her best friend were wondering about frequency of different letters in the English language. To help them answer the question, and to give them some exposure to coding, we wrote some R code to analyze text from books in the Gutenberg Project. We decided we should check if frequency was different between a contemporary novel and something older. For our contemporary novel, we chose "Call Me Montana".

http://self.gutenberg.org/eBooks/WPLBN0001235250-Call-Me-Montana-by-John-Richman.txt

The text of that book and a frequency table derived from it are also included as inputs.

dickens, toadstools, and bible

Our "older book" was "Great Expectations" by Charles Dickens. I've also found two other books to include as test cases: The King James version of the Bible and "Our Edible Toadstools and Mushrooms and How to Distinguish Them".

http://www.gutenberg.org/files/1400/1400-0.txt
http://www.gutenberg.org/files/46514/46514-0.txt
http://www.gutenberg.org/cache/epub/10/pg10.txt

Creating Frequency Tables

I used the R script in the "scripts" folder to create frequency tables from the test cases above. I also created the following fake frequency tables

- "backward" – reverses the frequencies so that the most frequent letters get the longest codes
- "skew" – built to give the tallest Huffman encoded tree possible
- "uniform" – all frequencies set to 1

Frequency tables side-by-side

key	given	uniform	toadstools	skew	montana	bible	backward
E	42	1	30,783	2^25 = 33554432	23,173	412,232	1
S	35	1	19,390	2^24 = 16777216	10,326	190,029	2
R	25	1	15,953	2^23 = 8388608	10,693	170,327	3
T	25	1	22,140	2^22 = 4194304	18,576	317,744	5
N	24	1	16,930	2^21 = 2097152	12,467	225,055	5
W	21	1	4,117	2^20 = 1048576	5,262	65,487	8
L	20	1	11,576	2^19 = 524288	7,358	129,938	10
A	19	1	19,185	2^18 = 262144	14,859	275,727	11
M	19	1	7,302	2^17 = 131072	5,025	79,940	12
O	18	1	20,387	2^16 = 65536	15,130	243,185	13
C	17	1	8,837	2^15 = 32768	3,780	55,067	14
H	17	1	11,898	2^14 = 16384	11,288	282,678	15
B	16	1	4,202	2^12 = 4096	3,109	48,875	16
I	16	1	20,295	2^13 = 8192	12,548	193,959	16
U	15	1	9,110	2^11 = 2048	5,562	83,473	17
G	14	1	5,137	2^10 = 1024	4,508	55,301	17
P	13	1	6,666	2^9 = 512	2,919	43,254	18
F	12	1	6,625	2^8 = 256	3,467	83,543	19
D	11	1	9,087	2^7 = 128	9,021	158,094	19
K	10	1	1,230	2^6 = 64	2,277	22,292	20
Y	8	1	4,462	2^5 = 32	4,131	58,576	21
J	5	1	257	2^4 = 16	429	8,889	24
V	5	1	2,335	2^2 = 4	1,424	30,365	25
Z	3	1	185	2^3 = 8	49	2,972	25
X	2	1	573	2^1 = 2	190	1,478	35
Q	1	1	362	2^0 = 1	124	964	42

Frequency tables showing relative ranks of keys

key	given	toadstools	skew	montana	bible	backward
E	1	1	1	1	1	26
S	2	5	2	9	8	25
R	3	8	3	8	9	24
T	4	2	4	2	2	23
N	5	7	5	6	6	22
W	6	20	6	13	15	21
L	7	10	7	11	11	20
A	8	6	8	4	4	19
M	9	14	9	14	14	18
O	10	3	10	3	5	17
C	11	13	11	17	18	16
H	12	9	12	7	3	15
B	13	19	13	19	19	13
etc.						

Bad Messages
An encoded message is "bad" when there are trailing bits that do not resolve to a leaf in the Huffman tree. I've written my code purposefully to ignore these. If this is to be used to encode secret messages, trailing nonsense bits might help to throw wannabe saboteurs.

Tie-Breaking
The code currently supports two methods of tie-breaking: the default, as given in the assignment, and one called "mirror". "Mirror" is the exact opposite of the default. It doesn't appear to alter the expected average bits-per-character of the resulting encoding. For example, here is the "default" versus "mirror" methods compared for the "given" frequency table.

Given Tree with Different Tie-Breaker Methods

default	mirror
ABCDEFGHIJKLMNOPQRSTUVWXYZ:413	ABCDEFGHIJKLMNOPQRSTUVWXYZ:413
DEFJKLMNVW:169	ADEFJKLNVW:169
JKLMVW:80	AJKLVW:80
LM:39	AJKV:39
M:19	A:19
L:20	JKV:20
JKVW:41	JV:10
JKV:20	V:5
K:10	J:5
JV:10	K:10
J:5	LW:41
V:5	L:20
W:21	W:21
DEFN:89	DEFN:89
E:42	E:42
DFN:47	DFN:47
DF:23	DF:23
D:11	D:11
F:12	F:12
N:24	N:24
etc.	

This is just left sub-tree, but it illustrates how the tree shape is different because of the ties being handled differently. The child nodes at each level have a one-to-one correspondence, when matched on frequency. But, the keys can be different. For example, the "default" method has JKLMCW:80 where the "mirror" has AJKLVW:80. This ultimately comes down to A and M having the same frequency.

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**

Of course, if the text being encoded has more A's than the frequency table would imply then "mirror" would be a better option. But, if the actual frequency mirrors the expected then they will be equally efficient.

The bigger the corpus used to create the frequency table, the less likely it is that the tie-breaking method will even have an effect on the final shape of the tree.

Results

Running all input files against all available frequency tables gives this table of "bits per character".

Freq	Input bible	given	montana	toadstools
backward	5.78	5.93	5.79	5.80
bible	4.12	4.40	4.22	4.24
given	4.41	4.36	4.42	4.43
montana	4.13	4.40	4.21	4.25
skew	8.58	8.87	8.61	8.61
toadstools	4.16	4.31	4.23	4.22
uniform	4.93	4.96	4.91	4.92

The files "bible", "montana", and "toadstools" have their best compression rate when using the frequency table derived from that file. The uniform frequency table is pretty consistent across all of the inputs. The skew table provides for encryption that is worse off than ANSI encoding. "Skew" was purposefully derived to be really bad, but it still preserves the relative order of keys given in the "given" frequency table. It could be worse by doing a skewbackward.

key	skewbackward
E	1
S	2
R	4
T	8
N	16
W	32
L	64
A	128
M	256
O	512
C	1,024
H	2,048
B	4,096
I	8,192
U	16,384
G	32,768
P	65,536
F	131,072
D	262,144
K	524,288
Y	1,048,576
J	2,097,152
V	4,194,304
Z	8,388,608
X	16,777,216
Q	33,554,432

Freq	Input bible	given	montana	toadstools
backward	5.78	5.93	5.79	5.80
bible	4.12	4.40	4.22	4.24
given	4.41	4.36	4.42	4.43
montana	4.13	4.40	4.21	4.25
skew	8.58	8.87	8.61	8.61
toadstools	4.16	4.31	4.23	4.22
uniform	4.93	4.96	4.91	4.92
skewbackward	18.28	17.95	18.26	18.27

"Skewbackward" gives results that are really horrible. But, this is a completely contrived example. It appears that any frequency table that is built on real English text would be OK. Even uniform provides for a space savings when encrypted.

Wrap Up

Things Learned
I continue to learn how much I like to code in Java. Learned some new methods of reading and writing data. Of course, I learned a lot about Huffman Encoding!

What would I do differently?
I am very happy with this lab and would probably not do anything differently.

IDE

My IDE was notepad++ and the command line. Instructions on how to build and run the code, from the command line at the project base folder, are in the README file.

github

I used github, a private repo, to backup and track important chunks of development.