1a.

Index	0	1	2	3	4	5	6	7	8	9
Value	1400, 8980	3341, 4321	7652	1093	5674					

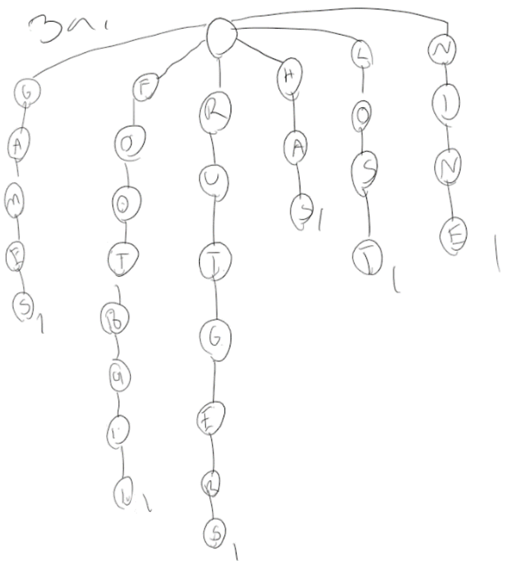
1b.

Index	0	1	2	3	4	5	6	7	8	9
Value	1400	3341	7652	1093	4321	5674	8980			

2.

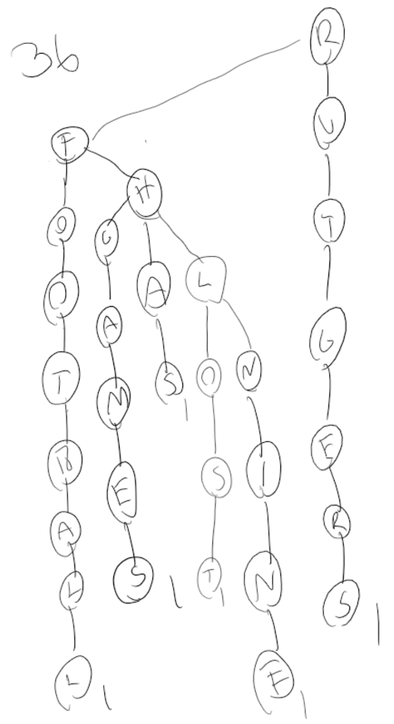
The ideal way to solve this problem is to use a Bloom field. A bloom field is a Boolean vector of size n where each index can be true or false (or 0/1). This allows you to only store the state true or false instead of the entire object to determine if an object is in the set. This drastically reduces the space requirements when the objects are long strings, which requires several bytes to store in a traditional hash table. You use k independent hash functions where each function returns a different number between 0 and n-1 at random. Each hash function cannot return the same value for the same object. For example, if there were two hash functions and the word “app” was processed the first and the second hash functions can’t return the same value. Once you run the hash functions on the object you set the corresponding index of the vector to true. Again using “app” as an example. If the first function returned 1 and the other function returned 5, you would set index 1 and 5 of the vector to true. If another object returns some indexes that are already true, they do not have to be changed. Once the bloom field is constructed you can search for a word using the hash functions. If the indexes returned by the hash functions are all set true then the item is in the set. If any index returned is not set true the item is not in the set. False negatives can’t occur in a bloom field, but false positives can occur. The optimal number of hash functions for a low probability of false positives is equal to $\frac{n}{m} * \ln(2)$ where m is number of elements stored in the set. Assuming the optimal number of hash functions is used, and you know what you want the probability of a false positive to be, you can determine what the size of the bloom field (n) should be using the following equation: $n = -\frac{m * \ln(p)}{\ln(2)^2}$ where p is the desired probability of a false positive. For example, if the client wanted the probability of a false positive to be 0.001%, the values used in the above formula would be 0.00001 for p, and one million for m.

3a.



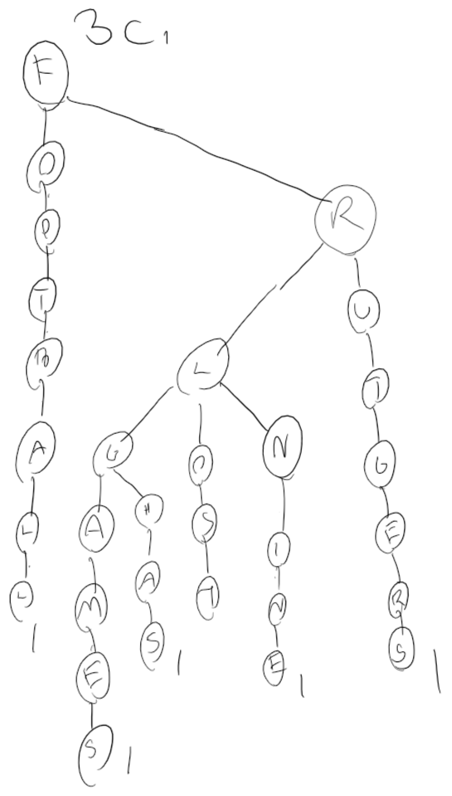
The height of trie is equal to 8.

3b.



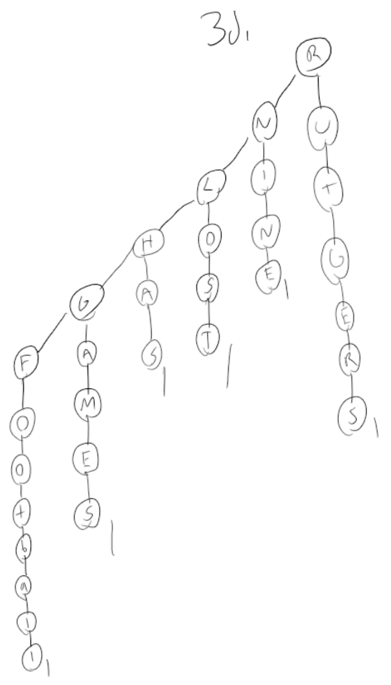
If root is still has a depth of 0 the height is equal to 8

3c.



The insertion order that minimizes the height is FOOTBALL, RUTGERS, LOSE, NINE, GAMES, HAS. The resulting height under the assuming the depth of the root is 0 is equal to 7

3d.



The insertion order that maximizes the height is RUTGERS, NINE, LOST, HAS, GAMES, FOOTBALL. The resulting height under the assuming the depth of the root is 0 is equal to 12

4.

Since Huffman uses prefix codes it allows you to read the encoded bit stream from left to right. This is because as soon as you hit a codeword you know nothing else starts with the same code. For example, if the code for e was 00 and the code word for d is 01 and you are decodeing the bit stream 0001 you know that after reading 00 it has to correspond to e since nothing else contains 00 and after reading 01 it has to correspond to e. If the Huffman codes weren't prefix codes, you wouldn't be able to decode the bitstream by reading from left to right. For example, if the code for e was now 0 and the code for d was still 00 and the encoding bit stream was 0000 and you read the bitstream from left to right you wouldn't know if after reading 0 you found e or if that is still part of d. This means that the decoded output of this stream could be "eeee", "ede", "dd", etc.

5a.

The best case occurs when all N bits are 1s. For example, the bitstream 111111 would produce the best possible compression ratio for 6 bits. The compression ratio in the best case is $B = \frac{8*\left(int\left(\frac{N}{256}\right)+1\right)}{N}$ where B is the number of bits after the compression and int rounds to the lowest number. The worst case would occur if the order of 0s and 1s consistently alternated. For example, the bitstream 01010101 would result in the worst compression ratio for 8 bits. The compression ratio in the worst case is $8 * N$.

5b.

The base case occurs when there are only two different characters in the string since the prefix code will only be 1 bit long. For example, the string “aababa” would result in the best compression ratio for six strings. The compression ratio in the best case is $B = N$, where B is the number of bits after the compression and N is the number of characters. The worst case would occur when the frequency of the characters follows the Fibonacci sequence because it causes the resulting Huffman tree to be the most unbalanced. For example, the string “abccddd” would result in the worst possible compression ratio for 7 strings. It is not possible to compute the compression ratio in the worst case since it is unbounded. This means the ratio will continue to get worse as N increases.

5c.

The best compression ratio occurs when all the characters are the same (case sensitive) this ensures that the user expanded dictionary is used as much as possible. For example, the string “aaaa” would result in the best possible compression ratio for four characters. The compression ratio in the best case is $R = round(\sqrt{N})$, where R is the number of hexadecimal pairs after the compression and N is the number of characters. The worst-case compression ratio occurs when every character is different since nothing from the user expanded dictionary will be used. For example, the string “abcd” would result in the worst possible compression ratio for 4 characters. The worst-case compression ratio is equal to 1 since nothing gets compressed in the worst situation.

6.

Expanded Dictionary

Key	Value
AB	81
BA	82
AA	83
AC	84
CD	85
DF	86
FA	87
ABA	88
ACD	89

LZW compression = 41 42 41 41 43 44 46 81 84 44 80