1. a. Separate Chaining (storing multiple integers in a hash slot), N=10

1093 % 10 = 3
1400 % 10 = 0
3341 % 10 = 1
7652 % 10 = 2
4321 % 10 = 1
5674 % 10 = 4
8980 % 10 = 0

Chain the numbers with the same remainder…

Index	0	1	2	3	4
Hashed	8980, **1400**	4321, **3341**	7652	1093	5674

b. Linear Probing (resolving collisions in hash tables), N=10

(index range = mod -1 = 9, so index goes from 0-8)

h(x) = x% N = (a # in array) % 10

1093 % 10 = 3
1400 % 10 = 0
3341 % 10 = 1
7652 % 10 = 2
4321 % 10 = 1
5674 % 10 = 4
8980 % 10 = 0

Index	0	1	2	3	4	5	6
Hashed	1400	3341	7652	1093	4321	5674	8980

2. To solve the client problem using hashing without storing the actual values, there is a thing called "minimal perfect hashing", in which if the set of keys (k) planned to query the hash table is known in advance (in this case it is 1 million objects in the set), then one can construct a hash function which ensures $k1 \neq k2$, $h(k1) \neq h(k2)$, which makes the hash function injective (one-one and preserving distinctiveness). If this can be arranged, then there is no need to store the objects in the set and so the hash function is hence "perfect".

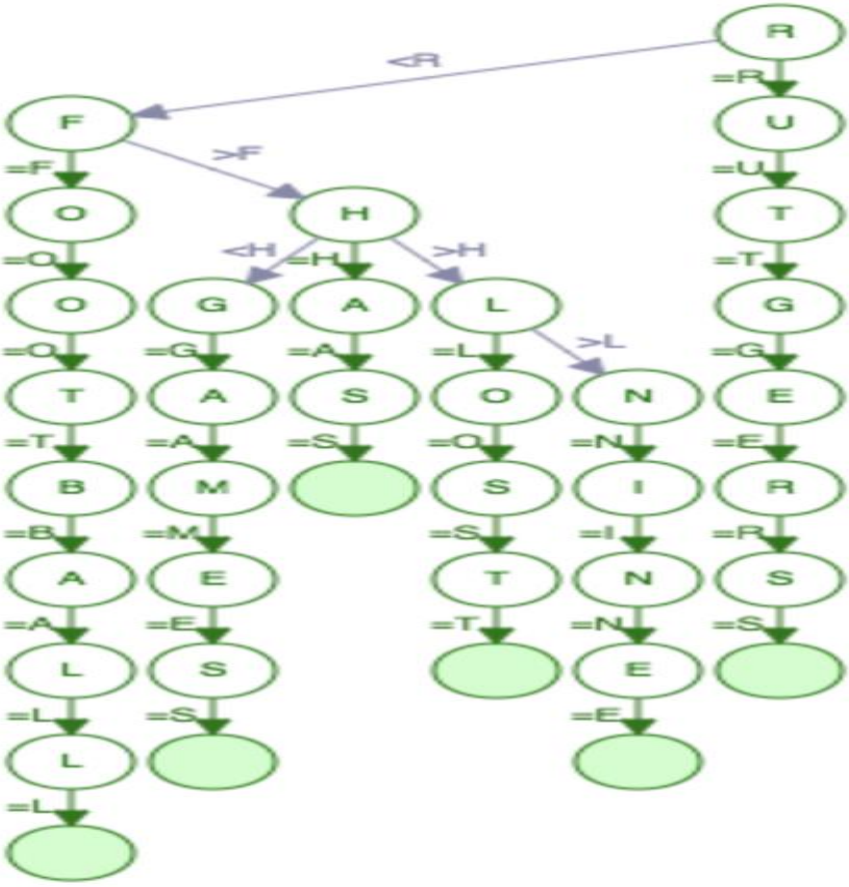
Several different hashing functions will be used and each object in the set will go through the different hashing functions. The value of the remainders for these hashing functions for each object is in the set, then all they would have to do is run that object through the same hash functions and then they could search through the hash table to see if there is any key in there that matches the remainders that the object has.

For example, if a potential number in the set (k) is 96 then that number is run through the hash function that is given. Going through the hash function will give several different remainders and that will then be checked though the hash table to see if any index has same keys. If they do match, then it is most likely in the given set.

3. Sequence of strings: {RUTGERS, FOOTBALL, HAS, LOST, NINE, GAMES } are inserted into an R-way trie, with values all equal to '1' (All values =1, Root =0)

a. using R-way trie, Height of tree = 8, the height is determine by the string that takes the most nodes to get to the key, here it is the word "FOOTBALL"

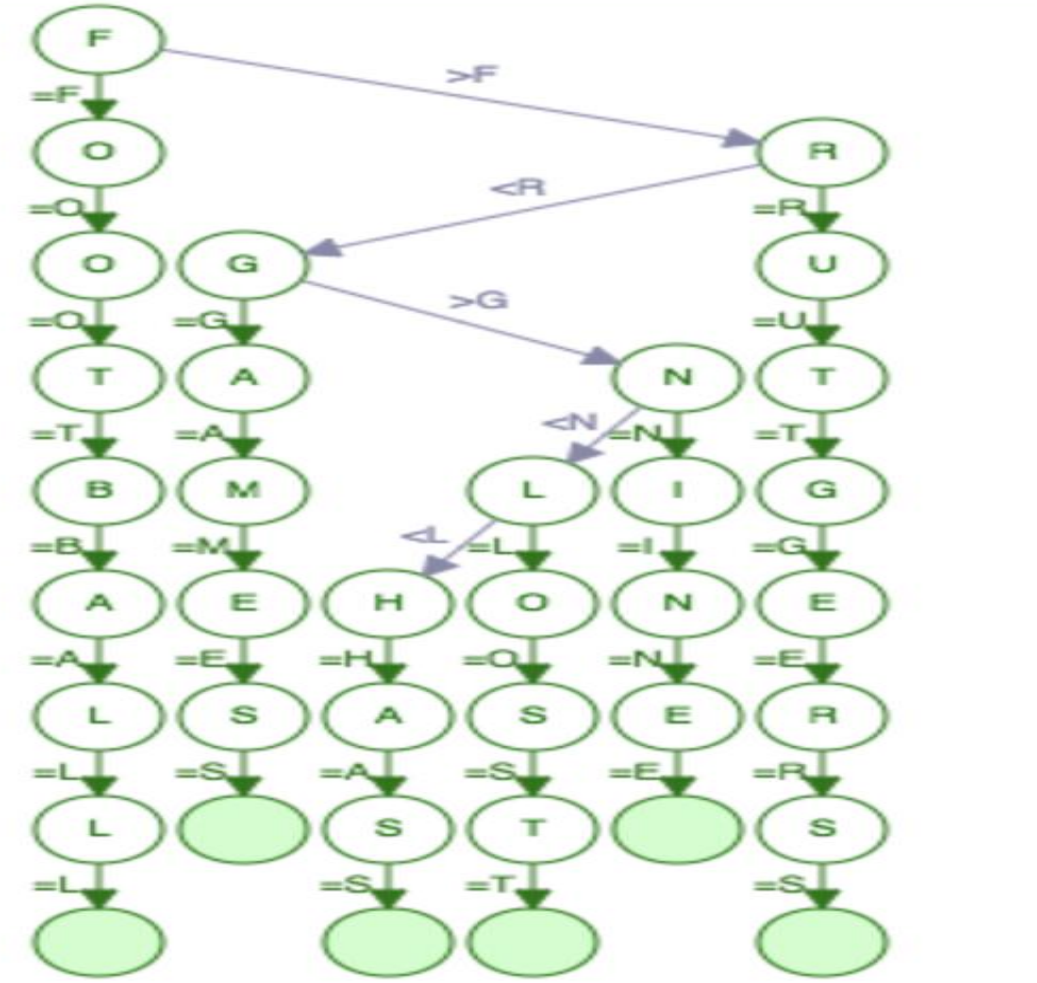
b. The TST of the string of words, when put in order also has a height equaling to 8



c. Insertion order that minimizes the height of the TST = regular sequential order of the set of strings

Inserting the phrases in terms of descending length of the word will minimize the height of the TST

Height = 7



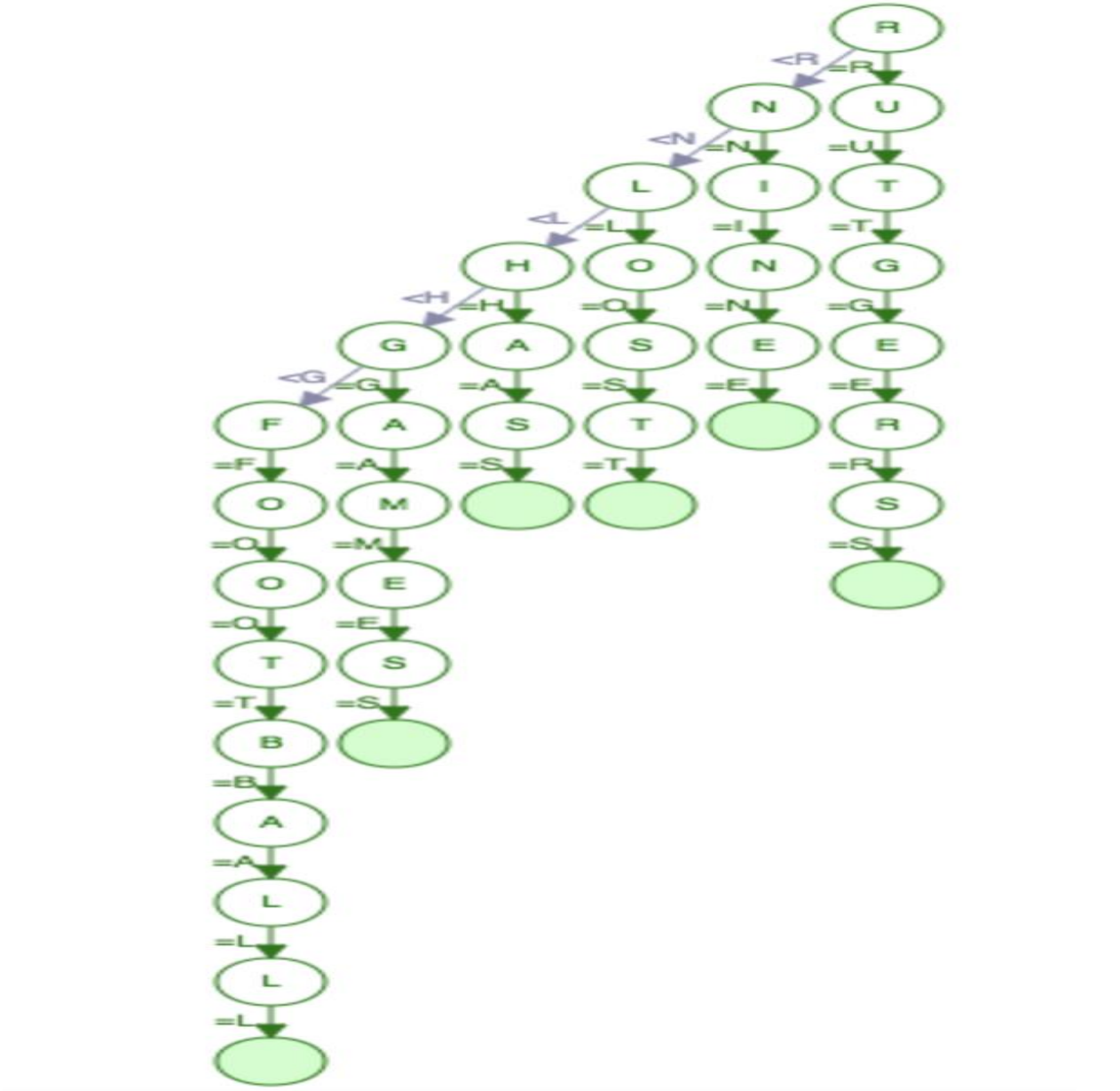
d. do not include the first node

insertion order that maximizes the height of the TST =

RUTGERS → NINE → LOST → HAS → GAMES → FOOTBALL

Adding the words in order of decreasing value of the root will maximize height of tree

Height = 12



4. If prefix free codes aren't used in Huffman coding, there would be a problem with both encoding and decoding. A problem with variable-length encodings is to figure out where code word ends. When decoding a file, it would not be known whether/how to interpret the code

position/meaning. This is why Huffman Encoding is there to remove this ambiguity by producing prefix-free codes.

5. Compression algorithms

a. Run-length coding of B-bit counts for inputs of N bits

best case scenario of RLC is when we have an
input sequence $N = 2^B-1$, which can be encoded by (B) bits, for $(B / (2^B-1))$

The principle works of counting consecutive characters which are the same
The best case when all the characters are the same (for example, AAAAAA is A6)
The worst case when no 2 consecutive characters are the same (for example BOY is B1O1Y1) so compression ratio is negative and double length
So original size is N and new size is 2N (double of N), so compression ratio is -100% since
(Original size – New Size) / (Original size) = (3-6) / 3 = -3/3 = -1 which is -100%
Worst case ratio occurs for non-compression, like a string where the next character is different from the current character → N=B

Best case ratio occurs when identical characters follow each other

b. Huffman Coding of B-bit characters for inputs of N characters

The principle works of frequency of each of characters.
The best case when all the characters are the same (for example, AAAAAA is A6 and A is converted to single bit representation and N characters take N bits so NB originally
The worst case when frequency of all characters is same and no characters is absent which makes compression ratio = 0 → B/2N
So original size is N and new size is N, so compression ratio is 0% since

(Original size – New Size) / (Original size) = 0

c. LZW compression of B-bit characters for inputs of N characters with L-bit codewords

the codebook size is fixed as a function of bit code word length, B
The principle works of using patterns which have previously occurred already

The best case when all the characters are the same (for example, AAAAAA is A6 and A is converted to single bit representation and N characters take N bits so NE originally
The worst case when a lot of different words come without any fixed pattern (absolutely random)
B bits per character
Total characters = N
Characters in 1 word (bit codeword) = L characters
Number of bits per word = (Characters in 1 word(bit codeword)) • (bits per character) = L • B
Number of words = (Total characters) / (Characters in 1 word(bit codeword)) = N / L
Original size in bits = (Total characters) • (bits per character) = N • B
new size = number of words • (new word length in bits) = (N/B) • [$\log_R(N/B)$ + (B•L-1)]
Therefore compression ratio is 0 since
(Original size – New Size) / (Original size) = 0

The new size depends on the length of each and number of words, and so does compression ratio

6. Expand dictionary and encode this sequence using LZW expression:

A B A A C D F A B A C D

Corresponding to each letter → 41 42 41 41 43 44 46 41 42 41 43 44

AB = 83
A = 41
A = 41
C = 43
D = 44
F = 46
AB = 83
A = 41
C = 43
D = 44

Encoded → 83 41 41 43 44 46 83 41 43 44