Assignment 6: Hash Tables, Data Compression, Tries

In this assignment, you will solve problems on directed and undirected graphs and submit the solution as a single PDF through Github.

You will push all of your solutions to the Assignment 4 repository through the Github Classroom. Organize your repository in such a way that it contains:

Root – Assignment 6 directory

- Assignment 6.PDF

Note: You do not have to submit any runnable code.

There is no restriction on the programming language that you use.

For the problems, you will be implementing your code as an API through which the graders will make function calls.

Include a README.md file for the GitHub repository that has your Name as the title and a description of what is inside the repository, generally.

!! INVITATION LINK: https://classroom.github.com/a/35yN32h7!!

For anyone new to Github, we encourage you to use **Github Desktop** as a simple GUI for interacting: https://help.github.com/desktop/guides/getting-started-with-github-desktop/

For people who want to be more advanced and up to industry standards, you can use the Git command line:

https://git-scm.com/downloads

A very barebones introduction to the command line (very readable, I recommend it):

http://rogerdudler.github.io/git-gu ide/

1. An array {1093, 1400, 3341, 7652, 4321, 5674, 8980} is given. Assuming the hashcodes for the integers is the same as the value of those integers, use the hash function $h(x) = x \% N$, where N is the size of the hash table to solve these problems:
    a. Separate Chaining, N = 10
        i. Represent your answer as a two row table, the first row contains the index and the row contains a comma separated list of all values hashed to that index. The list should in order of first inserted to last inserted.

i	0	1	2	3	4	5	6	7	8	9
table[i]	1400,8980	3341,4321	7652	1093	5674					

    b. Linear Probing, N = 10
        i. Represent your answer as a two row table, the first row contains the index and the second row contains the value hashed to that index.

i	0	1	2	3	4	5	6	7	8	9
table[i]	1,400	3,341	7652	1093	3321	5674	8980			

2. Your client has one million objects in a set, and wants to support queries on the set to check whether or not an object exists in the set. Your client is averse to hashing the information since the hash solutions in Problem 1 requires storing the hashed values, thus costing too much space (for example, reference space requirements of hashing in the hash tables, tries, and data compression slides). The client is fine saying an object is in the set when it is not, but only if the possibility of that occurring is very small. Explain how to solve this problem using hashing without storing the actual values. Your description does not require any pseudocode, but should follow logically and be explained fully. Give an example of how your solution might work in your explanation.

The optimal solution is to use a Bloom Filter. Construct N bit vectors for N hash functions. Run N hash functions on each value in the set and set true/false values for the bit vectors for each hash index. While it is possible to collide for two values in one hash function, it is less likely at two has functions, more unlikely at 3, … etc. This saves space because these would be only N*L 1-bit vectors.

3. Suppose the sequence of strings: {RUTGERS, FOOTBALL, HAS, LOST, NINE, GAMES } are inserted into an R-way trie, with values all equal to '1'.
    a. What is the height of the trie? Consider the root to be level '0'.
        i. **The height is 8, determined by the length of the string 'Football' while holding the root as 0.**
    b. Insert the values in-sequence into a TST. What is the height of the trie?
        i. **The height is still 8. "R" will be at root level, for height 0, so the length of the Rutgers string will be perceived as its length -1. Football is inserted to the left subtree of 'R' and then directly down. The longest potential sequence is when we insert 'NINE', which is rooted in the right subtree of 'F', being a level lower than 'FOOTBALL', 'HAS', and 'LOST'. Even then, the height of that is 4 (NINE) + 3 (F,H,L). So the height is 8.**
    c. What insertion order minimizes the height of the TST, and what is the height?
        i. **We have to consider how a trie works. Trie heights are bounded by the length of the longest continuous chain in the tree. Typically, this has been 'Football'.**

What we should prioritize is: (1) String Length, inserting the largest one first and, (2) common prefixes. For example, inserting ALGO and ALGORITHM. The height of this trie is bounded by ALGORITHM, but the entire prefix of ALGO is captured in the trie. Sorting by length gives: {FOOTBALL, RUTGERS, GAMES, NINE, LOST, HAS}. Insertion in this order would bound the height of the tree at 7 rather than 8 (FOOTBALL at node level '0' gives its max length as 7, then 'RUTGERS' at node level '1' has length = 7).

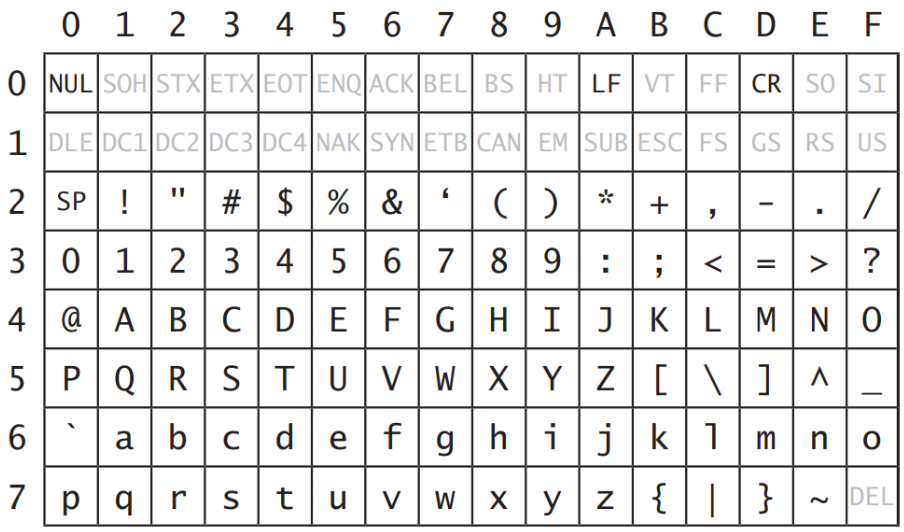
  d. What insertion order maximizes the height of the TST, and what is the height?
     i. **The answer for the trie is the same as the answer for worst case height of the BST: in-order insertion. This creates a long, one-sided chain that balloons the trie height. Looking at this, we clearly want 'FOOTBALL' to be the last value inserted into the trie. We can do a lexicographical ordering of the strings with a priority on length: {FOOTBALL,GAMES,HAS,LOST,NINE,RUTGERS}. Insert this in reverse would make the maximum height of the tree as N + L + H + G + FOOTBALL = 4+8 = 12. The 'R' at root is level 0.**

4. What happens if prefix free codes aren't used in Huffman coding – is it a problem with encoding or decoding or both?
   Prefix free codes are to prevent ambiguity or confusion when decoding sequences. If we have codes that are prefixed by other codes, then it is unclear when examining a bit string what is the proper code priority to use. While encoding works fine, the message comes out distorted when decoding. Another consequence of a greedy algorithm requiring specific constraints to work properly.

5. Explain the sequences that generate the worst and best case compression ratios occur for the below compression algorithms:
  a. Run-length coding of N bits
     i. **Best case is when N is a sequence of 2^(B)-1 0s or 2^(B)-1 1s. We can then encode it with exactly B bits, for: $\frac{B}{2^B-1}$.**
     ii. **Worst case is when we have to encode an alternating sequence and can never increase the run count beyond '1'. So we waste B bits to encode exactly one bit numbers every time. This is for N = 1, so $\frac{B}{1}$ = B.**
  b. Huffman Coding of N characters
     i. **Best case is when a single character appears all the time, being encoded with exactly one bit. This gives us $\frac{1}{B}$.**
     ii. **If we're encoding with B bits, then the worst case is when the N = 2^(B)-1 occurs equally and we have to waste B bits to encode each one of them. This ratio is then B bits to encode per B bits, for $\frac{B}{B} = 1$.**
  c. LZW of N characters
     i. **The best case scenario is when we are able to fill the codeword table with a repeating sequence of the same character. We recognize 'N', then we recognize 'NN', then 'NNN' … and so on until the entire codeword table is filled. This is a B-bit character that occurs 2^(B) times. The codeword table is then filled at $2^L - 2^B$ of these characters. Then, we continue reading that character in and can encode every sequence of these with L bits. This works out to $\frac{L}{B(2^L-2^B)}$.**
     ii. **The worst case is when we can no longer add new codewords to a table that is filled up with useless ones. Table fills up after $2^L - 2^B$. If we assume initially**

we get a best case run of one character, as in the best case, then it flips to a best case run for another character, we are unable to encode that $2^L - 2^B$ run for the second set. We have to waste an L bit codeword on a B-bit sequence. Totally,. This is $\frac{L}{B}$.

6. In the lecture for LZW Compression, I said to take it for granted that ‘41’ maps to A, ‘42’ maps to B, and so forth. The reason is because the authors did not include the codeword dictionary for converting between hexadecimal and ASCII. This is pictured below:



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	$	%	&	‘	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

This table is used by looking up the row first, then the column (A is row 4, column 1).
Using this table for referencing the base characters, expand the dictionary and encode the below sequence using LZW compression:

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

The expanded codeword table is:

81	AB
82	BA
83	AA
84	AC
85	CD
86	DF
87	FA

We can encode sequences after we’ve already seen them, but not before:

A	B	A	A	C	D	F	AB	AC	D	END
41	42	41	41	43	44	46	81	84	44	80