CS1501 Fall 2018

Solutions of Practice Questions for Midterm Exam

Fill in the Blanks Complete the statements below with the MOST APPROPRIATE words/phrases.

a) Given N keys, each with b bits, in a digital search tree, the WORST CASE search time for a key in the tree requires _____**b**_____ key comparisons, while the AVERAGE CASE search time requires ___**Theta(lgN)**___ key comparisons.

b) Delete is a problem with open-addressing hashing because ___**if a value within a cluster is deleted, values after it in the cluster may not be found**___.

c) If I have a linear probing hash table of size M, and a cluster of size C, the probability that a random key will be Inserted into the location immediately after the cluster is ___**(C+1)/M**_____.

d) The mismatched character heuristic of the Boyer-Moore algorithm has a best case run-time of ___**N/M**_____.

e) If an encoding scheme is prefix-free, it is certain that ____**no codeword is a prefix of any other codeword**_____.

f) In the 8-Queens problem each recursive call attempts to _____**place a queen somewhere in the next column of the board**_____

g) Given a multiway radix search trie in which 32-bit keys are compared 4 bits at a time, the maximum height of the tree is _____**8**_____ and interior nodes will each have up to __**2^4=16**____ children.

h) Consider an empty separate chaining hash table of size 100. If we hash 200 keys into this table, the average chain length will be _____**2**____ and the worst case chain length will be ___**200**_____.

i) Given a pattern of length M and a text of length N, the brute-force string matching algorithm discussed in lecture will require time _____Theta(N)_____ in the normal (average) case and time _____Theta(MN)_____ in the worst case.

True/False Indicate whether each of the following statements is True or False. For False answers, **explain why it is false.**

a) The brute-force algorithm to crack an n-digit PIN number tries every permutation of n different digits and has an upper-bound run-time of Theta(n!). **FALSE – the upper bound is 10^n.**

b) A multiway trie that considers k bits at a time has 2^k branches at each node. **TRUE**

c) A good hash function should utilize the entire key. **TRUE**

d) The KMP string matching algorithm improves over the brute-force algorithm in the normal (average) case. **FALSE – it improves the worst case but the normal case is still linear.**

e) When a file is compressed, its entropy level is decreased. **FALSE – it is increased**

f) The Theta asymptotic bound is an exact bound (within a constant factor) – it is both an upper and a lower bound on the asymptotic performance. **True**

g) Pruning is a technique that improves the asymptotic run-times of exhaustive search algorithms. **False. Pruning can improve the exhaustive search in practice, but typically does not change the asymptotic runtime.**

h) The Java String operator "+" appends one String to another in constant time (relative to the lengths of the Strings). **False. It requires linear time (in the length of the strings) since a new object must be created**.

i) A problem with regular multiway radix search tries is that they consume large amounts of memory due to the extensive branching at each node. **True**

j) I can avoid collisions in hashing as long as the size of my hash table, M is greater than the amount of data I am storing, N. **False. I can only avoid collisions if M is greater than or equal to the size of the key space.**

Short Answers and Calculations

1) You have two programs, Program A, which runs in time k_1N and Program B, which runs in time $k_2\log_2(N)$ for some constants k_1 and k_2. Assume that for a problem of size N_o, both programs take X seconds to execute. Approximately, how much time would each program take to run if we double the problem size? Show your work.

For Program A, since the time is linear, we know that if we double the problem size the run-time should also double. Thus, we can say 2X seconds for Program A. For Program B it is more complicated, since Program B runs in logarithmic time. However, we can still solve this with some math:

We know: $k_2\log_2(N_o) = X$

And we want to solve $k_2\log_2(2N_o) = ?$

Remembering properties of logarithms, we can rewrite the problem as follows:

$k_2\log_2(2N_o) = k_2[\log_2(2) + \log_2(N_o)] = k_2 + k_2\log_2(N_o) = k_2 + X$

2) Define what it means to have a collision in a hash table, and why we cannot usually prevent them from occurring.

A collision occurs in a hash table if, for two keys, x_1 and x_2, $h(x_1) = h(x_2)$, with $x_1 != x_2$. Collisions cannot usually be prevented, since, in most instances, the key space being used (all possible keys) is

greater in size than the table size, and, by the Pigeonhole Principle, at least two distinct keys must map to the same table location.

3) Consider a file containing the following text data:
AAABBBAAB
Trace the LZW encoding process for the file (in the same way done in handout lzw.txt, so each "step" produces a single codeword). Assume that the extended ASCII set will use codewords 0-255. For each step in the encoding, be sure to show all of the information indicated below. Note: The ASCII value for 'A' is 65.

STEP #	PREFIX MATCHED	CW OUTPUT	ADD TO DICT.
1	A	65	(AA, 256)
2	AA	256	(AAB, 257)
3	B	66	(BB, 258)
4	BB	258	(BBA, 259)
5	AAB	257	--

4) Consider 2-pass Huffman compression. How would it perform on each of the following files and why? Be specific by giving approximate compression ratios in each case.
a) A file containing 1000 of each ASCII character
b) A file containing 1000000 A's

a) Since all characters have the same frequencies, Huffman will obtain no compression, since it depends on frequency disparities to be effective.

b) Now, since all characters are the same, Huffman will approach its optimal ratio of 8/1 – since the Huffman tree will have only a single edge, thereby requiring only 1 bit to encode A, as opposed to the 8 bits required in ASCII. Note that the tree information will take up some space, but the compression should still be close to the optimal amount.

5) Consider the **mismatched character heuristic** of the **Boyer-Moore** string matching algorithm. For the pattern and text strings shown below, state and **justify** how many **total character comparisons** must be done in order to match the pattern shown within the text string. Justify your answer using the **right array** for the pattern.

Text: ABCDXABCDYABCDZABCDE

Pattern: ABCDE

right(A)=0, right(B)=1, right(C)=2, right(E)=3 and right(E)=4. All other right array entries are -1. The total number of character comparisons is 8. For each of the first 3 mismatches the maximum skip value of 5 is used. The final 5 comparisons are used to match the pattern right to left.

6) Justify in detail how many character comparisons are required to find a string in a DLB in the worst case. Assume that your DLB has N strings, each with a maximum of K characters, and that your alphabet has S possible characters in it.

Recall that a DLB "node" consists of a number of "nodelets" – one "nodelet" per possible character for a given prefix in the dictionary. In the worst case, all S characters in a given "node" are used, thereby requiring S "nodelets". If a character being searched for in a given "node" happens to be in the last "nodelet", S character comparisons will be required in the examination of a single position within the string. If this worst case occurs for each position within the string, a total of SK character comparisons

will be required in total. Note that this worst case is extremely unlikely, since after the first few levels the "nodes" typically have very few "nodelets"' in them (which is why we use the DLB in the first place).

7) Consider the crossword puzzle algorithm that you implemented in Assignment 1. Consider the board below.

	0	1	2	3
0	H	E	L	P
1	A	N	+	+
2	+	+	+	+
3	+	+	+	+

You are currently looking at square [1][2] and proceeding through the board in a row-wise fashion.

a) Assuming that you will not backtrack to square [1][1], what is the maximum number of recursive calls (to square [1][3]) that is possible from this square? Justify your answer.

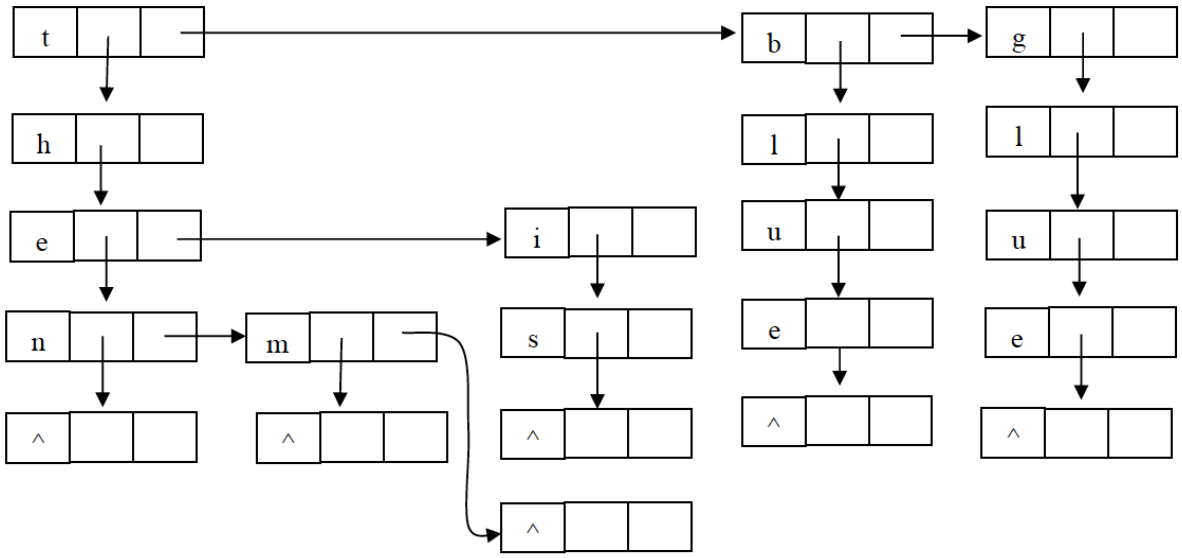
A maximum of 26 recursive calls can be made from square [1][2], one for each possible letter in the alphabet.

b) Assuming a "typical" dictionary of English words, how likely is this maximum number to occur? Thoroughly justify your answer.

The maximum number is not very likely. This is because to make a recursive call the character at position [1][2] must form a valid prefix in both row [1] and in column [2]. For most of the letters in English this is not the case. For example, the letter 'B' in position [1][2] would generate strings "ANB" and "LB". "ANB" is not a prefix to a word in most English dictionaries and LB is definitely not a prefix to a word.

8) Consider an emtpy de la Briandais Tree, which uses the lower case letters (plus a string termination character) as its alphabet, using the implementation that we discussed in lecture. Also consider the following strings: **then them the this blue glue**. Draw the de la Briandais tree that results after inserting the strings shown in the order shown above.

Note: Empty squares indicate null references and the ^ is the terminator character



9) Consider the two open addressing hash tables, with **h(x) = x mod 13**, shown below. Also, consider the following keys (in order): 23, 17, 36, 24, 26, 37, 49.

a) Assume that **linear probing** is being used for collision resolution. Show the table after the keys shown above are inserted in the order shown above.

b) Assume now that **double hashing** is being used for collision resolution, with **h_2(x) = (x mod 11) + 1**. Show the table after the keys shown above are inserted in the order shown above. For full credit for each collision indicate the h_2(x) value for the key.

Linear Probing	
Index	key
0	**26**
1	**37**
2	**49**
3	
4	**17**
5	
6	
7	
8	
9	
10	**23**
11	**36**
12	**24**

Double Hashing	
Index	key
0	**26**
1	**36**
2	
3	**37**
4	**17**
5	
6	
7	
8	
9	**49**
10	**23**
11	**24**
12	

h2(36) = 4

h2(37) = 5

h2(49) = 6

Coding

1)

```
public boolean find(String item)
{
        int index = h(item);

        for (int i = 0; i < table.length; i++)
        {
               int curr = (index + i) % table.length;
               if (table[curr] == null)
                    return false;
               else if (table[curr].equals(item))
                    return true;
        }
        return false; // cycled through all locations - not found
}
```