CPSC 221: Sample Final Exam Questions for Practice: Solutions

Updated:
April 13, 2017 @ 01:20 – Q5d: divide by 106+1 rather than 106 – no change to final answer. Q1a: better explanation.
April 12, 2017 @ 20:35 (divide by 2 in part of Question 3(b)'s solution)

Please don't look at these solutions until you've made a serious attempt at answering the questions.

Question 1

a) If the sequences (chains) are not sorted then the worst-case running time is n * O(1) = O(n). It does not matter if we insert at: (i) the start of each chain (in this case, use each chain's head pointer to perform the insertion in O(1)) time; or (ii) at the end of each chain (in this case, keep a tail pointer for each chain so that you don't have to traverse the whole chain to do the insertion). It also doesn't matter if all the values get hashed to one chain or not, we do O(1) steps for every insertion no matter where it goes.

b) If each chain is stored in sorted order, then the worst-case running time is O(n^2) because each new entry has to traverse all or most of the linked list before being inserted into the list. Note that it is not O(n lg n) time.

c) If we used a BST whose root node is anchored at the hash cell location, then our answer would not change in the worst-case. That's because, in the worst-case, we could get a "stick" as our BST, which would be the same as a linked list. This would be the case, for example if we inserted nodes that already were in ascending or descending order in the input.

Question 2

To count the number of 1's in A, we can do a binary search on each row of A to determine the position of the last 1 (or first 0) in that row. Then we can simply maintain a sum (based on the index values) to determine the total number of 1's in A. It takes O(lg n) time to find the last 1 in *each* row because we can perform a binary search (i.e., we can simply test whether or not the next value is a zero or the end of the list). When we consider all n rows, this takes O(n lg n) time—plus O(n) time to add the results of the n rows together. Overall, this is O(n lg n + n) = O(n lg n) time.

Question 3

a) $\Theta(n^2)$ for Insertion Sort

b) Input:

01010101... There are $n/2$ such "01" pairs.

or

111...1 00...0 Each of these 2 strings contains $n/2$ entries.

These cases demonstrate worst-case behaviour because we must go through at least $n/2$ entries for $n/2$ of the numbers in the list.

For example, just considering the cases for even n (the odd case is similar):

For the first input expression (01010101...) during Insertion Sort:

For the 1's, we do 1 comparison each—this is $n/2$ comparisons in all.

For the 0's, we do $0 + 2 + 3 + 4 + \ldots + n/2$ comparisons $= (n/2)(n/2 + 1)/2 - 1 = n^2/8 + n/4 - 1 = n^2/8 + n/4 - 1$ comparisons.

Let $T(n)$ be the sum of these two expressions.

$T(n) = n/2 + n^2/8 + n/4 - 1 \leq n^2$ for all $n \geq 1$.

Therefore, $T(n) \in O(n^2)$.

Similarly, for Big-Ω:
$T(n) = n/2 + n^2/8 + n/4 - 1 \geq 1/8\ n^2$ for all $n \geq 2$.
Therefore, $T(n) \in \Omega(n^2)$.

Finally, because $T(n) \in O(n^2)$ and $T(n) \in \Omega(n^2)$, it follows that $T(n) \in \Theta(n^2)$ for all $n \geq 2$.

And a similar—but even simpler—argument can be applied to the 111...1 00...0 input string shown above.

c) Input:

000... There are n such entries.

or

111... There are n such entries.

Note that in both of these cases, we always get a bad partition split, and this causes Quicksort to be $\Theta(n^2)$.

Question 4

Let us assume we are working with an array, for convenience. (Note that we can copy the elements from another list to an array in O(n) time.) First, we sort the objects of A using a good sorting algorithm like Mergesort or Heapsort. This takes O(n lg n) time. Then, we can go through the sorted sequence, from start to end (index $k = 0$ to $n - 1$), and remove all duplicates. It takes O(n) time to read through the list (searching for equal values among adjacent objects). We can remove the duplicates in place by maintaining a separate index j ($j \leq k \leq n$) to shift-left the elements (as needed), or by simply copying unique values to a new/final array, vector, or linked list. Overall, this is an O(n lg n) algorithm because sorting is the most expensive step and it takes O(n lg n) time, and copying/iterating takes O(n) time.

Question 5

a) The number of data entries (DEs) per leaf page is given by:
`floor`((4096 – 3(8)) bytes/page / (30 + 5(4))) bytes/DE)
= `floor`(4072 / 50) DE/page
= 81 DE/page … but we want an even number, so that would be:
80 data entries per leaf page

b) `ceiling`(10,000 DE / 80 DE/page) = 125 pages (these will be level-one internal pages)

c) The internal pages have similar but not identical structure to the leaf pages. They still need the 3 regular pointers (parent, left sibling, and right sibling); but, they have one extra child pointer in addition to the key-pointer (i.e., key and child pointer) pairs. That's because each *key* that's found in the internal pages of this index can discriminate among two children, and therefore each key can be used to direct the search to either the left child or the right child, as appropriate. In summary, we will need $k + 1$ child pointers if we have k keys; and this is equivalent to saying k key-pointer pairs + 1 extra pointer for the rightmost child.

`floor`((4096 – 3(8) – 8) bytes/page / (30 + 8) bytes/key-pointer) where the –8 is for the extra 8-byte child pointer for the right child of the final key
= `floor`(4064 / 38) key-pointer pairs
= `floor`(106.95) key-pointer pairs
= 106 key-pointer pairs per internal page (and we already accounted for the extra pointer—however, if we didn't, just make sure there's room for it)

We will divide the key-pointer pairs among these two level-one parents, so that they are at least half full each.

d) We have 125 leaf pages. How many parent pages are needed to support 125 leaf pages?

`ceiling`(125 leaf pages / (106+1) key-child pairs/parent page)

= 2 parent pages (also known as level-one internal pages)
… because there is a 1:1 relationship between a leaf page and a parent's child pointer.

Going one level up the tree: How many parents are needed to support 2 level-one pages?

`ceiling`(2 level-one pages / (106+1) key-child pairs/parent page)
= 1 parent page (also known as the root page)

Since we have reached a single parent page, that means we've reached the root. We stop here.

Summary:

- 125 leaf pages
- 2 level-one pages
- 1 root page

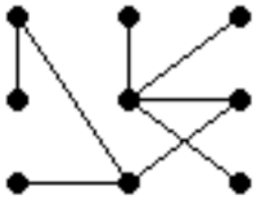
e) If there are 3 levels, then at a minimum, we need:

a. One root page that has at least one key in it. Note that the root page is not subject to the "half full" rule; it is the only page that can have this quality.
b. Two level-one pages. You cannot have only one level-one page because the root key must discriminate between two pages (i.e., two children). Furthermore, these two level-one pages have to be at least half full. That means that we have to have a minimum of 53 key-child pairs in each of the two level-one pages.
c. Since there are 53 keys in the first level-one page, it has to have 54 children. This means it has 54 leaf pages that are at least half full. Similarly for the other level-one page. This means we have 54 + 54 = 108 leaf pages that are at least half full, which means we must have a minimum of 108(40) = 4320 records.

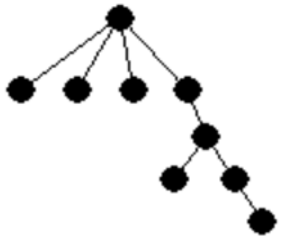
Thus, there are at least 4320 bands' records in a 3-level B+ tree index having the above constraints.

Question 6

a) Here is one example for Graph A. For our root node, we can pick the center node in the following diagram (same diagram as shown in the question).



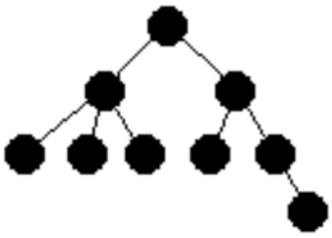
… and this yields the following tree:



This tree is easy to draw because no nodes share the same parent, and all nodes in the graph are connected.

To provide a balanced tree for Graph A, we need to be more careful with our selection of root node. The above diagram does not work because leaves appear at more than the last two levels. Above, for example, we have leaves at levels 1, 3, and 4—where level 0 is defined as the root node.

Instead, let us pick the rightmost node of level-one (from the preceding diagram) as our new root node. This will yield a balanced tree because the leaves will appear on the last two levels only:



b) For Graph B, we note that one of the nodes has two parents. This means that there is a violation of the tree rule: there must be a unique path between any two nodes in the tree. Therefore, we cannot turn graph B into a tree; and therefore, we cannot turn it into a balanced tree either.

Question 7

There are $f * m * l$ unique (complete) names. Therefore, by the Pigeonhole Principle, we must have at least $fml + 1$ people before we are guaranteed that we will get a duplicate name for a person. As long as the number of people n in the original problem is greater than $fml + 1$, we are guaranteed to have a "collision". If we do not have at least this many people, then we *might* still get a collision, but we do not necessarily have to get a collision.

Question 8

A single entry $c_{i,j}$ in the result requires n multiplications because k goes from 1 to n.

The matrix is an $n \times n$ matrix; therefore, there are n^2 $c_{i,j}$'s in the matrix.

Therefore, we have $T(n) = n^2\ n = n^3$ multiplications $\Rightarrow \Theta(n^3)$ multiplications.