Final Exam in Data Structures

Department of Information Technology
Uppsala University
Lecturers: Parosh Aziz Abdulla, Jonathan Cederberg
Location: PB
Time: 5 h
No books or calculator allowed

Directions:

1. Answer only one problem on each sheet of paper
2. Do not write on the back of the paper
3. Write your name on each sheet of paper
4. **Important** Unless explicitly stated otherwise, justify you answer carefully!! Answers without justification do not give any credits.

Good Luck!

WARNING! Do not believe what you read! These solutions were developed by request from students. They might contain errors. Actually, it is very likely that they do. If you discover such errors whilst reading, please drop us a note at `jonathan.cederberg@it.uu.se`.

Problem 1 (10p)

Order these functions in order of asymptotic growth rate, with the most rapidly growing first. If two of them have the same asymptotic growth rate, state that fact. No proofs are needed.

$$\lg(n^2) \qquad 0.000001n \qquad \lg n \qquad 2^{2^n} \qquad 4^n \qquad n \lg n \qquad \lg(2^n)$$

Solution to 1 We start by reducing the functions to their simplest form. This means mainly simplifying logarithms. We can also remove any leading constants to figure out the tight asymptotic bound:

- $\lg(n^2) = 2\lg n$ so asymptotic growth is $\Theta(\lg n)$ since multiplication by a constant does not affect the growth rate.
- $0.000001n$ cannot be simplified, but has again a constant multiplier. Thus asymptotic growth is $\Theta(n)$.
- $\lg n$ is obviously $\Theta(\lg n)$.
- 2^{2^n} cannot be simplified, so it is $\Theta(2^{2^n})$.
- 4^n cannot be simplified, so it is $\Theta(4^n)$.
- $n \lg n$ cannot be simplified, so it is $\Theta(n \lg n)$.
- $\lg(2^n) = n \lg 2 = n \cdot 1 = n$ is obviously $\Theta(n)$.

With this preprocessing, we can easily just write down the answer, namely

Answer

$$\underbrace{\lg n \qquad \lg(n^2)}_{\text{same growth rate}} \qquad \underbrace{0.000001n \qquad \lg(2^n)}_{\text{same growth rate}} \qquad n\lg n \qquad 4^n \qquad 2^{2^n}$$

Problem 2 (10p)

a) Prove or disprove: $n^2 + 14n + 3 = \mathcal{O}(n^2)$

b) A function f is monotonic if $m \leq n \implies f(m) \leq f(n)$ Let f and g be monotonic functions. Prove or disprove: $\min(f(n), g(n)) = \Omega(f(n) + g(n))$

Solution to 2

a) This is true, by all our standard knowledge. To prove it, we apply the defintion of $\mathcal{O}$:

$$\begin{aligned}
& n^2 + 14n + 3 = \mathcal{O}(n^2) \\
\Leftrightarrow \quad & \exists c, n_0 : \forall n > n_0 : n^2 + 14n + 3 \leq c \cdot n^2 \\
\Leftrightarrow \quad & \exists c, n_0 : \forall n > n_0 : 14n + 3 \leq c \cdot n^2 - n^2 = (c-1) \cdot n^2 \\
\Leftrightarrow \quad & \exists c, n_0 : \forall n > n_0 : 0 \leq (c-1)n^2 - 14n - 3 = \\
& = (c-18) \cdot n^2 + 14n^2 - 14n + 3n^2 - 3 = \\
& = \underbrace{(c-18) \cdot n^2}_{\geq 0 \text{ if } c \geq 18 \text{ and } n \geq 0} + \underbrace{14n \cdot (n-1)}_{\geq 0 \text{ if } n \geq 1} + \underbrace{3(n^2-1)}_{\geq 0 \text{ if } n \geq 1}
\end{aligned}$$

So we can conclude that picking $c = 18$ and $n_0 = 1$ will make the inequality hold, and thus the required numbers exists

b) Ok, this is a tricky one. Let's start by understanding the question. We have two monotonic functions. We want to know whether it is possible to "squeeze" the min of these two functions from below (that is what Ω means), using the sum.

Now, how does the min function behave? It will always take on the smallest value of the two functions. In particular, if for example $f(n) \leq g(n)$ for all n, then $\min(f(n), g(n)) = f(n)$. We know that if one function grows faster than another, it will eventually dominate in terms of magnitude. Therefore, in the limit, $\min(f(n), g(n))$ will behave like the slowest growing of the two. We have thus concluded that the function we want to squeeze behaves like the smaller one.

Let's now look at the function $f(n) + g(n)$. We want to know the asymptotic behaviour. We know from the course material that a sum grows like the fastest growing term in the sum.

Now combine these two insights. We want to use something that behaves like the fastest growing of the two, to bound the slowest growing of the two. This is clearly not possible.

Ok, this reasoning might be hard, but the quest here is not to be rigorous, but to decide whether the statement is true or false in order do proceed to a proof or counter example. We have concluded that we need to do the latter.

How do we find this counter example? Let us just try something very simple and see if it works. Just make sure to pick two functions with different growth rate. Suppose $f(n) = n$ and $g(n) = n^2$. Se we have $n \leq n^2$ for all positive n, we have $min(f(n), g(n)) = n$. We can also conclude that $f(n)+g(n) = n+n^2$. Going back to the original question, we get $n = \Omega(n^2)$ which is not true.

If you are not comfortable with lower bounds, then you can convert the last statement to the equivalent statement $n^2 = \mathcal{O}(n)$ which is of course also not true.

Problem 3 (10p)

What is the maximum number of times during the execution of quicksort that the largest element can be moved, for an array of N elements. Explain your answer in no more than **three** lines.

Solution to 3

WARNING! The question states that you should give your answer in **at most three lines**! Failure to do so will get you 0 points!

This question requires us to know the inner workings of quicksort. Let's recap: the procedure is recursive, and works in a divide-and-conquer manner. The procedure successively picks a pivot element, partitions the remaining elements depending on their relation to the pivot element. In pseudo-code:

```
PARTITION($A, p, r$)
$x \leftarrow A[r]$
$i \leftarrow p - 1$
for $j \leftarrow p$ to $r - 1$
    do if $A[j] \leq x$
        then $i \leftarrow i + 1$
             exchange $A[i] \leftrightarrow A[j]$
exchange $A[i + 1] \leftrightarrow A[r]$
return $i + 1$
```

Now, where are elements moved? We know that the QUICKSORT procedure itself does not move elements, so all movements are made in PARTITION. In there, in turn, the only movements are by means of exchanges, as can be immediately seen by the psuedo-code.

One more thing about movements that we can see (which might be a bit harder though) but more importantly that should be clear from our intuition of the inner workings of QUICKSORT, is that bigger elements always move to the right.

This last observation immediately gives us an upper bound on the number of movements of the biggest element. Since it always moves to the right it can at most move $n-1$ times for an array of size n.

This being an upper bound, if we believe that this is actually the answer, we need to devise an arrangement of keys that results in that many movements. To start with, it has to initially be placed in the rightmost position. Now it has to be moved exacly one step at a time. How can this be done?

If you have no intuition at this stage of how the arrangement should be, my suggestion is that you try some random assignment. Why not try the sequence $[9, 1, 5, 2, 6, 7, 4]$?

1. 4 is pivot element, resulting array: [1,2,4,9,6,7,5], and 2 movements of 9.
2. Recurse: partition $[1, 2]$ and $[9, 6, 7, 5]$.
3. Result: $[1, 2]$ and $[5, 6, 7, 9]$. and one movement of the maximum element.

We got 3 moves in total. This might not be that enlightening. If not, then try a bigger example. The key is that we want to move the largest element in as small a step as possible. How do we achieve this?

We can see from the example above, and in general from the pseudo-code, that moving the element just one step means that i and j should have a difference of 1, i.e. $j = i + 1$. So what is i and j? They are indicies describing where the subarray of elements smaller and larger than the pivot element end, respectively. In particular, the size of the subarray of larger elements is exactly $j - i$, which means that in our case the subarray of elements larger than the pivot element should be 1. Final conclusion: the pivot element needs to be the second largest, and the largest should be at the first position. Then the largest element will be moved one step in each iteration of the loop on lines 3-6. Finally it will be moved once more when it is swapped for the pivot element on line 7.

Now we have a lengthy reasoning, but the justification should be 3 lines only. A suggestion could look like:

Answer

Answer: $n-1$
Since larger elements are always moved to the right, $n-1$ is the max. If $A[1]$ is the largest and $A[N]$ is the second largest, it will be moved each iteration and finally be swapped for the pivot.

Problem 4 (10p)

a) Simulate counting sort on the keys below. Show all steps.

$$10 \quad 3 \quad 6 \quad 4 \quad 8 \quad 7$$

b) Describe in one sentence when using counting sort is a bad idea.

Solution to 4

b) We know that counting sort is a linear sorting algorithm, and that the complexity is $\Theta(n+k)$. Here n is the number of elements to sort, and k is the size of the largest key. If the key size is linearly bounded by the number of elements, i.e. $k = \mathcal{O}(n)$, we can reduce the complexity to just $\Theta(n)$. However, if this is not true, this is no longer true. In other words, if the key size is not linearly bounded by the number of elements, we no longer have linear complexity for counting sort. Put more succinctly, as an answer to the question:

Answer

When the key size is not linearly bounded by the number of elements.

or slightly less technically put, and not as precise (but still yielding full points)

Answer

When the key size is larger than the number of elements.

Problem 5 (15p)

Is the operation of insertion in a binary search tree commutative in the sense that inserting x and then y into a binary search tree gives the same tree as inserting y and then x? Argue why it is so (in no more than fives lines) or give a counter-example.

Solution to 5

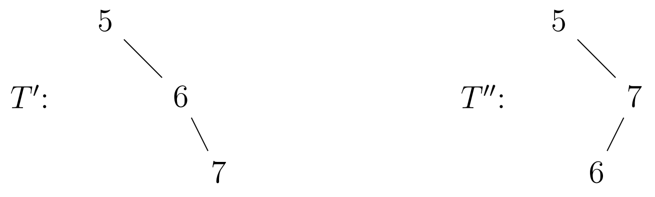
WARNING! The question states that you should give your answer in **no more than five lines**! Failure to do so will get you 0 points!

This problem requires us to know the basics of tree insertion. The key observation is that tree insertion does not alter the structure of the tree that is already there. *It only adds new leaves.* Also, the procedure is completely deterministic, so an element will always be put as the same leaf in two copies of the same tree.

From these fact, we can conclude that if x and y happened to be destined for the same leaf position in the tree T, the trees $T_x = \textsc{Tree-Insert}(T, x)$ and $T_y = \textsc{Tree-Insert}(T, y)$ would be identical in terms of the shape. However, they would clearly differ in the content of the leaf just added. We have already noted that the part of the tree already present will not change due to additional insertions, so no matter where y and x ends up in T_x and T_y, they will already be different.

Ok, we have concluded that the answer is "No". So, how do we derive a counter example? The answer is: make it minimal. It is a good strategy in general to start with a very simple example to see if it works out as the desired counter example.

Just pick any root element, say 5. That is a very small tree, so we use it as a start. Then we are to decide x and y. According to our previous reasoning, they should "be destined for the same leaf position". This means concretely that they should both end up in the same subtree, i.e. both be smaller or bigger than the root. Let's pick them larger: $x = 6$ and $y = 7$. Then inserting first x and then y would yield the tree T' below, whereas the reverse order of insertion would yield the tree T''.

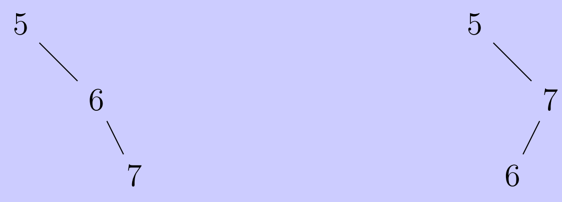


The two trees resulting from inserting 6 and 7

So, apparently we have the desired counter example. A reasonable presentation of this answer could be

Answer

No, insertion is not commutative in this sense. A counter-example would be insertion of the elements 6 and 7 into the tree containing only the element 5. The two trees depicted below shows the two possible trees resulting from this.



Problem 6 (15p)

Consider a hash table H of a given size $n > 0$. Also assume a hash function $h(k) = (k \mod n)$. Does increasing the size of H to $2n$ (and modifying the hash function accordingly) necessarily mean that the number of collisions decreases by approximately one half? (Your answer should not be longer than three lines).

Solution to 6

WARNING! The question states that you should give your answer in **at most three lines**! Failure to do so will get you 0 points!

This question might be tricky, but it is important to think of the main concept here. We start by asking ourselves the fundamental question "when do we get collisions?". If we do not know this, then answering the question is not possible.

The answer is of course that we get collisions when the hash function evaluates to the same value for two keys. Formally, we get a collisions whenever we for two keys k_1, k_2 have $h(k_1) = h(k_2)$.

Now, we might have some immediate intuition about this particular question, along the lines that "well yeah, since the range of the hash function has doubled, there is twice as much room for the elements". However, we must think one step further: having more room in our hash table is not enough to have the number of collision decrease. It must also be the case that the hash functions actually maps keys to these new positions. Will this happen? Well, it depends, as we shall see.

The key is to note that for some sets of keys, the hash function will actually evaluate to the same value, no matter whether the size is n or $2n$. For example, the keys $1, 5, 9, 13$ and 17 will hash to position 1 both when the size of the hash table is 2 and 4.

To sum up the insights, the very key fact to keep in mind here is that the effectiveness of using a hash table depends three things:

1. the size of the hash table
2. the hash function
3. the input distribution

We can see from these three points that the answer to the original question is negative, and a very concise answer would be:

Answer

No, because the keys might not be hashed differently in the hash table. For example, all numbers of the form $2nm + k$ will hash to the same position (k), no matter if the size is n or $2n$.

Problem 7 (15p)

Consider inserting the following keys into a hash table of length $m = 10$:

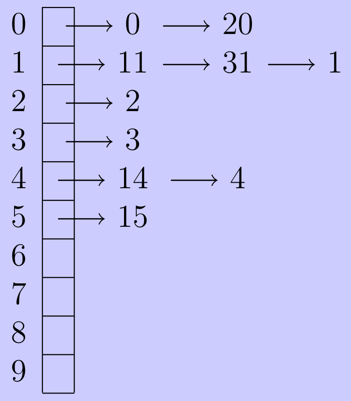
$$2 \;\; 4 \;\; 1 \;\; 20 \;\; 15 \;\; 31 \;\; 14 \;\; 3 \;\; 0 \;\; 11$$

The auxiliary hash function is given by $(k \mod m)$. Draw the resulting hash table if we use chaining for collision resolution.

Solution to 7 There is not much to say here, just compute the keys and put them in the right place. One common error is to not insert the elements at the head of the list, but instead at the end. This is not correct, and no minor flaw either. The reason for this is that it takes linear time to insert into the end of a list, and constant time to insert at the head.

For example, the resulting hash table above should be

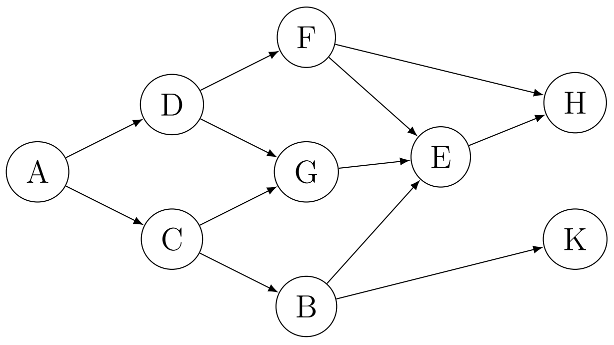
Answer



and nothing else.

Problem 8 (15p)

Give 2 possible DFS traversals of the graph below, listing the nodes in the order they are discovered. A should be the starting vertex.



Solution to 8 Any two of the following exhaustive list of traversals is ok.

Answer

- $ACBKEHGDF$
- $ACBEHKGDF$
- $ACGEHBKDF$
- $ADGEHFCBK$
- $ADFEHGCBK$
- $ADFHEGCBK$