Question 1: [5 points] Put a checkmark in front of the statements that are mathematically correct.

Solution:

	correct
☑	$n - 4\sqrt{n} + \log n \in \Theta(n)$
☐	$\sqrt{\log n} \in \Omega(\log \sqrt{n})$
☑	$\sum_{i=0}^{n} i \in O(n^3)$
☑	$\log n^n \in \Theta(n \log n)$
☐	If $f(n) \in \Omega(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in \Omega(h(n))$.

Question 2: [4 points] In this question we consider the effect of using min-heaps or sorted arrays to maintain a set of integers in a data structure. (a) Describe an operation on a set of integers that can be executed more efficiently when using a min-heap to store the set than when using a sorted array. (b) Describe an operation on a set of integers that can be executed more efficiently when using a sorted array to store the set than when using a min-heap. Explain your answers by giving the running time of the operations for both structures.

Solution: Possible answers:

(a):

	Min-heap	Sorted array
Extract-Min	$\Theta(\log n)$	$\Theta(n)$

Motivation (additional info for practice exam): We can remove the minimum element from a min-heap in $\Theta(\log n)$ time by moving it from the root of the heap to the final position. Afterwards we call min-heapify on the root which restores heap properties in $O(\log n)$ time. Removing the minimum from an array requires that we shift all other elements one position to the left which takes $\Theta(n)$ time.

Insert	$\Theta(\log n)$	$\Theta(n)$

Motivation (additional info for practice exam): We can insert an element in a min-heap in $\Theta(\log n)$ time by inserting it at the end of the heap as a key with value ∞ and then calling reduce-key which takes $\Theta(\log n)$ time. Inserting an element in a sorted array may take up to $\Theta(n)$ time as we need to make space by shifting all adjacent elements by one position.

(b):

	Min-heap	Sorted array
Maximum	$\Theta(n)$	$\Theta(1)$

Motivation (additional info for practice exam): The maximum of a min-heap can be in any of the $\Omega(n)$ leaves. Finding it takes $\Theta(n)$ time in the worst case. In a sorted array the maximum value is in the last cell of the array which we can access in $\Theta(1)$ time.

	Min-heap	Sorted array
Successor	$\Theta(n)$	$\Theta(1)$

Motivation (additional info for practice exam): As there is no structure in a heap besides the parent-child relation the successor of a node can be 'anywhere' in a heap. Particularly for the second-largest item the successor could be in any of the leaves of the heap. Finding it then also may take $\Theta(n)$ time. For a sorted array the successor of the element at index i is at index $i + 1$ which we can access in $O(1)$ time.

Question 3: [2+2+2 points] Give asymptotic upper and lower bounds for $T(n)$ for the following two recurrences. Make your bounds as tight as possible, and justify your answers.

(a) $T(n) = 4T(n/2) + n^3$

Solution: We have $a = 4$, $b = 2$ and $f(n) = n^3$. Therefore, we also have $\log_b a = \log_2 4 = 2$, and thus $f(n) = \Omega(n^{\log_b a + \varepsilon})$.
We thus consider case 3 of the MT. Pick $\varepsilon = 0.5$ then $f(n) = n^3 = \Omega(n^{2+0.5})$. If the regularity also holds then we can use case 3 of the MT. We need that $af(n/b) \leq c \cdot f(n)$ for $0 < c < 1$. We need that $4(n/2)^3 \leq c \cdot n^3$, so $0.5n^3 \leq c \cdot n^3$, which is true for all $1 > c \geq 0.5$. Take for example $c = 0.5$. By MT case 3 we have that $T(n) = \Theta(f(n)) = \Theta(n^3)$.

(b) $T(n) = 9T(n/3) + n \log^2 n$

Solution: We have $a = 9$, $b = 3$ and $f(n) = n \log^2 n$. Therefore, we also have $\log_b a = 2$, and thus $f(n) = n \log^2 n = O(n^{\log_b a - \varepsilon}) = O(n^{2-\varepsilon})$. Pick $\varepsilon = 0.5$, then indeed as required $n \log^2 n = O(n^{2-0.5})$ Then by case 1 of MT, $T(n) = \Theta(n^{\log_b a}) = \Theta(n^2)$.

(c) Prove that the solution to the recurrence $T(n) = T(n-1) + n - 2$ with $T(1) = 1$ is $T(n) = O(n^2)$.

Solution: We use substitution. We need to prove that $T(n) \leq c \cdot n^2$ for all $n \geq n_0$, where $c > 0$, $n_0 > 0$.

Base case:
For $n = 1$ we need that $T(1) = 1 \leq c \cdot 1^2$, rewriting gives $c \geq 1$. So as long as $c \geq 1$ the base case holds.

Inductive step:

$$
\begin{array}{lll}
T(n) = & T(n-1) + n - 2 & \Rightarrow \quad \{\text{by IH}\} \\
T(n) \leq & c \cdot (n-1)^2 + n - 2 & \Leftrightarrow \quad \{\text{reorder terms}\} \\
T(n) \leq & cn^2 - 2cn + c + n - 2 & \Leftrightarrow \quad \{\text{reorder terms}\} \\
T(n) \leq & cn^2 + (1-2c)n + c - 2 & \Rightarrow \quad \{\text{if } T(n) \leq x - 2, \text{ then also } T(n) \leq x\} \\
T(n) \leq & cn^2 + (1-2c)n + c & \Rightarrow \quad \{\text{if } n \geq 1, \text{ then } cn \geq c\} \\
T(n) \leq & cn^2 + (1-2c)n + cn & \Leftrightarrow \quad \{\text{reorder terms}\} \\
T(n) \leq & cn^2 + (1-c)n & \Leftrightarrow \quad \{\text{if } c \geq 1, \text{ then } (1-c)n \leq 0\} \\
T(n) \leq & cn^2 &
\end{array}
$$

Let $c = 1$ and $n_0 = 1$ then by induction we have that $T(n) \leq cn^2$ for all $n \geq n_0$. Thus we have that $T(n) = O(n^2)$.

Question 4: [3 points] Assume that algorithm A and algorithm B both solve the same problem. In the best case algorithm A takes $T_A(n)$ steps and algorithm B takes $T_B(n)$ steps with $T_A(n) \in \Theta(n)$ and $T_B(n) \in \Theta(\sqrt{n})$. Does algorithm B always takes fewer steps than algorithm A for any input of size n? Give at least two reasons why or why not.

Solution:

- The statements are only about the best-case, that does not say anything about the worst-case behavior.
- There are constants and lower-order terms hidden in the asymptotic notation which may matter for small values of n.
- The input that gives the best-case performance for A, need not be a good input for B. It is possible this is the worst-case input for B.
- For small values of n the bounds don't need to hold as asymptotic functions only hold from a certain n_0 upwards.

Question 5: [5 points] You are given a binary search tree T with n nodes having integer keys. With each node $x \in T$ you would like to store the sum of all keys in the subtree rooted at x as satellite data. Write an algorithm that computes this value for all nodes in the tree in $O(n)$ time total and stores it with each node x as satellite data x_{sum}.
Don't forget to analyze the running time and prove the correctness of your algorithm.

Solution:
Algorithm:
Run a PostOrderTreeWalk where at each internal node we compute the sum as $n_{sum} = left[n]_{sum} + right[n]_{sum} + n.key$ and at each NIL-leaf l the value is $l_{sum} = 0$.

Runtime analysis:
A treewalk takes $O(n)$ time assuming the time spent per node is $O(1)$. Indeed, we can compute the value of n_{sum} based on all locally available computed values (including in the children). Hence our algorithm takes $O(n)$ time.

Proof of correctness:
Base case:
The sub-tree rooted at a NIL-leaf does not contain any elements. Hence trivially the sum of all those elements is 0 and l_{sum} is correctly set to 0.

Step:
When at an internal node, PostOrderTreeWalk first visits the left and right child and by IH it correctly computes the sum of all elements for these subtrees. But then $left[x]_{sum}$ is the sum of all elements in the subtree rooted at $left[x]$ and $right[x]_{sun}$ is the sum of all elements in the subtree rooted at $right[x]$. The sum of all elements in the tree at x contains the sum of all elements in the subtree $left[x]$, the sum of all elements in the subtree rooted at $right[x]$ and x. Thus $x_{sum} = x + left[x]_{sum} + right[x]_{sum}$. Indeed the algorithm computes the right value for every internal node.

By induction the algorithm computes the right value for all nodes in a binary tree with $n \geq 1$ nodes.

Question 6: [5 points] You have downloaded your grades for all the n homework assignments. Each homework assignment has an integer grade g_i, where $0 \leq g_i \leq 4n$. Describe an $O(n)$ time algorithm that detects if there are at least two homework assignments where you received the same grade. You get as input the unsorted array A containing the integer grades for all n homework assignments. Don't forget to analyze the running time and prove the correctness of your algorithm.

Solution:
Algorithm:
First run CountingSort on the grades to get a sorted list. Then iterate over all items and compare all adjacent pairs of items. If two adjacent items are equal then return TRUE. After iterating over the entire array return FALSE.

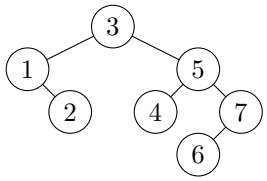
Runtime:
CountingSort runs $O(n+k)$ time. As we have that $k = 4n$, in this case CountingSort runs in $O(n+4n) = O(n)$ time. Iterating over the array takes $n - 1 \cdot O(1) = O(n)$ time. Returning the answer takes $O(1)$ time. In total we use $O(n)$ time.

Proof of Correctness:
In a sorted array two equal values need to be adjacent. Thus it is sufficient to check all adjacent pairs of grades to see if there is a duplicate grade.

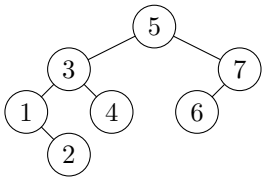
Question 7:

(a) [2 points] Consider binary search trees with the seven keys $1, 2, \ldots, 7$. Draw two distinct binary search tree with these seven keys that have exactly the nodes with keys 2, 4, and 6 as leaves. All other nodes *must* be internal.

Solution:



Solution:



(b) [2 points] Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf.

Solution: Let p be the length of a shortest path from x to a descendant leaf, and let q be the length of a longest path from x to a descendant leaf. By definition $p \geq bh(x)$, where $bh(x)$ denotes the black height of x. The longest path also contains exactly $bh(x)$ black nodes. Moreover, as every red node on the path must have a black child there are at most as many red nodes as black nodes on the longest path. Thus the longest path has length at most $2bh(x)$. Hence $q \leq 2bh(x) \leq 2p$.

Question 8: We want to augment a red-black tree T. For each of the following examples determine if this additional information can be maintained under insertions and deletions without affecting the $O(\log n)$ running time. Explain your answers.

You can use theorems from the book and from the lecture without proving them. You do, however, have to apply them correctly!

(a) [2 points] Every node v is augmented with a field f that stores the largest key in the subtree rooted at v.

Solution: Yes, this is possible.
Motivation: As we have $v.f = \max\{left[v].f, right[v].f, v\}$ for an internal node and $v.f = -\infty$ for a NIL leaf by RB-tree augmentation theorem this can be maintained in $O(\log n)$ as the value depends only on the node itself and it's children.

(b) [2 points] Every node v is augmented with a field f that stores the depth of the node in the tree.

Solution: No, this is not possible
Motivation: If we repeatedly insert a new smallest value then at some point a rotation on the root will occur. When we perform a right-rotation on the root r, then the depth of all nodes in the right subtree will change. The subtree rooted at $right[r]$ has black-height $\Theta(\log n)$ and thus the subtree contains at least $2^{\Theta(\log n)} \geq 2^{c \log n} = (2^{\log n})^c = n^c$ nodes. But then the depth of $\Omega(n^c)$ nodes changes, and updating this takes $\Omega(n^c)$ time which is already more time than $O(\log n)$.

(c) [2 points] Every node v already stores an additional field *shape* (either *square* or *round*). Each node v is now augmented with a field f that stores the sum of all keys of *square* nodes in the subtree rooted at v.

Solution: Yes, this is possible.
Motivation: $v.f = left[v].f + right[v].f + v.key$ if v is *square* and $v.f = left[v].f + right[v].f$ is v is *round* and $v.f = 0$ for a NIL-leaf. By RB-tree augmentation theorem this can be maintained in $O(\log n)$ as the value depends only on the node ifself and it's children.

Question 9:

(a) [3 points] Use the hash function $h(k,i) = (\,h'(k) + 2i^2 + 2i\,) \bmod 11$, with $h'(k) = k \bmod 17$ to insert the numbers 23, 43, and 47 in this order into this hash table using open addressing:

Solution:

0	1	2	3	4	5	6	7	8	9	10
43		36	20	**47**		19			9	**23**

Motivation (not needed on exam):
$h(k,i) = h(23,0) = ((23 \mod 17) + 2*0^2 + 2*0) \mod 11 = 6$
$h(k,i) = h(23,1) = ((23 \mod 17) + 2*1^2 + 2*1) \mod 11 = 10$
$h(k,i) = h(43,0) = ((43 \mod 17) + 2*0^2 + 2*0) \mod 11 = 9$
$h(k,i) = h(43,1) = ((43 \mod 17) + 2*1^2 + 2*1) \mod 11 = 2$
$h(k,i) = h(43,2) = ((43 \mod 17) + 2*2^2 + 2*2) \mod 11 = 10$
$h(k,i) = h(43,3) = ((43 \mod 17) + 2*3^2 + 2*3) \mod 11 = 0$
$h(k,i) = h(47,0) = ((47 \mod 17) + 2*0^2 + 2*0) \mod 11 = 2$
$h(k,i) = h(47,1) = ((47 \mod 17) + 2*1^2 + 2*1) \mod 11 = 6$
$h(k,i) = h(47,2) = ((47 \mod 17) + 2*2^2 + 2*2) \mod 11 = 3$
$h(k,i) = h(47,3) = ((47 \mod 17) + 2*3^2 + 2*3) \mod 11 = 4$

(b) [2 points] Let $T[0..m-1]$ be a hash table in which we want to store a set of keys. We decide to resolve collisions with open addressing using linear probing. We have $T[j] =$ NIL if no key is stored at position j. Assume that key k is stored at position j, that is, $T[j] = k$. We want to delete k from the hash table. Explain why we cannot simply set $T[j]$ to be NIL.

Solution: Suppose that while key k was stored in the hash table, some other value m during insertion probed $T[j]$ and found a non-empty spot. Then it would continue probing and be inserted somewhere else in the table. If we simply set $T[j] =$NIL then if we search for m we will find that $T[j]$ is empty in our probing sequence and conclude that m is not stored in the table. Thus we may not be able to find elements back from our hash-table.

(c) [2 points] What are the advantages of double hashing over linear or quadratic probing for open addressing?

Solution: If two values map to the same index, they don't necessarily follow the same probing sequence. Particularly in a hash-table with m slots double hashing ensures there can be up to $O(m^2)$ different hash-sequences instead of the $O(m)$ different sequences for linear and quadratic probing. That means that it is less likely that many probe sequences follow the same sequence of probes. This prevent primary and secondary clustering, which causes long sequences of slot to fill up and in turn slows down INSERT, DELETE

and SEARCH operations on the table.

Question 10: [5 points] Give an algorithm that takes as input a graph $G = (V, E)$, stored in an adjacency list, and that returns TRUE if there exists at least one cycle in G and FALSE otherwise. Your algorithm should run in $O(V)$ time. Don't forget to analyze the running time and prove the correctness of your algorithm.

Solution:
Algorithm:
Run DFS and stop as soon as you find a back-edge.

Runtime:
DFS normally runs in $O(V + E)$ time. However, if the graph contains no cycles then it must either be a undirected, connected, acyclic graph (a tree), or a set of trees. A set of $k \geq 1$ trees on $|V|$ vertices has $\sum_{i=1}^{k} |V_i| - 1 = |V| - k$ edges. Thus the runtime for a graph without cycles is $O(V + V - k) = O(V)$.

If the graph does contain a cycle, then at the latest at the $|V| - k + 1$-st edge traversed we must complete a cycle. This edge is a back-edge (see PoC). Thus after $|V| - k + 1$ edges we will detect the graph has a cycle and we stop DFS. Thus the runtime for a graph with cycles is $O(V + V - k + 1) = O(V)$ as well.

Proof of Correctness:
DFS can only find tree edges or back edges. When only WHITE nodes are discovered when traversing the edges then only tree edges have been explored. It must then be that the graph is acyclic and we can safely return that there are no cycles.

When DFS find a back edge then there is a way to reach an ancestor w of the currently explored node v. But then there is a path from the ancestor w to v along GRAY nodes. Moreover there is a path from v to w along the edge vw. Thus there exists a cycle and therefore the graph is not acyclic and we can safely return that there is a cycle.