Delft University of Technology
EEMCS Faculty

CSE1305 Algorithms & Data Structures

Final Written Exam

29 January 2020, 13:30–15:30

Examiners:

Examiner responsible: Joana Gonçalves and Robbert Krebbers
Examination reviewer: Stefan Hugtenburg

Parts of the examination and determination of the grade:

Exam part	Number of questions	Question specifics	Grade (%)	Grade (points)
Multiple-choice	22 questions (equal weights)	One correct answer per question	50%	$5 \cdot \frac{\text{score}}{22}$
Open questions	3 questions (different weights)	Multiple parts	50%	$5 \cdot \frac{\text{score}}{30}$

Use of information sources and aids:

- A hand-written double-sided A4 cheat sheet can be used during the exam.
- No other materials may be used, including but not limited to books, lecture slides in any form, or devices such as laptops and phones.
- Scrap paper sheets are provided at the beginning of the exam. Additional scrap paper can be requested.

General instructions:

- Solve the exam on your own. Any form of collaboration is prohibited.
- You cannot leave the examination room during the first 30 minutes.
- If you are eligible for extra time, place the "Verklaring Tentamentijd Verlenging" on your desk.

Instructions for writing down your answers:

- You should answer the questions using the provided answer sheets.
- Write your name and student number on every sheet of paper.
- **For multiple-choice questions**, mark the answers on this exam paper first, and copy them to the answer form after revising.
- **For open questions**, provide all requested information and always give an explanation. Avoid irrelevant data, it could lead to deductions.
- **For proofs**, make sure your proof is properly structured and sufficiently explained. Statements or steps without justification could lead to point deductions.

Multiple-choice questions (50%, 22 points)

1. (1 point) Which of the following statements about asymptotic algorithmic complexities is **false**?

   A. n is $\Omega(1)$

   B. n is $\Omega(n \log n)$

   C. 2^{n+1000} is $\mathcal{O}(2^n)$

   D. $\log n$ is $\mathcal{O}(n \log n)$

Answer:

A. True. The function n is larger than a constant when n grows arbitrarily large.

B. False. The function $n \log n$ grows faster than n when n grows arbitrarily large.

C. True. $2^{n+1000} = 2^{1000} \cdot 2^n$. Since 2^{1000} is a constant, the big-Oh is $\mathcal{O}(2^n)$.

D. True. The function $n \log n$ is an asymptotic upper bound on $\log n$, i.e. grows faster when n grows arbitrarily large.

2. (1 point) Which of the following statements about asymptotic algorithmic complexities is **false**?

   A. $\lceil f(n) \rceil$ is $\mathcal{O}(f(n))$, if $f(n)$ is a positive non-decreasing function that is always greater than 1.

   B. If $d(n)$ is $\mathcal{O}(f(n))$, then $a \cdot d(n)$ is $\mathcal{O}(f(n))$ for any constant $a > 0$.

   C. If $p(n)$ is a polynomial in n, then the tightest upper bound for $\log(p(n))$ is $O(n \log n)$.

   D. $h(n)$ is $\mathcal{O}(\max\{f(n), g(n)\})$ if and only if $h(n)$ is $\mathcal{O}(f(n) + g(n))$.

Answer:

A. True. If $f(n)$ is a positive nondecreasing function greater than 1, then $\lceil f(n) \rceil \leq 2f(n)$.

B. True. There are constants c and n_0 such that $d(n) \leq c \cdot f(n)$ for $n \geq n_0$. Thus, $a \cdot d(n) \leq a \cdot c \cdot f(n)$ for $n \geq n_0$.

C. False. The tightest upper bound is $\mathcal{O}(\log n)$. Recall that $\log n^k = k \log n$.

D. True.

- For $\mathcal{O}(\max\{f(n), g(n)\})$ is $\mathcal{O}(f(n) + g(n))$:
  Since $\max\{f(n), g(n)\}$ is either $f(n)$ or $g(n)$, and $f(n) \leq f(n) + g(n)$ and $g(n) \leq f(n) + g(n)$ (with $f(n) > 0$ and $g(n) > 0$), then $\max\{f(n), g(n)\} \leq f(n) + g(n)$.
- For $\mathcal{O}(f(n) + g(n))$ is $\mathcal{O}(\max\{f(n), g(n)\})$:
  Since $f(n) \leq \max\{f(n), g(n)\}$ and $g(n) \leq \max\{f(n), g(n)\}$, then $f(n) + g(n) \leq 2 \cdot \max\{f(n), g(n)\}$.
- Since $\mathcal{O}(\max\{f(n), g(n)\})$ is $\mathcal{O}(f(n)+g(n))$ and $\mathcal{O}(f(n)+g(n))$ is $\mathcal{O}(\max\{f(n), g(n)\})$, it follows that $\mathcal{O}(\max\{f(n), g(n)\}) = \mathcal{O}(f(n) + g(n))$.

3. (1 point) Consider the Java interface `AnInterface` below, which defines an abstract data type (ADT) for one of the data structures studied in this course.

```
1  public interface AnInterface<E> {
2    public int size();
3    public boolean isEmpty();
4    public E first();
5    public E last();
```

```
6    public void addFirst(E e);
7    public void addLast(E e);
8    public void removeFirst(E e);
9  }
```

Which of the following data structures **cannot** be efficiently implemented by only reusing methods from the data structure that implements this interface `AnInterface` (no additional methods allowed)? Note that the data structure implements every method in this interface with $\mathcal{O}(1)$ time complexity.

A. Priority queue.

B. Queue.

C. Stack.

D. Circularly-linked list.

Answer: Note that the given interface corresponds to the ADT of a singly-linked list (SLL).

A. A priority queue cannot be implemented efficiently by only reusing this interface, since it does not allow direct access, insertion, or removal at arbitrary positions (necessary to insert an element when using a sorted list implementation, or to retrieve the element with minimal/maximal key when using an unsorted list implementation).

B. A queue can be efficiently implemented by reusing the methods of this interface. For the non-trivial methods, `enqueue` uses `addLast` and `dequeue` uses `removeFirst`. See Section 6.2 "Queues" of the book for the full implementation.

C. A stack can be efficiently implemented by reusing the methods of this interface. For the non-trivial methods, `top` uses `first`, `push` uses `addFirst`, and `pop` uses `removeFirst`. See Section 6.1 "Stacks" of the book for the full implementation.

D. A circularly-linked list (CLL) can be efficiently implemented by reusing the methods of this interface. The only additional operation offered by a CLL relative to a SLL is `rotate`, which can be implemented using calls to `removeFirst` and `addLast`. The implementation is not as efficient as the one of an actual CLL, but it does have the same asymptotic time and space complexities.

4. (1 point) Consider the most time-efficient algorithm that uses $\mathcal{O}(1)$ space to calculate and print the sum of every pair of consecutive elements in a sequence S with n elements. Example: if $S = [1, 2, 3, 4, 5]$, the algorithm should print the sums $3, 5, 7, 9$ corresponding to $1 + 2$, $2 + 3$, $3 + 4$, and $4 + 5$. The sequence can change in between, but its final state needs to be the same as it was before the calculations. What is the tightest worst-case **time** complexity of such algorithm when S is implemented using either a fixed-size array, or a singly-linked list without a tail reference. The list has the following ADT: `addFirst(E e)`, `addLast(E e)`, `removeFirst()`, `first()`, `size()`. Note that the ADT **does not allow** direct access to its nodes, the methods `removeFirst` and `first` return the element directly.

A. array: $\mathcal{O}(n)$ **singly-linked list:** $\mathcal{O}(n)$

B. array: $\mathcal{O}(n)$ **singly-linked list:** $\mathcal{O}(n^2)$

C. **array:** $\mathcal{O}(n^2)$ **singly-linked list:** $\mathcal{O}(n)$

D. **array:** $\mathcal{O}(n^2)$ **singly-linked list:** $\mathcal{O}(n^2)$

Answer:

- Array: The algorithm sums pairs of consecutive elements by accessing their indices directly. This runs in $\mathcal{O}(n)$ time. Sample code:

```
1  // create example list
2  int[] array = new int[]{1,2,3,4,5};
3
4  // sum and print
5  for (int i = 1; i < array.length; i++)
6    System.out.print(array[i-1]+array[i]);
```

- SLL in $\mathcal{O}(n)$: One can make sure no more space is required by first repeatedly removing the first element from the list and adding this as the first element to a new list. This means the original list shrinks as the new list grows, meaning no extra space is required. This results in a list in reverse order, so we need to do this a second time to get the list back to its original order. Sample code:

```
1  // create example list
2  LinkedList<Integer> list = new LinkedList<Integer>();
3  for (int i = 1; i < 6; i++)
4    list.addLast(i);
5
6  LinkedList<Integer> temp = new LinkedList<Integer>();
7  // sum and print
8  for (int i = list.size(); i > 1; i--) {
9    int first = list.removeFirst();
10   System.out.print(first+list.getFirst());
11   temp.addFirst(first);
12 }
13 temp.addFirst(list.removeFirst());
14 // reverse temp back to original list.
15 for (int i = list.size(); i > 0; i--) {
16         list.addFirst(temp.removeFirst());
17 }
```

  Since this solution is slightly more involved than intended, we have also accepted answer B as correct.

- SLL in $\mathcal{O}(n^2)$: The algorithm sums pairs of consecutive elements by removing the first element first using `removeFirst`, so that it can also access the second (using `first`. The first element is added again to the end of the list using `addLast`, so that the list will be in the same state after all elements have been processed. The algorithm repeats these operations for every position in the list, so $\mathcal{O}(n)$ times. Since `removeFirst` and `first` are $\mathcal{O}(1)$, and `addLast` is $\mathcal{O}(n)$ (no tail reference), the overall runtime is $\mathcal{O}(n^2)$. Sample code:

```
1  // create example list
2  LinkedList<Integer> list = new LinkedList<Integer>();
3  for (int i = 1; i < 6; i++)
4    list.addLast(i);
5
6  // sum and print
7  for (int i = list.size(); i > 1; i--) {
8    int first = list.removeFirst();
9    System.out.print(first+list.getFirst());
10   list.addLast(first);
11 }
12 list.addLast(list.removeFirst());
```

  Note that the method `addLast` of Java's `LinkedList` class is $\mathcal{O}(1)$ time. Here we consider an implementation of `LinkedList` with $\mathcal{O}(n)$ time complexity for `addLast`.

5. (1 point) Consider the following Java method `calculate`.

```
1  public long calculate(int n) {
2    if (n <= 1)
3      return n;
4    return calculate(n-2) + calculate(n-1);
5  }
```

What growth rate corresponds to the tightest big-Oh **time** complexity of method `calculate` as a function of the input size n? Tip: look at the number of recursive calls made for different values of n.

A. Logarithmic.

B. Linear.

C. Quadratic.

D. Exponential.

Answer: Method `calculate` makes at least $2^{n/2}$ recursive calls to compute the n^{th} Fibonacci number, since it recomputes intermediate Fibonacci numbers multiple times instead of doing it once and reusing the result. See the example in the book for more details (Section 5.5 "Pitfalls of Recursion").
Define number of calls to calculate n^{th} Fibonacci number as c_n:
$c_0 = 1$
$c_1 = 1$
$c_2 = 1 + c_0 + c_1 = 1 + 1 + 1 = 3$
$c_3 = 1 + c_1 + c_2 = 1 + 1 + 3 = 5$
$c_4 = 1 + c_2 + c_3 = 1 + 3 + 5 = 9$
$c_5 = 1 + c_3 + c_4 = 1 + 5 + 9 = 15$
$c_6 = 1 + c_3 + c_4 = 1 + 9 + 15 = 25$
$c_7 = 1 + c_3 + c_4 = 1 + 15 + 25 = 41$
$c_8 = 1 + c_4 + c_5 = 1 + 25 + 41 = 67$

6. (1 point) Consider again the method `calculate` from question 5, as well as the method `calculate2` provided below. Which of the following statements is **true** about methods `calculate` and/or `calculate2`?

```
1  public long[] calculate2(int n) {
2    if (n <= 1) {
3      long[] res = {n, 0};
4      return res;
5    }
6    long[] temp = calculate2(n-1);
7    long[] res = {temp[0]+temp[1], temp[0]};
8    return res;
9  }
```

A. Method `calculate` uses tail recursion, while method `calculate2` does not.

B. Method `calculate2` runs in $\mathcal{O}(n)$ time.

C. Method `calculate2` uses more stack frames than method `calculate`.

D. Both methods `calculate` and `calculate2` use binary recursion.

Answer:

A. False. None of them is tail recursive, since both perform calculations after returning from the recursive calls.

B. True. Each recursive call to method `calculate2` decreases the argument n by 1, therefore the recursive trace includes a sequence of n method calls. Since the non-recursive work per call takes $\mathcal{O}(1)$ time, the overall runtime is $\mathcal{O}(n)$.

C. False. Method `calculate2` uses less stack frames, because it makes $O(n)$ recursive calls, while method `calculate` makes $O(2^n)$ recursive calls.

D. False. Method `calculate` uses binary recursion because it makes 2 recursive calls (line 5), but method `calculate2` uses linear recursion (line 6).

7. (1 point) Which of the following average case time complexities does not involve probability or probabilistic analysis?

A. $\mathcal{O}(1)$ for getting an entry from a hashtable.

B. $\mathcal{O}(1)$ for adding an element at the end of an array list implemented using a dynamic array with proportional resizing.

C. $\mathcal{O}(n)$ for randomized quick-select.

D. $\mathcal{O}(n \log n)$ for randomized quicksort.

Answer:

A. The average case for getting an entry from a hashtable has a probabilistic basis: if the hash function is good, we expect that entries are uniformly distributed in the N cells of the bucket array. Thus, to store n entries, the expected number of keys in a bucket would be $\lceil n/N \rceil$, which is $\mathcal{O}(1)$ if n is $O(N)$.

B. This complexity is the result of amortized complexity analysis, which does not involve probability. The cost of adding an element at the end of an array list using a dynamic array is deterministic: we know that it corresponds to the cost $\mathcal{O}(1)$ of adding one element (and occasionally plus the cost $\mathcal{O}(i)$ of increasing the array size if the array is full). The average time complexity is obtained by aggregating the cost over many insertion operations, such that the cost of the expensive but occasional resizing operations gets diluted over the insertions.

C. The introduction of randomization in the pivot choice implies that there is a high probability of choosing good partitions, which in turn leads to an expected time complexity of $\mathcal{O}(n)$.

D. Same reasoning as above, but leading to an expected time complexity of $\mathcal{O}(n \log n)$. See book Section "13.2.1 Randomized quicksort" for a detailed analysis.

8. (1 point) Which data structures are used for the following tasks: expression parsing (matching parentheses), and breadth-first search?

A. **parsing:** stack **breadth-first:** stack

B. parsing: stack breadth-first: queue

C. **parsing:** queue **breadth-first:** stack

D. **parsing:** queue **breadth-first:** queue

Answer: For matching parentheses we use a stack, since each new right parenthesis encountered needs to be matched to the last or most recently encountered left parenthesis (therefore, we need LIFO behaviour).
For breadth-first search we use a queue, since we need to explore nodes/vertices in the order they were first encountered (therefore, we need FIFO behaviour).

9. (1 point) Which of the following statements presents the implementations of a priority queue from most to least worst-case time-efficient (left to right) for an **insertion** operation?

   A. Sorted list, unsorted list, heap.

   B. Heap, sorted list, unsorted list.

   C. Unsorted list, sorted list, heap.

   D. Unsorted list, heap, sorted list.

> **Answer:** Sorted list insertion is $\mathcal{O}(n)$, since the element needs to be inserted at its final position. Unsorted list insertion is $\mathcal{O}(1)$, since order does not matter, so insertion occurs where it is most efficient. Binary heap insertion is $\mathcal{O}(\log_2 n)$, bounded by the height of the tree).

10. (1 point) Consider the `clone` method below for cloning objects of class `SinglyLinkedList`.

```
 1  public SinglyLinkedList<E> clone() throws CloneNotSupportedException {
 2    SinglyLinkedList<E> other = (SinglyLinkedList<E>) super.clone();
 3    if (size > 0) {
 4      other.head = new Node<>(head.getElement(), null);
 5      Node<E> walk = head.getNext();
 6      Node<E> otherTail = other.head;
 7      while (walk != null) {
 8        Node<E> newest = new Node<>(walk.getElement(), null);
 9        otherTail.setNext(newest);
10        otherTail = newest;
11        walk = walk.getNext();
12      }
13    }
14    return other;
15  }
```

Which of the following statements is **true** about method `clone`?

   A. If a **node** of the clone list is replaced, the corresponding node in the original list will also change.

   B. If an **element** of the clone list is replaced, the corresponding element in the original list will also change.

   C. If an instance field of an element in the clone list is changed, then the field of the corresponding element in the original list will also change.

   D. If the **head** of the clone list is changed to point to the tail of the clone list, then the head of the original list will also point to the tail of the original list.

> **Answer:** The nodes of the cloned list are new, while the elements are the same. This means that replacing the nodes or the elements of the cloned list does not have an effect on the original list (note that the element objects are the same, but the references of the nodes to the element objects are independent), while changing instance fields of an element will change in the original element as well.

11. (1 point) Consider an array-based implementation of a binary tree. For which kind of binary tree structure does this implementation use the least **space** (in absolute terms, not asymptotic)?

   A. Proper or full binary tree.

   B. Complete binary tree.

   C. Balanced binary tree.

   D. Degenerate binary tree, in which every internal node has only one child.

Answer: In an array-based representation of a binary tree, the elements of the tree are stored in an array-based list such that the element at position p is found at index equal to the level number $f(p)$ of position p, defined as follows:

- If p is the root, then $f(p) = 0$.
- If p is the left child of position q, then $f(p) = 2f(q) + 1$.
- If p is the right child of position q, then $f(p) = 2f(q) + 2$.

From the definition of each of the binary tree structures in the different options (see below), one can conclude that all structures except the complete binary tree can lead to an array representation with empty spaces in between, that is, indexes in the array for which there are no nodes in the binary tree. This means that they may need to use more space than the number of nodes. The only tree for which this does not happen is the complete binary tree (a heap is an example of a complete binary tree).

A. A proper binary tree is a binary tree in which every internal node has either 0 or 2 children.

B. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.

C. A balanced binary tree is a binary tree in which the heights of the left and right subtrees differ by at most one, the left subtree is balanced, and the right subtree is balanced.

D. A degenerate binary tree, as defined, is a tree in which every internal node has only one child.

12. (1 point) Which of the statements about tree traversals is **false**?

A. It is possible for the postorder and preorder traversal to visit the nodes of a tree with more than one node in the same order.

B. It is possible for the postorder and preorder traversals to visit the nodes of a tree with more than one node in **reverse** order.

C. It is not possible for the postorder and preorder traversals to visit the nodes of a **proper** binary tree in **reverse** order.

D. The inorder traversal is only defined for binary trees.

Answer:

A. False. It is not possible for the postorder and preorder traversal of a tree with more than one node to visit the nodes in the same order. A preorder traversal will always visit the root node first, while a postorder traversal node will always visit an external node first.

B. True. It is possible for a preorder and a postorder traversal to visit the nodes in the reverse order. Consider the case of a tree with only two nodes.

C. True. It is not possible for the postorder and preorder traversals to visit the nodes of a proper binary tree in the reverse order. Let r be the root of a proper binary tree and let T_1 and T_2 be the left and right subtrees. A postorder traversal would visit the postorder traversal of T_1, the postorder traversal of T_2 and then node r while the preorder traversal would visit node r, the preorder traversal of T_1 and then the preorder traversal of T_2. Clearly the postorder and preorder traversals cannot be the reverse of each other since in both cases, all the nodes of T_1 are visited before all the nodes of T_2.

D. True.

13. (1 point) Consider the sorting of sequence $[4, 7, 12, 9, 1, 3]$ in increasing order with the in-place heap sort algorithm using an array-based heap. What is the content of the array after the first two removals from the heap (including any necessary bubbling operations)?

    A. $[1, 7, 4, 3, 9, 12]$

    B. $[7, 1, 4, 3, 9, 12]$

    C. $[7, 3, 4, 1, 9, 12]$

    D. $[1, 3, 4, 7, 12, 9]$

Answer: Original sequence $[4, 7, 12, 9, 1, 13]$
After building max-heap: $[12, 9, 4, 7, 1, 3]$
After removal of max (root swaps with last position): $[3, 9, 4, 7, 1, 12]$ (12 swaps with 3)
After down-heap on root (3): $[9, 7, 4, 3, 1, 12]$ (first swap with 9, then swap with 7)
After removal of max: $[1, 7, 4, 3, 9, 12]$ (9 swaps with 1)
After down-heap on root (1): $[7, 3, 4, 1, 9, 12]$ (first swap with 7, then swap with 3)

14. (1 point) Consider the following method `csort`, which implements a sorting algorithm. The input array is guaranteed to contain integer values between 0 and k.

```
 1  public static void csort(int[] array, int k) {
 2    int temp[] = new int[k + 1];
 3
 4    for (int e : array)
 5      temp[e]++;
 6
 7    int ndx = 0;
 8    for (int i = 0; i < temp.length; i++)
 9      while (temp[i] > 0) {
10        array[ndx] = i;
11        ndx++;
12        temp[i]--;
13     }
14  }
```

Which of the following statements about method `csort` and the sorting algorithm it implements is **true**?

A. It is a comparison-based sorting algorithm.

B. It is an in-place sorting algorithm with $\mathcal{O}(1)$ space complexity.

C. It runs in $\mathcal{O}(n)$ time, where n is the length of the input array.

D. At each index, the array `temp` accumulates the number of elements from the original input array that are identical to such index.

Answer:

A. False. It does not perform comparisons, it uses elements as indices of an auxiliary array $temp$. The element at each index accumulates the number of elements from the input array which are identical to such index.

B. False. The algorithm declares a new array with k positions, so it uses $\mathcal{O}(k)$ space.

C. False. The algorithm runs in $\mathcal{O}(n + k)$ time. This is only $O(n)$ if k is $\mathcal{O}(n)$.

D. True.

15. (1 point) Consider again the method given in question 14. Between method `csort` and the algorithm merge sort, which one has the fastest running time in each of the following two situations? First, sorting 1 million bytes. Second, sorting 100 integers in the range between 0 and 1 million.

    A. first: `merge sort`; second: `merge sort`

    B. first: `merge sort`; second: `csort`

    C. first: `csort`; second: `csort`

    D. first: `csort`**; second:** `merge sort`

> **Answer:** Merge sort has time complexity $\mathcal{O}(n \log n)$, while `csort` runs in $\mathcal{O}(n + k)$ time. In the first situation, $n = 10^6$ and $k = 256$, so `csort` runs in $\mathcal{O}(n)$ time. In the second situation, $n = 100$ and $k = 10^6$; since $k = n^3$, the runtime of `csort` is $\mathcal{O}(n^3)$. Merge sort remains $\mathcal{O}(n \log n)$.

16. (1 point) Which of the following statements about sorting and selection algorithms is **true**?

    A. Heap sort, merge sort, and bucket sort are naturally stable sorting algorithms.

    B. The time complexity of insertion sort is $\mathcal{O}(n + m)$, where n is the size of the input sequence and m is the number of inversions. In the worst-case, this can be $\mathcal{O}(n^2)$.

    C. LSD radix sort may not need to process all the individual keys of a composite key in order to determine the final ordering.

    D. The quick-select, merge sort, and quicksort algorithms follow the divide-and-conquer paradigm.

> **Answer:**
>
> A. False. Merge and bucket sort are stable. Heap sort is not, since the order of elements with the same key cannot be guaranteed due to the bubbling operations.
>
> **B. True.**
>
> C. False. This is true for MSD, not LSD, since MSD processes keys from the most to the least significant position. See example given in the lectures.
>
> D. Merge and quick sort use divide-and-conquer, since they both split the input into smaller sequences and recursively sort those sequences, after which they combine the results into the result for the larger input sequence. Quick-select uses decrease-and-conquer or prune-and-search, since it prunes away a portion of the input and then recursively solves the smaller problem.

17. (1 point) Which of the statements about hashing is **true**?

A. If `a.hashCode() == b.hashCode()` is true, then `b.equals(a)` should be true.

B. If `b.equals(a)` is true, then `a.hashCode() == b.hashCode()` should be true.

C. When only using the `get` and `put` operations, hash maps are always better than AVL-trees.

D. When using separate chaining, the load factor can be at most 1.

Answer:

A. Incorrect. That would limit us to `Integer.MAX_VALUE` different entries.

B. Correct. Two objects that are the same should also hash the same.

C. Incorrect. For example, periodic rehashing can be problematic in real-time systems.

D. Incorrect. There can be arbitrarily many entries in the secondary container.

18. (1 point) Consider the following fixed size hash table using linear probing (associated values are omitted):

0	1	2	3	4	5	6	7	8	9
9		11					36	16	18

The hash function is $h(k) = (k + 1) \mod 10$. Assume we first insert an entry with key 27, then delete the entry with key 16, and then insert an entry with key 6. In which bucket will the inserted entry appear?

A. 1

B. 3

C. 6

D. 8

Answer: After inserting an entry with key 27 (hash code 8), the hash table is:

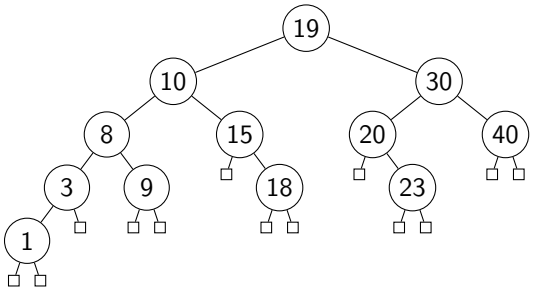
0	1	2	3	4	5	6	7	8	9
9	27	11					36	16	18

After removing the entry with key 16 (hash code 7), the hash table is:

0	1	2	3	4	5	6	7	8	9
9	27	11					36	×	18

Here, × represents a DEFUNCT bucket. When inserting an entry with key 6 (hash code 7), it will probe until it finds an empty or DEFUNCT bucket, which will appear at position 8.

19. (1 point) Consider the following AVL tree:



When deleting **40** from the given AVL tree, how many tri-node restructurings are caused in the tree? If the node that is to be deleted has two non-null children, the node will be replaced with the in-order predecessor (i.e. the maximal node in the left child).

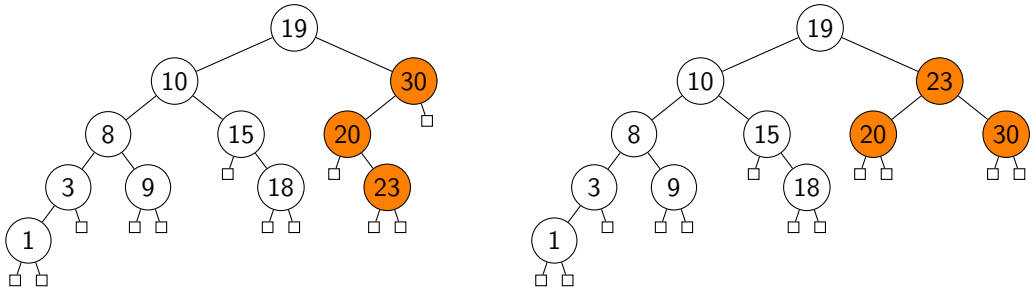
A. None.

B. One tri-node restructuring.

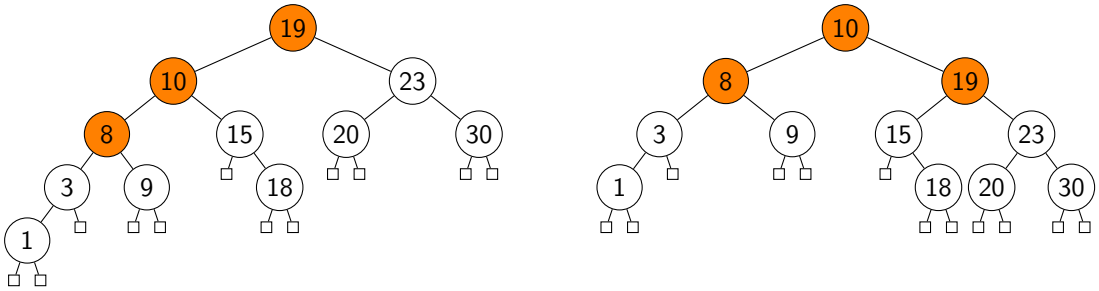
C. Two tri-node restructurings.

D. Three tri-node restructurings.

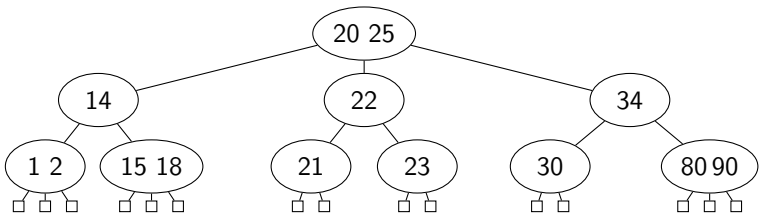
Answer: After removing 30, we obtain a tree that needs to be fixed by a double rotation, i.e. one tri-node restructuring:



Subsequently, the root needs to be fixed by a single rotation, i.e. another tri-node restructuring:



20. (1 point) Consider the following (2,4) tree:



When deleting **34** from the tree, how many underflows are caused in the tree? If the node of the entry that is to be deleted has has non-null children, the node will be replaced with the in-order predecessor (i.e. the maximal entry in the left child of the entry).

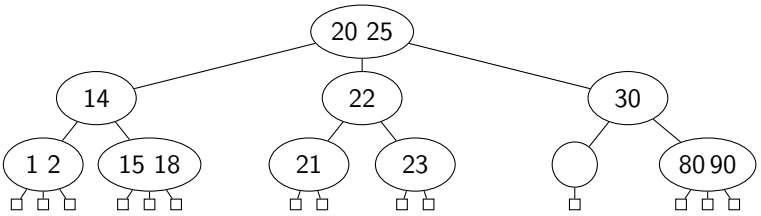
A. One underflow, which needs to be fixed by a fusion.

B. One underflow, which needs to be fixed by a tranfer.

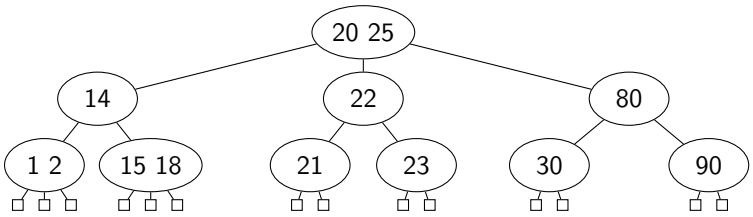
C. Two underflows, which need to be fixed by a transfer and fusion.

D. Two underflows, which need to be fixed by two transfers.

Answer: After removing 34, we end up with:



After performing a transfer, we end up with:



There are no more underflows in this tree, so we are done.

21. (1 point) Consider Kruskal's algorithm operating on a graph $G = (V, E)$ where $n = |V|$ is the number of vertices and $m = |E|$ is the number of edges. Which of the following statements is **true**?

A. One needs an adaptable priority queue to implement the algorithm.

B. The edge with the highest weight in E is never added to the tree by the algorithm.

C. The `union` operation on partitions is called $\mathcal{O}(\min(n, m))$ times.

D. Kruskal's algorithm starts with n clusters, and terminates when there is 1 cluster left.

Answer:

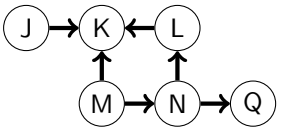
A. Incorrect. To implement Kruskal one only needs the `insert` and `removeMin` operations, not the `replaceKey` operation.

B. Incorrect. Consider a graph E that is already a tree, in this case Kruskal removes no edges.

C. Correct. Each iteration of the algorithm calls the `union` operation at most once. The algorithm terminates after either n iterations, or in case the priority queue is empty. The priority queue initially contains m elements.

D. Incorrect. Consider an unconnected graph. In the lecture we have however only applied Kruskal on connected graphs, so we have decided to consider this answer correct when grading the exam.

22. (1 point) Consider the following directed acyclic graph (DAG):



In general, DAGs can have multiple topological orders. How many different topological orders does the above graph have?

A. 11

B. 14

C. 17

D. 20

Answer: In order to enumerate all topological orders, we start with the sequence M, N, L, K, which has to appear in order. We then insert the vertex Q into the sequence. The vertex Q should appear after N, so there are 3 choices for inserting Q. We then insert the vertex J into the sequence. The vertex J should appear before K, so depending on where Q has been inserted there are 4 or 5 choices. In case Q has been inserted before K, there are 5 choices. In case Q has been inserted after K, there are only 4 choices. This gives 14 choices in total.

Open questions (50%, 30 points)

23. Consider the following Java implementation of method `methodX`.

```
1  public static int methodX(Graph<Integer,Integer> g) {
2    int x = 0;
3    Iterable<Vertex<Integer>> vertices = g.vertices();
4    for (Vertex<Integer> v : vertices) {
5      Iterable<Edge<Integer>> edges = g.outgoingEdges(v);
6      for (Edge<Integer> e : edges) {
7        x += e.getElement();
8      }
9    }
10   return x;
11 }
```

(a) (5 points) Assume that graph g is implemented using an edge list data structure, and does not have parallel edges or self-loops. Give the polynomial expressing the tightest worst-case **time** complexity of `methodX` as a function of the number of vertices n, number of edges m, and degree $\deg(v)$ of a vertex $v \in V$ of graph g. Define all variables and constants, and explain which lines of code contribute to each term of the polynomial.

Answer:

$$T(n) = c_0 + c_1 n + c_2 nm + c_3 \sum_{v \in V} \deg(v)$$

where:

- c_0 accounts for the primitive instructions associated with calling the method `methodX` (line 1), assignment of variables `x` (line 2) and `vertices` (line 3), and returning from the method `methodX` (lines 10);
- $c_1 n$ accounts for the call to `g.vertices` (line 5) and the constant-time operations within the first `for` loop (line 4-9) and, e.g. assignment of variable `edges` (line 5);
- $c_2 nm$ accounts for the call to `g.outgoingEdges` (line 5). We have $c_2 nm$, since the complexity of `g.outgoingEdges` is $O(m)$ on edge lists, and it appears within the first `for` loop.
- $c_3 \sum_{v \in V} \deg(v)$ accounts for the operations within the second `for` loop (line 7), e.g. the addition to variable `x`.

(b) (2 points) Use the polynomial in your answer to question 23(a) to derive the tightest worst-case **time** complexity in Big-Oh notation. You should explain, but you do not need to give a proof.

Answer: The polynomial from the answer to 23(a) can be simplified as follows:

$$\begin{aligned} T(n) &= c_0 + c_1 n + c_2 nm + c_3 \sum_{v \in V} \deg(v) \\ &\leq c_0 + c_1 n + c_2 nm + c_3 2m \end{aligned}$$

Here, we make use of $\sum_{v \in V} \deg(v) = 2m$, which holds because the graph does not have parallel edges or self loops.

The constants in the simplified polynomial can be disregarded, since $\{c_0, c_1, c_2, c_3\} \ll n, m$. The term nm grows faster than any other term in the polynomial when $n, m \to \infty$, therefore the time complexity of method `methodX` in Big-Oh notation is $\mathcal{O}(nm)$.

(c) (2 points) If the graph g was implemented using an adjacency list data structure, what would be the tightest worst-case **time** complexity of `methodX` in Big-Oh notation? You do not need to write down the polynomial nor give a proof, but you should motivate your answer.

Answer: When using an adjacency list, the worst-case time complexity of `outgoingEdges(`v`)` is $\mathcal{O}(\deg(v))$. Therefore, the worst-case time complexity of `methodX` is $\mathcal{O}(n + m)$.

24. Consider the following Java implementation of a recursive algorithm.

```
1 private static int methodY(int[] data, int i, int n) {
2   if (n == 0) return 0;
3   if (n == 1) return data[i];
4   return methodY(data, i, n/2) + methodY(data, i + n/2, n - n/2);
5 }
6
7 public static int methodZ(int[] data) {
8   return methodY(data, 0, data.length);
9 }
```

(a) (5 points) State the base and recurrence equations for the worst-case **space** complexity of the recursive method `methodY`. In your answer, clearly state what you consider to be the input size, and refer to the relevant parts of the code to justify why your equations are correct.

Answer:

$$S(1) = c_0$$
$$S(n) = c_1 + S(n/2) \quad \text{if } n > 1$$

where:

- n is the input size, expressed as the value of the parameter `n`;
- c_0 accounts for the stack frame in the base case;
- c_1 accounts for the stack frame in the recursive case;
- $S(n/2)$ accounts for the recursive calls (line 4).

(b) (6 points) Derive the closed form of your recurrence equation by repeatedly unfolding it.

Answer:

$$
\begin{aligned}
S(n) &= c_1 + S(n/2) && \text{by } S(n) = c_1 + S(n/2) \\
&= c_1 + (c_1 + \cdot S(n/4)) && \text{by } S(n/2) = c_1 + S(n/4) \\
&= 2c_1 + S(n/4) && \text{by arithmetic} \\
&= kc_1 + S(n/2^k) && \text{repeat } k \text{ times} \\
&= \log_2 n \cdot c_1 + S(1) && \text{by letting } k = \log_2 n \\
&= \log_2 n \cdot c_1 + c_0 && \text{by } S(1) = c_0
\end{aligned}
$$

(c) (2 points) State the tightest worst-case **space** complexity of method `methodY` in Big-Oh notation. You do not have to give a proof, but should clearly justify your answer.

Answer: Recall that the closed form is:

$$S(n) = \log_2 n \cdot c_1 + c_0$$

The constants can be disregarded, since $\{c_0, c_1\} \ll n$, therefore the time complexity of method `methodY` in Big-Oh notation is $\mathcal{O}(\log_2 n)$.

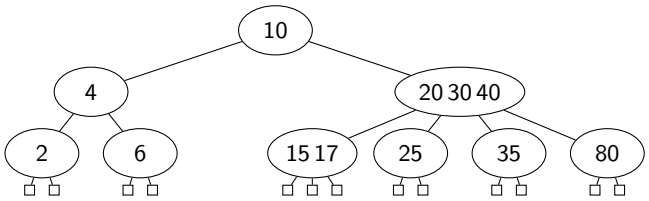
(d) (2 points) Describe an improved version of `methodY`, which performs the same calculation as `methodY` with better big-Oh worst-case **space** complexity and the same big-Oh worst-case **time** complexity. Give the big-Oh **space** complexity of your improved solution.

Answer: Method `methodY(data,i,n)` computes the sum of the first `n` elements starting at index `i` in array `data`. A more efficient version can be written using a loop:

```
1  private static int methodY(int[] data, int i, int n) {
2    int sum = 0;
3    for (int k = i; k < i + n; k++)
4      sum += data[k];
5    return sum;
6  }
```

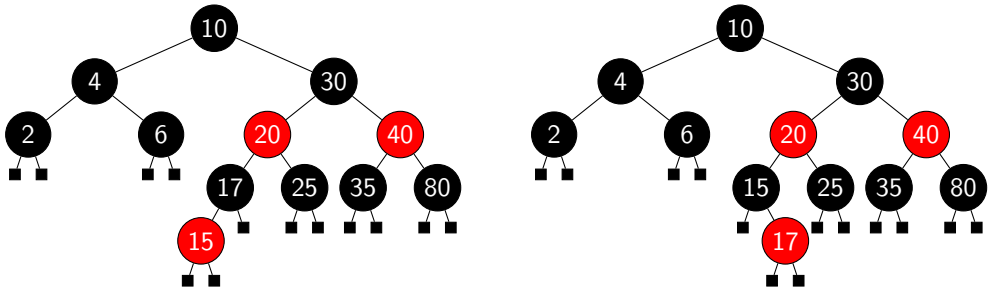
The space complexity of this version is $\mathcal{O}(1)$, and the time complexity remains unchanged.

25. Consider the following (2,4) tree.



(a) (1 point) Give a red-black tree that corresponds to the given (2,4) tree.

Answer:



(b) (1 point) How many different red-black trees correspond to the given (2,4) tree? Explain.

Answer: For each 3-node in the (2,4) tree, there are two choices. The given (2,4) tree has one 3-node, so there are 2 red-black trees that correspond to it.

(c) (2 points) List the properties of AVL trees.

Answer:

- An AVL tree should be a *binary search tree*. That is, each node with key k satisfies the following properties:
  - Keys stored in the left subtree are less (not equal) than k, and
  - Keys stored in the right subtree are greater (not equal) than k.
- An AVL tree should satisfy the *AVL balance condition*. That is, the heights of the children of each node differ by at most 1.

(d) (2 points) Is every red-black tree also an AVL tree?

- If you answer "yes", explain why the properties of AVL trees hold for every red-black tree.

- If you answer "no", give a red-black tree as a counter example, and explain which properties of AVL trees do not hold.

Answer: No. A counterexample is the red-black tree in the answer of question 25(a). This tree does not satisfy the AVL balance condition, the left subtree of the root has height 3 and the left subtree of the root has height 5.

End of the exam