Exam IN2010 fall 2025

5. December 2025

About the exam

- The exam consists of a tiny warm-up, followed by two main parts.
- The first part consists of small exercises, which are automatically corrected. No distinction is made between leaving a question unanswered and providing an incorrect answer; this means there is no reason to leave a question unanswered.
- The second part consists of more comprehensive exercises, where you to a greater extent have to provide pseudocode and written explanations.
- No aids are allowed.
- The full set of exercises is attached as a PDF.
- An English translation of the full set of exercises is attached as a PDF.

Comments and tips

- *Read the exercise very carefully*.
- Make sure you answer precisely what the exercise asks of you.
- Explain any assumptions you make.
- Make sure that your delivery is clear, precise, and easy to understand, both in terms of structure and content.
- If you get stuck on a problem, then you might want to proceed to another problem and return later.
- All exercises that require an implementation should be answered in *pseudo-code*. The important thing is that the pseudo-code is *easy to comprehend, unambiguous, and precise*.
- A *easily understandable, unambiguous and precise* explanation in natural language, may be worth more points than pseudocode that is difficult to comprehend, ambiguous or imprecise.
- In implementation tasks, lower runtime complexity is worth more points.
- You may assume that algorithms and data structures known from the syllabus are available, unless otherwise specified.

Warm-up 2 poeng

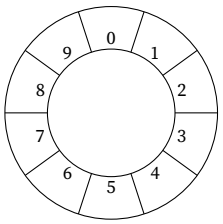
(a) What is an algorithm? Keep your answer brief (maximum four sentences).

(b) What is a data structure? Keep your answer brief (maximum four sentences).

Here, we are not after a “textbook answer". We are interested in seeing your understanding of the terms. Any reasonable answer gives a full score.

Separate chaining 10 poeng

(a) We start with an empty array of size $N = 10$.



The hash function you shall use is $h(k, N) = k \bmod N$, which for this example becomes $h(k, 10) = k \bmod 10$. That is, a number hashes to its last digit.

Use *separate chaining* to insert these key- and value pairs in the given order:

$$(21, a), (54, b), (82, c), (10, d), (20, e), (44, f), (10, g)$$

Write down the result after the last insertion, with each line in the form:

$$i\colon (k_1, v_1), (k_2, v_2), \ldots$$

where i is an index in the array, and $(k_1, v_1), (k_2, v_2), \ldots$ are key- and value pairs.

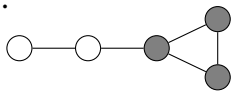
(b) Briefly explain how the algorithm for insertion with separate chaining works, and outline the algorithm with pseudocode. You may assume that you have a hash function h such that $h(k, N)$ yields a number between 0 and $N - 1$ for an arbitrary key k. You do not need to account for rehashing.

```
Input: An array A of size N, a key k and value v
Output: An array where (k, v) is inserted with separate chaining
1 Procedure SeparateChainingInsert(A, k, v)
  | // ...
```

(c) Briefly account for the runtime complexity of insertion with separate chaining.

Cycles in graphs 11 poeng

(a) Assume that the graph $G = (V, E)$ is an *undirected* graph. The graph contains a cycle if there exists a path from a node u back to the same node u that passes through at least one other node v.

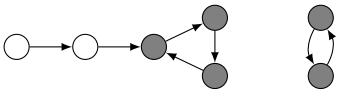


Write a procedure that determines whether the graph contains a cycle.

Input: An undirected graph $G = (V, E)$
Output: Returns **true** if the graph contains a cycle, and **false** otherwise

```
1 Procedure isCyclicUndirected(G)
  | // ...
```

(b) Assume that the graph $G = (V, E)$ is a *directed* graph. The graph contains a cycle if there exists a path from a node u back to the same node u.



Write a procedure that determines whether the graph contains a cycle.

Input: A directed graph $G = (V, E)$
Output: Returns **true** if the graph contains a cycle, and **false** otherwise

```
1 Procedure isCyclicDirected(G)
  | // ...
```

Anti-AVL 12 poeng

AVL trees use left and right rotations to ensure that the binary search tree ends up balanced after insertion and deletion. Here is insertion for a variant of AVL trees that we call *Anti-AVL*.

```
Input: The root node v of an Anti-AVL tree B
       and an element x
Output: An Anti-AVL tree with x inserted
1  Procedure InsertAntiAVL(v, x)
2      if v = null then
3          v ← new Node(x)
4      else if x < v.element then
5          v.left ← InsertAntiAVL(v.left, x)
6      else if x > v.element then
7          v.right ←
             InsertAntiAVL(v.right, x)
8
9      if v.left ≠ null then
10         return RightRotate(v)
11
12     return v
```

```
Input: A node z
Output: The root of the
        right-rotated tree
1  Procedure RightRotate(z)
2      y ← z.left
3      T2 ← y.right
4
5      y.right ← z
6      z.left ← T2
7
8      return y
```

(a) Insert the numbers $3, 1, 4, 2, 5$ with `InsertAntiAVL`.

- Briefly describe what the resulting Anti-AVL tree looks like.
- State the total number of rotations that occur during the insertions.

(b) Describe more generally what Anti-AVL trees look like.

(c) A binary search tree is a binary tree where the element in each node is:

- greater than all elements in the left subtree, and
- smaller than all elements in the right subtree.

Do Anti-AVL trees have the same property? Explain.

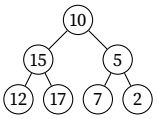
(d) What is the runtime complexity of lookup in Anti-AVL trees? You may assume that lookup is implemented identically to ordinary binary search trees and AVL trees. Briefly justify your answer.

(e) Briefly describe how insertion in Anti-AVL trees differs from insertion in AVL trees.

Level-order 12 poeng

A *level-order* traversal visits the nodes in a tree by *depth*. That is, the root is visited first, then all the children of the root, then all the children of the root's children, and so on.

A *level-order* traversal of the following tree can, for example, visit the nodes in the order $10, 15, 5, 12, 17, 7, 2$.



(a) Assume that the tree is a *binary heap*, represented by an array A with n elements. Give an algorithm that returns a list with the elements from A in *level-order*.

```
Input: A binary heap, represented by an array A with n elements
Output: A list with the elements in the heap in level-order
Procedure LevelOrderHeap(A)
    // ···
```

(b) Assume that the tree is a *binary search tree* with n nodes, where each node v has the following fields:

- v.`element` — the value stored in the node
- v.`left` — the left child of v
- v.`right` — the right child of v

Give an algorithm that returns a list with the elements from the tree in *level-order*.

```
Input: The root v of a binary search tree with n nodes
Output: A list with the elements in the search tree in level-order
Procedure LevelOrderBST(v)
    // ···
```

(c) State the runtime complexity for your algorithms in (a) and (b).

(d) Consider the following algorithm:

```
Input: The root v of a binary search tree
Output: A new binary search tree
Procedure ConstructBST(v)
    B ← empty binary search tree
    nodes ← LevelOrderBST(v)
    for u in nodes do
        B ← Insert(B, u)
    return B
```

What can you say about the relation between the input tree and the output tree?

Find the k smallest — 12 poeng

Given an array A with n comparable elements, we want to find the k smallest elements in A where $1 \leq k \leq n$.

Input: An array A of n comparable elements, and an integer $1 \leq k \leq n$
Output: A list with the k smallest elements from A

```
1 Procedure FindMin(A, k)
  | // ···
```

Write a procedure that computes `FindMin(`A`, `k`)` in:

(a) $\mathcal{O}(k \cdot n)$

(b) $\mathcal{O}(n \cdot \log(n))$

(c) $\mathcal{O}(n \cdot \log(k))$ *or* $\mathcal{O}(n + k \cdot \log(n))$

Public Transport in Oslo — 15 poeng

At the latest City Council meeting several resolutions were passed regarding the public transport system in Oslo. Let the public transport system be represented as a connected, undirected, weighted graph $G = (V, E)$, where $u \in V$ is a stop, an edge $\{u, v\} \in E$ is a *segment*, and the weight function w specifies the travel time (in minutes) for a segment.

(a) The graph is *connected* and *undirected*. Briefly explain what each of these properties means for someone travelling by public transport in Oslo.

(b) The City Council has decided that new stops will be built along all *segments* that exceed k minutes. During the construction work it will not be possible to travel along any of these segments. Write an algorithm that takes the graph representing the current public transport system, as well as a threshold k, and checks whether it will be possible to travel between all pairs of stops during the construction work.

(c) It turns out that the algorithm from subproblem (b) answers **`false`** for the k the City Council had in mind. Give an algorithm the City Council can use to find the smallest value of k for which the algorithm from subproblem (b) answers **`true`**.

(d) In the past year there have been two incidents where there was a total shutdown at a stop. This resulted in travellers being unable to reach their destination. The City Council has adopted a guarantee: if there is a total shutdown at a stop, it should still be possible for travellers to reach their destination via an alternative route. The guarantee applies to all stops.

Explain how we can use an algorithm known from the syllabus to check whether the public transport system provides this guarantee.

(e) The City Council meeting dragged on for too long, so they ordered pizza, and after that new creative ideas started pouring out. The last one in this series is the following: to relieve the most heavily trafficked routes, they want to give *negative ticket prices* on the least trafficked segments.

Before initiating the project they must ensure that it is not possible to accumulate arbitrarily large amounts in negative ticket prices by travelling around and around.

The ticket price for a segment is given by a weight function c. A segment can be heavily trafficked in one direction but lightly trafficked in the other, so $c(u, v)$ may differ from $c(v, u)$ for a segment $\{u, v\} \in E$.

Explain how we can use an algorithm known from the syllabus to check whether there exist ways to accumulate arbitrarily large amounts in negative ticket prices.