Data structures and algorithms, exam

(DIT962, DAT038, DIT182/3, DAT495, DAT525, TDA417)

2025-10-29, 14:00–18:00

Teachers: **Jonas Duregård, 031–772 1028**
Yehia Abd Alrahman, 031-772 6054
Peter Ljunglöf, 031–772 1065 (for questions during exam)
Alex Gerdes, Christian Sattler

We will visit each exam room at least twice.

Notes *Answer directly on the question sheets!*
If you need additional space, you can use extra pages.
Refer to them from the question sheet so that we don't miss them.

Write your anonymous code (not your name) on the first question sheet (and on all extra pages).

Don't tear out pages.

You may answer in English or Swedish.
You may write algorithms in pseudocode or your favourite programming language.

Excessively complicated answers may be rejected.
Write legibly, we need to be able to read your answer!

Questions and solutions will be published afterwards here:
https://github.com/ChalmersGU-data-structure-courses/past-exams

Allowed aids One *hand-written* A4 sheet of notes (double-sided).
If you make use of a sheet, you must hand it in along with your answers.

Review When the exams have been graded and shipped, they will be available for review and collection at the CSE Student Office during their opening hours.

If you want to discuss the grading, contact the examiner via email.
Remarks about the grading should be explained thoroughly. Attach a picture of your answer.

Grading The six basic questions are graded either **correct** or **incorrect**.
To pass the exam, you must correctly answer 5 of the 6 basic questions.

The three advanced questions are each awarded one of **0**, **1**, **2** advanced points.
For a higher grade, you must pass and also obtain a certain number of advanced points:

- 4: 2 out of 6 advanced points.
- 5: 4 out of 6 advanced points.
- VG: 3 out of 6 advanced points.

Anonymous code:_____

Basic question 1: Mergesort

Consider the following 8-element array:

0	1	2	3	4	5	6	7
12	7	14	9	10	11	6	2

We use merge sort to sort the elements in increasing order.

Show every merge step in the order that they are performed (the two input arrays and the merged output) and the final sorted array. Use the following table and use as many rows as you need for all the merge steps.

Array 1	**Array 2**	**Merged output**

Basic question 2: Hash tables

The hash table to the right uses linear probing with lazy deletion. The hash table uses modular compression (hash(x) % 10), but the hash function is unknown.

0	<deleted>
1	A
2	
3	
4	C
5	
6	E
7	B
8	
9	D

a) Which one of these is a possible hash function used to build the table?
Note that this is **not** the order values were added, just the hash function used.
(select only one option)

- ☐ hash(A)=29, hash(C)=4, hash(E)=16, hash(B)=7, hash(D)=21
- ☐ hash(A)=39, hash(C)=4, hash(E)=27, hash(B)=6, hash(D)=11
- ☐ hash(A)=49, hash(C)=4, hash(E)=36, hash(B)=7, hash(D)=19
- ☐ hash(A)=89, hash(C)=4, hash(E)=47, hash(B)=6, hash(D)=39
- ☐ hash(A)=71, hash(C)=4, hash(E)=56, hash(B)=7, hash(D)=11
- ☐ hash(A)=21, hash(C)=4, hash(E)=66, hash(B)=6, hash(D)=81
- ☐ hash(A)=11, hash(C)=4, hash(E)=76, hash(B)=7, hash(D)=99
- ☐ hash(A)=11, hash(C)=4, hash(E)=86, hash(B)=6, hash(D)=9

b) Add F with hash(F)=46 to the table (without resizing). You can add the value directly to the table above.

Basic question 3: BSTs

We start from an empty binary search tree (BST). We add values to it using the standard insertion procedure, in the following order: 50, 30, 70, 20, 40, 60, 80.

a) Draw the resulting **binary search tree**.

b) List the **nodes** in the order they are processed in a (recursive) pre-order and post-order traversal.
Hint: Pre-order traversal processes a node before making the recursive calls on its left and right children, while post-order [missing in printed version] traversal processes it after the recursive calls.

Pre-order: _____

Post-order: _____

Basic question 4: Data structure complexity

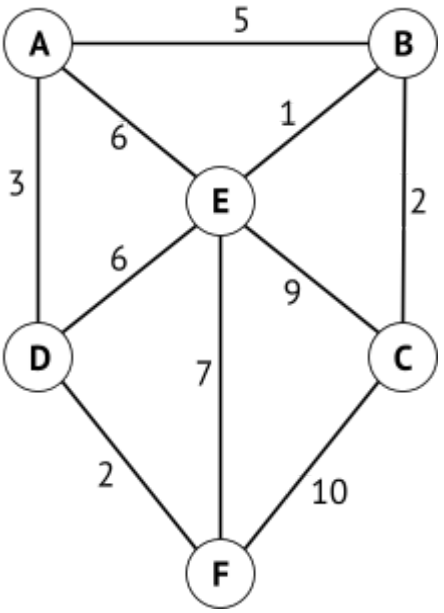
Below is a table (partially filled in) of the asymptotic complexity of some operations in efficient implementations of some data structures. The asymptotic complexity is expressed in terms of the number n of elements of the data structure.

Fill in the rest of the table.

Data structure	**getMin() (Finding smallest element, without modifications)**	**contains(x) (Searching for a specified element)**	**add(x) (Adding a new element)**
Sorted linked list			$O(n)$
Binary min-heap			amortized $O(\log(n))$
Dynamic array (not sorted)		$O(n)$	
AVL-tree		$O(\log(n))$	

Basic question 5: Graphs

We run this familiar piece of code on the graph to the right:




```
visited = new set of nodes
result = new list of node pairs
agenda = new min-priority queue of edges ordered by their weights
agenda.add( (null,0,E) )    // add a dummy edge for start node E
while agenda is not empty:
        (v, weight, w) = agenda.removeMin()
        if not visited.contains(w):
                visited.add(w)
                for (from, weight, to) in outgoingEdges(w):
                        if not visited.contains(to)
                                agenda.add( (from, weight, to) )
                result.append( (v, w) )
return result
```

a. What is this algorithm computing?

b. What are the contents of the set ***visited*** and the list ***result***, initially and after each of the first four iterations of the while loop? (Some values are pre-filled for explanatory purposes.)

   Note: you must report all of the contents of ***visited*** and ***result*** *in each iteration*.

Iteration	***visited***	***result***
before the loop	—	—
after iteration 1	E	(null, E)
after iteration 2		
after iteration 3		
after iteration 4		

Basic question 6: Complexity

Below are two (mostly nonsense) functions, each taking an integer array as a parameter.

State the asymptotic time complexity of:

- Algorithm 1 in terms of the **parameter k**.
- Algorithm 2 in terms of the **length *N* of the array**.

In each case, **briefly state** how you concluded this (you may do this by annotating the programs).

***Notes*:**

- Assume that all arithmetic operations and array indexing take O(1) time.
- Answer in O-notation, and as tight and simple as possible.

Algorithm 1

```
function f(x, arr, offs, k):
    if n [typo, should be k] ⩽ 0:
        return offs
    half = k // 2      (note: // is integer division)
    m = offs + half
    if arr[m] < x:
        offs = offs + half
    return f(x, arr, offs, half)
```

Complexity:

Justification:

Algorithm 2

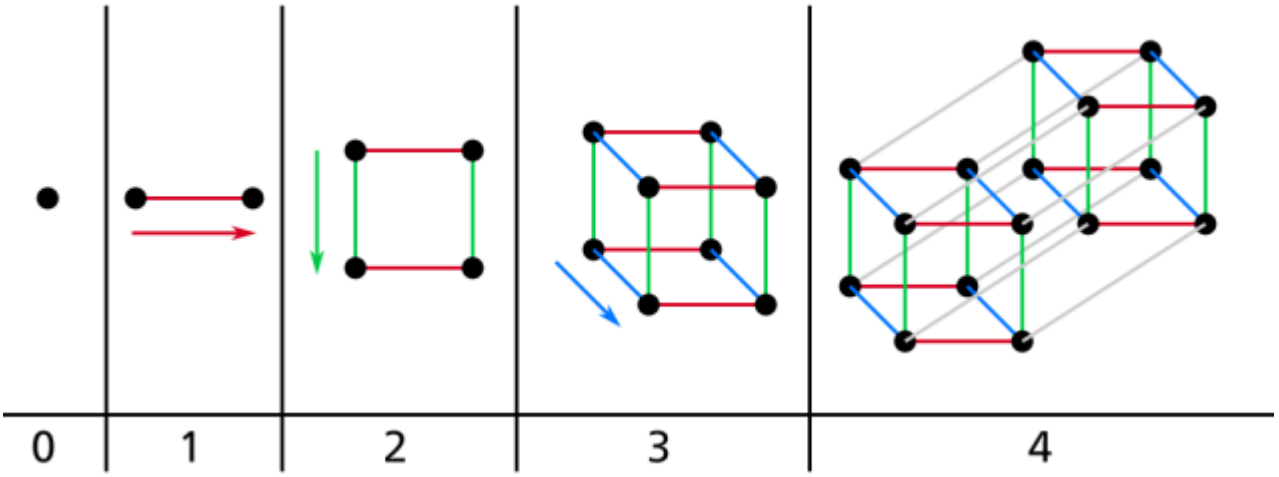
```
function rolling(arr):
    res = new dynamic array
    for i from 0 to arr.length−7:
        sum = 0
        for offset from 0 to 6:
            sum = sum + arr[i+offset]
        res.add(sum/7)
    return res
```

Complexity:

Justification:

Advanced question 7: Hypercubes

An N-dimensional hypercube can be considered as a graph. Here is an illustration for $0 \leqslant N \leqslant 4$:



Suppose you have an N-dimensional hypercube with undirected weighted edges, and you want to run an $O(E \log(E))$ implementation of Kruskal's algorithm on it (where E is the number of edges). What is the asymptotic complexity of doing so in terms of N, the number of dimensions of the hypercube? You should answer in a simplified O-notation, and explain your reasoning.

Hint: Notice how an N-dimensional cube is two copies of an $(N-1)$-dimensional cube, with exactly one additional edge to/from each node. This means all nodes in an N-dimensional cube will have a certain property in common that will help you calculate the number of edges.

Hint: This is going to be a slow complexity class.

a) Answer these questions about the graph for the N-dimensional hypercube:

Number V of nodes (in terms of N): _____

Number E of edges (in terms of N): _____

b) What is the asymptotic complexity of running Kruskal's algorithm on an N-dimensional hypercube? (answer in O-notation, and remember to simplify as much as possible)

_____

Explanation:

Advanced question 8: LinkedList Splitting

You are given a singly linked list that contains *n* elements. The class for LinkedList is as follows:

```
class LinkedList<Item>:
    Node head          // points to the head of the list
    class Node:
        Node next      // points to the next list node
        Item value     // the value of this node
```

Design an algorithm to split the list into two halves so that the following requirements are met:

- If n is even, both halves have *n*/2 nodes.
- If n is odd, the first half has one more node than the second.
- You may only use a single loop with at most *n* loop iterations (and no recursion).
- Your loop can use multiple pointers to traverse the list, but you may only use O(1) extra space (so no arrays or secondary lists are allowed).

Note: the number of nodes in the list is not known in advance. The class definition above is complete, there are no extra methods or variables you can use.

a) One attempt at a solution is to first count the number of nodes, and then split at the middle node. Why does this not meet the requirements?

b) Propose an algorithm that satisfies all conditions above. Use pseudocode or a detailed text description. Your algorithm should take a Node object as input and return two Nodes (the heads of the two halves).

c) Show that your algorithm runs in O(n) time and O(1) extra space.

d) Demonstrate your algorithm using the following example list:

1 → 4 → 7 → 9 → 12 → 15 → null

Show the steps your algorithm takes to split the above list in halves.

Write your solution on a separate sheet of paper.

Advanced question 9: Updateable priority queue

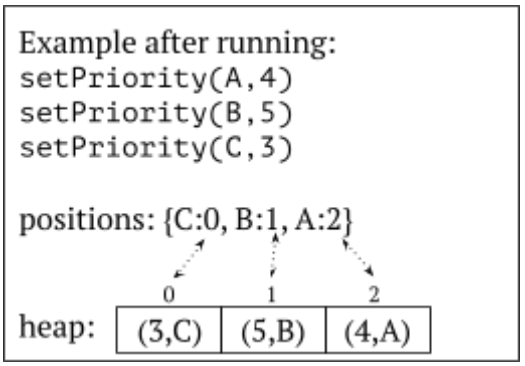
Here is a draft implementation of a priority queue of unique keys with updatable priorities. We use a heap together with an AVL-map (positions) to quickly find where keys are stored in the heap. In addition to the heap invariant, that means we have the following invariant:

If (and only if) *heap[i]=(prio,key)* then *positions.get(key) = i*

```
class MinPrioMap<Key, Priority>:
    positions: AVL-map from Key to integers        // maps keys to positions in the heap
    heap: Dynamic Array of (Priority, Key)-pairs   // binary heap ordered by priorities

    function setPriority(Key key, Priority prio):
        if positions.containsKey(key):             // key is already present, update priority
            i = positions.get(key)
            (oldPrio, key) = heap[i]
            if prio < oldPrio:                     // we only update priority if the new priority is lower
                heap[i] = (prio, key)
                swim(i)
        else:        // this is a new key, extend the heap
            heap.add( (prio, key) )
            swim(heap.length-1)
```

The illustration to the right shows the connection between the positions-map and the heap when the invariant holds.



a) Implement the method **swim**, such that **setPriority** preserves the invariants.

b) What is the (amortized) complexity of **setPriority** using your **swim** function, in terms of the number of values in the queue *n*?