DAT038/TDA417
Data structures and algorithms
Written examination

Thursday, 2021-01-14, 14:00–18:00, in Canvas and Zoom

Examiner(s)	Peter Ljunglöf, Nick Smallbone, Christian Sattler.
Instructions	Read all instructions on the front page of the Canvas exam room.
Allowed aids	Course literature, other books, the internet. ***You are not allowed to discuss the problems with anyone!***
Individualised	In most questions you will be asked to work on an individual problem instance. You will get your individualised problem from a specific Canvas page.
Submitting	Submit your answers as answers to Canvas quizzes. There is one Canvas quiz per question.
Pen & paper	Some questions are easier to answer using pen and paper, which is fine! In that case, take a photo of your answer and upload to your quiz answer. Make sure the photo is readable!
Assorted notes	You may answer in English or Swedish. Excessively complicated answers might be rejected. We need to be able to read and understand your answer!
Exam review	If you want to discuss the grading, please contact Peter via email.

There are 6 basic, and 3 advanced questions, and two points per question. So, the highest possible mark is 18 in total (12 from the basic and 6 from the advanced). Here is what you need for each grade:

- To **pass** the exam, you must get 8 out of 12 on the basic questions.
- To get a **four**, you must get 9 out of 12 on the basic questions, plus 2 out of 6 on the advanced.
- To get a **five**, you must get 10 out of 12 on the basic questions, plus 4 out of 6 on the advanced.

Grade	Total points	Basic points	Advanced
3	⩾ 8	⩾ 8	—
4	⩾ 11	⩾ 9	⩾ 2
5	⩾ 14	⩾ 10	⩾ 4

Question 1: Complexity

Get a program and a part B question from here ("Q1: complexity"):
https://chalmers.instructure.com/courses/13943/pages/individualised-problem-instances

Part A

- What is the worst-case time complexity of the above program?
- Justify that the program has this complexity.

Note: Your complexity should be as simple as possible, and as low as possible.

Part B

Get a part B question from the link above.

- What is your answer to the question?
- Justify your answer.

Question 2: Sorting

In this question, you will show how the *partitioning* algorithm in quicksort works.

Start by getting an individualised array from here ("Q2: Sorting"):
https://chalmers.instructure.com/courses/13943/pages/individualised-problem-instances

Part A

Take the array you got from the link above, and partition it, using the *median-of-three* strategy to choose the pivot. (Do not make any recursive calls to quicksort.) Your answer should show:

- The sequence of swaps that the partitioning algorithm performs.
- What the array looks like after partitioning.
- Which parts of the array are the two partitions.
  You can indicate the partitions using square brackets like this: [11 22 33] 44 [55 77 66].

If you use a different partitioning algorithm from the one in the course, write down what algorithm you used (either its name or a description).

Part B

Now go back to the array you were given at the start of the question (*not* your answer to Part A).

Swap two array elements, so that afterwards the array has the following property: if we partition it using the median-of-three strategy, one of the partitions will only contain one element.

Note: you are only allowed to swap two elements, not more.

Write down which elements you swapped, and briefly explain why partitioning using the median-of-three strategy will produce a partition containing only one element.

Hint: Think about which element must be chosen as the pivot for this to happen.

Question 3: Search trees

In this question you will show how insertion works in AVL trees.

Start by getting an individualised AVL tree from here ("Q3: Search trees"):
https://chalmers.instructure.com/courses/13943/pages/individualised-problem-instances

Part A

Using the AVL insertion algorithm, insert an element into the AVL tree, so that:

- During insertion, **the tree becomes unbalanced**,
  and the AVL insertion algorithm rebalances it
- The node that gets rebalanced is either a **left-left or right-right case**

You choose which element to insert, **but it must create either a left-left or right-right unbalanced tree.** You should then use the AVL insertion algorithm to rebalance the tree.

Your answer should show:

- What element you inserted
- The unbalanced tree after the element is added (in other words, the tree that would have been created if you just used ordinary BST insertion)
- Which node the AVL insertion algorithm needs to rebalance, and which kind of imbalance it has (left-left or right-right)
- What rotations the algorithm performed to rebalance the tree
- The final balanced tree

If you have trouble scanning your answers, you can instead write your tree as a nested list (but pictures are fine too!). The individualised AVL tree you got comes with an example of this.

Part B

Repeat part A, but this time **create either a left-right or right-left unbalanced tree**. Start from the individualised AVL tree you got at the beginning, *not* the tree you created in Part A.

Just the same as Part A, write down:

- What element you inserted
- The unbalanced tree after the element is added (in other words, the tree that would have been created if you just used ordinary BST insertion)
- Which node the AVL insertion algorithm needs to rebalance, and which kind of imbalance it has (left-right or right-left)
- What rotations the algorithm performed to rebalance the tree
- The final balanced tree

Question 4: Priority queues

In this question, you will implement an algorithm that takes as input two priority queues, and returns a *sorted list* containing all the elements of both priority queues, in ascending order. For example, if the algorithm is called on the inputs {1,2,3,5} and {1,2,4}, it should return a list containing {1,1,2,2,3,4,5}.

Start by getting an unfinished implementation of the algorithm from here ("Q4: Priority queues"): https://chalmers.instructure.com/courses/13943/pages/individualised-problem-instances

The code defines a method `mergePQs` which takes two `PriorityQueue`s as input, and returns its result in a dynamic array. It's unfinished – the main loop of the method currently just reads **"???"**. Read the implementation and make sure you understand what it does so far.

To keep the code simple, the method `mergePQs` only works on priority queues containing integers.

Part A

Complete the code of `mergePQs`, by filling in the part labelled **???**. After `mergePQs` has finished running, it should return the combined contents of both input priority queues, *in ascending order*. You are allowed to modify the priority queues, for example by adding or removing elements.

Hint: remove one item in each iteration of the while-loop. Remember to handle the case when one of the priority queues is empty.

In your solution, you are *only allowed to call the following methods* from the course API:

- PriorityQueue interface: add, getMin, removeMin, isEmpty, size
- List interface: addFirst, addLast

When you call these methods, *you do not need to write out their implementation*. In addition, you can use ordinary integer operations such as + and < on `Int`.

Give your answer in *pseudocode* or a suitable programming language. An example of pseudocode can be seen in question 1.

Part B

What is the worst-case complexity of `mergePQs`, in big-O notation? Assume that both priority queues have a size of n, and are implemented using binary heaps. Briefly justify your answer.

Note: Your complexity should be as simple as possible, and as low as possible.

Question 5: Hash tables

Get an individualised hash table and hash values from here ("Q5: hash tables"): https://chalmers.instructure.com/courses/13943/pages/individualised-problem-instances

Part A

Assume that the open addressing hash table you got has never been resized, and that no elements have been deleted from it. Also assume that we use the hash value modulo the array size to get the array index, and standard linear probing for conflict resolution (i.e., probe by increasing by one).

- In which order can the elements have been added to the hash table, under the conditions above? It's enough that you give one possible answer.
- Explain what hash collisions there were, and how they were resolved.

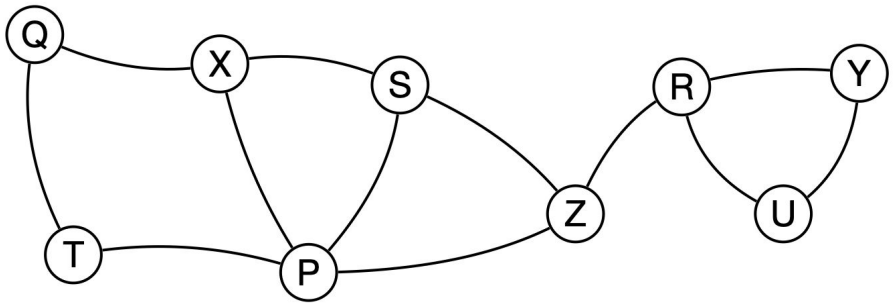
Part B

Now, add the same elements in the order from part A, to a *separate chaining hash table of size 5*. Once again, use modulo the array size to get the array index. Assume that you use linked lists as the container.

- How will the hash table look like when all elements have been added?
- Explain how elements are added to the linked list (in the front or back or somewhere else), and what conflicts you got when adding the elements to the hash table.

Question 6: Graphs

Assume the following undirected graph:



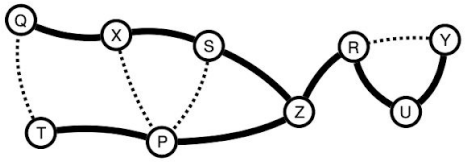
This graph is actually weighted, so get your individualised edge weights from here ("Q6: graphs"): https://chalmers.instructure.com/courses/13943/pages/individualised-problem-instances

Part A

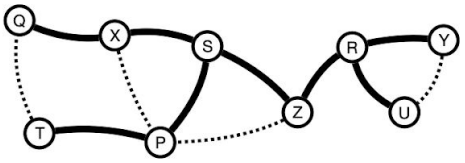
Classify each of the following subgraphs A–F of the graph above as:

- **MST**: a minimum spanning tree
- **SPT from S**: not an MST, but a shortest path tree starting from node S
- **ST**: neither an MST nor a SPT from S, but a spanning tree
- **None**: not a spanning tree at all

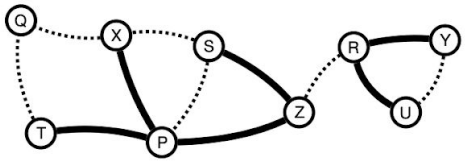
A)



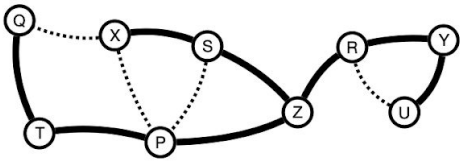
D)



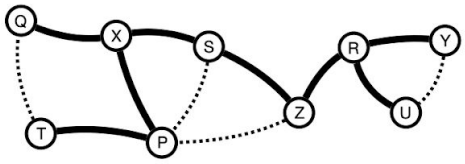
B)



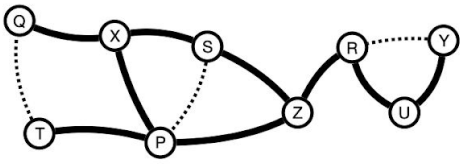
E)



C)



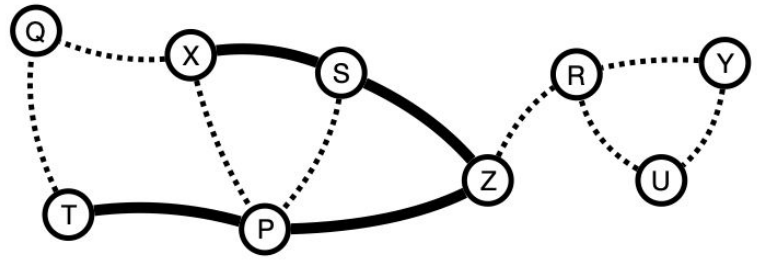
F)



For each of the graphs, briefly justify why it is classified as MST, SPT, ST or None.

Part B

Assume that you are running Prim's or Kruskal's algorithm on the graph, and after some iterations you have reached the following state (where the bold edges are the ones already added to the final MST):



Which edge will be added next to the MST by each of the algorithms?

- by Prim's algorithm? (Assume that the search started in node S)
- by Kruskal's algorithm?

Briefly justify your answers.

Advanced question 7: Complexity

Get an individualised two-variable orders of growth A, B, C, D, E from here ("Q7: Complexity II"): https://chalmers.instructure.com/courses/13943/pages/individualised-problem-instances

Part A

Simplify the given orders of growth. Use O-notation. Justify your answer.

Part B

Draw, or list the edges of, a directed graph that describes the relationships between the given orders of growth: there should be a path from O(X) to O(Y) exactly if X is O(Y). Justify why these relations hold and no other.

Advanced question 8: Cyclic order binary search

Let **A** be an array of $n \geqslant 1$ integers. Assume that **A** is in *strict cyclic order*: $\mathbf{A}[i] < \mathbf{A}[(i + 1) \bmod n]$ for all indices i except one.

(Note that this implies that all elements of **A** are distinct.)

Part A

Find an algorithm that computes the index of the minimum of **A** and runs in time $O(\log n)$.

```
int findMinIndex(int[] A)
```

Part B

Find an algorithm that checks if **A** contains an element v and runs in time $O(\log n)$:

```
boolean contains(int[] A, int v)
```

Give your algorithms in pseudocode or a suitable programming language.

Justify why your algorithms work, and why they have these complexities.

Advanced question 9: Bidirectional search

Modify the UCS/Dijkstra algorithm so that it can do *bidirectional search*:

- "Bidirectional search is a graph search algorithm that finds a shortest path from an initial vertex to a goal vertex in a directed graph. It runs two simultaneous searches: one forward from the initial state, and one backward from the goal, stopping when the two meet. The reason for this approach is that in many cases it is faster" (https://en.wikipedia.org/wiki/Bidirectional_search)

Here's a formulation of the UCS algorithm, similar to the one from the lecture slides:

```
List<Node> searchUCS(Node start, Node goal):
    pqueue = new PriorityQueue(comparing (e) -> e.costToHere)
    pqueue.add(new PQEntry(start, 0, null))
    Set<Node> visited = new Set()

    while pqueue is not empty:
        PQEntry entry = pqueue.removeMin()

        if entry.node == goal:
            return extractPath(entry)

        if entry.node not in visited:
            visited.add(entry.node)
            for Edge edge in graph.outgoingEdges(entry.node):
                costToNext = entry.costToHere + edge.weight
                pqueue.add(new PQEntry(edge.to, costToNext, entry))

    return null // Failure
```

It uses an auxiliary class of priority queue entries, and a method that extracts a path from the start node to the current:

```
class PQEntry:
    Node node
    double costToHere
    PQEntry backPointer

List<Node> extractPath(PQEntry entry):
    ... // you don't have to know how this method works
```

The graph API is the following:

```
class Edge<Node>:
    Node from
    Node to
    double weight

class Graph<Node>:
    Collection<Edge<Node>> outgoingEdges(Node from)
    Graph<Node> reversedCopy()
```

This graph API is very similar to the one from lab 4 or the lecture slides, with one important addition: the `reversedCopy()` method. This will create a new copy of the graph with the directions of all edges reversed (so that from becomes to and to becomes from). The existing graph will not be changed by calling this method – it creates a new graph instead.

Hints:

- Don't try to use concurrency or parallelisation. Instead alternate between the forward and the backward search – remove from the first, then from the second, then from the first, then from the second, etc.
- Think about how you can make use of the "active" nodes – the nodes that are currently in the priority queue, but have not yet been visited.
- You can use any class or method from the course API or the Java standard API (java.lang.* or java.util.*).

Give your algorithm in pseudocode or a suitable programming language.

Explain how your algorithm works.

Side note: This simple version of bidirectional search does not guarantee that it will return the optimal solution, as explained in footnote 1 of the Wikipedia article. There are some additional things you have to do to get an optimal algorithm, but your solution will return an almost-optimal solution in most cases.