Solution suggestions for reexam 2024-08-22

Basic question 1: Selection sort

Perform in-place *selection sort* on the following array (i.e., *not* insertion sort!):

0	1	2	3	4	5	6	7
48	57	14	83	64	35	20	74

Write down how the array looks after each iteration of the outer loop (*i* is the outer loop variable) of the selection sort algorithm.

Mark which cells are modified compared to the previous content, by circling or underlining them.

	0	1	2	3	4	5	6	7
i=0	14	57	48	83	64	35	20	74
i=1	14	20	48	83	64	35	57	74
i=2	14	20	35	83	64	48	57	74
i=3	14	20	35	48	64	83	57	74
i=4	14	20	35	48	57	83	64	74
i=5	14	20	35	48	57	64	83	74
i=6	14	20	35	48	57	64	74	83

Note: the green cells mark the prefix that is guaranteed to be sorted after each iteration.

Basic question 2: Hash tables

You have the following linear probing hash table supporting lazy deletion:

0	1	2	3	4	5	6
London			Rom	Oslo	Paris	Köln

The hash function is the length of the string (modulo the size of the array).

a) Delete **Köln** from the table.

What cell indices do you have to look at, and in which order, to find **Köln** in the table?

4 (Oslo), 5 (Paris), 6 (Köln)

How does the table look after **Köln** has been deleted?

0	1	2	3	4	5	6
London			Rom	Oslo	Paris	DEL

Note: the green cells are the ones that are not changed.

b) Now insert **Ulm** into the table from part (a).

What cell indices do you look at, and in which order, to find a place where to insert **Ulm**?

3 (Rom), 4 (Oslo), 5 (Paris), 6 (DEL), 0 (London), 1 (empty)

How does the table look after **Ulm** has been inserted?

0	1	2	3	4	5	6
London	Ulm		Rom	Oslo	Paris	DEL

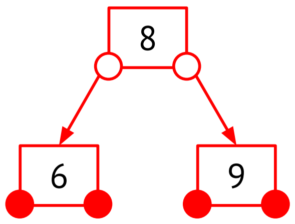
Or alternatively:

0	1	2	3	4	5	6
London			Rom	Oslo	Paris	Ulm

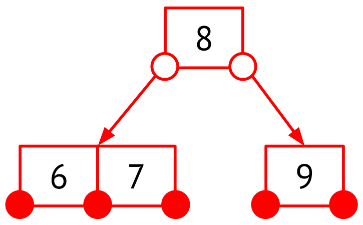
Basic question 3: 2–3 trees

Insert the numbers **8**, **9**, **6**, **7**, and finally **5** into an initially empty 2–3 tree, in that order. Show the state of the 2–3 tree after the following steps.

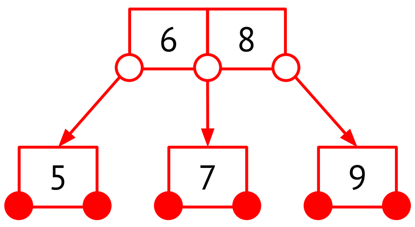
a) State of the 2–3 tree after inserting **8**, **9**, and **6**:



b) State of the 2–3 tree after inserting **7**:



c) Final state of the 2–3 tree (after inserting **5**):



Basic question 4: Binary heaps

A (min) binary heap is an efficient implementation of a (min) priority queue. Here are some other ideas for implementing a priority queue. What asymptotic complexities would you expect for the basic priority queue operations in these implementations?

a) *Unsorted linked list*: Add new elements to the front of the linked list, and perform a linear search for the minimal element when needed.
b) *Sorted dynamic array*: The array elements are in *descending* priority order, meaning that the minimal element will always be the *last* element.
c) *Binary heap*: And for comparison, what are the complexities if you use a binary heap?

Write the asymptotic complexities in the following table, using the size N of the priority queue as the parameter. Use amortised complexity where it makes a difference. Use O-notation and be as exact and simple as possible.

	`insert`	`removeMin`	`findMin`
(a) unsorted linked list	$O(1)$	$O(N)$	$O(N)$
(b) sorted dynamic array	$O(N)$	$O(1)$	$O(1)$
(c) binary heap	$O(\log N)$	$O(\log N)$	$O(1)$

Justification for the complexities in (a) and (b):

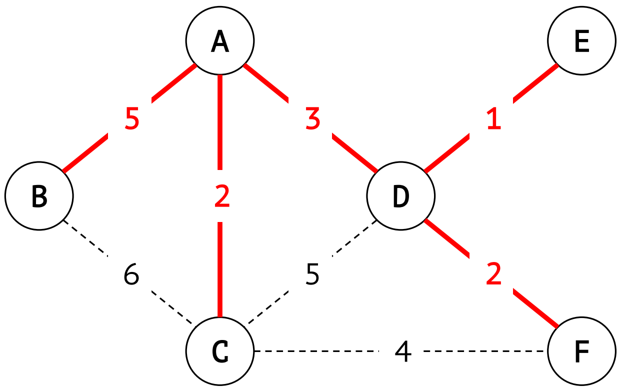
(a) unsorted linked list
- `insert`: adding to the front of a linked list is constant time
- `removeMin`: to remove an element we first have to find it
- `findMin`: linear search is linear time

(b) sorted dynamic array
- `insert`: in the worst case we have to insert the new element at position 0, and then we have to move every existing element one step to give room for the new one, which has linear complexity
- `removeMin`, `findMin`: finding and removing the last element in a dynamic array is constant time

Basic question 5: Prim’s algorithm

Run Prim’s algorithm twice
on the graph to the right:

- the first time starting at vertex **A**,
- the second time starting at vertex **F**.



a) Draw the minimum spanning tree obtained from one of these runs by marking the edges directly on the graph: make the edges thicker or coloured.

b) In which order are the edges added, if you start from vertex **A**?
(Write the edge between X and Y as XY)

AC, AD, DE, EF, AB

c) In which order are the edges added, if you start from vertex **F**?

DF, DE, AD, AC, AB

Basic question 6: Complexity

Here is a function that calculates the *disjunctive union* (also called *symmetric difference*) $X \oplus Y$, which is the set of all elements that occur in either X or Y but not in both.

Instead of returning a new set, the function updates X so that it is the disjunctive union of Y and the original state of X.

```
function disjunctive_union(X,Y):
    Z = new empty set           $O(1)$                 = $O(1)$
    for each x in X:            $O(n)$ ×               = $O(n \log n)$
        if Y contains x:          [$O(\log n)$ +
            add x to Z             $O(\log n)$]
    for each y in Y:            $O(n)$ ×               = $O(n \log n)$
        add y to X                 $O(\log n)$
    for each z in Z:            $O(n)$ ×               = $O(n \log n)$
        remove z from X            $O(\log n)$
```

What the function does is to add all Y elements to X, and then remove all elements that were common to both X and Y (which is the set Z). Note that this is not a very good implementation, see the next question if you want to improve it.

Assume that the sets X and Y have the same number of elements n. What is the asymptotic worst-case time complexity of the function in terms of n, if the sets are represented as **AVL trees**? Be as exact and simple as possible. Justify your answers.

Complexity: $O(n \log n)$

Justification:

(You can also annotate directly in the code above, but it has to be understandable!)

See the line annotations in the code above.

Each loop body is multiplied with the number of times the loop is iterated, which is the same for each loop, $O(n) \times O(\log n) = O(n \log n)$. Note that the if-clause is treated as a sequence, not a loop, so we sum their complexities, $O(\log n) + O(\log n) = O(\log n)$.

The for-loops are run in sequence, which means that we sum their time complexities, $O(n \log n) + O(n \log n) + O(n \log n) = O(n \log n)$.

Advanced question 7: Disjunctive union

The implementation for the disjunctive union in question 6 is not very good, for some reasons:

- it creates a temporary set *Z*, which is unnecessary
- it is not the most efficient, which becomes evident when the sets have different sizes

a) Let *n* be the size of *X* and *m* be the size of *Y*. Assume that *X* is larger than *Y* (so that $n > m$). What is the asymptotic time complexity of the implementation in question 6 in terms of *n* and *m*? Analyse the three loops separately, and be as exact and simple as possible.

```
function disjunctive_union(X,Y):
    Z = new empty set
    for each x in X:          }
        if Y contains x:      }  O( n log m )
            add x to Z        }
    for each y in Y:          }  O( m log n )
        add y to X            }
    for each z in Z:          }  O( m log n )
        remove z from X       }
```

Optional justification:

The size of *Z* is at most the size of *Y*, so the last loop is iterated O(*m*) times.

b) Now give a better in-place implementation of the function. It should still modify the first set *X*, but it should not create any temporary set (or other data structure), and also not modify the set *Y*. Furthermore it must have a strictly better time complexity than the original function above (e.g., since $m < n$, O(*m*) is strictly better than O(*n*)).

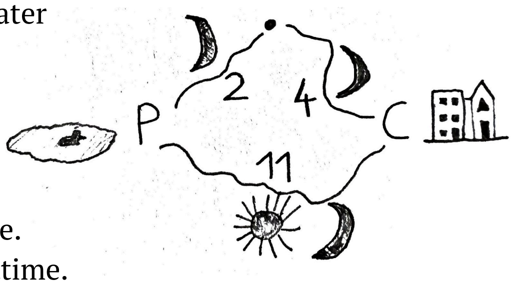
```
function disjunctive_union(X,Y):
    for each y in Y:            O(m) ×
        if y in X:                [O(log n) +
            remove y from X        O(log n) +
        else:
            add y to X             O(log n)]
```

c) What is the asymptotic complexity of your implementation in terms of *n* and *m*? (The sets are still represented as AVL trees.)

Complexity: O(*m* log *n*)

Justification can be annotated alongside with your code above.

Advanced question 8: Escape from the wilderness



You awaken at the pond (location P), the only source of water in the wilderness. You must reach the city (location C). Luckily you have a good map.

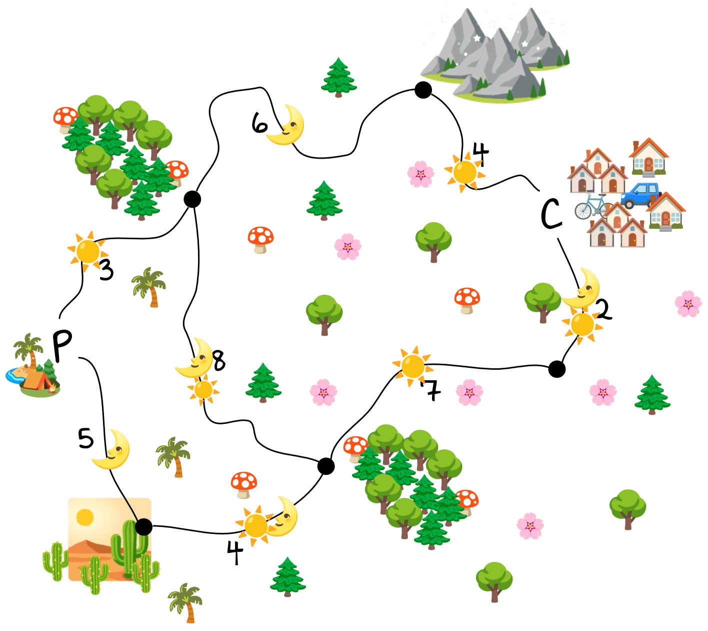
Your map displays all the locations and trails in the wilderness. Each trail goes from a location to another location and requires a certain amount of water to traverse. Some trails are only safe to travel during daytime or nighttime. You start out at daytime. After traversing a trail, daytime and nighttime switch. At any location, you may wait through daytime or nighttime for 3 water.

Design an algorithm that computes the smallest amount of water you need to take from the pond (your starting location) for your journey to the city. It should run in O(*n* log(*n*)) where *n* is an upper bound for the numbers of locations and trails.

```
type Location      // supports comparison in O(1) time

class Trail:
    from, to: Location
    day, night: boolean     // can the trail be traversed during the day/night?
    water: int              // water you consume walking the trail (positive number)

constant P, C: Location     // the locations for the pond and the city

constant trails: Map from Location to (List of Trail)
        // trails.get(x) returns the list of trails starting at location x in O(1) time
```

Notes:

- Describe the algorithm clearly, for example using pseudocode or Java/Python/Haskell.
- You may freely use data structures and algorithms from the course without implementing them yourself.



See next page for a solution suggestion

Solution suggestion for advanced question 8

A short solution is to use ordinary uniform-cost search in a custom graph whose type of vertices includes information on whether it is daytime or nighttime.

```
enum Time = {day, night}

type Vertex = pairs of Location and Time

def outgoingEdges(⟨location, time⟩: Vertex) → list of pairs of target vertex and cost
    if time is day:
        yield ⟨location, night⟩, cost 3
        for trail in trails.get(location):
            if trail.day:
                yield ⟨trail.to, night⟩, cost trail.water

    if time is night:
        yield ⟨location, day⟩, cost 3
        for trail in trails.get(location):
            if trail.night:
                yield ⟨trail.to, day⟩, cost trail.water
```

Here, **yield** is shorthand for adding an item to the result list (the function could be a generator).

Now we call uniform-cost search (Dijkstra's algorithm) with start vertex ⟨P, daytime⟩ and goal vertices ⟨C, daytime⟩ and ⟨C, nighttime⟩ (we can stop the search when either is found). The cost of the shortest path returned is the amount of water needed for the journey.

Advanced question 9: Min-stack

It is possible to make a data structure that is a stack, extended with a `getMin` operation to get the minimal element currently in the stack, such that `push`, `pop` and `getMin` are all O(1) operations.

A colleague of yours suggests just using a standard stack along with a single variable for storing the current minimum, updating it when the stack is modified, and just returning it for `getMin`. Here's how it might look for a min-stack of integers:

```
class MinStack:
   stack: Stack
   minValue: int

   push(x): ...

   pop(): ...

   getMin(): ...
```

a) Impress your boss by explaining why your colleague's solution would not work without making at least one operation slower than O(1).

The problematic case is when we want to pop, and the minimum value is on the top.

Popping the stack itself is not a problem, but then we have to assign a new minimum value. And since the only information we have is the stack itself, we can only do a linear search to find the (new) minimum.

b) Impress further by suggesting an implementation that would work.

Hint: You can use more than one stack as part of the implementation.

There are many possibilities. One idea is to use an additional stack of all the previous minimum values. So instead of having a single minValue we have a stack of minValues.

```
class MinStack:
   stack: Stack
   minValues: Stack

   push(x):
     stack.push(x)
     if minValues.empty() or x <= minValues.peek():  //x is a new minimum
       minValues.push(x)                             // so we push x to min-values

   pop():
     x = stack.pop()
     if x == minValues.peek():          // x is a current minimum,
       minValues.pop()                  // so we pop it off the min-values
     return x

   getMin():
     return minValues.peek()
```

Another idea is to use the stack to store pairs of values of minimum up to this point.

```
class MinStack:
   stack: Stack of Pair of int and int

   push(x):
       stack.push((
           x,
           x if stack.empty() else min(x, getMin())
       ))

   pop():
       return stack.pop().first

   getMin():
       return stack.peek().second
```