Exam for DIT182 (and re-exam for DIT181, LET375, DAT038, DAT037, TDA417, TDA416, and DAT525) Datastrukturer och algoritmer

Tuesday, 2022-03-15, 14:00–18:00

Solution suggestions

Teachers DIT182 and DIT181: Christian Sattler
LET375: Pelle Evensen
The rest: Peter Ljunglöf and Nick Smallbone

Allowed aids None

Exam review When the exams have been graded, they are available for review in the CSE student office at Johanneberg (DAT, TDA) or Lindholmen (DIT, LET).

You can discuss the grading at the exam review on Friday, March 25, 15–17:
- Johanneberg: EDIT building, room 6128 (6th floor)
- Lindholmen: Jupiter building, room 424 (4th floor)

In that case, leave your exam in the student office until after the review.
We will bring all exams to the review meeting.

Notes Write your anonymous code (not your name) on every page.
You may answer in English or Swedish.
Excessively complicated answers might be rejected.
Write legibly – we need to be able to read your answer!
You can write explanations on the question sheet or on separate pages.

There are **6 sections**, each containing **2 questions**, making **12 questions total**.
Each question is graded as **correct** or **incorrect**. Here is what you need to do to get each grade:

Grade	Sections with ⩾1 correct answer	Sections with both answers correct
3	5	—
4	5	2
5	6	4

Good luck!

Section 1: Complexity

Question 1A

What is the asymptotic complexity of the following code in terms of n?

Write your answer in O-notation. Be as exact and simple as possible. Justify that the complexity of the program has this order of growth.

```
M = new red-black map
for i from 1 to n:
    M.put(i, 0)
    for j from 1 to i*i:
        M.put(i, M.get(i)+j)
```

Answer: $O(n^3 \log n)$

Justification:

The outer loop has $O(n)$ iterations. In each iteration, M gets a new entry. So M has size $O(n)$. That means *get* and *put* in M have complexity $O(\log(n))$.

The inner loop has $O(n^2)$ iterations and a body of complexity $O(\log(n))$, so has complexity $O(n^2 \log(n))$. The line before it is $O(\log(n))$, which we ignore because it is smaller. The outer loop has $O(n)$ iterations, so has complexity $O(n^3 \log(n))$.

Question 1B

This question is about the order of growth of a mathematical expression. Consider the following expression in x, y and z:

$$3xy \log_2(x) + (z + 1)^2 + xyz$$

a. Suppose that y and z are constants, but the value of x can vary.
   What is the order of growth in terms of x?
b. Now suppose that x is a constant, but the values of y and z can vary.
   What is the order of growth in terms of y and z together?

Write your answers in O-notation, as simply and exactly as possible. Explain your reasoning, e.g., by showing the steps you took to simplify the function to an order of growth.

Answer (a): $O(x \log(x))$

Answer (b): $O(z(y + z))$ [or $O(yz + z^2)$, or $O(z \cdot \max(y, z))$, or $O(\max(yz, z^2))$]

Explanation:

(a) $O(3xy \log(x) + (z+1)^2 + xyz) = O(x \log(x)) + O(1) + O(x) = O(x \log(x))$.
(b) $O(3xy \log(x) + (z+1)^2 + xyz) = O(y) + O(z^2) + O(yz) = O(z^2) + O(yz)$ [since O(y) is smaller than O(yz)] $= O(z(y+z))$.

Section 2: Using data types and data structures

Question 2A

Pair each of the following four abstract data types (ADTs) with a use case for which it is well-suited. Note that each ADT goes with exactly one use case, and each use case goes with exactly one ADT.

A: stack	**P:** a queue for an emergency room at a hospital, where the sickest patients should be treated first
B: queue	**Q:** a database of people, where you can find information about a person by giving their personnummer
C: map	**R:** testing if a string contains matching parentheses and brackets e.g. correct: `([](([])())[()])`, incorrect: `[())`
D: priority queue	**S:** a waiting list for a course – when someone drops out of the course, the student who signed up earliest takes their place

You can e.g. draw lines between each ADT and its use case.

For each ADT, say which operations the ADT supports that are important for the use case:

A. **use case R**: A stack supports LIFO inserting (*push*) and removing (*pop*), which is necessary to remember which was the most recent opening parenthesis when reading a closing parenthesis.

B. **use case S**: Queues are FIFO, which means that the students who have waited (*enqueue*) the longest get their turn (*dequeue*) first.

C. **use case Q**: Using maps, we can efficiently look up (*get*) information about a key (in this case, the personnummer). (We can also efficiently update the database using *put* and *remove*.)

D. **use case P**: If the priority is how sick a patient is, then a priority queue supports removing the sickest patient from the queue first (*removeMin*). Patients can enter (*add*) in arbitrary order.

Question 2B

In this question, you will show how to implement a data structure you may not have seen before, called a *bidirectional map*. A bidirectional map is a map which supports two kinds of lookup: given a key, you can find the corresponding value, and given a value, you can find the corresponding key. All entries (key-value pairs) in a bidirectional map must have a different key (like in an ordinary map) and also a ***different value***.

A bidirectional map supports the following operations from maps:

- `put(k, v)`: add the entry k ↦ v to the map.
  Any existing item with key k ***or value v*** is removed.
- `get(k)`: if the map contains an entry k ↦ v, return v.

plus the following *reverse lookup* operation:

- `reverseGet(v)`: if the map contains an entry k ↦ v, return k.

The following table shows an example sequence of operations.

Operation	**Result**
create new map	Map is {}
`put(1, "hello")`	Map is {1 ↦ "hello"}
`put(2, "world")`	Map is {1 ↦ "hello", 3 ↦ "world"}
`get(1)`	Returns "hello"
`put(3, "hello")`	Map is {2 ↦ "world", 3 ↦ "hello"}. Notice that the item 1 ↦ "hello" is replaced by 3 ↦ "hello" since it is not allowed for two entries to have the same value.
`reverseGet("world")`	Returns 2

We can implement a bidirectional map using *two* maps:

- `forward` is a map from keys to values.
  In the example above, at the end, it contains 2 ↦ "world" and 3 ↦ "hello".
- `backward` is a map from *values* to *keys*.
  In the example above, it contains "world" ↦ 2 and "hello" ↦ 3.

The data structure has the following invariant: `forward` contains the entry k ↦ v if and only if `backward` contains the entry v ↦ k. In other words, the implementation must make sure that `forward` and `backward` are always "mirror images" of one another.

On the next page is an incomplete pseudocode implementation of bidirectional maps. The `get` and `reverseGet` operations have already been implemented. Your job is to implement `put`! Make sure that your code works on the previous example — there is an important detail which is easy to miss.

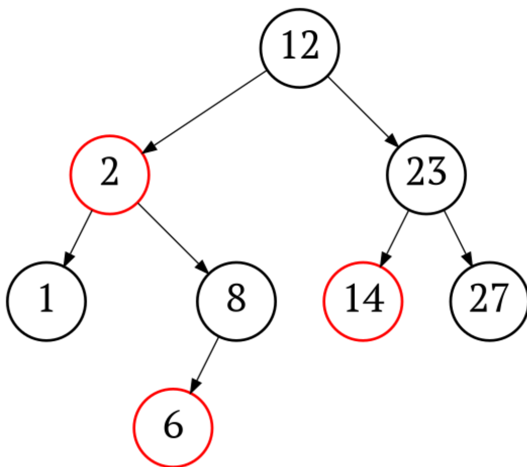
In this code, `forward` and `backward` are implemented as AVL trees. You can use all of the standard map operations on `forward` and `backward`, such as `get`, `put`, `containsKey` and `remove`.

```
class BidirectionalMap[Key, Value]:
    // This class implements a bidirectional map where the keys are
    // objects of class 'Key' and the values are objects of class 'Value'.

    // We implement the maps 'forward' and 'backward' as AVL trees,
    // but only remember that they are maps.
    forward: Map[Key, Value] = empty AVL map
    backward: Map[Value, Key] = empty AVL map

    // Look up the value for a given key.
    // Note on syntax: this declares that 'get' is a method, taking one
    // parameter 'key' of type 'Key', returning an object of type 'Value'.
    get(key: Key) -> Value:
        return forward.get(key)

    // Look up the key for a given value.
    reverseGet(value: Value) -> Key:
        return backward.get(value)

    // Add an entry to the bidirectional map.
    // This overwrites any existing entry with that key or value.
    put(key: Key, value: Value):
        // Your code goes here! Write your answer below,
        // or on a separate piece of paper.
        // You can use Java or Python syntax, or pseudo-code,
        // as long as it is clear enough.
        if forward.containsKey(key):
            oldValue: Value = forward.get(key)
            forward.remove(key)  // optional
            backward.remove(oldValue)

        if backwards.containsKey(value):
            oldKey: Key = backward.get(value)
            forward.remove(oldKey)
            backward.remove(value)  // optional

        // These insertions also effect the removals marked optional above.
        forward.put(key, value)
        backward.put(value, key)
```

Section 3: Search trees

In this section, the key ordering of search trees is the natural ordering of integers.

Question 3A

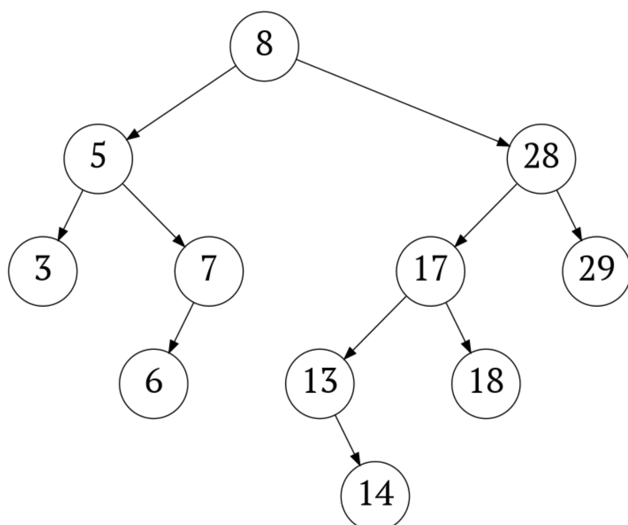


Is this a red-black tree?

☐ Yes.
☑ No.

If not, point out a concrete problem (be specific):

The rightmost branch (12→23→27) contains 3 black nodes, but the other paths contain only 2 black nodes.



Is this an AVL tree?

☐ Yes.
☑ No.

If not, point out a concrete problem (be specific):

The subtree rooted at 28 violates the balancing invariant: its left subtree has height 3, while its right subtree has height 1.

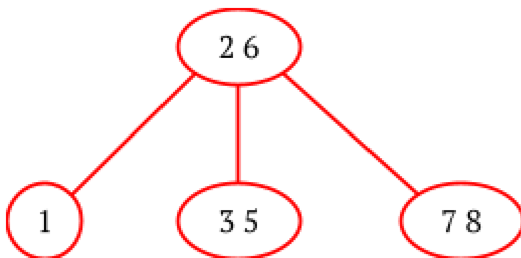
Question 3B

Find a 2-3 tree with the following properties:

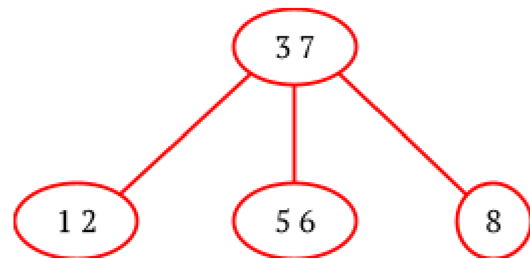
- Its keys are the integers 1–8 except 4. In particular, each integer 1, 2, 3, 5, 6, 7, 8 appears exactly once in the tree.
- Inserting the key 4 will increase the height by one.

Draw the tree before inserting 4:

Alternative 1:

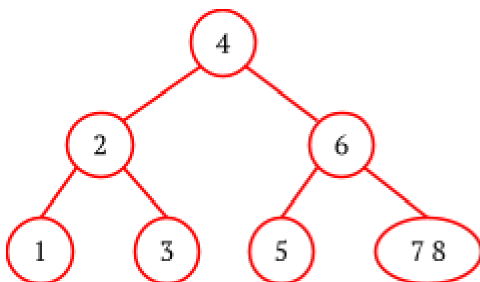


Alternative 2:

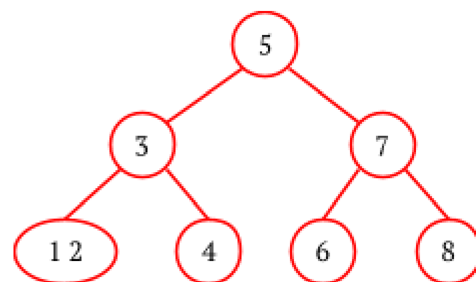


Draw the resulting 2-3 tree after inserting 4:

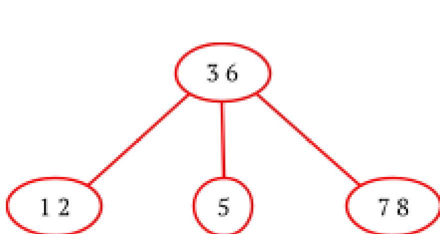
Alternative 1:

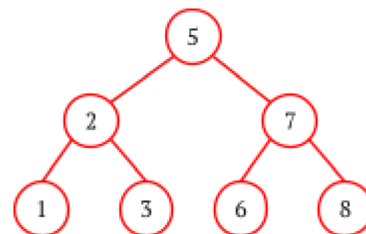


Alternative 2:



The following 2-3 trees are ***not*** solutions because adding 4 will not change the height:





Section 4: Sorting

Question 4A

Let `arr` be an array of `n` integers. The following algorithm is supposed to build a sorted version `sorted` of the array `arr`. Unfortunately, your cat has eaten the delicious parts of the algorithm:

```
sorted = new array of size n
for i from 0 to n:
    minIndex = i
    for j from i+1 to n:
        if arr[j] <= arr[minIndex]:
            minIndex = j
    sorted[i] = arr[minIndex]
    arr[minIndex] = arr[i]
```

(Clarification: A loop `for i from a to b` iterates over the values `a`, `a+1`, ..., `b-1`.)

Repair the algorithm by filling in the missing expressions.

Question 4B

In this question, we consider sorting a list of items by some kind of *key*. For example, we may want to sort a list of strings by their length, so that shorter strings come first. More formally, we have a function *key* whose values we can compare, and we sort the list so that *x* comes before *y* if *key(x)* < *key(y)*. In this example, *key(x)* would be the length of the string *x*.

A key-based sorting algorithm is *stable* if any two objects with the same key have the same relative order before and after sorting. That is, if *x* and *y* have the same key, and *x* comes before *y* in the input list, then *x* comes before *y* in the sorted output list. For example, consider the following list of strings:

> *big, cats, eat, icecream, some, tuesdays*

If we sort these strings by their length, using a stable sorting algorithm, the result will be:

> *big, eat, cats, some, icecream, tuesdays*

Notice that *big* and *eat* both have length 3, and *big* comes before *eat* both before and after sorting. Similarly, *cats* comes before *some*, and *icecream* comes before *tuesdays*. The strings are sorted in order of length, but strings with the same length have the same relative order as they did originally.

The following version of insertion sort takes an array `arr` of size `n` and sorts it by `key`:

```
for i from 0 to n:
    j = i
    while j > 0 and key(arr[j]) <= key(arr[j-1]):
        swap arr[j] and arr[j-1]
        j = j-1
```

Unfortunately, it is not stable.

Your tasks:

a. The list "*big cats...*" above is an example for which stability is violated. Show how the resulting list looks after applying the sorting algorithm above.
b. Change a single line in the pseudocode so that the algorithm becomes stable.
c. Justify why your improved version is stable.

Answer:

(a) *eat, big, some, cat, tuesdays, icecream*
(b) Change <= to < in the condition of the while-loop.
(c) This change makes the insertion stop once it reaches an item having the same key.

Section 5: priority queues, hash tables

Question 5A

Exactly one of the following arrays is **not** a binary min-heap.

A	1	2	1	3	2	2	1	4	3	3
B	1	2	3	4	8	5	6	6	7	7
C	0	3	2	5	4	8	9	7	6	5

Which one is it? Point out a concrete problem with it (be specific).

Array **B** is not a binary min-heap.

Explanation using 0-based indexing: Index 4 has value 8. This is not a lower bound for its left child (index $2 \cdot 4 + 1 = 9$), which has value 7.

Explanation using 1-based indexing: Index 5 has value 8. This is not a lower bound for its left child (index $2 \cdot 5 = 10$), which has value 7.

Question 5B

We consider an open addressing hash table with linear probing and modular compression. The maximum load factor is 75%, and when the table becomes overloaded, its array size grows by 50%.

Initially, the hash table looks like this:

0	1	2	3	4	5	6	7	8	9
18	39		13	63				28	9

Insert the following elements into the table, in this order: 30, 33, 14. Use the numbers themselves as hash values (modulo the array size to get the array index).

Draw the hash table *after each individual insertion*. If the hash table needs to be resized, draw it *both before and after* resizing. (This means that you should draw at least 3 hash tables below.)

Note: A simple way of calculating modulo N is to repeatedly subtract N until you get a value below N.

After inserting 30:

0	1	2	3	4	5	6	7	8	9
18	39	30	13	63				28	9

After inserting 33 and before resizing:

0	1	2	3	4	5	6	7	8	9
18	39	30	13	63	33			28	9

Load factor = 80% > 75%, so we have to resize to 15 cells.
We reinsert all elements using their new indices calculated by modulo 15:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
30			18	63	33				39	9			13	28

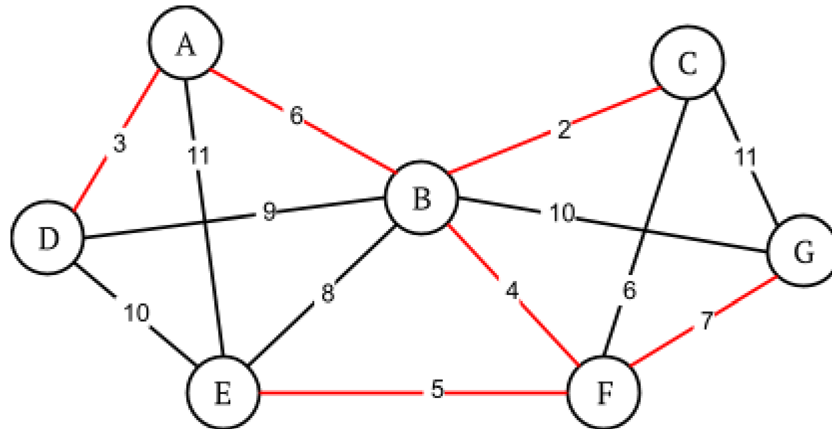
After inserting 14:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
30	14		18	63	33				39	9			13	28

Section 6: Graphs

Question 6A

Perform Kruskal's algorithm on the following weighted undirected graph.



(resulting minimum spanning tree is drawn in red; not needed for answer)

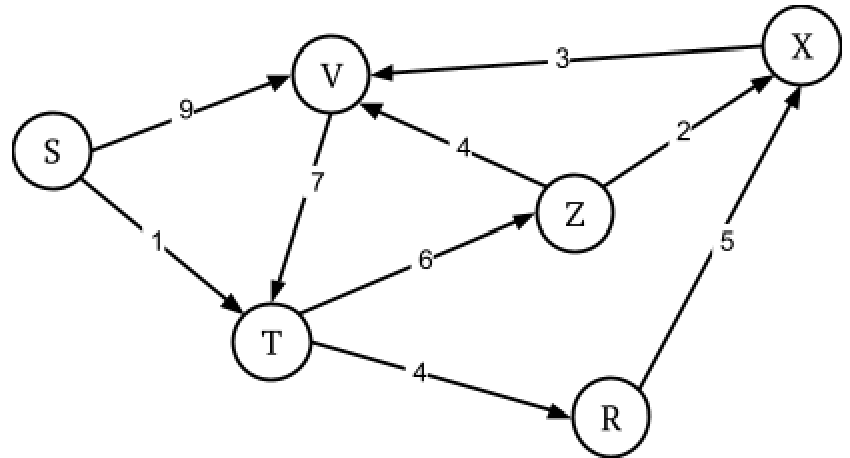
List the edges of the minimum spanning tree in the order they are added:

B—C
A—D
B—F
E—F
A—B
F—G

Note: The table has more rows than needed.

Question 6B

Here is a weighted directed graph:



And here is a trace of a run of Dijkstra's algorithm (UCS) on the graph, starting at node **S**:

Removed node	Added node(s)	Visited nodes	Priority queue after adding node(s)
—	S	—	S:0
S	T, V	S	T:1, V:9
T	R, Z	S, T	R:4, Z:6, V:9
R	X	R, S, T	X:5, Z:6, V:9
X	V	R, S, T, X	V:3, Z:6, V:9
V	T	R, S, T, V, X	Z:6, T:7, V:9
Z	V, X	R, S, T, V, X, Z	X:2, V:4, T:7, V:9
X	—	R, S, T, V, X, Z	V:4, T:7, V:9
V	—	R, S, T, V, X, Z	T:7, V:9
T	—	R, S, T, V, X, Z	V:9
V	—	R, S, T, V, X, Z	—

However, the person implementing the algorithm made some errors. Because of this, the above trace does not produce the correct shortest path tree.

Question 6B (continued)

a. Where in the trace does the algorithm start going wrong?

The first three nodes (S, T, R) are visited correctly, but then it should not visit X, but Z instead. Indeed, the priority of X should be 10, not 5. So the first mistake is in the row for R.

b. What is the problem with the implementation?

The implementation uses as priority only the weight of the edge leading to the node — it should be the length of the whole path from the starting node instead.

c. Show the trace for a corrected implementation:

This is one possibility, but there are other UCS/Dijstra algorithms that are also correct.

Removed node	**Added node(s)**	**Visited nodes**	**Priority queue after adding node(s)**
—	S	—	S:0
S	T, V	S	T:1, V:9
T	R, Z	S, T	R:5, Z:7, V:9
R	X	R, S, T	Z:7, V:9, X:10
Z	V, X	R, S, T, Z	V:9, X:9, X:10, V:11
V[1]	T[2]	R, S, T, V, Z	X:9, X:10, V:11, T:16
X[1]	V[2]	R, S, T, V, X, Z	X:10, V:11, V:12, T:16
X	—	R, S, T, V, X, Z	V:11, V:12, T:16
V	—	R, S, T, V, X, Z	V:12, T:16
T	—	R, S, T, V, X, Z	T:16
V	—	R, S, T, V, X, Z	—

[1] These could be removed in any order

[2] Depending on the algorithm, you don't have to add these nodes since they are already visited