Exam – Datastrukturer

DIT961, VT-22
Göteborgs Universitet, CSE

Day: 2022-06-04, *Time:* 8:30-12.30, *Place:* Johanneberg

Course responsible

Pelle Evensen will visit at around 9:30. Alex Gerdes is available via telephone 031-772 6154 during the entire exam.

Allowed aids

You may bring a dictionary.

Grading

The exam consists of *six questions*. For each question you can get a U, a G or a VG. To get a G on the exam, you need to answer at least *five* questions to G or VG standard. To get a VG on the exam, you need to answer at least three questions to VG standard (and the remaining part of a question to a G standard).

A fully correct answer for a question, including the parts labelled "For a VG", will get a VG. A correct answer, without the "For a VG" parts, will get a G. An answer with minor mistakes might be accepted, but this is at the discretion of the marker. A correct answer on the VG part may compensate a mistake in the G part, also at the discretion of the marker. An answer with large mistakes will get an U.

Inspection

When the exams have been graded they are available for review in the student office on floor 4 in the EDIT building. If you want to discuss the grading, please contact the course responsible and book a meeting. In that case, you should leave the exam in the student office until after the meeting.

Note

– Begin each question on a new page and write your anonymous code (not your name) on every page.

– When a question asks for pseudocode, you don't have to write precise code, such as Java. But your answer should be well structured. Indenting and/or using brackets is a good idea. Apart from being readable your pseudocode should give enough detail that a competent programmer could easily implement the solution, and that it's possible to analyse the time complexity.

– Excessively complicated answers might be rejected.

– *Write legibly!* Solutions that are difficult to read are not evaluated!

Exercise 1 (complexity)

Your task is to describe the worst-case time complexity of the following three code fragments. The complexity should be expressed in terms of n, the size of the input. You should express the complexity in the simplest form possible. All answers need to be well motivated!

a)

```
void fun(int n) {
  for (int i = 12; i < n; i++) {
    /* statement with O(1) time complexity */
  }
}
```

b)

```
void fun(int[] a) {
  for (int i = 0; i < a.length; i++) {
    for (int j = 1; j < a.length; j = j * 2) {
      /* statement with O(n) time complexity */
    }
  }
}
```

c)

```
void fun(int[] a) {
  int i = a.length;
  while (i > 0) {
    for (int j = 0; j < i; ++j) {
      /* statement with O(1) time complexity */
    }
    i--;
  }
}
```

For a VG only:

Consider the following Haskell function that drops all the elements from a list that are not in ascending order (i.e., which are not sorted). The function returns a tuple of lists: the first element in the tuple is a (sorted) list with elements from the input list that were in ascending order, and the second element is a list that contains the remaining items of the input list.

```
dropit :: Ord a => [a] -> ([a], [a])
dropit = go [] []
 where
  go sorted dropped (x:y:ys)
    | x <= y           = go (x:sorted) dropped (y:ys)
    | otherwise        = go sorted (y:dropped) (x:ys)
  go sorted dropped xs = (reverse (xs ++ sorted), dropped)
```

Your task is to write a *recurrence relation* for the `dropit` function and give the corresponding worst-case time complexity. Motivate your answer!

Exercise 2 (sorting)

Perform a quicksort-partitioning of the following array:

9	27	21	25	2	8	20	5	20
0	1	2	3	4	5	6	7	8

Show the resulting array after partitioning it with the following pivot elements:

a) first element

b) middle element (swap the first and middle element before partitioning)

c) last element (swap the last and first element before partitioning)

You should only use the partitioning strategy that we used during the course. Also, highlight which subarrays that need to be sorted using a recursive call. (for example by drawing a line under each subarray)

For a VG only:

Recall the `dropit` function:

```
dropit :: Ord a => [a] -> ([a], [a])
dropit = go [] []
 where
  go sorted dropped (x:y:ys)
    | x <= y           = go (x:sorted) dropped (y:ys)
    | otherwise        = go sorted (y:dropped) (x:ys)
  go sorted dropped xs = (reverse (xs ++ sorted), dropped)
```

Your tasks are:

a) Use the `dropit` function to define a function `sort` that sorts a list of elements. It should have the following type signature:

```
sort :: Ord a => [a] -> [a]
```

You may assume that the `merge` function that merges two sorted lists:

```
merge :: Ord a => [a] -> [a] -> [a]
```

is available and has a linear worst-case time complexity. You can solve the problem in less than 10 lines of code.

b) What is the best- and worst-case input for the `sort` function?

c) What is the the worst-case time complexity of the `sort` function? Motivate your answer!

Exercise 3 (basic data structures: hash table)

Suppose you have the following hash table, implemented using *linear probing*. For the hash function we are using the identity function, $h(x) = x \bmod 9$, where mod is the modulo operator that calculates the remainder of a division.

35	71			4	68	60	70	44
0	1	2	3	4	5	6	7	8

a) In which order could the elements have been added to the hash table? There are several correct answers, and you should choose them all.

A) 44, 35, 68, 71, 60, 70, 4

B) 35, 68, 71, 44, 60, 4, 70

C) 68, 70, 44, 35, 71, 60, 4

D) 68, 71, 44, 35, 60, 70, 4

E) 4, 35, 68, 71, 70, 60, 44

b) Add 79 to the hash table (assume there is no rehashing). What does the hash table look like now?

c) **For a VG only:** Remove 35 from the hash table. What does it look like now? Remember that we use *lazy deletion*.

d) **For a VG only:** After you have removed 35, perform a *rehash* by increasing the size of the array to 13. Notice that the hash function will change to $h(x) = x \bmod 13$. Show the resulting array.

Exercise 4 (heaps)

Which array out of A, B and C represents a binary heap? Only one answer is right.

	0	1	2	3	4	5	6	7	8	9	10
A	5	19	7	69	14	64	58	76	74	81	
B	6	14	12	42	24	55	36	75	61	43	
C	14	48	38	22	58	62	49	70	69	65	

a) Write out that heap as a binary tree.

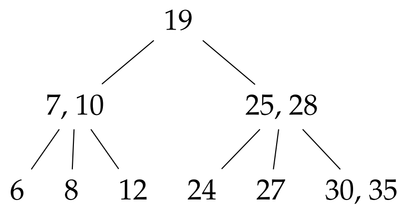
b) Add 10 to the heap, making sure to restore the heap invariant. How does the array look now?

c) **For a VG only:** A binary heap is a complete binary tree that satisfies the heap property. What does 'complete' mean in this context? Why is it important to have a complete tree for a binary heap?

d) **For a VG only:** Describe the procedure how we can build a heap from an arbitrary array. Answer either in pseudocode or English (or Swedish).

Exercise 5 (search trees)

Consider the following 2-3 tree:



Your tasks are:

a) Insert the element 9 into the given tree and show the resulting tree.

b) After you have inserted the element 9, now insert the element 32 into the tree and write down the resulting tree.

c) Convert the given 2-3 tree (without the insertions of 9 and 32) to a corresponding AA-tree. You need to draw the resulting AA-tree in which you annotate every node with its *level*.

d) **For a VG only:**

The following data declaration can be used for defining a 2-3 tree in Haskell:

```
data Tree a
  = Empty
  | Node2 (Tree a) a (Tree a)
  | Node3 (Tree a) a (Tree a) a (Tree a)
  deriving Show
```

Write a function that combines (i.e., folds) all values in a 2-3 tree:

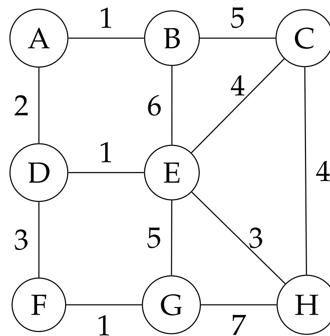
```
foldTree :: (a -> b -> b) -> b -> Tree a -> b
```

The `foldTree` function should traverse the argument 2-3 tree in an *inorder* manner. For example, applying the expression `foldTree (:) []` on the given 2-3 tree above results in the following list:

```
[6,7,8,10,12,19,24,25,27,28,30,35]
```

Exercise 6 (graphs)

You are given the following undirected weighted graph:



For a G you may choose one of the following questions, for a VG you need to answer both.

a) Compute a minimal spanning tree for the following graph by manually performing Prim's algorithm using H as starting node.

Your answer should be the set of edges which are members of the spanning tree you have computed. The edges should be listed in the order they are added as Prim's algorithm is executed. Refer to each edge by the labels of the two nodes that it connects, e.g. DF for the edge between nodes D and F.

b) Suppose we perform Dijkstra's algorithm starting from node A. In which order does the algorithm visit the nodes, and what is the computed distance to each of them? Explain your answer.

Suggested solutions – Exam – Datastrukturer

DIT961, VT-22
Göteborgs Universitet, CSE

Day: 2022-06-04, *Time:* 8:30-12.30, *Place:* Johanneberg

Exercise 1 (complexity)

a) This function is linear $O(n)$, the fact that the loop starts at 12 does not influence the complexity.

b) The body of the inner `for`-loop has given $O(n)$ complexity and it loops $O(\log n)$ times, so the complexity for the entire inner loop is $O(n \log n)$. The outer `for`-loop is linear $O(n)$, which makes the complexity for `fun` $O(n^2 \log n)$.

c) The inner `for`-loop has linear complexity and the outer `while`loop has a linear complexity as well. So the total complexity is quadratic $O(n^2)$.

For a VG only:

For the `dropit` function uses the `go` function for which we can write the following recurrence relation:

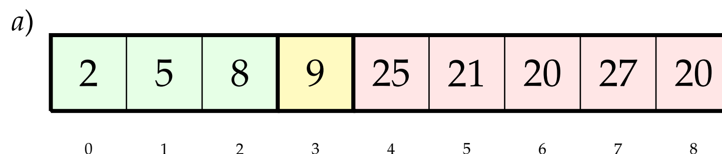
$$T(n) = 1 + T(n-1)$$

due to the fact that at every call the input list, the list `xs` on which the `go` function recurses, is reduced by one. (the other parameters are just accumulating parameters) In the base case we call the operator `(++)` on `xs` and the sorted list. Although the append operator is linear in its first argument this will take constant time, because `xs` contains at most one element. The `reverse` function, which is linear, is on the sorted list, but only once. So the `dropit` function is *linear* $O(n)$.

Exercise 2 (sorting)

It is important that all elements to the left of the pivot are *smaller* and to the right are *larger*, that is, the pivot should be in the right place. The elements in the to be sorted arrays should be in the correct order, reflecting how the quicksort algorithm works. The subarrays next to the pivot should *not* necessarily be sorted.

a)



b)

2	27	21	25	9	8	20	5	20
0	1	2	3	4	5	6	7	8

c)

8	9	5	20	2	20	25	21	27
0	1	2	3	4	5	6	7	8

For a VG only:

```haskell
module DropSort where

import qualified Data.List as List
import Test.QuickCheck

dropit :: Ord a => [a] -> ([a], [a])
dropit = go [] []
 where
  go sorted dropped (x:y:ys)
    | x <= y           = go (x:sorted) dropped (y:ys)
    | otherwise        = go sorted (y:dropped) (x:ys)
  go sorted dropped xs = (reverse (xs ++ sorted), dropped)

merge :: Ord a => [a] -> [a] -> [a]
merge xs []    = xs
merge [] ys    = ys
merge (x:xs) (y:ys)
  | x <= y    = x : merge xs (y:ys)
  | otherwise = y : merge (x:xs) ys

sort :: Ord a => [a] -> [a]
sort [] = []
sort xs = let (ys, zs) = dropit xs in ys `merge` sort zs

prop_model :: [Int] -> Property
prop_model xs = List.sort xs === sort xs
```

(you only need to write the `sort` function)

The best-case input for the `sort` function is when the list is already sorted (or nearly sorted). The `dropit` function will not drop any of the elements and the list of dropped elements will be empty, so `sort` does not need to go in recursion. For sorted lists, the `sort` function is linear $O(n)$.

The worst-case is a reverse-sorted list. The `dropit` function will drop all but the first element. So `sort` function is called recursively on a list that has one less element. This

makes that the `sort` function is quadratic $O(n^2)$.

Exercise 3 (basic data structures)

a) The correct answers are A and C.

b)

35	71	79		4	68	60	70	44
0	1	2	3	4	5	6	7	8

For a VG only:

c)

XX	71	79		4	68	60	70	44
0	1	2	3	4	5	6	7	8

The number 35 is replaced by a ‘deleted’ marker.

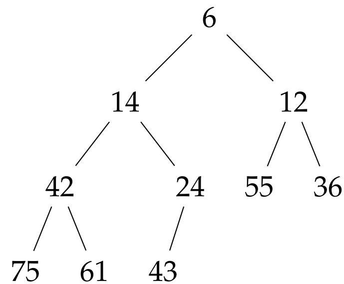
d)

	79		68	4	70	71	44	60				
0	1	2	3	4	5	6	7	8	9	10	11	12

Exercise 4 (heaps)

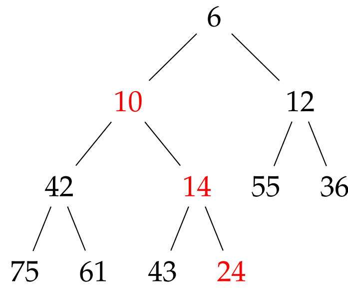
B is the only array that represents a binary heap, which can be drawn as a tree as follows:

A



B

The moved bits and new value are drawn in red.



As array: `[6, 10, 12, 42, 14, 55, 36, 75, 61, 43, 24]`

For a VG only:

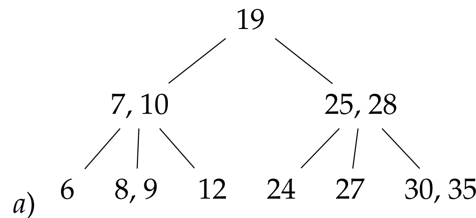
C

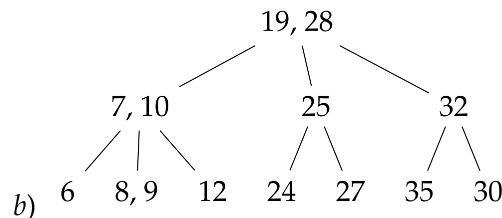
A complete tree means that means that all levels except the bottom one are full, and the bottom level is filled from left to right. This makes sure that the tree is balanced and that a new element has only one place to go. Moreover, we can implement it using an array!

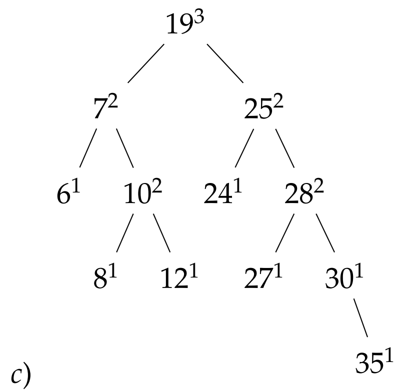
D

The heap property states that each element must be less than its children. If we have an element that violates the heap property, we can restore it by 'sifting down' the element (also called 'sink'). So, we loop through the array and sift down each element in turn. However, when sifting down the children must already be in heap order. To make sure that the children have the heap property we loop through the array in *reverse* order.

Exercise 5 (search trees)







c)

d) **For a VG only:**

```
foldTree :: (a -> b -> b) -> b -> Tree a -> b
foldTree op b t = let go = foldTree op in case t of
  Empty           -> b
  Node2 l x r     -> go (x `op` go b r) l
  Node3 l x m y r -> go (x `op` go (y `op` go b r) m) l
```

Exercise 6 (graphs)

a) $\{EH, DE, AD, AB, DF, FG, CH\}$

the last edge can also be CE

b) A : 0, B : 1, D : 2, E : 3, F : 5, G : 6, C : 6, H : 6

Depending on the priority queue implementation nodes with equal shortest distances may be visited in a different order.