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:

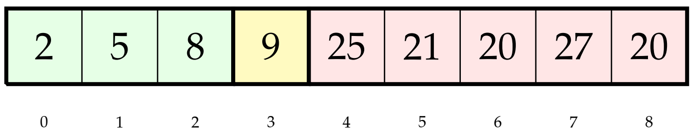
$$T(n) = 1 + T(n-1)$$

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* $O(n)$.

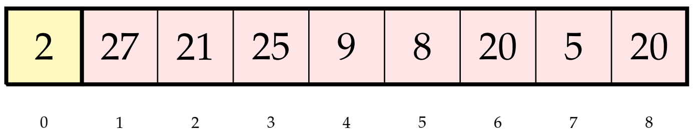
Exercise 2 (sorting)

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.

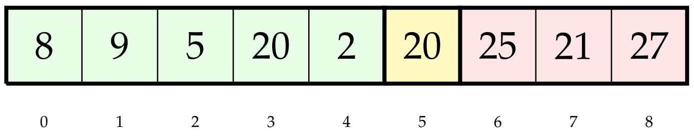
a)



b)



c)



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.

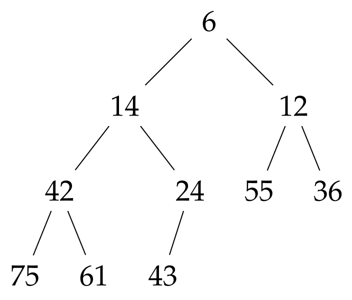
d)

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

Exercise 4 (heaps)

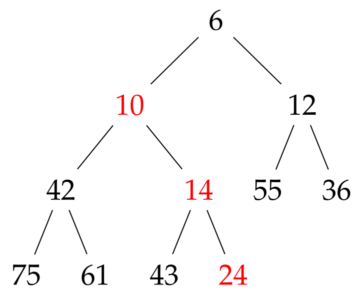
B is the only array that represents a binary heap, which can be drawn as a tree as follows:

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

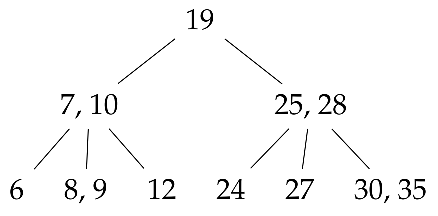
A complete tree means that means that all levels except the bottom one are full, and the bottom level is filled from left to right. This makes sure that the tree is balanced and that a new element has only one place to go. Moreover, we can implement it using an array!

D

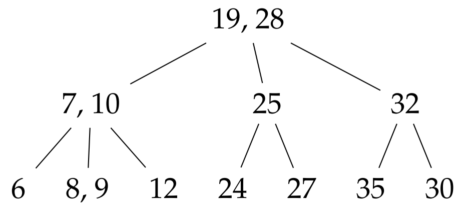
The heap property states that each element must be less than its children. If we have an element that violates the heap property, we can restore it by 'sifting down' the element (also called 'sink'). So, we loop through the array and sift down each element in turn. However, when sifting down the children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

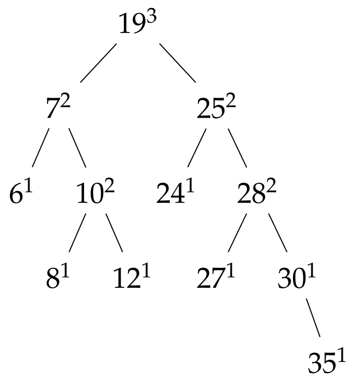
Exercise 5 (search trees)

a)



b)





c)

d) **For a VG only:**

```haskell
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.