Suggested solutions – Exam – Datastrukturer

DIT960 / DIT961, VT-18
Göteborgs Universitet, CSE

Day: 2018-08-24, *Time:* 8:30-12.30, *Place:* J

Exercise 1

The complexity of the `disjoint` function is $O(n \log n)$. Insertion into an AVL tree takes $O(\log n)$, so to build the AVL tree `tree` we call the `insert` function n times, which results in $O(n \log n)$. To create the list of booleans `bs` to check if an element is present in the other list, is $O(n \log n)$ as well, so the overall complexity is $(n \log n)$. The function `null` takes constant time.

a) The insertions take $O(m \log m)$ time. There are n membership tests and the tree will have m elements so that part takes $O(n \log m)$ time. The total is $O((m + n) \log m)$.

b) Using the complexity above, it goes faster if you minimise m (the size of the tree). So case 2 is faster.

Exercise 2

A possible implementation for the two functions:

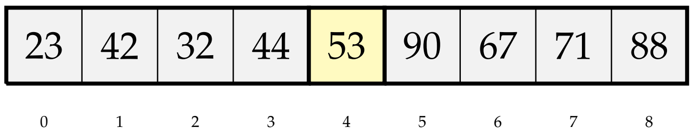
```haskell
module Tree where

data Tree a = Nil | Node a (Tree a) (Tree a) deriving Show

greatest :: Ord a => Tree a -> Maybe a
greatest tree = case tree of
  Nil          -> Nothing
  Node x _ Nil -> Just x
  Node _ _ r   -> greatest r

delete :: Ord a => a -> Tree a -> Tree a
delete _ Nil = Nil
delete x (Node y l r)
  | x < y     = Node y (delete x l) r
  | x > y     = Node y l (delete x r)
  | otherwise = maybe r (\g -> Node g (delMax l) r) $ greatest l
```

```
delMax :: Ord a => Tree a -> Tree a
delMax tree = case tree of
  Nil          -> Nil
  Node _ l Nil -> l
  Node _ _ r   -> delMax r
```

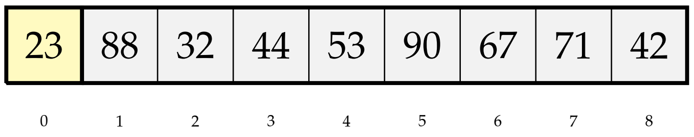
Exercise 3

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.

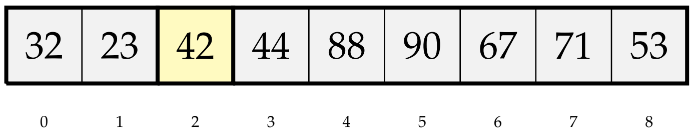
a)



b)



c)



d) It is important that we choose a pivot element that divides the array into subarrays of reasonable size. If we choose for example the smallest element we end up with a subarray that is just one element smaller, which gives rise to quadratic complexity. This can occur in a (nearly) sorted array.

One of the possible precautions is to randomize the array. Another is to take the median of the first, middle, and last element.

e) Both algorithms have an average case runtime of $O(n \log n)$. Quicksort does not preform well on sorted list and has quadratic worst case complexity. Mergesort has high constant factors, due the copying to an auxiliary array, but is not that sensitive to the type of data. Moreover, mergesort is stable, whereas quicksort is not.

Exercise 4

a) AD, DE, EG, AB, BC, CI, CH, DF

b) F : 0, D : 6, E : 7, A : 7, G : 9, B : 10, H : 14, C : 15, I : 16

Exercise 5

a) Because of probing, an element is either stored at the index corresponding to its hash or at a *next* index, which wraps around. The hash function computes the rest of a division with 12. The correct answers are: E (11) and F (36).

b) Answer: no. Strings with the same length will have the same hashcode. If we insert lots of strings with the same length, then lookup will take $O(n)$ time instead of $O(1)$.

(This hash function was apparently used by early versions of PHP!
see: htps:/lwn.net/Articles/577494/)

Exercise 6

For a G

Represent the set as e.g. a balanced tree, such as an AVL tree. Implement the operations as follows:

- `new`: create an empty AVL tree. $O(1)$
- `insert`: use AVL insertion. $O(\log n)$
- `member`: use binary search for a tree. $O(\log n)$
- `delete`: use AVL deletion. $O(\log n)$
- `increaseBy`: traverse the tree and increase every element with the given amount. This doesn't change the relative order of any nodes, so the BST and AVL invariants still hold afterwards.

For a VG

Use an AVL tree plus an integer variable "offset", which stores the total of all calls to `increaseBy`. Implement the operations as follows:

- `new`: create new AVL tree and intialise "offset" to 0. $O(1)$
- `insert`: insert the given value minus "offset" in the tree. $O(\log n)$
- `member`: search for the given value minus "offset". $O(\log n)$

- `delete`: same as for `insert` but do a delete instead of an insert. $O(\log n)$
- `increaseBy`: add given value to ”offset”. $O(1)$