Suggested solutions – Exam – Datastrukturer

DIT960 / DIT961, VT-18
Göteborgs Universitet, CSE

Day: 2017-05-31, *Time:* 8:30-12.30, *Place:* SB-MU

Exercise 1

a) an ordinary binary search tree?

$O(n^2)$ – We do n insert and delete operations, each of which is $O(\text{height of tree})$. In case of an ordinary BST the maximum possible height is $O(n)$, if it is unbalanced.

b) an AVL tree?

$O(n \log n)$ – The insert and delete operations are $O(\log n)$ for an AVL tree, which is balanced.

c) an unsorted linked list?

$O(n^2)$ – Insert is $O(1)$ for an unsorted linked list, so the `for`-loop takes $O(n)$ time. However, `deleteMin` takes $O(n)$ time so the `while`-loop takes $O(n^2)$ time.

d) a 2-3 tree?

$O(n \log n)$ – Same reasoning as for AVL trees.

e) a sorted array?

$O(n^2)$ – Insert takes $O(n)$ time so the `for`-loop takes $O(n^2)$ time. Deleting an element takes $O(n)$, if we delete the first element, so that means that the `while`-loop is $O(n^2)$ as well. If we store the elements in *reverse* order then the deletion would take $O(1)$ and the `while`-loop becomes $O(n)$, but the entire method would still be $O(n^2)$ due to the cost of `insert`.

Exercise 2

Adapting the example code, to store both keys and values in the nodes:

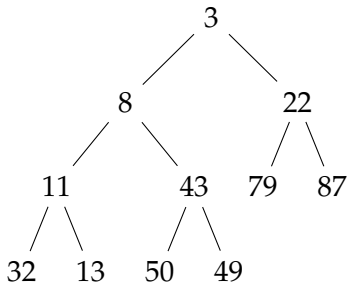
```
module Map where

import Prelude hiding (lookup)
```

```haskell
data Map k v
  = Empty
  | Node k v (Map k v) (Map k v)
  deriving Show

lookup :: Ord k => k -> Map k v -> Maybe v
lookup _ Empty = Nothing
lookup k (Node k' v l r)
  | k < k'    = lookup k l
  | k > k'    = lookup k r
  | otherwise = Just v

update :: Ord k => k -> v -> Map k v -> Map k v
update k v Empty = Node k v Empty Empty
update k v (Node k' v' l r)
  | k < k'    = Node k' v' (update k v l) r
  | k > k'    = Node k' v' l (update k v r)
  | otherwise = Node k v l r

-- O(log n), every case only considers one subtree, if balanced
delete :: Ord k => k -> Map k v -> Map k v
delete _ Empty = Empty
delete k (Node k' v l r)
  | k < k'    = Node k' v (delete k l) r
  | k > k'    = Node k' v l (delete k r)
  | otherwise = maybe l liftMin $ deleteMin r
 where
  liftMin (mk, mv, r') = Node mk mv l r'

-- O(log n), recurrence relation: T(n) = O(1) + T(n/2), if balanced
deleteMin :: Map k v -> Maybe (k, v, Map k v)
deleteMin Empty              = Nothing
deleteMin (Node k v Empty r) = Just (k, v, r)
deleteMin (Node k v l r)     = do
  (k', v', l') <- deleteMin l
  return (k', v', Node k v l' r)

empty :: Map k v -> Bool
empty Empty = True
emtpy _     = False
```

Exercise 3

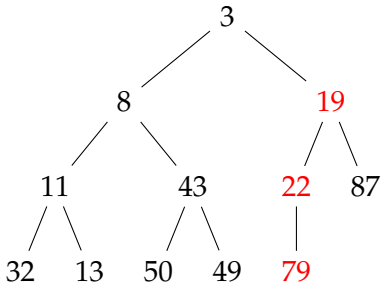
C is the only array that represents a binary heap, which can be drawn as a tree as follows:

A



B

The moved bits and new value are drawn in red.



As array: `[3, 8, 19, 11, 43, 22, 87, 32, 13, 50, 49, 79]`

C

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 4

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

b) F : 0, D : 5, G : 6, A : 7, E : 8, B : 11, C : 15, H : 16, I : 18

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. So the deleted value's hash could have been 0, 1 or 9. The correct answers are B (9), C (20), E (41).

The exercise mistakenly named index 6, so giving only A (26) is also correct.

b) One example: find the smallest element in the hash table. (Or get the elements in increasing order.)

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. $O(\log n)$
- `deleteLarge`: Rebuild the tree with strings smaller than 43: create a new empty tree, remove an element from the original tree, and add it to the new tree. Do this for all elements. $O(n \log n)$

  Alternatively, create an `iterator` instance and loop over all strings and call `delete`.

  (You cannot traverse the tree (which would be $O(n)$) and delete a string, because you need to rebalance the tree before continuing, if not you will probably return a tree which violates the AVL invariant.)

For a VG

Represent the set as *two* AVL trees, one called `smallStrings` that holds the strings smaller than 43, and one called `largeStrings` that stores the other strings. Implement the operations as follows:

- `new`: initialise `smallStrings` and `largeStrings` to empty trees. $O(1)$
- `insert`: use AVL insertion to insert the string into `largeStrings` if it is larger than 42, otherwise insert it into `smallStrings`. $O(\log n)$
- `member`: perform a binary search on the appropriate tree, based on the length of the string. $O(\log n)$

- `delete`: same as for `insert` but do a delete instead of an insert. $O(\log n)$
- `deleteLarge`: assign `largeStrings` to an empty tree. $O(1)$