Exam (with answers)
Data structures DAT036/DAT037/DIT960

Time	Thursday 20th August 2015, 14:00–18:00
Place	Maskinhuset
Course responsible	Nick Smallbone, tel. 0707 183062

The exam consists of **six questions**.

For a **3** (Chalmers) or **G** (GU), you need to answer **three** questions correctly. You can ignore any parts labelled **"For a 4"** or **"For a 5/VG"**.

For a **4** (Chalmers only), you need to answer **four** questions correctly. You must also answer all parts labelled **"For a 4"** in those questions but can ignore any parts labelled **"For a 5/VG"**.

For a **5** (Chalmers) or **VG** (GU), you need to answer **five** questions correctly. You must also answer all parts labelled **"For a 4/5/VG"** in those questions.

For an answer to be considered correct, it must contain no major mistakes. Minor mistakes might be accepted, but this is at the discretion of the marker.

When a question asks for **pseudocode**, you can use a mixture of English and programming notation to describe your solution, and should give enough detail that a competent programmer could easily implement your solution.

Allowed aids One A4 piece of paper of hand-written notes, which should be handed in after the exam. You may write on both sides.

You may also bring a dictionary.

Note Begin each question on a new page.

Write your anonymous code (*not* your name) on every page.

Good luck!

1. Consider the following algorithm for sorting an array a with the help of a priority queue.

```
q = new priority queue
for every element x in a
  q.insert(x)
i = 0
while q is not empty
  a[i] = q.findMinimum()
  q.deleteMinimum()
  i = i+1
```

Assuming that n is the length of the input array, what is the big-O complexity of this algorithm, if the priority queue is implemented using:

a) an unsorted array,

$O(n^2)$. Insert takes (amortised) $O(1)$ time so the for-loop takes $O(n)$ time. But findMinimum and deleteMinimum take $O(n)$ time so the while-loop takes $O(n^2)$ time.

b) a binary heap,

$O(n \log n)$. Insert takes (amortised) $O(\log n)$ so the for-loop takes $O(n \log n)$ time. DeleteMinimum also takes $O(\log n)$ so the while-loop takes $O(n \log n)$ time.

c) **For 4/5/VG only:** a sorted array?

$O(n^2)$. Insert takes $O(n)$ time so the for-loop takes $O(n^2)$ time.

2. Consider the following hash table implemented using *linear probing*, where the hash function is the identity, $h(x) = x$ (modulo 10).

0	1	2	3	4	5	6	7	8	9
10	1	11		14	5	XXX	7	18	

a) The value that was previously at index 6 has been deleted, which is represented by the **XXX** in the hash table.

Which value might have been stored there, before it was deleted? There may be several correct answers, and you should write down **all** of them.

A) 26 **B)** 2 **C)** 17 **D)** 24 **E)** 6 **F)** 13

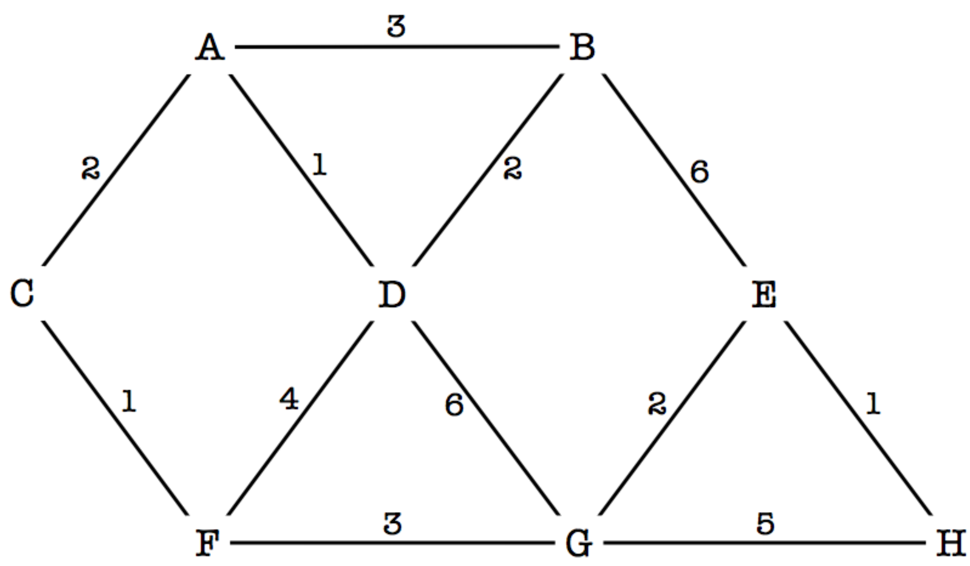
Because of probing, an element is either stored at the index corresponding to its hash or a *greater* index. So the deleted value's hash could have been 4, 5 or 6. The correct answers are A (26), D (24), E (6).

b) **For 4/5/VG only:**
Hash tables typically have better performance than balanced binary search trees. Even so, both are widely used in practice. One reason is that a hash table does not support all the operations that a BST does.

Give an example of an operation which can be efficiently implemented for a binary search tree but not for a hash table.

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

3. You are given the following weighted graph:



a) Suppose we perform Dijkstra's algorithm starting from node **D**. In which order does the algorithm visit the nodes, and what is the computed distance to each of them?

Node	D	A	B	C	F	G	E	H
Distance	0	1	2	3	4	6	8	9

b) In its basic form, Dijkstra's algorithm computes the *distance* from one node to all other nodes, but it doesn't tell you the shortest path.

Suppose you have run Dijkstra's algorithm on a graph, in other words you know the shortest distance from a node x to all other nodes in the graph. How can you use this information to find the shortest path from x to another node y?

Answer either in pseudocode or English. If you want you can use finding the shortest path from **D** to **H** as an example – but your answer should make it clear how to do this for any graph.

The idea: work out which node is immediately before y on the shortest path.

To do this we look at all nodes z that have an edge to y. Node z is immediately before y on the shortest path if: distance from x to z + length of edge from z to y = distance from x to y. Having found z, we recursively find the shortest path to z, and then follow the edge from z to y.

For example, to go from D to H (distance 9), we look at nodes E and G:

- **Distance from D to E + length of edge from E to H = 8 + 1 = 9**
- **Distance from D to G + length of edge from G to H = 6 + 5 = 11**

So we should take the shortest path to E (which we find by recursion), and then follow the edge from E to H.

4. In this question you should design a data structure that is almost a priority queue, except that you can find and remove the *second-smallest* element. It should support the following operations:

   - `new()`: create a new, empty priority queue
   - `insert(x)`: add an integer `x` to the priority queue
   - `findSecondSmallest()`: return the second-smallest element
   - `deleteSecondSmallest()`: remove the second-smallest element

   You may freely use standard data structures and algorithms from the course in your solution, without explaining how they are implemented.

   You should say what design or existing data structure you have chosen, and give **pseudocode** for each of the operations – you don't need to write fully detailed Java code.

   The operations must have the following time complexities:

   - **For a 3/G:**
     O(1) for `new`,
     O(log n) for `insert`,
     O(log n) for `findSecondSmallest`
     O(log n) for `deleteSecondSmallest`
     (where n is the number of elements in the priority queue)

     One way is to use a priority queue and simply remove the smallest element whenever you want to look at the second smallest (remembering to put it back afterwards). So `findSecondSmallest` would go:

     ```
     x = deleteMinimum()
     y = findMinimum()
     insert(x)
     return y
     ```

     and `deleteSecondSmallest` would go:

```
x = deleteMinimum()
deleteMinimum()
insert(x)
```

- **For a 4:**
  as for G but the complexity of `findSecondSmallest` must be O(1).

  Use a priority queue and store *all the elements except the smallest* in it. Store the smallest element in a separate variable. If the queue is called `q` and the variable is called `min`, then `findSecondSmallest` is simply `q.findMinimum()` and `deleteSecondSmallest` is `q.deleteMinimum()`. For `insert(x)` you have to check if the inserted element is the smallest element or not:

  ```
  if x < min then
    q.insert(min)
    min = x
  else
    q.insert(x)
  ```

For a 5/VG (**you may wish to answer the earlier part first**)**:**
Generalise your data structure so that it supports finding and removing the *k*th-smallest element, where *k* is a constant chosen by the user when the data structure is created. The time complexity should be:

O(1) for `new` and `findKthSmallest`;
O(log n) for `insert` and `deleteKthSmallest`.

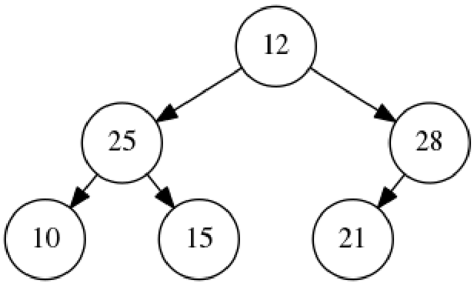
Hint for a 5/VG: it might help to store the *k* smallest elements separately from the rest.

The idea is similar to the one for a 4. We have a min heap `q` for all the elements except the *k-1* smallest ones. We store the *k-1* smallest elements in a *max* heap `smallest`. `findKthSmallest` is `q.findMinimum` and `deleteKthSmallest` is `q.deleteMinimum`. For `insert(x)`, we have to check if the new element is one of the *k-1* smallest or not:

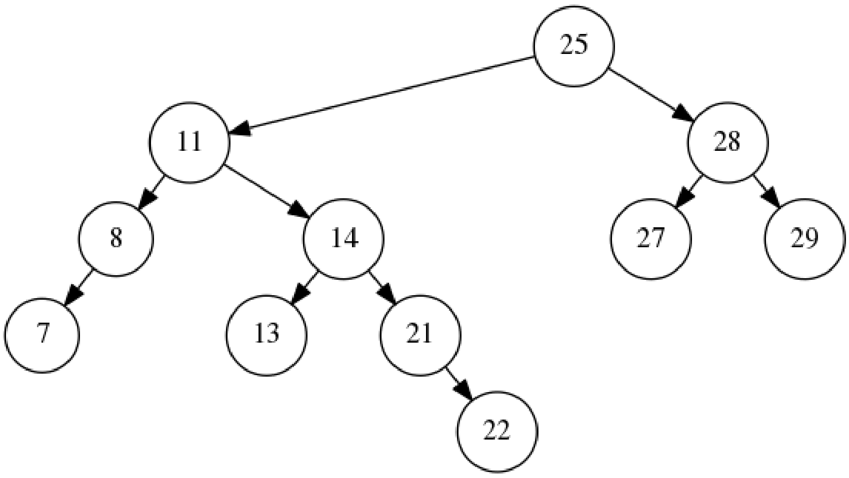
```
if smallest.size() < k then
  smallest.insert(x)
else if smallest.size() = k and x < smallest.findMaximum() then
  q.insert(smallest.deleteMaximum())
else
  q.insert(x)
```

5. Have a look at the following three binary trees.

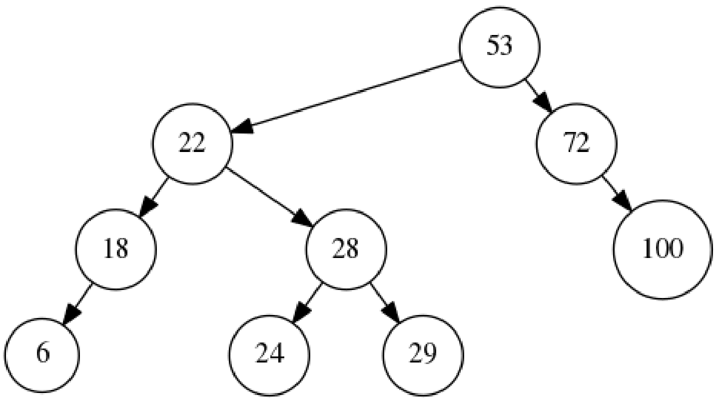
Tree A:



Tree B:



Tree C:

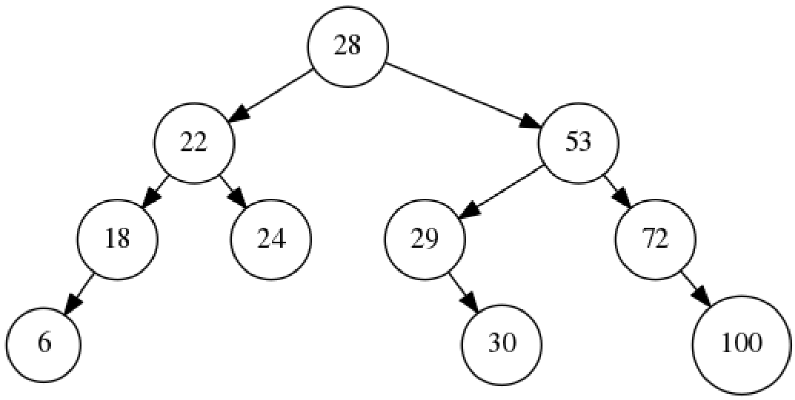


a) One of these trees is an AVL tree. Which one?

C is an AVL tree. A is not a binary search tree (25 > 12) and B is not balanced enough (height of left child of root = 4, height of right child of root = 2)

b) Insert 30 into the tree using the AVL insertion algorithm. Write down the final tree.

After inserting 30 using BST insert, the tree is unbalanced (root's left child height = 4, right child height = 2). It's a left-right tree so a double rotation fixes it:



6. Suppose we are given the following type of binary search trees in Haskell:

```
data Tree a = Nil | Node a (Tree a) (Tree a)
```

a) Implement a Haskell function

```
deleteGreater :: Ord a => a -> Tree a -> Tree a
```

which takes an element `x` and a tree `t`, and returns `t` with all elements greater than or equal to `x` removed. For example on a tree `t` containing 1, 2, 3, 4, 5, `deleteGreater 3 t` should return a tree containing 1 and 2.

The complexity of your function should be O(height of tree), i.e., O(log n) for balanced trees, O(n) for unbalanced trees.

Hint: the correct solution is very small – mine is four lines of code. Try drawing some example trees and look for the general pattern of what is deleted. Think of what happens with a single node – which parts should be kept and which parts should be deleted.

Suppose we call `deleteGreater x t`. If the root node's value is ≥ x then the entire right subtree must also be ≥ x. We should therefore delete the right subtree and the root and recurse into the left subtree. If the root's value is < x then the left subtree must also be < x. So we should keep the root and the left subtree and recurse into the right subtree.

```
deleteGreater _ Nil = Nil
deleteGreater x (Node y l r)
  | y ≥ x = deleteGreater x l
  | otherwise = Node y l (deleteGreater x r)
```

b) **For 5/VG only:**
Does your function work on AVL trees, i.e., does it preserve the AVL invariant? If yes, explain why; if no, give an example that shows why not.

No. Take tree C from q5. If we do **`deleteGreater 29 t`**, then the entire right subtree is deleted and the tree is no longer balanced.