Exam
Data structures DIT960/DAT036

Time	Monday 26th May 2014, 14:00–18:00
Place	Hörsalsvägen
Course responsible	Nick Smallbone, tel. 0707 183062

The exam consists of **six questions**. Some of the questions have parts marked **for VG**, which you should only answer if you are trying for a VG.

For *Godkänd* you need to answer at least **three questions** correctly.
You do not need to answer any VG-only parts.

For *Väl Godkänd* you need to answer at least **five questions** correctly.
You must also answer the VG-only parts of those questions.

For Chalmers students: a G corresponds to a 3, and a VG to a 5. To get a 4, you must answer four questions correctly, including any VG-only parts.

Allowed aids One A4 piece of paper of hand-written notes.
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.

Excessively complicated answers might be rejected.

Write legibly!

1. Consider the following algorithm that sorts a list of n elements by using a binary search tree:

   ```
   initialise t to be an empty binary search tree
   for each element x in the list,
       add x to t
   while t is not empty,
       remove and print the smallest element of t
   ```

   What is the worst-case time complexity of this algorithm, if the tree is implemented using:

   a) an ordinary binary search tree?

   b) an AVL tree?

   You may assume that printing out a value takes constant time.

   Recall that you can find the smallest value in a binary search tree by starting at the root and following the left child until you find a node without a left child – this is the node with the smallest value.

2. Your task is to implement a *map* from keys to values in Haskell using a binary search tree. Your solution should define a type `BSTMap k v` that represents a map from keys `k` to values `v`, together with two functions:

   ```
   data BSTMap k v = …
   lookup :: Ord k => k -> BSTMap k v -> Maybe v
   update :: Ord k => k -> v -> BSTMap k v -> BSTMap k v
   ```

   The `lookup` function looks up a key in the map, while `update` adds a key/value pair to the map, or updates the value if the key already exists in the map.

   You may like to start from the following Haskell code which implements a *set* using a binary search tree.

   ```
   data BST a = Empty
              | Node a (BST a) (BST a)

   member :: Ord a => a -> BST a -> Bool
   member x Empty = False
   member x (Node a left right)
     | x < a = member x left
     | x > a = member x right
     | otherwise = True

   insert :: Ord a => a -> BST a -> BST a
   insert x Empty = Node x Empty Empty
   insert x (Node a left right)
     | x < a = Node a (insert x left) right
     | x > a = Node a left (insert x right)
     | otherwise = Node x left right
   ```

3. Design a data structure for storing a set of integers. It should support the following operations:

   - `new`: create a new, empty set
   - `insert`: add an integer to the set
   - `member`: test if a given integer is in the set
   - `delete`: delete an integer from the set
   - `deleteOdd`: delete **all odd numbers** from the set

   You may use existing standard data structures as part of your solution – you don't have to start from scratch.

   Write down the data structure or design you have chosen, plus **pseudocode** showing how the operations would be implemented. The operations must have the following time complexities:

   - **For G:**
     O(1) for `new`,
     O(log n) for `insert/member/delete`,
     O(n log n) for `deleteOdd`
     (where n is the number of elements in the set)
   - **For VG:**
     as for G but `deleteOdd` must take O(1) time

4. Suppose you have the following hash table, implemented using *linear probing*. The hash function we are using is the identity function, *h(x)* = *x*.

0	1	2	3	4	5	6	7	8
9	18		12	3	14	4	21	

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

A 9, 14, 4, 18, 12, 3, 21

B 12, 3, 14, 18, 4, 9, 21

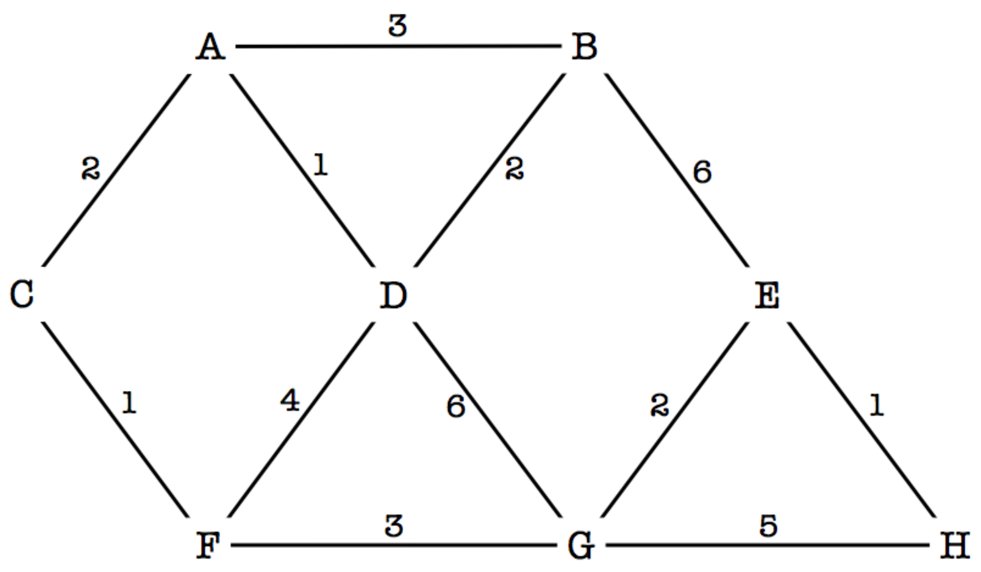
C 12, 14, 3, 9, 4, 18, 21

D 9, 12, 14, 3, 4, 21, 18

E 12, 9, 18, 3, 14, 21, 4

b) Remove 3 from the hash table, and write down how it looks afterwards.

5. You are given the following weighted graph:



a) Suppose we perform Dijkstra's algorithm starting from node **H**. In which order does the algorithm visit the nodes, and what is the computed distance to each of them? There are several possible orders the algorithm might visit the nodes in – you may choose any of them.

b) Use Prim's algorithm to construct a minimum spanning tree for the graph, starting from a node of your choice, and draw the tree.

6. A *double-ended priority queue* is a priority queue that supports removing both the minimum and the maximum element. It provides the following operations:

   ◦ `insert` – add an element to the priority queue

   ◦ `findMin/deleteMin` – find or delete the minimum element

   ◦ `findMax/deleteMax` – find or delete the maximum element

   While writing a program, you discover you need a double-ended priority queue. Your friend suggests a way to implement one:

   > Maintain two priority queues, one of them a min heap and the other a max heap[1]. To insert an item, insert it into both heaps. To implement `findMin/deleteMin` simply call `findMin/deleteMin` on the min heap. To implement `findMax/deleteMax` call `findMax/deleteMax` on the max heap.

   The following Java code illustrates the idea:

   ```
   MinHeap minheap = new MinHeap();
   MaxHeap maxheap = new MaxHeap();
   void insert(E x) { minheap.insert(x); maxheap.insert(x); }
   E findMin() { return minheap.findMin(); }
   E findMax() { return maxheap.findMax(); }
   void deleteMin() { minheap.deleteMin(); }
   void deleteMax() { maxheap.deleteMax(); }
   ```

   Unfortunately, this idea does not work. Once you see why, write down an example where this design would give the wrong answer.

   For VG:

   A *min-max heap* is a binary tree with the following invariant:

   - The value of any node at an *even* level in the tree is *less than or equal to* all values in the node's subtree;
   - the value of any node at an *odd* level in the tree is *greater than or equal to* all values in the node's subtree.

1 Recall that a max heap supports the operations `insert`, `findMax` and `deleteMax`. A min heap is an ordinary binary heap and supports `insert`, `findMin` and `deleteMin`.

The *level* of a node is defined as follows: the root of the tree is at level 0, its children are at level 1, its grandchildren are at level 2, and so on.

Describe how to find the minimum and maximum elements in a min-max heap. Make sure you consider the case where the heap has one element. You do not have to worry about insertion or deletion, only `findMin`/`findMax`.