Exam
Data structures DIT960

Time	Thursday 30th May 2013, 14:00–18:00
Place	V-huset
Course responsible	Nick Smallbone, tel. 0707 183062

The exam consists of **six easy questions (nos. 1–6), two hard questions (nos. 7–8)** and **one extra hard question (no. 9), which counts as two hard questions.**

For *Godkänd* you need to answer at least **four easy questions** correctly.

For *Väl Godkänd* you *also* need to answer **two hard questions** correctly. Answering question 9 correctly counts as two hard questions; a partial answer to question 9 may count as one hard question.

Allowed aids One A4 piece of paper of notes, which may be hand-written or typed. 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. Suppose you have the following hash table, implemented using *linear probing*. For the hash function we are using 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 choose them all.

- ☐ 9, 14, 4, 18, 12, 3, 21
- ☐ 12, 3, 14, 18, 4, 9, 21
- ☐ 12, 14, 3, 9, 4, 18, 21
- ☐ 9, 12, 14, 3, 4, 21, 18
- ☐ 12, 9, 18, 3, 14, 21, 4

b) Add 30 to the hash table (assume there is no rehashing). What does the hash table look like now?

0	1	2	3	4	5	6	7	8

c) Remove 3 from the hash table. What does it look like now?

0	1	2	3	4	5	6	7	8

2. Implement binary search in Java. Your method should have the following type:

   ```
   int search(Object[] array, Object key)
   ```

   It should return the *index* of the object, or -1 if the object could not be found. It should take O(log n) time, and must not use recursion, or any Java standard library methods.

   Warning: there are many different ways to write a *wrong* binary search! To pass this question, your code must be correct. *Make extra sure that you cannot get into an infinite loop*. You do not, however, have to consider integer overflow.

   Answer on a separate sheet of paper. You may want to start from the following skeleton program:

   ```
   int search(Object[] array, Object key) {
      int low = 0;
      int high = array.length - 1;
      while(...) {
         int mid = ...;
         if (array[mid].compare(key) < 0) {
            // array[mid] < key
            ...
         } else if (array[mid].compare(key) > 0) {
            // array[mid] > key
            ...
         } else {
            // array[mid] == key
            ...
         }
      }
      ...
   }
   ```

3. Which array out of A, B and C represents a binary heap? Only one answer is right.

☐ A =

0	1	2	3	4	5	6	7	8	9	10	11
1	4	54	8	45	76	65	44	11	47	57	

☐ B =

0	1	2	3	4	5	6	7	8	9	10	11
1	8	45	13	43	47	44	65	23	76	57	

☐ C =

0	1	2	3	4	5	6	7	8	9	10	11
1	13	47	23	65	54	67	45	32	76	57	

a) Write the heap out as a binary tree.

b) Add 46 to the heap, making sure to restore the heap invariant. How does the array look now?

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

4. Perform a *selection sort* on the following array. Your sort must be in-place, so you must not use a temporary array.

0	1	2	3	4	5	6	7	8
53	71	32	67	90	23	44	88	42

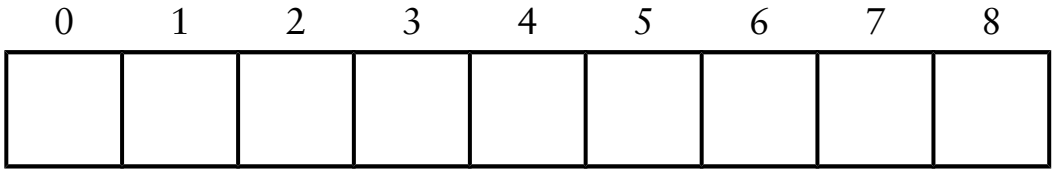
Show how the array looks after each execution of the outer loop. Also show which part of the array is sorted, for example by underlining it.

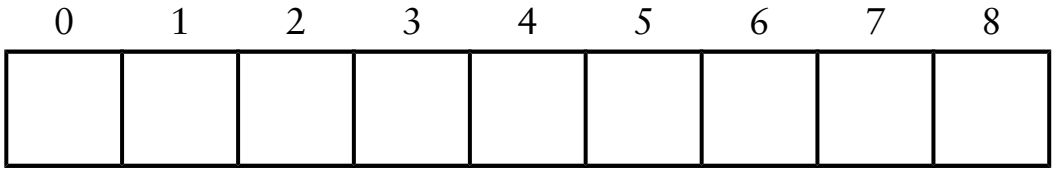
0	1	2	3	4	5	6	7	8

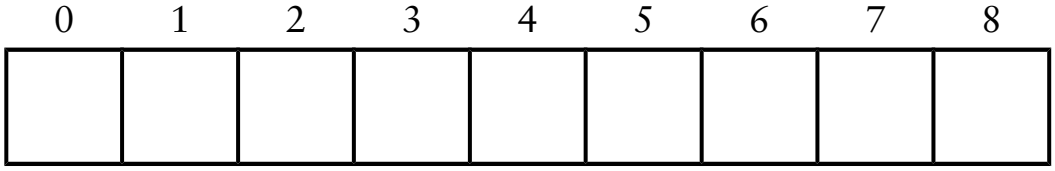
0	1	2	3	4	5	6	7	8

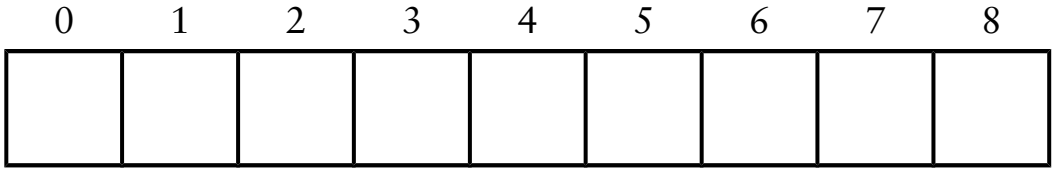
0	1	2	3	4	5	6	7	8

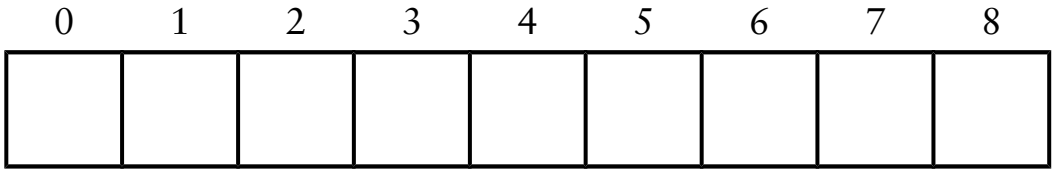
0	1	2	3	4	5	6	7	8


0 1 2 3 4 5 6 7 8

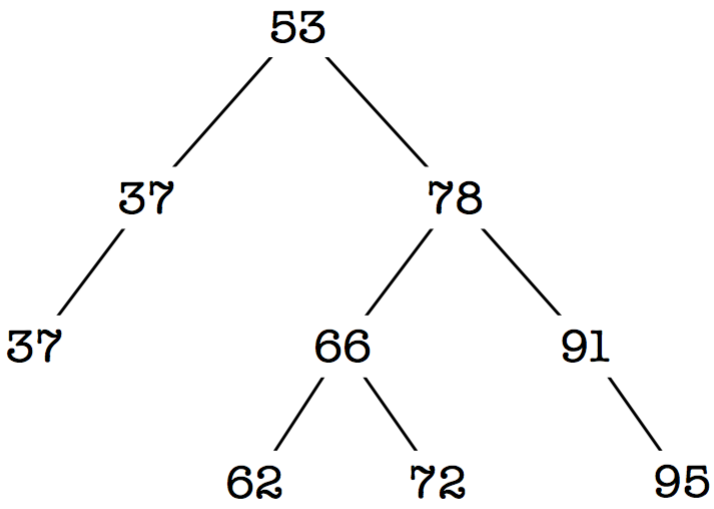

0 1 2 3 4 5 6 7 8


0 1 2 3 4 5 6 7 8


0 1 2 3 4 5 6 7 8


0 1 2 3 4 5 6 7 8

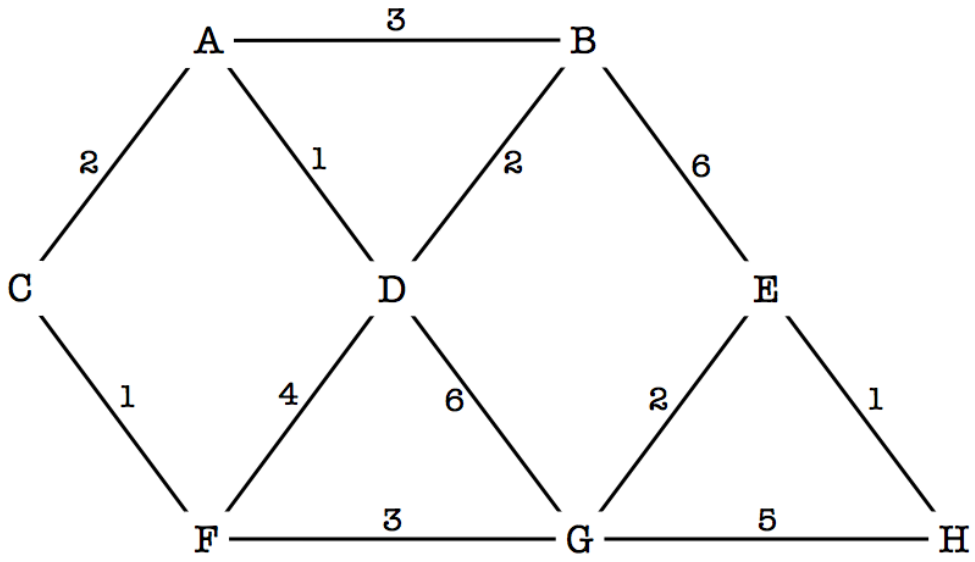
5. You are given the following AVL tree.



a) Mark each node with its AVL balance (left height minus right height).

b) Insert 60 into the tree, and balance it to restore the invariant. Write down the final tree.

6. You are given the following weighted graph:



a) Perform Dijkstra's algorithm starting from node **A**. At each step the algorithm visits a new node. In which order are the nodes visited, and what is the distance to each of them? There are several possible orders to visit the nodes in – you may choose any of them.

Node	A							
Distance	0							

b) What is the shortest path to node **H**? List all the nodes along the way.

c) Use Prim's algorithm to construct a minimum spanning tree for the graph, starting from whichever node you like. Draw the tree below.

A B

C D E

F G H

7. (**hard**) Your job is to implement a queue using a circular array. Your queue should implement the interface `Queue<E>` shown below, which is based on the Java queue interface.

```
public interface Queue<E> {
  // Inserts the specified element at the back of the queue.
  void add(E e);

  // Retrieves and removes an element from the front of the
  // queue, or returns null if the queue is empty.
  E poll();

  // Returns the number of elements in the queue.
  int size();
}
```

Note that:

- Your queue should not have any capacity restrictions.
- You should include a constructor for your queue.
- You may not use Java standard library functions in your solution.

Hint: you may find it useful to define a separate method `void resize()` that doubles the size of the array.

Answer on a separate sheet of paper.

8. (**hard**) 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 function

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

that returns the greatest element in a non-empty binary search tree (in an empty binary search tree it is allowed to crash).

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

b) Write a function to delete an element. It will take two parameters, which are the element to delete and the tree, and have the following type:

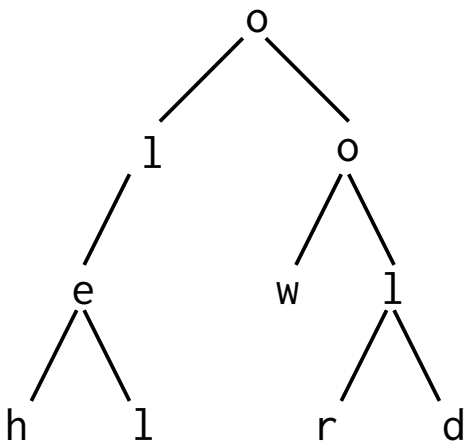
```
delete :: Ord a => a -> Tree a -> Tree a
```

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: it will help to use `greatest` when implementing `delete`.

Answer on a separate sheet of paper.

9. (**extra hard, worth two hard questions**)
Suppose we want to add an array-like indexing operation to a binary tree. The idea is that `get(i)` will return the `i`th element that would be traversed in an in-order traversal of the tree. For example, in the tree to the right, `get(0)` would return `h`, `get(1)` would return `e`, `get(2)` would return `l`, `get(3)` would return `l`, `get(4)` would return `o`, and so on.



Your job is to efficiently implement `get`. Your implementation should have complexity O(height of tree), i.e. O(log n) for balanced trees, O(n) for unbalanced trees. You will have to work out by yourself how to implement it. If you get part of the way, write down what you've found: you may get partial credit for it.

Here are some hints to help you:

- `get` should be a recursive function. It will either recurse into the left or right child, or return the value of the current node. First try to work out the *base case*: when is the current node the one we are looking for?

- To implement `get` efficiently, you will need to know the *size of each node*. Can you think how it helps to know the size of each node? Think especially about the size of the left child.

- If you are stuck, take the tree above, label each node with its size and index and look for a pattern – something that tells you which child to recurse into and with what index. Look at what happens with `get(3)`, `get(4)` and `get(5)`.

- Try writing down pseudocode once you have the basic idea.

You may choose either Java or Haskell. For Java, you should start from the following tree datatype (which records the size of each node) and skeleton implementation:

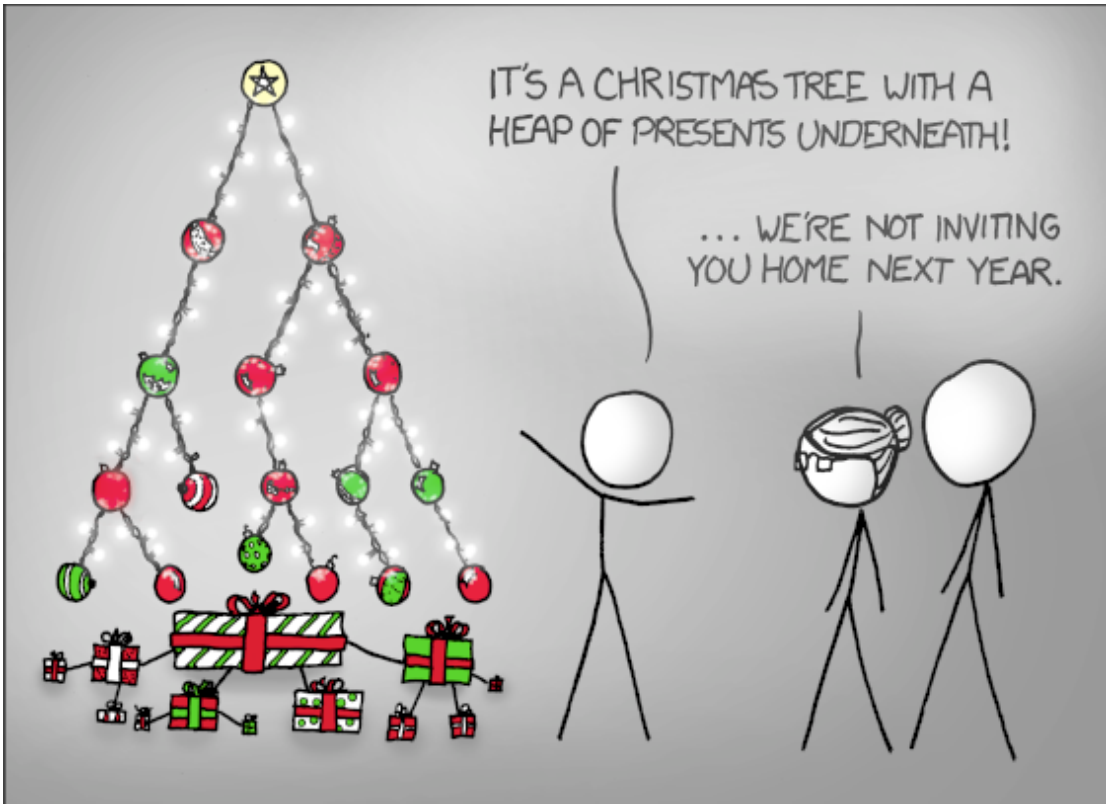
```
class Tree<E> {
 E value;
 int size; // The size of the node
 Tree<E> left, right;

 E get(int index) { /* your code goes here */ }
}
```

For Haskell, you should use the following datatype (which records the size of each node) and skeleton implementation:

```
data Tree a = Nil | Node Int a (Tree a) (Tree a)
 -- The 'Int' field is the size of the node
get :: Int -> Tree a -> a
get index tree = -- your code goes here
```

You may find it useful to write a function `size :: Tree a -> Int` that returns the size of any tree.

Answer on a separate sheet of paper.



http://xkcd.com/835/