Re-exam – Datastrukturer

DIT960/DIT961, VT-19
Göteborgs Universitet, CSE

Day: 2019-08-23, *Time:* 8:30-12.30, *Place:* J

Course responsible
Alex Gerdes, tel. 031-772 6154. Will visit at around 9:30.

Allowed aids
One hand-written sheet of A4 paper (both sides). You may also bring a dictionary.

Grading
The exam consists of *six questions*. For each question you can get a U, a G or a VG. To get a G on the exam, you need to answer at least *five* questions to G or VG standard. To get a VG on the exam, you need to answer at least five questions to VG standard (and the remaining question to a G standard).

A fully correct answer for a question, including the parts labelled ”For a VG”, will get a VG. A correct answer, without the ”For a VG” parts, will get a G. An answer with minor mistakes might be accepted, but this is at the discretion of the marker. A correct answer on the VG part may compensate a mistake in the G part, also at the discretion of the marker. An answer with large mistakes will get an U.

Inspection
When the exams have been graded they are available for review in the student office on floor 4 in the EDIT building. If you want to discuss the grading, please contact the course responsible and book a meeting. In that case, you should leave the exam in the student office until after the meeting.

Note

- Begin each question on a new page.
- Write your anonymous code (not your name) on every page.
- When a question asks for pseudocode, you don’t have to write precise code, such as Java. But your answer should be well structured. Indenting and/or using brackets is a good idea. Apart from being readable your pseudocode should give enough detail that a competent programmer could easily implement the solution, and that it’s possible to analyse the time complexity.
- Excessively complicated answers might be rejected.
- *Write legibly!* Solutions that are difficult to read are not evaluated!

Exercise 1 (complexity)

Which of the following statements are correct?

1. The function $f(n) = n^2 + n + 1$ is $O(n^2)$
2. The function $f(n) = n^2$ is $O(n^2 + n + 1)$
3. The function $f(n) = n^3$ is $O(2^n)$
4. The function $f(n) = n \log n$ is $O(n + \log n)$
5. The function $f(n) = n + \log n$ is $O(n \log n)$

Motivate your answer!

For a VG only:

The following Haskell function that groups consecutive duplicates of list elements into sublists.

```haskell
pack :: Eq a => [a] -> [[a]]
pack []     = []
pack (x:xs) = (x : takeWhile (== x) xs) : pack (dropWhile (== x) xs)
```

If a list contains repeated elements they should be placed in separate sublists. For example:

```
> pack [2,2,1,3,3,3]
[[2,2],[1],[3,3,3]]
```

Your task is to give the worst-case complexity in terms of n, the size of the input list, of the `pack` function. Begin by writing a recurrence relation. Motivate your answer!

Note that `takeWhile` applied to a predicate p and a list `xs`, returns the longest prefix (possibly empty) of `xs` of elements that satisfy p, whereas `dropWhile` removes the longest prefix. Both `takeWhile` and `dropWhile` have $O(n)$ worst-case time complexity, where n is the number of elements in the input list.

Exercise 2 (sorting)

Perform a quicksort-partitioning of the following array:

45	26	16	12	71	33	18	3	53
0	1	2	3	4	5	6	7	8

Show the resulting array after partitioning it with the following pivot elements:

a) first element

b) middle element (swap the first and middle element before partitioning)

c) last element (swap the last and first element before partitioning)

Also, highlight which subarrays that need to be sorted using a recursive call. (for example by drawing a line under each subarray)

For a VG only:

Define a Haskell function `merge` that merges two *sorted* lists. It should have the following type signature:

```
merge :: Ord a => [a] -> [a] -> [a]
```

You can solve the problem in less than 10 lines of code. Give the worst-case time complexity for this function.

Exercise 3 (basic data structures)

Assume we have the following interface for stacks in Java:

```
interface Stack<E> {
  void push(E elem);
  E pop();
  E top();
  boolean isEmpty();
}
```

a) What is the worst-case time complexity (in terms of n, the number of elements on the stack) for the different methods if you implement the interface with a (singly) linked list?

b) Implement a generic method in Java that calculates the number of elements on a stack:

```
public static <E> int size(Stack<E> s) { ... }
```

Note that you only have access to the methods defined in the interface, not to methods in the actual implementation of the stack.

c) What is the worst-case time complexity of the `size` method you implemented in *b)* if the stack is implemented with a linked list?

d) In many cases a stack interface already incorporates a `size` method, which can be implemented with $O(1)$ complexity. How can we implement this method with constant time complexity? What are the advantages and disadvantages of such an implementation compared to the one you implemented in *b)*?

For a VG only:

Give an implementation for a (singly) linked list that implements the above `Stack` interface.

Exercise 4 (heaps)

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

A

5	8	15	13	10	74	69	72	36	52
0	1	2	3	4	5	6	7	8	9

B

12	50	14	49	36	66	75	56	76	58
0	1	2	3	4	5	6	7	8	9

C

3	17	20	39	19	66	72	77	31	55
0	1	2	3	4	5	6	7	8	9

a) Write out that heap as a binary tree.

b) **For a VG only:** Add 7 to the heap, making sure to restore the heap invariant. How does the array look now?

c) **For a VG only:** A binary heap is a complete binary tree that satisfies the heap property. What does 'complete' mean in this context? Why is it important to have a complete tree for a binary heap?

Exercise 5 (search trees)

Assume we have the following binary search tree in Java:

```java
public class BST<E extends Comparable<? super E>> {
  private class Node {
    E data;
    Node left;
    Node right;

    Node(E data) {
      this.data = data;
    }
  }

  private Node root;

  public int height() { ...}

  public void insert(E data) { ... }

  public boolean invariant() { ... }
}
```

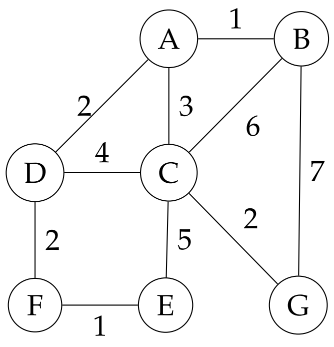
Your task is to complete the `height` and `insert` methods. The `height` method returns the height of the tree, whereas the `insert` method adds an element to the search tree, while maintaining the binary search tree invariant.

For a VG only:

Complete the method `invariant` that checks if the tree has the binary search tree property. That property states that for every node the element should be larger than all the nodes in the left sub-tree, and smaller than all the nodes in the right sub-tree.

Exercise 6 (graphs)

You are given the following undirected weighted graph:



For a G you may choose one of the following questions, for a VG you need to answer both.

a) Compute a minimal spanning tree for the following graph by manually performing Prim's algorithm using B as starting node.

Your answer should be the set of edges which are members of the spanning tree you have computed. The edges should be listed in the order they are added as Prim's algorithm is executed. Refer to each edge by the labels of the two nodes that it connects, e.g. DF for the edge between nodes D and F.

b) Suppose we perform Dijkstra's algorithm starting from node G. In which order does the algorithm visit the nodes, and what is the computed distance to each of them? (explain your answer)