Re-exam – Datastrukturer

DIT961, VT-22
Göteborgs Universitet, CSE

Day: 2022-08-18, *Time:* 8:30-12.30, *Place:* Johanneberg

Course responsible

Alex Gerdes will visit at around 9:30 and is available via telephone 031-772 6154 during the entire exam.

Allowed aids

You may 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 three questions to a VG standard and all remaining questions 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 and 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)

The following function produces a copy of an array of values sorted according to a separately specified array of keys. We assume that the keys and values arrays have the same length n. We are interested in the asymptotic complexity of `sort_values` in terms of n. We assume that comparing two keys takes constant time.

```
function sort_values(keys : Array, values : Array) -> Array:
        map = new SortedMap()
        i = 0
        while i < length(keys):
                map.put(keys[i], values[i])
                i = i + 1
        result = new Array of size length(keys)
        i = 0
        for each key, value in the entries of map:
                result[i] = value
                i = i + 1
        return result
```

a) What is the asymptotic complexity if `SortedMap` is a Binary Search Tree (BST)?

b) What is the asymptotic complexity if `SortedMap` is an AVL Tree?

Write your answer in O-notation. Be as exact and simple as possible. Justify why the complexity of the function has this order of growth.

For a VG only:

Three more questions about the function `sort_values`:

c) Give an example that displays the worst-case complexity when `SortedMap` is a BST. Answer with example arrays `keys` and `values` that make `sort_values` perform at the worst-case complexity you answered in 1*a*.

Note that since the complexity depends on the size n of the input, your example should be generic in n, that is, you should make it clear how the `keys` and `values` arrays would look for any value of n.

d) The function will not work as intended if `SortedMap` is a hash table. Explain why.

e) However, if we used a hash table, the function would still be able to run. What would the worst-case complexity of `sort_values` be in that case (assuming a perfect hash function)?

Exercise 2 (sorting)

Perform one level of split+sort+merge when *merge sorting* the following list:

```
[42,12,54,72,36,82,99,66,6,27]
```

Show the following:

a) The result of splitting the input list.

b) The result of calling the merge sort function on each of the splits.

c) The result of merging the sorted sublists. Explain how you do it.

For a VG only:

Your tasks are:

d) Define a `merge` function in Haskell that merges two sorted lists into a new sorted list:

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

The function should have a linear worst-case time complexity.

e) Consider the following alternative implementation of merge sort:

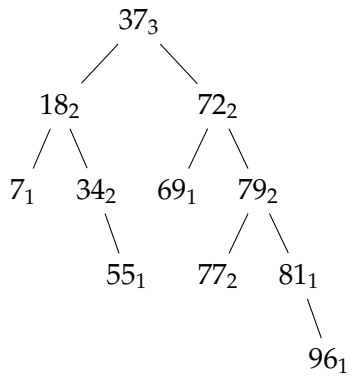
```haskell
mergeSort :: Ord a => [a] -> [a]
mergeSort = go []
 where
  go acc []     = acc
  go acc (x:xs) = go (merge [x] acc) xs
```

What is the worst-case time complexity of this function? Motivate your answer!

f) What is the best- and worst-case input for the `mergeSort` function?

Exercise 3 (search trees)

The tree below is neither a valid AVL tree nor is it a valid AA tree (the sub-scripts denote the AA tree level of a node).



You must answer the following questions:

a) Why is the tree not a valid AA tree? Explain how and where (in the tree) the invariant is invalidated.

b) Is it possible to change the level of the nodes such that the tree becomes a valid AA tree? Either give a new tree with updated levels or explain why the tree can't be made into a valid AA tree.

c) Why is the tree not a valid AVL tree? Explain how and where (in the tree) the invariant is invalidated. You can ignore the AA tree level annotations.

d) Perform one tree rotation to restore the AVL invariant and draw the resulting tree. Explain which rotation you performed and where it was done in the tree.

For a VG only:

Now consider the above tree as a plain BST.

e) Delete the root node (67) from the tree, and show the resulting tree.

f) There are two possible ways to delete a node which has two children. Now show the resulting tree after deleting 67 from the initial tree in the alternative way.

g) The following data type models a binary search tree in Haskell:

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

Implement the function

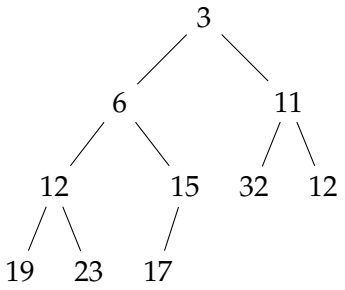
```
delete :: Ord a => a -> Tree a -> Tree a
```

which removes the given value from the tree (if it exists). You can pick any of the two possible ways to delete nodes with two children.

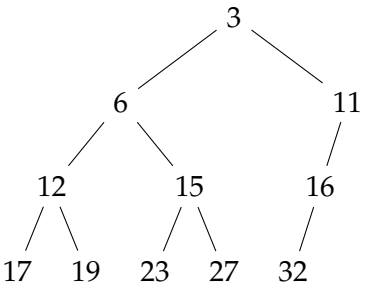
Exercise 4 (heaps)

Which of the following trees represent a binary min-heap?

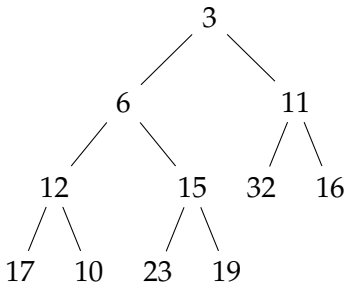
A)



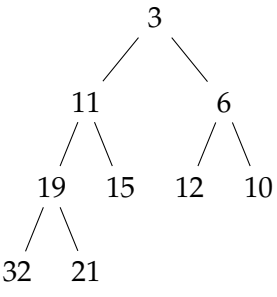
C)



B)



D)



For each of the trees:

- if it is not a binary heap, explain why not – i.e. point out a concrete specific problem with it,
- if it is a binary heap, write it in array notation.

For a VG only:

a) Explain how to represent a binary min-heap as an array. Make sure to describe all invariant(s).

b) Give pseudocode for inserting a new element into a heap represented as a dynamic array.

Exercise 5 (basic data structures: hash table and queue)

For each of the following questions, select the correct answer (1, 2 or 3).

Assume that you don't know anything at all about the hash function except that it takes constant time to calculate hash values.

a) What is the worst-case asymptotic time complexity, in terms of n, of adding one element to a separate-chaining hash table containing n elements?

(a) Constant, $O(1)$

(b) Logarithmic, $O(\log n)$

(c) Linear, $O(n)$

b) What is the most efficient way of resizing an open-addressing hash table that is close to full?

(a) Create a new table, roughly twice as large. Add each element from the old table to the new table, using normal hash table insertion.

(b) Create a new table, roughly twice as large. Add each element from the old table to the cell that is indicated by the new hash value.

(c) Create a new table, roughly twice as large. Since the hash values are stable, you can simply put the elements in the same positions as they were in the old table.

c) Which expression most accurately describes the worst-case time complexity of searching for an element in a separate-chaining hash table, with table size m, containing n elements, and buckets as linked lists of size at most k?

(a) Linear in the table size, $O(m)$

(b) Linear in the number of elements, $O(n)$

(c) Linear in the bucket size, $O(k)$

d) How can you delete an element `d` in an open-addressing hash table using linear probing?

(a) Find the element `d`, and clear the cell.

(b) Find the element `d`, and set the value of the cell to a special "deleted" value.

(c) Find the element `d`, then continue searching for the last element `x` in the cluster. Move `x` to the cell where `d` is (thus deleting `d`), and then clear the old `x` cell.

For a VG only:

Assume we have the following interface for queues in Java:

```
interface Queue<E> {
  void enqueue(E elem);
  E dequeue();
  boolean isEmpty();
}
```

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

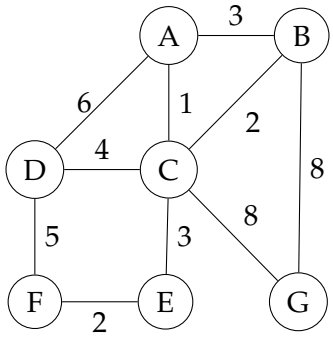
Hint: we can have multiple pointers to different nodes in the queue.

b) Give an implementation for the above `Queue` interface that uses a singly linked list.

c) Add a `size` method, which returns the number of elements in the queue, to the implementation. The method must have a constant worst-case time complexity $O(1)$.

Exercise 6 (graphs)

You are given the following undirected weighted graph:



Suppose we perform Dijkstra's algorithm starting from node A. In which order does the algorithm visit the nodes, and what is the computed distance to each of them? In addition, show the contents of the priority queue after each visited node. Explain your answer.

For a VG only:

Describe an algorithm (in pseudocode or your favorite programming language) that detects if a directed graph contains a cycle.

Assume that the graph is given in adjacency list representation:

- you can iterate over each vertex (e.g. `for each vertex v`),
- for each vertex `v`, you can iterate over each outgoing edge (e.g. `for each edge e from v`).

Your algorithm should be reasonably efficient, close to $O(V + E)$ in terms of V vertices and E edges. You can use any data structure from the course.

Suggested solutions – Re-exam – Datastrukturer

DIT961, VT-22
Göteborgs Universitet, CSE

Day: 2022-08-18, *Time:* 8:30-12.30, *Place:* Johanneberg

Exercise 1 (complexity)

a) Both loops iterate over each element in the array, so n times. The first loop calls `map.put` for each element, and in the worst case BST is unbalanced and so put has linear complexity.

So the first loop is $O(n) * O(n) = O(n^2)$, and the second is $O(n)$. In the end the complexity of `sort_values` is quadratic, $O(n^2)$

b) This time the tree will always be balanced, so adding elements to it is logarithmic in the worst case. Therefore the first loop (and the whole program) is $O(n) * O(\log n) = O(n \log n)$

For a VG only:

c) A simple example is if the `keys` array is sorted (in any order). Then the BST will be completely unbalanced (to the left or the right), and adding new elements will take $O(n)$ time. How the `values` array looks doesn't matter at all.

d) The keys in a hash table are not stored in order, so when we iterate through the map entries we will get them in an unknown order. So `sort_values` will not sort the values according to the keys.

e) If we have a perfect hash function, adding elements to a hash table is constant time, $O(1)$. So the first loop becomes $O(n) * O(1)$, and therefore the complexity of `sort_values` will be $O(n)$.

Exercise 2 (sorting)

a) The easiest way is to split in half, so you get the two lists: `[42, 12, 54, 72, 36]` and `[82, 99, 66, 6, 27]`.

b) Calling merge sort on each half gives us two sorted lists: `[12, 36, 42, 54, 72]` and `[6, 27, 66, 82, 99]`.

c) Merging compares the heads of each list and puts the smallest in a new list:

```
1.  [12,36,42,54,72] + [6,27,66,82,99] -> [6]
2.  [12,36,42,54,72] + [27,66,82,99] -> [6,12]
3.  [36,42,54,72] + [27,66,82,99] -> [6,12,27]
4.  [36,42,54,72] + [66,82,99] -> [6,12,27,36]
5.  [42,54,72] + [66,82,99] -> [6,12,27,36,42]
6.  [54,72] + [66,82,99] -> [6,12,27,36,42,54]
7.  [72] + [66,82,99] -> [6,12,27,36,42,54,66]
8.  [72] + [82,99] -> [6,12,27,36,42,54,66,72]
9.  [] + [82,99] -> [6,12,27,36,42,54,66,72,82]
10. [] + [99] -> [6,12,27,36,42,54,66,72,82,99]
```

For a VG only:

d) The `merge` function can be defined as follows:

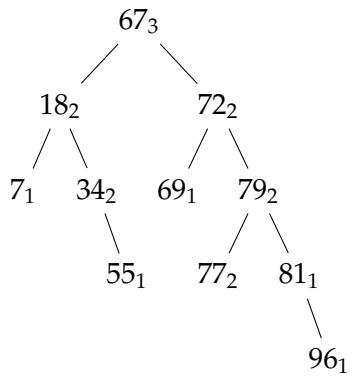
```
merge :: Ord a => [a] -> [a] -> [a]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
  | x <= y    = x : merge xs (y:ys)
  | otherwise = y : merge (x:xs) ys
```

e) This version of merge sort is in essence the same as insertion sort. The input list does not get divived into sublists of (nearly) equal size. In the worst case the to be merged singleton list contains an element that is greater than all elements in the other list in which case merge takes $O(n)$, which needs to be done n times, so the complexity is quadratic $O(n^2)$.

f) The best case is a reverse sorted list and the worst case a sorted list.

Exercise 3 (search trees)

The given tree in the exercise contained a small typo, which has quite a big impact on the questions, unfortunately. The root of the tree was supposed to be 67 and not 37:



This makes the questions c, d, e, and f more difficult to answer, so all attempts were accepted (that is, these questions are ignored).

a) The tree in the exam (with the incorrect root) is not a valid AA tree because does not satisfy the BST property (55 is larger than 37).

Answers that considered 67 as the root should point out that the level of 34 is wrong, it should be 1 (making it a proper three-node). The level of 77 is incorrect as well, and should be 1.

b) The tree given on the exam cannot be made into a AA tree because of the BST violation.

Otherwise one should use the suggestions from the previous answer.

c) The node 72 is imbalanced since its right child has height 3 and its left child has height 1.

d) Perform a single left rotation at node 72.

For a VG only:

e) Delete the root by replacing it by the largest element from the left subtree, which is 55.

f) Delete the root by replacing it by the smallest element from the right subtree, which is 69.

g)
```
delete :: Ord a => a -> Tree a -> Tree a
delete x Empty = Empty
delete x (Node l y r)
  | x < y     = Node (delete x l) y r
  | x > y     = Node l y (delete x r)
  | otherwise = maybe r (\(l', m) -> Node l' m r) (deleteMax l)
 where
  deleteMax Empty = Nothing
  deleteMax (Node l x r) = case r of
    Empty -> Just (l, x)
    _     -> deleteMax r
```

Exercise 4 (heaps)

a) This is a binary heap and the array looks like: [3, 6, 11, 12, 15, 32, 12, 19, 23, 17].

b) This is not a binary heap: the node 10 is smaller than its parent 12.

c) This is not a binary heap: it is not a complete tree.

d) This is a binary heap and the array looks like: [3, 11, 6, 19, 15, 12, 10, 32, 21].

For a VG only:

a) With the 0-based array indexing convention, an array `x` represents a heap exactly if for every valid array index *i*, we have `x[(i - 1) / 2 (rounded down)] <= x[i]`.

b) Call the dynamic array `x` and the element to be inserted `a`. Then we do:

```
i = size(x)
x.append(a)
loop:
        j = (i - 1) / 2 (rounded down)
        if (not j >= 0) or x[j] <= x[i]:
                break out of loop
        swap positions j and i of x
        i = j
```

Exercise 5 (basic data structures: hash table and queue)

a) 3, because the hash function could be very bad

b) 1

c) 3

d) 2, this is the idea with lazy deletion

For a VG only:

a) All methods have constant time worst case complexity $O(1)$. The usual implementation of a queue with a singly-linked list has both a pointer to the start as well as the end of the queue.

b) See source code below.

c) See source code below.

```java
import java.util.NoSuchElementException;

class QueueSingle<E> implements Queue<E> {
  private class Node {
    E data;
    Node next;
  }

  private Node first, last;
  private int size;

  public QueueSingle() {
    first = new Node();  // sentinel node
    last = first;
    size = 0;
  }

  public void enqueue(E elem) {
    Node n = new Node();
    n.data = elem;
```

```
    last.next = n;
    last = n;
    size++;
  }

  public E dequeue() {
    if (isEmpty()) throw new NoSuchElementException();

    Node n = first.next;
    E d = n.data;
    first.next = n.next;
    size--;

    return d;
  }

  public boolean isEmpty() {
    return size == 0;
  }

  public int size() {
    return size;
  }
}
```

Exercise 6 (graphs)

Node	Distance	Predecessor	Queue
A	0	-	{B-3, C-1, D-6}
C	1	A	{B-3, D-5, E-4, G-9}
B	3	A/C	{D-5, E-4, G-9}
E	4	C	{D-5, F-6, G-9}
D	5	C	{F-6, G-9}
F	6	E	{G-9}
G	9	C	{}

The priority queue may have extra duplicate nodes (with different distances) depending on the implementation.

For a VG only:

Here's an efficient solution. We use a variant of depth-first search that processes every vertex only the first time it is visited. With every vertex `v`, we store two Booleans flags (both initially false):

- ancestor: does `v` appear as an ancestor with respect to the current position in the

graph?

- visited: have we already processed v?

```
function has_cycle() -> bool:
        unmark all vertices (as both visited and ancestor)
        for each vertex v:
                if vertex_has_cycle(v):
                        return true
        return false

function vertex_has_cycle(v : Vertex) -> bool:
        if v is marked as ancestor:
                return true
        if v is not marked as visited:
                mark v as visited
                mark v as ancestor
                for each edge e from v:
                        if vertex_has_cycle(target of e):
                                return true
                unmark v as ancestor
        return false
```

We call `vertex_has_cycle(v)` for every vertex `v`. If this does not report a cycle, then no cycle exists.

We now check (not required) that this algorithm is $O(V + E)$. For every vertex `v`, the `for`-loop in the function `vertex_has_cycle` is executed only once. Thus, the loop body is executed only once for every edge. This also means that the visit function is called at most $V + E$ many times (E from this loop body and V from the outer loop). The remaining statements in an invocation of `vertex_has_cycle` are constant time.