Algorithms and Data Structures (DAT3/SW3)

Exam Assignments

Manfred Jaeger

12 February 2015

Full name:	
CPR-number:	
E-mail at student.aau.dk:	

This exam consists of three problems and there are three hours to solve them. When answering the questions in problem 1, mark or fill in the boxes on this paper. Remember also to put your name and your CPR number on any additional sheets of paper you will use for problems 2 and 3.

- *Read* carefully *the text of each problem before solving it!*
- *For problems 2 and 3, it is important that your solutions are presented in a readable form. Note that problems 2.B and 3.B have different requirements for the level of detail of the solution. If you don't have enough time to give full solutions for problems 2 and 3, then give a solution outline in a few lines of text.*
- *Make an effort to use a readable handwriting and to present your solutions neatly.*

[ItoA] refers to T.H. Cormen, Ch. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (both 2nd and 3rd editions).

During the exam you are allowed to consult books, printed lecture slides, and notes. The use of any kind of electronic devices, including calculators, is not permitted.

Problem 1 [50 points in total]

1. (*8 points*)

1.1. $\frac{1}{3}n^3 + n^2 \lg n + \sqrt{n^5}$ is:

[X] **a)** $\Theta(n^3)$ [ ] **b)** $\Theta(n \lg n)$ [ ] **c)** $\Theta(n^5)$ [ ] **d)** $\Theta(n^3 \lg n)$

1.2. $n\sqrt{\lg n}$ is:

[ ] **a)** $\Omega(n \lg n)$ [ ] **b)** $\Theta(n \lg n)$ [X] **c)** $O(n \lg n)$ [ ] **d)** $O(n)$

2. (*6 points*)
Consider the following recurrence relation:

$$T(n) \quad = \quad 2T(n/4) + n^{1/3} \quad (n > 1).$$

Mark the correct solution. $T(n) =$

[X] **a)** $\Theta(\sqrt{n})$ [ ] **b)** $\Theta(n^{1/3})$ [ ] **c)** $\Theta(n^3)$ [ ] **d)** $\Theta(\sqrt{n} \lg n)$

3. (*8 points*)
Consider the following algorithm:

```
   input  : An Integer n
   output: An Integer Z
    COMPUTE(n)
1  S = newStack()
2  for i=1 to n do
3      S.push(i)
4  Z = 0
5  while S.isEmpty() == false do
6      n = S.pop()
7      for i=1 to n do
8          Z = Z + i
9  return Z
```

3.1 Enter the return value of COMPUTE(4) in the box below:

COMPUTE(4) = [20]

3.2 Complete the following statement by entering the correct expression within the parentheses: the complexity of COMPUTE as a function of n is

$$\Theta\left(n^2 \quad\right)$$

4 (*7 points*)

Let A be a Max-Heap given by the array

3	8	6	4	2	1	5
1	2	3	4	5	6	7

and $A.\textit{heap-size} = 7$. Write into the boxes below the contents of the array A after the operation MAX-HEAPIFY$(A, 1)$ has been performed:

8	4	6	3	2	1	5
1	2	3	4	5	6	7

5 (*7 points*)

Consider the result of inserting the keys

$$9,\ 5,\ 13,\ 12,\ 16$$

(in this order) into a hash table of size $m = 7$ using open addressing with auxiliary hash function

$$h'(k) = k \bmod 7$$

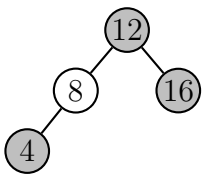
and linear probing. Write into the boxes below the contents of the hash table after the keys have been inserted:

12		9	16		5	13
0	1	2	3	4	5	6

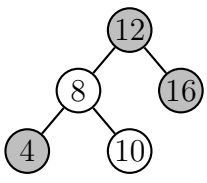
6 (*6 points*)

Which of the following is a valid red-black tree (● are black nodes, ○ are red nodes)?

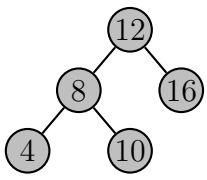
☐ **a)**



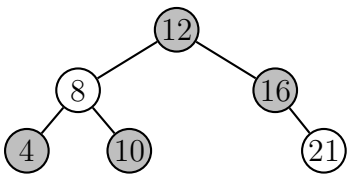
☐ **b)**



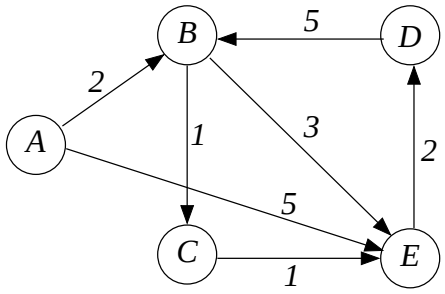
☐ **c)**



☒ **d)**



7 (*8 points*)
Consider the operation of the Bellman-Ford algorithm as given in [ItoA] on the following weighted graph with A as the source node:



Assume that in line 3 the algorithm enumerates the edges of the graph in the following order:

$$B \to C,\ A \to B,\ B \to E,\ A \to E,\ E \to D,\ C \to E,\ D \to B$$

Fill in the values that the $.d$ attributes of the nodes have after the completion of the second iteration of the **for** loop of lines 2-4 (i.e. the iteration $i = 2$ is completed):

$A.d =$ 0

$B.d =$ 2

$C.d =$ 3

$D.d =$ 7

$E.d =$ 4

Problem 2 [25 points]

Let us define an abstract data type *Integer Bag (IB)* as follows: the data that an IB stores consists of *multisets of integers*, i.e. collections of integer values where the same value can appear more than once. For example:

$$myMoI = [0, 3, 1, 0, 1, 2, 1]$$

is a multiset of integers. The following operations must be supported by a data structure implementing the IB abstract data type:

Name	Specification
void *add*(*Integer* i)	add i to the multiset
Integer *number_of*(*Integer* i)	returns the number of occurrences of i in the multiset
void *delete*(*Integer* i)	deletes one occurrence of i from the multiset (does nothing if the multiset does not contain any i)
Boolean *same_elements*(*IB* C)	returns *true* if this multiset and the multiset C contain the same elements (but maybe with different number of occurrences)

For example, if B is a new IB object that is initialized as the empty multiset, then after the sequence of operations

$$B.add(0), B.add(1), B.add(2), B.add(1), B.add(3), B.add(0), B.add(1)$$

B will represent the set *myMoI* from above. $B.number\_of(1)$ should now return 3. When next the operation $B.delete(1)$ is performed, then afterwards $B.number\_of(1)$ must return 2. $B.same\_elements(C)$ must return *true*, e.g. for $C = [0, 1, 2, 2, 2, 3, 3]$, and *false* for $C = [0, 2, 2, 3]$.

Following steps **A**-**C** below, describe a data structure that provides an efficient implementation of the IB abstract data type. You may make use of standard data structures described in [ItoA], and the operations they support. In that case, precisely show how the existing data structures are used, and what modifications or extensions of them you need to make.

A Give a description of how a multiset of integers is stored in your data structure, and make a sketch of how the example multiset myMoI would look like in your data structure.

B Give pseudo-code descriptions for the implementations of the *add* and *same_elements* operations.

C What is the complexity of the *add* and *same_elements* operations in your implementation? Depending on what is more appropriate for your implementation, you

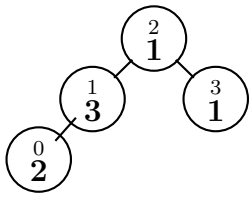
may give the complexity in terms of the total number of elements of a multiset, or the number of distinct elements in a multiset.

Solution

A

We represent IBs as augmented binary search trees (ABST), where a node in the tree has an additional *count* attribute that represents the number of times a given integer (key) value is contained in the IB.

myMoI then can look as follows (values (keys) in italics, counts in bold):



B

The *add* operation is implemented as a slight modification of the TREE-INSERT method of BSTs ([ItoA] p. 294):

```
ADD(i)
1  y = nil
2  x = this.root
3  found = false
4  while x ≠ nil & !found do
5      y = x
6      if x.key = i then
7          x.count ← x.count + 1
8          found = true
9      if i < x.key then
10         x = x.left
11     else
12         x = x.right
13 if !found then
14     z = new node(i, 1) /* creates a new node with key i and count 1 */
15     if y=nil then
16         this.root = z
17     else if z.key < y.key then
18         y.left = z
19     else
20         y.right = z
```

ADD operates like TREE-INSERT with the following modifications: while traversing the tree we also consider the case that for the key i to be inserted already a node exists in the tree (line 6), in which case the counter of that node is just incremented.

If no existing node for i is found, then a new node for that key is created (line 14) and inserted just as in TREE-INSERT.

For the *same_elements* method, we first define a slight modification of INORDER-TREE-WALK ([ItoA] p. 288) in which the keys are not printed but appended to a linked list:

INORDER-TREE-WALK($node\ x, Linked\ List\ ll$)

```
1 if x ≠ nil then
2     INORDER-TREE-WALK(x.left, ll)
3     ll.add(x.key)
4     INORDER-TREE-WALK(x.right, ll)
```

Now *same_elements* is implemented by comparing the linked lists produced by the tree walks on the two ABSTs:

SAME-ELEMENTS($ABSTC$)

```
1 myll = newLinkedList()
2 Cll = newLinkedList()
3 INORDER-TREE-WALK(this.root, myll)
4 INORDER-TREE-WALK(C.root, myll)
5 if myll.equals(Cll) then
6     return true
7 else
8     return false
```

In line 5 we assume that $ll1$.*equals*($ll2$) returns *true* if the linked lists $ll1$ and $ll2$ contain the same elements in the same order.

C

The *add* operation is linear in the height of the tree, which is $O(n)$ for n the number of distinct elements in the IB. Using an augmented red-black tree instead of a plain BST would reduce that complexity to $O(\lg n)$. The *same_elements* method is linear in the sizes of the two trees, i.e. $O(n_1 + n_2)$ where n_1 is the number of distinct elements in the first tree ("*this*"), and n_2 is the number of distinct elements in the second tree ("C")

Alternative Solutions:

Many proposed solutions for this problem are based on linked lists. Depending on to what extent the following critical elements are taken into account in a linked-list solution, such a solution can also receive the maximal number of points:

- Similar to the augmented BST, a linked list implementation for IBs should use only one list element for each distinct value in the multiset, and a *count* attribute for the number of occurrences of that value in the multiset.
- The linked list should be maintained as an ordered list.

With these two elements, *add* can be implemented as an $O(n)$ operation (n the number of *distinct* values in the multiset), and *same_elements* reduces to the trivial $O(n)$ list comparison that we also have at the end of the search-tree based implementation.

Problem 3 [25 points]

Suppose *Instwitr* is a social network with registered users who can *follow* one another.

Consumer electronics company *Beans* wants to market a new product to the users of *Instwitr*. To this end, *Beans* buys advertising space on *Instwitr*, so that a *Beans* ad will be displayed to a selected set of *Instwitr* users. *Beans* wants to maximize the effect of the advertising campaign at minimal cost. The cost of the campaign is proportional to the number of *Instwitr* users to whom the ad is displayed on their *Instwitr* home pages. To assess the effect of the campaign, *Beans* defines the *reach* of the campaign as follows: every user to whom the ad is displayed directly is in the *reach* of the campaign. If user U is in the *reach* of the campaign, and user V *follows* user U, then V is in the *reach* of the campaign. Users to whom none of the above two rules apply are not in the *reach* of the campaign.

A subset *MinSeed* of users is a *minimal seed set*, if the the following two conditions are satisfied: if the ad is displayed to all users in *MinSeed*, then the *reach* of the campaign is the set of all *Instwitr* users. And: there exists no smaller subset of users than *MinSeed* for which the first condition also holds.

A

Propose a formalization of this scenario in terms of a mathematical model or abstract data type. Give an illustration (drawing) of your formalization for the following example: *Instwitr* has 6 users A, B, C, D, E, F, which *follow* each other according to the table:

User	*follows* users
A	B
B	A
C	D
D	C
E	B, F
F	D, E

What is a minimal seed set in this network?

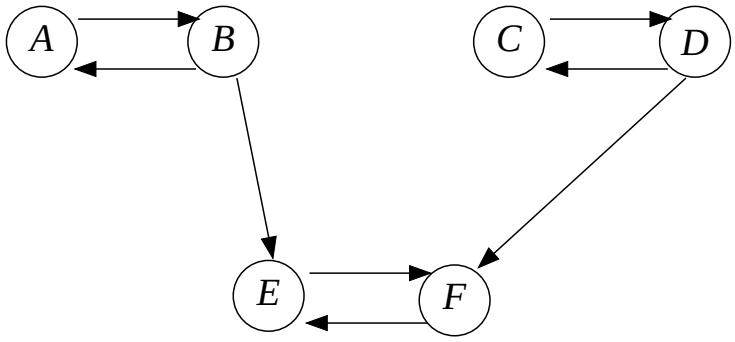
B

Give a description of an algorithm that computes a minimal seed set. A clear textual description (no detailed pseudo-code) is sufficient. What is the complexity of your algorithm?

Solution

A

The social network can be formalized as a directed graph where the nodes are users, and there is a directed link from user A to user B if B *follows* A. The example then looks as follows:



Minimal seed sets are $\{A, C\}$, $\{A, D\}$, $\{B, C\}$, and $\{B, D\}$.

B

We consider the *component graph* of the network graph ([ItoA] p. 617). Call a node of the component graph (i.e. a strongly connected component of the original graph) a root of the component graph, if it has no incoming edges. In our example, the component graph has three nodes corresponding to the strongly connected components $\{A, B\}$, $\{C, D\}$, and $\{E, F\}$. $\{A, B\}$ and $\{C, D\}$ are roots. A minimal seed set now is any set of nodes that contains exactly one node from each strongly connected component that is a root of the component graph.

This leads to the following algorithm:

Step 1: compute the strongly connected components of the social network graph using the algorithm [ItoA] p. 617

Step 2: determine the roots of the component graph. This can be done by checking whether the depth-first tree corresponding to a strongly connected component has any incoming cross edges ([ItoA] p. 697).

Step 3: Add an arbitrary node from each root component of the component graph to the seed set.

Each of the 3 steps is linear, so the overall algorithm also is linear ($O(|V| + |E|)$)

Alternative Solutions:

In part **A** it is most often suggested to point the direction of the edges from the follower to the user being followed. As a mathematical formalization this is just as good. One only has to be careful that in part **B** then one has to reverse the edges, if, for example, one wants to use depth-first-search to determine the reach of the campaign given a specific initial user to whom the ad is displayed.