Datastrukturer/Data structures DAT037

Time	Tuesday 15th January 2019, 14:00–18:00
Place	Hörsalsvägen
Course responsible	Nick Smallbone, tel. 0707 183062 (will visit at approximately 15:00 and 17:00)
Examiner	Nils Anders Danielsson

The exam consists of six questions, and you may answer in **English** or **Swedish**.

For each question you can get a **3**, **4** or **5**. Your grade on the exam is determined as follows:

- To get a 3 on the exam, you must get a 3 on 3 questions.
- To get a 4 on the exam, you must get a 4 on 4 questions.
- To get a 5 on the exam, you must get a 5 on 5 questions.

To get a 5 on a question, you must answer it correctly, including any parts marked **for a 4** or **for a 5**. Skipping a part marked **for a 4** or **for a 5** will prevent you from getting that grade on that question. Minor mistakes *might* be accepted; a larger mistake will lead to a reduced grade or a U on that question. Excessively complicated answers may be rejected.

Allowed aids	One A4 piece of paper of hand-written notes, which should be handed in after the exam. You may write on both sides.
Exam review	The exams will be available for review in the student office on floor 4 of the EDIT building. If you want to discuss the grading of your exam, contact Nick within 3 weeks of getting your result. In that case, do not take the exam from the student office.
Note	Begin the answer to each question on a new page. Write your anonymous code (*not* your name) on every page. Write legibly – we need to be able to read your answer!

Good luck!

1. Here is an algorithm to test if two arrays `X` and `Y` are disjoint (contain no elements in common):

```
boolean isDisjoint(X, Y) {
  S = new empty set data structure
  for every element x in X,
      S.insert(x)
  for every element y in Y,
      if S.member(y) then
          return false
  return true
}
```

What is the worst-case time complexity of this algorithm, if the set data structure is implemented with:

a) a binary search tree?

b) an AVL tree?

Give your answer in big-O (or big-Θ) notation. Write the complexity as simply as possible, and *not* as a recurrence relation; unnecessarily complex answers will be rejected. You do not need to give a proof. You may assume that comparisons take O(1) time.

For a 3, you may assume that `X` and `Y` have the same length, n.
Give the complexity in terms of n.

For a 4 or 5, don't assume that `X` and `Y` have the same length.
Give the complexity in terms of m and n, where m is the length of `X` and n is the length of `Y`.

For a 5, also answer the following question. Suppose that we want to minimise the worst-case runtime of the algorithm. When is it better to call `isDisjoint(Y, X)` rather than `isDisjoint(X, Y)`?

2. Consider the following hash table implemented using *linear probing*, where the hash function is the identity, $h(x) = x$ (modulo 10).

0	1	2	3	4	5	6	7	8	9
20	XXX	12		14	5			8	19

a) The value that was previously at index 1 has been deleted, which is represented by the **XXX** in the hash table. No other elements have been deleted from the hash table.

Which of the following values might have been stored there, before it was deleted? There may be several correct answers, and you should write down **all** of them.

A) 4 **B**) 9 **C**) 11 **D**) 17 **E**) 2 **F**) 10

b) The **XXX** in the array above illustrates *lazy deletion*: when a value is deleted from a hash table using linear probing, the hash table records that there used to be a value there. By contrast, index 3 did not previously contain a value and is represented by a blank space.

How are deleted values treated differently from "blank spaces" when searching (checking for membership) in a hash table using linear probing? If it helps you to explain, you may give an example referring to the hash table above.

c) **For a 5:** suppose that we are implementing linear probing but we do not do lazy deletion. That is, when removing a value from the hash table, we replace it with a blank space rather than an **XXX**. Show that this does not work, by giving a sequence of hash table operations, starting from an empty hash table of a size of your choice, that gives the wrong answer.

3. You are given a binary tree where each node has an extra field `parent` that should contain a reference to its parent node (the parent of the root node should be `null`). However, in this binary tree, the `parent` field has not been filled in and each node's `parent` field is currently set to `null`.

   Implement an algorithm that takes such a binary tree and sets the `parent` field of each node to point at the node's parent. **For a 4 or 5** you must also show (with an explanation, not necessarily a formal proof) that your algorithm takes linear time in the size of the tree.

   For this question, only **detailed code** (not necessarily Java) will be accepted, **not pseudocode**. You may not use other methods or procedures apart from ones you have implemented yourself.

   You may assume that the tree class is defined as follows:

```
// Binary tree with parent pointers.
public class Tree<A> {
  // Tree nodes; null represents an empty tree.
  class Node {
    A contents;  // Value stored in the node.
    Node left;   // Left child.
    Node right;  // Right child.
    Node parent; // Parent; is currently set to null.
  }

  // The root.
  private Node root;

  // Your task.
  // After calling setParents(), the parent field of each
  // node should contain the node's parent (and, in particular,
  // root.parent should be null).
  public void setParents() {
    ...
  }
}
```

4. In this question you should design a data structure that is almost a priority queue, except that you can find and remove the *second-smallest* element. Your data structure should support the following operations:

    - `empty()`: create a new, empty priority queue. If you prefer, you can model this as a constructor: `new PriorityQueue()`.
    - `add(x)`: add an integer `x` to the priority queue.
    - `findSecondSmallest()`: return the second-smallest element. You may assume that the priority queue has at least two elements.
    - `deleteSecondSmallest()`: delete the second-smallest element. You may assume that the priority queue has at least two elements.

    Example: After running `new(); add(1); add(5); add(4)`, calling `findSecondSmallest()` should return 4. If we then call `add(1)`, then calling `findSecondSmallest()` should return 1 (1 occurs twice in the priority queue, so it is both the smallest and the second-smallest element).

    The operations must have the following time complexities:

    - **For a 3**: O(1) for `new`, O(log n) for `add`, O(log n) for `findSecondSmallest`, O(log n) for `deleteSecondSmallest` (where n is the number of elements in the priority queue)
    - **For a 4 or 5:** as for a 3 but the complexity of `findSecondSmallest` must be O(1).

    You should write down what design you have chosen (e.g., what fields or variables the class would have or what data structure you are using), as well as how each operation is implemented.

    You should express your answer as either code or **pseudocode**, i.e. a mixture of code and textual description. Your answer must be detailed enough that a competent programmer could easily implement it. **You may freely use standard data structures and algorithms from the course in your answer, without explaining how they are implemented.**

    You may assume that comparisons take O(1) time, and that a high-quality hash function taking O(1) time is available.

5. Take a look at the following arrays.

A =

0	1	2	3	4	5	6	7	8	9	10
1	5	6	7	9	12	7	8	5	10	11

B =

0	1	2	3	4	5	6	7	8	9	10
1	8	7	21	43	11	9	20	22	46	58

C =

0	1	2	3	4	5	6	7	8	9	10
1	5	8	13	6	11	9	25	14	7	58

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

b) Write the heap out as a binary tree.

c) Remove the smallest element from the heap, using the standard binary heap algorithm for doing so. How does the array look now?

6. Define a method

   ```
   void printAscendingNoDuplicates(int[] x) { ... }
   ```

   that takes an array of integers, and prints its elements in ascending order, except that if a value occurs several times in the input array, it is only printed once. You may assume that a method `void print(int x)` exists that prints a single integer.

   For example, `printAscendingNoDuplicates({3,1,4,1,5,9,2,6,5})` should print the following sequence of numbers: `1 2 3 4 5 6 9`.

   For a 3, your solution should take **O(n log n)** time, where n is the length of the input array.

   For a 4 or 5, your solution should take **O(n log m)** time, where n is the length of the input array, and m is the number of *distinct* values in the array. For example, the array {1,1,3,3,4} has a length of 5 but contains only 3 distinct values.

   You should express your answer as either code or **pseudocode**, i.e. a mixture of code and textual description. Your answer must be detailed enough that a competent programmer could easily implement it. **You may freely use standard data structures and algorithms from the course in your answer, without explaining how they are implemented.**

   You may assume that comparisons take O(1) time, and that a high-quality hash function taking O(1) time is available.