# Data Structures - Assignment no. 3

**Remarks:**

- **Write both your name and your ID number very clearly on the top of the exercise. Write your exercises in pen, or in clearly visible pencil. Please write *very* clearly.**
- **Give correctness and complexity proofs for every algorithm you write.**
- **For every question where you are required to write pseudo-code, also explain your solution in words.**

1. **De-amortization.** In the lecture you learned the "doubling" method that allows to implement a stack using an array without placing a limit on the size of the stack, such that the amortized complexity of each operation is $O(1)$. The method is that every time the array gets full, a new array is allocated whose size is twice the size of the old array, and the old array is copied to the new array.

   In this question we ask you to *de-amortize* the doubling technique. Specifically, give an implementation of a stack using an array, such that the size of the stack can increase as much as needed (obviously, the array should increase its size once in a while), but the running time of each operation should be $O(1)$ *in the worst case*. We assume that for any value $k$, you can allocate an array of size $k$ in one time unit. However, this array is initialized with arbitrary values: copying the old array to the new array still takes time linear in the length of the old array.
   Hint: The data structure should maintain two arrays simultaneously.

2. Describe an algorithm that prints the $k$ smallest elements in a Heap. You can assume that the heap is represented as an array or as a tree, whichever is more comfortable for you. As usual, you may also assume that no key appears more than once in the heap. The algorithm should take $O(k \log k)$ time. The algortihm should not modify the heap. Give: (i) pseudo-code; (ii) an explanation of the algorithm; (iii) an explanation why it is correct; and (iv) an explanation why the running time is indeed $O(k \log k)$.
   Note: Observe that getting an algorithm that runs in time $O(k \log n)$, where $n$ is the size of the heap, is easy – just perform $k$ delete-mins. (In order to avoid modifying the heap, you need to undo your actions, which takes another $O(k \log n)$ time).

3. Describe an algorithm that solves the following problem. You are given $k$ sorted lists $A_1, \ldots, A_k$, each of length $n$. The output should be one sorted list which contains the keys of all input lists. The algorithm should take $O(nk \log k)$ time. You may assume that no key appears more than once in the input. Give: (i) an explanation of the algorithm; (ii) an explanation why it is correct; and (iii) an explanation why the running time is indeed $O(nk \log k)$.
   Hint: The merge procedure that is used as a subroutine in *merge-sort* (which you learned in the course "extended introduction to CS") answers this question for $k = 2$ in time $O(n)$.

4. (a) On an initially empty Fibonacci heap, carry out the following sequence of operations: insert(27), insert(17), insert(19), insert(20), insert(24), insert(12), insert(11), insert(10), insert(14), insert(18), deletemin, decreasekey(19, 7), delete (17), decrease-key(24,5), deletemin.

After each operation, draw the resulting structure of the Fibonacci heap. (Whenever elements enter the root list, they are inserted to the right of the current minimum. Successive linking of the root list also starts with the element to the right of the removed minimum and continues cyclicly.)

(b) Let $H$ be a Fibonacci heap consisting of a single tree. The tree consists of three nodes with keys $x = 7$, $y = 25$, and $z = 31$, such that $y = parent[z]$, and $x = parent[y]$. The node that contains $y$ is marked. Find a sequence of legal operations (insert, deletemin, decreasekey, delete) by which $H$ could have emerged from an empty Fibonacci heap.

5. Professor Cohen has devised a new data structure based on Fibonacci heaps. A *Cohen heap* has the same structure as a Fibonacci heap and implements exactly the same abstract data type. The implementations of the operations are the same as for Fibonacci heaps, except that insert and meld perform successive linking as their last step. What are the worst-case and amortized running times of all operations on Cohen heaps?

6. We wish to augment a Fibonacci heap $H$ to support two new operations without changing the amortized running time of any other Fibonacci-heap operations.

   - Give an efficient implementation of the operation FIB-CHANGE-KEY$(H, x, k)$, which changes the key of node $x$ to the value $k$. Analyze the amortized running time of your implementation for the cases in which $k$ is greater than, less than, or equal to key$[x]$.
   - Give an efficient implementation of FIB-PRUNE$(H, r)$, which deletes $min(r, n[H])$ nodes from $H$. Which nodes are deleted should be arbitrary. Analyze the amortized running time of your implementation. (Hint: You may need to modify the data structure and potential function.)