Data Structures and Algorithms I

Thu 2.3.2017

Exercise 6

No mandatory X-task this week, but there will be on last week. Draw a picture of all exercises.

31. Write an algorithm that creates a "two out of three" intersection of sets. Algorithm gets as parameter three sets, and creates a new set that contains those elements that are contained in exactly two input sets (not all three, not less than two). You can use either JavaAPI sets, or TRA-library set. Do not change the input sets. What is the time complexity of your algorithm?

32. Write a linear time algorithm *retainAll(LinkedList A, Collection B)* which removes from list A all those elements that are not in in set B. Hint: create a set (*HashSet*) of elements of collection B. Then it is easy to decide which of the elements of A should be removed. Do not use ready *retainAll*() operation.

33. Write a linear time algorithm *retainAll(ArrayList A, Collection B)* which removes from array-based list A all those elements that are not in in set B. Hint: create a set (*HashSet*) of elements of collection B. Then it is easy to decide which of the elements of A should be removed. Compared to task 32 you cannot, however, call *remove*() for each element to remove, as it would destroy the linear time complexity. Do not use ready *retainAll*() operation.

34. Write an algorithm that finds from a collection of floating point numbers those two different numbers that have the lowest (mutual) difference. Return the two values as a list of two elements or *null* if there are less than two values in the input collection. Hint: put all values to a sorted set and iterate that. What is the time complexity of your algorithm?

35. Use quicksort manually to sort array `A[16] = { M, L, D, Z, R, H, B, A, S, G, P, F, J, K, C, X }`. Write the contents of the array after each partition phase (before recursive calls). Mark down each pivot element and each recursive call. "Parallel" recursive calls can be drawn side by side. You can select pivot element randomly (or use the first element initially for simplicity).

36. Write an algorithm that sorts efficiently an array (or *Vector/ArrayList*) that is in almost ascending order. Here "almost" means that most of the array is in correct ascending order, but at most k last elements are out of order and can contain also small elements. Use a priority queue as helper by putting the k last elements to priority queue. Then merge the contents of the priority queue and the ascending majority of the array to the array (from end to start). Time complexity should be $O(n+k\log k)$.