Data Structures (IT205) 2013-14
Second Midsemester-semester Exam
4^{th} October, 2013

Time: 2 hours **marks: 80**

This question paper consists of TWO parts. Part A contains three elementary questions of which you are expected to answer any two. Part B contains three advanced questions of which you are expected to answer any two. All questions across the sections carry 20 marks each. You may not answer extra questions in Part A in lieu of a question in Part B. However, you may answer an extra question in Part B instead of a question from Part A and it will be considered. This question paper is printed on two pages on a single paper back-to-back and consists of a total of six questions. Check that your question paper is complete.

PART A

1. (a) Draw a binary search tree with the key values $1, 2, 3, 4, 5, 6$ such that across all nodes the sum of the **height** of that node and the **rank** of the key element it contains is same.

   (b) Draw a binary search tree with the key values $1, 2, 3, 4, 5, 6$ such that across all nodes the sum of the **depth** of that node and the **rank** of the key element it contains is same.

2. (a) Draw a binary max heap and a binary min heap each with the key values $1, 2, 3, 4, 5, 6$ in such a way that the number of positions in which the key values in the two heaps are identical is as **many** as possible. How many positions are they identical at, in this case?

   (b) Draw a binary max heap and a binary min heap each with the key values $1, 2, 3, 4, 5, 6$ in such a way that the number of positions in which the key values in the two heaps are identical is as **few** as possible. How many positions are they identical at, in this case?

3. Draw a 7 node complete binary tree and fill it with the key values $2, 4, 6, 8, 10, 12, 14$ according to the binary search tree key value property. Give four elements missing in the tree such that the search path along the tree for those elements terminates via the four different leaves of the tree. This search process is like binary search using the binary search tree starting at the root.

PART B

4. Assume you have a specific Binary Search Tree (BST) T consisting of n nodes and distinct key values. You are also given as input two integers i and j, such that $1 \leq i \leq j \leq n$. Write an efficient algorithm to determine whether or not there is a **subtree** of T which consists of **precisely** the keys of ranks $i, \ldots, j$. Analyse the running time of your algorithm.

5. Write pseudocode for **tripartitioning** an input array using **two pivots**. The pivots can be selected as the leftmost and rightmost elements of the selected subarray. If the right pivot is less than the left pivot then the two are first exchanged, before the rest of the operations. If the subarray has at most two elements, then after the above step it should terminate. After exchanging the two pivots (if necessary), let the pivots be copied to variables x and y. At the end of the tripartition procedure, a contiguous left portion, must contain **all** elements less than x, then x must go to its correct position followed by a contiguous portion of **all** elements whose value is between x and y, then y and finally the elements greater than y. Instead of the two index variables i and j used in the standard partition algorithm, you should use three index variables i, j, k. As in the standard partition, i varies in every iteration. Here j represents the rightmost index position in the region for elements less than x. k represents the rightmost index of elements whose value is between x and y. You may assume that all elements have distinct values. Analyse the running time of your routine.

   If this tripartition routine is used to implement quick-sort then instead of dividing the array into two portions at each level there will be a division into three subarrays. Write a recurrence relation and indicate the best and worst case running times for this modified quicksort.

6. Consider a scenario where we have an input array A of n elements. The array is scanned treating it as a binary max heap starting at the root. However, whenever we encounter a node (say at array position i) which has key value greater than its parent, instead of resolving the violations by exchanging key values with the parent and moving up the tree, we start a new heap by treating that position as index number 1. Thus we get one heap of the elements from 1 to $i - 1$. We start a new heap at position i treating it as element 1 and again include as many elements as we can until we encounter another violation.

   In this way we partition the input array into a set of maximal disjoint binary max heaps. Write a pseudocode to compute for an arbitrary input array the number of such heaps it decomposes into. The process should be efficient. Write down its running time in order notation.