CS21003 Algorithms-1
Tutorial 6
September 8th, 2017

1. Let us add the facility to a priority queue that the priority of an item may change after insertion. Provide an algorithm *changePriority* that, given the index of an element in the supporting array and a new priority value, assigns the new priority value to the element, and reorganizes the array so that heap ordering is restored. Your algorithm should run in O(log n) time for a heap of n elements.

2. Demonstrate that the *makeHeap* algorithm may fail to work if you invoke the heapify function for i = 0, 1, 2, . . . , n − 1 (in that order).

3. Design an O(nlogk)-time algorithm to merge k sorted linked lists having a total of n items. You must make use of heaps.

4. Let u, v be two nodes in a BST T. A common ancestor of u, v is a node w in T such that both u and v lie in the subtree rooted at w. The lowest common ancestor (LCA) of u, v is the common ancestor of u, v that is as far away from the root as possible. Design an O(h(T))-time algorithm to
compute the LCA of two nodes in T. Assume that the nodes in T do not maintain parent pointers.

5. Write a function that, given a binary search tree T with n nodes and an integer k ∈ {**1**, **2**, . . . , n},
returns the k-th largest element in the tree.

6. Consider a big array where elements are from a small set and in any range, i.e. there are many repetitions. How to efficiently sort the array? A **Basic Sorting** algorithm like MergeSort, HeapSort would take O(nLogn) time where n is number of elements, can we do better? **Hints:** it can be done O(nLogm) time, m is the number of distinct elements. Use AVL Trees with extra information.

```
Input:  arr[] = {100, 12, 100, 1, 1, 12, 100, 1, 12, 100, 1, 1}
Output: arr[] = {1, 1, 1, 1, 1, 12, 12, 12, 100, 100, 100, 100}
```

7. Write a function to count number of smaller elements on right of each element in an array. Given an unsorted array arr[] of distinct integers, construct another array countSmaller[] such that countSmaller[i] contains count of smaller elements on right side of each element arr[i] in array. Time complexity should be O(nLogn)

```
Input:   arr[] =  {12, 1, 2, 3, 0, 11, 4}

Output:  countSmaller[]  =  {6, 1, 1, 1, 0, 1, 0}
```

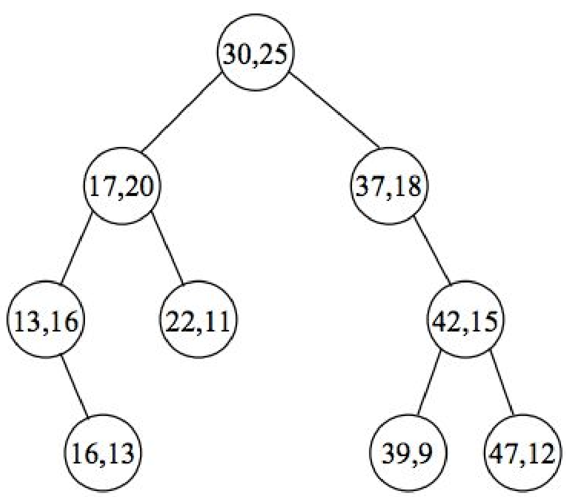
8. A node in a binary tree is an only-child if it has a parent node but no sibling node (Note: The root does not qualify as an only child). The "loneliness-ratio" of a given binary tree T is defined as the following ratio: LR(T) = (The number of nodes in T that are only children) / (The number of nodes in T). **Prove that for any non-empty AVL tree T we have that LR(T)≤1/2.**

Practice Problems:

1. You are given a rooted tree T. The width of T is the maximum number of nodes at a level in the tree. For example, consider a tree of height three on ten nodes a, b, c, d, e, f, g, h, i, j, where a is the root having three children b, c, d, node b has two children e, f, node d has three children g, h, i, and h has one child j. In this tree, the numbers of nodes at levels 0, 1, 2, 3 are respectively 1, 3, 5, 1. The width of this tree is therefore 5. You are given T in the first-child-next-sibling representation. Design an algorithm to compute the width of T in O(n) time, where n is the number of nodes in T.

2. You are given a binary tree T in the standard pointer-based representation. Each node in the tree consists of a key and two pointers (left and right). Write a function that, upon the input of a pointer to the root node, returns a suitable value indicating whether T is structurally an AVL tree. You do not need to look at the keys to identify whether T satisfies BST ordering (this is already covered in the class). Do not add any extra space in the nodes of the tree. If there are n nodes in the tree, your function must run in O(n) time.

3. A treap T is a binary search tree with each node storing (in addition to a value) a priority. The priority of any node is not smaller than the priorities of its children. The root is the node with the highest priority. Unlike heaps, a treap is not forced to satisfy the heap-structure property. An example of a treap is given in the adjacent figure, where the pair (x, y) stored in a node indicates that x is the value of the node, and y is the priority of the node. The x values satisfy binary-search-tree ordering, and the y values satisfy heap ordering. An example of a treap is given below.



Design an O(h(T))-time algorithm to insert a value x with priority y in a treap. (Hint: Use rotations.)