Massachusetts Institute of Technology
6.854J/18.415J: Advanced Algorithms
David Karger and Aleksander Mądry

Handout 2
Wednesday, September 7, 2016

Problem Set 1

Due: Friday, September 16, 2016.

Collaboration policy: Collaboration is *strongly encouraged.* However, remember that

1. You must write up your own solutions, independently.
2. You must record the name of every collaborator.
3. You must actually participate in solving all the problems. This is difficult in very large groups, so you should keep your collaboration groups limited to 3 or 4 people in a given week.
4. **No bibles. This includes solutions posted to problems in previous years.**

Problem 1. You are given a set of n closed intervals $[a_i, b_i]$ of the real line.

(a) Using a persistent-data-structure approach, develop a data structure that uses $O(n)$ space and can be constructed in $O(n \log n)$ time and can list all intervals containing a query point x in $O(k+\log n)$ time, where k is the number of containing intervals. Interval trees are a known solution to this problem, but persistent data structures provide a different one.

(b) Show that if the interval endpoints are integers in the range $1, \ldots, n$, the construction time can be improved to $O(n)$ and the running time can be improved to $O(k)$ (interval trees can't do this).

Problem 2. The *least common ancestor* (LCA) of nodes v and w in an n node rooted tree T is the node furthest from the root that is an ancestor of both v and w.

Consider first an offline variant of the least common ancestor problem. In this problem, we are given a set of m query pairs, and want to compute the LCA of each of these pairs of nodes quickly. This variant of the problem can be solved efficiently using the Union-Find data structure. (If you are not familiar with this data structure, you should review it (cf. [CLRS], Chapter 21) and note in particular the union-by-rank heuristic.)

Specifically, this algorithm for offline LCA problem works as follows. It associates with each node of the input tree T an extra field "name". Then, it processes the nodes of T in postorder. To process a node, it considers all of the query pairs that this node belongs to. For each such pair, if the other endpoint has not yet been processed, it does nothing. Otherwise, i.e., if the other endpoint has been processed, it executes a Find operation on that other node, and records the "name" of the result as the LCA of this pair. After considering in this way all of these pairs, it executes a Union operation on the just processed node with its parent, and sets the "name" of the set representative to be the parent.

We leave it as an exercise (not to be turned in), that this algorithm is correct, and takes $O((n+m)\alpha(n))$ time, where m is the number of query pairs. (If you haven't met α before, it is an inverse of Ackerman's function and grows VERY, VERY slowly—even slower than $\log^* n$. It is only 4 on the number of particles in the universe.)

However, in some instances we would like to find least common ancestors in an *online*. That is, we aren't told all of the pairs up front; we get queries one at a time.

(a) Show how to use the techniques of persistent data structures to preprocess a tree in $O(n \log n)$ time so as to allow LCA queries to be answered in $O(\log n)$ time. Aim for a simple solution here, even when solving part (b).

Hint: Path compression (which is used it union-find data structure) is messy from the point of view of persistent data structures, and is not necessary to achieve $O(\log n)$ time for union and find operations. Note also that nodes have arbitrary indegree, so path copying won't work.

(b) **(OPTIONAL)** Improve your solution to take $O(n)$ preprocessing time (while still having $O(\log n)$ query time).

Problem 3. Some splaying counterexamples.

(a) In class, we stated that single rotations "don't work" for splay trees. To demonstrate this, consider a degenerate n-node "linked list shaped" binary tree where each node's right child is empty. Suppose the (only) leaf is splayed to the root by *single* rotations. Show the structure of the tree after this splay.

To generalize this example, argue that there is a sequence of $n/2$ splays that each take at least $n/2$ work.

(b) Now, for the same starting tree, show the final structure after splaying the leaf with (zig-zig) double rotations. Explain how this splay has made much more progress than single rotations in "improving" the tree.

(c) Given the theorem about access time in splay trees, it is tempting to conjecture that splaying does not create trees in which it would take a long time to find an item. Show that this conjecture is false by showing that for large enough n, it is possible to restructure any binary tree on n nodes into any other binary tree on n nodes by a sequence of splay operations. Conclude that it is possible to make a sequence of requests that cause the splay tree to achieve any desired shape.

Hint: Start by showing how you can use splay operations to make a specified node into a leaf; then recurse.

Problem 4. Describe a data structure that represents an ordered list of elements under the following three types of operations:

Access(k)**:** Return the kth element of the list (in its current order).

Insert(k, x)**:** Insert x (a new element) after the kth element in the current version of the list.

Reverse(i, j) Reverse the order of the ith through jth elements.

For example, if the initial list is $[a, b, c, d, e]$, then Access(2) returns b. After Reverse(2,4), the represented list becomes $[a, d, c, b, e]$, and then Access(2) returns d.

Each operation should run in $O(\log n)$ amortized time, where n is the (current) number of elements in the list. The list starts out empty.

Hint: First consider how to implement Access and Insert using splay trees. Then think about a special case of reverse in which the $[i, j]$ range is represented by a whole subtree. Use these ideas to solve the real problem. Remember, if you store extra information in the tree, you must state how this information can be maintained under various restructuring operations.

Problem 5. In this problem, we're going to develop a less slick but hopefully more intuitive analysis of why splay trees have low amortized search cost. We need to argue that on a long search, all but $O(\log n)$ of the work of the descent and of the follow-on splay is paid for by a potential decrease from the splay. We'll use the same potential function as before—the sum of node ranks (multiplied by some constant). Also as before, we'll analyze one double rotation at a time, and argue that the potential decrease for each double rotation cancels the constant work of doing that double rotation.

(a) As in class, a double rotation will involve 3 nodes on the search path: x, its parent y, and its grandparent z. Call this triple *biased* if over $9/10$ of z's descendants are below x, and *balanced* otherwise. Argue that along the given search path, there can be at most $O(\log n)$ balanced triples.

(b) Argue that when a biased triple is rotated, the potential decreases by a constant, paying for the rotation. Do so by observing that rank of x only increases by a small constant, while the ranks of y or z decrease by a significantly larger constant. Do this for both the "zig-zig" and "zig-zag" rotations.

(c) Argue that when a balanced triple is rotated, the potential increases by at most $2(r(z) - r(x))$ (again, consider both rotation types).

(d) Conclude that enough potential falls out of the system to pay for all the biased rotations, while the real work and amount of potential introduced by the balanced rotation is $O(\log n)$, which thus bounds the amortized cost.

Problem 6. How long did you spend on this problem set? Please answer this question using the Google form that is located on the course website. This problem is mandatory, and thus counts towards your final grade. It is due by the Monday 2:30pm after the pset due date.