

SOLUTIONS: Homework 6, CS520, Spring '09

1. Recall that we showed how to implement a **reverse** operation on the path data type, reversing the list. The trick is to keep a parity bit in each node that tells which of its two subtrees should be listed in an inorder traversal. In order to avoid having to change all the parity bits when we reverse the list, we use the *Delta* scheme for keeping *Delta parity bits*. Each Delta parity bit, other than the one at the root, is 1 if and only if the ordinary parity bit differs from its parent.

We showed that the Delta parity bits can be updated during splaying by the usual trick that I suggested as an alternative to Tarjan's description: replace all bits on the splay path and their neighbors with their actual parity bits, perform the splay, updating the parity bits in the obvious way, and then go back through and replace the affected parity bits with their new Delta values.

That gives us a new operation on the *path* data type, which we may add without conflict with those we already have: Given a path, reverse the order of its elements.

Now that that's out of the way, describe how to reroot a linking/cutting tree by picking it up at a new node. Your description should consist of calls to **expose** and the path data type's **reverse** operation. Get a time bound for it.

Solution: *Between each pair of vertices in a tree, there is a unique path. A rooted tree can be viewed as an orientation of the edges of an unrooted one, so that they point from child to parent. If r is the old root and r' is the new root, the edges that get reversed by the rerooting are exactly those on the unique path between r and r' . Do an **expose** operation on r' to get r and r' into the same solid path. This takes $O(\log n)$, amortized. Then reverse exactly this set of edges by reversing this path. You must update the pig credits on this path. Some hoggers can become runts and vice versa. Any hogger that becomes a runt needs a credit, since he has a solid edge. There are $O(\log n)$ runts on any path to the root, so you can maintain the credit invariant on them with $O(\log n)$ credits. You can toss the pig credits on any runt that became a hogger.*

2. Show how to update a minimum spanning tree of a graph efficiently when a new edge is added or an edge weight is lowered.

The strategy should be based on the following observations. If you lower the weight of an edge that's part of a minimum spanning tree, the spanning tree remains a minimum one. Otherwise, adding the edge to the minimum spanning tree creates a unique cycle consisting of edges of the tree and the added edge. Removing a minimum-weight edge of this cycle leaves a tree that must be a minimum-spanning tree in the new graph. (Those of you who have had CS420 can easily prove this, but you may assume it for the assignment.)

Solution: *Implement the spanning tree with a linking/cutting tree. If uv is the added edge, reroot the tree at v , then perform an **expose** at u to get the path from u to v to be one instance of the **path** data type. If each vertex stores the weight of its parent edge, then finding a maximum-weight edge xy on this path takes $O(\log n)$ time, amortized. Compare this with uv . If it is larger, cut the tree at xy , and join the resulting trees with uv . All of these operations take $O(\log n)$ time, amortized.*

3. Show how to perform a *reduce-key* operation on an element of a leftist heap. You are given a pointer to the element. You may assume that each element has a pointer to its parent, so you can remove the subtree rooted at the element, yielding a new leftist heap, reduce the root of that heap, and then meld it back in. It looks like an $O(\log n)$ bound.

There is a missing piece of the analysis. Identify what it is, and show how to deal with it so that it doesn't mess up the time bound.

Solution: *You have to restore the leftist property to the tree that had the subtree removed from it, since some ancestors may now have shorter paths down to the null pointer dummy node that previously corresponded to the removed subtree. There are $O(\log n)$ such ancestors: if u is such an ancestor, giving a buzz cut to its subtree at the level of the removed node results in a complete subtree rooted at u ; otherwise the removed node can't affect the shortest path from u to a leaf. Therefore, u must be among the first $\log_2 n$ ancestors above the removed node. Updating the leftist property and labels at these ancestors therefore takes $O(\log n)$ time.*