

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.

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.)

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.

4. The last two problems are quite a bit harder, and I think only a fraction of the class will get them. They can be regarded as extra credit.

Let a *nested expression* be an expression of the form $E(x) = (a_1 op_1 (a_2 op_2 (a_3 op_3 \dots op_{k-1} (a_k op_k x) \dots)))$, where each op_i is addition or multiplication and x and the a_i 's are reals. Let us call k the *degree* of the expression.

For instance, $E(x) = (1/5 * (-4 + (2 * (3 + x))))$ is a nested expression of degree 4. We get nothing new by adding subtraction and division, since we can simulate these with negative numbers and reciprocals.

If $E_1(x)$ and $E_2(x)$ are nested expressions, let the *composition* $E_1E_2(x)$ be the result of replacing x in E_1 with $E_2(x)$. For instance, if $E_1(x) = (5 + (4 * (2 - (1 + x))))$ and $E_2(x) = (2 * (3 + x))$ then $E_1E_2(x) = (5 + (4 * (2 - (1 + (2 * (3 + x))))))$.

If $E(x) = (a_1op_1(a_2op_2(a_3op_3...op_{k-1}(a_kop_kx)...)))$, then its *list representation* is $((a_1op_1)(a_2op_2)...(a_kop_k))$. For instance, $E(x) = (1/5 * (-4 + (2 * (3 + x))))$ can be represented as $((1/5*), (-4+), (2*), (3+))$. The two representations are interchangeable.

The list representation of the composition of two expressions is just the concatenation of their list representations. This implies that composition is associative.

This gives a way of using a tree to represent a nested expression $E(x) = (a_1op_1(a_2op_2(a_3op_3...op_{k-1}(a_kop_kx)...)))$: pick some $((a_iop_i))$ to be the root, represent $((a_1op_1)(a_2op_2)...(a_{i-1}op_{i-1}))$ recursively as the left subtree, and represent $((a_{i+1}op_{i+1}), \dots, (a_k, op_k))$ recursively as the right subtree. The whole expression is the composition of the left subtree's expression, the root's expression, and the right subtree's expression.

- (a) Show by induction on the degree that every nested expression can be rewritten as $ax + b$. Let us call this expression the expression's *short form*.
- (b) Show how, given the short forms of E_1 and E_2 , you can find the short form of the composition E_1E_2 in $O(1)$ time.
- (c) Show how, given the short forms of E_1 and E_1E_2 , you can find the short form of E_2 in $O(1)$ time. Similarly, show how, given the short form of E_2 and E_1E_2 , you can find the short form of E_1 in $O(1)$ time.
- (d) Let us represent a nested expression with a data structure that is labeled with its short form and equipped with a some way to manage its list representation. Give the best time bounds you can get for each of the following:
 - i. **MakeNestedExpression** (v, op): Make a nested expression of degree 1.
 - ii. **Compose** (E_1, E_2): Given the data structures for E_1 and E_2 , combine them to form the data structure for E_1E_2 .
 - iii. **Split** (p): Given a pointer to an element p in the list representation of an expression E , split the list representation after p and return the data structures for the resulting subexpressions E_1 and E_2 .
 - iv. **Evaluate**(E): Given the data structure for a nested expression E , find the value of $E(x)$.

To avoid the need for lengthy answers, let me tell you what I'm looking for: Describe your data structure for the list representation; carefully describe any labels that elements of the data structure carry; briefly make reference to the different transformations that need to be performed on the data structure, and describe how you can update your labels efficiently as they are carried out; then briefly describe how the above operations can be carried out within your time bounds.

5. Let's consider fully parenthesized expressions involving addition, subtractions, multiplication, and division of constants, such as $((2 + 1) * 3) + (5 / (2 - 4))$. Such an expression has a binary parse tree.

Show how to obtain $O(\log n)$ time amortized bounds for each of the following:

- (a) **MakeExpression(a)**: Create an expression that has a single constant and no operator.
- (b) **Combine ($E_1 op E_2$)**: Given two expressions E_1 and E_2 and an operator op , combine them to form $(E_1 op E_2)$.
- (c) **Evaluate(E)**: Return the current value of the expression E .
- (d) **ChangeValue(E, v, x)**: v is a pointer to one of the constants in the expression E , and x is a new constant. Replace v with x in the expression.