

## HW2, CS 520, McConnell, Spring 12

1. Write C, C++, or Java code that prints out the costs (keys) of elements in the `path` data type, in order, in  $O(n)$  time, where  $n$  is the number of elements in the list.
2. Describe what additional data element to include at each node of the splay tree to support the following additional operation on the `path` data type so that it runs in  $O(\log n)$  amortized time:

`reverse(p)`: Reverse the order of elements in path `p`

Notice that, because `join` and `split` operation, including this operation allows you to reverse the order of elements of a *section* of `p`: `split p` before and after the section, making three lists, one of which is the elements of the section to be reversed. Reverse this list, and then concatenate the three lists back together. If you reverse different sections of `p`, the list can get pretty scrambled. Yet you must be able to pull it off in  $O(\log n)$  amortized time per reversal, and be able to print out the elements of the list in the order in which the user understands them, in  $O(n)$  time.

*There is a simple trick that you can describe in a couple of sentences. It's a variant of tricks you have already seen.*

In class, we talked about how to implement arraylist-type operations in  $O(\log n)$  amortized worst-case bounds. This exercise will be about an alternative way of implementing the following in  $O(\log n)$  amortized worst-case time. The purpose of the exercise is to give you practice at getting an amortized time bound.

We will focus on a small subset of arraylist operations.

```
list makeList(key)  -- create a one-element list containing key
int  retrieve(p, i)  retrieve the element at position i of p
void add(p, i, key)  increment the position numbers of elements at
                    positions i or greater, creating a space at position i; insert key at
                    position i
```

To avoid having to deal with tedious issues such as error checking while we think about the design of the algorithm, we will assume a precondition that  $0 \leq i \leq \text{length}(p) - 1$  for `retrieve` and  $0 \leq i \leq \text{length}(p)$  for `add`.

3. Describe how to do an `add` operation in  $O(\log n)$  time *if the tree currently has height  $O(\log n)$* , without doing a splay operation. The `add` operation should just add the new key at a new leaf node. If your tree requires an extra labels at each node, describe what information they give. You can assume that these labels are correct in the tree before the operation, but you must describe how to update them after an `add` operation so that they are still correct after the operation.

*Review your class notes.*

Unfortunately, the assumption that the tree has height  $O(\log n)$  will not be justified unless we take measures to ensure that this continues to be so after a sequence of **add** operations.

Let us define the *size* of a node to be the number of nodes in the subtree rooted at it, and *degradation measure* of a node  $v$  to be the maximum of the sizes of the two children of  $v$ , divided by the size of  $v$ .

For instance, if  $v$  has size 100, and children of sizes 24 and 75, then its degradation measure is  $75/100 = 3/4$ .

Be sure you understand why, if we ensure that no node has degradation measure larger than  $3/4$ , the height of the tree will be at most  $\log_{4/3} n = O(\log n)$ .

Here is an algorithm for balancing your data structure. Using an operation similar to what you did for Problem 1, store the elements of your list in an array  $A[0..n-1]$ , in order, in  $O(n)$  time. Now call the `build(A, 0, n-1)`, where `build` is defined as follows:

```
// precondition: 0 <= i <= j < A.length
// postcondition: return the root of a binary tree representing
// the list A[i..j], where every node's degradation measure is at most 1/2.

Node build (int [] A, int i, int j)
{
    int mid = (i+j)/2;
    Node n = new Node();
    n.key = A[mid];
    if (mid-1 >= i)
        n.leftChild = build(A,i,mid-1);
    if (mid+1 <= j)
        n.rightChild = build(A,mid+1,j);
}
```

We can see by induction on  $n = j - i + 1$  that every node in the returned tree has degradation measure at most  $1/2$ , as follows. This is true when  $n = 1$ , since both children have size 0. Adopt as the induction hypothesis that it's true whenever the method is called on fewer than  $n$  elements. When it is called on  $n$  elements, every node in the left subtree has degradation measure at most  $1/2$  and every node in the right subtree has degradation measure at most  $1/2$ , by the induction hypothesis. The left subtree has size  $\lfloor (n-1)/2 \rfloor$  and the right subtree has size  $\lceil (n-1)/2 \rceil$ . Since each of these is at most  $n/2$ , the root has degradation measure at most  $(n/2)/n = 1/2$ .

It takes  $O(n)$  time, because it takes  $O(1)$  time to create each node and assign it its children.

4. Modify this code so that it labels the nodes with correct **size** labels.

Our strategy will be to maintain the invariant that every node has degradation measure at most  $3/4$ . That way, the tree will have height at most  $\log_{4/3} n = O(\log n)$ . Adding an element will therefore take  $O(\log n)$  time. This also takes  $O(\log n)$  time, since the length of

the path to it is  $O(\log n)$ . However, the degradation measure of some ancestors may now be greater than  $3/4$ . We find the highest such ancestor  $v$  and rebuild its subtree so that no node in it has degradation measure exceeding  $1/2$ , as described above. The root of this rebuilt tree replaces  $v$  in our binary tree.

If there are  $k$  nodes in the subtree, this takes  $O(k)$  time, which is not  $O(\log n)$ .

5. Show that that the *amortized* worst-case cost of **add** operations is  $O(\log n)$  as follows.

You must deposit credits on nodes during an **add** operation in such a way that, whenever a node's  $v$ 's degradation measure exceeds  $3/4$ , the number of credits in it is at least equal to  $size(v)$ , so that you can then use these credits to pay the  $O(size(v))$  cost of rebuilding the subtree.

Note that after  $v$ 's subtree is rebuilt, we have spent the credits at  $v$ . You can't assume that any of the nodes in the subtree have any credits on them; they may have recently spent their credits also.

**Hint:** *If the current size of  $v$  is  $k$ , then one of its children,  $w$ , has size greater than  $3k/4$ . What is an upper bound on the size  $w$  had the last time  $v$  had zero credits?*

6. Tell the number of credits you must be given at the start of an **add** operation. If this is  $O(\log n)$ , then you've shown the amortized bound for all the rebuilding.
7. In class, we will show that an **expose** operation on the linking and cutting trees takes  $O(\log^2 n)$  amortized worst-case time. Tarjan gives a proof that the bound is actually  $O(\log n)$  amortized worst-case time, and we will accept that bound without going over the proof, which is more difficult.

Using the tighter bound, describe how to implement the following in  $O(\log n)$  amortized worst-case time:

```
Node lca (Node v, Node w) // find the lowest common ancestor of v and w.
    // Return null if v and w are in different trees
```

*Express the answer in terms of other operations we have described. You should be able to do it in a couple of sentences.*

8. Describe how to add the following operation to the set of operations he gives for linking and cutting trees:

```
reroot(v): make v be the new root of its tree
```

The operation must also take  $O(\log n)$  amortized time. Conceptually, the operation is similar to conceptualizing the tree as a bunch of balls and strings and "picking up" the tree at  $v$ , letting all the other balls dangle down from it. Notice that the other operations will subsequently need to take into account the new root, and still operate in  $O(\log n)$  amortized worst-case time.

*Express your algorithm in terms of other operations we have described. You should be able to express it in a couple of sentences.*