Homework 4

Aleck Zhao

September 28, 2017

1 Move to Front (34 points)

Amortized analysis can be used not just to give absolute bounds on running times, but also bounds *compared to other algorithms*. In this problem we'll see an example of this.

Suppose we have n items $x_1, x_2, \ldots, x_n$ that we wish to store in a linked list. The cost of a *lookup* operation is the position in the list, i.e. if we lookup some element x_i we pay 1 if it is the first element, 2 if it is the second element, 3 if it is the third element, etc. This models the time it takes to scan through the list to find the element.

For example, suppose there are four items and we lookup x_1 twice, x_2 three times, x_3 once, and x_4 four times. Then if we store them in the list (x_1, x_2, x_3, x_4) the total cost is $1(2) + 2(3) + 3(1) + 4(4) = 27$. On the other hand, if we stored them in the list (x_4, x_2, x_1, x_3) the cost would be $1(4) + 2(3) + 3(2) + 4(1) = 20$. It is easy to see that if we knew the number of times each element was looked up, the optimal list is simply sorting by the number of lookups, i.e. the first element is the one looked up most often, then the element looked up next most often, etc.

But what if we do not know how many times each element will be looked up? It turns out that a good strategy is *Move-to-Front* (MTF): when we lookup an item, we also move it to the head of the list. Let's say that moving an element is free (it is easy enough to implement with a constant number of pointer switches, in any case). So if, for example, we start with the list $(x_1, x_2, \ldots, x_n)$ and then lookup x_4, we pay a cost of 4 and afterwards the list will be $(x_4, x_1, x_2, x_3, x_5, x_6, \ldots, x_n)$. Then if we lookup x_4 again we only have to pay a cost of 1.

1. Consider a sequence of m operations (think of m as being much larger than n). Let C_{init} be the cost of these operations if we used the original list $(x_1, x_2, \ldots, x_n)$ the entire time (i.e. we do not use MTF). Let C_{MTF} be the cost if we use MTF on the same operations starting from the original list. Prove that $C_{MTF} \leq 2C_{init}$.

   Hint: Think of each item x_i as having a "piggy bank" in which the number of tokens is the number of elements before it in the current list that come after it in the initial list, or a potential function equal to the number of reversals.

2. Now let C_{static} be the smallest possible cost among all fixed lists. In other words, given the same sequence of lookups, each fixed list (without MTF) has some cost, and let C_{static} be the minimum of those costs. Note that this will correspond to the list that is sorted by the number of accesses, like in the first example. Prove that $C_{MTF} \leq 2C_{static} + n^2$.

   Hint: think of a new bank or potential function which is relative to the unknown static best fixed list, rather than relative to the initial list. What does this imply about the initial amount of money in the bank / potential?

2 Amortized analysis of 2-3-4 trees (33 points)

Recall that in a 2-3-4 tree, whenever we insert a new key we immediately split (on our way down the tree) any node we see that is full (has 3 keys in it). In the worst case, every node that we see is full and has to be split, so the number of splits that we do in a single operation can be $\Omega(\log n)$. Prove that the *amortized* number of splits is only $O(1)$.

Hint: Using a piggy bank at every node which stores a token if the node is full (or equivalently, a potential function which is equal to the number of full nodes) does not work. Why not? Can you modify the banks/potential functions?

Note: this does not mean that the amortized running time of an insert is $O(1)$ (since this is not true); it just means that the amortized number of splits is $O(1)$. So think of "cost" not as "time", but as "number of splits". Since we didn't talk about them in class, feel free to assume there are no delete operations.

3 More stacks (33 points)

In this problem we have two stacks A and B. In what follows, we will use n to denote the number of elements in stack A and use m to denote the number of elements in stack B. Suppose that we use these stacks to implement the following operations:

- PushA(x): Push element x onto A.
- PushB(x): Push element x onto B.
- MultipopA(k): Pop $\min(n, k)$ elements from A.
- MultipopB(k): Pop $\min(m, k)$ elements from B.
- Transfer(k): Repeatedly pop one element from A and push it into B, until either k elements have been moved or A is empty.

We are using the stacks as a black box – you may assume that PushA, PushB, MultipopA(1), and MultipopB(1) each take one unit of time (i.e. it takes one time step to push or pop a single element).

Design a potential function $\Phi(n, m)$, and use it to prove that the amortized running time is $O(1)$ for every operation.