Advanced Algorithms

Class Notes for Monday, September 24, 2012

Bernard Moret

1 Amortized Analysis

Amortized analysis is used mostly with data structures; it is a type of worst-case analysis, but, instead of focusing on the worst-case performance of a single data structure operation, it analyzes the worst-case performance of a long series of operations. The question is how? Two main approaches are used: the cost-accounting approach and the physics potential approach.

To illustrate the first, consider the familiar stack data structure, augmented with one additional operation: we will have not just `create`, `is\_empty`, `push`, and `pop`, but also `empty`, which empties the entire stack through a simple loop that calls on `pop` and tests whether the resulting stack is empty, repeating the loop until such is the case. Now, assume that each of the first four operations takes one unit of time; if the stack has n items in it, then the operation `empty` clearly takes $2n+1$ units of time. However, we claim that *all five* operations take constant amortized time: that is, a series of k stack operations (of any kind), for k large enough, takes $\Theta(k)$ time, so that each operation can be viewed as taking $\Theta(1)$ amortized time. We will prove this result by sharing the burden of `empty` among all operations, artificially making the other four operations slightly more expensive in order to "pay" for the `empty` operations. Although `push` and `pop` each take only one unit of time, we will charge three units per operation; then building a stack with n items in it incurs a minimum charge of $3n$ units, one third of it real running time, and two thirds of it additional charges against future work—we can think of these additional charges as down payments put into a bank account. Now if we call `empty` on such a stack, the actual cost is $2n+1$ units, but we can withdraw from our bank account to lessen that cost; we have a minimum of $2n$ units in the bank, so we can withdraw $2n-2$ units and subtract them from the actual cost of the operation to obtain a charge of just three units, just like the other four operations. In this analogy, operations with small actual running time are charged extra to build up our bank account, while operations with large actual running time withdraw from the account to reduce their charge—the bank account acts as an equalizing (amortizing) device. The key to such an analysis is to figure out the right amount to charge or withdraw for each operation so that any sequence of operations will reach a proper balance.

The second approach uses a physics analogy based on potential. For instance, the gravitational potential of an object is simply a measure of how far that object could fall. Raising the object increases its potential, lowering the object reduces it. Consider the problem of incrementing an integer: $i \leftarrow i+1$. Depending on i, this operation can take from 1 bit change (when i is even) to $1+\log(i+1)$ bit changes (when i is one less than a power of two). However, we expect that a typical incrementation will take constant time.

We could, in fact, prove that the average operation takes constant expected time, but we can make a much stronger statement: for sufficiently large n, a sequence of n increments takes $O(n)$ time in the worst case.

Intuitively, incrementation amortizes well because every second incrementation operation is cheap (as every second step we have an even number to increment), while, in contrast, the most expensive operations are very rare, as they only occur on successive powers of 2. How can we capture that? Note that the binary number 111111111111110 is even and thus "cheap" to increment, but that the result of this incrementation is the worst-case for an operation: one less than a power of 2. In contrast, note that incrementing 1111111111111, while the most expensive possible, yields 100000000000, which is going to be cheap to increment for quite a while, thanks to all those zeros. Thus the number of zeros in the binary representation of the number is a measure of how well set up we are—how good the current state of the data structure (here just a number representation) is; conversely, the number of ones is a measure of how bad the data structure has become. We will embody this vague notion of "bad structure" in a measure we call the "potential" of a data structure—potential to cause trouble, that is! Potentials are traditionally denoted by the Greek letter Φ, so in our case we write

$$\Phi(i) = \#\text{“1”}s \text{ in the binary representation of } i$$

Now a specific incrementation by 1 achieves two things: first, it takes some amount of work to carry out (between 1 and $\log i$, as we saw), and secondly it alters the quality of the data structure. In fact, the two compensate each other to within a constant: if incrementation takes $k+1$ bit changes, it is because it turned k ones into zeroes, then the first encountered zero into a one, so that the potential decreased by k and then increased by one, for a total change of $\Delta\Phi = -(k-1)$. Thus the sum of actual running time and change in potential is

$$k+1+\Delta\Phi = k+1-(k-1) = 2$$

This is what we claim to be the amortized time per operation, for the following reason. Consider summing the actual running times over a sequence of n operations; carrying the sum on both sides of the equation, we get

$$\sum_{i=1}^{n}(\text{actual time at step } i) + \sum_{i=1}^{n}(\Phi_i - \Phi_{i-1}) = 2n$$

But the Φ terms all cancel, except for the first and last, so we have

$$\sum_{i=1}^{n}(\text{actual time at step } i) + \Phi_n - \Phi_0 = 2n$$

Note that the overall $\Delta\Phi = \Phi_n - \Phi_0$ is $O(\log n)$, as the number reaches n and so has $\log n$ digits, and as the potential cannot exceed the number of digits nor be smaller than zero. Thus, as n gets large, the overall $\Delta\Phi$ becomes negligible compared to the right-hand side and we get, asymptotically

$$\sum_{i=1}^{n}(\text{actual time at step } i) \approx 2n$$

showing that the worst-case running time over any sequence of n incrementations is linear, as desired.

In general, we will want to establish an inequality of the form

$$\text{actual time} + \Delta\Phi \leq \text{amortized time}$$

by choosing a suitable definition of potential, then establish what is the largest potential difference possible over a sequence of n operations in order to verify that the $|\Phi_n - \Phi_0|$ vanishes against the sum of amortized times.

It is very important to note that the "amortized time" on the right-hand side of the inequality above has no direct meaning for any single operation—it is a convenient shorthand and an accounting method, but the only assertion about actual running time we can make is an assertion about the running time of a *sequence* of such operations.

2 Designing Amortized Data Structures

Amortization finds uses in many aspects of algorithm analysis, but it also guides the design of simpler data structures, data structures designed to have excellent worst-case performance over long sequences of operations, but not necessarily on an operation-per-operation basis. We shall look at two types of data structures for which amortization works particularly well: priority queues and equivalence classes. The first is one of the most common and most useful data structures; the second is an important component of runtime libraries for object-oriented languages and a crucial structure in the design of graph algorithms.

2.1 Priority queues

A priority queue has, at a minimum, the operations `create`, `insert`, `min_elt` (or max), and `deletemin`; in addition, it may also have a general `delete` and a `adjust_priority`. The classic array-based binary heap implements all of these operations in worst-case logarithmic time each, using very minimal space. It is in all respects an outstanding data structure, but it suffers from one serious drawback: it cannot be merged efficiently—any merging routine will take linear time, which is far too high. In order to design mergeable (usually called meldable) priority queues, we need to switch to a pointer-based design. Several such designs have been published over the years, none of which is competitive with the binary heap; however, all of them can be simplified and streamlined using an amortized perspective. We shall look at two such designs: leftist trees, which will give rise to skew heaps, and binomial queues, which will give rise to Fibonacci heaps.

2.1.1 Leftist trees

A leftist tree is a binary, heap-ordered tree; that is, every node has 0 or 2 children and the priority of the parent is never worse than the priority of either child. What makes it leftist is a third property, the leftist property: from any node in the tree, a shortest path to a leaf can always be chosen as the rightmost path (following only right pointers from the node to a leaf). A moment's thought shows that the leftist property implies that the rightmost path from the root of a leftist tree of n nodes has length $O(\log n)$. Moreover, because of the heap property, this path is a sorted linked list. This suggests melding two such leftists trees by merging the two linked lists defined by their rightmost paths: since these paths have a logarithmic bound on length and since we can merge two sorted lists to produce a sorted merged list in time linear in the size of the resulting list, this merging operation would run in $O(\log n)$ time too. The problem is that such a merging could easily produce a tree that violates the leftist property at some of the nodes along its rightmost path. Thus we need a way to detect such violations and a method for correcting them. Since a violation of the leftist property involves lengths of shortest paths to a leaf, we are going to have to store (and maintain) at every node the length of its shortest path to a leaf (i.e., the length of its rightmost path). Now the rest is relatively simple. We first merge the two sorted lists in time linear in the sum of their lengths; on reaching the last node on one or the other original list, we start backing up the merged list, checking at every node that the value stored in its right child does not exceed the value stored in its right child; if it does, we simply switch the two children (the two pointers); in either case, we set the value of the current node to the value of its right child plus 1, and then move back to the previous node, if any. The following pseudocode formalizes this process.

```
function Meld(p, q: PtrToNode): PtrToNode;
  (* p and q point to leftist trees. *)
  begin
    (* Call functions key and depth to avoid special cases for nil
       (key(nil) = infinity and depth(nil) = -1). *)
    (* Make things uniform--make p point to node with smaller key. *)
    if key(p) > key(q) then swap(p,q);
    if q <> nil
      then begin
             p^.right := Meld(p^.right,q);
             if depth(p^.left) < depth(p^.right)
               then swap(p^.left, p^.right);
             p^.depth := depth(p^.right) + 1
           end;
    Meld := p
  end; (* Meld *)
```

The maintenance work done at each node along the rightmost path is constant, so the overall work remains $O(\log n)$. Thus we can meld two leftist trees in logarithmic time. It now remains to see how this MELD operation is used to provide the classic operations of a priority queue, before altering the data structure with an eye to amortized behavior rather than worst-case per operation.