W4231: Analysis of Algorithms

10/7/1999

- Amortized Analysis of Data Structures

Main Idea of Amortized Analysis

Suppose you have a data structure such that every time you do a sequence of n operations starting from the empty structure, the total time is $O(n)$ (worst-case over all possible sequences of n operations).

This is almost as good as having a data structure with $O(1)$ worst case time per operation.

In fact it is *exactly as good* if the data structure is used only within an algorithm.

Amortized Running Time — Aggregate Method

We say that a data structure has operations running in amortized time $T(\cdot)$, if every time we do n operations the total running time is $O(nT(n))$.

This is the "aggregate method" approach to the definition of amortized complexity. It does not allow to differentiate the complexity of more or less efficient operations.

More General Definition

In general, say that you have implemented operations $op_1, \ldots, op_i$.

You can say that they have, respectively, amortized running time $t_1(), \ldots, t_i()$ if for any sequence of operations, where you are doing in total n_1 applications of op_1, n_2 applications of op_2 etc. the total time is at most

$$n_1 t_1(n) + n_2 t_2(n) + \cdots + n_i t_i(n)$$

where $n = n_1 + \ldots n_i$.

Why Amortized Analysis is Useful

Typically a data structure allows efficient `find` and `find-min` operations when it has a nice structure (think of a sorted vector). But it is hard to maintain a nice structure when `insert` and `delete` operations are allowed.

If the data structure is allowed to be a mess (a linked list) then `insert` is easy (also `delete` if you have a pointer), but `find` and `find-min` are hard.

Good worst-case efficient data structures are a compromise. They do have a structure, so `find` is efficient, and it is not too rigid, so `insert` and `delete` are efficient.

If we can use amortized analysis, we can think of a data structure that we allow to grow into a bit of a mess for a little while, but we periodically tidy it up.

Even if each tidying up takes a lot of time, provided that they occur sparsely, one can still prove good amortized upper bounds.

Implementing a Stack with a Vector

Consider the standard implementation of a stack.

We have a vector $V[1,\ldots,k]$ and a pointer $p \in \{0,\ldots,k\}$.

To push an element a we increase p and put a in $V[p]$.

To pop an element we read $V[p]$ and then decrease p.

When $p > k$ we are in trouble.

Heuristic

When the stack is full, a reasonable heuristic is to allocate a new vector twice bigger, and then copy all the elements in the new vector (and free the memory of the old one). This costs $O(k)$ time.

Suppose initially we create the empty stack using a vector of size one.

What is amortized time of operations?

Suppose we do n operations. Each pop is $O(1)$ time in the worst case. We have to account for the push operations.

Let N be the final size of the vector. The last time the vector was doubled, there were more than $N/2$ elements in the stack.

At no time there were more than n elements. Then $N \leq 2n$.

If we start from vector of size 1, we end with vector of size N, we did a create-and-copy operation to go to size 2, 4, . . . , N. (Note that N is a power of two.)

When we create a new vector of size 2^i, the time is $\leq c2^i$, where c is some constant.

The total creation time is

$$\sum_{i=1}^{\log N} c2^i \leq c2^{(\log N)+1} = 2cN \leq 4cn$$

So the total time for n operations is $O(n)$ for the pop, $O(n)$ for the creations, and $O(n)$ for the push after creations.

Constant amortized time. The heuristic makes sense.

Counter

Suppose we want to maintain a counter.

The counter is initially zero. Each `inc` operation increases its value by 1.

The implementation must be able to work with an unbounded number of increases.

Representing a big integer

We represent the counter as a linked list of bits.

We start with the least significant bit.

Initially, length-zero list containing only a zero.

Implementation of `inc`:

move a pointer from the beginning of the list.

If the pointer is on a 1, change to 0 and move on.

If the pointer is on a 0, change to 1 and stop.

If the pointer reaches the end of the list create a new digit and put 1.

Running time

Worst case: when the value of the counter is of the form $n = 2^k - 1$, `inc` takes $O(k) = O(\log n)$ time.

Amortized analysis: when we do n applications of `inc`, half of the times we just do one step; 1/4 of the times we need two steps, . . .

The total running time is a big-Oh of

$$= n + n/2 + n/4 + n/8 + n/16 + \ldots + n/2^k + \ldots + 1 < 2n$$

So the amortized time of `inc` is O(1)

A Queue with two Stacks

Let's see how to implement a queue by using two stacks as a black box.

We have stacks A and B. We can execute operations `push` and `pop`. We want to use A and B to implement a queue supporting operations `insert` and `remove`.

Implementation

- `insert`(x): `push`(x, A).
- `remove`$()$:
  - if B is not empty, return $x =$ `pop`(B).
  - if B is empty, then repeat the following until A is empty: $x =$ `pop`(A); `push`(x, B).
    Finally, return $x =$ `pop`(B).

Correctness is clear(?), worst-case running time is $O(n)$, where n is the size of A at the time an unfortunate `remove` is executed.

Amortized Analysis

We want to prove that `insert` has amortized running time 3 and `remove` has amortized running time 1.

(i.e. we want to show that after n_1 `insert` and n_2 `remove`, we have done no more than $3n_1 + n_2$ elementary operations)

Potential Method

At any time, we associate to the data structure (A, B) a non-negative integer value $\Phi(A, B)$ that we call the *potential*.

We want a definition satisfying the following properties:

1. The potential is always ≥ 0, and it is zero for the empty data structure.

2. The actual cost of each operation is (at most) equal to its claimed amortized running time plus the difference between the potential before the operation and potential after the operation.

Our Case

We define the potential as twice the number of elements in A.

We want to prove the definition satisfies properties 1 and 2. 1 is clear, have to prove 2.

Consider `insert`: its actual cost is 1, and it increases the potential by 2. Its claimed amortized running time is 3, we are OK.

Consider `remove`: if B is not empty, then the cost is 1, the potential stays the same, the claimed amortized running time is 1, we are fine.
If B is not empty, the cost is $1 +$ twice the number of elements of A, the potential goes down by a factor of twice the elements of A, the claimed amortized running time is 1, OK.

Then . . .

This gives a proof that the claimed amortized running times are true.

Suppose we do a sequence of n operations, having claimed amortized running times $a_1, \ldots, a_n$. Let the potential be initially $p_0 = 0$ and then $p_1, p_2, \ldots, p_n$.

The actual running time (by property 2) is at most

$$(a_1 + p_0 - p_1) + (a_2 + p_1 - p_2) + \cdots + (a_n + p_{n-1} - p_n)$$

$$= a_1 + \ldots + a_n + p_0 - p_n \leq a_1 + \ldots + a_n$$

Generality

This is a general method.

We define a data structure and implement operators.

We attach a potential function to each instance of the data structure and a cost to each operator.

We prove that properties 1 and 2 hold, and this gives a proof that the costs are actually the amortized running times of the operators.

In practice, we define the potential so that property 1 is true, and then we set the amortized time of an operator as the worst case of the expression

`actual running time` $+$ `old potential - new potential`

This guarantees that property 2 is true.

Binomial Heaps Again

Claim: if `delete-min` is not allowed, then the `insert` operator takes $O(1)$ amortized time.

Recall that `insert`(x, H) is the special case of `union` where we take the union of a binomial heap H (say of size n) with a "singleton" binomial heap made only by x.

Also recall that the trees in H are in 1-1 correspondence with the 1s in the binary representation of n.

`insert` performs a sequence of tree-unions, each one taking constant time.

The running time of `insert` is proportional to the running time of `inc` applied to a counter whose value is n.

In the same way `inc` uses $O(1)$ amortized time (if no decrease operation is allowed), the same is true for `insert`.

Possible Improvements

It would be trivial to implement `find-min` in $O(1)$ time.

We can implement `insert` in $O(1)$ amortized time even when `delete` $-$ `min` is allowed.

Define the potential of a Binomial Heap to be (a suitable constant times) the number of trees.

Under this definition, `insert` takes constant amortized time.

Also `delete-min` is still having a logarithmic cost.

So does `union` (one has to consider the sum of the potentials of the two initial heaps).

So our amortized bounds are $O(1)$ for `find-min` and `insert`, and logarithmic for `decrease-key`, `union` and `delete-min`.

Next (and last) Data Structure

With Fibonacci Heaps we can do everything in $O(1)$ amortized except `delete-min`.

Note that it is impossible (with comparison-based implementations) to do both `insert` and `delete-min` in $o(\log n)$ amortized time, otherwise we could sort in $o(n \log n)$ time.