Amortized Analysis

Ming-Hwa Wang, Ph.D.
COEN 279/AMTH 377 Design and Analysis of Algorithms
Department of Computer Engineering
Santa Clara University

Amortized Time Bound

- The worst-case running time for any sequence of M operations, degenerate behavior cannot occur repeatedly.
- Amortized bounds are weaker than the corresponding worst-case bounds, because there is no guarantee for any single operation. But amortized bounds are stronger than the equivalent average-case bounds, because average-case cannot guarantee any M operations will behave as M * average running time.
- Amortized analysis is tricky because it requires us to look at an entire sequence of operations instead of just one. Sometimes it is easier to solve a problem indirectly than directly.
- The insight into a particular data structure gained by performing an amortized analysis can help in optimizing the design.

Aggregate Method

- A sequence of n operations takes worst-case time T(n) in total. The average cost, or amortized cost, per operation is therefore T(n)/n.
- The amortized cost applies to each operation, even when there are several types of operations in the sequence.

Accounting Method

- Assign differing charges (amortized cost) to different operations. When an operation's amortized cost exceeds its actual cost, the difference is assigned to the data structure as credit. Credit can be used later on to help pay for operations whose amortized cost is less than their actual cost.
- The total amortized cost of a sequence of operations must be an upper bound on the total actual cost of the sequence. The total credit must be nonnegative at all time.

Potential method

- Represent the prepaid work as potential energy that can be released to pay for future operations. The potential is associated with the data structure as a whole.
- Let the initial data structure be D_0, c_i be the actual cost of the i^{th} operation, and D_i be the data structure that results after applying the i^{th} operation to data structure D_{i-1}. A potential function Φ maps each data structure D_i to a real number $\Phi(D_i)$, which is the potential associated with data structure D_i.
- The amortized cost γ_i of the i^{th} operation with respect to potential function Φ is defined by $\gamma_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$. The total amortized cost of the n operations is $\Sigma_{i=1}^{n}\gamma_i = \Sigma_{i=1}^{n}(c_i + \Phi(D_i) - \Phi(D_{i-1})) = \Sigma_{i=1}^{n}c_i + \Phi(D_n) - \Phi(D_0)$
- If $\Phi(D_n) \geq \Phi(D_0)$, then $\Sigma_{i=1}^{n}\gamma_i$ is an upper bound on the total actual cost. But we don't always know n, therefore, we require $\Phi(D_i) \geq \Phi(D_0)$ (which is 0 for convenience) for all i, then we guarantee that we pay in advance.

Stack Operations

- Push(S, x), Pop(S) and Multipop(S, k), which removes the k top objects of stack S, or pops the entire stack if it contains less than k objects. If stack has s objects, then min(s, k) of objects popped off S.
- Aggregate method: Each object can be popped at most once for each time it is pushed. O(n)/n = O(1).
- Counting method: assign amortized cost Push 2, Pop 0 and Multipop 0.
- Potential method: Φ = the number of objects in stack S.

Incrementing a Binary Counter

- A k-bit binary counter that counts upward from 0. We use an array $A[0..k-1]$ of bits, where length[A] = k, then $x = \Sigma_{i=0}^{k-1}A[i] * 2^i$.
- Aggregate method: $A[0]$ flips each time increment is called. $A[i]$ flips $\lfloor n/2^i \rfloor$ time in a sequence of n increment operations on an initially zero counter.
- Counting method: assign amortized cost, 2 for set 2 and 0 for reset.
- Potential method: define the potential of the counter after the i^{th} increment operation to be b_i, the number of 1's in the counter after the i^{th} operations. The actual cost of the operation is therefore at most t_i+1, since in addition to resetting t_i bits, it sets at most one bit at 1.

Dynamic Table

- the cost of the i^{th} operation is c_i

$$c_i = \begin{cases} i & \text{if } i-1 \text{ is an exact power of 2} \\ 1 & \text{otherwise} \end{cases}$$

$\Sigma_{i=1}^{n}c_i < n + \Sigma_{j=0}^{\lfloor \lg n \rfloor}2^j < 3n$

- double the table size when an item is inserted into a full table, but halve the table size when a deletion causes the table to become less than ¼ full
- using potential method

$$\Phi(T) = \begin{cases} 2\,\text{num}[T] - \text{size}[T] & \text{if } a(T) \geq 1/2 \\ \text{size}[T] / 2 - \text{num}[T] & \text{if } a(T) < 1/2 \end{cases}$$

Binomial Queues

- An insertion that costs c units results in a net increase of 2 - *c* trees in the forest, i.e., expensive insertion remove trees, while cheap insertion create trees.
- Let C_i be the cost of the i^{th} insertion. Let T_i be the number of trees after the i^{th} insertion. $T_0 = 0$ is the number of trees initially. $C_i + (T_i - T_{i-1}) = 2$, $\Sigma_{i=1}^{N} C_i + T_n - T_0 = 2N$, and $\Sigma_{i=1}^{N} C_i \leq 2N$. Thus, O(1).

Skew Heaps

Fibonacci Heaps

Splay Trees