Amortized Analysis

The time required to perform a sequence of (data structure) operations is averaged over all the operations performed.

No probabilistic assumptions - not an average case analysis!

Stack Operations

Push(S, x) - Push x to the stack S.

Pop(S, x) - Pop the top of stack S.

Multipop(S, k)

1. While not empty and $k \neq 0$ do

   1.1 Pop(S);
   1.2 $k \leftarrow k - 1$;

A call to Multipop(S, k) takes $O(k)$ time.

A "naive" analysis of a sequence of of n operations Push, Pop, and Multipop on an empty stack give $O(n^2)$, since the worst case time of one stack operation is $O(n)$.

Binary Counter

k bit binary counter $A[0, ..., k-1]$.

$k = length[A]$

$x = \sum_{i=0}^{k-1} A[i]2^i$

Increment(A):

1. $i \leftarrow 0$;

2. While $i < length[A]$ and $A[i] = 1$ do

   2.1 $A[i] \leftarrow 0$
   2.2 $i \leftarrow i + 1$

3. If $i < length[A]$ then $A[i] \leftarrow 1$.

What's the cost of n increments?

A worst case increment takes $O(k)$ steps, thus $O(nk)$ time.

Amortized Analysis - The Aggregate Method

Compute $T(n)$, the total work in n operations and take $T(n)/n$.

Stack Operations: The total amount of work is O(number of Push operations), thus $T(n) = O(n)$, and amortized work is $O(1)$.

Counter increments: In n increments the i-th bit $(i = 0, 1, ..., k-1)$ is flipped only every 2^i increments. Thus

$$T(n) = \sum_{i=0}^{\log n} \lfloor \frac{n}{2^i} \rfloor \leq n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n$$

Thus, the amortized work is $T(n)/n = O(1)$.

Amortized Analysis - The Accounting Method

We associate a **charge** with each operation with the invariant that at any step the total charges in the new accounting is not smaller than the total real cost up to that point.

Stack Operations:

Real cost: Push - 1; Pop - 1; Multipop(k, S) - min(k, s).

New charge: Push - 2; Pop - 0; Multipop(k, S) - 0.

Since the number of Pops is bounded by the number of Push operations, the total new charge at any given time is never smaller than the real cost.

A simple $O(n)$ bound on the total amortized cost.

Counter increments:

New charge: 2 for each flip to 1, 0 otherwise.

Since we start with all bits set to 0, this give an upper bound on the real cost.

In each increment operation only one bit is flipped to 1 - $O(1)$ amortized cost.

Amortized Analysis - The Potential Method

Let D_0 be the initial data structure.

Consider a set of n operations: c_i the cost of operation i, D_i the data structure after operation i.

The **potential function**:

$$\Phi : \{D_i \mid i = 0, ..., n\} \rightarrow R$$

The amortized cost of operation i is

$$A(c_i) = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

The total amortized cost:

$$\begin{aligned}\sum_{i=1}^{n} A(c_i) &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

If we can show that $\Phi(D_i) \geq \Phi(D_0)$ for all i, then

$$\sum_{i=1}^{n} A(c_i) \geq \sum_{i=1}^{n} c_i$$

Stack Operations

Define $\Phi(D_i) =$ number of items in the stack.

$\Phi(D_i) \geq \Phi(D_0) = 0$.

The amortized cost of a push operation (starting with s items in the stack) is:

$$A(c_i) = 1 + (s + 1) - s = 2$$

The amortized cost of a Multipop(S, k) ($k' = min(k, s)$)

$$A(c_i) = k' + (s - k') - s = 0$$

Thus, all operations have amortized cost $O(1)$, the total cost of n operations is $O(n)$.

Counter Increment

Let $\Phi(D_i) =$ number of 1's in the counter.

If counter starts with 0: $\Phi(D_i) \geq \Phi(D_0) = 0$.

Assume that the i-th operation (increment) flipped t_i bits to 0.

The total cost of that operation is $t_i + 1$.

$A(c_i) = c_i + \Phi(D_i) - \Phi(D_{i-1}) = t_i + 1 + 1 - t_i = 2$.

The total amortized cost of n operations is $O(n)$.

If the counter didn't start with 0:

$$\begin{aligned}\sum_{i=1}^{n} A(c_i) &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0)\end{aligned}$$

or

$$\begin{aligned}\sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} A(c_i) + \Phi(D_0) - \Phi(D_n) \\ &\leq \sum_{i=1}^{n} A(c_i) + k\end{aligned}$$

Dynamic Tables

Allocate storage to a table dynamicly. When the allocated space is full, it is expanded by allocating more space, when large part of the space is empty the space allocated to the table shrinks.

Table-Insert - add one item to the table.

Table-Delete - delete a given item from the table.

We analyze a heuristics that uses space allocations in powers of 2.

$size[T] =$ current size of the table.

$num[T] =$ number of items currently in the table.

Initially $size[T] = num[T] = 0$.

The cost of expanding/contraction of the table equal the number of elements in the table at the time of the operation.

Potential Function Analysis for Expansion Only

Double the size of the table when inserting to a full table.

The cost of expanding the table equal the number of elements in the table at the time of expansion.

$\Phi(T) = 2 \cdot num[T] - size[T]$

$\Phi(T) = 0$ initially, and right after expansions.

Since at least half of the table is always full $\Phi(T) \geq 0$.

Let mun_i, $size_i$ and Φ_i be the values after the i-th operation:

If the i-th step doesn't trigger an expansion;

$$\begin{aligned} A(c_i) &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) \\ &= 1 + (2num_i - size_i) - (2(num_i - 1) - size_{i-1}) \\ &= 3 \end{aligned}$$

If the i-th step triggers an expansion;

$$\begin{aligned} A(c_i) &= c_i + \Phi_i - \Phi_{i-1} \\ &= num_i + (2num_i - size_i) - (2num_{i-1} - size_{i-1}) \\ &= num_i + (2num_i - 2(num_i - 1)) \\ &\qquad -(2(num_i - 1) - (num_i - 1)) \\ &= 3 \end{aligned}$$

Thus the amortized work per step is $O(1)$.

Potential Function Analysis for Expansion and Contraction

Let $\alpha(T) = \frac{num[T]}{size[T]}$.

Double the size of the table when inserting to a full table ($\alpha[T] = 1$), halve the table size when $\alpha[T] < 1/4$.

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

$\Phi(T) \geq 0$.

Let α_i be the value of $\alpha[T]$ after operation i.

Table-Insert

If $\alpha_{i-1} \geq 1/2$ then as before $A(c_i) = 3$.

If $\alpha_{i-1} < 1/2$ AND $\alpha_i < 1/2$

$$\begin{aligned} A(c_i) &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) \\ &= 0 \end{aligned}$$

If $\alpha_{i-1} < 1/2$ AND $\alpha_i \geq 1/2$

$$
\begin{aligned}
& A(c_i) = \\
= \quad & c_i + \Phi_i - \Phi_{i-1} \\
= \quad & 1 + (2num_i - size_i) - (size_{i-1}/2 - num_{i-1}) \\
= \quad & 1 + (2(num_{i-1} + 1) - size_{i-1}) \\
& \qquad\qquad -(size_{i-1}/2 - num_{i-1}) \\
= \quad & 3num_{i-1} - \frac{3}{2}size_{i-1} + 3 \\
= \quad & 3\alpha_{i-1}size_{i-1} - \frac{3}{2}size_{i-1} + 3 \\
< \quad & 3
\end{aligned}
$$

Thus, amortized cost of Table-Insert is $O(1)$.

Table-Delete

$num_i = num_{i-1} - 1$, and if $\alpha_{i-1} < 1/2$ there might be a contraction.

If there is no contraction:

$$\begin{aligned} A(c_i) &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\ &= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i + 1)) \\ &= 2 \end{aligned}$$

If there is a contraction, $c_i = num_i + 1$, and $size_i/2 = size_{i-1}/4 = num_i + 1$.

$$\begin{aligned} A(c_i) &= c_i + \Phi_i - \Phi_{i-1} \\ &= (num_i + 1) + (size_i/2 - num_i) \\ &\qquad -(size_{i-1}/2 - num_{i-1}) \\ &= (num_i + 1) + ((num_i + 1) - num_i) \\ &\qquad -((2num_i + 2) - (num_i + 1)) \\ &= 1 \end{aligned}$$

If $\alpha_i > 1/2$ no contraction.

In all cases $A(c_i) = O(1)$.

Data Structures for Disjoint Sets

Maintain a **Dynamic** collection of disjoint sets.

Each set has a unique representative (an arbitrary member of the set).

Make-Set(x) - Create a new set with one member x.

Union(x, y) - Combine the two sets, represented by x and y into one set.

Find-Set(x) - Find the representative of the set containing x.

Computing Connected Components

Given a graph $G = (V, E)$ compute the connected components of G.

Algorithm Connected-Components(G);

for each vertex $v \in V$ **do**
Make-Set(v);
for each edge $(u, v) \in E$ **do**
If Find-Set(u) $\neq$ Find-Set(v) **then** Union(u, v);

Linked-List Representation

Each set is represented as a linked list. The head of the list is the representative element.

Each item in the list has a pointer to the next element in the list and to the head of the list.

Make-List and Find-List take $O(1)$ steps.

Union is implemented by appending one list to the other. The pointers of all the appended elements need to be updated.

m Union operation can take $O(m^2)$ time.

$O(m)$ amortized cost.

Weighted Union

A representative stores the size of its list.

In Union(x, y) we always append the smaller set to the larger one.

Theorem 1. *Any m operations on a collection of sets with a total of up to n elements takes $O(m + n \log n)$ operations.*

Proof.

We bound the number of times a pointer of an element is updated.

When a pointer is updated the element at least doubles the size of its set.

No more than $\log n$ updates per element. $\square$