Amortized Analysis

Adam W. Bargteil

February 17, 2022

In an **amortized analysis**, we average the tiem required to perform a sequence of data-structure operations over all the operations performed. Even if a single operation may be expensive, we can sometimes show that the average cost is small.

Amortized analysis differs from *average-case* analysis; an amortized analysis give the average cost of each operation in the *worst case.*

Aggregate analysis: we show that for all n, a sequence of n operations takes *worst-case* time $T(n)$ total. The **amortized cost**, per operation is $T(n)/n$.

Stack: To Push(S, x) and Pop(S) add MultiPop(S, k).

```
MultiPop $(S, k)$
1: while not Stack-Empty($S$) and $k > 0$ do
2:     Pop($S$)
3:     $k = k - 1$
4: end while
```

MultiPop could take $O(n)$ time in the worst case. Naive analysis says there are n operations each takes $O(n)$ time in the worst case, so total time $O(n^2)$, giving $n^2/n = O(n)$ average time.

However, even though a single MultiPop operation can be expensive, any sequence of n Push, Pop, MultiPop operations on an initially empty stack can cost at most O(n). We can pop each element fromt the stack at most once for each time we push it. Therefore, the number of times Pop is called, including calls from MultiPopis at most the number of Pushoperations, which is at most n. For any value of n, any sequence of n Push, Pop, and MultiPopoperations takes a total of $O(n)$ time. The average cost of an operation is $O(n)/n = O(1)$.

Note that we did not use probablistic reasoning, the $O(n)$ bound is *worst-case.*

Incrementing a binary counter: Implement a k-bit binary counter that counts upward from 0.

```
Increment $(A)$
1: while $i < A.length$ and $A[i] == 1$ do
2:     $A[i] = 0$
3:     $i = i + 1$
4: end while
5: if $i < A.length$ then
6:     $A[i] = 1$
7: end if
```

Naive analysis: A single execution of Increment takes time $\Theta(k)$ in the worst case, in which A contains all 1s. Thus, a sequence of n Increment operations on an initially zero counter takes time $O(nk)$ in the worst case. Again the bound is sound, but not tight.

Increment does not always flip all bits. $A[0]$ does flip each time Increment is called, but $A[1]$ flips only every other time, giving $\lfloor n/2 \rfloor$ flips. $A[2]$ flips every fourth time, or $\lfloor n/4 \rfloor$ times in a sequence of n Increment operations. For $i = 0, 1, \ldots, k-1$, bit $A[i]$ flips $\lfloor n/2^i \rfloor$ times in a sequence of n Increment operations on an initially zero counter. The total number of flips is thus.

$$\sum_{i=0}^{k-1} \lfloor \frac{n}{2^i} \rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n.$$

The worst-case time for a sequence of n Increment operations on an initially zero counter is therefor $O(n)$. The average cost of each operation, and therefore the amortized cost per operation is $O(n)/n = O(1)$.

The Accounting Method: Assign differing charges to different operations, with some operations being charged more or less than they actually cost. We call the amount we charge an operation its amortized cost. When amortized cost exceeds actual cost, we assignt he difference to specific objects in the data structure as **credit**. Credit pays for operations whose amortized cost is less than their actual cost. We must ensure that the total amortized cost of a sequence of operations provides an upper bound on the total actual cost of the sequence. This must be true for all sequences of operations. More precisely, if the actual cost of the ith operation is c_i and the amortized cost is $\hat{c}_i$ then,

$$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i$$

for all sequences of n operations. The total credit stored in the data structure is the difference between the total amortized cost and the total actual cost, or $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i$. So, the total credit must be nonnegative at all times or there would be a sequence of operations that would violate the inequality.

Stack Operations Actual costs: Push= 1, Pop= 1, MultiPop= $\min(k, s)$ (where s is stacksize). Amortized: Push= 2, Pop= 0, MultiPop= 0. Analogy of paying \$1 to Push and adding a credit of \$1 when pushing an item onto a stack. When we Pop the item, we use the \$1 credit. When we MultiPop, we use multiple credits. We “prepay” for the Pop and MultiPop. Each item in the stack has a \$1 credit, so the balance is always nonnegative. Thus, for *any* sequence of n Push, Pop, MultiPop operations, the total amortized cost is an upper bound on the total actual cost. The total amortized cost is $O(n)$.

Incrementing a binary counter Pay \$2 to set a bit to 1. Use \$1 to set the bit and leave a \$1 credit to be used later to reset to 0. The number of 1s never becomes negative. So amortized cost $\geq$ actual cost. Amortized cost is at most $2\times$ the number of calls to Increment bacause Increment only sets one bit to 1. Thus, for n Increment operations, the total amortized cost is $O(n)$, which bounds the total actual cost.

The potential method: Instead of representing prepaid work as credit stored with specific objects in the data structure, the **potential method** of amortized analysis represents teh prepaid work as “potential energy,” or just “potential,” which can be released to pay for future operations. We associate the potential with the data structure as a whole, rather than with specific objects within the data structure.

We perform n operations, starting with an initial data structure D_0. For each $i = 1, , 2, \ldots, n$, we let c_i be the actual cost of the ith operation and D_i be the data structure that results from the ith operation. A **potential function** $\Phi : D_i \to \mathbb{R}$ is the potential associated with D_i. The **amortized cost** $\hat{c}_i$ of the ith operation with respect to potential function Φ is defined by

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

The amortized cost of each operation is its actual cost plus the change in potential.

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n}(c_i + \Phi(D_i) - \Phi(D_{i-1})) \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(0).$$

If we can define a potential function Φ so that $\Phi(D_n) \geq \Phi(D_0)$, then the total amortized cost $\sum_{i=1}^{n} \hat{c}_i$ gives an upper bound on the total actual cost $\sum_{i=1}^{n} c_i$.

Given arbitrary sequences, we must require that $\forall i, \Phi(D_i) \geq \Phi(D_0)$. This guarantees that we "pay in advance."

If the change in potential is positive, its an overcharge for the ith operation, if it is negative, it is an undercharge and the change in potential "pays for" the actual cost of the operation.

The costs depends on the choice of potential function. The best potential function to use depends on the desired time bounds.

Stack Operations We define the potential function to be the number of items on the stack. $\Phi(D_0) = 0$ and $\forall i, \Phi(D_i) \geq 0$. So the total amortized cost of the various stack operations with respect to Φ represents an upper bound on the actual cost.

Amortized cost. If the ith operation on a stack containing s objects is a Push, then the potential difference is

$$\Phi(D_i) - \Phi(D_{i-1} = (s+1) - s = 1.$$

The amortized cost of this Pushis

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + 1 = 2.$$

If the ith operation is MultiPop(S, k) which causes $k' = \min(k, s)$ Popoperations the actual cost is k'. and the potential difference is

$$\Phi(D_i) - \Phi(D_{i-1}) = -k'.$$

Thus, the amortized cost of MultiPopis

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = k' - k' = 0.$$

Popis also 0

The amortized cost of each operation is $O(1)$ and so the total amortized cost of a sequence of n operation is $O(n)$. Since $\Phi(D_i) \geq \Phi(D_0)$, the total amortized cost of n operations is an upper bound on the total actual cost. The worst-case cost of n operations is therefore $O(n)$.

Incrementing binary counter Potential is b_i, the number of 1s in the counter after the ith operation.

Suppose the ith operation resets t_i bits. The actual cost is at most $t_i + 1$, since in addition to resetting t_i bits it sets at most one bit to 1. (If $b_i = 0$, then the ith operation resets all k bits, and so $b_{i-1} = t_i = k$; if $b_i > 0$, then $b_i = b_{i-1} - t_+ 1$.) $b_i \leq b_{i-1} - t_i + 1$, potential difference is

$$\Phi(D_i) - \Phi(D_{i-1}) \leq (b_{i-1} - t_i + 1) - b_{i-1} = 1 - t_i.$$

The amortized cost is therefore

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2.$$

If the counter starts at zero, then $\Phi(D_0) = 0$. Since $\Phi(D_i) \geq 0 \forall u$, the total amortized cost of a sequence of n INCREMENToperations is an upper bound on the total actual cost, and so the worst-case cost of n INCREMENToperations is $O(n)$.

This approach allows us to examine the case when the counter does not start at zero. We start with b_0 1s and after n INCREMENToperations b_n 1s, where $0 \leq b_0, b_n \leq k$.

$$\sum_{i=1}^{n} c_i = \sum_{i=1}^{n} \hat{c}_i - \Phi(D_n) + \Phi(D_0).$$

We have $\hat{c}_i \leq 2 \forall 1 \leq i \leq n$. The total cost of n INCREMENToperations is

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} 2 - b_n + b_0 = 2n - b_n + b_0.$$

Since $b_0 \leq k$ as long as $k = O(n)$, the total actual cost is $O(n)$. So, if we execute at least $n = \Omega(k)$ INCREMENToperations, the total actual cost is $O(n)$.

Dynamic Tables Dynamically expanding and contracting a table. TABLE-INSERT and TABLE-DELETE. **load factor** $\alpha(T) =$ number of items stored divided by size (number of **slots**). Empty table has 0 slots and $\alpha = 1$. If α is bounded below by a constant than the unused space is never more than a constant fraction of the total space.

Table **expand**. $T.num$ is the number of items in the table and $T.size$ is the allocated space.

```
TABLE-INSERT $(T, x)$
 1: if $T.size == 0$ then
 2:     allocate $T.table$ with 1 slot
 3:     $T.size = 1$
 4: end if
 5: if $T.num == T.size$ then
 6:     allocate $new - table$ with $2 \cdot T.size$ slots
 7:     insert all items in $T.table$ into $new - table$
 8:     free $T.table$
 9:     $T.table = new - table$
10:     $T.size = 2 \cdot T.size$
11: end if
12: insert $x$ into $T.table$
13: $T.num = T.num + 1$
```

Two "insertion" procedures: TABLE-INSERT and the **elementary insertion** in line 7 and 12. We analyze the running time based on the elementary insertions, assigning a cost of 1 to each. Line 6-10 are called an **expansion**.

Analyze a sequence of n TABLE-INSERT operations on an initially empty table. c_i the cost of the ith insertion. If the current table has room, then $c_i = 1$. If the current table is full then $c_i = i$. Cost of insertion plus cost of copy. Naive analysis: n operations, worst case of each $O(n)$ total upper bound for n operations is $O(n^2)$.

Aggregate analysis.

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

The total cost is therefore

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

The amortized cost of a single operation is at most 3.

We can gain intuition through the accounting method. We charge \$3 for each insertion. Intuitively, each item pays for 3 elementary operations: inserting itself into the current table, moving itself when the table expands, and moving another item that has already been moved once when the table expands. Suppose that the size of the table is m immediately after an expansion. Then the table holds $m/2$ items and no credit. We charge \$3 for each insertion. The elementary insertion costs \$1, we place another \$1 credit on the item inserted. We place the third \$1 as credit on one of the $m/2$ items already in the table. The table will not fill again until we insert another $m/2 - 1$ items; by the time the table contains m items and is full, we will have placed a \$1 on each item to pay to reinsert it during the expansion.

Using the potential method we set

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

Immediately after expansion, we have $T.num = T.size$ and thus $\Phi(T) = T.num$. The initial value of the potential is 0, and since the table is always at least half full, $T.num \ge T.size/2$, so $\Phi(T)$ is always nonnegative.

Let num_i denote the number of items stored in the table, $size_i$ denote the total size of the table, and Φ_i the potential; all after the ith insertion. If the ith insertion does not trigger an expansion, we have:

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} = 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) = 1 + (2 \cdot num_i - size_i) - (2(num_i - 1) - size_i) = 3.$$

If an expansion is triggered, then $size_i = 2 \cdot size_{i-1}$, $size_{i-1} = num_{i-1} = num_i - 1$, and $size_i = 2 \cdot (num_i - 1)$.

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} = num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1})$$

$$= num_i + (2 \cdot num_i - 2 \cdot (num_i - 1)) - (2(num_i - 1) - (num_i - 1)) = num_i + 2 - (num_i - 1) = 3.$$

Table expansion and contraction Two goals:

- The load factor is bounded below by a positive constant
- the amortized cost of a table operation is bounded above by a constant

If we double the table on expansion, should we contract when the load factor goes below 0.5? No, consider a sequence on a T where the number of operations n is a power of 2. The first $n/2$ operations are insertions, resulting in $T.num = T.size = n/2$. The rest of the operations follow the pattern "insert, delete, delete, insert, insert, delete, delete, insert, insert,..." The first insertion causes an expansion, the following two deletes cause a contraction, the following two insertion cause an expansion, etc. There are $\Omega(n)$ expansions and contractions, each costing $\Omega(n)$.

Intuitively, after expanding the table, we do not delete enough items to pay for a contractions. Likewise, after contracting, we do not insert enough to pay for an expansion.

Instead we only contract when the load factor is less than $1/4$.

We need a potential function that is 0 at a load factor of $1/2$. As the load factor deviates from $1/2$ the potential increases so that when we perform an expansion or contraction there is enough potential to cover the operation. So, the potential needs to be $T.num$ whenever the load factor reaches 1 or $1/4$. After an expansion or contraction, the load factor goes to $1/2$ and the potential goes to 0.

Denote the load factor of table T by $\alpha(T) = T.num/T.size$. Since for an empty table $T.num = T.size = 0$ and $\alpha(T) = 1$, we always have $T.num = \alpha \cdot T.size$. Let

$$\Phi(T) = \begin{cases} 2 \cdot T.num - T.size & \text{if} \alpha(T) \ge 1/2 \\ T.size/2 - T.num & \text{if} \alpha(T) < 1/2 \end{cases}$$

Observe that the potential of an empty table is 0 and that the potential is never negative. Thus, the total amortized cost of a sequence of operations with respect to Φ provides an upper bound on the actual cost of the sequence.

- When $\alpha = 1, T.size = T.num \implies \Phi(T) = T.num$, and expansion can be paid for.
- When $\alpha = 1/4, T.size = t \cdot T.num \implies \Phi(T) = T.num$, and contraction can be paid for.

After the ith operation let:

- c_i actual cost
- $\hat{c}_i$ amortized cost wrt Φ
- num_i items in the table
- $size_i$ total size of the table
- $alpha_i$ load factor
- Φ_i potential

Initially, $num_0 = 0, size_0 = 0, \alpha_0 = 1, \Phi_0 = 0$.

For $\alpha_{i-1} \geq 1/2$ analysis is identical to above. If $\alpha_{i-1} < 1/2$, the table cannot expand. If $\alpha_i < 1/2$ then

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} = 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1})$$
$$= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) = 0.$$

IF $\alpha_{i-1} < 1/2$ but $\alpha_i \geq 1/2$, then

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} = 1 + (2 \cdot num_i - size_i) - (size_{i-1}/2 - num_{i-1})$$
$$= 1 + (2(num_{i-1} + 1) - size_{i-1}) - (size_{i-1}/2 - num_{i-1}) = 3 \cdot num_{i-1} - \frac{3}{2} size_{i-1} + 3$$
$$= 3\alpha_{i-1} size_{i-1} - \frac{3}{2} size_{i-1} + 3 < \frac{3}{2} size_{i-1} - \frac{3}{2} size_{i-1} + 3 = 3$$

Table-Delete $num_i = num_{i-1} - 1$. If $\alpha_{i-1} < 1/2$, then we must consider whether the operation causes the table to contract. If it does not, then $size_i = size_{i-1}$ and

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1})$$
$$= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i + 1)) = 2$$

If $\alpha_{i-1} < 1/2$ and the ith operation does trigger a contraction, then the actual cost is $c = num_i + 1$, since we delete one item and move num_i items. We have $size_i/2 = size_{i-1}/4 = num_{i-1} = num_i + 1$ and

$$\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} = (num_i + 1) +) size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1})$$
$$= (num_i + 1) + ((num_i + 1) - num_i) - ((2 \cdot num_i + 2) - (num_i + 1)) = 1.$$