1 September 5th, 2019

1.1 Amortization Analysis

A lot of time, with the right data structure, algorithm design can be made easier, such as **heapsort**. The analysis of these algorithms are also usually a lot easier, e.g. in heapsort, Insert and Extract-Min are both $O(\log n)$, making heapsort $O(n \log n)$.

The default analysis is worst-case analysis. However, sometimes the worst case is too pessimistic.

Example 1.1
We have a stack with the two operations:
- PUSH(S, x)
- POP(S)

Suppose we want to add a new operation

$$\text{MULTIPOP}(S, k).$$

Which just pops k times or until the stack is empty.

The worst case cost for MULTIPOP in the sequence is $O(n)$, since the stack size is at most n, making the cost of the sequence $O(n^2)$. This is correct, but we can improve it, since we know the structure of the data structure.

Intuitively, if we make n operations, we cannot have all taking $O(n)$ time. Observing that an object can only POPed once, we have the that the number of POPs is at most the number of PUSHs, which is at most n.

As such, the **amortized** cost of an operation is $O(n)/n = O(1)$.

Remark 1.2 — The reason why we are able to improve the bound is because we know more information about the inner workings of the data structure, otherwise if we only know the interface, $O(n^2)$ the only thing we can have.

Definition 1.3. The **amortized cost** of an operation on a data structure is:

$$A(n) = \frac{1}{n} \max_{\sigma, |\sigma| = n} T(\sigma),$$

where the max is taken over all sequences σ of n such operations, and $T(\sigma)$ is the total running time over σ.

Remark 1.4 — In other words, we are finding the longest running time of any σ and dividing per operation.

Definition 1.5. When there are k types of operations, op_i, they have amortized costs $A_i(n)$, respectively, if for any $\sum_i n_i = n$,

$$n_1A_1(n) + n_2A_2(n) + \ldots + n_kA_k(n) \geq \max_{\sigma} T(\sigma).$$

Where the max is taken over all sequences σ of n operations that consists of n_i op_i's.

Example 1.6

Suppose we have a binary counter A, initialized to all 0. We have the operation INCREMENT(A), which increments the counter to the next number by changing the bits.

Note that a single execution of INCREMENT takes $O(\log n)$ time in the worst case (when A contains all 1's), giving it a $O(n \log n)$ bound in the worst case. However, note that the total number of flips is:

$$\sum_{i=0}^{\log n - 1} \frac{n}{2^i} = O(n).$$

Making the amortized cost $O(1)$.

Remark 1.7 — The reason why people use amortized instead of average, since average is reserved for the **average case analysis**, which is over random input sequences. While amortized analysis has no randomness.

This is also contrasted with **expected case analysis**, as it is the average running time with internal random numbers on the worst input.

There are three methods for amortized analysis:

- **Aggregate Analysis**
  - Total cost of n operations divided by n.
- **Accounting Method**
  - Assign each type of operation with a (possibly different) amortized cost.
  - We should overcharge some operations and store overcharge as credit on specific objects.
  - Then we will use the credit for compensation for some later operations.
- **Potential Method**
  - Same as the accounting method, but we don't associate the credit with specific objects.

1.2 Accounting Method

Let's apply the accounting method to the binary counter:

We will show that the amortized cost of each INCREMENT is \$2.

- Let \$1 represent each unit of cost (i.e., the flip of one bit).

- Whenever a bit flips from 0 to 1, we spend \$2: \$1 for the actual cost, and store another \$1 on the 1-bit as credit.
- When a bit is set to 0, the stored \$1 pays the cost.
- At most one bit flips from 0 to 1 in each INCREMENT operation, os the amortized cost of \$2 is enough to cover all the flips.

Remark 1.8 — The summary of the accounting method is:
- Come up with a conjecture of the amortized cost of each type of operation.
- For some operations, the amortized cost will be greater than the actual cost.
- In others, the amortized cost will be less than the actual cost.
- If we have a surplus, we attached the surplus to some object in the data structure. This allows us to use the invariant argument.

Example 1.9

Actual cost:
- PUSH: 1
- POP: 1
- MULTIPOP: k

Let's assign the following amortized costs:
- PUSH: 2
- POP: 0
- MULTIPOP: 0

Suppose \$1 represent a unit of cost. When pushing a plate, we use \$1 to pay the actual cost and leave \$ on the plate as credit.

When POPing a plate the \$1 on the plate is used to pay the actual cost of the POP (note that al plates must have a dollar from when they are pushed).

Remark 1.10 — The reason why we can assign 0 cost to some operation is because for amortized costs, we always consider **sequences of operations**.

1.3 Potential Method

There isn't a difference between accounting method and potential method, the only difference is that **we don't have to attach the surplus to an object**.

Note that the accounting method argument still works if we have enough money, we don't have to associate the money with each plate.

Remark 1.11 — The name **potential** comes from physics, where we inject energy and extract to do things.

Definition 1.12. We have:
- Initial data structure D_0 and n operations, resulting in $D_0, D_1, \ldots, D_n$ with costs $c_1, c_2, \ldots, c_n$.
- A potential function $\phi : \{D_i\} \to \mathbb{R}$ ($\phi(D_i)$ is called the potential of D_i).

- Amortized cost c_i' of the i-th operation is:

$$c_i' = c_i + \phi(D_i) - \phi(D_{i-1}), \quad \text{(the actual cost + potential change)}.$$

- $\phi(D_i) \geq 0$

In the end we need to show that:

$$\sum_i c_i' \geq \sum_i c_i.$$

In other words:

$$\sum_{i=1}^{n} c_i + \phi(D_n) - \phi(D_0).$$

We can do this by showing that:

$$c_i' \geq c_i + \phi(D_i) - \phi(D_{i-1}).$$

In general we start by defining the potential. This is the key to a potential proof, as we just need to show that the equality holds.

Example 1.13

We define the potential for a stack to be the number of objects in the stack. So:

$$\phi(D_0) = 0, \quad \text{and } \phi(D_i) \geq 0.$$

Example 1.14

For the binary counter, we define the potential of the counter to be the number of 1's.

Remark 1.15 — In the potential method, we don't guess the amortized cost, instead we guess the potential.

We will now see an example of amortized analysis with the following scenario:

- A table (e.g. an array, not linked list)
- We do not know how large in advance
- Insertion at the end: $O(1)$ time if there is space
- Deletion at the end: $O(1)$ time
- Rebuilding the table takes time linear to the table size

Goal:

- $O(1)$ amortized cost
- Load factor $\alpha \geq$ constant fraction

Definition 1.16. The Load factor is the fraction of elements vs size of table

One natural strategy is the doubling strategy: when the table becomes full, double its size and rebuild. This guarantees $\alpha \geq \frac{1}{2}$.

With the aggregate analysis:

- Total cost of n insertions: $O(n)$
- Total cost of all rebuilds: $1 + 2 + 4 + 8 + \ldots + n = O(n)$

Thus is has the amortized cost of $O(1)$.

However, this does not work when we want to support deletion. One natural strategy is as follows:

- Double the size when inserting to a full table
- Halve the size when less than half full
- This guarantees that $\alpha \in [\frac{1}{2}, 1]$.

However consider where we:

- Fill the table
- insert (doubling the table)
- 2 deletes (halving the table)
- 2 inserts (doubling the table)
- $\vdots$

This would give us a cost $\Theta(n^2)$. The problem is that we don't perform enough operations after expansion or contraction to pay for the next one.

To fix this, we have a buffer, where we only halve when $\alpha \leq \frac{1}{4}$. However aggregate analysis might run into problems, since we have to consider both insertion and deletion. As such, we must use the accounting or potential method.

1.3.1 Dynamic Table: Accounting Method

- Suppose the table size is m after last expansion/contraction
- Charge each insertion/deletion an amortized cost of 5 (4+1), which pays for the actual cost of the insertion/deletion, and stores 4 extra credits.
- Table expands/contracts again after a least $\frac{1}{2}m$ operations, so we have $2m$ credits. This is because it takes at at least m insertions to double, and $\frac{1}{2}m$ deletions to halve.
- These $2m$ credits will be enough to cover the next expansion/contraction ($2m$ to expand, m to contract).

Remark 1.17 — Note that there is still very little math needed. However, we need to find the constants. Usually, we just use asymptotic notation, e.g.

- Suppose the table size is m after last expansion/contraction
- Charge each insertion/deletion an amortized cost of $O(1)$ (sufficiently large), which pays for the actual cost of the insertion/deletion, and stores $O(1)$ (sufficiently large) extra credits.
- Table expands/contracts again after a least $\Omega(m)$ (some given constant) operations, so we have $\Omega(m)$ (sufficiently large, its $O(1) \times \Omega(m)$) credits.
- These $\Omega(m)$ credits will be enough to cover the next expansion/contraction which costs $O(m)$.