Amortized Analysis

Even though the worst case cost of a single operation is large, the average cost of an operation (averaged over a sequence of operations) can be small. Amortized analysis guarantees the average performance of each operation in the worst case.

Aggregate Analysis

We show, for all n, a sequence of n operations takes worst case $T(n)$ time in total. Then, the amortized cost per operation is $T(n)/n$.

Stack Operations

- PUSH(S, x): $O(1)$
- POP(S): $O(1)$
- MULTIPOP(S, k): $O(\min(s, k)) = O(n)$

By worst case analysis, we can say that the worst case complexity of any stack operation is $O(n)$, and thus, the worst case running time of n operations will be $O(n^2)$. However, this is not a tight upper bound for the running time of n operations.

By aggregate analysis, we can observe, that we can pop an object at most once, for each time we have pushed to the stack. Since there are n operations, there can be at most n pushes to the stack, so there can be at most n pops from the stack (including those due to MULTIPOP calls). Thus, the total running time can be bounded by $T(n) = O(n)$.

The average cost of an operation therefore, is $O(n)/n = O(1)$. So we say, all three stack operations have a constant amortized cost.

What does this all mean?

Let $\langle \text{op}_1, \text{op}_2, \ldots, \text{op}_n \rangle$ be a sequence of n operations, with

- c_i = actual cost of op_i
- $\hat{c}_i$ = *amortized cost* of op_i
- $T(n) = \sum c_i$ = total running time of n operations.

In **amortized analysis**, we assign values to $\hat{c}_i$ such that the following inequality holds:

$$T(n) = \sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \hat{c}_i$$

or in words: **For any sequence of n operations, the total running time must be upper bounded by the sum of the amortized costs of the individual operations.**

For some operation op_i, $\hat{c}_i$ is its *amortized cost*. Note that we do not make any claims about the running time of an individual operation. The actual running time of an operation (c_i) can be greater than the amortized cost $(\hat{c}_i)$. But the guarantee we do have, is that on performing n operations, the total actual running time $(T(n))$ will be bounded by the total amortized costs.

In case of **aggregate analysis**, we assign $\hat{c}_i = T(n)/n$. This is valid, because the inequality holds trivially: $\sum_{i=1}^{n} \hat{c}_i = n(T(n)/n) = T(n) \geq T(n)$.

Note that, aggregate analysis assigns the same *amortized cost* to all operations, even if there are operations of different types. Other methods of finding amortized cost may assign different costs to different types of operations, as we will see later.

Incrementing a binary counter

Consider a k-bit binary counter that counts up from 0, with the following INCREMENT procedure:

INCREMENT(A)

```
1  i = 0
2  while i < A.length and A[i] == 1
3      A[i] = 0
4      i = i + 1
5  if i < A.length
6      A[i] = 1
```

The cost of an INCREMENT operation is linear in the number of bits flipped. In the worst case, a single run of INCREMENT can take $\Theta(k)$ (when A contains all 1s), so a sequence of n INCREMENT operations on an initially zero counter takes time $O(nk)$ in the worst case.

We can tighten our analysis to get a worst case cost of $O(n)$ for a sequence of n increment operations by observing a pattern in when a bit flip occurs:

- $A[0]$ flips every time INCREMENT is called.
- $A[1]$ flips every other call, so this bit will flip $\lfloor n/2 \rfloor$ times in a sequence of n INCREMENT calls.
- ...
- $A[i]$ will flip $\left\lfloor n/2^i \right\rfloor$ times in a sequence of n INCREMENT calls.

So, the total number of bit flips that happen in n INCREMENT calls is:

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor \leq \sum_{i=0}^{\infty} \frac{n}{2^i} = 2n$$

Therefore, the worst case time for a sequence of n increment operations (on a initially 0 counter) is $O(n)$, so the amortized cost for each operation is $O(n)/n = O(1)$.

Accounting Method

In this method, we assign differing charges to different operations, with some operations charged more/less than their actual cost. This hypothetical charge is called the **amortized cost**. The difference in an operation's amortized cost and actual cost is assigned as **credit** which can be used later when the amortized cost is less than the actual cost.

We require, that the following relation holds for any sequence of n operations:

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

or in other words, the total credit stored in the data structure must be non negative at all times.

This ensures that our sum of amortized costs does bound the total running time of n operations, thus our assigned amortized costs are valid.

Stack Operations

Recall that the actual costs of the operations are (in terms of number of elementary operations)

- PUSH: 1
- POP: 1

- MULTIPOP: $\min(s, k)$

Consider a sequence of n stack operations. We need to assign $\hat{c_i}$ values to each type of operation, so that $\sum_{i=1}^{n} \hat{c_i} - \sum_{i=1}^{n} c_i$ is **always** nonnegative.

We may use the following observation: Any element that has been pushed can be popped at most once (either by the POP or the MULTIPOP call). So, the total actual cost can never be more than twice the number of PUSH calls in the sequence.

So, we assign these amortized costs:

- PUSH: 2
- POP: 0
- MULTIPOP: 0

We can show that we can pay for any sequence of stack operations by charging these amortized costs. We start with an empty stack. Whenever we push an element we charge 2 units, we pay 1 unit to pay for the push, and we are left with 1 unit of credit. So every element in the stack holds 1 unit of credit which serves as the cost for popping it later.

When we execute a POP operation, we don't need to charge anything, and pay for the pop by using the credit associated to that element on the stack. Similarly for multipop, its actual cost is the number of elements removed, which we pay using the credits on those elements. Therefore, we have always charged enough to pay for any sequence of n operations, hence the total amortized cost is an upper bound on the total actual cost.

Thus we can conclude, that PUSH, POP, MULTIPOP have an $O(1)$ amortized cost.

Incrementing a binary counter

We only have one type of operation (INCREMENT). We need to assign some amortized cost to this operation, so that any sequence of n operations can be paid for, by charging the amortized cost.

Initially the counter is 0. The cost of an operation is linear in the number of bits flipped. Observe, that if a bit is being flipped from $1 \to 0$, then it must have been flipped from $0 \to 1$ before at some point. So, let's charge 2 units for every $0 \to 1$ flip, and 0 units for every $1 \to 0$ flip. By doing so, every $1 \to 0$ flip can be paid for by the credit stored due to its preceding $0 \to 1$ flip.

In each operation, observe that there is at most 1 flip which turns a 0 to a 1. So we can assign an amortized cost of 2 units for each operation. By doing so, a sequence of n operations can be paid for as

- We charge 2 units for each operation, 1 unit is used for the bit flip from 0 to 1, and 1 unit is stored as a credit.
- Each 1 bit in the counter holds 1 unit of credit. (invariant)
- Any 1 to 0 bit flips are paid for by the credits held by the corresponding bit.

Therefore, we conclude that the cost of INCREMENT is $O(1)$ amortized.

Potential Method

In this method, we associate the state of the data structure D_i with a value $\Phi(D_i)$ known as its **potential**, and Φ is called the **potential function**.

Consider a sequence of n operations, the ith operation has cost c_i. Let the state of the data structure be D_i after the ith operation, and its initial state be D_0. We define the amortized cost $\hat{c_i}$ values as follows:

$$\begin{aligned}\hat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= c_i + \Delta\Phi(D_i)\end{aligned}$$

We need,

$$\begin{aligned}\sum_{i=1}^{n} \hat{c_i} &\geq \sum_{i=1}^{n} c_i \\ \Rightarrow \sum_{i=1}^{n} (\hat{c_i} - c_i) &\geq 0 \\ \Rightarrow \sum_{i=1}^{n} \Delta\Phi(D_i) &\geq 0 \\ \Rightarrow \Phi(D_n) &\geq \Phi(D_0)\end{aligned}$$

WLOG we can take $\Phi(D_0) = 0$. Since we require that for any sequence of n operations, the sum of amortized costs should be at least the sum of total costs, so we can establish a stronger condition:

$$\Phi(D_i) \geq 0 \text{ (for all } i)$$

Intuitively, if the potential difference $\Delta\Phi(D_i)$ of the ith operation is positive, then the amortized cost $\hat{c_i}$ represents a overcharge. If the potential difference is negative, amortized cost represents a undercharge, and the change in potential pays for the actual cost of the operation.

Stack operations

Define the potential of a stack to be the number of elements in the stack. For the empty stack D_0, we have $\Phi(D_0) = 0$ naturally. Since the number of elements in the stack is always non negative, we have $\Phi(D_i) \geq 0$ for all i.

We can now compute the amortized costs of each operation.

- PUSH:

$$\begin{aligned}\hat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + 1 = 2\end{aligned}$$

- POP:

$$\begin{aligned}\hat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 - 1 = 0\end{aligned}$$

- MULTIPOP:

$$\begin{aligned}\hat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k - k = 0\end{aligned}$$

So the amortized costs of all the operations are constant, and the worst case cost of n operations is therefore $O(n)$.

This choice of potential is apt, because when we perform an expensive operation (MULTIPOP), the change in potential is negative (number of elements in the stack decrease), and the decrease in potential pays for the operation (cost of the operation is covered by the decreased potential).

Incrementing a binary counter

In this case, let us define the potential function $\Phi(D_i)$ to be the number of 1s in the counter, b_i. This is always nonnegative, and for the starting state, $\Phi(D_0) = b_0 = 0$.

To calculate the amortized cost of an increment operation, say that the number of trailing 1s in the current counter is t_i, then total bit flips is $1 + t_i$. Also note that $b_{i-1} - b_i = t_i - 1$ (the change in the number of 1s by the ith operation is 1 less than the number of trailing ones).

Then,

$$\begin{aligned}\hat{c_i} &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq 1 + t_i - (t_i - 1) = 2\end{aligned}$$

So, each INCREMENT operation is amortized constant time, and a sequence of n operations is $O(n)$ worst case.

We can also analyse this situation if the initial state is not 0. In this case, our constraint that $\Phi(D_i) \geq \Phi(D_0)$ may not hold, but still, we have the following equality (even if the counter doesn't start at 0! Why?):

$$\begin{aligned}\sum_{i=1}^{n} c_i &= \sum_{i=1}^{n} \hat{c_i} - \Phi(D_n) + \Phi(D_0) \\ \Rightarrow \sum_{i=1}^{n} c_i &\leq 2n - b_n + b_0\end{aligned}$$

So, given that $k = O(n)$, the total actual cost is $O(n)$.

Dynamic Tables

A dynamic table is a data structure which is a dynamically expanding and contracting memory block. Using amortized analysis, we can show such a data structure can have amortized constant time insertions and deletions, even though the actual costs of these operations can be large when an expansion or a contraction is triggered.

We define $\alpha(T)$ (load factor) of an non empty table as:

$$\alpha(T) = \frac{n}{m} = \frac{\text{number of items stored}}{\text{total slots}}$$

Define the load factor of an empty table to be 1.

Our dynamic table supports the following operations:

- TABLE-INSERT: inserts an item into the table, which occupies one slot.
- TABLE-DELETE: removes an item from the table, freeing a slot.

Expansion

Consider a dynamic table which only supports TABLE-INSERT calls. Assume that memory for a table is allocated as an array, and when all slots are used up, we need to **expand** the table by

reallocating memory. Heuristically, we choose to double the size of the table upon expansion; this ensures that the amount of unused space never exceeds half of the total space in the table.

We can reason about the complexity of TABLE-INSERT by the number of elementary insertions required.

Consider a sequence of n TABLE-INSERT operations. What is the cost c_i of the ith operation? If the current table has room for the new item, then $c_i = 1$. Else, expansion occurs and $c_i = i$. The worst case cost of an operation is $O(n)$, so the total running time of n operations is bounded by $O(n^2)$.

We can get a tighter bound by using aggregate analysis. Observe that the ith operation causes an expansion only when $i - 1$ is a power of 2. Thus,

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

Therefore, the total cost of n TABLE-INSERT operations is

$$T(n) = \sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2j < n + 2n = 3n$$

Thus, the amortized cost for each operation is at most 3.

We can justify this cost by the accounting method: Consider we charge 3 units per TABLE-INSERT operation. Each item pays for 3 elementary insertions: one unit for itself, one unit for moving it when the table needs to be expanded, and one for those elements which have already exhausted their credits (in the first half of the table).

We can also use the potential method to analyse a sequence of n TABLE-INSERT operations. We define a potential fuction that is 0 immediately after expansion, but is equal to the table size when it is full.

- When the table is full, that's when the next operation is the expensive operation, so we need to have high potential (so that the drop in potential can pay for the expensive operation).
- Suppose the current size of the table is m and it is full. Then, the next operation costs m units for expansion, and 1 more unit for adding the new element. So the drop in potential must at least pay for m. And at the point when we are exactly half full, we don't need to keep any additional credits (next insertions will be paid for by their charges), so we can take the potential of the half-filled table to be 0.
- Naturally, the potential of a full table of size m must be m.

This leads to the following definition of the potential:

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

Since the load factor of the table is $\geq \frac{1}{2}$, this ensures that the potential is always positive. Thus the sum of amortized costs of n TABLE-INSERT operations will give an upper bound for the sum of actual costs.

If the ith operation does not trigger an expansion, then

$$\begin{aligned} \hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + 2 \cdot \text{num}_i - \text{size}_i - 2 \cdot (\text{num}_i - 1) + \text{size}_i \\ &= 3 \end{aligned}$$

If the ith operation does trigger an expansion, then

$$\begin{aligned}\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= \text{num}_i + 2 \cdot \text{num}_i - \text{size}_i - 2 \cdot (\text{num}_i - 1) + \frac{\text{size}_i}{2} \\ &= \text{num}_i + 2 \cdot \text{num}_i - 2 \cdot (\text{num}_i - 1) - 2 \cdot (\text{num}_i - 1) + \text{num}_i - 1 \\ &= 3\end{aligned}$$

So, all operations have an amortized cost of 3, and the total cost is bounded by $O(n)$.

Expansion and Contraction

To implement TABLE-DELETE, we may simply remove the element from the table, and still we have all amortized constant time operations. However, we may wish to limit the unused space by contracting the table when the load factor becomes too small. We can achieve this by ensuring that the load factor is bounded below by a constant, and the amortized costs of the operations are constant time.

Suppose we say, that in addition to expanding the table whenever it is full, we also contract the table (to half its size) whenever the table becomes less than half full. However, this causes the amortized cost to be large. We can consider a scenario where the table is one more than half full, and then we do the following sequence operations:

delete, **delete**, insert, **insert**, delete, **delete**, insert, **insert**, ...

In this case, we can notice that every other operation (bolded) will trigger an expansion or a contraction which is an expensive operation, and takes $O(n)$ actual cost. So, the total cost of the sequence of operations will be at least $n^2/4 = \Omega(n^2)$, which cannot be bounded by a linear function. Thus, the amortized costs are strictly worse than constant time.

We can improve upon this strategy by not immediately contracting, but waiting until the load factor drops below some other threshold. Say, we half the table whenever the table becomes less than a quarter full. Then, the load factor of the table is bounded below by $1/4$.

We need to come up with a suitable potential function for this situation.

- The expensive operations are expansion and contraction, which happen when the load factor is 1 and $1/4$ respectively. The actual costs of the expensive operations is num_i in both cases.
- After performing an expansion/contraction, the table is exactly half full. So this is the state of minimum potential.
- During expansion, we need the potential drop to pay for the cost of the expansion (num_i). So when the table is completely full, the potential should be at least $\text{size}_i = \text{num}_i$.
- During contraction, we need the potential drop to pay for the cost of the contraction (num_i). So when the table is a quarter full, the potential should be at least $\text{size}_i/4 = \text{num}_i$.

These observations lead us to the following potential function:

$$\Phi(T) = \begin{cases} 2 \cdot \text{num}_i - \text{size}_i & \text{if } \alpha_i \geq 1/2 \\ \text{size}_i/2 - \text{num}_i & \text{if } \alpha_i < 1/2 \end{cases}$$

The potential of an empty table is 0, and the potential function is always nonnegative; thus the sum of amortized costs of the function give an upper bound for the worst case actual cost of the sequence of n operations.

Consider a sequence of n operations. Let us compute the amortized costs of the operations.

1. The ith operation is TABLE-INSERT.

   If $\alpha_{i-1} \geq 1/2$ (the table was at least half full before this insertion), then the next insertion may or may not trigger an expansion. The computation of $\hat{c}_i$ is exactly the same as done in the expansion-only case. So we know that the $\hat{c}_i$s are constant.

   If $\alpha_{i-1} < 1/2$ (the table was less than half full before this insertion), then this insertion couldn't have triggered an expansion.

   - If $\alpha_i \geq 1/2$, then

   $$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot \text{num}_i - \text{size}_i) - \left(\frac{\text{size}_i}{2} - \text{num}_i + 1\right) \\ &= 3 \cdot \text{num}_i - \frac{3}{2}\text{size}_i \\ &= 3 + 3 \cdot \text{num}_{i-1} - \frac{3}{2}\text{size}_{i-1} \\ &= 3 + 3\alpha_{i-1}\text{size}_{i-1} - \frac{3}{2}\text{size}_{i-1} \\ &< 3 + \frac{3}{2}\text{size}_{i-1} - \frac{3}{2}\text{size}_{i-1} = 3\end{aligned}$$

   - If $\alpha_i < 1/2$, then

   $$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + \frac{\text{size}_i}{2} - \text{num}_i - \left(\frac{\text{size}_i}{2} - \text{num}_i + 1\right) = 0\end{aligned}$$

   Again, $\hat{c}_i$ is bounded by a constant. Hence, TABLE-INSERT runs in amortized constant time.

2. The ith operation is TABLE-DELETE.

   If $\alpha_{i-1} < 1/2$ (the table was less than half full before this deletion), then the next deletion may or may not trigger an contraction.

   - No contraction ($\text{size}_i = \text{size}_{i-1}$, $\text{num}_i = \text{num}_{i-1} - 1$)

   $$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + \frac{\text{size}_i}{2} - \text{num}_i - \left(\frac{\text{size}_i}{2} - \text{num}_i - 1\right) = 2\end{aligned}$$

   - Contraction ($\text{size}_i = \text{size}_{i-1}/2$, $\text{num}_i = \text{num}_{i-1} - 1$, $\text{num}_i \leq \text{size}_{i-1}/4 = \text{size}_i/2$)

   $$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= \text{num}_i + \frac{\text{size}_i}{2} - \text{num}_i - (\text{size}_i - \text{num}_i - 1) \\ &= 1 + \text{num}_i - \frac{\text{size}_i}{2} \\ &\leq 1\end{aligned}$$

   So, the amortized cost is bounded by a constant in this case.

If $\alpha_{i-1} \geq 1/2$ (the table was at least half full before this deletion), then this deletion couldn't have triggered a contraction.

- If $\alpha_i \geq 1/2$, then

$$\begin{aligned}\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot \text{num}_i - \text{size}_i) - (2 \cdot (\text{num}_i + 1) - \text{size}_i) \\ &= -1\end{aligned}$$

- If $\alpha_i < 1/2$, then

$$\begin{aligned}\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + \left(\frac{\text{size}_i}{2} - \text{num}_i\right) - (2 \cdot (\text{num}_i + 1) - \text{size}_i) \\ &= \frac{3}{2} \cdot \text{size}_i - 3 \cdot \text{num}_i - 1 \\ &= \frac{3}{2} \cdot \text{size}_{i-1} - 3 \cdot \text{num}_{i-1} + 2 \\ &= \frac{3}{2} \cdot \text{size}_{i-1} - 3 \cdot \alpha_{i-1}\text{size}_{i-1} + 2 \\ &\leq \frac{3}{2} \cdot \text{size}_{i-1} - \frac{3}{2} \cdot \text{size}_{i-1} + 2 = 2\end{aligned}$$

Again, $\hat{c_i}$ is bounded by a constant. Hence, TABLE-DELETE runs in amortized constant time.

Problems

1. Recall the dynamic table data structure discussed in class. Suppose that instead of contracting a table by halving its size when its load factor drops below $\frac{1}{4}$, we contract it by multiplying its size by $\frac{2}{3}$ when its load factor drops below $\frac{1}{3}$. Show that insert and delete operations take $O(1)$ amortized time.

Here, expansion happens when the table is full ($\alpha = 1$), and contraction happens when the table is a third full ($\alpha = \frac{1}{3}$). After expansion/contraction, we reach a situation where the table is exactly half full. So we need to find a potential function which is 0 when $\text{num}_i = \text{size}_i/2$; num_i when the table is full; and num_i when $\text{num}_i = \text{size}_i/3$.

So, we can use the following potential function:

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

Clearly, the potential function is non negative everywhere, so the sum of amortized costs of a sequence of operations upper bounds the sum of actual costs.

Let us calculate the amortized costs for the operations.

TABLE-INSERT:

- If $\alpha_{i-1} \geq 1/2$,
  - If an expansion took place,

$$\begin{aligned}\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= \text{num}_i + (2 \cdot \text{num}_i - \text{size}_i) - (2 \cdot \text{num}_{i-1} - \text{size}_{i-1}) \\ &= \text{num}_{i-1} + 1 + (2 \cdot \text{num}_{i-1} + 2 - 2 \cdot \text{size}_{i-1}) - (2 \cdot \text{num}_{i-1} - \text{size}_{i-1}) \\ &= 3 + \text{num}_{i-1} - \text{size}_{i-1} \\ &= 3\end{aligned}$$

  - If no expansion took place,

$$\begin{aligned}\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot \text{num}_i - \text{size}_i) - (2 \cdot (\text{num}_i - 1) - \text{size}_i) \\ &= 3\end{aligned}$$

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

$$\begin{aligned}\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (\text{size}_i - 2 \cdot \text{num}_i) - (\text{size}_i - 2 \cdot (\text{num}_i - 1)) \\ &= -1\end{aligned}$$

  - If $\alpha_i \geq 1/2$,

$$\begin{aligned}\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot \text{num}_i - \text{size}_i) - (\text{size}_i - 2 \cdot (\text{num}_i - 1)) \\ &= 1 + 2 \cdot \text{num}_i - \text{size}_i - \text{size}_i + 2 \cdot \text{num}_i - 2 \\ &= 4 \cdot \text{num}_i - 2 \cdot \text{size}_i - 1 \\ &= 4 \cdot \text{num}_{i-1} - 2 \cdot \text{size}_{i-1} + 3 \\ &= 4\alpha_{i-1}\text{size}_{i-1} - 2 \cdot \text{size}_{i-1} + 3 \\ &< 2 \cdot \text{size}_{i-1} - 2 \cdot \text{size}_{i-1} + 3 = 3\end{aligned}$$

Thus, in all cases, the amortized costs is bounded by a constant. So, TABLE-INSERT is $O(1)$ amortized.

TABLE-DELETE:

- If $\alpha_{i-1} < 1/2$
  - If contraction takes place

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= \text{num}_i + (\text{size}_i - 2 \cdot \text{num}_i) - (\text{size}_{i-1} - 2 \cdot \text{num}_{i-1}) \\ &= \text{num}_{i-1} - 1 + \frac{2}{3} \cdot \text{size}_{i-1} - 2 \cdot \text{num}_{i-1} + 2 - \text{size}_{i-1} + 2 \cdot \text{num}_{i-1} \\ &= 1 + \text{num}_{i-1} - \frac{1}{3} \cdot \text{size}_{i-1} \\ &= 2 + \text{num}_i - \frac{1}{3} \cdot \text{size}_{i-1} \\ &\leq 2\end{aligned}$$

  - If no contraction takes place

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (\text{size}_i - 2 \cdot \text{num}_i) - (\text{size}_{i-1} - 2 \cdot \text{num}_{i-1}) \\ &= 1 + (\text{size}_i - 2 \cdot \text{num}_i) - (\text{size}_i - 2 \cdot \text{num}_i - 2) \\ &= 3\end{aligned}$$

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

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (2 \cdot \text{num}_i - \text{size}_i) - (2 \cdot \text{num}_{i-1} - \text{size}_{i-1}) \\ &= 1 + (2 \cdot \text{num}_i - \text{size}_i) - (2 \cdot \text{num}_i + 2 - \text{size}_i) \\ &= -1\end{aligned}$$

  - If $\alpha_i < 1/2$

$$\begin{aligned}\hat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + (\text{size}_i - 2 \cdot \text{num}_i) - (2 \cdot \text{num}_{i-1} - \text{size}_{i-1}) \\ &= 1 + (\text{size}_i - 2 \cdot \text{num}_i) - (2 \cdot \text{num}_i + 2 - \text{size}_i) \\ &= 2 \cdot \text{size}_i - 4 \cdot \text{num}_i - 1 \\ &= 2 \cdot \text{size}_{i-1} - 4 \cdot \text{num}_{i-1} + 3 \\ &= 2 \cdot \text{size}_{i-1} - 4\alpha_{i-1} \cdot \text{size}_{i-1} + 3 \\ &\leq 2 \cdot \text{size}_{i-1} - 2 \cdot \text{size}_{i-1} + 3 = 3\end{aligned}$$

Again, in all cases the amortized cost is upper bounded by a constant.

Thus, the amortized costs for both insertions and deletions is $O(1)$.

2. Design a data structure to support the following operations for a dynamic multiset S of integers, which allows duplicate values:
   - INSERT(S, x): inserts x into S.
   - DELETE-LARGER-HALF(S): deletes the largest $\lceil |S|/2 \rceil$ elements from S.

   Explain how to implement this data structure so that the amortized time complexity of INSERT and DELETE-LARGER-HALF is $O(1)$.

We use a linked list/array as the underlying data structure for storing the elements, to support $O(1)$ insertions.

- INSERT(S, x): Insert x to the linked list. ($O(1)$, actual cost: 1)
- DELETE-LARGER-HALF(S): ($O(n)$, actual cost bounded by: kn (say))
  1. Find the element at rank $\lceil n/2 \rceil$ by using linear time selection (quickselect / median of medians).
  2. Delete all elements that are greater than or equal to this pivot.

Consider any sequence of n operations. Let us assign the following charges to each operation:

- INSERT: $\hat{c_i} = 1 + 2k$
- DELETE-LARGER-HALF: $\hat{c_i} = 0$

We can show that by charging these amounts for each operation, we can pay for all operations while keeping a nonnegative credit.

For each element stored in the data structure, we keep $2k$ credit per item. When we have a DELETE-LARGER-HALF call, we can use up the $\lceil n/2 \rceil \times 2k \geq nk$ credits of the deleted nodes to perform the deletion itself. The total credits always is nonnegative since number of elements in the array is nonnegative.

Hence, both operations are amortized constant time.

3. Consider an ordinary binary min-heap data structure with n elements supporting the instructions INSERT and EXTRACT-MIN in $O(\log n)$ worst case time. Show that the amortized cost of INSERT is $O(\log n)$ and EXTRACT-MIN is $O(1)$ time.

Since every EXTRACT-MIN operation corresponds to an INSERT operation, we can charge $2 \log n$ units for each INSERT operation, so that each element present in the heap holds enough credits, to pay for the corresponding EXTRACT-MIN operation. Thus, we can assign the following amortized costs:

- INSERT: $2 \log n$
- EXTRACT-MIN: 0

Since the total credits present is always nonnegative, the sum of the amortized costs upper bounds the total actual cost for any sequence of operations. Therefore, the amortized cost of INSERT is $O(\log n)$ and EXTRACT-MIN is $O(1)$.

4. What is the total cost of executing n stack operations PUSH, POP and MULTIPOP, assuming that the stack begins with s_0 objects and ends with s_n objects?

Let us take the potential function $\Phi(D_i) = s_i$ to be the number of elements in the stack. We have amortized costs:

- PUSH: 2
- POP: 0
- MULTIPOP: 0

Then, we have the relation:

$$\sum_{i=1}^{n} \hat{c_i} = \sum_{i=1}^{n} c_i + s_n - s_0$$

$$\Rightarrow \sum_{i=1}^{n} c_i \leq 2n + s_0 - s_n$$

So, the total cost is bounded by $2n + s_0 - s_n$, or $O(n + s_0)$

5. Show that if a DECREMENT operation is included in the k-bit counter example, n operations can cost as much as $\Theta(nk)$ time.

Consider the following sequence of $n = 2^k$ operations: A series of 2^{k-1} INCREMENT operations, followed by an alternating sequence of 2^{k-1} DECREMENT and INCREMENT operations.

The initial sequence of INCREMENT operations bring the counter to the value 2^{k-1}. This takes $O(n)$ time, since each INCREMENT is amortized constant time.

Then, we are followed by pairs of DECREMENT and INCREMENT operations. Both of them run in $\Theta(k)$ (actual cost), since both these operations need to change all k bits of the number. Since we have $\frac{n}{2}$ such operations, the total cost is $\Theta(nk)$.

Therefore, the total cost of all n operations takes $O(n) + \Theta(nk) = \Theta(nk)$ time.

6. If the set of stack operations includes a MULTIPUSH operation, which pushes k items onto the stack, does the $O(1)$ bound on the amortized cost of stack operations continue to hold?

No. Similar to the previous problem, we can have an alternating sequence n of MULTIPUSH(S, k) and MULTIPOP(S, k) operations, whose actual running time will be $\Theta(nk)$. Thus, the $O(1)$ bound on the amortized costs don't hold.

7. You perform a sequence of PUSH and POP operations on a stack whose size never exceeds k. After every k operations, a copy of the entire stack is made automatically, for backup purposes. Show that the cost of n stack operations, including copying the stack, is $O(n)$ by assigning suitable amortized costs to the various stack operations.

Assign the following charges to the operations:

- PUSH: 2
- POP: 2
- Backup: 0

Then, we know that each push and pop operation costs 1 unit, and the other unit of cost accumulates as credit. Each backup operation costs k units, but that can be paid for by the k credits from the last k operations. Thus, the sum of amortized costs upper bounds the total running time of n operations, hence, the total cost is $O(n)$.

8. You wish not only to increment a counter but also to reset it to 0 (i.e., make all bits in it 0). Counting the time to examine or modify a bit as $\Theta(1)$, show how to implement a counter as an array of bits so that any sequence of n INCREMENT and RESET operations takes $O(1)$ time on an initially zero counter.

We know that charging 2 units per operation is enough to pay for n increment operations. Now, we also need to account for RESET operations, which can be $O(k)$ in the worst case.

Firstly, for it to not always take $O(k)$, we should maintain an index `max_bit` to know the highest order bit that needs to be reset; all higher order bits are zero and thus don't need to be modified.

After the ith operation, let b_i be the number of 1s in the counter, and k_i be the index of the highest 1 bit.

Define the potential function as:

$$\Phi(D_i) = b_i + k_i$$

Justification:

- An expensive increment operation should be compensated by the drop in potential. The cost of the increment operation is $b_{i-1} - b_i + 2$ which is covered by the drop in first term of the potential $(b_{i-1} \to b_i)$.
- A reset operation should be compensated by the change in potential. The cost of the reset operation is k_{i-1}. The drop in the second term of the potential covers this cost $(k_{i-1} \to 0)$.

Since the potential is always nonnegative, the sum of amortized costs gives a bound on the total running time of n operations.

Amortized costs:

- INCREMENT

$$\begin{aligned}\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= b_{i-1} - b_i + 2 + b_i - b_{i-1} + k_i - k_{i-1} \\ &= 2 + k_i - k_{i-1} \\ &\leq 3\end{aligned}$$

- RESET

$$\begin{aligned}\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= k_{i-1} + b_i - b_{i-1} + k_i - k_{i-1} \\ &= -b_{i-1} \\ &\leq 0\end{aligned}$$

Thus, both INCREMENT and RESET work in amortized $O(1)$ cost.

9. **[Queue using two stacks]** A queue is implemented using two stacks:
   - Stack S_1 stores newly enqueued elements.
   - Stack S_2 is used for deletions.
   - If S_2 is empty during dequeue, all elements of S_1 are moved to S_2.

   1. What is the worst case cost of one dequeue? Answer in Θ notation and justify.
   2. Prove that both enqueue and dequeue have an amortized cost of $O(1)$.

Let us consider elementary push and pop operations on each stack to cost 1 unit.

Suppose the queue contains n elements. For a DEQUEUE operation, the worst case is when S_2 is empty and all n elements are stored in S_1. Then, first we need to move all elements from S_1 to S_2 and then pop out the top element in S_2 to delete the earliest inserted element. This runs in $\Theta(n)$ time (worst case time for DEQUEUE).

For amortized analysis, let us define a potential function on the state of the two stacks. Here, the expensive operation is the worst case dequeue operation, i.e. a dequeue that takes place when S_2 is empty. In this case, we need to move all elements from S_1 and then pop one element from S_2, so the cost of this operation is $2n + 1$.

Cost of enqueue operation, and a non expensive dequeue operation is 1. So, to show that all operations are amortized $O(1)$, we need to choose our potential function, such that the drop in potential due to an expensive pays for the operation's cost.

Let us choose the potential function:

$$\Phi(Q) = 2|S_1|$$

i.e. twice the number of elements in the stack S_1.

Clearly, the potential function is always nonnegative, and the starting potential is 0 (both stacks are empty). Thus, the sum of amortized costs will bound the total running time of n operations.

Let us calculate the amortized costs.

- ENQUEUE:

$$\begin{aligned}\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + 2 = 3\end{aligned}$$

- Inexpensive DEQUEUE:

$$\begin{aligned}\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= c_i \\ &= 1\end{aligned}$$

- Expensive DEQUEUE:

$$\begin{aligned}\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= 2n + 1 + 0 - 2n \\ &= 1\end{aligned}$$

So, we assign the following amortized costs:
- ENQUEUE: 3
- DEQUEUE: 1

Therefore, both the operations run in amortized constant time.

We can also give an intuitive justification for these costs by the accounting method. With every enqueue we push an element to stack S_1, we charge 3 units, out of which 1 unit is paid for the push, and 2 units are stored as credit for that element.

Whenever we perform a dequeue operation, we only charge 1 unit, which is used for the final pop operation on S_2. Before that, we may need to move every element from S_1 to S_2. This will be paid for by the 2 units of credits stored on each element in S_1. Since all operations can be performed with the total credits staying non negative, both operations have an amortized $O(1)$ cost.

10. **[Dynamic Table]** Consider the dynamic table data structure discussed in class. Suppose we double the table size when the table becomes $\frac{3}{4}$th full and half the table when it becomes $\frac{1}{4}$th full. Prove that the amortized time complexity of both TABLE-INSERT and TABLE-DELETE is $O(1)$.

In this case, expansion happens when $\alpha = \frac{3}{4}$. Immediately after the expansion, $\alpha = \frac{3}{8}$. Contraction happens when $\alpha = \frac{1}{4}$, and just after contraction, $\alpha = \frac{1}{2}$.

We can require the potential function to be num_i at the extreme points of the load factor, i.e. when $\text{num}_i = \text{size}_i \cdot \frac{3}{4}$ and $\text{num}_i = \text{size}_i \cdot \frac{1}{4}$, and 0 right after expansion/contraction ($\frac{\text{num}_i}{\text{size}_i} = \frac{3}{8}$ and $\frac{1}{2}$) so that the drop in potentials during the expensive operations pay for their actual costs.

Let us choose the following potential function:

$$\Phi(D_i) = \begin{cases} \frac{3}{4}\text{size}_i - 2\text{num}_i & \text{if } \alpha \leq \frac{3}{8} \\ 0 & \text{if } \frac{3}{8} < \alpha \leq \frac{1}{2} \\ 3\text{num}_i - \frac{3}{2}\text{size}_i & \text{if } \alpha > \frac{1}{2} \end{cases}$$

We can verify:

- at $\alpha = \frac{3}{4}$, $\Phi = \frac{3}{4}\text{size}_i = \text{num}_i$

- at $\alpha = \frac{1}{4}$, $\Phi = \frac{1}{4}\text{size}_i = \text{num}_i$
- at $\alpha = \frac{3}{8}$, $\Phi = 0$
- at $\alpha = \frac{1}{2}$, $\Phi = 0$

Since the potential is always non negative, sum of amortized costs of n operations will upper bound their total actual costs. To compute the amortized costs:

TABLE-INSERT:

- If $\alpha_{i-1} > 1/2$,
  - If an expansion took place,

$$\begin{aligned}\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= \text{num}_i + 0 - \left(3 \cdot \text{num}_{i-1} - \frac{3}{2} \cdot \text{size}_{i-1}\right) \\ &= 4\end{aligned}$$

  - If no expansion took place,

$$\begin{aligned}\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + \left(3 \cdot \text{num}_i - \frac{3}{2} \cdot \text{size}_i\right) - \left(3 \cdot (\text{num}_i - 1) - \frac{3}{2} \cdot \text{size}_i\right) \\ &= 4\end{aligned}$$

- Standard processing applies for configurations where $\alpha_{i-1} \leq 1/2$, e.g.

$$\hat{c_i} \leq 1 + \left(\frac{3}{4} \cdot \text{size}_i - 2 \cdot \text{num}_i\right) - \left(\frac{3}{4} \cdot \text{size}_i - 2 \cdot (\text{num}_i - 1)\right) = -1$$

TABLE-DELETE:

- If $\alpha_{i-1} \leq 3/8$,
  - If contraction takes place,

$$\begin{aligned}\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= \text{num}_i + 0 - \left(\frac{3}{4} \cdot \text{size}_{i-1} - 2 \cdot \text{num}_{i-1}\right) \\ &= 2\end{aligned}$$

  - If no contraction takes place,

$$\begin{aligned}\hat{c_i} &= c_i + \Phi_i - \Phi_{i-1} \\ &= 1 + \left(\frac{3}{4} \cdot \text{size}_i - 2 \cdot \text{num}_i\right) - \left(\frac{3}{4} \cdot \text{size}_i - 2 \cdot (\text{num}_i + 1)\right) \\ &= 3\end{aligned}$$

- Standard processing applies for configurations where $\alpha_{i-1} > 3/8$, e.g.

$$\hat{c_i} \leq 1 + \left(3 \cdot \text{num}_i - \frac{3}{2} \cdot \text{size}_i\right) - \left(3 \cdot (\text{num}_i + 1) - \frac{3}{2} \cdot \text{size}_i\right) = -2$$

Therefore, both operations are of amortized $O(1)$ complexity.