17 Amortized Analysis

- We average the time required to perform a sequence of data-structure operations over all the operations performed.
- Thus, we can show that the average cost of an operation is small, even if a single operation within the sequence might be expensive.
- Amortized analysis differs from average-case analysis in that probability is not involved.
  - Amortized analysis guarantees the average performance of each operation in the worst case.

17.1 Aggregate analysis

- We show that for all $n$, a sequence of $n$ operations takes worst-case time $T(n)$ in total.
- In the worst case, average cost, or amortized cost, per operation is therefore $T(n)/n$.
- This amortized cost applies to each operation, even when there are several types of operations in the sequence.
- The other two methods we shall study, may assign different amortized costs to different types of operations.
**Stack operations**

- The fundamental stack operations $\text{PUSH}(S, x)$ and $\text{POP}(S)$ each takes $O(1)$ time.
- Let’s consider the cost of each operation to be 1.
- The total cost of a sequence of $n$ PUSH and POP operations is therefore $n$, and the actual running time for $n$ operations is therefore $\Theta(n)$.
- Now we add the stack operation $\text{MULTIPOP}(S, k)$, which removes the $k$ top objects of stack $S$, popping the entire stack if the stack contains fewer than $k$ objects.

The operation $\text{STACK-EMPTY}$ returns TRUE if there are no objects currently on the stack, and FALSE otherwise.

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

1. while not $\text{STACK-EMPTY}(S)$ and $k > 0$
2. $\text{POP}(S)$
3. $k \leftarrow k - 1$

- The total cost of $\text{MULTIPOP}$ is $\min(s, k)$, and the actual running time is a linear function of this.
Let us analyze a sequence of \( n \) PUSH, POP, and MULTIPOP operations on an initially empty stack.

- The worst-case cost of a MULTIPOP operation is \( O(n) \), since the stack size is at most \( n \).
- The worst-case time of any stack operation is \( O(n) \), and hence a sequence of \( n \) operations costs \( O(n^2) \).
- Although this analysis is correct, the \( O(n^2) \) result is not tight.
- Using aggregate analysis, we can obtain a better upper bound that considers the entire sequence of \( n \) operations.

We can pop each object from the stack at most once for each time we have pushed it onto the stack.

- The number of times that POP can be called on a nonempty stack, including calls within MULTIPOP, is at most the number of PUSH operations, which is at most \( n \).
- Any sequence of \( n \) PUSH, POP, and MULTIPOP operations takes a total of \( O(n) \) time.
- The average cost of an operation \( O(n)/n = O(1) \).
Incrementing a binary counter

- Consider the problem of implementing a $k$-bit binary counter that counts upward from 0.
- We use an array $A[0..k-1]$ of bits, where $A.length = k$, as the counter.
- A binary number $x$ that is stored in the counter has its lowest-order bit in $A[0]$ and its highest-order bit in $A[k-1]$, so that
  \[ x = \sum_{i=0}^{k-1} A[i] \cdot 2^i \]

Initially, $x = 0$: $A[i] = 0$ for $i = 0, 1, ..., k-1$

To add 1 (modulo $2^k$) to the value in the counter, we use the following procedure:

INCREMENT($A$)

1. $i \leftarrow 0$
2. while $i < A.length$ and $A[i] = 1$
3. $A[i] \leftarrow 0$
4. $i \leftarrow i + 1$
5. if $i < A.length$
6. $A[i] \leftarrow 1$
At the start of each iteration of the **while** loop (lines 2–4), we wish to add a 1 into position $i$.

- If $A[i] = 1$, then adding 1 flips the bit to 0 in position $i$ and yields a carry of 1, to be added into position $i + 1$ on the next iteration of the loop.
- Otherwise, the loop ends, and then, if $i < k$, we know that $A[i] = 0$, so that line 6 adds a 1 into position $i$, flipping the 0 to a 1.
- The cost of each **INCREMENT** operation is linear in the number of bits flipped.
A cursory analysis yields a bound that is correct but not tight.

Single execution of \textsc{Increment} takes time $\Theta(k)$ in the worst case, when array $A$ contains all 1s.

Thus, a sequence of $n$ \textsc{Increment} operations on an initially zero counter takes time $O(nk)$.

Tighten the analysis to yield a worst-case cost of $O(n)$ by observing that not all bits flip each time \textsc{Increment} is called.

$A[0]$ does flip each time \textsc{Increment} is called.

The next bit up, $A[1]$, flips only every other time — a sequence of $n$ \textsc{Increment} operations on zero counter causes $A[1]$ to flip $\lfloor n/2 \rfloor$ times.

Similarly, bit $A[2]$ flips only every fourth time, or $\lfloor n/4 \rfloor$ times in a sequence of $n$ \textsc{Increment} operations.

In general, bit $A[i]$ flips $\lfloor n/2^i \rfloor$ times in a sequence of $n$ \textsc{Increment} operations on an initially zero counter.

For $i \geq k$, bit $A[i]$ does not exist, and so it cannot flip.
The total number of flips in the sequence is thus
\[ \sum_{i=0}^{k-1} \left| \frac{n}{2^i} \right| < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n \]
by infinite decreasing geometric series \( \sum_{k=0}^{\infty} x^k = 1/(1-x) \), where \( |x| < 1 \).

Worst-case time for a sequence of \( n \) INCREMENT operations on an initially zero counter is therefore \( O(n) \).

The average cost of each operation, and therefore the amortized cost per operation, is \( O(n)/n = O(1) \).

17.2 The accounting method

Assign differing charges to different operations, some charged more/less than they actually cost.

When an operation’s amortized cost exceeds its actual cost, we assign the difference to specific objects in the data structure as credit.

Credit can help pay for later operations.

We can view the amortized cost of an operation as being split between its actual cost and credit that is either deposited or used up.

Amortized cost of operations may differ.
We want to show that in the worst case the average cost per operation is small by analyzing with amortized costs.

- 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.
- Moreover, this relationship must hold for all sequences of operations.

Let $c_i$ be the actual cost of the $i$th operation and $\hat{c}_i$ its amortized cost, require for all $n$-sequences

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

Total credit stored in the data structure is the difference between the total amortized cost and the total actual cost, $\sum_{i=1}^{n} \hat{c}_i - \sum_{i=1}^{n} c_i$.

Total credit associated with the data structure must be nonnegative at all times.

If we ever were to allow the total credit to become negative, then the total amortized cost would not be an upper bound on the total actual cost.

We must take care that the total credit in the data structure never becomes negative.
Stack operations

- Recall the actual costs of operations; PUSH: 1, POP: 1, and MULTIPOP $\min(k, s)$, where $k$ is the argument of MULTIPOP and $s$ is the stack size.
- Let us assign the following amortized costs: PUSH: 2, POP: 0, and MULTIPOP 0.
- The amortized cost of MULTIPOP is a constant, whereas the actual cost is variable.
- In general, the amortized costs of the operations under consideration may differ from each other, and they may even differ asymptotically.

We can pay for any sequence of stack operations by charging the amortized costs:

- We start with an empty stack.
- When we push on the stack, we use 1 unit of cost to pay the actual cost of the push and are left with a credit of 1.
- The unit stored serves as prepayment for the cost of popping it from the stack.
- When we execute a POP operation, we charge the operation nothing and pay its actual cost using the credit stored in the stack.
• We can also charge MULTIOPS nothing
• To pop the 1st element, we take 1 unit of cost from the credit and use it to pay the actual cost of a POP operation
• To pop a 2nd element, we again have an unit of cost in the credit to pay for the POP operation,…
• Thus, we have always charged enough up front to pay for MULTIPOP operations
• For any sequence of \( n \) operations, the total amortized cost is an upper bound on actual cost
• Since the total amortized cost is \( O(n) \), so is the total actual cost

Incrementing a binary counter

• The running time is proportional to the number of bits flipped, which we use as our cost
• Let us charge an amortized cost of 2 units to set a bit to 1
• We use 1 unit to pay for the setting of the bit, and we place the other unit on the bit as credit to be used later when we flip the bit back to 0
• At any point in time, every 1 in the counter has a unit of credit on it, and thus we can charge nothing to reset a bit to 0
The amortized cost of \textsc{Increment}:

- The cost of resetting the bits within the \textbf{while} loop is paid for by the units on the bits that are reset.
- The \textsc{Increment} procedure sets at most one bit (line 6) and therefore the amortized cost of an \textsc{Increment} operation is at most 2 units.
- The number of 1s in the counter never becomes negative, and thus the amount of credit stays nonnegative at all times.
- For \( n \) \textsc{Increment} operations, the total amortized cost is \( O(n) \), which bounds the total actual cost.

### 17.3 The potential method

- We represent the prepaid work as “potential energy,” or just “potential,” which can be released to pay for future operations.
- Associate the potential with the data structure as a whole rather than with specific objects.
- We will perform \( n \) operations, starting with an initial data structure \( D_0 \).
- Let \( c_i \) be the actual cost of the \( i \)th operation and \( D_i \) the data structure that results after applying the \( i \)th operation to data structure \( D_{i-1} \).
- **Potential function** $\Phi$ maps data structure $D_i$ to a real number $\Phi(D_i)$, the potential of $D_i$.
- The amortized cost $\hat{c}_i$ of the $i$th operation with respect to potential function $\Phi$ is defined by
  \[ \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \]
- The actual cost plus the change in potential.
- The total amortized cost of the $n$ operations is
  \[ \sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) \]
- 2nd equality follows because the terms telescope.

If we can define a potential function $\Phi$ 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 $\sum_{i=1}^{n} c_i$.
- In practice, we do not always know how many operations might be performed.
- Therefore, if we require that $\Phi(D_i) \geq \Phi(D_0)$ for all $i$, then we guarantee, as in the accounting method, that we pay in advance.
- We usually just define $\Phi(D_0)$ to be 0 and then show that $\Phi(D_i) \geq 0$ for all $i$. 
• Intuitively, if the potential difference $\Phi(D_i) - \Phi(D_{i-1})$ of the $i$th operation is positive, then the amortized cost $\tilde{c_i}$ represents an overcharge to the $i$th operation, and the potential of the data structure increases.

• If the potential difference is negative, then the amortized cost represents an undercharge to the $i$th operation, and the decrease in the potential pays for the actual cost of the operation.

• Different potential functions may yield different amortized costs yet still be upper bounds on the actual costs.

**Stack operations**

• We define the potential function $\Phi$ on a stack to be the number of objects in the stack.

• For the empty stack $D_0$, we have $\Phi(D_0) = 0$.

• Since the number of objects in the stack is never negative, the stack $D_i$ that results after the $i$th operation has nonnegative potential, and thus $\Phi(D_i) \geq 0 = \Phi(D_0)$.

• The total amortized cost of $n$ operations with respect to $\Phi$ therefore represents an upper bound on the actual cost.
If the $i$th operation on a stack containing $s$ objects is a PUSH operation, then the potential difference is

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

The amortized cost of this PUSH operation is

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

Suppose that the $i$th operation on the stack is MULTIPOP$(S, k)$, which causes $k' = \min(k, s)$ objects to be popped off the stack.

The actual cost of the operation is $k'$, and the potential difference is

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

Thus, the amortized cost of the MULTIPOP operation is

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

Similarly, the amortized cost of an ordinary POP operation is 0.

The amortized cost of each of the three operations is $O(1)$, and thus the total amortized cost of a sequence of $n$ operations is $O(n)$.

Since we have already argued that $\Phi(D_i) \geq \Phi(D_0)$, the total amortized cost of $n$ operations is an upper bound on the total actual cost.

The worst-case cost of $n$ operations is $O(n)$. 
Incrementing a binary counter

- We define the potential of the counter after the $i$th INCREMENT to be $b_i$, the number of 1s in the counter after the $i$th operation.
- Suppose that the $i$th INCREMENT operation resets $t_i$ bits.
- 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 to 1.
- If $b_i = 0$, then the $i$th operation resets all $k$ bits.

In this situation $b_{i-1} = t_i = k$.
- If $b_i > 0$, then $b_i = b_{i-1} - t_i + 1$.
- In either case, $b_i \leq b_{i-1} - t_i + 1$, and the 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
  $$\tilde{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \leq (t_i + 1) + (1 - t_i) = 2$$
- If the counter starts at zero, then $\Phi(D_0) = 0$.
- $\Phi(D_i) \geq 0$ for all $i$, the total amortized cost of $n$ INCREMENTS is an upper bound on the total actual cost, and so the worst-case cost is $O(n)$.
The potential method gives us a way to analyze the counter even when it does not start at zero.

The counter starts with $b_0$ 1s, and after $n$ increments it has $b_n$ 1s, where $0 \leq b_0, b_n \leq k$.

Rewrite total actual cost in terms of amortized cost as

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

We have $\widehat{c}_i \leq 2$ for all $1 \leq i \leq n$.

Since $\Phi(D_0) = b_0$ and $\Phi(D_n) = b_n$, the total actual cost of $n$ increments is $\leq 2n - b_n + b_0$.

Note that since $b_0 \leq k$, as long as $k = O(n)$, the total actual cost is $O(n)$.

I.e., if we execute at least $n = \Omega(k)$ increment operations, the total actual cost is $O(n)$, no matter what initial value the counter contains.
17.4 Dynamic tables

- Let us now study the problem of dynamically expanding and contracting a table.
- We show that the amortized cost of insertion/deletion is only $O(1)$.
- Though the actual cost of an operation is large when it triggers an expansion or a contraction.
- Moreover, we see how to guarantee that the unused space in a dynamic table never exceeds a constant fraction of the total space.

Let the dynamic table support the operations 
TABLE-INSERT and TABLE-DELETE.

- It is convenient to use the load factor $\alpha(T)$
  - $\alpha(T)$ of a nonempty table $T$ is the number of items stored in the table divided by the size (number of slots) of the table.
- We assign an empty table size 0, and we define its load factor to be 1.
- If $\alpha(T)$ is bounded below by a constant, the unused space in $T$ is never more than a constant fraction of the total amount of space.
17.4.1 Table expansion

- Storage for a table is allocated as array of slots
- A table fills up when all slots have been used or, equivalently, when its load factor is 1
- Upon inserting an item into a full table, we can expand the table by allocating a new table with more slots than the old table had
- We need the table to reside in contiguous memory, thus, we must allocate a new array and then copy items into the new table

- A common heuristic allocates a new table with twice as many slots as the old one
- If the only operations are insertions, then the load factor is always at least 1/2, and the amount of wasted space never exceeds half the space in the table
- The attribute $T.table$ contains a pointer to the block of storage representing the table
- $T.num$ contains the number of items in the table
- $T.size$ gives the total number of slots in the table
- Initially, the table is empty: $T.num = T.size = 0$
TABLE-INSERT \((T, x)\)

1. if \(T.\text{size} = 0\)
2. allocate \(T.\text{table}\) with 1 slot
3. \(T.\text{size} \leftarrow 1\)
4. if \(T.\text{num} = T.\text{size}\)
5. allocate new-table with \(2 \cdot T.\text{size}\) slots
6. insert all items in \(T.\text{table}\) into new-table
7. free \(T.\text{table}\)
8. \(T.\text{table} \leftarrow \text{new-table}\)
9. \(T.\text{size} \leftarrow 2 \cdot T.\text{size}\)
10. insert \(x\) into \(T.\text{table}\)
11. \(T.\text{num} \leftarrow T.\text{num} + 1\)

Let us analyze a sequence of \(n\) TABLE-INSERT operations on an initially empty table.

- If the current table has room for the new item, then the cost \(c_i\) of the \(i\)th operation is 1, since we only perform one elementary insertion.
- If the current table is full and an expansion occurs, then \(c_i = i\): cost of 1 for the elementary insertion plus \(i - 1\) for the items that we copy from the old table to the new table.
- The worst-case cost of an operation is \(O(n)\) ⇒ upper bound of \(O(n^2)\) on the total running time for \(n\) operations.
• This bound is not tight; we rarely expand the table in the course of \( n \) TABLE-INSERT operations.
• The \( i \)th operation causes an expansion only when \( i - 1 \) is an exact power of 2.
• The amortized cost of an operation is in fact \( O(1) \), as we can show using aggregate analysis.
• The cost of the \( i \)th operation is
\[
c_i = \begin{cases} 
  i & \text{if } i - 1 \text{ is an exact power of } 2 \\
  1 & \text{otherwise}
\end{cases}
\]

The total cost of \( n \) TABLE-INSERT operations is therefore
\[
\sum_{i=1}^{n} c_i \leq n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j < n + 2n = 3n,
\]
because at most \( n \) operations cost 1 and the costs of the remaining operations form a geometric series.
• Since the total cost of \( n \) TABLE-INSERT operations is bounded by \( 3n \), the amortized cost of a single operation is at most 3.
By using the accounting method, we can gain some feeling for why the amortized cost of a **TABLE-INSERT** operation should be 3.

Intuitively, each item pays for 3 elementary insertions:
- 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.

For example, suppose that the size of the table is $m$ immediately after an expansion.

Then it holds $m/2$ items, and contains no credit.

We charge 3 units of cost for each insertion
- The elementary insertion that occurs immediately costs 1 unit
- We place another unit as credit on the item inserted
- We place the third unit as credit on one of the $m/2$ items already in the table.

The table will not fill again until we have inserted another $m/2 - 1$ items, and thus, by the time the table contains $m$ items and is full, we will have placed a unit on each item to pay to reinsert it during the expansion.
Let us use the potential method to analyze a sequence of \( n \) TABLE-INSERT operations.

Potential function \( \Phi \) is 0 after an expansion but builds to table size by the time the table is full.

- \( \Phi(T) = 2 \cdot T.\ num - T. size \) is one possibility.
- Immediately after an expansion, we have \( T.\ num = T. size/2 \), and \( \Phi(T) = 0 \), as desired.
- Before expansion, we have \( T.\ num = T. size \), and \( \Phi(T) = T.\ num \) as desired.
- Table is always at least half full, \( T.\ num \geq T. size/2 \), which implies that \( \Phi(T) \) is always nonnegative.
- The sum of the amortized costs of \( n \) TABLE-INSERTS upper bounds the sum of the actual costs.

Let, after the \( i \)th operation,

- \( num_i \) be the number of items stored in the table,
- \( size_i \) be the total size of the table, and
- \( \Phi_i \) be the potential after the operation.

Initially, \( num_0 = 0, size_0 = 0 \), and \( \Phi_0 = 0 \).

- If the \( i \)th TABLE-INSERT operation does not trigger an expansion, then we have \( size_i = size_{i-1} \) and the amortized cost of the operation is

\[
\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} \\
= 1 + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1}) \\
= 1 + (2 \cdot num_i - size_i) - (2(num_i - 1) - size_i) \\
= 3
\]
If the \( i \)th operation **does** trigger an expansion,

\[
size_i = 2 \cdot size_{i-1}
\]

and

\[
size_{i-1} = num_{i-1} = num_i - 1,
\]

which implies that

\[
size_i = 2(num_i - 1)
\]

Thus, the amortized cost of the operation is

\[
\hat{c}_i = c_i + \Phi_i - \Phi_{i-1}
= num_i + (2 \cdot num_i - size_i) - (2 \cdot num_{i-1} - size_{i-1})
= num_i + (2 \cdot num_i - 2(num_i - 1)) - (2(num_i - 1) - (num_i - 1))
= num_i + 2 - (num_i - 1) = 3
\]
17.4.2 Table expansion and contraction

- To implement TABLE-DELETE, it is enough to remove the specified item from the table.
- To limit wasted space, we wish to contract the table when the load factor becomes too small.
- Table contraction is analogous to expansion.
- Ideally, we would like to preserve two properties:
  - the load factor of the dynamic table is bounded below by a positive constant, and
  - the amortized cost of a table operation is bounded above by a constant.

One might double the table size upon insertion into a full table and halve the size when a deletion would cause the table to become less than half full.

This would guarantee that the load factor is always above $1/2$, but can cause quite large amortized cost.

Consider that we perform $n$ operations on a table $T$, where $n$ is an exact power of 2.

- The first $n/2$ operations are insertions, which by our previous analysis cost a total of $\Theta(n)$.
- At the end of this sequence, $T.\text{num} = T.\text{size} = n/2$.
- For the second $n/2$ operations, we perform the following sequence: insert, delete, delete, insert, insert, delete, delete, insert, insert, …
• First the table expands to size \( n \)
• The two following deletions cause the table to contract back to size \( n/2 \)
• Further insertions cause another expansion, …
• The cost of each expansion and contraction is \( \Theta(n) \), and there are \( \Theta(n) \) of them
• Thus, the total cost of the \( n \) operations is \( \Theta(n^2) \), making the amortized cost of an operation \( \Theta(n) \)
• Downside of this strategy is that after expanding, we do not delete enough items to pay for contraction
• Likewise, for contracting the table

• We can allow the load factor to drop below \( 1/2 \)
• We still double the table size upon insertion into a full table, but halve the size when deletion causes the table to become less than \( 1/4 \) full
• The load factor of the table is therefore bounded below by the constant
• Intuitively, a load factor of \( 1/2 \) seems to be ideal, and the table’s potential would then be \( 0 \)
• As the load factor deviates from \( 1/2 \), the potential increases so that by the time we change the table, it has garnered sufficient potential to pay for copying all the items
• Thus, we will need a potential function that has grown to $T.\ num$ by the time that the load factor has either increased to 1 or decreased to $1/4$

• After either expanding or contracting the table, the load factor goes back to $1/2$ and the table’s potential reduces back to 0

• Code for TABLE-DELETE is analogous to TABLE-INSERT

• We assume that whenever the number of items in the table drops to 0, we free the storage for the table

• That is, if $T.\ num = 0$, then $T.\ size = 0$

Let us denote the load factor of a nonempty 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 $T.\ num = \alpha(T) \cdot T.\ size$, whether the table is empty or not

• We shall use as our potential function

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

• The potential of an empty table is 0 and it is never negative

• The total amortized cost of a sequence w.r.t. $\Phi$ provides an upper bound on the actual cost
• When the load factor is 1/2, the potential is 0
• When $\alpha(T) = 1$, we have $T.\ size = T.\ num$, which implies $\Phi(T) = T.\ num$, and the potential can pay for an expansion if an item is inserted
• When $\alpha(T) = 1/4$, we have $T.\ size = 4 \cdot T.\ num$, which implies $\Phi(T) = T.\ num$, and the potential can pay for a contraction if an item is deleted
• When the $i$th operation is TABLE-INSERT the analysis is identical to the earlier one for table expansion if $\alpha_{i-1} \geq 1/2$
  – Whether the table expands or not, the amortized cost of the operation $\hat{c}_i = 3$

• If $\alpha_{i-1} < 1/2$, the table cannot expand, since it expands only when $\alpha_{i-1} = 1$
• If $\alpha_i < 1/2$ as well, then the amortized cost of the $i$th operation is

\[
\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} \\
= 1 + (size_i/2 - num_i) - (size_{i-1}/2 - num_{i-1}) \\
= 1 + (size_i/2 - num_i) - (size_i/2 - (num_i - 1)) \\
= 0
\]
If $\alpha_{i-1} < 1/2$, but $\alpha_i \geq 1/2$, then

$$
\hat{c}_i = c_i + \Phi_i - \Phi_{i-1} \\
= 1 + (2 \cdot \text{num}_i - \text{size}_i) - (\text{size}_{i-1}/2 - \text{num}_{i-1}) \\
= 1 + (2(\text{num}_{i-1} + 1) - \text{size}_{i-1}) - (\text{size}_{i-1}/2 - \text{num}_{i-1}) \\
= 3 \cdot \text{num}_{i-1} - \frac{3}{2} \text{size}_{i-1} + 3 \\
= 3 \alpha_{i-1} \text{size}_{i-1} - \frac{3}{2} \text{size}_{i-1} + 3 \\
< \frac{3}{2} \text{size}_{i-1} - \frac{3}{2} \text{size}_{i-1} + 3 \\
= 3
$$

Thus, the amortized cost of a TABLE-INSERT operation is at most 3

When the $i$th operation is a TABLE-DELETE, the amortized cost is also bounded above by a constant

In summary, since the amortized cost of each operation is bounded above by a constant, the actual time for any sequence of $n$ operations on a dynamic table is $O(n)$