Introduction to Algorithms

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 400

Amortized Analysis

- In an **amortized analysis**, we average the time required to perform a sequence of data-structure operations over all the operations performed.
- With amortized analysis, we can show that the average cost of an operation is small, if we average over a sequence of operations, even though a single operation within the sequence might be expensive.
- The difference between amortized analysis and average-case analysis is that, in the former, the average performance of each operation is guaranteed in the worst-case, whereas the latter involves a probabilistic analysis.

Subsection 1

The Aggregate Method

Aggregate Analysis

- In **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, the average cost, or **amortized cost**, per operation is therefore $\frac{T(n)}{n}$.
- This amortized cost applies to each operation, even when there are several types of operations in the sequence.
- In contrast, the accounting method and the potential method, studied later, may assign different amortized costs to different types of operations.

Stack Operations

- We analyze stacks that have been augmented with a new operation.
- The two fundamental stack operations, each of which takes $O(1)$ time are:

  $\textsc{Push}(S, x)$: pushes object x onto stack S;
  $\textsc{Pop}(S)$: pops the top of stack S and returns the popped object. Calling $\textsc{Pop}$ on an empty stack generates an error.
- Since each of these operations runs in $O(1)$ time, we consider the cost of each to be 1.
- The total cost of a sequence of n $\textsc{Push}$ and $\textsc{Pop}$ operations is therefore n. The actual running time for n operations is $\Theta(n)$.

The Multipop Operation

- We add the stack operation $\textsc{MultiPop}(S, k)$, which removes the k top objects of stack S, popping the entire stack if the stack contains fewer than k objects. If $k \leq 0$ nothing changes.
- $\textsc{StackEmpty}$ returns TRUE if there are no objects currently on the stack, and FALSE otherwise.

$\textsc{MultiPop}(S, k)$

1. while not $\textsc{StackEmpty}(S)$ and $k > 0$
2. $\quad \textsc{Pop}(S)$
3. $\quad k = k - 1$

- The actual running time of $\textsc{MultiPop}(S, k)$ on a stack of s objects is linear in the number of $\textsc{Pop}$ operations actually executed. Assume costs of 1 each for $\textsc{Push}$ and $\textsc{Pop}$. The number of iterations of the while loop is the number $\min(s, k)$ of objects popped. Each iteration makes one call to $\textsc{Pop}$. Thus, the total cost of $\textsc{MultiPop}$ is $\min(s, k)$ and the actual running time is linear in this cost.

Rough Analysis of a Sequence of Operations

- We analyze a sequence of n PUSH, POP and MULTIPOP operations on an initially empty stack.
- The worst-case cost of a MULTIPOP operation in the sequence is $O(n)$, since the stack size is at most n.
- The worst-case time of any stack operation is therefore $O(n)$.
- Hence, a sequence of n operations costs $O(n^2)$, since we may have $O(n)$ MULTIPOP operations costing $O(n)$ each.
- This $O(n^2)$ result, obtained by considering the worst-case cost of each operation individually, is not tight.
- Using aggregate analysis, we can obtain a better upper bound that considers the entire sequence of n operations.

Careful Aggregate Analysis of a Sequence of Operations

- A single MULTIPOP operation can be expensive, but any sequence of n PUSH, POP and MULTIPOP operations on an initially empty stack can cost at most $O(n)$.

  We can pop each object from the stack at most once for each time we have pushed it onto the stack.
  Therefore, 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.
  For any value of n, any sequence of n PUSH, POP and MULTIPOP operations takes a total of $O(n)$ time.
- The average cost of an operation is $\frac{O(n)}{n} = O(1)$.
- In aggregate analysis, we assign the amortized cost of each operation to be the average cost.
- So all three stack operations have an amortized cost of $O(1)$.

Incrementing a Binary Counter

- We implement a k-bit binary counter that counts upward from 0.
- We use an array $A[0 \ldots k-1]$ of bits, where $A.\text{length} = k$, as the counter.
- A binary number x stored in the counter has its lowest-order bit in $A[0]$ and its highest-order bit in $A[k-1]$: $x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$.
- Initially, $x = 0$, and, thus, $A[i] = 0$, for $i = 0, 1, \ldots, k-1$.
- To add 1 (modulo 2^k) to the value, we use the following:

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
```

Example

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

The binary counter on the left is incremented 16 times, starting with the initial value 0 and ending with the value 16.
At the start of each iteration of the while loop in 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.

Aggregate Analysis of INCREMENT

- A cursory analysis yields a bound that is correct but not tight.

  A single execution of INCREMENT takes time $\Theta(k)$ in the worst case, in which array A contains all 1s.

  Thus, a sequence of n INCREMENT operations on an initially zero counter takes time $O(nk)$ in the worst case.
- We can get worst-case cost of $O(n)$ for a sequence of n operations by observing that not all bits flip each time INCREMENT is called.
  - $A[0]$ does flip each time INCREMENT is called.
  - $A[1]$ flips only every other time: a sequence of n operations on an initially zero counter causes $A[1]$ to flip $\lfloor \frac{n}{2} \rfloor$ times.
  - Similarly, $A[2]$ flips only every fourth time, or $\lfloor \frac{n}{4} \rfloor$ times in a sequence of n operations.
  - In general, for $i = 0, 1, \ldots, k-1$, bit $A[i]$ flips $\lfloor \frac{n}{2^i} \rfloor$ times in a sequence of n INCREMENT operations on an initially zero counter.
  - For $i \geq k$, bit $A[i]$ does not exist, and so it cannot flip.

Amortizing the Costs

- The total number of flips in the sequence is thus

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

- The worst-case time for a sequence of n INCREMENT operations on an initially zero counter is therefore $\mathrm{O}(n)$.
- The average cost of each operation, and therefore the amortized cost per operation, is $\frac{\mathrm{O}(n)}{n} = \mathrm{O}(1)$.

Subsection 2

The Accounting Method

The Accounting Method

- In the **accounting method** of amortized analysis, we assign differing charges to different operations, with some operations charged more or less than they actually cost.
- We call the amount we charge an operation its **amortized 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 be used to pay for later operations whose amortized cost is less than their actual cost.
- Thus, 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.
- Since different operations may have different amortized costs, the method differs from aggregate analysis, in which all operations have the same amortized cost.

Assigning Amortized Costs

- We must choose the amortized costs of operations carefully.
- If 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.
- As in aggregate analysis, this relationship must hold for all sequences of operations.
- If we denote the actual cost of the i-th operation by c_i and the amortized cost of the i-th operation by $\widehat{c}_i$, we require $\sum_{i=1}^{n} \widehat{c}_i \geq \sum_{i=1}^{n} c_i$, for all sequences of n operations.
- The total credit stored in the data structure is $\sum_{i=1}^{n} \widehat{c}_i - \sum_{i=1}^{n} c_i$.
- The total credit must be nonnegative at all times, since, if it was allowed to become negative, the total amortized costs incurred at that time would be below the total actual costs incurred.

Stack Operations and Amortized Costs

- Recall that the actual costs of the operations are 1 for PUSH, 1 for POP, and $\min(k, s)$ for MULTIPOP, where k is the argument supplied to MULTIPOP and s is the stack size when it is called.
- We assign amortized costs: 2 for PUSH, 0 for POP and 0 for MULTIPOP. (Note the amortized cost of MULTIPOP is constant (0), whereas the actual cost is variable.)
- We show that we can pay for any sequence of stack operations by charging the amortized costs.
  - We start with an empty stack.
  - When we push an item on the stack, we use 1 unit to pay the actual cost of the push and are left with a credit of 1 attached to the item.
  - At any point in time, every item on the stack has 1 unit of credit on it, which 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.

By charging the PUSH a little more, we can charge the POP nothing.

The MultiPop Cost and Amortization

- We can also charge MultiPop operations nothing.
  - To pop the first item, we take the unit of credit off the item and use it to pay the actual cost of a Pop operation.
  - To pop a second item, we again have a unit of credit on the item to pay for the Pop operation, and so on.

  Thus, we have always charged enough up front to pay for MultiPop operations.
- It follows that, for any sequence of n Push, Pop and MultiPop operations, the total amortized cost is an upper bound on the total actual cost.
- Since the total amortized cost is $O(n)$, so is the total actual cost.

Incrementing a Binary Counter

- The running time of INCREMENT on a binary counter that starts at zero is proportional to the number of bits flipped.
- We use as the unit cost the flipping of a bit.
- For the amortized analysis:
  - We charge an amortized cost of 2 units to set a bit to 1.
  - When a bit is set, we use 1 unit to pay for the actual setting of the bit, and we place the other on credit to be used when the bit is reset 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.
- To determine the amortized cost of INCREMENT, note that the cost of resetting the bits within the while loop is paid for by the credit on the bits that are reset. The procedure sets at most one bit, whence the amortized cost is at most 2 units.

  Since the amount of credit stays nonnegative at all times, for n INCREMENT operations, the total amortized cost is $O(n)$, which bounds the total actual cost.

Subsection 3

The Potential Method

Potential Functions

- Instead of representing prepaid work as credit stored with specific objects, the **potential method** of amortized analysis represents the prepaid work as "potential energy", or just "potential", which can be released to pay for future operations.
- The potential is associated with the data structure as a whole rather than with specific objects.
  - 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 i-th operation.
  - For each $i = 1, 2, \ldots, n$, we let D_i be the data structure that results after applying the i-th operation to data structure D_{i-1}.
- A **potential function** Φ maps each data structure D_i to a real number $\Phi(D_i)$, which is the **potential** associated with data structure D_i.

Potential Functions and Amortized Costs

- The **amortized cost** $\widehat{c}_i$ of the i-th operation with respect to potential function Φ is defined by

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

  I.e., the amortized cost of each operation is its actual cost plus the change in potential due to the operation.

- The total amortized cost of the n operations is

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

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

The Potential in Practice

- Since, in practice, we do not always know how many operations might be performed, we require that $\Phi(D_i) \geq \Phi(D_0)$, for all i, so as to guarantee that we pay in advance.
- Usually we define $\Phi(D_0) = 0$ and show that $\Phi(D_i) \geq 0$, for all i.
- Intuitively, the idea is that:
  - If the potential difference $\Phi(D_i) \geq \Phi(D_{i-1})$ of the i-th operation is positive, then the amortized cost $\widehat{c}_i$ 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.
- Note that the amortized costs depend on the choice of the potential function Φ, i.e., different potential functions may yield different amortized costs that may all be upper bounds on the actual costs.

Potential for Stack

- We revisit the example of the stack operations PUSH, POP and MULTIPOP.
- We define the potential function Φ 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, $\Phi(D_i) \geq 0 = \Phi(D_0)$.
- This implies that the total amortized cost of n operations with respect to Φ represents an upper bound on the actual cost.
- We now compute the amortized costs of the various stack operations.

Amortized Costs for Stack Operations

- If the i-th operation on a stack containing s objects is a PUSH, then the potential difference is

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

  Thus, the amortized cost of this PUSH operation is

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

- If the i-th operation is MULTIPOP(S, k), it causes $k' = \min(k, s)$ objects to be popped. The actual cost of the operation is k'. The potential difference is

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

  Thus, the amortized cost of the MULTIPOP is

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

- Similarly, the amortized cost of a POP operation is 0.

Total Amortized Costs for Stack Operations

- The amortized cost of each of the three operations is $O(1)$.
- So the total amortized cost of a sequence of n operations is $O(n)$.
- Since 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 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; So $b_{i-1} = t_i = k$.
  - 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

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

- Clearly, the total amortized cost of a sequence of n operations is an upper bound on the total actual cost. Thus, the worst-case cost of n INCREMENT operations is $O(n)$.

Counter Not Starting at Zero

- Suppose the counter starts with b_0 1s, and after n INCREMENT operations it has b_n 1s, where $0 \leq b_0, b_n \leq k$.
- For the total amortized cost, we have

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

- 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 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 $k = \mathrm{O}(n)$, the total actual cost is $\mathrm{O}(n)$. I.e., if we execute at least $n = \Omega(k)$ INCREMENT operations, the total actual cost is $\mathrm{O}(n)$, no matter what initial value the counter contains.

Subsection 4

Dynamic Tables

Dynamic Table Allocation

- If we do not know in advance how many objects an application will store in a table, we might want to be able to dynamically expand and contract an initially allocated table.
- The actual cost of an insertion or a deletion operation may be large when it triggers an expansion or a contraction.
- Using amortized analysis, we show that the amortized cost of insertion and deletion is only $O(1)$.
- We also see how to guarantee that the unused space in a dynamic table never exceeds a constant fraction of the total space.

Dynamic Table Operations and the Load Factor

- We assume that the dynamic table supports the operations TableInsert and TableDelete:
  - TableInsert inserts into the table an item that occupies a single slot.
  - TableDelete removes an item from the table, thereby freeing a slot.
- We define the **load factor** $\alpha(T)$ of a nonempty table T to be 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 the load factor of a dynamic table is bounded below by a constant, the unused space in the table is never more than a constant fraction of the total amount of space.

Table Expansion

- We start by analyzing a dynamic table in which we only insert items.
- We assume that storage for a table is allocated as an array of slots.
- A table fills up when all slots have been used or, equivalently, when its load factor is 1.
- We assume that, upon inserting an item into a full table, our software environment can expand the table by allocating a new table with more slots than the old table had.
- Because we always need the table to reside in contiguous memory, we must allocate a new array for the larger table and then copy items from the old table into the new table.
- A common heuristic allocates a new table with twice as many slots as the old one.
- If the only table operations are insertions, then the load factor of the table is always at least $\frac{1}{2}$, and thus the amount of wasted space never exceeds half the total space in the table.

The Procedure TableInsert

- Assume that T is an object representing the table.
  - T.table is 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, $T.\text{num} = T.\text{size} = 0$.

TableInsert(T, x)

```
1.  if T.size == 0
2.      allocate T.table with 1 slot
3.      T.size = 1
4.  if T.num == T.size
5.      allocate new-table with 2 · T.size slots
6.      insert all items in T.table into new-table
7.      free T.table
8.      T.table = new-table
9.      T.size = 2 · T.size
10. insert x into T.table
11. T.num = T.num + 1
```

Cost of Insertion and Expansion

- Notice that we have two "insertion" procedures:
  - The TableInsert procedure itself;
  - The elementary insertion into a table in Lines 6 and 10.
- We can analyze the running time of TableInsert in terms of the number of elementary insertions by assigning a cost of 1 to each elementary insertion.
- We assume that the actual running time of TableInsert is linear in the time to insert individual items, so that:
  - The overhead for allocating an initial table in Line 2 is constant;
  - The overhead for allocating and freeing storage in Lines 5 and 7 is dominated by the cost of transferring items in Line 6.
- We call the event in which Lines 5-9 are executed an **expansion**.

Rough Estimate for the Total Cost

- Let us analyze a sequence of n TableInsert operations on an initially empty table.
- For the cost c_i of the i-th operation:
  - If the current table has room for the new item, then $c_i = 1$, since we need only perform the one elementary insertion in Line 10.
  - If the current table is full, an expansion occurs, whence $c_i = i$:
    - The cost is 1 for the elementary insertion in Line 10
    - The cost is $i - 1$ for the items that we must copy from the old table to the new table in Line 6.
- If we perform n operations, the worst-case cost of an operation is $O(n)$. This leads to an upper bound of $O\left(n^2\right)$ on the total running time for n operations.

Amortizing the Total Cost

- The $O(n^2)$ bound is not tight, because we rarely expand the table in the course of n TableInsert 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 $O(1)$.

  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 TableInsert operations is therefore

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

- Since the total cost of n TableInsert operations is bounded by $3n$, the amortized cost of a single operation is at most 3.

Applying the Accounting Method

- By using the accounting method, we can gain some feeling for why the amortized cost of a TableInsert operation should be 3.
- Intuitively, each item pays for 3 elementary insertions:
  - Inserting itself into the current table;
  - Moving itself when the table expands;
  - Moving another item that has already been moved once when the table expands.

Example: Suppose that the size of the table is m immediately after an expansion. Then the table holds $\frac{m}{2}$ items, and it contains no credit. We charge 3 units 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 $\frac{m}{2}$ items in the table.

The table will not fill again until another $\frac{m}{2}$ items are inserted. Thus, by the time the table contains m items and is full, we will have placed a unit of credit on each item to pay to reinsert it during the expansion.

Applying the Potential Method

- We can use the potential method to analyze a sequence of n TableInsert operations.
- We define a potential function Φ that is 0 immediately after an expansion but builds to the table size by the time the table is full, so that we can pay for the next expansion by the potential.
- One possibility is $\Phi(T) = 2 \cdot T.\text{num} - T.\text{size}$.
  - Immediately after an expansion, we have $T.\text{num} = \frac{T.\text{size}}{2}$, and, thus, $\Phi(T) = 0$, as desired.
  - Immediately before an expansion, we have $T.\text{num} = T.\text{size}$, and, thus, $\Phi(T) = T.\text{num}$, as desired.
- The initial value of the potential is 0, and since the table is always at least half full, $T.\text{num} \geq \frac{T.\text{size}}{2}$, which implies that $\Phi(T)$ is always nonnegative.
- Thus, the sum of the amortized costs of n TableInsert operations gives an upper bound on the sum of the actual costs.

The Amortized Cost

- We analyze the amortized cost of the i-th TableInsert operation.
- We let
  - num_i denote the number of items stored in the table after the i-th operation;
  - size_i denote the total size of the table after the i-th operation;
  - Φ_i denote the potential after the i-th operation.
- Initially, $\text{num}_0 = 0$, $\text{size}_0 = 0$ and $\Phi_0 = 0$.
- If the i-th TableInsert operation does not trigger an expansion, then we have $\text{size}_i = \text{size}_{i-1}$.

  So the amortized cost is

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

The Amortized Cost (Cont'd)

- If the i-th operation does trigger an expansion, then we have

$$\begin{aligned} \text{size}_i &= 2 \cdot \text{size}_{i-1}; \\ \text{size}_{i-1} &= \text{num}_{i-1} = \text{num}_i - 1. \end{aligned}$$

These imply $\text{size}_i = 2(\text{num}_i - 1)$.

So the amortized cost is

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

Adding the TableDelete Operation

- To implement a TableDelete operation, it is simple enough to remove the specified item from the table.
- In addition, when the number of items in the table drops too low, we allocate a new, smaller table, and copy the items into it.
- We would like to preserve two properties:
  - The load factor of the dynamic table is bounded below by a constant;
  - The amortized cost of a table operation is bounded above by a constant.
- We assume that cost can be measured in terms of elementary insertions and deletions.
- We follow the strategy for expansion and contraction:
  - Double the table size when an item is inserted into a full table;
  - Halve the size when a deletion would cause the table to become less than half full.
- Then we guarantee that the load factor of the table never drops below $\frac{1}{2}$, but it can result in rather large amortized cost.

Shortcomings of the Strategy

- Consider the following scenario:
  - We perform n operations on a table T, where n is an exact power of 2.
    - The first $\frac{n}{2}$ operations are insertions, which by our previous analysis cost a total of $\Theta(n)$.
      At the end of this sequence of insertions, $T.\text{num} = T.\text{size} = \frac{n}{2}$.
    - For the second $\frac{n}{2}$ operations, we perform the following sequence: insert, delete, delete, insert, insert, delete, delete, insert, insert, ...

    The first insertion causes the table to expand to size n. The two following deletions cause the table to contract back to size $\frac{n}{2}$. Two further insertions cause another expansion, and so forth.
    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 $\Theta(n^2)$.

  This makes the amortized cost of an operation $\Theta(n)$.
- The downside of this strategy is obvious: After expanding the table, we do not delete enough items to pay for a contraction. Likewise, after contracting the table, we do not insert enough items to pay for an expansion.

Devising a New Strategy

- We can improve upon this strategy by allowing the load factor of the table to drop below $\frac{1}{2}$.
  - We double the table size upon inserting an item into a full table;
  - We halve the table size when deleting an item causes the table to become less than $\frac{1}{4}$ full.
- The load factor of the table is now bounded below by $\frac{1}{4}$.
- Intuitively, we consider a load factor of $\frac{1}{2}$ to be ideal, and the table's potential would then be 0.

  As the load factor deviates from $\frac{1}{2}$, the potential increases so that by the time we expand or contract the table, the table has sufficient potential to pay for copying all the items into the new table.
- We need a potential function that:
  - Grows to T.num by the time that the load factor has either increased to 1 or decreased to $\frac{1}{4}$;
  - After either expanding or contracting the table, the load factor goes back to $\frac{1}{2}$ and the table's potential reduces back to 0.

The Analysis: Adopting a Potential Function

- We assume that whenever the number of items drops to 0, we free the storage for the table, i.e., if $T.\text{num} = 0$, then $T.\text{size} = 0$.
- We define a potential function Φ that:
  - Is 0 immediately after an expansion or contraction;
  - Builds as the load factor increases to 1 or decreases to $\frac{1}{4}$.
- Denote the load factor of a nonempty table T by $\alpha(T) = \frac{T.\text{num}}{T.\text{size}}$.
- For an empty table, $T.\text{num} = T.\text{size} = 0$ and $\alpha(T) = 1$.
- So, it holds always $T.\text{num} = \alpha(T) \cdot T.\text{size}$.
- We define

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

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

The Analysis: Properties of the Potential Function

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

  The following hold:
  - When the load factor is $\frac{1}{2}$, the potential is 0.
  - When the load factor is 1, we have $T.\text{size} = T.\text{num}$.
    This gives $\Phi(T) = T.\text{num}$.
    Thus, the potential can pay for an expansion if an item is inserted.
  - When the load factor is $\frac{1}{4}$, we have $T.\text{size} = 4 \cdot T.\text{num}$.
    This gives $\Phi(T) = T.\text{num}$.
    Thus, the potential can pay for a contraction if an item is deleted.

Setting and Initializing the Variables

- To analyze a sequence of n TableInsert and TableDelete operations, we let:
  - c_i denote the actual cost of the i-th operation;
  - $\widehat{c}_i$ denote its amortized cost with respect to Φ;
  - num_i denote the number of items stored in the table after the i-th operation;
  - size_i denote the total size of the table after the i-th operation;
  - α_i denote the load factor of the table after the i-th operation;
  - Φ_i denote the potential after the i-th operation.
- Initially, $\text{num}_0 = 0$, $\text{size}_0 = 0$, $\alpha_0 = 1$, and $\Phi_0 = 0$.

i-th Operation Insertion

- We start with the case in which the i-th operation is TableInsert.
- If $\alpha_{i-1} \geq \frac{1}{2}$, the analysis is identical to that for table expansion. Whether the table expands or not, the amortized cost $\widehat{c}_i$ of the operation is at most 3.
- If $\alpha_{i-1} < \frac{1}{2}$, the table cannot expand as a result of the operation, since the table expands only when $\alpha_{i-1} = 1$.
  - If $\alpha_i < \frac{1}{2}$ as well, then the amortized cost of the i-th operation is

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

i-th Operation Insertion (Cont'd)

- We continue with case $\alpha_{i-1} < \frac{1}{2}$, in which the table cannot expand.
  - If $\alpha_{i-1} < \frac{1}{2}$ but $\alpha_i \geq \frac{1}{2}$, then

$$\begin{aligned}
\widehat{c}_i &= c_i + \Phi_i - \Phi_{i-1} \\
&= 1 + (2 \cdot \mathsf{num}_i - \mathsf{size}_i) - (\tfrac{\mathsf{size}_{i-1}}{2} - \mathsf{num}_{i-1}) \\
&= 1 + (2(\mathsf{num}_{i-1} + 1) - \mathsf{size}_{i-1}) - (\tfrac{\mathsf{size}_{i-1}}{2} - \mathsf{num}_{i-1}) \\
&= 3\mathsf{num}_{i-1} - \tfrac{3}{2}\mathsf{size}_{i-1} + 3 \\
&= 3\alpha_{i-1}\mathsf{size}_{i-1} - \tfrac{3}{2}\mathsf{size}_{i-1} + 3 \\
&< \tfrac{3}{2}\mathsf{size}_{i-1} - \tfrac{3}{2}\mathsf{size}_{i-1} + 3 \\
&= 3.
\end{aligned}$$

Thus, the amortized cost of a TableInsert operation is at most 3.

i-th Operation Deletion

- If the i-th operation is TABLEDELETE, $\text{num}_i = \text{num}_{i-1} - 1$.
- We handle the case $\alpha_{i-1} < \frac{1}{2}$.
  - If operation does not cause the table to contract, then $\text{size}_i = \text{size}_{i-1}$. The amortized cost of the operation is

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

i-th Operation Deletion (Cont'd)

- We continue with the case $\alpha_{i-1} < \frac{1}{2}$.
  - If $\alpha_{i-1} < \frac{1}{2}$ and the i-th operation does trigger a contraction, then the actual cost of the operation is $c_i = \text{num}_i + 1$, since we delete one item and move num_i items.
    We have
    $$\frac{\text{size}_i}{2} = \frac{\text{size}_{i-1}}{4} = \text{num}_{i-1} = \text{num}_i + 1.$$

    The amortized cost of the operation is

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

- When the i-th operation is a TableDelete and $\alpha_{i-1} \geq \frac{1}{2}$, the amortized cost is also bounded above by a constant.