

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. Amortized analysis differs from average-case analysis in that probability is not involved; an amortized analysis guarantees the *average performance of each operation in the worst case*.

The first three sections of this chapter cover the three most common techniques used in amortized analysis. Section 17.1 starts with aggregate analysis, in which we determine an upper bound $T(n)$ on the total cost of a sequence of n operations. The average cost per operation is then $T(n)/n$. We take the average cost as the amortized cost of each operation, so that all operations have the same amortized cost.

Section 17.2 covers the accounting method, in which we determine an amortized cost of each operation. When there is more than one type of operation, each type of operation may have a different amortized cost. The accounting method overcharges some operations early in the sequence, storing the overcharge as “prepaid credit” on specific objects in the data structure. Later in the sequence, the credit pays for operations that are charged less than they actually cost.

Section 17.3 discusses the potential method, which is like the accounting method in that we determine the amortized cost of each operation and may overcharge operations early on to compensate for undercharges later. The potential method maintains the credit as the “potential energy” of the data structure as a whole instead of associating the credit with individual objects within the data structure.

We shall use two examples to examine these three methods. One is a stack with the additional operation `MULTIPOP`, which pops several objects at once. The other is a binary counter that counts up from 0 by means of the single operation `INCREMENT`.

While reading this chapter, bear in mind that the charges assigned during an amortized analysis are for analysis purposes only. They need not—and should not—appear in the code. If, for example, we assign a credit to an object x when using the accounting method, we have no need to assign an appropriate amount to some attribute, such as $x.credit$, in the code.

When we perform an amortized analysis, we often gain insight into a particular data structure, and this insight can help us optimize the design. In Section 17.4, for example, we shall use the potential method to analyze a dynamically expanding and contracting table.

17.1 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 $T(n)/n$. Note that 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 in this chapter, the accounting method and the potential method, may assign different amortized costs to different types of operations.

Stack operations

In our first example of aggregate analysis, we analyze stacks that have been augmented with a new operation. Section 10.1 presented the two fundamental stack operations, each of which takes $O(1)$ time:

PUSH(S, x) pushes object x onto stack S .

POP(S) pops the top of stack S and returns the popped object. Calling POP on an empty stack generates an error.

Since each of these operations runs in $O(1)$ time, let us consider the cost of each 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 MULTIPOP(S, k), which removes the k top objects of stack S , popping the entire stack if the stack contains fewer than k objects. Of course, we assume that k is positive; otherwise the MULTIPOP operation leaves the stack unchanged. In the following pseudocode, the operation STACK-EMPTY returns TRUE if there are no objects currently on the stack, and FALSE otherwise.

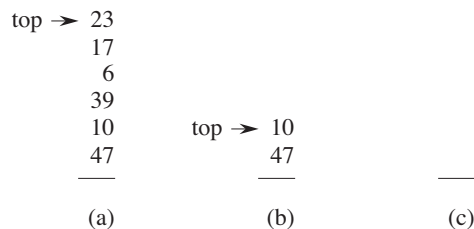


Figure 17.1 The action of MULTIPOP on a stack S , shown initially in (a). The top 4 objects are popped by MULTIPOP(S , 4), whose result is shown in (b). The next operation is MULTIPOP(S , 7), which empties the stack—shown in (c)—since there were fewer than 7 objects remaining.

MULTIPOP(S , k)

```

1  while not STACK-EMPTY( $S$ ) and  $k > 0$ 
2      POP( $S$ )
3       $k = k - 1$ 

```

Figure 17.1 shows an example of MULTIPOP.

What is the running time of MULTIPOP(S , k) on a stack of s objects? The actual running time is linear in the number of POP operations actually executed, and thus we can analyze MULTIPOP in terms of the abstract costs of 1 each for PUSH and POP. The number of iterations of the **while** loop is the number $\min(s, k)$ of objects popped off the stack. Each iteration of the loop makes one call to POP in line 2. Thus, the total cost of MULTIPOP is $\min(s, k)$, and the actual running time is a linear function of this cost.

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 in the sequence is $O(n)$, since the stack size is at most n . The worst-case time of any stack operation is therefore $O(n)$, and hence a sequence of n operations costs $O(n^2)$, since we may have $O(n)$ MULTIPOP operations costing $O(n)$ each. Although this analysis is correct, the $O(n^2)$ result, which we 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. In fact, although a single MULTIPOP operation can be expensive, any sequence of n PUSH, POP, and MULTIPOP operations on an initially empty stack can cost at most $O(n)$. Why? 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 $O(n)/n = O(1)$. In aggregate

analysis, we assign the amortized cost of each operation to be the average cost. In this example, therefore, all three stack operations have an amortized cost of $O(1)$.

We emphasize again that although we have just shown that the average cost, and hence the running time, of a stack operation is $O(1)$, we did not use probabilistic reasoning. We actually showed a *worst-case* bound of $O(n)$ on a sequence of n operations. Dividing this total cost by n yielded the average cost per operation, or the amortized cost.

Incrementing a binary counter

As another example of aggregate analysis, consider the problem of implementing a k -bit binary counter that counts upward from 0. We use an array $A[0 \dots k - 1]$ 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, $x = 0$, and thus $A[i] = 0$ for $i = 0, 1, \dots, k - 1$. To add 1 (modulo 2^k) to the value in the counter, we use the following 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       $A[i] = 1$ 

```

Figure 17.2 shows what happens to a binary counter as we increment it 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.

As with the stack example, 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 tighten our analysis to yield a worst-case cost of $O(n)$ for a sequence of n INCREMENT operations by observing that not all bits flip each time INCREMENT is called. As Figure 17.2 shows, $A[0]$ does flip each time INCREMENT is called. The next bit up, $A[1]$, flips only every other time: a sequence of n INCREMENT

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

Figure 17.2 An 8-bit binary counter as its value goes from 0 to 16 by a sequence of 16 INCREMENT operations. Bits that flip to achieve the next value are shaded. The running cost for flipping bits is shown at the right. Notice that the total cost is always less than twice the total number of INCREMENT operations.

operations on an initially 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 INCREMENT operations. In general, for $i = 0, 1, \dots, k-1$, bit $A[i]$ flips $\lfloor n/2^i \rfloor$ 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. The total number of flips in the sequence is thus

$$\begin{aligned} \sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor &< n \sum_{i=0}^{\infty} \frac{1}{2^i} \\ &= 2n, \end{aligned}$$

by equation (A.6). The 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

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 help 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. Different operations may have different amortized costs. This method differs from aggregate analysis, in which all operations have the same amortized cost.

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. Moreover, 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 \hat{c}_i , we require

$$\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i \tag{17.1}$$

for all sequences of n operations. The total credit stored in the data structure is the difference between the total amortized cost and the total actual cost, or

$\sum_{i=1}^n \hat{c}_i - \sum_{i=1}^n c_i$. By inequality (17.1), the 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 (the result of undercharging early operations with the promise of repaying the account later on), then the total amortized costs incurred at that time would be below the total actual costs incurred; for the sequence of operations up to that time, the total amortized cost would not be an upper bound on the total actual cost. Thus, we must take care that the total credit in the data structure never becomes negative.

Stack operations

To illustrate the accounting method of amortized analysis, let us return to the stack example. Recall that the actual costs of the operations were

PUSH 1 ,
 POP 1 ,
 MULTIPOP $\min(k, s)$,

where k is the argument supplied to MULTIPOP and s is the stack size when it is called. Let us assign the following amortized costs:

PUSH 2 ,
 POP 0 ,
 MULTIPOP 0 .

Note that the amortized cost of MULTIPOP is a constant (0), whereas the actual cost is variable. Here, all three amortized costs are constant. In general, the amortized costs of the operations under consideration may differ from each other, and they may even differ asymptotically.

We shall now show that we can pay for any sequence of stack operations by charging the amortized costs. Suppose we use a dollar bill to represent each unit of cost. We start with an empty stack. Recall the analogy of Section 10.1 between the stack data structure and a stack of plates in a cafeteria. When we push a plate on the stack, we use 1 dollar to pay the actual cost of the push and are left with a credit of 1 dollar (out of the 2 dollars charged), which we leave on top of the plate. At any point in time, every plate on the stack has a dollar of credit on it.

The dollar stored on the plate 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. To pop a plate, we take the dollar of credit off the plate and use it to pay the actual cost of the operation. Thus, by charging the PUSH operation a little bit more, we can charge the POP operation nothing.

Moreover, we can also charge MULTIPOP operations nothing. To pop the first plate, we take the dollar of credit off the plate and use it to pay the actual cost of a POP operation. To pop a second plate, we again have a dollar of credit on the plate to pay for the POP operation, and so on. Thus, we have always charged enough up front to pay for MULTIPOP operations. In other words, since each plate on the stack has 1 dollar of credit on it, and the stack always has a nonnegative number of plates, we have ensured that the amount of credit is always nonnegative. Thus, 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

As another illustration of the accounting method, we analyze the INCREMENT operation on a binary counter that starts at zero. As we observed earlier, the running time of this operation is proportional to the number of bits flipped, which we shall use as our cost for this example. Let us once again use a dollar bill to represent each unit of cost (the flipping of a bit in this example).

For the amortized analysis, let us charge an amortized cost of 2 dollars to set a bit to 1. When a bit is set, we use 1 dollar (out of the 2 dollars charged) to pay for the actual setting of the bit, and we place the other dollar 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 dollar of credit on it, and thus we can charge nothing to reset a bit to 0; we just pay for the reset with the dollar bill on the bit.

Now we can determine the amortized cost of INCREMENT. The cost of resetting the bits within the **while** loop is paid for by the dollars on the bits that are reset. The INCREMENT procedure sets at most one bit, in line 6, and therefore the amortized cost of an INCREMENT operation is at most 2 dollars. The number of 1s in the counter never becomes negative, and thus the amount of credit stays nonnegative at all times. Thus, for n INCREMENT operations, the total amortized cost is $O(n)$, which bounds the total actual cost.

17.3 The potential method

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

The potential method works as follows. We will perform n operations, starting with an initial data structure D_0 . For each $i = 1, 2, \dots, n$, we let c_i be the actual cost of the i th operation and 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 . The *amortized cost* \hat{c}_i of the i th operation with respect to potential function Φ is defined by

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) . \quad (17.2)$$

The amortized cost of each operation is therefore its actual cost plus the change in potential due to the operation. By equation (17.2), the total amortized cost of the n operations is

$$\begin{aligned} \sum_{i=1}^n \hat{c}_i &= \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0) . \end{aligned} \quad (17.3)$$

The second equality follows from equation (A.9) because the $\Phi(D_i)$ terms telescope.

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

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 . (See Exercise 17.3-1 for an easy way to handle cases in which $\Phi(D_0) \neq 0$.)

Intuitively, if the potential difference $\Phi(D_i) - \Phi(D_{i-1})$ of the i th operation is positive, then the amortized cost \hat{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.

The amortized costs defined by equations (17.2) and (17.3) depend on the choice of the potential function Φ . Different potential functions may yield different amortized costs yet still be upper bounds on the actual costs. We often find trade-offs that we can make in choosing a potential function; the best potential function to use depends on the desired time bounds.

Stack operations

To illustrate the potential method, we return once again to 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 with which we start, 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

$$\begin{aligned}\Phi(D_i) &\geq 0 \\ &= \Phi(D_0) .\end{aligned}$$

The total amortized cost of n operations with respect to Φ therefore represents an upper bound on the actual cost.

Let us now compute the amortized costs of the various stack operations. If the i th operation on a stack containing s objects is a PUSH operation, then the potential difference is

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

By equation (17.2), the amortized cost of this PUSH operation is

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

Suppose that the i th operation on the stack is $\text{MULTIPOP}(S, k)$, 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

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

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 therefore $O(n)$.

Incrementing a binary counter

As another example of the potential method, we again look at incrementing a binary counter. This time, we define the potential of the counter after the i th INCREMENT operation to be b_i , the number of 1s in the counter after the i th operation.

Let us compute the amortized cost of an INCREMENT 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, and 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

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

The amortized cost is therefore

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

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

The potential method gives us an easy way to analyze the counter even when it does not start at zero. The counter starts with b_0 1s, and after n INCREMENT

operations it has b_n 1s, where $0 \leq b_0, b_n \leq k$. (Recall that k is the number of bits in the counter.) We can rewrite equation (17.3) as

$$\sum_{i=1}^n c_i = \sum_{i=1}^n \hat{c}_i - \Phi(D_n) + \Phi(D_0) . \quad (17.4)$$

We have $\hat{c}_i \leq 2$ 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 INCREMENT operations is

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

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

We do not always know in advance how many objects some applications will store in a table. We might allocate space for a table, only to find out later that it is not enough. We must then reallocate the table with a larger size and copy all objects stored in the original table over into the new, larger table. Similarly, if many objects have been deleted from the table, it may be worthwhile to reallocate the table with a smaller size. In this section, we study this problem of dynamically expanding and contracting a table. Using amortized analysis, we shall show that the amortized cost of insertion and deletion is only $O(1)$, even though the actual cost of an operation is large when it triggers an expansion or a contraction. Moreover, we shall see how to guarantee that the unused space in a dynamic table never exceeds a constant fraction of the total space.

We assume that the dynamic table supports the operations TABLE-INSERT and TABLE-DELETE. TABLE-INSERT inserts into the table an item that occupies a single *slot*, that is, a space for one item. Likewise, TABLE-DELETE removes an item from the table, thereby freeing a slot. The details of the data-structuring method used to organize the table are unimportant; we might use a stack (Section 10.1), a heap (Chapter 6), or a hash table (Chapter 11). We might also use an array or collection of arrays to implement object storage, as we did in Section 10.3.

We shall find it convenient to use a concept introduced in our analysis of hashing (Chapter 11). 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 (one with no items) 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.

We start by analyzing a dynamic table in which we only insert items. We then consider the more general case in which we both insert and delete items.

17.4.1 Table expansion

Let us 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 1.¹ In some software environments, upon attempting to insert an item into a full table, the only alternative is to abort with an error. We shall assume, however, that our software environment, like many modern ones, provides a memory-management system that can allocate and free blocks of storage on request. Thus, 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. 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 $1/2$, and thus the amount of wasted space never exceeds half the total space in the table.

In the following pseudocode, we assume that T is an object representing 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, and $T.size$ 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  $T.size == 0$ 
2      allocate  $T.table$  with 1 slot
3       $T.size = 1$ 
4  if  $T.num == T.size$ 
5      allocate new-table with  $2 \cdot T.size$  slots
6      insert all items in  $T.table$  into new-table
7      free  $T.table$ 
8       $T.table = \textit{new-table}$ 
9       $T.size = 2 \cdot T.size$ 
10 insert  $x$  into  $T.table$ 
11  $T.num = T.num + 1$ 
```

¹In some situations, such as an open-address hash table, we may wish to consider a table to be full if its load factor equals some constant strictly less than 1. (See Exercise 17.4-1.)

Notice that we have two “insertion” procedures here: the TABLE-INSERT procedure itself and the *elementary insertion* into a table in lines 6 and 10. We can analyze the running time of TABLE-INSERT 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 TABLE-INSERT is linear in the time to insert individual items, so that the overhead for allocating an initial table in line 2 is constant and 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*.

Let us analyze a sequence of n TABLE-INSERT operations on an initially empty table. What is the cost c_i of the i th operation? If the current table has room for the new item (or if this is the first operation), then $c_i = 1$, since we need only perform the one elementary insertion in line 10. If the current table is full, however, and an expansion occurs, then $c_i = i$: the cost is 1 for the elementary insertion in line 10 plus $i - 1$ 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 $O(n)$, which leads to an upper bound of $O(n^2)$ on the total running time for n operations.

This bound is not tight, because we rarely expand the table in the course of n TABLE-INSERT operations. Specifically, 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

$$\begin{aligned} \sum_{i=1}^n c_i &\leq n + \sum_{j=0}^{\lfloor \lg n \rfloor} 2^j \\ &< n + 2n \\ &= 3n, \end{aligned}$$

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 the table holds $m/2$ items, and it contains

no credit. We charge 3 dollars for each insertion. The elementary insertion that occurs immediately costs 1 dollar. We place another dollar as credit on the item inserted. We place the third dollar 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 dollar on each item to pay to reinsert it during the expansion.

We can use the potential method to analyze a sequence of n TABLE-INSERT operations, and we shall use it in Section 17.4.2 to design a TABLE-DELETE operation that has an $O(1)$ amortized cost as well. We start by defining 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. The function

$$\Phi(T) = 2 \cdot T.num - T.size \quad (17.5)$$

is one possibility. Immediately after an expansion, we have $T.num = T.size/2$, and thus $\Phi(T) = 0$, as desired. Immediately before an expansion, we have $T.num = T.size$, and thus $\Phi(T) = T.num$, as desired. The initial value of the potential is 0, and since the table is always at least half full, $T.num \geq T.size/2$, which implies that $\Phi(T)$ is always nonnegative. Thus, the sum of the amortized costs of n TABLE-INSERT operations gives an upper bound on the sum of the actual costs.

To analyze the amortized cost of the i th TABLE-INSERT 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, and Φ_i denote the potential after the i th operation. Initially, we have $num_0 = 0$, $size_0 = 0$, 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

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

If the i th operation does trigger an expansion, then we have $size_i = 2 \cdot size_{i-1}$ and $size_{i-1} = num_{i-1} = num_i - 1$, which implies that $size_i = 2 \cdot (num_i - 1)$. Thus, the amortized cost of the operation is

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

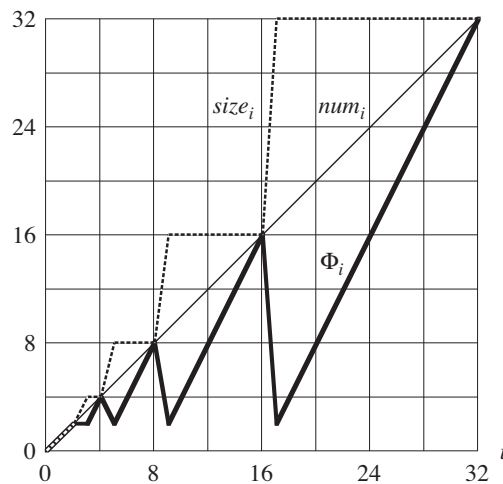


Figure 17.3 The effect of a sequence of n TABLE-INSERT operations on the number num_i of items in the table, the number $size_i$ of slots in the table, and the potential $\Phi_i = 2 \cdot num_i - size_i$, each being measured after the i th operation. The thin line shows num_i , the dashed line shows $size_i$, and the thick line shows Φ_i . Notice that immediately before an expansion, the potential has built up to the number of items in the table, and therefore it can pay for moving all the items to the new table. Afterwards, the potential drops to 0, but it is immediately increased by 2 upon inserting the item that caused the expansion.

Figure 17.3 plots the values of num_i , $size_i$, and Φ_i against i . Notice how the potential builds up to pay for expanding the table.

17.4.2 Table expansion and contraction

To implement a TABLE-DELETE operation, it is simple enough to remove the specified item from the table. In order to limit the amount of wasted space, however, we might wish to **contract** the table when the load factor becomes too small. Table contraction is analogous to table expansion: when the number of items in the table drops too low, we allocate a new, smaller table and then copy the items from the old table into the new one. We can then free the storage for the old table by returning it to the memory-management system. 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.

We assume that we measure the cost in terms of elementary insertions and deletions.

You might think that we should double the table size upon inserting an item into a full table and halve the size when deleting an item would cause the table to become less than half full. This strategy would guarantee that the load factor of the table never drops below $1/2$, but unfortunately, it can cause the amortized cost of an operation to be quite large. Consider the following scenario. 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 of insertions, $T.num = T.size = n/2$. For the second $n/2$ 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 $n/2$. 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 $\Theta(n^2)$, making 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.

We can improve upon this strategy by allowing the load factor of the table to drop below $1/2$. Specifically, we continue to double the table size upon inserting an item into a full table, but we halve the table size when deleting an item causes the table to become less than $1/4$ full, rather than $1/2$ full as before. The load factor of the table is therefore bounded below by the constant $1/4$.

Intuitively, we would consider a load factor of $1/2$ 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 expand or contract the table, the table has garnered sufficient potential to pay for copying all the items into the newly allocated table. 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.

We omit the code for TABLE-DELETE, since it is analogous to TABLE-INSERT. For our analysis, we shall 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 $T.size = 0$.

We can now use the potential method to analyze the cost of a sequence of n TABLE-INSERT and TABLE-DELETE operations. We start by defining a potential function Φ that is 0 immediately after an expansion or contraction and builds as the load factor increases to 1 or decreases to $1/4$. Let us denote the load fac-

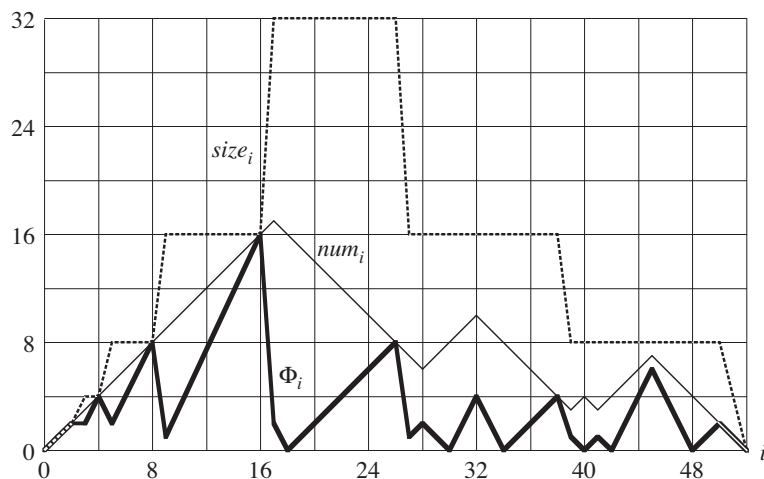


Figure 17.4 The effect of a sequence of n TABLE-INSERT and TABLE-DELETE operations on the number num_i of items in the table, the number $size_i$ of slots in the table, and the potential

$$\Phi_i = \begin{cases} 2 \cdot num_i - size_i & \text{if } \alpha_i \geq 1/2, \\ size_i/2 - num_i & \text{if } \alpha_i < 1/2, \end{cases}$$

each measured after the i th operation. The thin line shows num_i , the dashed line shows $size_i$, and the thick line shows Φ_i . Notice that immediately before an expansion, the potential has built up to the number of items in the table, and therefore it can pay for moving all the items to the new table. Likewise, immediately before a contraction, the potential has built up to the number of items in the table.

tor 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

$$\Phi(T) = \begin{cases} 2 \cdot T.num - T.size & \text{if } \alpha(T) \geq 1/2, \\ T.size/2 - T.num & \text{if } \alpha(T) < 1/2. \end{cases} \quad (17.6)$$

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

Before proceeding with a precise analysis, we pause to observe some properties of the potential function, as illustrated in Figure 17.4. Notice that when the load factor is $1/2$, the potential is 0. When the load factor is 1, we have $T.size = T.num$, which implies $\Phi(T) = T.num$, and thus the potential can pay for an expansion if an item is inserted. When the load factor is $1/4$, we have $T.size = 4 \cdot T.num$, which

implies $\Phi(T) = T.num$, and thus the potential can pay for a contraction if an item is deleted.

To analyze a sequence of n TABLE-INSERT and TABLE-DELETE operations, we let c_i denote the actual cost of the i th operation, \hat{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, and Φ_i denote the potential after the i th operation. Initially, $num_0 = 0$, $size_0 = 0$, $\alpha_0 = 1$, and $\Phi_0 = 0$.

We start with the case in which the i th operation is TABLE-INSERT. The analysis is identical to that for table expansion in Section 17.4.1 if $\alpha_{i-1} \geq 1/2$. Whether the table expands or not, the amortized cost \hat{c}_i of the operation is at most 3. If $\alpha_{i-1} < 1/2$, the table cannot expand as a result of the operation, since the table expands only when $\alpha_{i-1} = 1$. If $\alpha_i < 1/2$ as well, then the amortized cost of the i th operation is

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

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

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

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

We now turn to the case in which the i th operation is TABLE-DELETE. In this case, $num_i = num_{i-1} - 1$. If $\alpha_{i-1} < 1/2$, then we must consider whether the operation causes the table to contract. If it does not, then $size_i = size_{i-1}$ and the amortized cost of the operation is

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

If $\alpha_{i-1} < 1/2$ 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 $\text{size}_i/2 = \text{size}_{i-1}/4 = \text{num}_{i-1} = \text{num}_i + 1$, and the amortized cost of the operation is

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

When the i th operation is a TABLE-DELETE and $\alpha_{i-1} \geq 1/2$, the amortized cost is also bounded above by a constant. We leave the analysis as Exercise 17.4-2.

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 $O(n)$.