The remaining issue to consider is how large of a new array to create. A commonly used rule is for the new array to have twice the capacity of the existing array that has been filled. In Section 7.2.3, we will provide a mathematical analysis to justify such a choice.

To complete the revision to our original ArrayList implementation, we redesign the add method so that it calls the new resize utility when detecting that the current array is filled (rather than throwing an exception). The revised version appears in Code Fragment 7.5.

```
28   /** Inserts element e to be at index i, shifting all subsequent elements later. */
29   public void add(int i, E e) throws IndexOutOfBoundsException {
30     checkIndex(i, size + 1);
31     if (size == data.length)                    // not enough capacity
32       resize(2 * data.length);                  // so double the current capacity
...    // rest of method unchanged...
```

**Code Fragment 7.5:** A revision to the ArrayList.add method, originally from Code Fragment 7.3, which calls the resize method of Code Fragment 7.4 when more capacity is needed.

Finally, we note that our original implementation of the ArrayList class includes two constructors: a default constructor that uses an initial capacity of 16, and a parameterized constructor that allows the caller to specify a capacity value. With the use of dynamic arrays, that capacity is no longer a fixed limit. Still, greater efficiency is achieved when a user selects an initial capacity that matches the actual size of a data set, as this can avoid time spent on intermediate array reallocations and potential space that is wasted by having too large of an array.

### 7.2.3 Amortized Analysis of Dynamic Arrays

In this section, we will perform a detailed analysis of the running time of operations on dynamic arrays. As a shorthand notation, let us refer to the insertion of an element to be the last element in an array list as a ***push*** operation.

The strategy of replacing an array with a new, larger array might at first seem slow, because a single push operation may require $\Omega(n)$ time to perform, where $n$ is the current number of elements in the array. (Recall, from Section 4.3.1, that big-Omega notation, describes an asymptotic lower bound on the running time of an algorithm.) However, by doubling the capacity during an array replacement, our new array allows us to add $n$ further elements before the array must be replaced again. In this way, there are many simple push operations for each expensive one (see Figure 7.4). This fact allows us to show that a series of push operations on an initially empty dynamic array is efficient in terms of its total running time.

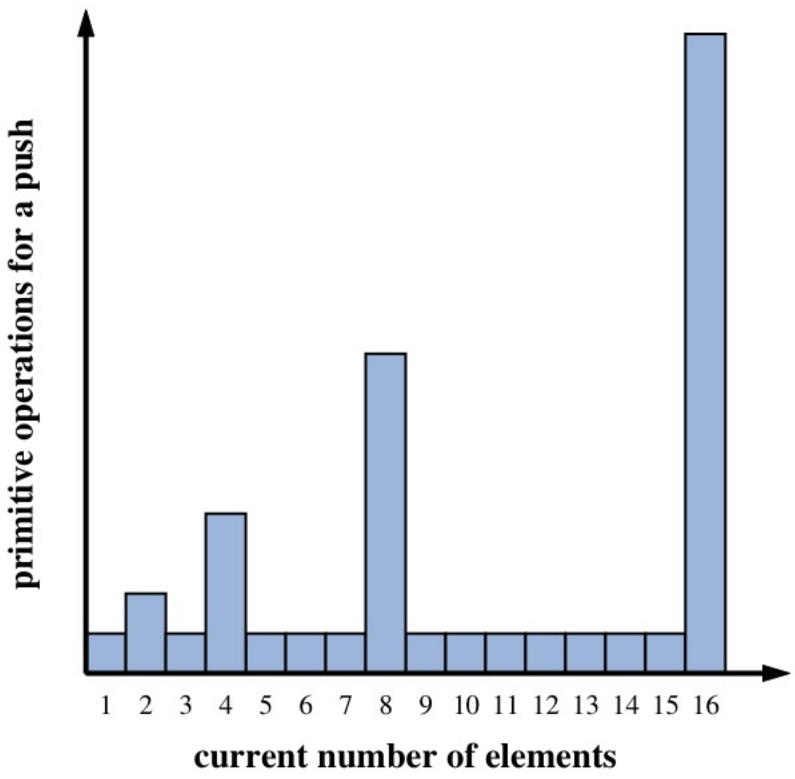


**Figure 7.4:** Running times of a series of push operations on a dynamic array.

Using an algorithmic design pattern called ***amortization***, we show that performing a sequence of push operations on a dynamic array is actually quite efficient. To perform an ***amortized analysis***, we use an accounting technique where we view the computer as a coin-operated appliance that requires the payment of one ***cyber-dollar*** for a constant amount of computing time. When an operation is executed, we should have enough cyber-dollars available in our current "bank account" to pay for that operation's running time. Thus, the total amount of cyber-dollars spent for any computation will be proportional to the total time spent on that computation. The beauty of using this analysis method is that we can overcharge some operations in order to save up cyber-dollars to pay for others.

**Proposition 7.2:** *Let $L$ be an initially empty array list with capacity one, implemented by means of a dynamic array that doubles in size when full. The total time to perform a series of $n$ push operations in $L$ is $O(n)$.*

**Justification:** Let us assume that one cyber-dollar is enough to pay for the execution of each push operation in $L$, excluding the time spent for growing the array. Also, let us assume that growing the array from size $k$ to size $2k$ requires $k$ cyber-dollars for the time spent initializing the new array. We shall charge each push operation three cyber-dollars. Thus, we overcharge each push operation that does not cause an overflow by two cyber-dollars. Think of the two cyber-dollars profited in an insertion that does not grow the array as being "stored" with the cell in which the element was inserted. An overflow occurs when the array $L$ has $2^i$ elements, for some integer $i \geq 0$, and the size of the array used by the array representing $L$ is $2^i$. Thus, doubling the size of the array will require $2^i$ cyber-dollars. Fortunately, these cyber-dollars can be found stored in cells $2^{i-1}$ through $2^i - 1$. (See Figure 7.5.)

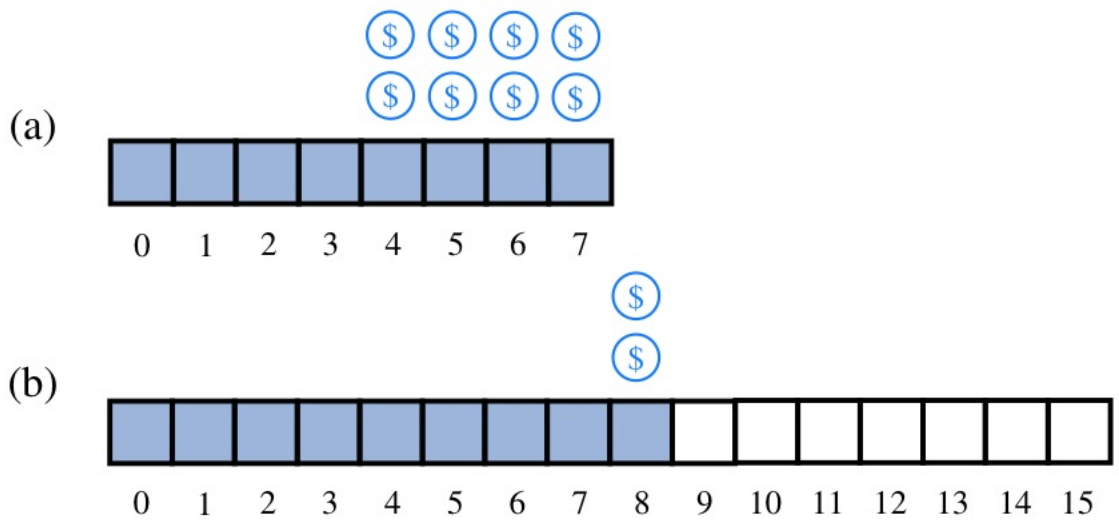


**Figure 7.5:** Illustration of a series of push operations on a dynamic array: (a) an 8-cell array is full, with two cyber-dollars "stored" at cells 4 through 7; (b) a push operation causes an overflow and a doubling of capacity. Copying the eight old elements to the new array is paid for by the cyber-dollars already stored in the table. Inserting the new element is paid for by one of the cyber-dollars charged to the current push operation, and the two cyber-dollars profited are stored at cell 8.

Note that the previous overflow occurred when the number of elements became larger than $2^{i-1}$ for the first time, and thus the cyber-dollars stored in cells $2^{i-1}$ through $2^i - 1$ have not yet been spent. Therefore, we have a valid amortization scheme in which each operation is charged three cyber-dollars and all the computing time is paid for. That is, we can pay for the execution of $n$ push operations using $3n$ cyber-dollars. In other words, the amortized running time of each push operation is $O(1)$; hence, the total running time of $n$ push operations is $O(n)$. ■

### Geometric Increase in Capacity

Although the proof of Proposition 7.2 relies on the array being doubled each time it is expanded, the $O(1)$ amortized bound per operation can be proven for any geometrically increasing progression of array sizes. (See Section 2.2.3 for discussion of geometric progressions.) When choosing the geometric base, there exists a trade-off between runtime efficiency and memory usage. If the last insertion causes a resize event, with a base of 2 (i.e., doubling the array), the array essentially ends up twice as large as it needs to be. If we instead increase the array by only 25% of its current size (i.e., a geometric base of 1.25), we do not risk wasting as much memory in the end, but there will be more intermediate resize events along the way. Still it is possible to prove an $O(1)$ amortized bound, using a constant factor greater than the 3 cyber-dollars per operation used in the proof of Proposition 7.2 (see Exercise R-7.7). The key to the performance is that the amount of additional space is proportional to the current size of the array.