The Stack ADT

An abstract data type (ADT) is an abstraction of a data structure. It specifies the type of data stored and different operations that can be performed on the data. It’s like a Java interface—it specifies the name and definition of the methods, but hides their implementations.

In the stack ADT, the data can be arbitrary objects and the main operations are push and pop which allow insertions and deletions in a last-in, first-out manner. One way to implement a stack is to use an array. Note that an array has a fixed size, so in a fixed capacity implementation of the stack, the number of items in the stack can be no more than the size of the array. This implies that to use the stack one must estimate the maximum size of the stack ahead of time. To make the stack have unlimited capacity, we will adjust dynamically the size of the array so that it is both sufficiently large to store all of the items and not so large so as to waste an excessive amount of space. We will consider two strategies—the incremental strategy in which the array capacity is increased by a constant amount when the stack size equals the array capacity and the doubling strategy, in which the size of the array is doubled when the stack size equals the array capacity. Since arrays cannot be dynamically resized, we have to create a new array of increased size and copy the elements from the old array to the new array.

We will first analyze the incremental strategy.

push(obj)

```
// s: stack size
// a: array capacity
// c: initial array size and also the increment in array size
A[s] = obj
s = s + 1
if s == a then
    a = a + c
    copy contents of old array to the new array
```

An example sequence of operations is below:
Let $c$ be the initial size of the array and the amount by which the array size is increased during each expansion. Consider a sequence of $n$ push operations. Note that after every $c$ push operations, the array expansion happens. The $n$ push operations cost $n$. The cost of the first expansion is $c + 2c$ (since $c$ elements are being copied and we need to allocate a new array of size $2c$), the cost of the second expansion is $2c + 3c$ (since $2c$ elements are being copied and we need to allocate a new array of size $3c$), and so on. We can separate these expressions into two parts: the cost of allocations and the cost of copying. The cost of allocations is $2c + 3c + ... + n + (n + c)$ and the cost of copying is $c + 2c + ... + n$.

Thus the total cost of $n$ consecutive push operations is given by

$$T(n) = \text{cost of pushes} + \text{cost of copying} + \text{cost of allocations}$$

$$= n + (c + 2c + ... + n) + (2c + 3c + ... + n + c)$$

$$= n + c(1 + 2 + ... + n/c) + c(2 + 3 + ... + n/c + 1)$$

$$= n + c \cdot \left( \frac{n/c(n/c + 1)}{2} \right) + c \left( \frac{(n/c + 1)(n/c + 1)}{2} - 1 \right)$$

$$= \Theta(n^2)$$

since $c$ is a constant.

We will now analyze the doubling method. The pseudocode is:

```plaintext
push(obj)
   // s: stack size
   // a: array capacity
   A[s] = obj
   s = s + 1
   if s == a then
     a = 2 * a
     copy contents of old array to the new array
```
We will be using amortized analysis, which means that we will be finding the time-averaged cost for a sequence of operations. In other words, the amortized runtime of an operation is the time required to perform a sequence of operations averaged over all the operations performed. Let $T(n)$ be the total time needed to perform a series of $n$ push operations. Then the amortized time of a single push operation is $T(n)/n$. Note that this is different from the notion of “average case analysis”—we’re not making any assumptions about inputs being chosen at random, nor are we assuming any probability distribution over the input. We are just averaging over time. Also note that the total real cost of a sequence of operations will be bounded by the total of the amortized costs of all the operations.

Let $s$ denote the number of objects in the stack at any given time and let $a$ be the array capacity at any given time. When $s < a$, push(obj) is a constant time operation. However, when the stack is full, i.e. when $s = a$, we double the size of the array. Thus the cost of push(obj) in this case is $O(s)$, as we have to copy $s$ items from the old array into the new array (the cost of allocating and freeing the array is $O(s)$). The worst case cost of a push operation is $O(n)$, and hence the cost of $n$ push operations is $O(n \log n)$ (since there are $O(\log n)$ expansions). Is this tight?

![Figure 2: A sequence of Push operations using the doubling strategy.](image)

The above analysis did NOT use amortization. Let’s see if we can get a better bound using amortized analysis. If we start from an empty stack, what is the cost of a sequence of $n$ push operations? As before, the cost of the $n$ push operations ignoring expansions is $n$. The cost of allocating new arrays is at most $1 + 2 + 4 + \ldots + n + 2n < 4n$ and the cost of copying elements is at most $1 + 2 + 4 + \ldots + n/2 + n < 2n$. Thus the total cost is at most $7n = O(n)$. The amortized cost of an operation is $7 = O(1)$, even though the worst case time complexity of a single push operation is $O(n)$.

Another way to analyze the doubling scheme is as follows: Each element will be allocated 7 dollars. Once the element is pushed, it uses 1 dollar. When the array needs to be doubled, only elements that have never
been moved before pay for all the elements that are being moved to the new array. Since the number of such elements is exactly half the number of all elements in the stack, each never Been moved element pays a cost of 6 for the move—2 for allocating space for itself and copying itself over, 2 for allocating space of one of the elements in the first half of the stack and copying it over, and 2 for allocating two more empty slots since the array is being doubled. Thus the total cost incurred by each element is 7, and hence $T(n) \leq 7n = O(n)$.

Similarly, in $\text{pop}()$, after removing the object from the stack, if the stack size is significantly less than the array capacity, then we resize the array (we don’t want to waste space). More specifically, when the stack size is equal to one-fourth of the array size, then we reduce the size of the array to half its current capacity. After resizing, the array is still half full and can accommodate a substantial number of push and pop operations before having to resize again. The pseudocode for the pop operations is as follows:

```plaintext
$.\text{pop}()$
item = A[s]
s = s - 1
if s < a/4 then
  a = a/2
```

Why do we only resize when the array is less than one fourth full, rather than one half full? Consider what would happen if a malicious user pushed elements onto the stack until it resized up. Then it popped a single element—this would trigger another resizing down. Then it pushed a single element—this would trigger another resizing up. And so on. In this worst case, every push/pop operation would require copying elements, leading to very bad running times. Resizing the array when it is less than one-fourth full prevents this “thrashing” problem.

### Queues

A queue is a collection of objects that is based on a first-in, first-out policy. The main operations in a Queue ADT are $\text{enqueue(obj)}$ and $\text{dequeue}$. The $\text{enqueue}$ operation inserts an element at the end of the queue. The $\text{dequeue}$ operation removes and returns the element at the front of the queue. Queues can also be implemented using expandable arrays. However, unlike a stack, in a queue we need to keep track of both the head and the tail of the queue. The head of the queue points to the first element in the queue and the tail points to the last element in the queue. Note that when we dequeue an element, the element is removed from the head of the queue and when an element is enqueued, that element is inserted at the tail of the queue. When the tail points to the end of the array, it may not mean that the array is full. This is because some elements may have been popped off and the head of the queue may not be pointing to the beginning of the array. That is, there may be room at the beginning of the array. To address this we use a wrap-around implementation. This way, we expand the array only when every slot in the array contains an element, i.e., when the queue size equals the array capacity. When copying the queue elements into the new array, we can “unwind” the queue so that the head points to the beginning of the array.

Note that this is only one possible implementation of a queue. There are countless others, including singly- or doubly-linked lists, circular linked lists, etc.