Data Structures and Algorithms
Lecture Outline
February 20, 2017

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` that 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 – *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
```

Let c be the intial size of the array and let c be 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 (since c elements are being copied), the cost of the second expansion is $2c$ (since $2c$ elements are being copied), and so on. Thus the total cost $T(n)$ is given by

$$T(n) = n + c + 2c + 3c + \ldots + n = n + c(1 + 2 + \ldots + n/c) = n + \frac{c(n/c)(n/c+1)}{2} = \Theta(n^2)$$

We will now analyze the doubling method (pseudo code given below).

```
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 use *amortized analysis* which means that we will be finding the time-averaged cost for a sequence of operations. In other words, it 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 push operation is $T(n)/n$. Note that this is different from our usual notion of "average case analysis" – we're not making any assumptions about inputs being chosen at random – we're just averaging over time. 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 move 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 if $O(n \log n)$ (since there are $O(\log n)$ expansions). Is this tight?

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 is n. The cost of expansion (doubling) is at most $1 + 2 + 4 + \ldots + n/2 + n < 2n$. Thus the total cost is at most $3n = O(n)$. Thus the amortized cost of an operation is 3, 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 object pays for the operation performed on it. When the array is doubled, only the 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 element pays a cost of 2 for the move. Thus the total cost incurred by each element is 3 and hence $T(n) = O(n)$.

Similarly, in `pop()` after removing the object from the stack, if the stack size is significantly less than the array capacity then we resize the array. 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 the array. The pseudocode for the pop operation is as follows.

```
pop()
item = A[s]
```

```
s = s-1
if (s < a/4) then
  a = a/2
```

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 `enqueue(obj)` and `dequeue`. The enqueue operation inserts an element at the end of the queue. The 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. 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, an 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, meaning that 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 "unwind" the queue so that the head points to the beginning of the array.