TIRGUL 2 in Data Structure

Elad Verbin

February 28, 2007

1 Amortized Analysis

Exercise: Give an implementation of a stack ADT using an array and $O(1)$ additional space, such that each operation costs $O(1)$ amortized running time, and the size of the array is always $O(n)$, where n is the number of elements in the stack.
Remark: This is like the extendable arrays that you have seen in class, except that here we require that the array is always of length $O(n)$. Here is a formal proof that the extendable arrays that you saw in class do not have this property (this is somewhat technical, so you may safely ignore it. It is in fact an exercise in proving that $f \neq O(g)$): Suppose that the extendable arrays were always of size $O(n)$. In the O-notation there is a constant hidden, say C. So the extendable arrays are always of size $\leq Cn$. Now push $C + 800$ elements into the stack, and then pop all of them but 1. The size of the array is still at least $C + 800$, but $n = 1$, and thus we have a contradiction to the size of the array always being $\leq Cn$.
Solution: The idea is that like in class, when the array is full and we wish to insert an element, we double its size. Now, also, whenever the array is only $\frac{1}{4}$-full, we halve its size. Here is the implementation in pseudo-code:

Data-Structure Representation:
maxsize – the size of the array
size – the size of the stack
A - an array of size maxsize

Initialization
size $\leftarrow$ 0
maxsize $\leftarrow$ 2
A $\leftarrow$ array of size 2

Push(S,x)
IF $(size < maxsize)$
    $A[size + 1] \leftarrow x$
    $size \leftarrow size + 1$
ELSE
    $A' \leftarrow$ array of size $maxsize * 2$
    copy A to A'
    $A \leftarrow A'$
    $maxsize \leftarrow maxsize * 2$
    $A[size + 1] \leftarrow x$
    $size \leftarrow size + 1$

Pop(S,x)
    $toreturn \leftarrow A[size]$

$size \leftarrow size - 1$
IF ($size < maxsize/4$ AND $maxsize > 2$)
$A' \leftarrow$ array of size $maxsize/2$
copy A to A'
$A \leftarrow A'$
$maxsize \leftarrow maxsize/2$
RETURN toreturn

Theorem 1.1 *The amortized cost of this data structure is $O(1)$. That is, any sequence of m operations costs $O(m)$ time.*

The intuition behind this is the following: Suppose we have just performed an expensive operation. Then now the array is exactly half-full (up to ± 1), and we need to perform at least $maxsize/4$ cheap operations before the next expensive operation, so the cheap operations can pay for the expensive operation that follows them. More formally:

Lemma 1.2 *Suppose op_1 is an expensive operation, and op_2 is the expensive operation that follows it. Suppose M is the size of the array after op_1. Then the number of (cheap) operations between op_1 and op_2 is at least $M/4$*

Proof. After op_1 there array is half-full, so the size of the stack is $M/2$. If op_2 is a POP operation, then it is performed when the size of the stack is $M/4$, so $M/4$ POP operations must have been performed in the meanwhile. If op_2 is a PUSH operation, then it is performed when the size of the stack is M, so $M/2$ PUSH operations must have been performed in the meanwhile. □

Now let us prove the theorem: In the bank method, we have a bank. Sometimes it is convenient to imagine that the tokens in the bank are "placed" in some manner over, say, the elements of the array, but we will not do this here. The bank begins at 0, and each operation can deposit or withdraw tokens from the bank. In the proof we need to give a protocol that says, for each operation, how much it deposits and how much it withdraws. The bank must always be non-empty. Then, we define the amortized running time of a single operation to be

$$amortized(op) = actual(op) - (\text{amount withdrawn}) + (\text{amount deposited}) \ .$$

Our goal here is to prove that $amortized(op) = O(1)$ for all operations.

In our proof, the protocol for deposits and withdrawals is as follows: we deposit 4 tokens in the bank in every operation, and in every expensive operation, we withdraw a number of tokens that is equal to the number of elements we copy in the expensive array-copy operation. So it is clear that for all operations $amortized(op) = O(1)$, and we just need to prove that the bank is never in overdraft. The reason for this is that, according to the Lemma, the cheap operations before an expensive operation always pay for it. Specifically: Let op_2 be the expensive operation, and let M be the $maxsize$ before it. Then the cost of the expensive copy operation is $maxsize$. But we have proved that before it there were made at least $maxsize/4$ cheap operations, and each one deposited 4 tokens, so we're ok.

Now let us prove the Theorem using the potential method. In the potential method, one defines a potential function Φ, which is a function from the set of possible configurations of the data structure, to the integers. The potential of the initial data structure should be 0, and the potential should always be non-negative. Define

$$amortized(op) = actual(op) - \Delta(\Phi) = actual(op) - \Phi(after) + \Phi(before) \ .$$

Our goal here is to prove that $amortized(op) = O(1)$ for all operations.

The bank method can always be used to get a valid potential, which is equal to the current balance in the bank. But to understand how the two methods are slightly different, try to actually give a closed expression for the balance in the bank at some point of time as a function of the operations that came before it. This is difficult. In the potential method, we give a function that is probably similar to the balance in the bank, but can be calculated easily given only the current status of the data structure.

Our potential here will be:

$$\Phi = \left(\begin{cases} 2(size - \frac{maxsize}{2}) & \text{if } size > maxsize/2, \\ \frac{maxsize}{2} - size & \text{if } size < maxsize/2 \end{cases} \right) - 1.$$

(the purpose of the -1 here is so that the potential at time 0 is equal to 0).

One can see that when the array is half-empty, the potential is 0. This is important, since our goal in defining a potential function is that the potential slowly rise when cheap operations are performed, and then sharply drop when expensive operations are performed. And indeed, after an expensive operation, the potential is always 0.

What is the potential before an expensive operation? If the array was full, then $size = maxsize$, and the potential is equal to $maxsize$, which is a good thing, since in the next operation we will incur a cost of $maxsize$ for copying the array. If the array was 1/4-full, then $size = maxsize/4$, and the potential is equal to $maxsize/4$, which is a good thing, since in the next operation we will incur a cost of $maxsize/4$ for copying the array. So it is easy to see that $amortized(op) = O(1)$ for all operations. It is also easy to see that the potential is always non-negative, so we are finished.

The reason that this potential works is that after an expensive operation we always have $\Phi = 0$. Then, when moving towards the situation where expensive operations need to be performed again, it slowly rises, until, before the expensive operation, it gets to a level where it can pay for the expensive operation.

2 Binary Search Trees

Exercise: Give a procedure that, given a binary search tree, prints all of the keys in it in increasing order. The operation should take $O(n)$ time, where n is the number of nodes in the tree.
Solution: We use a recursive procedure.

```
IN-ORDER(v)
IF (v = null) RETURN
ELSE
   IN-ORDER(v.left)
   PRINT v.key
   IN-ORDER(v.right)
   RETURN
```

The solution to the exercise is to run IN-ORDER on the root of the tree. It is not hard to see that this prints the keys of the tree in increasing order, and costs $O(n)$ time.