Dynamic Array

Arnab Ganguly, Assistant Professor
Department of Computer Science, University of Wisconsin – Whitewater
Data Structures (CS 223)

1 Dynamic Array

Given an array A, we want to support the following operations:

- $access(i)$: retrieve $A[i]$,
- $update(i, val)$: set $A[i] = val$,
- $insertAtEnd(val)$: add val at the end of the array,
- $deleteLast()$: remove the last element of A

1.1 The Algorithm

Implementing, *access* and *update* operations are straightforward. The other two operations are the trickier ones. The technique for them can be summarized as follows:

- We maintain three variables –
  - an array A to store the values,
  - $length$, which is the current length of A, and
  - $numElements$, which is the current number of values in the dynamic array.
- To implement $insertAtEnd$, the idea is to re-size A to double its length, whenever $length$ equals $numElements$.
- To implement *delete*, the idea is to re-size A to half its length, whenever $length$ equals $4 * numElements$.

See the detailed description of the methods in Algorithm 1.

Algorithm 1 Description of Dynamic Array

```
1: int A[ ];
2: int length; // the length of A, which is initialized to a power of 2 such as 1, 2, 4, 8, 16, ...,
3: int numElements; // initialized to 0;
4:
5: function ACCESS(int i)
6:     if (i < numElements) then
7:         return A[i]
8:     else
9:         print "Cannot return an element outside array range"
10:
11: function UPDATE(int i, int val)
12:     if (i < numElements) then
13:         A[i] = val
14:     else
15:         print "Cannot update outside array range"
16:
17: function INSERTATEND(int val)
18:     if (numElements == length) then
19:         update length to 2 * length
20:         create an array B of size length
21:         use a loop to copy first numElements numbers from A to B
22:         set A to B
23:     A[numElements++] = val
24:
25: function DELETELAST()
26:     if (numElements == 0) then
27:         print "Cannot delete from an empty array"
28:     else if (numElements == 1) then
29:         set numElements to 0
30:         set length to 1
31:         resize array A to 1
32:     else
33:         decrement numElements
34:         if (4 * numElements == length) then
35:             update length to 2 * numElements
36:             create an array B of size length
37:             use a loop to copy first numElements numbers from A to B
38:             set A to B
```

2 What is Amortized Analysis?

In the worst-case analysis of a data structure, we seek to answer questions of the following sort: *what is worst-case time (in the Big-O sense) to support an operation on the data structure?* For example, binary-search on an n-length array $A[\,]$ takes $O(\log n)$ time. Accessing $A[i]$ takes $O(1)$ time. Reporting all elements in $A[\,]$ takes $O(n)$ time.

In the best-case analysis, we want answer: *what is best-case time (in the Big-O sense) to support*

an operation on the data structure? For example, binary-search on an n-length array $A[\ ]$ takes $O(1)$ time. Accessing $A[i]$ takes $O(1)$ time. Reporting all elements in $A[\ ]$ takes $O(n)$ time.

However, worst-case or best-case complexity is often not a good indication of the performance, because they are either too pessimistic or too optimistic. In most cases, we want to find: **how a data structure behaves on average for a bulk of operations?**[1]

Typically, what we do is: take a large enough sequence of operations, and then average the time for the sequence of operations. So, in some sense amortized analysis means: *instead of looking at the complexity of each operation in a data structure, look at the total complexity of a suitably large number of operations in succession.*

Side remark: Here, we will describe one of the techniques for amortized analysis, known as the *Aggregate Method*. The idea is to sum up the cost for a suitably large number of operations and take the average. Two other techniques are *Accounting Method* and *Potential Method*.

2.1 Analysis of Dynamic Array

Let $N = numElements$ be the number of elements currently in the array A. Let $L = length$ be the total length allocated for A. Note that the following inequality holds:

$$\frac{L}{2} \leq N \leq L$$

- Operations *access* and *update* are easy to analyze. We only carry out a constant number of steps in each operation. Hence, each operation takes $O(1)$ time in the worst case. Obviously, best-case and amortized-case are both $O(1)$ per operation.

- Let us assume that we are only carrying out *insertAtEnd* operation on the data structure. Each insert takes $O(1)$ time in the best-case (when no resizing is required). In the worst-case, however, it takes $O(N)$ time, because we have to copy all the elements of A into B, reallocate space for A, and then copy everything back.

  Crucially, note that we will have to copy A to B, and back, only when we have consumed A completely, i.e., $N = L$.

  Suppose, at one instant in time, we carry out this array-to-array copying process, thereby spending at most $c_1 * N$ time for some constant integer $c_1 > 1$. For the next N insert operations, we will spend at most $c_2 * N$ time for some constant integer $c_2 > 1$, because we have N free slots where we can insert before re-sizing again. Therefore, over the N insertions, our total cost is $(c_1 N + c_2 N)$, because every other operation costs $O(1)$ time. The amortized cost per operation is

  $$\frac{c_1 * N + c_2 * N}{N} = c_1 + c_2 = O(1)$$

- Using similar arguments as above, we can show that *deleteLast* also needs $O(1)$ amortized time per operation. More careful and complicated techniques (such as by using potential method) can be used to show that both operations are $O(1)$ amortized time, even when they are interspersed.

[1] This is different from Average Case Analysis, which uses probabilistic techniques.