

## COMP2054-ADE

# The Vector Data Type, and Amortised Analysis

Andrew Parkes

<http://www.cs.nott.ac.uk/~pszajp/>

1

Today we do two topics – a type of data structure called a “Vector”, and a different kind of analysis called “Amortised Analysis”

## Vector ADT

- The “Vector” is an Abstract Data Type, ADT, corresponding to generalising the notion of the “Array” (Concrete Data Type, CDT)
- Key idea:
  - The “index” of an entry in an array can be thought of as the “number of elements preceding it”
  - E.g. in an array A, then A[2] has two elements, A[0], A[1] that precede it
  - In these lectures it is then called “rank”
  - The notion of “rank” can be more general than the idea of index:
    - Not necessarily implemented as “C-style with pointers”  $A[i]=*(A+i)$

2

Firstly, in abstract terms – think of the “public interface” in Java. The main idea of Vector is to make a more general version of a standard Array.

So as not to be totally tied to an actual array, instead of thinking of an “index” we think in terms of the rank of an element.

To do this just note that the index of an element in array is equal to the number of elements before it.

But then we call it the “rank” just to make it clear that it is not necessarily the same as the C-Style computation with pointers.

## The Vector ADT

- The **Vector** ADT is based on the array CDT, and stores a sequence of arbitrary objects
- An element can be accessed, inserted or removed by specifying its **rank, i.e. the number of elements preceding it**
- An exception is thrown if an incorrect rank is specified (e.g., a negative rank)
- Main vector operations:
  - object **elemAtRank(integer r)**: returns the element at rank r without removing it
  - object **replaceAtRank(integer r, object o)**: replace the element at rank with o and return the old element
  - **insertAtRank(integer r, object o)**: insert a new element o to have rank r
  - object **removeAtRank(integer r)**: removes and returns the element at rank r
- Additional operations **size()** and **isEmpty()**

3

The operations that we need to be able to perform are just the obvious ones.

We need to be able to read elements at an arbitrary rank in the Vector, but also to be able to insert and delete/remove.

We also have the usual auxiliary of something that gives the size – or if the data structure is empty.

## Vector as a Stack

- A common usage of a Vector is as a “Stack”
  - add and remove elements at the end
  - i.e. maximum rank values:
  - push(o) - `insertAtRank(size(), object o)` – hence the new element is after all the existing ones
  - pop() - `removeAtRank(size())` – hence the element removed is after all the existing ones
- This is very useful if reading elements from a file, and not knowing how many there will be

4

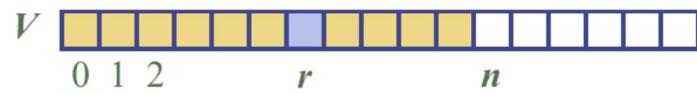
A lot of what we do today is concerned with

## Applications of Vectors

- There is not an automatic limit on the storage size
  - unlike arrays of a fixed size
- Direct applications
  - Sorted collection of objects (elementary database)
- Indirect applications
  - Auxiliary data structure for many algorithms
  - Components of other data structures

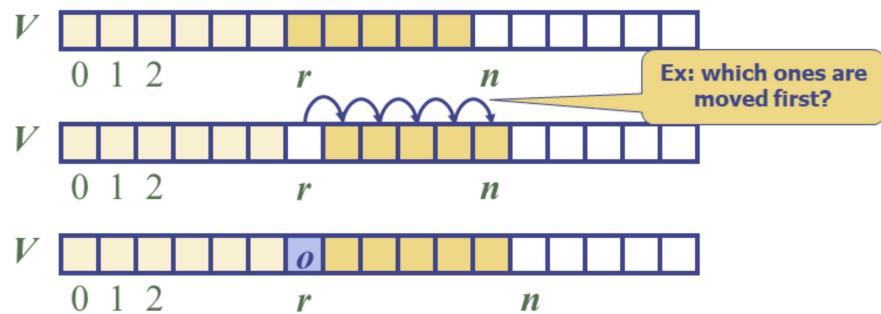
## Array-based Vector

- Use an array  $V$  of size  $N$  as the CDT
- A variable  $n$  keeps track of the size of the vector (number of elements currently stored)
- Operation  $\text{elemAtRank}(r)$  is implemented in  $O(1)$  time by simply returning  $V[r]$



## Insertion

- In operation  $\text{insertAtRank}(r, o)$ , we need to make room for the new element by shifting forward the  $n - r$  elements  $V[r], \dots, V[n - 1]$
- In the worst case ( $r = 0$ ), this takes  $O(n)$  time

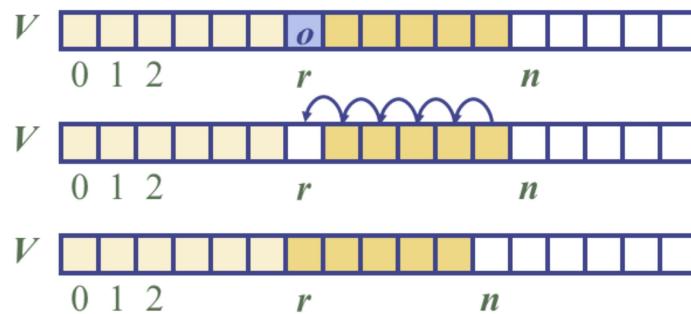


7

It is somewhat more efficient to do the moves from the right-hand side first and working towards the left, as this saves having to keep a temporary copy when moving.  
If confused, then try writing the fragment of code.

## Deletion

- In operation  $\text{removeAtRank}(r)$ , we need to fill the hole left by the removed element by shifting backward the  $n - r - 1$  elements  $V[r + 1], \dots, V[n - 1]$
- In the worst case ( $r = 0$ ), this takes  $O(n)$  time



8

## Performance

- In the array-based implementation of a Vector
  - The space used by the data structure is  $O(n)$
  - `size`, `isEmpty`, `elemAtRank` and `replaceAtRank` run in  $O(1)$  time
  - `insertAtRank` and `removeAtRank` run in  $O(n)$  time
  - `push` runs in  $O(1)$  time, as do not need to move elements
    - unless need to resize the array
  - `pop` runs in  $O(1)$  time
- In an `insertAtRank` operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one

## Growable Array-based Vector

- In a push (`insertAtRank(n)`) operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
  - But the resizing has a high cost –  $O(n)$  as need to copy all  $n$  elements
- How large should the new array be?
  - **incremental strategy:** increase size by a constant  $c$
  - **doubling strategy:** double the size

**Algorithm `push(o)`**

```
if  $n = V.length - 1$ 
then
     $A \leftarrow$  new array of
    size ...
    for  $i \leftarrow 0$  to  $t$  do
         $A[i] \leftarrow V[i]$ 
     $V \leftarrow A$ 
     $n \leftarrow n + 1$ 
     $V[t] \leftarrow o$ 
```

10

## Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time  $T(n)$  needed to perform a series of  $n$  push operations
  - “push” means “add an element at the end” – treat it as a stack
- We assume that we start with an empty stack represented by an array of size 1
- We call **amortized** time of a push operation the average time taken by a push over the series of operations, i.e.,  $T(n)/n$

11

## Meaning of “Amortize”

- See <http://www.thefreedictionary.com/amortize> or similar if you are not familiar with the word.
- It refers to writing off, or paying off, debts over a period of time.
- Similar to the way a mortgage for a house is paid back over many years, as opposed to needing to pay all in one go.

12

## Remarks on Amortised Analysis

- Suppose some individual operation (such as ‘push’) takes time  $T$  in the worst-case
- Suppose do a **sequence** of operations:
  - Suppose  $s$  such operations take total time  $T_s$
  - Then  $sT$  is an upper-bound for the total time  $T_s$
  - But, such an upper-bound might not ever occur.
- The time  $T_s$  might well be  $o(sT)$  even in the worst-case
  - the average time per operation,  $T_s / s$  can be the most relevant quantity in practice
  - E.g. if ‘push’ing a long sequence of elements into a Vector – e.g. when reading from a file

13

## Question (pause and try):

- Why is amortised analysis different from the average case analysis?

14

## Question:

- Why is amortised analysis different from the average case analysis?
- Answer:
  - “Amortised”: (long) **real sequence** of dependent operations
  - vs.
  - “Average”: **Set** of (possibly independent) operations

15

## Describing amortised costs

Note the usual big-Oh family is still used to describe amortised analysis

- Recall Big-oh is just describing functions
  - It is not limited to “worst case of an algorithm”!
- We have different measures of the runtime cost:
  - Worst-case “cost per operation of a sequence” not just
  - “Worst case of a single operation”

## Incremental: Example

- Take  $c=3$ , with start capacity of 3, then a sequence of pushes might have costs for each push as follows:

$1,1,1,3+1,1,1,6+1,1,1,9+1,1,1,12+1,1,1,15+$   
 $1,1,1,18+1,\dots$

- A constant fraction,  $1/3$ , of the pushes have cost  $O(n)$ .
- Average, per push operation, is  $O(n)$ .
- Much worse than  $O(1)$  cost without resizing!

17

The 'push' is because we are thinking of using the array as a Stack.

The "start capacity" is the size of the array when it is created – how many entries can be added before it needs to be resized.

The entry "3+1" is a sum of:

Cost 3 to copy the previous 3 entries to the new array

Cost 1 to then do the usual push.

And so one, but with the cost of copy increasing every time.

## Example: In detail

- [ - , - , - ] n=0 Starting point
  - empty, but capacity=3
  - “-” means “not yet used”
- Suppose do a sequence of “push(0)”, we get the sequence of costs (primitive operations):
  1. [ 0 , - , - ] n=1, **cost=1**
  2. [ 0 , 0 , - ] n=2, **cost=1**
  3. [ 0 , 0 , 0 ] n=3, **cost=1**
  4. [ 0 , 0 , 0 , 0 , - , - ] n=4, **cost=3+1**
    - cost= “3 for the copy” + “1 for the push(0)”
  5. [ 0 , 0 , 0 , 0 , 0 , - ] n=5, **cost=1**
  6. [ 0 , 0 , 0 , 0 , 0 , 0 ] n=6, **cost=1**
  7. [ 0 , 0 , 0 , 0 , 0 , 0 , 0 , - , - ] n=7, **cost=6+1**
    - cost= “6 for the copy” + “1 for the push(0)”
  8. etc, etc...

18

## Incremental Strategy Analysis

- We replace the array  $k = n/c$  times
- Each “replace” costs the current size
- The total time  $T(n)$  of a series of  $n$  push operations is proportional to

$$\begin{aligned}n + c + 2c + 3c + 4c + \dots + kc &= \\n + c(1 + 2 + 3 + \dots + k) &= \\n + ck(k + 1)/2\end{aligned}$$

- Since  $c$  is a constant,  $T(n)$  is  $O(n + k^2)$ , i.e.,  $O(n^2)$
- The amortized time of a push operation is  $O(n)$
- This is bad as the normal cost of a push is  $O(1)$ .

19

The first term ‘n’ is the cost for the  $n$  pushes, of  $O(1)$  each, and not taking account of the need to copy the array.

The remaining terms are to account for the costs of copying the array.

For the sum “ $1+2+3+\dots$ ” it is an arithmetic series; see the tutorial on series.

## Doubling: Example

- With start capacity of 3, then a sequence of pushes starting from an empty vector, might have costs for each push in turn of

1,1,1,3+1,1,1,6+1,1,1,1,1,1,12+1,1,1,1,1,1,...

- The 3,6,12,... are the costs for the resizing
- The fraction of pushes having cost  $O(n)$  drops with  $n$ .
- What is the average?

20

The idea is the same as for the incremental analysis, but because of the doubling, the cost of a copy is much less frequent.

## Doubling: Example

- With start capacity of 2, then a sequence of pushes might have costs

1,1,2+1,1,4+1,1,1,1,8+1,1,1,1,1,1,1,1,...,16+  
1,...

- For every push of cost  $O(n)$  we will be able to do another  $O(n)$  pushes of cost  $O(1)$  before having to resize again.
- The  $O(n)$  cost on resizing can be 'amortised' (spread) over  $n$  other  $O(1)$  operations
- Gives an average of  $O(1)$  per operation.

21

The start capacity 2 just gives a sequences of powers of 2, and so is a closer match to the next slide.

## Doubling: Example

- With start capacity of 2, then a sequence of pushes might have costs  
 $1, 1, 2+1, 1, 4+1, 1, 1, 1, \dots$
- The cost of the doubling can be “spread” over the later operations and so might be counted as  
 $1, 1, 1+1, \textcolor{red}{1+1}, 1+1, \textcolor{red}{1+1}, 1+1, \textcolor{red}{1+1}, \dots$   
where the red is a cost that has been moved.  
Analogy: save £1/day “for a rainy day”
- This ‘view’ makes it clearer that the net effect will just be a (rough) doubling of the original costs

22

The start capacity 2 just gives a sequences of powers of 2, and so is a closer match to the next slide.

## Doubling Strategy Analysis

- Question: for  $n$  ‘pushes’ how many times is the array grown?
- For simplicity assume  $n$  is a power of 2
  - We replace the array  $k = \log_2 n$  times
- The total time  $T(n)$  of a series of  $n$  push operations is proportional to
$$n + 1 + 2 + 4 + 8 + \dots + 2^{k-1}$$

23

## Examples

- $n=1$  #doublings=0
  - $\log_2 n = 0$
- $n=2$  #doublings=1
  - $\log_2 n = 1$
- $n=4$  #doublings=2
  - $\log_2 n = 2$

## Example: n=4

'|' is 'End of stack' marker

- start [ | \_ ]
  - capacity=1, size=0
- push(A) [ A | ]
- push(B) [ A B | ]
- needed: 1 double – 1 copies
- push(C) [ A B C | \_ ]
  - needed: 1 double – 2 copies
- push(D) [ A B C D | ]

Exercise (offline): Do this in full detail; as done earlier for the incremental strategy.

## Recall: Geometric sums

Want to find  $S$

$$S = 1 + 2 + 2^2 + 2^3 + \dots + 2^k$$

Standard trick:

$$2S = 2 + 2^2 + 2^3 + \dots + 2^k + 2^{k+1}$$

So

$$2S - S = 2^{k+1} - 1$$

$$\text{Hence } S = 2^{k+1} - 1$$

I.e. "the next term minus one"

## Doubling Strategy Analysis

- We replace the array  $k = \log_2 n$  times
- The total time  $T(n)$  of a series of  $n$  push operations is proportional to

$$\begin{aligned}n + 1 + 2 + 4 + 8 + \dots + 2^{k-1} = \\n + 2^k - 1 = 2n - 1\end{aligned}$$

$T(n)$  is  $O(n)$

- Amortized time of a single push operation is  $O(1)$
- That is, no worse than if all the needed memory was pre-assigned!
  - (Big-Oh hides the constant factor '2' extra cost.)

## Offline Exercises/Discussion

- When might you still want to use incremental increase rather than doubling?
- Why is it doubling rather than tripling? Or increasing by some other constant factor?
  - Try redoing the analysis with an arbitrary growth factor  $b$
  - As ever: *implement it all!*

## C/C++ comments

- Doubling vector size might be made even more efficient if use realloc rather than malloc or new
  - Internally it can (often not always) just extend the space allocated to the array, and so avoid the need for a copy
  - It can use “memcpy” which is direct copy rather than via individuals

## Expectations

- The “Vector” data structure
- The big-Oh costs of various single operations
- The amortised cost of a sequence of operations
- The amortised complexities of different strategies for resizing of the underlying array

30