Vectors

Eleni Mangina

Room B2.05
School of Computer Science and Informatics
University College Dublin, Ireland

Introduction

- Stacks and Queues are examples of ADTs that support insertion and removal at pre-specified points.
  - Stacks: insertion / removal at the top
  - Queues: insertion at the tail / removal at the head

- We have explored two basic techniques for implementing these two ADTs:
  - Arrays
  - Links

- Next, we will explore some ADTs that support insertion and removal at various points.
  - Collectively, these are known as **Sequence ADTs**.

Sequence ADTs

- Vectors
  - Support insertion / removal by rank (index).
- Lists
  - Support insertion / removal by position (I.e. you can insert a new item relative to an existing item).
- Sequences
  - Combination of a Vector & a List
- Iterators
  - Auxiliary ADT that supports traversal of other Sequence ADTs

Vectors: Concept

- A Vector supports insertion, removal and accessing of objects based on **rank**.

- Terminology:
  - Rank: The **rank of an object e** is an integer value that specifies the number of objects that come before e in vector.

- Example:

Rank ->	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
	A	B	C	D	F	G				

- Vectors are similar to **but not the same as** arrays:
  - KEY DIFFERENCES: No fixed capacity / object ranks change

Vectors: Insertion

- Insertion is performed based on a given rank.
  - E.g. insert Z with rank 4

- After insertion, the rank of some of the other objects in the vector may change

- Example: “Insert Z with rank 4”

0	1	2	3	4	5	6	7	8	9
A	B	C	D	F	G				

Vectors: Insertion

- Insertion is performed based on a given rank.
  - E.g. insert Z with rank 4

- After insertion, the rank of some of the other objects in the vector may change

- Example: “Insert Z with rank 4”

0	1	2	3	4	5	6	7	8	9
A	B	C	D	F	G				

0	1	2	3	4	5	6	7	8	9
A	B	C	D	**Z**	F	G			

Vectors: Removal

- Removal is performed based on a given rank.
  - E.g. Remove the item with rank 4
- Again, after removal, the rank of some of the other objects in the vector may change
- Example: “Remove the item with rank 3”

0	1	2	3	4	5	6	7	8	9
A	B	C	D	Z	F	G			

Vectors: Removal

- Removal is performed based on a given rank.
  - E.g. Remove the item with rank 4
- Again, after removal, the rank of some of the other objects in the vector may change
- Example: “Remove the item with rank 3”

0	1	2	3	4	5	6	7	8	9
A	B	C	D	Z	F	G			

0	1	2	3	4	5	6	7	8	9
A	B	C	Z	F	G				

Vectors: Function Specification

- Core Operations:
  - elemAtRank(r): Return the object with rank r: an error condition occurs if the vector is empty or r < 0 or r > size()–1
  - replaceAtRank(r, e): Replace the object at rank r with e: an error condition occurs if the vector is empty or r < 0 or r > size() –1
  - insertAtRank(r, e): Insert a object e into the vector with rank r: an error condition occurs if r < 0 or r > size()
  - removeAtRank(r): Remove the object at rank r: an error condition occurs if S is empty or r < 0 or r > size() –1

- Support Operations:
  - isEmpty() Returns true if the vector is empty, or false otherwise
  - size() Returns the number of elements in the vector

Vectors: Java Interface

```
public interface Vector {
   public int size();

   public boolean isEmpty();

   public Object elemAtRank(int rank)
               throws RankOutOfBoundsException;

   public Object replaceAtRank(int rank, Object element)
               throws RankOutOfBoundsException;

   public void insertAtRank(int rank, Object element)
               throws RankOutOfBoundsException;

   public Object removeAtRank(int rank)
               throws RankOutOfBoundsException;
}
```

Vectors: Impl. Strategies

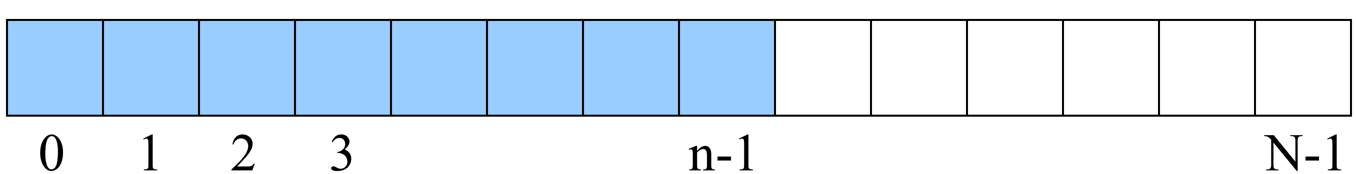
- Array-based Implementation:
  - Objects stored in an array
    - After each operation, the index of the object corresponds to the rank
  - Keep track of n, the current size of the vector
  - Finite Capacity (for now)

- Link-based Implementation:
  - Objects stored in special “nodes”
  - Nodes maintain ordering information
    - Link to the next and previous objects in the Vector.
    - Need auxiliary references for “front” and “rear” nodes.

Simple Array-based Impl.

- Approach: store the objects in an array, A, of size N
  - The rank of an object is very similar to the index of an array.
  - However, unlike an array, a Vector cannot have “gaps” between objects.
  - When an object is removed, the remaining objects must be squashed up to maintain the correspondance between the rank and index of an object.

- Below is a (Vector) array of size N that currently holds n objects.
  - The objects are located in indices 0 to n-1.



Vectors: Pseudo Code

```
Algorithm: insertAtRank(r, e):
    if (r < 0) or (r > n) then
      throw a RankOutOfBoundsException

    if (n == N) then
      throw a VectorFullException

    for i = n, n-2, …, r+1 do
      A[i] ← A[i-1]
    A[r] ← e
    n ← n + 1

Algorithm: elemAtRank(r):
    if (r < 0) or (r > n-1) then
      throw a RankOutOfBoundsException
    return A[r]

Algorithm isEmpty():
    return n == 0

Algorithm insertAtRank(r, e):
    if (r < 0) or (r > n-1) then
      throw a RankOutOfBoundsException
    e ← A[r]
    for i = r, r+1, …, n-2 do
      A[i] ← A[i+1]
    A[n-1] ← null
    n ← n – 1
    return e

Algorithm replaceAtRank(r,e):
    if (r < 0) or (r > n-1) then
      throw a RankOutOfBoundsException

    A[r] ← e

Algorithm size():
    return n
```

Vectors: Dry Runs

- Follow a similar pattern to other array-based implementations and record:
  - the array state
  - the number of objects in the array, and
  - the operations carried out

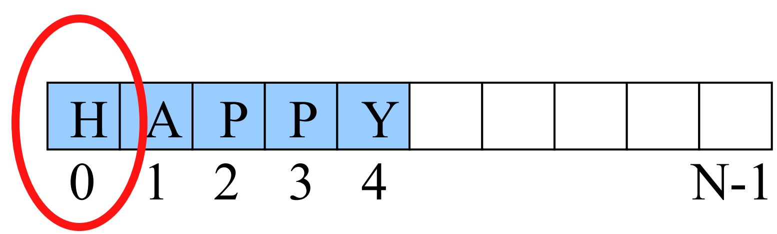
- Example: a Vector with array size 6

Operation	n	0	1	2	3	4	5
Initial State	0						
insertAtRank(0, A)	1	A					
insertAtRank(1, P)	2	A	P				
insertAtRank(2, Y)	3	A	P	Y			
insertAtRank(0, H)	4	H	A	P	Y		
insertAtRank(2, P)	5	H	A	P	P	Y	

Runtime Performance of Vectors

- The majority of methods for the Vector class have a running time of O(1).
- The exceptions to this are the insertAtRank(r,e) and the removeAtRank(r) methods.
  - These methods have O(n) running time because of the shift operations that must be performed on the arrays.
  - In the worst case, elements are inserted or removed with rank 0.
  - In this case, n-1 elements must be shifted, hence the O(n) running time.

Method	**Size**
size()	O(1)
isEmpty()	O(1)
elemAtRank(r)	O(1)
replaceAtRank(r,e)	O(1)
insertAtRank(r,e)	O(n)
removeAtRank(r)	O(n)



Extendable Arrays

- Current Approach: Array => Finite Capacity
  - At capacity throw an unchecked VectorFullException
- Alternative: Extendable Arrays
  - When the array is full, replace it with a new bigger array that is initialised to contain the current state of the Vector.
  - Typical Strategy: Double the array size
  - Impacts only the insertAtRank(r, e) operation
- Why double?
  - Fixed increments not efficient (10 + 1000 / 10000000 + 1000)
  - Doubling is the smallest integer multiple (vs tripling / quadrupling)
  - Tries to minimize wastage of space

Revised Insertion Algorithm

Algorithm: insertAtRank(r, e):

    if (r < 0) **or** (r > n) **then**

        throw a RankOutOfBoundsException

    if *n = capacity* ***then***

        capacity $\leftarrow$ *capacity * 2;*

        B $\leftarrow$ *new array of size capacity*

        for *i = 0, 1, …, n-1* ***do***

            B[i] $\leftarrow$ *A[i]*

        A $\leftarrow$ *B*

    for i = n-1, n-2, …, r **do**

        A[i+1] $\leftarrow$ A[i]

    A[r] $\leftarrow$ e

    n $\leftarrow$ n + 1

Amortization

- Goal: To calculate the running time for the algorithm that inserts items into an extendable array.
  - Let us consider only insertion at the end of the array (we can always add the O(n) shift operation later).
- Approach: Conduct an ***amortized analysis*** of the algorithm.
  - We adopt a view of the computer as a coin-operated appliance that takes one dollar as payment for a fixed amount of computing time.
  - When an operation is executed, we should have enough dollars in the "account" to pay for that operations running-time.
- NOTE: The idea behind this approach is to spread out the payments so that we get an idea for the overall perfomance of the algorithm.

Analysis of Insertion

- Assumptions:
  - One dollar is enough to pay for the insertion of an element into the vector S.
  - Growing the array from size k to size 2k costs k dollars.

- We charge 3 dollars per insertion.
  - So, we overcharge by 2 dollars, and store the additional cash in the "account".

- An overflow occurs when there are i elements in the array ($i \geq 0$).
  - Growing the array will cost 2i dollars.
  - Fortunately, this cash has accumulated in the account since the last overflow:

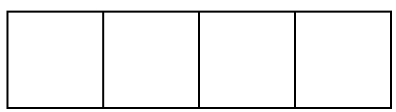
    2 dollars per operation * i operations = 2i dollars

Analysis of Insertion

- Start with an array of size 4, and add “H”, “A”, “P” , “P”, “Y”…

Account

Array



Analysis of Insertion

- Start with an array of size 4, and add “H”, “A”, “P” , “P”, “Y”…

Account $ $

Array

H			

Analysis of Insertion

- Start with an array of size 4, and add “H”, “A”, “P” , “P”, “Y”…

Account $ $
$ $

Array

H	A		

Analysis of Insertion

- Start with an array of size 4, and add “H”, “A”, “P” , “P”, “Y”…

Account $ $ $

$ $ $

Array

H	A	P	

Analysis of Insertion

- Start with an array of size 4, and add “H”, “A”, “P” , “P”, “Y”…

Account

$ $ $ $
$ $ $ $

Array

H	A	P	P

Analysis of Insertion

- Start with an array of size 4, and add “H”, “A”, “P” , “P”, “Y”…

Account $ $ $ $
$ $ $ $

Array

H	A	P	P

Overflow occurred adding “Y”…

Cost of extending an array from k to 2k costs k dollars

Currently the array is size 4, hence cost is 4

Analysis of Insertion

- Start with an array of size 4, and add “H”, “A”, “P” , “P”, “Y”…

Account $ $ $ $

Array

H	A	P	P				

But we still have to add “Y” at a price of 3

Analysis of Insertion

- Start with an array of size 4, and add “H”, “A”, “P” , “P”, “Y”…

Account

$ $ $

$ $ $

Array

H	A	P	P	Y			

The outstanding 4 dollars comes from the initial size of the array (4) – this never gets spent! Keep on adding “B” “I” “R” “T” to confirm!

Analysis of Insertion

- Now adding “B” “I” “R” “T” to confirm analysis

Account	$	$			$	$		
	$	$			$	$		
Array	H	A	P	P	Y	B		

Analysis of Insertion

- Now adding “B” “I” “R” “T” to confirm analysis

Account	$	$			$	$	$	
	$	$			$	$	$	
Array	H	A	P	P	Y	B	I	

Analysis of Insertion

- Now adding “B” “I” “R” “T” to confirm analysis

Account	$ $	$ $			$ $	$ $	$ $	$ $
Array	H	A	P	P	Y	B	I	R

Analysis of Insertion

- Now adding “B” “I” “R” “T” to confirm analysis

Account

\$	\$			\$	\$	\$	\$
\$	\$			\$	\$	\$	\$

Array

H	A	P	P	Y	B	I	R

Overflow occurred adding “T”…
Cost of extending an array from k to 2k costs k dollars
Currently the array is size 8, hence cost is 8

Analysis of Insertion

- Now adding “B” “I” “R” “T” to confirm analysis

Account

\$ \$
\$ \$

Array

H	A	P	P	Y	B	I	R					. . .

And finally, we add the “T”

Analysis of Insertion

- Now adding “B” “I” “R” “T” to confirm analysis

Account	$ $	$ $						$ $					
Array	H	A	P	P	Y	B	I	R	T				. . .

Analysis of Insertion

- This means that, 3 dollars per insertion covers the cost of extending the array (whenever that extension occurs).

- So, if we insert n items into an extendable vector, then the cost will be 3n dollars:
  - This means that inserting n items takes O(n) time.
  - This is because the ***amortized running time*** of each insertion is O(1)!

I DIDN’T EXPECT THAT ONE!

Linked Lists and Vectors

- Given the success we had with linked lists and Stacks and Queues, why don't we use them for Vectors?
- REMEMBER: In a linked list, we store direct references to only the first and the last nodes in the list.
  - To get to the other nodes, we must "follow the links".
- If there are "n" links in a linked list implementation of a Vector, then in the worst case, traversing the list to a given rank in the list will take n-1 steps.
  - Can improve this to at most n/2 steps but still not O(1)!

Linked Lists and Vectors

- Each Vector update method requires we traverse the list to a specified rank.
  - This means that ALL of these methods will have a running time of **O(n)**.
- For insertAtRank(r,e) and removeAtRank(r):
  - We have replaced the need for shift operation with the need for a traversal operation.
- For elemAtRank(r) and replaceAtRank(r,e):
  - We have ***introduced*** the need for a traversal operation.

Method	Size
size()	O(1)
isEmpty()	O(1)
elemAtRank(r)	O(n)
replaceAtRank(r,e)	O(n)
insertAtRank(r,e)	O(n)
removeAtRank(r)	O(n)

- This means that array-based implementations of Vectors are better!