Vectors and Array Lists
The Vector ADT (§5.1)

The Vector ADT extends the notion of array by storing a sequence of arbitrary objects. An element can be accessed, inserted or removed by specifying its rank (number of elements preceding it). An exception is thrown if an incorrect rank is specified (e.g., a negative rank).

Main vector operations:

- object \texttt{elemAtRank}(integer r): returns the element at rank \( r \) without removing it.
- object \texttt{replaceAtRank}(integer r, object o): replace the element at rank \( r \) with \( o \) and return the old element.
- \texttt{insertAtRank}(integer r, object o): insert a new element \( o \) to have rank \( r \).
- object \texttt{removeAtRank}(integer r): removes and returns the element at rank \( r \).

Additional operations \texttt{size()} and \texttt{isEmpty()}. 

© 2004 Goodrich, Tamassia

Vectors
Applications of Vectors

Direct applications

- Sorted collection of objects (elementary database)

Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures
Array-based Vector

- Use an array $V$ of size $N$
- A variable $n$ keeps track of the size of the vector (number of elements stored)
- Operation $elemAtRank(r)$ is implemented in $O(1)$ time by returning $V[r]$. 

![Diagram of array-based vector](image)
In operation \( \text{insertAtRank}(r, o) \), we need to make room for the new element by shifting forward the \( n - r \) elements \( V[r], \ldots, V[n - 1] \).

In the worst case (\( r = 0 \)), this takes \( O(n) \) time.
Deletion

In operation \textit{removeAtRank}(r), we need to fill the hole left by the removed element by shifting backward the \( n - r - 1 \) elements \( V[r + 1], \ldots, V[n - 1] \).

In the worst case \( (r = 0) \), this takes \( O(n) \) time.
Performance

In the array based implementation of a Vector
- The space used by the data structure is \( O(n) \)
- \textit{size}, \textit{isEmpty}, \textit{elemAtRank} and \textit{replaceAtRank} run in \( O(1) \) time
- \textit{insertAtRank} and \textit{removeAtRank} run in \( O(n) \) time

If we use the array in a circular fashion, \textit{insertAtRank}(0) and \textit{removeAtRank}(0) run in \( O(1) \) time

In an \textit{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 operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one.

How large should the new array be?

- incremental strategy: increase the size by a constant \( c \)
- doubling strategy: double the size

Algorithm `push(o)`

```plaintext
if t = S.length - 1 then
    A ← new array of size ...
    for i ← 0 to t do
        A[i] ← S[i]
    S ← A
    t ← t + 1
    S[t] ← o
```
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.
- 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., $T(n)/n$. 
Incremental Strategy Analysis

- We replace the array \( k = n/c \) times
- The total time \( T(n) \) of a series of \( n \) push operations is proportional to
  \[
  n + c + 2c + 3c + 4c + \ldots + kc = n + c(1 + 2 + 3 + \ldots + k) = n + ck(k + 1)/2
  \]
- 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 \( O(n) \)
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:
  \[ n + 1 + 2 + 4 + 8 + \ldots + 2^k = n + 2^{k+1} - 1 = 2n - 1 \]
- $T(n)$ is $O(n)$.
- The amortized time of a push operation is $O(1)$.