Array Lists
The Array List ADT

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

Main methods:
- `get(integer i)`: returns the element at index i without removing it.
- `set(integer i, object o)`: replace the element at index i with o and return the old element.
- `add(integer i, object o)`: insert a new element o to have index i.
- `remove(integer i)`: removes and returns the element at index i.

Additional methods:
- `size()`
- `isEmpty()`
Applications of Array Lists

- **Direct applications**
  - Sorted collection of objects (elementary database)

- **Indirect applications**
  - Auxiliary data structure for algorithms
  - Component of other data structures
Array-based Implementation

- Use an array $A$ of size $N$
- A variable $n$ keeps track of the size of the array list (number of elements stored)
- Operation $\text{get}(i)$ is implemented in $O(1)$ time by returning $A[i]$
- Operation $\text{set}(i, o)$ is implemented in $O(1)$ time by performing $t = A[i], A[i] = o$, and returning $t$.
Insertion

- In operation \textit{add}(i, o), we need to make room for the new element by shifting forward the \( n - i \) elements \( A[i], \ldots, A[n - 1] \).
- In the worst case \((i = 0)\), this takes \( O(n) \) time.
Element Removal

- In operation **remove**(i), we need to fill the hole left by the removed element by shifting backward the \( n - i - 1 \) elements \( A[i + 1], \ldots, A[n - 1] \)
- In the worst case (\( i = 0 \)), this takes \( O(n) \) time
Performance

- In the array based implementation of an array list:
  - The space used by the data structure is $O(n)$
  - `size`, `isEmpty`, `get` and `set` run in $O(1)$ time
  - `add` and `remove` run in $O(n)$ time in worst case
- If we use the array in a circular fashion, operations `add(0, x)` and `remove(0, x)` run in $O(1)$ time
- In an `add` operation, when the array is full, instead of throwing an exception, we can replace the array with a larger one
Growable Array-based Array List

- In an `add(o)` operation (without an index), we always add at the end.
- When the array is full, we 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 add(o)**

```plaintext
if t = S.length - 1 then
    A ← new array of size …
    for i ← 0 to n-1 do
        A[i] ← S[i]
    S ← A
    n ← n + 1
    S[n-1] ← 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 $n$ add(o) operations.
- We assume that we start with an empty stack represented by an array of size 1.
- We call amortized time of an add operation the average time taken by an add 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 \( n \) add 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 an add 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 $n$ add operations is proportional to
  
  $n + 1 + 2 + 4 + 8 + \ldots + 2^k = n + 2^{k+1} - 1 = 3n - 1$

- $T(n)$ is $O(n)$
- The amortized time of an add operation is $O(1)$