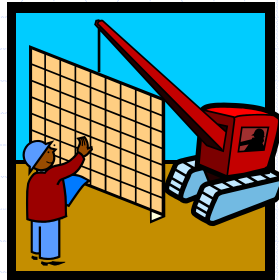


Array Lists



The Array List ADT

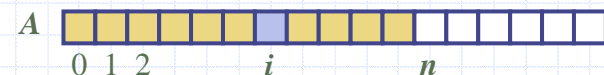
- The **Vector** or **Array List** ADT extends the notion of array by storing a sequence of objects
- An element can be accessed, inserted or removed by specifying its **index** (number of elements preceding it)
- An exception is thrown if an incorrect index is given (e.g., a negative index)
- Main methods:
 - **at**(integer i): returns the element at index i without removing it
 - **set**(integer i , object o): replace the element at index i with o
 - **insert**(integer i , object o): insert a new element o to have index i
 - **erase**(integer i): removes element at index i
- Additional methods:
 - **size**()
 - **empty**()

Applications of Array Lists

- Direct applications
 - Sorted collection of objects (elementary database)
- Indirect applications
 - Auxiliary data structure for algorithms
 - Component of other data structures

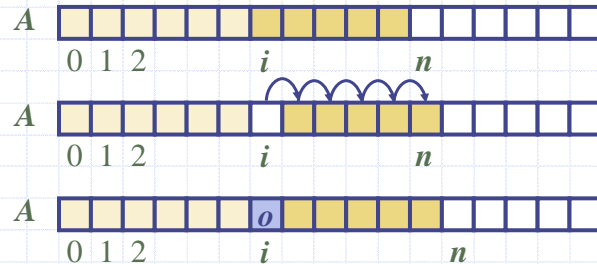
Array-based Implementation

- Use an array A of size N
- A variable n keeps track of the size of the array list (number of elements stored)
- Operation **at**(i) is implemented in $O(1)$ time by returning $A[i]$
- Operation **set**(i, o) is implemented in $O(1)$ time by performing $A[i] = o$



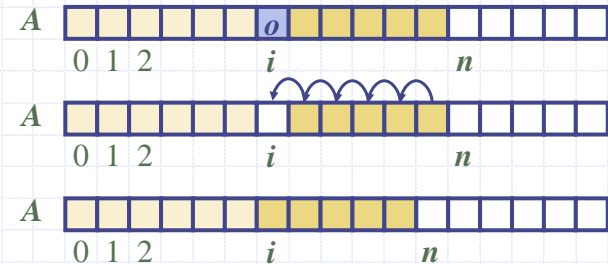
Insertion

- In operation **insert**(i, o), we need to make room for the new element by shifting forward the $n - i$ elements $A[i], \dots, A[n - 1]$
- In the worst case ($i = 0$), this takes $O(n)$ time



Element Removal

- In operation **erase**(i), we need to fill the hole left by the removed element by shifting backward the $n - i - 1$ elements $A[i + 1], \dots, A[n - 1]$
- 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**, **empty**, **at** and **set** run in $O(1)$ time
 - **insert** and **erase** run in $O(n)$ time in worst case
- If we use the array in a circular fashion, operations **insert**($0, x$) and **erase**($0, x$) run in $O(1)$ time
- In an **insert** 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 **insert**(o) operation (without an index), we always insert 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 insert(o)
  if  $t = S.length - 1$  then
     $A \leftarrow$  new array of size ...
    for  $i \leftarrow 0$  to  $n-1$  do
       $A[i] \leftarrow S[i]$ 
     $S \leftarrow A$ 
     $n \leftarrow n + 1$ 
     $S[n-1] \leftarrow o$ 
    
```

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 insert(o) operations
- We assume that we start with an empty stack represented by an array of size 1
- We call amortized time of an insert operation the average time taken by an insert 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 insert operations is proportional to

$$n + c + 2c + 3c + 4c + \dots + kc =$$

$$n + c(1 + 2 + 3 + \dots + k) =$$

$$n + ck(k+1)/2$$

- Since c is a constant, $T(n)$ is $O(n + k^2)$, i.e., $O(n^2)$
- The amortized time of an insert 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 insert operations is proportional to

$$n + 1 + 2 + 4 + 8 + \dots + 2^k =$$

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

$$3n - 1$$

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

geometric series

