ITMO UNIVERSITY

How to Win Coding Competitions: Secrets of Champions

Week 2: Computational complexity. Linear data structures
Lecture 3: Vector

Pavel Krotkov
Saint Petersburg 2016

Let's define operations we need for this data structure.

Let's define operations we need for this data structure.
O(1) operations

Let's define operations we need for this data structure.
O(1) operations

- inserting an element to the end of vector

Let's define operations we need for this data structure.
$O(1)$ operations

- inserting an element to the end of vector
- removing an element from the end of vector

Let's define operations we need for this data structure.
$O(1)$ operations

- inserting an element to the end of vector
- removing an element from the end of vector
- accessing an element by its index

Let's define operations we need for this data structure.
O(1) operations

- inserting an element to the end of vector
- removing an element from the end of vector
- accessing an element by its index

All other operations can be performed in linear time.

All elements are stored in some array.

- 0-th element of vector corresponds to 0-th element of array

All elements are stored in some array.

- 0-th element of vector corresponds to 0-th element of array
- 1-st element of vector corresponds to 1-st element of array

All elements are stored in some array.

- 0-th element of vector corresponds to 0-th element of array
- 1-st element of vector corresponds to 1-st element of array
- etc.

Array length will be always equal to some power of 2.

Array length will be always equal to some power of 2.

- we will have some space reserved for new elements.

Array length will be always equal to some power of 2.

- we will have some space reserved for new elements.

Example

- vector of 3 elements

3	6	2	

Extra space

Array length will be always equal to some power of 2.

- we will have some space reserved for new elements.

Example

- vector of 3 elements

3	6	2	

- vector of 5 elements

3	6	2	5	8			

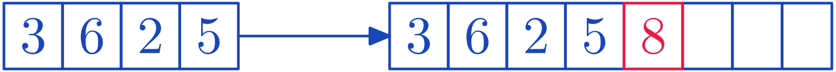
Almost always inserting an element is very simple.

Almost always inserting an element is very simple.

Special case: vector size is equal to array size.

Almost always inserting an element is very simple.

Special case: vector size is equal to array size.
Let's allocate array of doubled size, copy all elements and insert new element.



Let's analyze complexity of insert and access operations.

Let's analyze complexity of insert and access operations.

- accessing an element always takes $O(1)$ operations
    - just accessing corresponding element of array

Let's analyze complexity of insert and access operations.

- accessing an element always takes $O(1)$ operations
  - just accessing corresponding element of array
- removing an element always takes $O(1)$ operations
  - just decreasing current size of vector

Let's analyze complexity of insert and access operations.

- accessing an element always takes $O(1)$ operations
  - just accessing corresponding element of array
- removing an element always takes $O(1)$ operations
  - just decreasing current size of vector
- inserting an element almost always takes $O(1)$ operations
  - just increasing size of vector and assigning value to corresponding element of array

Let's analyze complexity of insert and access operations.

- accessing an element always takes $O(1)$ operations
  - just accessing corresponding element of array
- removing an element always takes $O(1)$ operations
  - just decreasing current size of vector
- inserting an element almost always takes $O(1)$ operations
  - just increasing size of vector and assigning value to corresponding element of array
  - except cases when size of vector is equal to size of array

Let's analyze complexity of insert and access operations.

- accessing an element always takes $O(1)$ operations
  - just accessing corresponding element of array
- removing an element always takes $O(1)$ operations
  - just decreasing current size of vector
- inserting an element almost always takes $O(1)$ operations
  - just increasing size of vector and assigning value to corresponding element of array
  - except cases when size of vector is equal to size of array
  - we'll call such operations *long* insert operations

Let's say we performed k insert operations.

Let's say we performed k insert operations.

- we performed some *long* insert operations
  - for sizes of vector $1, 2, \ldots, 2^{\lfloor log_2 k \rfloor}$

Let's say we performed k insert operations.

- we performed some *long* insert operations
  - for sizes of vector $1, 2, \ldots, 2^{\lfloor log_2 k \rfloor}$
- each *long* insert takes $O(n)$ operations, where n is current size of vector

Let's say we performed k insert operations.

- we performed some *long* insert operations
  - for sizes of vector $1, 2, \ldots, 2^{\lfloor log_2 k \rfloor}$
- each *long* insert takes $O(n)$ operations, where n is current size of vector
- total time of these insert operations is $O(1 + 2 + \ldots + 2^{\lfloor log_2 k \rfloor}) = O(2 \times 2^{\lfloor log_2 k \rfloor}) = O(k)$

Let's say we performed k insert operations.

- we performed some *long* insert operations
  - for sizes of vector $1, 2, \ldots, 2^{\lfloor log_2 k \rfloor}$
- each *long* insert takes $O(n)$ operations, where n is current size of vector
- total time of these insert operations is $O(1 + 2 + \ldots + 2^{\lfloor log_2 k \rfloor}) = O(2 \times 2^{\lfloor log_2 k \rfloor}) = O(k)$
- that means that *amortized* complexity of every insert operation is $O(1)$

We have a lot of unused allocated memory in case we did a lot of removes.

We have a lot of unused allocated memory in case we did a lot of removes.

- let's decrease size of array if we have less then $\frac{1}{3}$ of elements in use

We have a lot of unused allocated memory in case we did a lot of removes.

- let's decrease size of array if we have less then $\frac{1}{3}$ of elements in use
- it can be proved that even in this case *amortized* complexity of all operations is $O(1)$

Thank you
for your attention!