Basic Data Structures: Dynamic Arrays and Amortized Analysis

Data Structures

Data Structures and Algorithms

Outline

1. Dynamic Arrays
2. Amortized Analysis-Aggregate Method

Problem: static arrays are static!

```
int my_array[100];
```

Problem: static arrays are static!

```
int my_array[100];
```

Semi-solution: dynamically-allocated arrays:

```
int *my_array = new int[size];
```

Problem: might not know max size when allocating an array

Problem: might not know max size when allocating an array

> *All problems in computer science can be solved by another level of indirection.*

Problem: might not know max size when allocating an array

> *All problems in computer science can be solved by another level of indirection.*

Solution: *dynamic arrays* (also known as *resizable arrays*)

Idea: store a pointer to a dynamically allocated array, and replace it with a newly-allocated array as needed.

Definition

Dynamic Array:

Abstract data type with the following operations (at a minimum):

---

*must be constant time

Definition

Dynamic Array:

Abstract data type with the following operations (at a minimum):

- `Get`(*i*): returns element at location *i* *

---

*must be constant time

Definition

Dynamic Array:

Abstract data type with the following operations (at a minimum):

- `Get`(*i*): returns element at location *i* *
- `Set`(*i*, *val*): Sets element *i* to *val**

*must be constant time

Definition

Dynamic Array:

Abstract data type with the following operations (at a minimum):

- `Get`(*i*): returns element at location *i* *
- `Set`(*i*, *val*): Sets element *i* to *val**
- PushBack(*val*): Adds *val* to the end

---

*must be constant time

Definition

Dynamic Array:

Abstract data type with the following operations (at a minimum):

- `Get`(*i*): returns element at location *i* *
- `Set`(*i*, *val*): Sets element *i* to *val**
- PushBack(*val*): Adds *val* to the end
- `Remove`(*i*): Removes element at location *i*

---

*must be constant time

Definition

Dynamic Array:

Abstract data type with the following operations (at a minimum):

- `Get`(*i*): returns element at location *i* *
- `Set`(*i*, *val*): Sets element *i* to *val**
- PushBack(*val*): Adds *val* to the end
- `Remove`(*i*): Removes element at location *i*
- `Size`(): the number of elements

---

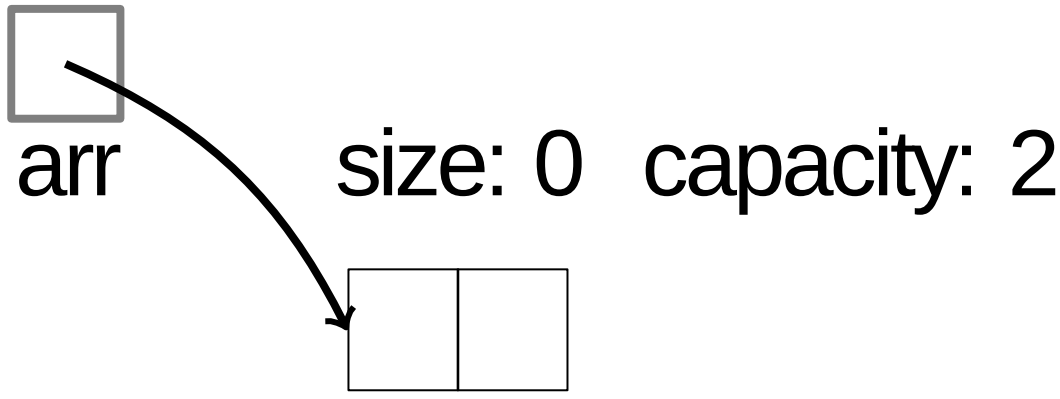
*must be constant time

Implementation

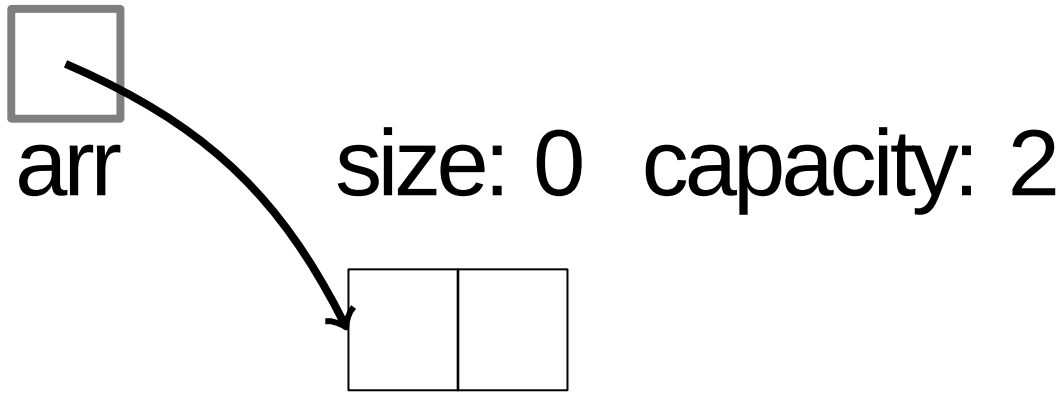
Store:

- `arr`: dynamically-allocated array
- `capacity`: size of the dynamically-allocated array
- `size`: number of elements currently in the array

Dynamic Array Resizing

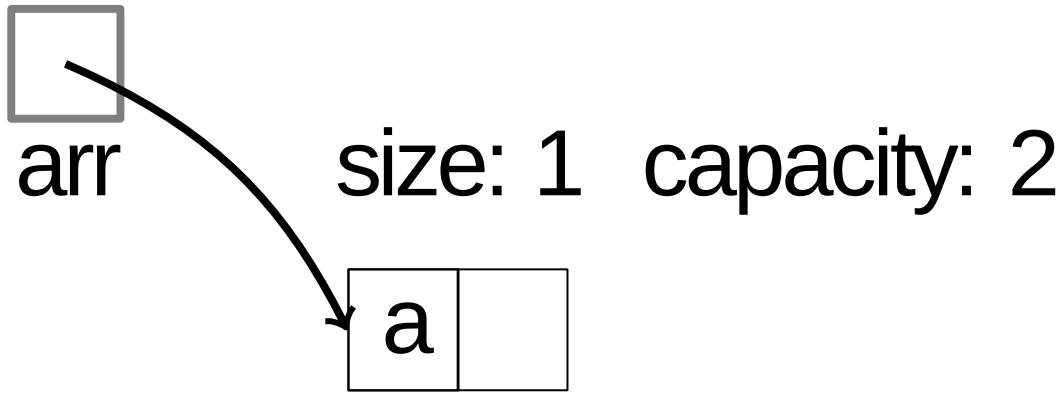


Dynamic Array Resizing



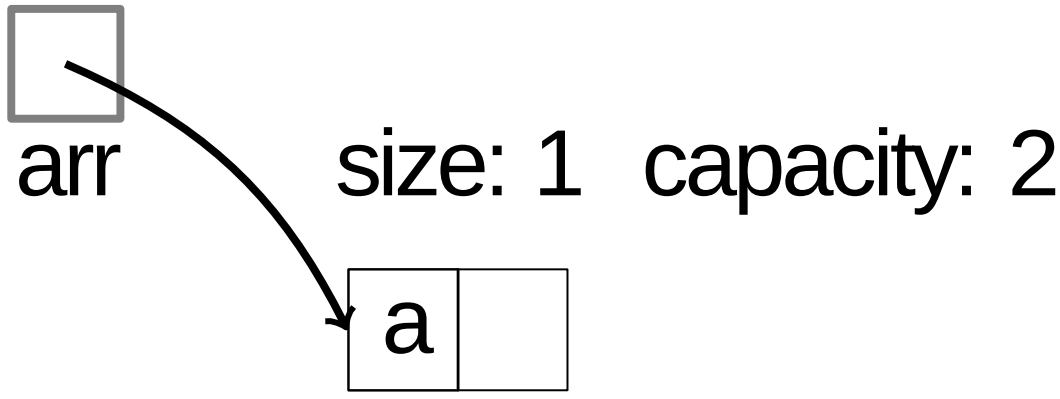
`PushBack(a)`

Dynamic Array Resizing

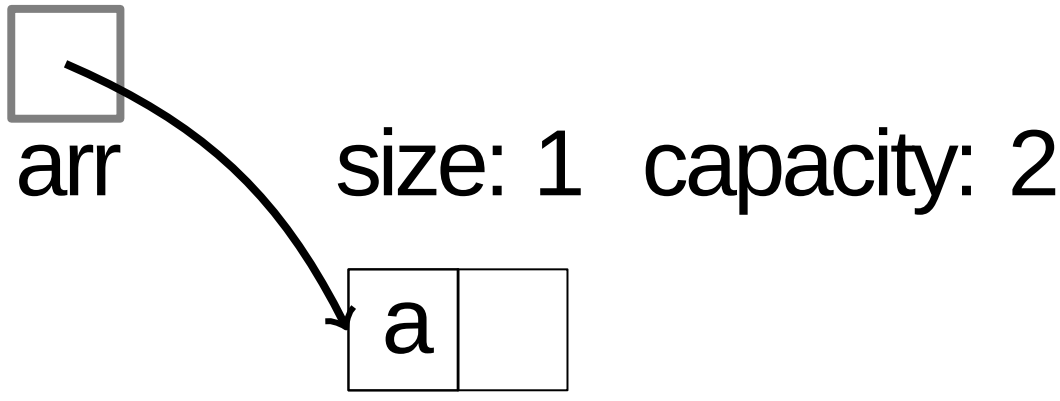


`PushBack(a)`

Dynamic Array Resizing

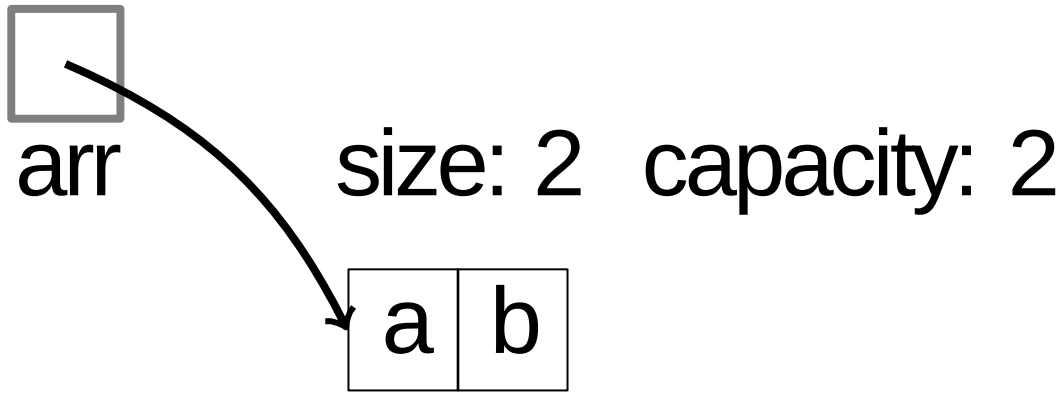


Dynamic Array Resizing



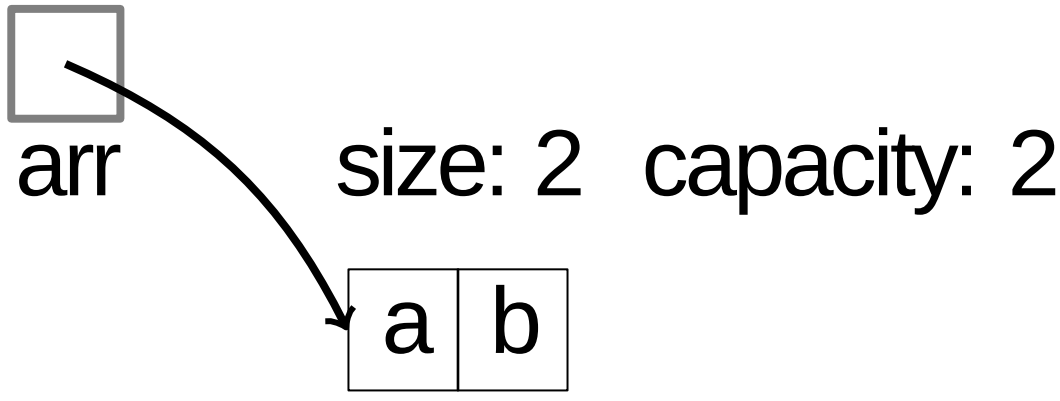
`PushBack(b)`

Dynamic Array Resizing

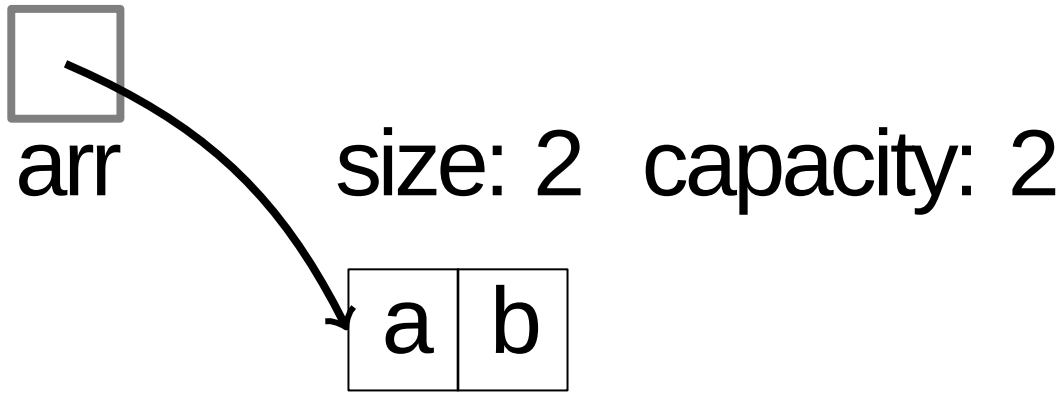


`PushBack(b)`

Dynamic Array Resizing

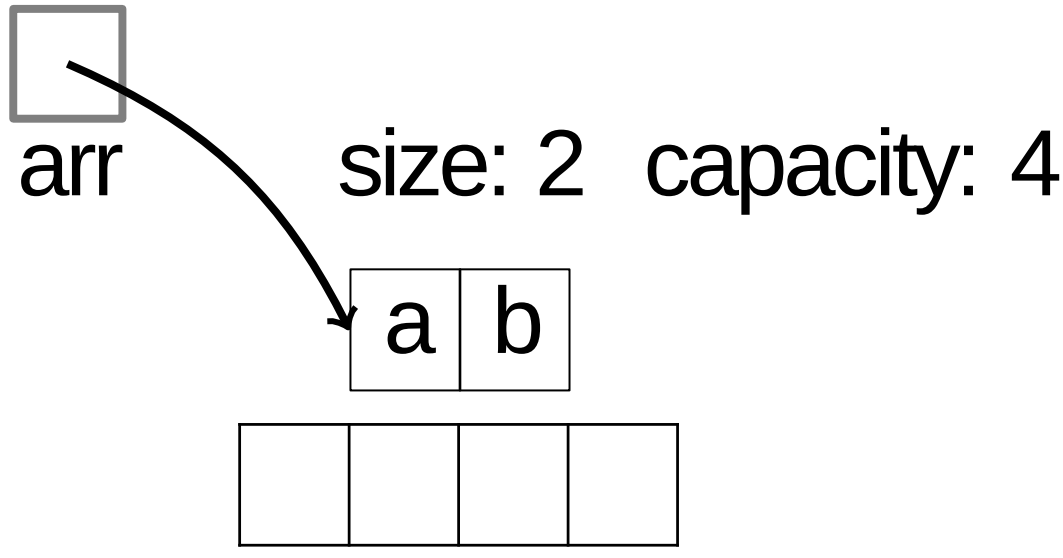


Dynamic Array Resizing



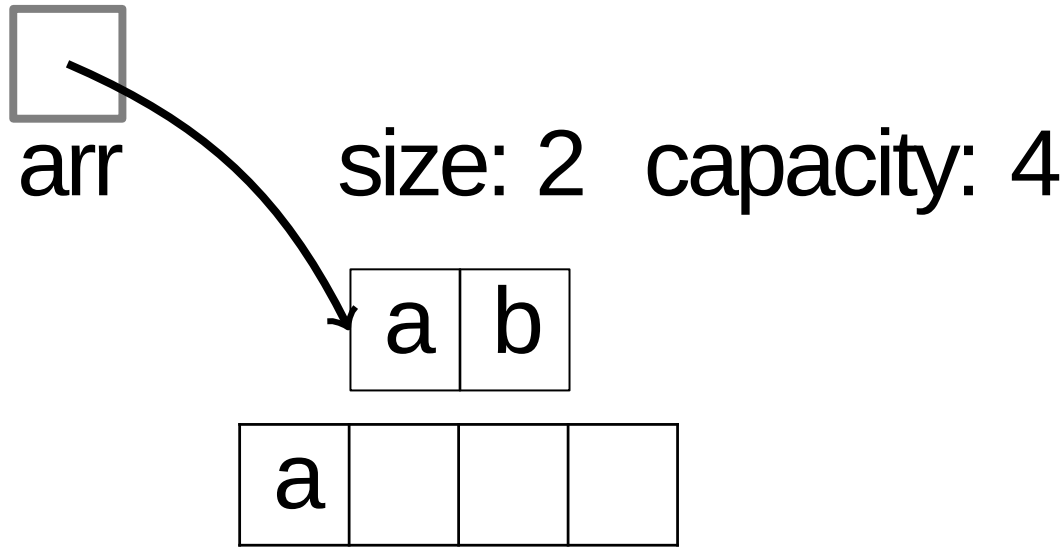
`PushBack(c)`

Dynamic Array Resizing



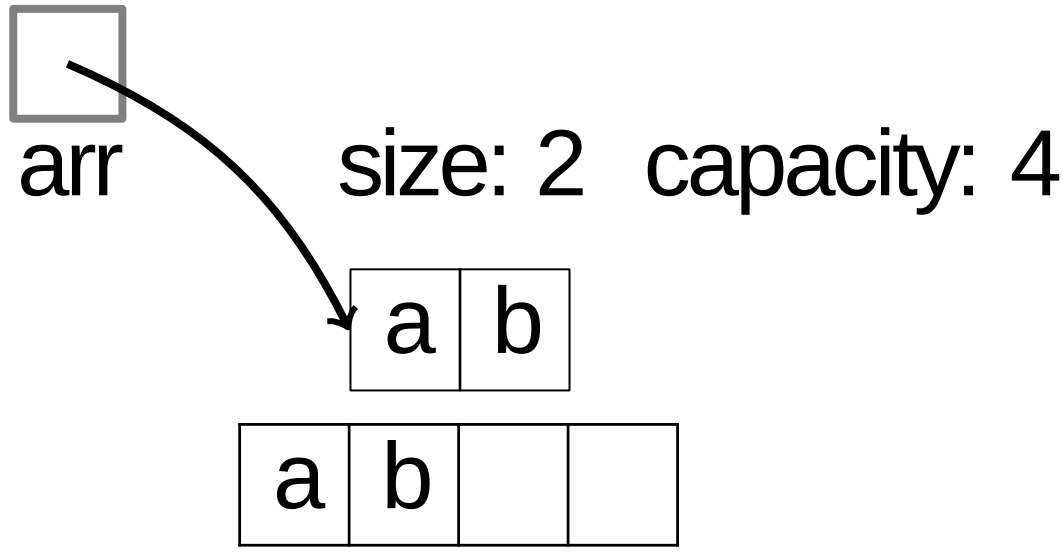
`PushBack(c)`

Dynamic Array Resizing



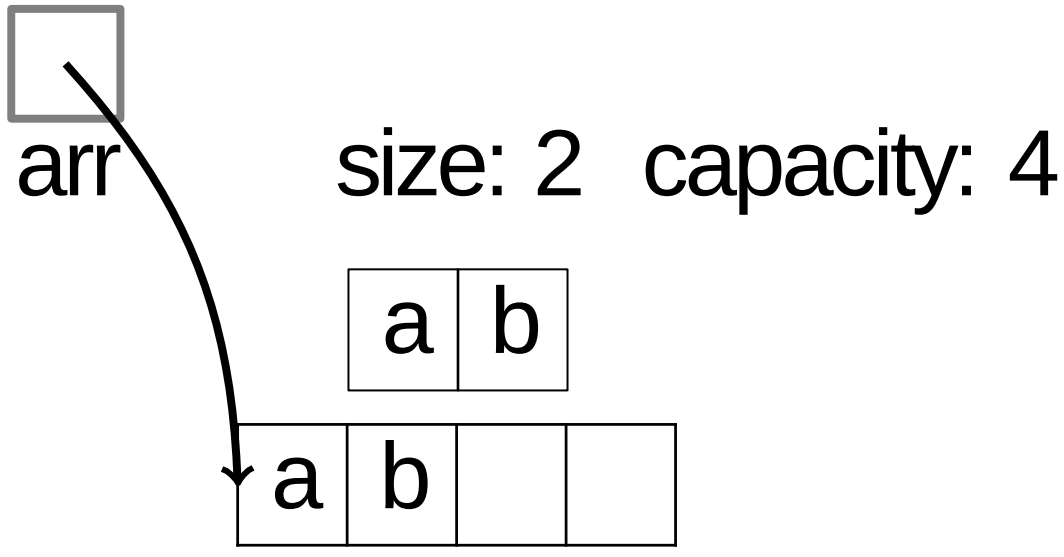
`PushBack(c)`

Dynamic Array Resizing



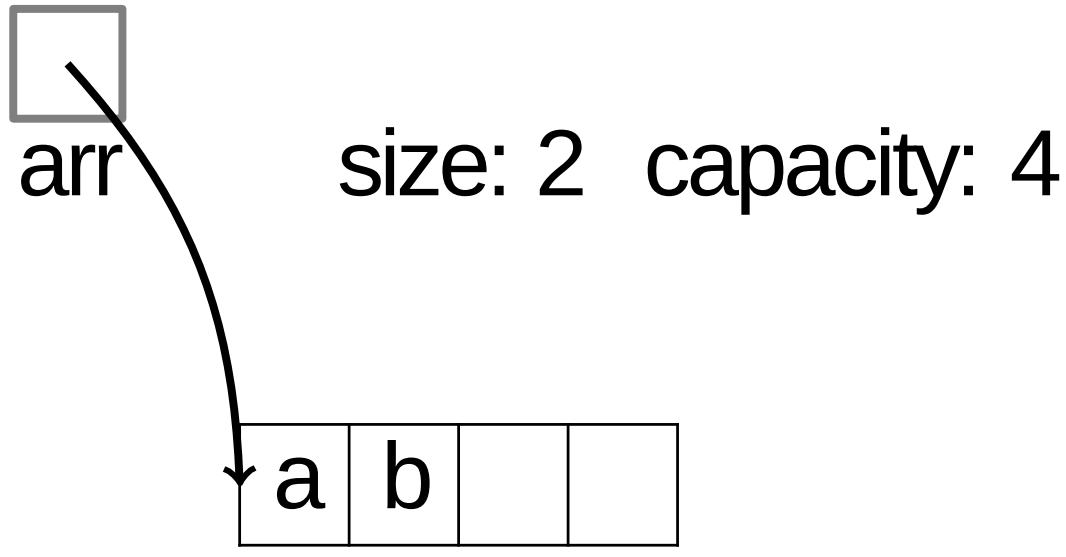
`PushBack(c)`

Dynamic Array Resizing



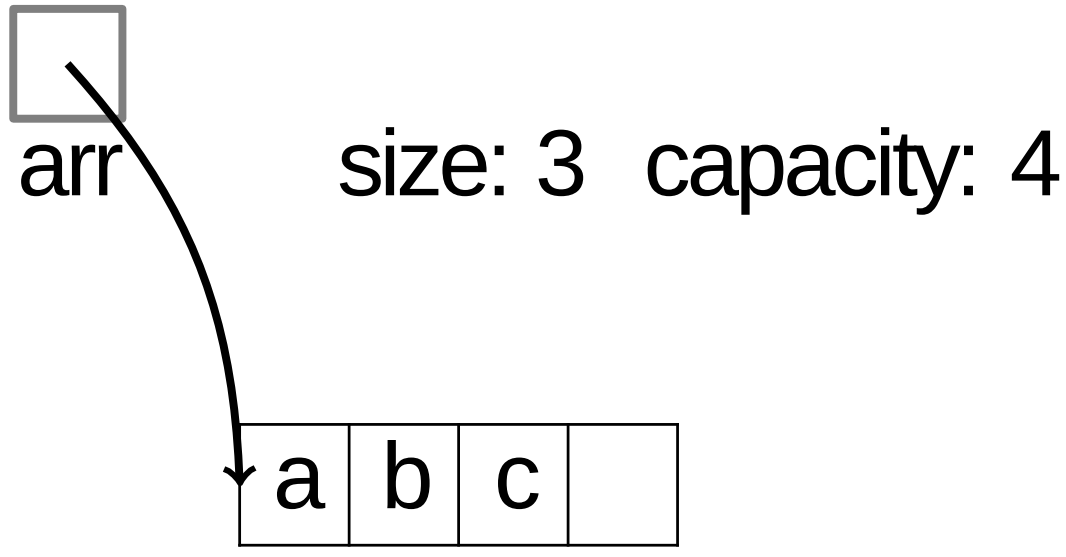
`PushBack(c)`

Dynamic Array Resizing



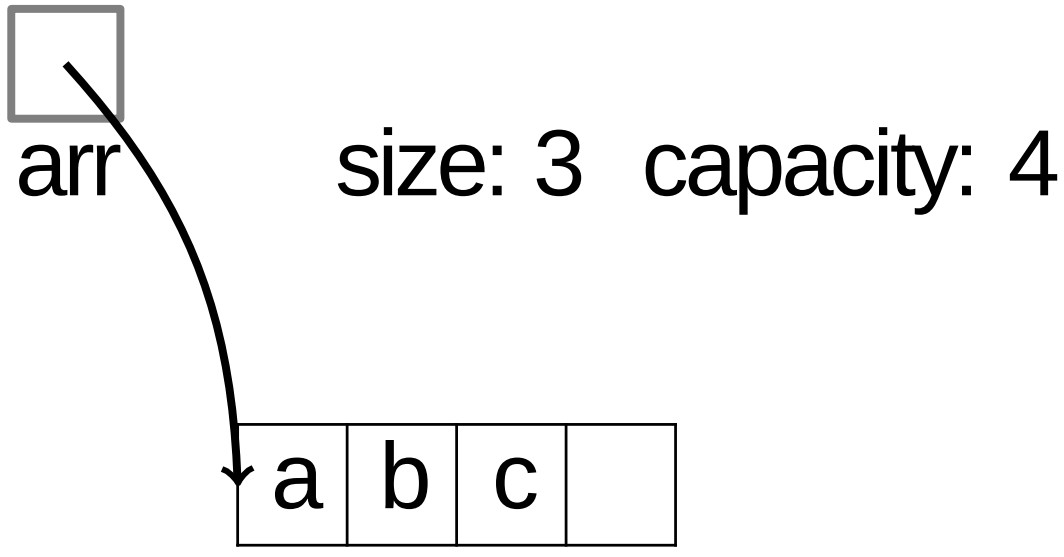
`PushBack(c)`

Dynamic Array Resizing

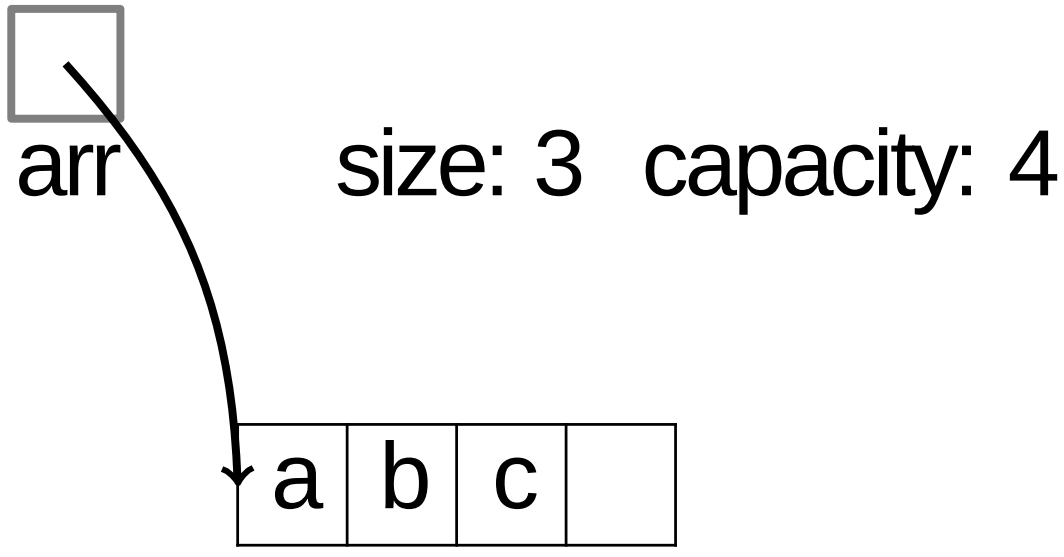


`PushBack(c)`

Dynamic Array Resizing

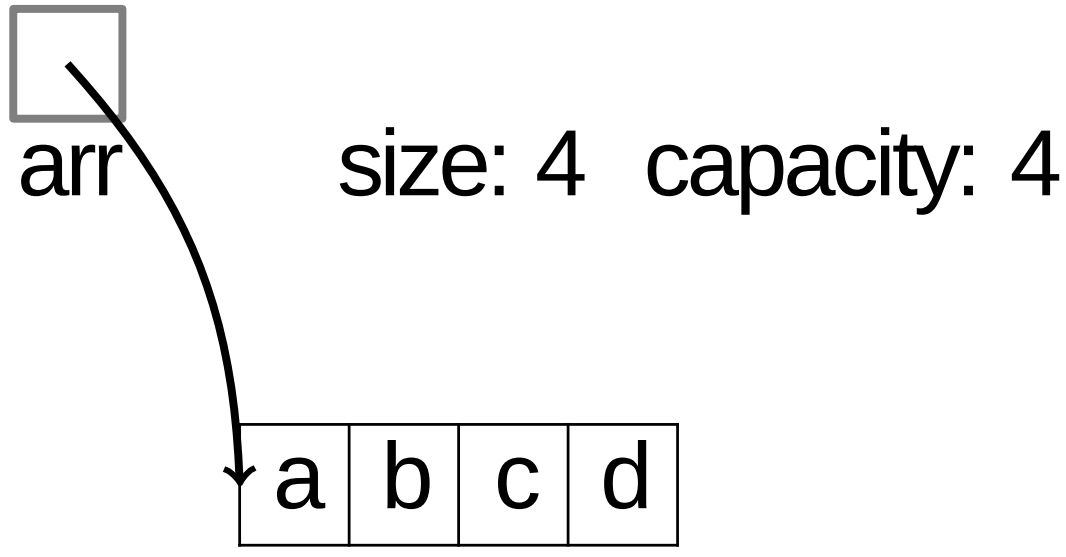


Dynamic Array Resizing



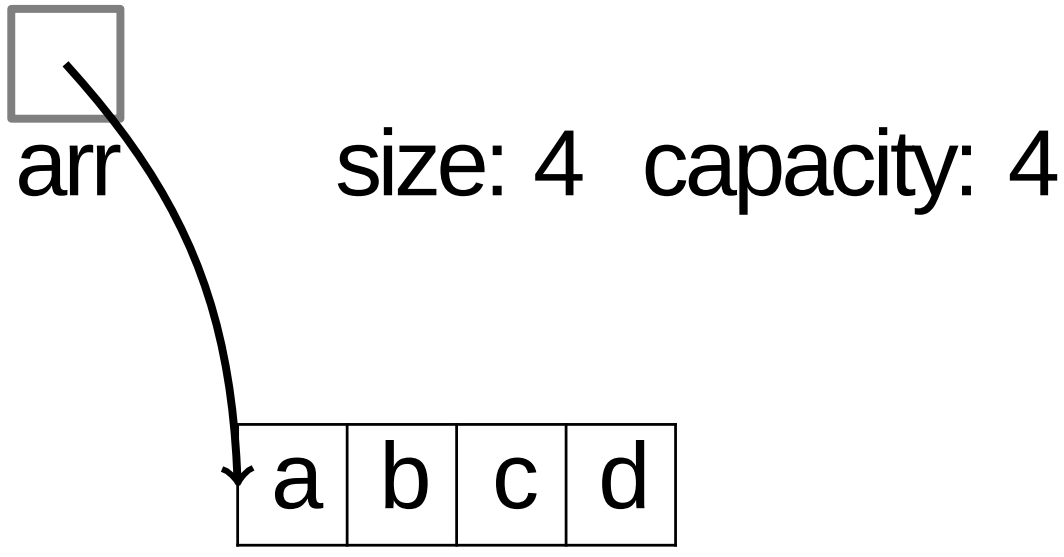
`PushBack(d)`

Dynamic Array Resizing

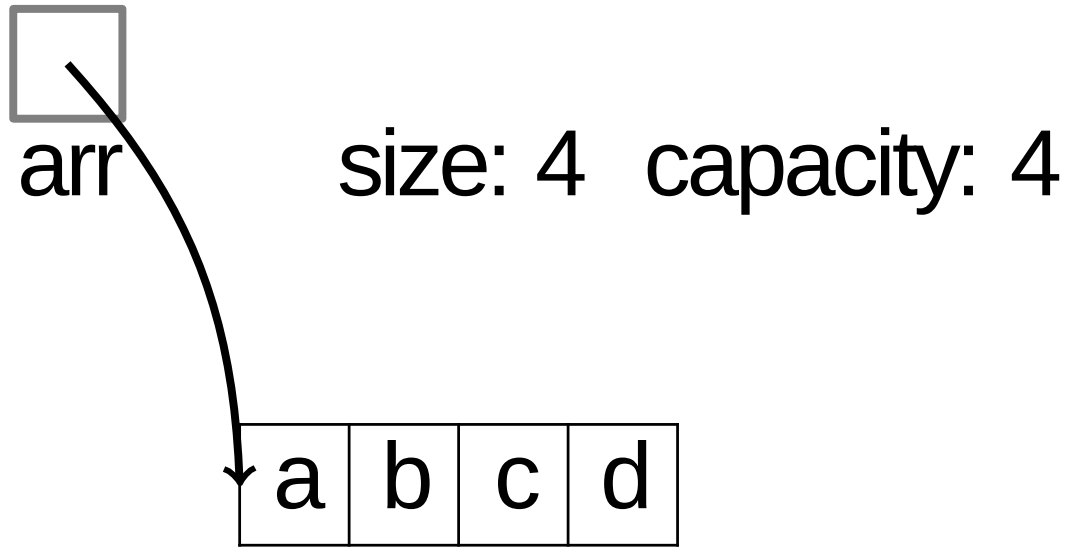


`PushBack(d)`

Dynamic Array Resizing

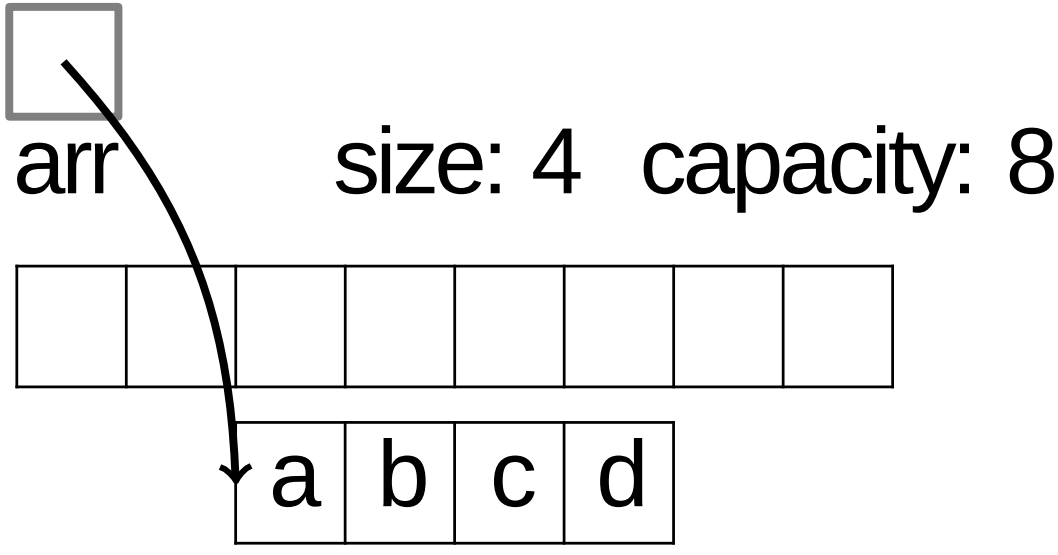


Dynamic Array Resizing



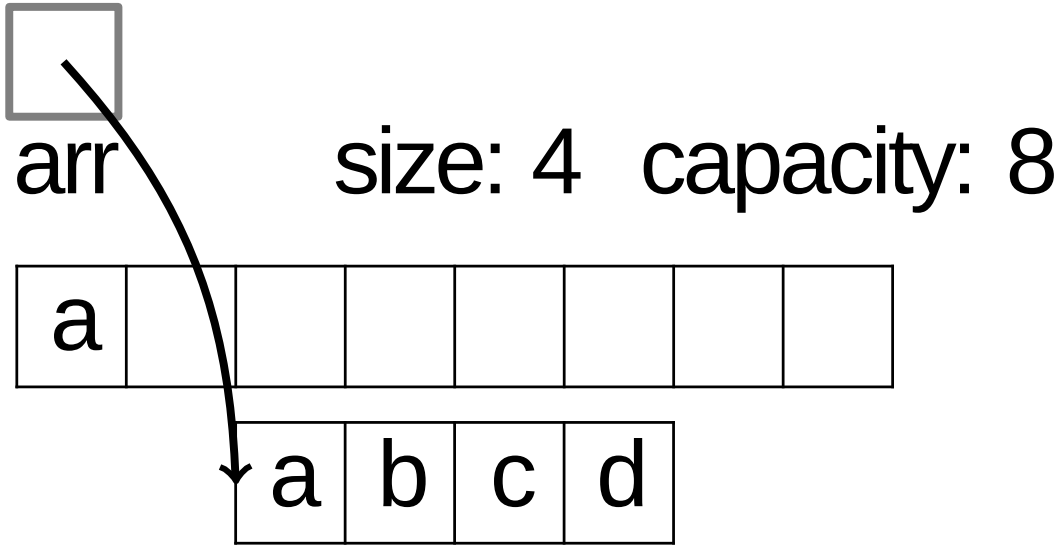
`PushBack(e)`

Dynamic Array Resizing



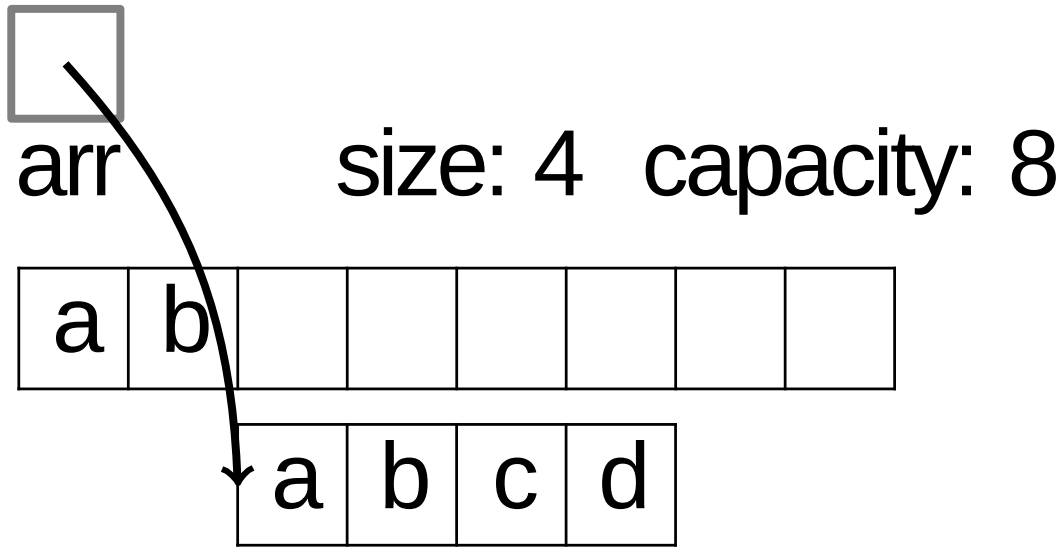
`PushBack(e)`

Dynamic Array Resizing



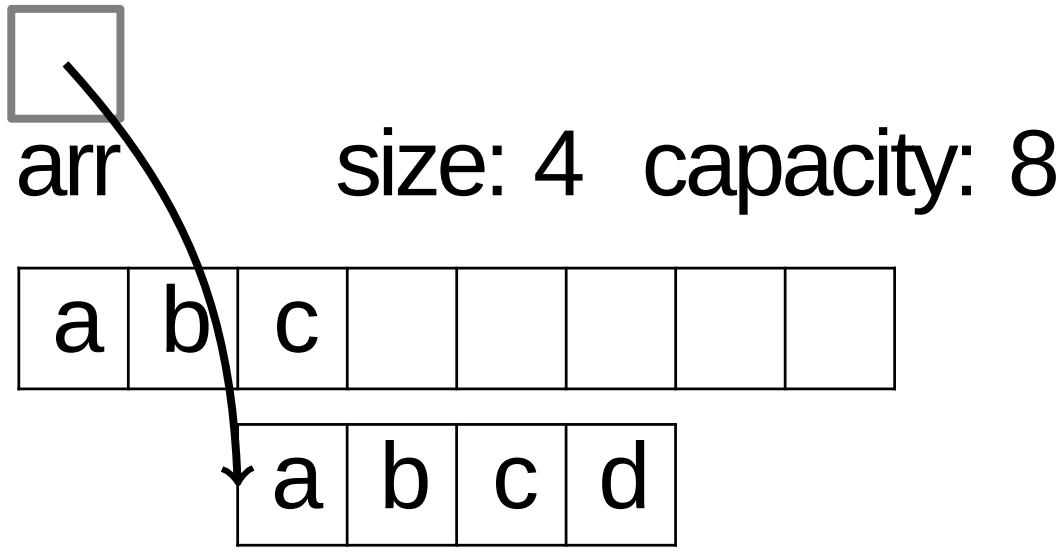
`PushBack(e)`

Dynamic Array Resizing



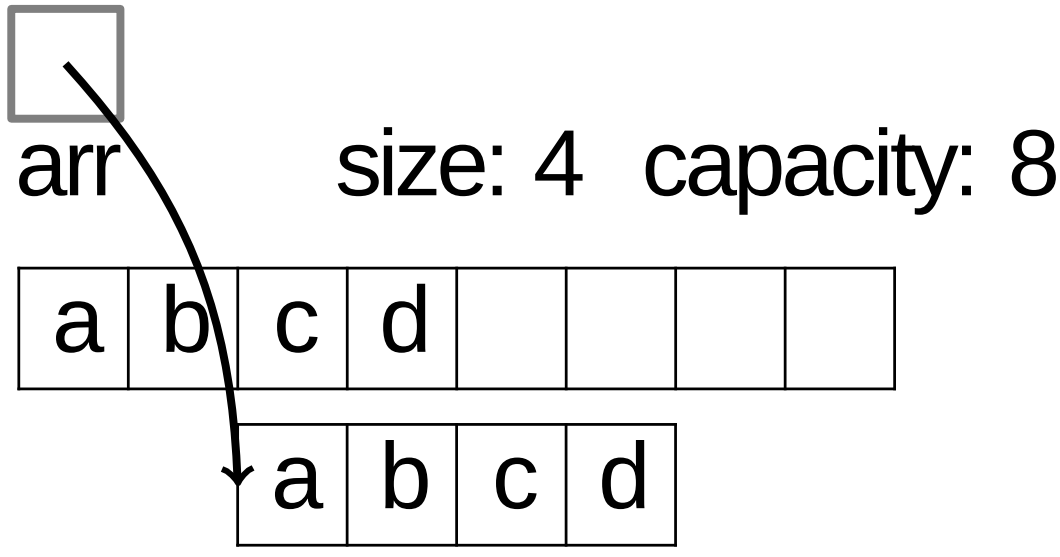
`PushBack(e)`

Dynamic Array Resizing



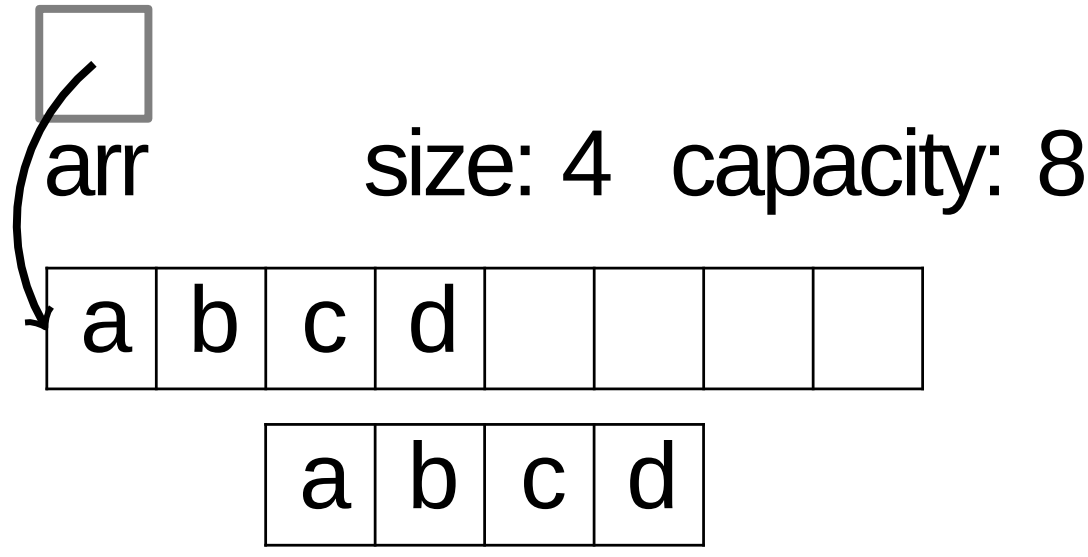
`PushBack(e)`

Dynamic Array Resizing



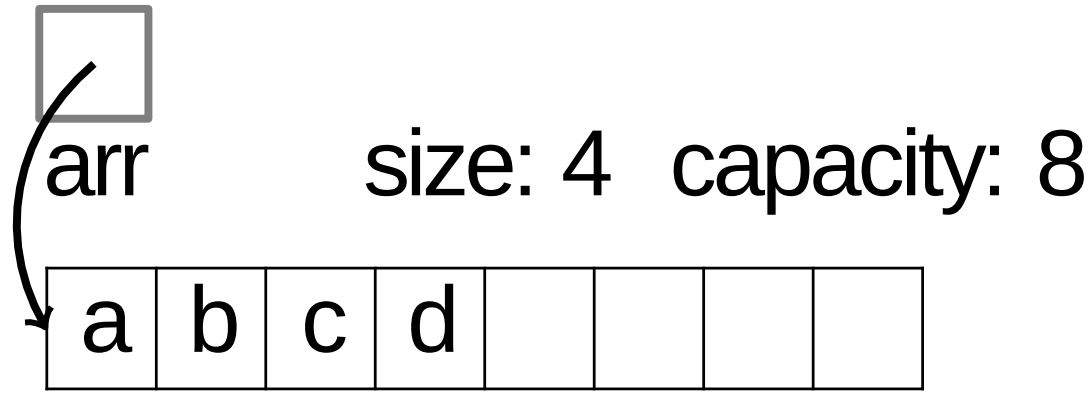
`PushBack(e)`

Dynamic Array Resizing



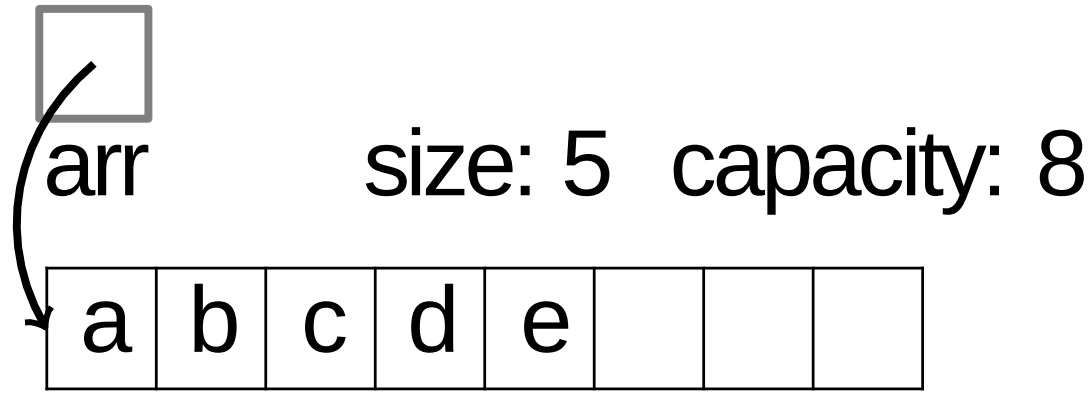
`PushBack(e)`

Dynamic Array Resizing



`PushBack(e)`

Dynamic Array Resizing



`PushBack(e)`

Get(*i*)

```
if i < 0 or i ≥ size:
   ERROR: index out of range
return arr[i]
```

Set(*i*, *val*)

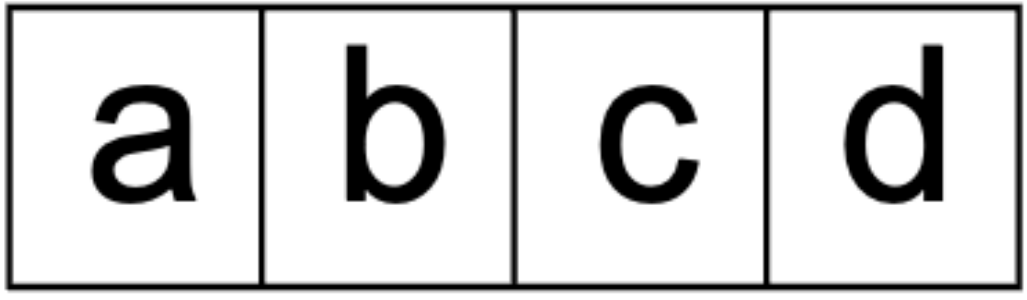
```
if i < 0 or i ≥ size:
   ERROR: index out of
   range
arr[i] = val
```

PushBack(*val*)

```
if size = capacity:
    allocate new_arr[2 × capacity]
    for i from 0 to size − 1:
        new_arr[i] ← arr[i]
    free arr
    arr ← new_arr; capacity ← 2 × capacity
arr[size] ← val
size ← size + 1
```

Remove(*i*)

```
if i < 0 or i ≥ size:
    ERROR: index out of range
for j from i to size :
    arr[j] ← arr[j + 1]
size ← size – 1
```



```
Size()

return size
```

Common Implementations

- **C++**: `vector`
- **Java**: `ArrayList`
- **Python**: `list` (the only kind of array)

Runtimes

Get(i)	$O(1)$

Runtimes

`Get`(*i*)	*O*(1)
`Set`(*i*, *val*)	*O*(1)

Runtimes

`Get`(*i*)	*O*(1)
`Set`(*i*, *val*)	*O*(1)
`PushBack`(*val*)	*O*(*n*)

Runtimes

`Get`(*i*)	*O*(1)
`Set`(*i*, *val*)	*O*(1)
`PushBack`(*val*)	*O*(*n*)
`Remove`(*i*)	*O*(*n*)

Runtimes

`Get(`*i*`)`	$O(1)$
`Set(`*i*`, `*val*`)`	$O(1)$
`PushBack(`*val*`)`	$O(n)$
`Remove(`*i*`)`	$O(n)$
`Size()`	$O(1)$

Summary

- Unlike static arrays, dynamic arrays can be resized.

Summary

- Unlike static arrays, dynamic arrays can be resized.
- Appending a new element to a dynamic array is often constant time, but can take $O(n)$.

Summary

- Unlike static arrays, dynamic arrays can be resized.
- Appending a new element to a dynamic array is often constant time, but can take $O(n)$.
- Some space is wasted

Summary

- Unlike static arrays, dynamic arrays can be resized.
- Appending a new element to a dynamic array is often constant time, but can take $O(n)$.
- Some space is wasted

Summary

- Unlike static arrays, dynamic arrays can be resized.
- Appending a new element to a dynamic array is often constant time, but can take $O(n)$.
- Some space is wasted-at most half.

Outline

1. Dynamic Arrays
2. Amortized Analysis-Aggregate Method
3. Amortized Analysis-Banker's Method
4. Amortized Analysis-Physicist's Method

Sometimes, looking at the individual worst-case may be too severe. We may want to know the total worst-case cost for a sequence of operations.

Dynamic Array

We only resize every so often.

Many $O(1)$ operations are followed by an

$O(n)$ operations.

What is the total cost of inserting many elements?

Definition

Amortized cost: Given a sequence of n operations, the amortized cost is:

$$\frac{\text{Cost}(n \text{ operations})}{n}$$

Aggregate Method

Dynamic array: n calls to `PushBack`

Aggregate Method

Dynamic array: n calls to `PushBack`

Let c_i = cost of i'th insertion.

Aggregate Method

Dynamic array: n calls to `PushBack`

Let c_i = cost of i'th insertion.

$$c_i = 1 + \begin{cases} i-1 & \text{if } i-1 \text{ is a power of 2} \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\sum_{i=1}^{n} c_i}{n} = \frac{n + \sum_{j=1}^{\lfloor \log_2(n-1) \rfloor} 2^j}{n} = \frac{O(n)}{n}$$

, $\sum_{j=1}^{\lfloor \log_2(n-1) \rfloor} 2^j = 2^{\lfloor \log_2(n-1) \rfloor + 1} - 2.$

Note that $2^{\lfloor \log_2(n-1) \rfloor + 1} = 2 \cdot 2^{\lfloor \log_2(n-1) \rfloor} \leq 2 \cdot (n-1)$, so the total sum is at most $2(n-1) - 2 = O(n)$.

Question

Which of the following is the tightest correct upper bound on the value of the sum. $\sum_{j=1}^{\lfloor \log_2(n-1) \rfloor} 2^j$? Recall that $\lfloor x \rfloor$ is the floor function of x - the largest integer that is not greater than x. Also recall that

$$2^{\log_2(x)} = x.$$

. You may want to read about the geometric series.

$O(n)$

$O(1)$

$O(\log n)$

$O(n^2)$.

Outline

1. Dynamic Arrays
2. Amortized Analysis-Aggregate Method

Alternatives to Doubling the Array Size

We could use some different growth factor (1.5, 2.5, etc.).

Could we use a constant amount?

Cannot Use Constant Amount

If we expand by 10 each time, then:

Let c_i = cost of i'th insertion.

Cannot Use Constant Amount

If we expand by 10 each time, then:

Let c_i = cost of i'th insertion.

Cannot Use Constant Amount

If we expand by 10 each time, then:

Let c_i = cost of i'th insertion.

$$c_i = 1 + \begin{cases} i-1 & \text{if } i-1 \text{ is a multiple of 10} \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{\sum_{i=1}^{n} c_i}{n} = \frac{n + \sum_{j=1}^{(n-1)/10} 10j}{n} = \frac{n + 10\sum_{j=1}^{(n-1)/10} j}{n}$$

$$= \frac{n + 10O(n^2)}{n} = \frac{O(n^2)}{n} = O(n)$$

Summary

- Calculate amortized cost of an operation in the context of a sequence of operations.

Summary

- Calculate amortized cost of an operation in the context of a sequence of operations.
- Three ways to do analysis:

Summary

- Calculate amortized cost of an operation in the context of a sequence of operations.
- Three ways to do analysis:
  - Aggregate method (brute-force sum)

Summary

- Calculate amortized cost of an operation in the context of a sequence of operations.
- Three ways to do analysis:
  - Aggregate method (brute-force sum)
  - Banker's method (tokens)

Summary

- Calculate amortized cost of an operation in the context of a sequence of operations.
- Three ways to do analysis:
  - Aggregate method (brute-force sum)
  - Banker's method (tokens)
  - Physicist's method (potential function, Φ)

Summary

- Calculate amortized cost of an operation in the context of a sequence of operations.
- Three ways to do analysis:
  - Aggregate method (brute-force sum)
  - Banker's method (tokens)
  - Physicist's method (potential function, Φ)
- Nothing changes in the code: runtime analysis only.