CS151 Intro to Data Structures

Queues

Lists

Announcements

- HW03 (Stacks & Queues) – due Friday 10/27
- Lab checkoff, deadline is when corresponding HW is due

Outline

- Queues
- Lists
- Iterators

Stack Property

First-in Last-out (FILO)

- Applications:
  - browser history (Ctrl+H)
  - Undo (Ctrl+Z)
- Applications where we don’t want FILO:
  - Queuing system
  - Cash register
  - Scheduling tasks

First-in First-out

The first item in, is the first item out

Add-to the back, remove from the front

This is a **Queue**

Inserting – `enqueue`

Removing - `dequeue`

Queue Interface

```
public interface Queue<E> {
  int size();
  boolean isEmpty();
  E first();
  void enqueue(E e);
  E dequeue();
}
```

- `null` is returned from `dequeue()` and `first()` when queue is empty

Example

Operation	Output	Q

Example

Operation	*Output*	*Q*
enqueue(5)		

Example

Operation	*Output*	*Q*
enqueue(5)	–	(5)

Example

Operation	*Output*	*Q*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)

Example

Operation	*Output*	*Q*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()		
enqueue(7)		
dequeue()		
first()		
dequeue()		
dequeue()		
isEmpty()		
enqueue(9)		
enqueue(7)		
size()		
enqueue(3)		
enqueue(5)		
dequeue()		

Skip slide

Skip slide

Example

Operation	*Output*	*Q*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	*5*	(3)
enqueue(7)	–	(3, 7)
dequeue()	*3*	(7)
first()	*7*	(7)
dequeue()	*7*	()
dequeue()	*null*	()
isEmpty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	*9*	(7, 3, 5)

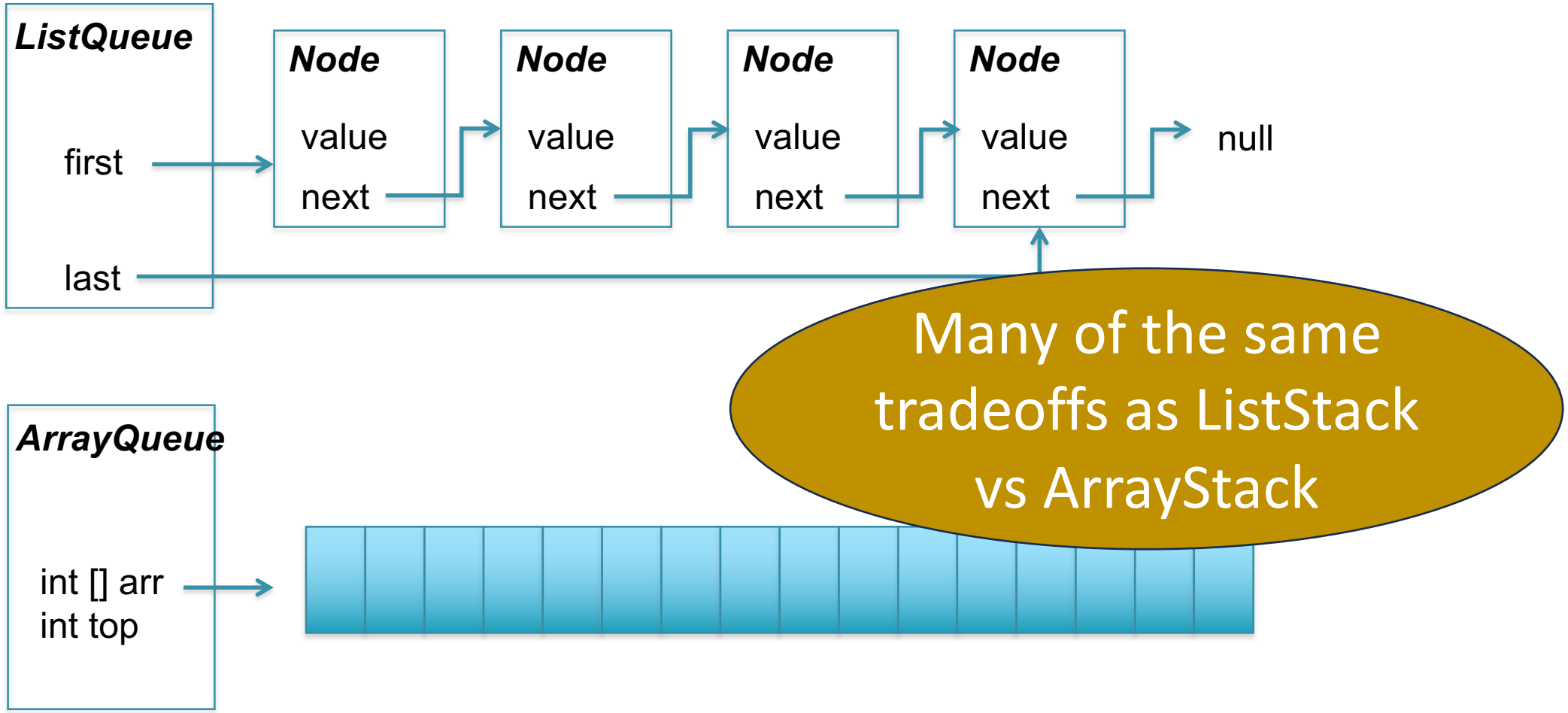
Queue Interface

```
public interface Queue<E> {
  int size();
  boolean isEmpty();
  E first();
  void enqueue(E e);
  E dequeue();
```

How would you implement this interface?

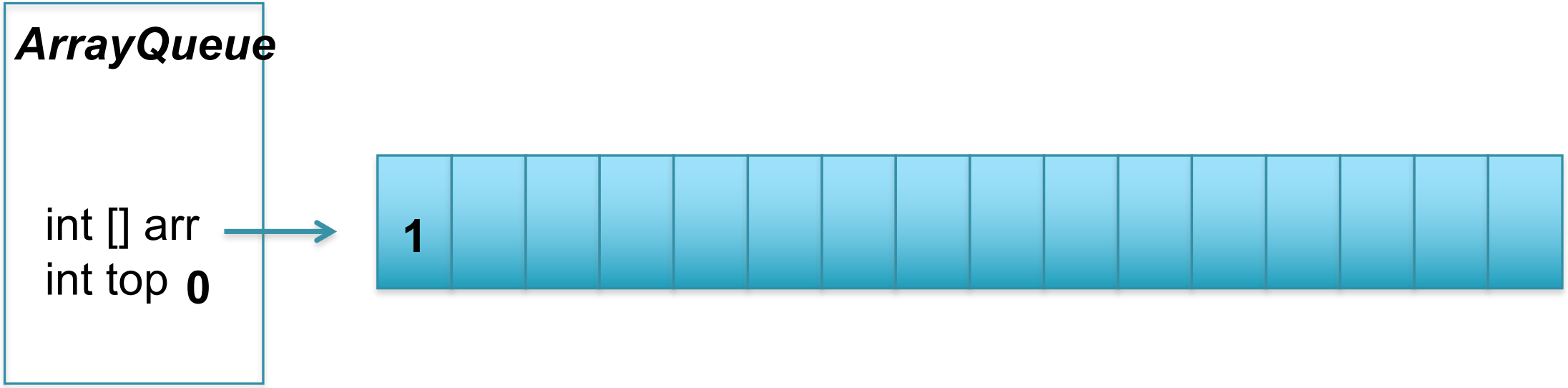
Why?

ListQueue vs ArrayQueue



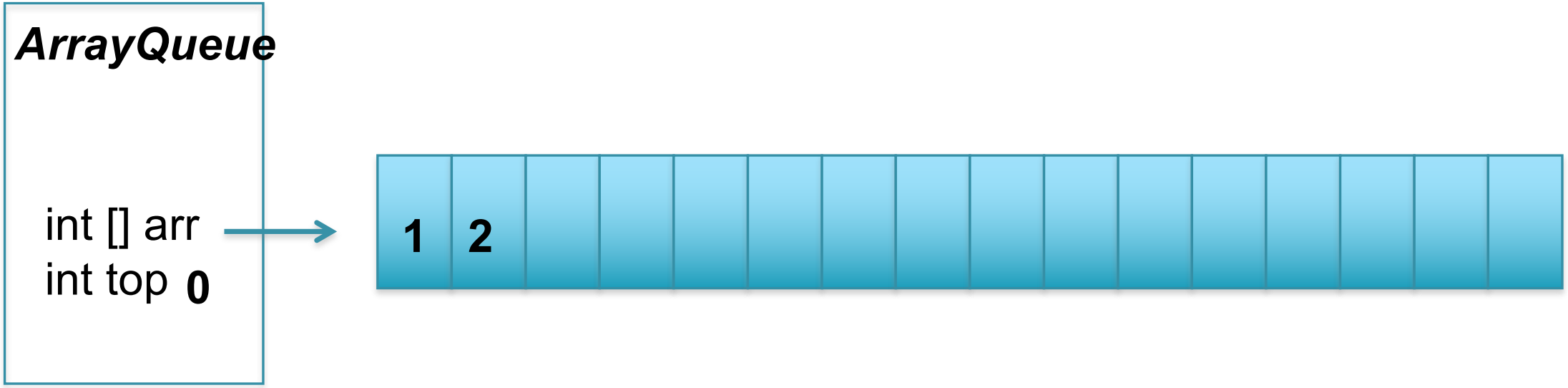
ArrayQueue

queue.enqueue(1);



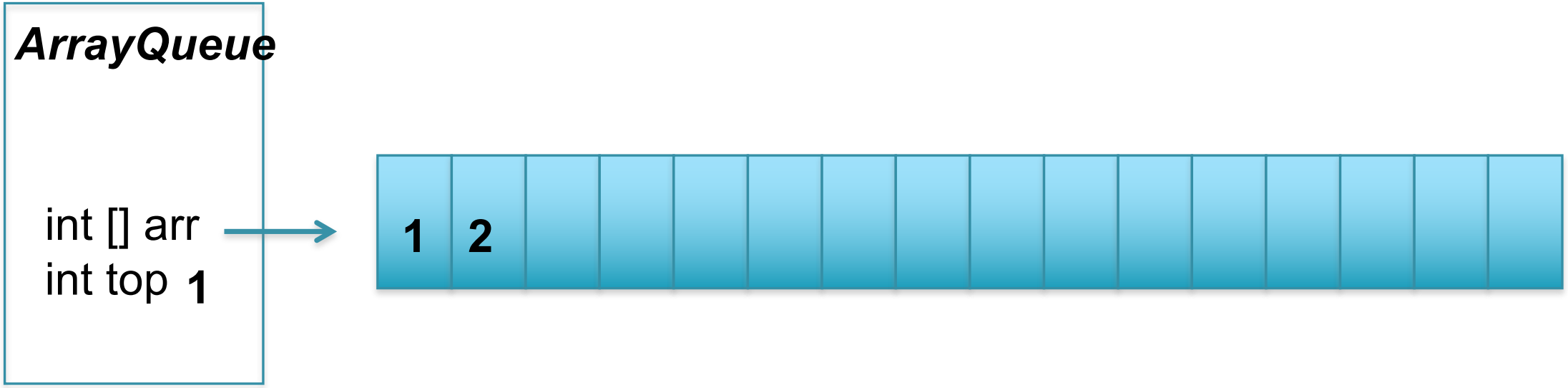
ArrayQueue

```
queue.enqueue(1);
queue.enqueue(2);
```



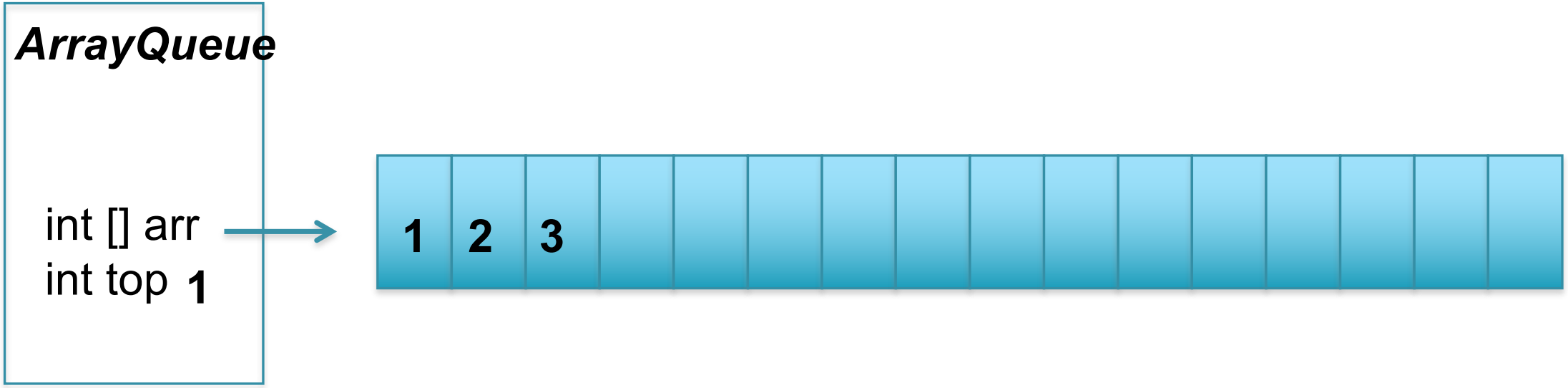
ArrayQueue

```
queue.enqueue(1);
queue.enqueue(2);
```



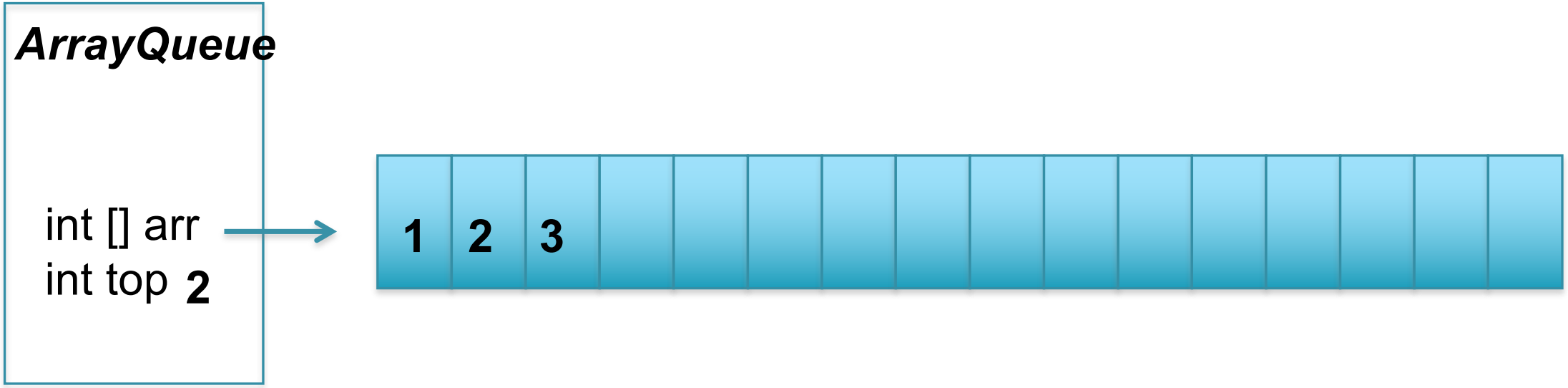
ArrayQueue

```
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);
```



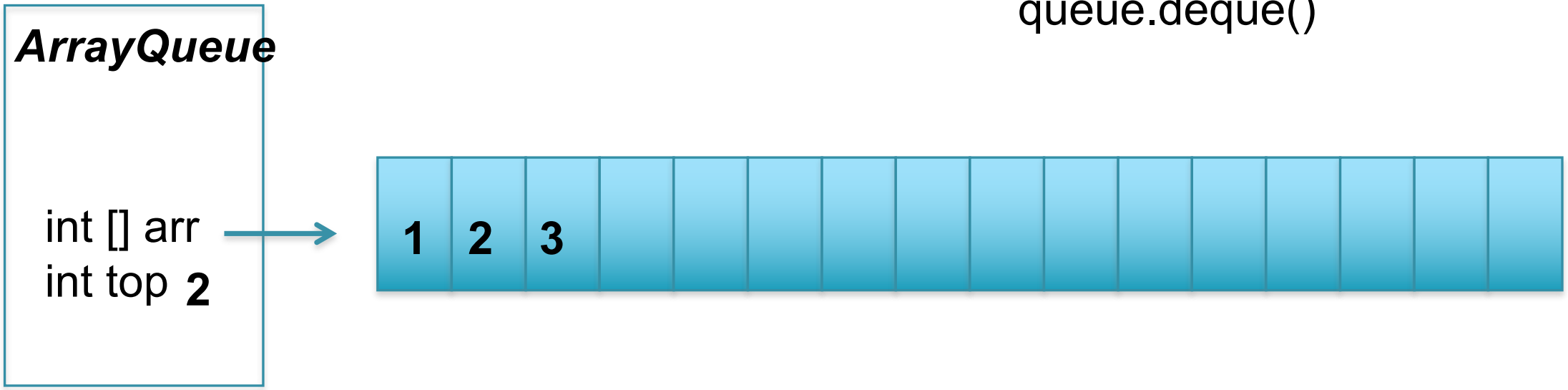
ArrayQueue

```
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);
```



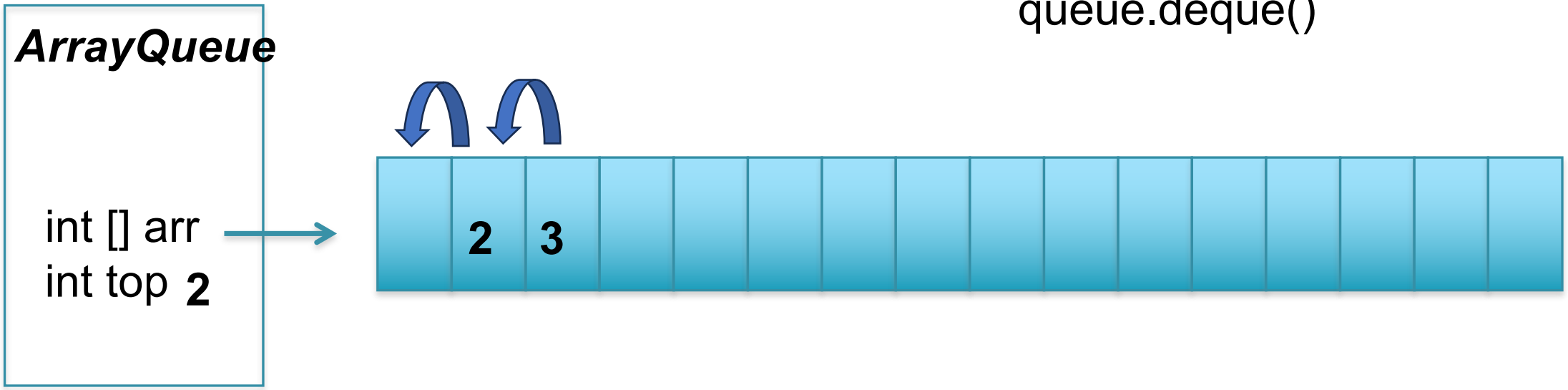
ArrayQueue

```
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);

queue.deque()
```



ArrayQueue

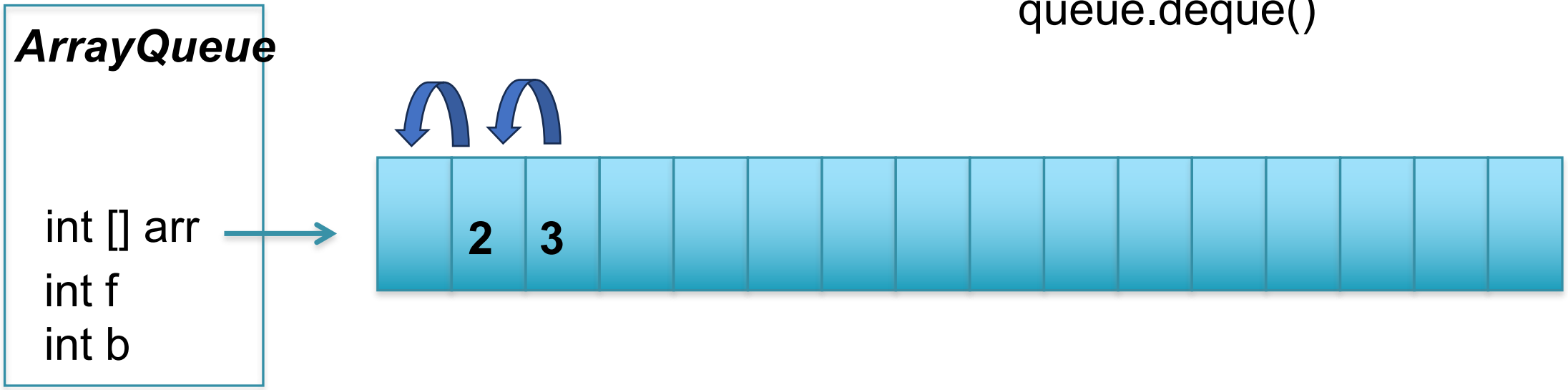
```
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);

queue.deque()
```



Whats wrong with copying?
How could we fix it?

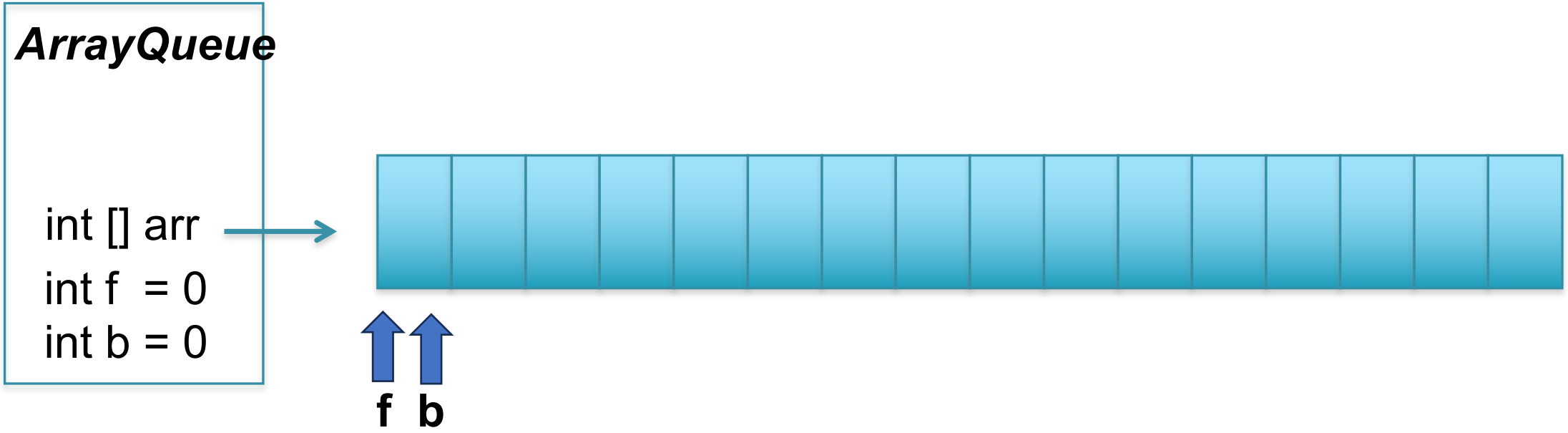
ArrayQueue

```
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);

queue.deque()
```



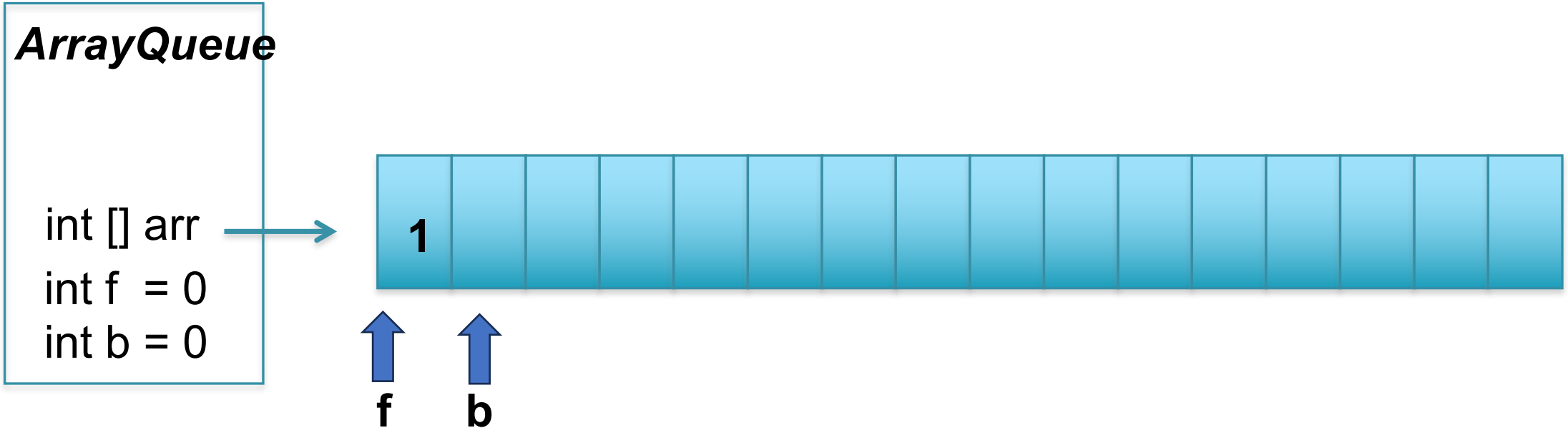
Whats wrong with copying?
How could we fix it?

ArrayQueue



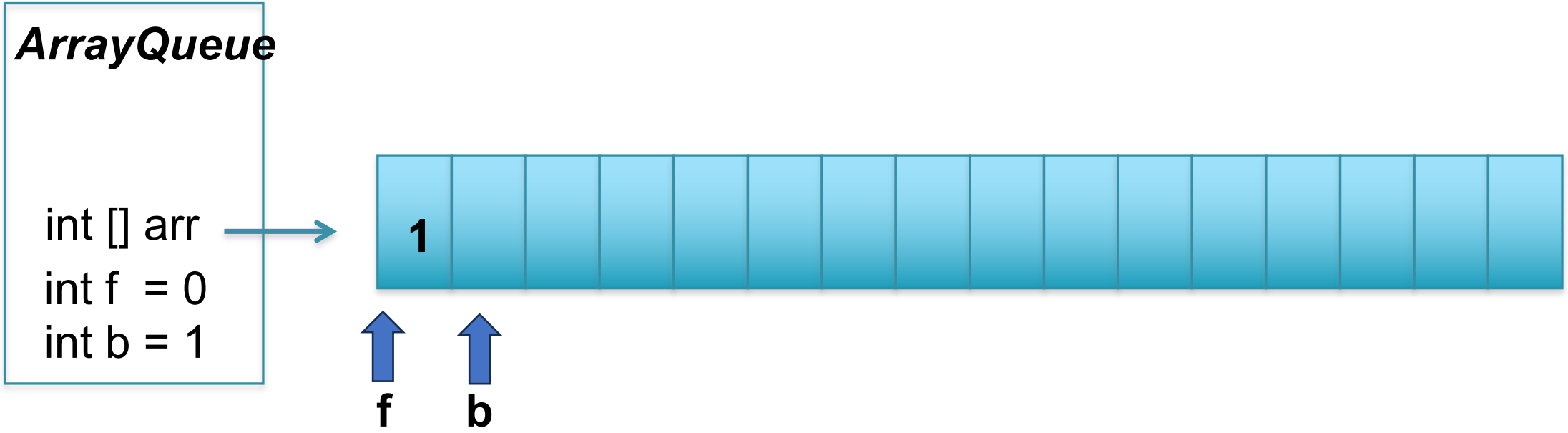
ArrayQueue

queue.enqueue(1);



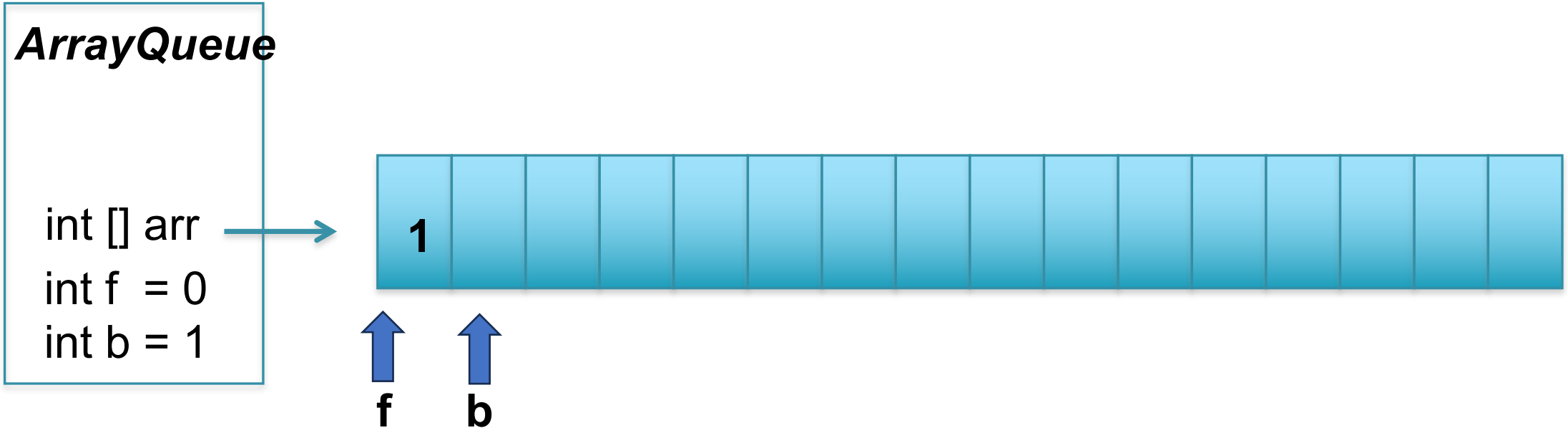
ArrayQueue

queue.enqueue(1);



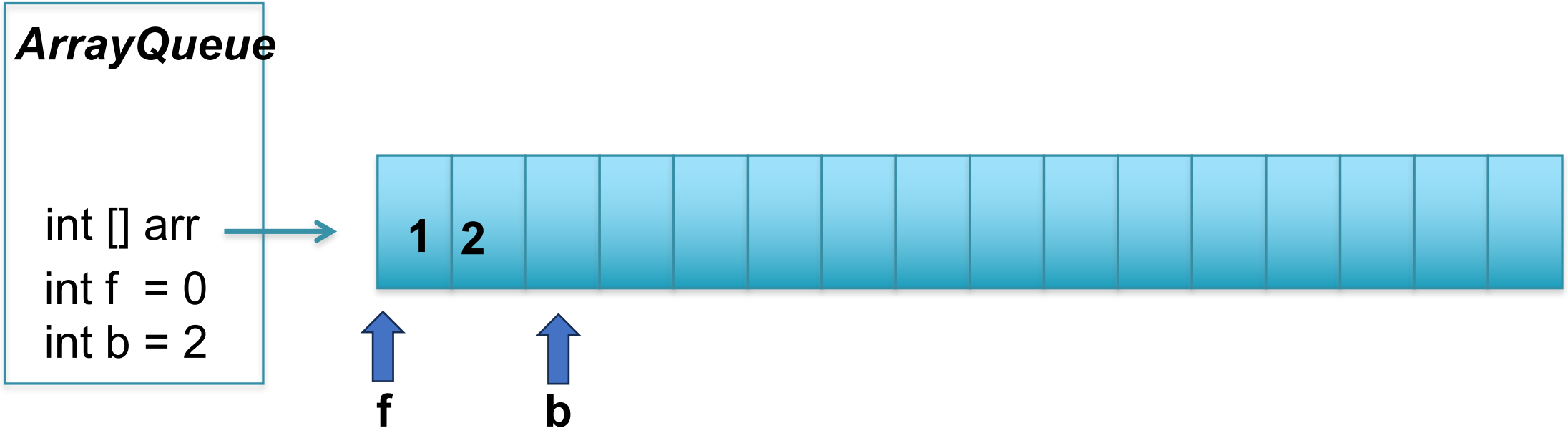
ArrayQueue

```
queue.enqueue(1);
queue.enqueue(2);
```



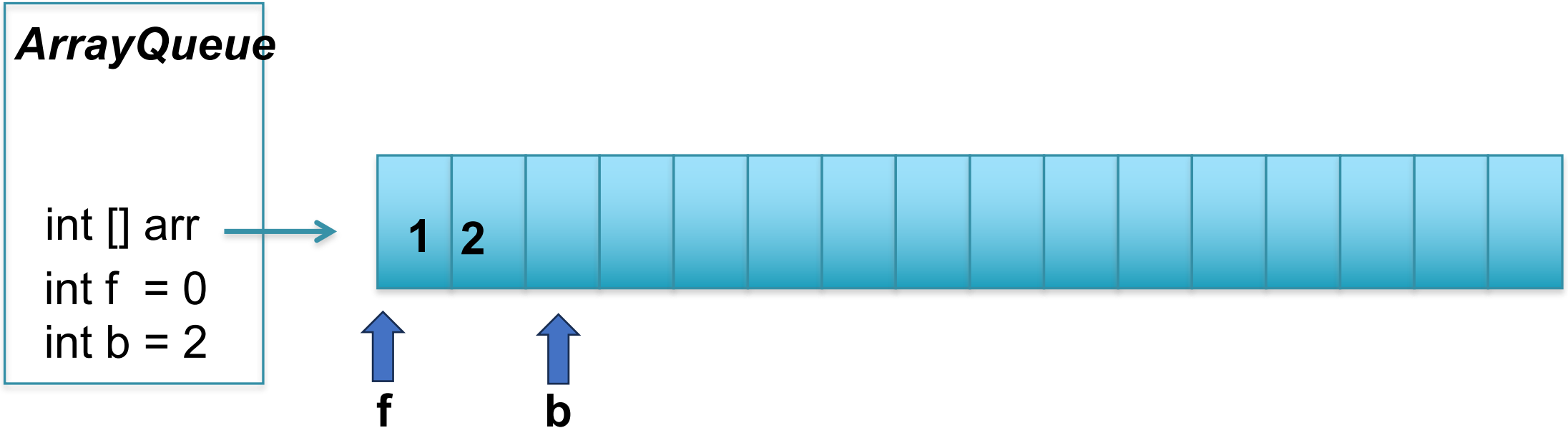
ArrayQueue

```
queue.enqueue(1);
queue.enqueue(2);
```



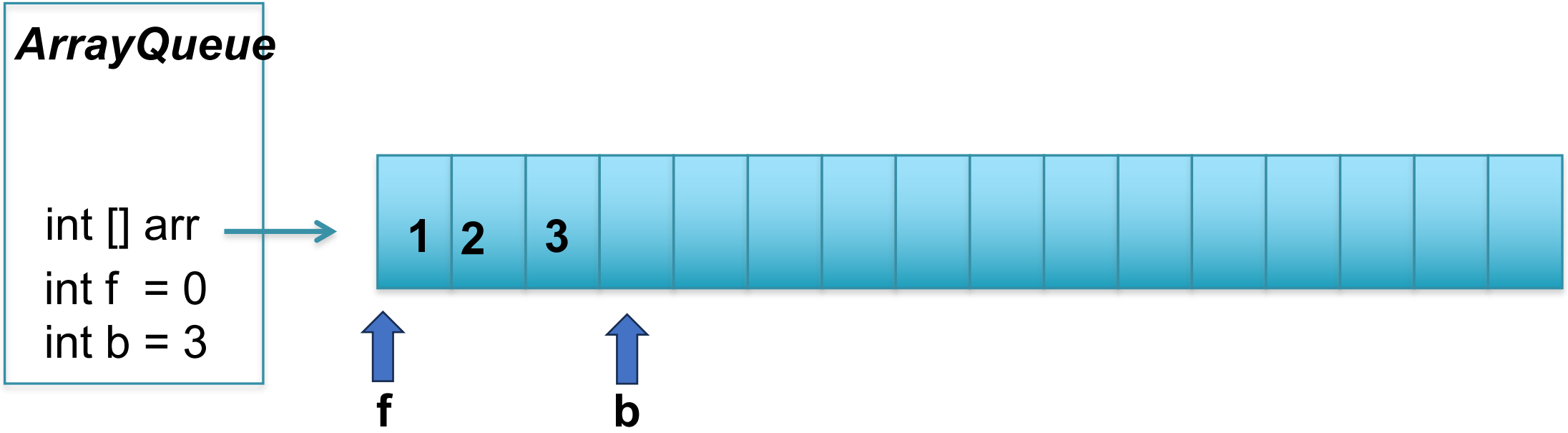
ArrayQueue

```
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);
```



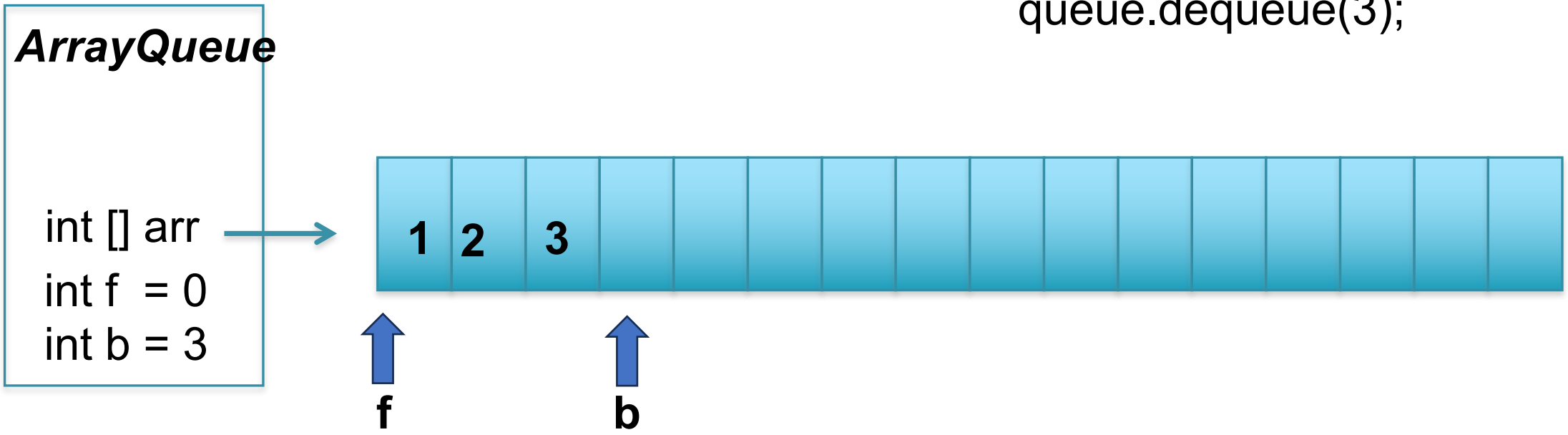
ArrayQueue

```
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);
```



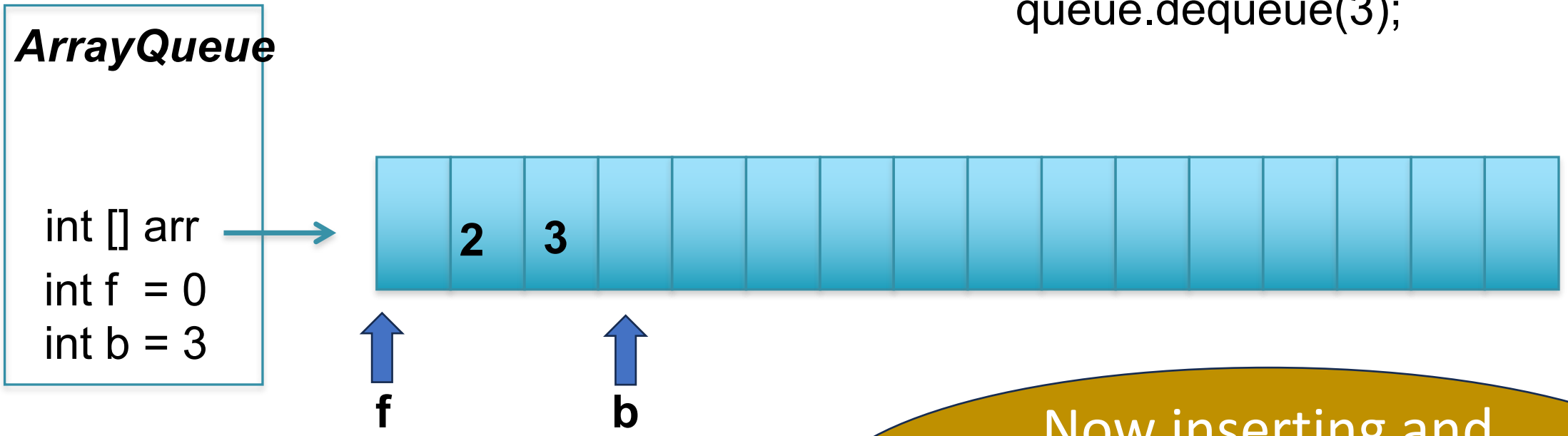
ArrayQueue

```
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);

queue.dequeue(3);
```



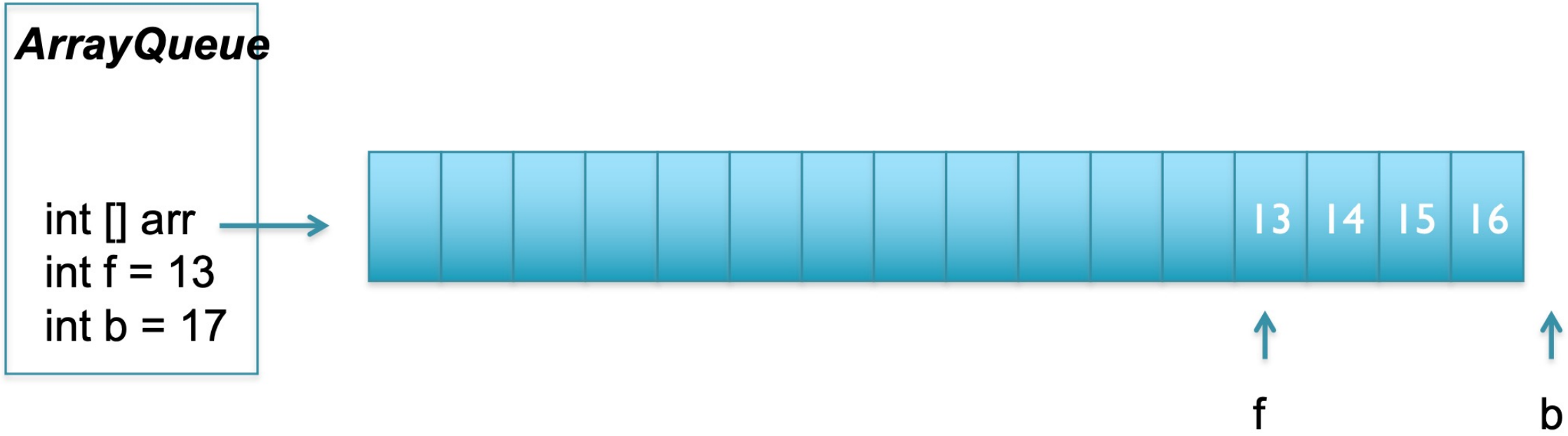
ArrayQueue

```
queue.enqueue(1);
queue.enqueue(2);
queue.enqueue(3);

queue.dequeue(3);
```



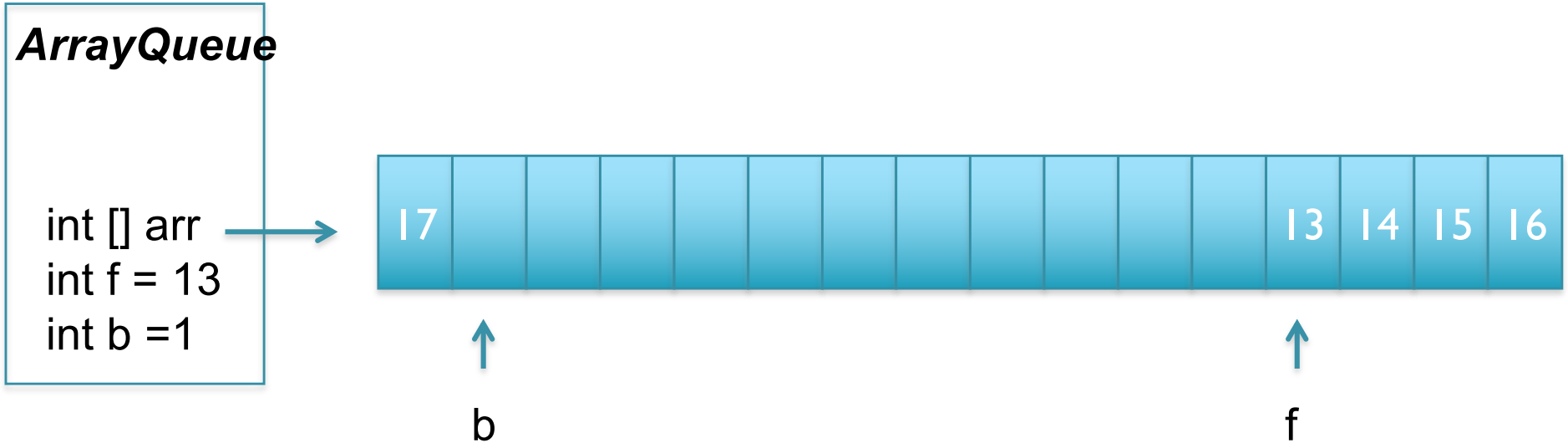
Now inserting and removing are both O(1) :)

ArrayQueue



What should we do if we insert?

ArrayQueue



queue.enqueue(17);

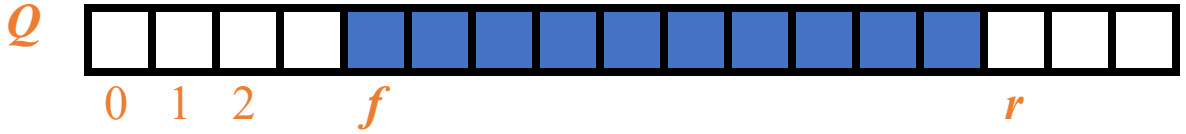
Circular Queue

When the queue has fewer than `n` elements, location `r`

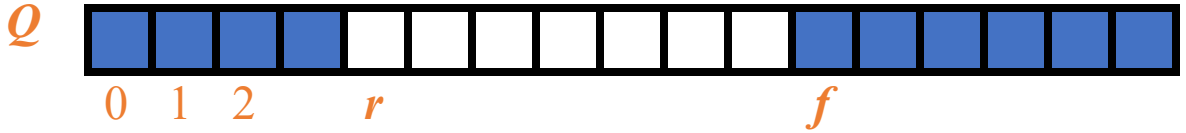
`back = back%n` is the first empty slot past the rear of the queue

Here `n` = length of the array

normal configuration



wrapped-around configuration



Performance and Limitations

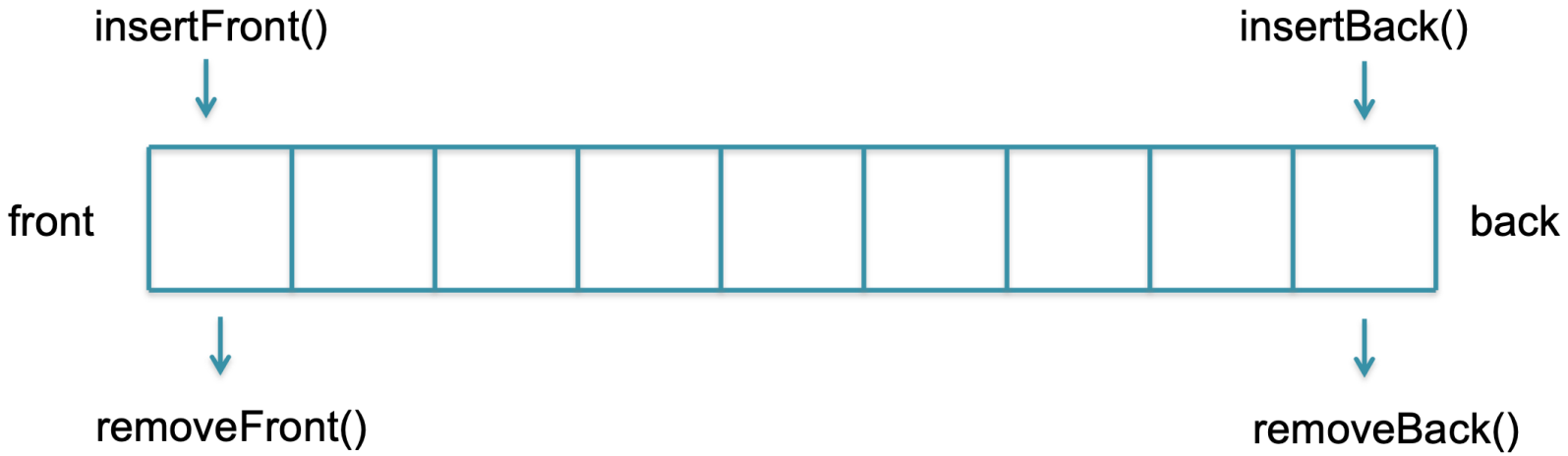
- Performance
  - let n be the number of objects in the queue
  - The space used is $O(n)$
  - Each operation runs in time $O(1)$

- Limitations
  - Depending on the implementation,
    - max size is limited and can not be changed
    - Or need to grow the array when out of room

Comparison to `java.util.Queue`

Our Queue ADT	Interface java.util.Queue	
	throws exceptions	returns special value
enqueue(e)	add(e)	offer(e)
dequeue()	remove()	poll()
first()	element()	peek()
size()	size()	
isEmpty()	isEmpty()	

Doubled-ended Queue (aka Deques aka Decks)



Dynamic Data Structure used for storing sequences of data

- Insert/Remove at either end in O(1)
- If you exclusively add/remove at one end, then ***it becomes a stack***
- If you exclusive add to one end and remove from other, then ***it becomes a queue***
- Many other applications:
  - browser history: deque of last 100 webpages visited

Outline

- Queues
- **Lists**
- Iterators

`java.util.List` ADT

size( ): Returns the number of elements in the list.

isEmpty( ): Returns a boolean indicating whether the list is empty.

get(i): Returns the element of the list having index i; an error condition occurs if i is not in range $[0, \text{size}() - 1]$.

set(i, e): Replaces the element at index i with e, and returns the old element that was replaced; an error condition occurs if i is not in range $[0, \text{size}() - 1]$.

add(i, e): Inserts a new element e into the list so that it has index i, moving all subsequent elements one index later in the list; an error condition occurs if i is not in range $[0, \text{size}()]$.

remove(i): Removes and returns the element at index i, moving all subsequent elements one index earlier in the list; an error condition occurs if i is not in range $[0, \text{size}() - 1]$.

Example

Method	Return Value	List Contents
add(0, A)		

Example

Method	Return Value	List Contents
add(0, A)	–	(A)

Example

Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)		

Example

Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)	–	(B, A)

Example

Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)	–	(B, A)
get(1)		
set(2, C)		
add(2, C)		
add(4, D)		
remove(1)		
add(1, D)		
add(1, E)		
get(4)		
add(4, F)		
set(2, G)		
get(2)		

Example

Method	Return Value	List Contents
add(0, A)	–	(A)
add(0, B)	–	(B, A)
get(1)	A	(B, A)
set(2, C)	"error"	(B, A)
add(2, C)	–	(B, A, C)
add(4, D)	"error"	(B, A, C)
remove(1)	A	(B, C)
add(1, D)	–	(B, D, C)
add(1, E)	–	(B, E, D, C)
get(4)	"error"	(B, E, D, C)
add(4, F)	–	(B, E, D, C, F)
set(2, G)	D	(B, E, G, C, F)
get(2)	G	(B, E, G, C, F)

ArrayList

- An array-based implementation of `List`
  - $O(n)$ space
  - indexing/random access is $O(1)$
  - add/remove is $O(n)$

Iterators

- Abstracts the process of scanning through a sequence of elements (traversal)

hasNext( ): Returns true if there is at least one additional element in the sequence, and false otherwise.

next( ): Returns the next element in the sequence.

traversal structure

```
while(iter.hasNext()) {
  iter.next();
}
```

`Iterable` Interface

An interface with a single method:

- `iterator()`: returns an iterator of the elements in the collection

Each call to `iterator()` returns a new iterator instance, thereby allowing traversals of a collection

`List` interface extends `Iterable`

`ArrayList` implements `List`

`Iterator` Interface

Another interface that supports iteration

- `boolean hasNext()`
- `E next()`
- `void remove()`

Example

- `Scanner` implements `Iterator<String>`
- `ArrayList` inner class `ArrayListIterator` implements `Iterator`

`Iterable` versus `Iterator`

- `Iterable`
  - `java.lang`
  - override `iterator()`
  - Doesn't store the iteration state
  - Removing elements during iteration isn't allowed
- `Iterator`
  - `java.util`
  - Override `hasNext()`, `next()`
  - Optional `remove()`
  - Stores iteration state (list cursor)
  - Removing elements during iteration supported

Iterations with ArrayList

```
public class ArrayList<E> implements List<E> {
  //instance vars, constructors and other methods not shown
  private class ArrayIterator implements Iterator<E> {
    private int j=0; //index of next element
    private boolean removable = false;
    public boolean hasNext() {return j<sz;}
    public E next() throws NoSuchElementException {
      if (j==sz) throw new NoSuchElementException();
      removable = true; // allow removal of this element
      return L[j++];
    }
    public void remove() throws IllegalStateException {
      if (!removable) throw new IllegalStateException();
      ArrayList.this.remove(j-1); j--; removable = false;
    }
    public Iterator<E> iterator() {
      return new ArrayIterator(); //create new instance
    }
  }
}
```