# Mini-Project 2: ArrayDeque61B

# Why use a backing array?

# Why use a backing array?

In Mini-Project 2, we'll be building another Deque! This time, the goal is to build a Deque with a circular backing array rather than a backing doubly linked list (with sentinel).

As we know, Deques should be able to handle any (nonnegative) number of elements, and have the ability to `addFirst`, `addLast`, `removeFirst`, and `removeLast`, among some other operations.

This implies that the backing data structure for our Deque should be able to dynamically size up and down.

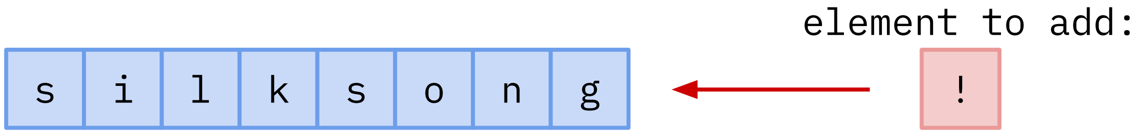
Discussion Question: what properties of arrays seem to make them bad for implementing a data structure like a Deque?

# Issue #1: Fixed Size

Arrays have a fixed size! Our Deque needs to have a dynamic size!

There is no way to resize an existing array.

Discussion Question: Can we come up with a workaround? What should we do if our initial backing array is at max capacity and we need to add another element?



element to add:

| s | i | l | k | s | o | n | g |
|---|---|---|---|---|---|---|---|

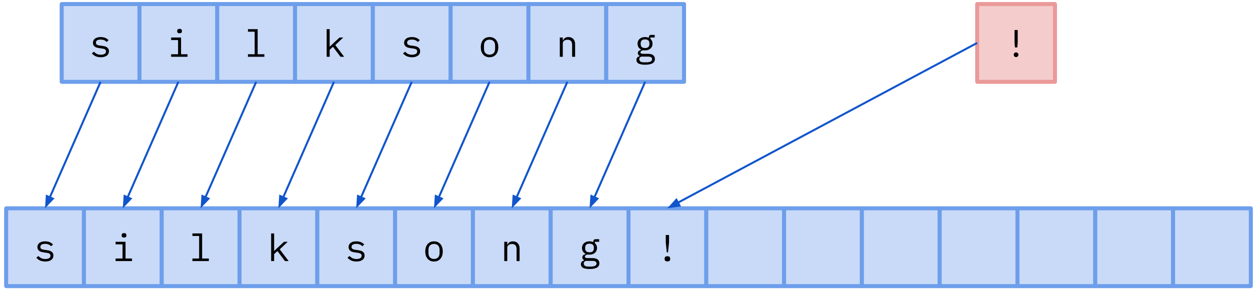
!

# Issue #1: Fixed Size

Arrays have a fixed size! Our Deque needs to have a dynamic size!

There is no way to resize an existing array.

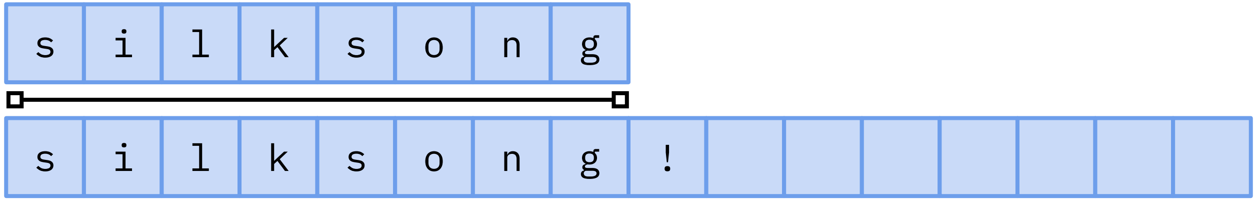
Discussion Question: Can we come up with a workaround? What should we do if our initial backing array is at max capacity and we need to add another element?



A good approach is to create a new, larger backing array, and copy the old elements in!

# Concept: Resizing

We want to resize up by a constant factor. In this case, our original array was of size 8 and our new array is of size 16, so we can say that it was scaled with a scaling factor of 2.

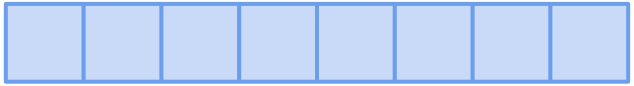


| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| s | i | l | k | s | o | n | g | | | | | | | | |
| s | i | l | k | s | o | n | g | ! | | | | | | | |

# There is still work to be done!

Okay, great! We can hold as many elements as we want now. Let’s try doing some Deque operations with a backing array!

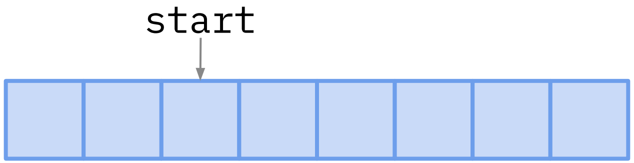
We’ll pick an arbitrary start position and call `addFirst` a few times.



# There is still work to be done!

Okay, great! We can hold as many elements as we want now. Let's try doing some Deque operations with a backing array!

We'll pick an arbitrary start position and call `addFirst` a few times.

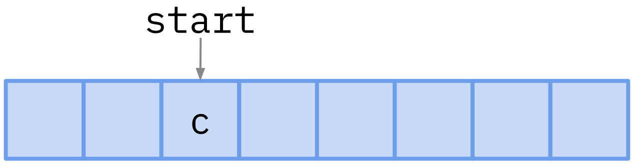


```
addFirst("c");
addFirst("s");
addFirst("6");
addFirst("1");
addFirst("b");
```

# There is still work to be done!

Okay, great! We can hold as many elements as we want now. Let’s try doing some Deque operations with a backing array!

We’ll pick an arbitrary start position and call `addFirst` a few times.

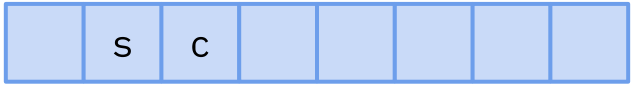


```
addFirst(“c”);
addFirst(“s”);
addFirst(“6”);
addFirst(“1”);
addFirst(“b”);
```

# There is still work to be done!

Okay, great! We can hold as many elements as we want now. Let's try doing some Deque operations with a backing array!

We'll pick an arbitrary start position and call `addFirst` a few times.



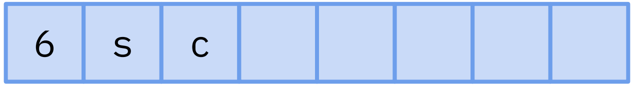
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | s | c | | | | | |

```
addFirst("c");
addFirst("s");
addFirst("6");
addFirst("1");
addFirst("b");
```

# There is still work to be done!

Okay, great! We can hold as many elements as we want now. Let’s try doing some Deque operations with a backing array!

We’ll pick an arbitrary start position and call `addFirst` a few times.



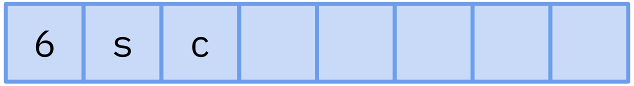
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | s | c | | | | | |

```
addFirst(“c”);
addFirst(“s”);
addFirst(“6”);
addFirst(“1”);
addFirst(“b”);
```

# There is still work to be done!

Okay, wait…

Where do we go now? If we imagine the array as a linear block of buckets, the answer is nowhere.
…but that’s not the only way to imagine the array!

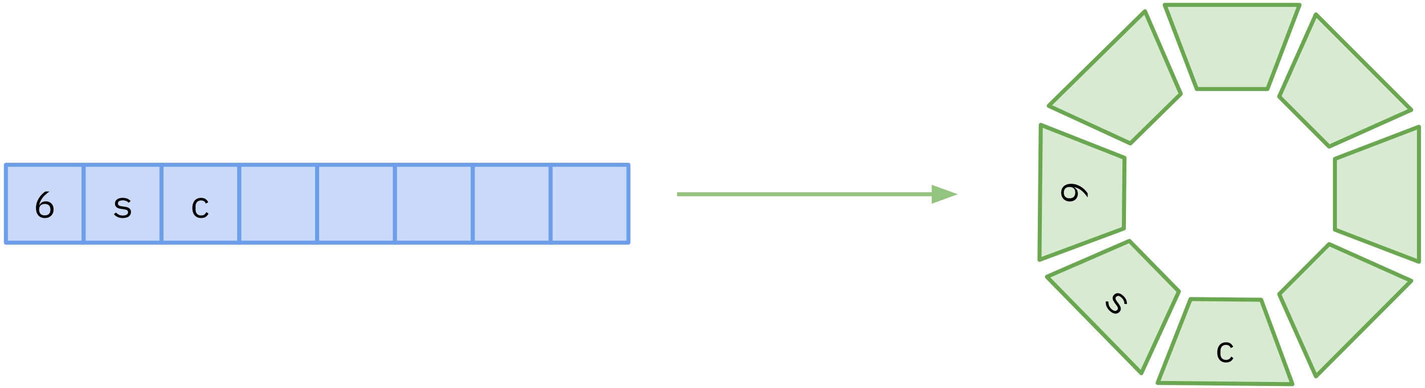


|   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|
| 6 | s | c |   |   |   |   |   |

```
addFirst(“c”);
addFirst(“s”);
addFirst(“6”);
addFirst(“1”);
addFirst(“b”);
```

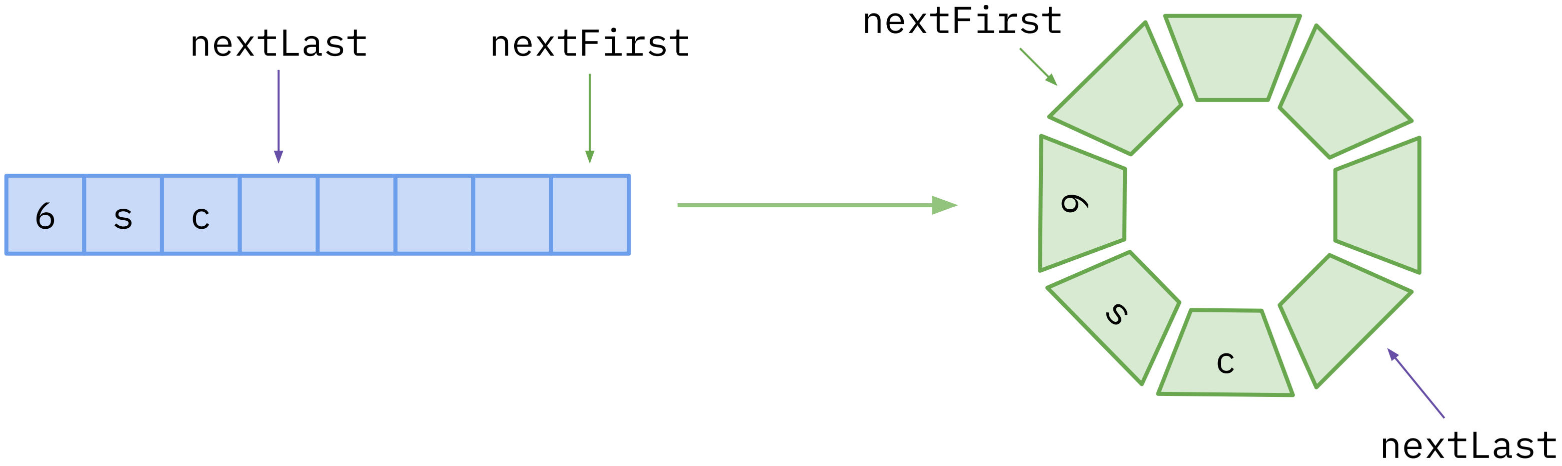
# There is still work to be done!

If we imagine the array as a big circle, then it’s pretty clear where we should go next!



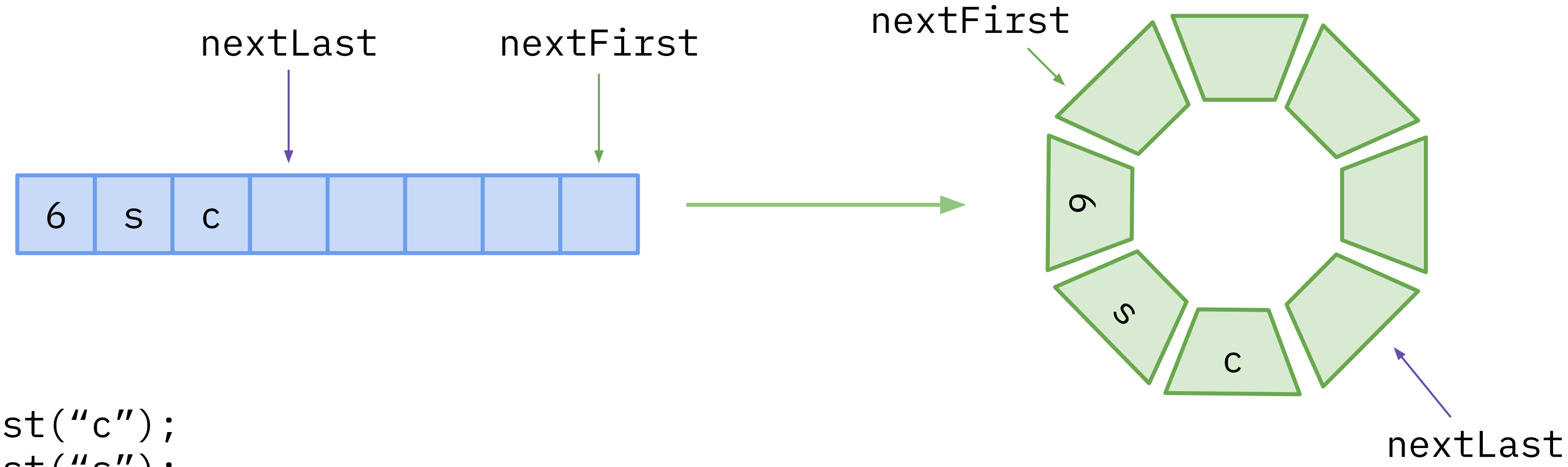
# There is still work to be done!

We can keep pointers (let's call them nextFirst and nextLast) to tell us where our next additions should be made!



# There is still work to be done!

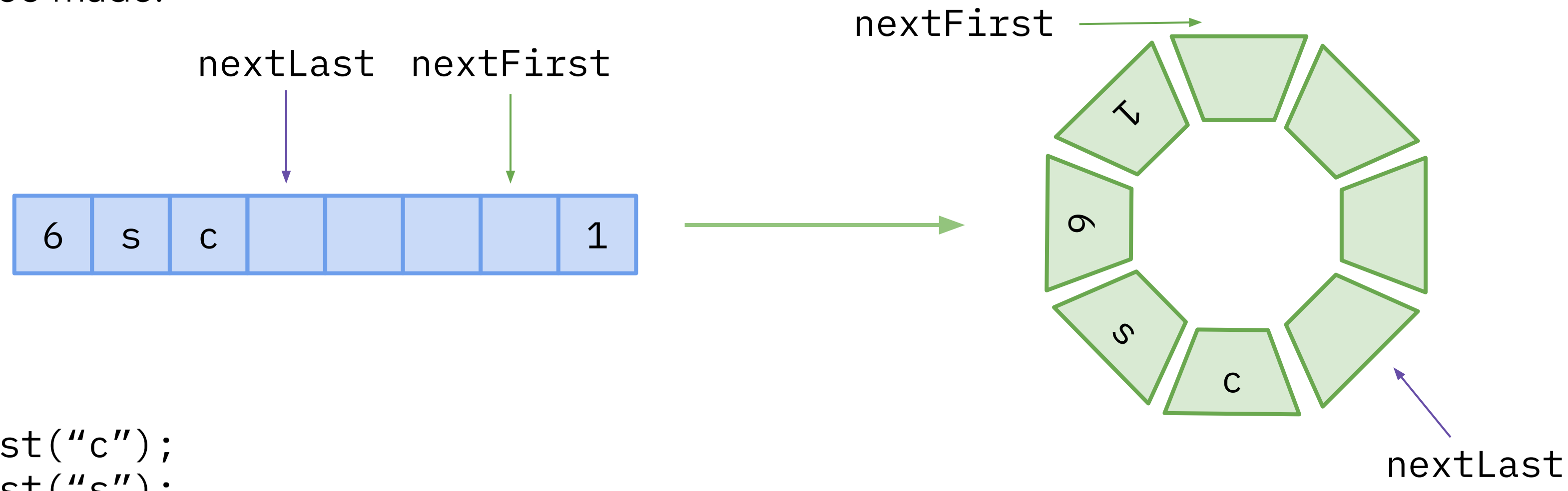
We can keep pointers (let’s call them nextFirst and nextLast) to tell us where our next additions should be made!



```
addFirst(“c”);
addFirst(“s”);
addFirst(“6”);
addFirst(“1”);
addFirst(“b”);
```

# There is still work to be done!

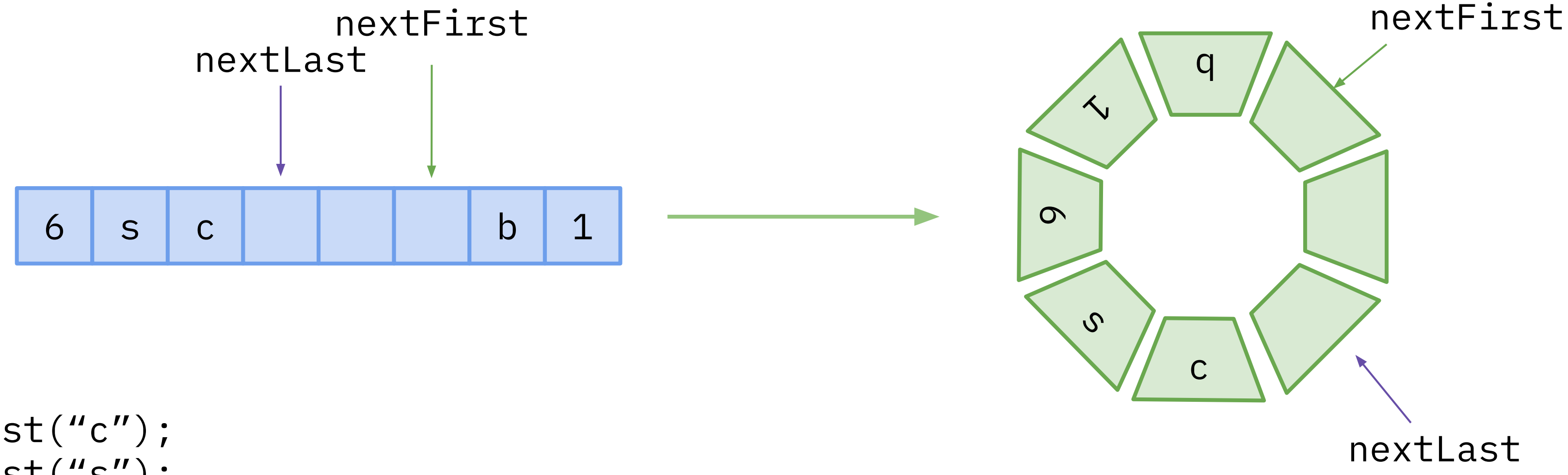
We can keep pointers (let's call them nextFirst and nextLast) to tell us where our next additions should be made!



```
addFirst("c");
addFirst("s");
addFirst("6");
addFirst("1");
addFirst("b");
```

# There is still work to be done!

We've managed to add all 5 items.



```
addFirst("c");
addFirst("s");
addFirst("6");
addFirst("1");
addFirst("b");
```

# Array Solutions

Combining the ideas of resizing and a circular array structure creates an architecture that lends itself very well to building a Deque.

- Keep `nextFirst` and `nextLast` pointers that move around the array in a circle.
- Resize the array up when the backing array is full.
  - Update the `nextFirst` and `nextLast` pointers to match the new start and end positions after a resize!

You primary task for Mini-Project 2 is implementing this behavior!

# Get

# Get

One of the methods in the Deque interface is `get`, which takes in an integer `i` and returns the `i`th element in the deque.

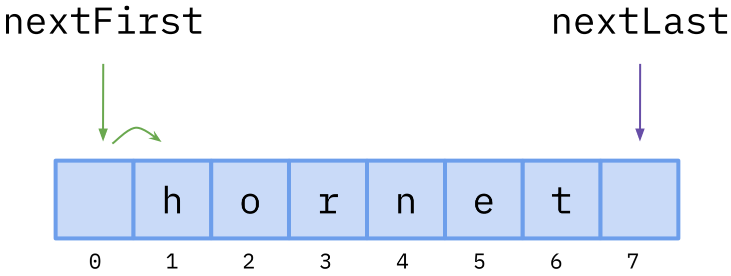
# Get

One of the methods in the Deque interface is `get`, which takes in an integer `i` and returns the `i`th element in the deque.

This is a surprisingly complex problem, given that the start of our Deque could be anywhere in the array. Let's answer a slightly easier question first: how do we get the index of the first item in our deque?

# Get

Our nextFirst pointer is always one spot before the Deque's first element, so maybe we could just try `nextFirst + 1.`

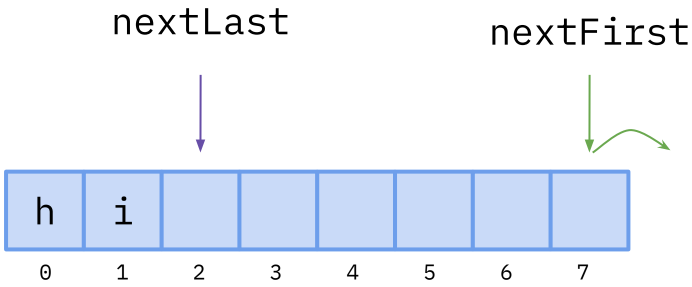


It actually almost works flawlessly! There is exactly one case where it fails.

- Can you think of the case where nextFirst + 1 fails?
- Given you have access to the backing array's length, how could you modify the expression to always work? *Hint: Use the modulus (%) operator!*

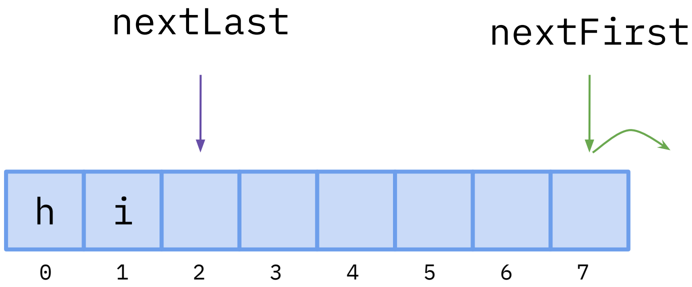
# Get

`nextFirst + 1` fails if nextFirst is pointing to the last place in the array!



# Get

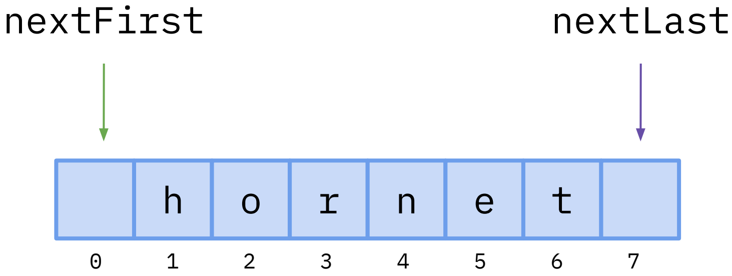
`nextFirst + 1` fails if `nextFirst` is pointing to the last place in the array!



A better expression is `(nextFirst + 1) % backingArray.length`

# Get

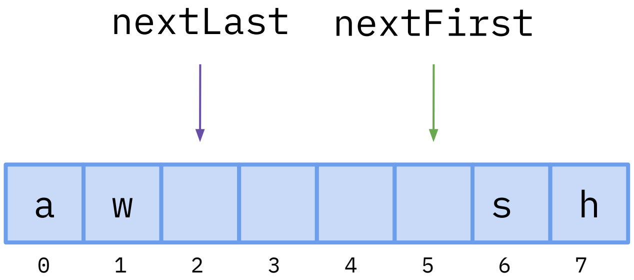
Now, given that you have access to `backingArray`, `nextFirst`, and `i`, could you write an expression to get the `i`th element of the Deque?



nextFirst nextLast

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | h | o | r | n | e | t | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

```
nextFirst = 0
nextLast = 7
length = 8
get(3) should return "n"
```



nextLast nextFirst

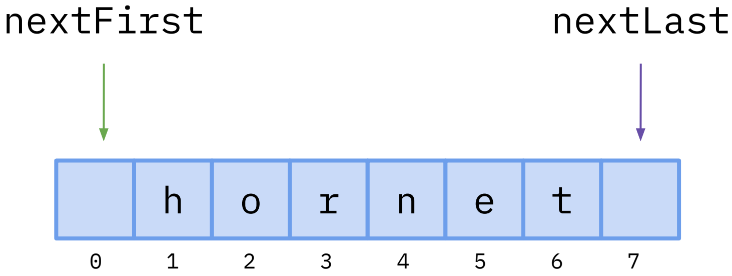
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| a | w | | | | | s | h |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

```
nextFirst = 5
nextLast = 2
length = 8
get(3) should return "w"
```

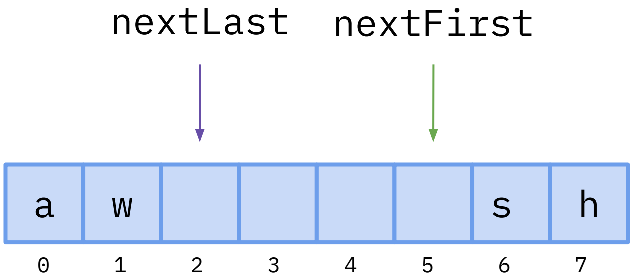
Talk to a partner and give it a try!

# Get

```
backingArray[(nextFirst + 1 + i) % length]
```



```
nextFirst = 0
nextLast = 7
length = 8
get(3) should return “n”
```



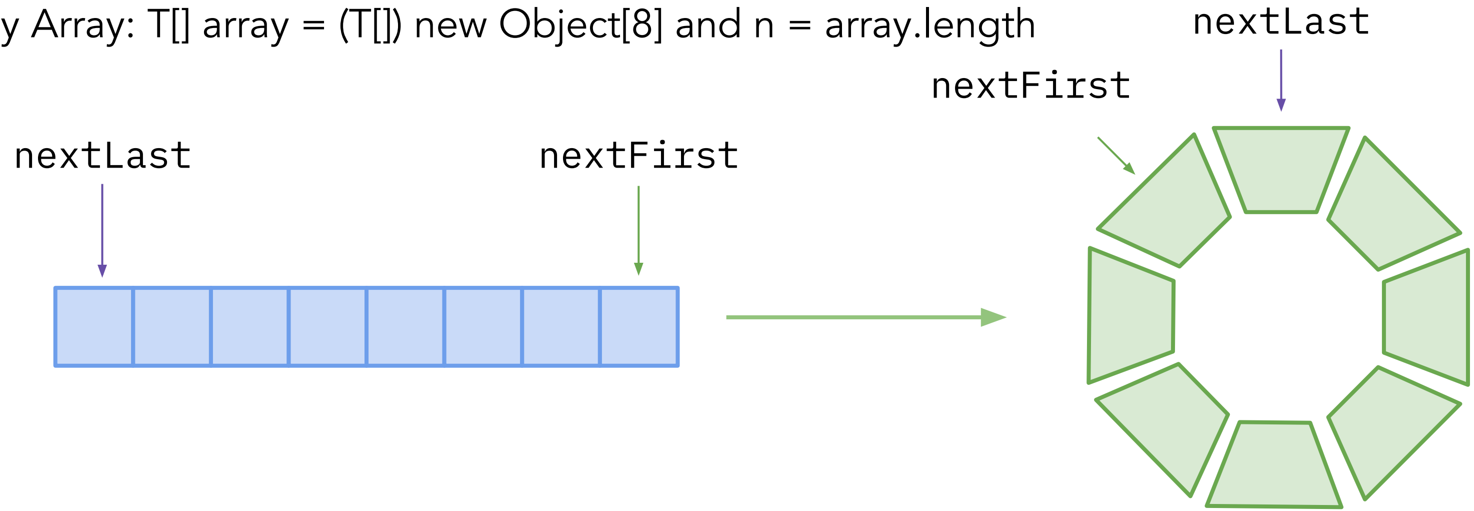
```
nextFirst = 5
nextLast = 2
length = 8
get(3) should return “w”
```

Now we have a functioning `get` method!

# ArrayDeque61B Implementation

# Circular backing array: empty array

Empty Array: T[] array = (T[]) new Object[8] and n = array.length

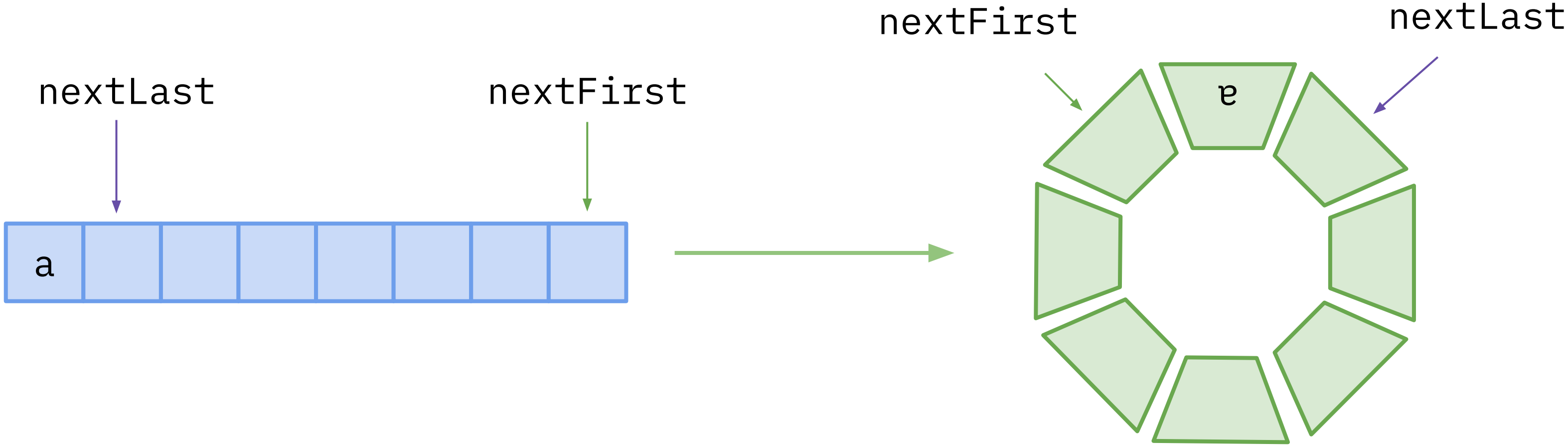


**Variants:**

```
nextLast = 0
nextFirst = n-1
```

# addLast(x)

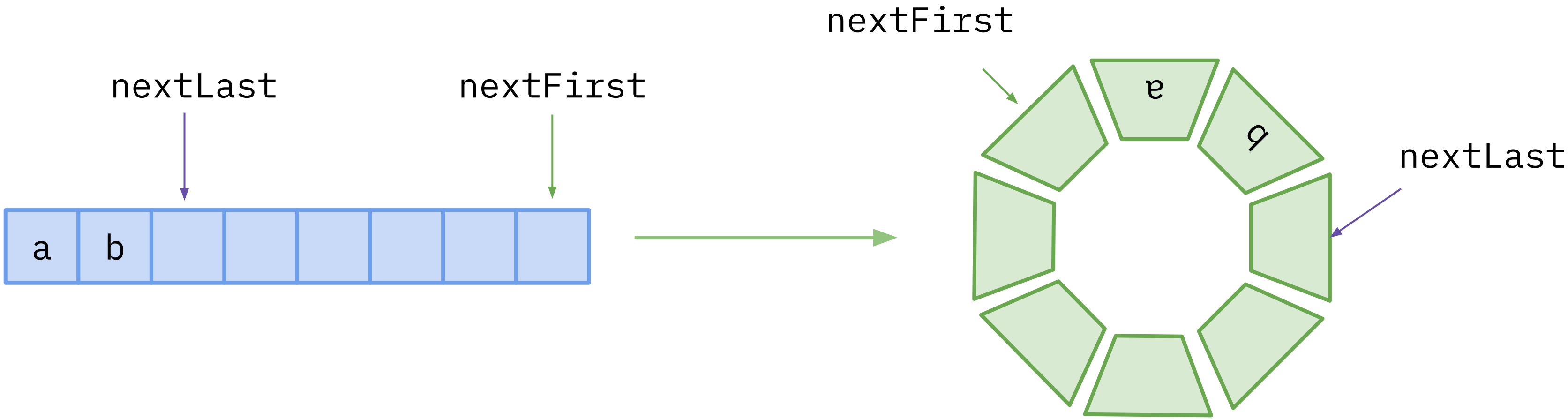
T[] array = (T[]) new Object[8] and n = array.length



```
addLast("a");
```

# addLast(x)

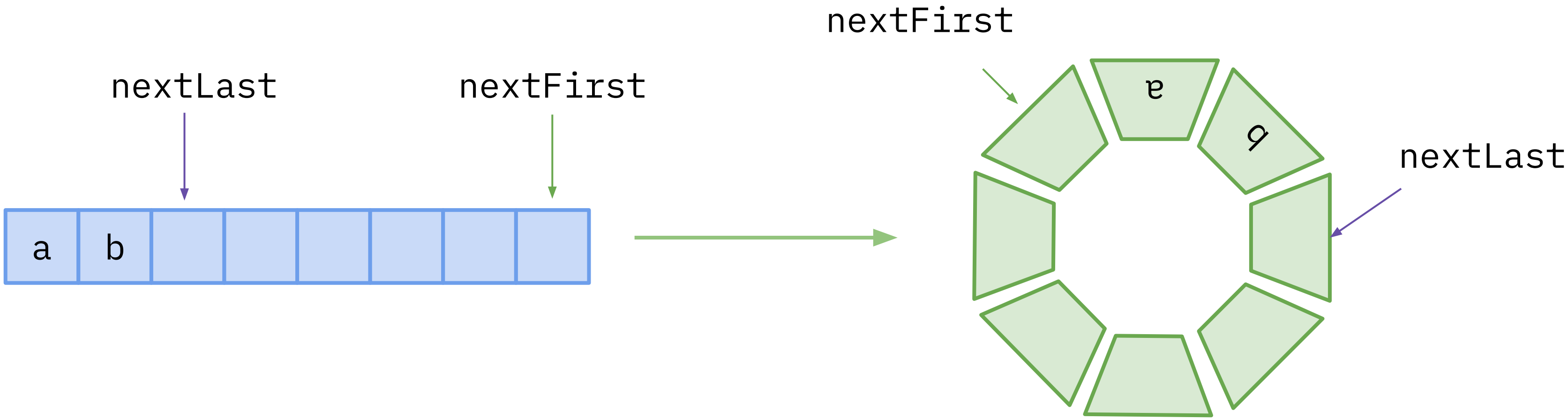
T[] array = (T[]) new Object[8] and n = array.length



```
addLast("a");
addLast("b");
```

# addLast(x)

T[] array = (T[]) new Object[8] and n = array.length



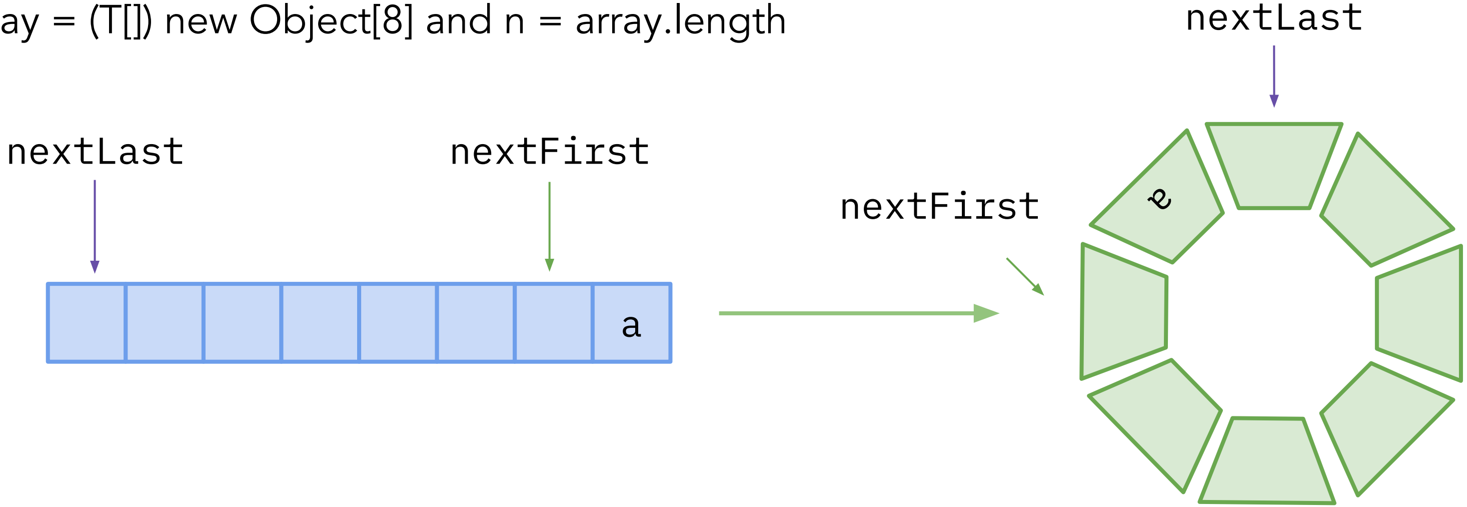
**Variants:**

```
array[nextLast] = x;
nextLast = (nextLast+1) % n;

E.g. if nextLast+1 = n, nextLast = (nextLast+1) % n = n % n = 0
```

# addFirst(x)

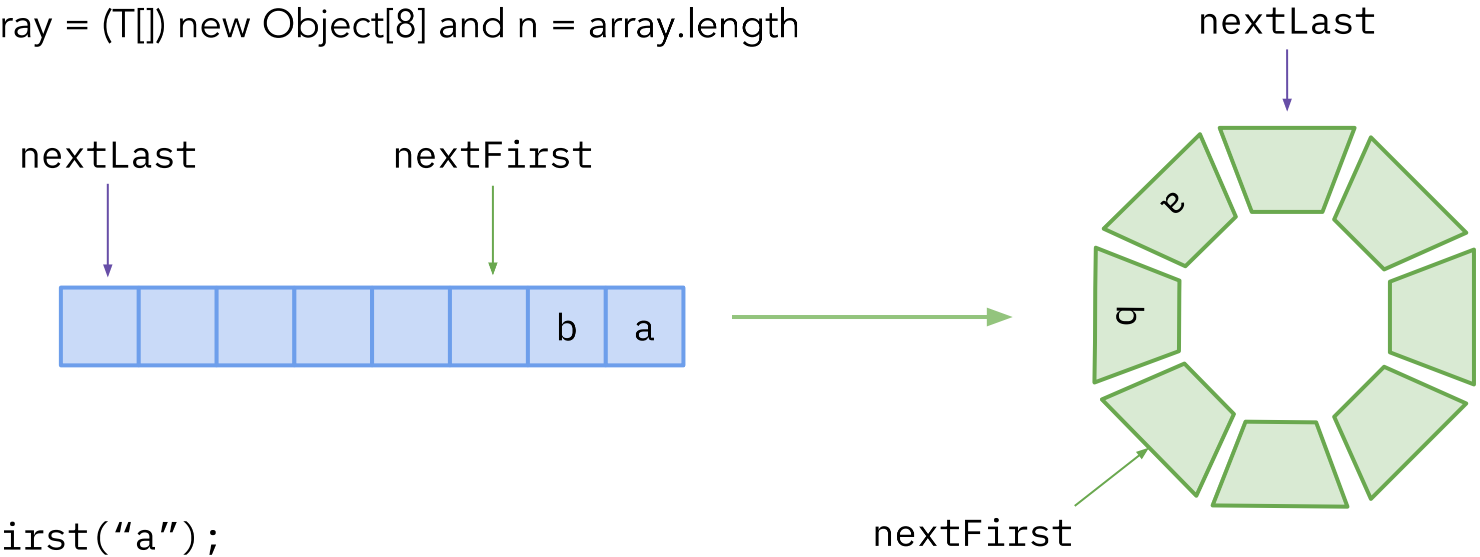
T[] array = (T[]) new Object[8] and n = array.length



```
addFirst("a");
```

# addFirst(x)

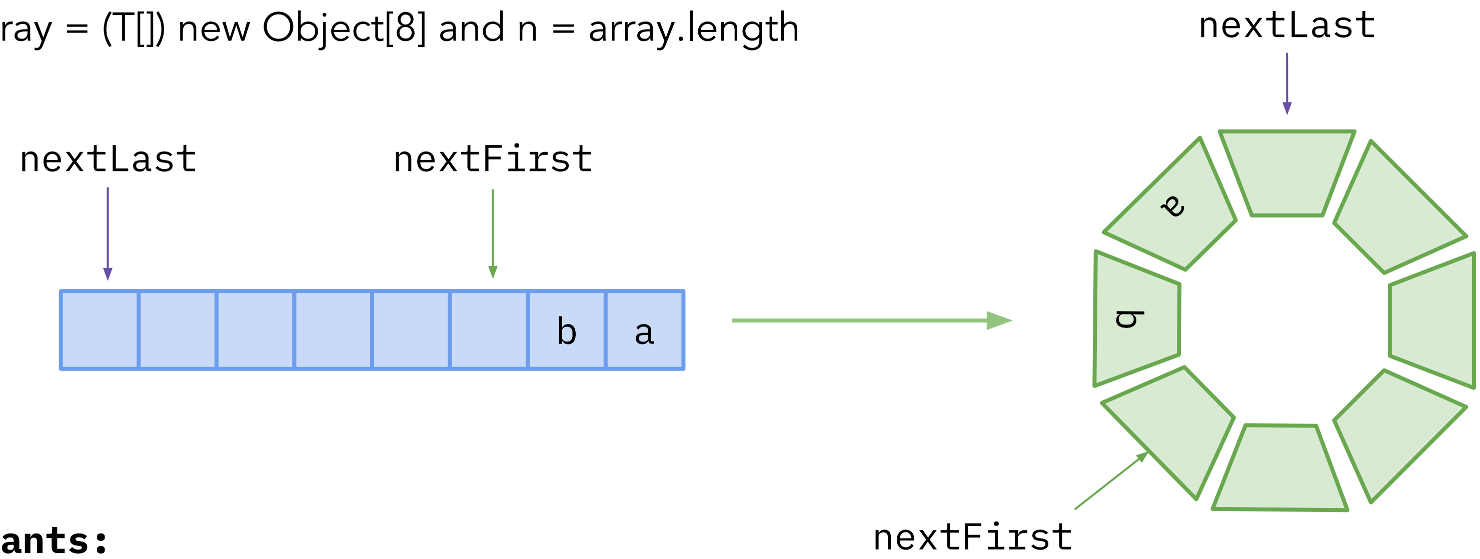
T[] array = (T[]) new Object[8] and n = array.length



```
addFirst(“a”);
addFirst(“b”);
```

# addFirst(x)

T[] array = (T[]) new Object[8] and n = array.length

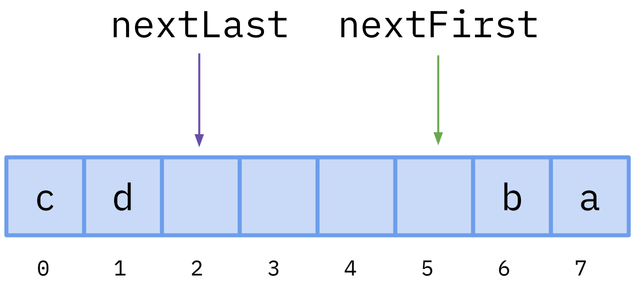


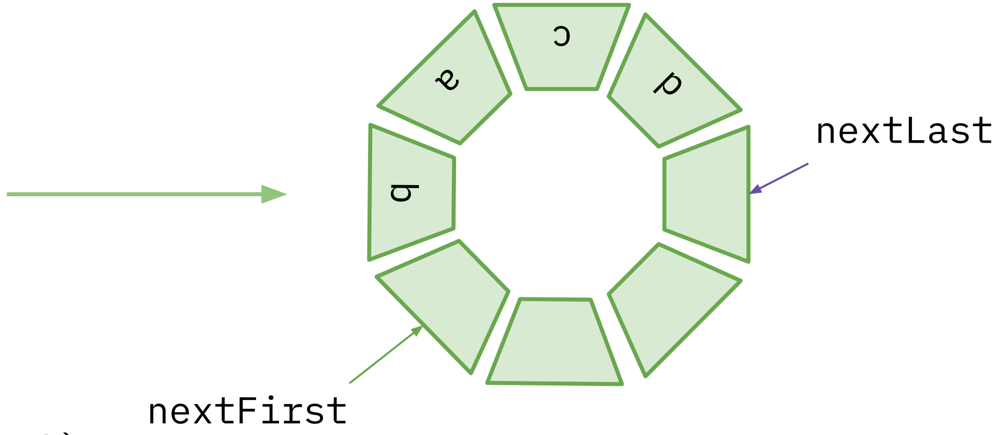
**Variants:**

```
array[nextFirst] = x;
nextFirst = (nextFirst-1+n) % n;
```

# get(index)

T[] array = (T[]) new Object[8] and n = array.length





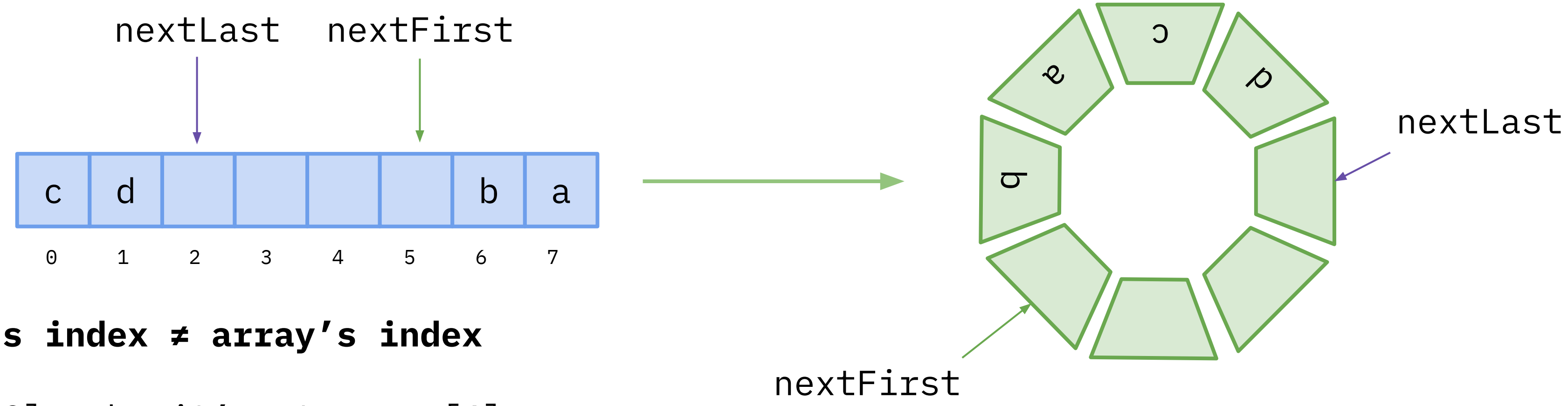
**deque ≠ array**

1.deque’s range is (nextFirst, nextLast),
but the array’s range is [0, n-1]

2.deque’s items are [b, a, c, d], but the array’s items are
[c, d, null, null, null, null, b, a]

# get(index)

T[] array = (T[]) new Object[8] and n = array.length



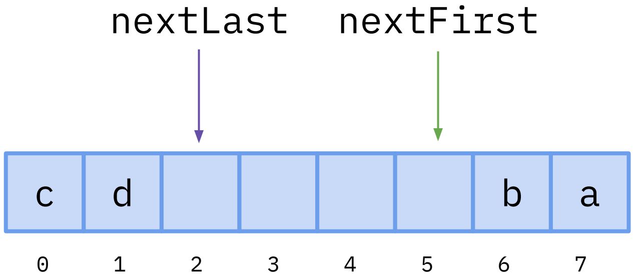
**deque’s index ≠ array’s index**

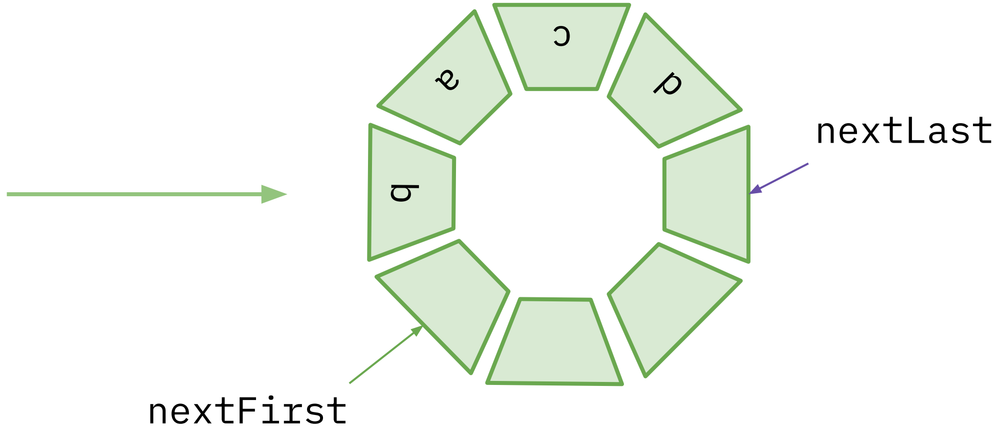
```
deque[0] = b, it’s at array[6]
deque[2] = c, it’s at array[0]

6 = nextFirst + 1 + 0 = 5 + 1 + 0
0 = (nextFirst + 1 + 2) % 8 = (5 + 1 + 2) % 8 = 8 % 8
```

# get(index)

T[] array = (T[]) new Object[8] and n = array.length





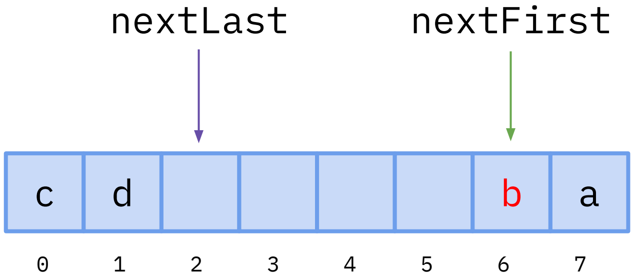
**Variants:**

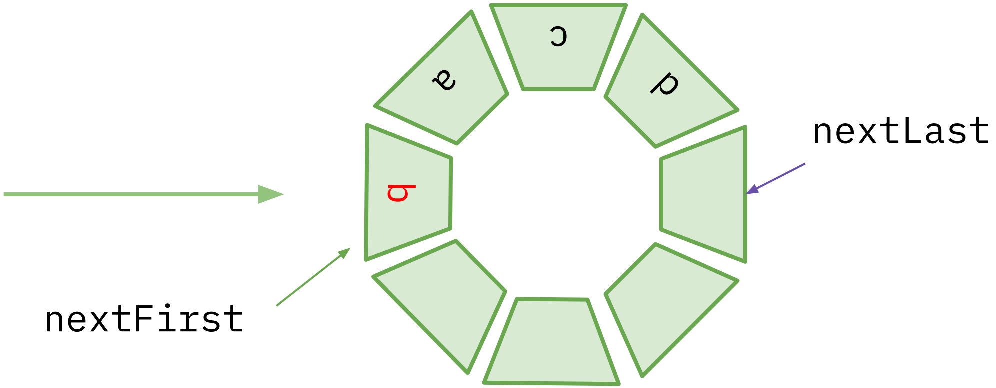
`deque[i] = array[(nextFirst+1+i)%n]`

`get(index) = array[(nextFirst+1+index)%n]`

# removeFirst()

T[] array = (T[]) new Object[8] and n = array.length

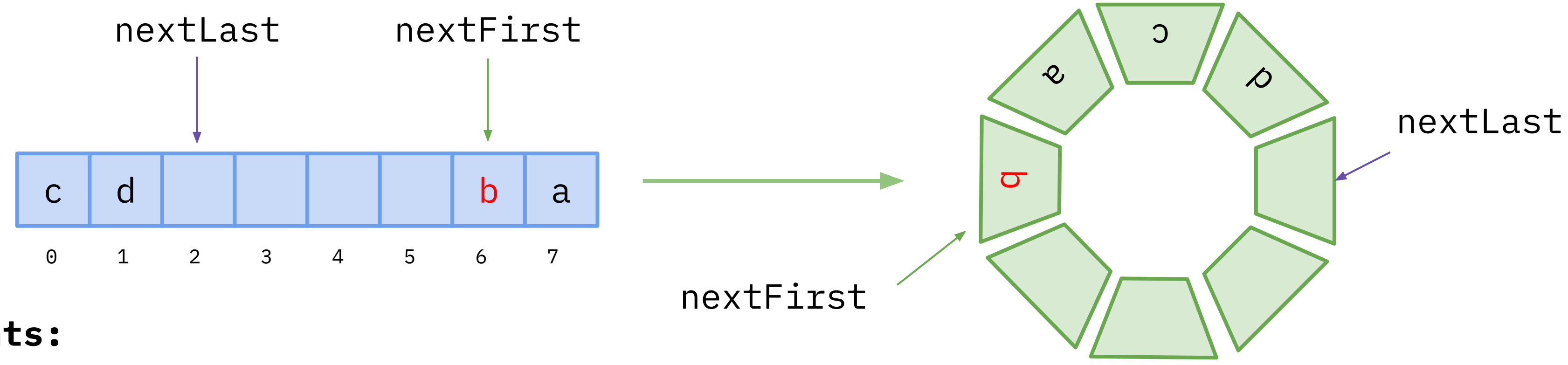




```
T first = removeFirst(); // b
```

# removeFirst()

T[] array = (T[]) new Object[8] and n = array.length



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| c | d |  |  |  |  | b | a |

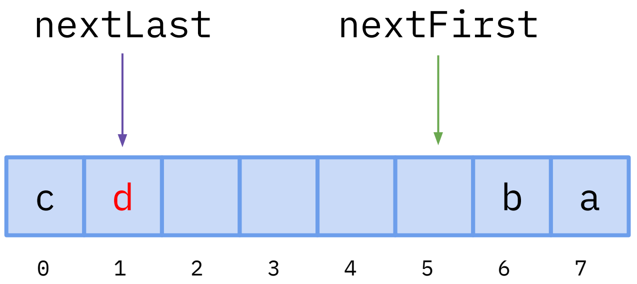
**Variants:**

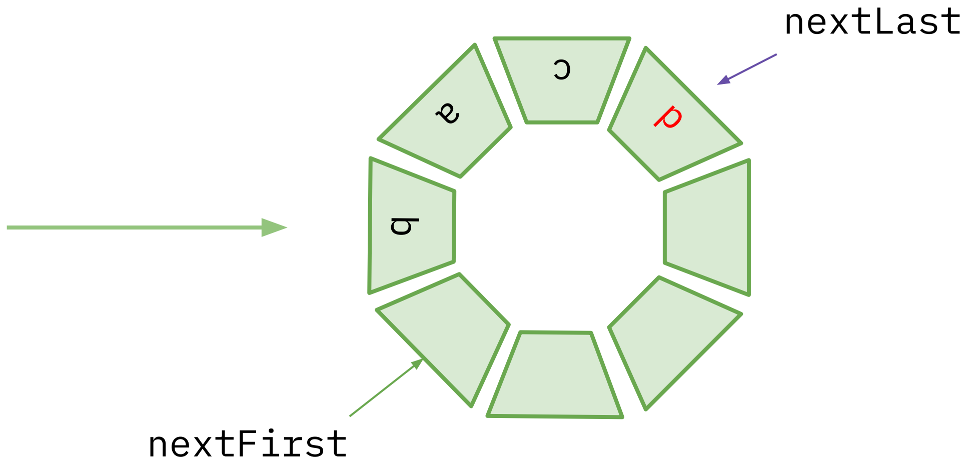
```
T first = get(0);                  // get(0) = deque(0)
nextFirst = (nextFirst + 1) % n; // opposite to addFirst()
array[nextFirst] = null;           // null out first to release memory

then return first;
```

# removeLast()

T[] array = (T[]) new Object[8] and n = array.length

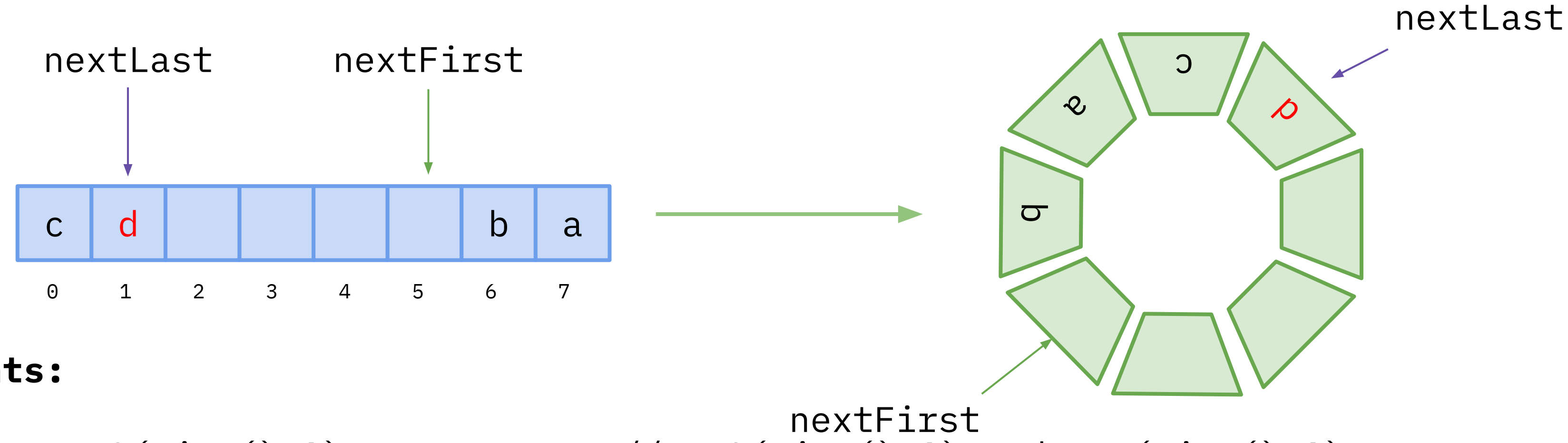




```
T last = removeLast(); // d
```

# removeLast()

T[] array = (T[]) new Object[8] and n = array.length



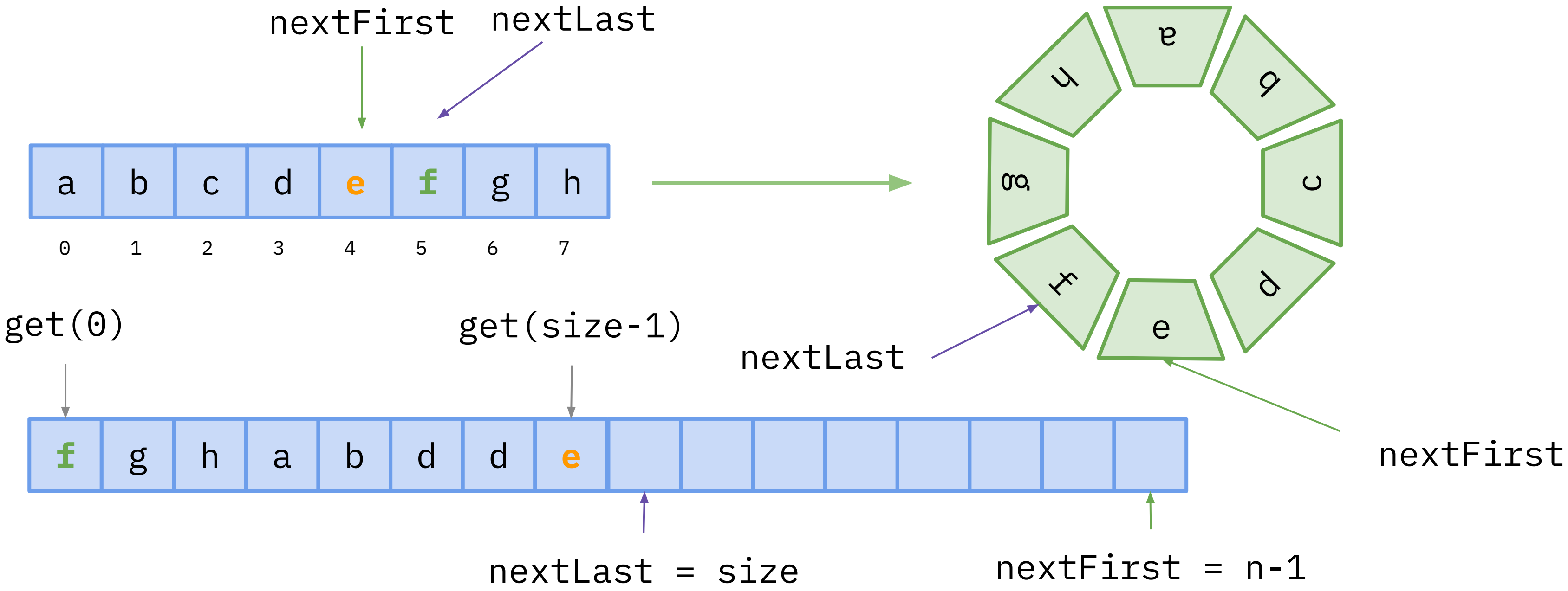
**Variants:**

```
T last = get(size()-1);                // get(size()-1) = deque(size()-1)
nextLast = (nextLast - 1 + n) % n;  // opposite to addLast()
array[nextLast] = null;                // null out last to release memory

then return last;
```

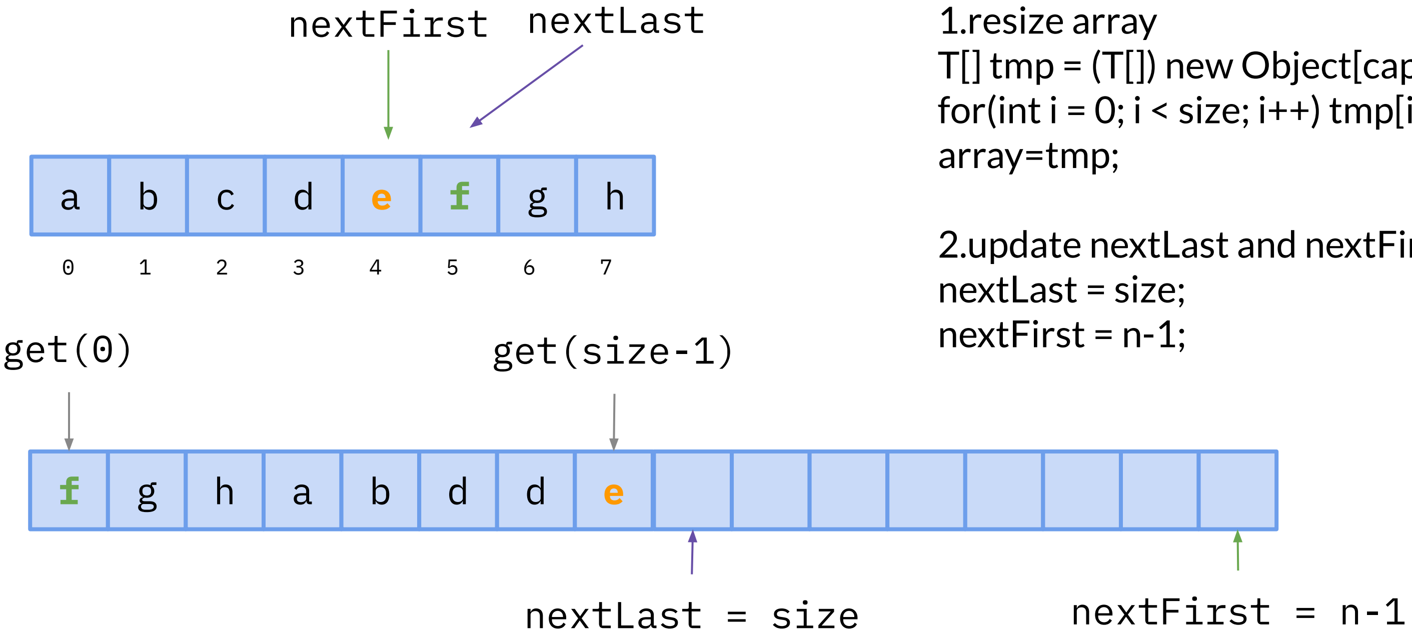
# resize(cap)

T[] array = (T[]) new Object[8] and n = array.length



# resize(cap)

T[] array = (T[]) new Object[8] and n = array.length



| a | b | c | d | e | f | g | h |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

| f | g | h | a | b | d | d | e | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

**Variants:**

1.resize array

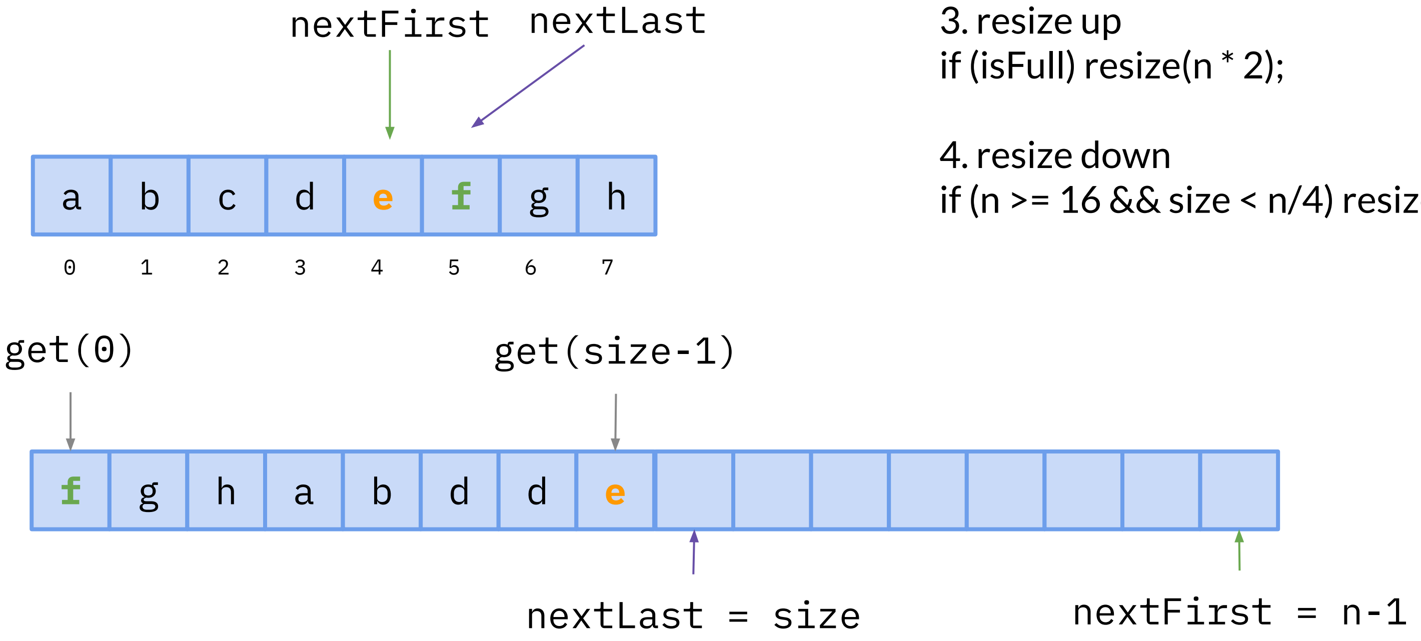
```
T[] tmp = (T[]) new Object[cap];
for(int i = 0; i < size; i++) tmp[i] = get(i);
array=tmp;
```

2.update nextLast and nextFirst

```
nextLast = size;
nextFirst = n-1;
```

# resize(cap)

T[] array = (T[]) new Object[8] and n = array.length



nextFirst nextLast

| a | b | c | d | e | f | g | h |
|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

get(0) get(size-1)

| f | g | h | a | b | d | d | e | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

nextLast = size

nextFirst = n-1

**Variants:**

3. resize up
if (isFull) resize(n * 2);

4. resize down
if (n >= 16 && size < n/4) resize(n / 4);