Array-based lists

part 2

Alina Shaikhet

Array-based implementations

of the **List** and **Queue** interfaces

	get(i) / set(i,x)	add(i,x) / remove(i)
ArrayStack	$\boldsymbol{O}(\mathbf{1})$	$\boldsymbol{O}(\mathbf{1}+\boldsymbol{n}-\boldsymbol{i})$
ArrayDeque	$\boldsymbol{O}(\mathbf{1})$	$\boldsymbol{O}(\mathbf{1}+\mathbf{min}\{\boldsymbol{i},\boldsymbol{n}-\boldsymbol{i}\})$
DualArrayDeque	$\boldsymbol{O}(\mathbf{1})$	$\boldsymbol{O}(\mathbf{1}+\mathbf{min}\{\boldsymbol{i},\boldsymbol{n}-\boldsymbol{i}\})$
RootishArrayStack	$\boldsymbol{O}(\mathbf{1})$	$\boldsymbol{O}(\mathbf{1}+\boldsymbol{n}-\boldsymbol{i})$

Review

we have seen

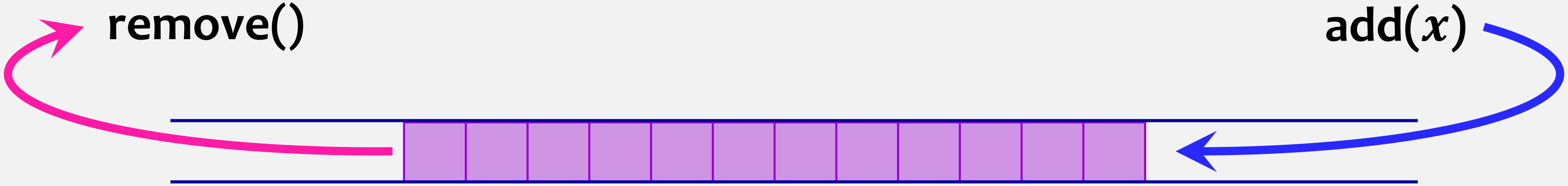
today

Implementations of the **List** interface

	get(i) / set(i,x)	add(i,x) / remove(i)
ArrayStack	$O(1)$	$O(1+n-i)$
LinkedList	$O(1+\min\{i,n-i\})$	$O(1+\min\{i,n-i\})$
ods ArrayDeque	$O(1)$	$O(1+\min\{i,n-i\})$

FIFO Queue

FIFO Queue represents a sequence of FIFO elements.
We add to the end of the queue and remove from the front.



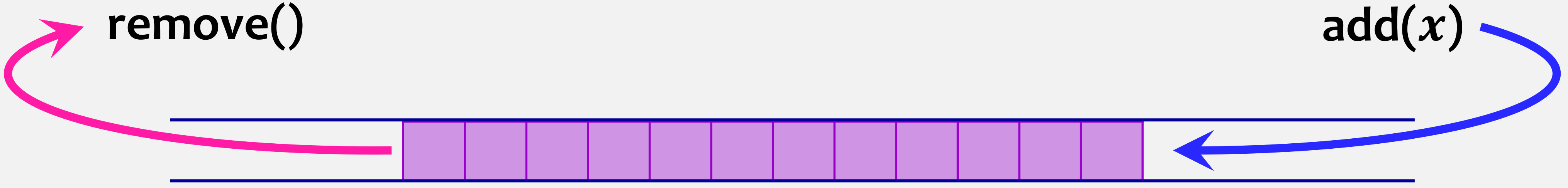
size()
add(x)
remove() – remove and return the "oldest" element

We can use **ArrayList** implementation and add/remove only to front/back. However, this won't get us our desired $\boldsymbol{O}(\mathbf{1})$ time for add/remove operations.

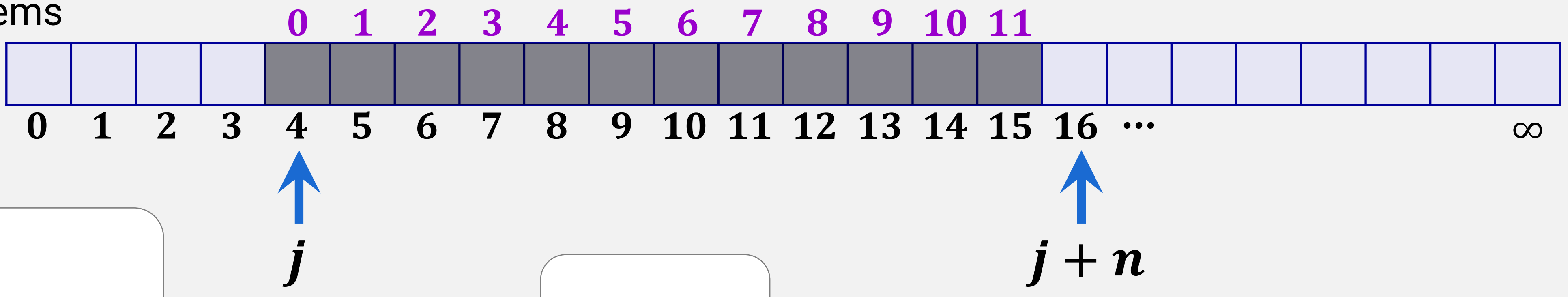
add(x)	**ArrayList.add(size(), x)**	or	**ArrayList.add(0, x)**
remove()	**ArrayList.remove(0)**		**ArrayList.remove(size() − 1)**

To avoid shifting we will use "pointer" to the front index of our queue elements ($\boldsymbol{j}$)

FIFO Queue



we store queue items at consecutive locations in this array



constructor:

a = **new T**[∞];
$n = 0$;
$j = 0$;

T[] a;
int n;
int j;

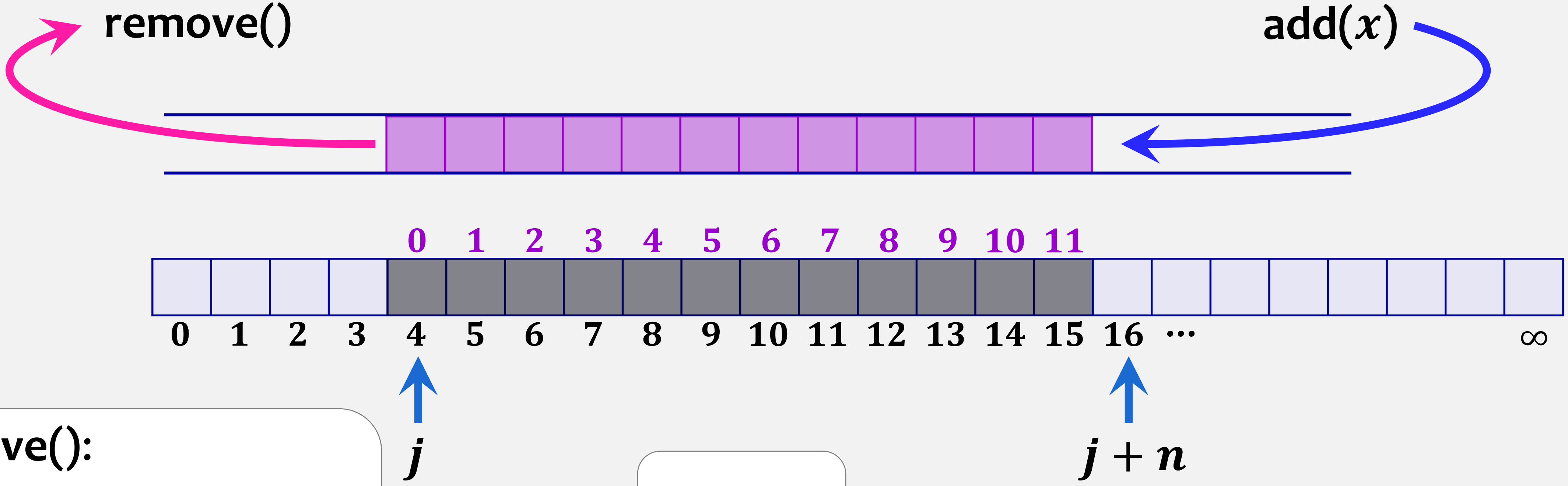
index at which queue begins

add(x) – which position do we add?

it is no longer at position n

in our example $n = 12,\ j = 4$

FIFO Queue



remove():

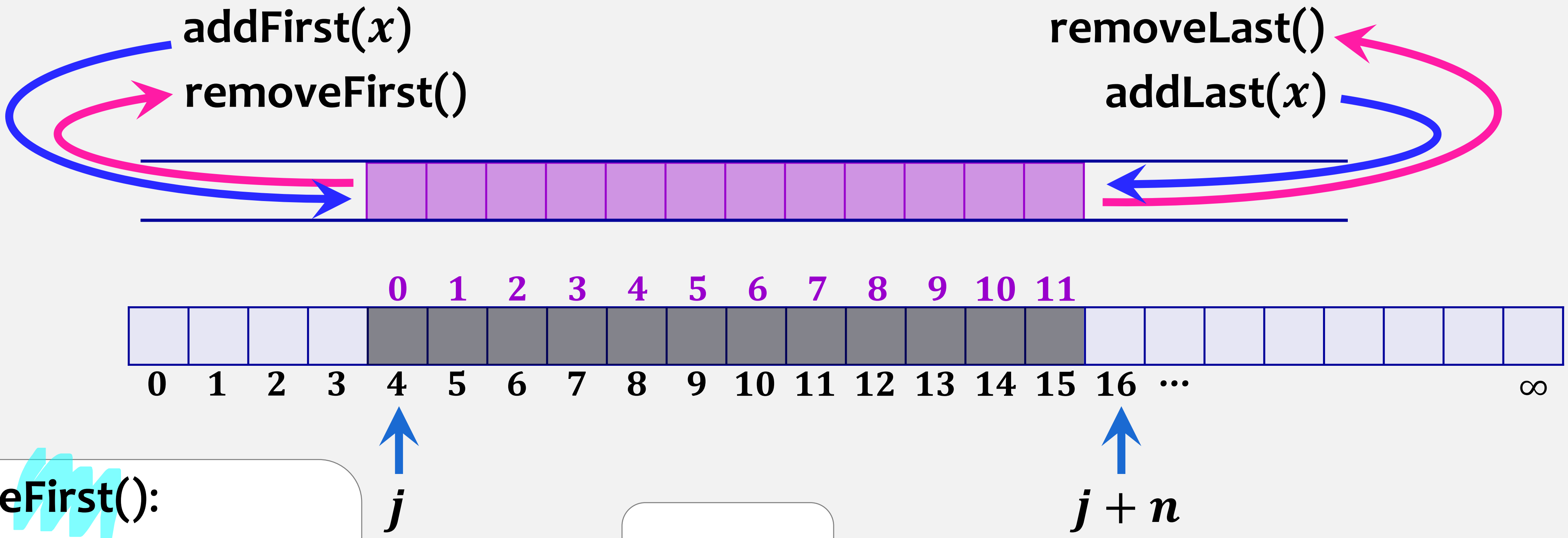
if $(n == 0)$ throw exception
T $x = a[j]$;
$a[j] = \mathbf{null}$;
$j++$;
$n--$;
return x;

T[] a;
int n;
int j;

add(x):

$a[j+n] = x$;
$n++$;

Deque



removeFirst():

if $(n == 0)$ throw exception
T $x = a[j]$;
$a[j] = \mathbf{null}$;
$j++$;
$n--$;
return x;

T[] a;
int n;
int j;

addLast(x):

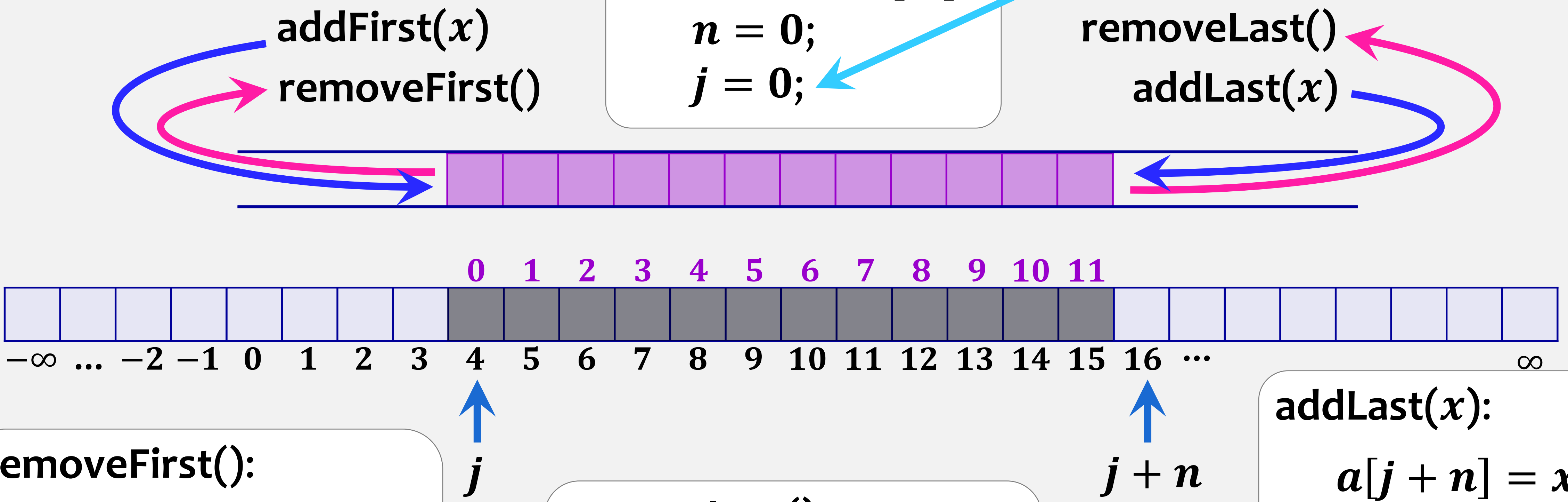
$a[j+n] = x$;
$n++$;

Deque

constructor:

$a = \text{new T}[\infty];$
$n = 0;$
$j = 0;$

but you can choose any other index



removeFirst():

if $(n == 0)$ throw exception
T $x = a[j];$
$a[j] = \text{null};$
$j++;$
$n--;$
return x;

removeLast():

if $(n == 0)$ throw exception
T $x = a[j+n-1];$
$a[j+n-1] = \text{null};$
$n--;$
return x;

addLast(x):

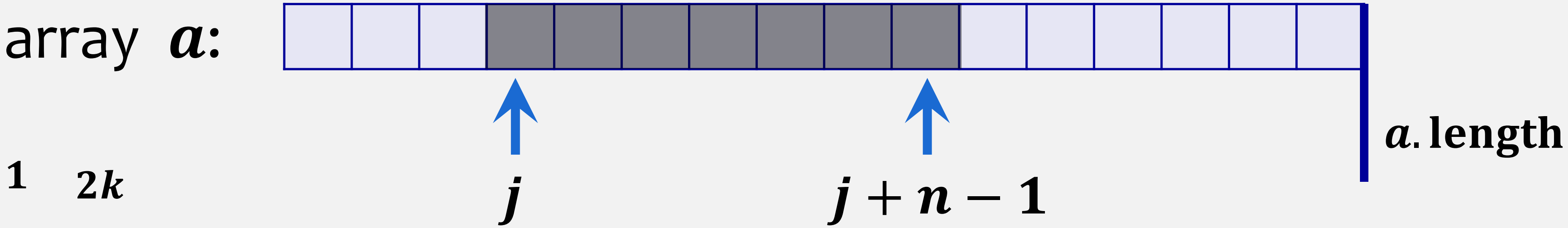
$a[j+n] = x;$
$n++;$

addFirst(x):

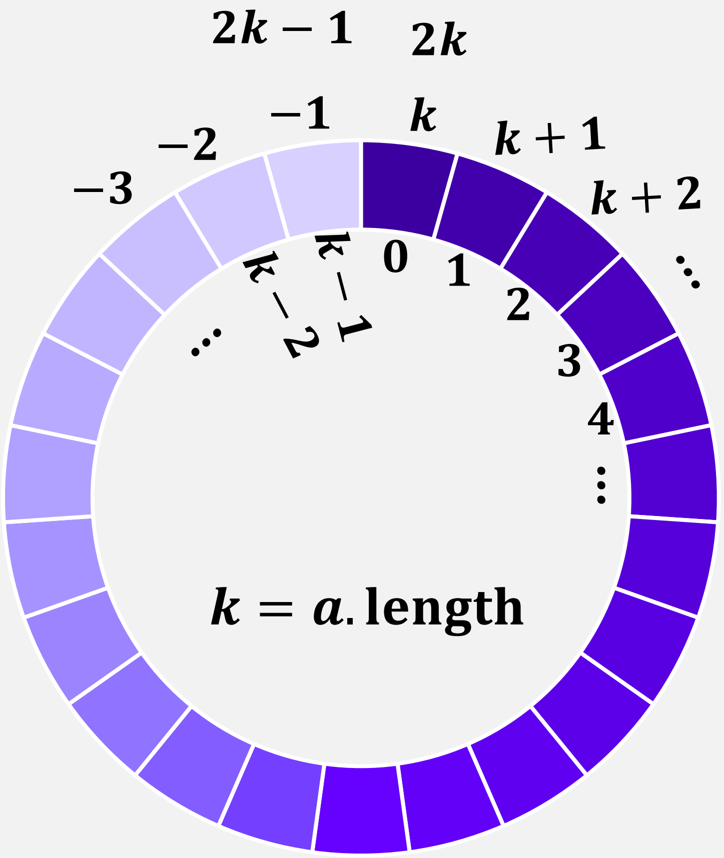
$j--;$
$a[j] = x;$
$n++;$

Circular Array & MOD

In Java we write $i\%k$. Modulus operator (%) returns the division remainder.



j – the first index
n – number of elements in the queue



circular array

$i \bmod k \in \{0, \dots, k-1\}$

$i \bmod k = r$ ← remainder

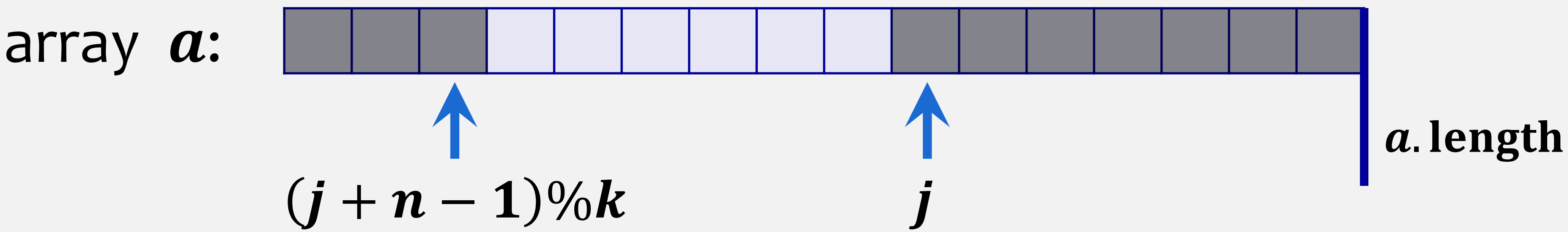
$r \in \{0, \dots, k-1\}$

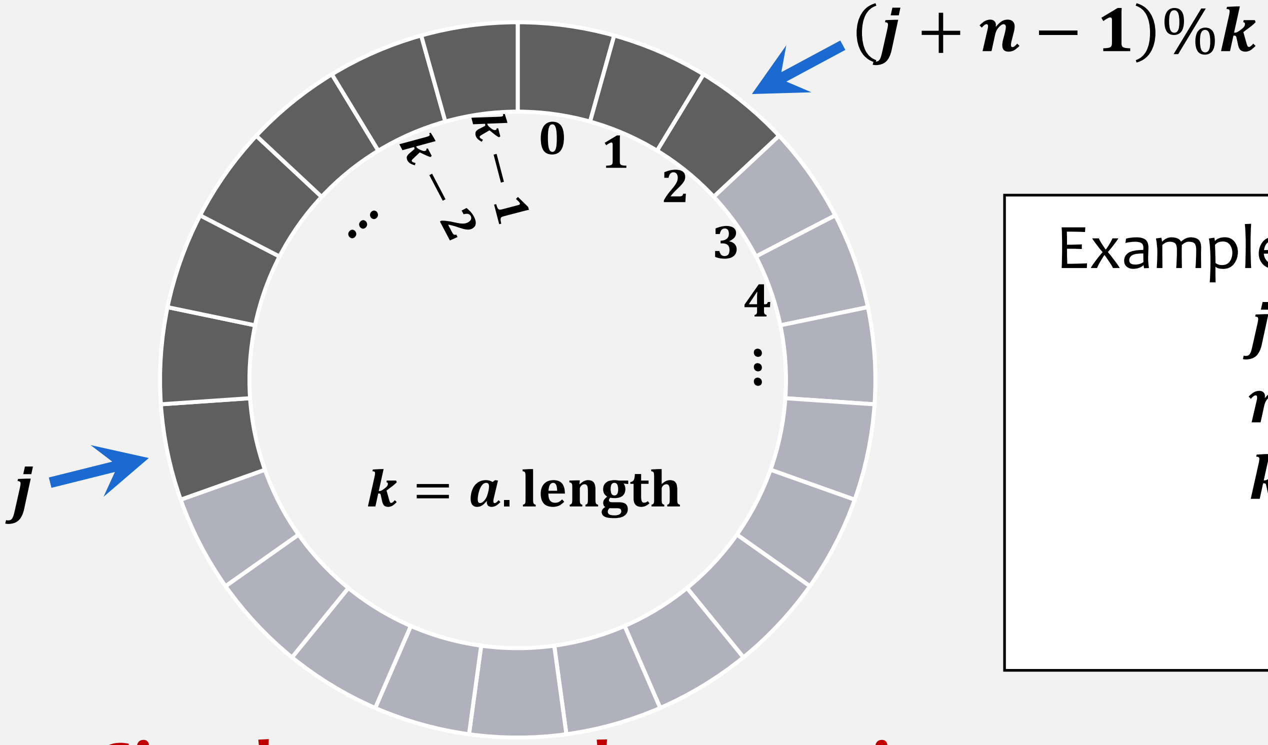
some integer

$i = p \cdot k + r$

$i \bmod k = i - p \cdot k$

Circular Array & MOD





Example:

$\boldsymbol{j}=\mathbf{17}$

$\boldsymbol{n}=\mathbf{11}$

$\boldsymbol{k}=\mathbf{24}$

$(\boldsymbol{j}+\boldsymbol{n}-\mathbf{1})\%\boldsymbol{k}$

$(\mathbf{17}+\mathbf{11}-\mathbf{1})\%\mathbf{24}$

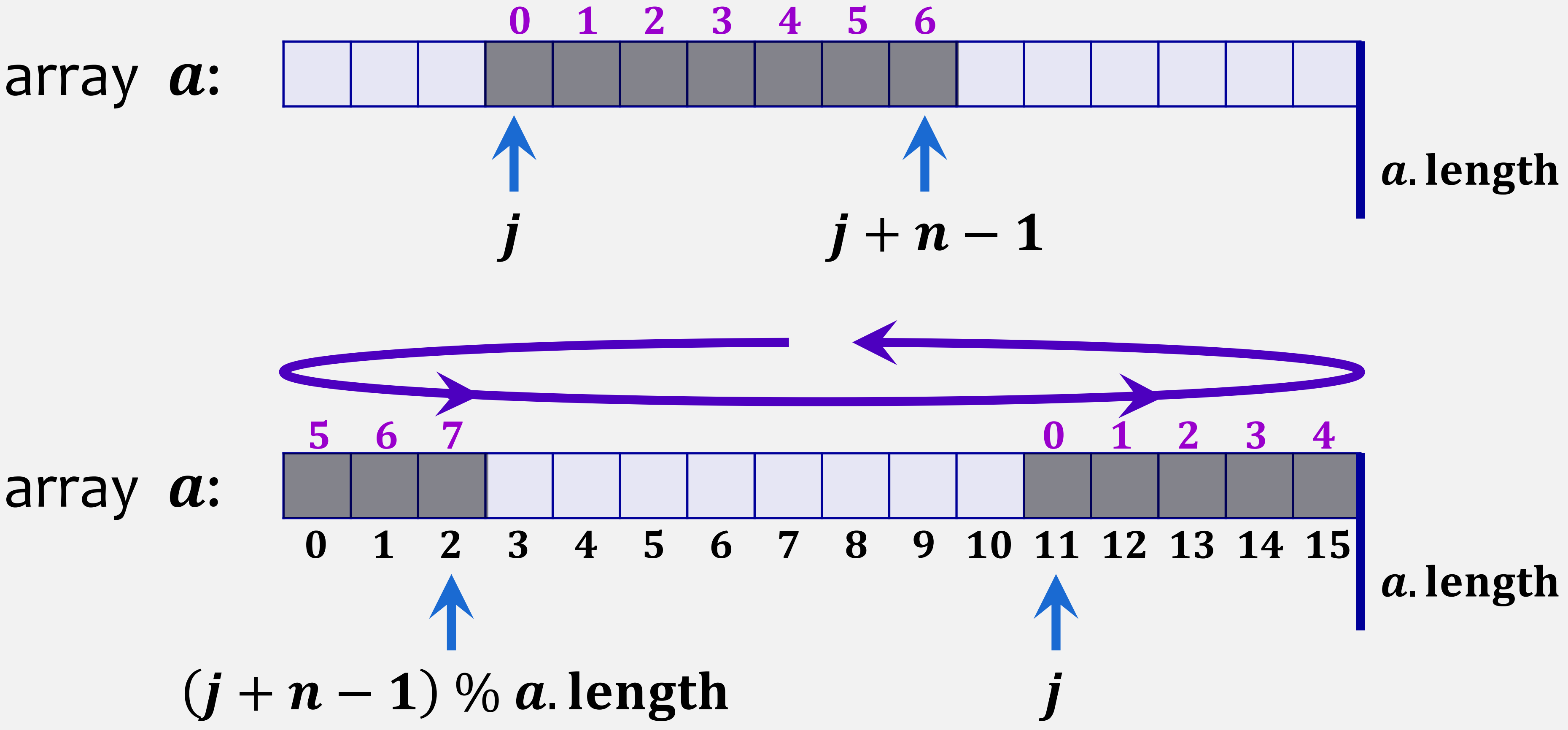
$\mathbf{27}\%\mathbf{24}$

$\mathbf{3}$

Circular arrays do not exist

$\boldsymbol{i}\ \mathbf{mod}\ \boldsymbol{k}=\boldsymbol{i}-\boldsymbol{p}\cdot\boldsymbol{k}$

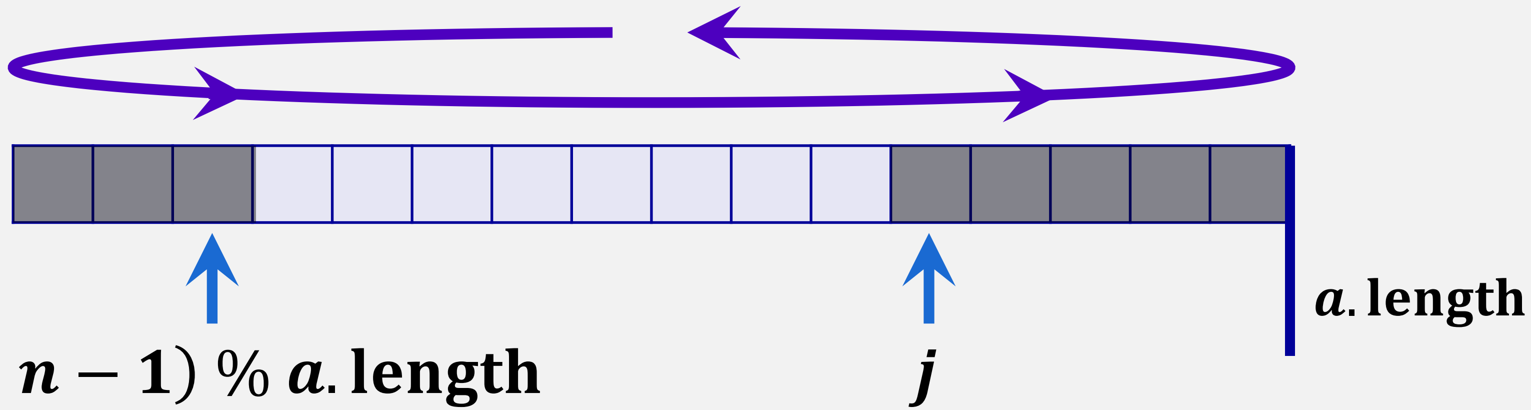
Circular Array & MOD



ArrayDeque in JCF

Deque

array $\boldsymbol{a}$:



constructor:

$\boldsymbol{a} = \text{new T}[1];$
$\boldsymbol{n} = 0;$
$\boldsymbol{j} = 0;$

We need to guarantee that $\boldsymbol{j}$ is a valid index

addFirst(x):

if $(n + 1 > a.\text{length})$ then **resize()**;
$j = (j == 0)\ ?\ a.\text{length} - 1 : j - 1;$
$a[j] = x;$
$n++;$

removeFirst():

if $(n == 0)$ throw exception
T $x = a[j];$
$a[j] = \text{null};$
$j = (j + 1)\ \%\ a.\text{length};$
$n--;$
if $(3n \le a.\text{length})$: **resize()**;
return x;

removeLast():

if $(n == 0)$ throw exception
T $x = a[(j + n - 1)\%\ a.\text{length}];$
$a[(j + n - 1)\%\ a.\text{length}] = \text{null};$
$n--;$
if $(3n \le a.\text{length})$: **resize()**;
return x;

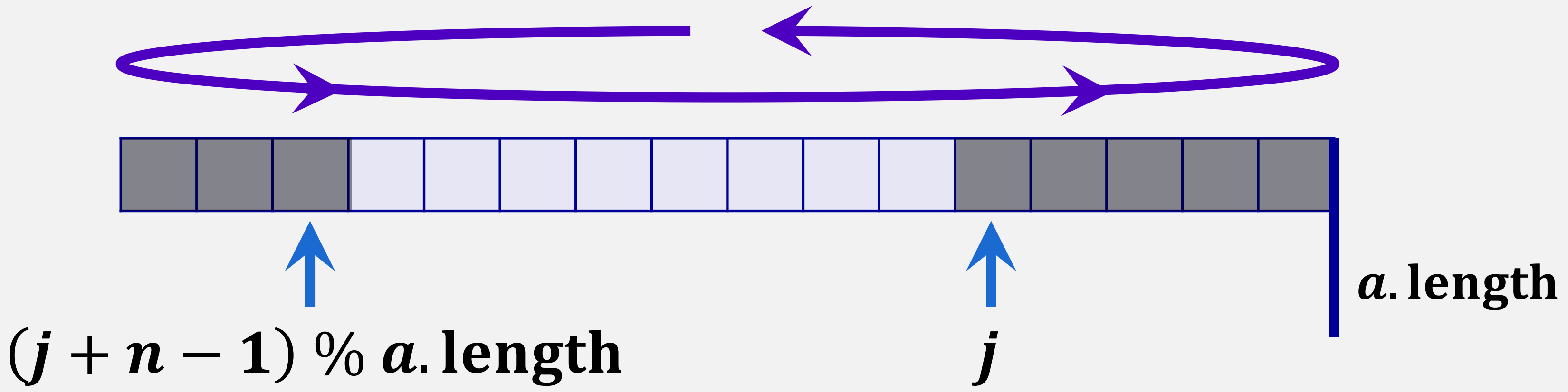
ignoring **resize()** $O(1)$

addLast(x):

if $(n + 1 > a.\text{length})$ then **resize()**;
$a[(j + n)\%\ a.\text{length}] = x;$
$n++;$

Alina Shaikhet – COMP 2402 – Carleton University

FIFO Queue



constructor:

$a = \text{new T}[1];$
$n = 0;$
$j = 0;$

Refer to ods textbook: **ArrayQueue** implements the FIFO Queue interface

remove():

if $(n == 0)$ throw exception
T $x = a[j];$
$a[j] = \textbf{null};$
$j = (j + 1)\ \%\ a.\text{length};$
$n - -;$
if $(3n \le a.\text{length})$: **resize();**
return x;

ignoring **resize()** $O(1)$

add(x):

if $(n + 1 > a.\text{length})$ then **resize();**
$a[(j + n)\%\ a.\text{length}] = x;$
$n + +;$

resize()

void **resize()**:

T[] $\boldsymbol{b} =$ new array$(\mathbf{max}\{\mathbf{2}\boldsymbol{n}, \mathbf{1}\})$

for $(\boldsymbol{i} = \mathbf{0}; \boldsymbol{i} < \boldsymbol{n};\ \boldsymbol{i}++)$

$\boldsymbol{b}[\boldsymbol{i}] = \boldsymbol{a}[(\boldsymbol{j} + \boldsymbol{i})\%\boldsymbol{a}.\mathbf{length}];$

$\boldsymbol{a} = \boldsymbol{b};$

$\boldsymbol{j} = \mathbf{0};$

to avoid 0-length array

executes $\boldsymbol{n}$ times

$\boldsymbol{O}(\boldsymbol{n})$

Theorem 2.2

An **ArrayQueue** implements the (FIFO) Queue interface. Ignoring the cost of calls to **resize()**, an **ArrayQueue** supports the operations **add($\boldsymbol{x}$)** and **remove()** in $\boldsymbol{O}(\mathbf{1})$ time per operation. Furthermore, beginning with an empty **ArrayQueue**, any sequence of $\boldsymbol{m}$ **add($\boldsymbol{x}$)** and **remove()** operations results in a total of $\boldsymbol{O}(\boldsymbol{m})$ time spent during all calls to **resize()**.

Review

Implementations of the **List** interface

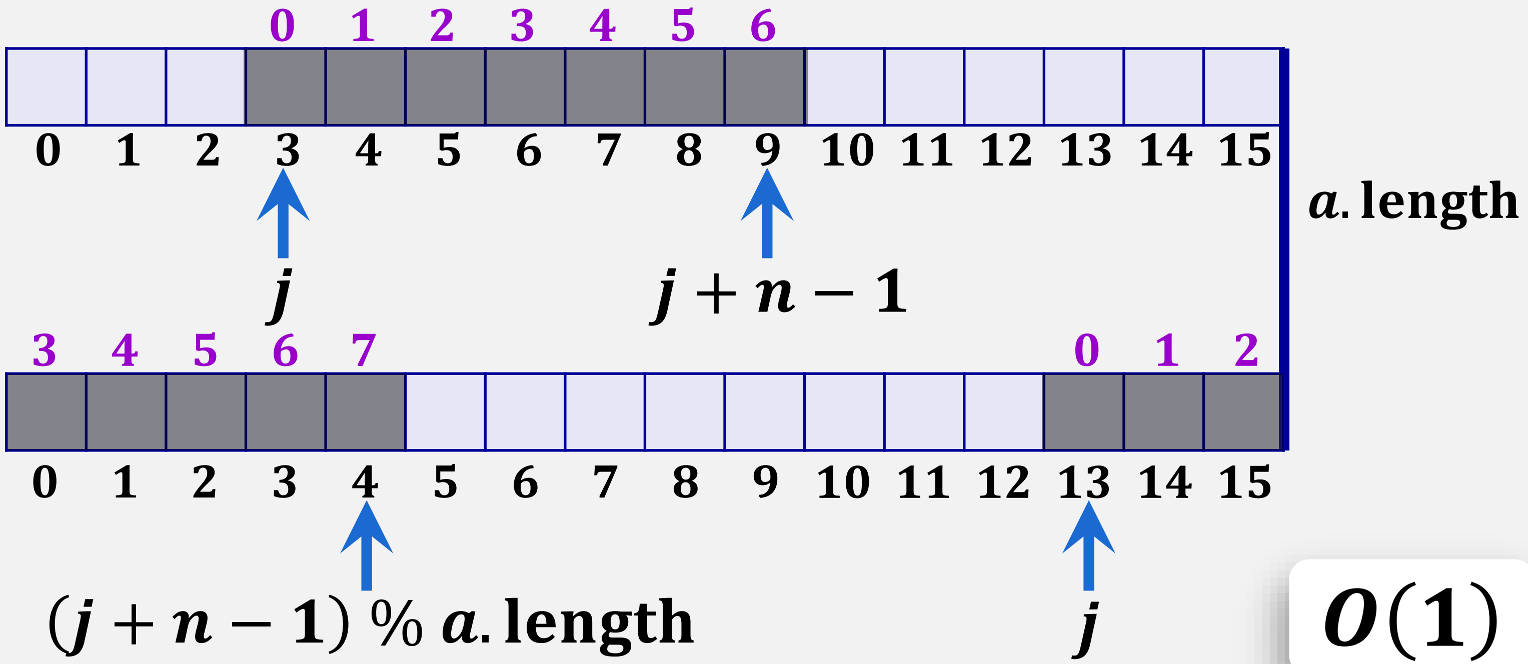
	get(i) / set(i,x)	add(i,x) / remove(i)
we have seen ArrayStack	$O(1)$	$O(1+n-i)$
LinkedList	$O(1+\min\{i,n-i\})$	$O(1+\min\{i,n-i\})$
today ods ArrayDeque	$O(1)$	$O(1+\min\{i,n-i\})$

ArrayDeque is faster then both ArrayStack and LinkedList

Alina Shaikhet – COMP 2402 – Carleton University

ArrayDeque

ArrayDeque implements the **List** interface using circular array with $O(1)$ amortized **Deque** operations (adding and removing to both ends of our sequence).



$O(1)$

T[] a;
int n;
int j;

keep j private

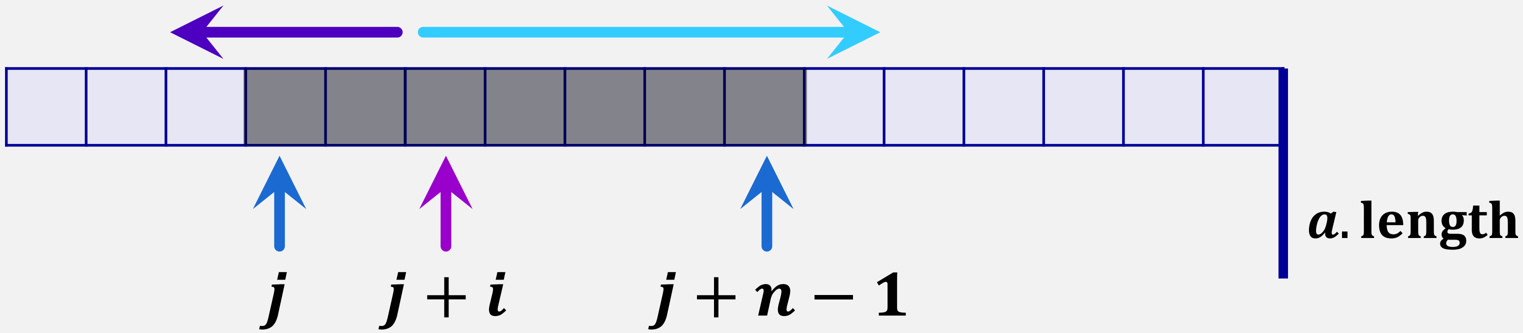
constructor:

a = **new** T[1];
$n = 0$;
$j = 0$;

get(i): check bounds;
return $a[(j + i)$ % $a.\mathbf{length}]$;

set(i,x): check bounds;
T $y = a[(j + i)$ % $a.\mathbf{length}]$; //T y = get(i);
$a[(j + i)$ % $a.\mathbf{length}] = x$;
return y;

ArrayDeque



add(i,x): if $(n+1 > a.\text{length})$ then **resize()**;

if $\left(i < \frac{n}{2}\right)$ then // i is in the first half

$j = (j == 0)\ ?\ a.\text{length} - 1 : j - 1;$ // move j left // $j = (j-1)\ \%\ a.\text{length}$

$O(i)$ shift to the left

for $(k = 0;\ k \leq i - 1;\ k++)$

$a[(j+k)\ \%\ a.\text{length}] = a[(j+k+1)\ \%\ a.\text{length}];$

else // i is in the second half

shift to the right

$O(n-i)$

for $(k = n;\ k > i;\ k--)$

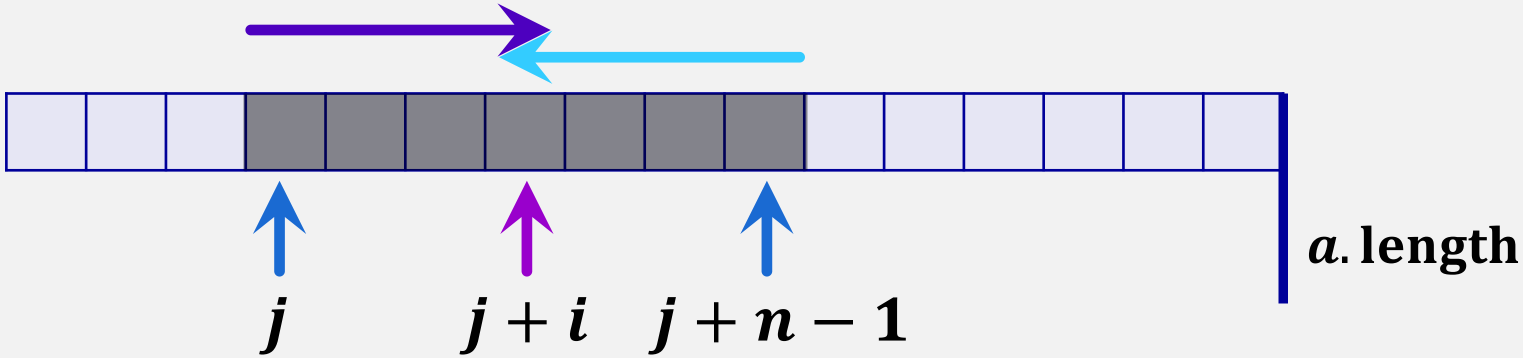
$a[(j+k)\ \%\ a.\text{length}] = a[(j+k-1)\ \%\ a.\text{length}];$

$a[(j+i)\ \%\ a.\text{length}] = x;$

$n++;$

Running time: $\boxed{O\big(1 + \min(i, n-i)\big)}$ **amortized**

ArrayDeque



remove(i):

$x = a[(j+i)\ \%\ a.\mathbf{length}];$

if $\left(i < \frac{n}{2}\right)$ then // shift $a[0],\ldots,[i-1]$ right one position

$\boxed{O(i)}$ shift $\left\{ \begin{array}{l} \text{for } (k = i;\ k > 0;\ k--) \\ \quad a[(j+k)\ \%\ a.\mathbf{length}] = a[(j+k-1)\ \%\ a.\mathbf{length}]; \end{array} \right.$

$j = (j+1)\ \%\ a.\mathbf{length}$ // move j right

else // shift $a[i+1],\ldots,a[n-1]$ left one position

$\boxed{O(n-i)}$ shift $\left\{ \begin{array}{l} \text{for } (k = i;\ k < n-1;\ k++) \\ \quad a[(j+k)\ \%\ a.\mathbf{length}] = a[(j+k+1)\ \%\ a.\mathbf{length}]; \end{array} \right.$

$n--;$

if $(3n < a.\mathbf{length})$ then **resize()**;

return x;

Running time: $\boxed{O\big(1+\min(i, n-i)\big)}$ **amortized**

Theorem 2.3

An **ArrayDeque** implements the **List** interface. Ignoring the cost of calls to **resize()**, an **ArrayDeque** supports the operations

- **get**$(\boldsymbol{i})$ and **set**$(\boldsymbol{i}, \boldsymbol{x})$ in $\boldsymbol{O}(\mathbf{1})$ time per operation; and
- **add**$(\boldsymbol{i}, \boldsymbol{x})$ and **remove**$(\boldsymbol{i})$ in $\boldsymbol{O}(\mathbf{1} + \mathbf{min}\{\boldsymbol{i}, \boldsymbol{n} - \boldsymbol{i}\})$ time per operation.

Furthermore, beginning with an empty **ArrayDeque**, performing any sequence of $\boldsymbol{m}$ **add**$(\boldsymbol{i}, \boldsymbol{x})$ and **remove**$(\boldsymbol{i})$ operations results in a total of $\boldsymbol{O}(\boldsymbol{m})$ time spent during all calls to **resize()**.