CECS 274: Data Structures
Array-Based Lists

Oscar Morales Ponce

California State University Long Beach

Example

- How to write a program to check if the parenthesis in the following expression are valid?

$$((a+b)*(c*d)+(a+(b*(c/d)))$$

- How to write a program to serve customers at a call center following the arrival order?

Array-Based Lists

- Implementations of the Stack, Queue, Deque and List interfaces using an array, called the *backing array*

	add(x) /remove()	add(i, x)/remove(i)
ArrayStack	$O(1)$	$O(n-i)$
ArrayQueue	$O(1)$	$O(\min(i, n-i))$
ArrayDeque	$O(1)$	$O(\min(i, n-i))$
ArrayList	N/A	$O(\min(i, n-i))$

Array

Suppose a is an array of 5 elements:

$a =$	23	43	13	65	0

Variable	Address (Hex)	Value
	0x00000000	
	0x0000FFFF	
	⋮	
a	0x01FFFFBB	23
	0x02FFFFBA	43
	0x03FFFFB9	13
	0x04FFFFB8	65
	0x05FFFFB7	0
n	0x06FFFFB6	4
	⋮	

Array

- a contains the reference $0x01FFFFBB$ to the memory address
  - $a[0]$ contains the value 23 at $0x01FFFFBB$
  - $a[1]$ contains the value 43 at $0x02FFFFBA$
  - $a[2]$ contains the value 13 at $0x03FFFFB9$
  - $a[3]$ contains the value 65 at $0x04FFFFB8$
  - $a[4]$ contains the value 0 at $0x05FFFFB7$
  - $a[5]$ will *crash* (stop the program) with an overflow exception
  - $a[-1]$ will *crash* with an underflow exception
- Accessing $a[i]$ takes $O(1)$ time.
- Arrays cannot expand or shrink

Methods

- Methods are used to access and modify the state of the object

```
method(parameters)
    Precondition:
    Effect:
```

- *Precondition*: The condition or *predicate* that must always be true prior to the execution. Otherwise, the effect becomes undefined
- *Effect*: Defines the steps that modify the state or access the state

ArrayStack: The Basics

- ArrayStack
  - State variables:
    - n: Number of elements in a. Initially set to 0.
    - a: Backing array where $a[i]$ stores the element i.
  - Invariant: At all times the length of a is no less than n and no greater than $3n$. Formally,

$$n \leq length(a) \leq 3n$$

b	r	e	d		
0	1	2	3	4	5

$n = 4$

$length(a) = 6$

ArrayStack: The Basics

- $initialize()$: Set all the state variables to the initial state
- $get(i)$: Return the value of the element i in the list
- $set(i, x)$: Set the value of the element i to x and returns the previous value

ArrayStack: The Basics

```
$\text{initialize}()$
    Allocate Memory to $a$
    Initialize the number of elements stored in $a$ to zero

$\text{get}(i)$
    return the element in $a$ at position $i$

$\text{set}(i, x)$
    keep in $y$ the element in $a$ at position $i$
    set the element in $a$ at position $i$
    return $y$
```

ArrayStack: The Basics

```
initialize()
    $a \leftarrow \text{new\_array}(1)$
    $n \leftarrow 0$

get($i$)
    Precondition: $0 \leq i < n$
    **return** $a[i]$

set($i, x$)
    Precondition: $0 \leq i < n$
    $y \leftarrow a[i]$
    $a[i] \leftarrow x$
    **return** $y$
```

ArrayStack: The Basics

```
initialize()
    $a \leftarrow \text{new\_array}(1)$
    $n \leftarrow 0$

get($i$)
    if $i < 0$ or $i \geq n$ then Exception
    return $a[i]$

set($i, x$)
    if $i < 0$ or $i \geq n$ then Exception
    $y \leftarrow a[i]$
    $a[i] \leftarrow x$
    return $y$
```

ArrayStack: Python

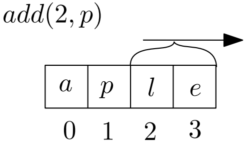
```
class ArrayStack(Stack):
    def __init__(self):
        self.a = self.new_array(1)
        self.n = 0

    def new_array(self, n: int):
        return numpy.zeros(n, numpy.object)

    def get(self, i : int) -> np.object:
        if i < 0 or i >= self.n:  raise IndexError()
        return self.a[i]

    def set(self, i : int, x : numpy.object) -> numpy.object:
        if i < 0 or i >= self.n:  raise IndexError()
        y = self.a[i]
        self.a[i] = x
        return y
```

ArrayStack: Java

```
public class ArrayStack implements Stack  {
    protected int n;
    protected Object[] a;

    public ArrayStack()  {
        n = 0;
        a = new_array(1);
    }

    public Object[] new_array(int n) {
        return new Object[n];
    }

    public Object get(int i) throws ArrayIndexOutOfBoundsException {
        if (i < 0 || i >= n)
                throw new ArrayIndexOutOfBoundsException();
        return  a[i];
    }

    public Object set(int i, Object x) throws ArrayIndexOutOfBoundsException  {
        if (i < 0 || i >= n)
            throw new ArrayIndexOutOfBoundsException();
        Object y = a[i];
        a[i]= x;
        return y;
    }
```

ArrayStack: $add(i, x)$

$add(i, x)$: Add space to insert x at position i.



$add(2, p)$

a	p	l	e
0	1	2	3

```
add(i, x)
    Check precondition: $0 \leq i \leq n$
    Check that the invariant holds
    Shift all the elements greater than $i$ one position to the right
    Set the value of $x$ to $a[i]$
    Increase $n$ by one
```

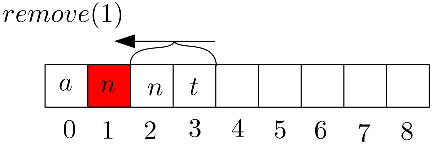
ArrayStack : $add(i, x)$

```
$\mathrm{add}(i, x)$
    if $i < 0$ or $i > n$ then Exception #Precondition
    if $n = \mathrm{length}(a)$ then $\mathrm{resize}()$ #Invariant
    $a[i+1, i+2, \ldots, n] \leftarrow a[i, i+1, \ldots, n-1]$ #Shift right
    $a[i] \leftarrow x$
    $n \leftarrow n + 1$
```

- How many operations?

ArrayStack: $remove(i)$

$remove(i)$: remove element i and recover space.



```
remove($i$)
    Check precondition: $0 \leq i < n$
    Let $x$ be the value of $a[i]$
    Shift all the elements greater than $i$ one position to the left
    Decrease $n$ by one
    Check that the invariant holds
    return $x$
```

ArrayStack: $remove(i)$

```
remove($i$)
    if $i < 0$ or $i \geq n$ then Exception #Precondition
    $x \leftarrow a[i]$
    $a[i, i+1, \ldots, n-2] \leftarrow a[i+1, i+2, \ldots, n-1]$ #Shift left
    $n \leftarrow n - 1$
    if $\mathrm{length}(a) \geq 3 \cdot n$ then $\mathrm{resize}()$ #Invariant
    return $x$
```

- How many operations?

ArrayStack: $resize()$

```
resize()
    Allocate a new array $b$ of length $2n$
    Copy all elements in $a$ to $b$
    Set the memory address of $a$ to $b$
```

Variable	Address (Hex)	Value
a	0x01FFFFBB	’a’
	0x02FFFFBA	’n’
b	0x1FFFFB800	’a’
	0x2FFFFB700	’n’
	0x2FFFFB700	
	0x2FFFFB700	

ArrayStack: Resize

```
resize()
    $b \leftarrow \mathrm{new\_array}(\max(1, 2 \cdot n))$ #cannot be empty
    $b[0, 1, \ldots, n-1] \leftarrow a[0, 1, \ldots, n-1]$ # Copy all elements of $a$ to $b$
    $a \leftarrow b$ # Set the memory address of $b$ to $a$
```

- How many operations?

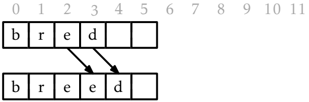
ArrayStack: Example

0	1	2	3	4	5	6	7	8	9	10	11
b	r	e	d								

`add(2,e)`

0 1 2 3 4 5 6 7 8 9 10 11

ArrayStack: Example



0	1	2	3	4	5	6	7	8	9	10	11
b	r	e	d								

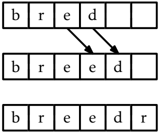
`add(2,e)`

b	r	e	e	d							

`add(5,r)`

0 1 2 3 4 5 6 7 8 9 10 11

ArrayStack: Example



0	1	2	3	4	5	6	7	8	9	10	11
b	r	e	d								
b	r	e	e	d							
b	r	e	e	d	r						

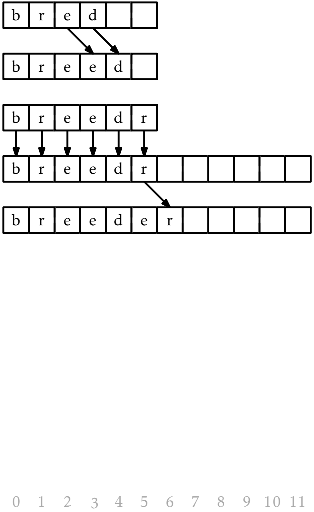
add(2,e)

add(5,r)

add(5,e)*

0 1 2 3 4 5 6 7 8 9 10 11

ArrayStack: Example



b r e d

add(2,e)

b r e e d

add(5,r)

b r e e d r

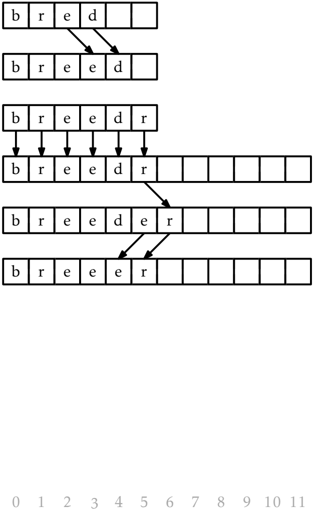
add(5,e)*

b r e e d r

b r e e d e r

remove(4)

ArrayStack: Example



add(2,e)

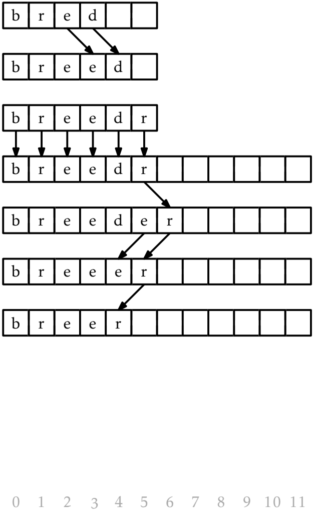
add(5,r)

add(5,e)*

remove(4)

remove(4)

ArrayStack: Example



add(2,e)

add(5,r)

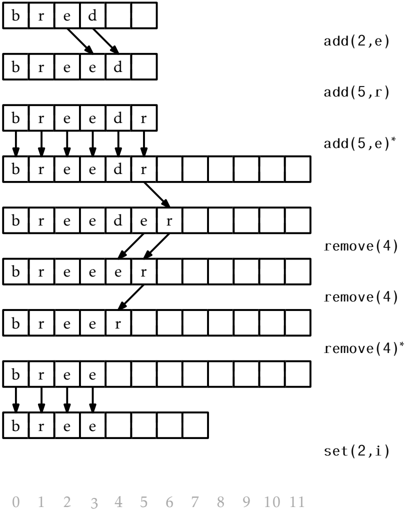
add(5,e)*

remove(4)

remove(4)

remove(4)*

ArrayStack: Example



add(2,e)

add(5,r)

add(5,e)*

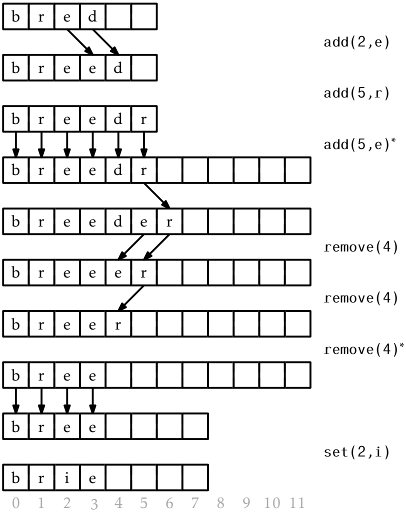
remove(4)

remove(4)

remove(4)*

set(2,i)

ArrayStack: Example



b	r	e	d									
												add(2,e)
b	r	e	e	d								
												add(5,r)
b	r	e	e	d	r							
												add(5,e)*
b	r	e	e	d	r							
b	r	e	e	d	e	r						
												remove(4)
b	r	e	e	e	r							
												remove(4)
b	r	e	e	r								
												remove(4)*
b	r	e	e									
b	r	e	e									
												set(2,i)
b	r	i	e									
0	1	2	3	4	5	6	7	8	9	10	11	

Amortized analysis

Lemma

If an empty ArrayStack is created and any sequence of $m \geq 1$ calls to $\mathrm{add}(i, x)$ and $\mathrm{remove}(i)$ are performed, then the total time spent during all calls to $\mathrm{resize}()$ is $O(m)$.

Proof. We claim that the number of calls between two $\mathrm{resize}()$ is at least $\frac{n}{2} - 1$.

Proof (1/3)

- Let n_i denote the value of n during the i-th call to $resize()$
- Let r denote the number of calls to $resize()$

$$\sum_{i=1}^{r}(\frac{n_i}{2} - 1) \leq m$$

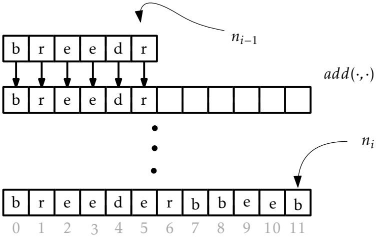
$$\sum_{i=1}^{r} n_i \leq 2m + 2r$$

- the total time spent during all calls to $resize()$ is

$$\sum_{i=1}^{r} O(n_i) \leq O(m + r) = O(m)$$

Proof (2/3): If $\mathrm{resize}()$ is being called by $\mathrm{add}(i, x)$.

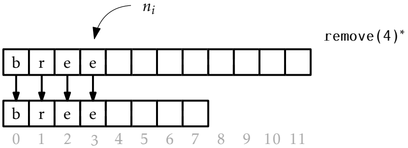
- Then, $\mathrm{length}(a) = n = n_i$ and there must have been at least $n_i/2$ calls to $\mathrm{add}(\cdot, x)$



Proof (3/3): If $\mathrm{resize}()$ is being called by $\mathrm{remove}(i)$

- After the previous call to resize, $n_{i-1} \geq \mathrm{length}(a)/2 - 1$.
- There are $n_i \leq \mathrm{length}(a)/3$ elements stored in a.
- Therefore, the number of operations between two consecutive resize is at least

$$\begin{aligned} R &\geq \mathrm{length}(a)/2 - 1 - \mathrm{length}(a)/3 \\ &= \mathrm{length}(a)(1/2 - 1/3) - 1 \\ &= \mathrm{length}(a)/6 - 1 \\ &\geq n_i/2 - 1 \end{aligned}$$



Theorem

Theorem

An ArrayStack implements the List interface. Ignoring the cost of calls to $\mathrm{resize}()$*,*

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

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

- The ArrayStack is an efficient way to implement a Stack.
  - $\mathrm{push}(x)$: $\mathrm{add}(n,x)$
  - $\mathrm{pop}()$: $\mathrm{remove}(n-1)$,

  these operations will run in $O(1)$ amortized time.

FastArrayStack: An Optimized ArrayStack

- Instead of using **for** loops we can use
  - In C: $\mathrm{memcpy}(d, s, n)$ and $\mathrm{memmove}(d, s, n)$ functions.
  - In C++: $\mathrm{stdcopy}(a_0, a_1, b)$ algorithm
  - In Java: $\mathrm{System.arraycopy}(s, i, d, j, n)$ method.

Infinite Array

- We could maintain one index j that keeps track of the next element to remove and n that counts the number of elements in the queue.
- The queue elements would always be stored in

$$a[j], a[j+1], \ldots, a[j+n-1]$$

ArrayQueue: Modular arithmetic

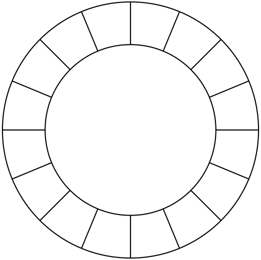
- An ArrayQueue simulates this by using a finite array a and *modular arithmetic*.
- We say that $10 + 5 = 15 \equiv 3 \pmod{12}$. "15 is congruent to 3 modulo 12"
- As binary operator $15 \bmod 12 = 3$. The operator in many programing languages is $\%$
- More generally, for an integer a and positive integer m, $a \bmod m$ is the unique integer $r \in \{0, \ldots, m-1\}$ such that $a = r + km$ for some integer k.
- The value r is the remainder we get when we divide a by m.

ArrayQueue: Simulating an Infinite Array

- Using modular arithmetic we can store the queue elements at array locations

$$a[j \bmod \mathrm{length}(a)], a[(j+1) \bmod \mathrm{length}(a)], \ldots$$

- This treats the array a like a *circular array*.



ArrayQueue: The Basics

- ArrayQueue
  - State variables:
    - n: Number of elements in a. Initially set to 0
    - j: Index of the first element in a. Initially set to 0
    - a: Backing array where $a[(i+j) \bmod length(a)]$ stores the element i.
  - Invariant: At all times the length of a is no less than n and no greater than $3n$. Formally,

$$n \leq length(a) \leq 3n$$

b	r	e	d		
0	1	2	3	4	5

$n = 4$
$length(a) = 6$

ArrayQueue: Initialize

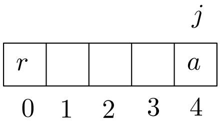
```
initialize()
    $a \leftarrow \mathrm{new\_array}(1)$
    $j \leftarrow 0$
    $n \leftarrow 0$
```

ArrayQueue: $add(x)$

$add(x)$: Add a to the tail of the queue.

$add(t)$

$n = 2$



r				a
0	1	2	3	4

```
$\mathrm{add}(x)$
    Check precondition
    Check that the invariant holds
    Set the value of $x$ to $a[(n + j) \bmod length(a)]$
    Increase $n$ by one
```

ArrayQueue: $add(x)$

```
add($x$)
    **if** $n > \text{length}(a)$ **then** resize()
    $a[(j+n) \bmod \text{length}(a)] \leftarrow x$
    $n \leftarrow n + 1$
    **return** $true$
```

- How many operations?

ArrayQueue: $remove()$ (Discussion Activity)

$remove()$: remove the element in the head

$remove()$

$n = 3$

				j
r	t			a
0	1	2	3	4

ArrayQueue: $remove()$

```
remove()
    if $0 \leq n$ then Exception #Precondition
    $x \leftarrow a[j]$
    $j \leftarrow (j+1) \bmod \text{length}(a)$
    $n \leftarrow n - 1$
    if $\text{length}(a) \geq 3 \cdot n$ then $\text{resize}()$
    return $x$
```

- How many operations?

ArrayQueue: $resize()$

```
resize()
    Allocate a new array $b$ of length $2n$
    Copy $a[(j+i) \bmod length(1)]$ to $b[i]$
    Set the memory address of $a$ to $b$
    Reset $j$
```

Variable	Address (Hex)	Value
a	0x01FFFFBB 0x02FFFFBA	 'a'
b	0x1FFFFB800 0x2FFFFB700	'a'

ArrayQueue: $resize()$

```
resize()
    $b \leftarrow \text{new\_array}(\max(1, 2 \cdot n))$
    **for** $k$ **in** $0, 1, 2, \ldots, n-1$ **do**
        $b[k] \leftarrow a[(j+k) \bmod \text{length}(a)]$
    $a \leftarrow b$
    $j \leftarrow 0$
```

- How many operations?

ArrayQueue: Example

$j = 2, n = 3$

0	1	2	3	4	5	6	7	8	9	10	11
		a	b	c							

add(d)

0 1 2 3 4 5 6 7 8 9 10 11

ArrayQueue: Example

	0	1	2	3	4	5	6	7	8	9	10	11
$j = 2, n = 3$			a	b	c							

add(d)

	0	1	2	3	4	5	6	7	8	9	10	11
$j = 2, n = 4$			a	b	c	d						

add(e)

0 1 2 3 4 5 6 7 8 9 10 11

ArrayQueue: Example

	0	1	2	3	4	5	6	7	8	9	10	11	
$j = 2, n = 3$			a	b	c								
													add(d)
$j = 2, n = 4$			a	b	c	d							
													add(e)
$j = 2, n = 5$	e		a	b	c	d							
													remove()

0 1 2 3 4 5 6 7 8 9 10 11

ArrayQueue: Example

	0	1	2	3	4	5	6	7	8	9	10	11	
$j = 2, n = 3$			a	b	c								
													add(d)
$j = 2, n = 4$			a	b	c	d							
													add(e)
$j = 2, n = 5$	e		a	b	c	d							
													remove()
$j = 3, n = 4$	e			b	c	d							
													add(f)

0 1 2 3 4 5 6 7 8 9 10 11

ArrayQueue: Example

	0	1	2	3	4	5	6	7	8	9	10	11	
$j = 2, n = 3$			a	b	c								`add(d)`
$j = 2, n = 4$			a	b	c	d							`add(e)`
$j = 2, n = 5$	e		a	b	c	d							`remove()`
$j = 3, n = 4$	e			b	c	d							`add(f)`
$j = 3, n = 5$	e	f		b	c	d							`add(g)`

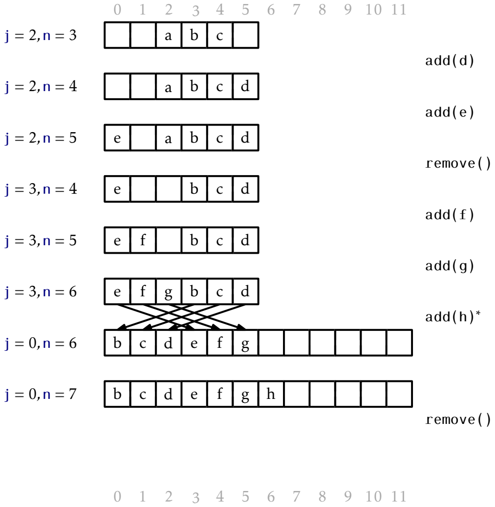
0 1 2 3 4 5 6 7 8 9 10 11

ArrayQueue: Example

	0	1	2	3	4	5	6	7	8	9	10	11	
$j = 2, n = 3$			a	b	c								add(d)
$j = 2, n = 4$			a	b	c	d							add(e)
$j = 2, n = 5$	e		a	b	c	d							remove()
$j = 3, n = 4$	e			b	c	d							add(f)
$j = 3, n = 5$	e	f		b	c	d							add(g)
$j = 3, n = 6$	e	f	g	b	c	d							add(h)*

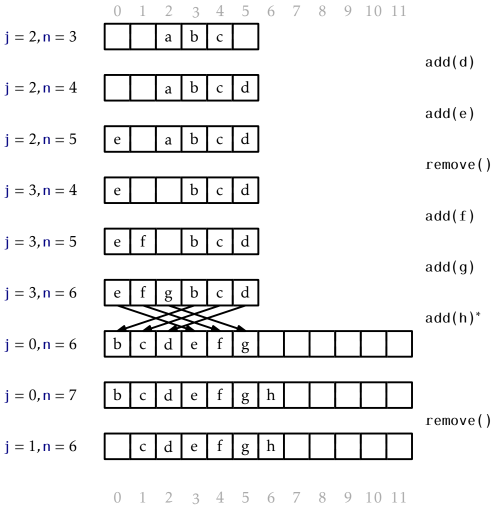
0 1 2 3 4 5 6 7 8 9 10 11

ArrayQueue: Example



	0	1	2	3	4	5	6	7	8	9	10	11	
$j = 2, n = 3$			a	b	c								
													add(d)
$j = 2, n = 4$			a	b	c	d							
													add(e)
$j = 2, n = 5$	e		a	b	c	d							
													remove()
$j = 3, n = 4$	e			b	c	d							
													add(f)
$j = 3, n = 5$	e	f		b	c	d							
													add(g)
$j = 3, n = 6$	e	f	g	b	c	d							
													add(h)*
$j = 0, n = 6$	b	c	d	e	f	g							
$j = 0, n = 7$	b	c	d	e	f	g	h						
													remove()
	0	1	2	3	4	5	6	7	8	9	10	11	

ArrayQueue: Example



	0	1	2	3	4	5	6	7	8	9	10	11	
$j = 2, n = 3$			a	b	c								
													add(d)
$j = 2, n = 4$			a	b	c	d							
													add(e)
$j = 2, n = 5$	e		a	b	c	d							
													remove()
$j = 3, n = 4$	e			b	c	d							
													add(f)
$j = 3, n = 5$	e	f		b	c	d							
													add(g)
$j = 3, n = 6$	e	f	g	b	c	d							
													add(h)*
$j = 0, n = 6$	b	c	d	e	f	g							
$j = 0, n = 7$	b	c	d	e	f	g	h						
													remove()
$j = 1, n = 6$		c	d	e	f	g	h						
	0	1	2	3	4	5	6	7	8	9	10	11	

ArrayQueue: Summary

Theorem

An ArrayQueue implements the (FIFO) Queue interface. Ignoring the cost of calls to $resize()$, an ArrayDeque supports the operations

- $add(x)$ *and* $remove()$ *in* $O(1)$ *time per operation.*

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

ArrayDeque: The Basics

- ArrayDeque
  - State variables:
    - n: Number of elements in a. Initially set to 0
    - j: Index of the first element in a. Initially set to 0
    - a: Backing array where $a[(i+j) \bmod length(a)]$ stores the element i.
  - Invariant: At all times the length of a is no less than n and no greater than $3n$. Formally,

$$n \leq length(a) \leq 3n$$

b	r	e	d		
0	1	2	3	4	5

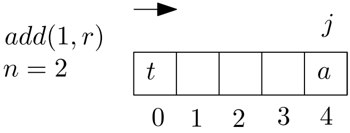
$n = 4$
$length(a) = 6$

ArrayDeque: $initialize()$, $get(i)$ and $set(i, x)$

```
initialize()
    $a \leftarrow \text{new\_array}(1)$
    $j \leftarrow 0$
    $n \leftarrow 0$

get($i$)
    return $a[(i + j) \bmod \text{length}(a)]$

set($i, x$)
    $y \leftarrow a[(i + j) \bmod \text{length}(a)]$
    $a[(i + j) \bmod \text{length}(a)] \leftarrow x$
    return $y$
```

ArrayDeque: $add(i, x)$

$add(i, x)$: Add space to insert x at position i.



```
add$(i, x)$
    Check precondition: $0 \leq i \leq n$
    Check that the invariant holds
    If $i \geq n/2$, shift elements greater than $i$ one position to the right
    If $i < n/2$ Shift all the elements less than $i$ one position to the left
    Set the value of $x$ to $a[i]$
    Increase $n$ by one
```

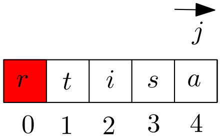
ArrayDeque: $add(i, x)$

```
add$(i, x)$
    **if** $i < 0$ **or** $i > n$ **then** Exception #Precondition
    **if** $n = \mathrm{length}(a)$ **then** resize()
    **if** $i < n/2$ **then**
        $j \leftarrow (j - 1) \bmod \mathrm{length}(a)$
        **for** $k$ **in** $0, 1, 2, \ldots, i - 1$ **do**
            $a[(j+k) \bmod \mathrm{length}(a)] \leftarrow a[(j+k+1) \bmod \mathrm{length}(a)]$
    **else**
        **for** $k$ **in** $n, n - 1, n - 2, \ldots, i + 1$ **do**
            $a[(j+k) \bmod \mathrm{length}(a)] \leftarrow a[(j+k-1) \bmod \mathrm{length}(a)]$
    $a[(j + i) \bmod \mathrm{length}(a)] \leftarrow x$
    $n \leftarrow n + 1$
```

ArrayDeque: $remove(i)$

$remove(i)$: remove the element i and recover space

$remove(1)$
$n = 5$



r	t	i	s	a
0	1	2	3	4

```
remove($i$)
   Check precondition: $0 < n$
   Let $x$ be the value of $a[(j + i) \bmod length(a)]$
   If $i \geq n/2$, shift all the elements greater than $i$ one position to the left
   If $i < n/2$, shift all the elements less than $i$ one position to the right
   Update $j$ accordingly
   Decrement $n$ by one
   Check that the invariant holds
   return $x$
```

ArrayQueue: Example

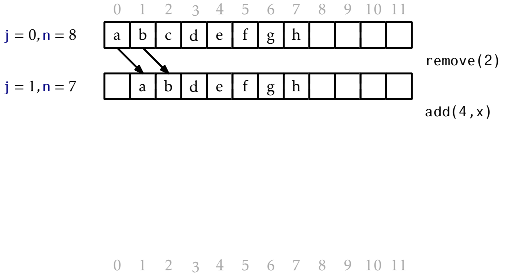
$j = 0, n = 8$

0	1	2	3	4	5	6	7	8	9	10	11
a	b	c	d	e	f	g	h				

`remove(2)`

0	1	2	3	4	5	6	7	8	9	10	11

ArrayQueue: Example



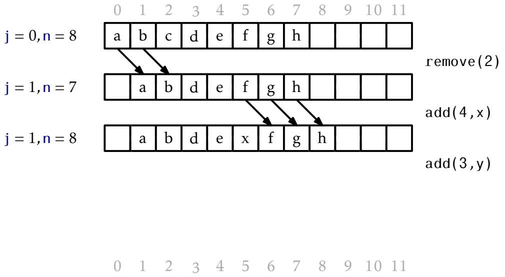
	0	1	2	3	4	5	6	7	8	9	10	11
$j = 0, n = 8$	a	b	c	d	e	f	g	h				
$j = 1, n = 7$		a	b	d	e	f	g	h				

remove(2)

add(4,x)

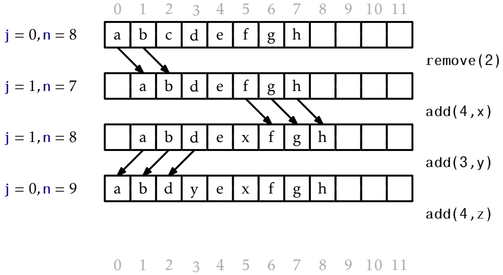
0 1 2 3 4 5 6 7 8 9 10 11

ArrayQueue: Example



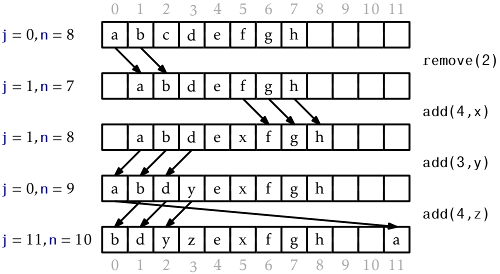
	0	1	2	3	4	5	6	7	8	9	10	11	
$j = 0, n = 8$	a	b	c	d	e	f	g	h					remove(2)
$j = 1, n = 7$		a	b	d	e	f	g	h					add(4,x)
$j = 1, n = 8$		a	b	d	e	x	f	g	h				add(3,y)
	0	1	2	3	4	5	6	7	8	9	10	11	

ArrayQueue: Example



	0	1	2	3	4	5	6	7	8	9	10	11
$j = 0, n = 8$	a	b	c	d	e	f	g	h				
$j = 1, n = 7$		a	b	d	e	f	g	h				
$j = 1, n = 8$		a	b	d	e	x	f	g	h			
$j = 0, n = 9$	a	b	d	y	e	x	f	g	h			

ArrayQueue: Example



	0	1	2	3	4	5	6	7	8	9	10	11	
$j = 0, n = 8$	a	b	c	d	e	f	g	h					`remove(2)`
$j = 1, n = 7$		a	b	d	e	f	g	h					`add(4,x)`
$j = 1, n = 8$		a	b	d	e	x	f	g	h				`add(3,y)`
$j = 0, n = 9$	a	b	d	y	e	x	f	g	h				`add(4,z)`
$j = 11, n = 10$	b	d	y	z	e	x	f	g	h			a	

ArrayDeque: Summary

Theorem

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

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

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

ArrayDeque: Summary

- The ArrayDeque is an efficient way to implement a Deque Interface
  - $add\_first(x)$ as $add(0, x)$
  - $add\_last(x)$ as $add(n, x)$
  - $remove\_first()$ as $remove(0)$
  - $remove\_last()$ as $remove(n-1)$

  these operations will run in $O(1)$ amortized time.

Example (Discussion Activity)

- How to write a program to check if the parenthesis in the following expression are valid?

$$((a+b)*(c*d)+(a+(b*(c/d)))$$

- How to write a program to serve customers at a call center following the arrival order?