Lecture 16

Stacks and Queues

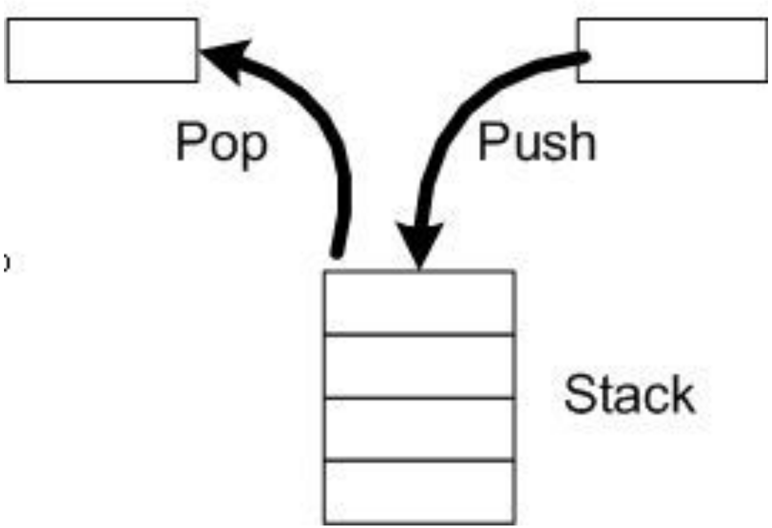
Stack

The Stack ADT (Abstract Data Type)

A **Stack** is a collection of objects inserted and removed according to the Last In First Out (LIFO) principle. Think of a stack of dishes.

Stack operations

Push and **Pop** are the two main operations



- When using push() operation to place the following items on a stack:

push(10)

push(20)

push(30)

push(0)

push(-30)

the output when popping from the stack is:

A: 10, 20, 30, 0 , -30

B: -30, 0, 10, 20, 30

C: 30, 10, 20, 0, -30

D: -30, 0, 30, 20, 10

E: 0, 30, -30, 10, 20

A lot of applications

- Think of the undo operation of an editor. The recent changes are pushed into a stack, and the undo operation pops it from the stack.
- Reverse strings
- The expression evaluation stacks are also used for parameter passing and local variable storage.
  - Think of ED diagrams and recursions!
- Check if a given expression has correct “(“ ,   “)” order.

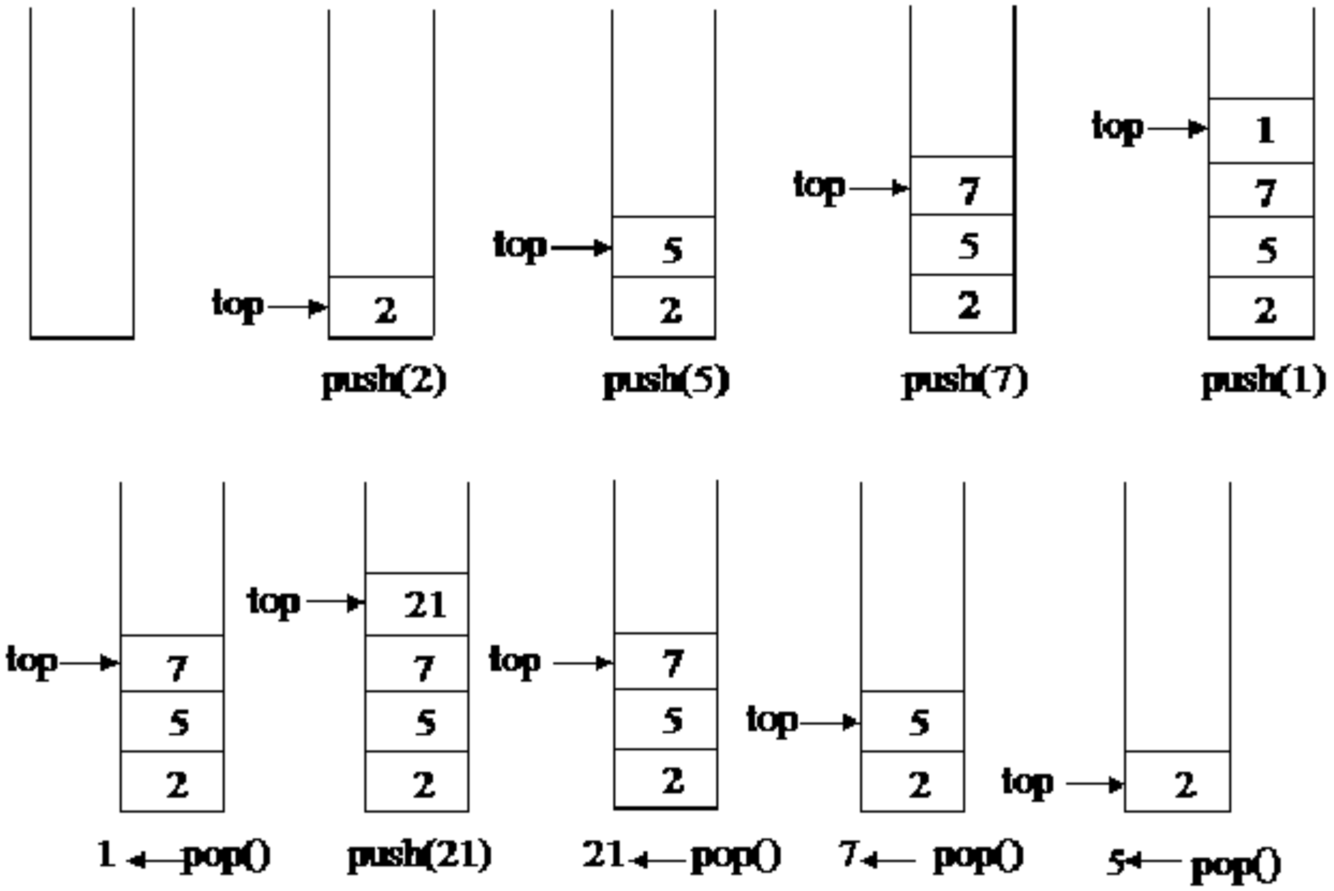
Implementation. Arrays

Main update methods:

- `Push(e)`
- `Pop( )`

Additional useful methods

- `Peek():` Same as pop, but does not remove the element
- `Empty():` Boolean, True when the stack is *empty*
- `Size():` Returns the size of the stack


top
2
push(2)
5
push(5)
7
push(7)
1
push(1)
1 pop()
21
push(21)
21 pop()
7 pop()
5 pop()

Implement class Stack using numpy array

```
import numpy as np

class Stack():
    """
    >>> stack = Stack()
    >>> stack.nelem
    0
    >>> stack.push(1)
    >>> stack.push(2)
    >>> stack
    2
    1
    >>> stack.pop()
    2
    >>> stack
    1
    >>> stack.pop()
    1
    >>> stack.pop() is None
    True
    """
```

Implement class Stack using numpy array

```
import numpy as np

class Stack():
    """
    >>> stack = Stack()
    >>> stack.nelem
    0
    >>> stack.push(1)
    >>> stack.push(2)
    >>> stack
    2
    1
    >>> stack.pop()
    2
    >>> stack
    1
    >>> stack.pop()
    1
    >>> stack.pop() is None
    True
    """
```

```
def __init__(self):

        self.items = np.empty(5, dtype = int)
        self.nelem = 0

def push(self, elem):
```

Implement class Stack using numpy array

```
import numpy as np

class Stack():
    """
    >>> stack = Stack()
    >>> stack.nelem
    0
    >>> stack.push(1)
    >>> stack.push(2)
    >>> stack
    2
    1
    >>> stack.pop()
    2
    >>> stack
    1
    >>> stack.pop()
    1
    >>> stack.pop() is None
    True
    """
```

```
def __init__(self):

        self.items = np.empty(5, dtype = int)
        self.nelem = 0

def push(self, elem):
         self.items[self.nelem] = elem
        self.nelem = self.nelem + 1
```

Implement class Stack using numpy array

```python
import numpy as np

class Stack():
    """
    >>> stack = Stack()
    >>> stack.nelem
    0
    >>> stack.push(1)
    >>> stack.push(2)
    >>> stack
    2
    1
    >>> stack.pop()
    2
    >>> stack
    1
    >>> stack.pop()
    1
    >>> stack.pop() is None
    True
    """
```

```python
def __init__(self):

        self.items = np.empty(5, dtype = int)
        self.nelem = 0

def push(self, elem):
        self.items[self.nelem] = elem
        # Be careful here
        self.nelem = self.nelem + 1
```

Implement class Stack using numpy array

```
import numpy as np

class Stack():
    """
    >>> stack = Stack()
    >>> stack.nelem
    0
    >>> stack.push(1)
    >>> stack.push(2)
    >>> stack
    2
    1
    >>> stack.pop()
    2
    >>> stack
    1
    >>> stack.pop()
    1
    >>> stack.pop() is None
    True
    """
```

```
def __init__(self):

        self.items = np.empty(5, dtype = int)
        self.nelem = 0

def push(self, elem):
        self.items[self.nelem] = elem
        # Be careful here
        self.nelem = self.nelem + 1


def pop(self):
```

Implement class Stack using numpy array

```python
import numpy as np

class Stack():
    """
    >>> stack = Stack()
    >>> stack.nelem
    0
    >>> stack.push(1)
    >>> stack.push(2)
    >>> stack
    2
    1
    >>> stack.pop()
    2
    >>> stack
    1
    >>> stack.pop()
    1
    >>> stack.pop() is None
    True
    """
```

```python
def __init__(self):

        self.items = np.empty(5, dtype = int)
        self.nelem = 0

def push(self, elem):
         self.items[self.nelem] = elem
         # Be careful here
         self.nelem = self.nelem + 1


def pop(self):
     if self.nelem == 0:
             return None
         else:
             value = self.items[self.nelem - 1]
             self.nelem = self.nelem - 1
             return value
```

Implement class Stack using numpy array

```
import numpy as np

class Stack():
    """
    >>> stack = Stack()
    >>> stack.nelem
    0
    >>> stack.push(1)
    >>> stack.push(2)
    >>> stack
    2
    1
    >>> stack.pop()
    2
    >>> stack
    1
    >>> stack.pop()
    1
    >>> stack.pop() is None
    True
    """
```

```
def __init__(self):

        self.items = np.empty(5, dtype = int)
        self.nelem = 0

def push(self, elem):
         self.items[self.nelem] = elem
         # Be careful here
         self.nelem = self.nelem + 1


def pop(self):
     if self.nelem == 0:
           return None
         else:
             value = self.items[self.nelem - 1]
             self.nelem = self.nelem - 1
             return value
def __repr__(self):
```

Advantage and Limitation

- **Advantages of Array-based Implementation Fast:**

  all operations are completed in one step. No loops are needed.

- **Limitations of Array-based Implementation:**

You have to know the upper bound of growth and allocate memory accordingly. If the array is full and there is another *push* operation then you encounter an exception (error).

Implementation: Linked List

- Do not have to worry about the size when the stack grows.
- Sky (i.e. the entire memory pool) is the limit :)
- Also can be implemented **fast**. No for loops needed!

Available Linked List methods

- `insert_front(lst, elem)`
- `insert_last(lst, elem)`
- `delete_last(lst)`
- `delete_first(lst)`
- `size(lst)`

Available Linked List methods. Need a loop to implement?

1. `insert_front(lst, elem)`
2. `insert_last(lst, elem)`
3. `delete_last(lst)`
4. `delete_first(lst)`
5. `size(lst)`

Yes for:

A: None of the below
B: 1, 3, 5
C: 2, 4
D: 1, 4
E: 2, 3, 5

Available Linked List methods. Best for stack?

```
1.  insert_front(lst, elem)
2.  insert_last(lst, elem)
3.  delete_last(lst)
4.  delete_first(lst)
5.  size(lst)
```

	Pop:	Push:
A:	4	1
B:	4	2
C:	3	1
D:	3	2
E:	Does not matter	

Announcements

- HW9 is longer compared to the prev. ones
- Start early!

Testing your code

Do you know that you need to test your code?

A: Yes

B: No

Testing your code

Do you know how to write doctests to test your code?

A: Yes

B: No

Testing your code

Do you know that you are required to write your own doctests for the homework?

A: Yes

B: No

Testing your code

How do you code?

A: Code first, then doctests, then submit

B: Doctests first, then code, then submit

C: Code first, IF THERE IS TIME doctests, submit

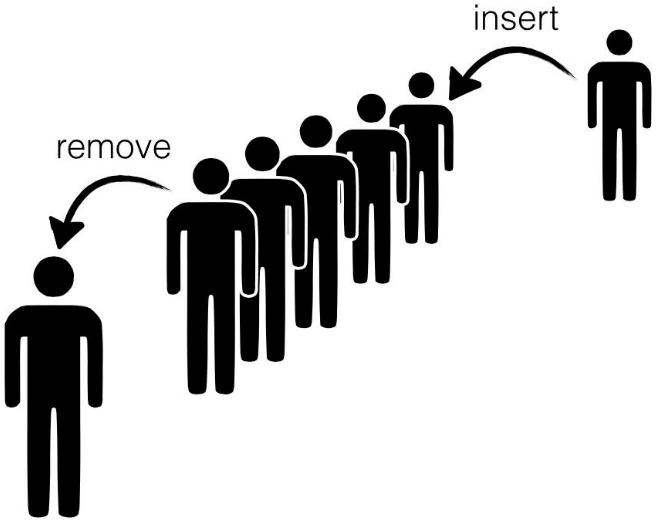
D: Code, submit (just use our doctests for correctness)

E: Other

Queues

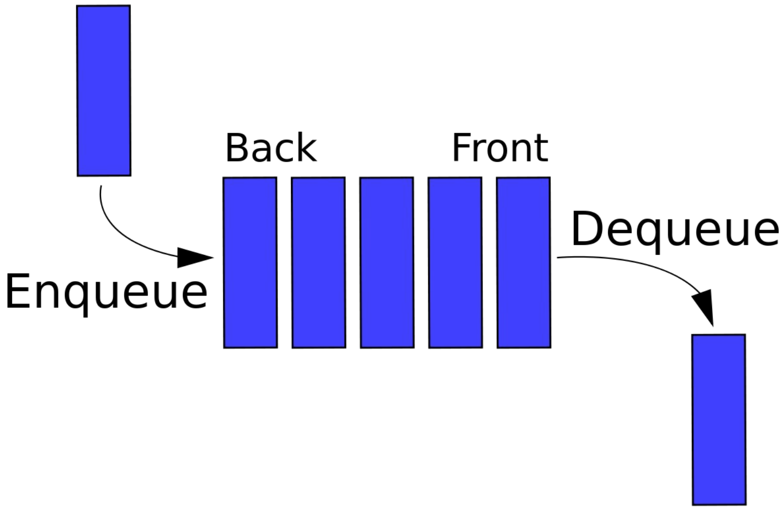
The Queue ADT

A **Queue** is a collection of objects inserted and removed according to the First In First Out (FIFO) principle. Think of a queue of people to Rubios.



Queue operations

Enqueue (insert) and **Dequeue (remove)** are the two main operations



Question

When using enqueue operation to place the following items in a queue:

enqueue(10)

enqueue(20)

enqueue(30)

enqueue(0)

enqueue(-30)

The output when dequeuing from the queue is:

A: 10, 20, 30, 0 , -30

B: -30, 0, 10, 20, 30

C: 30, 10, 20, 0, -30

D: -30, 0, 30, 20, 10

E: 0, 30, -30, 10, 20

Implementation. Linked Lists and Arrays

Main update methods:

- `Enqueue(e)`
- `Dequeue( )`

Additional useful methods

- `Peek():` Same as dequeue, but does not remove the element
- `Empty():` Boolean, True when the queue is *empty*
- `Size():` Returns the size of the queue

Implementation:

1. `insert_front(lst, elem)`
2. `insert_last(lst, elem)`
3. `delete_last(lst)`
4. `delete_first(lst)`
5. `size(lst)`

`dequeue` returns both:

- deleted element
- changed queue

```
def enqueue(q, elem):
    return____


def dequeue(q):
    return____
```

Implementation: Not efficient! $\Theta(n)$

1. `insert_front(lst, elem)`
2. `insert_last(lst, elem)`
3. `delete_last(lst)`
4. `delete_first(lst)`
5. `size(lst)`

`dequeue` returns both:

- deleted element
- changed queue

```
def enqueue(q, elem):
    return insert_last(q, elem)


def dequeue(q):
    return first(q),delete_first(q)
```

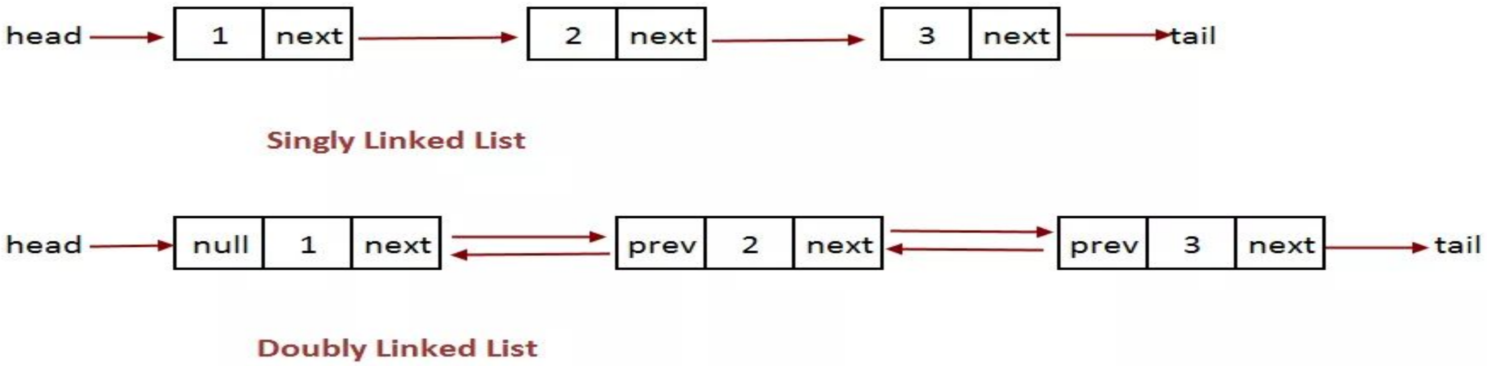
Circular array

Complexity for enqueue and dequeue

Efficient implementation of Queue ADT using either

- Array
- Linked Lists (Doubly Linked Lists)

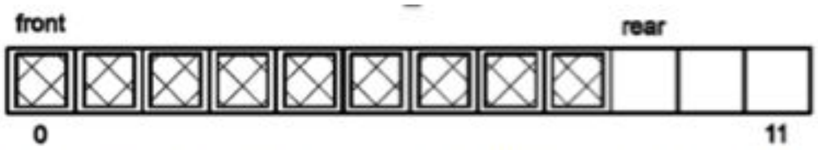
Assumes $\Theta(1)$ for both: enqueue and dequeue.



Regular array: dequeue

```python
def dequeue():
    # potential issue if empty
    # for now, assume not empty

    elem = array[front]
    # You code is here #
    return elem
```



(a) Queue.*front* is always at 0 – shift elements *left* on dequeue().

Select the correct code to delete from below:

A:

```python
front = front + 1
```

B:

```python
rear = rear - 1
```

C:

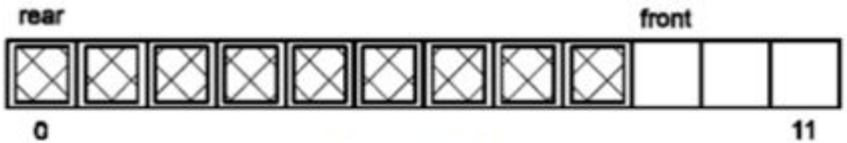
```python
for i in range(rear):
    array[i] = array[i+1]

rear = rear - 1
```

D: None of these are correct

Regular array: enqueue

```python
def enqueue(elem):
    # potential issue if full
    # for now, assume not empty
    # Your code is here #
    front = front + 1
```



(b) Queue.*rear* is always at 0 – shift elements *right* on enqueue().

Select the correct code to insert from below:

A:

```python
array[0] = elem
```

B:

```python
array[front] = elem
```

C:

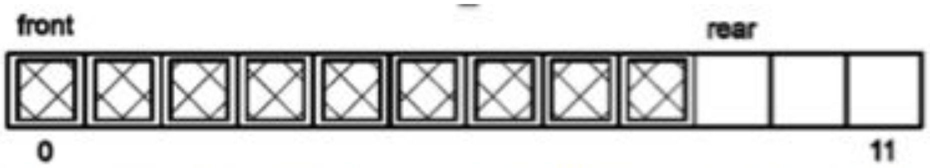
```python
for i in range(front):
    array[i+1] = array[i]

array[front] = elem
```

D: None of these are correct

Issues

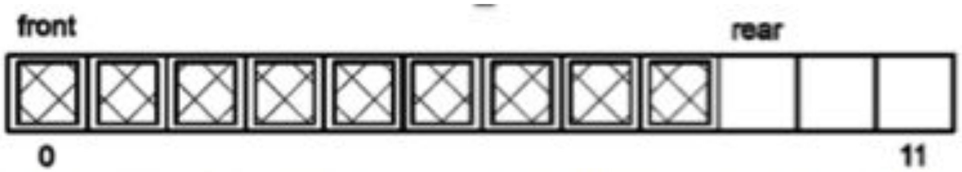
Dequeue:



(a) Queue.*front* is always at 0 – shift elements *left* on dequeue().

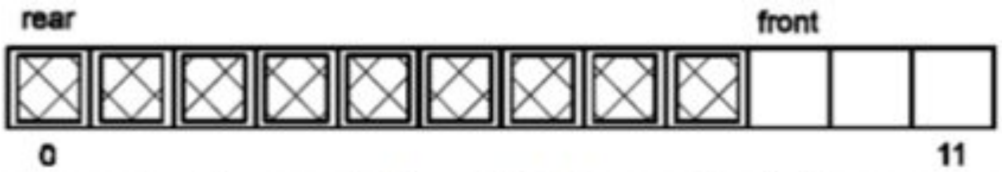
Issues

Dequeue:



(a) Queue.*front* is always at 0 – shift elements *left* on dequeue().

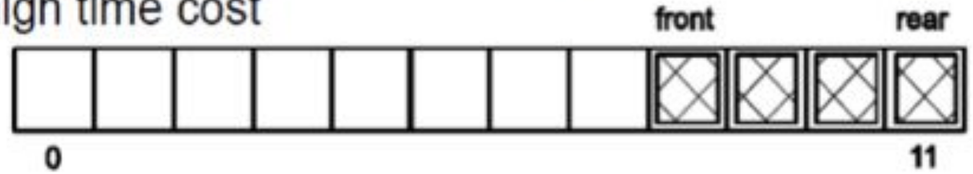
Enqueue:



(b) Queue.*rear* is always at 0 – shift elements *right* on enqueue().

Regular Array

- Neither of those solutions is very good as they both involve *moving all the existing* data elements, which has high time cost

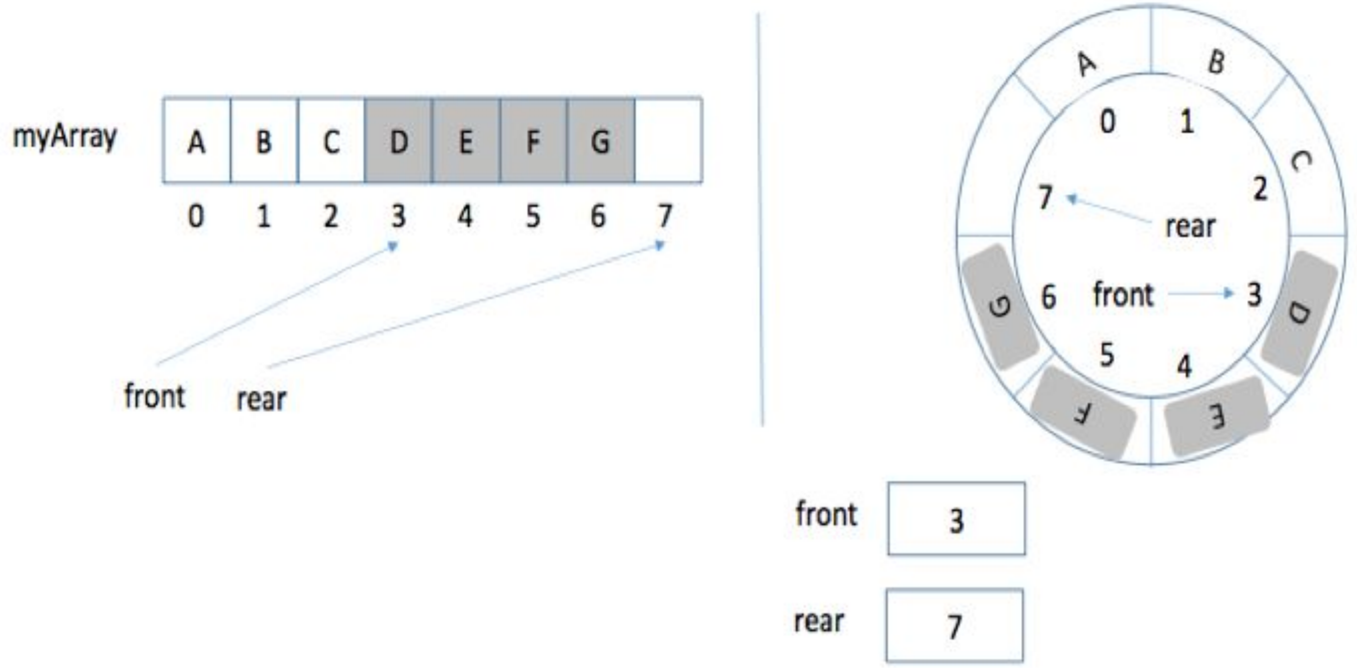


- Idea: Instead of moving data elements to a fixed position for *front* when removing, let *front* advance through the array

Hmmm….what do we do when we now add an element to that queue at the rear? What happens when we remove several elements, and *front* catches up with *rear*?...

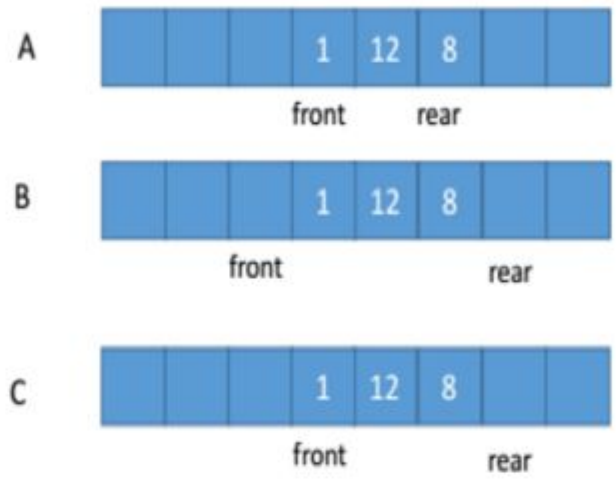
Making a linear array appear circular

front==head
rear == tail



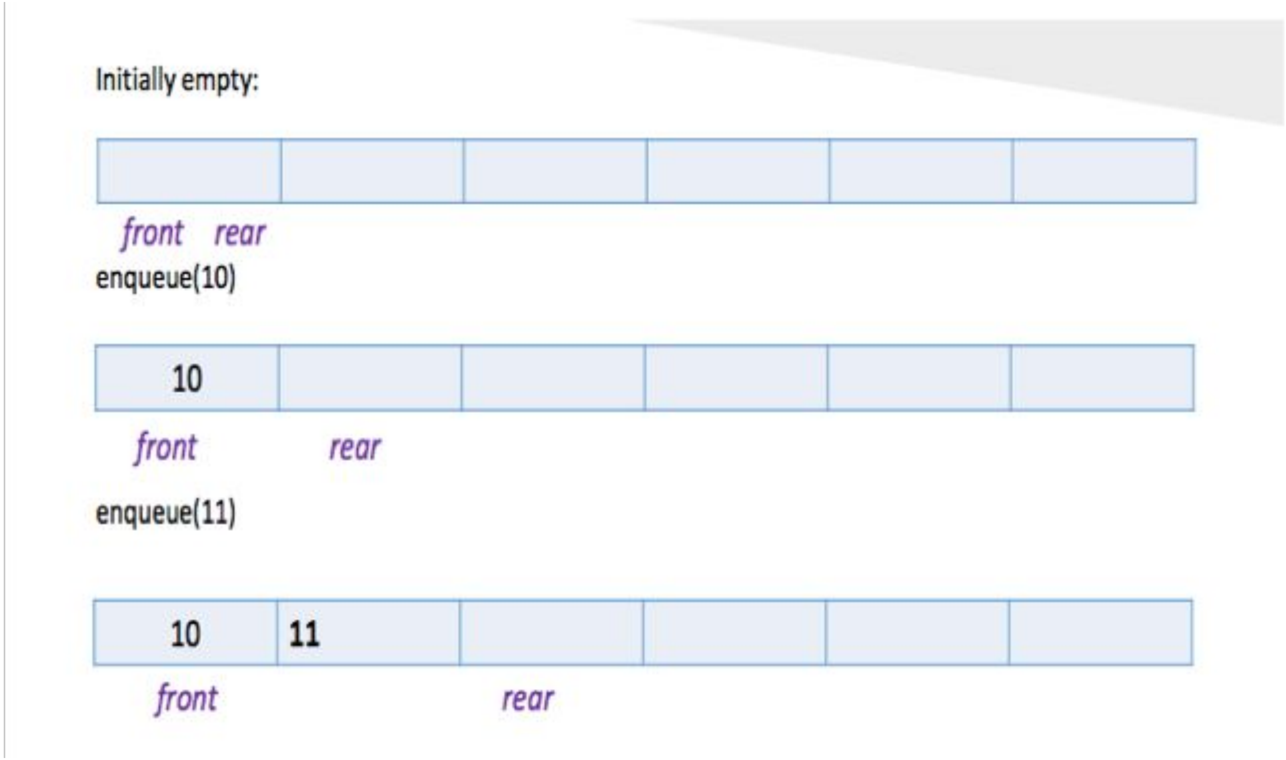
Design decisions: Where do front and rear point?

Which of these choices will work?



D Any of these could work

Queues using circular arrays

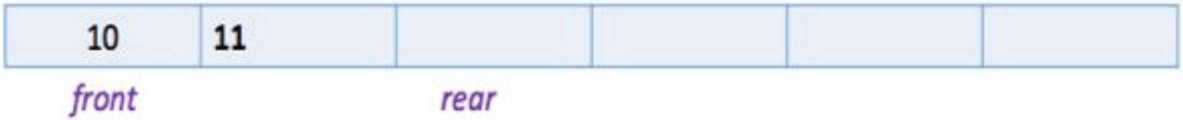


Queues using circular arrays

Initially empty:

enqueue(10)

enqueue(11)



What should be the value *of* front after the next dequeue?

A. 0 B. 1 C. 2 D. 5

Queues using circular arrays

Initially empty:

front *rear*

enqueue(10)

10					

front *rear*

enqueue(11)

10	11				

front *rear*

What should be the value stored at arr[0] after the next dequeue?

A. 10 B. 0 C. null D. It doesn't matter

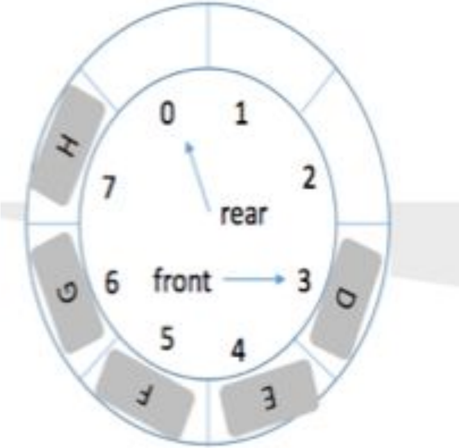
Queues using circular arrays

10 or null	11 or null	12	8	3	
		front			*rear*

enqueue(20)

What is the value of rear after this enqueue?

A. 5
B. 0
C. 1
D. 2
E. Other




```python
def dequeue():
    size = size - 1
    elem = array[front]
    # Your code #
    return elem
```

Select the correct code to insert from below:

A:

```python
front = front + 1
if (front == len(array)):
    front = 0
```

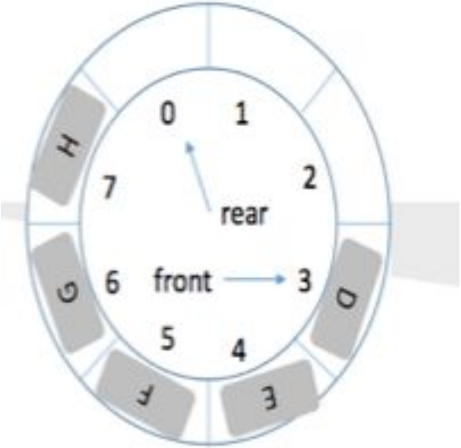
B:

```python
rear = rear - 1
if (rear < 0):
    rear = len(array) - 1
```

C:

```python
for in range(rear):
    array[i] = array[i+1]
rear = rear - 1
if (rear < 0):
    rear = len(array) - 1
```

D: None of these are correct




```
def enqueue(elem):
    #Your code is here#
    size = size + 1
```

Select the correct code to insert from below:

A:

```
rear = rear + 1
if (rear == len(array)):
    rear = 0
array[rear] = elem
```

B:

```
array[rear] = elem
rear = rear + 1
```

C:

```
for i in range(rear):
    array[i] = array[i+1]
array[rear] = elem
front = front - 1
```

D: None of these are correct