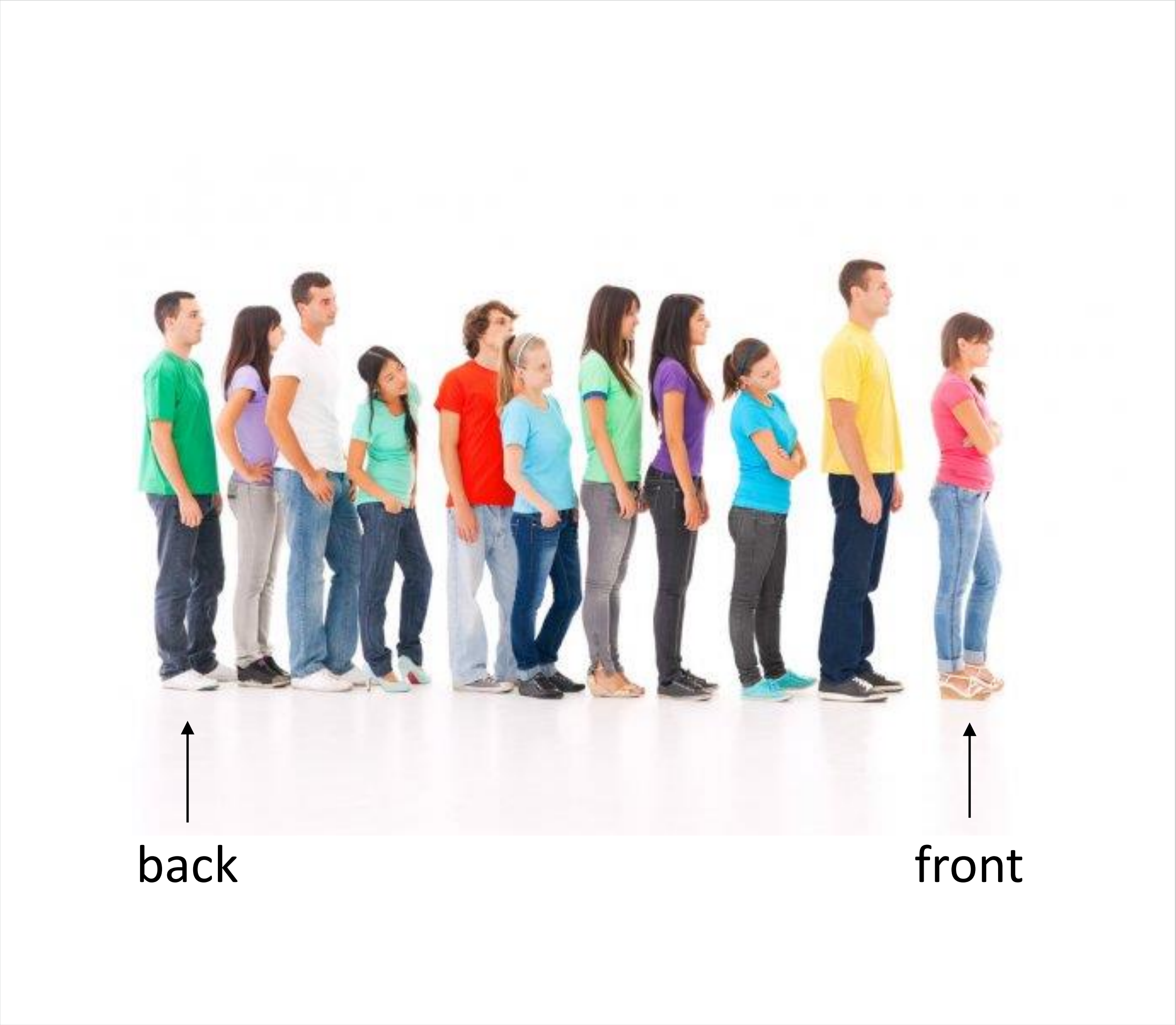
Queue

Another special kind of List

What is a Queue?

- The Queue is a List where all removals and retrievals take place at one end (front), and all insertions take place at the other (back).

- The Queue behaves in a First-In, First-Out (FIFO) manner.



back

front

Primary methods (IQueue)

```
// Add an item to the back of a queue (Insert)
public void Enqueue(T item)

// Remove an item from the front of a queue (Remove)
public void Dequeue()

// Return the front item of a queue (Retrieve)
public T Front()
```

Unlike List, position is NOT passed as a parameter

Supporting methods (IContainer)

```
// Resets the queue to empty
public void MakeEmpty()

// Returns true if the queue is empty; false otherwise
public bool Empty()

Returns the number of items in the queue
public int Size()
```

Generic Queue class

```
class Queue<T> : IQueue<T>, IContainer<T>
{
    ...
}
```

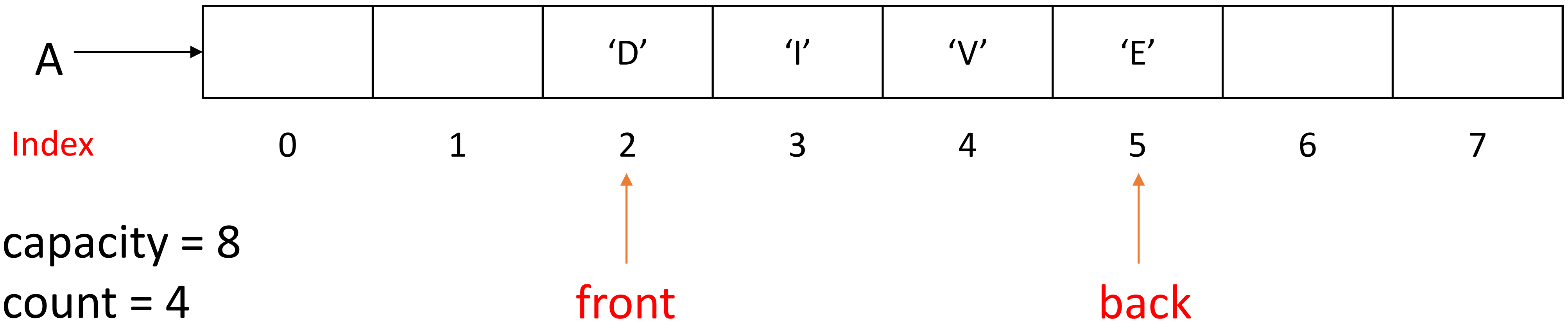
Note: In the code, IQueue inherits IContainer.

Data structures

- Circular array ←
- Singly linked list

Circular array

```
private T[] A;              // Linear array of items (Generic)
private int front, back;    // Indices of the front and back items
private int capacity;       // Maximum capacity of the queue
private int count;          // Actual number of items in the queue
```



A			'D'	'I'	'V'	'E'		
Index	0	1	2	3	4	5	6	7
			front			back		

capacity = 8

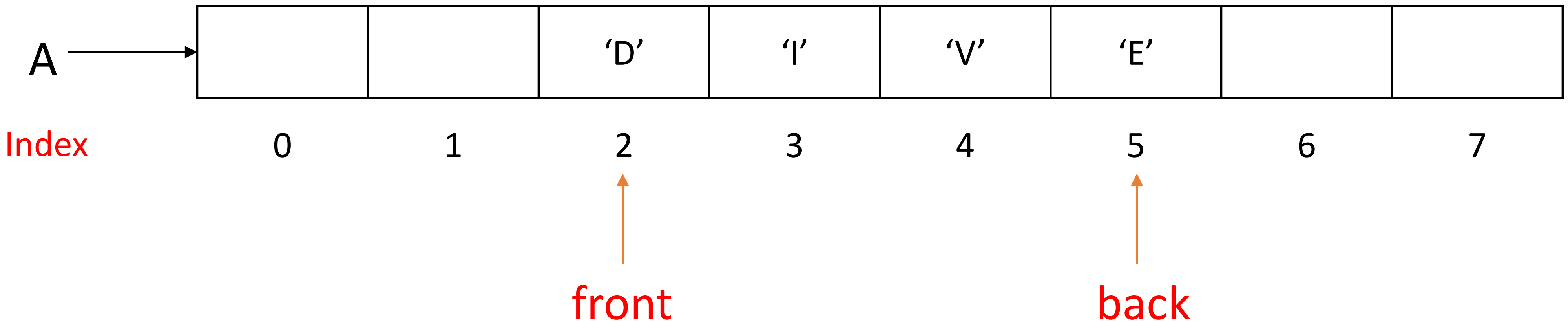
count = 4

Why circular?

- The queue below could be arrived at by enqueuing the characters ‘E’, ‘N’, ‘D’, ‘I’, ‘V’ and ‘E’ and then dequeuing twice (i.e., removing in turn the two front items of the queue).

capacity = 8

count = 4



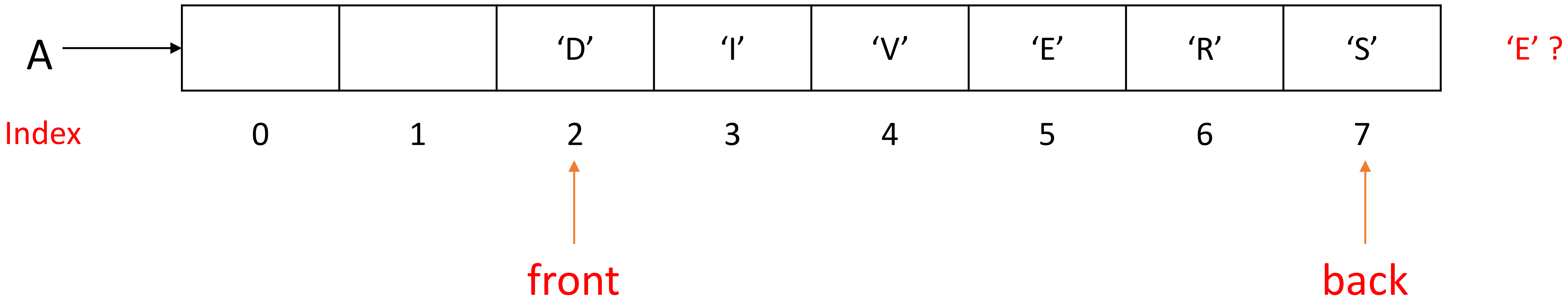
A			‘D’	‘I’	‘V’	‘E’		
Index	0	1	2	3	4	5	6	7
			front			back		

Why circular?

- When we try to enqueue three additional characters ‘R’, ‘S’ and ‘E’, we run into the following scenario.

capacity = 8

count = 6



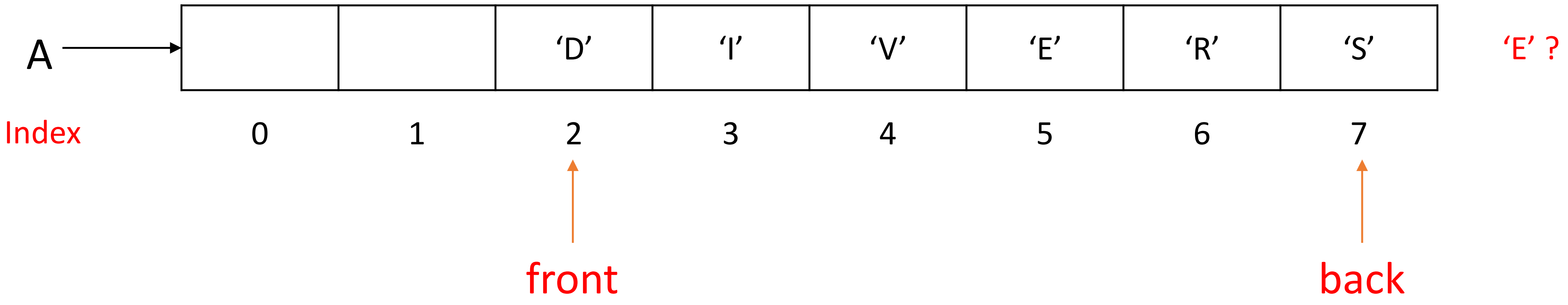
A			‘D’	‘I’	‘V’	‘E’	‘R’	‘S’	‘E’ ?
Index	0	1	2	3	4	5	6	7	
			front					back	

Why circular?

- The queue still has available capacity, but back has reached the end of the array. What do we do?

capacity = 8

count = 6



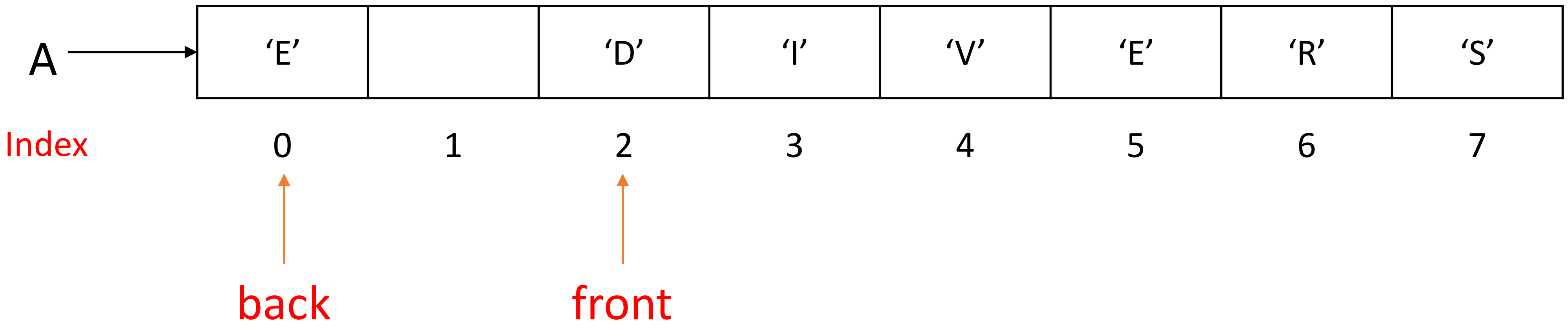
A			‘D’	‘I’	‘V’	‘E’	‘R’	‘S’	‘E’ ?
Index	0	1	2	3	4	5	6	7	

Why circular?

- Answer: Wrap back around to the beginning of the array and place ‘E’ at index 0.

capacity = 8

count = 7



A	‘E’		‘D’	‘I’	‘V’	‘E’	‘R’	‘S’
Index	0	1	2	3	4	5	6	7
	back		front					

Additional observations

- The front “chases” the back and so, needs to wrap around as well.

- To increment an index i, such as back or front, with wraparound, the following statement is used:

```
i = (i + 1) % capacity;
```

- Note that % is the modulus operator which returns the remainder of integer division. For example, 23 % 7 = 2, 6 % 7 = 6, and 7 % 7 = 0.

Constructor

- Basic Strategy
    - Create an array with a default capacity of 8.
    - Set count and front to 0.
    - Set back to capacity – 1 (end of the array). Why?

Implementation

```
// Constructor

public Queue()
 {
     A = new T[8];
     capacity = 8;
     MakeEmpty();
 }
 ...
```

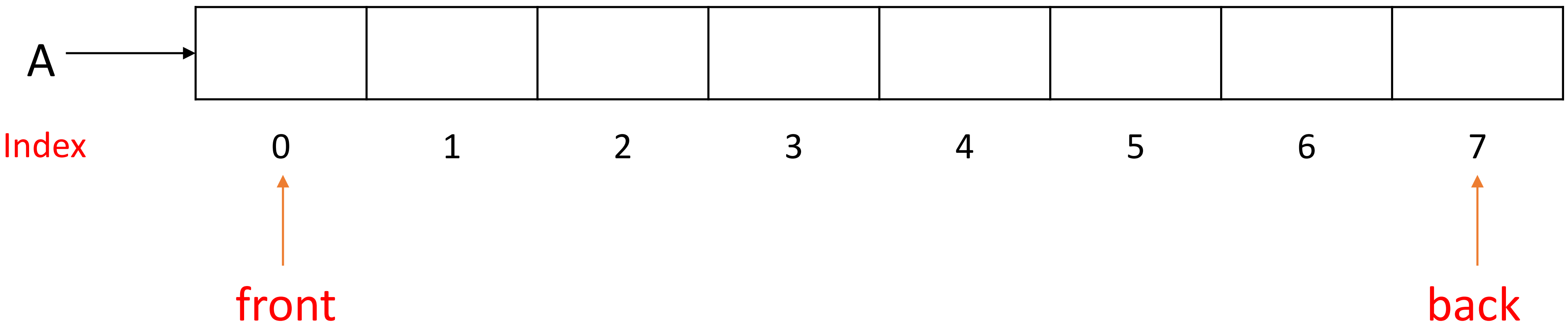
```
public void MakeEmpty()
 {
     front = 0;
     count = 0;
     back = capacity - 1;
 }
```

Initial configuration of an empty queue

Q = new Queue<int>();

capacity = 8

count = 0



A

Index

0 1 2 3 4 5 6 7

front

back

Enqueue an item to the back of a Queue

- Basic Strategy
  - If the queue is full (count == capacity) then double capacity
  - Increment back with wraparound.
  - Place item at A[back].
  - Increase count by 1.

Implementation

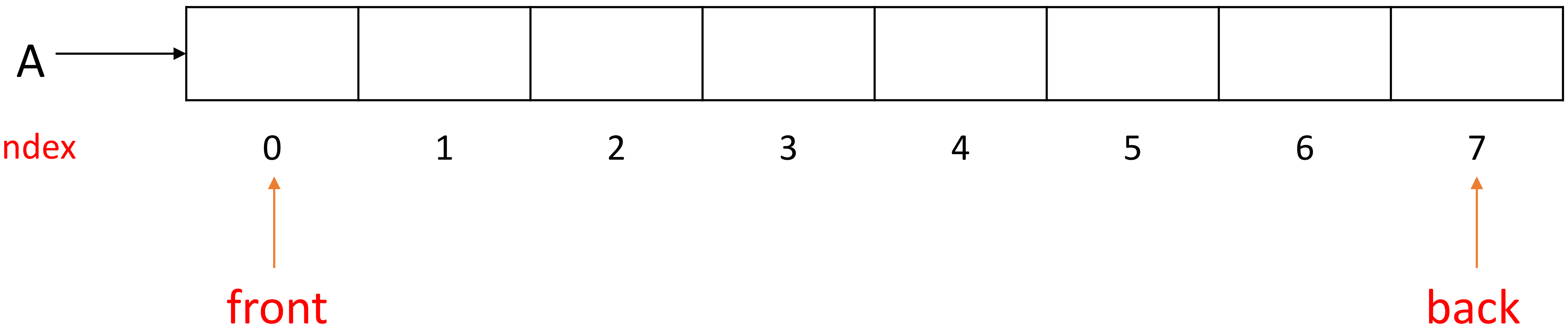
```
public void Enqueue(T item)
{
    if (count == capacity)
    {
        DoubleCapacity();
    }
    back = (back + 1) % capacity;    // Increment back with wraparound
    A[back] = item;
    count++;
}
```

Study the code for DoubleCapacity
How does it differ from DoubleCapacity in Stack?

Enqueue ‘E’ into an empty Queue

Before

capacity = 8
count = 0



A

Index 0 1 2 3 4 5 6 7

front

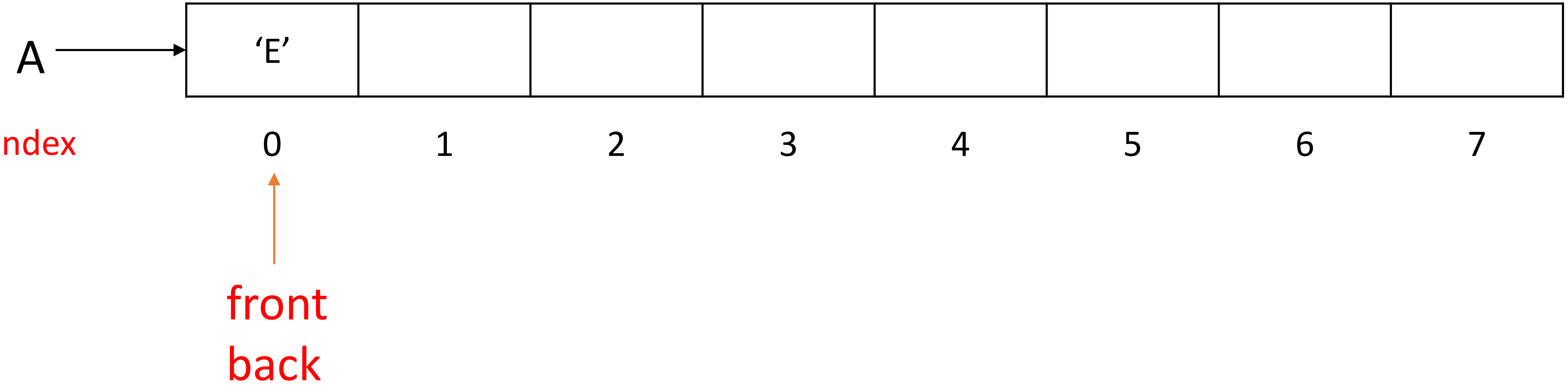
back

Enqueue ‘E’ into an empty Queue

After

capacity = 8

count = 1



A								
	‘E’							
Index	0	1	2	3	4	5	6	7

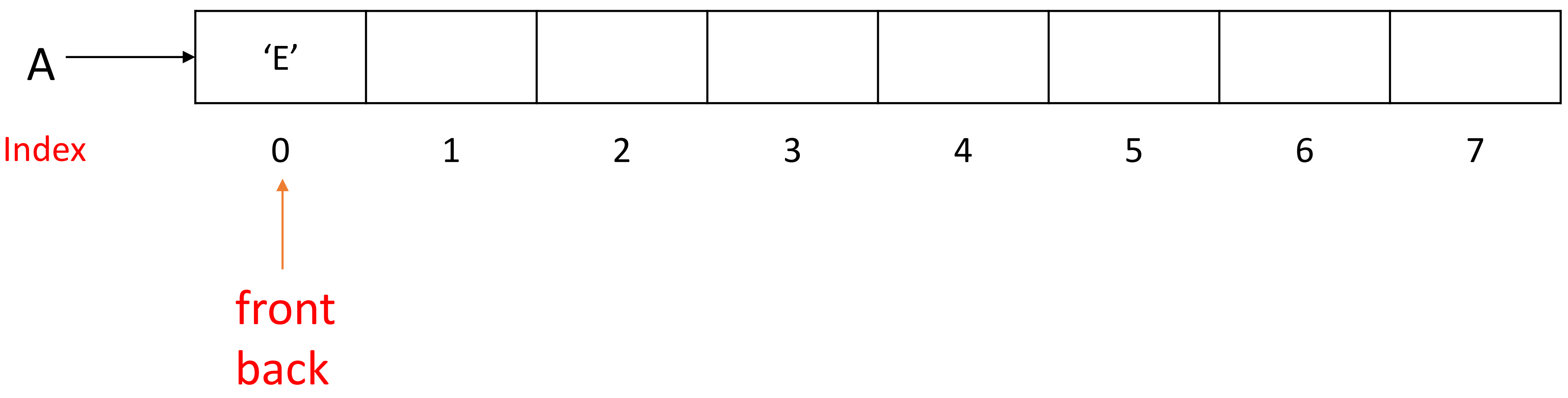
front

back

Enqueue ‘N’ onto the Queue

Before

capacity = 8
count = 1



A

‘E’							

Index	0	1	2	3	4	5	6	7

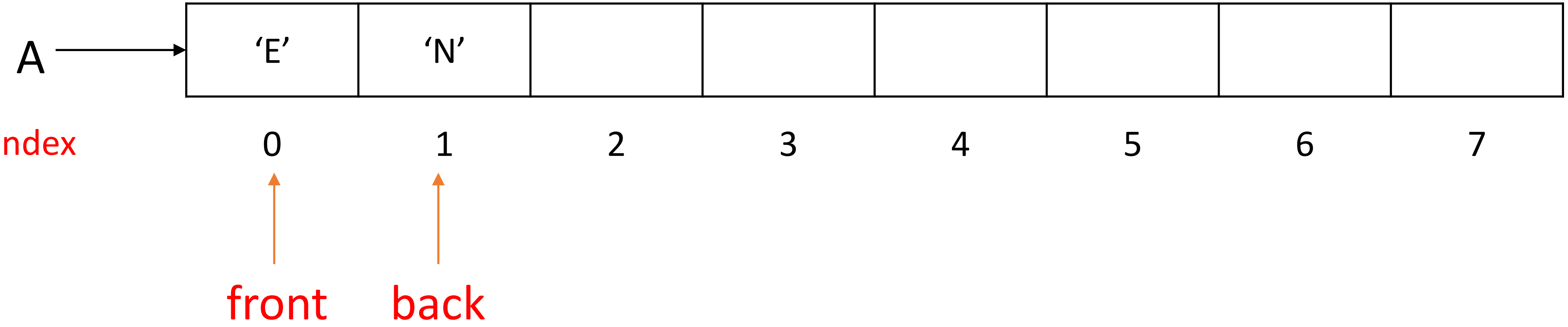
front
back

Enqueue ‘N’ onto the Queue

After

capacity = 8

count = 2



A	‘E’	‘N’						
Index	0	1	2	3	4	5	6	7
	front	back						

Dequeue an item from the front of a Queue

- Basic strategy
    - If the queue is empty (count = 0) then throw an exception
    - Else
        - Increment front with wraparound.
        - Decrease count by 1.

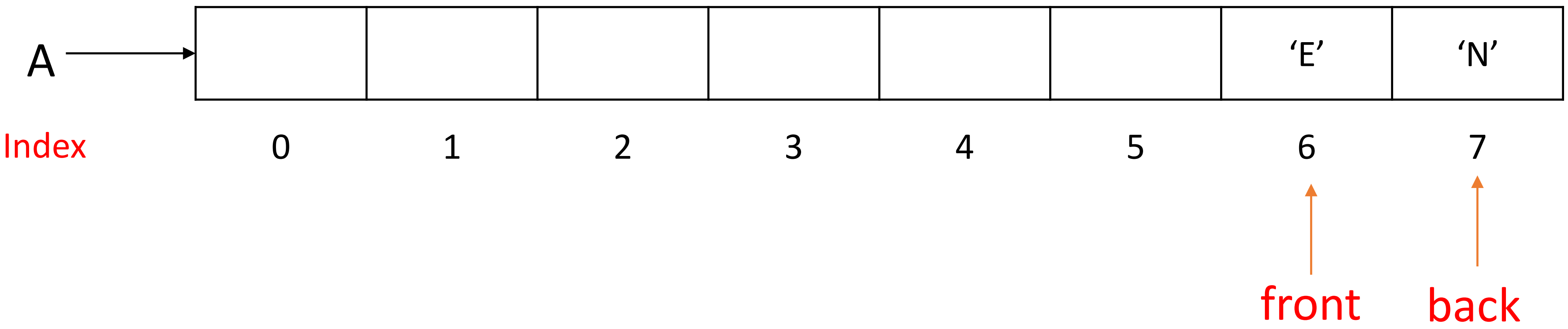
Implementation

```
public void Dequeue()
{
    if (Empty())
    {
        throw new InvalidOperationException("Queue is empty");
    }
    else
    {
        front = (front + 1) % capacity;  // Increment front with wraparound
        count--;
    }
}
```

Dequeue ‘E’ from the Queue

Before

capacity = 8
count = 2



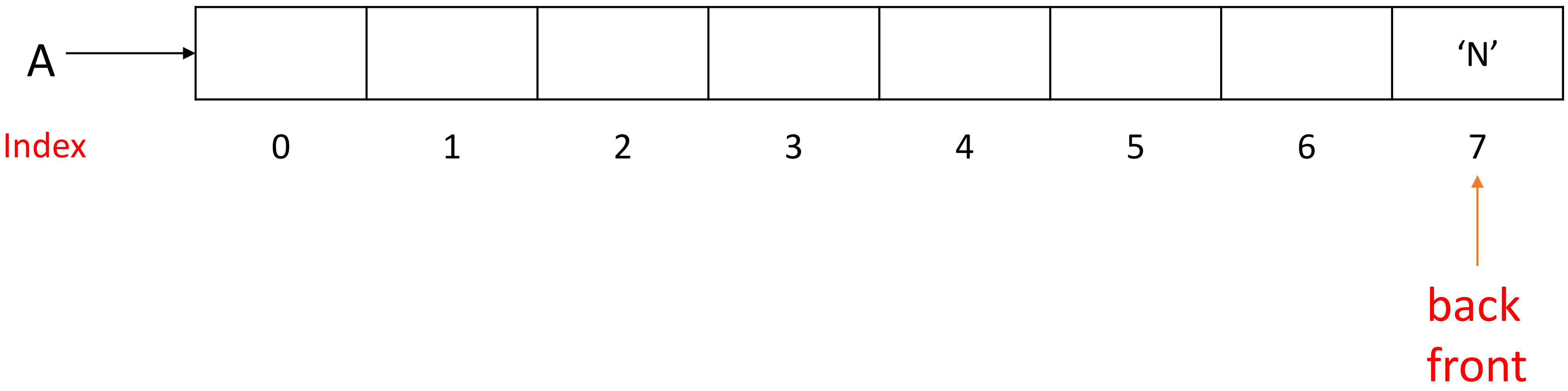
A							‘E’	‘N’
Index	0	1	2	3	4	5	6	7

Dequeue ‘E’ from the Queue

After

capacity = 8

count = 1

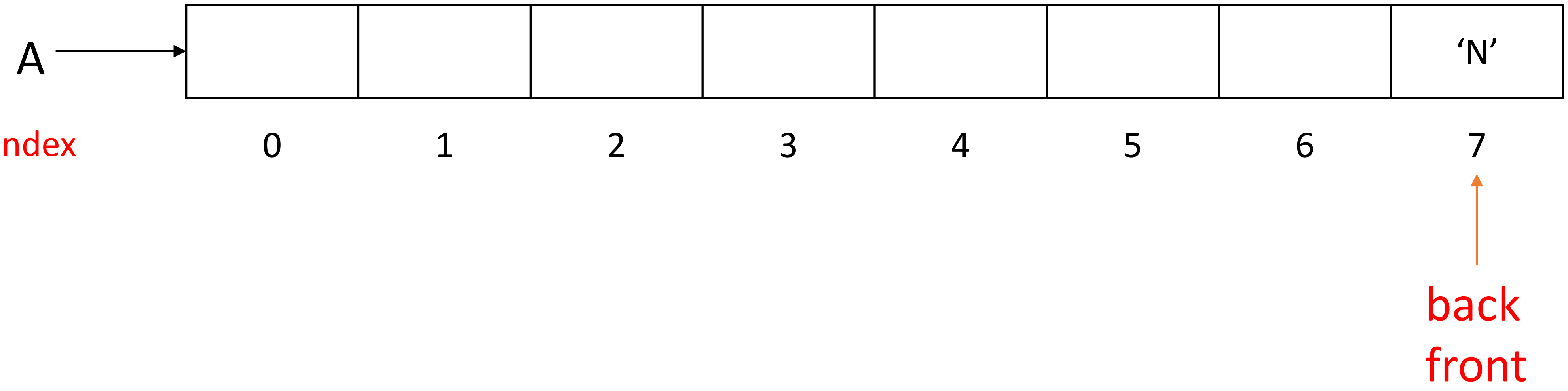


Dequeue ‘N’ from the Queue

Before

capacity = 8

count = 1



A

0	1	2	3	4	5	6	7
							‘N’

Index 0 1 2 3 4 5 6 7

back

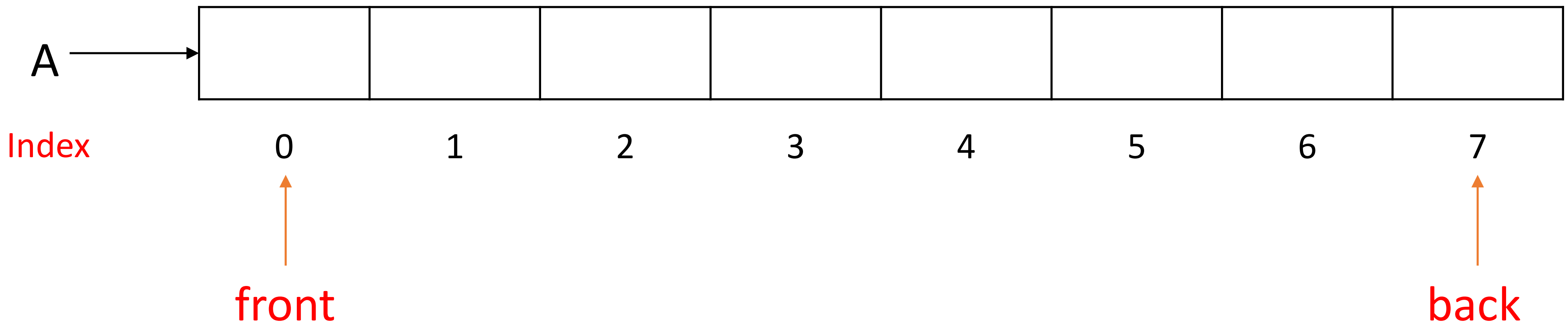
front

Dequeue ‘N’ from the Queue

After

capacity = 8

count = 0



A

Index 0 1 2 3 4 5 6 7

front

back

Only when the queue is
emptied does front “pass” back.

Retrieve the front item of a Queue

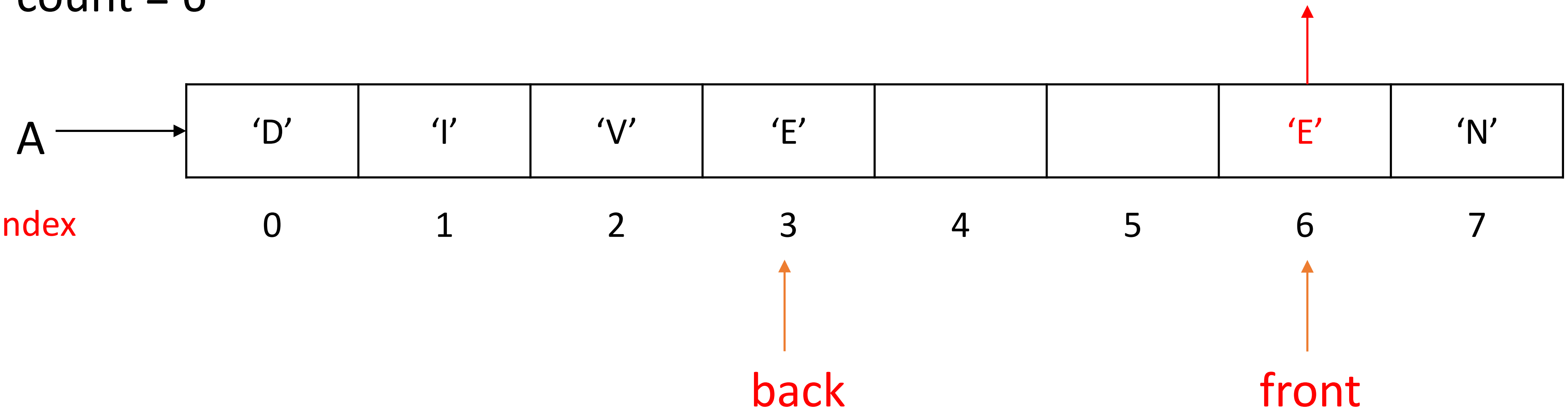
- Basic Strategy
    - If the queue is empty, then throw an exception else return A[front].

```
public T Front()
{
    if (Empty())
        throw new InvalidOperationException("Queue is empty");
    else
        return A[front];
}
```

Retrieve the front item of a Queue

capacity = 8

count = 6



A	‘D’	‘I’	‘V’	‘E’			‘E’	‘N’
Index	0	1	2	3	4	5	6	7

back

front

Supporting methods

- MakeEmpty
    - Resets count and front to 0
    - Sets back to capacity - 1
    - O(1)

- Empty
    - Returns true if count is 0; false otherwise
    - O(1)

- Size
    - Returns count
    - O(1)

Data structures

- Circular array
- Singly linked list ←

Singly linked list

Data members

```
public class Queue<T> : IQueue<T>
{
    ...                        // Node class
    private Node front;        // Reference to the front item
    private Node back;         // Reference to the back item
    private int count;         // Number of items in the queue
    ...
}
```

Node class (recap)

```
private class Node
{
    public T Item       { get; set; }
    public Node Next    { get; set; }

    public Node()
    {
        Next = null;
    }
    public Node(T item, Node next = null)
    {
        Item = item;
        Next = next;
    }
}
```

Read/Write Properties

Constructor

- Basic strategy:

  - Set front and back to null (no header node).
  - Set count to 0.
  - Use the MakeEmpty method.

Implementation

```
// Constructor

public Queue()
{
    MakeEmpty();
}
```

```
public void MakeEmpty()
{
    front = back = null;
    count = 0;
}
```

Initial configuration of an empty Queue

front ●

back ●

count = 0

Enqueue an item to the back of the Queue

- Basic Strategy
    - Create a new Node to store item and set its Next field to null.
    - If the queue is empty set back and front to the new Node
    - If the queue is not empty link the new Node to the end of the Queue and assign back to the new Node.
    - Increase count by 1.

Implementation

```
public void Enqueue(T item)
{
    Node newNode = new Node(item);
    if (count == 0)
    {
        front = back = newNode;
    }
    else
    {
        back = back.Next = newNode;
    }
    count++;
}
```

Case I: Empty Queue

Case II: Non-Empty Queue

Enqueue ‘B’ to the back of an empty Queue

Before

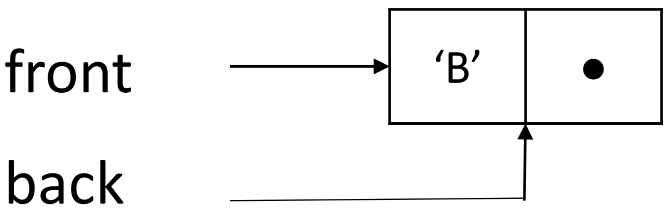
front ●

back ●

count = 0

Enqueue ‘B’ to the back of an empty Queue

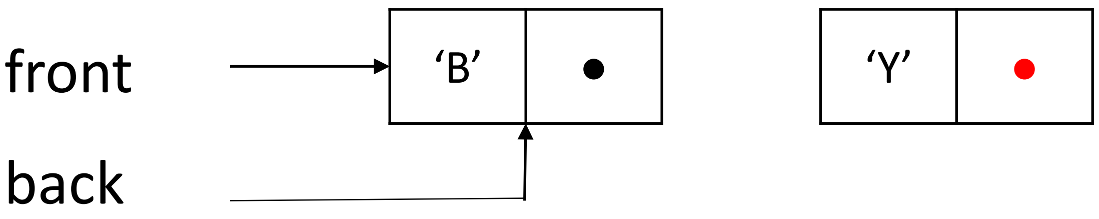
After (Case I)



count = 1

Enqueue ‘Y’ to the back of the Queue

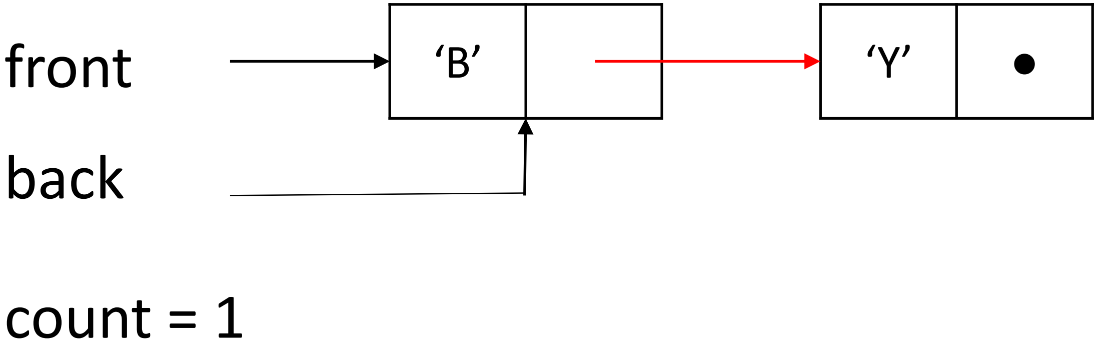
After (Case II)



count = 1

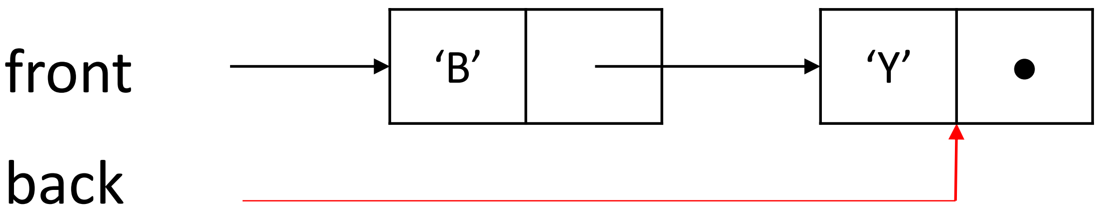
Enqueue ‘Y’ to the back of the Queue

After (Case II)



Enqueue ‘Y’ to the back of the Queue

After (Case II)



count = 2

back = back.Next = new Node<char>(‘Y’);

Dequeue an item from the front of a Queue

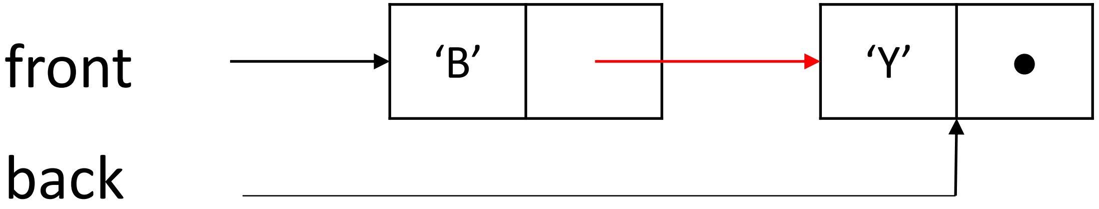
- Basic Strategy
  - If the Queue is empty, then throw an exception
  - Else
    - Move front to the next Node.
    - Decrease count by 1.

Implementation

```
public void Dequeue()
{
    if (Empty())
    {
        throw new InvalidOperationException("Queue is empty");
    }
    else
    {
        front = front.Next;
        count--;
    }
}
```

Dequeue an item from the Queue

Before



front

back

‘B’

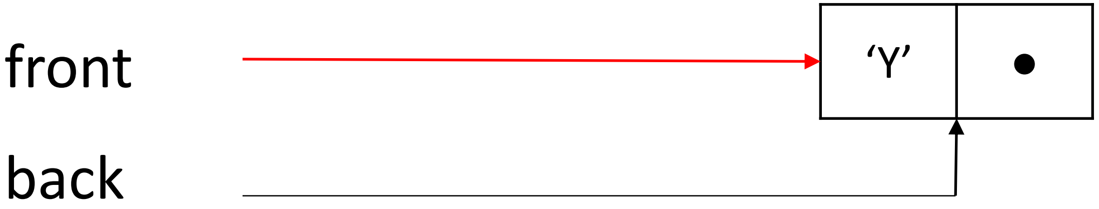
‘Y’

●

count = 2

Dequeue an item from the Queue

After



front

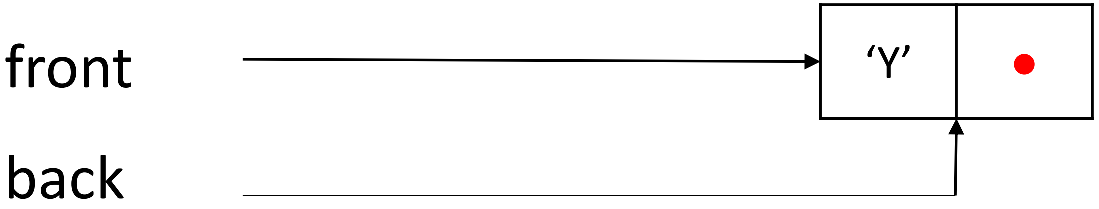
back

‘Y’ ●

count = 1

Dequeue an item from the Queue

Before



front

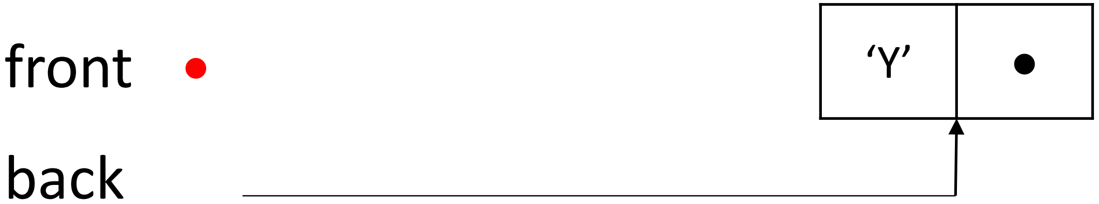
back

‘Y’

count = 1

Dequeue an item from the Queue

After



front

back

count = 0

Notice that back still refers to the last item.
Is this a problem?

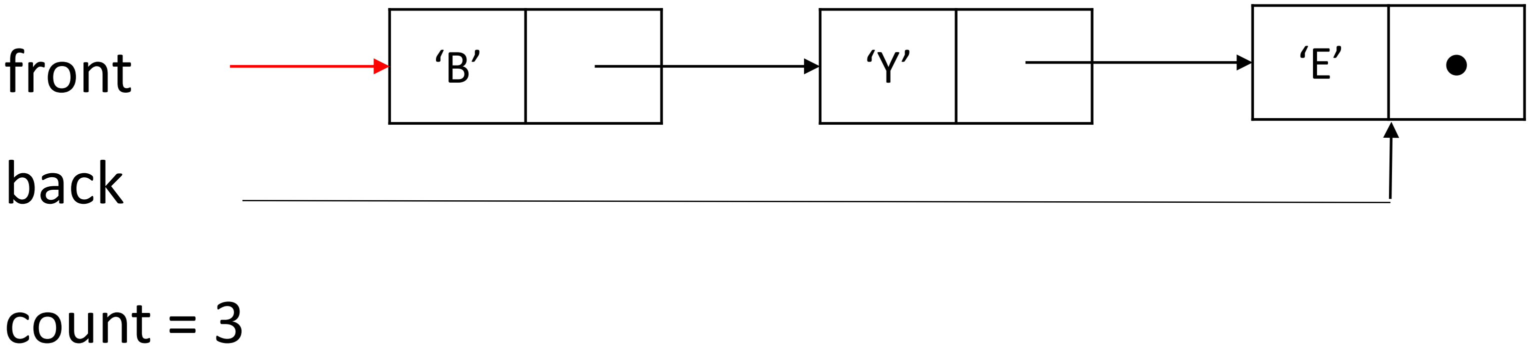
Retrieve the front item of a Queue

- Basic Strategy
  - If the Queue is empty, then throw an exception else return the front item.

```
public T Front()
{
    if (Empty())
        throw new InvalidOperationException("Queue is empty");
    else
        return front.Item;
}
```

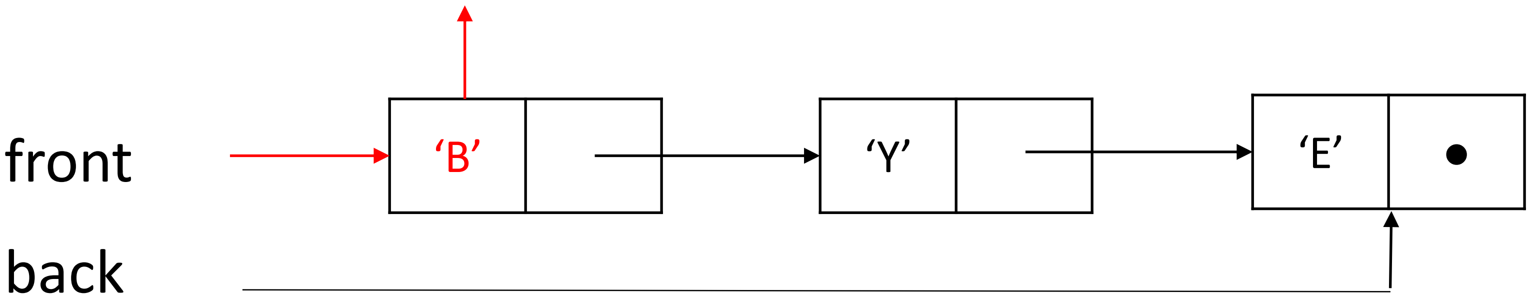
Retrieve the front item of the Queue

Before



Retrieve the front item of the Queue

After



front

back

‘B’

‘Y’

‘E’

count = 3

Supporting methods

- MakeEmpty
  - Sets front and back to null
  - Sets count to 0
  - O(1)

- Empty
  - Returns true if count is 0; false otherwise.
  - O(1)

- Size
  - Returns count.
  - O(1)

Time complexity

- The worst-case time complexity of all primary and supporting methods, except DoubleCapacity and Enqueue, is constant, i.e. O(1).
- DoubleCapacity is O(n).
- Enqueue has an amortized time complexity of O(1).

Exercises

1. Re-implement the class Queue using only two Stacks as data members.

2. How can you use the circular array to improve the average time to insert and remove items from a List?

3. If the back index is i % capacity and the front index is (i + 1) % capacity, is the queue empty, full or either?

Exercises (con’t)

4. Without count, what can you do to distinguish between an empty and a full queue?

5. Enqueue three items to an initially empty queue implemented as a singly linked list. Dequeue the three items and show the resultant queue. Why is it not necessary to set back to null when a queue is emptied?

6. Reimplement the Queue class using a header node.