CSE 373

SEPTEMBER 29 – STACKS AND QUEUES

DESIGN DECISIONS

- **Shopping list?**

DESIGN DECISIONS

- **Shopping list?**
  - What sorts of behavior do shoppers exhibit?
  - What constraints are there on a shopper?
  - What improvements would make a better shopping list?

DESIGN DECISIONS

- Shopping list?
- Stack?

DESIGN DECISIONS

- **Shopping list?**
- **Stack?**
    - What sorts of behavior does the ‘stack’ support?
    - What constraints are there on a stack user?
      (Is there a change in certainty?)
    - What improvements would make a better stack?
      (What problems might arise in a stack?)

STACK ADT

- **Important to know *exactly* what we expect from a stack.**

STACK ADT

- **Important to know *exactly* what we expect from a stack.**
  - Push(Object a) returns null; *(other options?)*
  - Pop() returns Object a: where a is the element on ‘top’ of the stack; also removes a from the stack
  - Top() returns Object a: where a is the element on ‘top’ of the stack without removing that element from the stack

STACK ADT

- **Important to know *exactly* what we expect from a stack.**
  - Push(Object a) returns null; *(other options?)*
  - Pop() returns Object a: where a is the element on ‘top’ of the stack; also removes a from the stack
  - Top() returns Object a: where a is the element on ‘top’ of the stack without removing that element from the stack
  - How long will these operations take?

STACK ADT

- **Important to know *exactly* what we expect from a stack.**
  - Push(Object a) returns null; *(other options?)*
  - Pop() returns Object a: where a is the element on ‘top’ of the stack; also removes a from the stack
  - Top() returns Object a: where a is the element on ‘top’ of the stack without removing that element from the stack
  - How long will these operations take?

That depends on the Data Structure and Implementation

QUEUE ADT

- **What behavior do we expect from the queue?**

QUEUE ADT

- **What behavior do we expect from the queue?**
    - `enqueue(Object toInsert):`
    - `dequeue():`
    - `front():`

QUEUE ADT

- **What behavior do we expect from the queue?**
  - `enqueue(Object toInsert):`
    adds to the queue
  - `dequeue():`
    removes the ‘next’ element from the queue
  - `front():`
    peeks at the ‘next’ element

QUEUE ADT

- **What behavior do we expect from the queue?**
  - `enqueue(Object toInsert):`
    adds to the queue
  - `dequeue():`
    removes the 'next' element from the queue
  - `front():`
    peeks at the 'next' element
- **Which element is 'next'?**

QUEUE ADT

- **What behavior do we expect from the queue?**
  - `enqueue(Object toInsert):`
    adds to the queue
  - `dequeue():`
    removes the ‘next’ element from the queue
  - `front():`
    peeks at the ‘next’ element
- **Which element is ‘next’?**
  - FIFO – ‘first in, first out’ ordering

STACK AND QUEUE ADT

- **Stacks and Queues both support the same functions**
  - insert: push() and enqueue()
  - remove: pop() and dequeue()
  - peek: top() and front()

STACK AND QUEUE ADT

- **Stacks and Queues both support the same functions**
  - insert: push() and enqueue()
  - remove: pop() and dequeue()
  - peek: top() and front()
- **This isn’t sufficient to distinguish them, their *behavior* is also a critical part of their ADT. *Which element do we expect to be ‘removed’?***
  - *FIFO v LIFO*

STACK AND QUEUE ADT

- The *ADT* describes the methods provided and the behavior we expect from them
- The *Data Structure* is a theoretical arrangement of the data that supports the functionality of the *ADT*

STACK AND QUEUE ADT

- What Data Structures might we use for Stacks and Queues?

STACK AND QUEUE ADT

- **What Data Structures might we use for Stacks and Queues?**
  - Arrays

STACK AND QUEUE ADT

- **What Data Structures might we use for Stacks and Queues?**
  - Arrays
  - *How many ways can we use arrays?*

STACK AND QUEUE ADT

- **What Data Structures might we use for Stacks and Queues?**
  - Arrays
  - *How many ways can we use arrays?*
  - *Which ways are efficient?*

QUEUE ADT

- Array implementation
- Unique problems?

QUEUE ADT

- **Array implementation**
- **Unique problems?**

What if the array is full?

QUEUE ADT

- Array implementation
- Unique problems?

What if the array is full?

What if we alternate enqueue() and dequeue()?

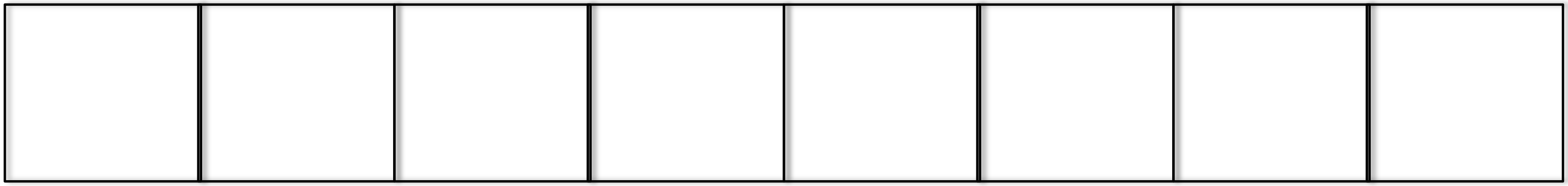
QUEUE ADT

- **Array implementation**
- **Unique problems?**
  - End of Array
- **Unique solutions?**

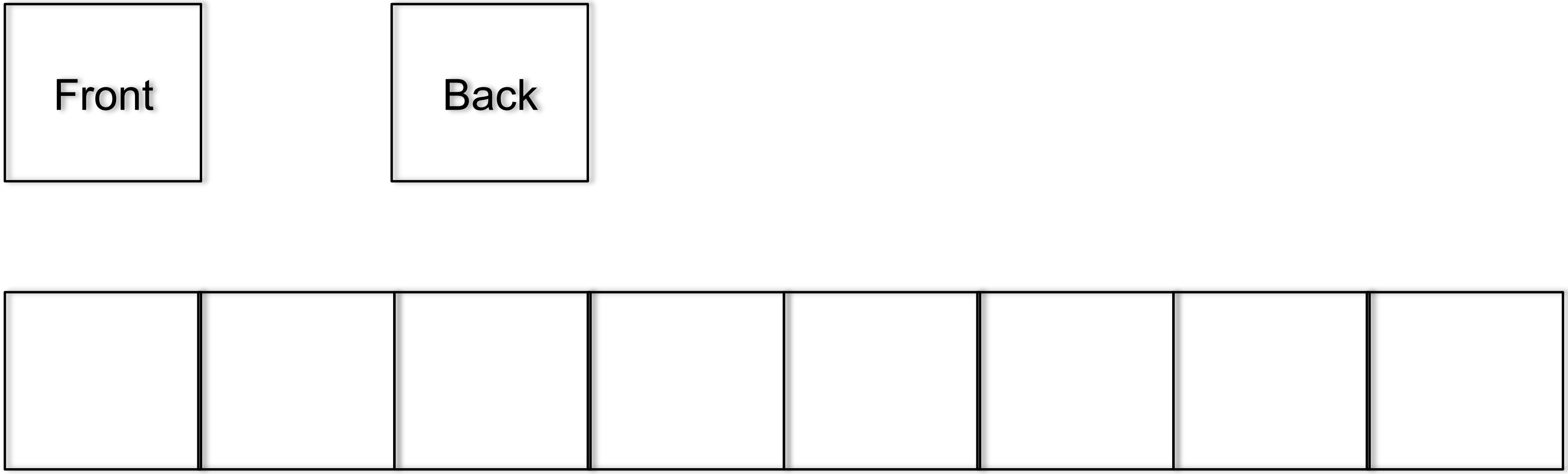
QUEUE ADT

- **Array implementation**
- **Unique problems?**
  - End of Array
- **Unique solutions?**
  - Resizing (costly!)
  - Circular Array (?)

CIRCULAR QUEUES



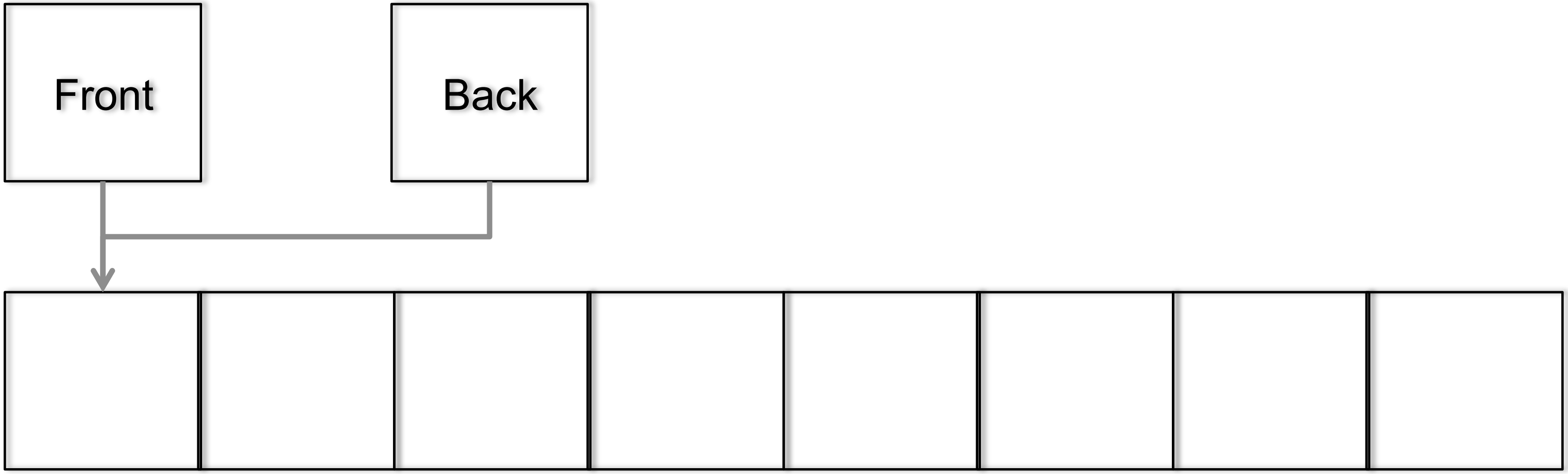
CIRCULAR QUEUES



Front

Back

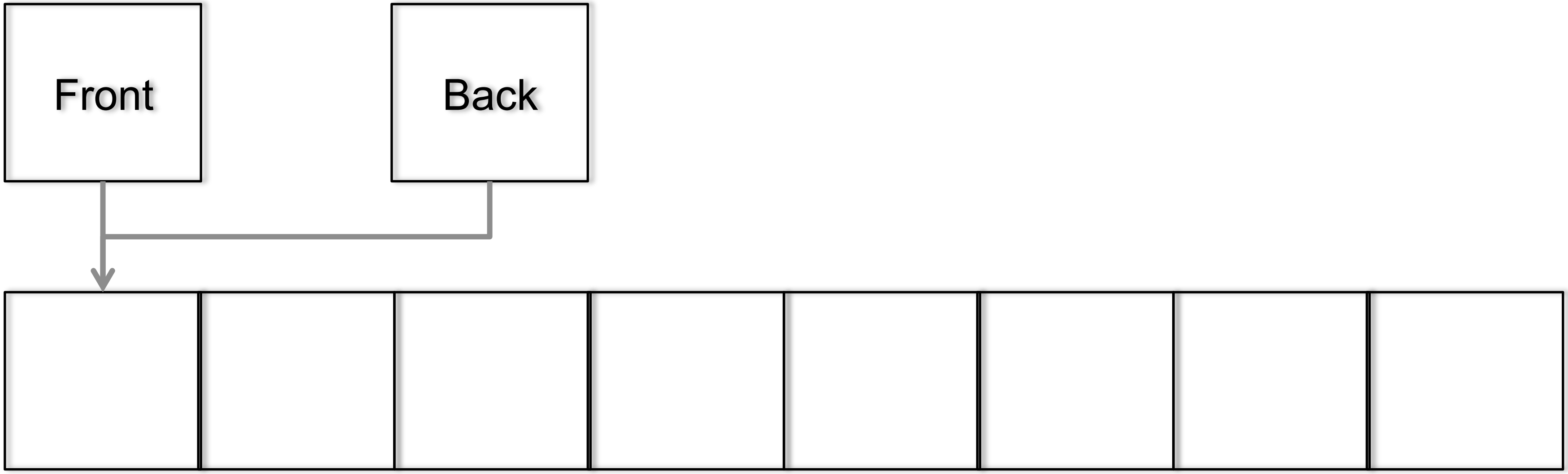
CIRCULAR QUEUES



Front

Back

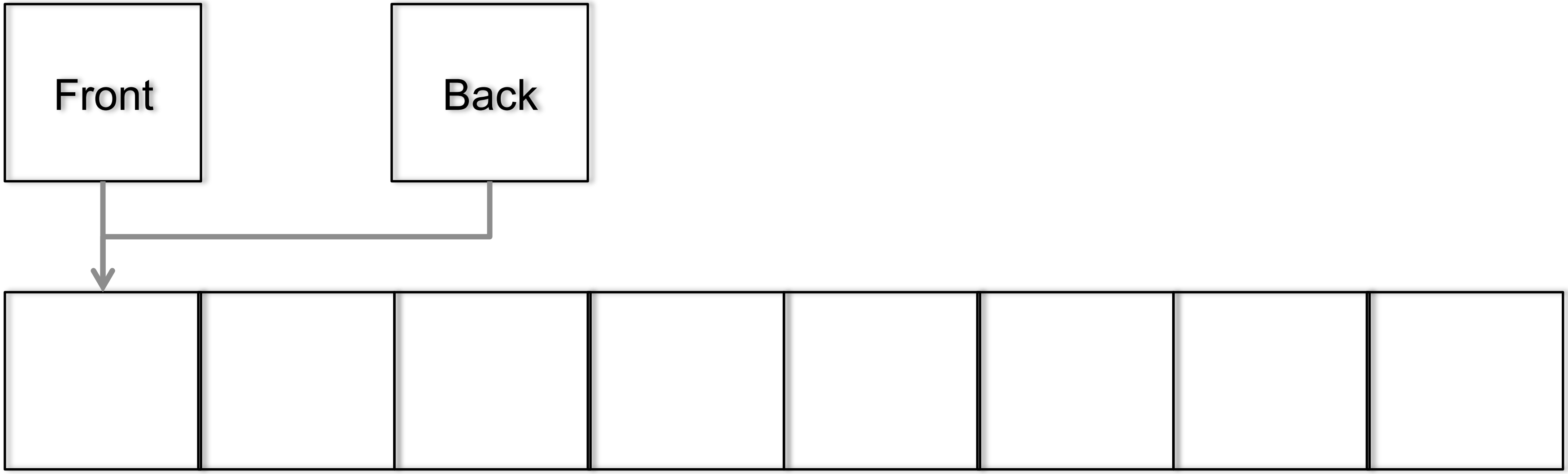
CIRCULAR QUEUES



Why this way?

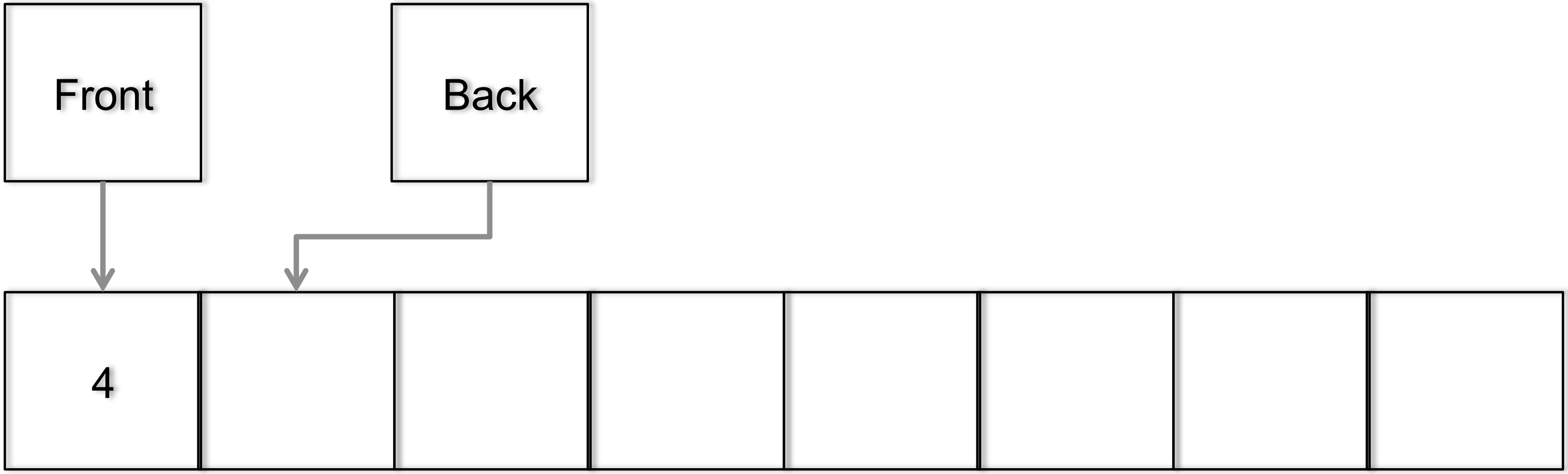
What function to front and back serve?

CIRCULAR QUEUES



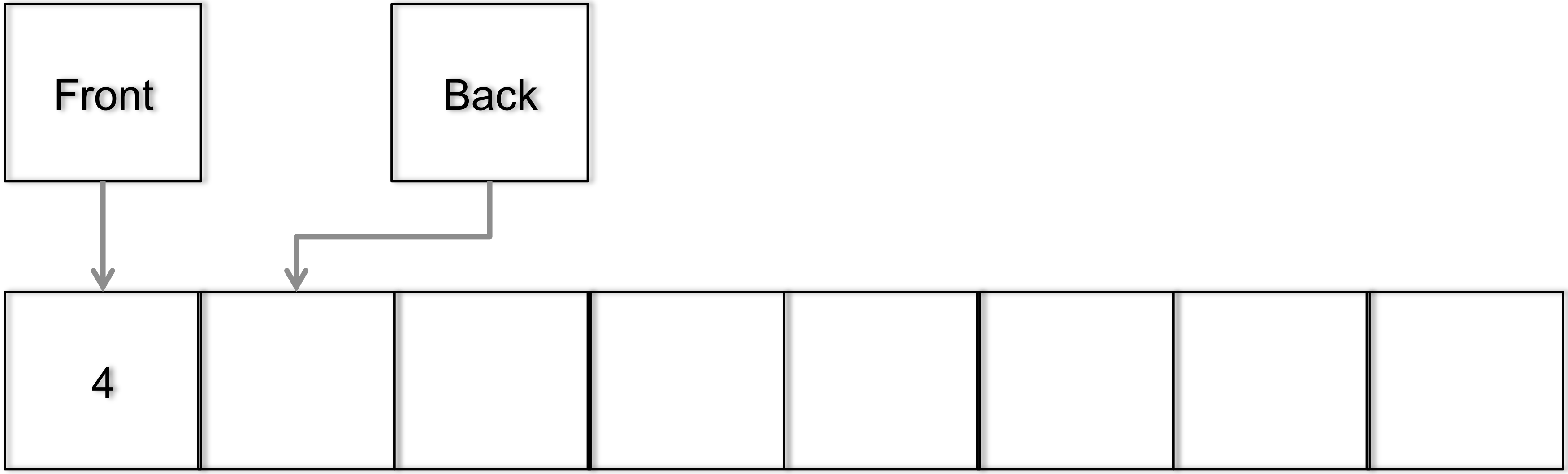
enqueue(4)

CIRCULAR QUEUES



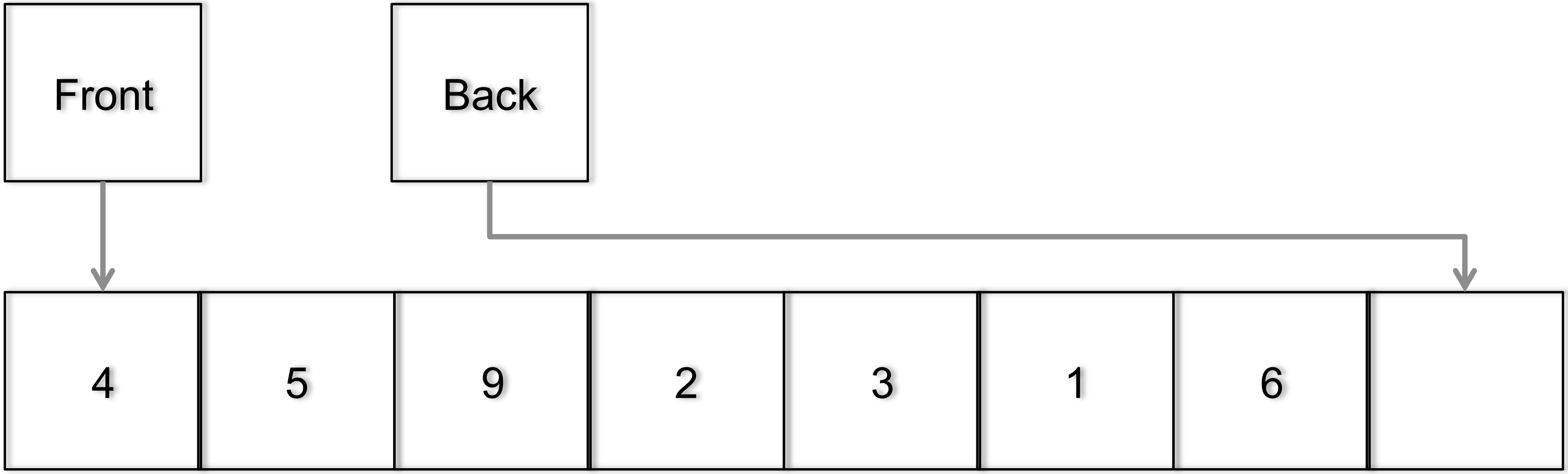
Which operations will move what pointers?

CIRCULAR QUEUES



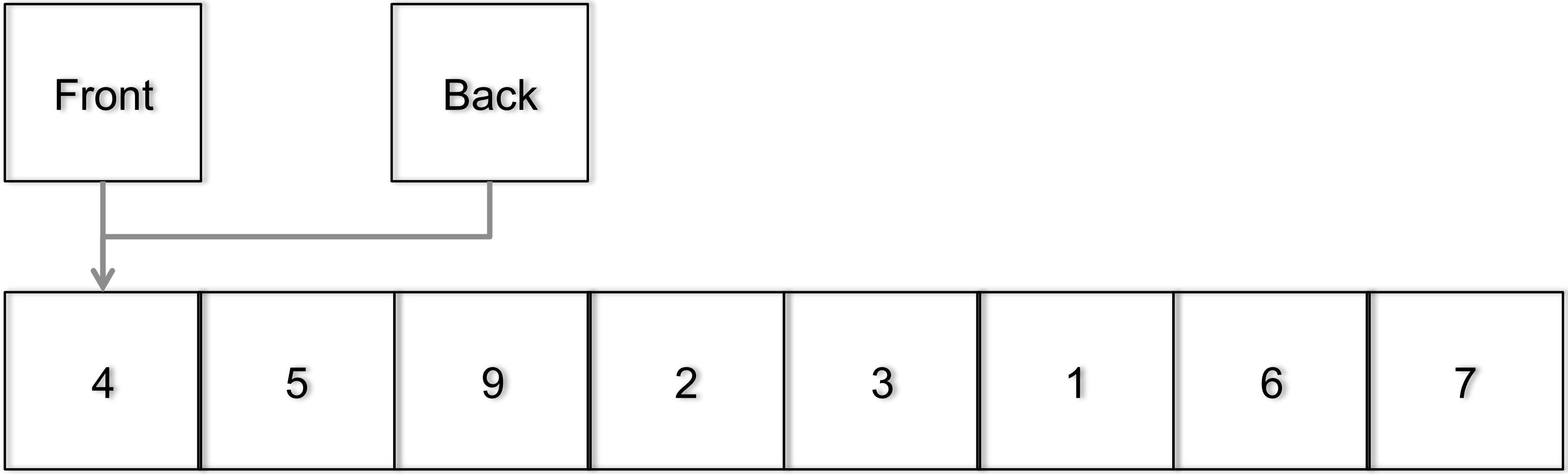
Let’s do several enqueues

CIRCULAR QUEUES



What happens now, on enqueue(7)?

CIRCULAR QUEUES



Front

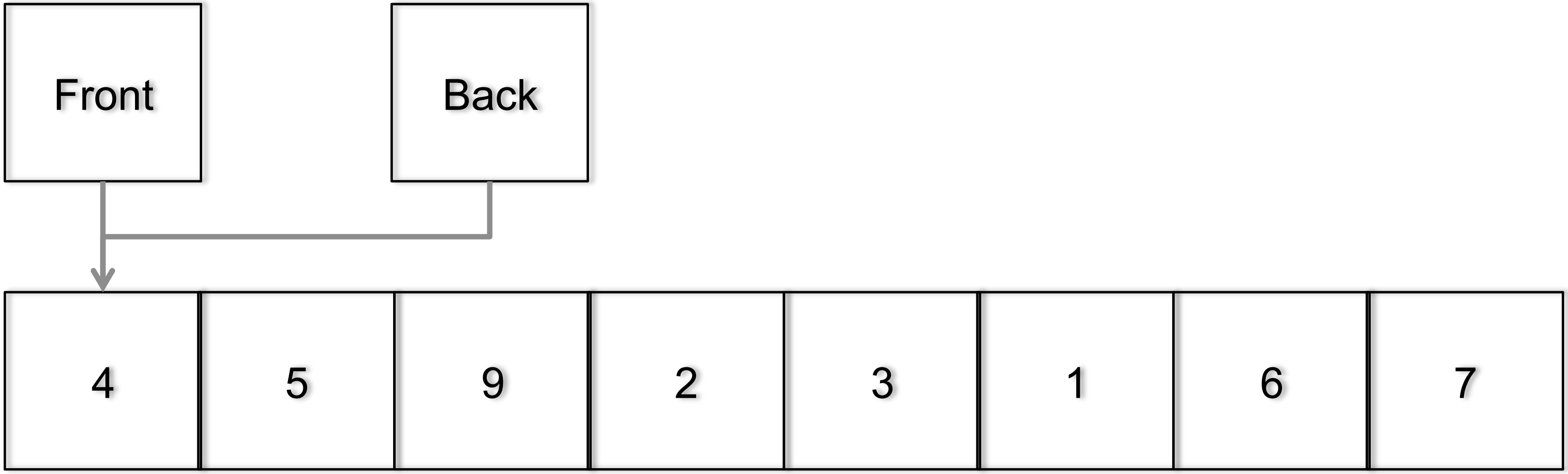
Back

4	5	9	2	3	1	6	7

Problems here?

How to implement?

CIRCULAR QUEUES



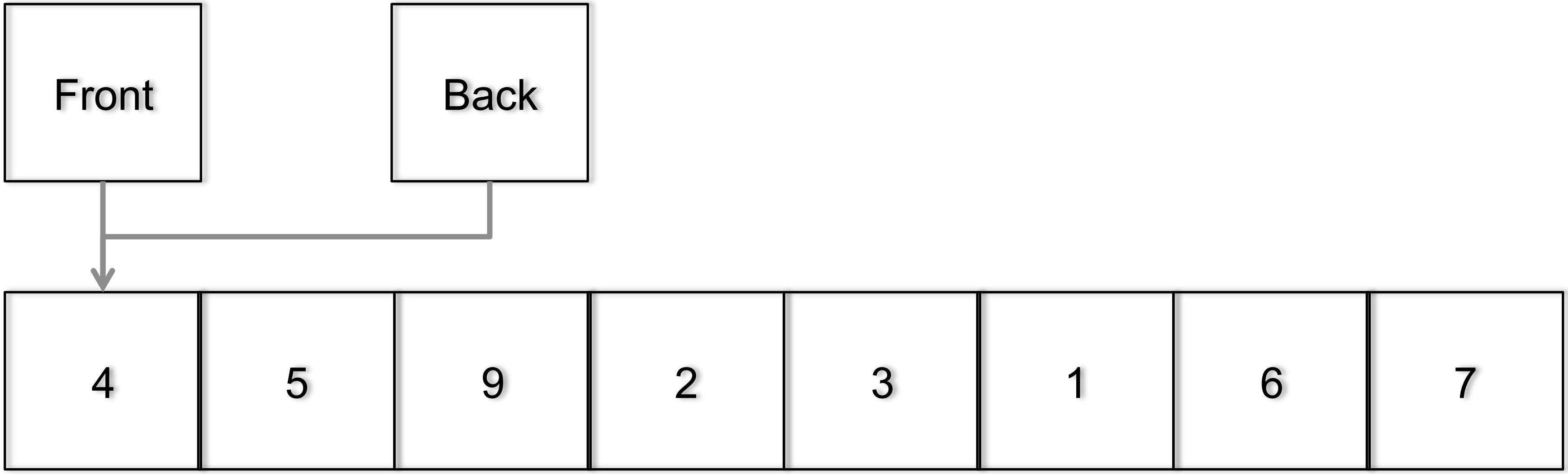
Front

Back

4 5 9 2 3 1 6 7

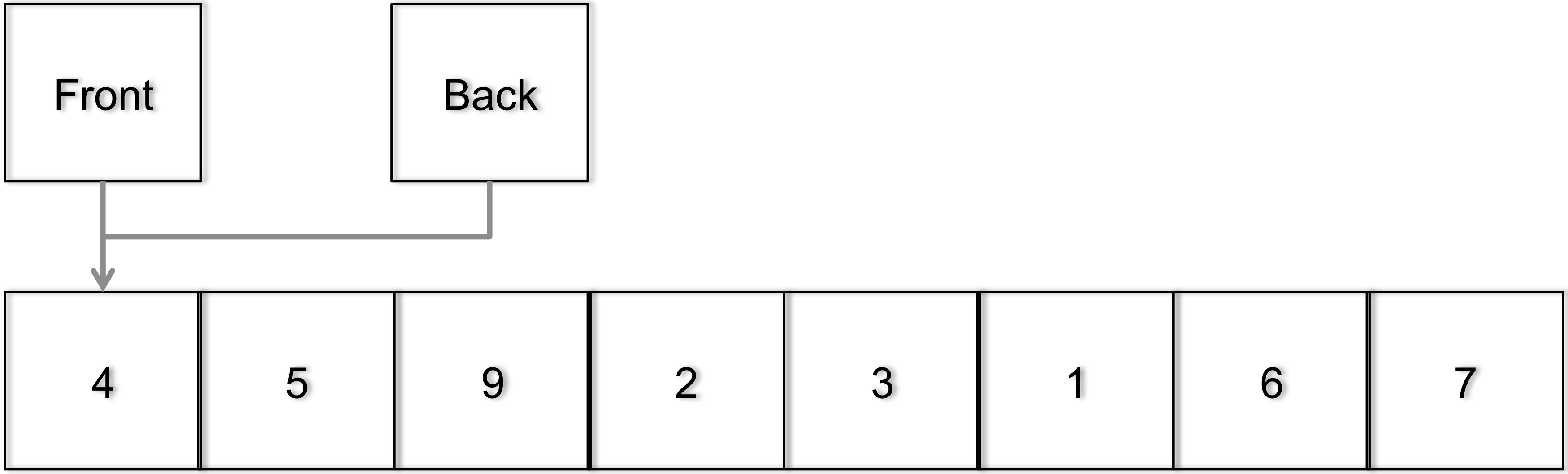
The queue is full, but it is the same situation (front == back) as when the queue is empty. This is a boundary condition.

CIRCULAR QUEUES



We have to resize the list (or deny the add) if we get another enqueue.

CIRCULAR QUEUES



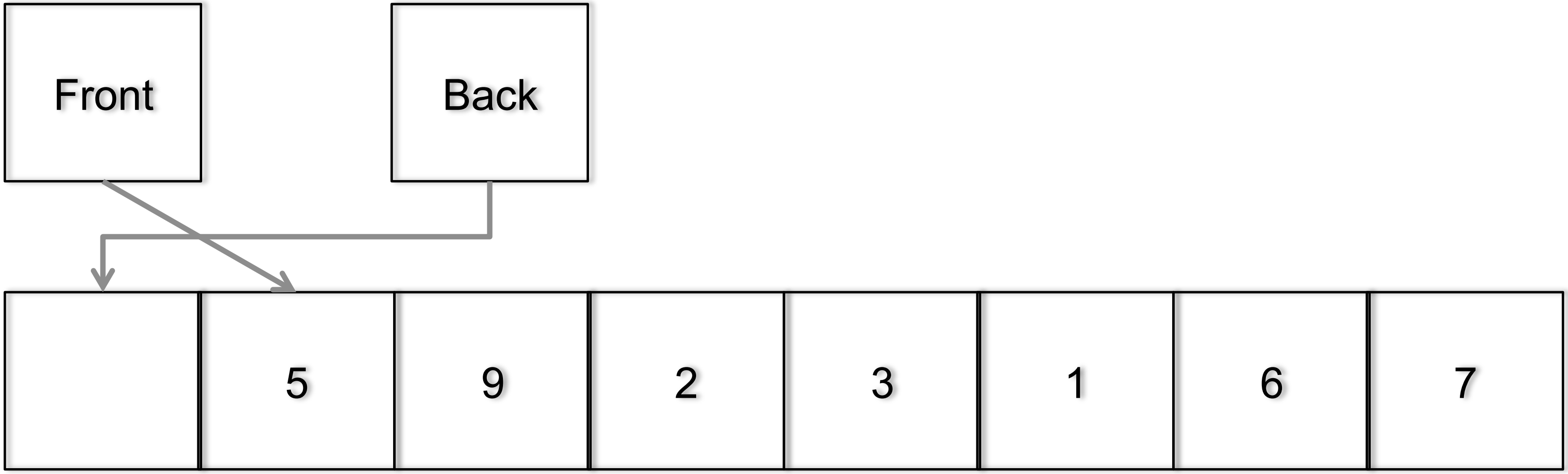
Front

Back

4	5	9	2	3	1	6	7

What if we dequeue some items?

CIRCULAR QUEUES



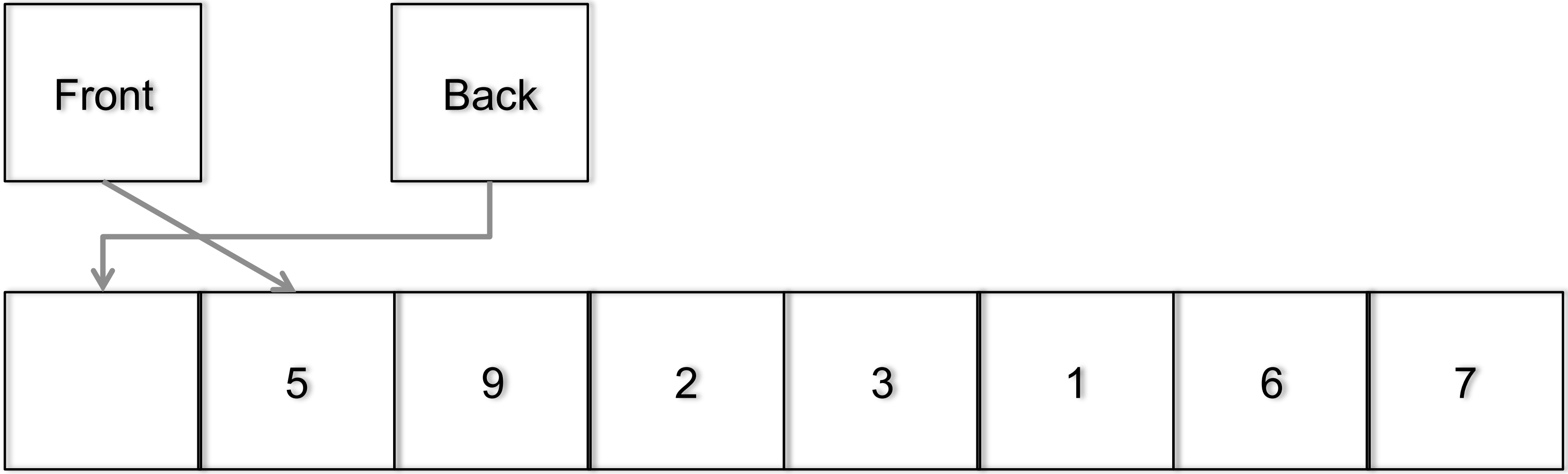
Front

Back

	5	9	2	3	1	6	7

Dequeue() outputs 4

CIRCULAR QUEUES



Front

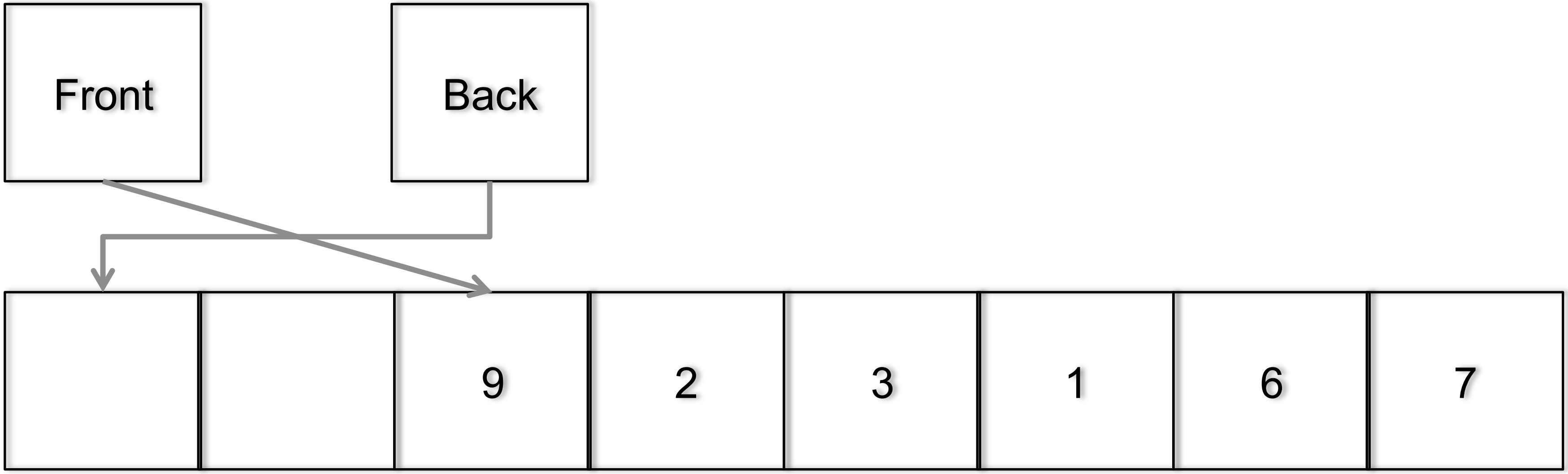
Back

	5	9	2	3	1	6	7

Dequeue() outputs 4

Is the 4 really “deleted”?

CIRCULAR QUEUES



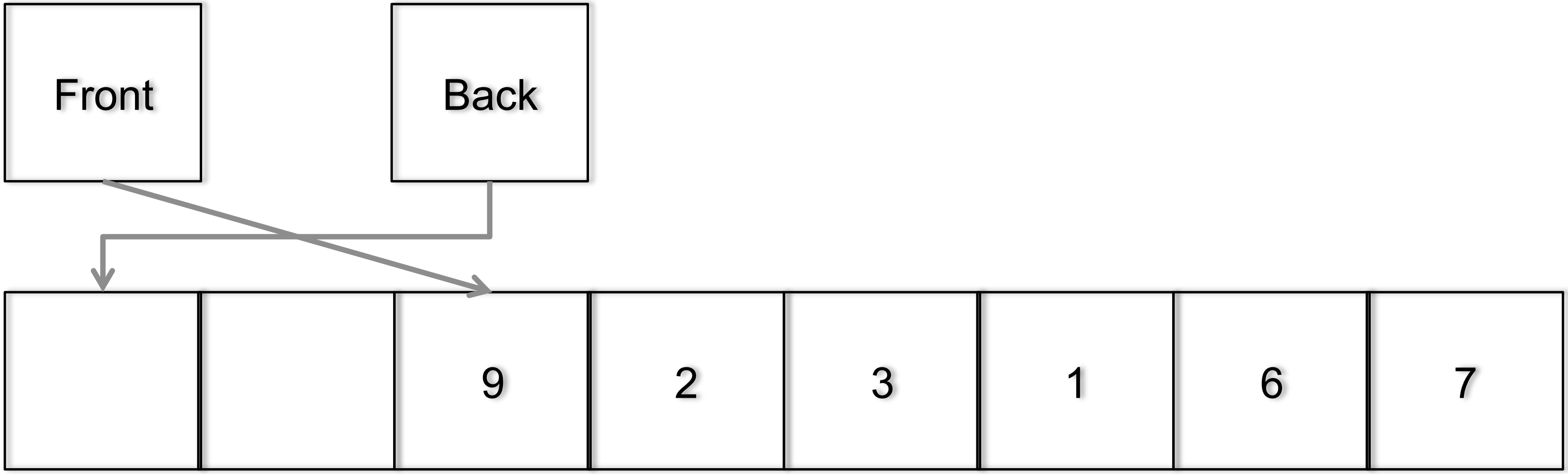
Front

Back

		9	2	3	1	6	7

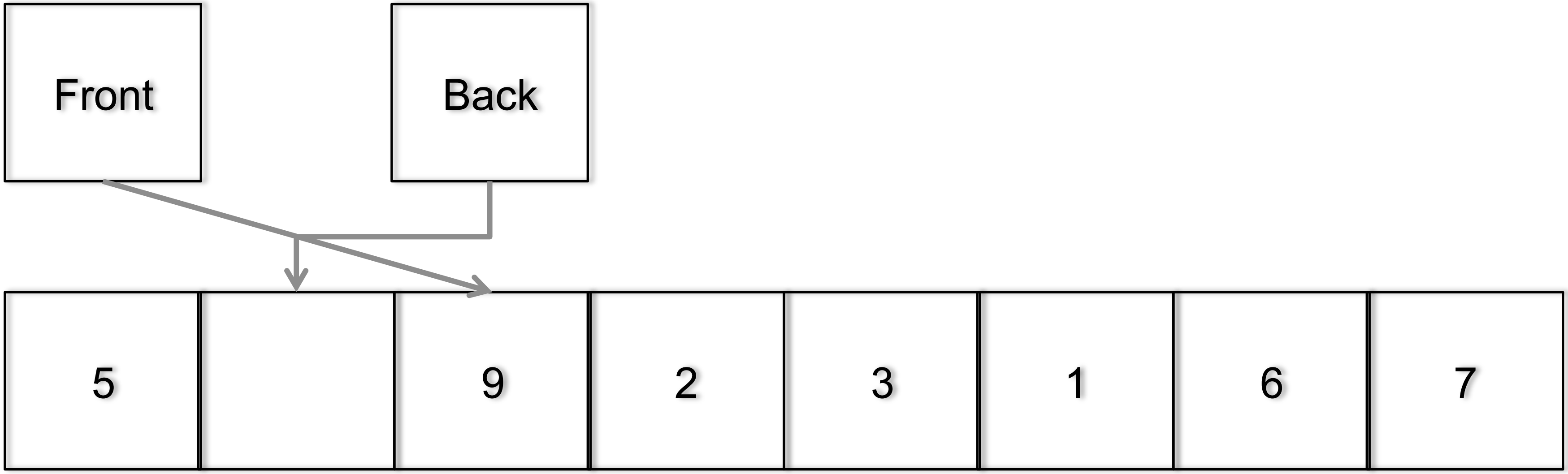
Output 5

CIRCULAR QUEUES



Now we’ve freed up some space and can enqueue more

CIRCULAR QUEUES



Front

Back

5		9	2	3	1	6	7

enqueue(5)

CIRCULAR QUEUES

- By moving the front and back pointers, we can utilize all of the space in the array
- Advantages over a linked list?

CIRCULAR QUEUES

- **By moving the front and back pointers, we can utilize all of the space in the array**
- **Advantages over a linked list?**
  - Fixed number of items
  - Small data (Memory efficiency)

- ***From Wednesday: What is the memory overhead of the linked list?***

TESTING

- Implementation is great if it works on the first try

TESTING

- **Implementation is great if it works on the first try**
- **In a large implementation, what is causing the problem?**
  - Data structure?
  - Client?
  - Wrapper?

TESTING

- Implementation is great if it works on the first try
- In a large implementation, what is causing the problem?
- Object oriented programming allows modularity – good testing can pinpoint bugs to particular modules

TESTING

- Two primary types of testing

TESTING

- **Two primary types of testing**
  - Black box
    - Behavior only, no peeking into the code
    - This usually tests ADT behavior
    - Can test performance/efficiency by using a timer

TESTING

- **Two primary types of testing**
  - White box (or clear box)
    - Where there is an understanding of the implementation that can be leveraged for testing
    - If you’re writing your own DS, you can peek into attributes that you would normally refuse access to the client

TESTING

- Isolate the problem

TESTING

- **Isolate the problem**
  - Write specific tests
  - Running the whole program doesn’t help narrow down problems

TESTING

- **Isolate the problem**
  - Write specific tests
  - Running the whole program doesn't help narrow down problems
- **What are expected test cases?**

TESTING

- **Isolate the problem**
  - Write specific tests
  - Running the whole program doesn't help narrow down problems

TESTING

- **Many test cases (and large ones)**
    - You can prove that an algorithm is correct, but you cannot necessarily prove an arbitrary implementation is correct

TESTING

- **Many test cases (and large ones)**
  - You can prove that an algorithm is correct, but you cannot necessarily prove an arbitrary implementation is correct
- **More inputs can increase certainty**
  - Adversarial testing
  - The client is not your friend

TESTING

- **Good things to test**
  - Expected behavior (at multiple sizes)
  - Forbidden input
  - Empty/Null
  - Side effects
  - Boundary/Edge Cases

NEW ADT

- Stacks and Queues are great, but they’re very simple.
- Data structures is about storing and managing data, but S/Q restrict access to that data
- What sort of behavior would be more general?

DICTIONARY ADT

- **Operates on two data types**
  - a key, our lookup data type
  - a value, the related data stored in the structure
- **Supports three main functions**
  - insert(K key, V value)
  - delete(K key)
  - find(K key)

DICTIONARY ADT

- **Example**
    - English Language Dictionary

DICTIONARY ADT

- **Example**
  - English Language Dictionary
    - What are keys and values?

DICTIONARY ADT

- **Example**
    - English Language Dictionary
        - Keys here are words (Strings)
        - Values are definitions (Strings)

DICTIONARY ADT

- **Example**
  - English Language Dictionary
    - Keys here are words (Strings)
    - Values are definitions (Strings)
  - Keys and Values can be the same data type

DICTIONARY ADT

- **Example**
  - English Language Dictionary
    - Keys here are words (Strings)
    - Values are definitions (Strings)
  - Keys and Values can be the same data type
  - find(String word) will return the definition of the word – provided that the <word,definition> pair was added to the dictionary

NEXT WEEK

- Dictionary/Map behavior and ADT
- Simple Implementations
- Analyzing behavior, what do we mean when we say an algorithm is efficient?