QUEUES

The Queue ADT

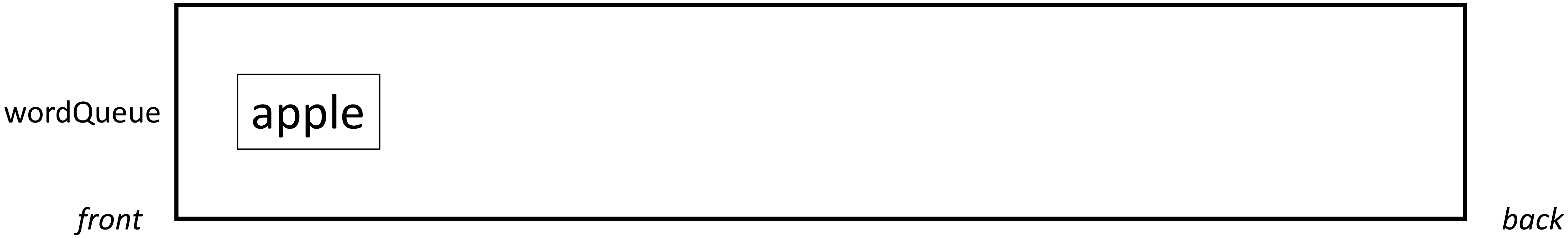
- A queue is a essentially a list in which you can only add to the back and remove from the front.
    - You cannot directly access elements in the middle.
- Queues are a first-in, first-out (FIFO) data structure.
    - This also known as a last-in, last-out (LILO) data structure.
    - This is essentially the reverse of a stack.

- Removing entries from a queue gives you the chronological ordering in which the elements were added.

Queue Methods

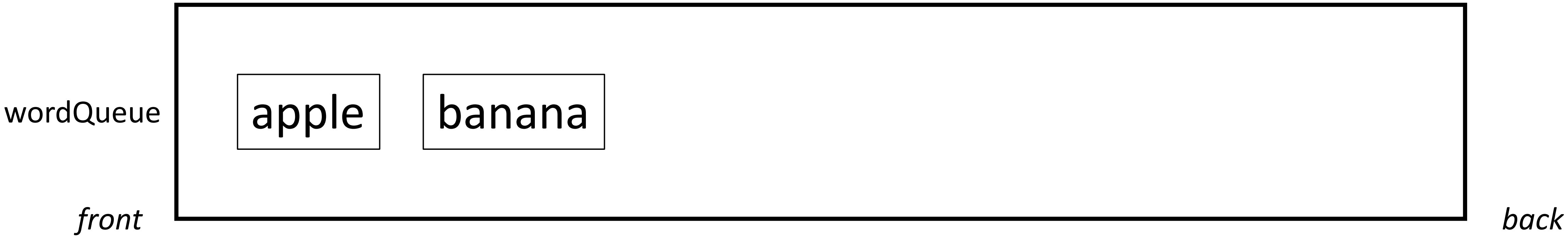
- enqueue
  - adds to the queue
  - similar to insertTail(obj) or add(obj)
- dequeue
  - removes from the queue
  - similar to removeHead() or remove(0)
- getFront
  - looks at the front of the queue but does not change the queue
  - similar to getEntry(1) or get(0)

Example



```
wordQueue.enqueue(“apple”);
wordQueue.enqueue(“banana”);
wordQueue.enqueue(“cantaloupe”);
System.out.println(wordQueue.getFront());
System.out.println(wordQueue.dequeue());
System.out.println(wordQueue.dequeue());
```

Example



wordQueue

apple

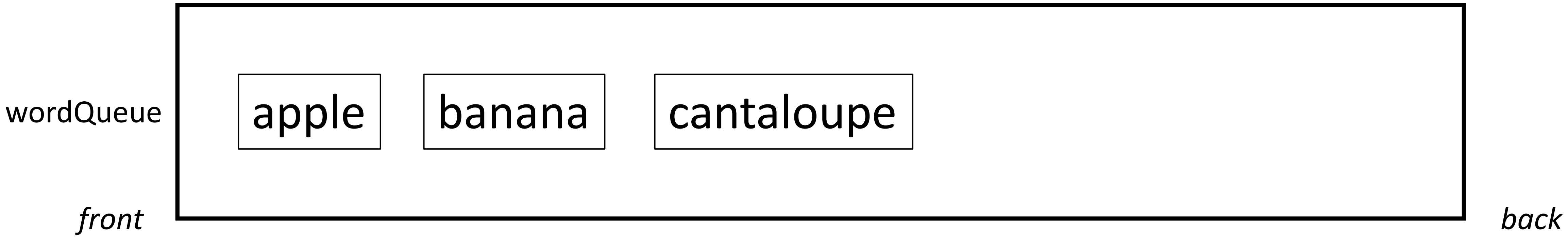
banana

front

back

```
wordQueue.enqueue("apple");
wordQueue.enqueue("banana");
wordQueue.enqueue("cantaloupe");
System.out.println(wordQueue.getFront());
System.out.println(wordQueue.dequeue());
System.out.println(wordQueue.dequeue());
```

Example



wordQueue

apple

banana

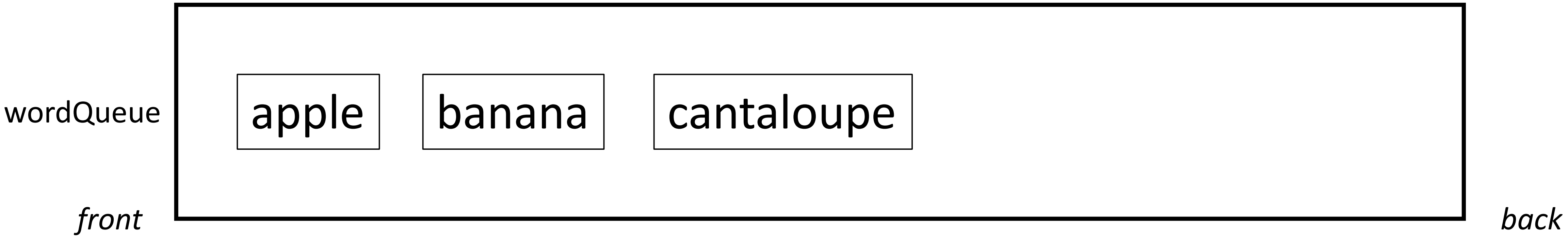
cantaloupe

front

back

```
wordQueue.enqueue(“apple”);
wordQueue.enqueue(“banana”);
wordQueue.enqueue(“cantaloupe”);
System.out.println(wordQueue.getFront());
System.out.println(wordQueue.dequeue());
System.out.println(wordQueue.dequeue());
```

Example



wordQueue

apple banana cantaloupe

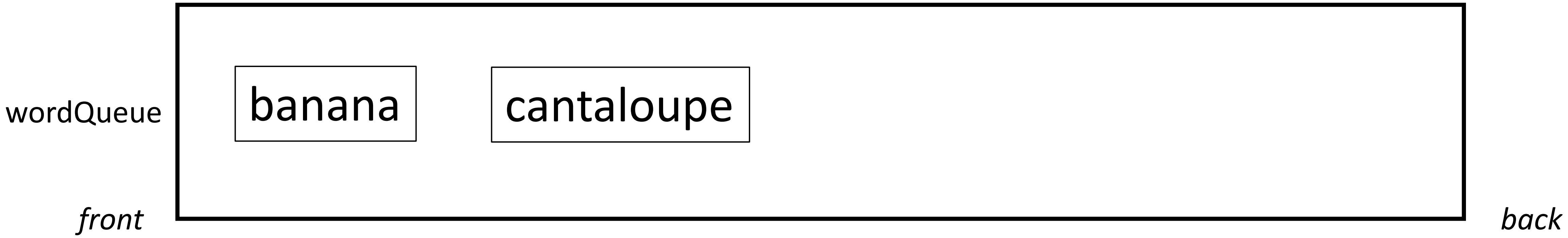
front

back

```
wordQueue.enqueue(“apple”);
wordQueue.enqueue(“banana”);
wordQueue.enqueue(“cantaloupe”);
System.out.println(wordQueue.getFront());
System.out.println(wordQueue.dequeue());
System.out.println(wordQueue.dequeue());
```

prints: apple

Example



wordQueue

banana

cantaloupe

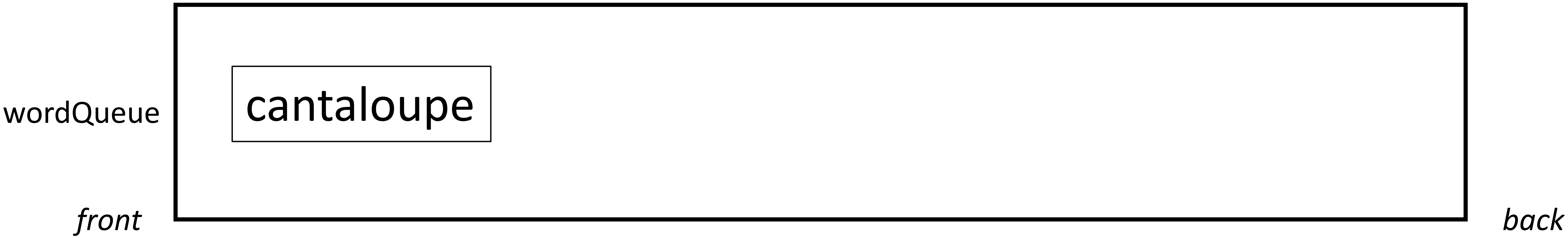
front

back

```
wordQueue.enqueue(“apple”);
wordQueue.enqueue(“banana”);
wordQueue.enqueue(“cantaloupe”);
System.out.println(wordQueue.getFront());
System.out.println(wordQueue.dequeue());
System.out.println(wordQueue.dequeue());
```

prints: apple

Example


wordQueue

cantaloupe

front

back

```
wordQueue.enqueue(“apple”);
wordQueue.enqueue(“banana”);
wordQueue.enqueue(“cantaloupe”);
System.out.println(wordQueue.getFront());
System.out.println(wordQueue.dequeue());
System.out.println(wordQueue.dequeue());
```

prints: banana

Queues

- Queues are often used to represent waiting behavior.
  - Tasks waiting to be executed
  - People waiting for service

Example: A Customer Queue

- You have five customers arriving at different times.
- Each customer needs 3 minutes to complete the task (e.g., buying tickets, getting helped, etc.).
- Only one customer can complete a task at a time (e.g., there is only one ticket window, only one customer service representative, etc.)
- Here is when each customer arrives:
    - Customer1: Time 5
    - Customer2: Time 6
    - Customer3: Time 8
    - Customer4: Time 9
    - Customer5: Time 10

Example: A Customer Queue

- Customer1: Time 5
- Customer2: Time 6
- Customer3: Time 8
- Customer4: Time 9
- Customer5: Time 10

Time: 0, 1, 2, 3, 4

Customer Being Helped:

customerQueue

front

back

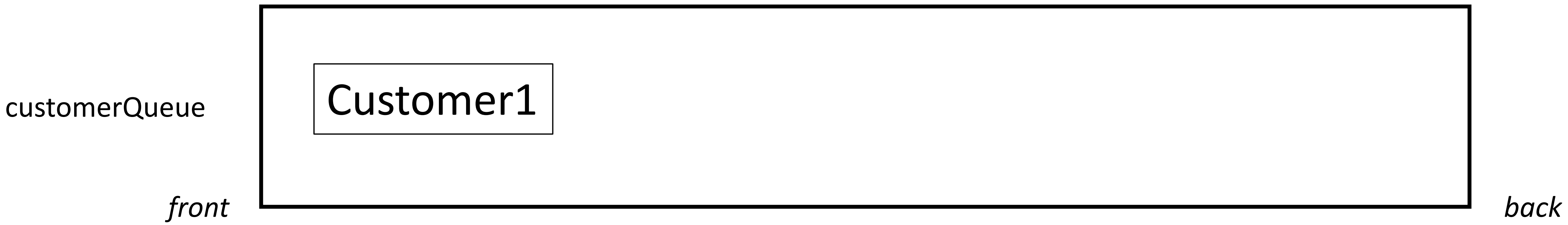
Example: A Customer Queue

- Customer1: Time 5
- Customer2: Time 6
- Customer3: Time 8
- Customer4: Time 9
- Customer5: Time 10

Time: 5

Customer Being Helped:

Customer1 (started at 5)



customerQueue

Customer1

front

back

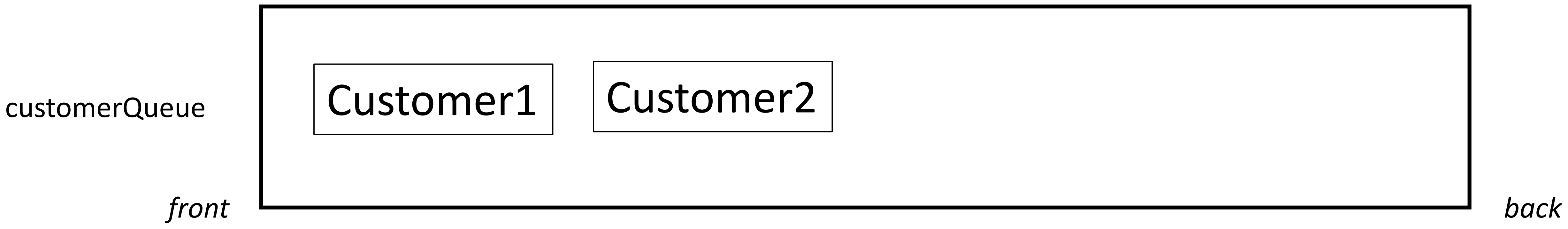
Example: A Customer Queue

- Customer1: Time 5
- Customer2: Time 6
- Customer3: Time 8
- Customer4: Time 9
- Customer5: Time 10

Time: 6

Customer Being Helped:

Customer1 (started at 5)



customerQueue

Customer1

Customer2

front

back

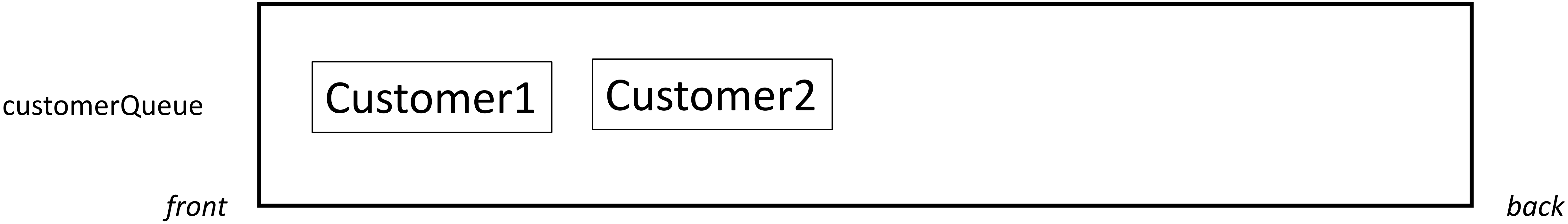
Example: A Customer Queue

- Customer1: Time 5
- Customer2: Time 6
- Customer3: Time 8
- Customer4: Time 9
- Customer5: Time 10

Time: 7

Customer Being Helped:

Customer1 (started at 5)



customerQueue

Customer1

Customer2

front

back

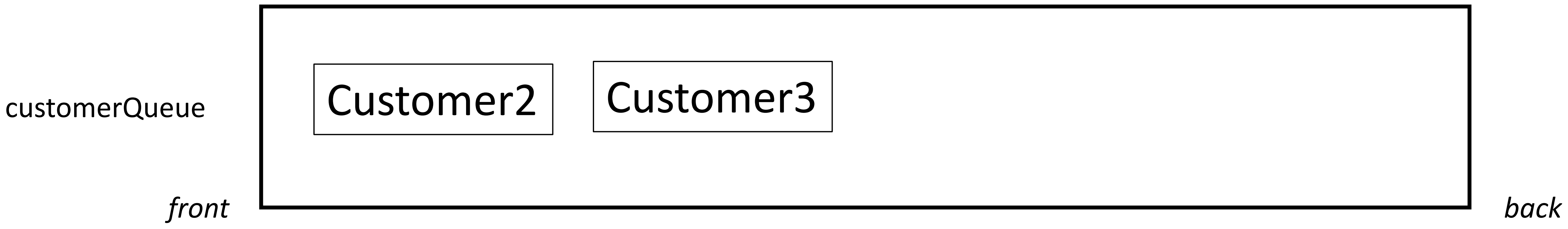
Example: A Customer Queue

- Customer1: Time 5
- Customer2: Time 6
- Customer3: Time 8
- Customer4: Time 9
- Customer5: Time 10

Time: 8

Customer Being Helped:

Customer2 (started at 8)



customerQueue

Customer2

Customer3

front

back

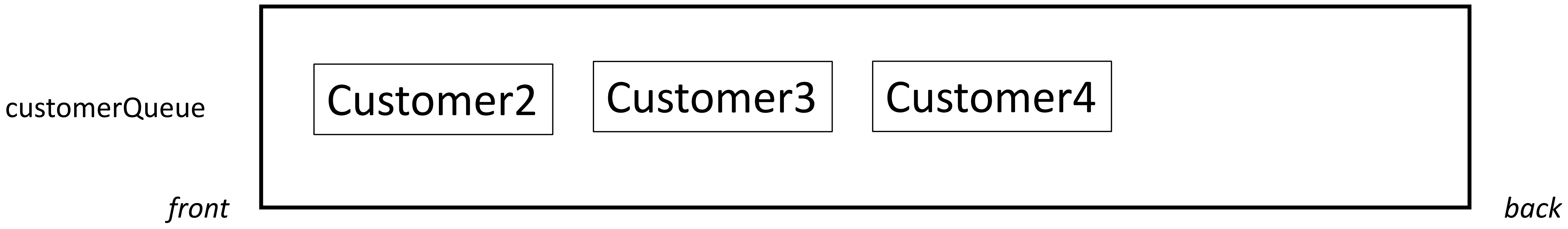
Example: A Customer Queue

- Customer1: Time 5
- Customer2: Time 6
- Customer3: Time 8
- Customer4: Time 9
- Customer5: Time 10

Time: 9

Customer Being Helped:

Customer2 (started at 8)



customerQueue

Customer2

Customer3

Customer4

front

back

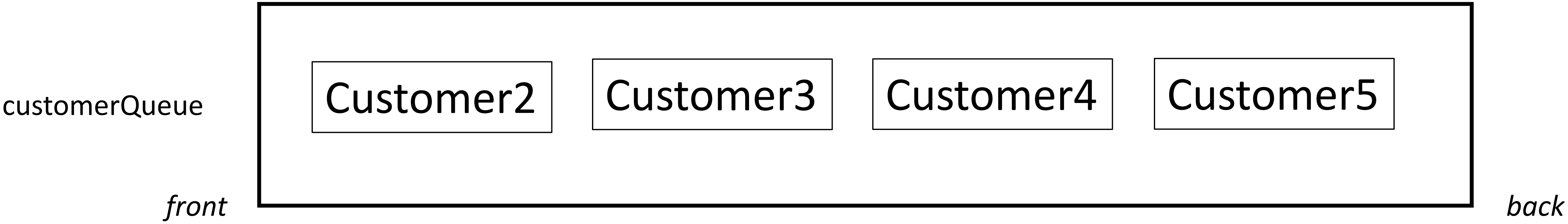
Example: A Customer Queue

- Customer1: Time 5
- Customer2: Time 6
- Customer3: Time 8
- Customer4: Time 9
- Customer5: Time 10

Time: 10

Customer Being Helped:

Customer2 (started at 8)



customerQueue

Customer2

Customer3

Customer4

Customer5

front

back

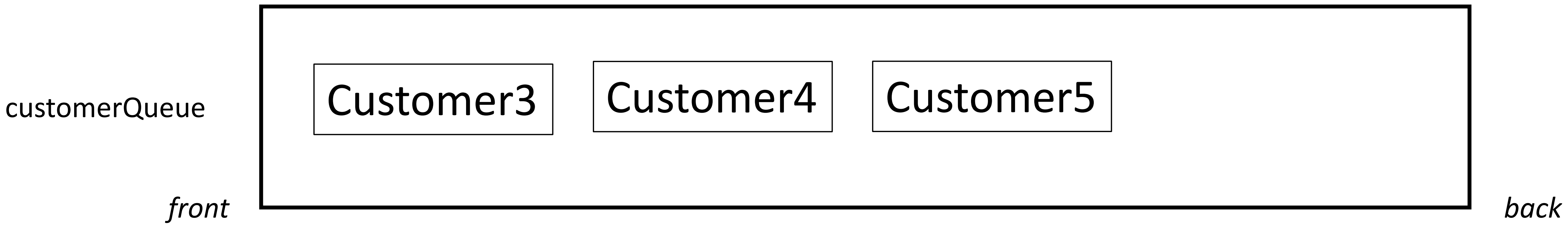
Example: A Customer Queue

- Customer1: Time 5
- Customer2: Time 6
- Customer3: Time 8
- Customer4: Time 9
- Customer5: Time 10

Time: 11

Customer Being Helped:

Customer3 (started at 11)



customerQueue

Customer3

Customer4

Customer5

front

back

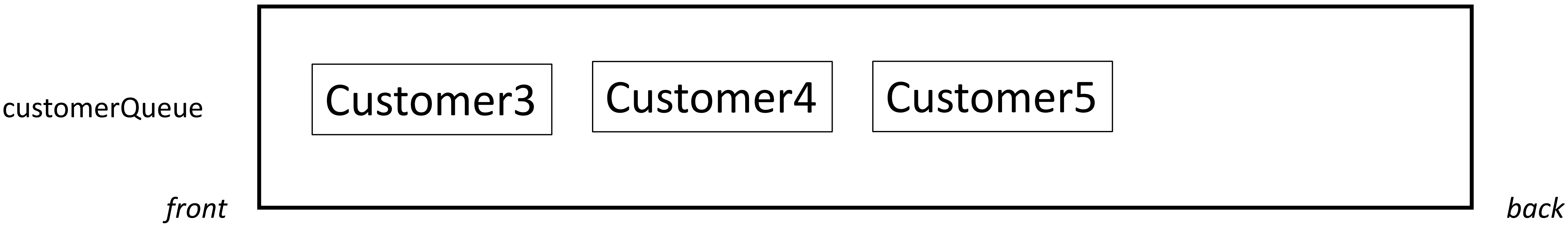
Example: A Customer Queue

- Customer1: Time 5
- Customer2: Time 6
- Customer3: Time 8
- Customer4: Time 9
- Customer5: Time 10

Time: 12, 13

Customer Being Helped:

Customer3 (started at 11)



customerQueue

Customer3

Customer4

Customer5

front

back

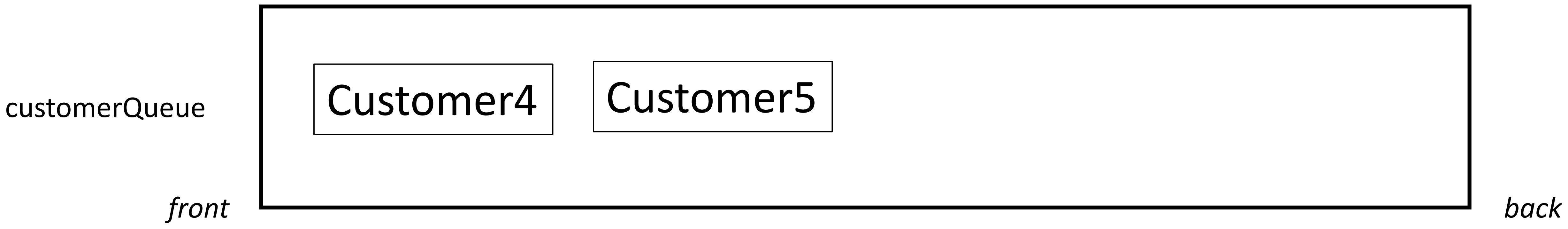
Example: A Customer Queue

- Customer1: Time 5
- Customer2: Time 6
- Customer3: Time 8
- Customer4: Time 9
- Customer5: Time 10

Time: 14

Customer Being Helped:

Customer4 (started at 14)



customerQueue

Customer4

Customer5

front

back

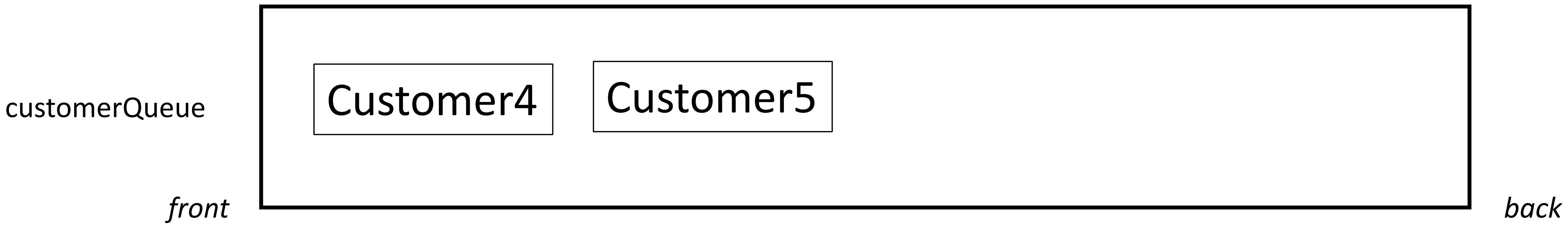
Example: A Customer Queue

- Customer1: Time 5
- Customer2: Time 6
- Customer3: Time 8
- Customer4: Time 9
- Customer5: Time 10

Time: 15, 16

Customer Being Helped:

Customer4 (started at 14)



customerQueue

Customer4

Customer5

front

back

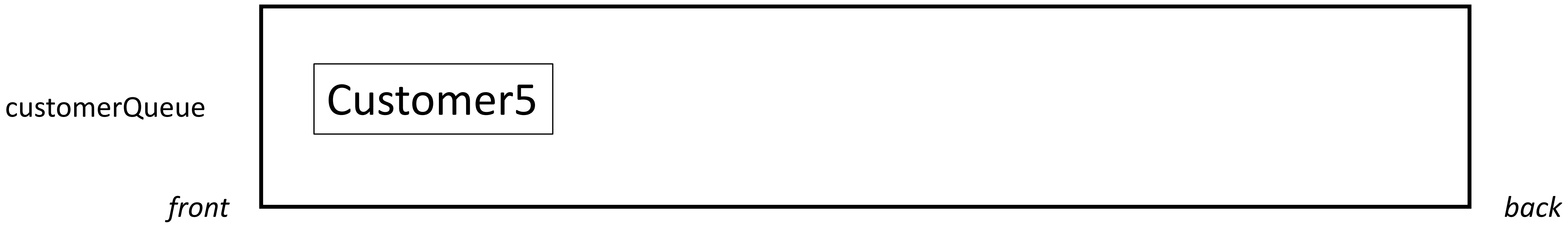
Example: A Customer Queue

- Customer1: Time 5
- Customer2: Time 6
- Customer3: Time 8
- Customer4: Time 9
- Customer5: Time 10

Time: 17

Customer Being Helped:

Customer5 (started at 17)



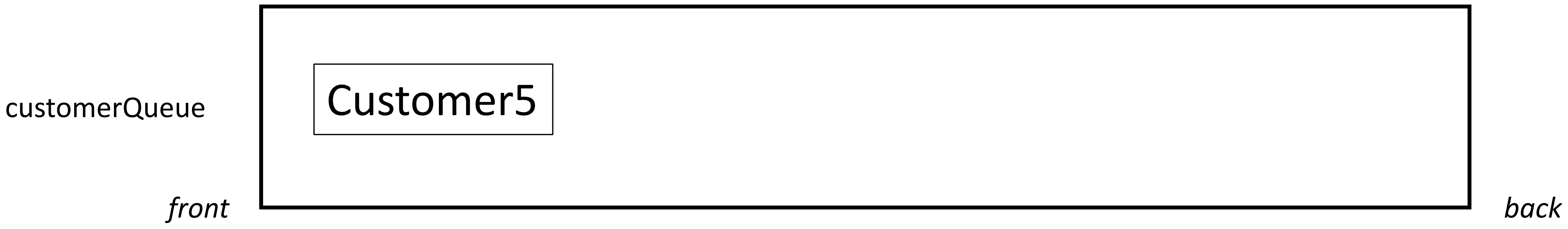
Example: A Customer Queue

- Customer1: Time 5
- Customer2: Time 6
- Customer3: Time 8
- Customer4: Time 9
- Customer5: Time 10

Time: 18, 19

Customer Being Helped:

Customer5 (started at 17)



customerQueue

Customer5

front

back

Example: A Customer Queue

- Customer1: Time 5
- Customer2: Time 6
- Customer3: Time 8
- Customer4: Time 9
- Customer5: Time 10

Time: 20

Customer Being Helped:

customerQueue

front

back

The Deque ADT

- Another kind of queue
- A double-ended queue
- Pronounced “deck”
- Allows adding and removing from both the front and the back
- Kind of like queue and stack functionality all in one data structure
- Still more restrictive than a list because no access to the middle elements

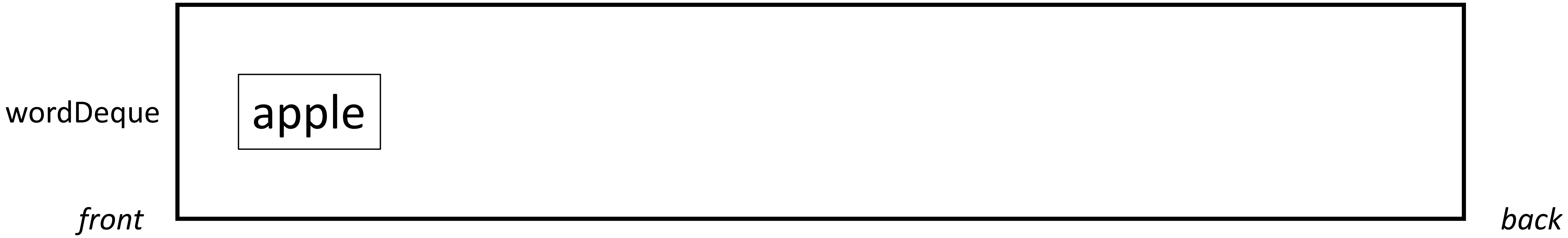
Deque Methods

- to add
  - addTo Front (like insertHead)
  - addToBack (like insertTail)
- to remove
  - removeFront (like deleteHead)
  - removeBack (like deleteTail)
- view
  - getFront
  - getBack

Example Uses of Deques

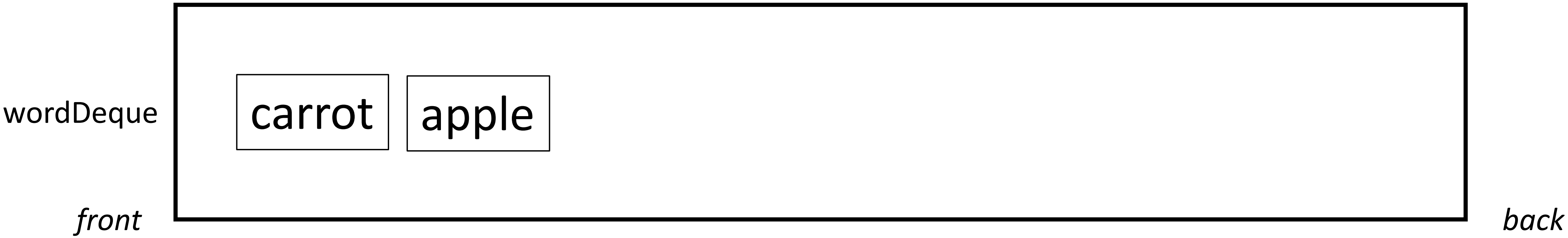
- Example: browser history
- Example: an undo button
- For both of these, you want the most recent item retrieved first.
    - This is LIFO behavior like a stack.
    - Use addFront
- However, you can only keep track of a limited number of elements.
    - When it’s time to drop elements, you want to drop the oldest ones.
    - Use removeBack

Example



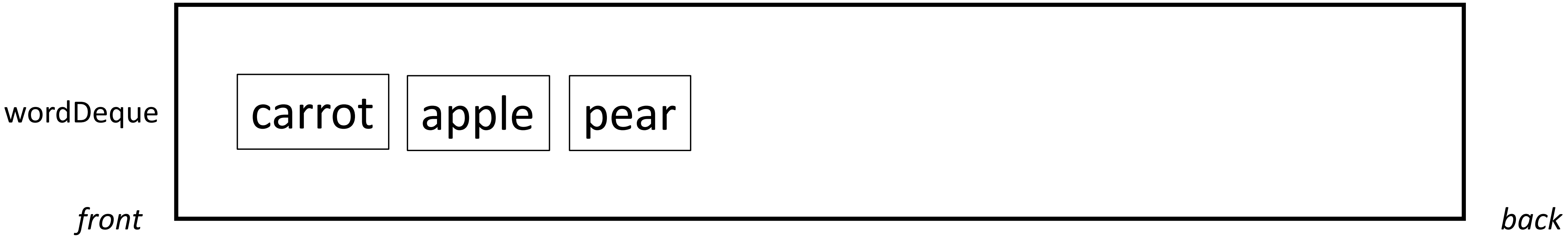
```
wordDeque.addFront(“apple”);
wordDeque.addFront(“carrot”);
wordDeque.addBack(“pear”);
wordDeque.addBack(“banana”);
wordDeque.addFront(“orange”);
System.out.println(wordDeque.getFront());
System.out.println(wordDeque.getBack());
System.out.println(wordDeque.removeFront());
System.out.println(wordDeque.removeBack());
```

Example



```
wordDeque.addFront("apple");
wordDeque.addFront("carrot");
wordDeque.addBack("pear");
wordDeque.addBack("banana");
wordDeque.addFront("orange");
System.out.println(wordDeque.getFront());
System.out.println(wordDeque.getBack());
System.out.println(wordDeque.removeFront());
System.out.println(wordDeque.removeBack());
```

Example



wordDeque

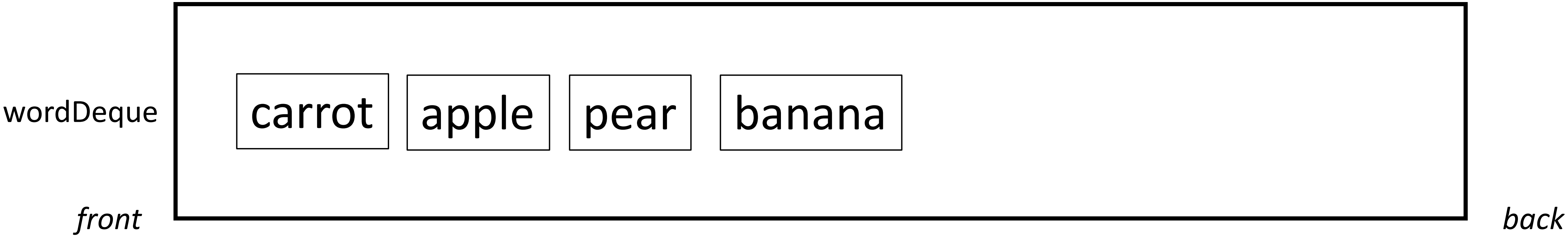
carrot apple pear

front

back

```
wordDeque.addFront(“apple”);
wordDeque.addFront(“carrot”);
wordDeque.addBack(“pear”);
wordDeque.addBack(“banana”);
wordDeque.addFront(“orange”);
System.out.println(wordDeque.getFront());
System.out.println(wordDeque.getBack());
System.out.println(wordDeque.removeFront());
System.out.println(wordDeque.removeBack());
```

Example



wordDeque

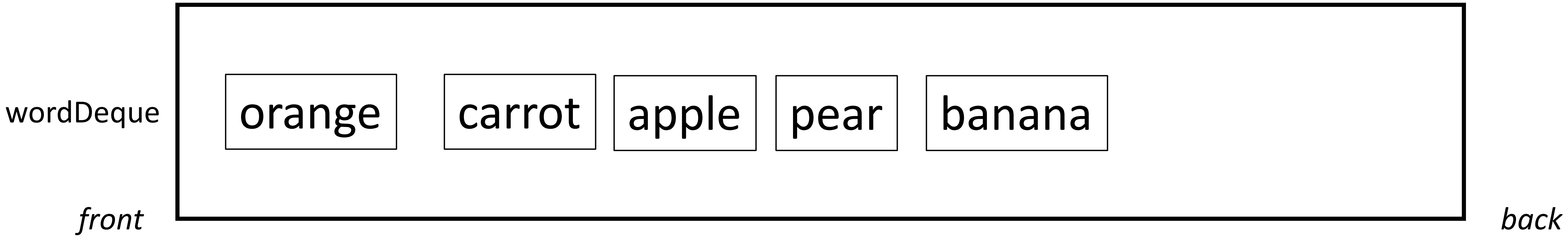
carrot apple pear banana

front

back

```
wordDeque.addFront(“apple”);
wordDeque.addFront(“carrot”);
wordDeque.addBack(“pear”);
wordDeque.addBack(“banana”);
wordDeque.addFront(“orange”);
System.out.println(wordDeque.getFront());
System.out.println(wordDeque.getBack());
System.out.println(wordDeque.removeFront());
System.out.println(wordDeque.removeBack());
```

Example



wordDeque

orange

carrot

apple

pear

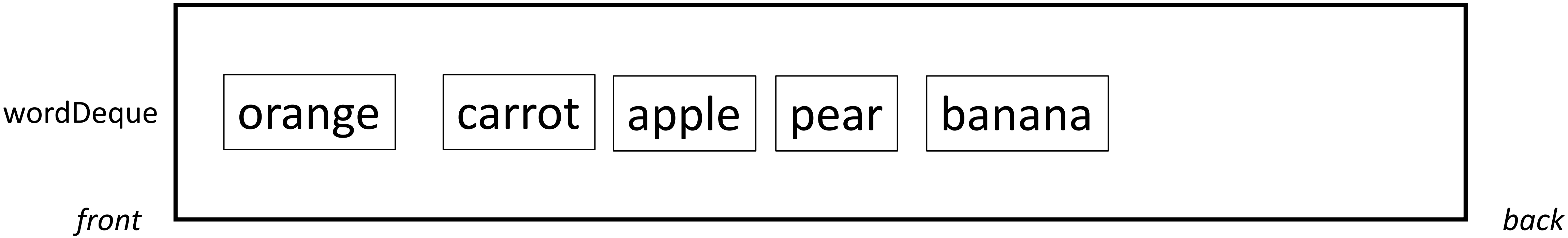
banana

front

back

```
wordDeque.addFront(“apple”);
wordDeque.addFront(“carrot”);
wordDeque.addBack(“pear”);
wordDeque.addBack(“banana”);
wordDeque.addFront(“orange”);
System.out.println(wordDeque.getFront());
System.out.println(wordDeque.getBack());
System.out.println(wordDeque.removeFront());
System.out.println(wordDeque.removeBack());
```

Example



wordDeque

orange carrot apple pear banana

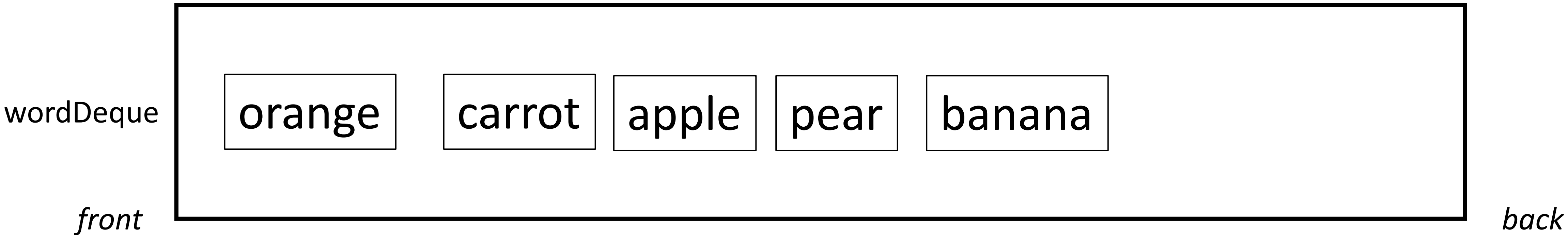
front

back

```
wordDeque.addFront(“apple”);
wordDeque.addFront(“carrot”);
wordDeque.addBack(“pear”);
wordDeque.addBack(“banana”);
wordDeque.addFront(“orange”);
System.out.println(wordDeque.getFront());
System.out.println(wordDeque.getBack());
System.out.println(wordDeque.removeFront());
System.out.println(wordDeque.removeBack());
```

prints: orange

Example



wordDeque

orange carrot apple pear banana

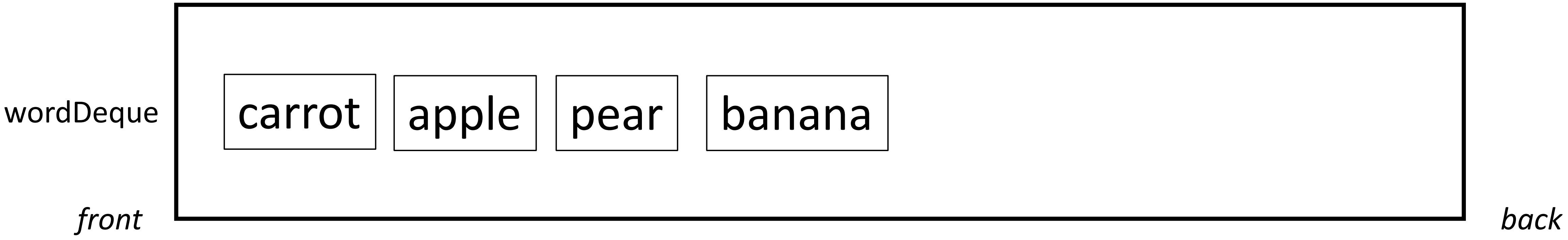
front

back

```
wordDeque.addFront("apple");
wordDeque.addFront("carrot");
wordDeque.addBack("pear");
wordDeque.addBack("banana");
wordDeque.addFront("orange");
System.out.println(wordDeque.getFront());
System.out.println(wordDeque.getBack());
System.out.println(wordDeque.removeFront());
System.out.println(wordDeque.removeBack());
```

prints: banana

Example



wordDeque

carrot apple pear banana

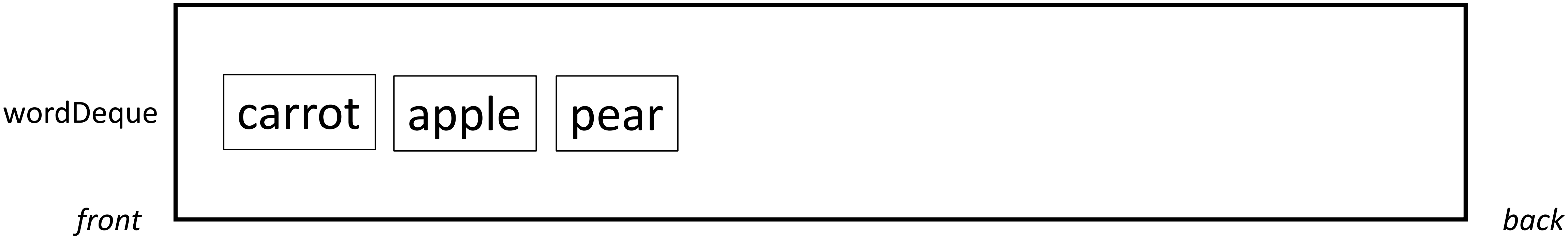
front

back

```
wordDeque.addFront(“apple”);
wordDeque.addFront(“carrot”);
wordDeque.addBack(“pear”);
wordDeque.addBack(“banana”);
wordDeque.addFront(“orange”);
System.out.println(wordDeque.getFront());
System.out.println(wordDeque.getBack());
System.out.println(wordDeque.removeFront());
System.out.println(wordDeque.removeBack());
```

prints: orange

Example



wordDeque

carrot apple pear

front

back

```
wordDeque.addFront("apple");
wordDeque.addFront("carrot");
wordDeque.addBack("pear");
wordDeque.addBack("banana");
wordDeque.addFront("orange");
System.out.println(wordDeque.getFront());
System.out.println(wordDeque.getBack());
System.out.println(wordDeque.removeFront());
System.out.println(wordDeque.removeBack());
```

prints: banana

Priority Queue

- Another kind of queue
- Elements are added/ordered based on some type of ordering
- Elements are still removed from the front
  - This will be the element with the “highest priority”

One Kind of Priority Queue

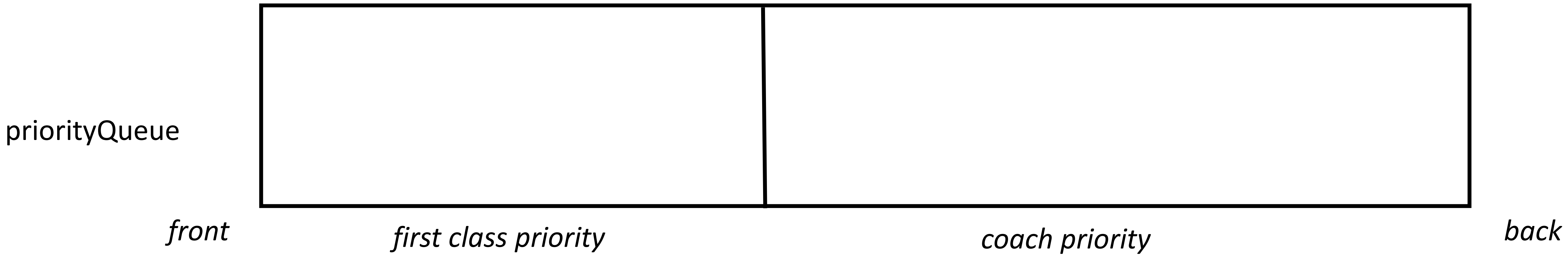
- A priority queue can also use *layered* ordering
- Elements are added **first** based on priority and only after that based on chronological ordering.
- Chronological ordering is significant only for items with the same priority.
- Priority queues retrieve the element with the highest priority; and the earliest element with that priority.

Example: Airplane Boarding

- Flyers line up to board in a first class or coach group.
  - They get in order within their appropriate line.
- To board, the plane boards all first class passengers first (in the order they got in line), followed by all coach passengers (in the order they got in line).

Example: Airplane Boarding

```
priorityQueue.add(first class flyer a)
priorityQueue.add(coach flyer b)
priorityQueue.add(coach flyer c)
priorityQueue.add(first class flyer d)
priorityQueue.add(coach flyer e)
priorityQueue.add(first class flyer f)
priorityQueue.add(coach flyer g)
```



priorityQueue

front

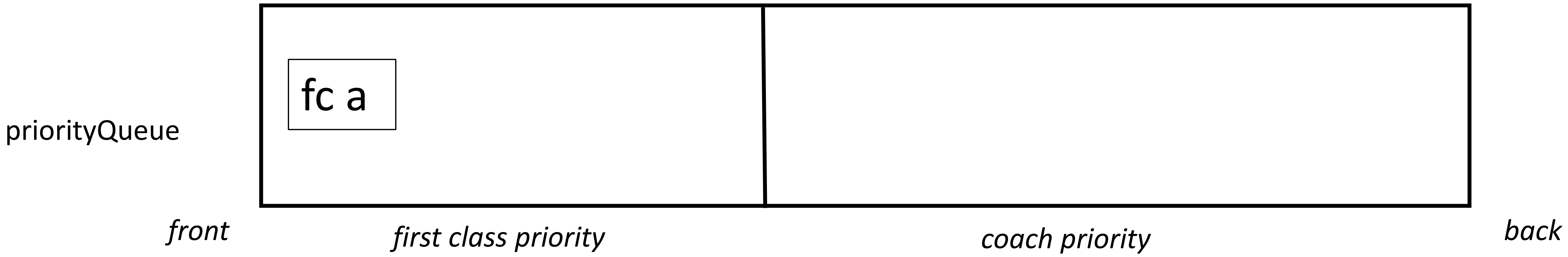
first class priority

coach priority

back

Example: Airplane Boarding

```
priorityQueue.add(first class flyer a)
priorityQueue.add(coach flyer b)
priorityQueue.add(coach flyer c)
priorityQueue.add(first class flyer d)
priorityQueue.add(coach flyer e)
priorityQueue.add(first class flyer f)
priorityQueue.add(coach flyer g)
```



priorityQueue

fc a

front

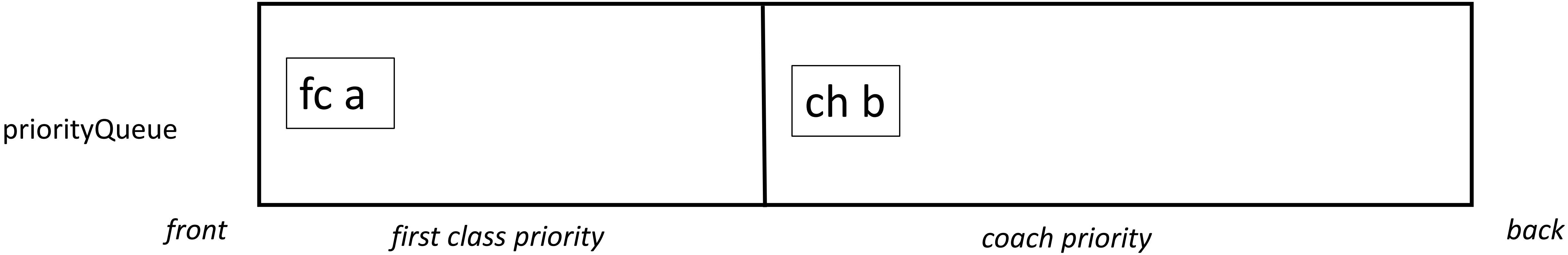
first class priority

coach priority

back

Example: Airplane Boarding

```
priorityQueue.add(first class flyer a)
priorityQueue.add(coach flyer b)
priorityQueue.add(coach flyer c)
priorityQueue.add(first class flyer d)
priorityQueue.add(coach flyer e)
priorityQueue.add(first class flyer f)
priorityQueue.add(coach flyer g)
```



priorityQueue

fc a

ch b

front

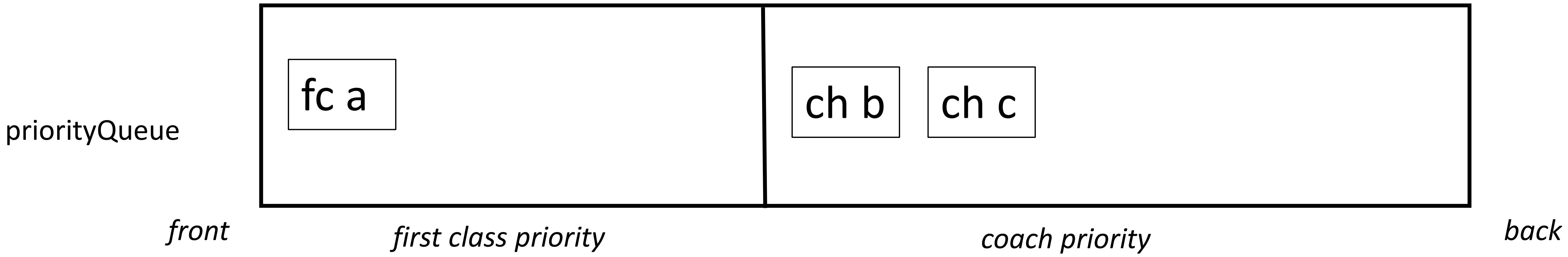
first class priority

coach priority

back

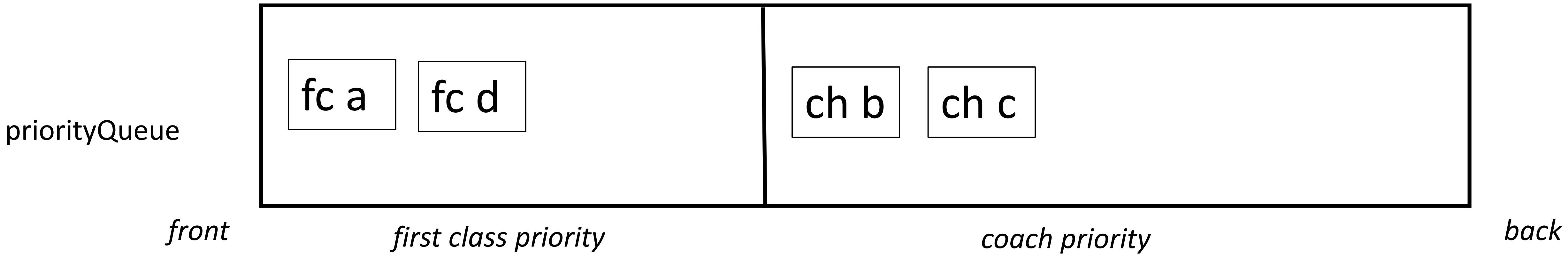
Example: Airplane Boarding

```
priorityQueue.add(first class flyer a)
priorityQueue.add(coach flyer b)
priorityQueue.add(coach flyer c)
priorityQueue.add(first class flyer d)
priorityQueue.add(coach flyer e)
priorityQueue.add(first class flyer f)
priorityQueue.add(coach flyer g)
```



Example: Airplane Boarding

priorityQueue.add(first class flyer a)
priorityQueue.add(coach flyer b)
priorityQueue.add(coach flyer c)
priorityQueue.add(first class flyer d)
priorityQueue.add(coach flyer e)
priorityQueue.add(first class flyer f)
priorityQueue.add(coach flyer g)



priorityQueue

fc a

fc d

ch b

ch c

front

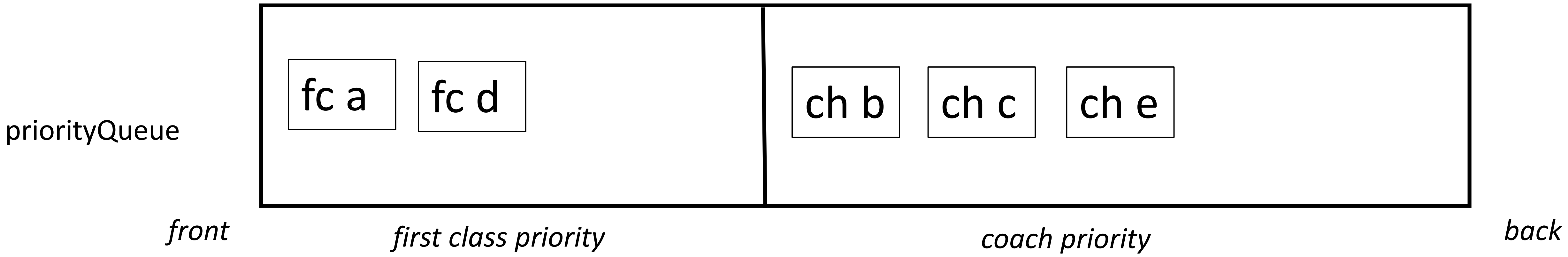
first class priority

coach priority

back

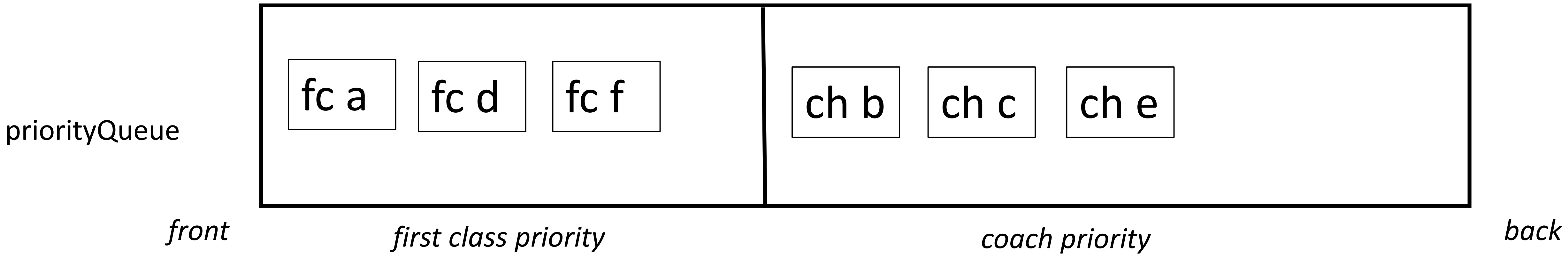
Example: Airplane Boarding

```
priorityQueue.add(first class flyer a)
priorityQueue.add(coach flyer b)
priorityQueue.add(coach flyer c)
priorityQueue.add(first class flyer d)
priorityQueue.add(coach flyer e)
priorityQueue.add(first class flyer f)
priorityQueue.add(coach flyer g)
```



Example: Airplane Boarding

```
priorityQueue.add(first class flyer a)
priorityQueue.add(coach flyer b)
priorityQueue.add(coach flyer c)
priorityQueue.add(first class flyer d)
priorityQueue.add(coach flyer e)
priorityQueue.add(first class flyer f)
priorityQueue.add(coach flyer g)
```



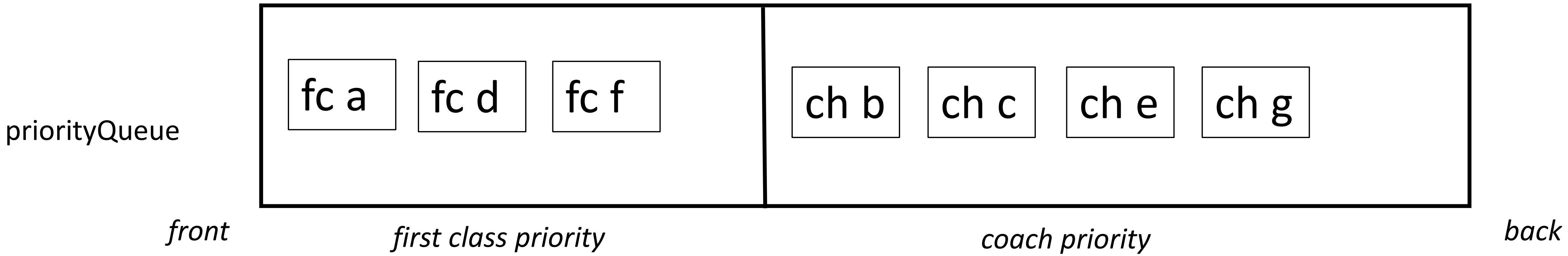
priorityQueue

fc a fc d fc f | ch b ch c ch e

front *first class priority* *coach priority* *back*

Example: Airplane Boarding

priorityQueue.add(first class flyer a)
priorityQueue.add(coach flyer b)
priorityQueue.add(coach flyer c)
priorityQueue.add(first class flyer d)
priorityQueue.add(coach flyer e)
priorityQueue.add(first class flyer f)
priorityQueue.add(coach flyer g)



priorityQueue

fc a fc d fc f

ch b ch c ch e ch g

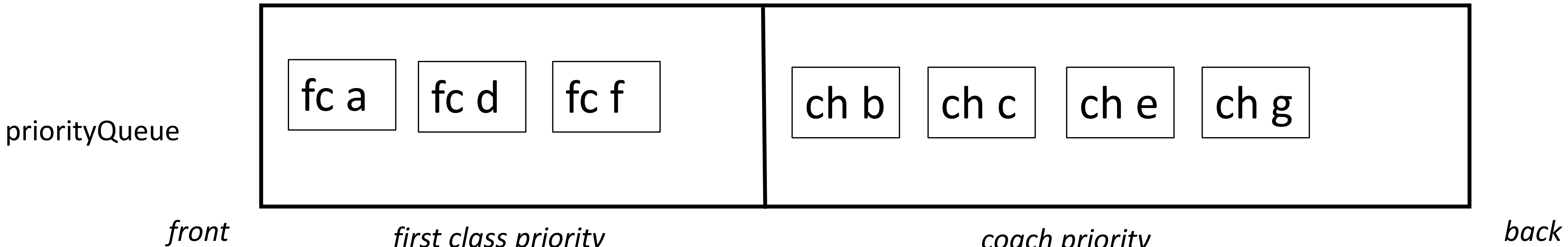
front *first class priority* *coach priority* *back*

Example: Airplane Boarding

boarding order:

(remove one at a time from the front of the whole queue)

first class flyer a
first class flyer d
first class flyer f
coach flyer b
coach flyer c
coach flyer e
coach flyer g



Implementing a Priority Queue

- If elements with different values always have different priorities, then a priority queue can be implemented with a sorted list.
  - Elements are maintained in “sorted” or “prioritized” order.
  - We still only remove from the front of the queue!
- Example: Strings prioritized alphabetically.

Implementing Queues with Nodes

- Implementing queues with linked nodes is straightforward.
  - Assuming you have a head AND tail pointer so you have access to both the front and back of the chain.
- firstNode (or head) is the front of the queue
- lastNode (or tail) is the back of the queue
- Enqueues are easy and O(1)
  - tail.next = newNode;
  - tail = newNode;
- Dequeues are easy and O(1)
  - currentData = first.data;
  - first = first.next;
- Very efficient!

Implementing Queues with Circular Linked Nodes

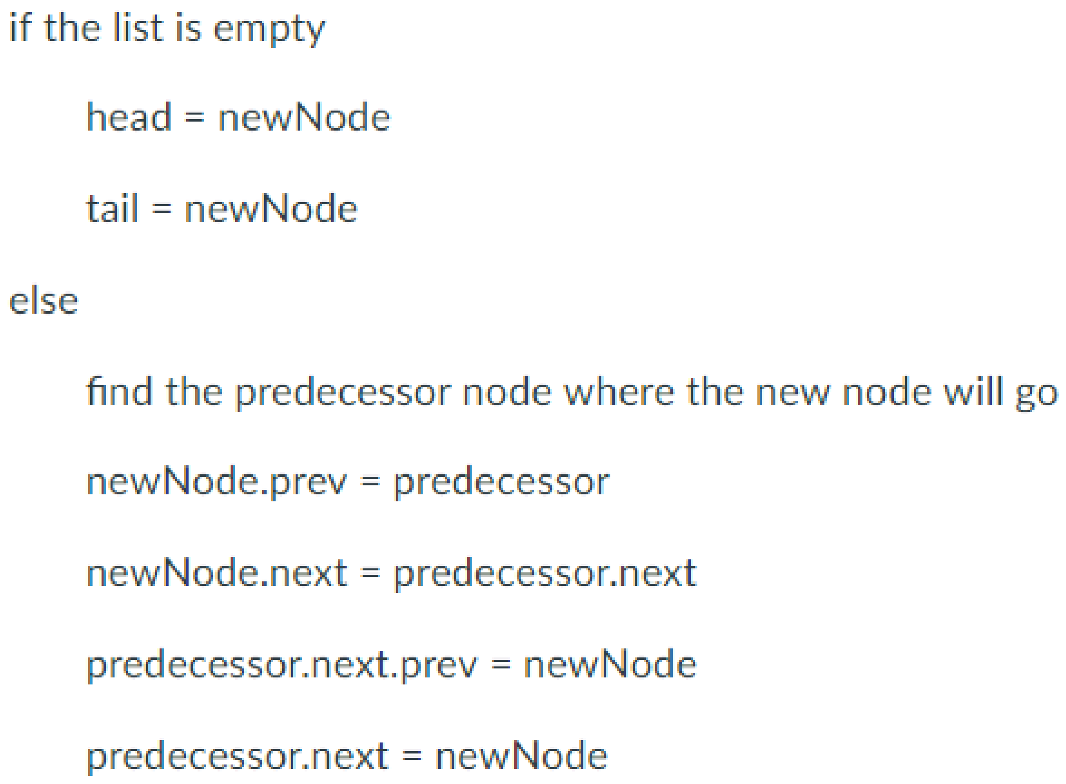
- Only keep track of tail (back)
- The head (front) is tail.next

Implementing Queues with Doubly Linked Nodes

- Use doubly-linked nodes
  - A doubly-linked node keeps track of previous **and** next
  - current == current.next.prev
  - current == current.prev.next
- These are good for deques because removeBack is O(1) instead of O(n)
  - So all adds and removes are O(1)

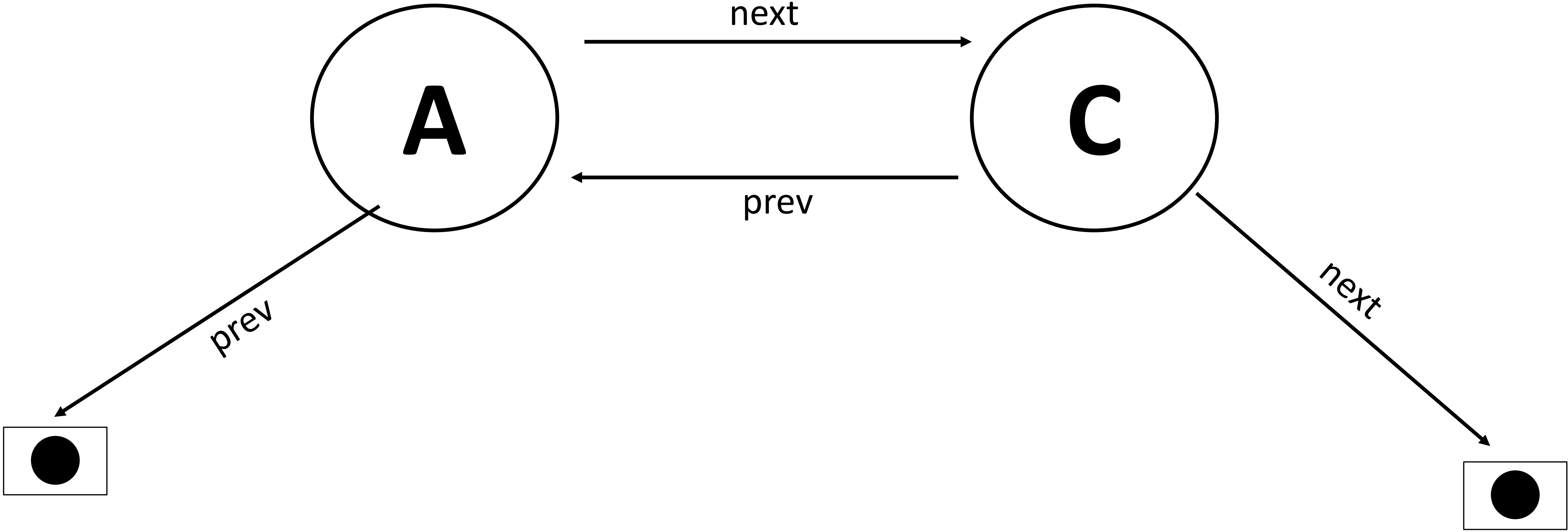
Using Doubly Linked Nodes

- Pseudocode for adding an element to a chain of doubly linked nodes
- This is not an example for a queue- just a general example of adding to a chain

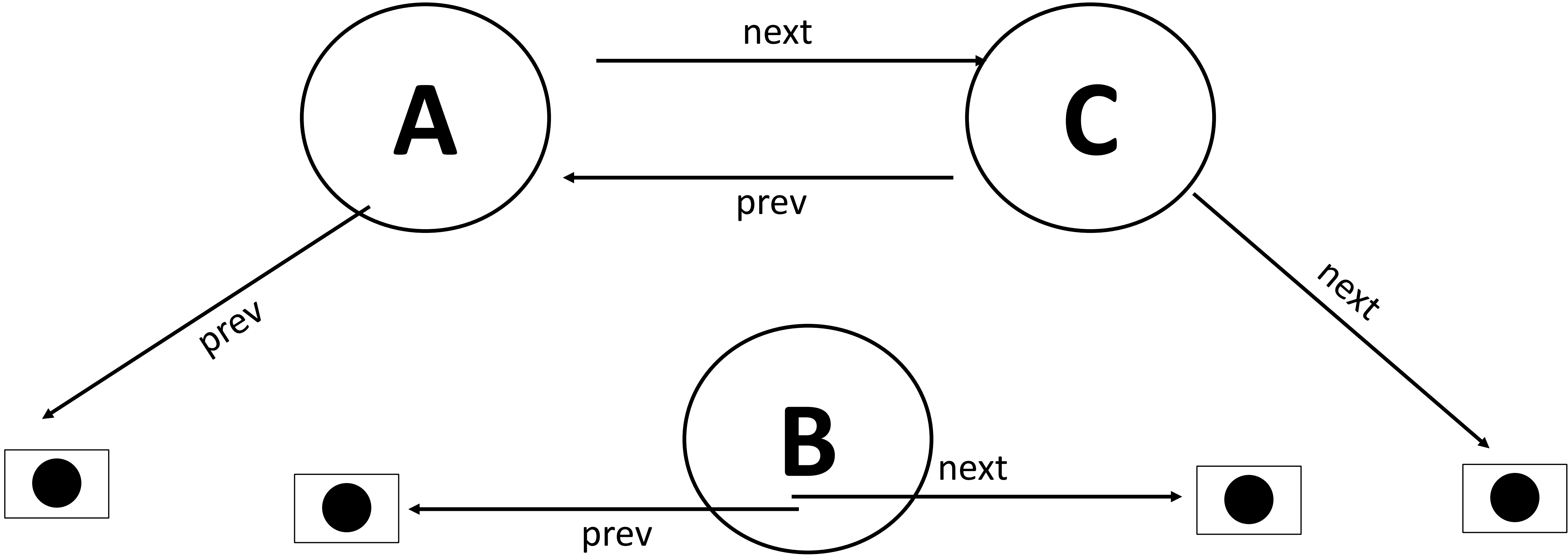


```
if the list is empty
    head = newNode
    tail = newNode
else
    find the predecessor node where the new node will go
    newNode.prev = predecessor
    newNode.next = predecessor.next
    predecessor.next.prev = newNode
    predecessor.next = newNode
```

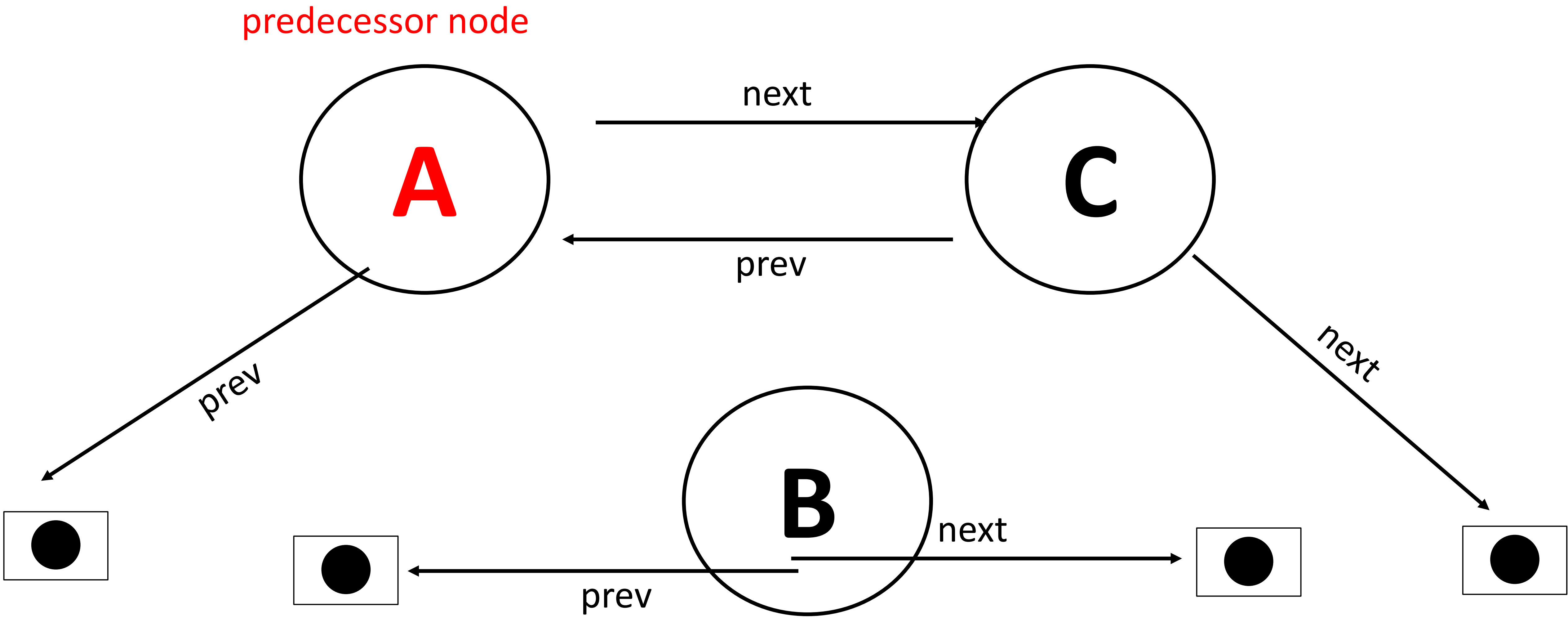
Example Trace of Doubly Linked Nodes



Example Trace of Adding to Doubly Linked Nodes

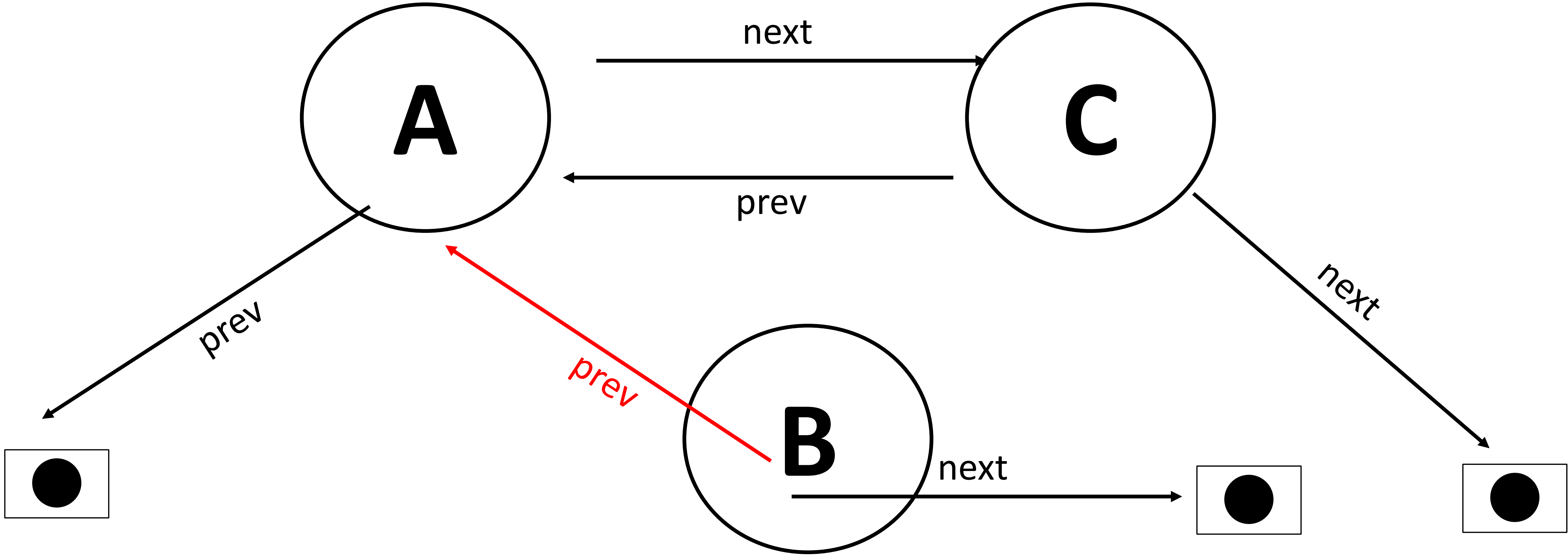


Example Trace of Adding to Doubly Linked Nodes



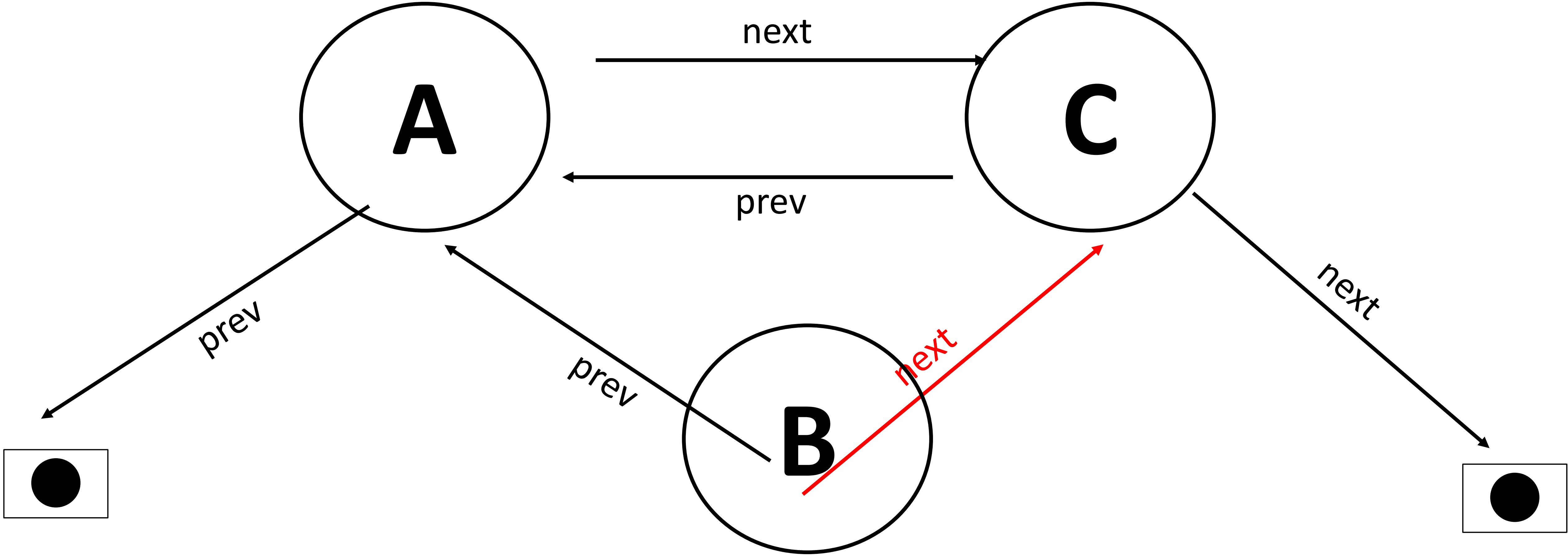
Example Trace of Adding to Doubly Linked Nodes

newNode.prev = predecessor



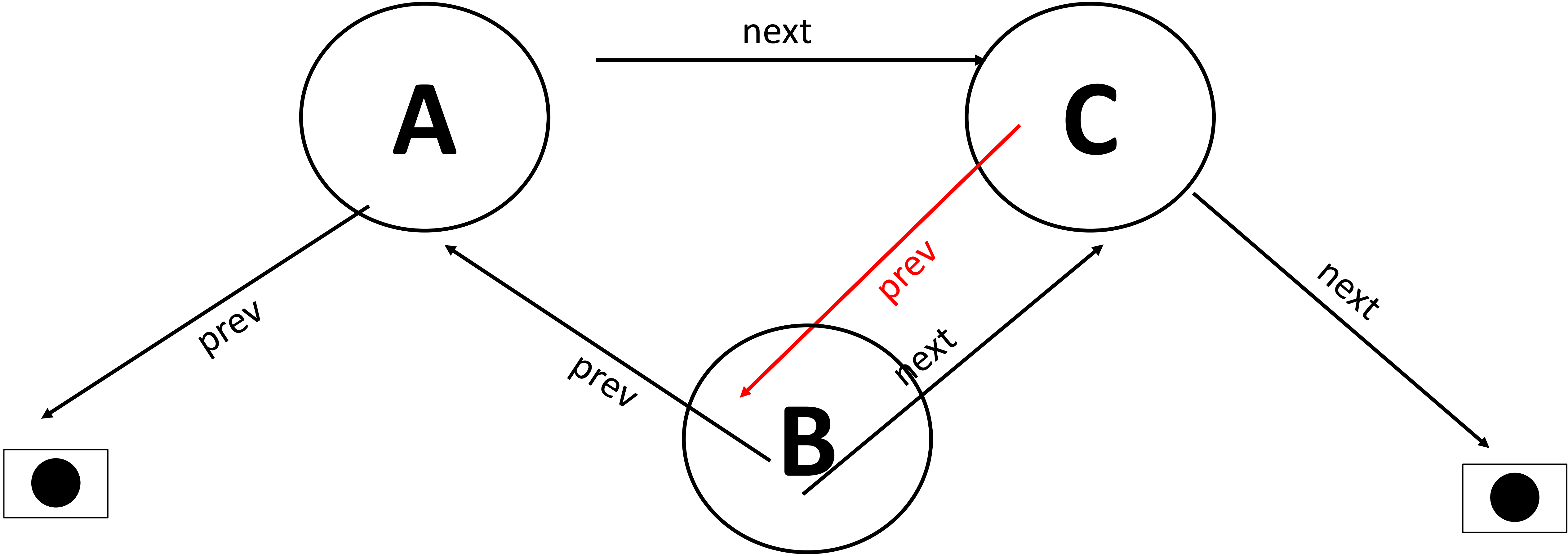
Example Trace of Adding to Doubly Linked Nodes

newNode.next = predecessor.next



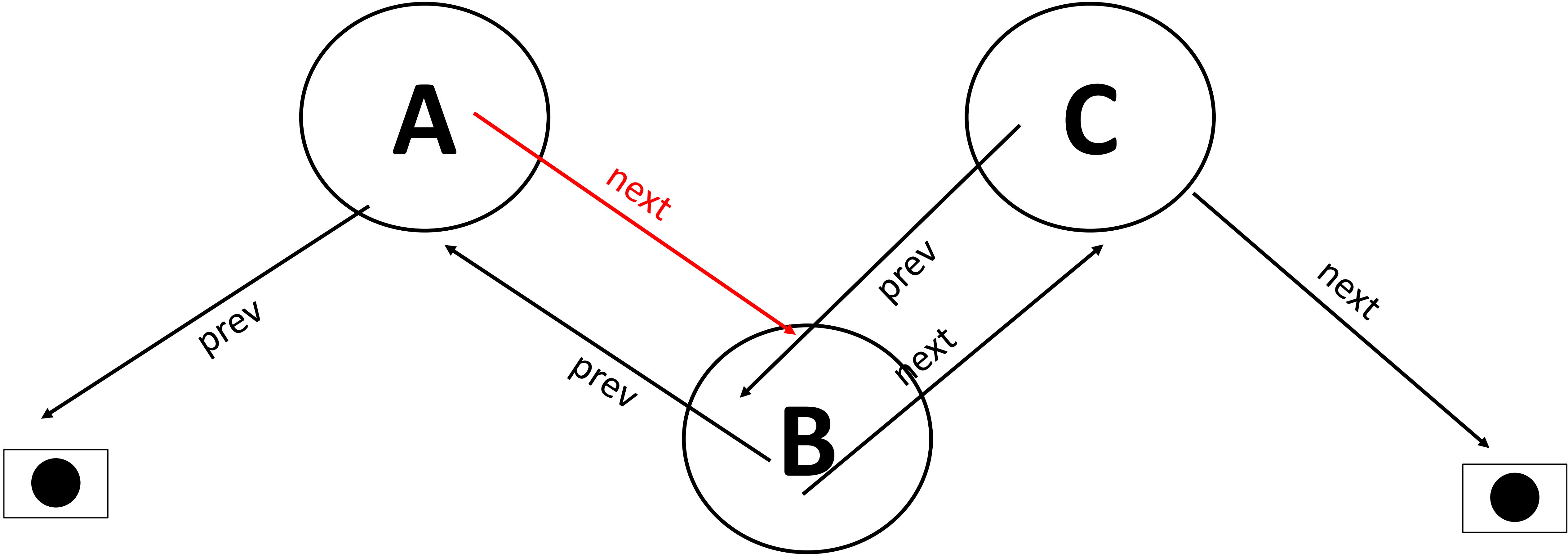
Example Trace of Adding to Doubly Linked Nodes

predecessor.next.prev = newNode



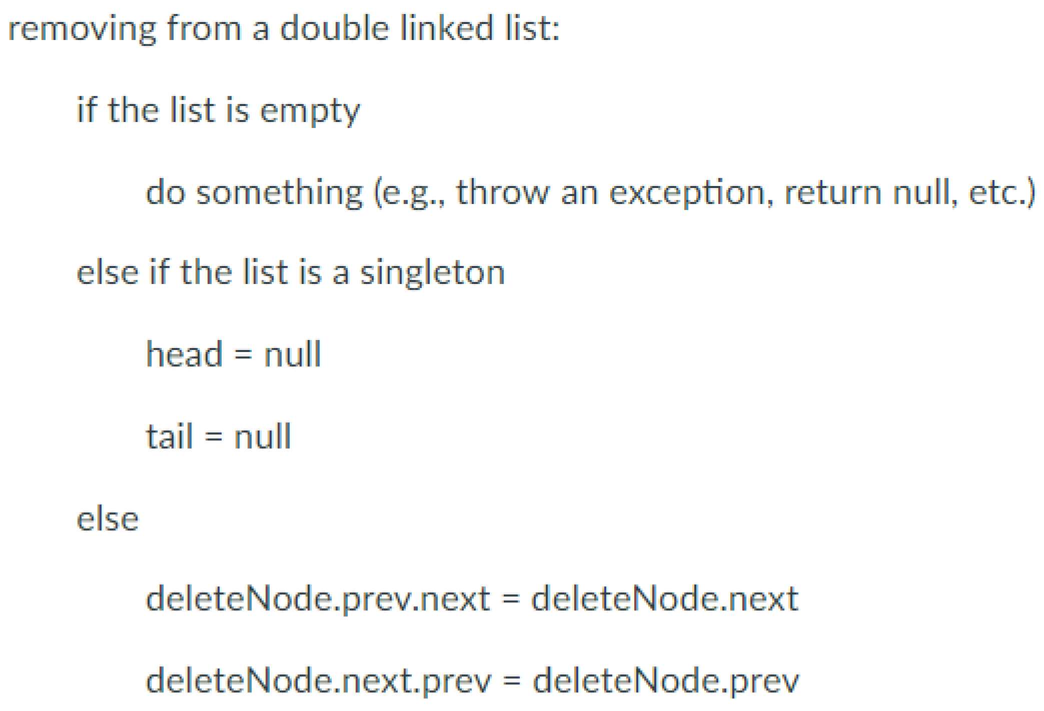
Example Trace of Adding to Doubly Linked Nodes

predecessor.next = newNode



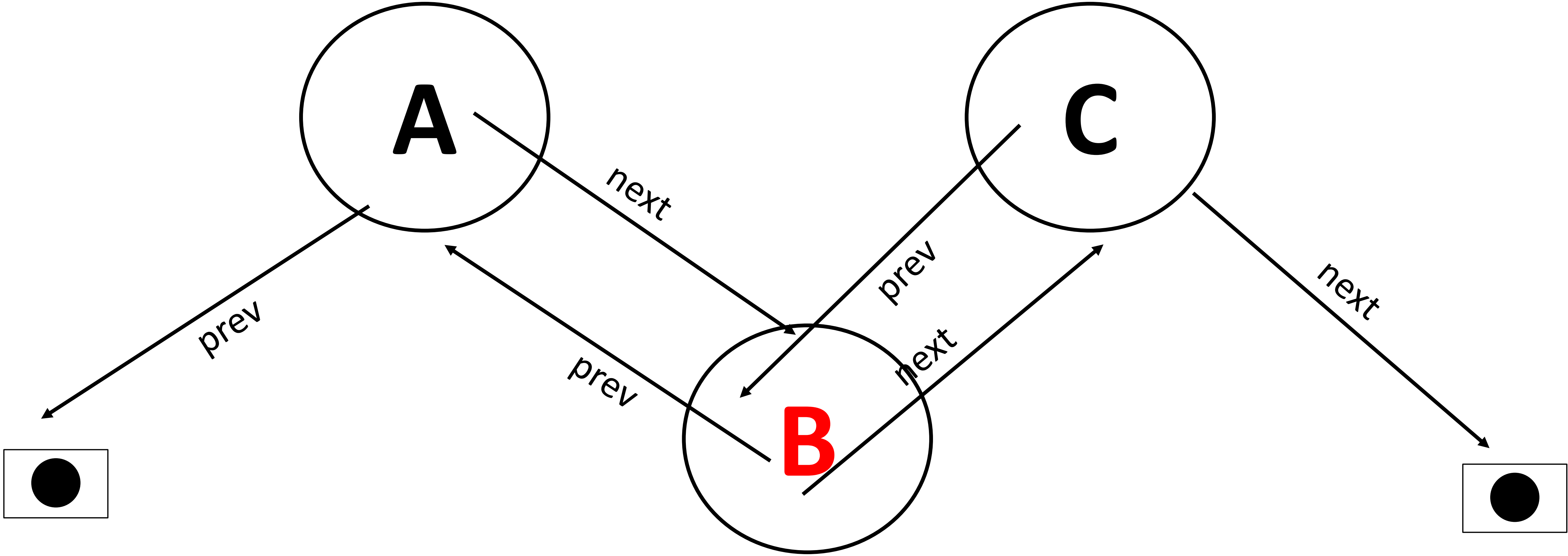
Using Doubly Linked Nodes

- Pseudocode for removing an element from a chain of doubly linked nodes
- This is not an example for a queue- just a general example of removing from a chain



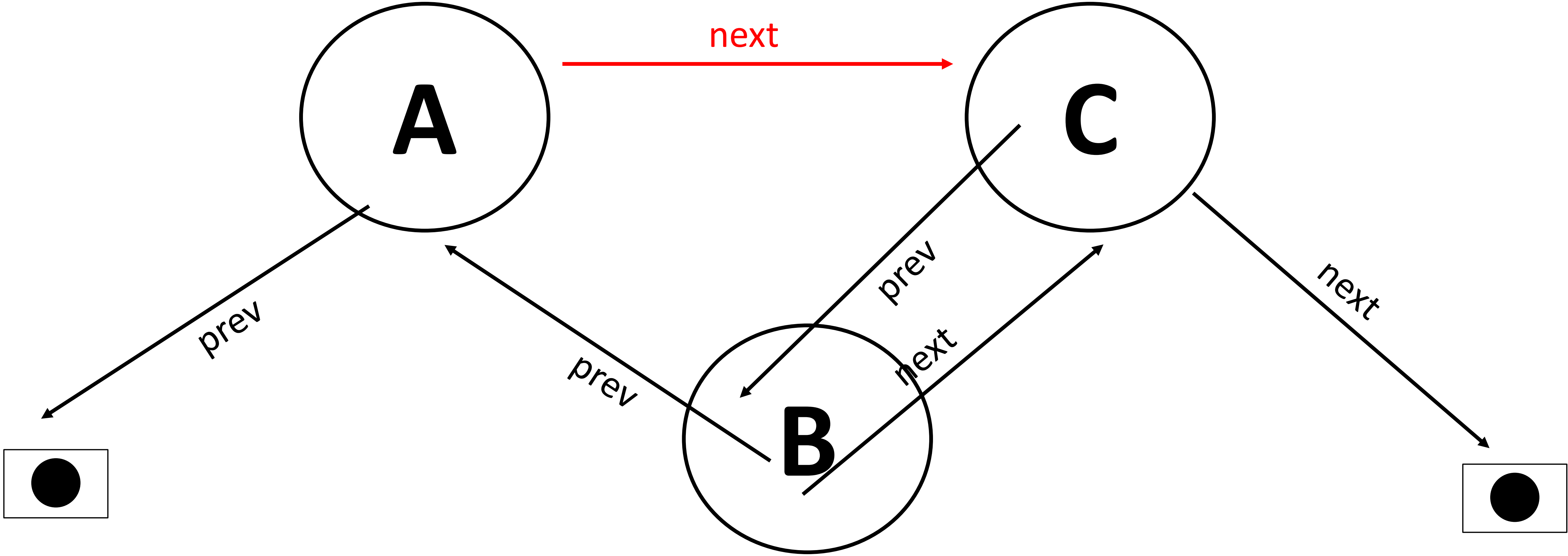
```
removing from a double linked list:
    if the list is empty
        do something (e.g., throw an exception, return null, etc.)
    else if the list is a singleton
        head = null
        tail = null
    else
        deleteNode.prev.next = deleteNode.next
        deleteNode.next.prev = deleteNode.prev
```

Example Trace of Removing



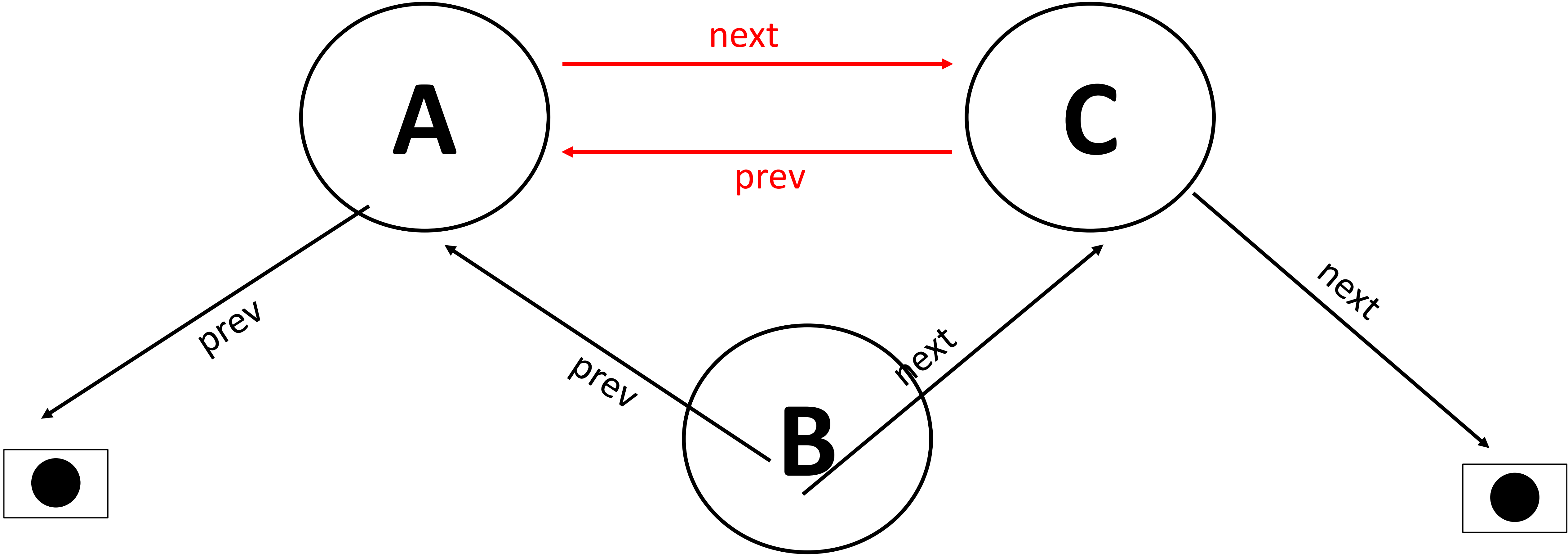
Example Trace of Removing

deleteNode.prev.next = deleteNode.next

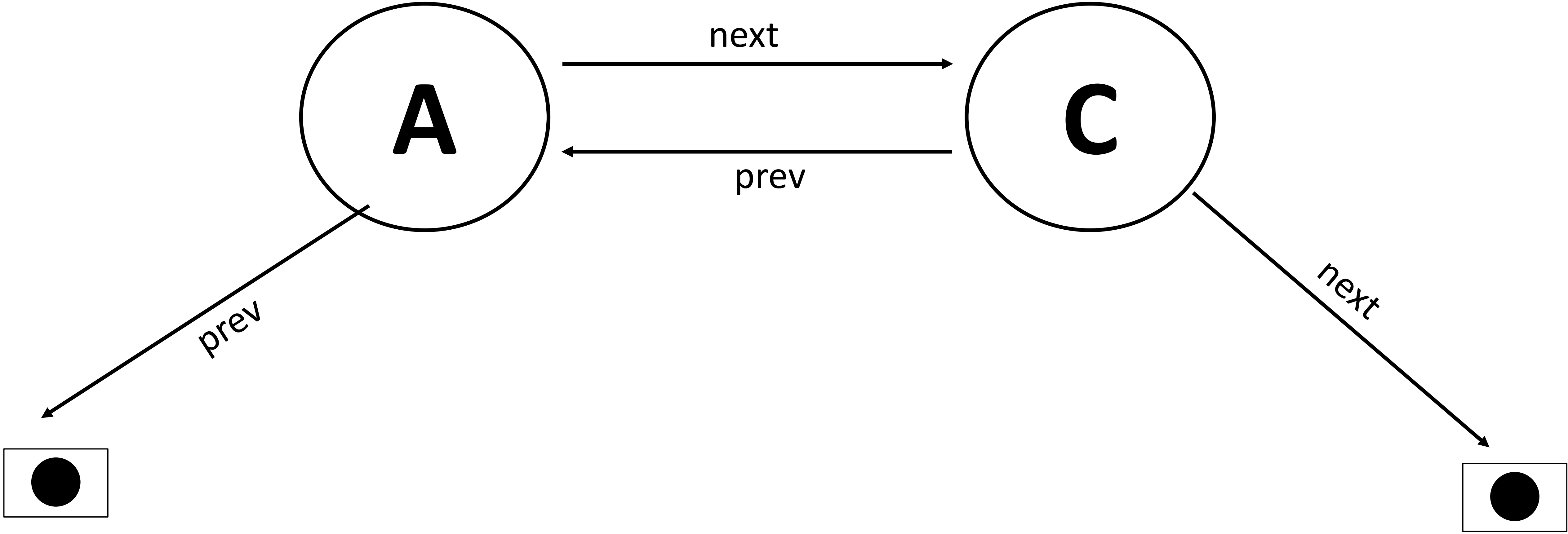


Example Trace of Removing

deleteNode.next.prev = deleteNode.prev



Example Trace of Removing



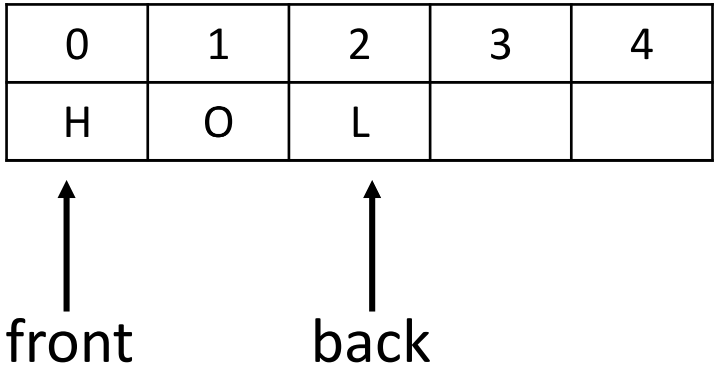
Implementing Queues with an Array

- Similar to a stack, this is a little trickier.

- If we keep the front of the queue at position 0 and the back of the queue towards array.length-1, we would need to shift elements when an element is dequeued.
- This would make removals O(n) instead of O(1).

Implementing Queues with an Array

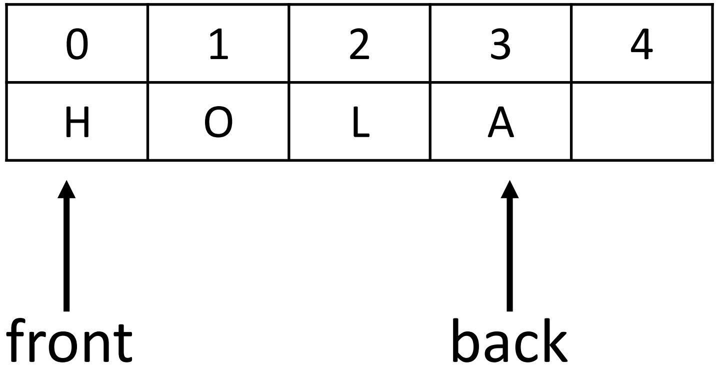
- Inefficient array implementation.



0	1	2	3	4
H	O	L		

Implementing Queues with an Array

- Inefficient array implementation.
- Enqueueing still O(1).



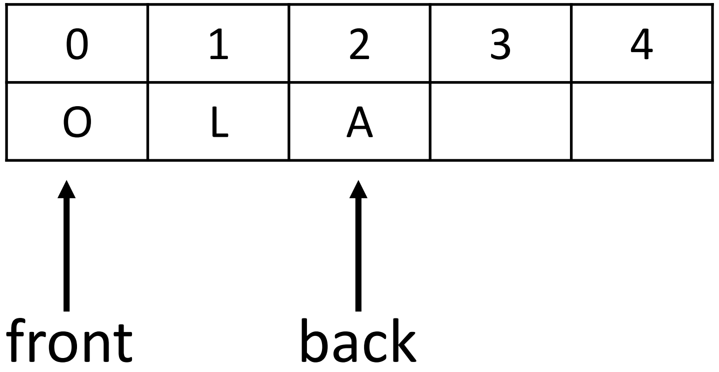
0	1	2	3	4
H	O	L	A	

front

back

Implementing Queues with an Array

- Inefficient array implementation.
- Dequeueing requires a shift!



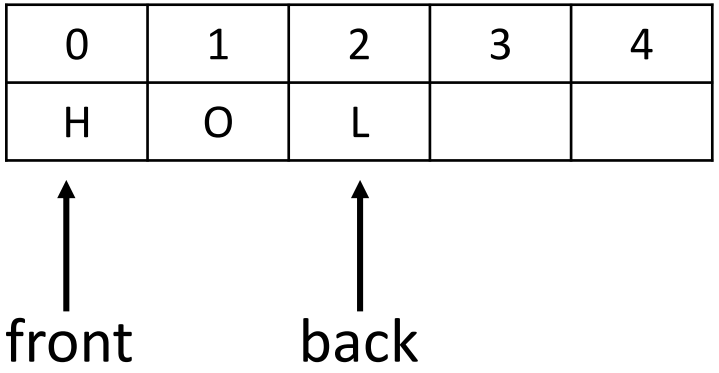
0	1	2	3	4
O	L	A		

front

back

Implementing Queues with an Array

- Instead, we can just update the index and keep track of which index holds the front of the queue and which index holds the back of the queue.



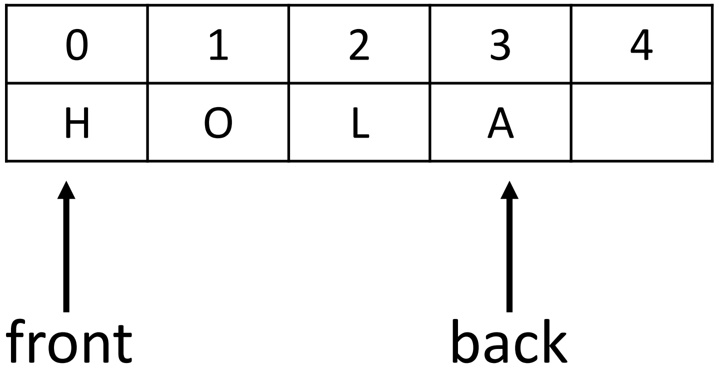
0	1	2	3	4
H	O	L		

front

back

Implementing Queues with an Array

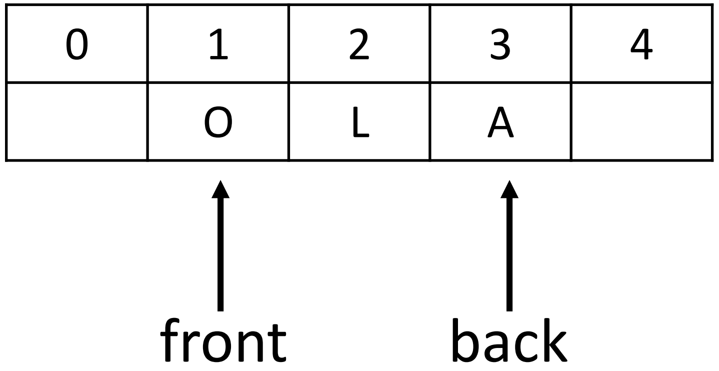
- Enqueueing still O(1)



0	1	2	3	4
H	O	L	A	

Implementing Queues with an Array

- Dequeueing is now also O(1)
  - We don’t shift anything! We just update front.



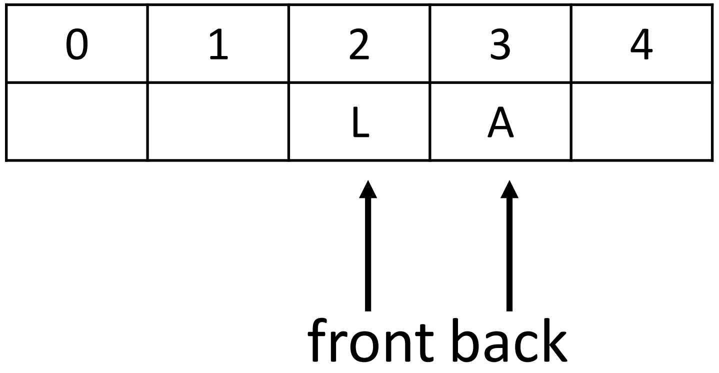
0	1	2	3	4
	O	L	A	

front

back

Implementing Queues with an Array

- Another dequeue

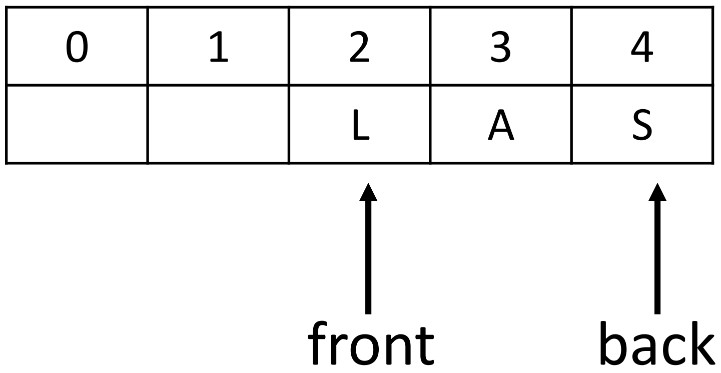


0	1	2	3	4
		L	A	

front back

Implementing Queues with an Array

- Another enqueue
- *Inchworm* effect- entries migrate to the end of the array



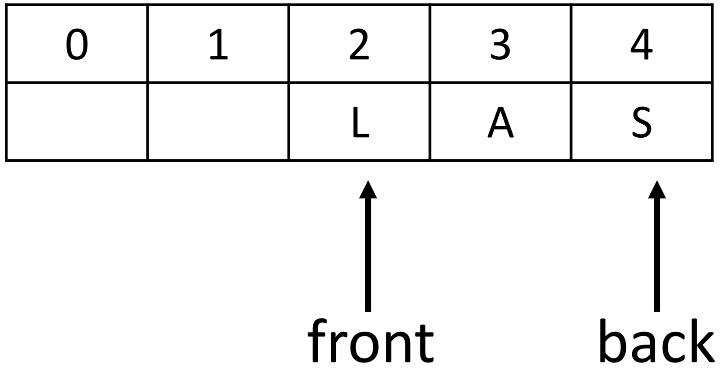
0	1	2	3	4
		L	A	S

front

back

Implementing Queues with an Array

- What if we want to enqueue X?
- No space left at the end. But there is space on the beginning!
- We can use a *circular array* to allow the elements to “wrap around.”
- Use the modulus operator to support this functionality.



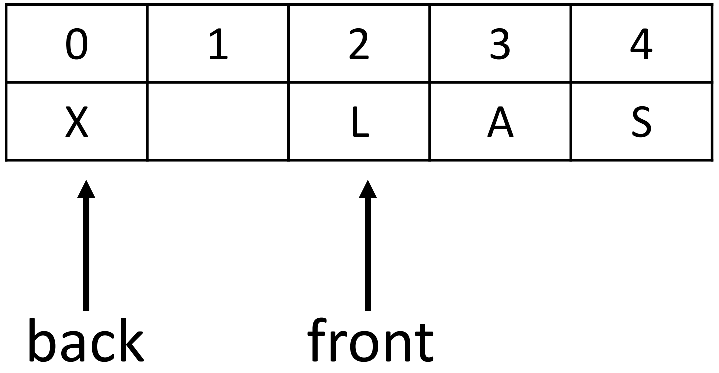
0	1	2	3	4
		L	A	S

Modulus Operator

- The remainder- what is left over after performing integer division
- Examples:
  - 10 % 3 = 1 because 10 / 3 = 3 with 1 left over
  - 4 % 9 = 4 because 4 / 9 = 0 with 4 leftover
  - 6 % 6 = 0 because 6 / 6 = 1 with 0 leftover

Implementing Queues with an Array

- Enqueue X
- back = (back + 1) % array.length
- back = (4+1) % 5



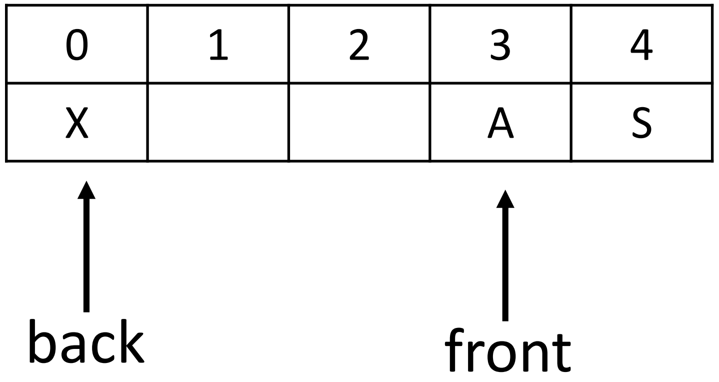
0	1	2	3	4
X		L	A	S

back

front

Implementing Queues with an Array

- Dequeue
- front = (front + 1) % array.length
- front = (2+1) % 5



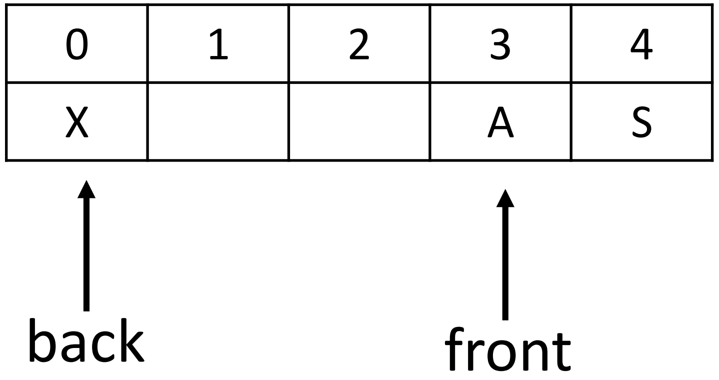
0	1	2	3	4
X			A	S

back

front

Implementing Queues with an Array

- Note that back can be less than front!!



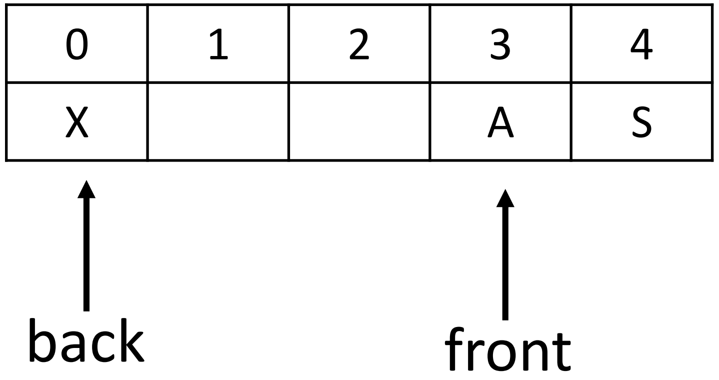
0	1	2	3	4
X			A	S

back

front

Implementing Queues with an Array

- How do we know when the array (and thus the queue) is full?



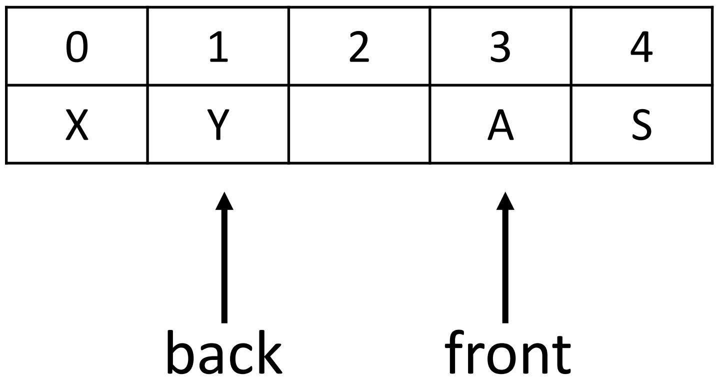
0	1	2	3	4
X			A	S

back

front

Implementing Queues with an Array

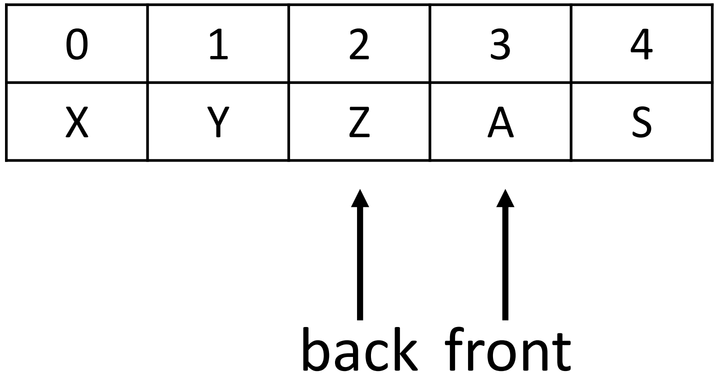
- Enqueue Y



0	1	2	3	4
X	Y		A	S

Implementing Queues with an Array

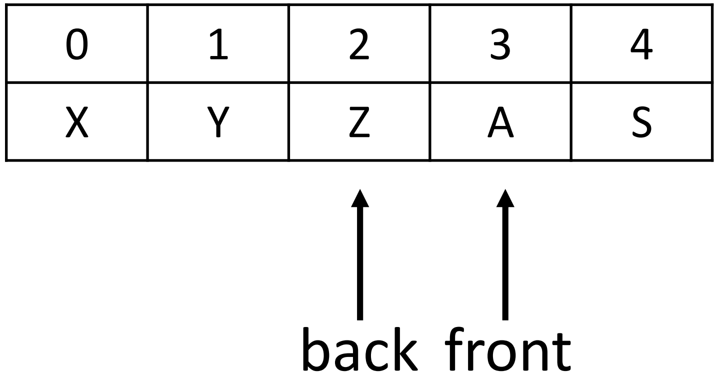
- Enqueue Z



0	1	2	3	4
X	Y	Z	A	S

Implementing Queues with an Array

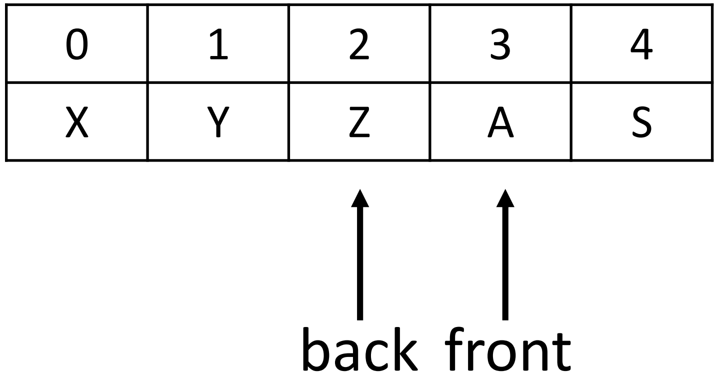
- Okay, so… can we determine if an array is full by testing:
    - front == (back+1) % length
- It works for this example! But…



0	1	2	3	4
X	Y	Z	A	S

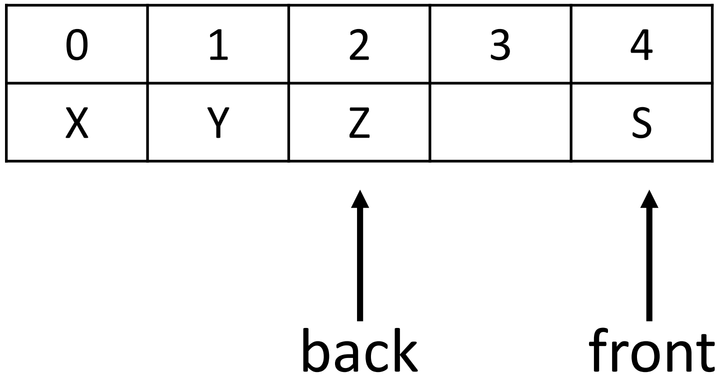
Implementing Queues with an Array

- Let’s dequeue everything to get an empty array (and queue).



0	1	2	3	4
X	Y	Z	A	S

Implementing Queues with an Array

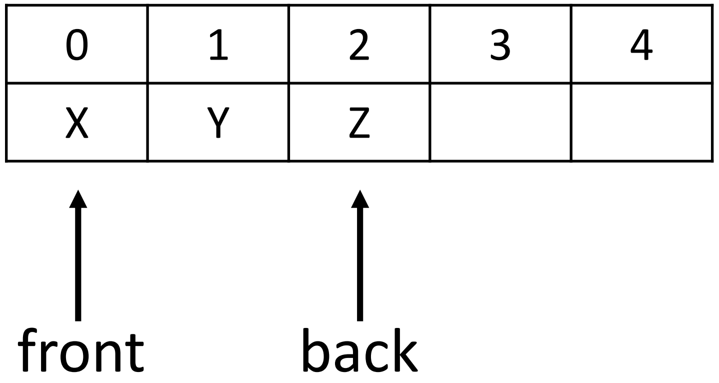


0	1	2	3	4
X	Y	Z		S

back

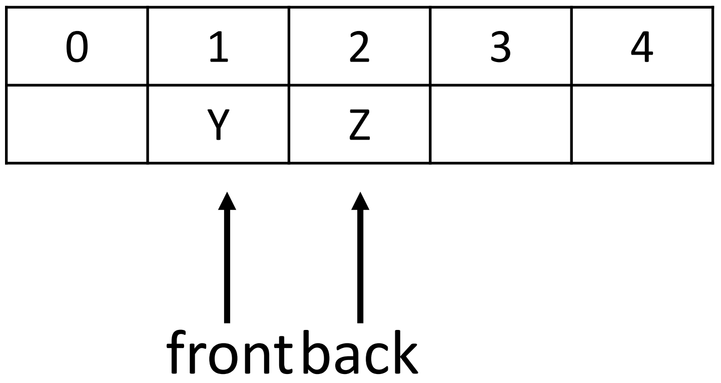
front

Implementing Queues with an Array



0	1	2	3	4
X	Y	Z		

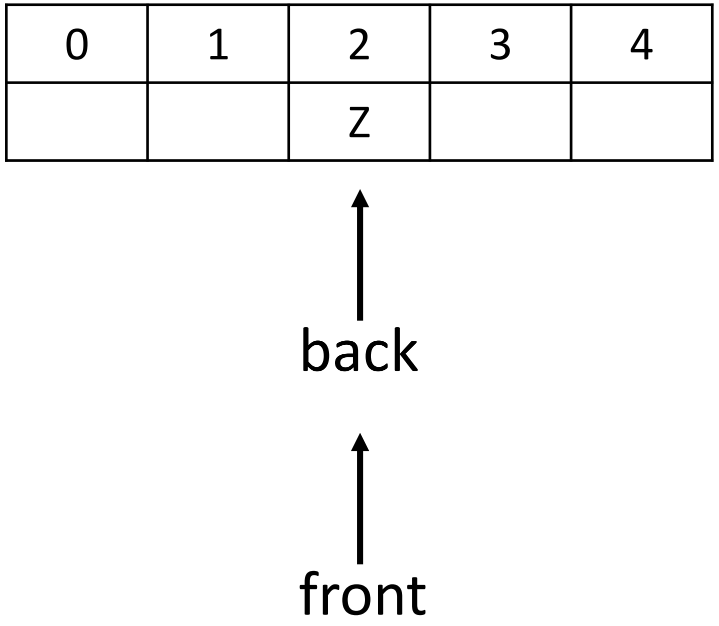
Implementing Queues with an Array



0	1	2	3	4
	Y	Z		

front back

Implementing Queues with an Array



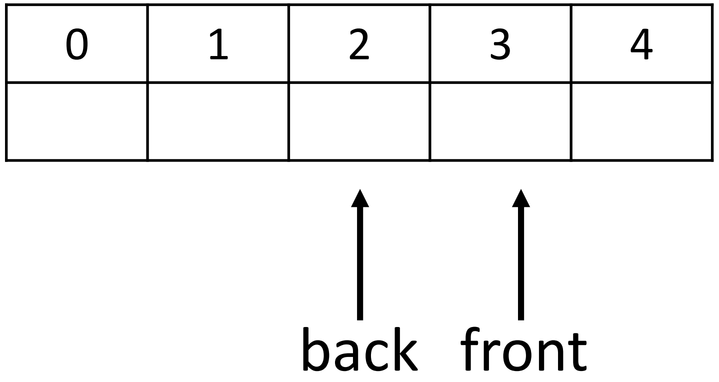
0	1	2	3	4
		Z		

back

front

Implementing Queues with an Array

- Our check for full is still true!!
    - front == (back+1) % length



0	1	2	3	4

Implementing Queues with an Array

- Two solutions:
  - Keep a separate variable counting elements
  - Always leave an empty space in the array

Implementing Queues with an Array

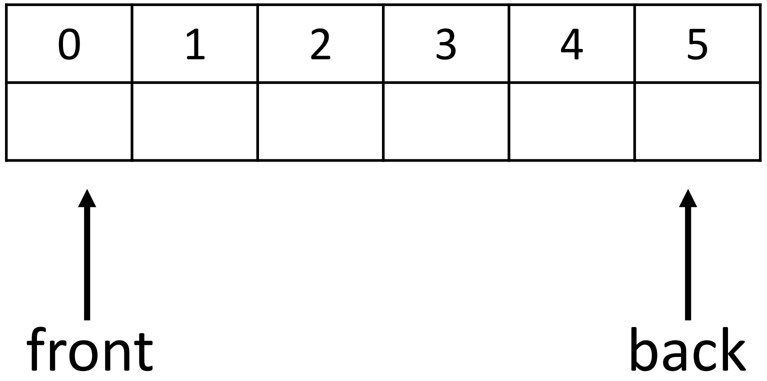
- Leave an empty space in the array

- Check for empty array:
  - front == (back+1) % array.length

- Check for full array:
  - front == (back+2) % array.length

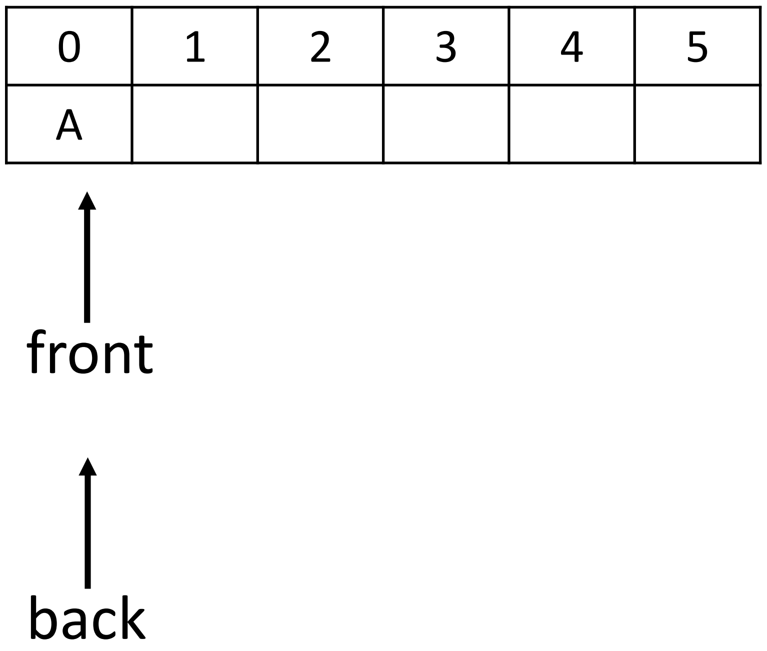
Implementing Queues with an Array

- Leave an empty space in the array
- empty?   front == (back+1) % array.length
- full?   front == (back+2) % array.length



Implementing Queues with an Array

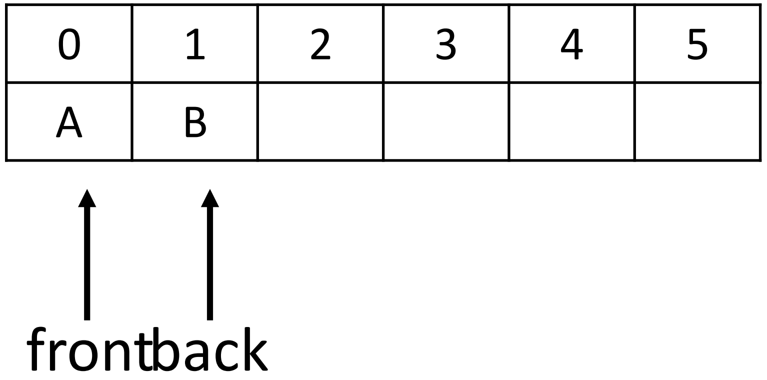
- Enqueue A



0	1	2	3	4	5
A					

Implementing Queues with an Array

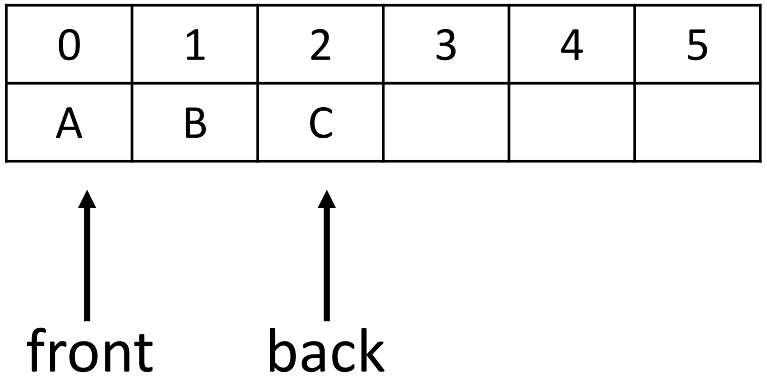
- Enqueue B



0	1	2	3	4	5
A	B				

Implementing Queues with an Array

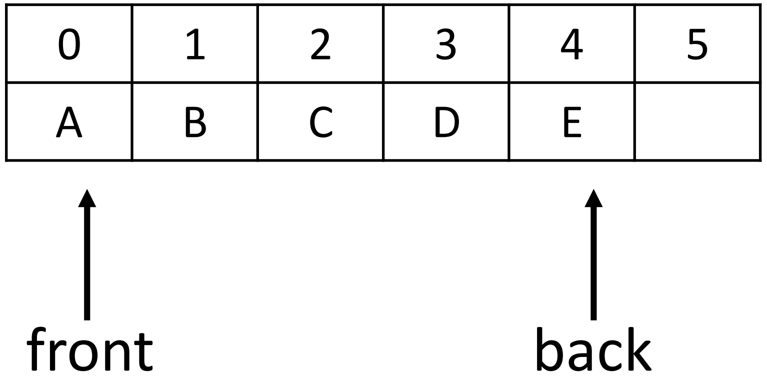
- Enqueue C



0	1	2	3	4	5
A	B	C			

Implementing Queues with an Array

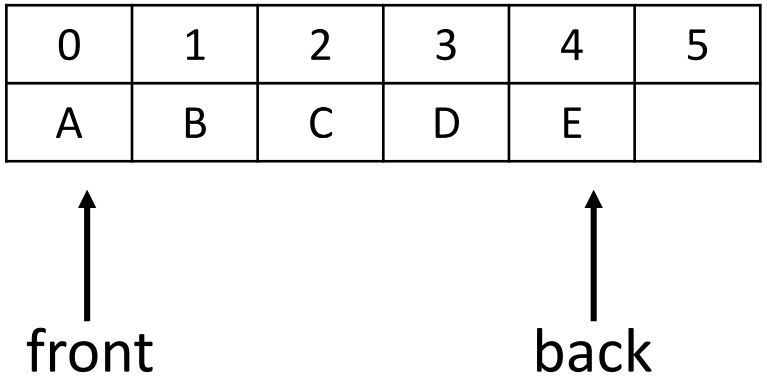
- Enqueue D and E



0	1	2	3	4	5
A	B	C	D	E	

Implementing Queues with an Array

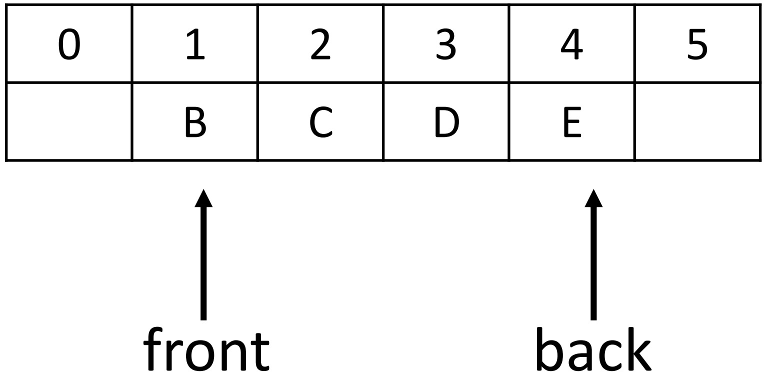
- empty? front == (back+1) % array.length
- full? front == (back+2) % array.length



0	1	2	3	4	5
A	B	C	D	E	

Implementing Queues with an Array

- Dequeue



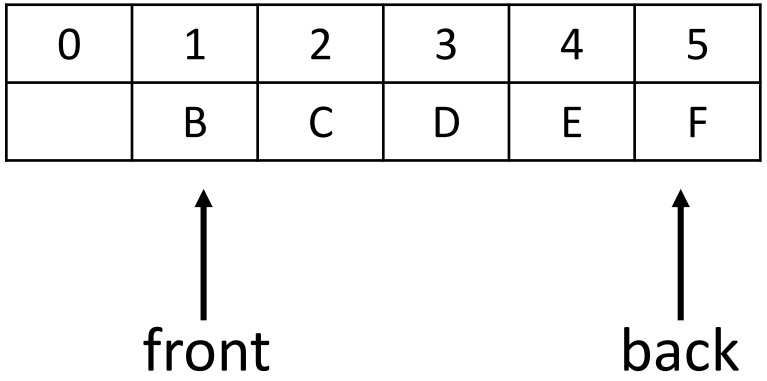
0	1	2	3	4	5
	B	C	D	E	

front

back

Implementing Queues with an Array

- Enqueue F



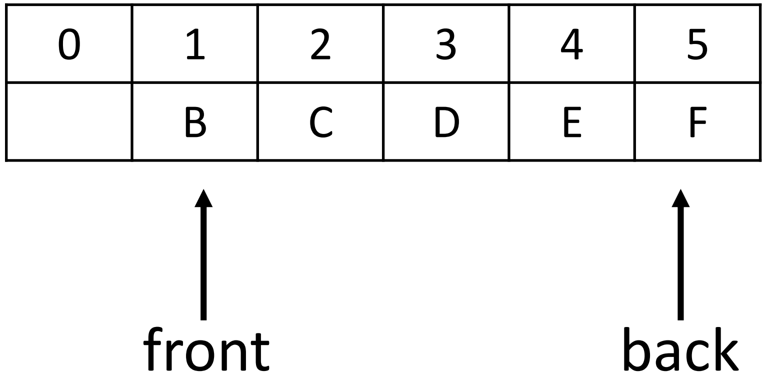
0	1	2	3	4	5
	B	C	D	E	F

front

back

Implementing Queues with an Array

- empty? front == (back+1) % array.length
- full? front == (back+2) % array.length



0	1	2	3	4	5
	B	C	D	E	F

front

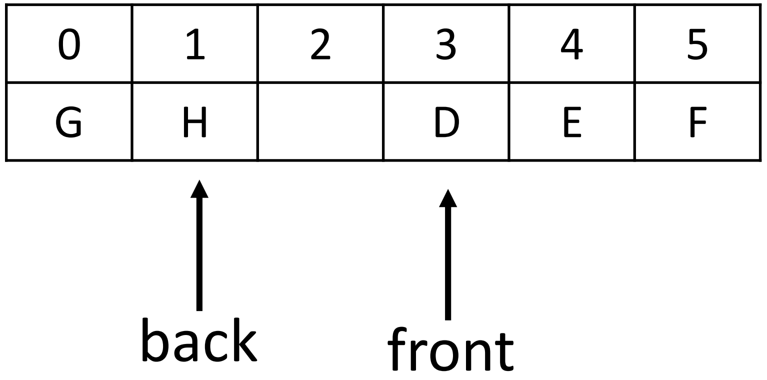
back

Implementing Queues with Arrays

- For an efficient solution, leave an empty space.
  - For a queue to hold n elements, create an array of size n+1.
- Initialize front=0 and back=array.length-1
- To add, update back **first** and then put in the array.
  - back = (back + 1) % array.length
  - array[back] = newElement
- To remove, get the element and **then** update front.
  - removeElement = array[front]
  - front = (front + 1) % array.length

Implementing Queues with Arrays

- To traverse the queue contents:
    - for(int i = front;   i != (back+1)%queue.length;   i = (i+1)%array.length)



0	1	2	3	4	5
G	H		D	E	F

back

front

Queues in Java Standard Library

- Java supports the queue data structure through several classes and interfaces.
- Queue interface
- enqueueing: add or offer
- dequeueing: remove or poll

- Many classes implement Queue

Queue Interface- add vs offer

- the add method throws an exception if the addition fails
  - use add when the queue is not limited in size
- the offer method returns false when the add fails
  - use offer when the queue is fixed in size
  - for a fixed-size queue, it's expected that at some point you might reasonably try to add to a full queue, so you don't want to throw an exception in that case- you just want to return false to indicate that the addition fail

Queue Interface- remove vs poll

- these methods work similarly as the enqueueing methods when trying to dequeue from an empty queue
- remove throws an exception when you try to dequeue from an empty queue
- poll returns null when you try to dequeue from an empty queue

LinkedList

- LinkedList is the most common class to use to implement Queue

  ```
  Queue<String> wordQueue = new LinkedList<String>();
  ```

- Declaring as type Queue restricts to the methods defined in the queue interface.
- Then you instantiate as the concrete class LinkedList.
- This is similar to how we use our textbook classes
  - QueueInterface<String> queue = new LinkedQueue<>())

The Deque Interface

- Java provides a Deque interface.
- ArrayDeque implements this interface using an array.
- LinkedList implements this interface using linked nodes.

- Create a deque the same way as you would a queue:

```
Deque<String> wordDeque = new LinkedList<String>();
Deque<String> wordDeque = new ArrayDeque<String>();
```

- The add and remove methods use “first” and “last” added onto the queue methods
  - Examples: addFirst, offerFirst, removeLast, pollLast

Priority Queues

- PriorityQueue class.
- Priority is defined by the class's compareTo method (the "natural ordering").
  - Smaller items get higher priority
  - Ties are broken arbitrarily
    - Note that this is different from an implementation where ties are resolved based on the chronological ordering.
- You can also create a queue by passing in a *comparator*, which specifies a different ordering than compareTo.