ITI 1121. Introduction to Computing II *

Marcel Turcotte
School of Electrical Engineering and Computer Science

Version of March 2, 2013

Abstract

- Queues
  - LinkedQueue

*These lecture notes are meant to be looked at on a computer screen. Do not print them unless it is necessary.

Definitions

A **queue** is a linear abstract data type such that insertions are made at one end, called the **rear**, and removals are made at the other end, called the **front**.

Queues are sometimes called FIFOs: *first-in first-out*.

`enqueue()` $\Rightarrow$ Queue $\Rightarrow$ `dequeue()`

The two basic operations are:

enqueue: adds an element to the rear of the queue;

dequeue: removes and returns the element at the front of the queue.

$\Rightarrow$ Software queues are similar to physical ones: queuing at the supermarket, at the bank, at cinemas, etc.

Applications

- Shared resources management
  (system programming):
  - Access to the processor;
  - Access to the peripherals such as disks and printers.

- Application programs:
  - Simulations;
  - Generating sequences of increasing length over a finite size alphabet;
  - Navigating through a maze.

Example

```
public class Test {

    public static void main( String[] args ) {

        Queue<Integer> queue = new QueueImplementation<Integer>();

        for ( int i=0; i<10; i++ )
            queue.enqueue( new Integer( i ) );

        while ( ! queue.isEmpty() )
            System.out.println( queue.dequeue() );

    }
}
```

$\Rightarrow$ What does it print? 0, 1, 2, 3, 4, 5, 6, 7, 9

```
q = new Q();
q.enqueue( a );
q.enqueue( b );
q.enqueue( c );
q.dequeue( );
-> a
q.dequeue( );
-> b
q.enqueue( d );
q.dequeue( );
-> c
q.dequeue( );
-> d
```

⇒ Elements of a queue are processed in the same order as the they are inserted into the queue, here “a” was the first element to join the queue and it was the first to leave the queue: *first-come first-serve*.

Implementations

Just like stacks, there are two families of implementations:

- Linked elements;
- Array-based.

ADT

```
public interface Queue {
    public abstract boolean isEmpty();
    public abstract void enqueue( Object o );
    public abstract Object dequeue();
}
```

ADT

```
public interface Queue<E> {
    public abstract boolean isEmpty();
    public abstract void enqueue( E o );
    public abstract E dequeue();
}
```

Using linked elements

```
public class LinkedQueue<T> implements Queue<T> {

    public boolean isEmpty() { ... }
    public void enqueue( T o ) { ... }
    public T dequeue() { ... }

}
```

Using linked elements

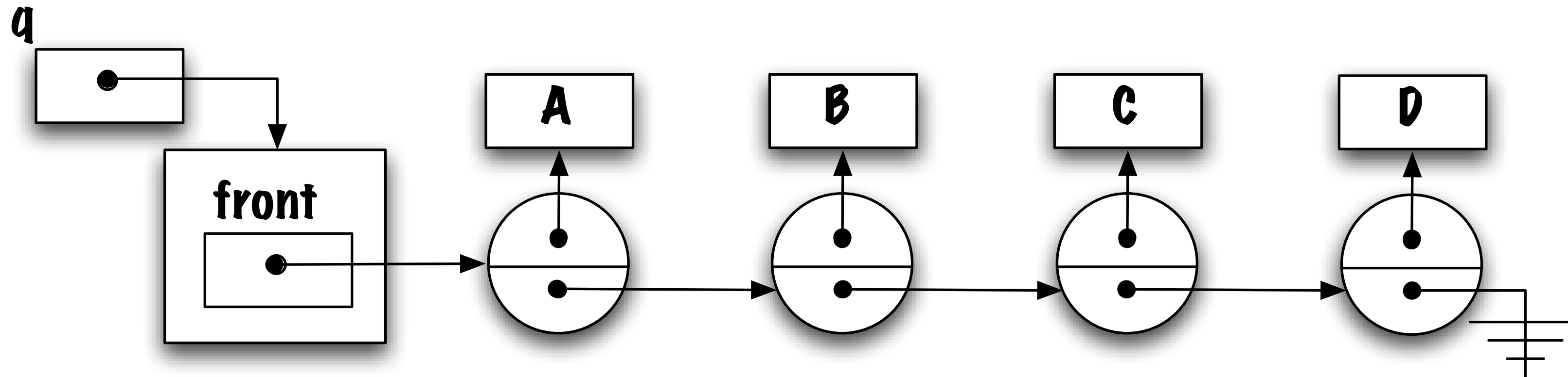
```
public class LinkedQueue<T> implements Queue<T> {

    private static class Elem<E> {
        private E value;
        private Elem<E> next;
        private Elem( E value, Elem<E> next ) {
            this.value = value;
            this.next = next;
        }
    }

    public boolean isEmpty() { ... }
    public void enqueue( T o ) { ... }
    public T dequeue() { ... }
}
```

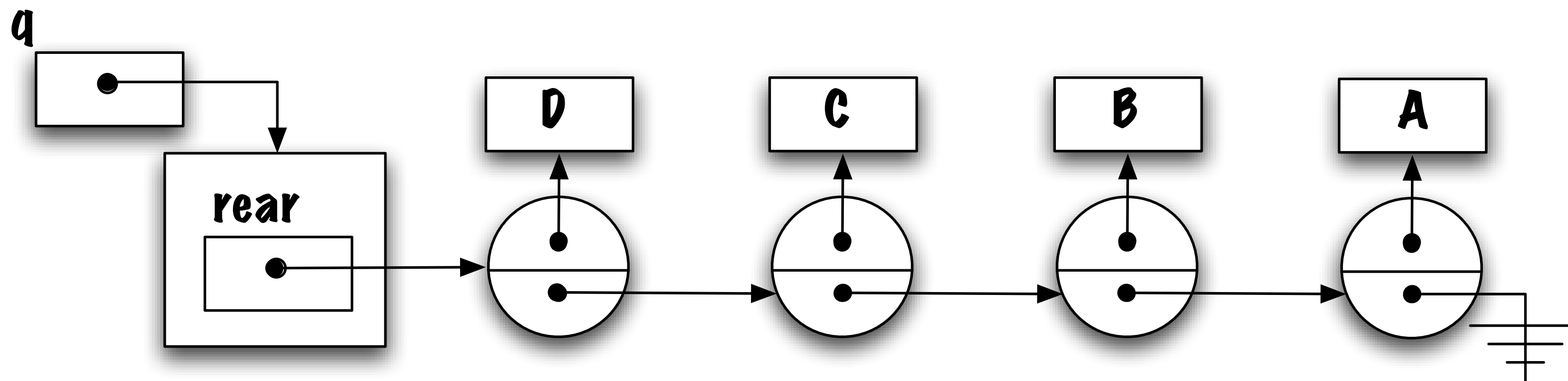
Using linked elements

```
public class LinkedQueue<T> implements Queue<T> {

    private static class Elem<E> {
        private E value;
        private Elem<E> next;
        private Elem( E value, Elem<E> next ) {
            this.value = value;
            this.next = next;
        }
    }

    private Elem<T> front; // or rear?

    public boolean isEmpty() { ... }
    public void enqueue( T o ) { ... }
    public T dequeue() { ... }
}
```

Using linked elements

Which implementation is preferable and why?





Using linked elements

With the first implementation, removing an element is easy (and fast) but adding an element to the rear of the queue will be more involved (and slow).

The other implementation simply reverses the situations, removal is costly while adding is fast.

Is this a dead-end?

What makes removing fast? Having a reference to the **front** element.

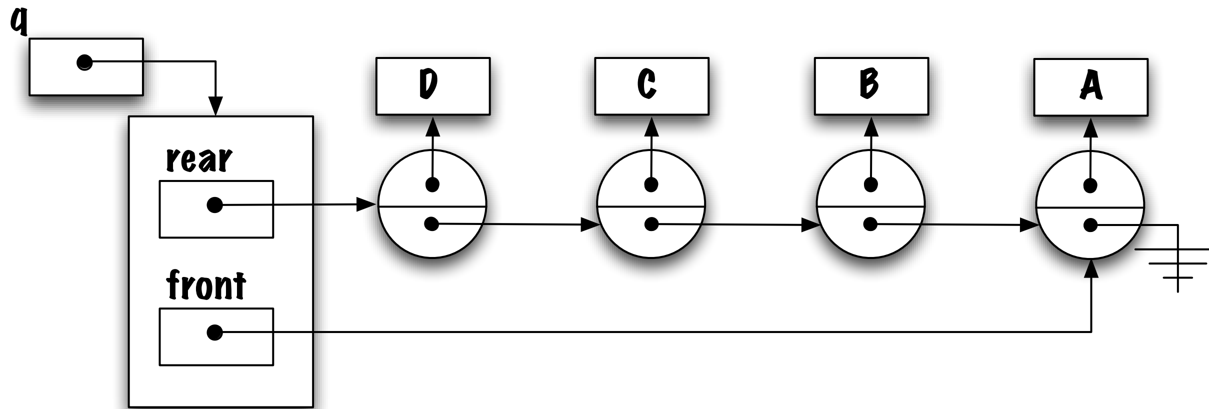
What makes adding an element fast? A reference to the **rear** element.

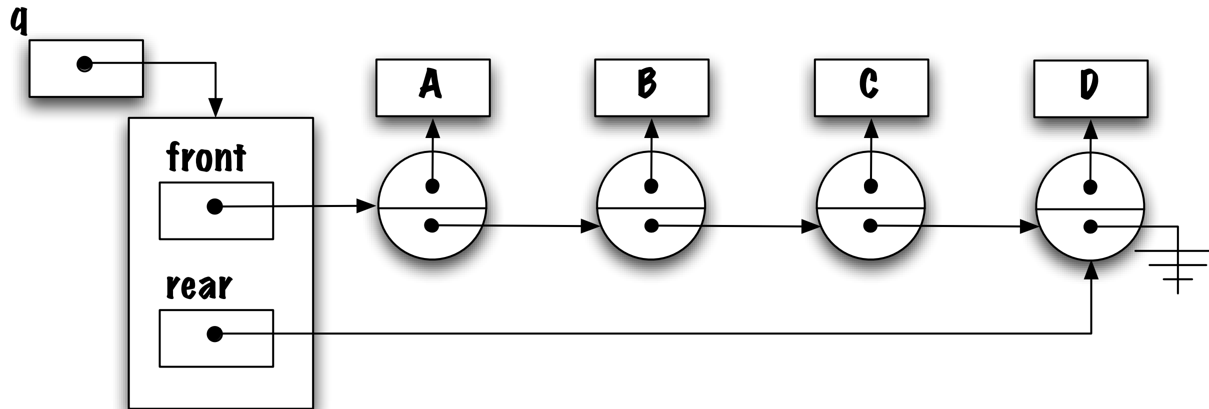
Using linked elements

Solution: the class **LinkedQueue** will be having two instance variables, **front** and **rear**, pointing at the front and rear elements, making both operations fast.
(is it really that easy?)

Using linked elements

Will these two implementations be equally fast?





Discussion

What are the consequences of these changes?

Impact on memory usage is not significant.

Implementing the methods will be slightly more complex.

Adding an element (enqueue)

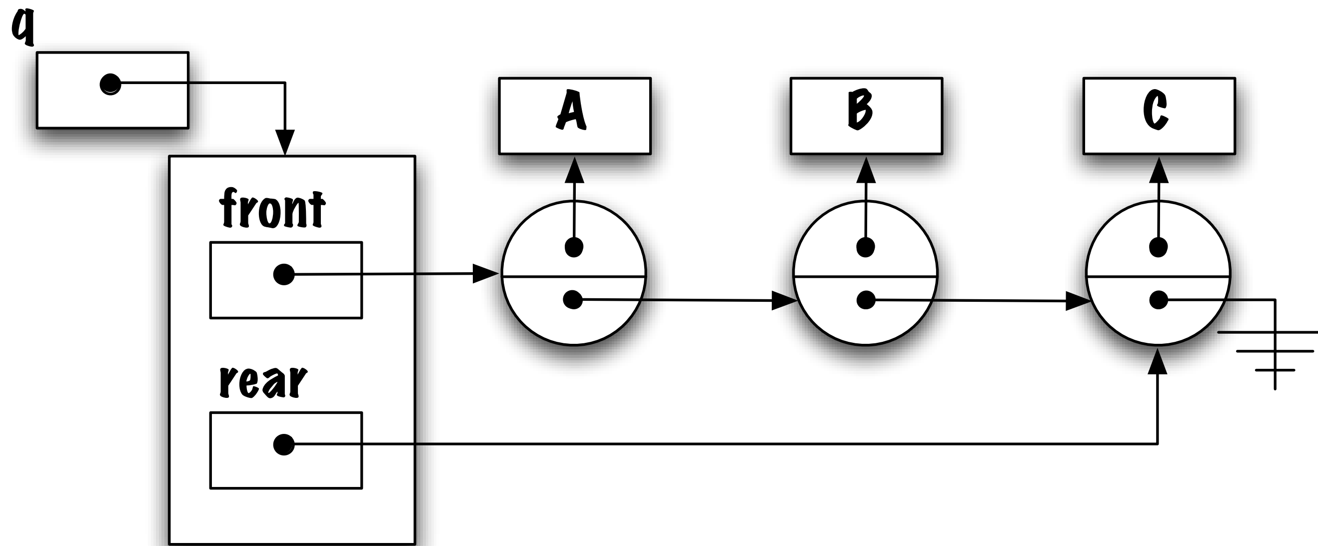
Identify the general case as well as the special case(s).

General case, consider a number of elements sufficiently large to represent the majority of the cases (3 or 4 elements generally suffice).

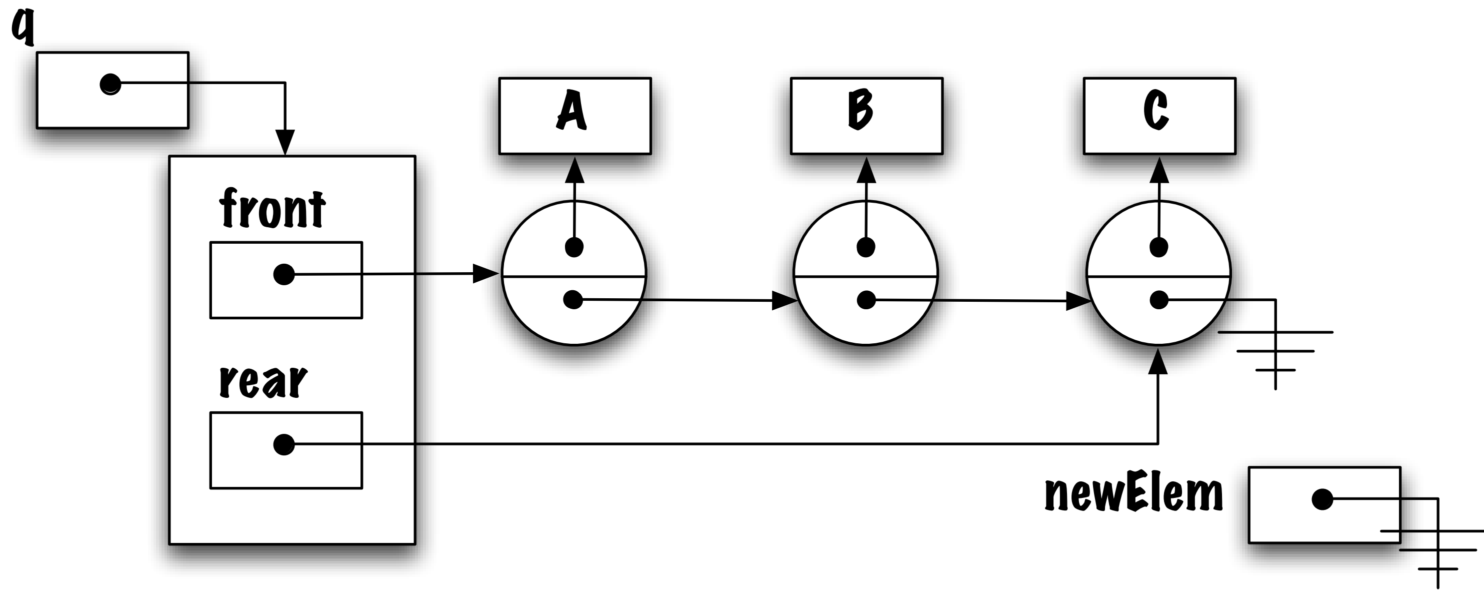
Special cases are the cases where the general strategy is not applicable.

For stacks, queues and lists, having no elements or one element is often a special case.

Adding an element (enqueue) (general case)

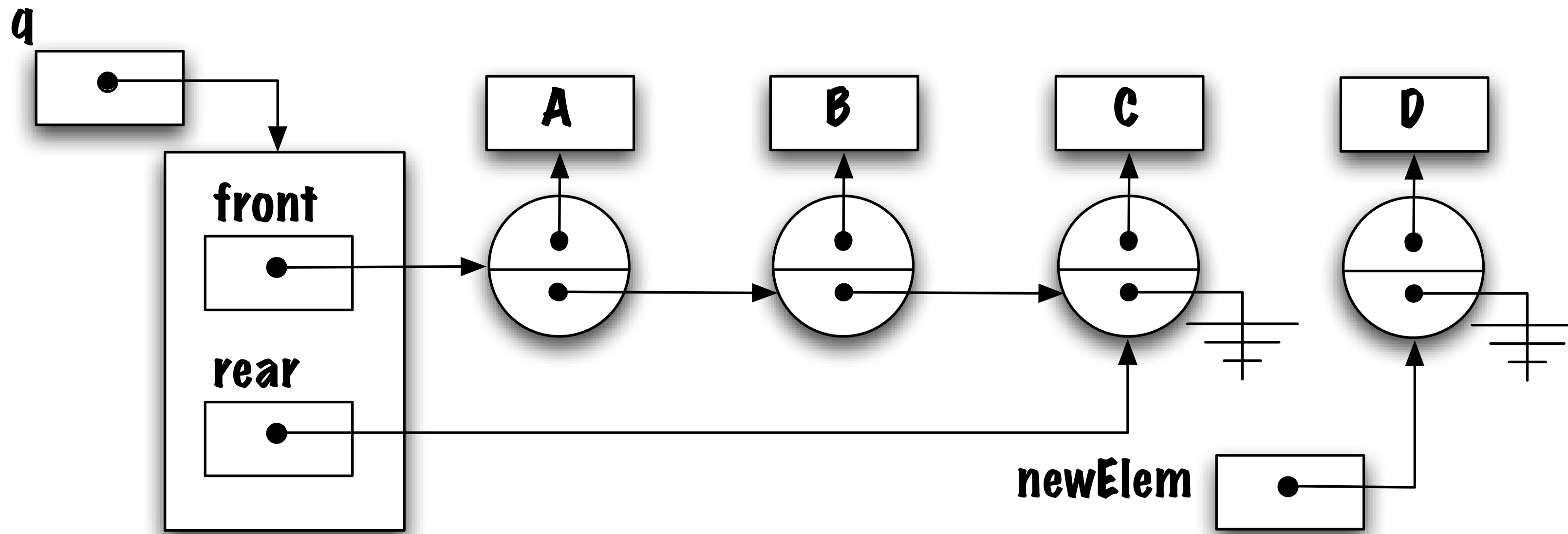


Adding an element (enqueue) (general case)

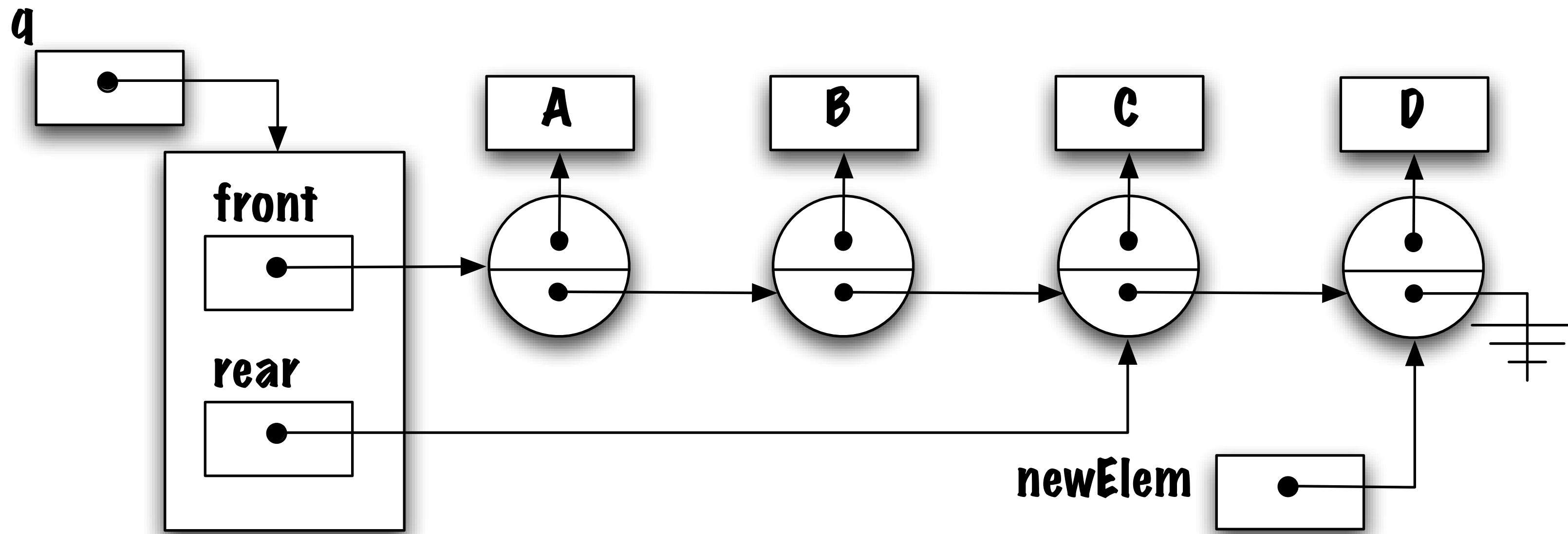


Using a local variable will make our work simpler.

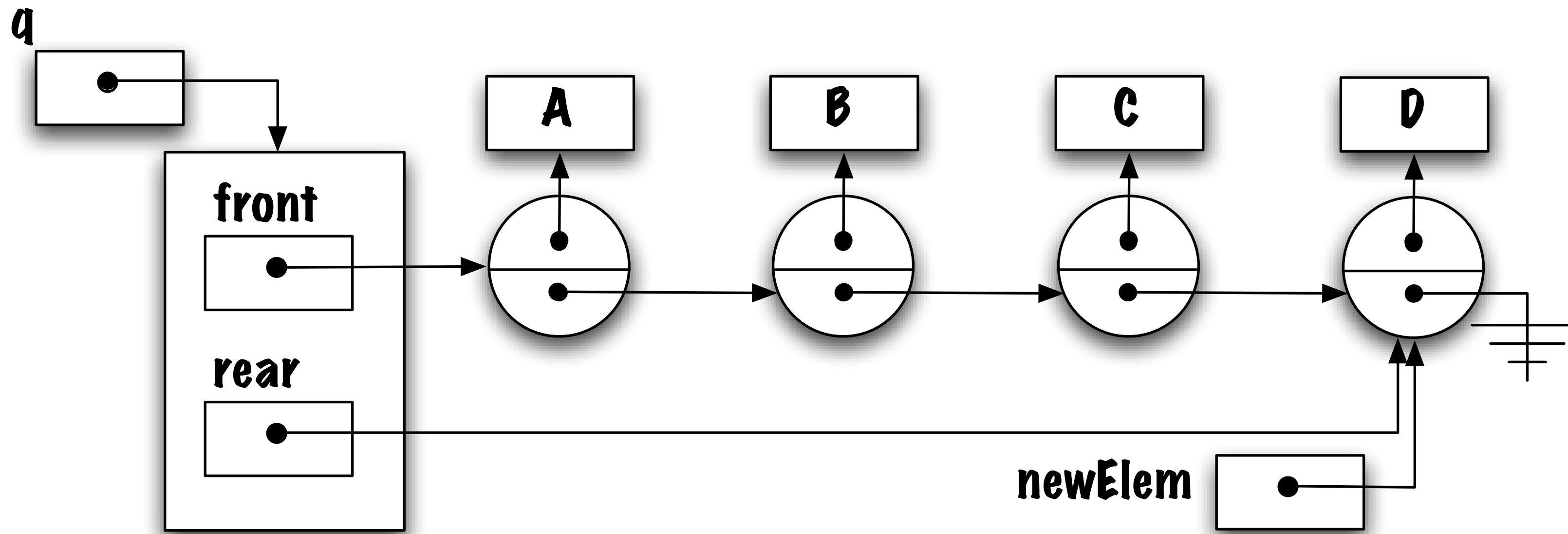
Adding an element (enqueue) (general case)



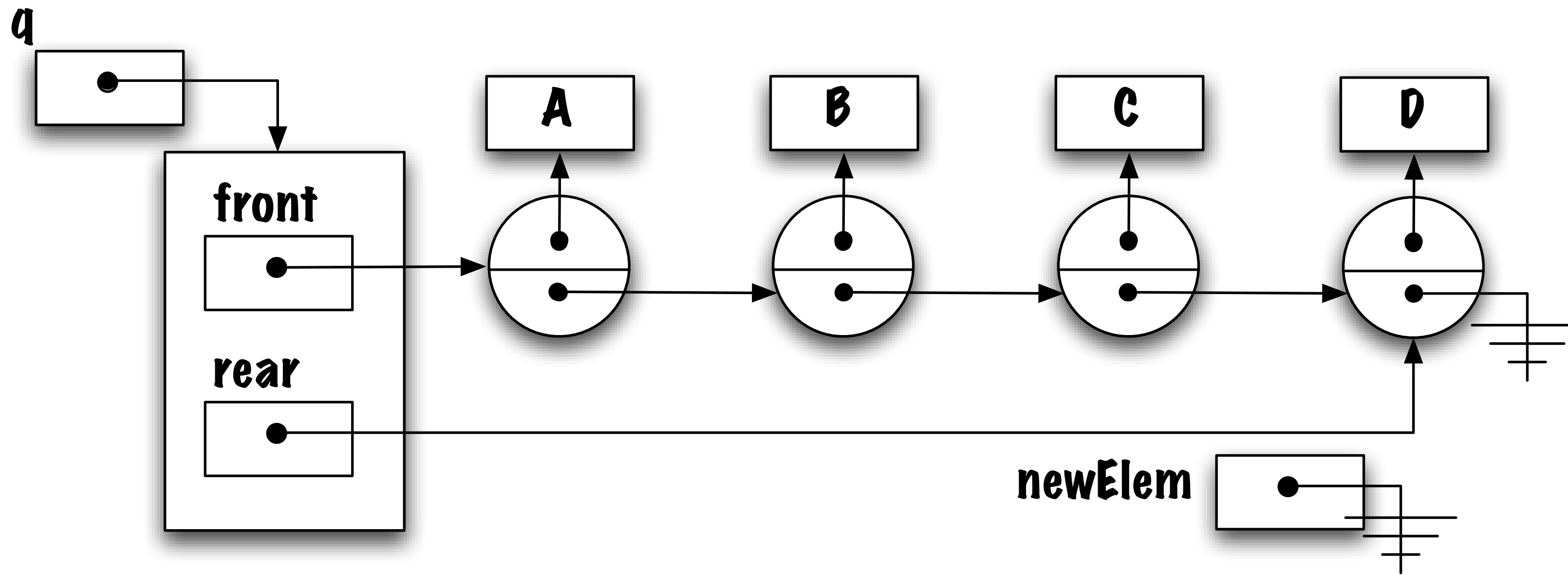
Adding an element (enqueue) (general case)



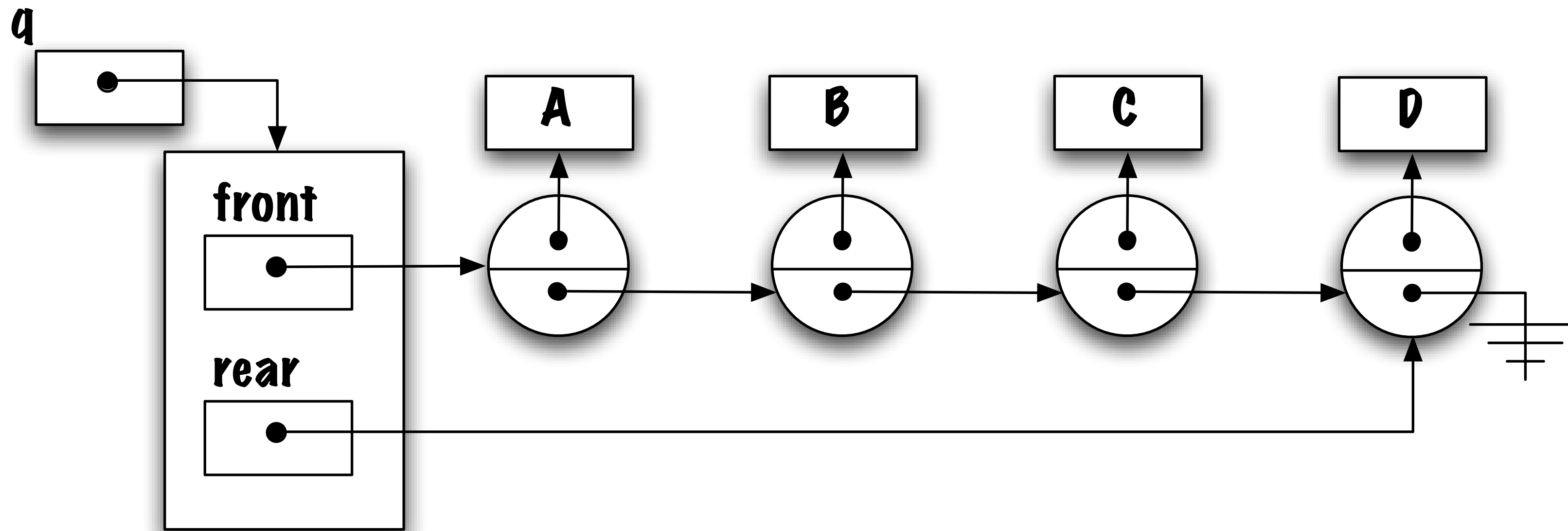
Adding an element (enqueue) (general case)



Adding an element (enqueue) (general case)

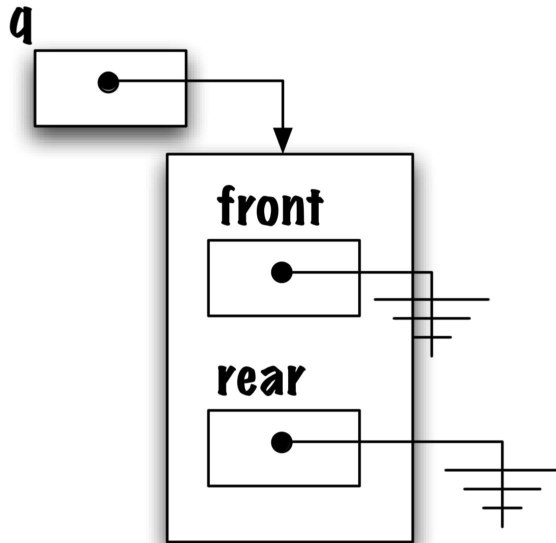


Adding an element (enqueue) (general case)



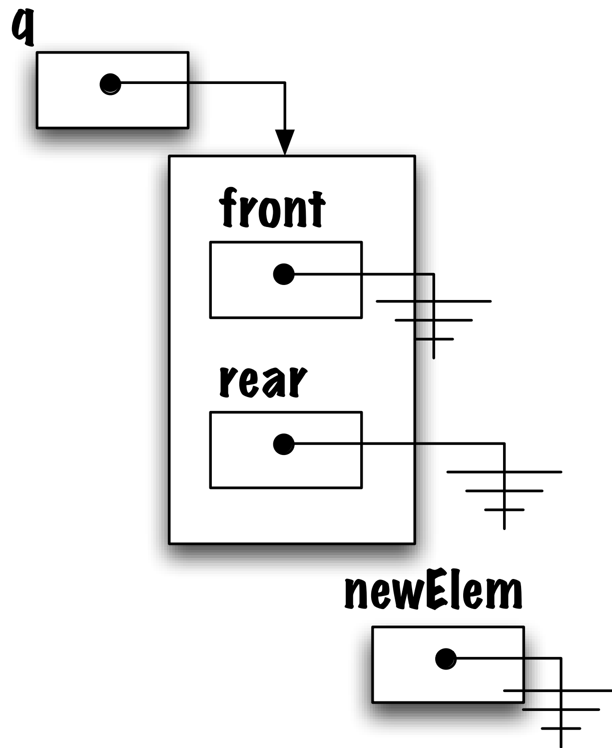
Adding an element (enqueue) (special case)

Draw the memory diagram representing the empty queue.

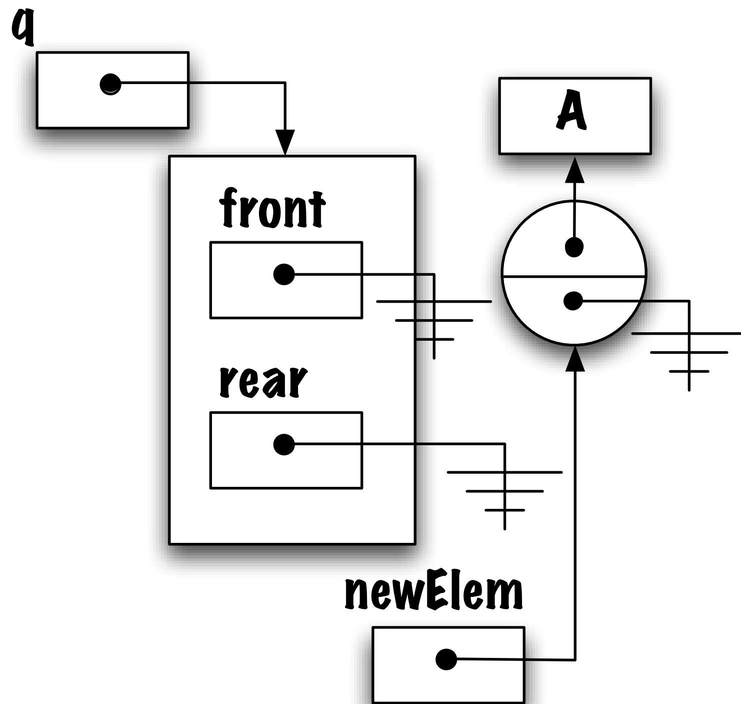


Which expression allows to identify the empty queue?

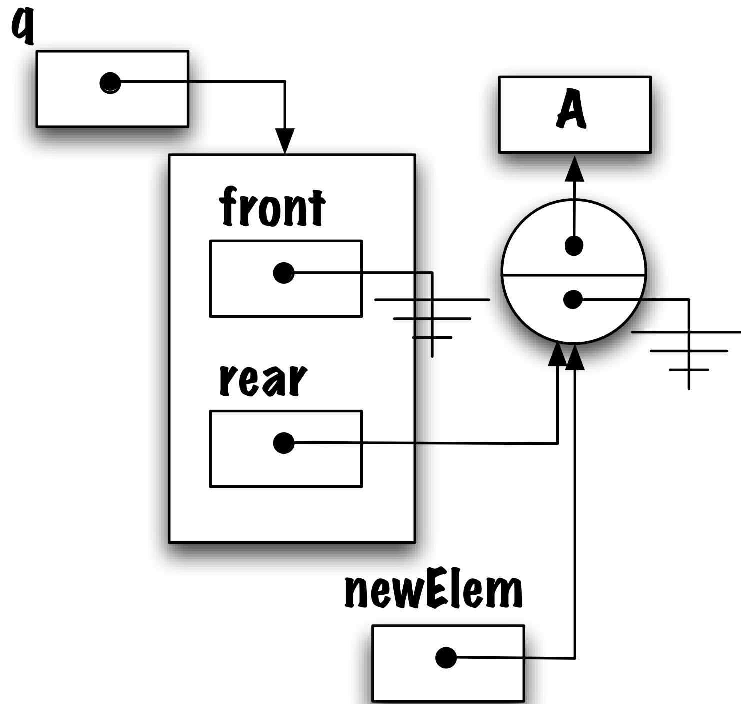
Adding an element (enqueue) (special case)



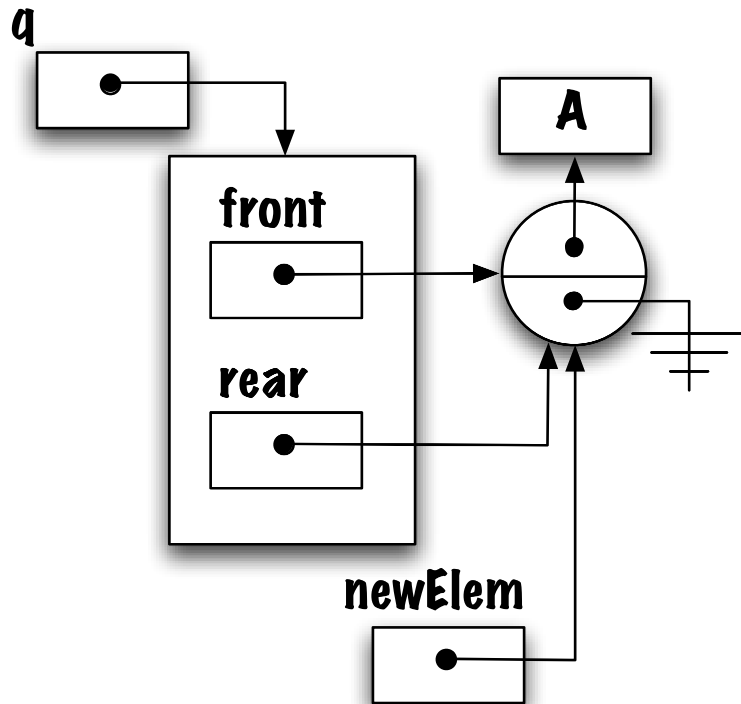
Adding an element (enqueue) (special case)



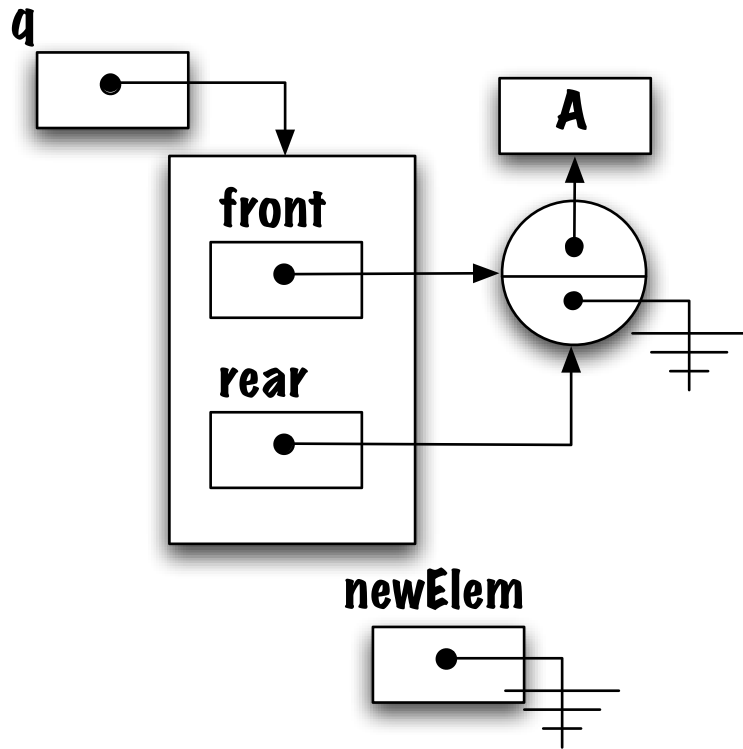
Adding an element (enqueue) (special case)



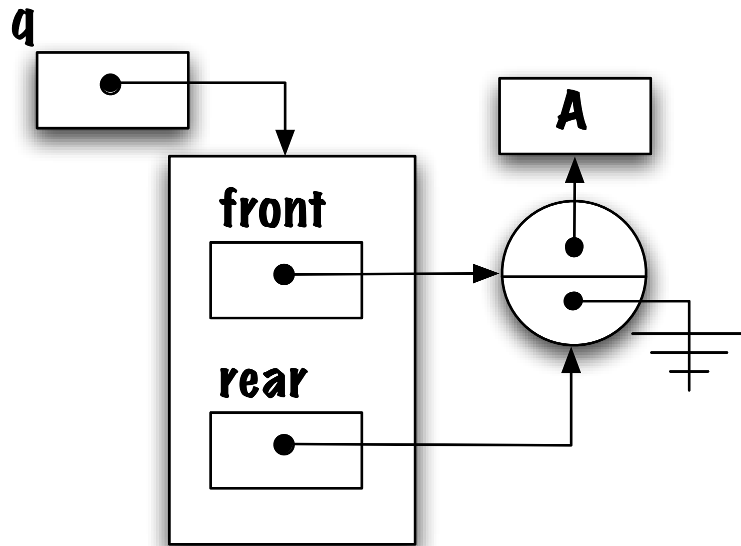
Adding an element (enqueue) (special case)



Adding an element (enqueue) (special case)



Adding an element (enqueue) (special case)



Removing an element

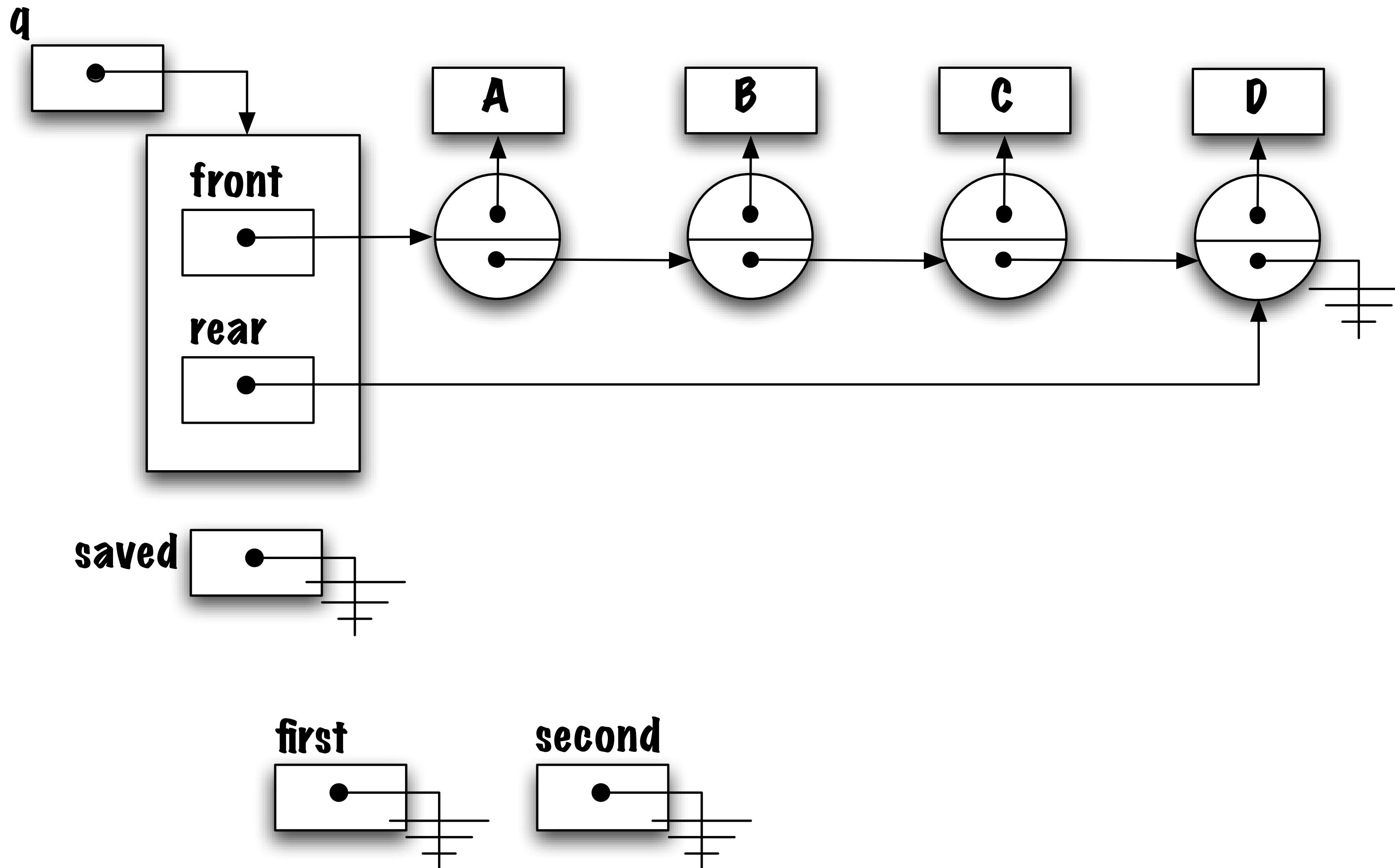
Identify the general case as well as the special case(s).

Is the empty queue a special case?

No, it is an illegal case, it should be handled by the pre-conditions and an exception should be thrown.

The queue containing a single element will be the special case.

Removing an element (general case)



Removing an element (general case)

q

A B C D

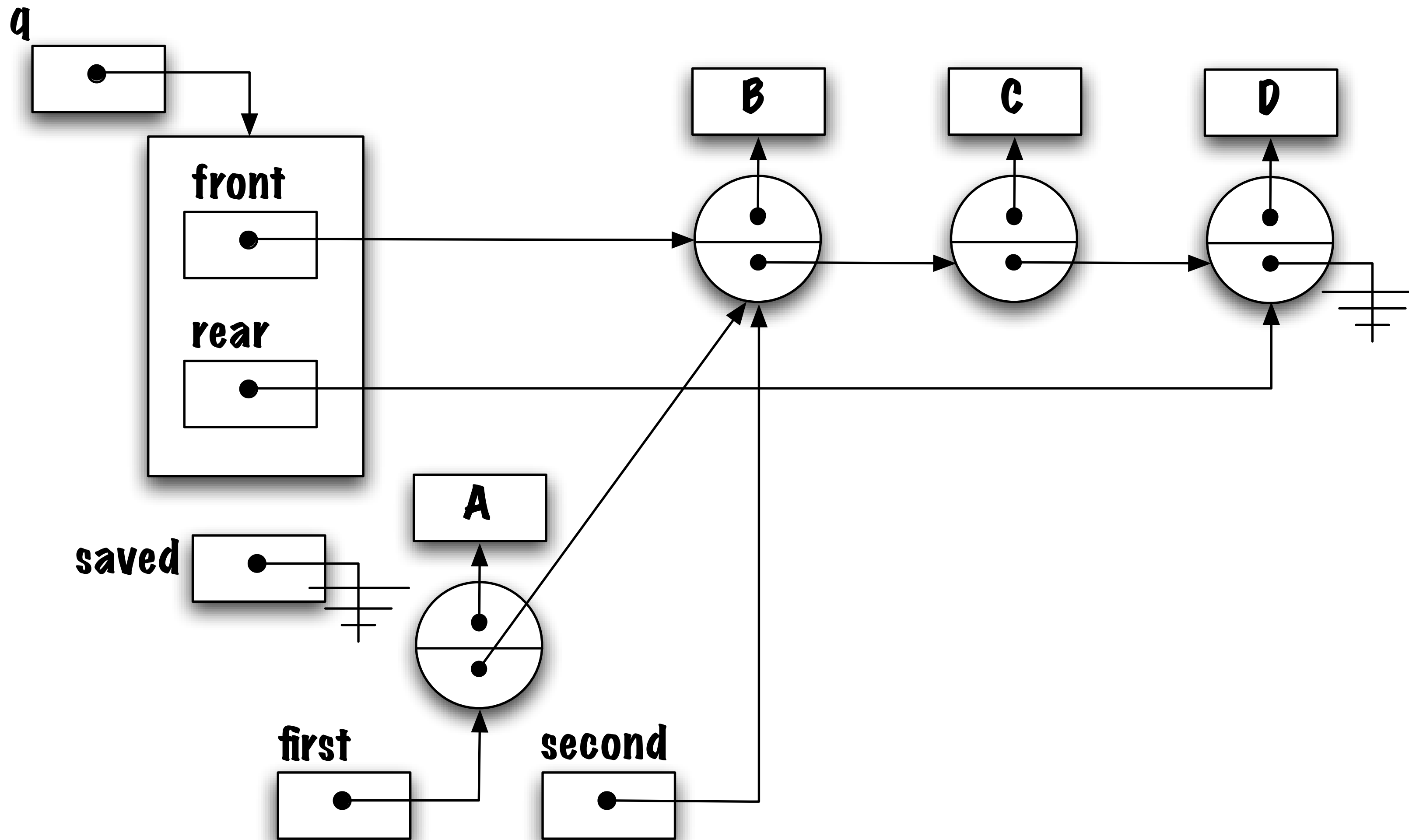
front

rear

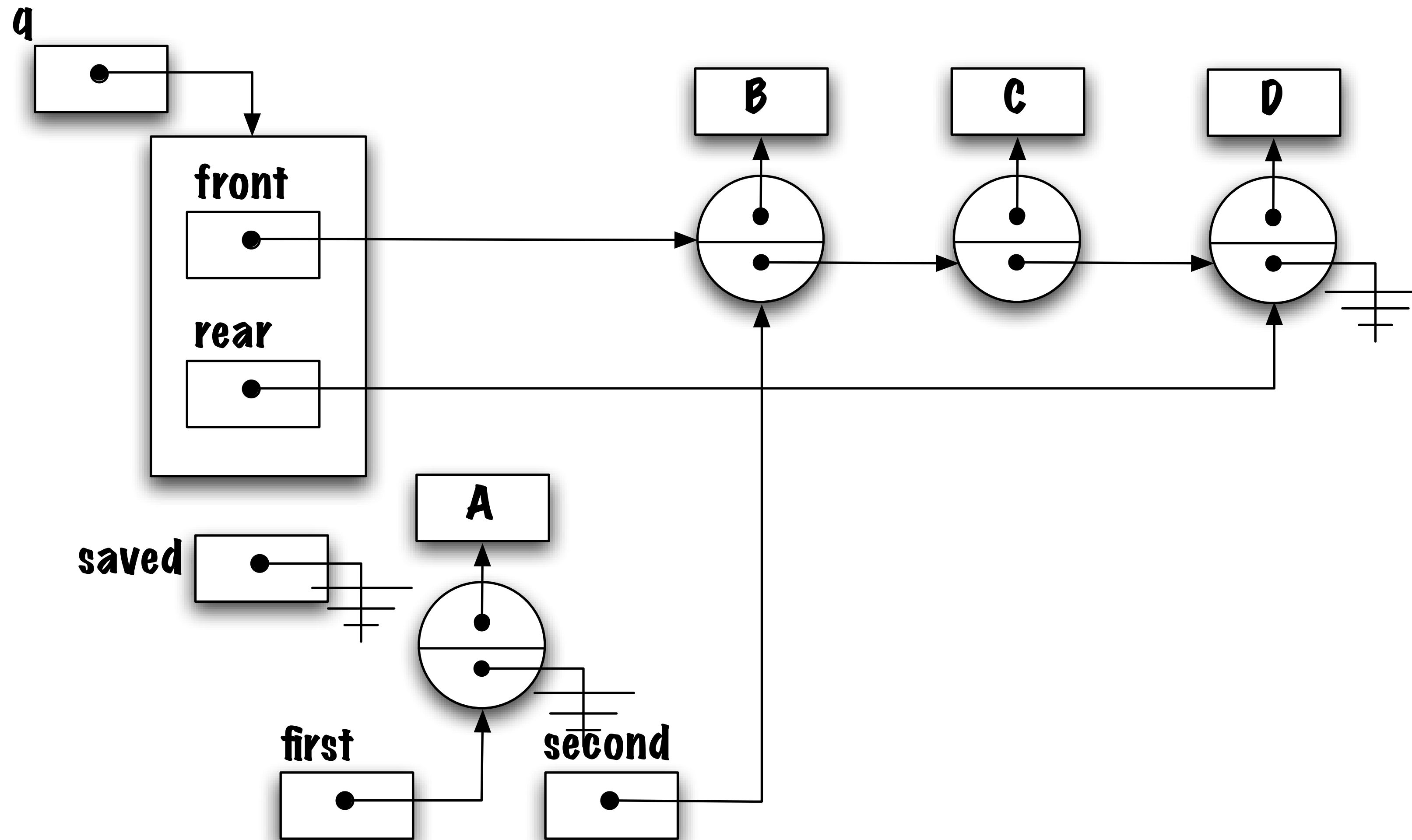
saved

first second

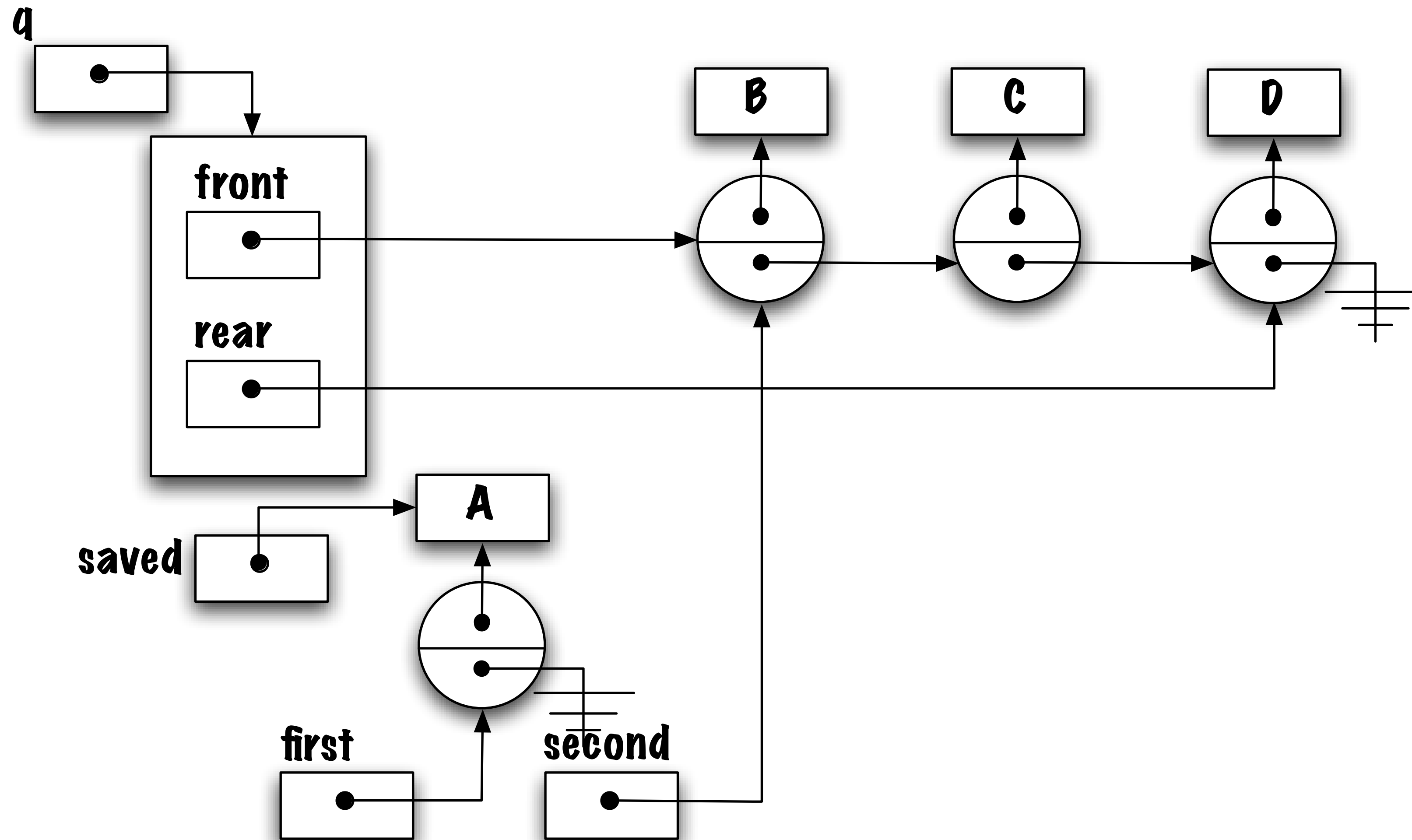
Removing an element (general case)



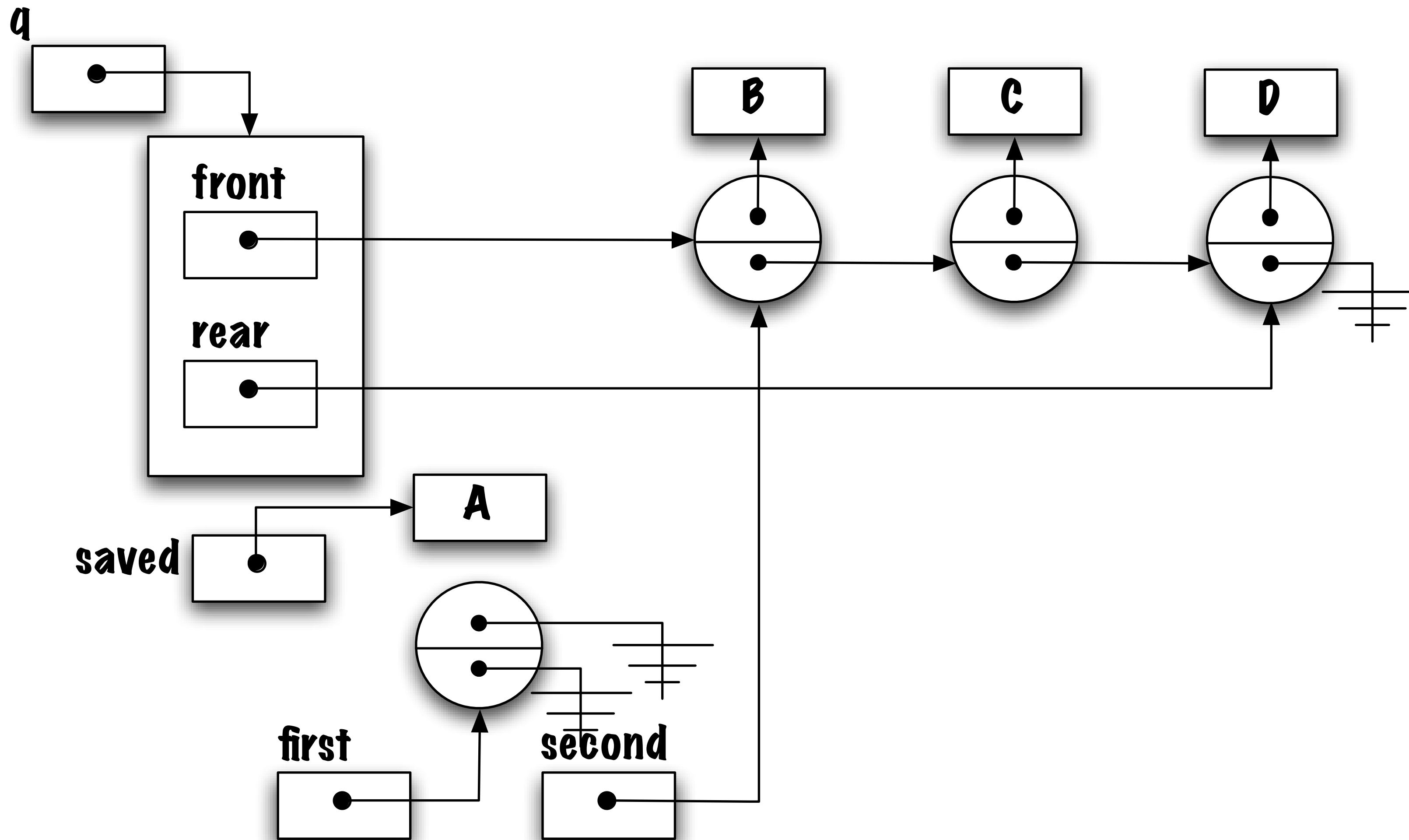
Removing an element (general case)



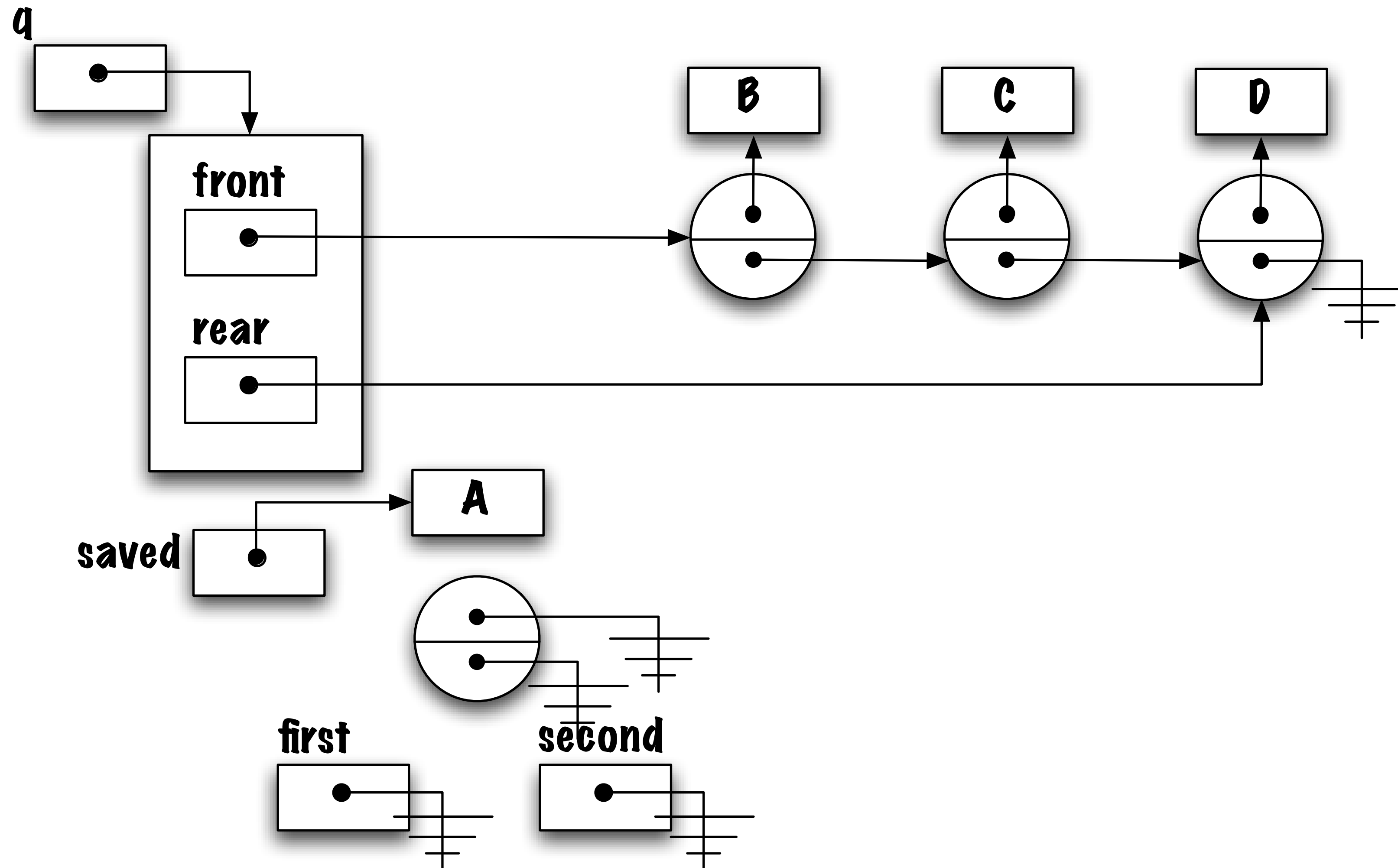
Removing an element (general case)



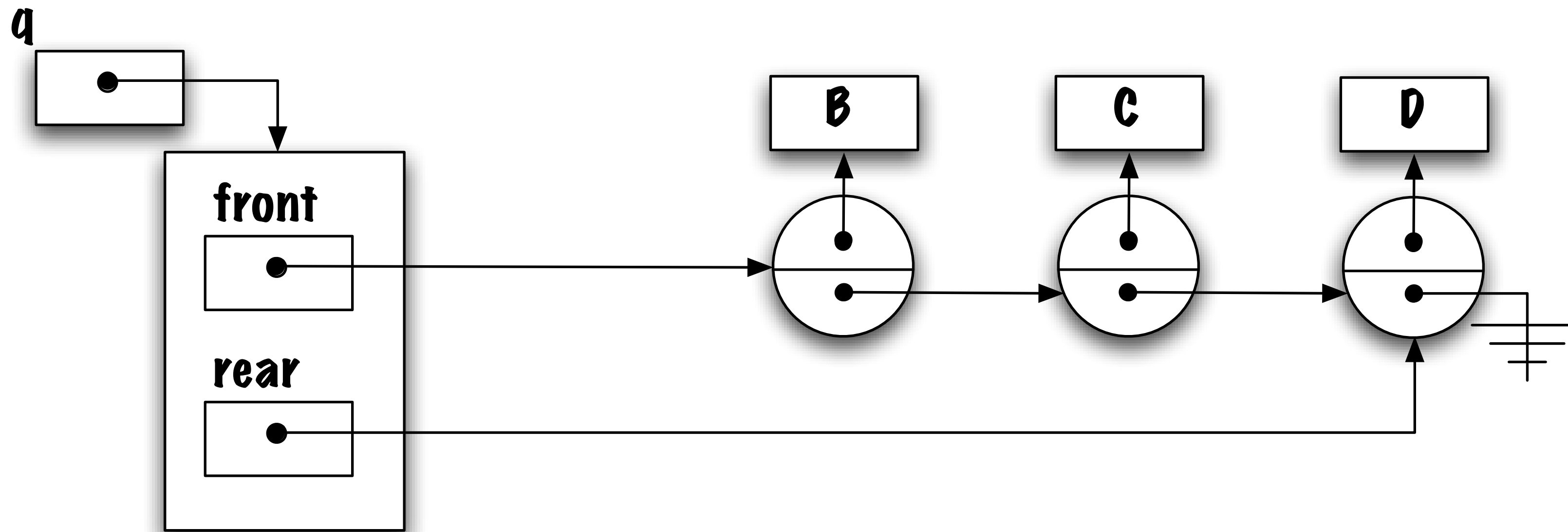
Removing an element (general case)



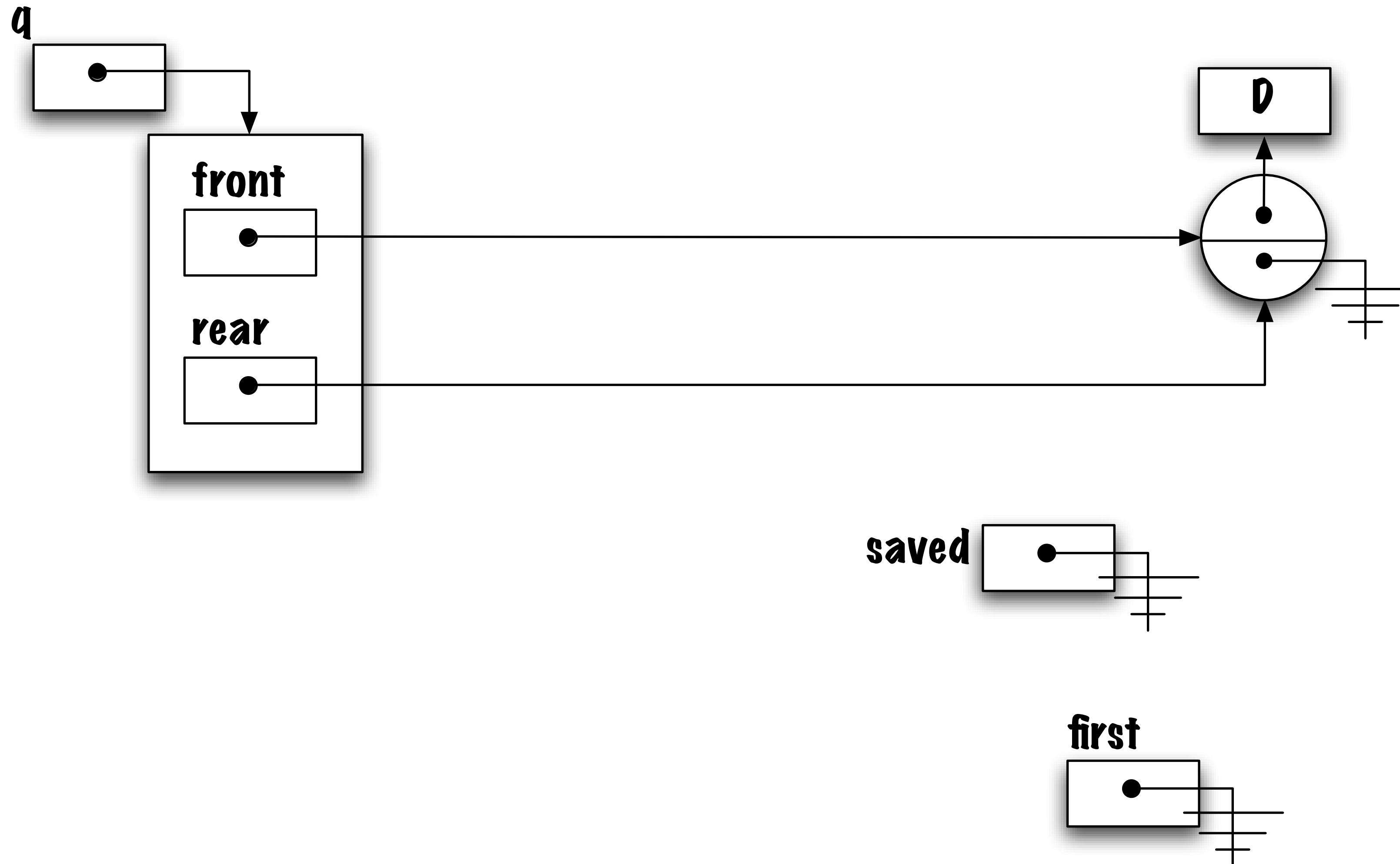
Removing an element (general case)



Removing an element (general case)

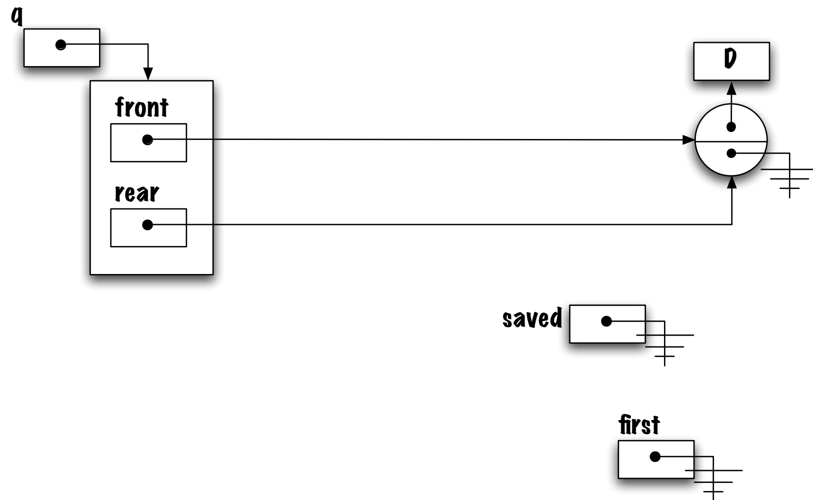


Removing an element (special case)



Which expression can be used to identify a queue containing a single element?

Removing an element (special case)



Pitfall. Which expression can be used to identify a queue containing a single element?

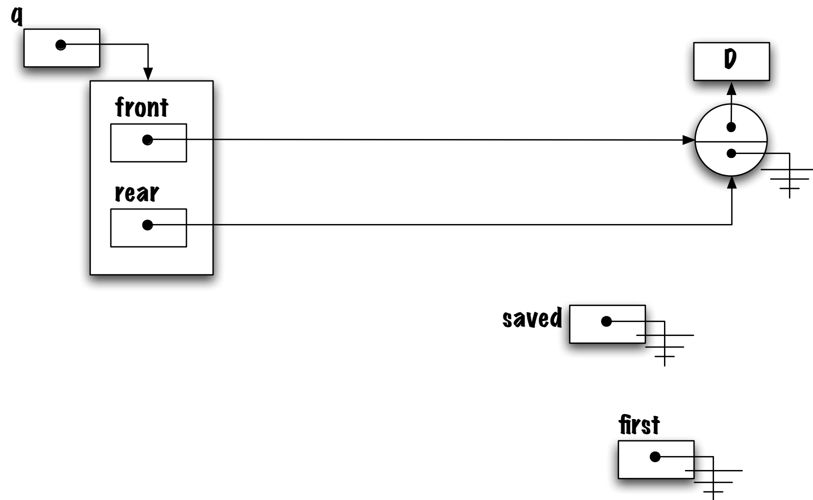
```
font != null && front.next == null
```

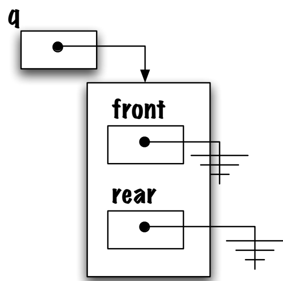
What about this?

```
front == rear
```

Removing an element (special case)

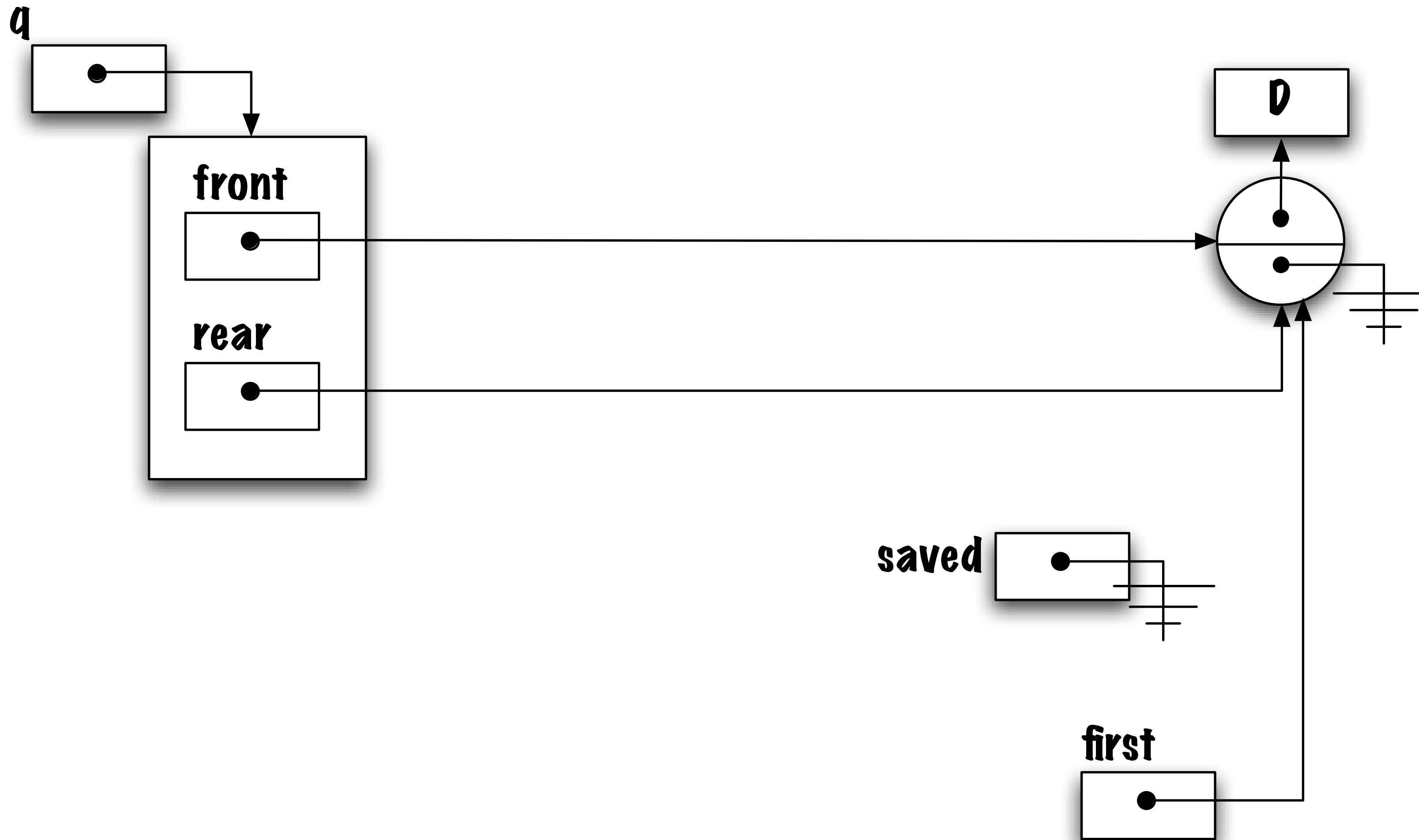
`front == rear` is also **true** for the empty queue!



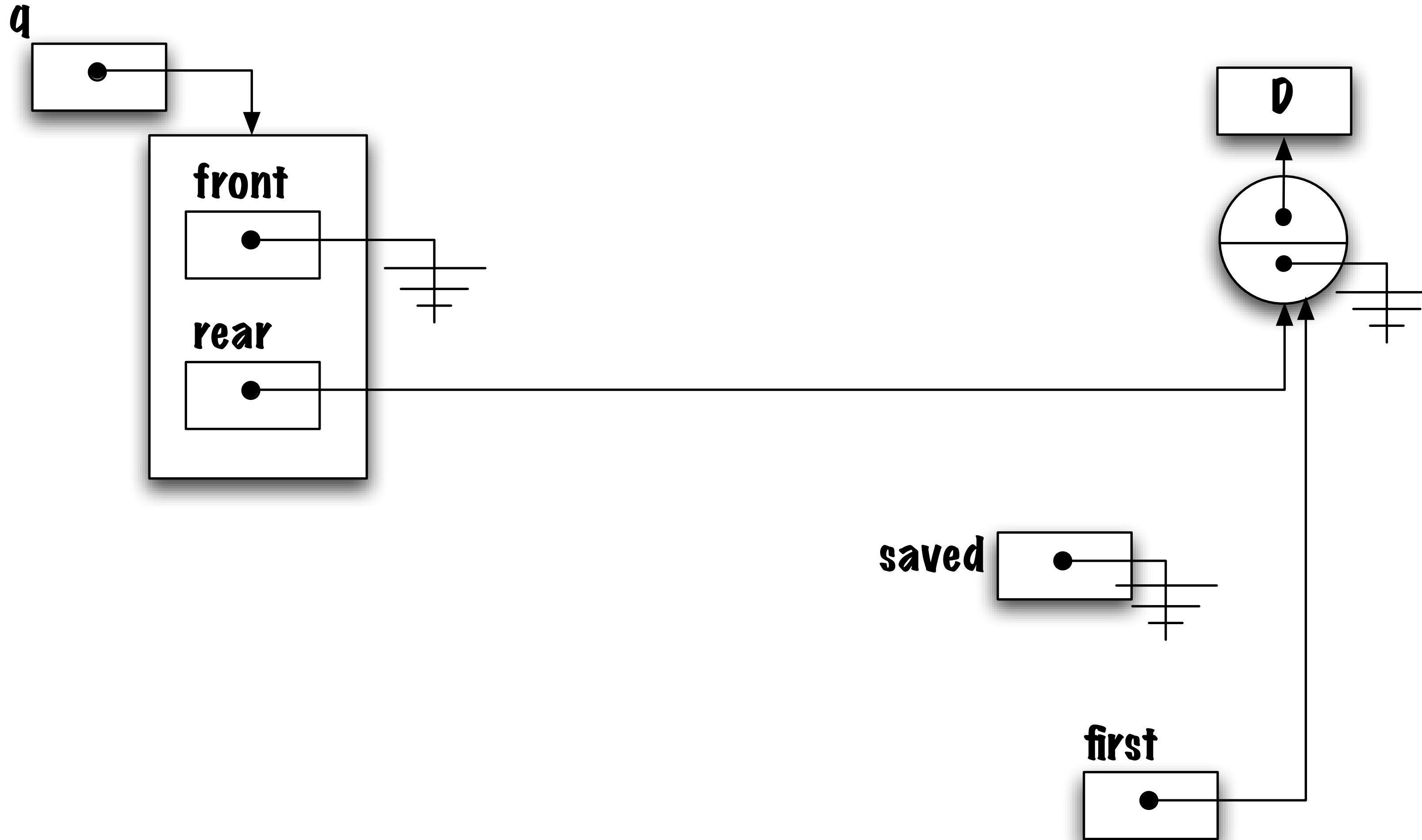


Solution: `front != null && front == rear`

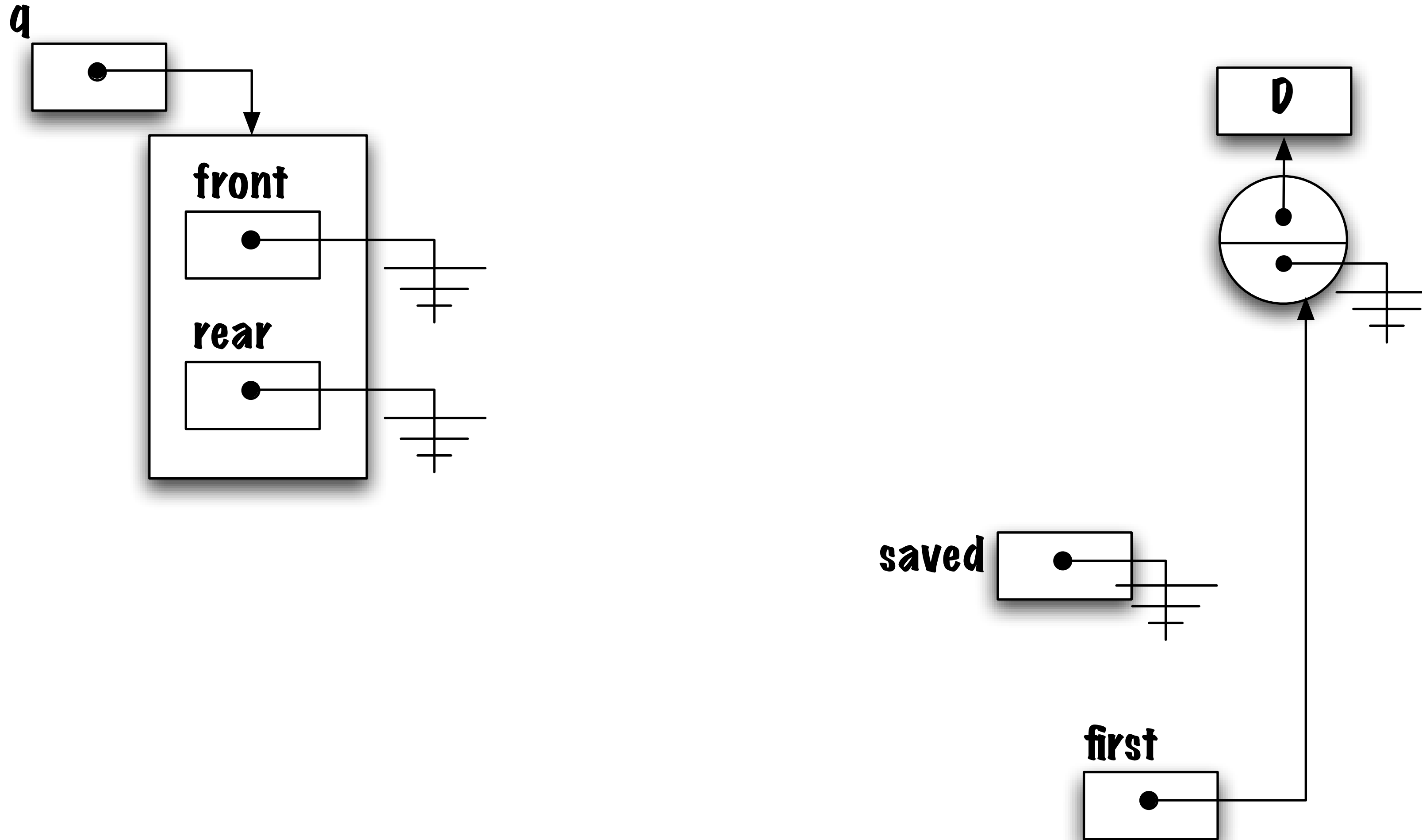
Removing an element (special case)



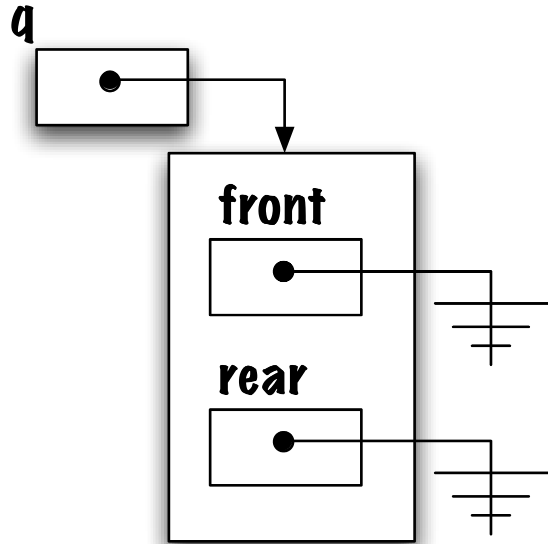
Removing an element (special case)

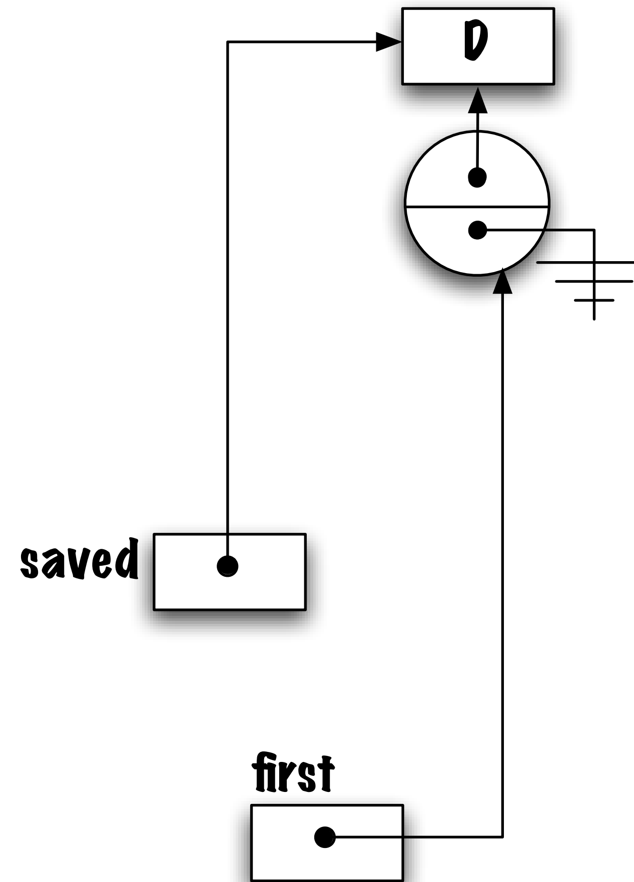


Removing an element (special case)

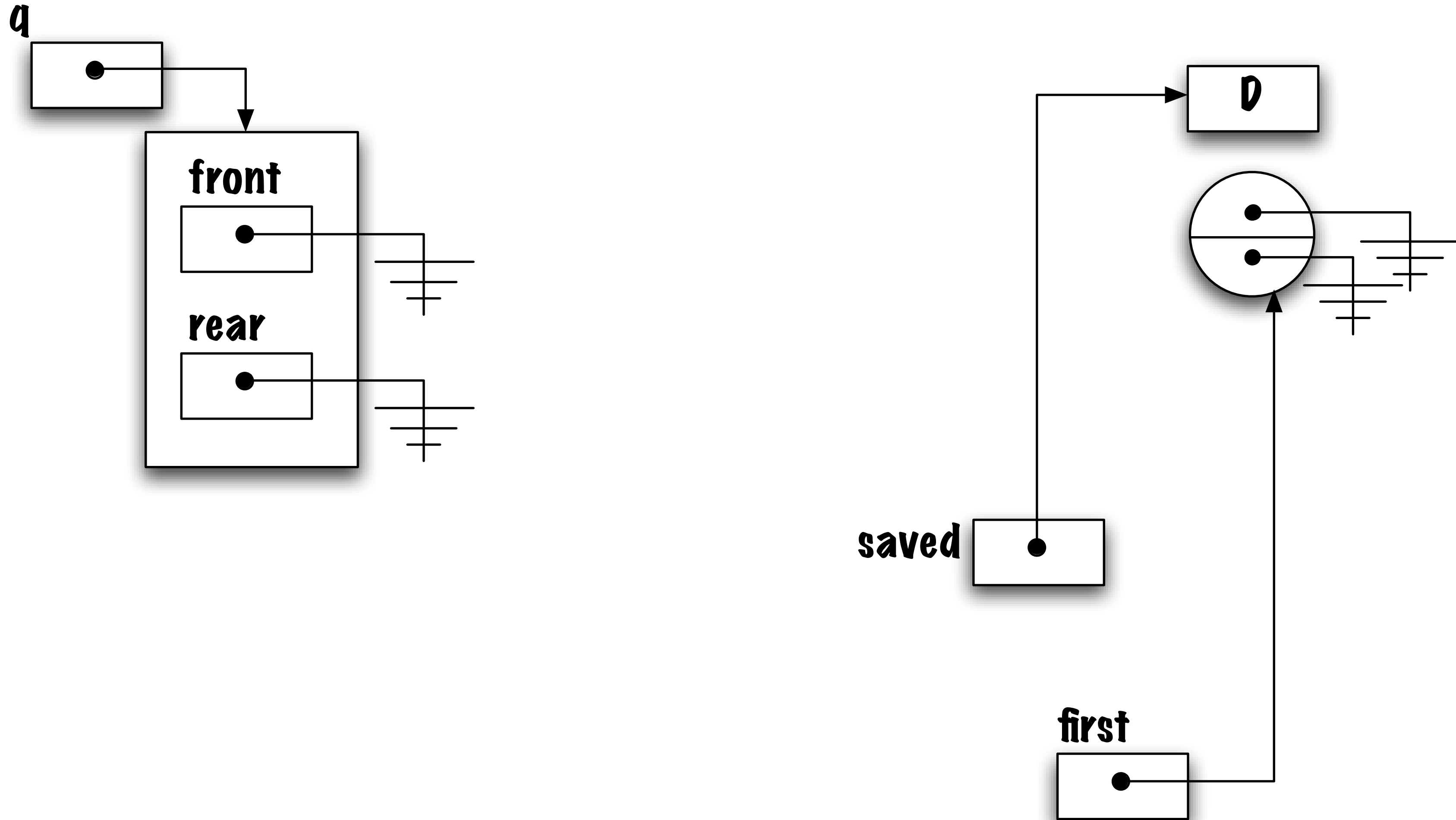


Removing an element (special case)

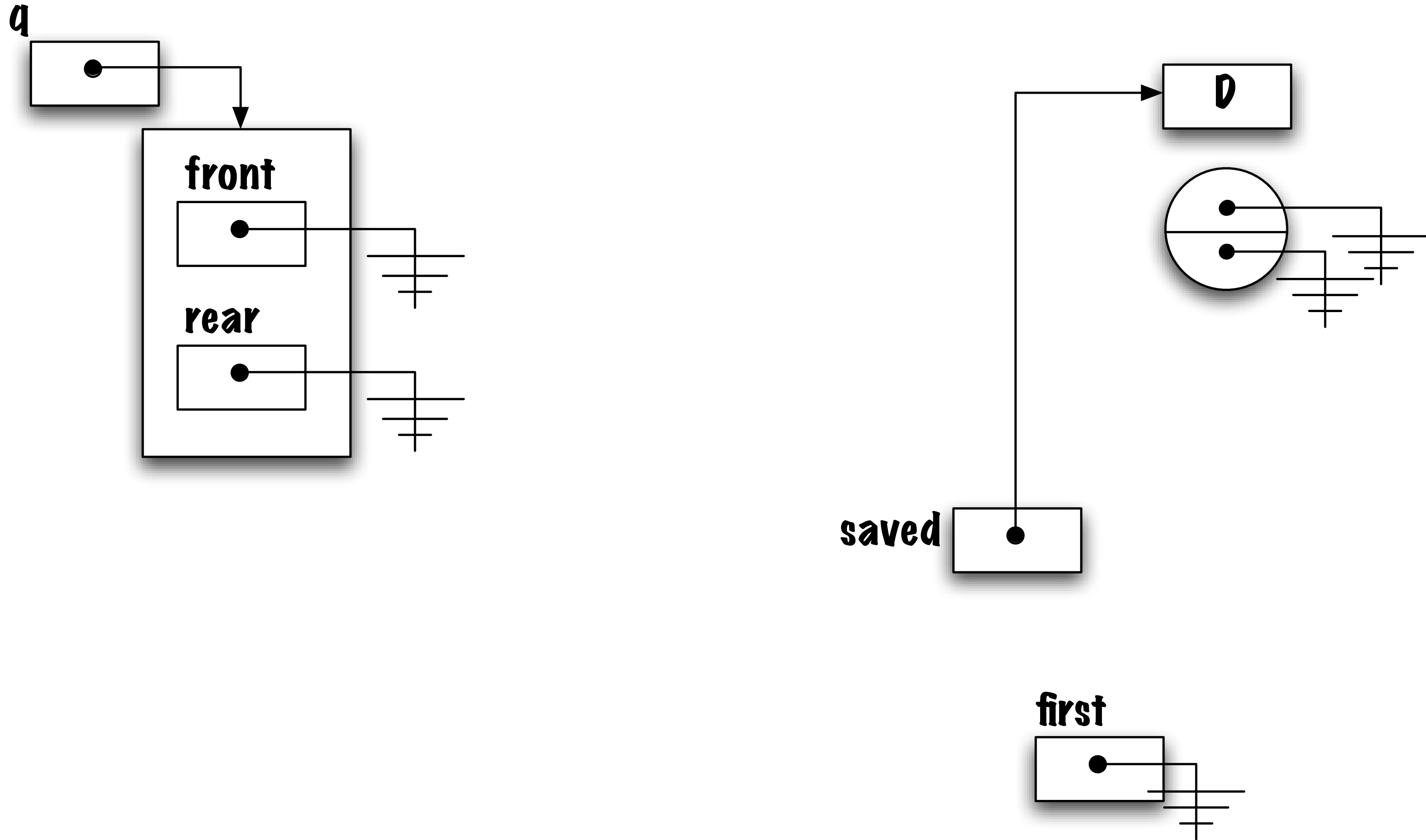




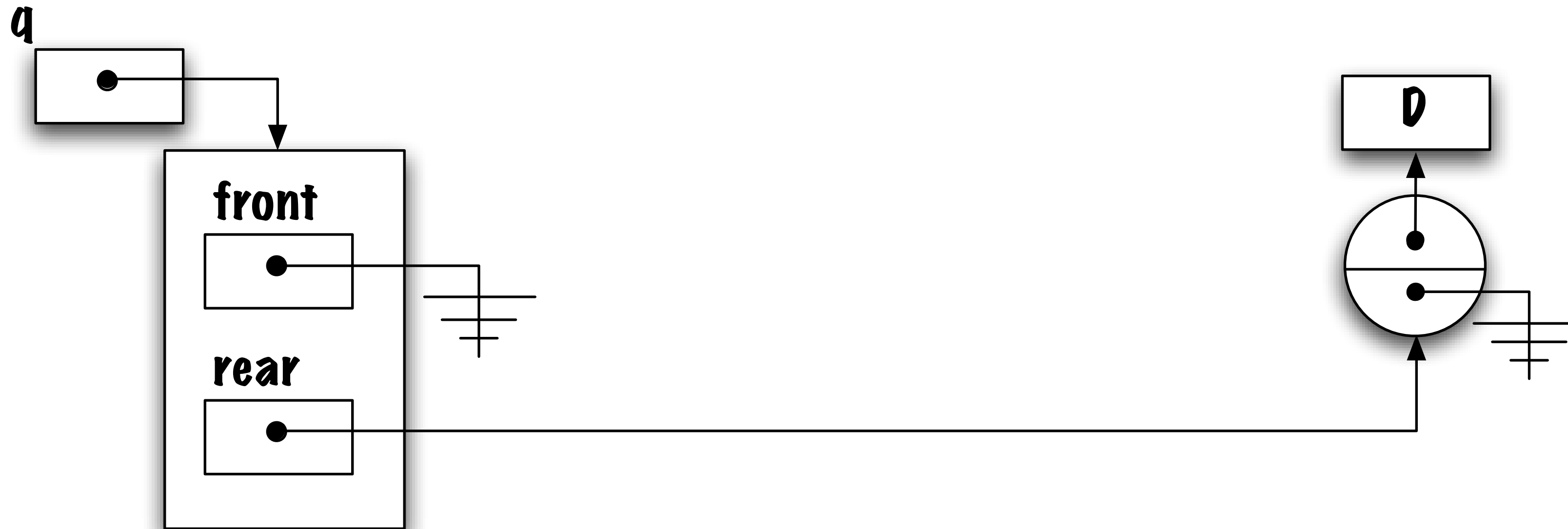
Removing an element (special case)



Removing an element (special case)



Pitfall!



This memory diagram illustrates the kinds of errors that occur frequently with linked elements.

What are the consequences?

Consider using the following test to detect the empty queue: `front == null && rear == null`.