Group 1 (Friday 8th)

- **Lab 5**: queues, linked lists
- (Lab 4 deadline)

Group 2 (Monday 11th)

- **Lab 5**
- (Lab 4 deadline)

Next week is **reading week for SD3**

- Thursday 14th, Friday 15th
- **Group 2 lab is on (Monday 11th)**

Software Development 3 (F27SG2)

Lecture 9

Queues

Rob Stewart

Outline

- By the end of this lecture you should know
  - about the Queue ADT
  - analysis of Queues in Big-O
  - how to use and implement the Queue ADT
    - Using arrays – 1st half
    - Using linked lists – 2nd half

Queues

- A **queue** is related to a stack
- Whilst a stack is LIFO, a queue is FIFO
  - First In First Out
- You are all too familiar with Queues
  - stand in them
- ... in *Computers*
  - access to printers & sending messages

The Queue ADT

- The data is the type of elements stored
- Main operations
  - elements enter a queue at the *rear* and are removed from the *front*
  - **enqueue**(object) - adds object to the rear of the queue
  - **dequeue**() - remove and returns element at the front
- Auxiliary operations
  - **front**() - return the front element
  - **size**() - return number of elements
  - **isEmpty**() - check if empty
- Error conditions: dequeue/front of empty queue

A Queue Example

A Queue Example

Operation	***Output***	*Q: (front,...,rear)*
enqueue(5)	–	(5)
enqueue(3)	–	(5, 3)
dequeue()	*5*	(3)
enqueue(7)	–	(3, 7)
dequeue()	*3*	(7)
front()	*7*	(7)
dequeue()	*7*	()
dequeue()	*“error”*	()
isEmpty()	*true*	()
enqueue(9)	–	(9)
enqueue(7)	–	(9, 7)
size()	*2*	(9, 7)
enqueue(3)	–	(9, 7, 3)
enqueue(5)	–	(9, 7, 3, 5)
dequeue()	*9*	(7, 3, 5)

The Queue ADT Interface

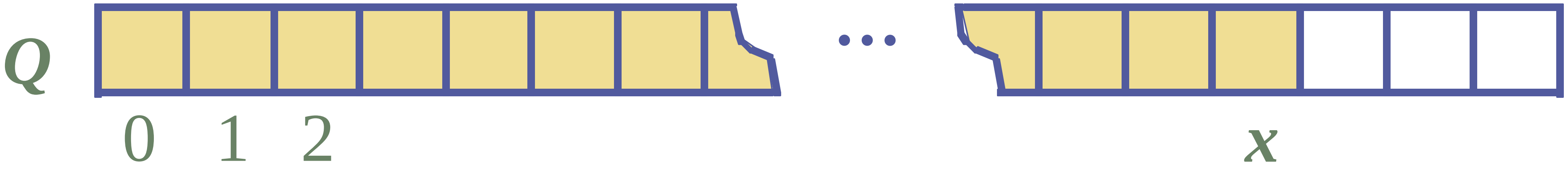
- We define a Queue interface
- Similarly to stacks, we throw a **QueueException** when accessing an empty queue

```
public interface QueueI {
  public int size();
  public boolean isEmpty();
  public Object front() throws QueueException;
  public void enqueue(Object element);
  public Object dequeue()  throws QueueException;
}
```

Naïve queue implementation with arrays

Queue as an Array

- We will show how to implement a queue using an *array Q*
- We could implement the queue in a similar way to stack where a variable **x** either
  1. Points to the **front** of the queue (and the 0 index is the rear), or
  2. Points to the **rear** of the queue (and the 0 index is the front).
- Can you see any problems with these solutions?

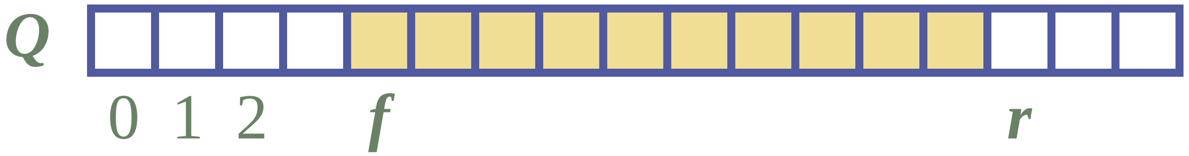


Queue Array Implementation

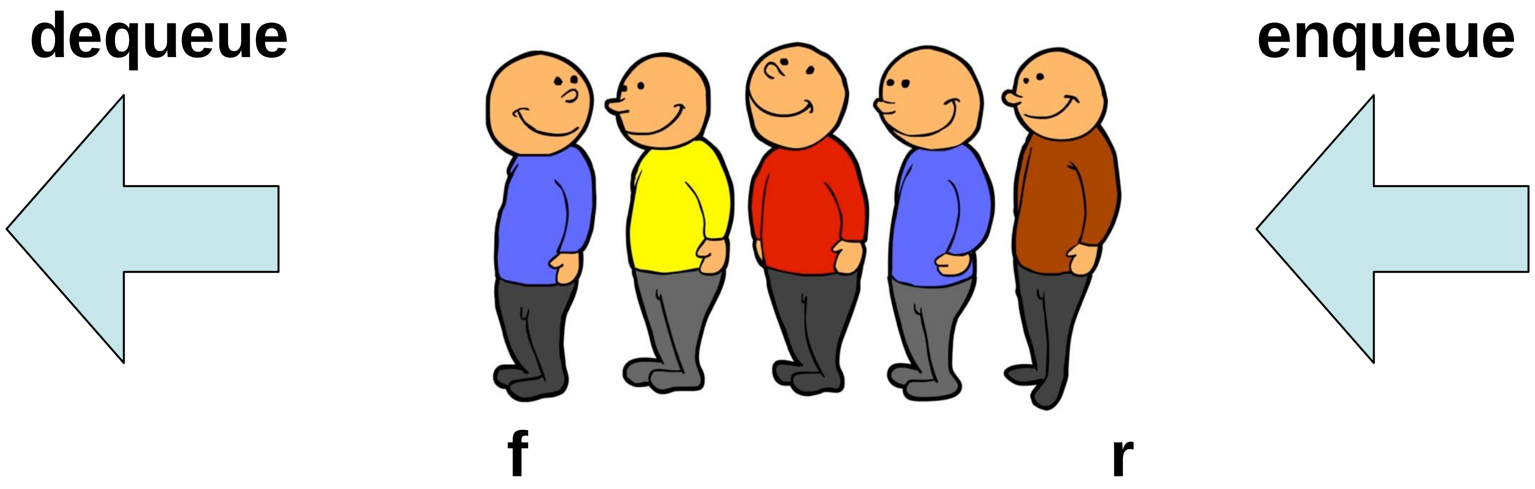
- We would like each operation to be O(1)
  - as with stacks
- These solutions will either result in
  1. Points to the **front** of the queue
     - too slow **enqueue()** operation (1)
       - all elements have to be moved one up when we enqueue
  2. Points to the **rear** of the queue
     - too slow **dequeue()** operation (2)
       - all elements have to be moved one down when we dequeue
  - One operation forces a shift of each element
    - **Linear time!**

Queue Array Implementation

- Instead we represent (array) Queue **Q** with 2 variables:
  - a variable **f** indicating the front of the queue
  - a variable **r** indicating the rear (next free element) of the queue



- New elements are added to the rear (increase **r**) and elements are removed from the front (increase **f**)



Queue Array Implementation

```
public class Queue implements QueueI {
   private int f; // front of queue
   private int r; // rear of queue
   private int capacity;
   private Object[] Q;
   private static int MAX = 100;
```

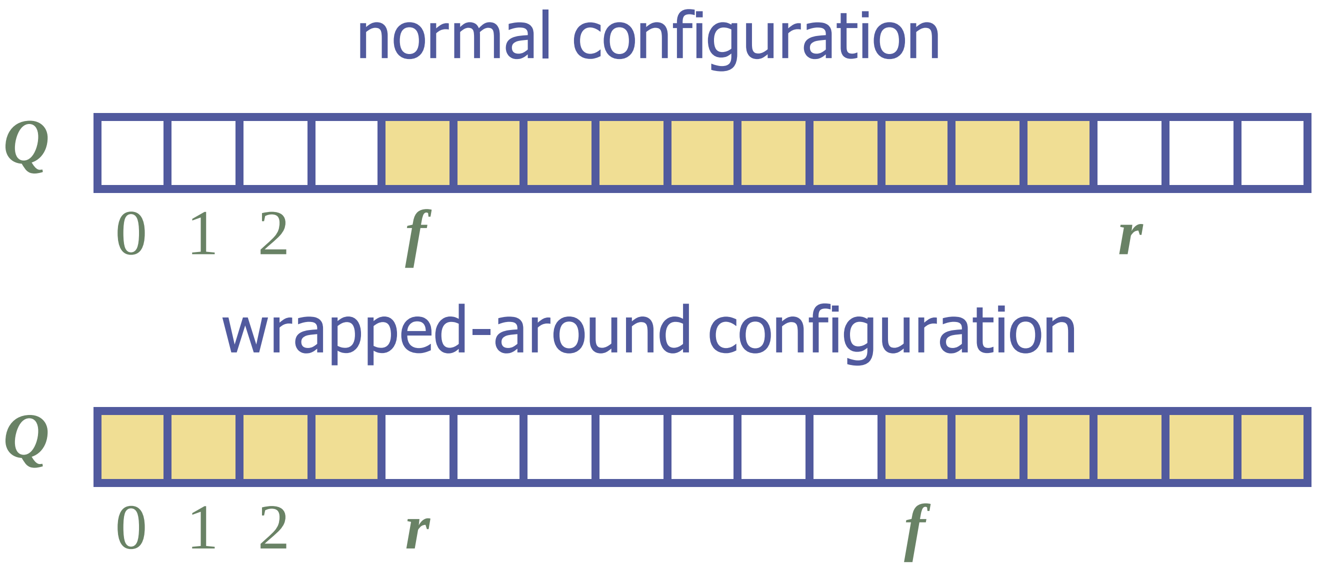
Queue Implementation Constructors

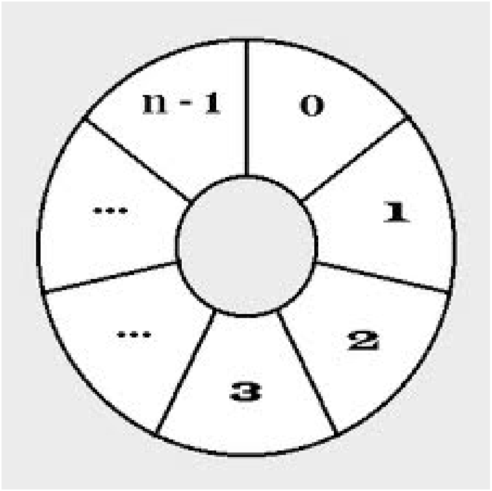
```
public Queue(int capacity){
   this.capacity = capacity;
   Q = new Object[capacity];
   f = 0;
   r = 0;
}

public Queue(){
   this(MAX);
}
```

Queue Implementation
“Wrap Around”

- What happens when **r** reaches the end of the array
  - … and there may still be free elements before **f**?
- A solution is to “wrap” **r** around to the front of the list





- This can be seen as a circular list
  - where the last element is followed by the first

Queue Implementation Full vs. Empty Queue

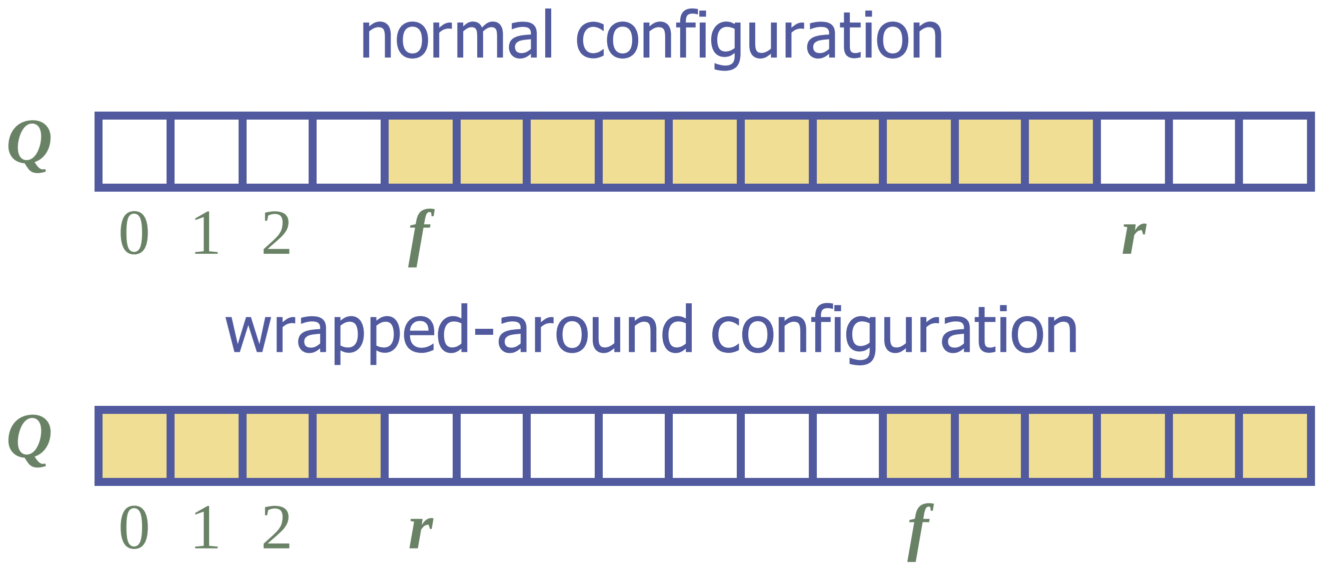
- Initially, the queue is **empty**, which we represented as **f=r**.
- How do we know if a queue is full?
- Technically, this is when **f=r,** but then how do we differentiate a full queue from an empty queue?
- Possible solutions
  1. boolean variable to indicate if the queue has elements
     - e.g. this can be **true** when the queue contains elements
  2. do not allow the queue to become full
     - there must always be one free element (i.e. size is N-1)
     - **We will implement this solution**

Queue Implementation the Use of Modulo

- How do we find the size of the queue
  - For a “normal configuration”: **r-f**
  - For a “wrap-around configuration”:

    (capacity - f + r) % capacity

Doc cam worked example...



Queue Array Implementation Operations

```
public int size(){
    return (capacity - f + r) % capacity;
}
public boolean isEmpty(){
    return (f==r);
}
```

Queue Array Implementation Operations: enqueue/dequeue

Queue Array Implementation Operations

```
public Object front() throws QueueException {
    if (isEmpty())
       throw new QueueException("Queue is empty.");
    return Q[f];
 }

public Object dequeue() throws QueueException {
    if (isEmpty())
      throw new QueueException("Queue is empty.");
    Object tmp = Q[f];
    Q[f] = null;
    f = (f+1) % capacity;
    return tmp
}
```

Queue Implementation Operations

```
public void enqueue(Object e) throws QueueException {
    if (size() == capacity - 1)
      throw new QueueException("Queue is full.");
    Q[r] = e;
    r = (r+1) % capacity;
}
```

Big-O for Queue Operations

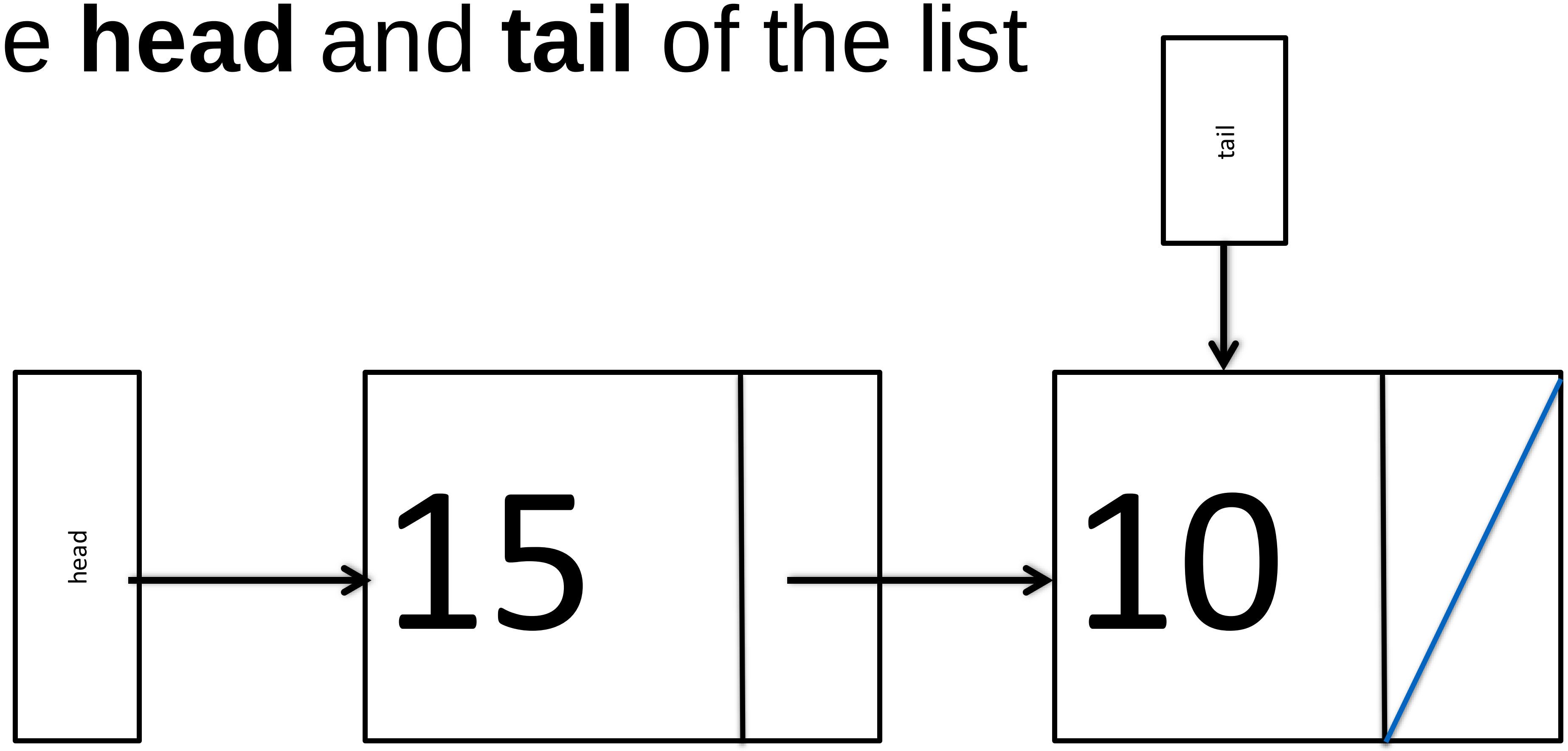
- The number of primitive operations
  - are the same for all queue operations

- All queue operations are constant time **O(1)**

- This can be shown in the same way as we did for stacks:
  1. primitive counting
  2. Simplify Big-Oh formula

Queues as linked lists

- We can also implement a queue as a **linked list**
- We will enqueue to one side of the list
    - and dequeue to the other end
- For efficiency reasons we therefore keep track of the **head** and **tail** of the list



Enqueue

- As for a stack we keep track of
  - the **size** as a separate variable.
- We will therefore only discuss two key operations:
  - Enqueue
  - Dequeue

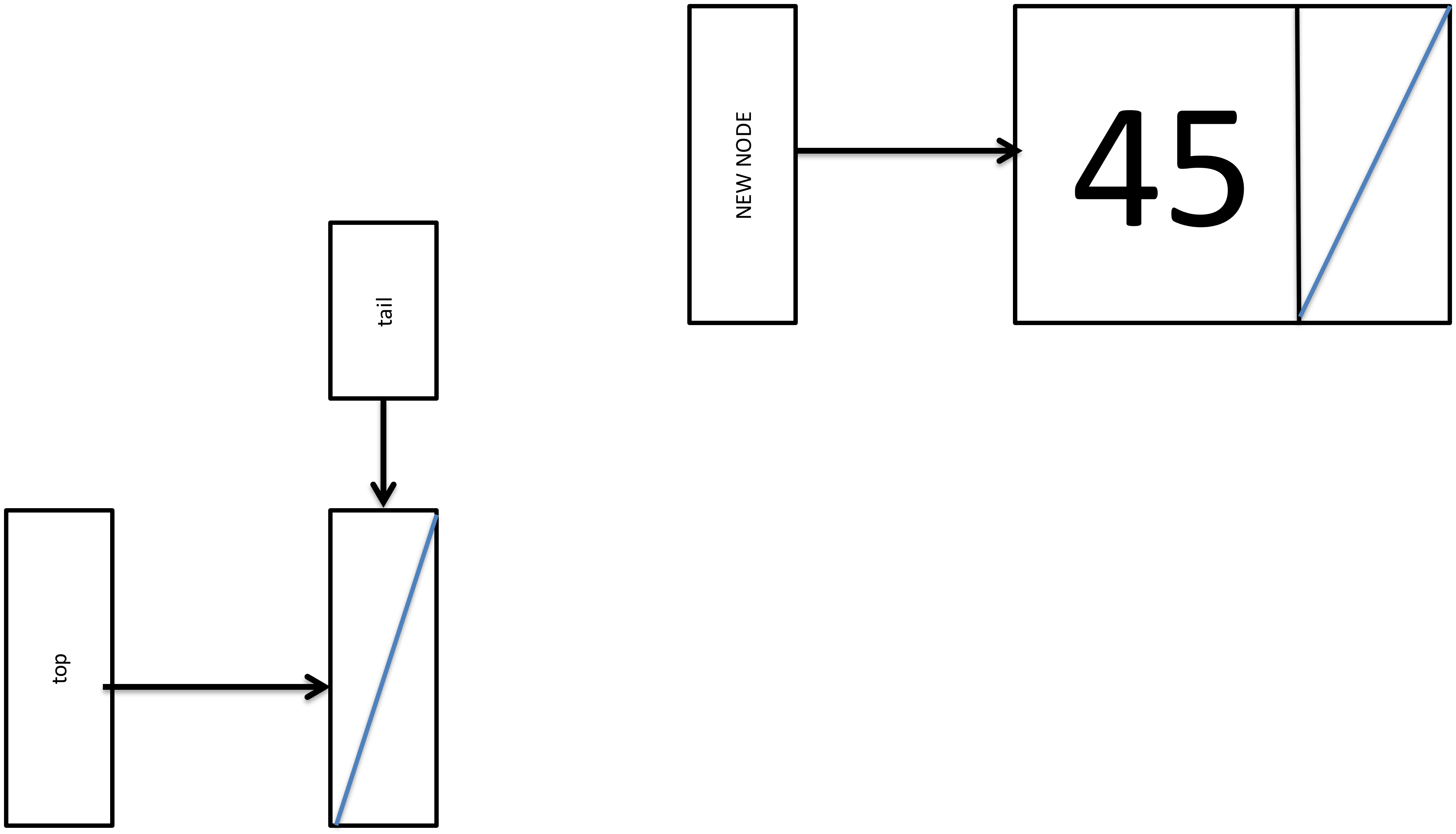
- Implementing this is **lab 5, question 3**
- Doc cam worked example

Enqueue an element

- A new object is **enqueued** to the linked lists
  - Generate a new node
  - And add it to the rear of the list
- Special care has to be taken when the list is initially empty

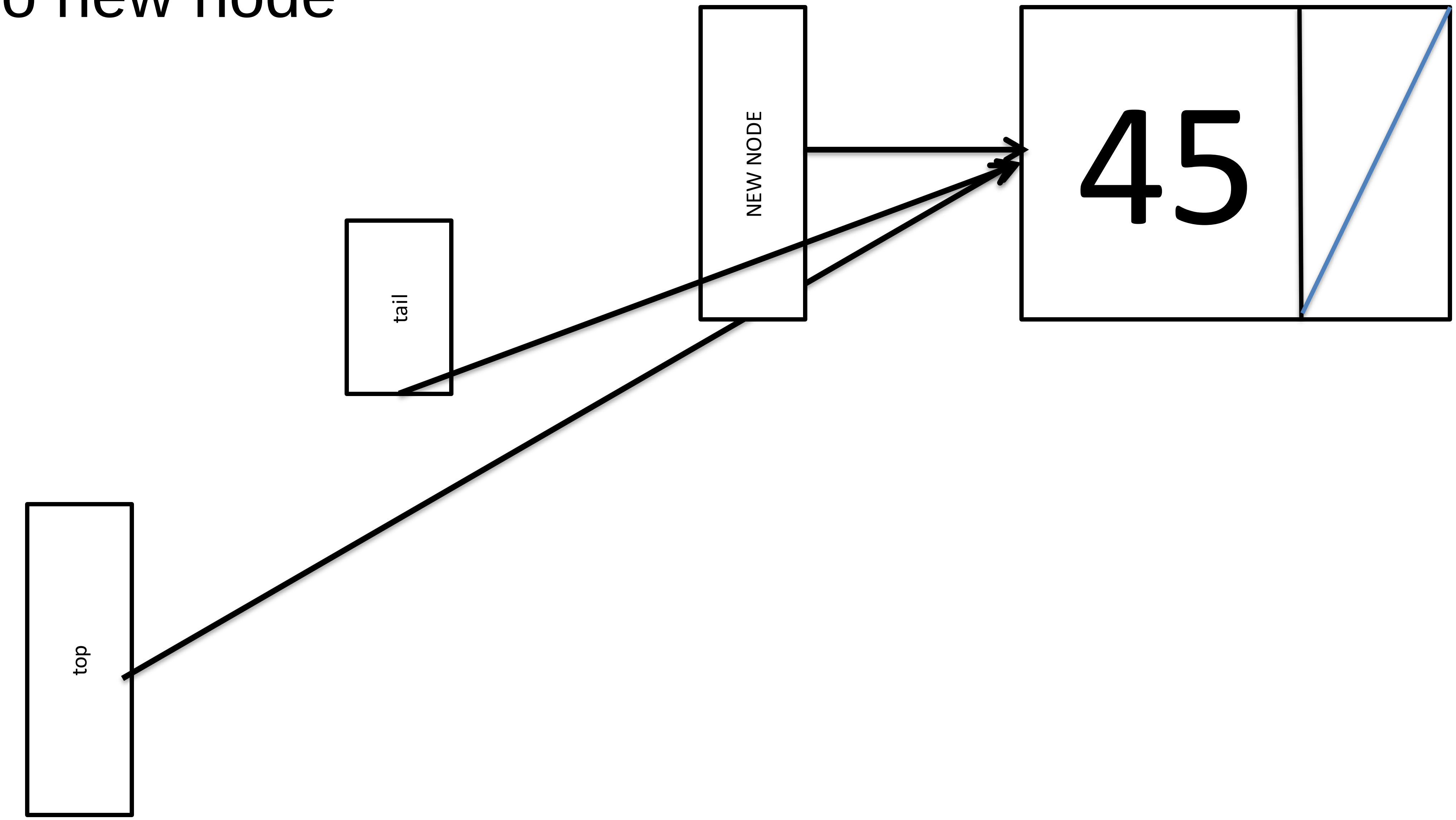
Enqueue for empty queue

1. Create the new node



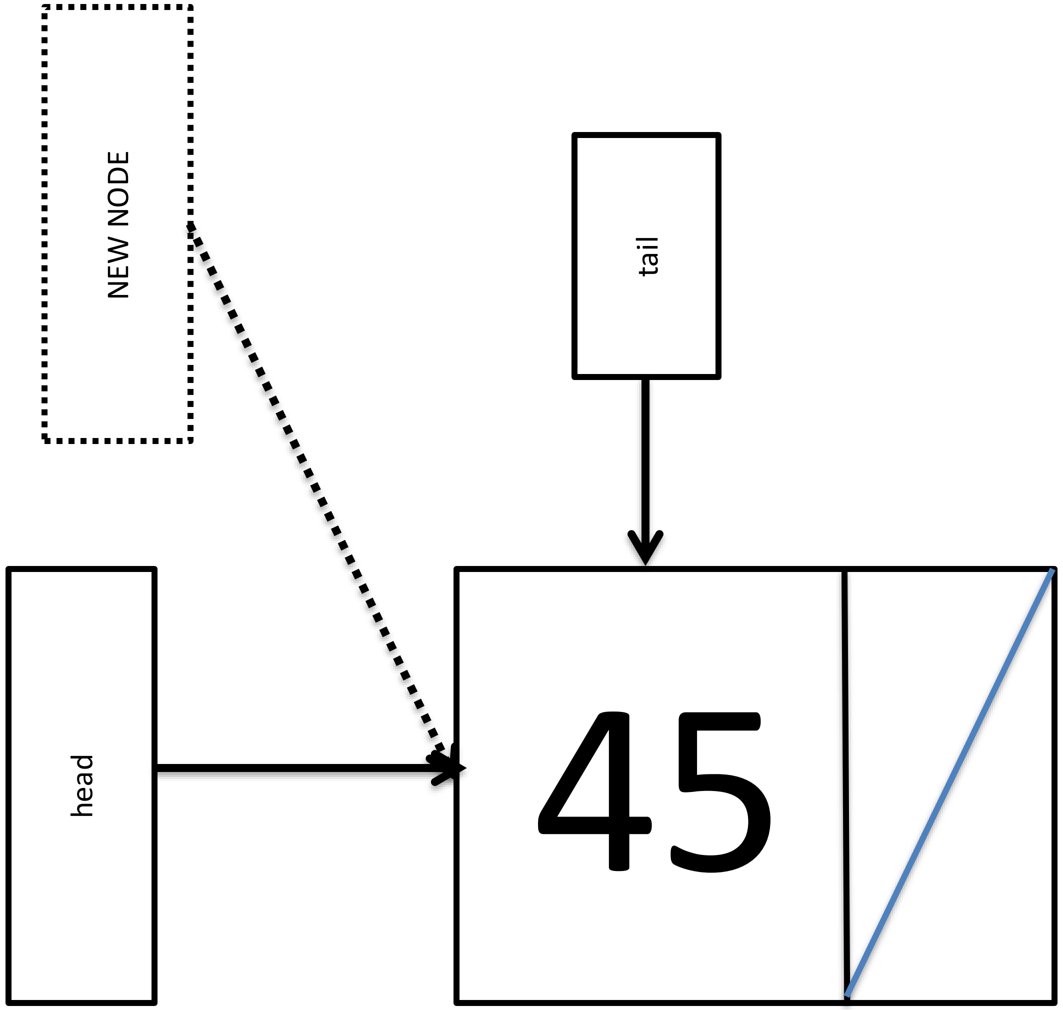
Enqueue for empty queue

2. Set head and tail to point to new node



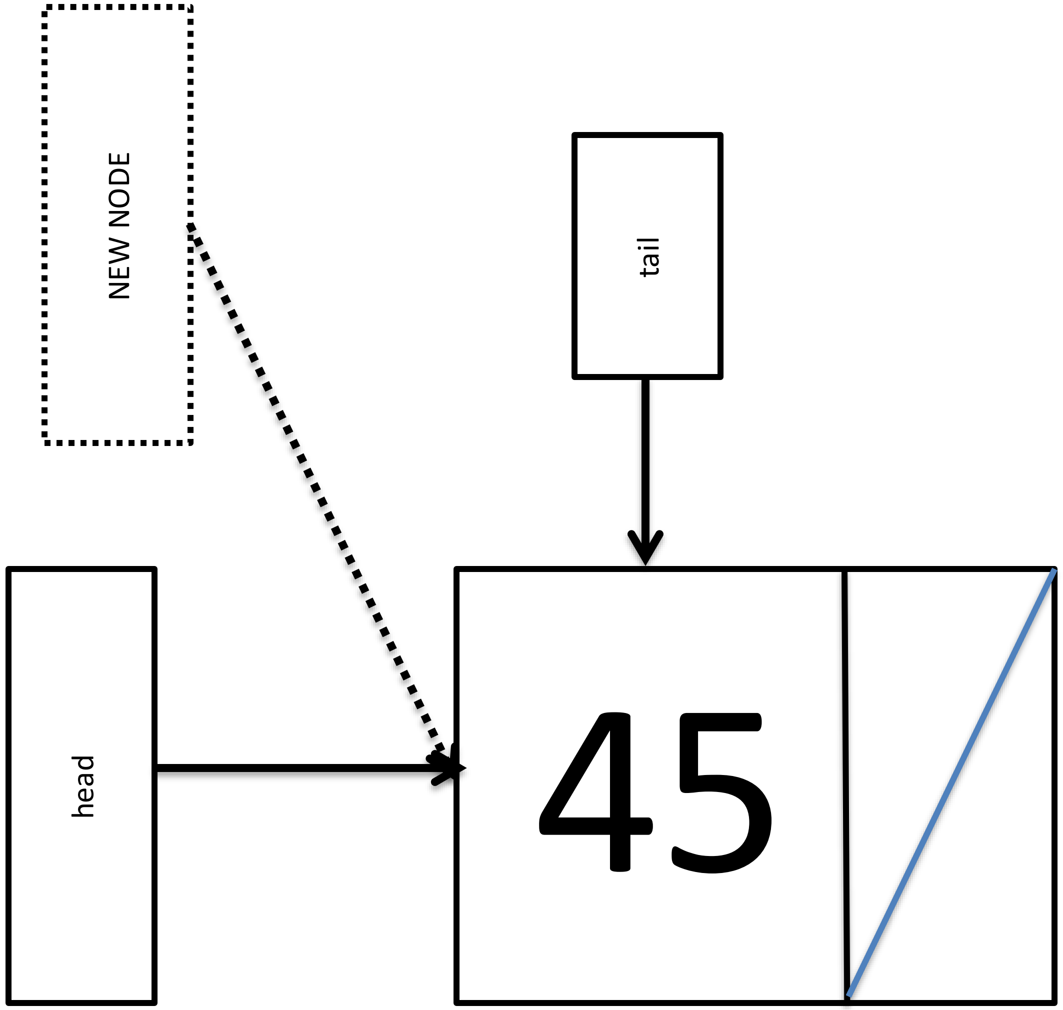
Enqueue for empty queue

Same as:



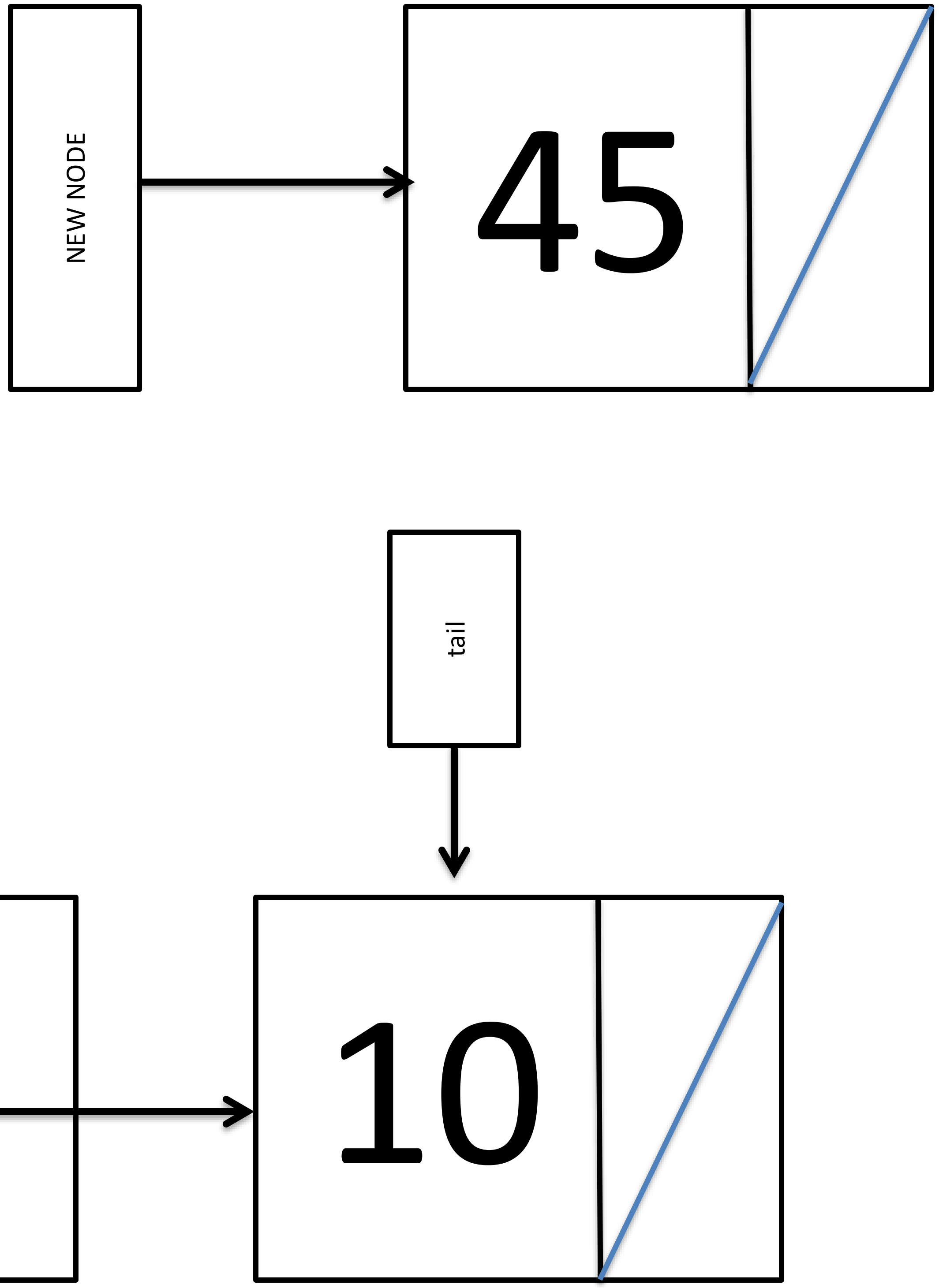
Enqueue for empty queue

3. Increment **size** by one



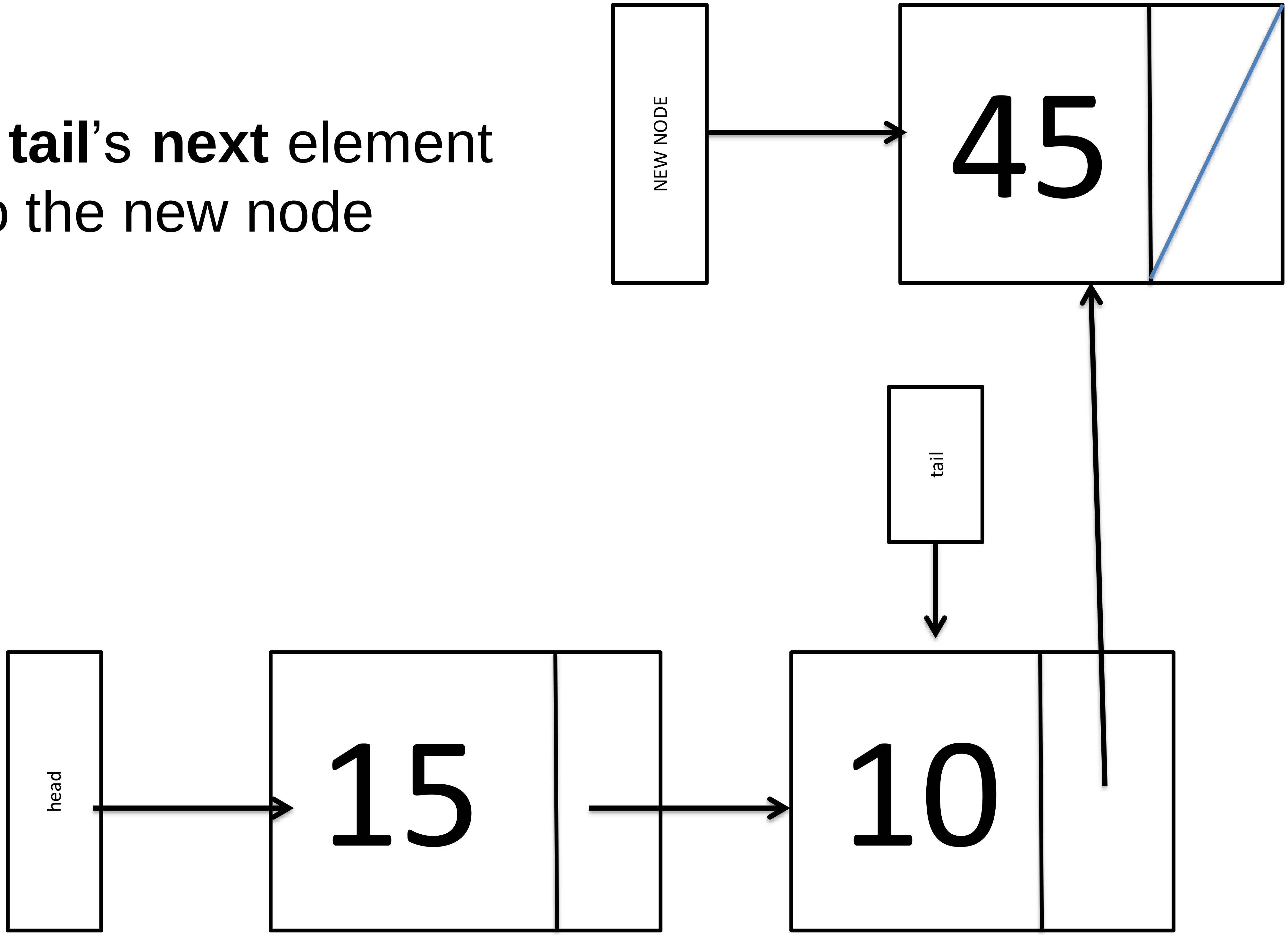
Enqueue non-empty queue

1. Create the new node



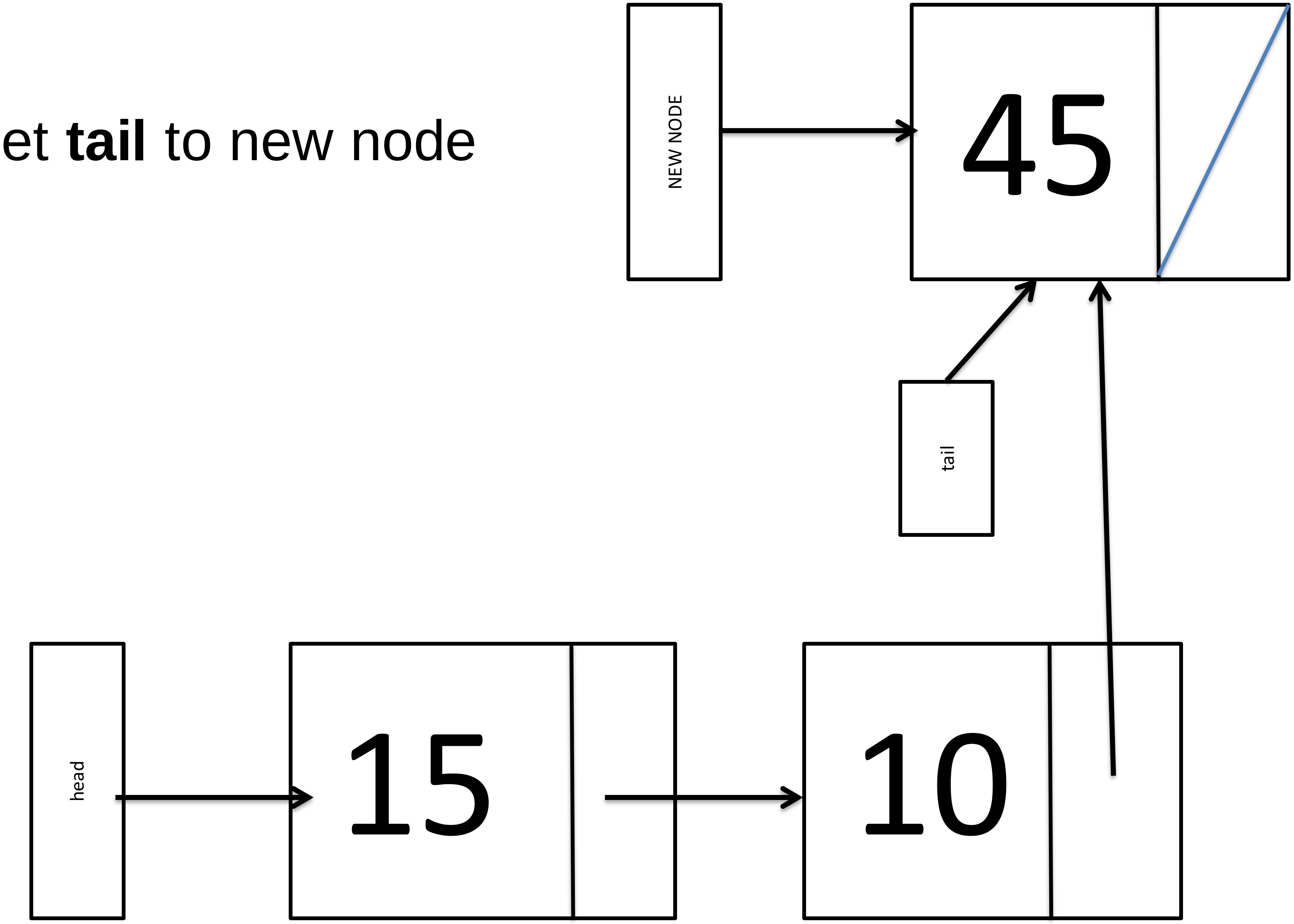
Enqueue non-empty queue

2. Set **tail**'s **next** element to the new node



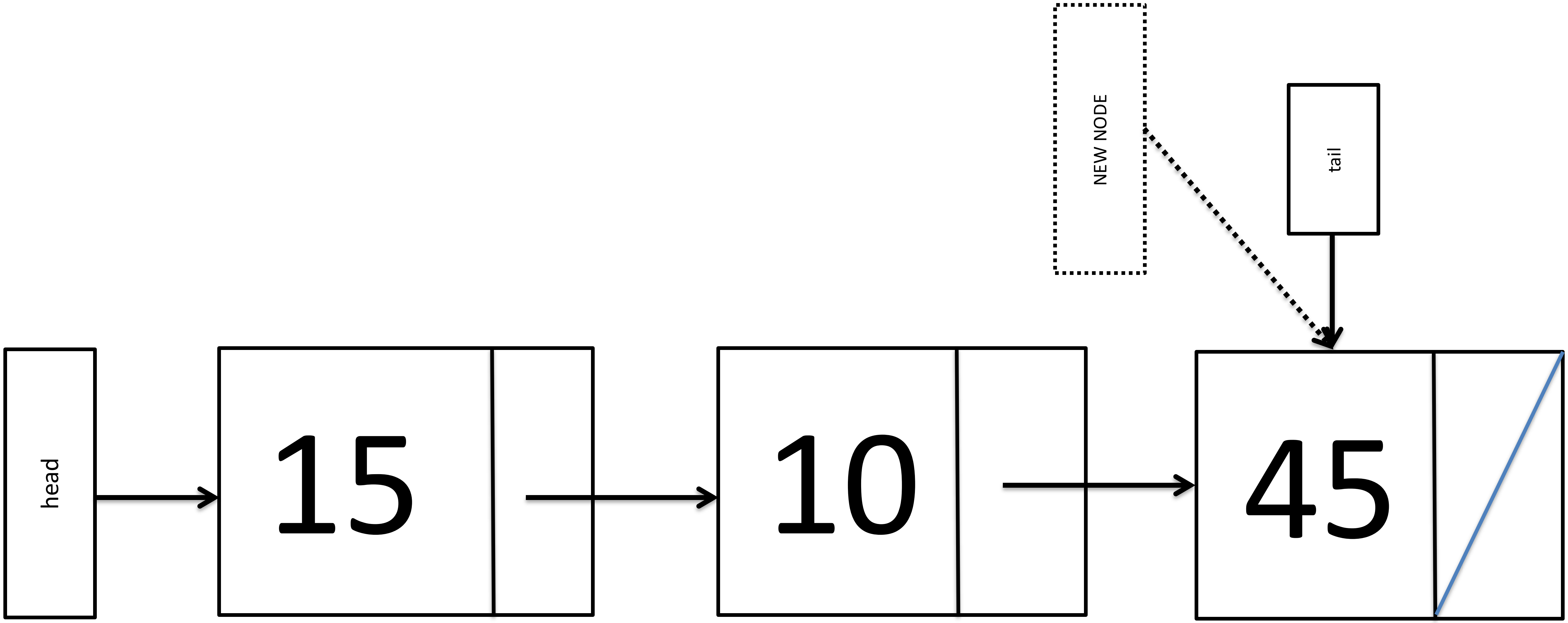
Enqueue non-empty queue

3. Set **tail** to new node



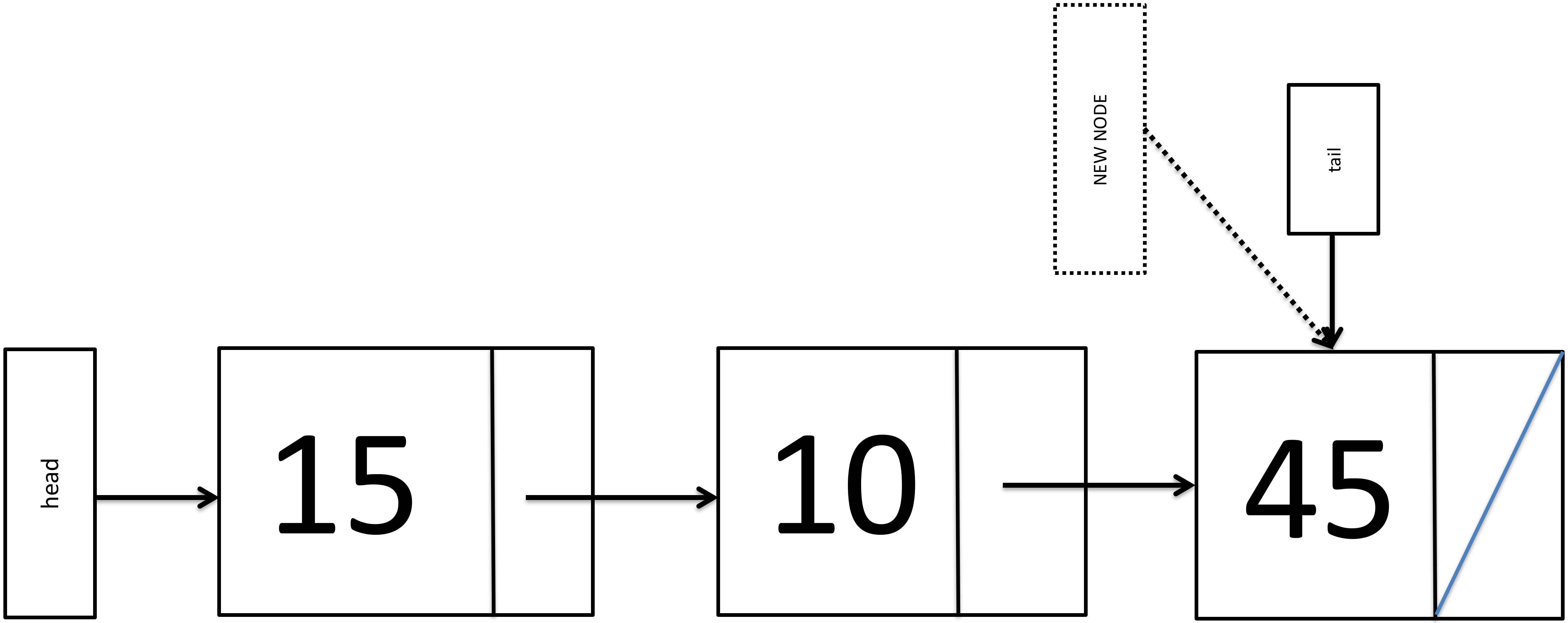
Enqueue non-empty queue

… which is the same as



Enqueue non-empty queue

4. Increment **size** by one

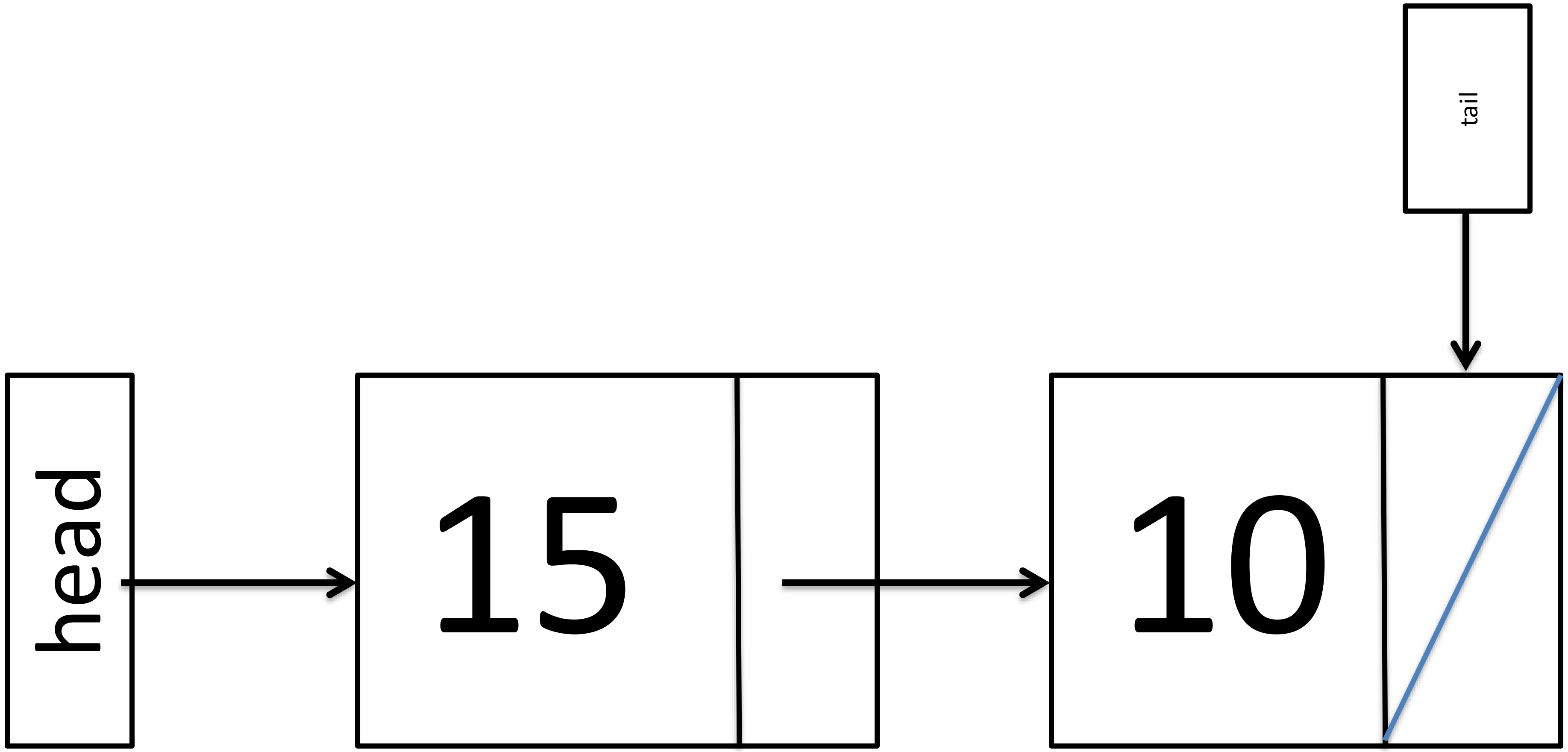


Dequeue an element

- An element is **dequeued** from the linked lists
  - Get the head element
  - Make head point to the next element
  - Decrement size
  - If new queue is empty, set **tail** to **null**
  - Return (old) head element
- If it is empty a **QueueException** is raised
- Note: almost identical to how we popped a stack

Dequeue

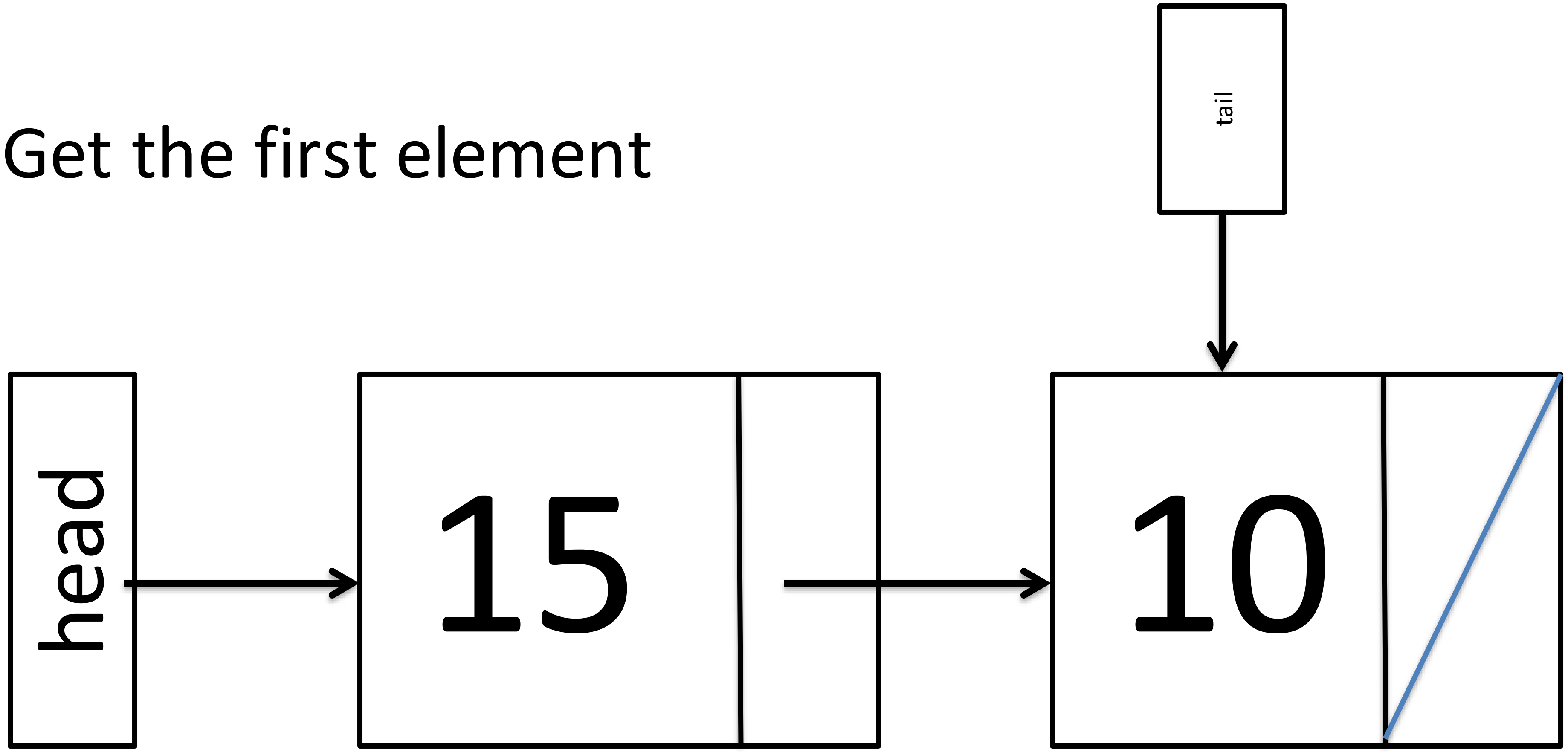
1. Check the queue isn’t Null



Head == NULL throws a **QueueException**

Dequeue

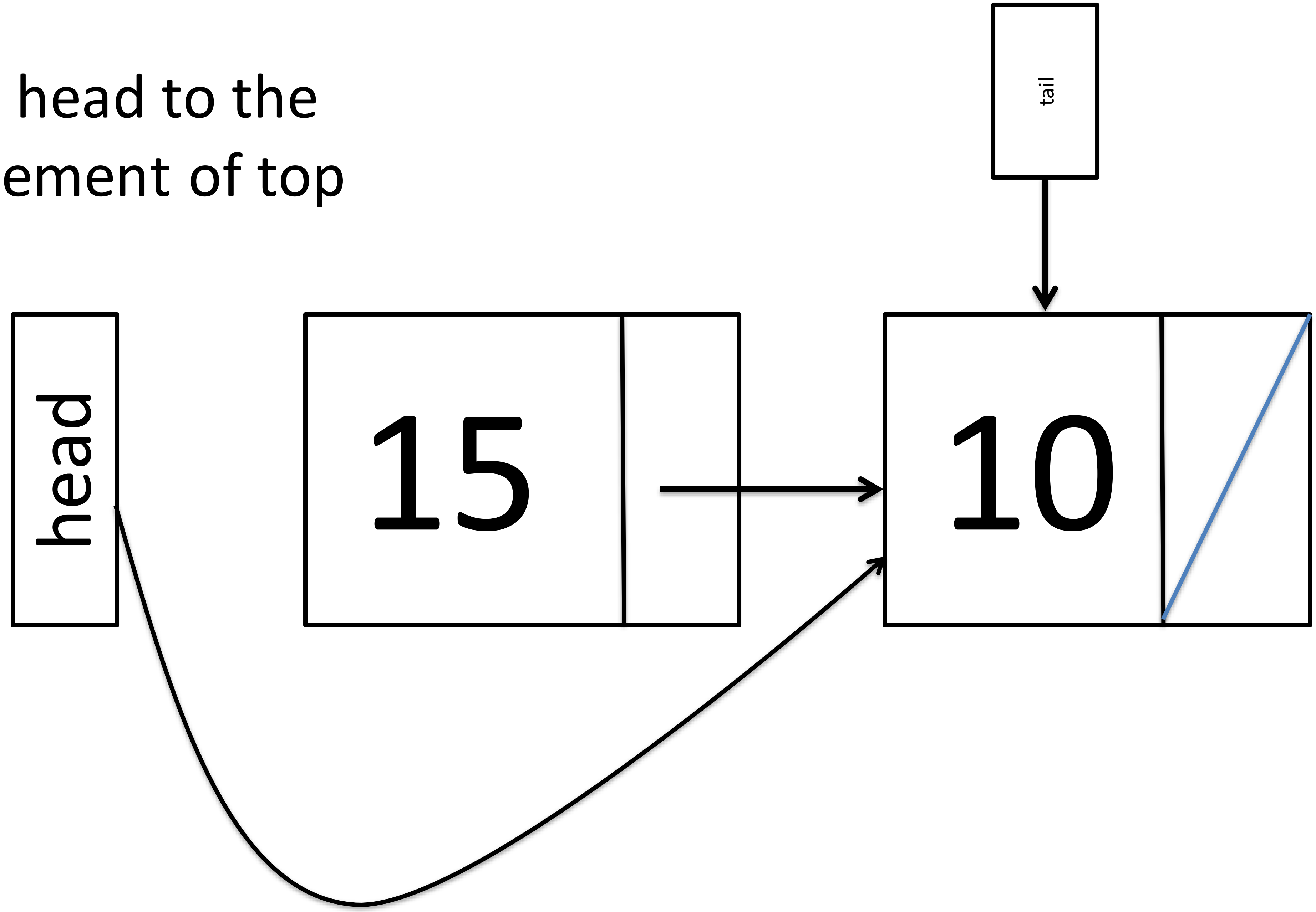
2. Get the first element



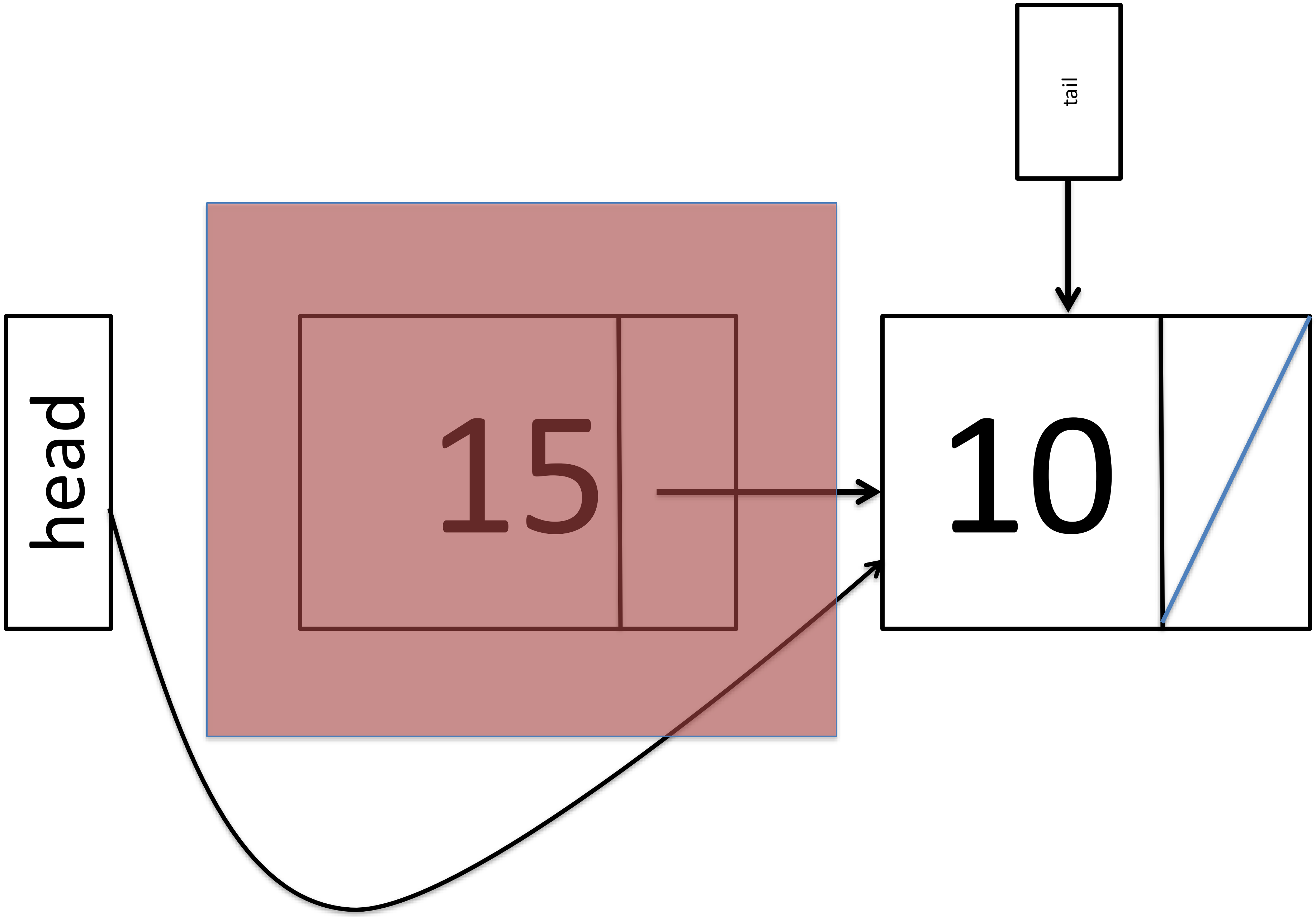
tmp = 15;

Dequeue

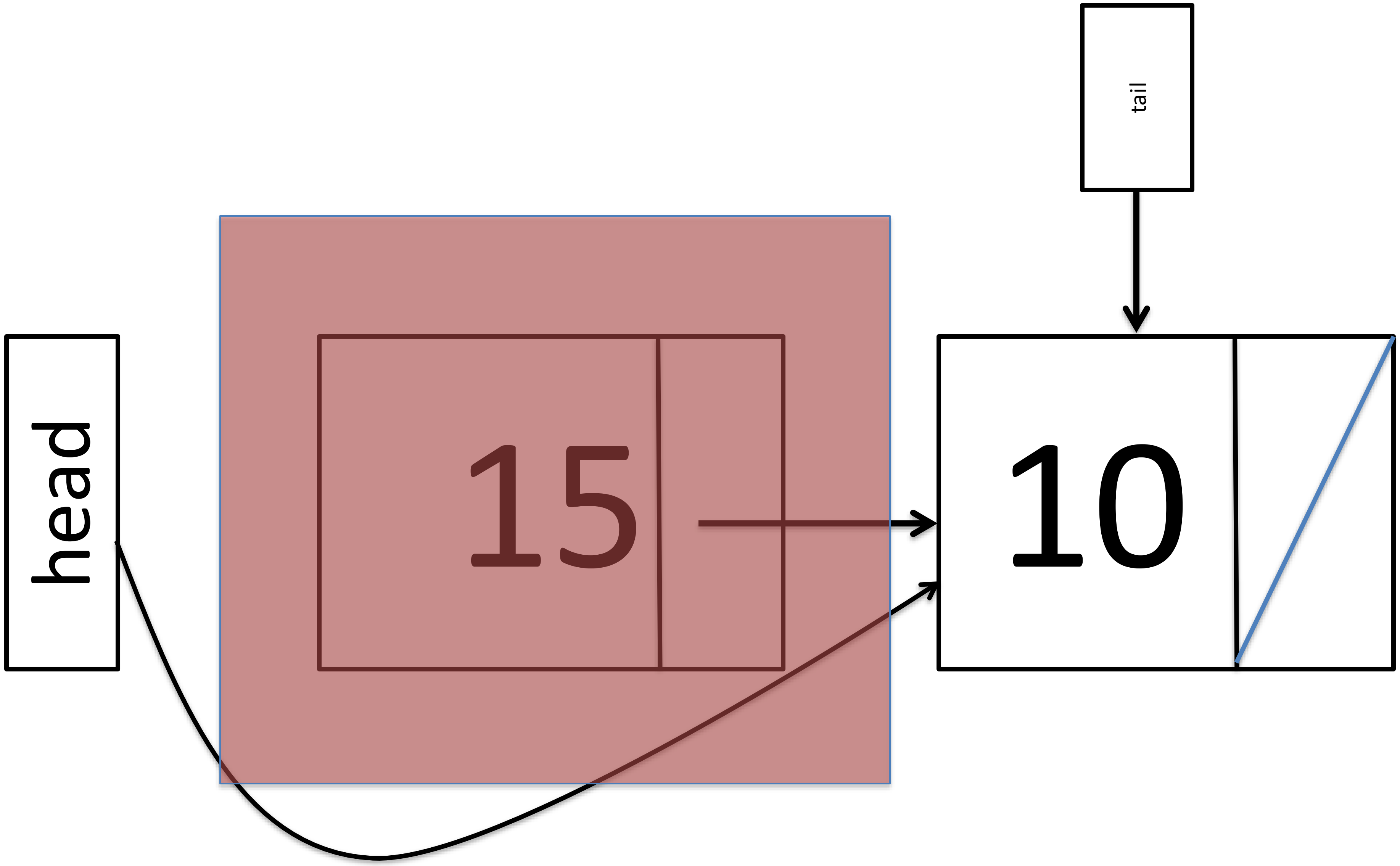
3. Set head to the next element of top



Dequeue

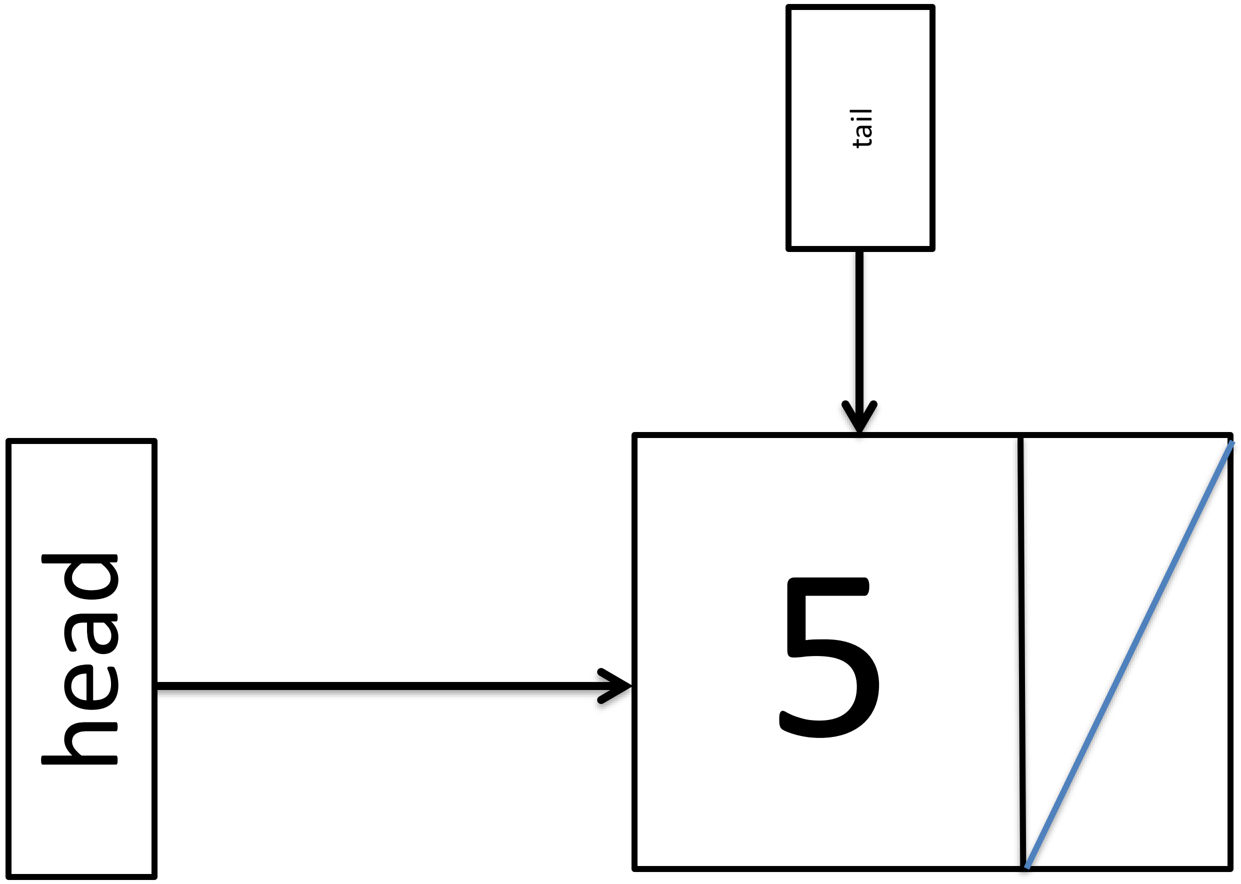


Dequeue



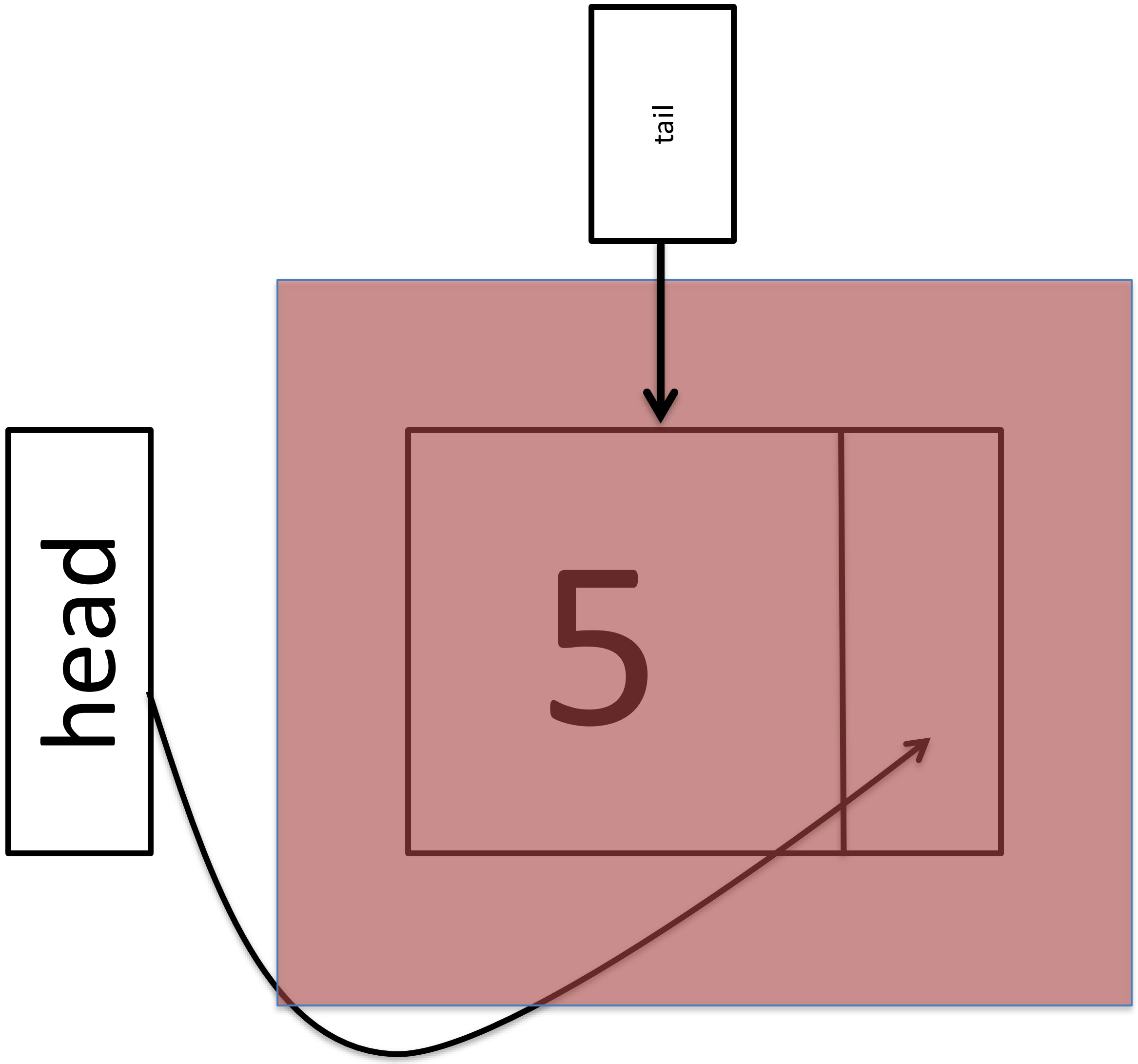
4. Set **tail** to NULL if new queue is empty

Dequeue (special case)



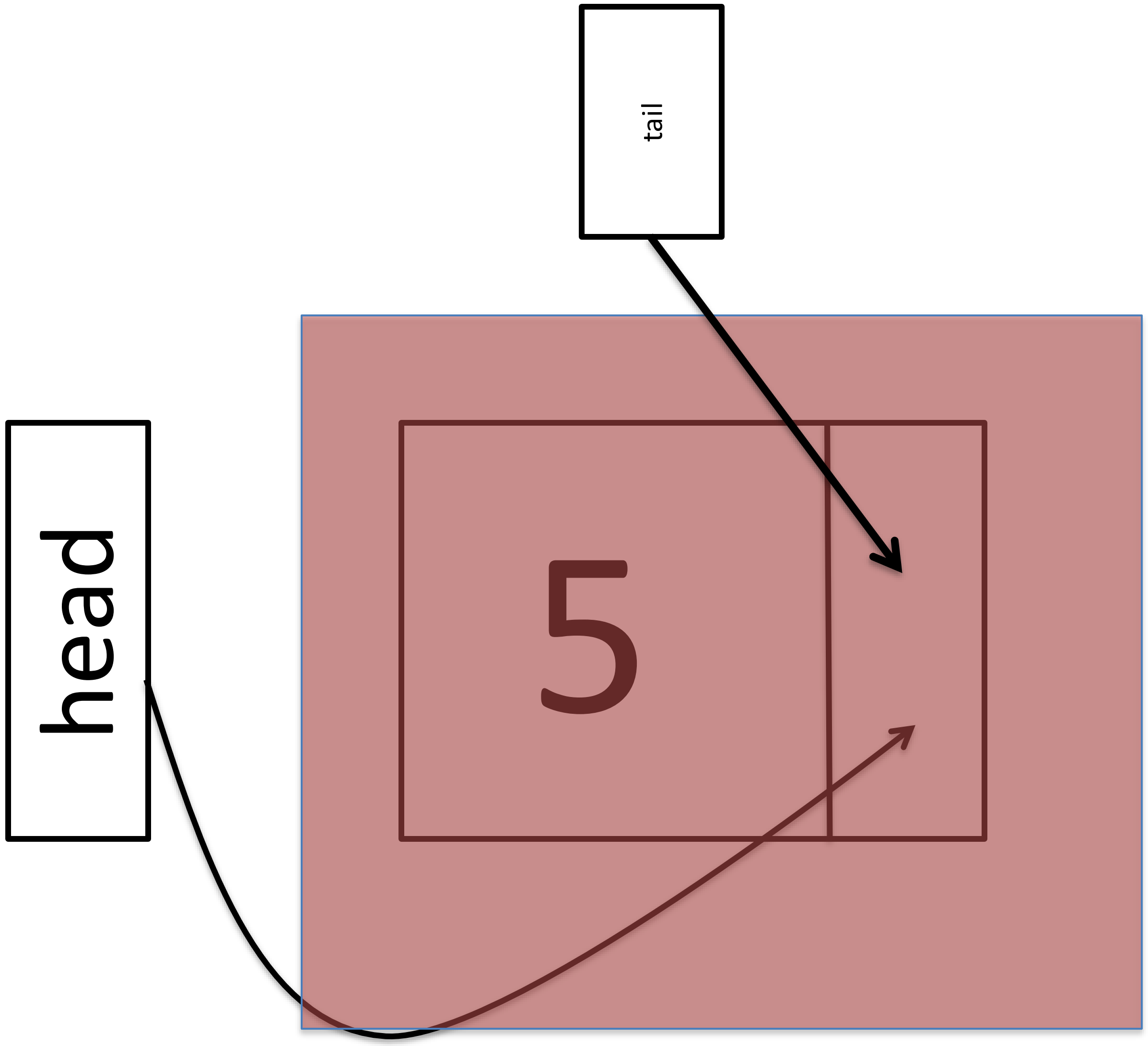
4. Set **tail** to NULL if new queue is empty

Dequeue (special case)



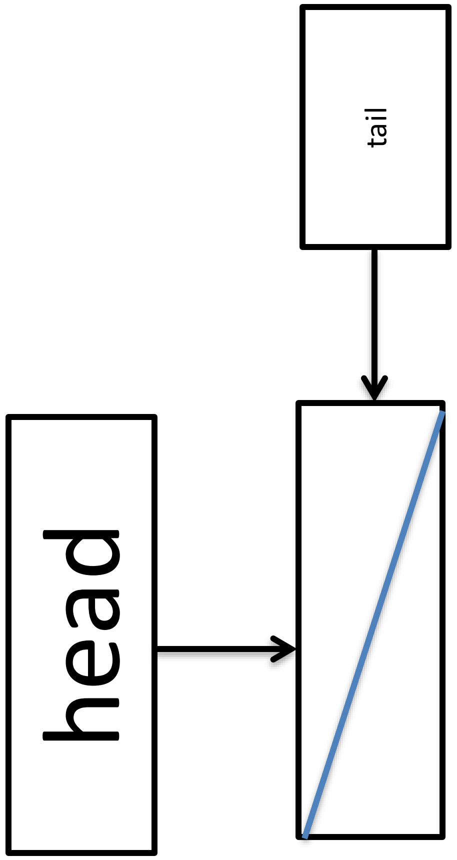
4. Set **tail** to NULL if new queue is empty

Dequeue (special case)



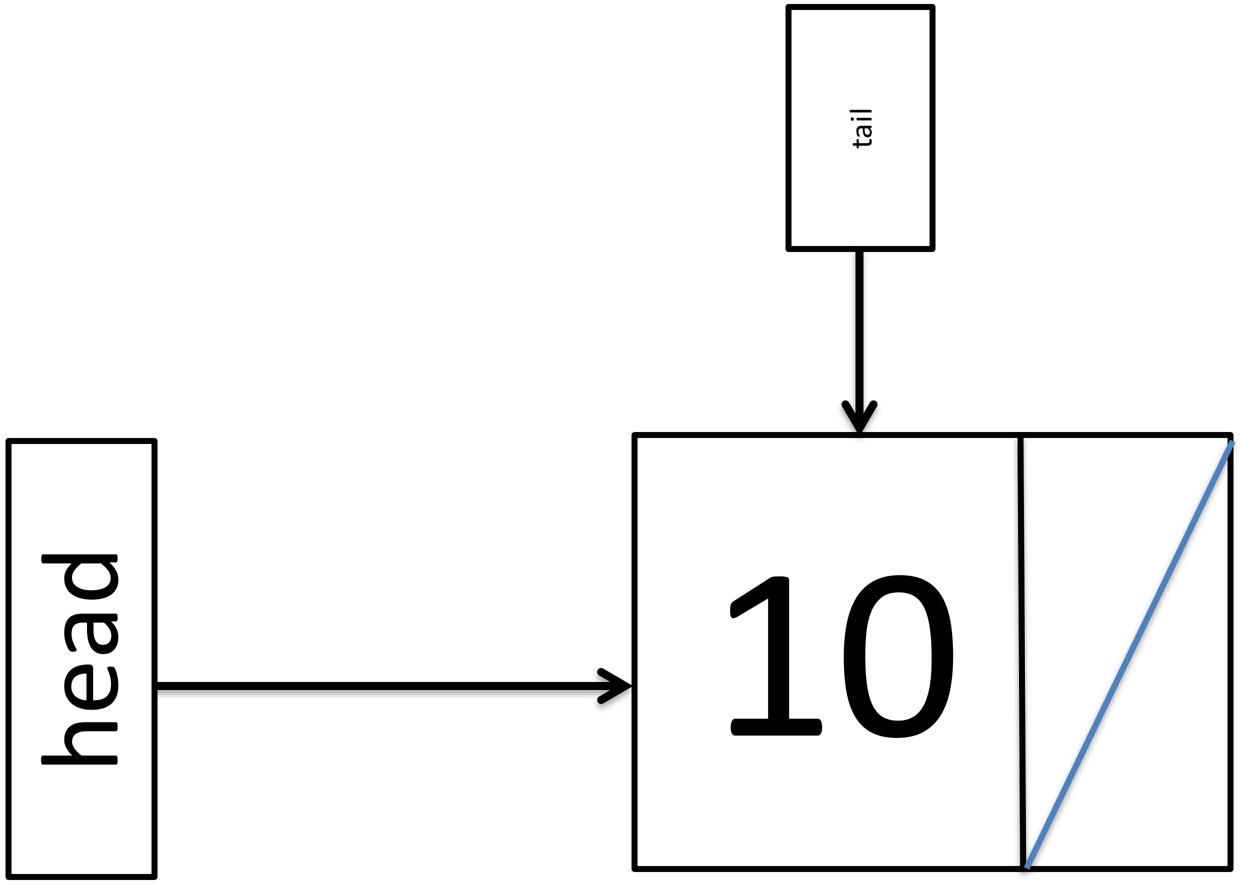
4. Set **tail** to NULL if new queue is empty

Dequeue (special case)



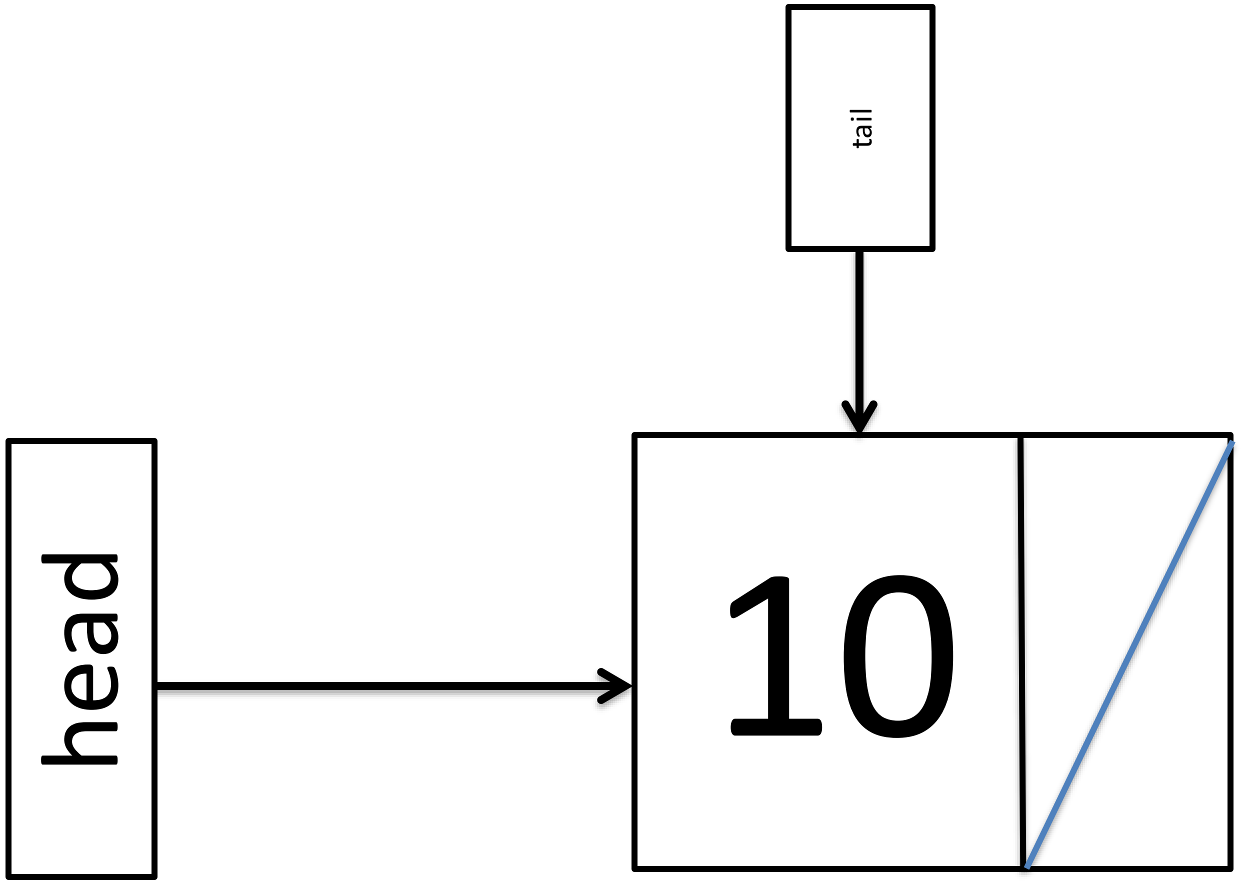
4. Set **tail** to NULL if new queue is empty

Dequeue



5. Decrement **size** by 1

Dequeue



6. Return the tmp element
(15 in this case)

Queue Exercise

- **enqueue**(object) - adds object to the rear of the queue
- **dequeue**() - remove and returns element at the front
- **front**() - return the front element
- **size**() - return number of elements
- **isEmpty**() - check if empty

Suppose an initial empty queue S applied: 32x enqueue operations, 10x front, 15x dequeue where 5 dequeue operations raised an exception since the queue was empty. What is the size of the queue after these operations?

www.socrative.com / `student login' / room: SD32018

Queues space race

Summary

- We have introduced the Queue ADT
- Discussed Big-O analysis of Queues
- We have shown how to represent queues
  - Using arrays (1st half)
  - Using linked lists (2nd
- We have shown how to implement queues
  - Using arrays

- Next lecture: doubly linked list