Introductory Data Structures

CO518: Algorithms, Correctness and Efficiency

Scott Owens (S.A.Owens@kent.ac.uk)

Warning: These slides do not necessarily follow the method names or descriptions from the Java standard library

Last updated: 17/10/17

Abstract Data Types

Separate interface from implementation

- Growable arrays
- Stacks (for graph searching, implementing recursive functions, managing resources, …)
- Queues (for graph searching, scheduling, …)
- Dictionaries (use practically everywhere)
- Priority queues (for graph shortest paths, scheduling, …)

Growable Arrays

- `T get(int i)`: get the element at position i
- `void add(T v)`: add v to the end, growing the array
- `int size()`: the current size
- `int put(int i, T v)`: assign the element at position i the value v, growing the array if necessary

Stacks

- LIFO (last in, first out)
- Analogy with a physical stack of things (plates, trays, etc.)
- `void push(T v) throws StackFull`: put v on top of the stack
- `T pop() throws StackEmpty`: remove the top element, if any
- `bool isEmpty()`: check if the stack is empty

Queues

- FIFO (first in, first out)
- Analogy with a physical queue of people
- `void enqueue(T v) throws QueueFull`: put v at the back of the queue
- `T dequeue() throws QueueEmpty`: remove the element at the front of the queue, if any
- `bool isEmpty()`: check if the queue is empty

Dictionary

- Analogy with a physical dictionary
- Associate *values* with *keys* (e.g., definitions with words, phone numbers with names, enrolment data with student id numbers)
- `void add(T1 key, T2 value)`: add an entry to the dictionary
- `T2 lookup(T1 key) throws KeyNotFound`: finds the value associated with the key
- `void remove(T1 key) throws KeyNotFound`: removes the entry for key

The types that can be used for keys can depend on how the dictionary is implemented

Priority Queues

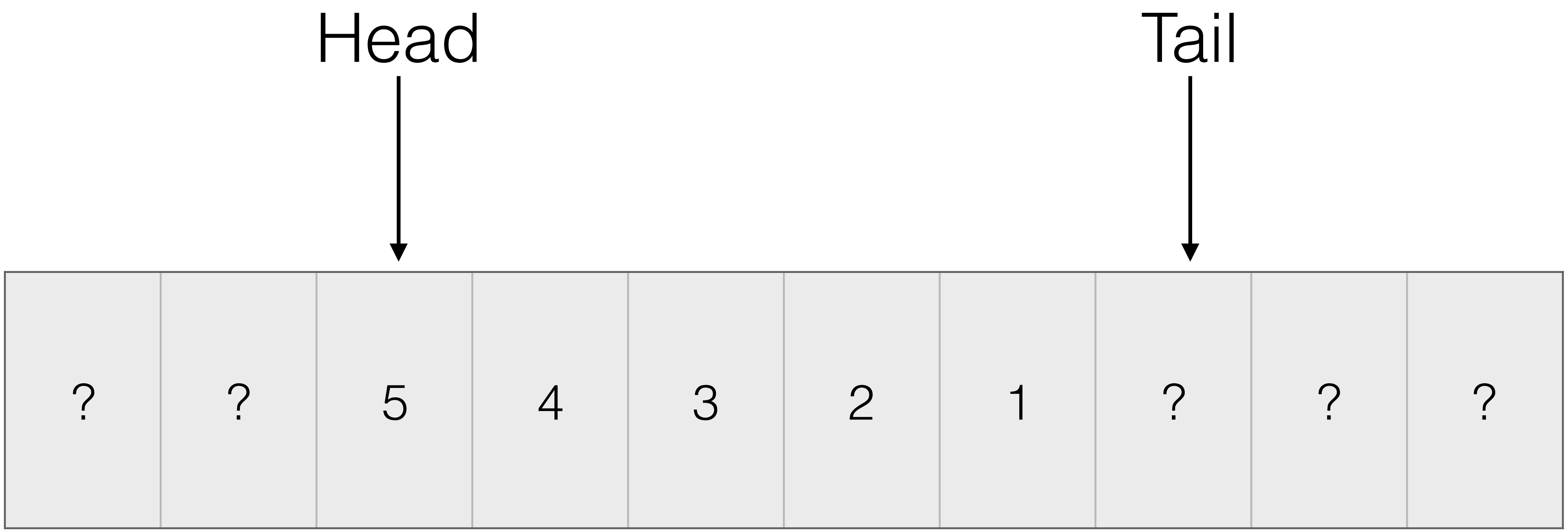
- `void enqueue(int p, T v) throws QueueFull`: put v on the queue with priority p

- `T getMax() throws QueueEmpty`: remove the element with the highest priority, if any

- `bool isEmpty()`: check if the queue is empty

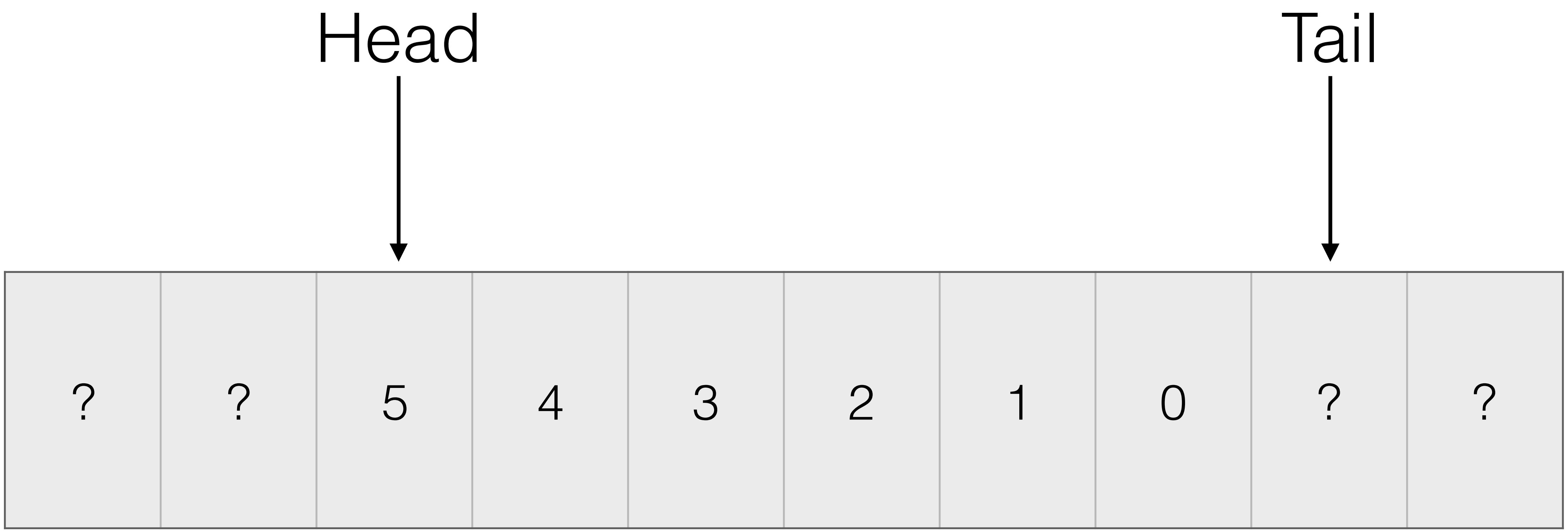
Queue Implementation

An array, plus two variables to track the head and tail



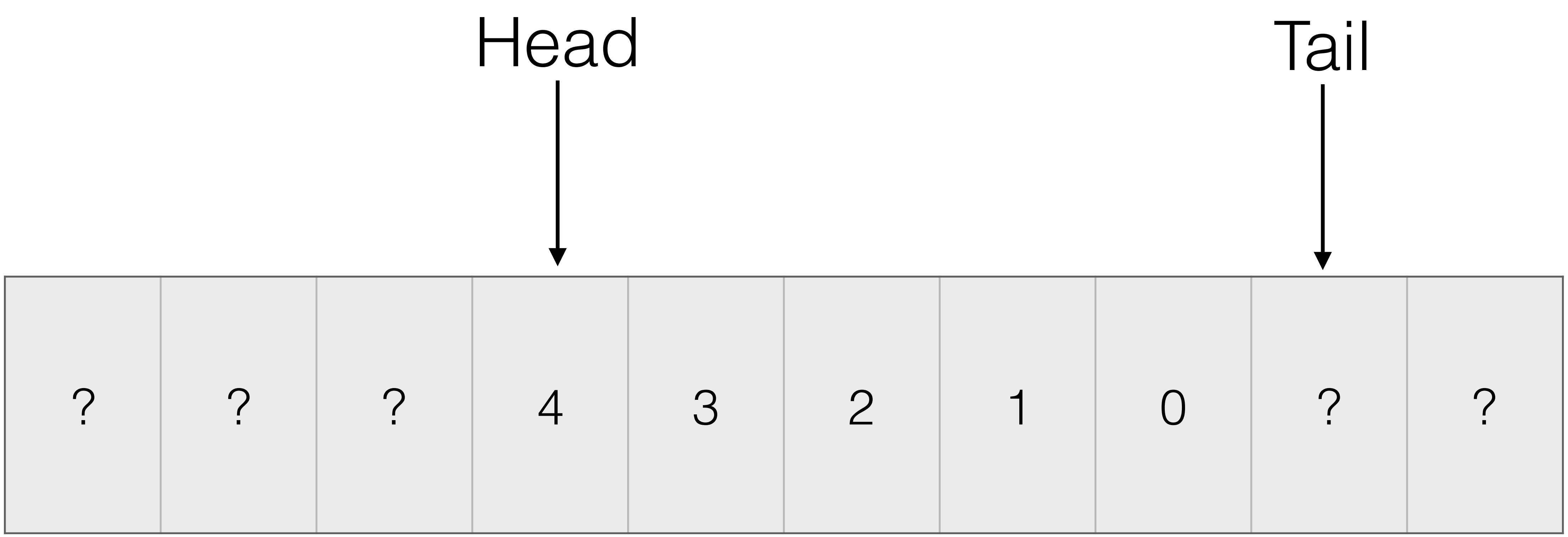
Queue Implementation

Add at the tail



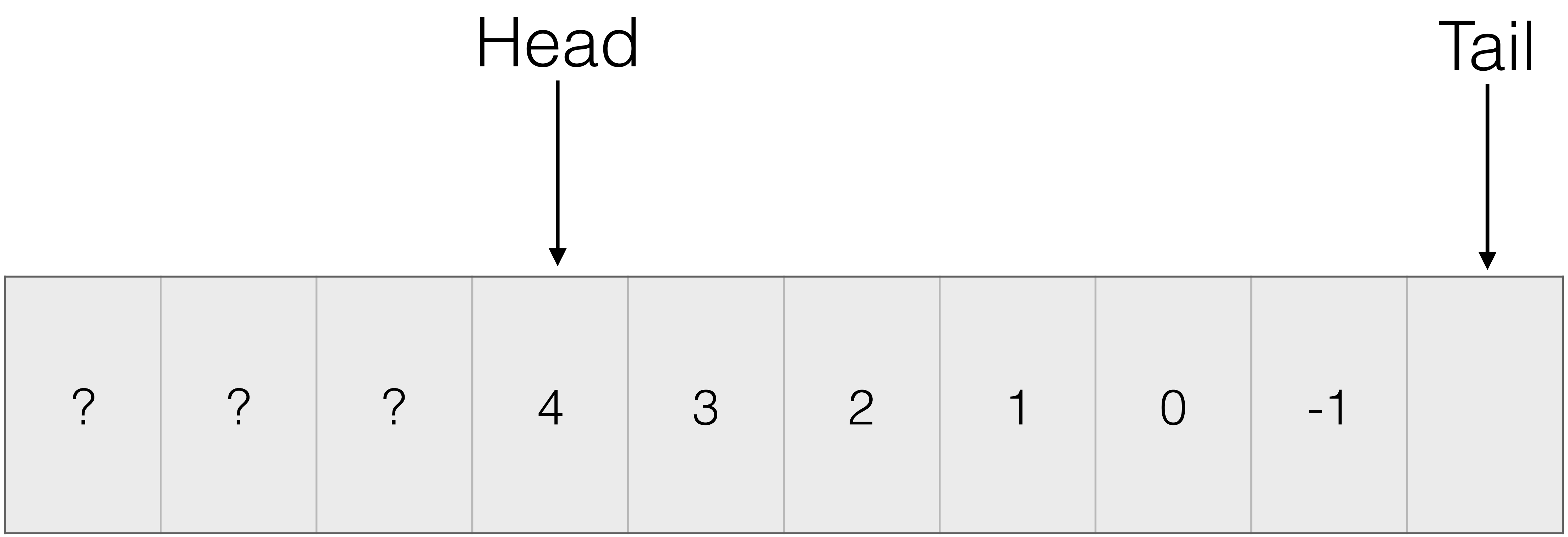
Queue Implementation

Remove from the head



Queue Implementation

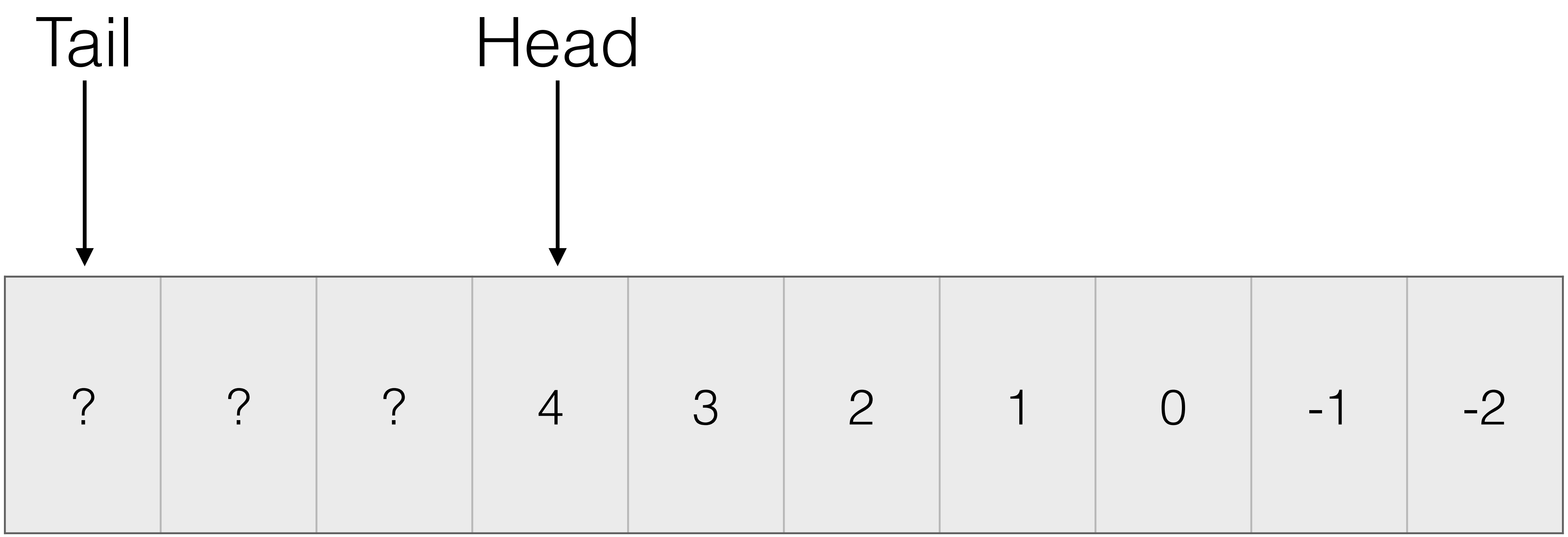
Special case 1: Tail hits the end



Queue Implementation

Special case 1: Tail hits the end

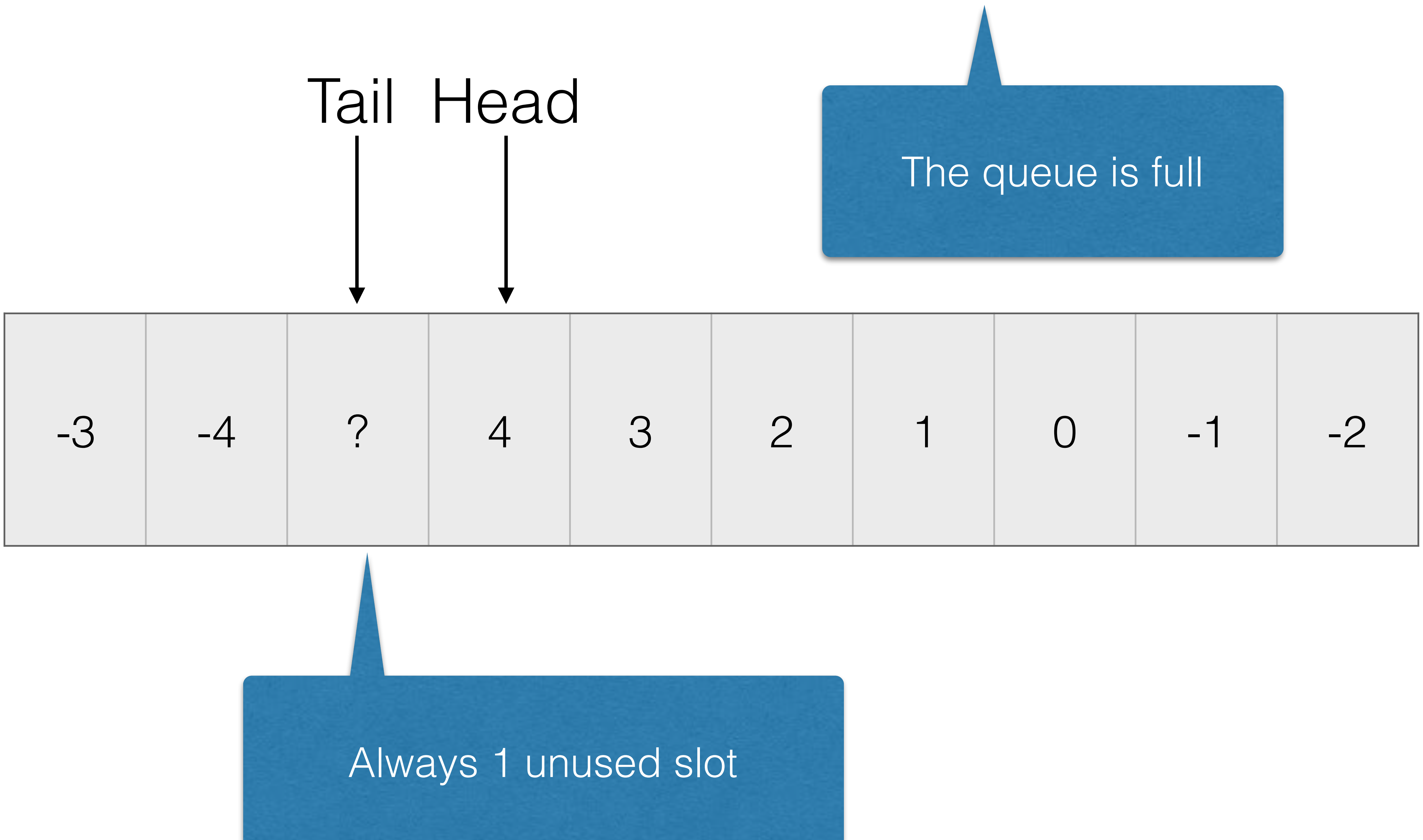
Wrap around



Also wrap if head hits the end

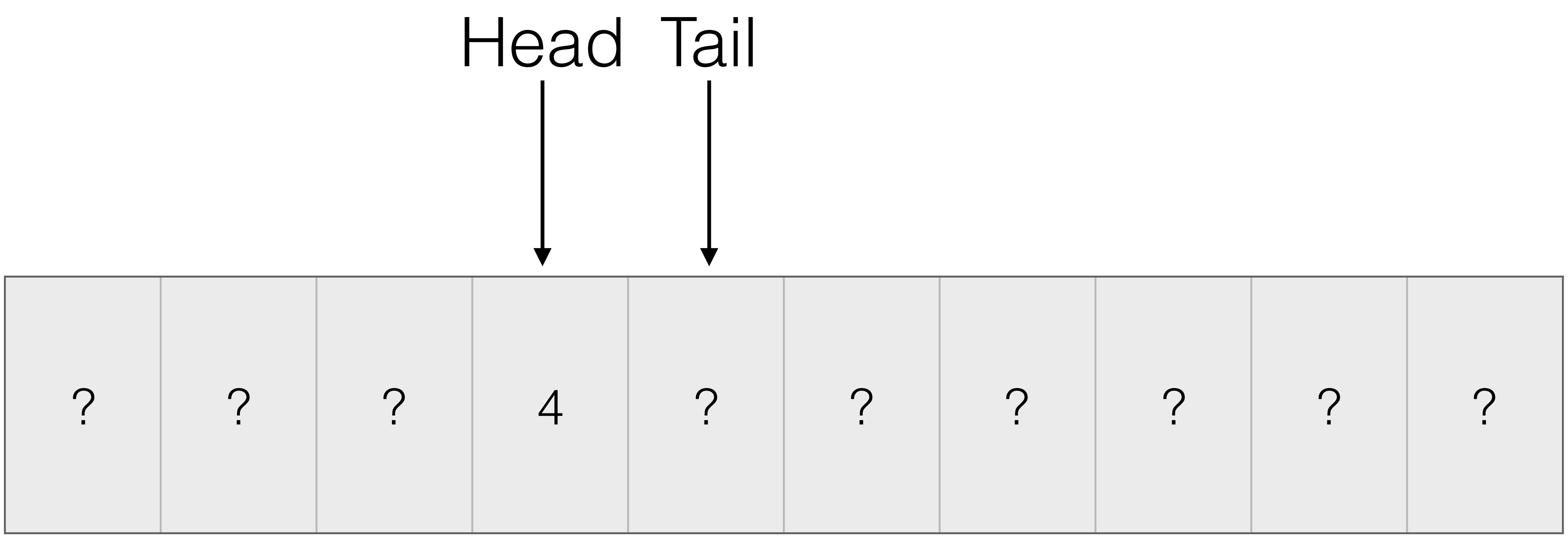
Queue Implementation

Special case 2: Tail reaches head



Queue Implementation

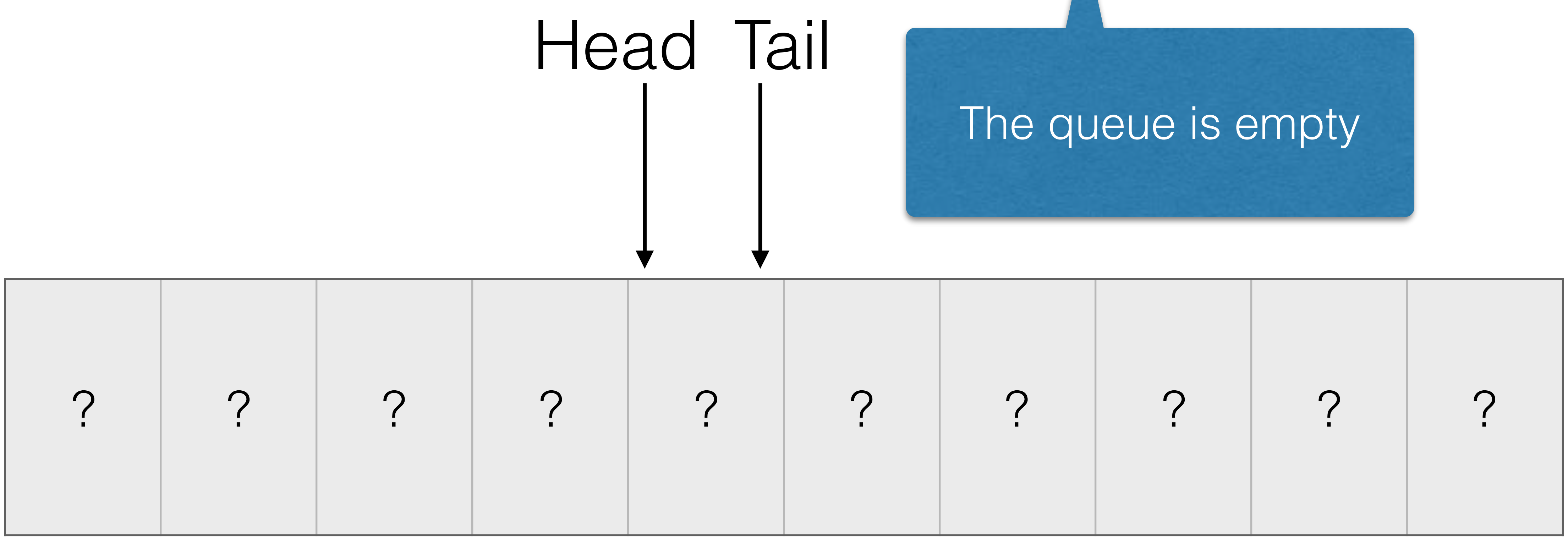
Special case 3: Head reaches tail



Queue Implementation

Special case 3: Head reaches tail

The queue is empty



Queue Summary

- Think of the array as circular (also called a *ring buffer*)
- Keep track of the head and tail as they follow each other around the buffer
- If they are on the same slot, the queue is empty
- If tail is one less than head, the queue is full

Dictionary Implementation

- If keys only support `.equals` then
  - `lookup` must use *linear search:* check all elements in order until desired key found
  - `add` just add to the growable array: amortised constant time
- If keys support `.compareTo` then keep the array in sorted order
  - `lookup` can use *binary search*
  - `add` must keep array sorted, uses linear amount of copying

Binary Search

Showing just the keys

2	5	16	20	25	70	101	130	145	146	180	200	210	222	345

index 0 7 14

Start looking for *k* between 0 and 14

1. Check at (0+14)/2 = 7
2. If *k* = array[7], finished
   if *k* < array[7], then check between 0 and 6
   if k > array[7], then check between 8 and 14

Binary Search

Showing just the keys

2	5	16	20	25	70	101	130	145	146	180	200	210	222	345

index 0 7 14

Start looking for *k* between *start* and *stop*

1. If *start == stop*, then just check that
2. Check at (*start*+*stop*)/2 = *middle*
3. If *k* = array[*middle*], finished
   if *k* < array[*middle*], then check *start* to *middle-1*
   if k > array[*middle*], then check *middle+1* to *stop*

Binary Search

Showing just the keys

	2	5	16	20	25	70	101	130	145	146	180	200	210	222	345
index	0							7							14

To find 25:
0–14, middle = 7, then
0–6, middle = 3, then
4–6, middle = 5, then
4–4, check that array[4] is 25

To find 23:
0–14, middle = 7, then
0–6, middle = 3, then
4–7, middle = 5, then
4–4, check that array[4] is 23

Binary Search Complexity

Divide and conquer: Each step halves the size of the problem

Array length	1	2	4	8	16	32	64	128	256	2^n	n
Max # steps	1	2	3	4	5	6	7	8	9	n	$\log_2(n)$

Logarithmic: If the size of the input doubles, the number of steps increased by a constant amount

Sorting

- *Sorting*: given an array of `Comparable` keys, put them into ascending (or descending) order
- We will look at *insertion sort, merge sort,* and *quick sort.* Of these, merge sort and quick sort are practical.
- Consider efficiency on random data, and on almost sorted data.
- *Stability*: Does the sort keep the ordering of keys that appear multiple times?

Insertion Sort

- Similar to adding to the sorted dictionary
- Repeatedly insert the next unsorted element into the sorted array

Insertion Sort Example

Showing just the keys

0				4	5									
2	5	16	20	25	1	1	2	145	27	12	122	500	499	345

Sorted here

Unsorted here

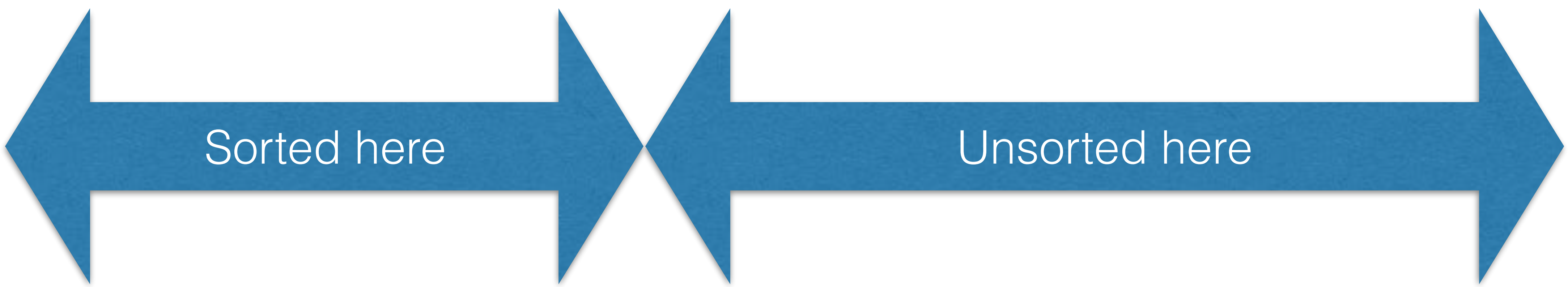
0					5									
1	2	5	16	20	25	1	2	145	27	12	122	500	499	345

Sorted here

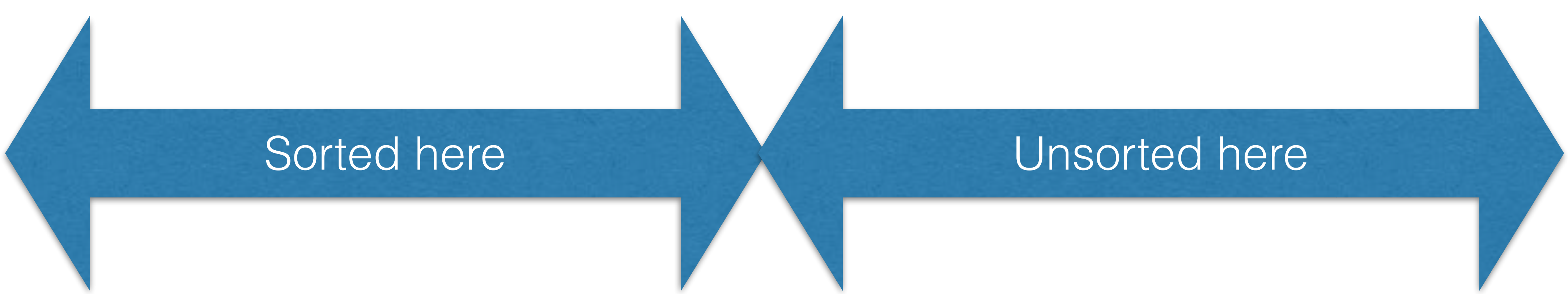
Unsorted here

Insertion Sort Example

1	2	5	16	20	25	1	2	145	27	12	122	500	499	345

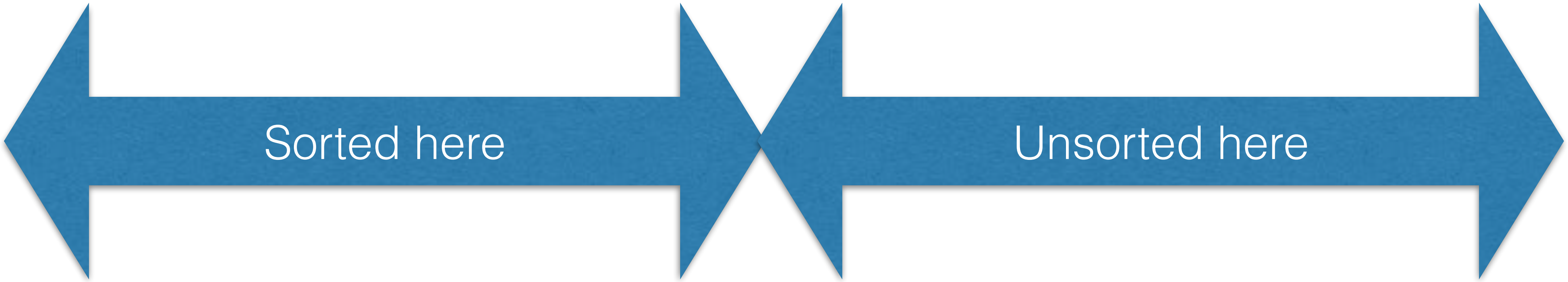


1	1	2	5	16	20	25	2	145	27	12	122	500	499	345

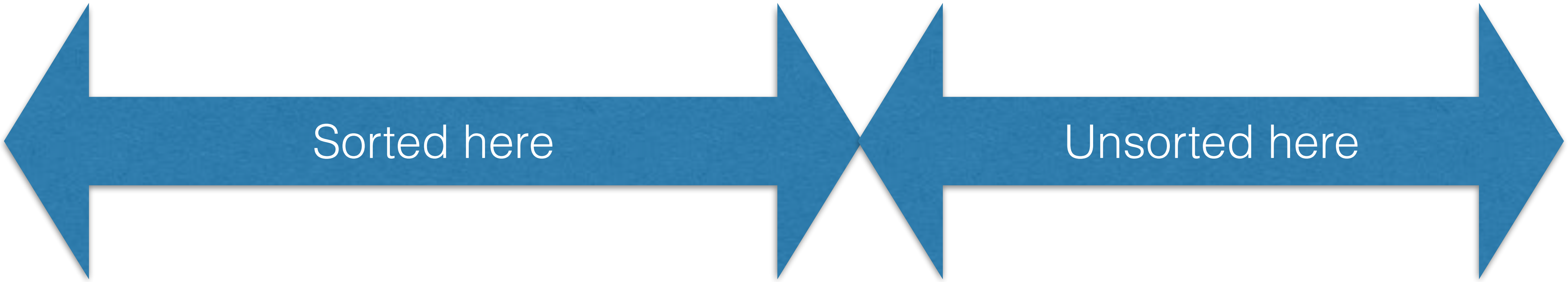


Insertion Sort Example

1	1	2	5	16	20	25	2	145	27	12	122	500	499	345



1	1	2	2	5	16	20	25	145	27	12	122	500	499	345



Insertion Sort Example

1	1	2	2	5	16	20	25	145	27	12	122	500	499	345

Sorted here

Unsorted here

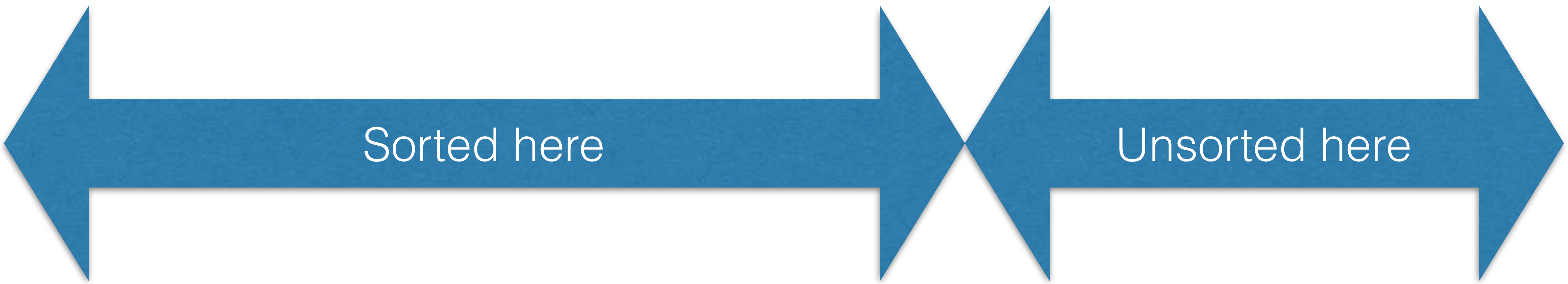
1	1	2	2	5	16	20	25	145	27	12	122	500	499	345

Sorted here

Unsorted here

Insertion Sort Example

1	1	2	2	5	16	20	25	145	27	12	122	500	499	345



1	1	2	2	5	16	20	25	27	145	12	122	500	499	345

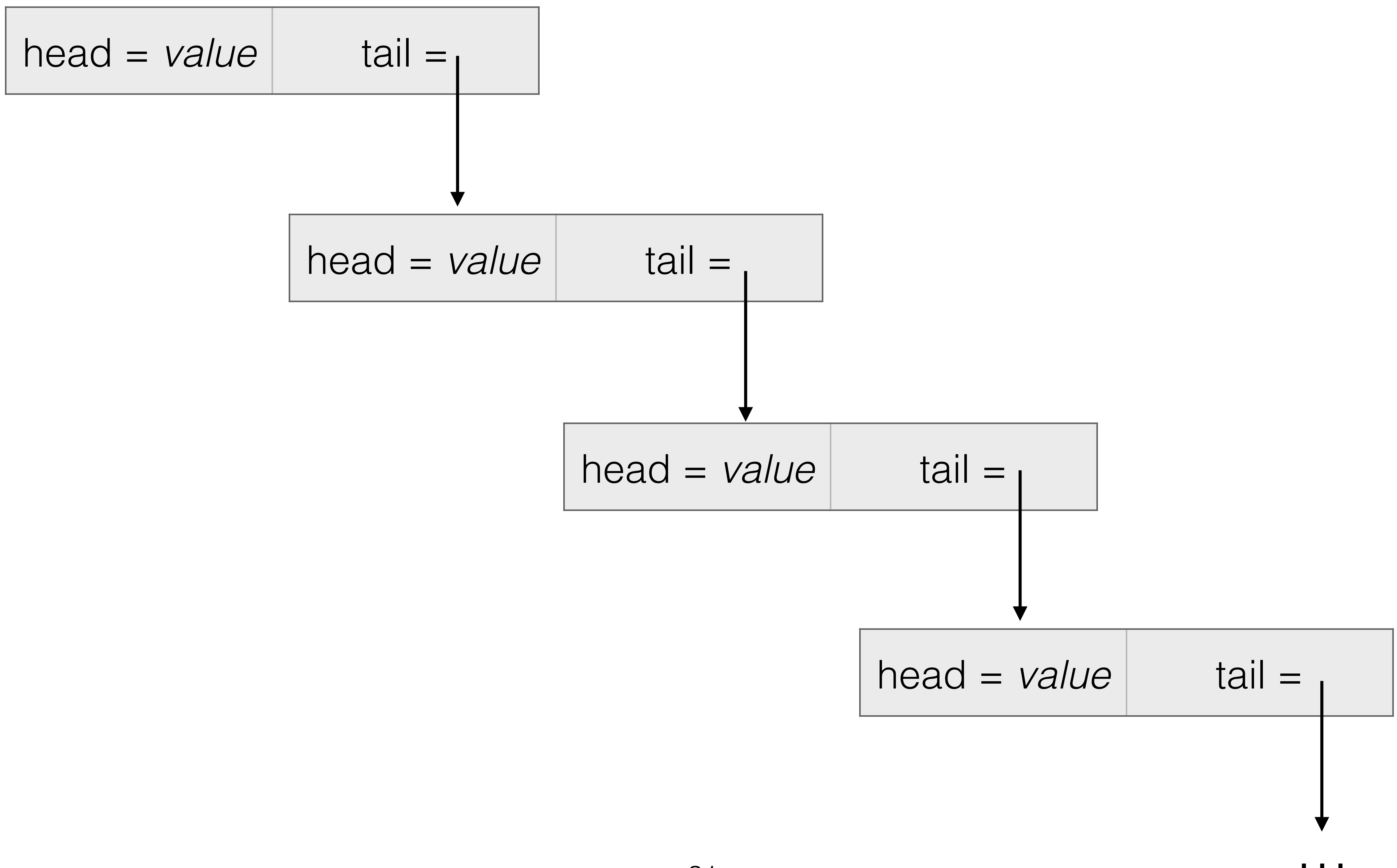
Insertion Sort Complexity

- For each element, insert it into an array quadratic in the worst case (which is that the array is sorted in reverse order)
- If the array is already sorted, linear (if we're careful)
- Stable (if we're careful)

Linked Lists

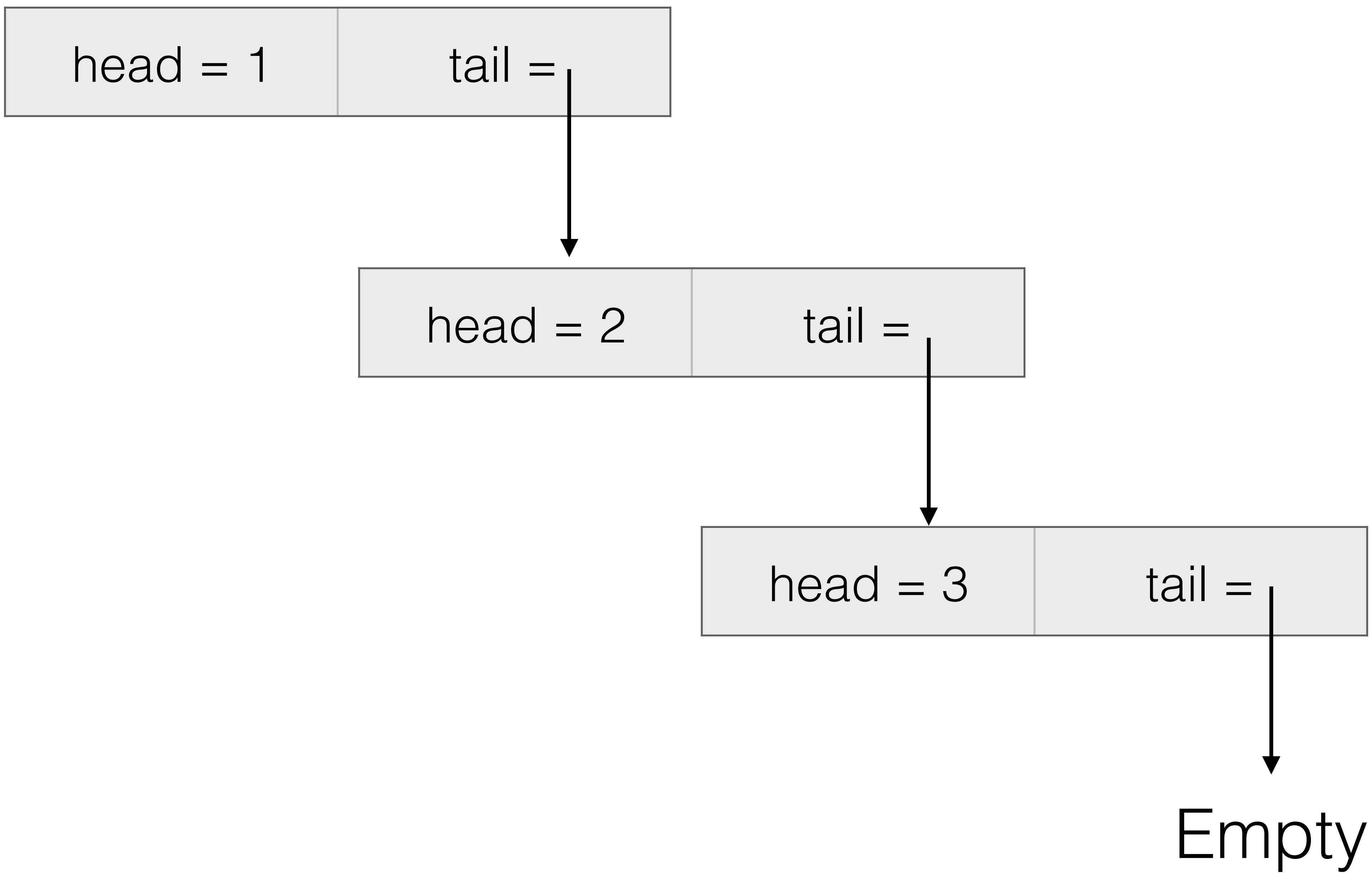
- There's more to life than just arrays
- Growable arrays can be tedious to use
    - Bad for concurrency to lock the whole array
- In a *linked list* each element is dynamically allocated
- But, no constant time direct access to elements

Linked List Idea



Linked List Example

A list of 1, 2, 3 would be



Linked List Primitive Operations

- Allocate a new list element, to lengthen the list
- Look at the value in the front element
- Look at the tail of the list

Linked List Implementation

Object Oriented Style

See **OOLinkedList.java** on the Moodle page

```
class EndOfList extends Exception {};

abstract class LinkedList<T> {
}

class Empty<T> extends LinkedList<T> {
}

class Node<T> extends LinkedList<T> {
  public T head;
  public LinkedList<T> tail;
  public Node(T h, LinkedList<T> t) {
    head = h;
    tail = t;
  }
}
```

Some Methods on Lists

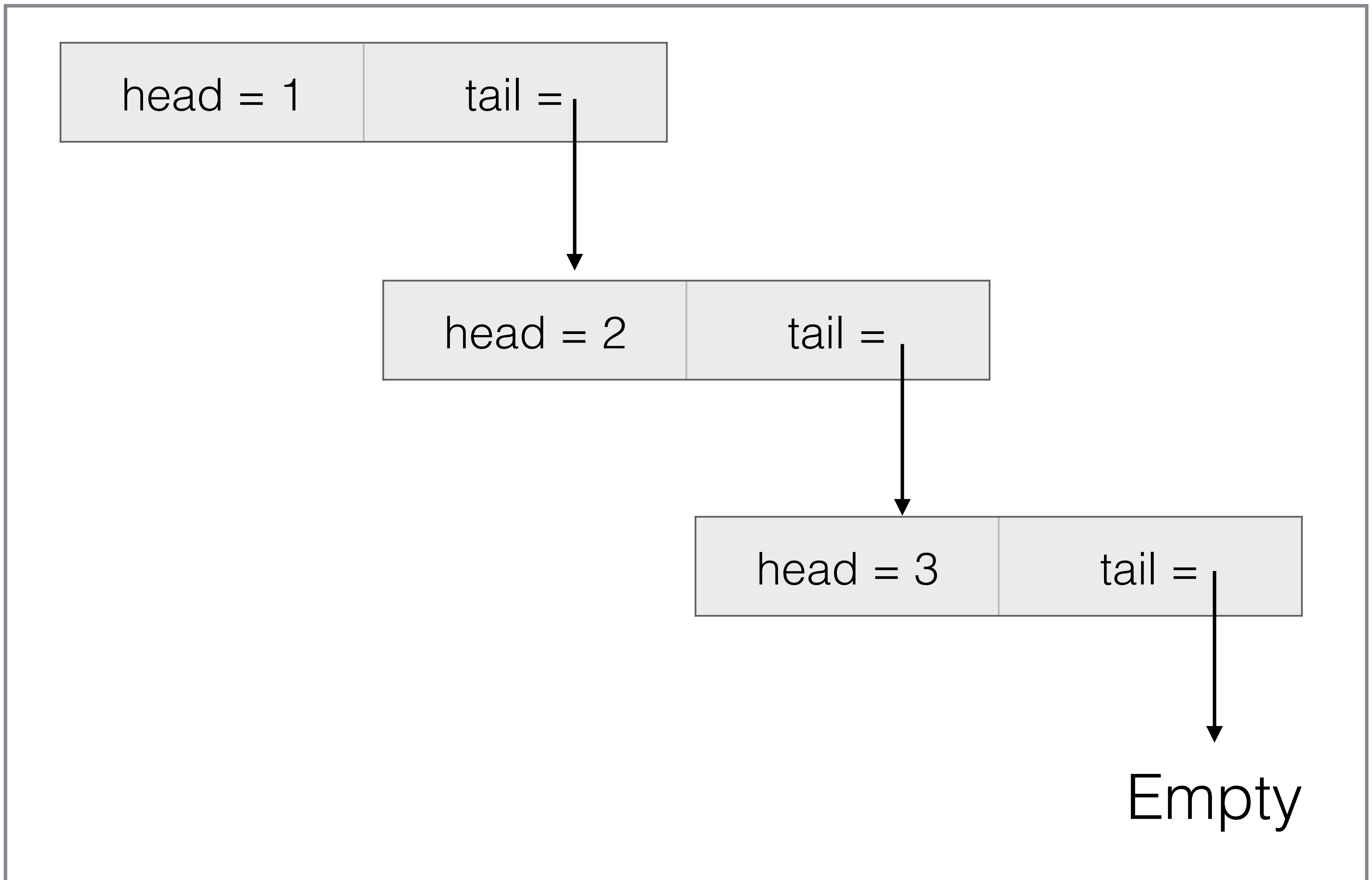
The **length** method is *recursive*

```
public int length() {
  // The length of the list is just one more
  // than the length of the tail
  return 1 + tail.length();
}
```

This call to **length** is said to be *recursive.*
tail is a LinkedList, and LinkedLists support **length** methods

Think about how to compute the length of this list in terms of the length of the tail. Similar thinking as to what the body of a for loop should do each time through

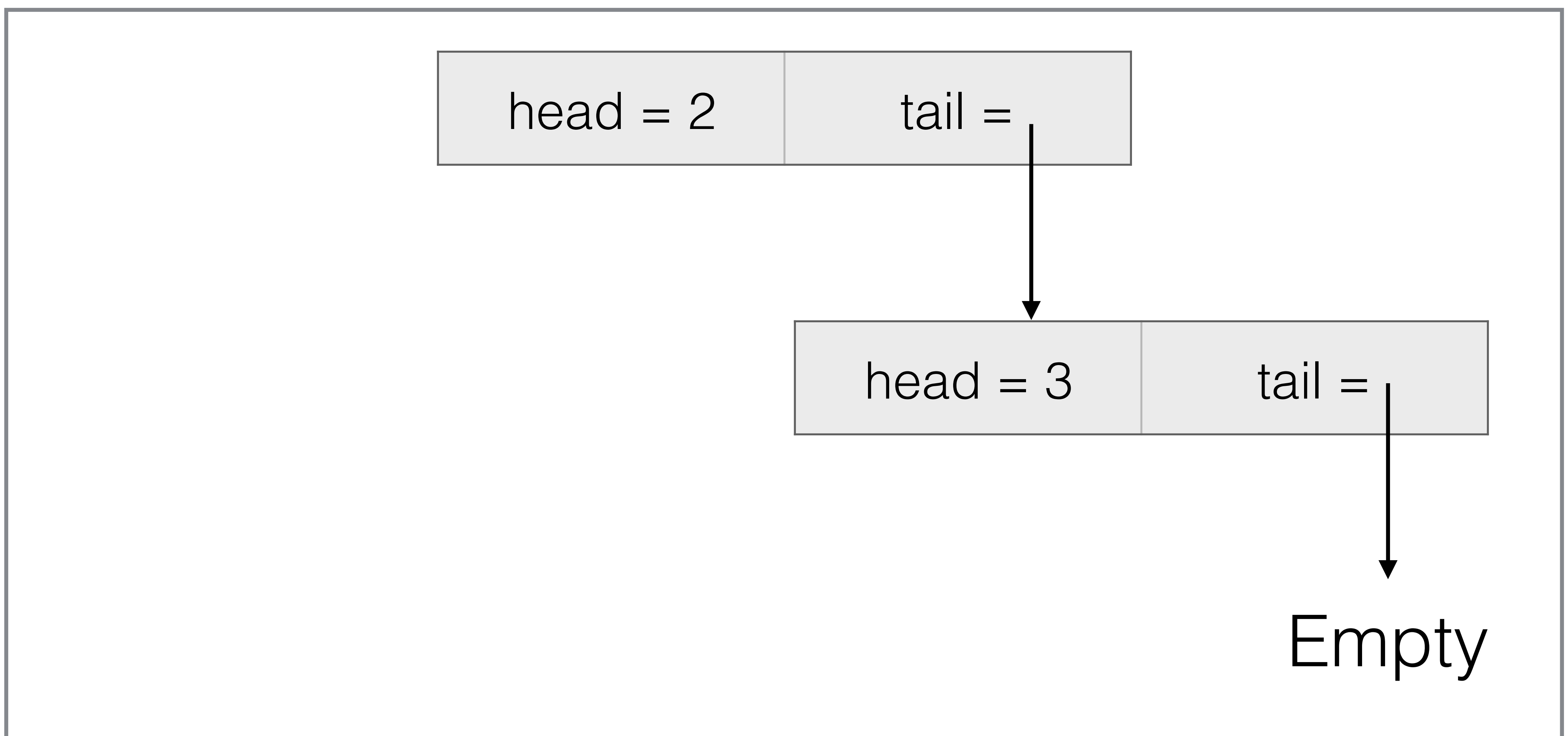
Length Example



.length()

Length Example

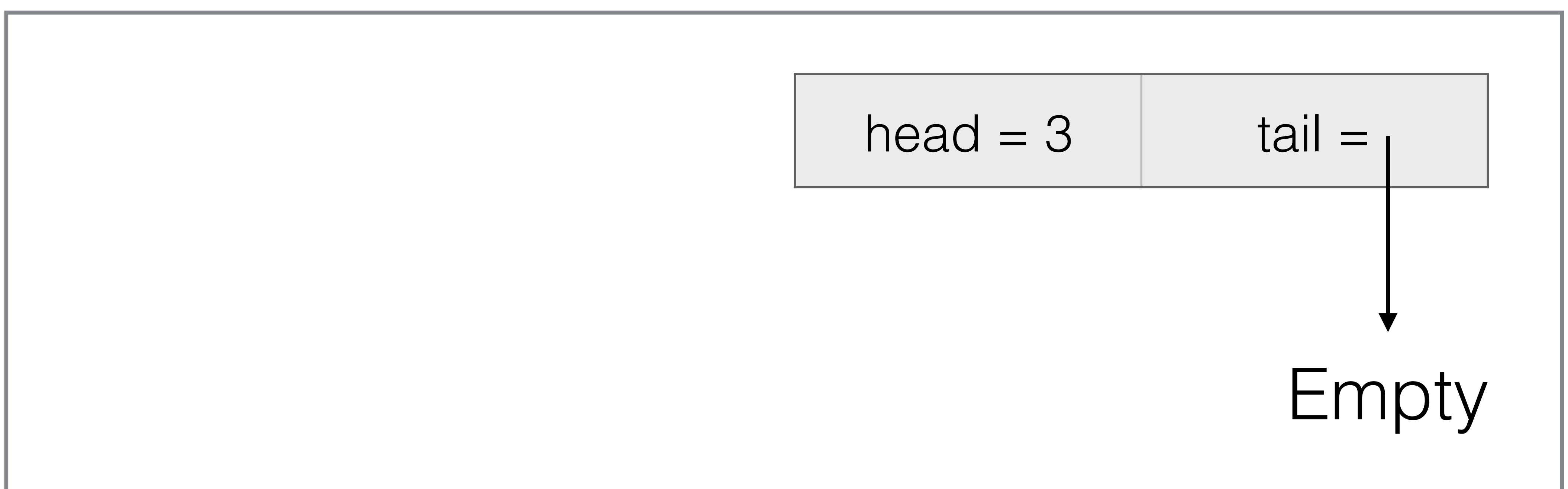
1 +



.length()

Length Example

1 + 1 +



.length()

Length Example

1 + 1 + 1 +

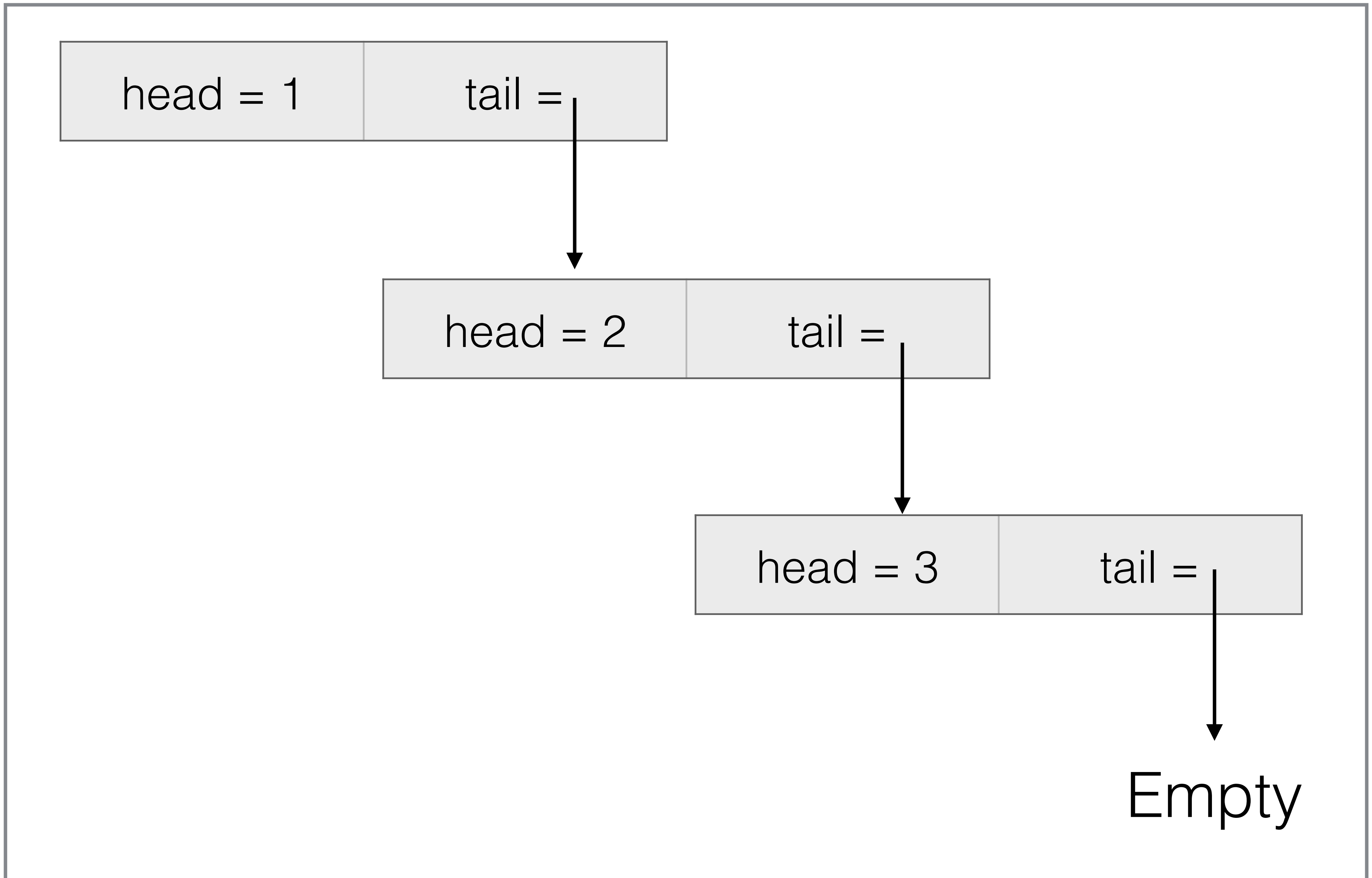
Empty

.length()

Length Example

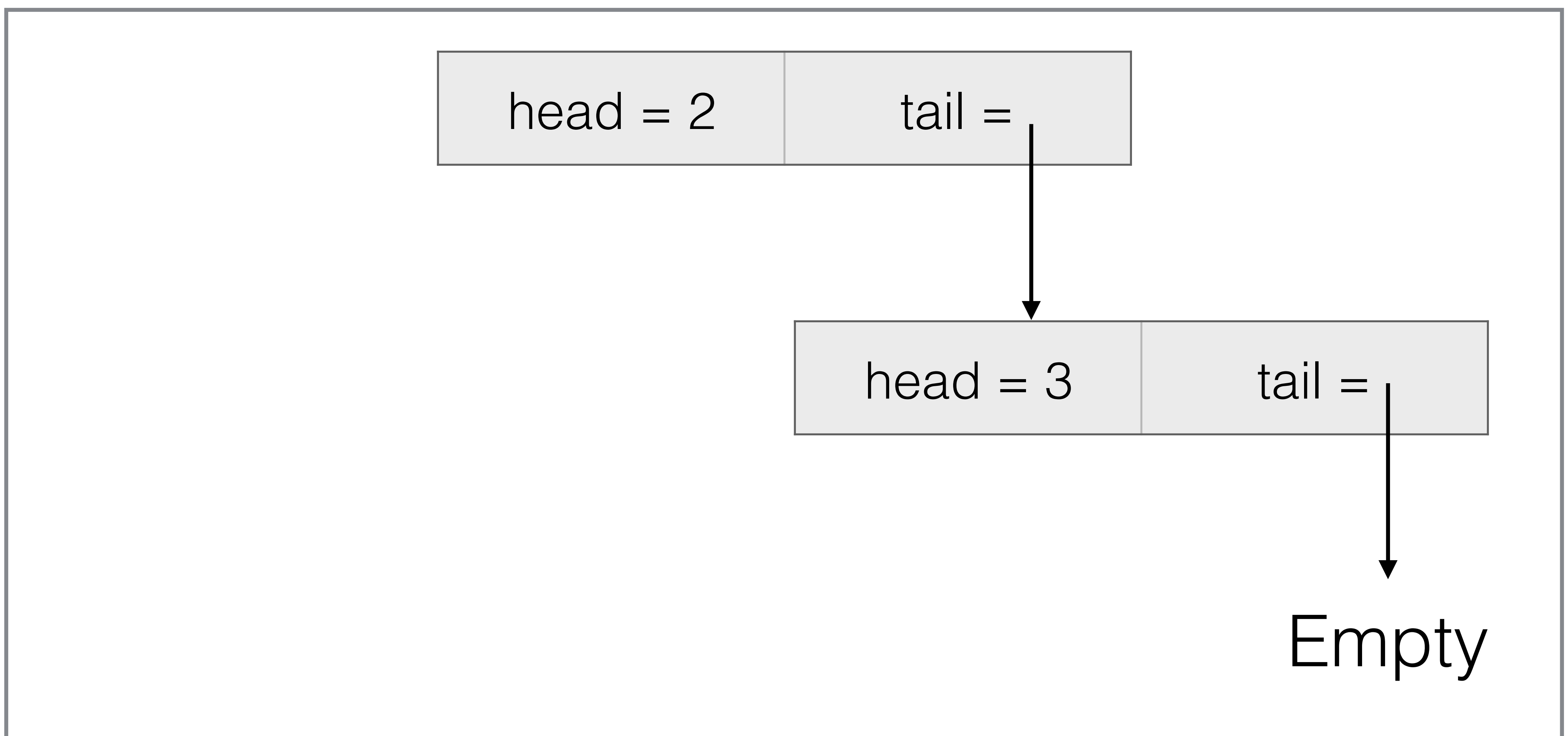
$1 + 1 + 1 + 0$

Get Example



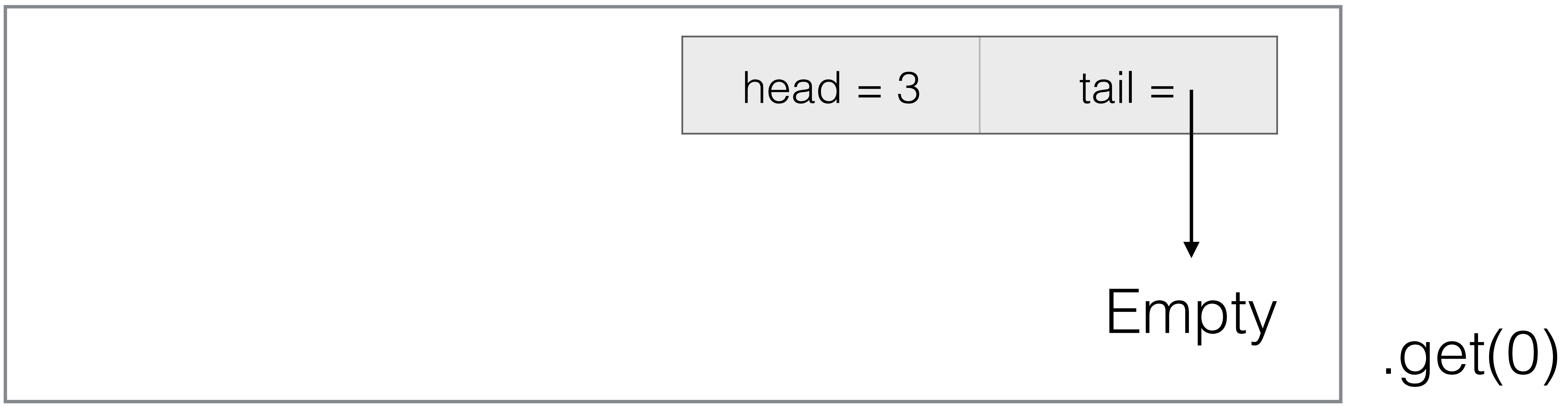
.get(2)

Get Example



.get(1)

Get Example



Get Example

3

Linked List Implementation

Procedural Style

See **LinkedList.java** on the Moodle page

```
class EndOfList extends Exception {};

class Node<T> {
  public T head;
  public Node<T> tail;
  public Node(T h, Node<T> t) {
    head = h;
    tail = t;
  }
}
```

Some Methods on Lists

The **length** method is *recursive*

```
public static <T> int length(Node<T> n) {
  if (n == null)
    return 0;
  else
    // The length of the list is just one more
    // than the length of the tail
    return 1 + length(n.tail);
}
```

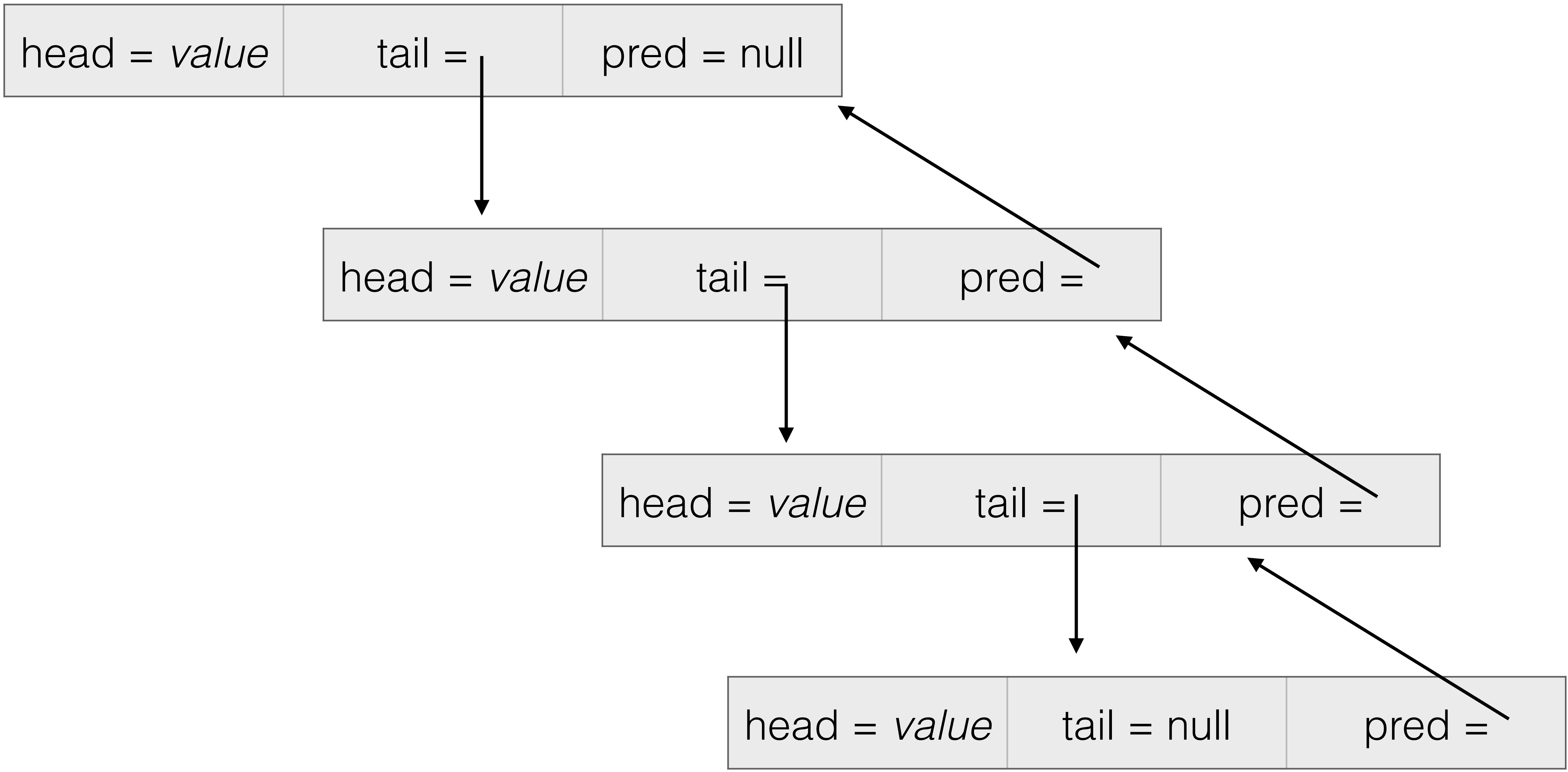
Note that the if detects the empty class, rather than letting the objects do that for us.

Stacks from Linked Lists

- Keep a pointer to the Node at the top of the stack
- **push** allocates a new node
- **pop** follows the tail pointer once
- See **LinkedStack.java**

Doubly Linked Lists

Only procedural style

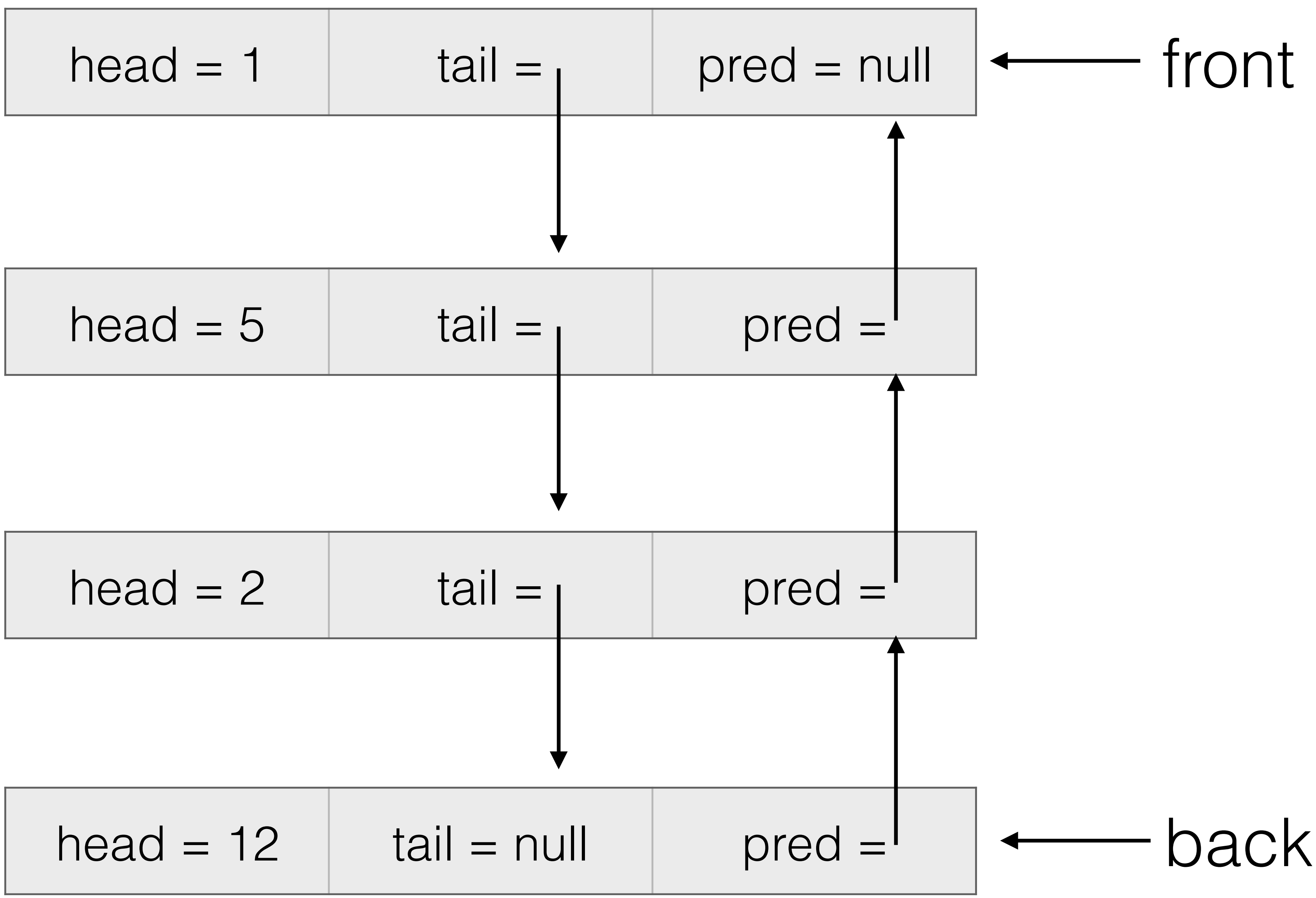


See **DoubleLL.java**

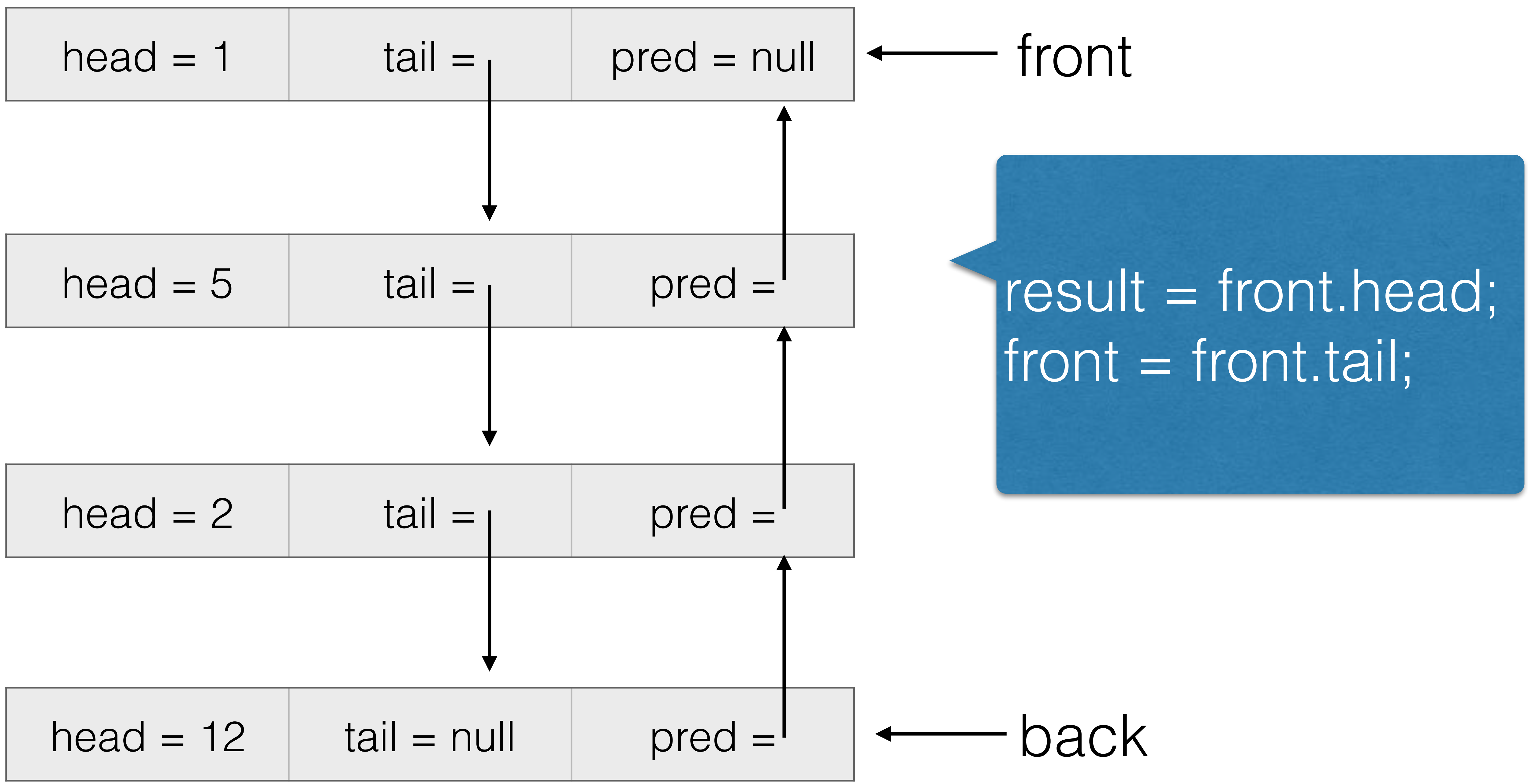
Queues from Doubly Linked Lists

- Keep a pointer to each end of the queue, front and back
- **enqueue** by adding to the back. Change the null **pred** of the previous back to the new back.
- **dequeue** by using the **pred** of the front to find the next oldest element
- See LinkedQueue.java
- Unbounded capacity, unlike the ring buffer queue

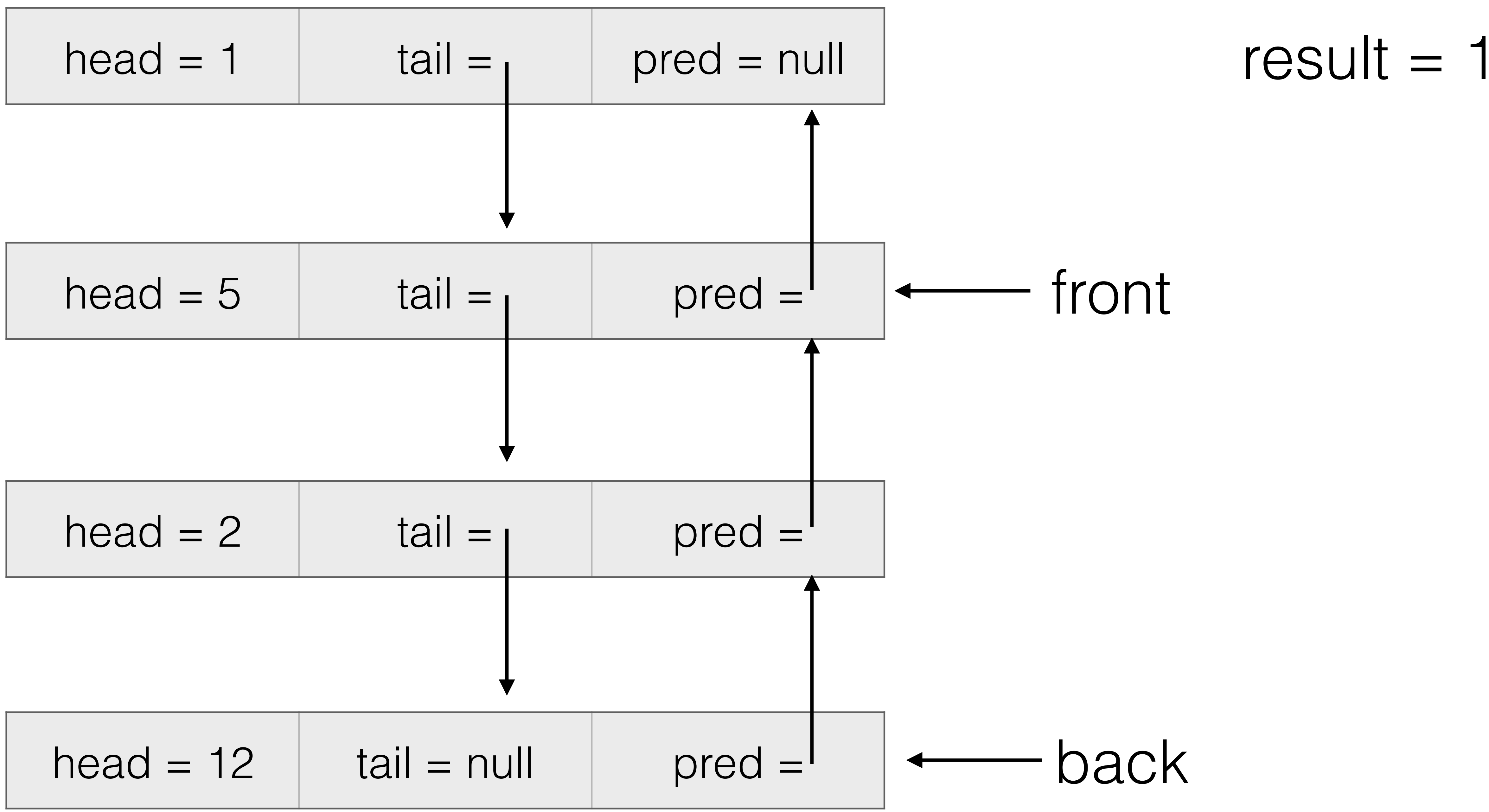
Dequeue


head = 1
tail =
pred = null
front
head = 5
tail =
pred =
head = 2
tail =
pred =
head = 12
tail = null
pred =
back

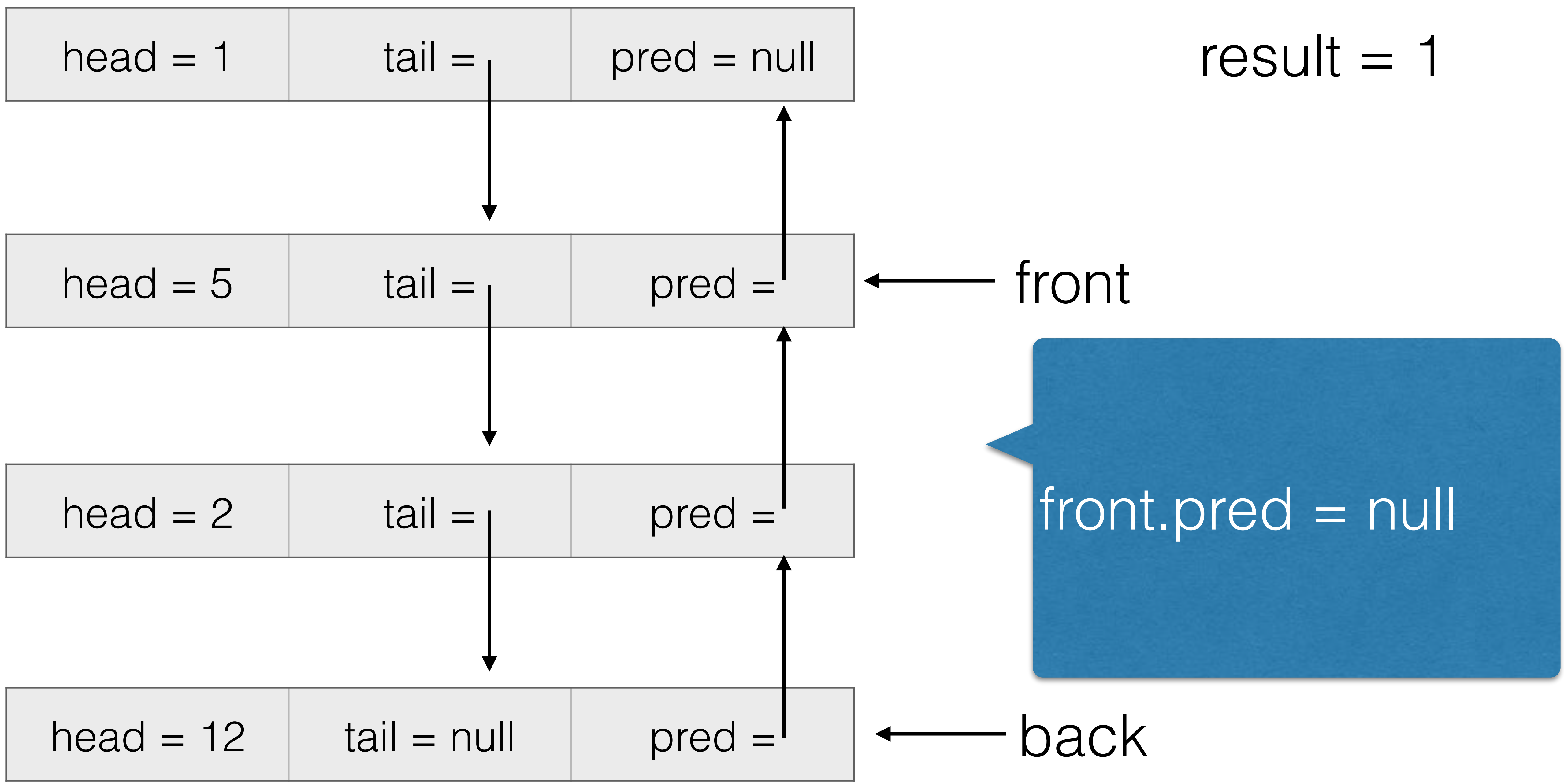
Dequeue


head = 1
tail =
pred = null
front
head = 5
tail =
pred =
result = front.head;
front = front.tail;
head = 2
tail =
pred =
head = 12
tail = null
pred =
back

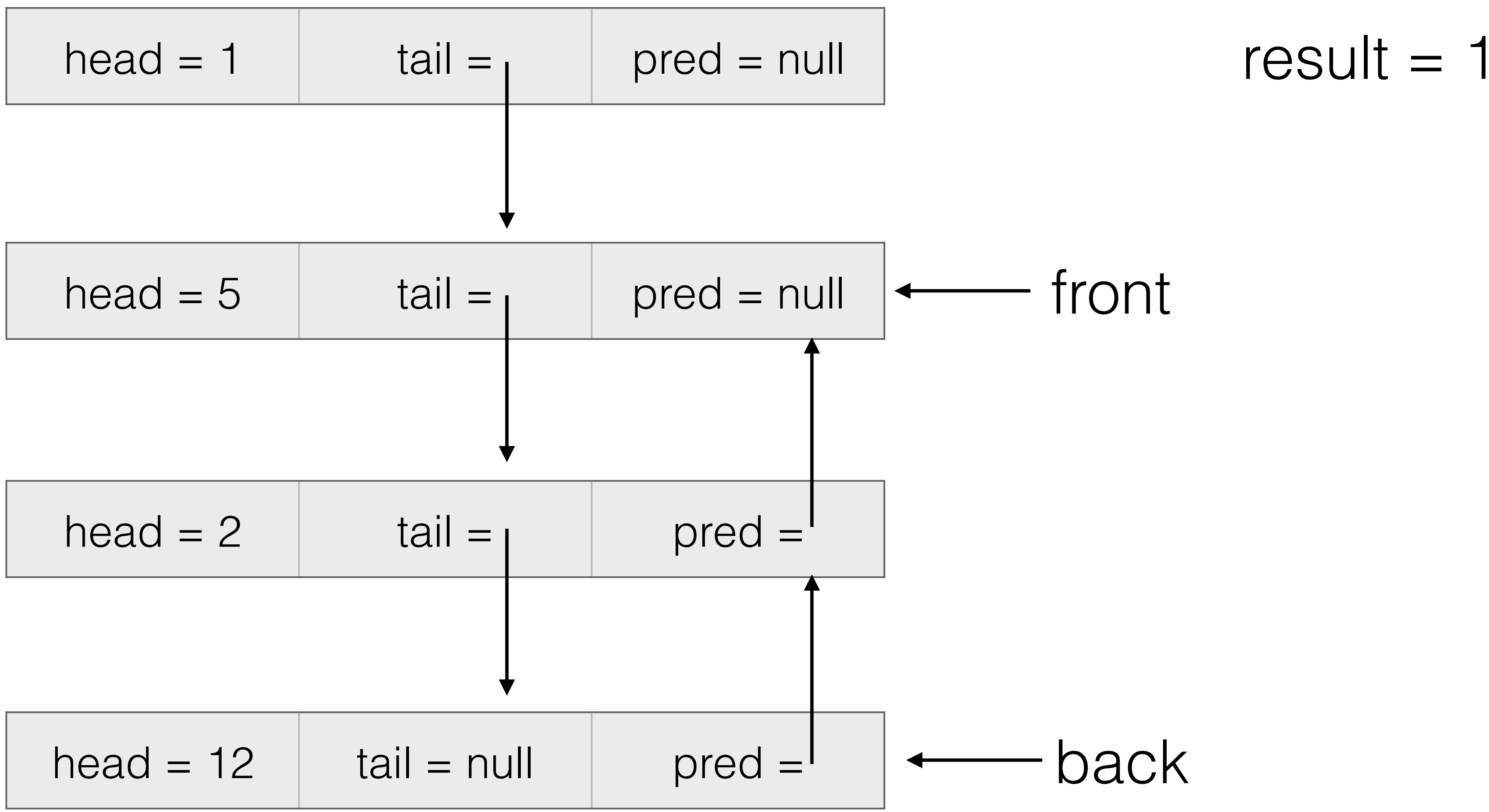
Dequeue


head = 1
tail =
pred = null
result = 1
head = 5
tail =
pred =
front
head = 2
tail =
pred =
head = 12
tail = null
pred =
back

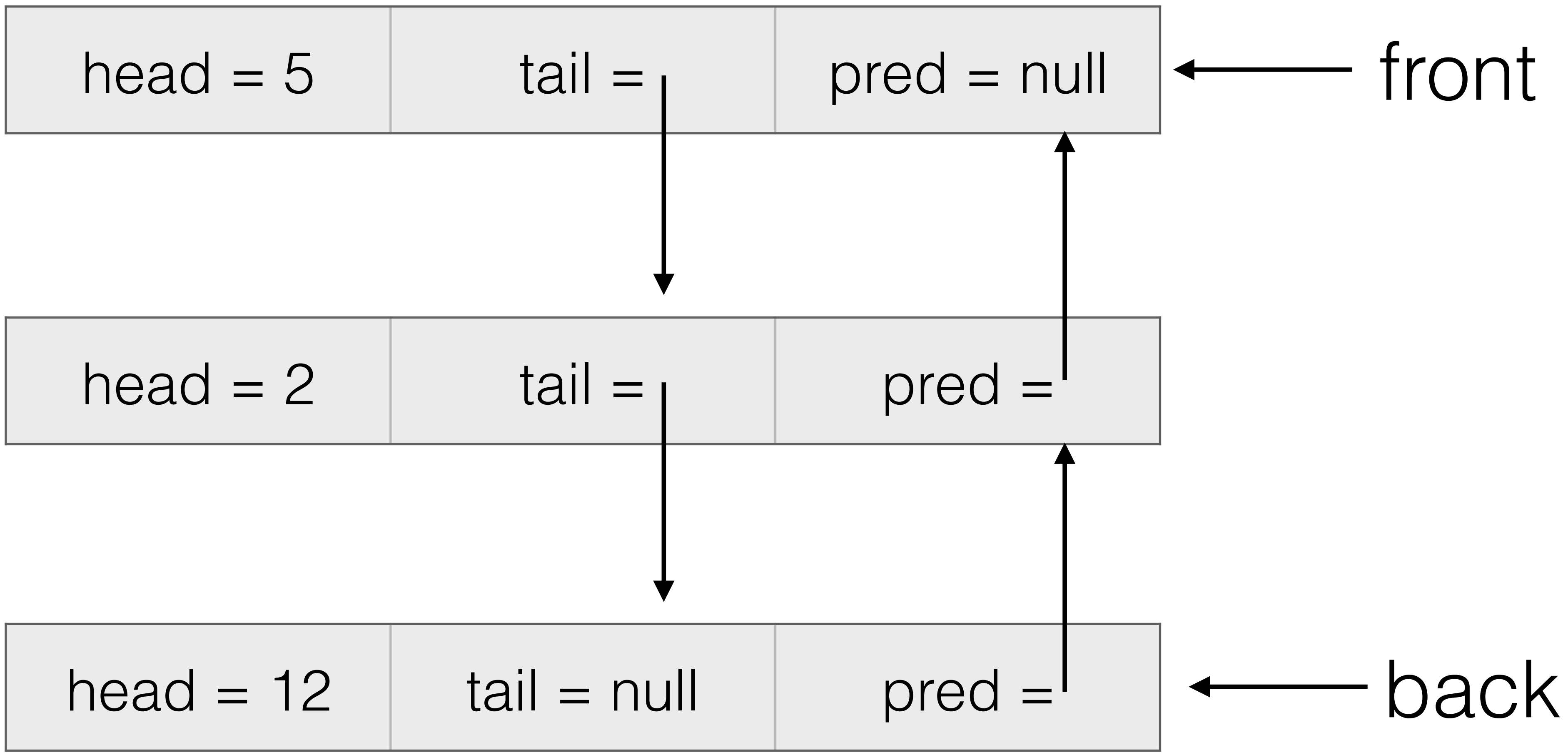
Dequeue


head = 1
tail =
pred = null
result = 1
head = 5
tail =
pred =
front
front.pred = null
head = 2
tail =
pred =
head = 12
tail = null
pred =
back

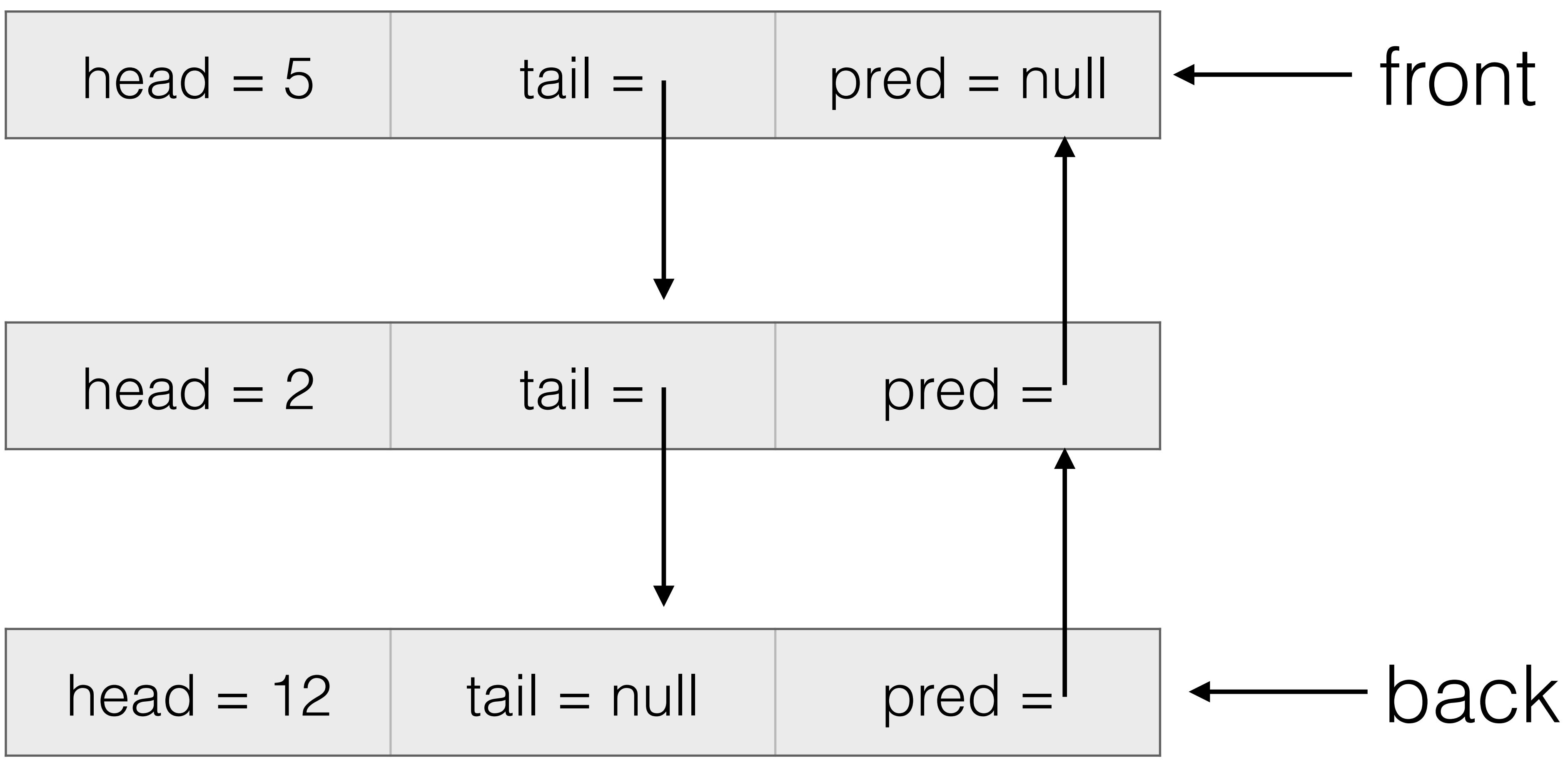
Dequeue


head = 1
tail =
pred = null
head = 5
tail =
pred = null
head = 2
tail =
pred =
head = 12
tail = null
pred =
front
back
result = 1

Enqueue

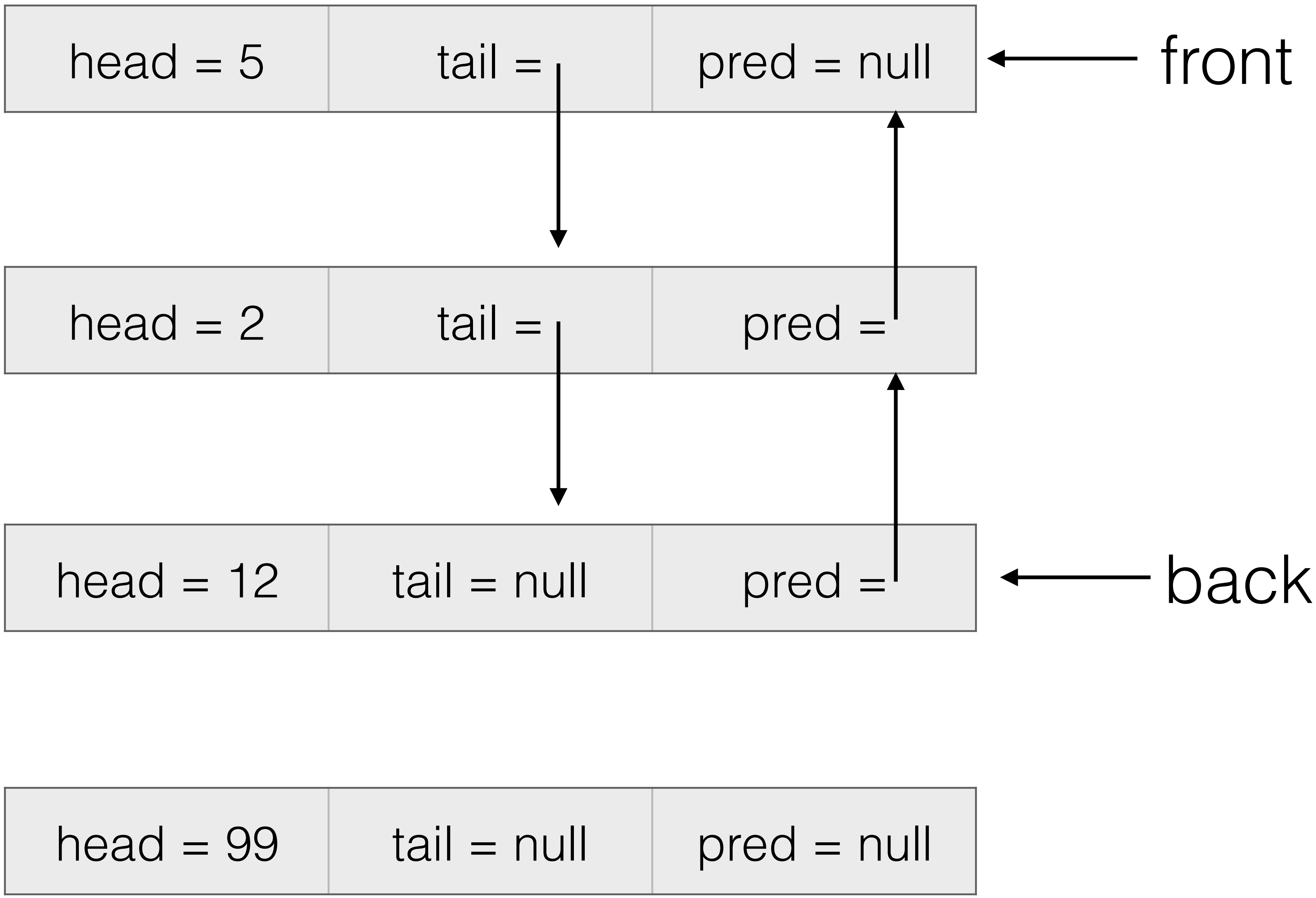

head = 5
tail =
pred = null
front
head = 2
tail =
pred =
head = 12
tail = null
pred =
back

Enqueue

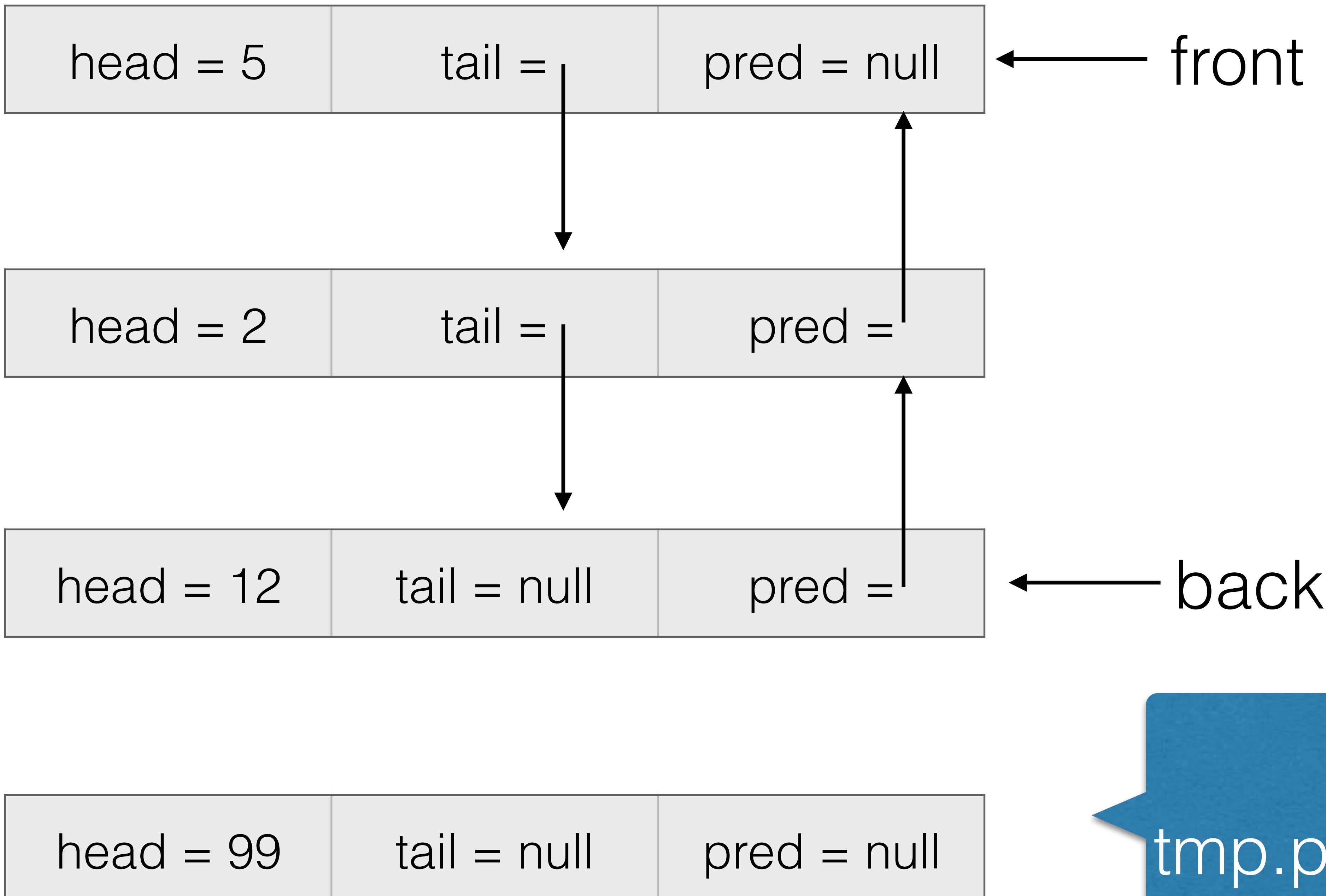


tmp = new Node(…)

Enqueue

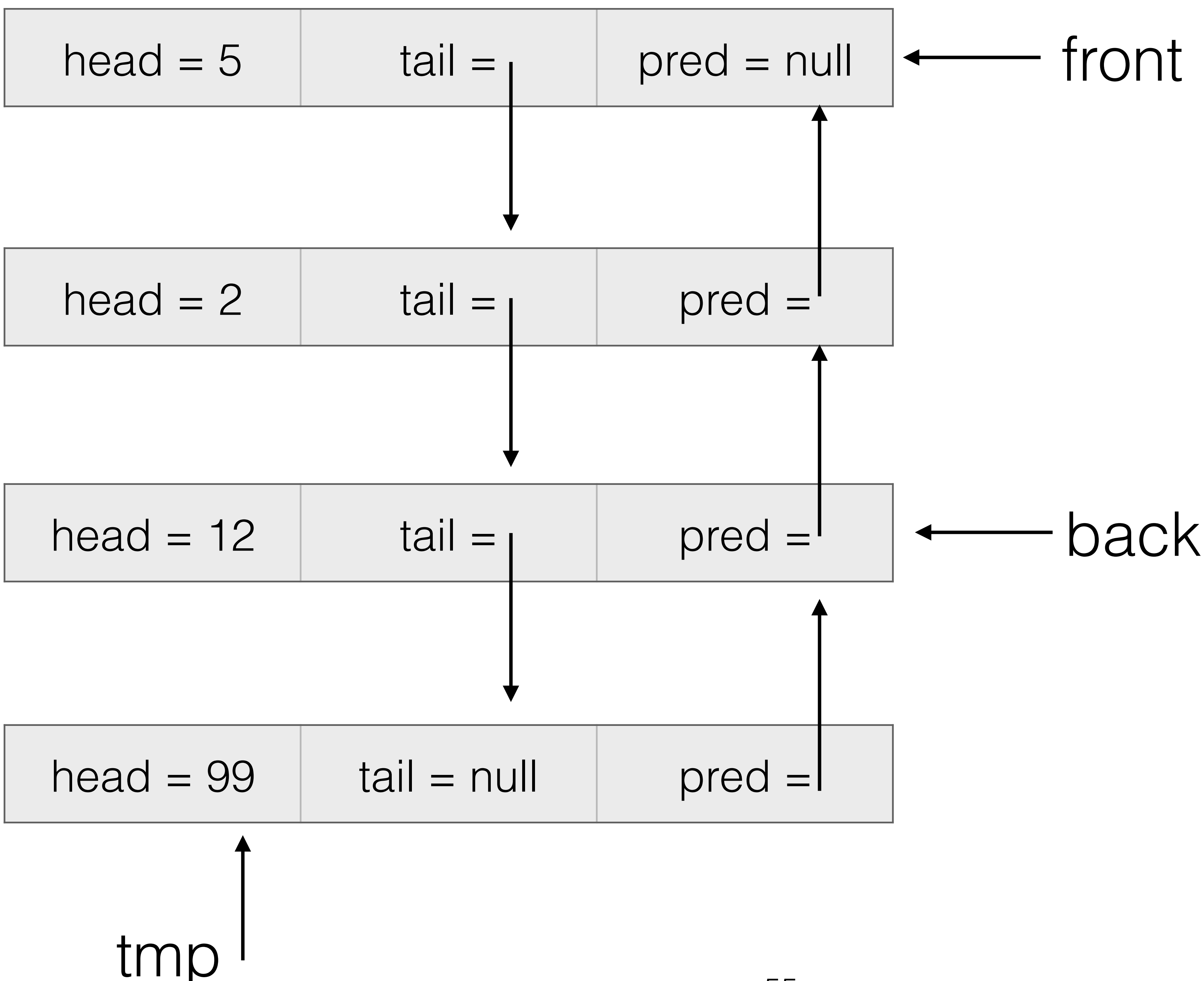

head = 5
tail =
pred = null
front
head = 2
tail =
pred =
head = 12
tail = null
pred =
back
head = 99
tail = null
pred = null

Enqueue

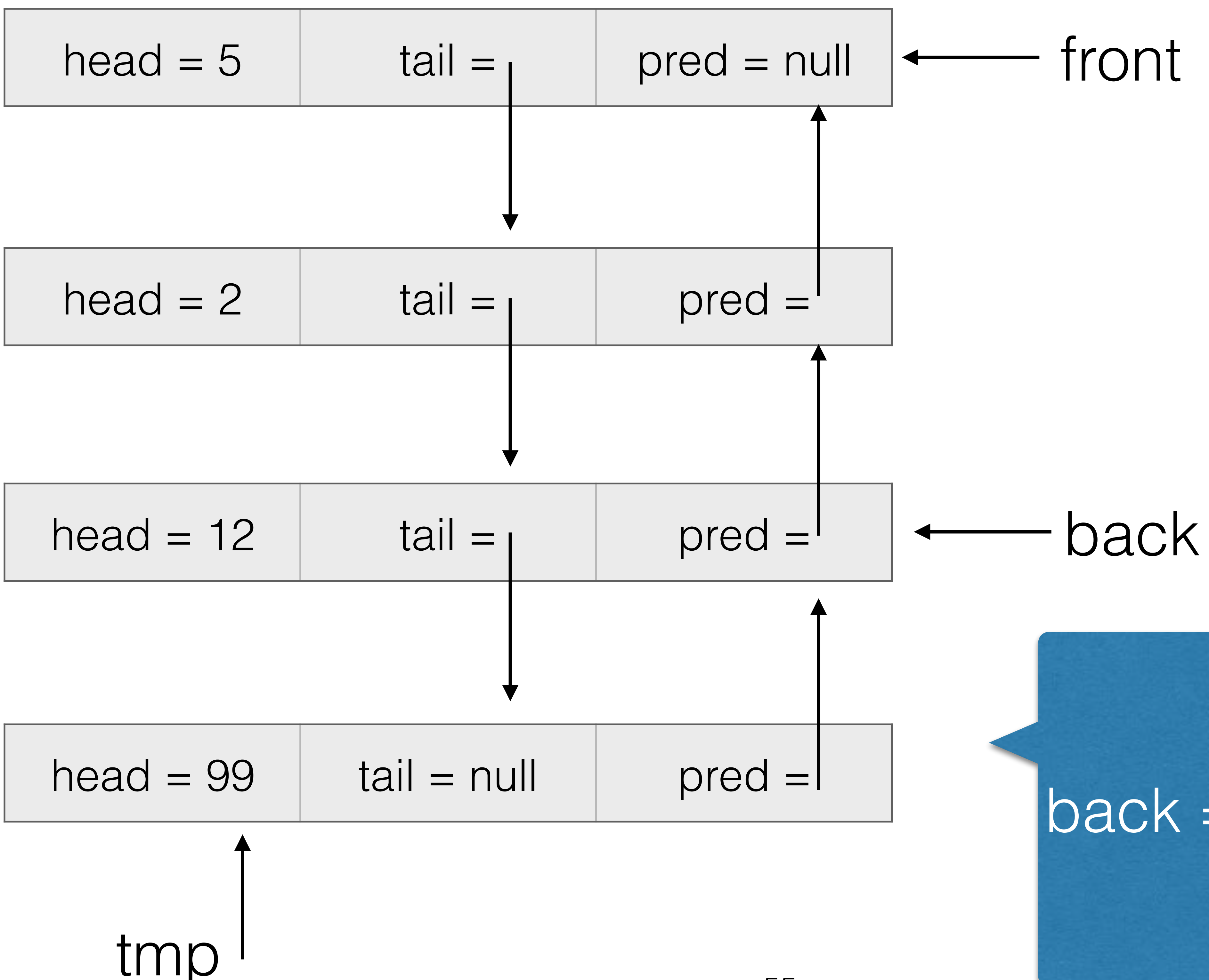


tmp.pred = back
back.tail = tmp

Enqueue

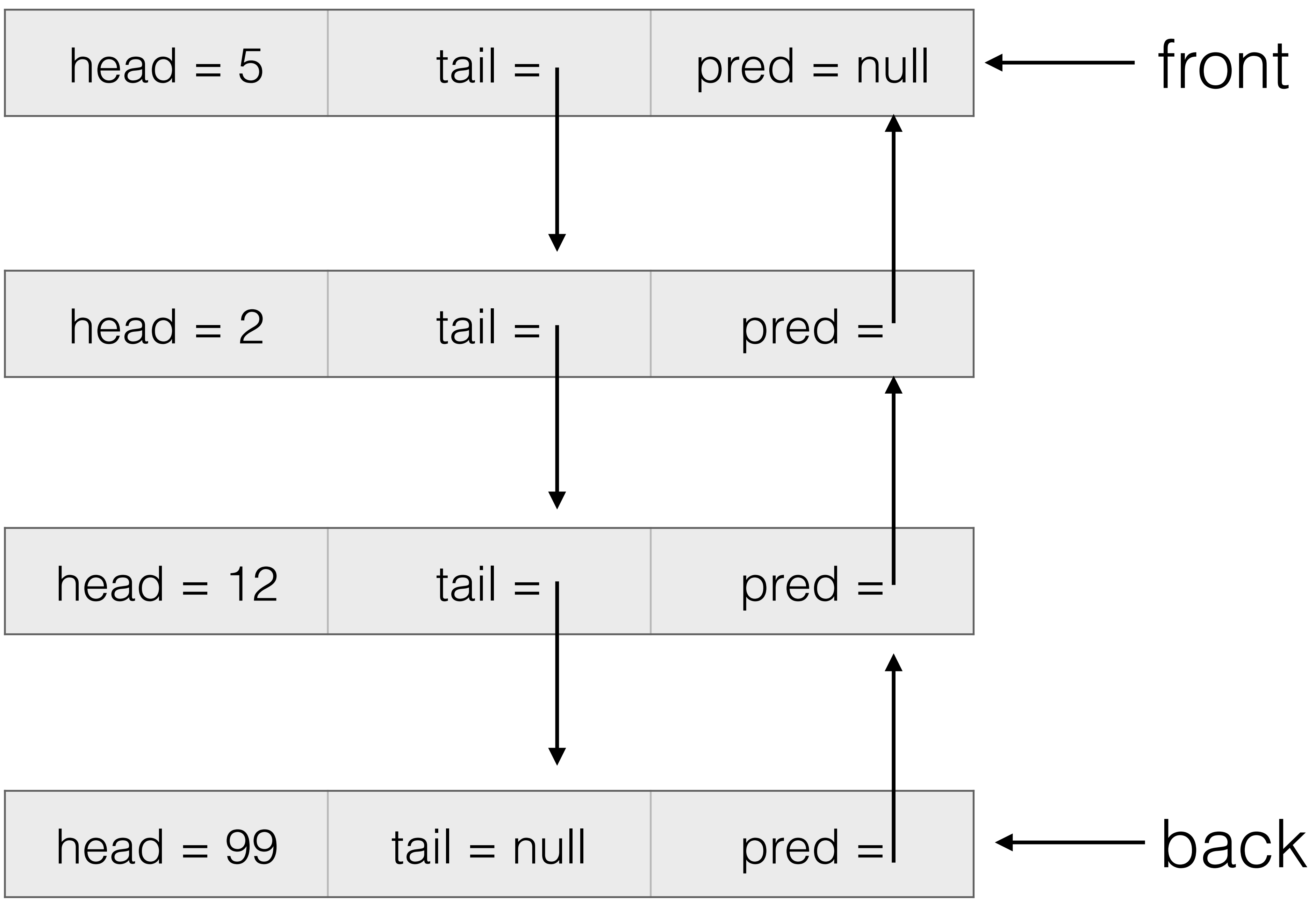

head = 5
tail =
pred = null
front
head = 2
tail =
pred =
head = 12
tail =
pred =
back
head = 99
tail = null
pred =
tmp

Enqueue



back = back.tail

Enqueue



head = 5
tail =
pred = null
front
head = 2
tail =
pred =
head = 12
tail =
pred =
head = 99
tail = null
pred =
back

Small Queues

- Empty and single element queues require special care
- Empty when front and back are both null
- Single element when front and back point to the same node
- Easy to test!

Quicksort

- Invented by Tony Hoare in 1960, inspired by the introduction of recursive function in Algol60
- Divide-and-conquer
- Essence:
    - choose a **pivot** value
    - **partition** the array into two parts: less than the pivot and greater than the pivot (not stable)
    - Sort each part separately, using two recursive calls

Quicksort Example

2	5	16	20	25	1	1	2	145	27	12	122	500	499	345

<= pivot

Choice of pivot: 25

>= pivot

2	5	16	20	25	1	1	2	145	27	12	122	500	499	345

Scan from the front

Find elements >= pivot

Lower

Upper

Scan from the end

Find elements <= pivot

Quicksort Example

Choice of pivot: 25

2	5	16	20	25	1	1	2	145	27	12	122	500	499	345

Lower

Upper

<= pivot

Swap

>= pivot

2	5	16	20	12	1	1	2	145	27	25	122	500	499	345

Lower

Upper

Quicksort Example

Choice of pivot: 25

<= pivot

>= pivot

2	5	16	20	12	1	1	2	145	27	25	122	500	499	345

Upper

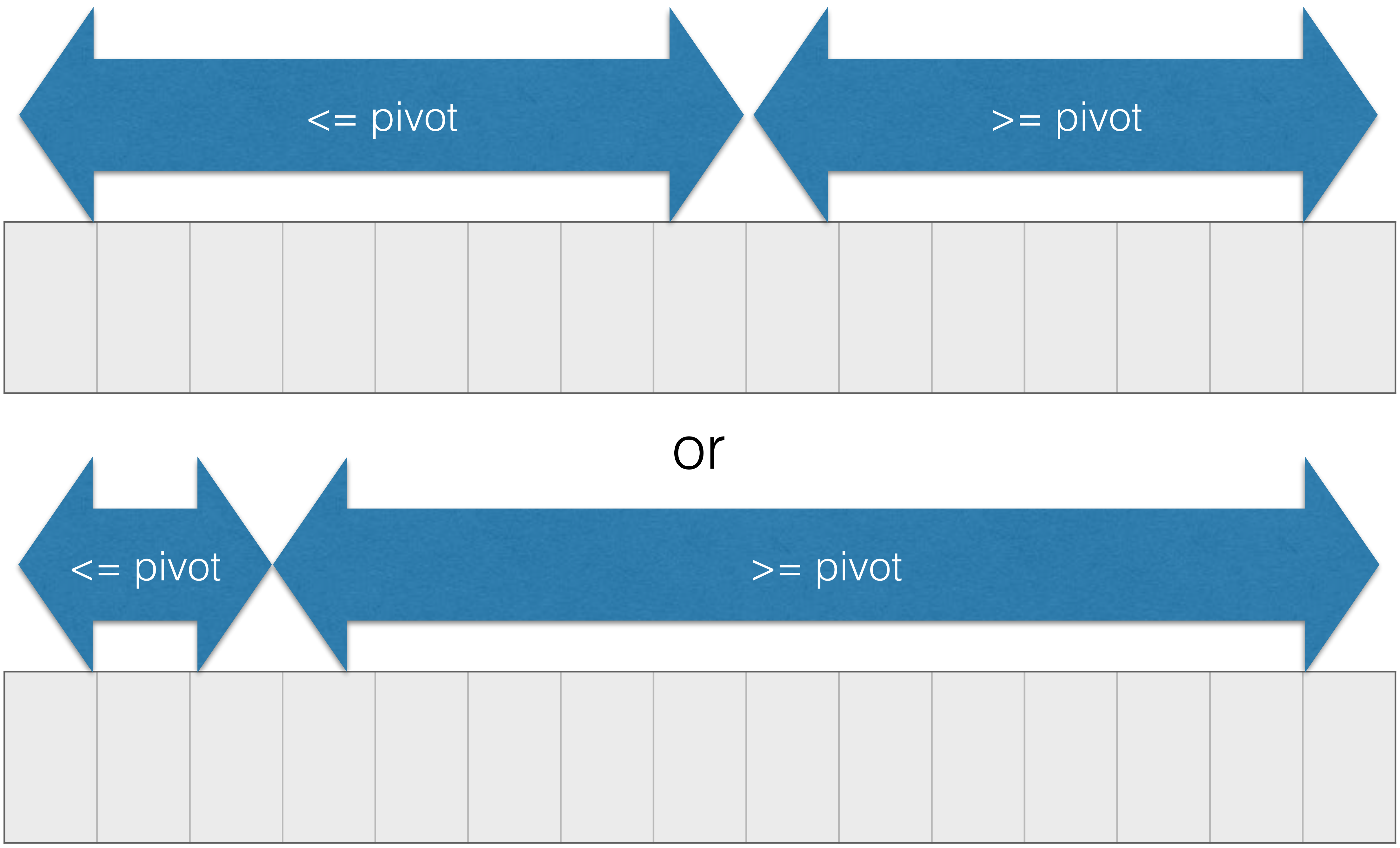
Lower

Stop when pointers cross

Now Quicksort the two sub-arrays

Quicksort

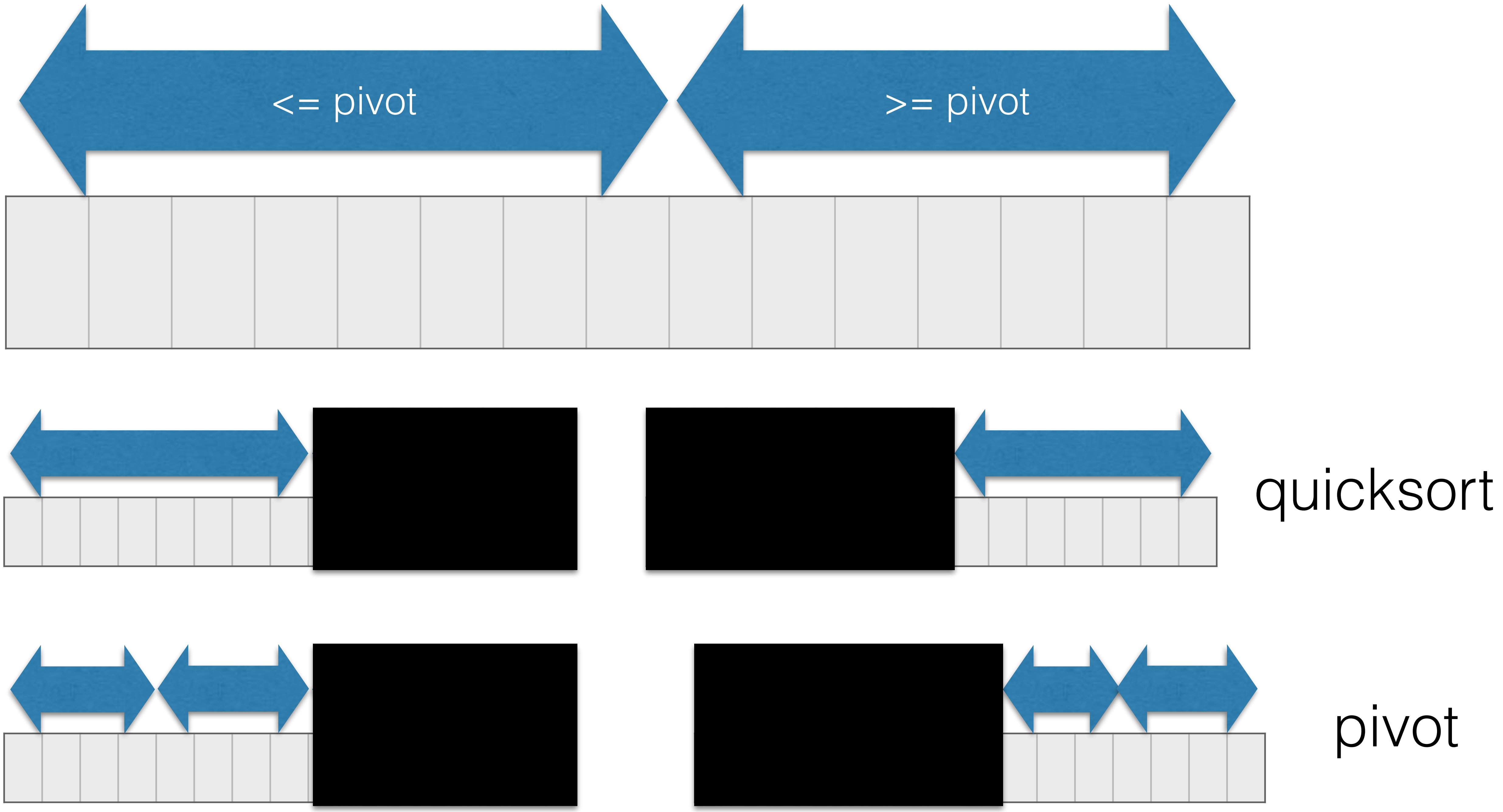
Do not know up front where the pivot will be



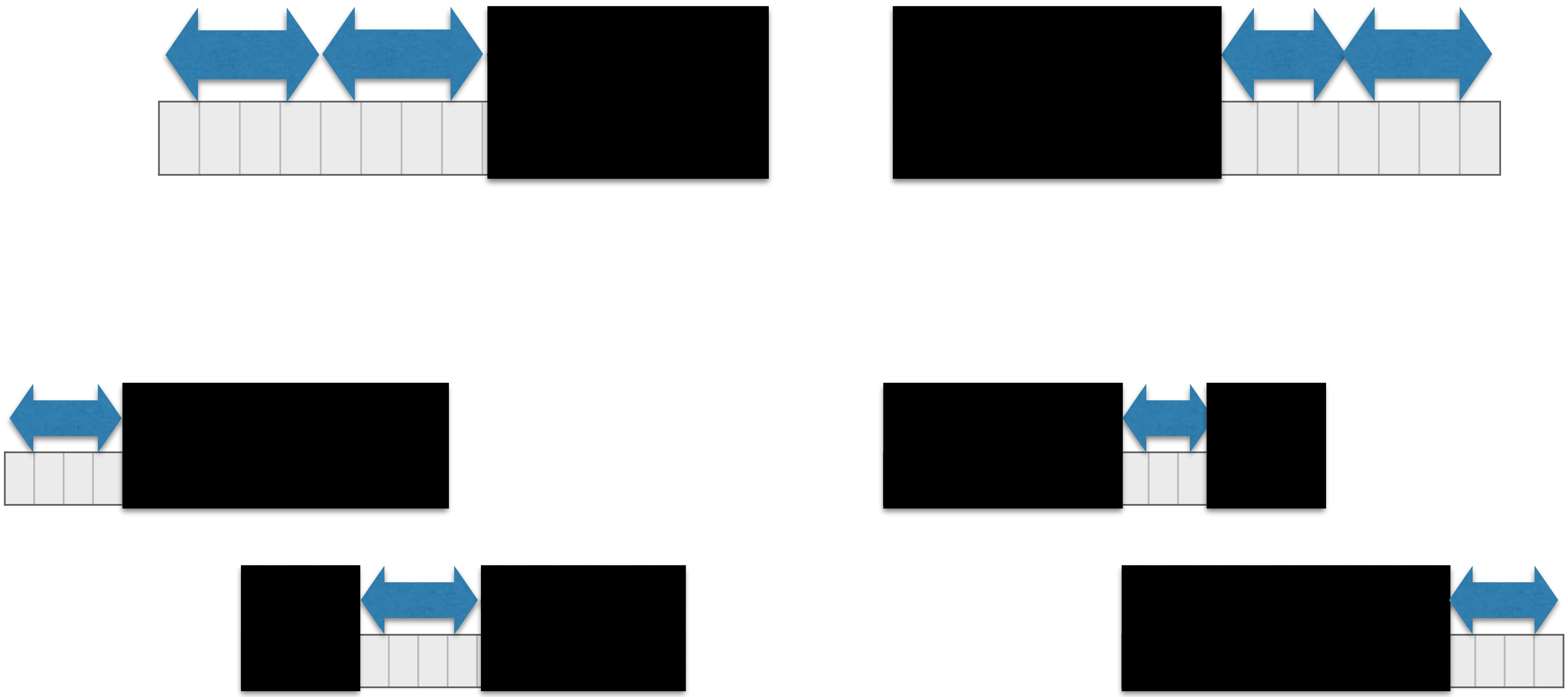
Quicksort Correctness

- After partitioning, everything to the left of the pivot is less than everything to the right
- So, once the left and right partition are sorted, the whole array is sorted
- The algorithms always terminated (no infinite loops) because each successive partition is smaller

Quicksort Call Tree



Quicksort Call Tree



The total time taken at each level is *linear*.
Each element of the array gets passed by at most one pointer during partitioning in the calls to quicksort at one level.

Quicksort Complexity

If the partitions split the array in half, Quicksorting an array of size *n* has *n*log(n)* run time.

n pointer steps at each level, and $log_2(n)$ levels (same reasoning as for binary search)

Quicksort Worst Case

If the partitions leave all-but-one elements on one side, Quicksorting an array is quadratic.

approx. *n* pointer steps at each level, and *n* levels

Quicksort is usually efficient, but care must be taken to avoid pathological cases.

Pivot Choice

- Consider Quicksorting an already sorted list (or a reverse sorted list)
    - Always choosing the first (or last) element as pivot gives bad partitioning
    - Choosing the middle element as pivot gives ideal partitions
- Could choose a random element as pivot
- Could take the middle value of three elements
- For any strategy, there will be some occasions where performance is bad