# Leveraging Concurrent Collections to Simplify Application Design

José Paumard
PHD, JAVA CHAMPION, JAVA ROCK STAR

@JosePaumard https://github.com/JosePaumard

# Agenda

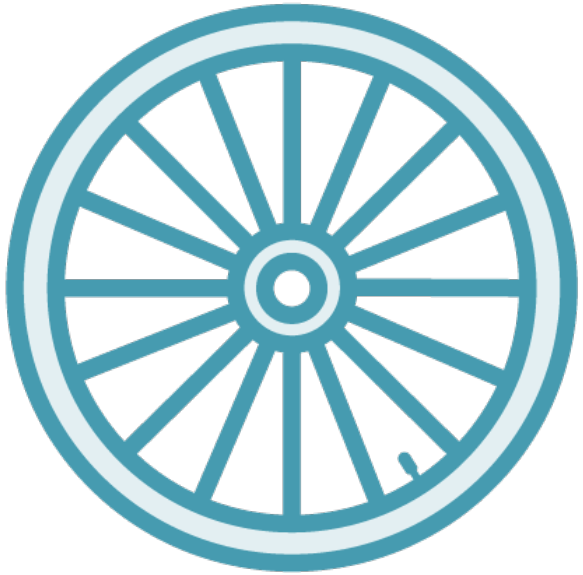
Concurrent collections and maps

Collections: Queue, BlockingQueue

Map: ConcurrentMap

And implementations

# Concurrent Interfaces

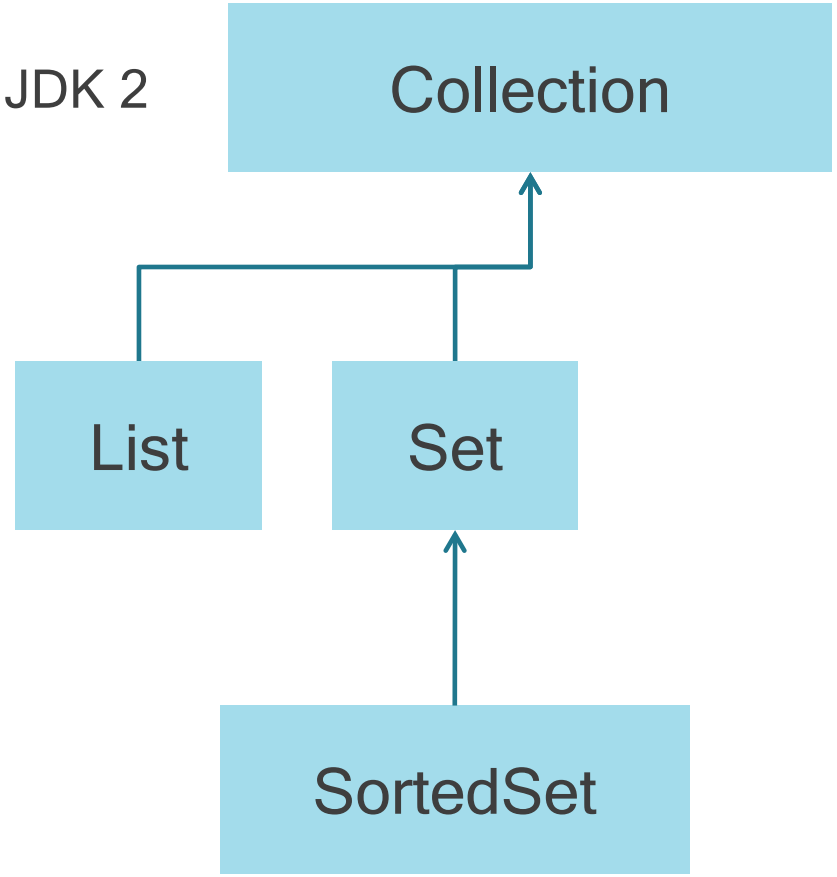


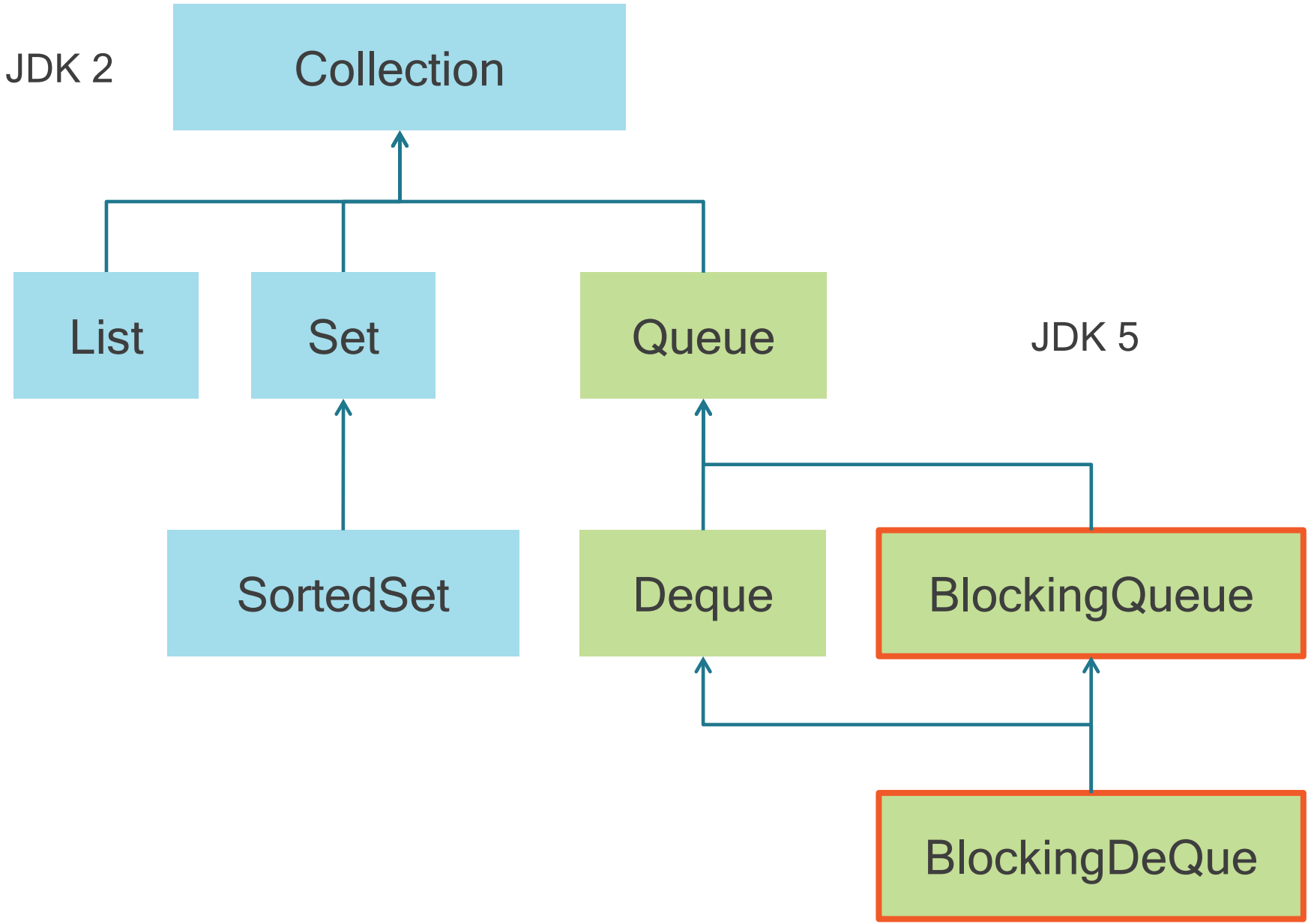
Implementing the Producer / Consumer at the API level vs the application level

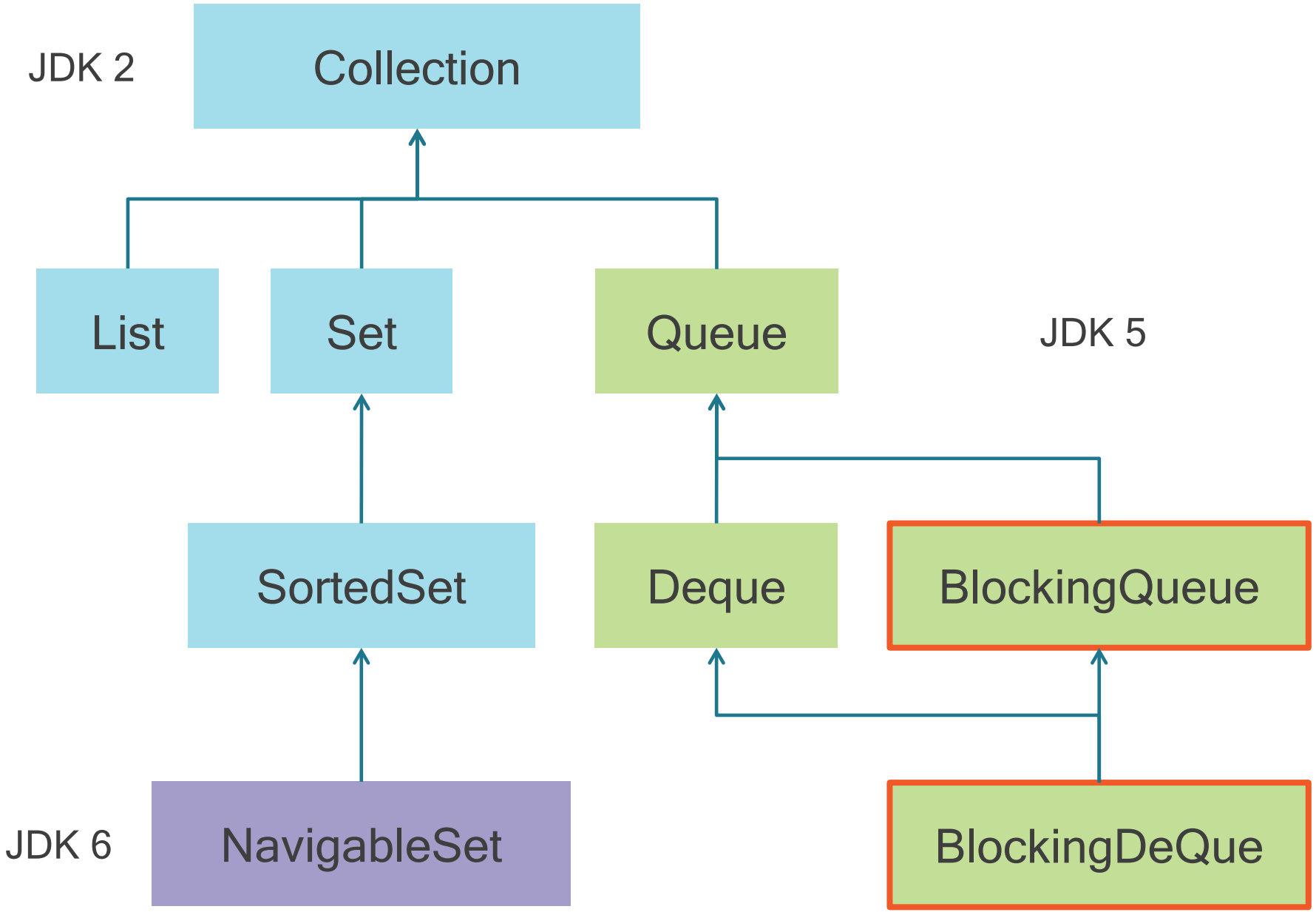
For that, we need new API, new Collections

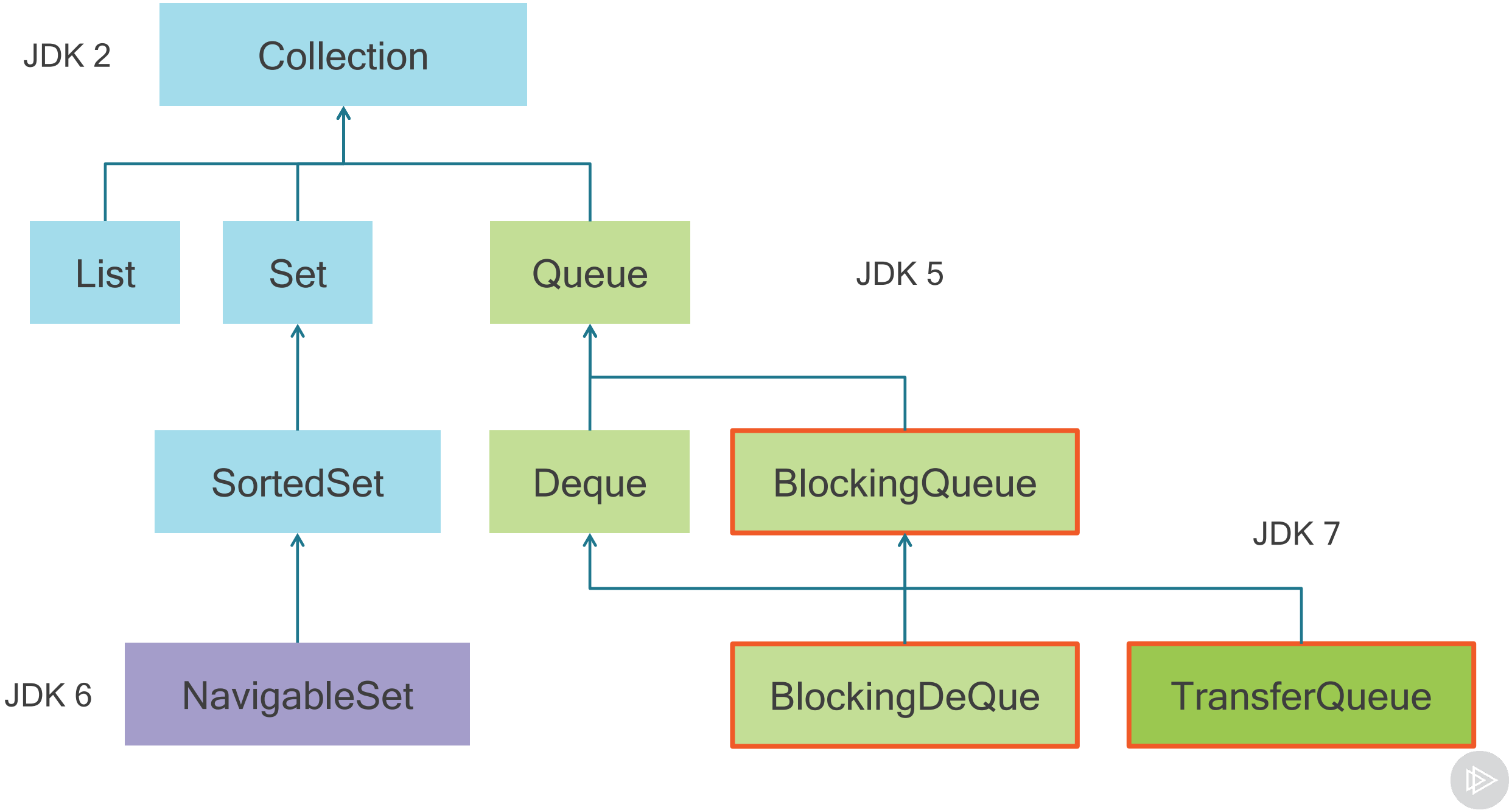
Two branches: Collection and Map

JDK 2

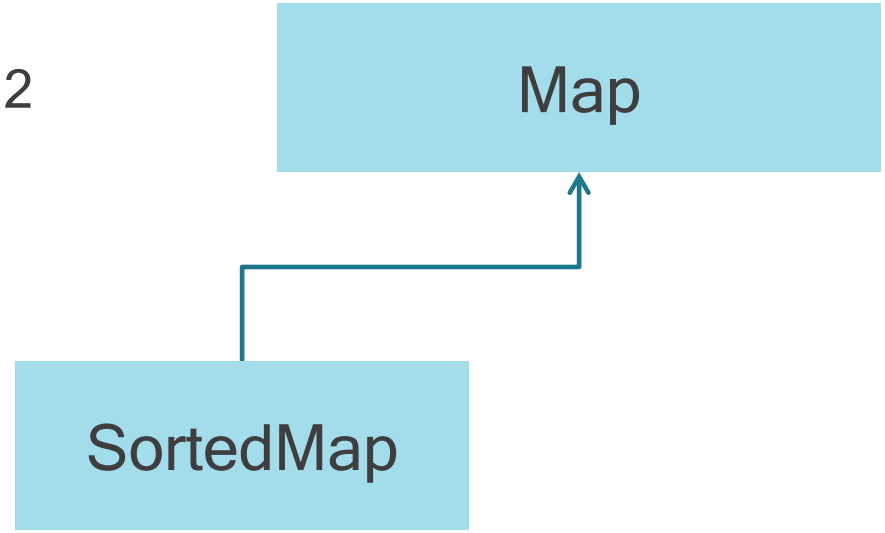


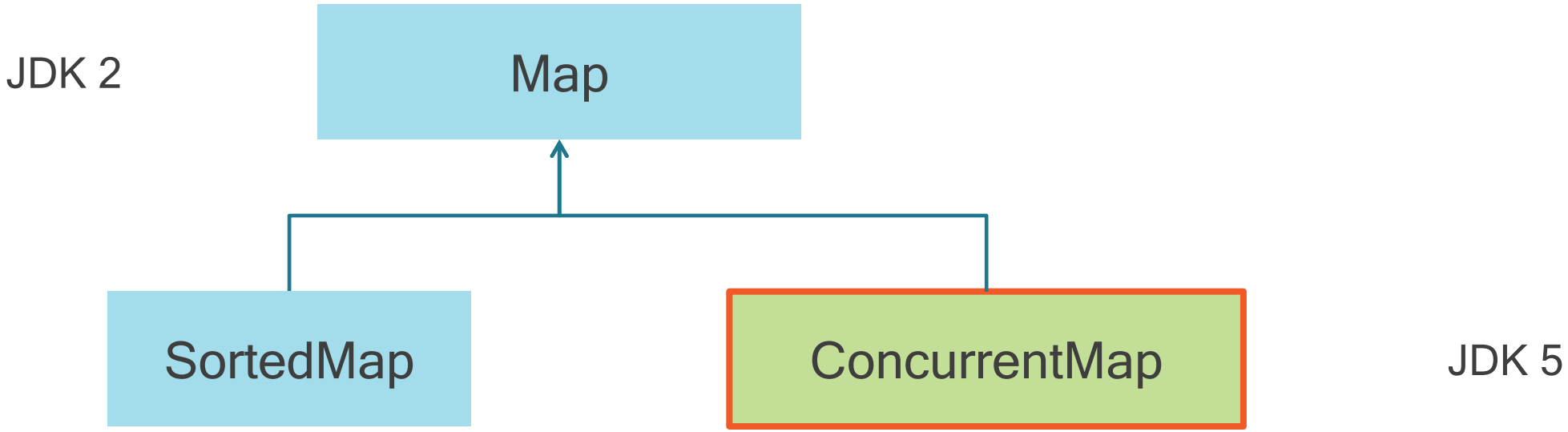

JDK 2
Collection
List
Set
Queue
JDK 5
SortedSet
Deque
BlockingQueue
BlockingDeQue

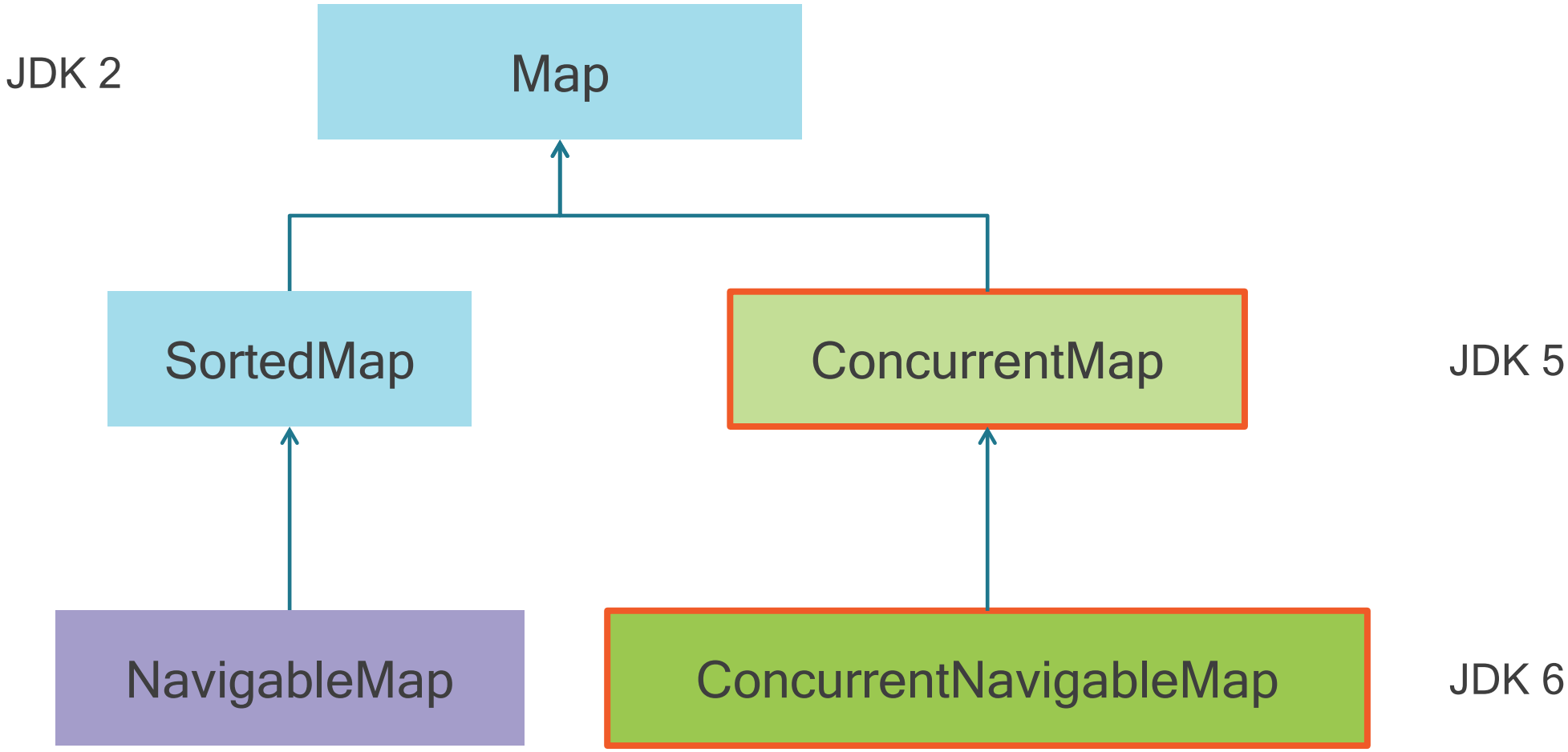

JDK 2
Collection
List
Set
Queue
JDK 5
SortedSet
Deque
BlockingQueue
JDK 6
NavigableSet
BlockingDeQue

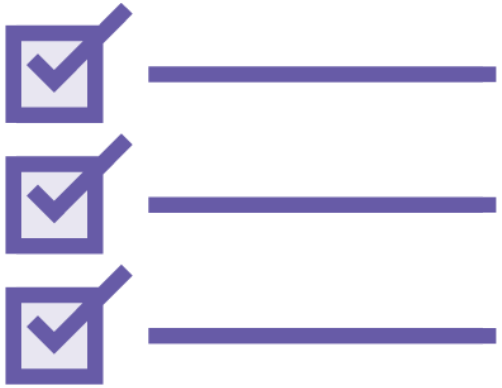

JDK 2
Collection
List
Set
Queue
JDK 5
SortedSet
Deque
BlockingQueue
JDK 7
JDK 6
NavigableSet
BlockingDeQue
TransferQueue

JDK 2




JDK 2
Map
SortedMap
ConcurrentMap
JDK 5


JDK 2
Map
SortedMap
ConcurrentMap
JDK 5
NavigableMap
ConcurrentNavigableMap
JDK 6



Concurrent interfaces, that define contracts in concurrent environments

And implementations that follow these contracts

But concurrency is complex!

Dealing with 10 threads is not the same as 10k threads...

So we need different implementations

# Concurrent Lists

# About Vectors and Stacks

There are thread-safe structures: Vector and Stack

They are legacy structures, very poorly implemented

They should not be used!

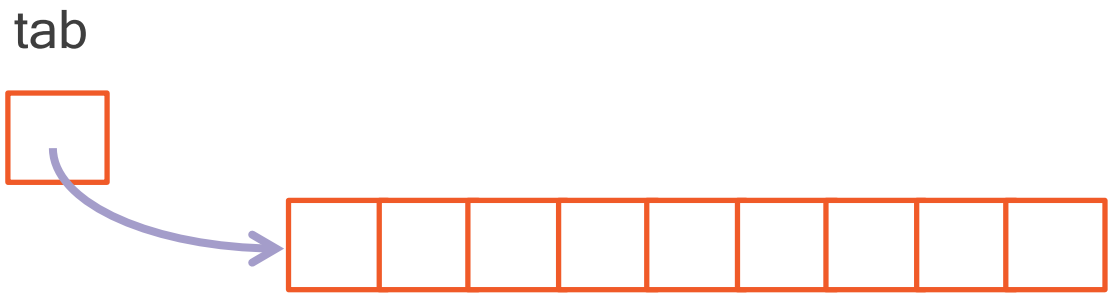
# Copy on Write

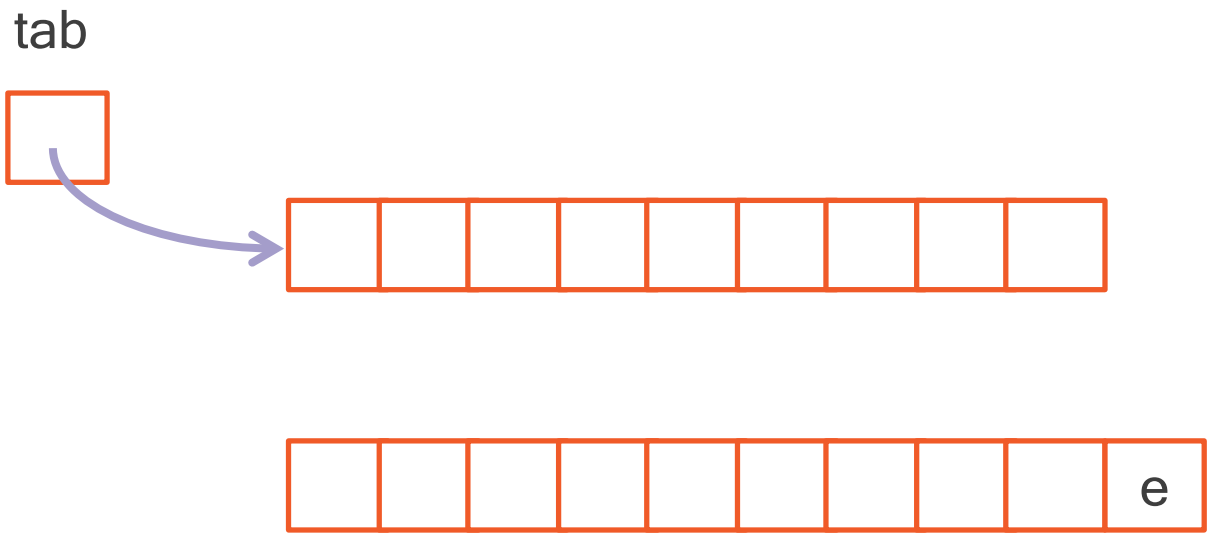
Exists for list and set

No locking for read operations

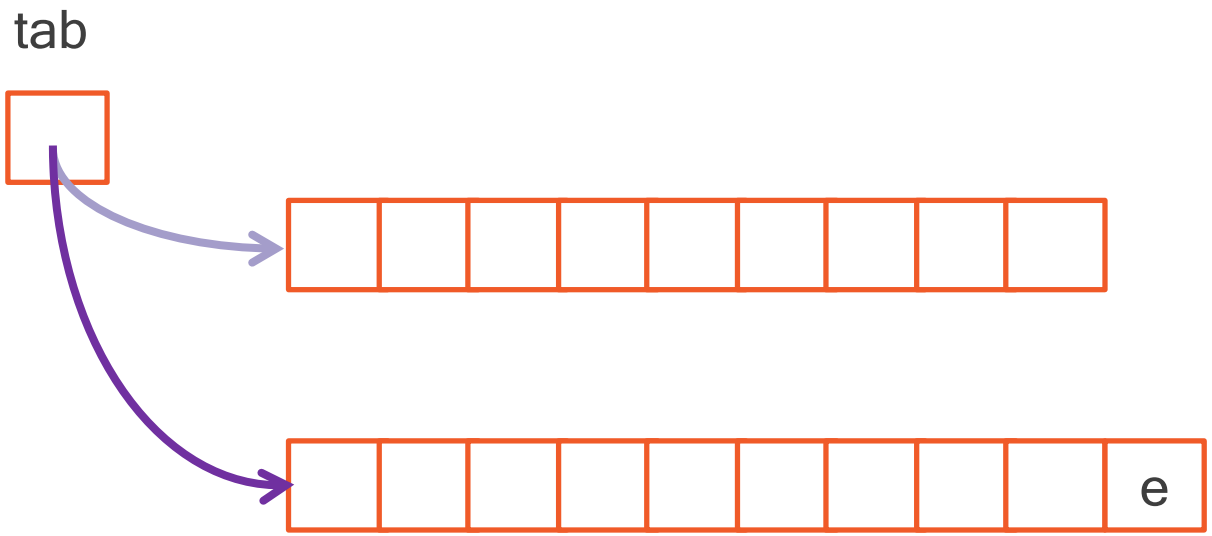
Write operations create a new structure

The new structure then replaces the previous one


tab




add(e)




add(e)

*synchronized*

# Copy on Write

The thread that already has a reference on the previous array will not see the modification

The new threads will see the modification

# Copy on Write

Two structures:

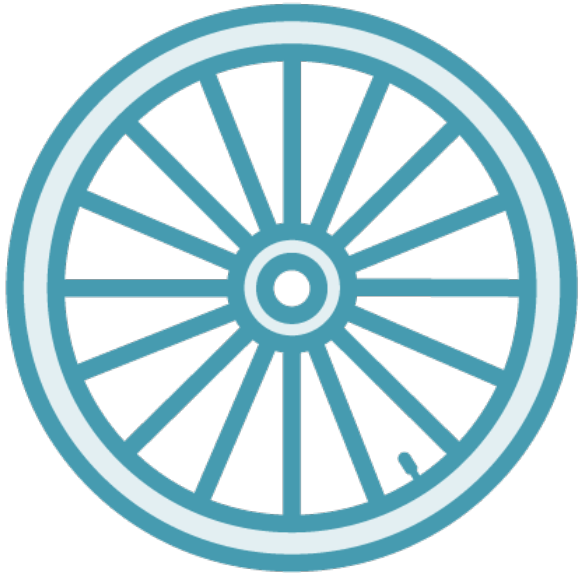
- CopyOnWriteArrayList
- CopyOnWriteArraySet

# Copy on Write Structures

Work well when there are many reads and very, very few writes

Example: application initialization

# Queues and Stacks



Queue and Deque: interfaces

ArrayBlockingQueue: a bounded blocking queue built on an array

ConcurrentLinkedQueue: an unbounded blocking queue

# How Does a Queue Work?

Two kinds of queues: FIFO (queue) and LIFO (stack)

In the JDK we have the following
- Queue: queue
- Deque: both a queue and a stack

There is no “pure” stack (the Stack class does not count)

Producer

*tail*

Consumer

*head*


Queue

Producer

*tail*

Consumer

*head*


Queue

# We Are in a Concurrent World

So we can have as many producers and consumers as we need

Each of them in its own thread

A thread does not know how many elements are in the queue...

# Two Questions

1) What happens if the queue / stack is full and we need to add an element to it?

2) What happens if the queue / stack is empty and we need to get an element from it?

# Adding an Element to a Queue That Is Full


```
boolean add(E e); // fail: IllegalArgumentException

// fail: return false
boolean offer(E e);
```

# If the Queue Is a BlockingQueue


```
boolean add(E e); // fail: IllegalArgumentException

// fail: return false
boolean offer(E e);

// blocks until a cell becomes available
void put(E e);
```

# If the Queue Is a BlockingQueue


```
boolean add(E e); // fail: IllegalArgumentException

// fail: return false
boolean offer(E e); boolean offer(E e, timeOut, timeUnit);

// blocks until a cell becomes available
void put(E e);
```

# Adding Elements at the Tail of a Queue

Two behaviors:

- Failing with an exception
- Failing and returning false

And for blocking queue:

- Blocking until the queue can accept the element

# We Also Have Deque And BlockingDeque

Deque can accept elements at the head of a queue:
- addFirst(), offerFirst(),

And for the BlockingDeque
- putFirst()

# Other Methods

Queues have also get and peek operations

Queue:
- Returns null: poll() and peek()
- Exception: remove() and element()

BlockingQueue:
- blocks: take()

# Other Methods

Queues have also get and peek operations

Deque:
- Returns null: pollLast() and peekLast()
- Exception: removeLast() and getLast()

BlockingDeque:
- blocks: takeLast()

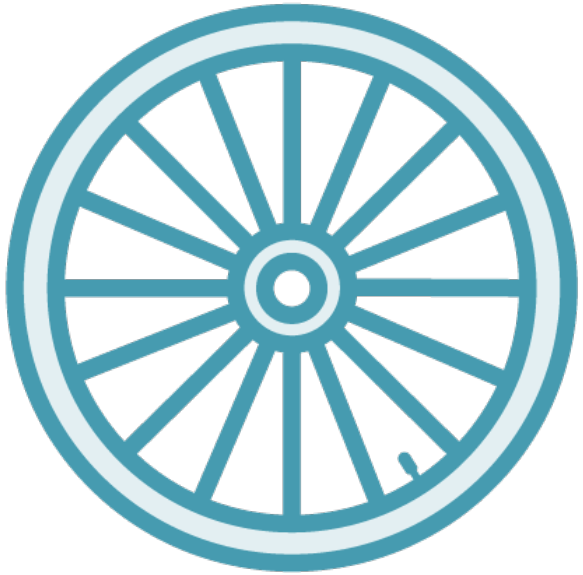
# Queue and BlockingQueue

Four different types of queues: they may be blocking or not, may offer access from both sides or not

Different types of failure: special value, exception, blocking

That makes the API quite complex, with a lot of methods

# Concurrent Maps



One interface:

ConcurrentMap: redefining the JavaDoc

Two implementations:

ConcurrentHashMap: JDK 7 & JDK 8

ConcurrentSkipListMap: JDK 6, no synchronization

# Atomic Operations

ConcurrentMap defines atomic operations:

- putIfAbsent(key, value)
- remove(key, value)
- replace(key, value)
- replace(key, existingValue, newValue)

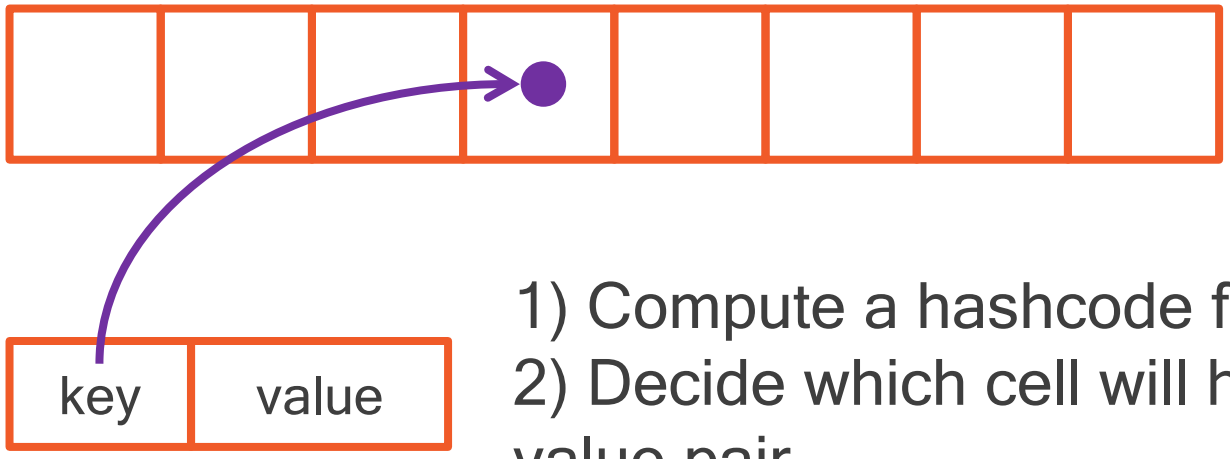
# ConcurrentMap Implementations

ConcurrentMap implementations are:

- Thread-safe maps
- Efficient up to a certain number of threads
- A number of efficient, parallel special operations

# How Does a HashMap Work?

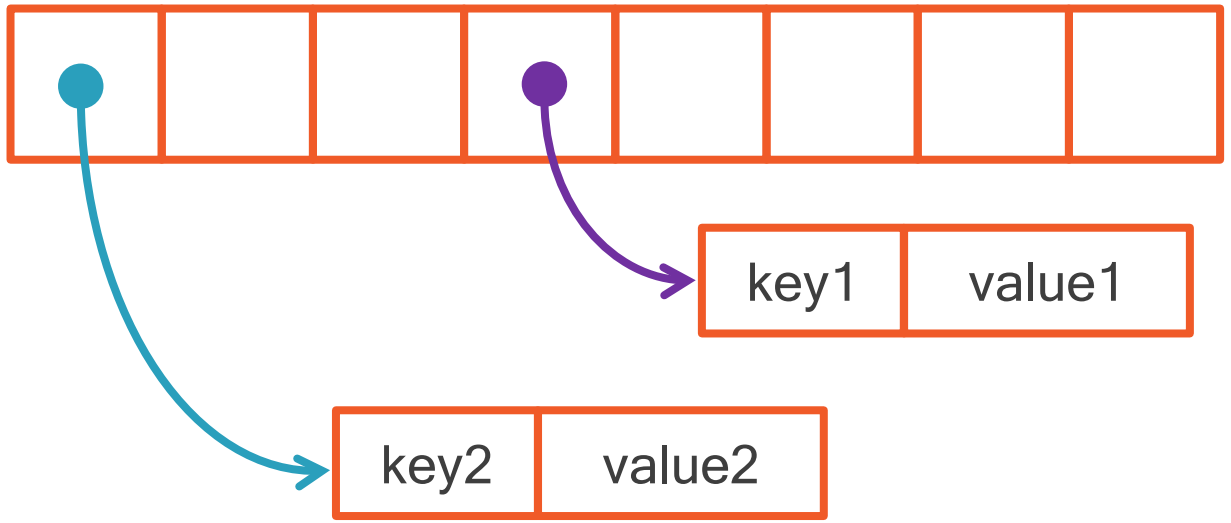
A hashmap is built on an array




1) Compute a hashcode from the key
2) Decide which cell will hold the key / value pair

# How Does a HashMap Work?

A hashmap is built on an array




Each cell is called a “bucket”

# Understanding the Problem

Adding a key / value pair to a map is a several steps problem

1) Compute the hashcode of the key

2) Check if the bucket is there or not

3) Check if the key is there or not

4) Update the map

In a concurrent map these steps must not be interrupted by another thread

# Understanding the Problem

The only way to guard an array-based structure is to lock the array

Synchronizing the put would work, but...

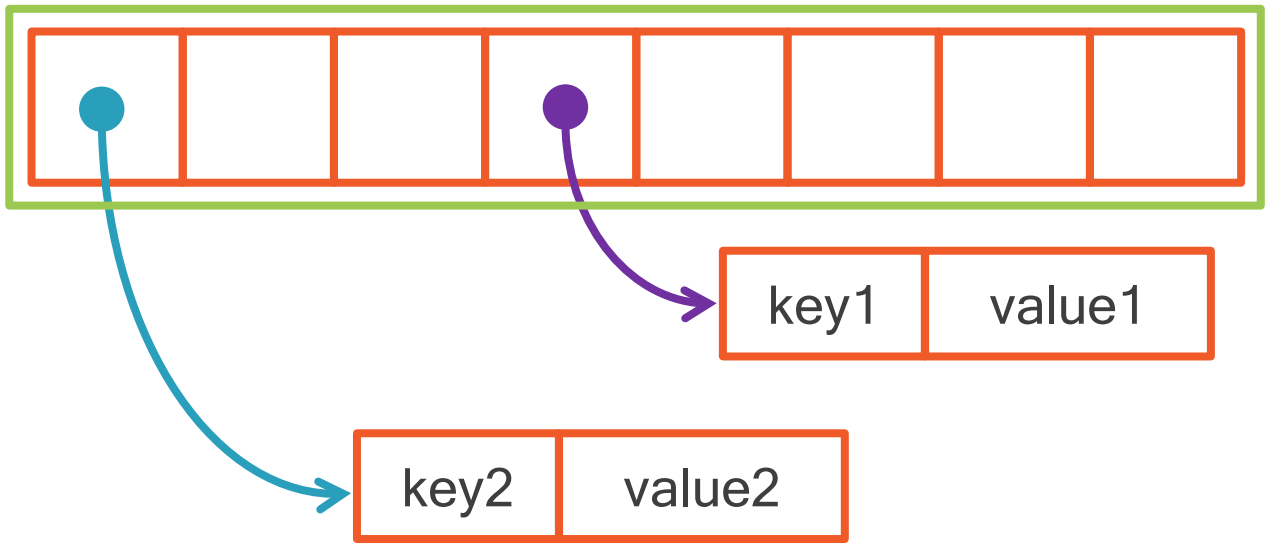
It would be very inefficient to block all the map:

- we should allow several threads on different buckets

- we should allow concurrent reads

# Synchronizing All the Map

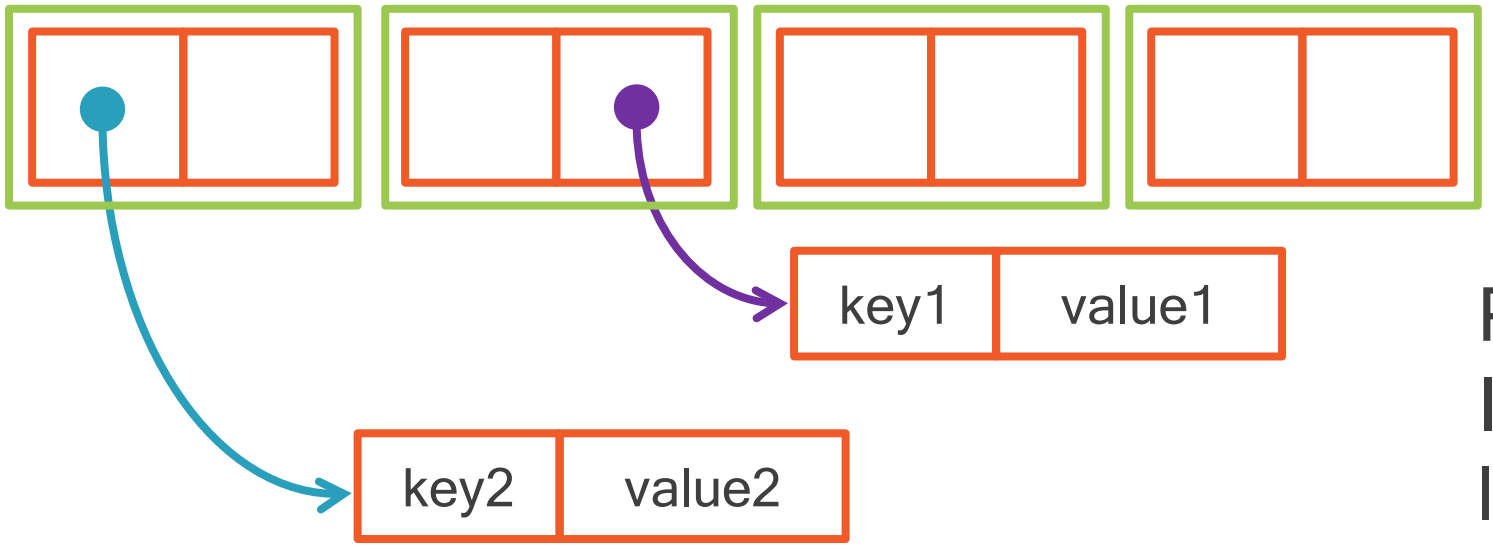
Synchronizing on the array itself




Pros: it works!
Cons: one write blocks everything

# Synchronizing All the Map

Synchronizing on parts of the array




Pros: it works!
It allows for a certain level of parallelism

# ConcurrentHashMap From JDK 7

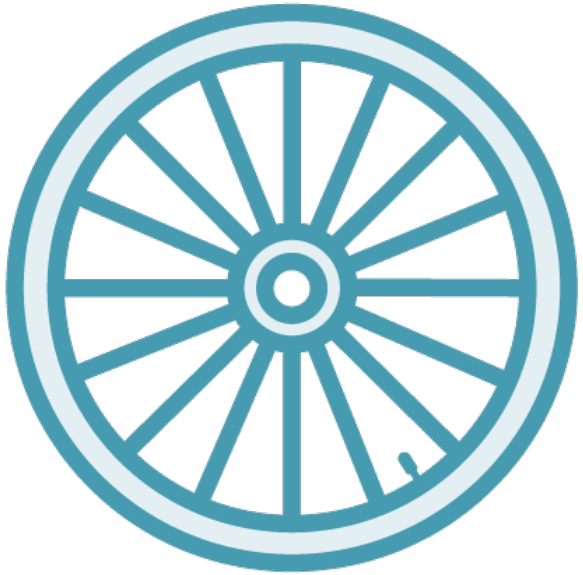
Built on a set of synchronized segments

Number of segments = concurrency level (16 - 64k)

This sets the number of threads that can use this map

The number of key / value pairs has to be (much) greater than the concurrency level

# ConcurrentHashMap JDK 8



The implementation changed completely

Serialization: compatible with JDK 7 in both ways

Tailored to handle heavy concurrency and millions of key / value pairs

Parallel methods implemented

```
ConcurrentHashMap<Long, String> map = ...; // JDK 8
String result =
   map.search(10_000,
            (key, value) ->
               value.startsWith("a") ? "a" : null
            );
```

# ConcurrentHashMap: Parallel Search

The first parameter is the parallelism threshold

The second is the operation to be applied

Also searchKeys(), searchValues(), searchEntries()

```
ConcurrentHashMap<Long, List<String>> map = ...; // JDK 8
String result =
   map.reduce(10_000,
            (key, value) -> value.size(),
            (value1, value2) -> Integer.max(value1, value2)
           );
```

# ConcurrentHashMap: Parallel Map / Reduce

The first bifunction maps to the element to be reduced

The second bifunction reduces two elements together

```
ConcurrentHashMap<Long, List<String>> map = ...; // JDK 8
String result =
   map.forEach(10_000,
            (key, value) -> value.removeIf(s -> s.length() > 20)
           );
```

# ConcurrentHashMap: Parallel for Each

The biconsumer is applied to all the key / value pairs of the map

Also forEachKeys(), forEachValues(), forEachEntry()

```
Set<String> set = ConcurrentHashMap.<String>newKeySet(); // JDK 8
```

# ConcurrentHashMap to Create Concurrent Sets

This concurrent hash map can also be used as a concurrent set

No parallel operations available

# ConcurrentHashMap From JDK 8

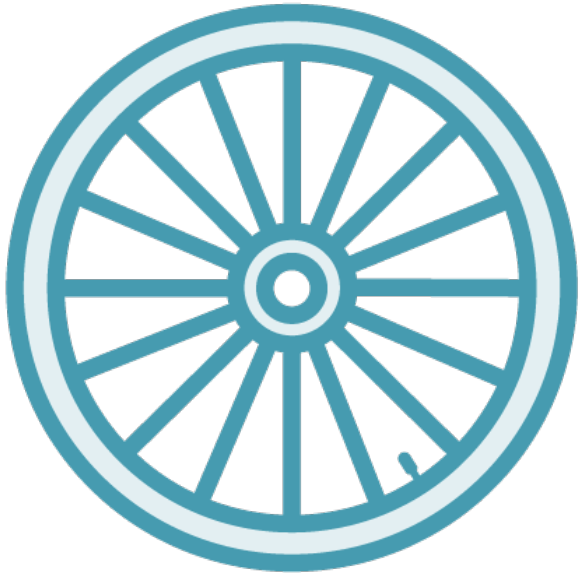
A fully concurrent map

Tailored to handle millions of key / value pairs

With built-in parallel operations

Can be used to create concurrent sets

# Concurrent Skip Lists



Another concurrent map (JDK 6)

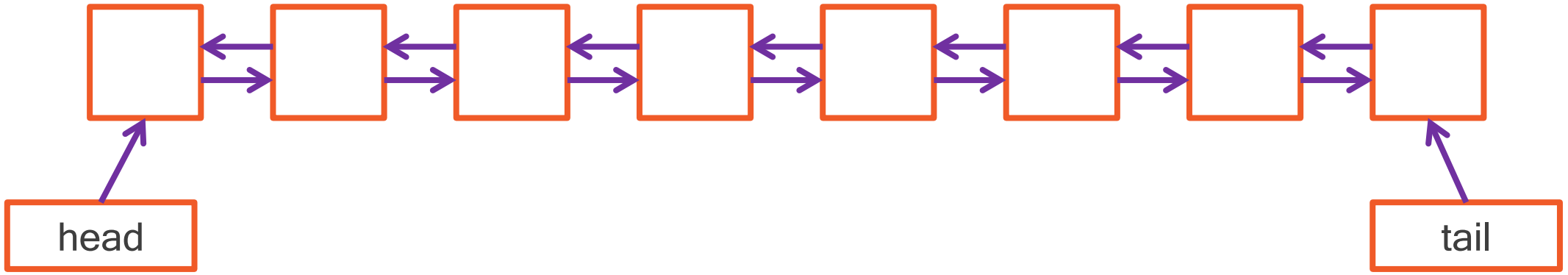
A skip list is a smart structure used to create linked lists

Relies on atomic reference operations, no synchronization

That can be used to create maps and sets

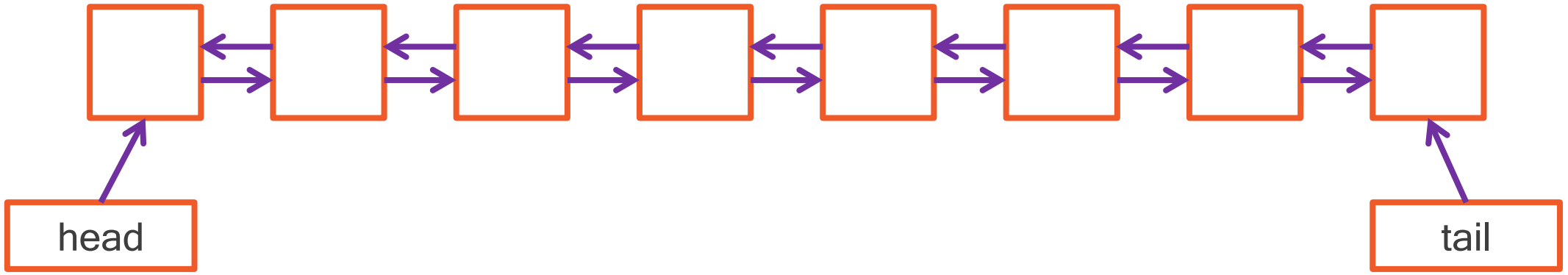
# Skip Lists

It starts with a classical linked list



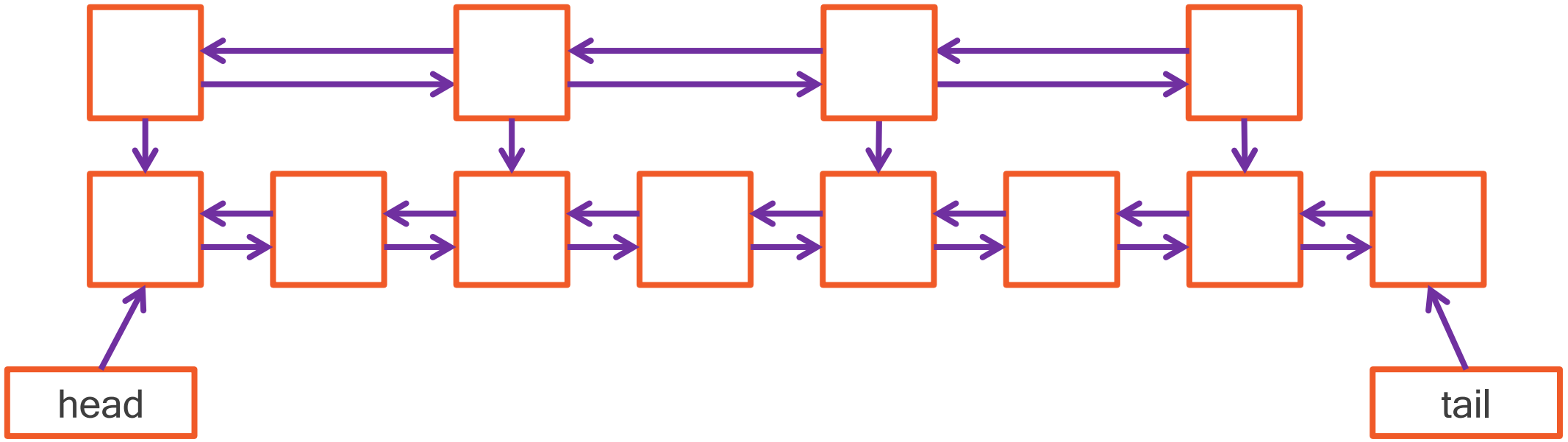
# Skip Lists

Problem: it takes time to reach element *N*
Complexity is O(*N*)



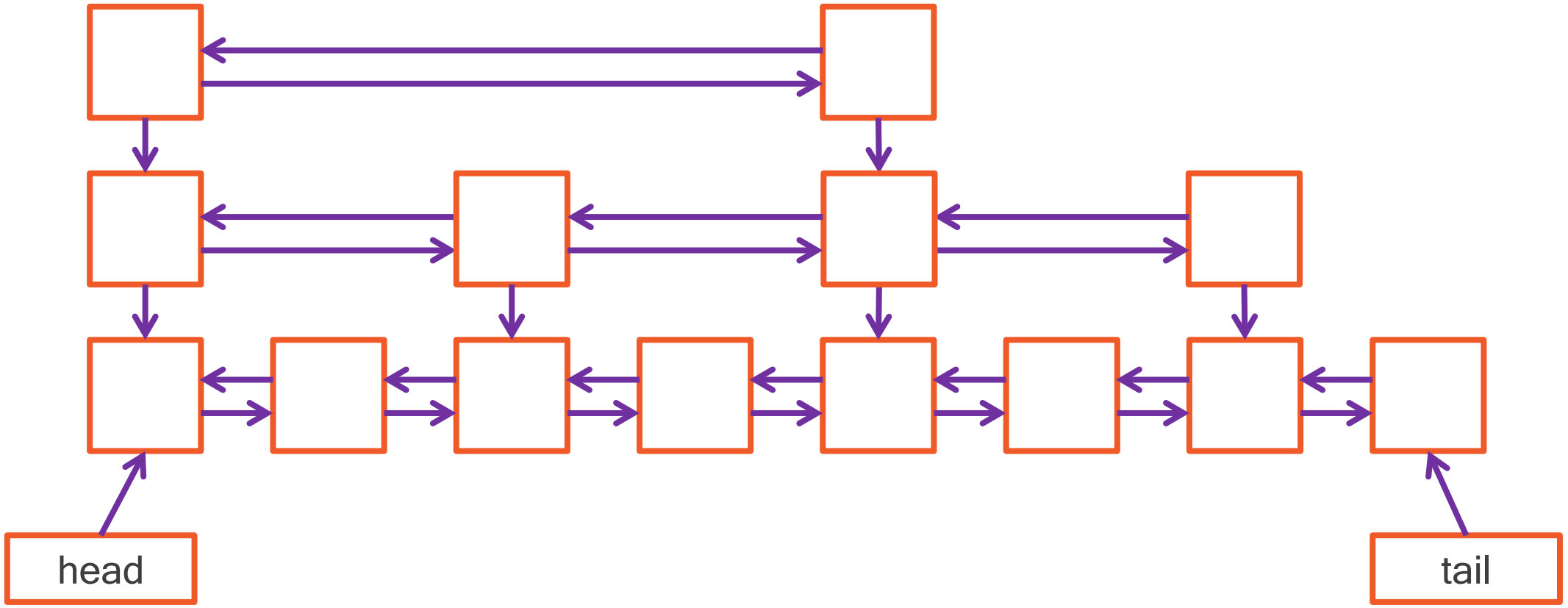
# Skip Lists

Solution: create a fast access list



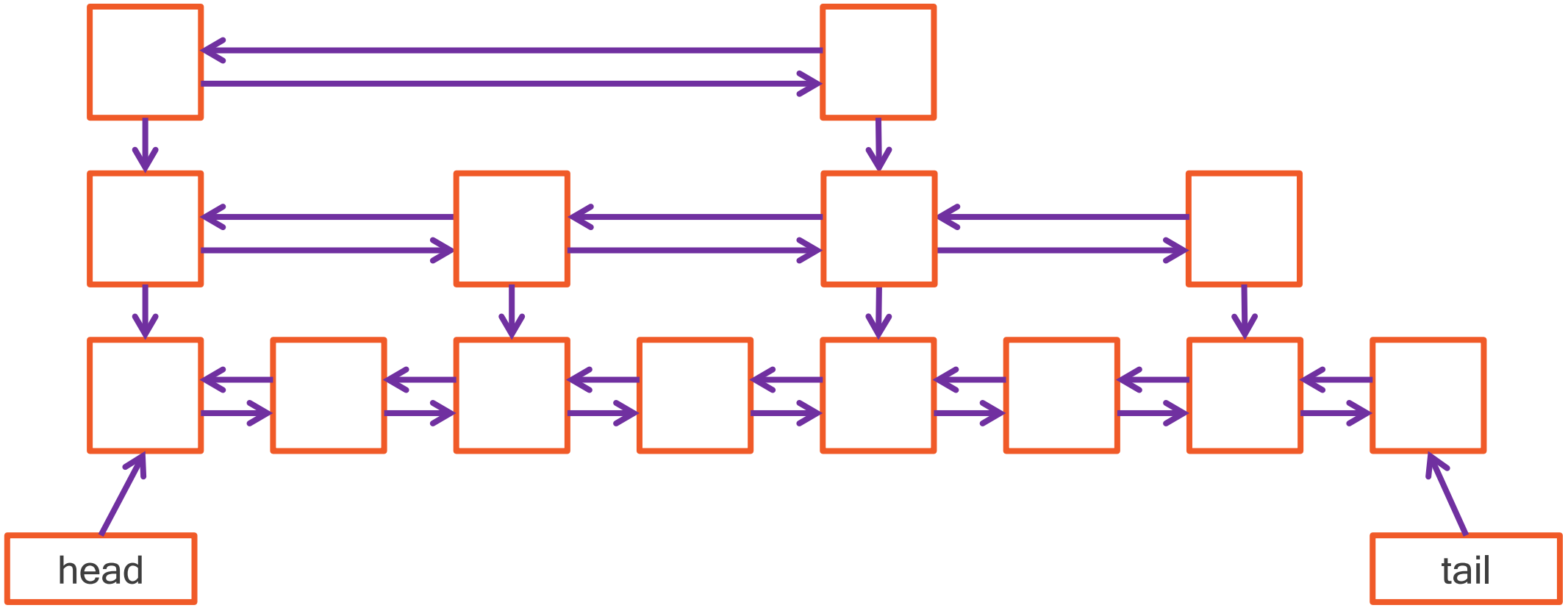
# Skip Lists

We can even create more than one



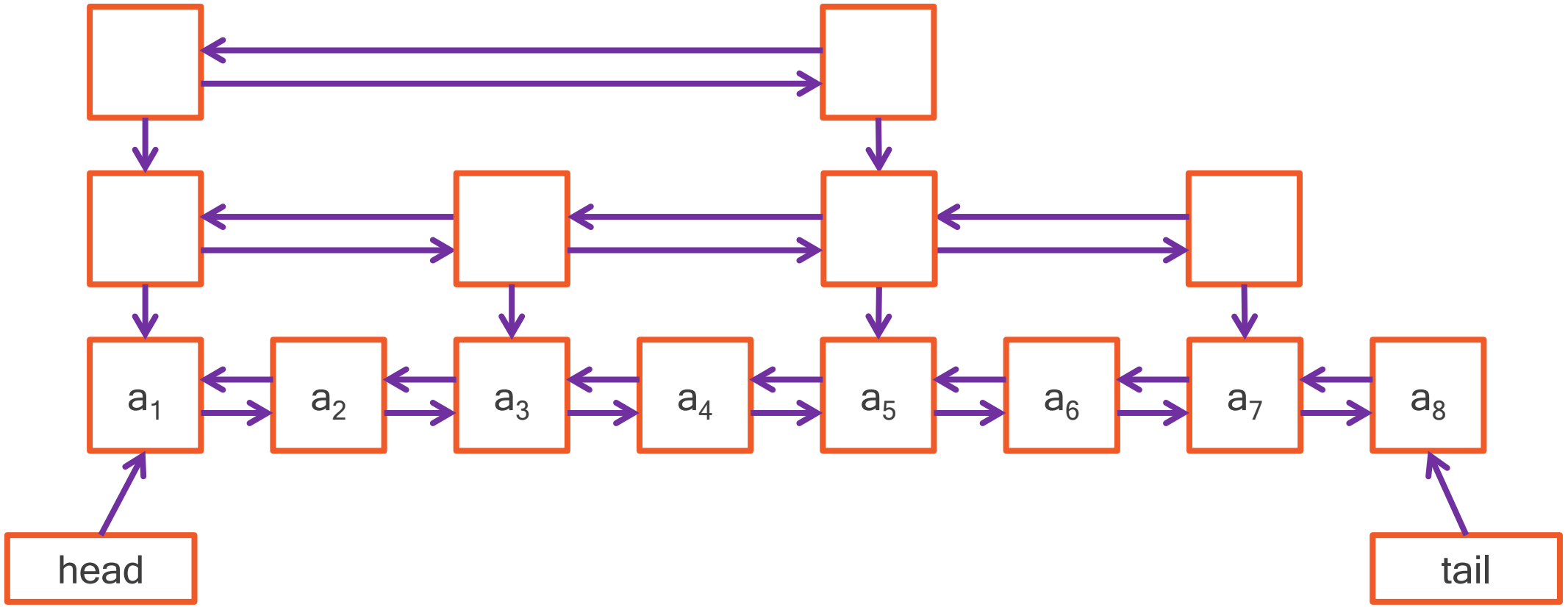
# Skip Lists

It assumes that the elements are sorted



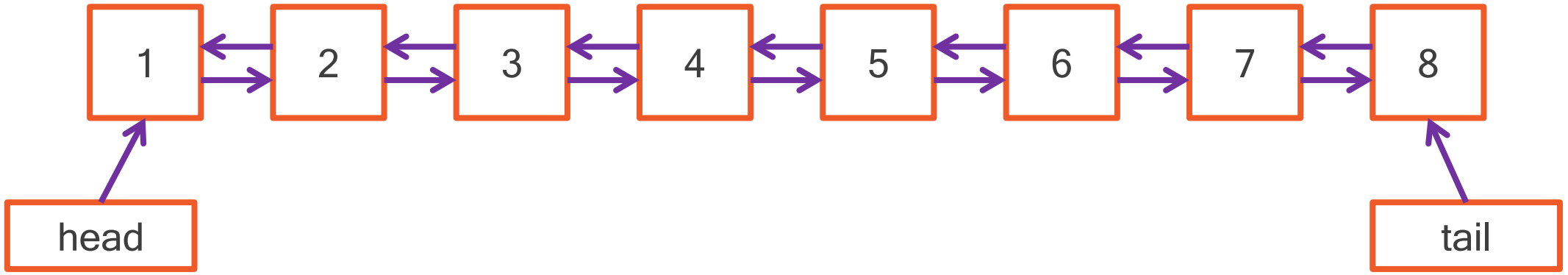
# Skip Lists

The access time is now in O(log(N))



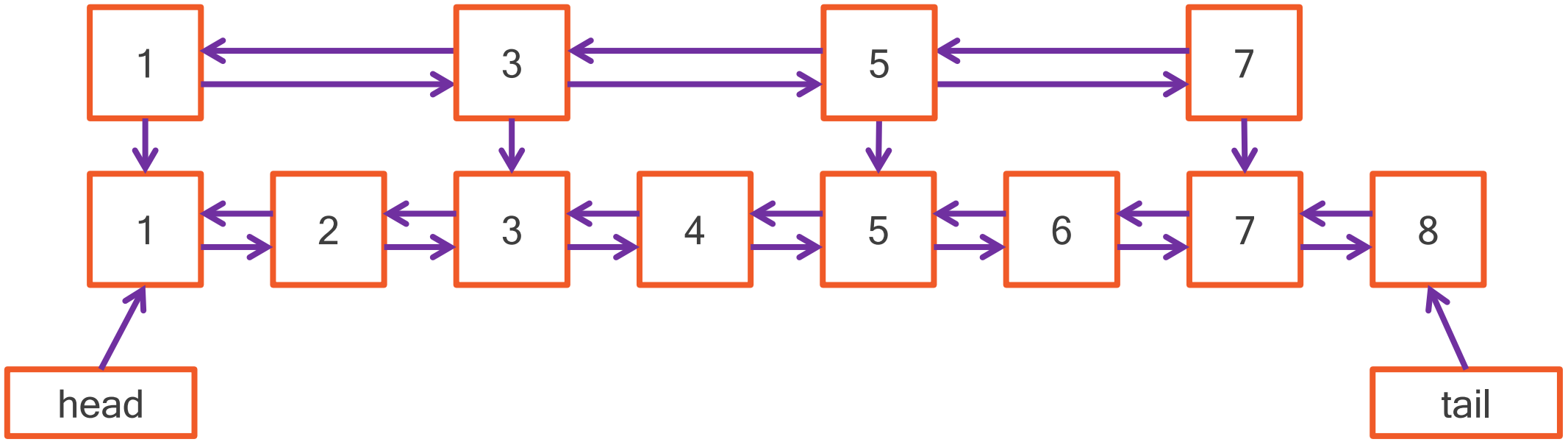
# Skip Lists

Let us see how it works



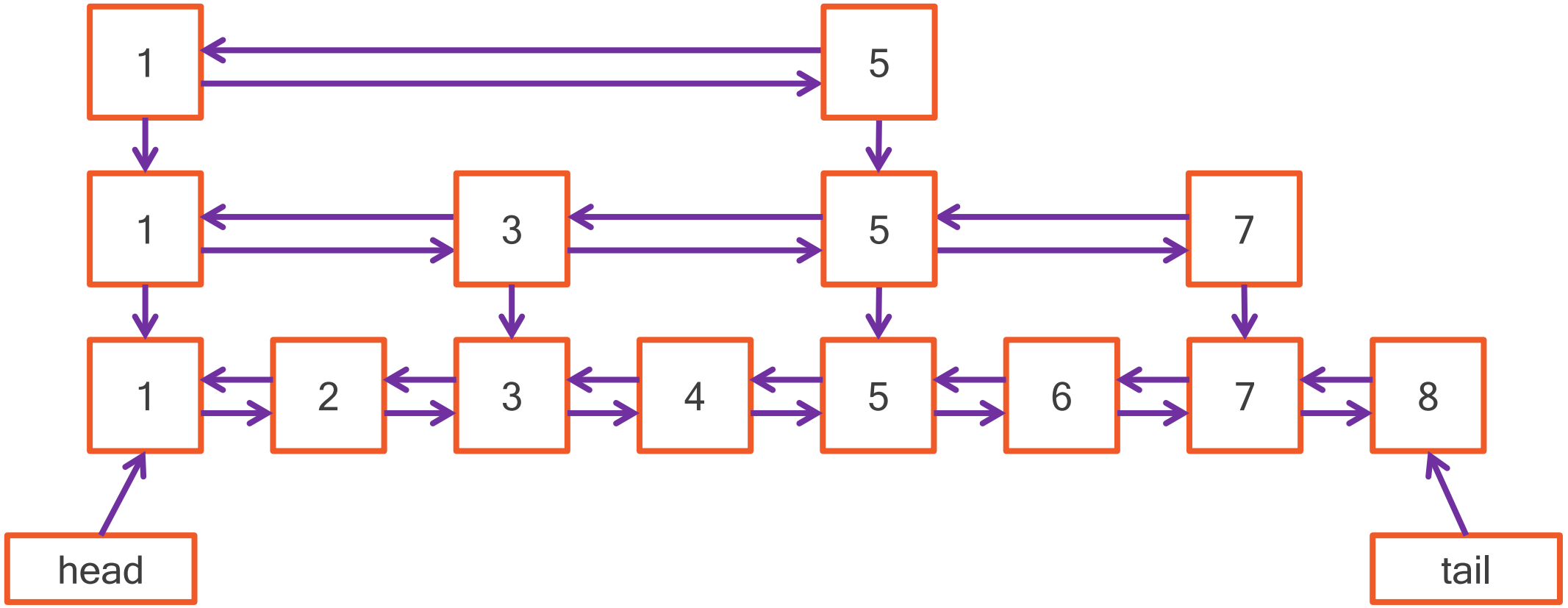
# Skip Lists

Let us create a first layer of fast access...



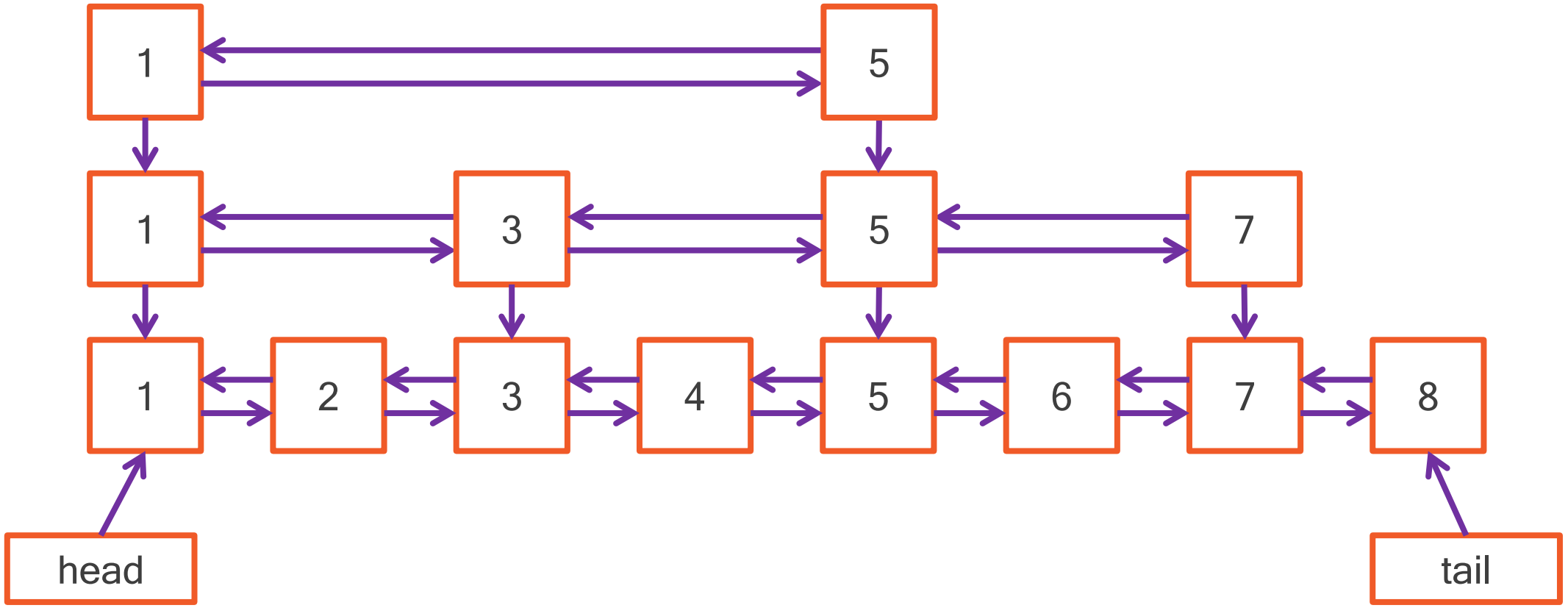
# Skip Lists

… and a second one



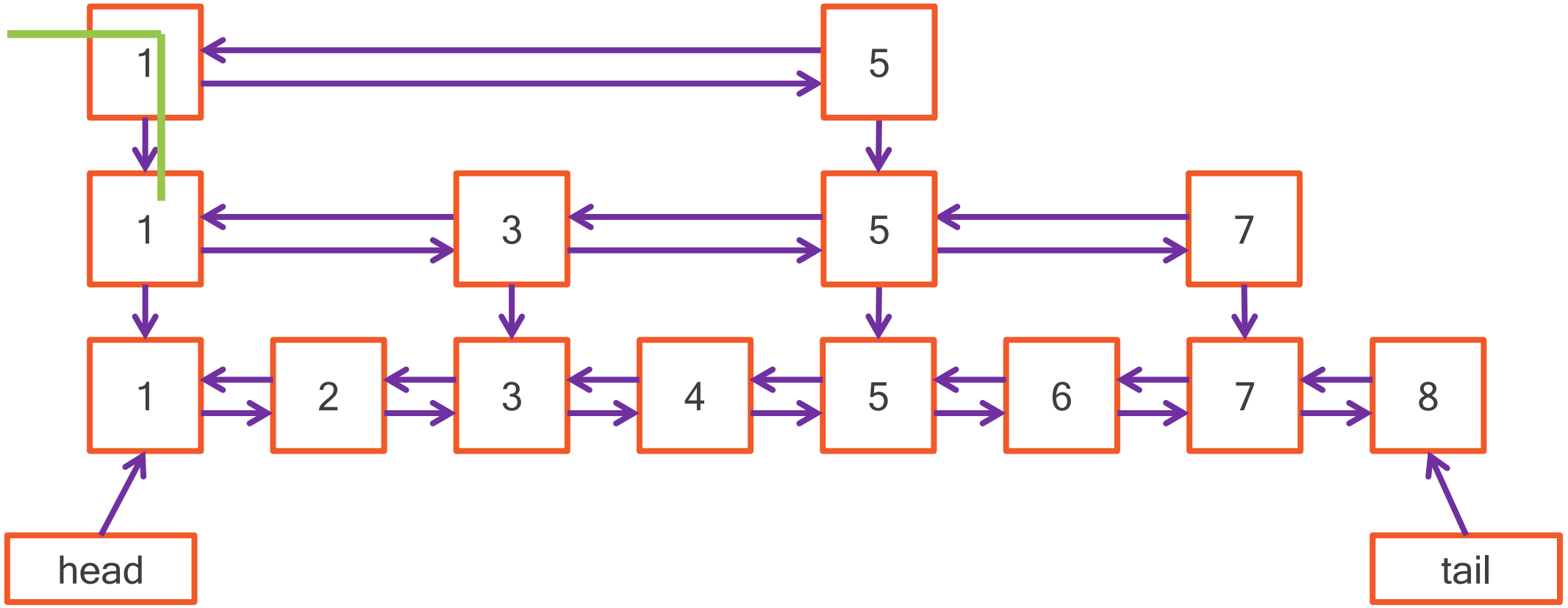
# Skip Lists

Now, suppose we need to locate 4



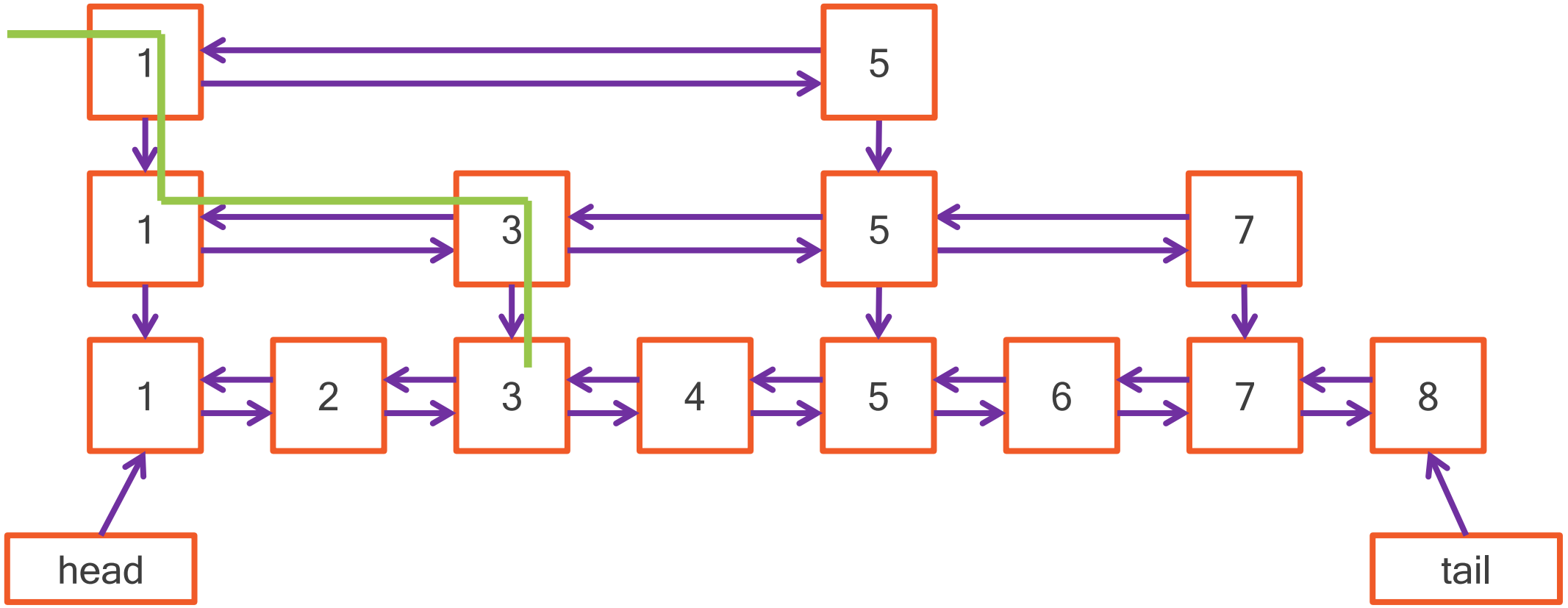
# Skip Lists

4 is between 1 and 5, so we go down one layer on 1



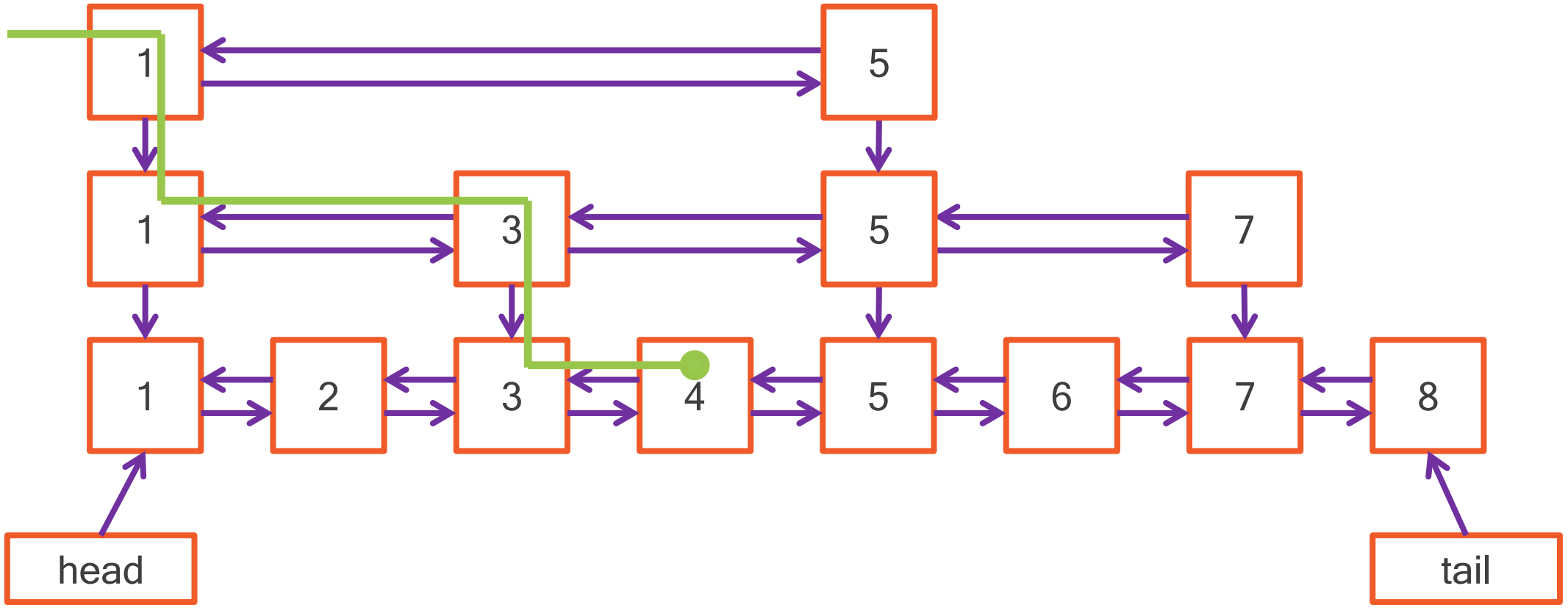
# Skip Lists

4 is between 3 and 5, so we go down one layer on 3



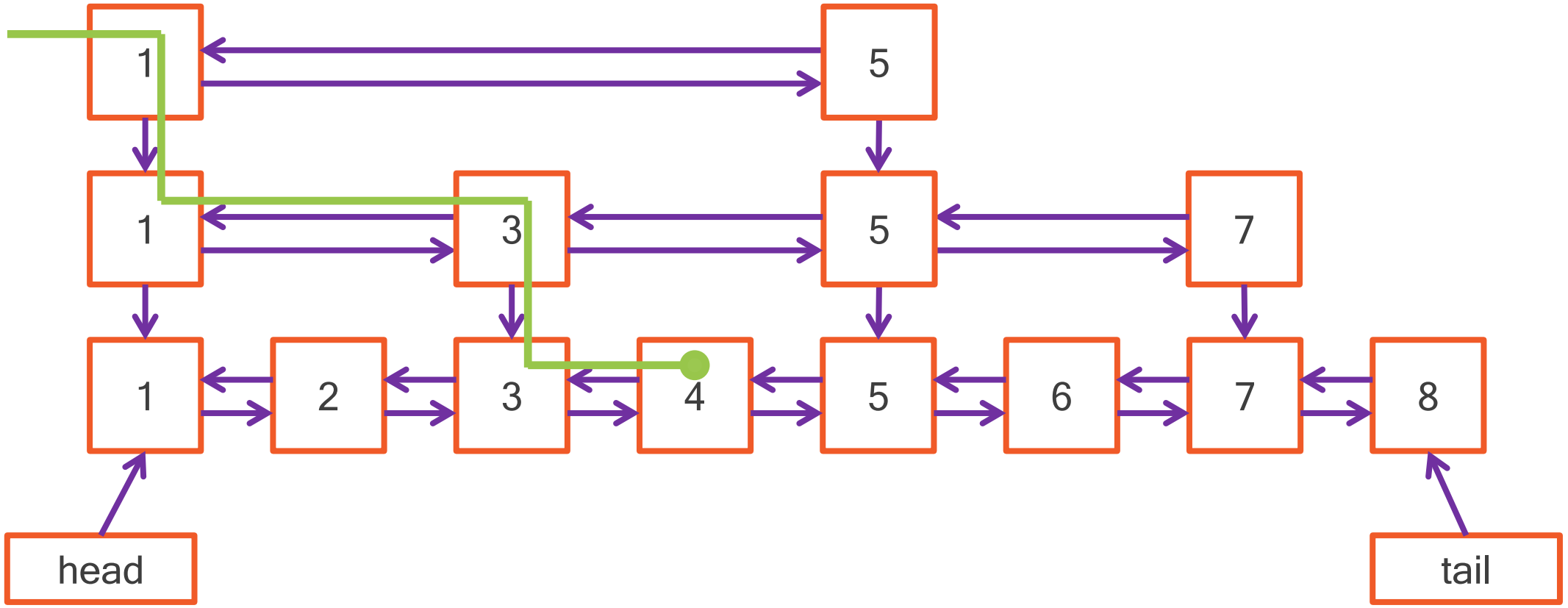
# Skip Lists

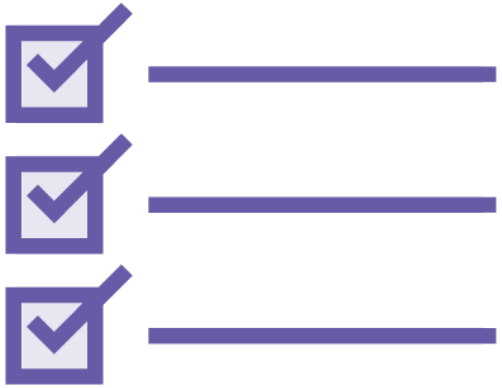
Then we can reach 4



# Skip Lists

The access time is now O(log(*N*))





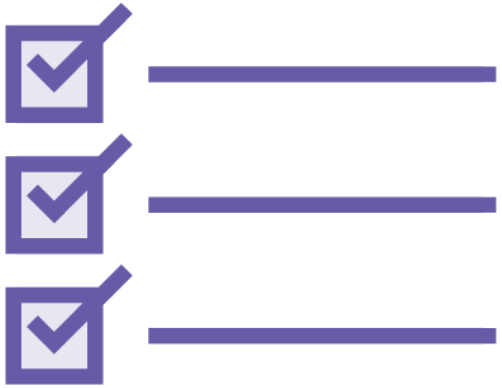
A skip list is used to implement a map

The keys are sorted

The skip list structure ensure fast access to any key

A skip list is not an array-based structure

And there are other ways than locking to guard it



The ConcurrentSkipListMap uses this structure

All the references are implemented with AtomicReference

So it is a thread-safe map with no synchronization

There is also a ConcurrentSkipListSet using the same structure

Tailored for high concurrency!

# Concurrent Skip Lists

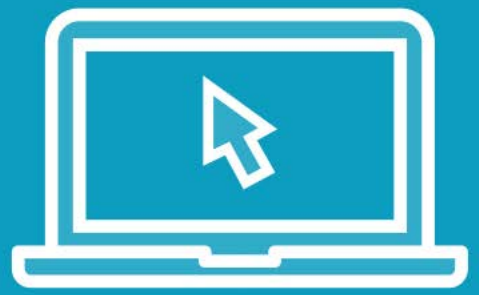
Used for maps and sets

Thread safety with no locking (synchronization)

Usable when concurrency is high

As usual: some methods should not be used (size())

# Demo



Let us see some code!

Let us implement a Consumer / Producer using an ArrayBlockingQueue

And see the ConcurrentHashMap from JDK8 in action

# Demo Wrapup

What did we see?

A producer / consumer implementation using concurrent queues

How to use the ConcurrentHashMap to look for information in a ~170k data set

# Module Wrapup

What did we learn?

We saw what the JDK has to offer as concurrent collections and maps

They can be used to solve concurrent problems while delegating thread safety to the API

We focused on blocking queues and concurrent maps

# Module Wrapup

Which structure for which case?

If you have very few writes, use copy on write structures

If you have low concurrency, you can rely on synchronization

In high concurrency, skip lists are usable with many objects, or ConcurrentHashMap

High concurrency with few objects is always problematic

# Course Wrapup

Be careful when designing concurrent code :

1) be sure to have a good idea of what your problem is

2) concurrent programming is different from parallel processing

3) try to delegate to the API as much as you can

4) know the concurrent collections well, as they solve many problems

# Course Wrapup

## Thank you!

@JosePaumard

https://github.com/JosePaumard