# Leveraging Concurrent Collections to Simplify Application Design

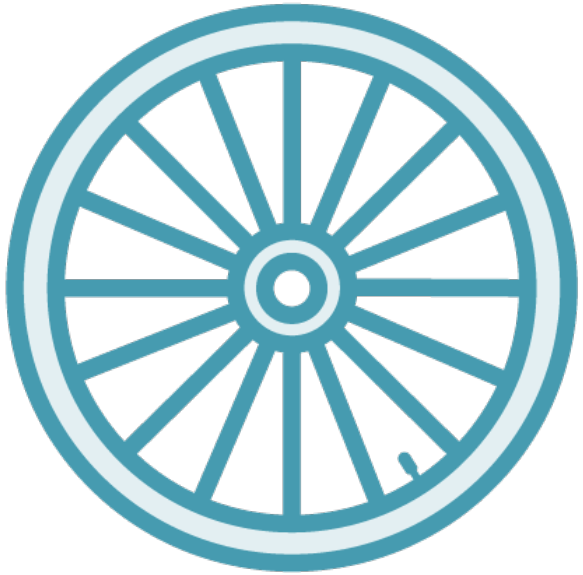
José Paumard
PHD, JAVA CHAMPION, JAVA ROCK STAR

@JosePaumard https://github.com/JosePaumard

# Agenda

- Concurrent collections and maps
- Collections: Queue, BlockingQueue
- Map: ConcurrentMap
- And implementations

# Concurrent Interfaces

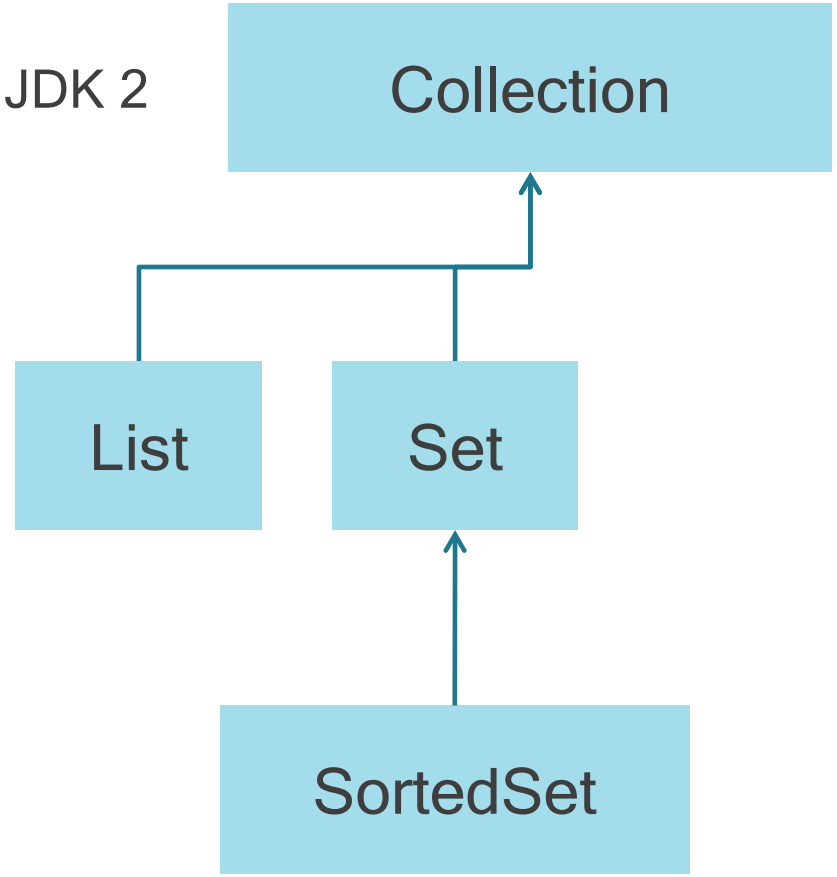


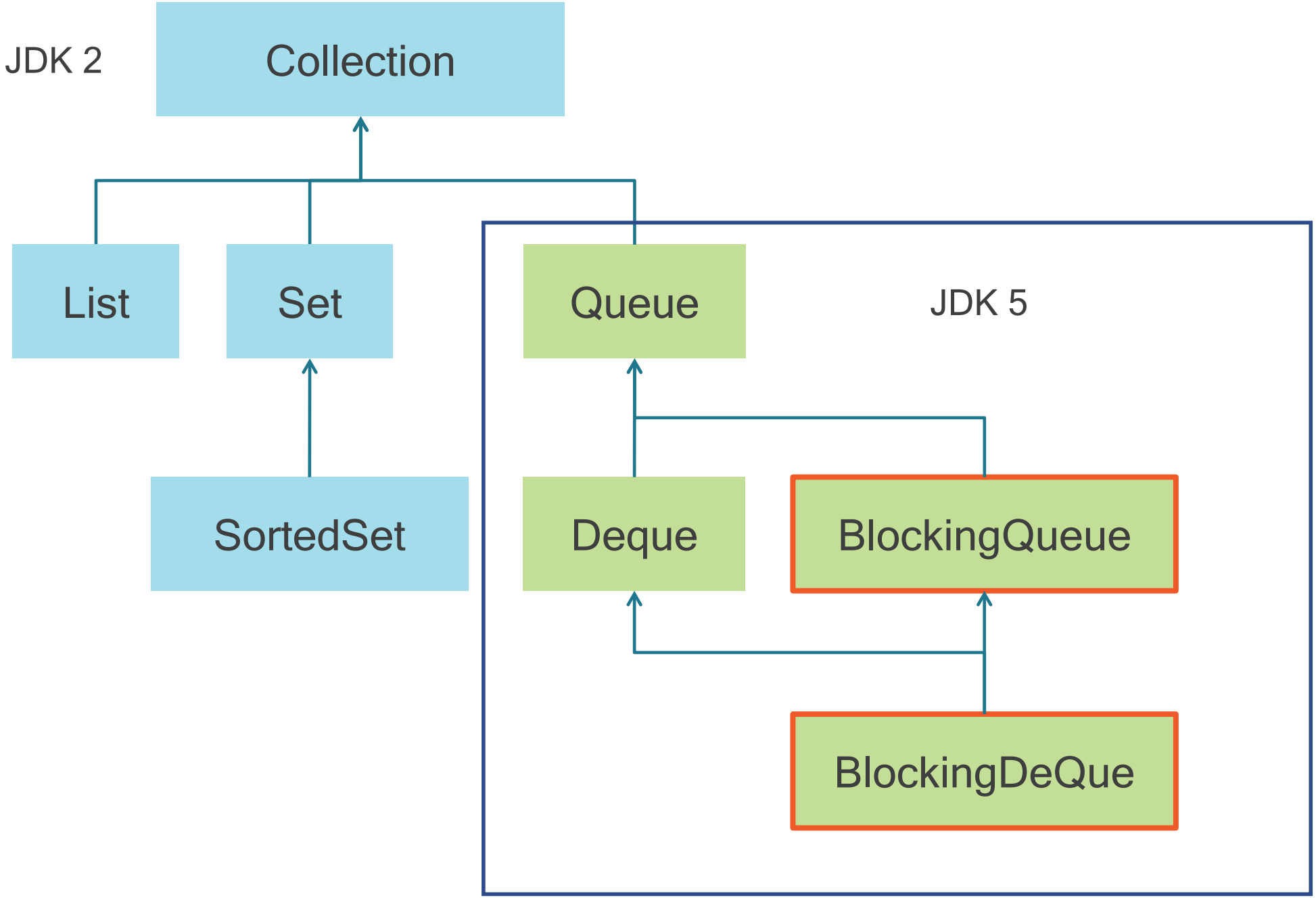
Implementing the Producer / Consumer at the API level vs the application level

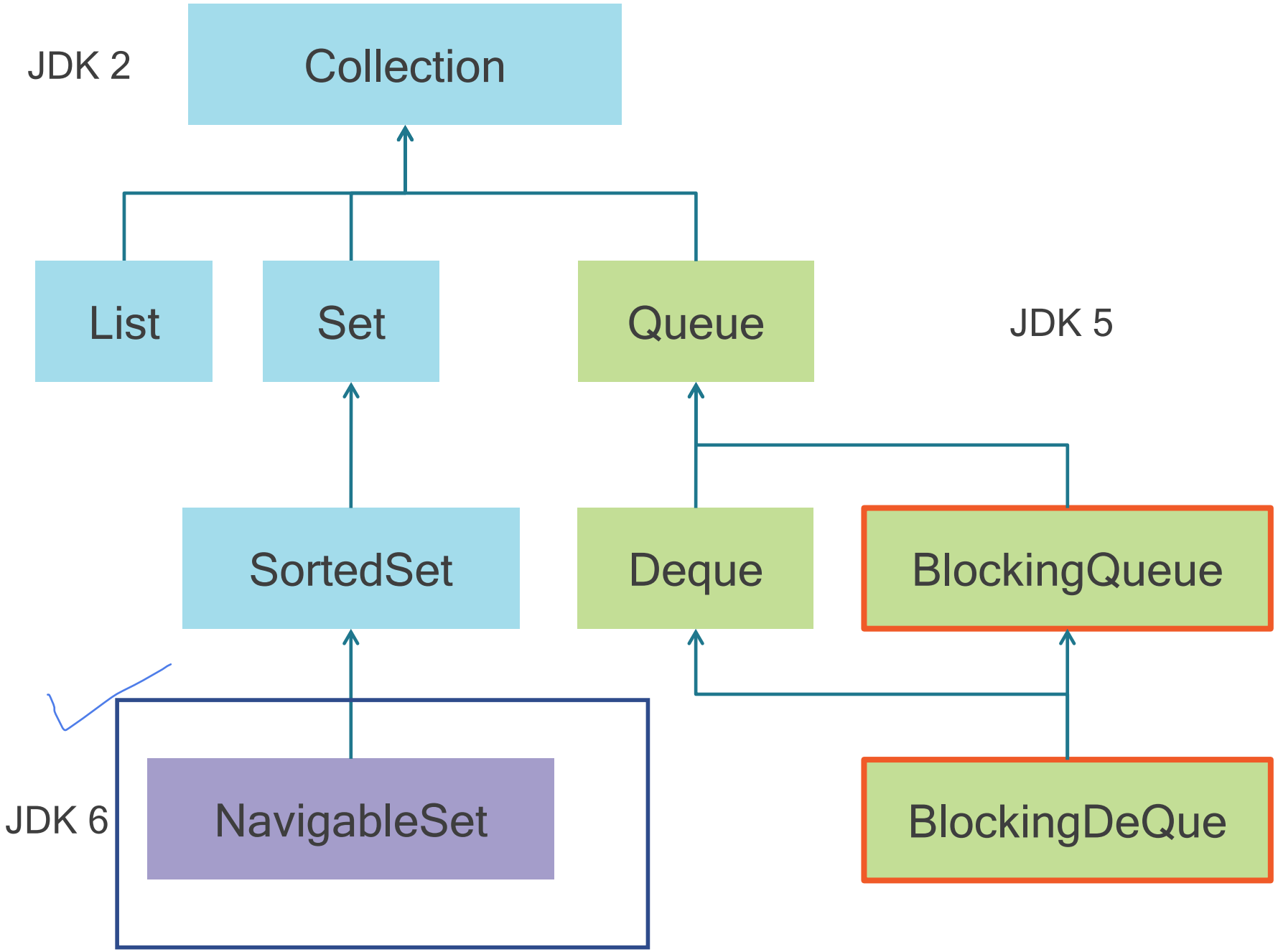
For that, we need new API, new Collections

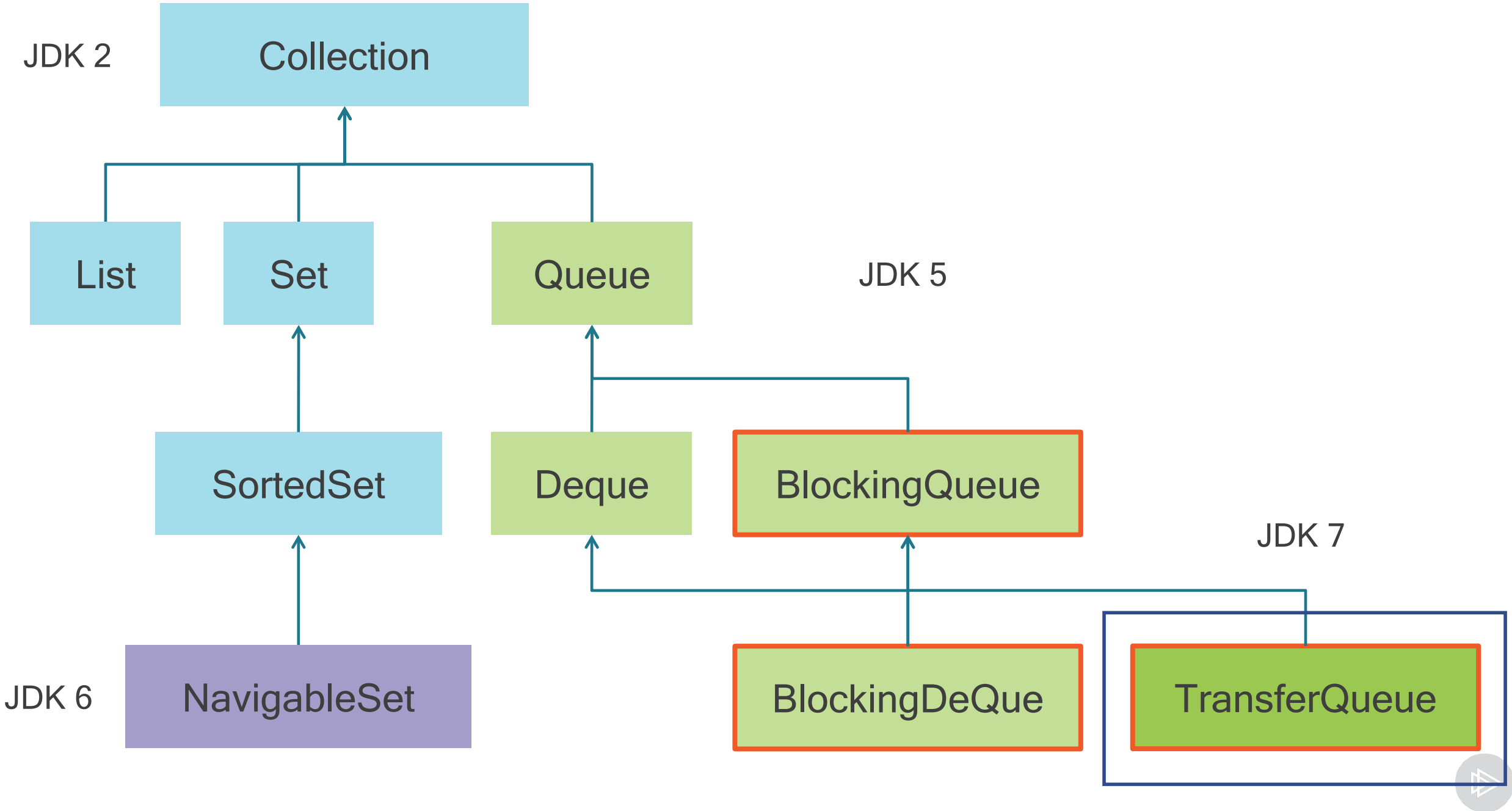
Two branches: Collection and Map

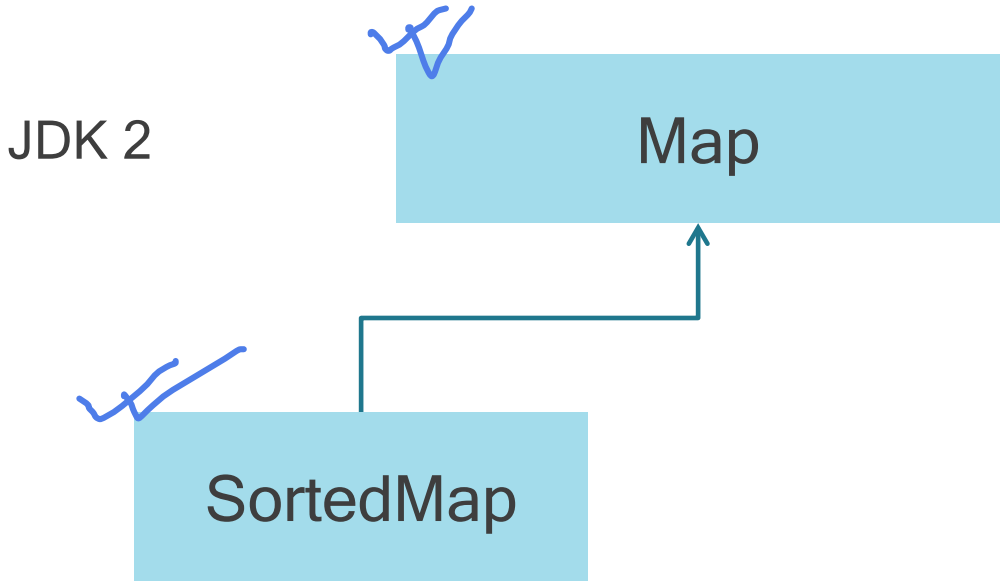
JDK 2

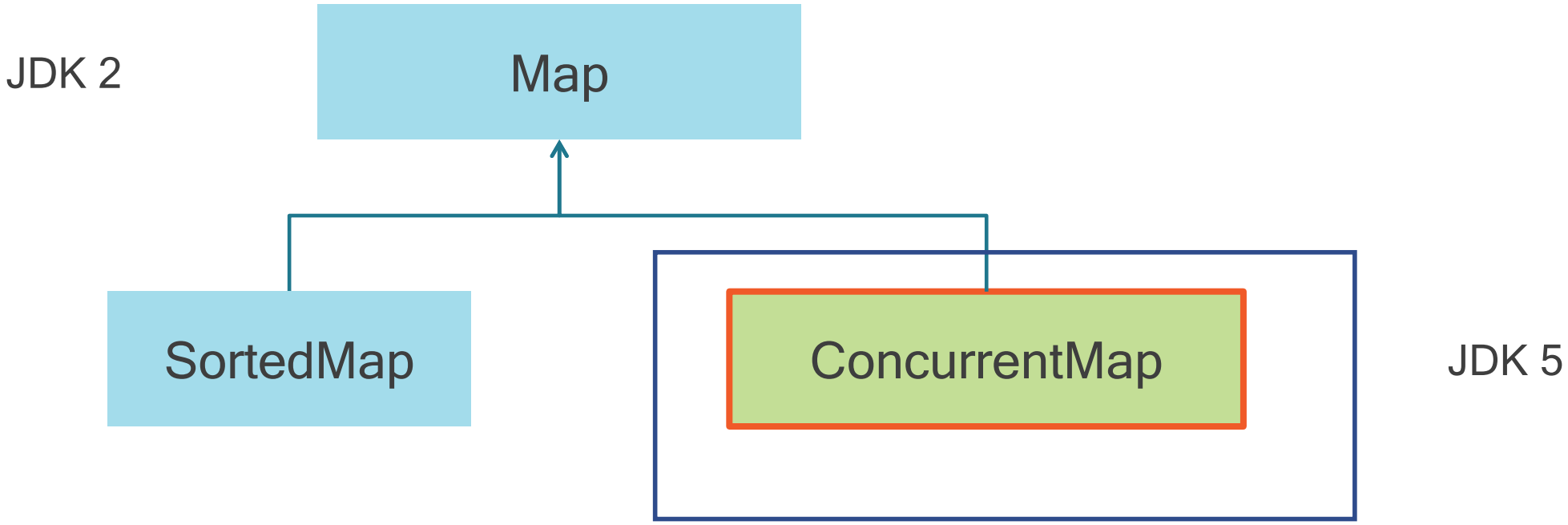


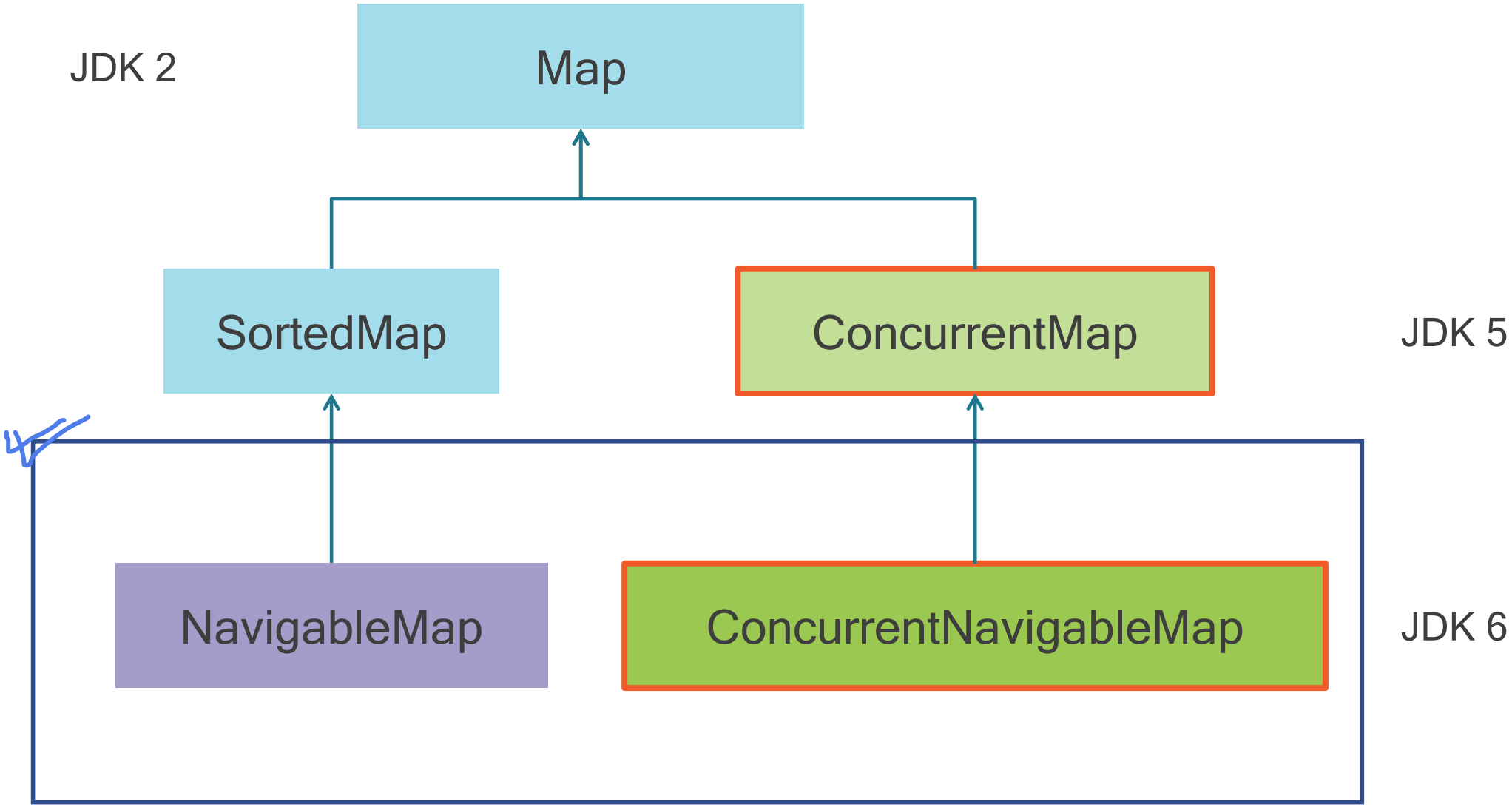

JDK 2
Collection
List
Set
SortedSet
Queue
JDK 5
Deque
BlockingQueue
BlockingDeQue

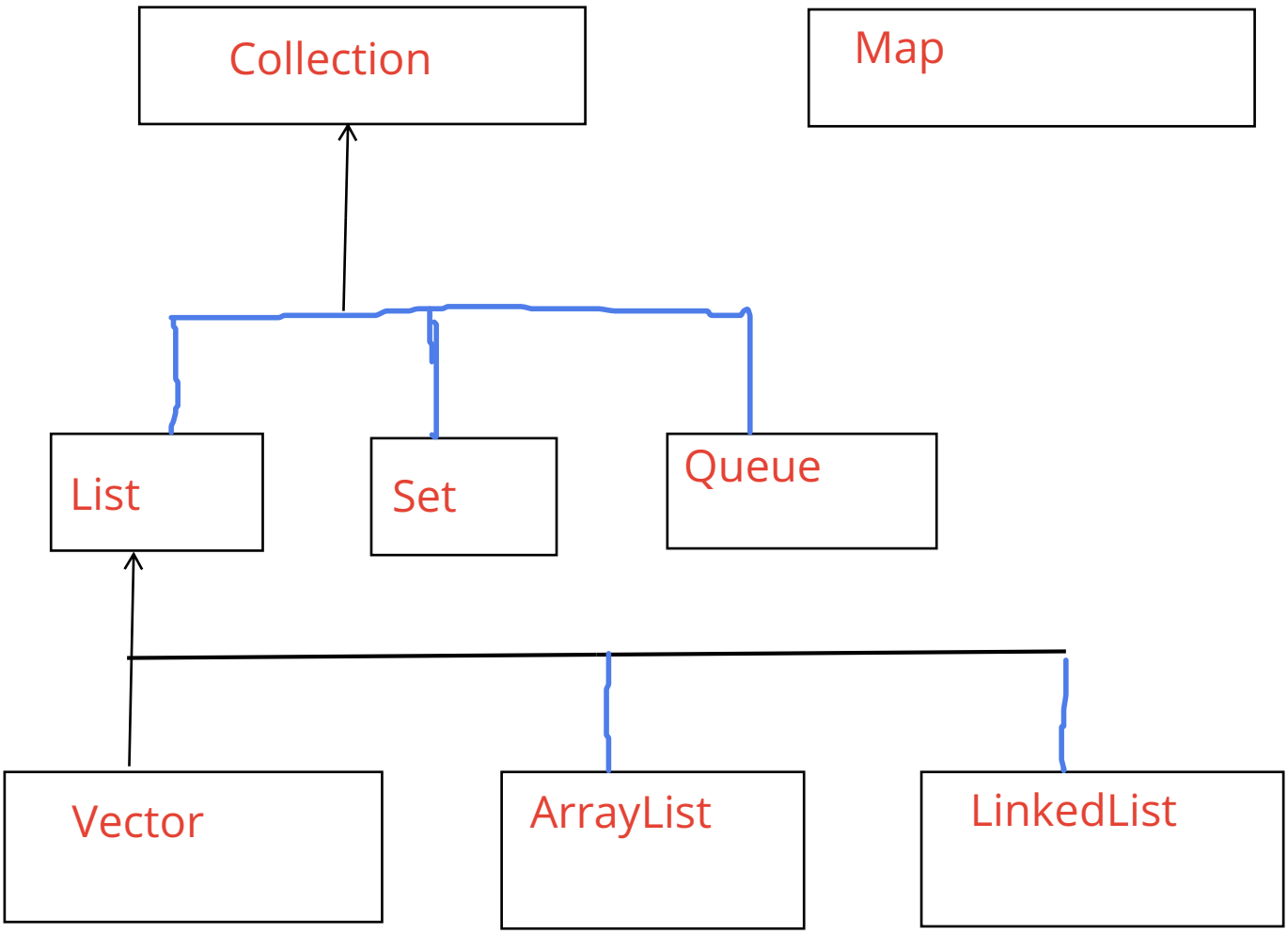

JDK 2
Collection
List
Set
Queue
JDK 5
SortedSet
Deque
BlockingQueue
JDK 6
NavigableSet
BlockingDeQue

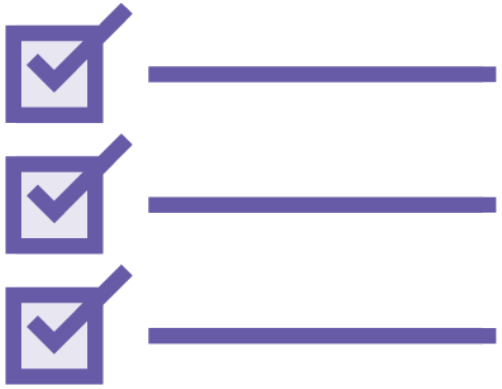

JDK 2
Collection
List
Set
Queue
JDK 5
SortedSet
Deque
BlockingQueue
JDK 7
JDK 6
NavigableSet
BlockingDeQue
TransferQueue


JDK 2
Map
SortedMap


JDK 2
Map
SortedMap
ConcurrentMap
JDK 5


JDK 2
Map
SortedMap
ConcurrentMap
JDK 5
NavigableMap
ConcurrentNavigableMap
JDK 6


Collection
Map
List
Set
Queue
Vector
ArrayList
LinkedList



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

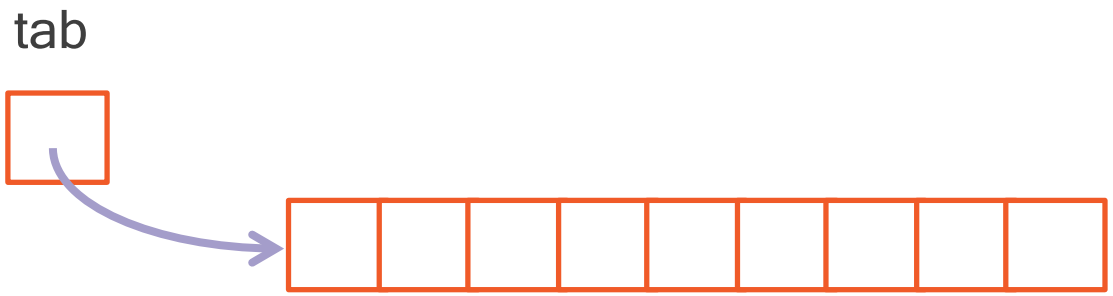
Exists only for List and Set

Exists for list and set

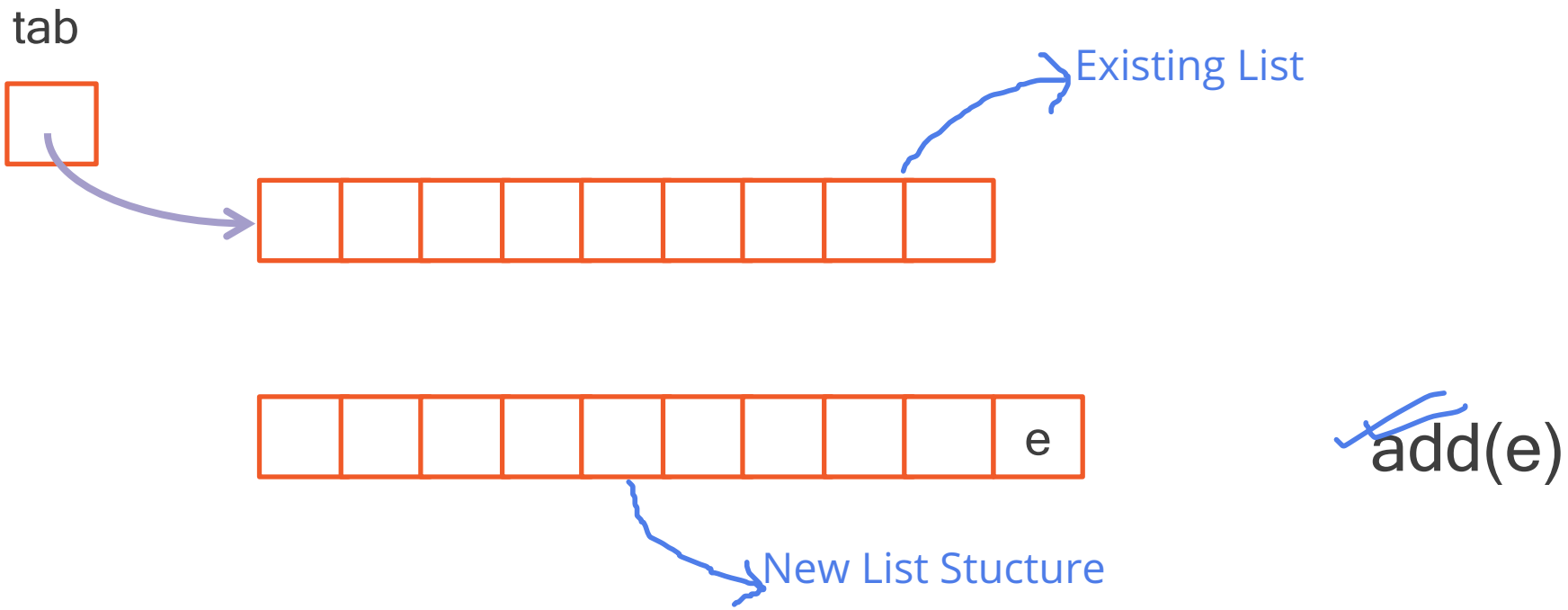
No locking for read operations

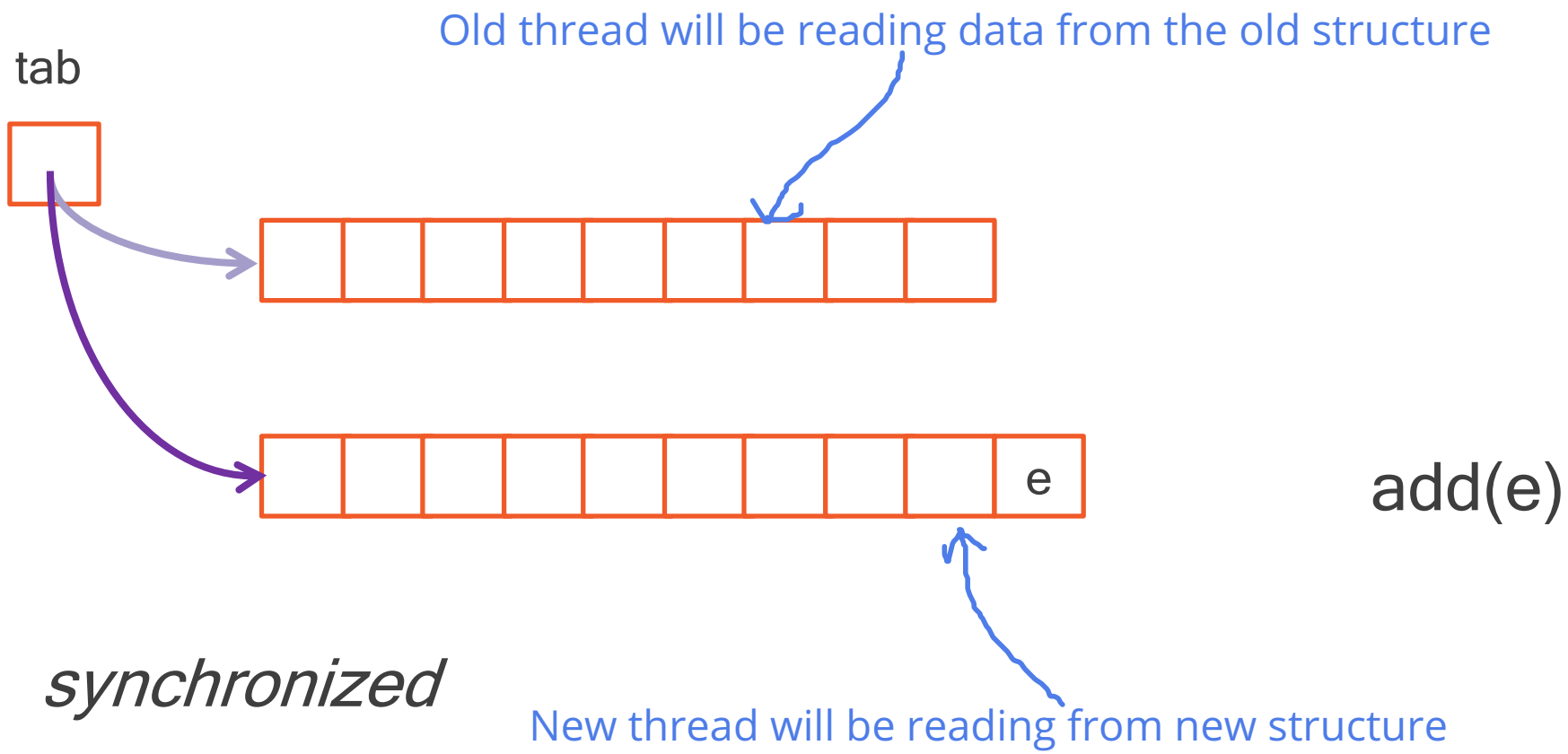
Write operations create a new structure

The new structure then replaces the previous one


tab

Use Case: Node e needs to be added at the end of the list




Old thread will be reading data from the old structure
tab
e
add(e)
synchronized
New thread will be reading from new structure

# 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

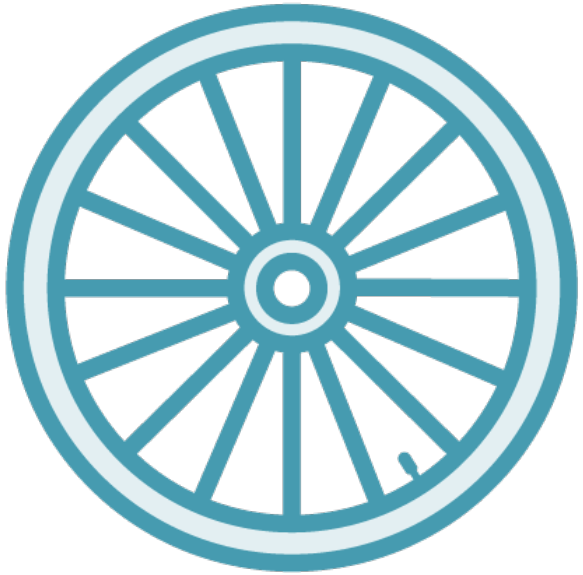
Two structures:

- 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

What is bounded and unbounded blocking Queue?

Array -> bounded as you need to mention the size of the array while initializing.

List -> List can be expanded dynamically at runtime. Hence it is unbounded.

Bounded means that we create a blocking queue with a certain amount of cells, a certain size of the array, and once this queue is full, it does not extend itself.

# How Does a Queue Work?

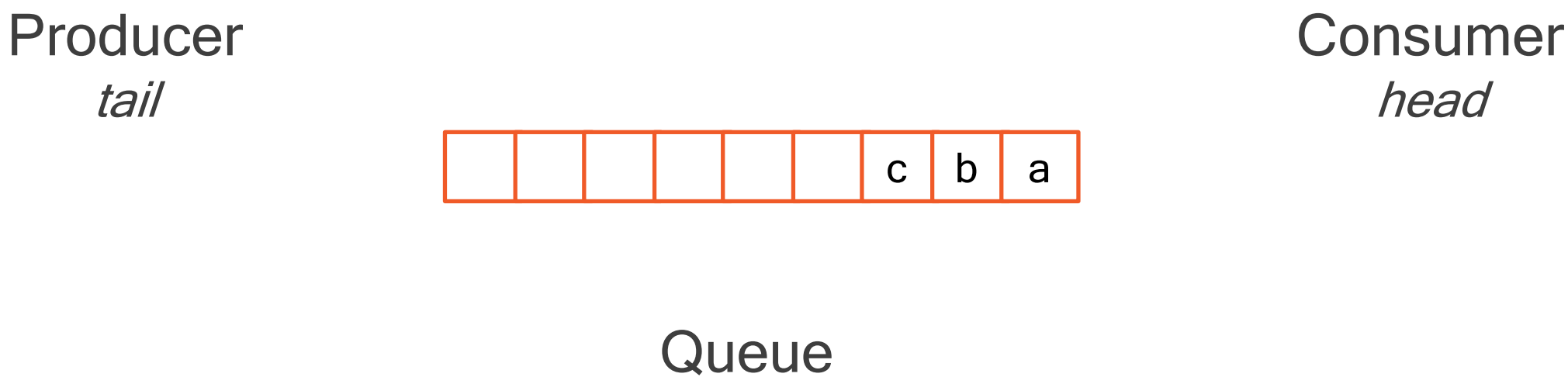
Two kinds of queues: FIFO (queue) and LIFO (stack)

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)

Suppose our queue is built on an array. The producer will add element from the tail and the consumer will consume them from the head. So we have several elements in our queue. Those elements are going to be consumed one by one by the consumer. But we are in a concurrent world. We can have as many producers and as many consumers as we need, and of course, each of them is in its own thread, so our queue or deque will be accessed in a concurrent way.




Of course, a thread does not know in advance how many elements there are in the queue or in a stack and querying a concurrent structure for the number of elements it has is not such a good idea because between the time we query that and the time where we use this information, this information might have changed dramatically. So this raises two questions. First, what happens if the queue or the stack is full and we need to add an element to it, and if you remember what we did to implement that using the lock interface, you might remember that we called the wait or the await method on the thread. And second question, what happens if the queue or the stack is empty and we need to get an element from it.

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);
```

Applicable for Blocking Queue

# 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

So in a nutshell, for the addition of an element at the Tail of the Queue, we have two behaviors. We can fail with an exception, we can fail by returning false, and if the queue is a blocking queue, we can block 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()

Deque can accept elements at the head of a queue with the method addFirst and offerFirst. And in a case of a BlockingDeque, putFirst, and the add offer and put behaviors are the same as the behaviors that we just saw that is fail with an exception, fail by returning false, and block until a cell is available. But we are not quite done. We also have the get and peek operation. Get is to remove an element from a queue and peek is just to examine an element without removing it from the queue. So for the queue poll and peek return null. Remove and element will throw exceptions.

And for the BlockingQueue, take will block until an element is available. And for the deque, returns null for pollLast and peekLast, exception for removeLast and getLast, and in a case of a BlockingDeque, takeLast will block until an element is available.

# 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()

1. Queue,
2. BlockingQueue,
3. Deque, and
4. BlockingDeque

# 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

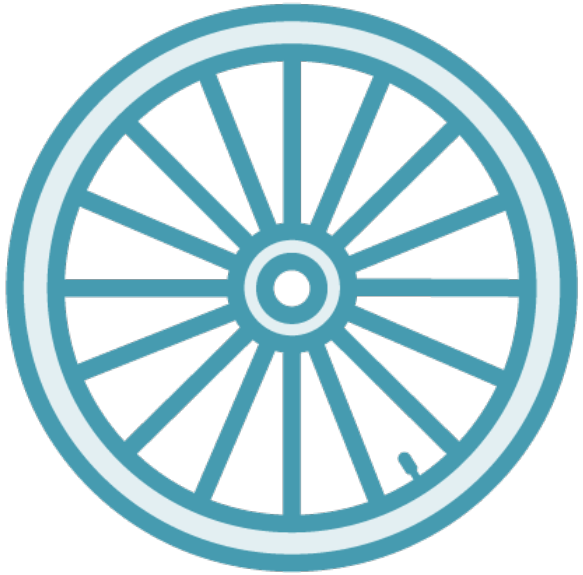
Let us see now the concept of concurrent maps. In fact, we have only one interface, ConcurrentMap, which is an extension of the map interface. The object of this ConcurrentMap is not to add any new method to the map interface, but merely to redefine the JavaDoc of those methods. We have two implementations of ConcurrentMap. The first is the ConcurrentHashMap. There is one available up to JDK 7, and in JDK 8, it has been completely we return. We are going to see that. And the ConcurrentSkipListMap introduced in Java 6, which does not rely on any synchronization. We are also going to see this structure in details.

## Atomic Operations Defined by the ConcurrentMap Interface

Besides being thread safe, ConcurrentMap also defines atomic operations. The first one is the putIfAbsent that takes a key and a value and that will add this key value pair to the map if the key is not already present in the map. This putIfAbsent is atomic in the sense that between the instant where the presence of the key is checked in the map and the adding of this key value pair, no other thread can interrupt this method. The same goes for the version of remove that takes a key and a value. This key value pair will be removed from the map if it is present. It is in fact the equivalent of remove if present. There is no interruption possible between the instant where the thread checks for the presence of the key in the map associated with the right value and the instance where the remove is performed. And then we have two last methods, the first one is replace that takes a key and value and that will replace the value currently associated with that key with this new value and the replace key, existingValue, and newValue that will replace existing value by new value if existingValue is already associated with that key in this map. Those two replace methods cannot be interrupted between the checking of the presence of the key in the map and the replacement of the value.

## Understanding Concurrency for a HashMap

Now let us take a closer look at the implementations and let us try to understand what is at stake with this implementation. We need implementations that are thread safe, efficient up to a certain number of threads, and we are going to see that this is a key point in dealing with maps, and a number of efficient, parallel special operations, and this is offered by the ConcurrentHashMap from Java 8. How does a hashmap work internally? This is important to understand to understand also concurrency. In the JDK, a hashmap is built on an array. When I want to add the key value pair to this array, first I compute a hashcode from the key. This hashcode will decide which cell of this array will hold the key value pair, and when this is done, a pointer will point from this cell to this value pair. So if I have two key value pairs in my hashmap, this structure will be the following. Note that each cell is called a bucket and a bucket can hold several key value pairs since different keys may have the same hashcode. So adding a key value pair to a map is a several steps problem. First, we need to compute the hashcode of the key. Second, we need to determine which cell of the array will hold this key value pair and we first need to check the bucket has been created or not. If it's not, then we create it and add the key value pair to it, pretty straightforward. If there is already a bucket, we need to check if this bucket is already holding this same key or not. If it is the case, then the value we try to add will replace the existing value. If it is not, there is a special process that is launched. In the hashmap, it will create a link list up to a certain number of key value pairs and will switch to a red-black tree past this number. This is the full step, the updating of the map. In a concurrent world, all these steps must not be interrupted by any other thread because if it is the case, it will just corrupt the map by corrupting either the bucket, either the linked list, either the red-black tree.



One interface:

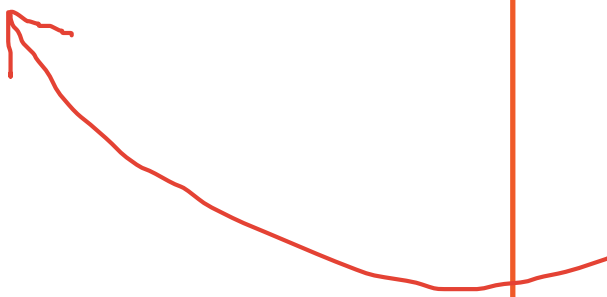
ConcurrentMap: redefining the JavaDoc

Two implementations:

ConcurrentHashMap: JDK 7 & JDK 8

ConcurrentSkipListMap: JDK 6, no synchronization

The first one is the putIfAbsent that takes a key and a value and that will add this key value pair to the map if the key is not already present in the map. This putIfAbsent is atomic in the sense that between the instant where the presence of the key is checked in the map and the adding of this key value pair, no other thread can interrupt this method



# Atomic Operations

ConcurrentMap defines atomic operations:

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

Remove takes a key and a value. This key value pair will be removed from the map if it is present. It is in fact the equivalent of remove if present. There is no interruption possible between the instant where the thread checks for the presence of the key in the map associated with the right value and the instance where the remove is performed.

replace takes a key and value and that will replace the value currently associated with that key with this new value

replace existing value by new value if existingValue is already associated with that key in this map. Those two replace methods cannot be interrupted between the checking of the presence of the key in the map and the replacement of the value.

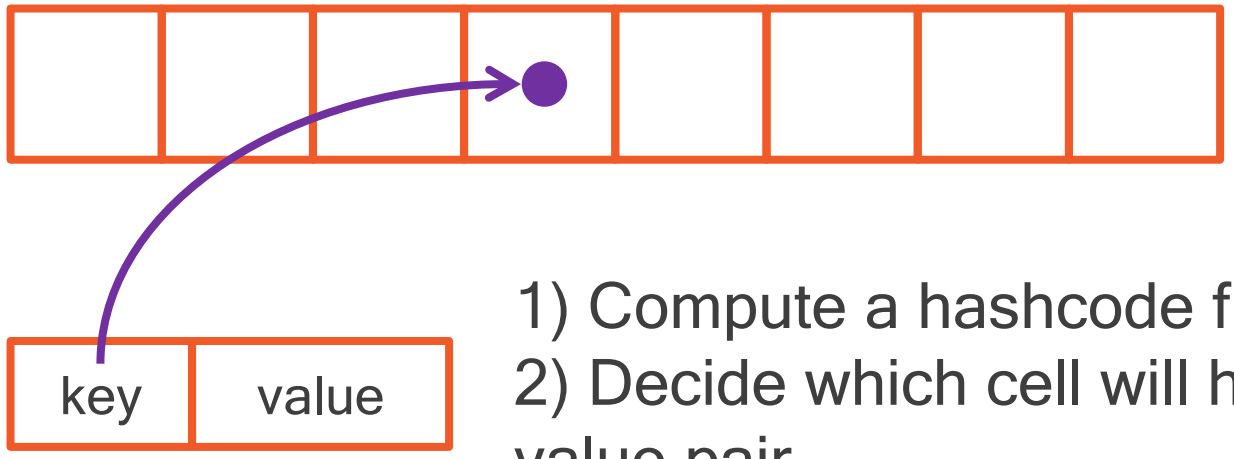
# ConcurrentMap Implementations

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

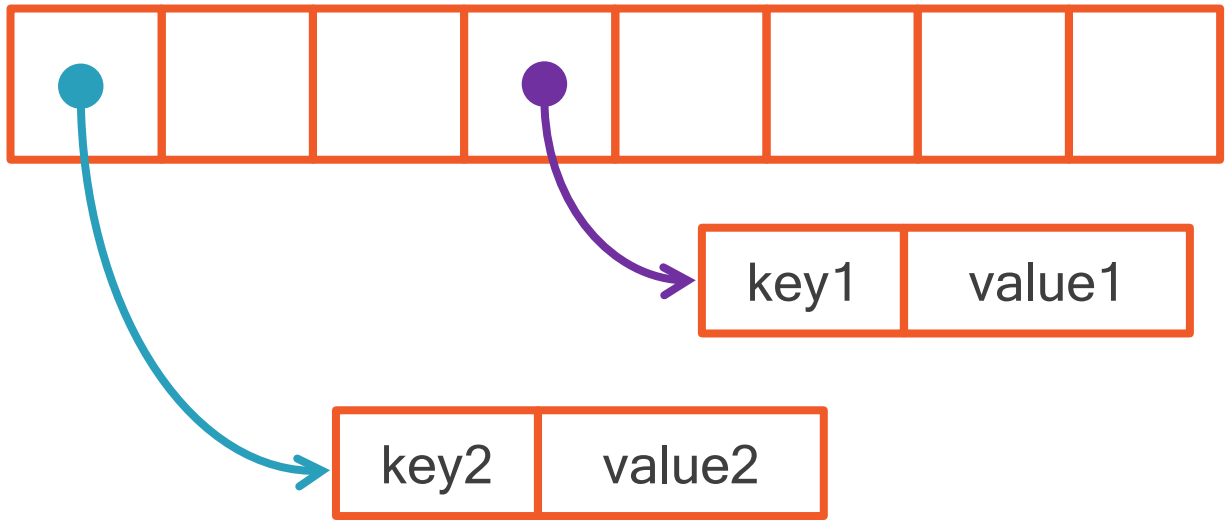



1) Compute a hashcode from the key
2) Decide which cell will hold the key / value pair

In the JDK, a hashmap is built on an array. When I want to add the key value pair to this array, first I compute a hashcode from the key. This hashcode will decide which cell of this array will hold the key value pair, and when this is done, a pointer will point from this cell to this value pair.

# How Does a HashMap Work?

A hashmap is built on an array




Each cell is called a “bucket”

So if I have two key value pairs in my hashmap, this structure will be the following. Note that each cell is called a bucket and a bucket can hold several key value pairs since different keys may have the same hashcode.

So adding a key value pair to a map is a several steps problem. First, we need to compute the hashcode of the key. Second, we need to determine which cell of the array will hold this key value pair and we first need to check the bucket has been created or not. If it's not, then we create it and add the key value pair to it, pretty straightforward.

# Understanding the Problem

If there is already a bucket, we need to check if this bucket is already holding this same key or not. If it is the case, then the value we try to add will replace the existing value. If it is not, there is a special process that is launched. In the hashmap, it will create a link list up to a certain number of key value pairs and will switch to a red-black tree past this number. This is the full step, the updating of the map. In a concurrent world, all these steps must not be interrupted by any other thread because if it is the case, it will just corrupt the map by corrupting either the bucket, either the linked list, either the red-black tree.

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...

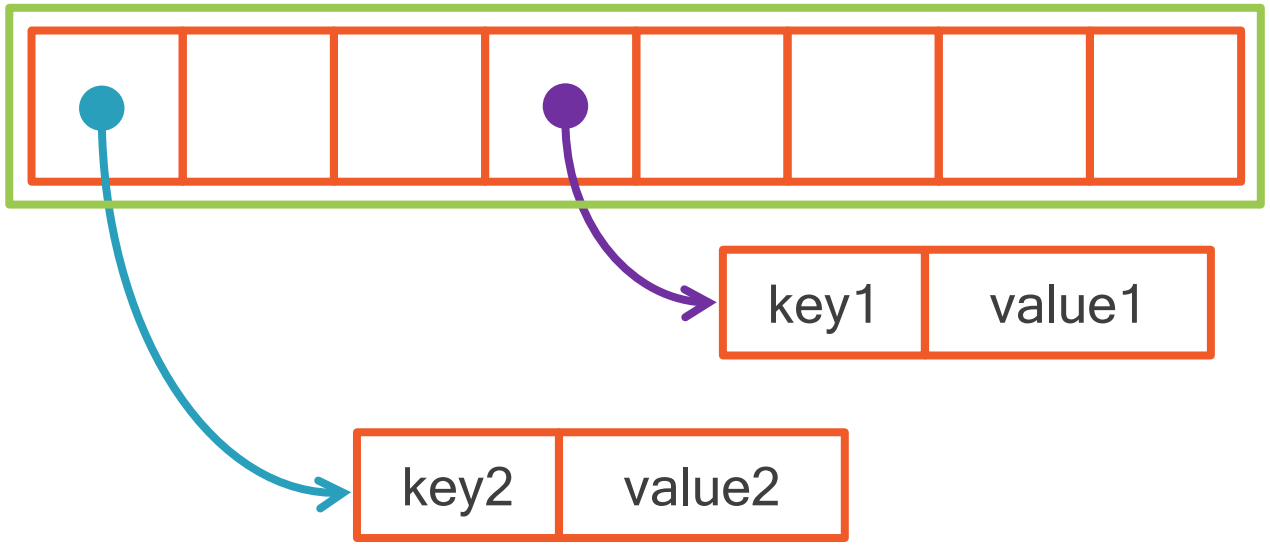
It would be very inefficient to block all the map:

- we should allow several threads on different buckets

- we should allow concurrent reads

# Synchronizing All the Map

Synchronizing on the array itself

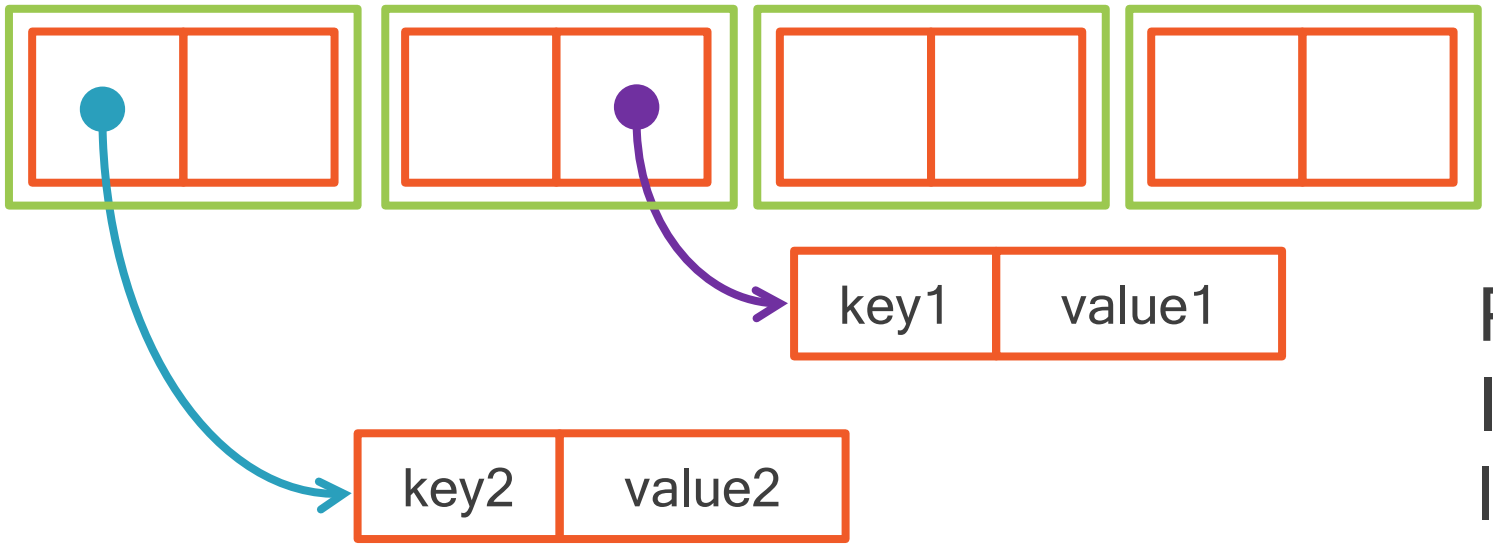



Pros: it works!

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

Insert text here

# 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

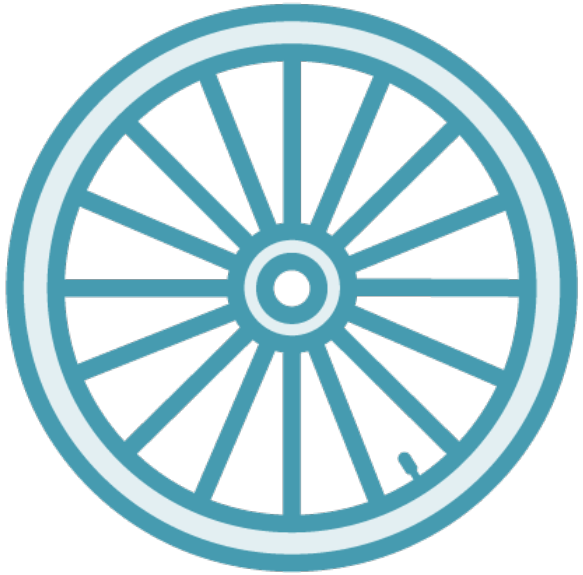
# ConcurrentHashMap JDK 8

## Creating a Concurrent Set on a Java 8 ConcurrentHashMap

We do not have concurrent hash set in the JDK, but we have a static factory method on the ConcurrentHashMap from JDK 8 called newKeySet that will create a set here, a set of string backed by this ConcurrentHashMap. So this ConcurrentHashMap can also be used as a concurrent set with the same kind of semantics. Note that the parallel operations of the ConcurrentHashMap are not available on this set. In fact, the implementation of this set is a public static class defined as a member of this ConcurrentHashMap with no parallel operations defined on it.

## Wrapping up the Java 8 ConcurrentHashMap

Let us do a quick wrap up on this ConcurrentHashMap. First, it is a fully concurrent map, which is nice. It has been made to handle very high concurrency and millions of key value pairs. It exposes built-in parallel operations, which is very nice. Those operations are not present in the map or concurrent map, and it can also be used to create very efficient and very large concurrent sets.



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
            );
```

Parallel Search on a Java 8 ConcurrentHashMap

The ConcurrentHashMap, for instance, has a method for parallel searching of key value pairs. The first parameter passed to this search method is called the parallelism threshold and the second one is the operation to be applied. If this operation returns a non-null value, this value will be returned by the search method and it will stop the exploration of the map.

# ConcurrentHashMap: Parallel Search

The first parameter is the parallelism threshold

The second is the operation to be applied

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

We also have other search methods, searchKeys, searchValues, and searchEntries. Now what is this parallelism threshold? It is the number of key value pairs in this map that will trigger a parallel search. In our case, if we have more than 10, 000 key value pairs in this map, this search will be conducted in parallel.

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

We also have a reduce method. The first bifunction maps each key value pair to an element that will be used for reduction, and the second bifunction, it is the reduction itself. It takes two elements returned by this mapping bifunction and reduce those two elements together. Of course, this second bifunction should be associative as in all reduction operation. And once again, this reduce method takes a parallel threshold, that is if we have more than 10, 000 key value pairs in this map, this reduction will be conducted in parallel.

# 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)
           );
```

And the last parallel method available on this ConcurrentHashMap in Java 8 is the forEach method. The first parameter is also a parallelism threshold, so here if we have more than 10, 000 key value pairs, this forEach operation will be computed in parallel. And the second and main parameter is a biconsumer that takes a key value pair and does something. This biconsumer is applied to all the key value pairs of the map.

# ConcurrentHashMap: Parallel for Each

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

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

Here, what does it do? The value is a list of string, so on all the values of the map, we will remove all the strings that are longer than 20 characters. We also have other versions of this forEach method, forEachKeys that takes a consumer of key, forEachValues that takes a consumer values, and forEachEntry that takes a consumer of entry, which is the object that model the key value pair.

We do not have concurrent hash set in the JDK, but we have a static factory method on the ConcurrentHashMap from JDK 8 called newKeySet that will create a set here, a set of string backed by this ConcurrentHashMap. So this ConcurrentHashMap can also be used as a concurrent set with the same kind of semantics. Note that the parallel operations of the ConcurrentHashMap are not available on this set. In fact, the implementation of this set is a public static class defined as a member of this ConcurrentHashMap with no parallel operations defined on it.

```
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

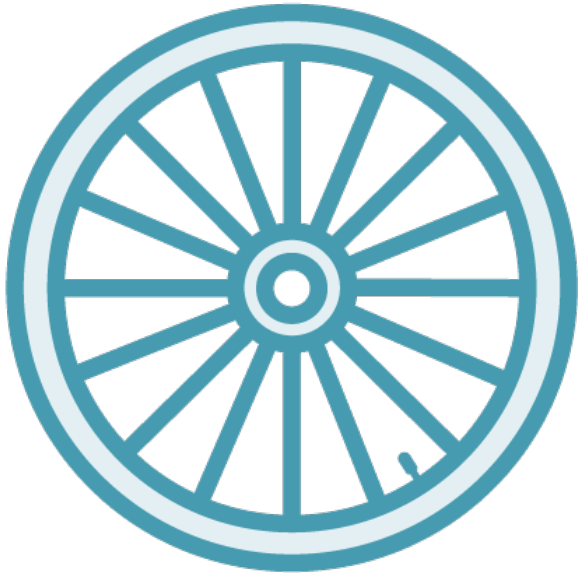
A fully concurrent map

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)

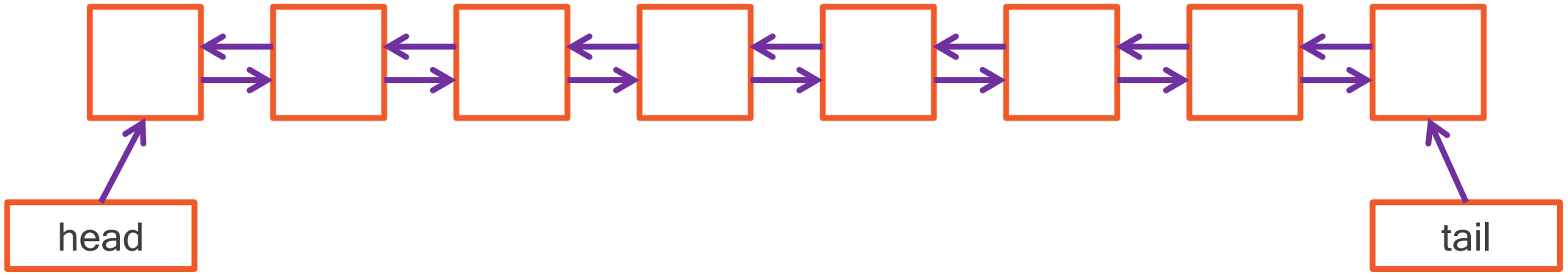
A skip list is a smart structure used to create linked lists

Relies on atomic reference operations, no synchronization

That can be used to create maps and sets

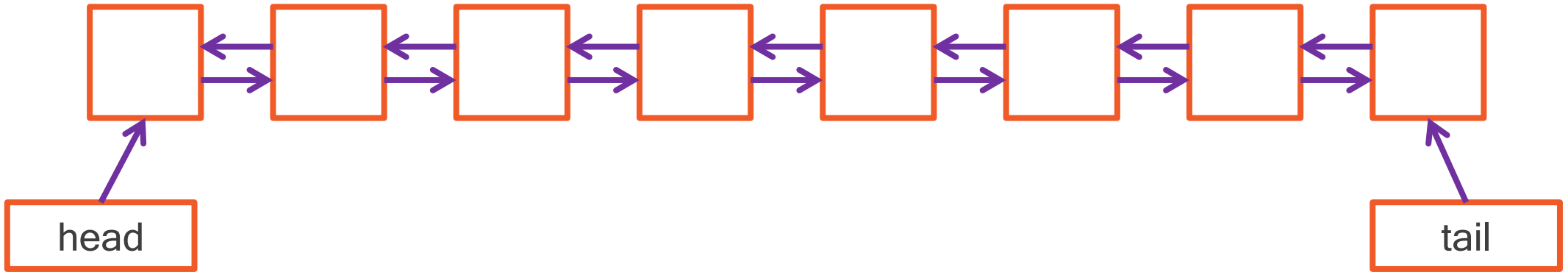
# Skip Lists

It starts with a classical linked list



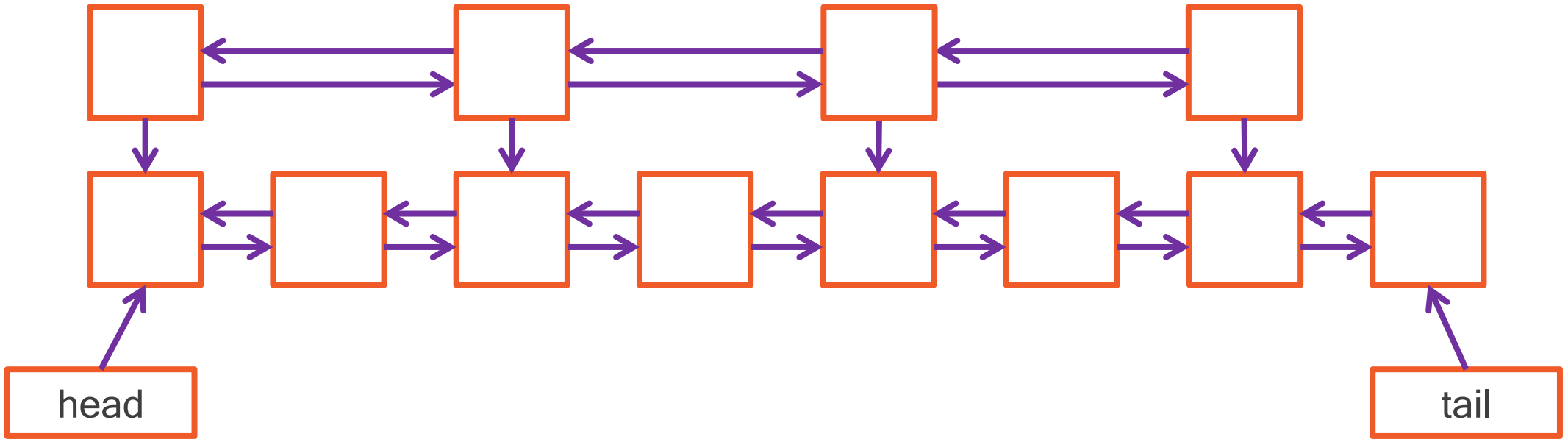
# Skip Lists

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



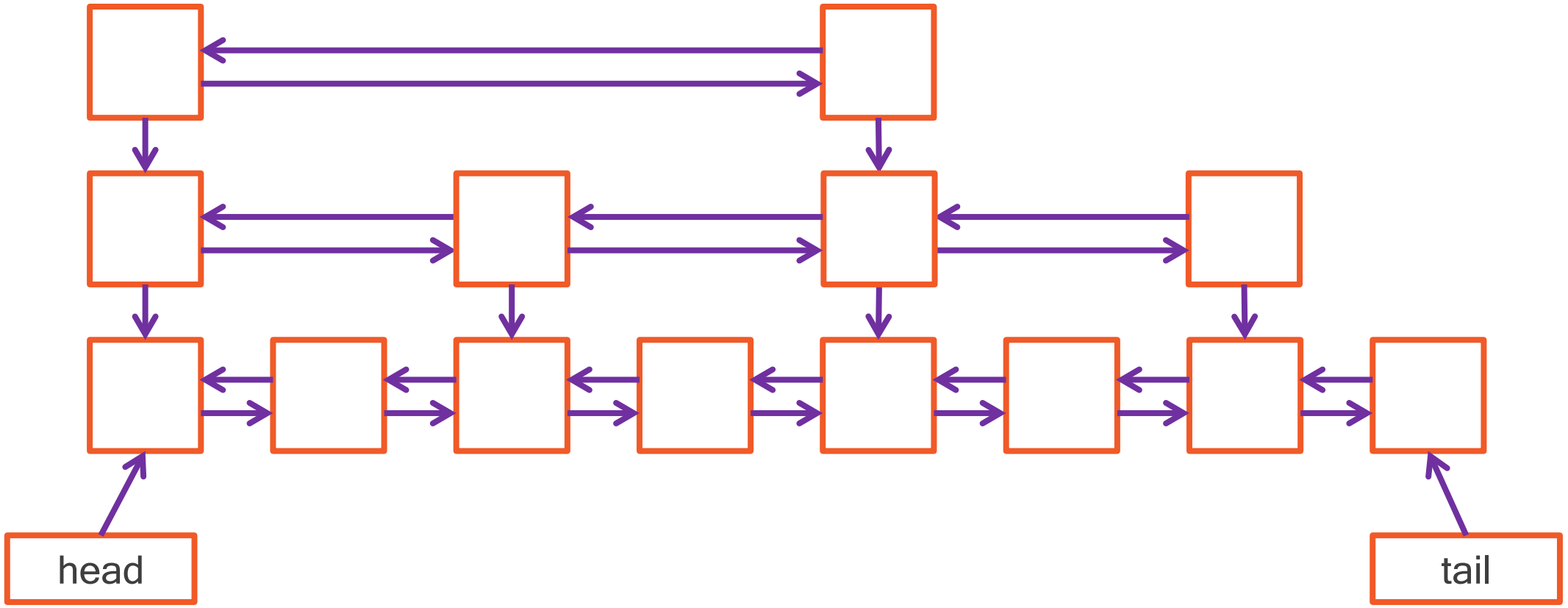
# Skip Lists

Solution: create a fast access list



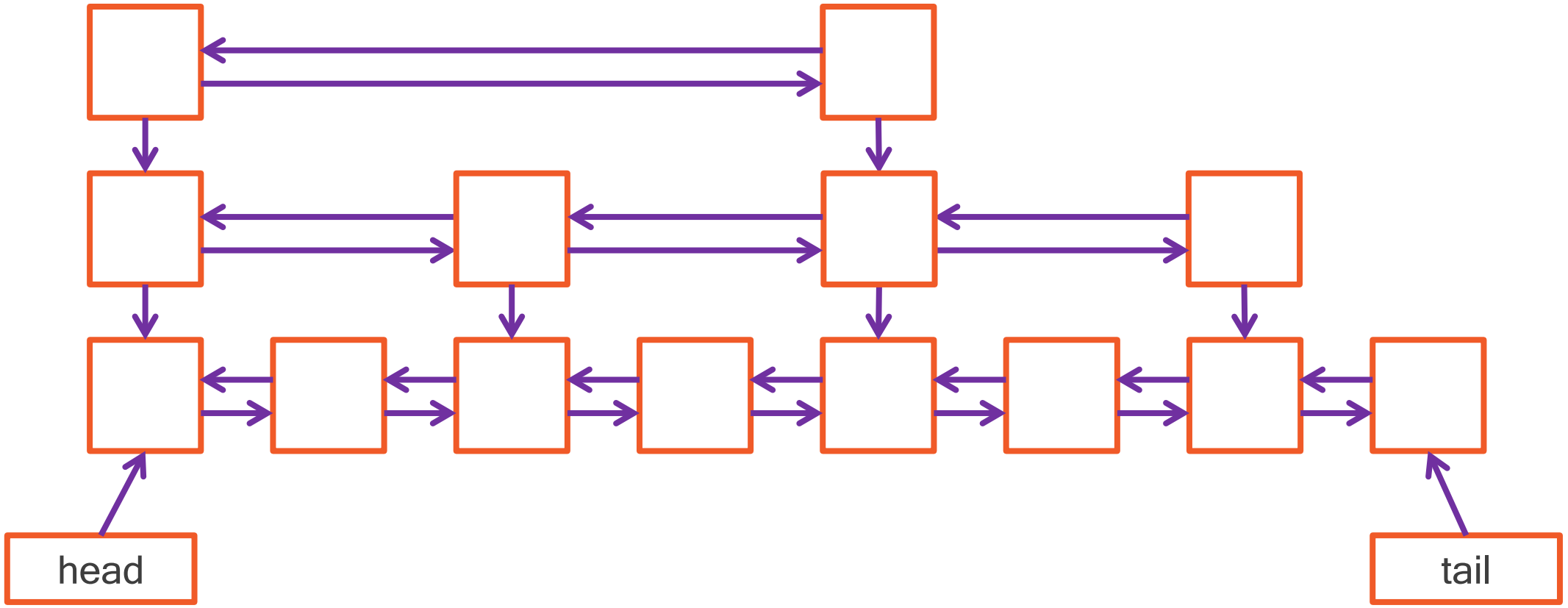
# Skip Lists

We can even create more than one



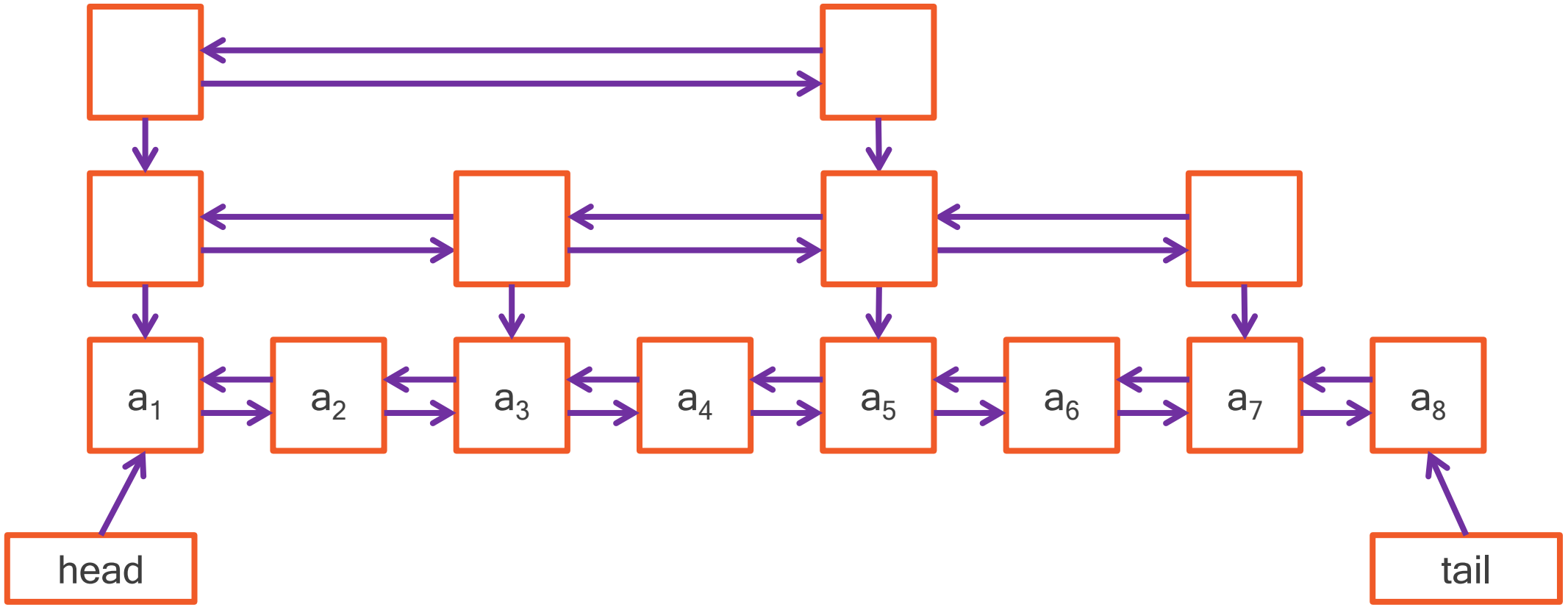
# Skip Lists

It assumes that the elements are sorted



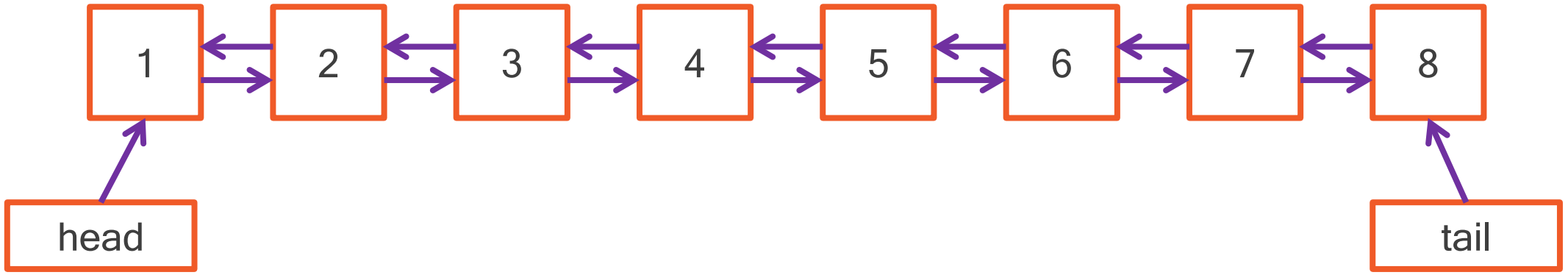
# Skip Lists

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



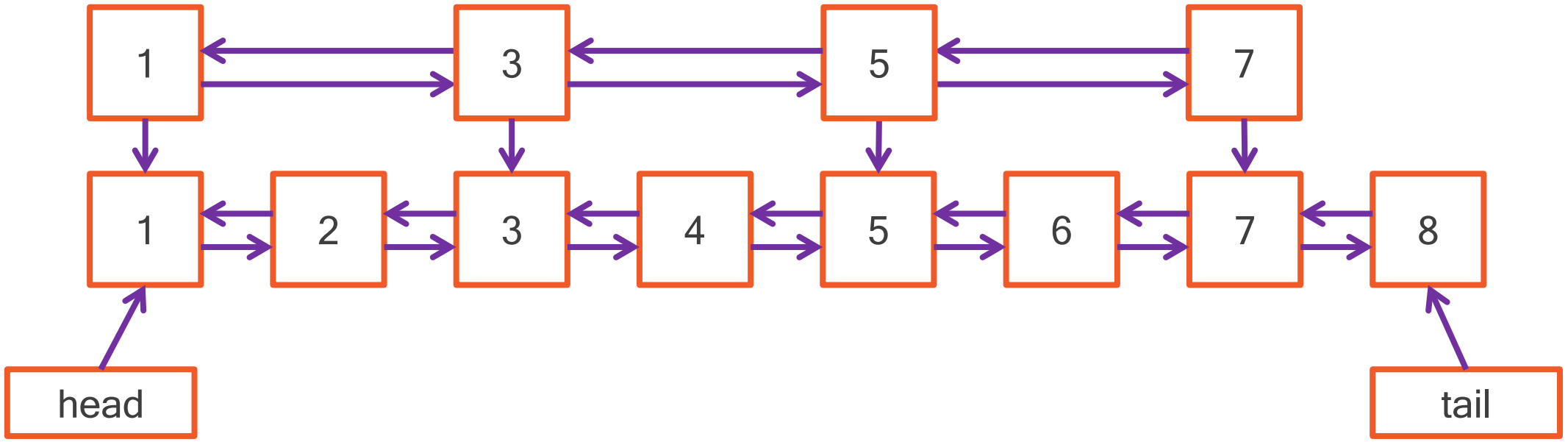
# Skip Lists

Let us see how it works



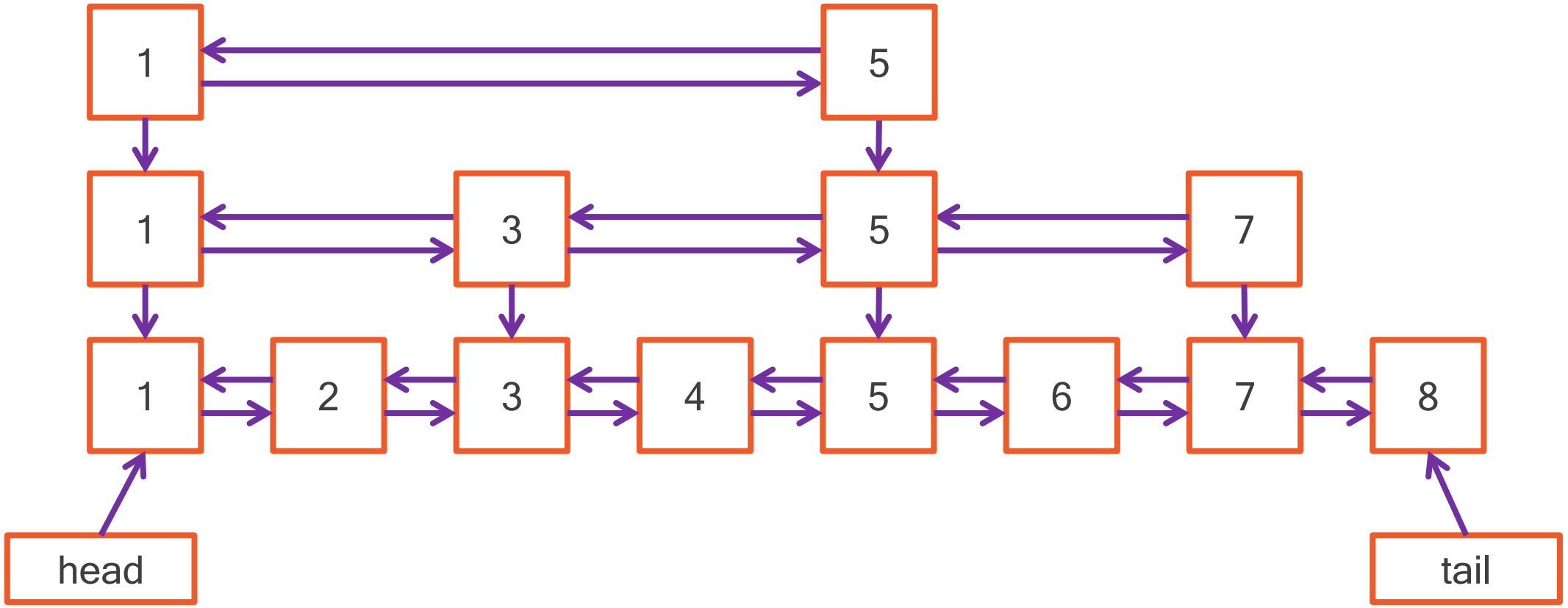
# Skip Lists

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



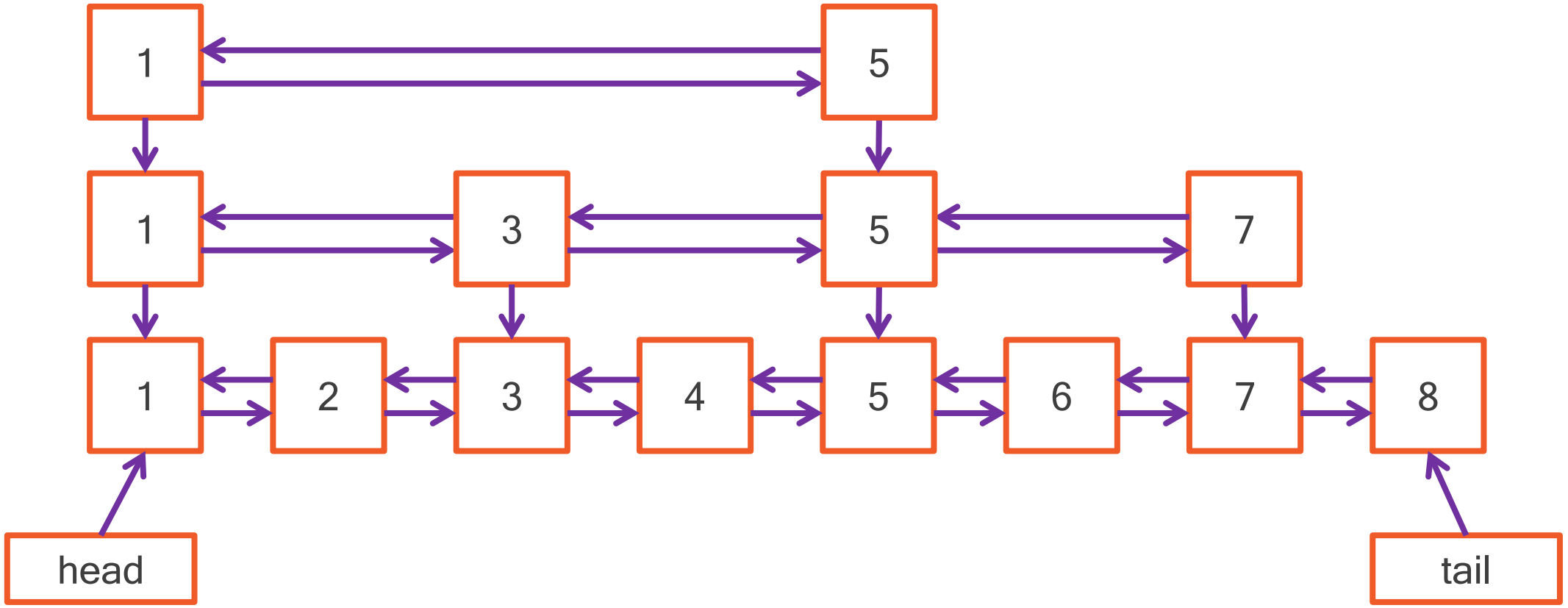
# Skip Lists

… and a second one



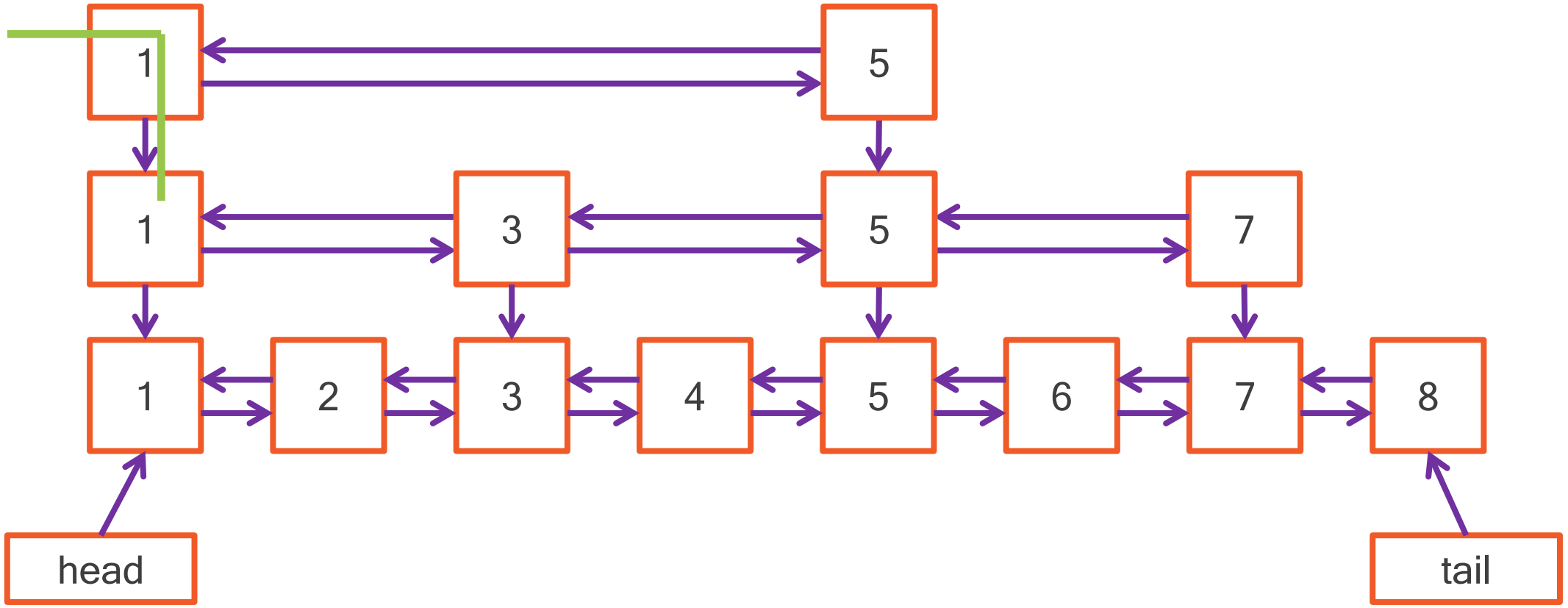
# Skip Lists

Now, suppose we need to locate 4



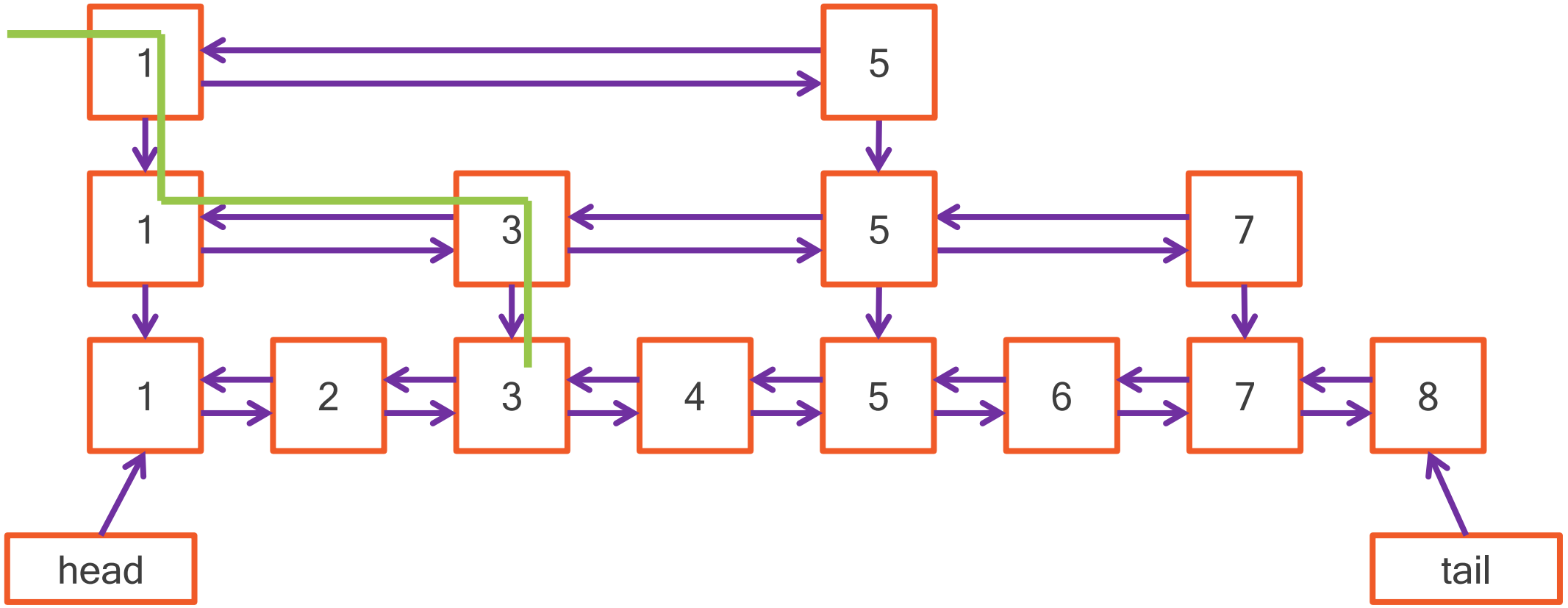
# Skip Lists

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



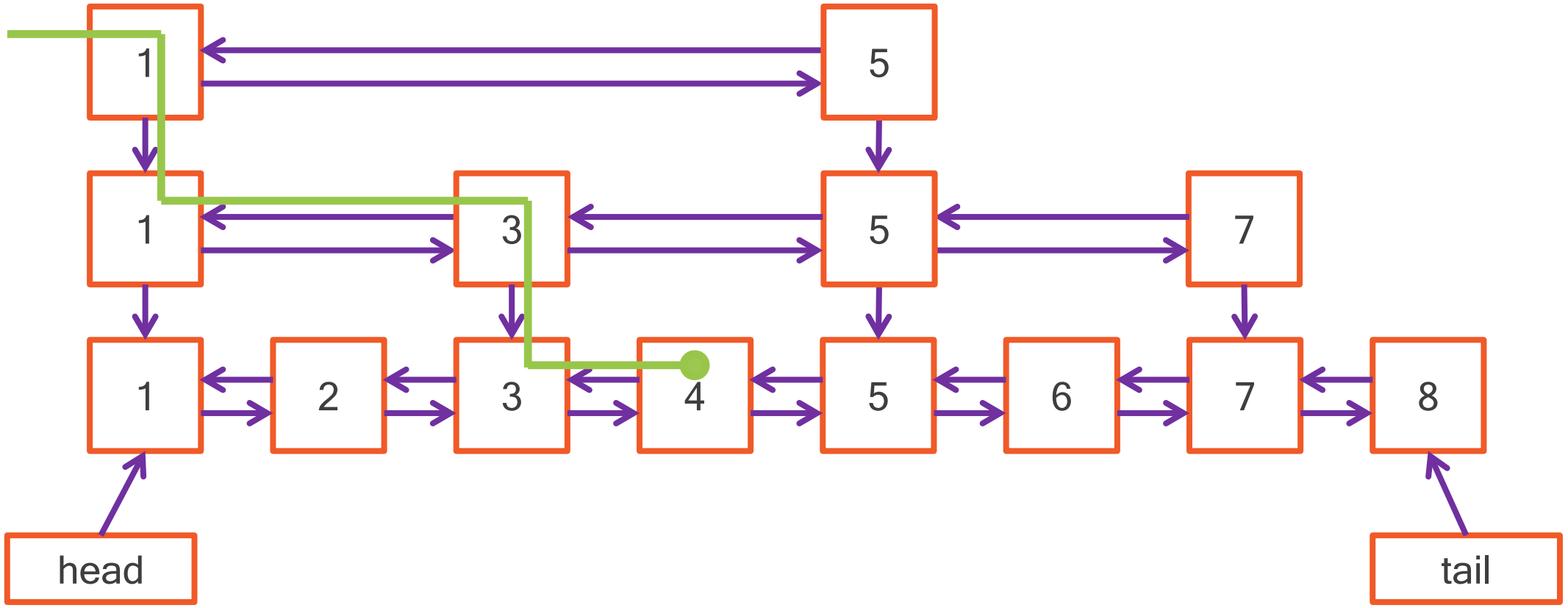
# Skip Lists

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



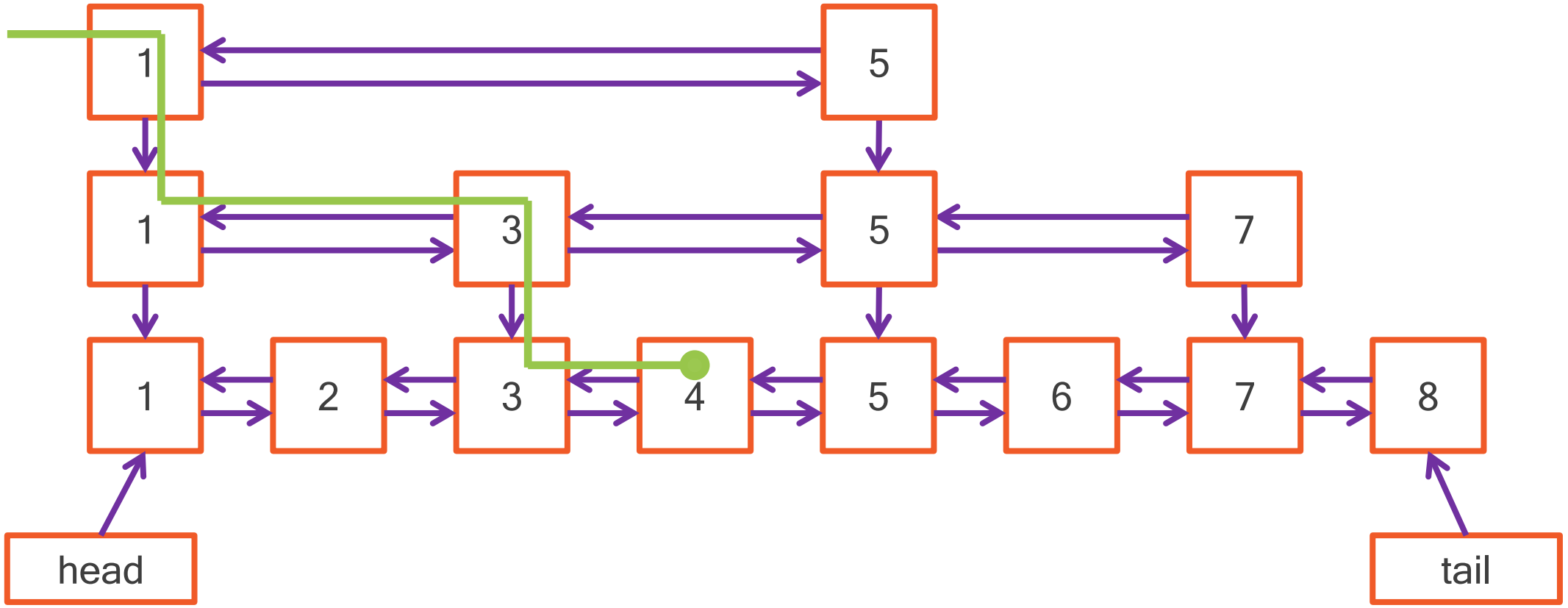
# Skip Lists

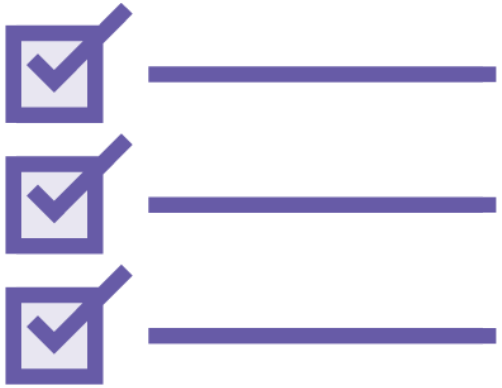
Then we can reach 4



# Skip Lists

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





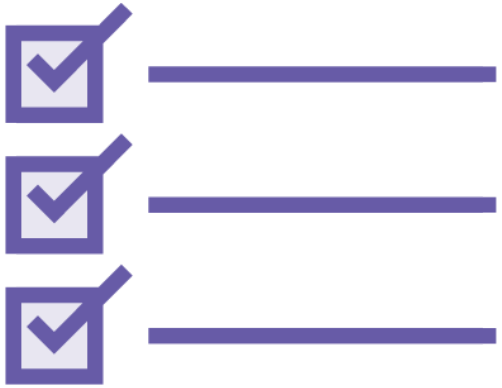
A skip list is used to implement a map

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

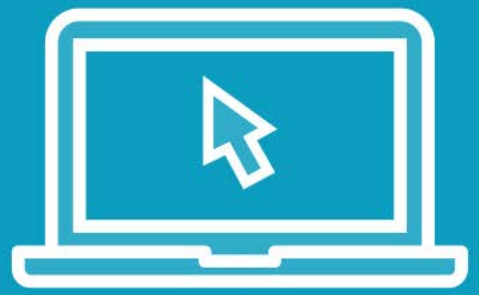
Used for maps and sets

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