Intro into Concurrent Programming (1/3)

Classic Stack and Queue Algorithms

Nikita Koval, Researcher @ JetBrains

Hydra 2022

Intro into Concurrent Programming

1. Classic Stack and Queue Algorithms

2. Modern Queues and Flat Combining

3. Relaxed Data Structures for Parallel Algorithms

Intro into Concurrent Programming: **1/3**

1. **Classic Stack and Queue Algorithms**
2. Modern Queues and Flat Combining
3. Relaxed Data Structures for Parallel Algorithms

Mutual Exclusion

- Aka *mutex* or *lock*
- At most one thread can hold the lock

Mutual Exclusion

- Aka *mutex* or *lock*
- At most one thread can hold the lock

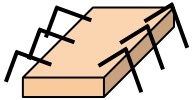
```
lock.lock()

// Critical section

lock.unlock()
```

Atomic Counter via Mutex

```
lock := Mutex()
value := 0

fun getAndIncrement(): Int {
    lock.lock()
    try {
        return value++
    } finally {
        lock.unlock()
    }
}
```

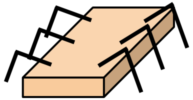
Atomic Counter via Mutex

```
lock := Mutex()
value := 0

fun getAndIncrement(): Int {
    lock.lock()
    try {
        return value++
    } finally {
        lock.unlock()
    }
}
```



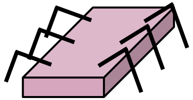
Atomic Counter via Mutex

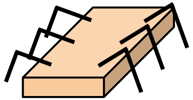
```
lock := Mutex()
value := 0

fun getAndIncrement(): Int {
    lock.lock()
    try {
        return value++
    } finally {
        lock.unlock()
    }
}
```



Atomic Counter via Mutex

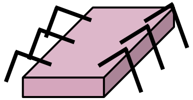
```
lock := Mutex()
value := 0

fun getAndIncrement(): Int {
    lock.lock()
    try {
        return value++
    } finally {
        lock.unlock()
    }
}
```

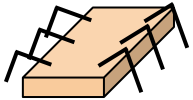




Atomic Counter via Mutex

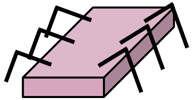
```
lock := Mutex()
value := 0

fun getAndIncrement(): Int {
    lock.lock()
    try {
        return value++
    } finally {
        lock.unlock()
    }
}
```

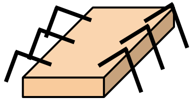




Atomic Counter via Mutex

```
lock := Mutex()
value := 0

fun getAndIncrement(): Int {
    lock.lock()
    try {
        return value++
    } finally {
        lock.unlock()
    }
}
```





Coarse-Grained Locking

```
lock := Mutex()
sequential_data_structure := ...

fun function(...): ... {
    lock.lock()
    try {
        return sequential_data_structure.function()
    } finally {
        lock.unlock()
    }
}
```

All functions should be guarded by lock

Coarse-Grained Locking

```
lock := Mutex()
sequential_data_structure := ...

fun function(...): ... {
    lock.lock()
    try {
        return sequential_data_structure.function()
    } finally {
        lock.unlock()
    }
}
```

All functions should be guarded by lock

No progress guarantee, not scalable

Compare-And-Set

- **Lock-freedom**: guarantees system-wide progress

Compare-And-Set

- **Lock-freedom**: guarantees system-wide progress

- The main building block is *`Compare-And-Set`* primitive:
  `CAS(&addr, expected, update): Boolean` tries to atomically replace the value located by address `addr` from `expected` to `update`.

Compare-And-Set

- **Lock-freedom**: guarantees system-wide progress

```
value := 0

fun getAndIncrement(): Int {
  while (true) {
    cur := value
    if CAS(&value, cur, cur + 1):
      return cur + 1
  }
}
```

Compare-And-Set

- **Lock-freedom**: guarantees system-wide progress

```
value := 0

fun getAndIncrement(): Int {
  while (true) {
    cur := value
    if CAS(&value, cur, cur + 1):
      return cur + 1
  }
}
```

If the `CAS` fails, another `getAndIncrement` has succeeded

Universal Construction via `CAS`

```
lock := Mutex()
sequential_data_structure := ...

fun function(...): ... {
    lock.lock()
    try {
        return sequential_data_structure.function()
    } finally {
        lock.unlock()
    }
}
```

Universal Construction via `CAS`

```
sequential_data_structure := ...

fun function(...): ... {
  while (true) {
    cur := sequential_data_structure
    copy := cur.makeCopy()
    result := copy.function()
    if CAS(&sequential_sata_structure, cur, copy):
      return result
  }
}
```

Universal Construction via `CAS`

```
sequential_data_structure := ...

fun function(...): ... {
  while (true) {
    cur := sequential_data_structure
    copy := cur.makeCopy()
    result := copy.function()
    if CAS(&sequential_sata_structure, cur, copy):
      return result
  }
}
```

Makes a new copy on each modification

Universal Construction via CAS

```
sequential_data_structure := ...

fun function(...): ... {
  while (true) {
    cur := sequential_data_structure
    copy := cur.makeCopy()
    result := copy.function()
    if CAS(&sequential_sata_structure, cur, copy):
      return result
  }
}
```

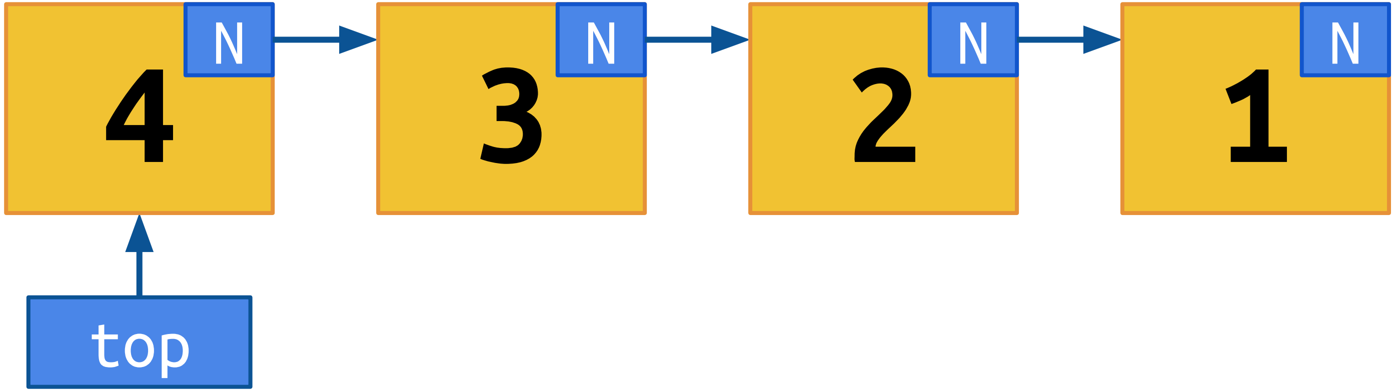
Makes a new copy on each modification

Provides lock-freedom non-blocking guarantee

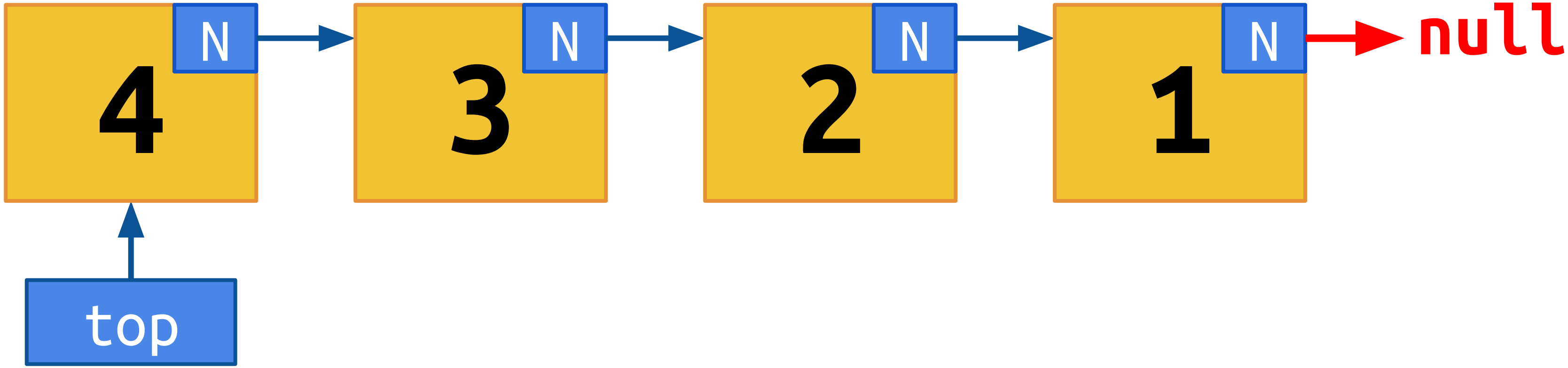
Treiber Lock-Free Stack

Treiber Stack: The Structure

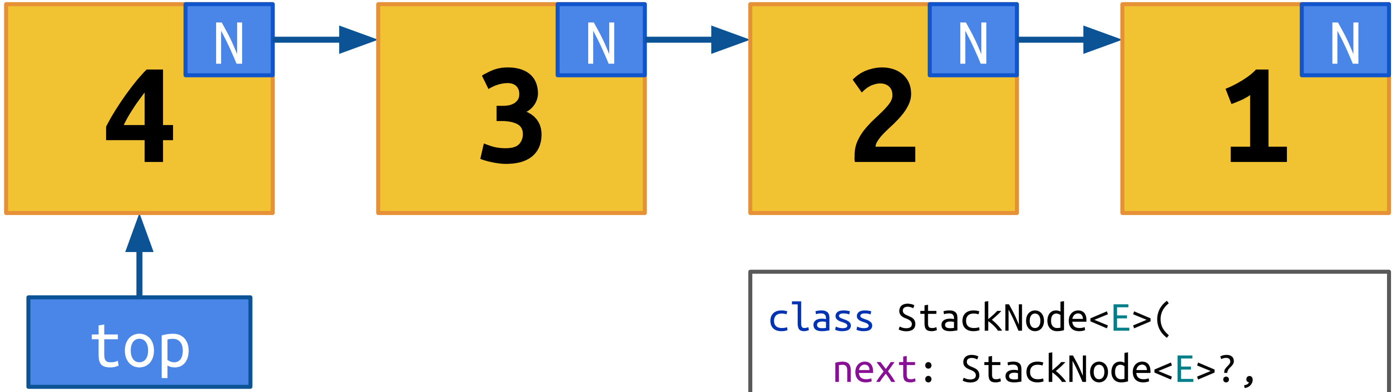
N = next



Treiber Stack: The Structure

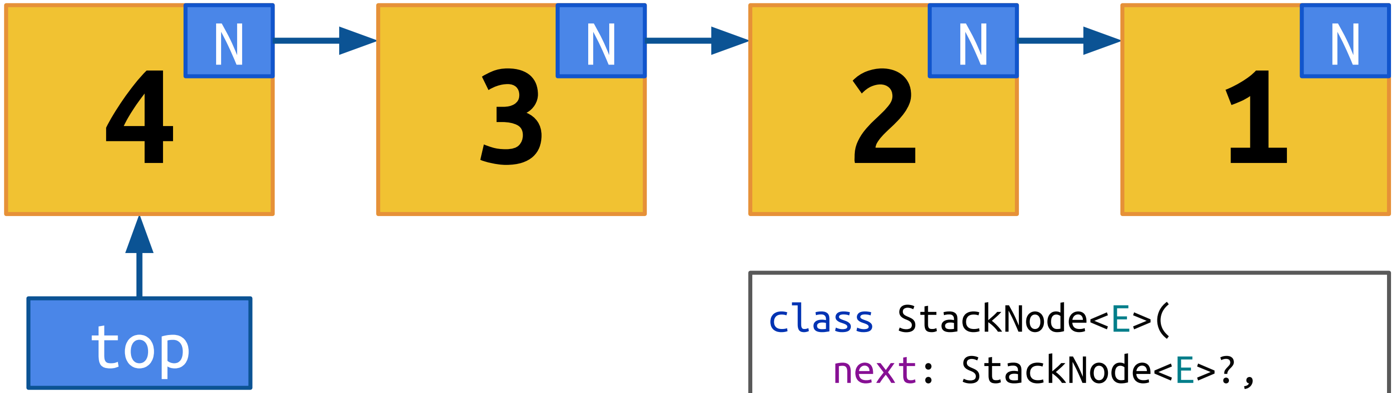


Treiber Stack: The Structure



```
class StackNode<E>(
   next: StackNode<E>?,
   value: E
)

class TreiberStack<E> {
   top: StackNode<E>? = null
}
```

Treiber Stack: The Structure

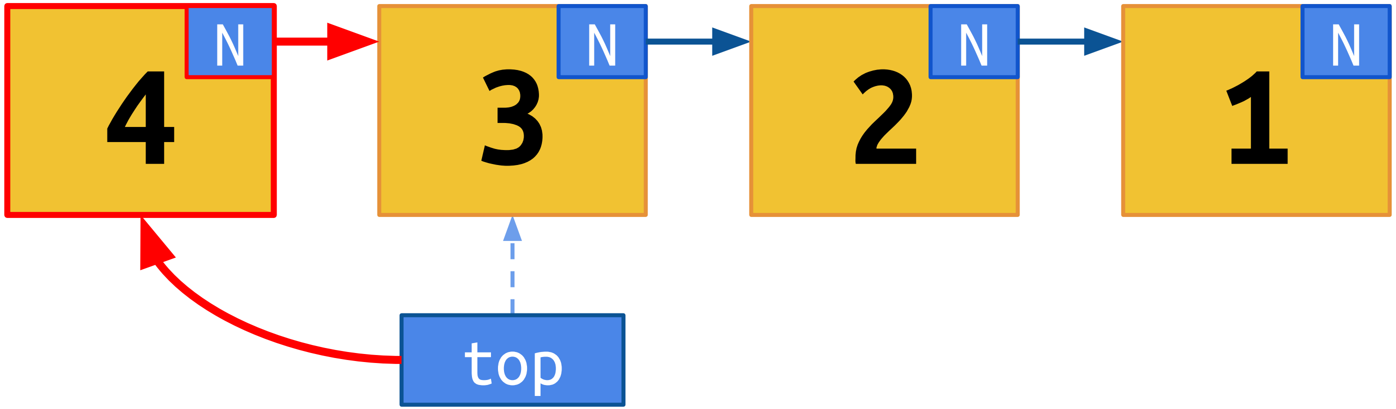


Empty stack ⇔ `top == null`

```
class StackNode<E>(
   next: StackNode<E>?,
   value: E
)

class TreiberStack<E> {
   top: StackNode<E>? = null
}
```

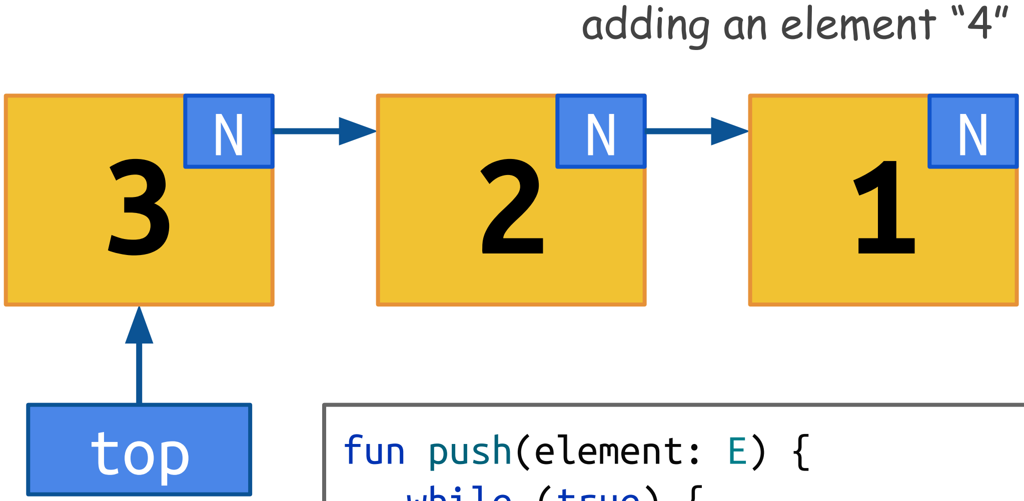
Treiber Stack: Push

adding an element “4”

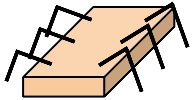


Create a new node with **`N = top`** and update **`top`**

Treiber Stack: Push

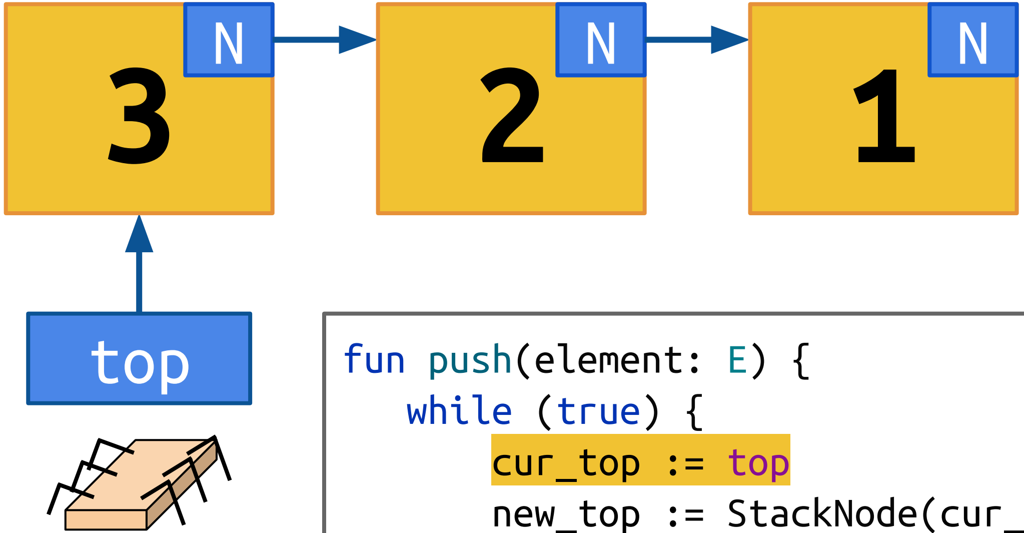


```
fun push(element: E) {
    while (true) {
        cur_top := top
        new_top := StackNode(cur_top, element)
        if CAS(&top,cur_top, new_top): return
    }
}
```



Treiber Stack: Push

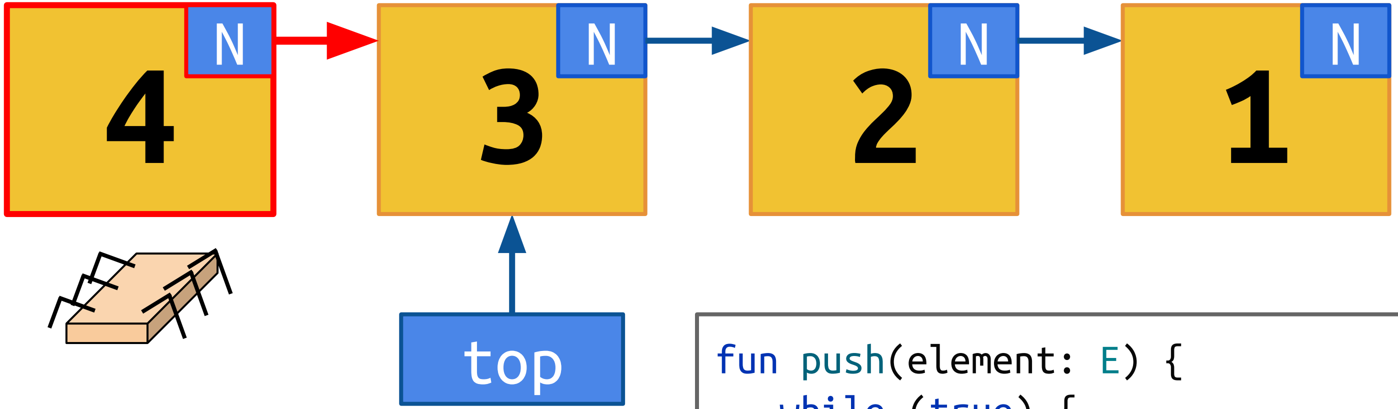
adding an element "4"



```
fun push(element: E) {
   while (true) {
       cur_top := top
       new_top := StackNode(cur_top, element)
       if CAS(&top,cur_top, new_top): return
   }
}
```

Treiber Stack: Push

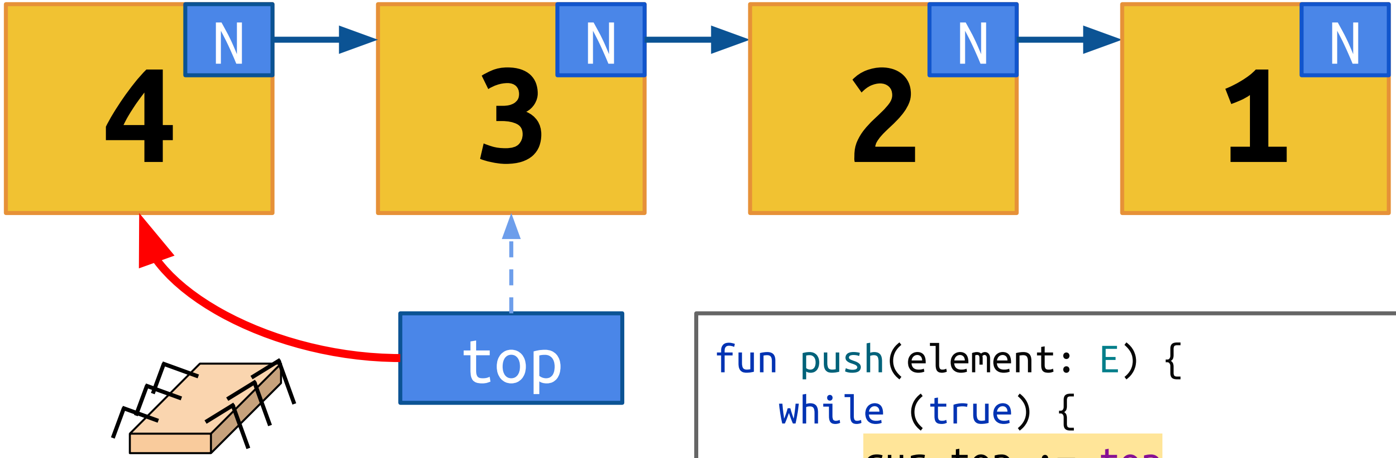
adding an element "4"



```
fun push(element: E) {
    while (true) {
        cur_top := top
        new_top := StackNode(cur_top, element)
        if CAS(&top,cur_top, new_top): return
    }
}
```

Treiber Stack: Push

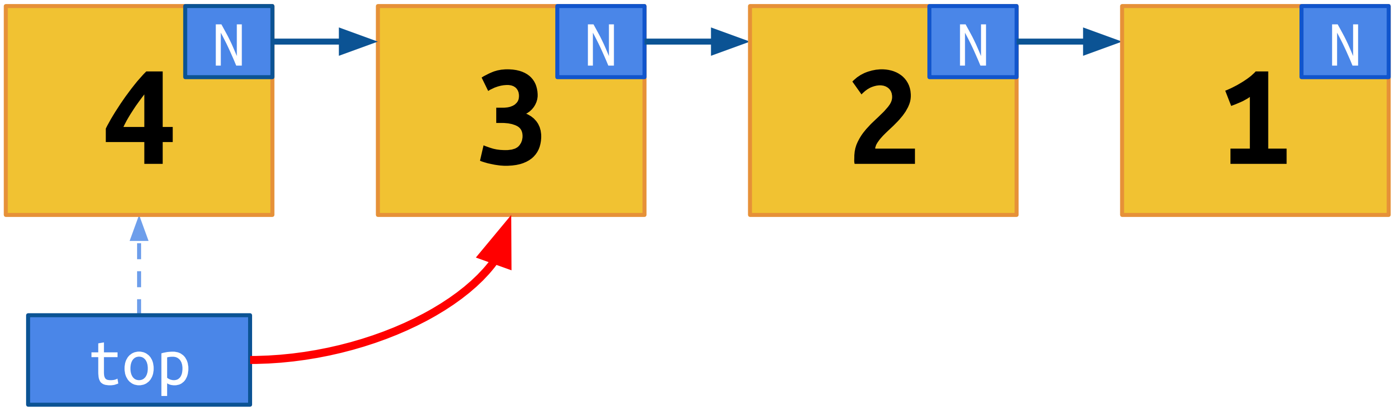
adding an element "4"



```
fun push(element: E) {
    while (true) {
        cur_top := top
        new_top := StackNode(cur_top, element)
        if CAS(&top,cur_top, new_top): return
    }
}
```

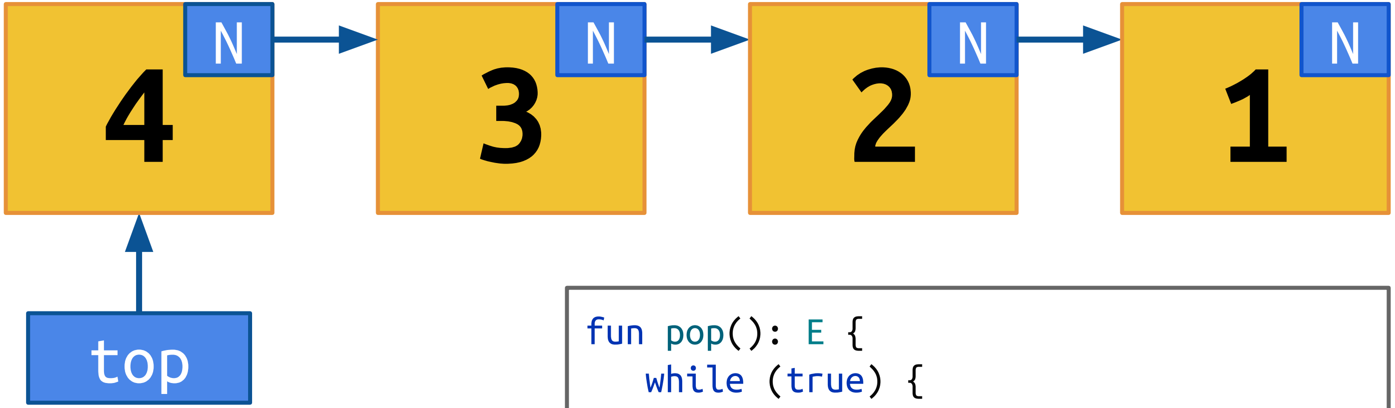
Treiber Stack: Pop

extracting the top element

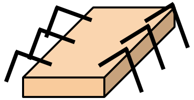


Move the **`top`** pointer forward

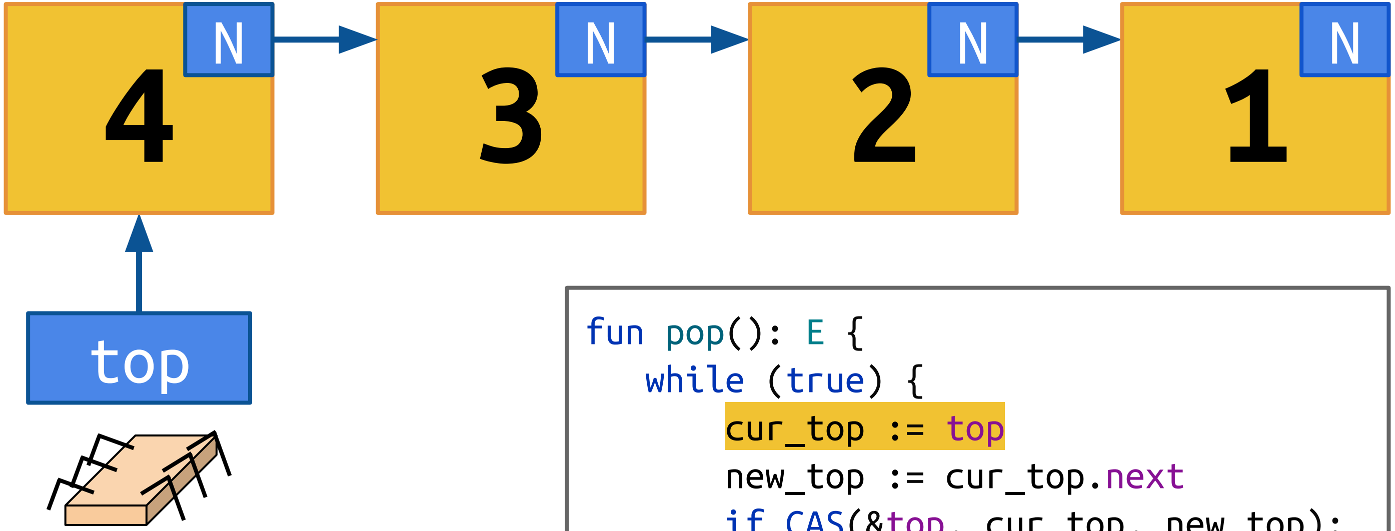
Treiber Stack: Pop



```
fun pop(): E {
   while (true) {
         cur_top := top
         new_top := cur_top.next
         if CAS(&top, cur_top, new_top):
               return cur_top.element
   }
}
```

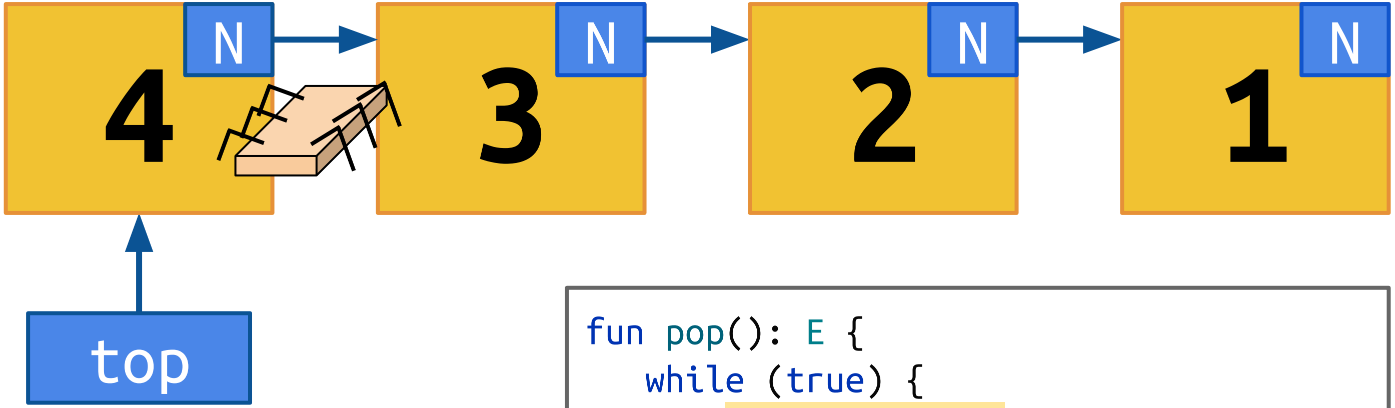


Treiber Stack: Pop



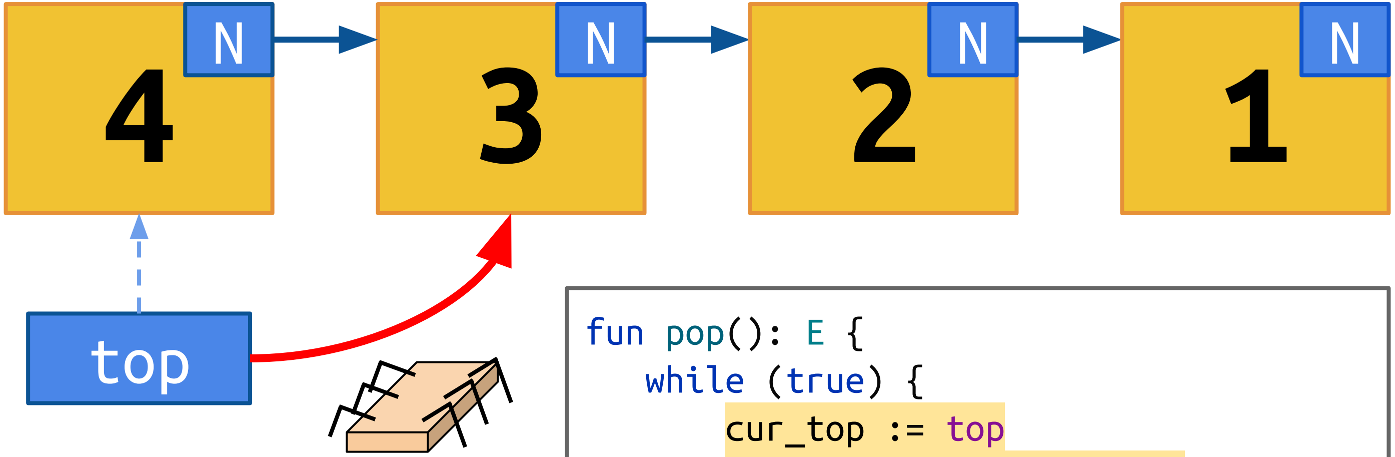
```
fun pop(): E {
    while (true) {
        cur_top := top
        new_top := cur_top.next
        if CAS(&top, cur_top, new_top):
            return cur_top.element
    }
}
```

Treiber Stack: Pop



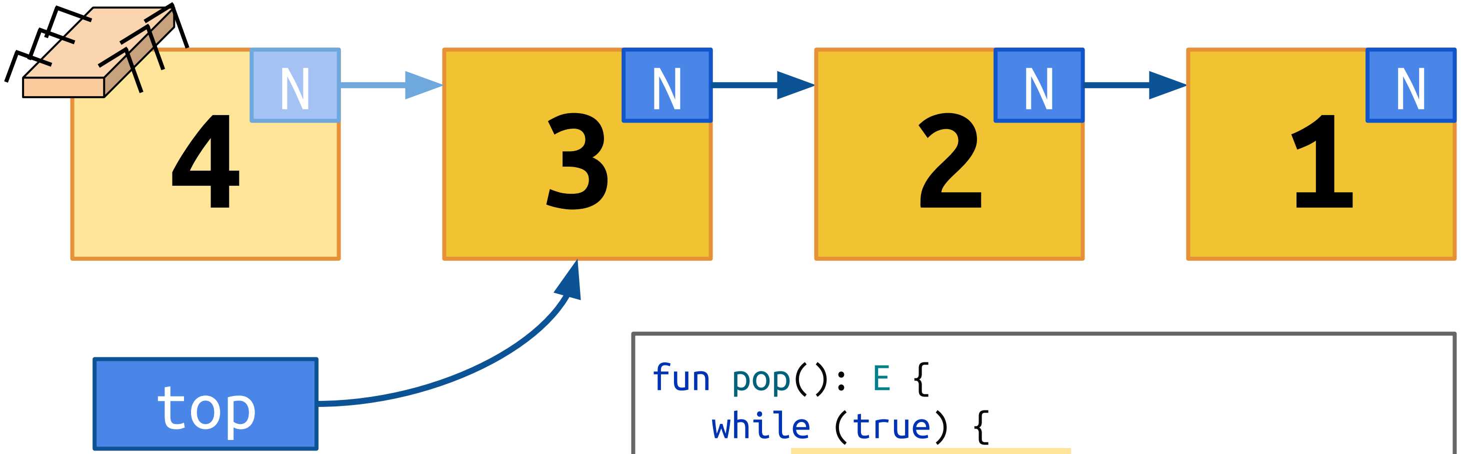
```
fun pop(): E {
   while (true) {
       cur_top := top
       new_top := cur_top.next
       if CAS(&top, cur_top, new_top):
           return cur_top.element
   }
}
```

Treiber Stack: Pop



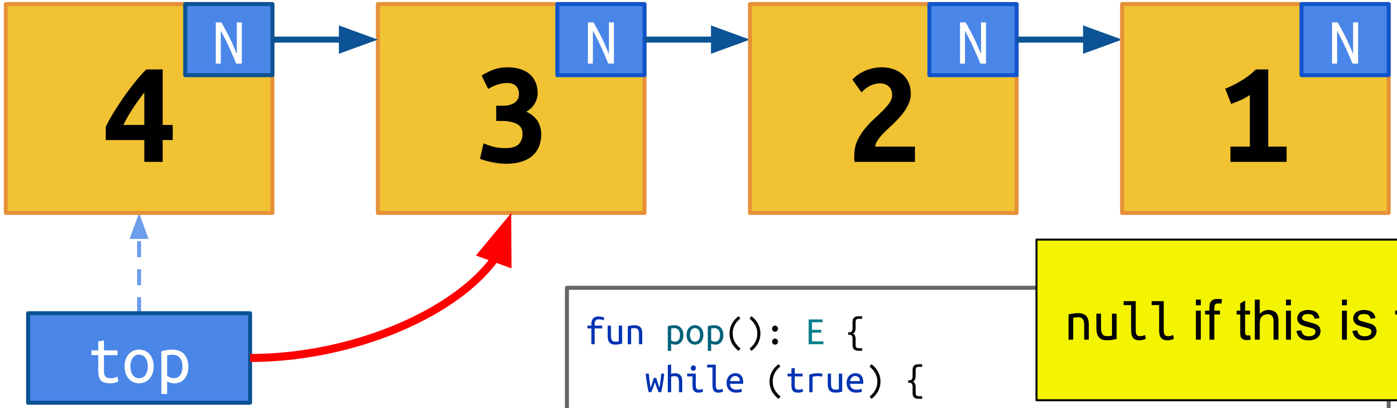
```
fun pop(): E {
    while (true) {
         cur_top := top
         new_top := cur_top.next
         if CAS(&top, cur_top, new_top):
              return cur_top.element
    }
}
```

Treiber Stack: Pop



```
fun pop(): E {
   while (true) {
      cur_top := top
      new_top := cur_top.next
      if CAS(&top, cur_top, new_top):
         return cur_top.element
   }
}
```

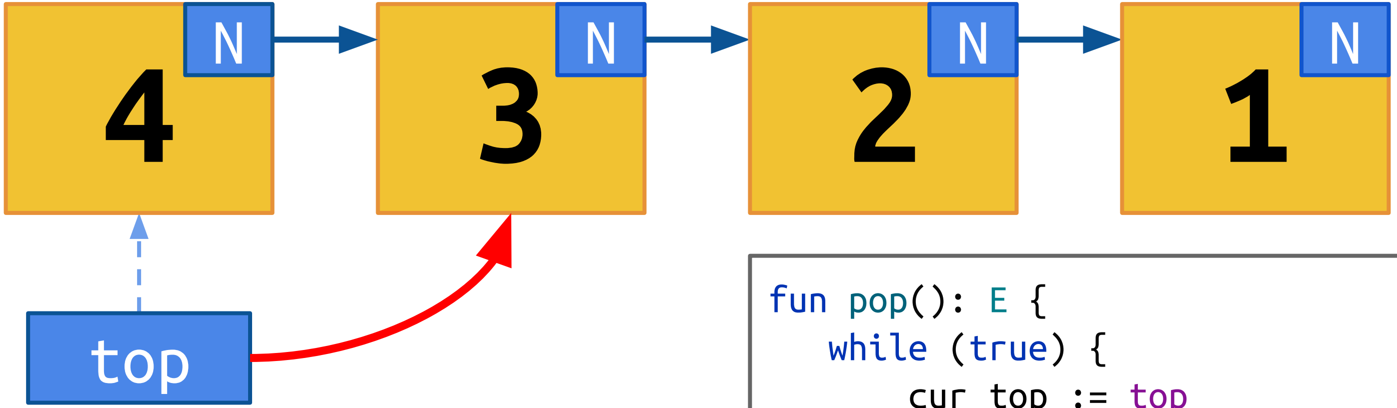
Treiber Stack: Pop



```
fun pop(): E {
    while (true) {
        cur_top := top
        new_top := cur_top.next
        if CAS(&top, cur_top, new_top):
            return cur_top.element
    }
}
```

null if this is the last node

Treiber Stack: Pop



Don’t forget to check for emptiness!

```
fun pop(): E {
   while (true) {
       cur_top := top
       if cur_top == null:
           throw EmptyStackException()
       new_top := cur_top.next
       if CAS(&top, cur_top, new_top):
           return cur_top.element
   }
}
```

Treiber Stack: Progress Guarantees

```
fun push(element: E) {
  while (true) {
      cur_top := top
      new_top := StackNode(cur_top, element)
      if CAS(&top,cur_top, new_top): return
  }
}
```

```
fun pop(): E {
  while (true) {
      cur_top := top
      if cur_top == null:
          throw EmptyStackException()
      new_top := cur_top.next
      if CAS(&top, cur_top, new_top):
          return cur_top.element
  }
}
```

Are these `push(..)` and `pop()` lock-free?

Treiber Stack: Correctness

```
fun push(element: E) {
  while (true) {
      cur_top := top
      new_top := StackNode(cur_top, element)
      if CAS(&top,cur_top, new_top): return
  }
}
```

```
fun pop(): E {
  while (true) {
      cur_top := top
      if cur_top == null:
          throw EmptyStackException()
      new_top := cur_top.next
      if CAS(&top, cur_top, new_top):
          return cur_top.element
  }
}
```

Is this stack implementation correct?

What is *correctness* in the world of concurrency?

Correctness for Concurrent Objects

```
s := TreiberStack<String>()
```

```
s.push("a")
s.pop() // “a”
```

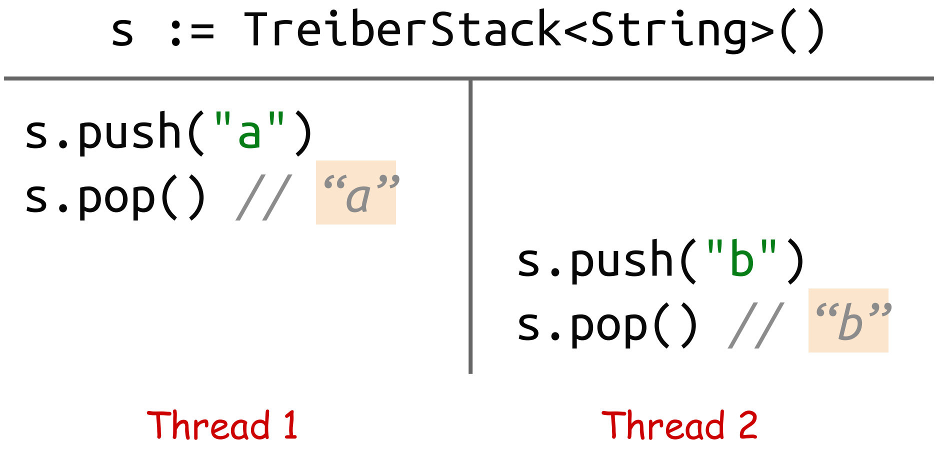
Thread 1

```
s.push("b")
s.pop() // “b”
```

Thread 2

Is this execution correct?

Correctness for Concurrent Objects



```
s := TreiberStack<String>()
```

```
s.push("a")
s.pop() // "a"
```

```
s.push("b")
s.pop() // "b"
```

Thread 1

Thread 2

Correctness for Concurrent Objects

```
      s := TreiberStack<String>()
-------------------------------------------
s.push("a")            |  s.push("b")
s.pop() // “b”         |  s.pop() // “a”
```

What about this one?

Correctness for Concurrent Objects

```
        s := TreiberStack<String>()
------------------------------------------
s.push("a")          |
                     |  s.push("b")
s.pop() // “b”       |
                     |  s.pop() // “a”
```

Correctness for Concurrent Objects

```
s := TreiberStack<String>()
```

`s.push("a")`
`s.pop() // “a”`
	`s.push("b")`
	`s.pop() // “a”`

Is this execution also correct?

Correctness for Concurrent Objects

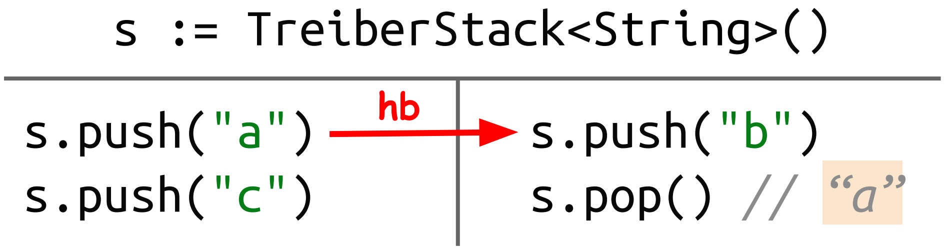
```
         s := TreiberStack<String>()
-----------------------------------------------
s.push("a")           | s.push("b")
s.pop() // “a”        | s.pop() // “a”
```

BANNED

Is this execution also correct?

Correctness for Concurrent Objects

hb = happens before

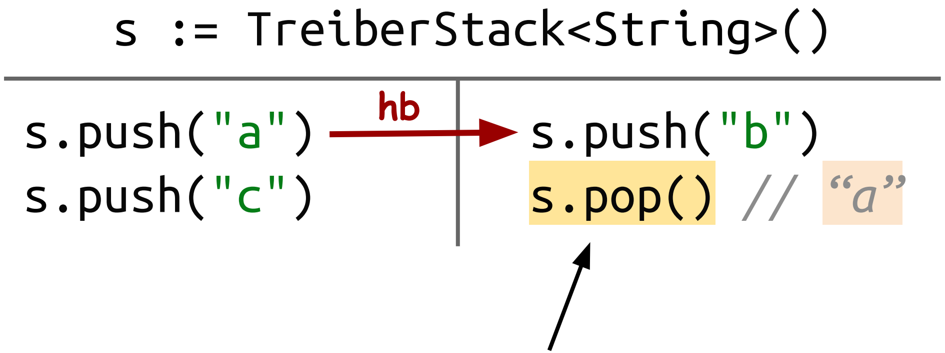


`s := TreiberStack<String>()`

`s.push("a")`	`s.push("b")`
`s.push("c")`	`s.pop() // "a"`

Correctness for Concurrent Objects

hb = happens before

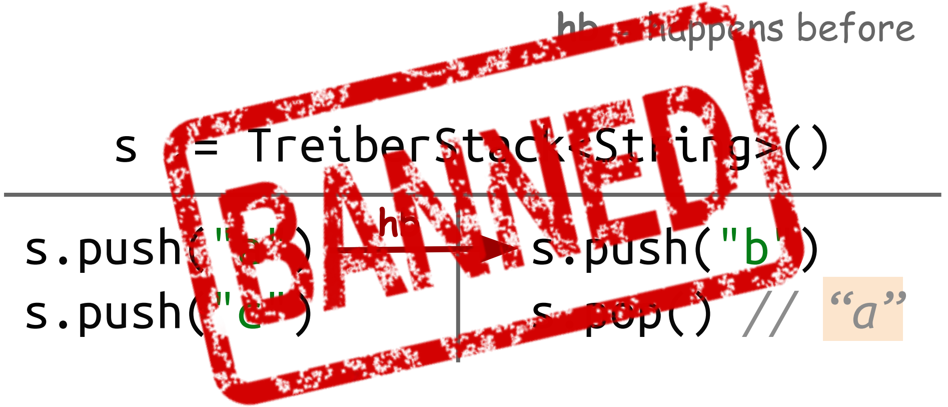


```
s := TreiberStack<String>()
```

`s.push("a")`	`s.push("b")`
`s.push("c")`	`s.pop()` // “a”

This pop() is eligible to extract either “b” or “c”

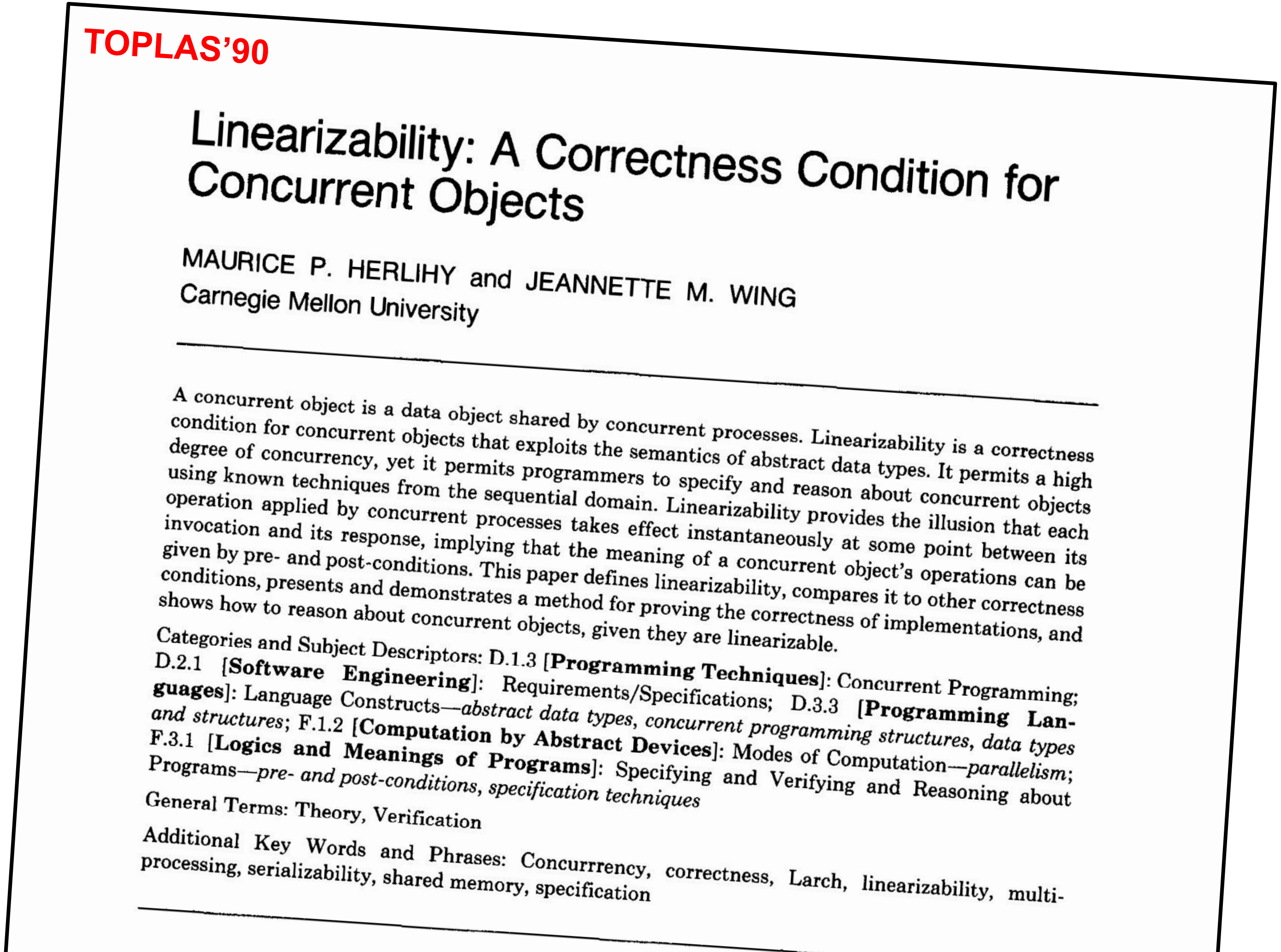
Correctness for Concurrent Objects



```
s = TreiberStack<String>()
```

```
s.push("a")          s.push("b")
s.push("c")          s.pop() // “a”
```

TOPLAS'90



Linearizability: A Correctness Condition for Concurrent Objects

MAURICE P. HERLIHY and JEANNETTE M. WING
Carnegie Mellon University

A concurrent object is a data object shared by concurrent processes. Linearizability is a correctness condition for concurrent objects that exploits the semantics of abstract data types. It permits a high degree of concurrency, yet it permits programmers to specify and reason about concurrent objects using known techniques from the sequential domain. Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response, implying that the meaning of a concurrent object's operations can be given by pre- and post-conditions. This paper defines linearizability, compares it to other correctness conditions, presents and demonstrates a method for proving the correctness of implementations, and shows how to reason about concurrent objects, given they are linearizable.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types, concurrent programming structures, data types and structures*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*pre- and post-conditions, specification techniques*

General Terms: Theory, Verification

Additional Key Words and Phrases: Concurrency, correctness, Larch, linearizability, multiprocessing, serializability, shared memory, specification

Correctness for Concurrent Objects: Linearizability

An execution is ***linearizable*** if it can be expressed with a *sequential* execution that:

1. does not violate the original happens-before order (program order + synchronization)
2. shows the same results

* the definition is simplified

Treiber Stack: Linearizability Points

```
fun push(element: E) {
  while (true) {
      cur_top := top
      new_top := StackNode(cur_top, element)
      if CAS(&top,cur_top, new_top): return
  }
}
```

```
fun pop(): E {
  while (true) {
      cur_top := top
      if cur_top == null:
          throw EmptyStackException()
      new_top := cur_top.next
      if CAS(&top, cur_top, new_top):
          return cur_top.element
  }
}
```

Successful CAS-s are linearizability points

Treiber Stack: Reading the Top

```
fun top(): E? {
   cur_top := top
   if cur_top == null: return null
   return cur_top.element
}
```

Treiber Stack: Reading the Top

Reading `top` is the linearizability point

```
fun top(): E? {
   cur_top := top
   if cur_top == null: return null
   return cur_top.element
}
```

Treiber Stack: Reading the Top

Reading `top` is the linearizability point

```
fun top(): E? {
    cur_top := top
    if cur_top == null: return null
    return cur_top.element
}
```

Is it lock-free?

Treiber Stack: Reading the Top

Reading `top` is the linearizability point

```
fun top(): E? {
    cur_top := top
    if cur_top == null: return null
    return cur_top.element
```

It is **wait-free!**

Is it lock-free?

Lock-Freedom vs Wait-Freedom

- **Lock-Freedom**: *there is* an operation that finishes in a bounded number of steps

- **Wait-Freedom:** *each* operation finishes in a bounded number of steps

Lock-Freedom vs Wait-Freedom

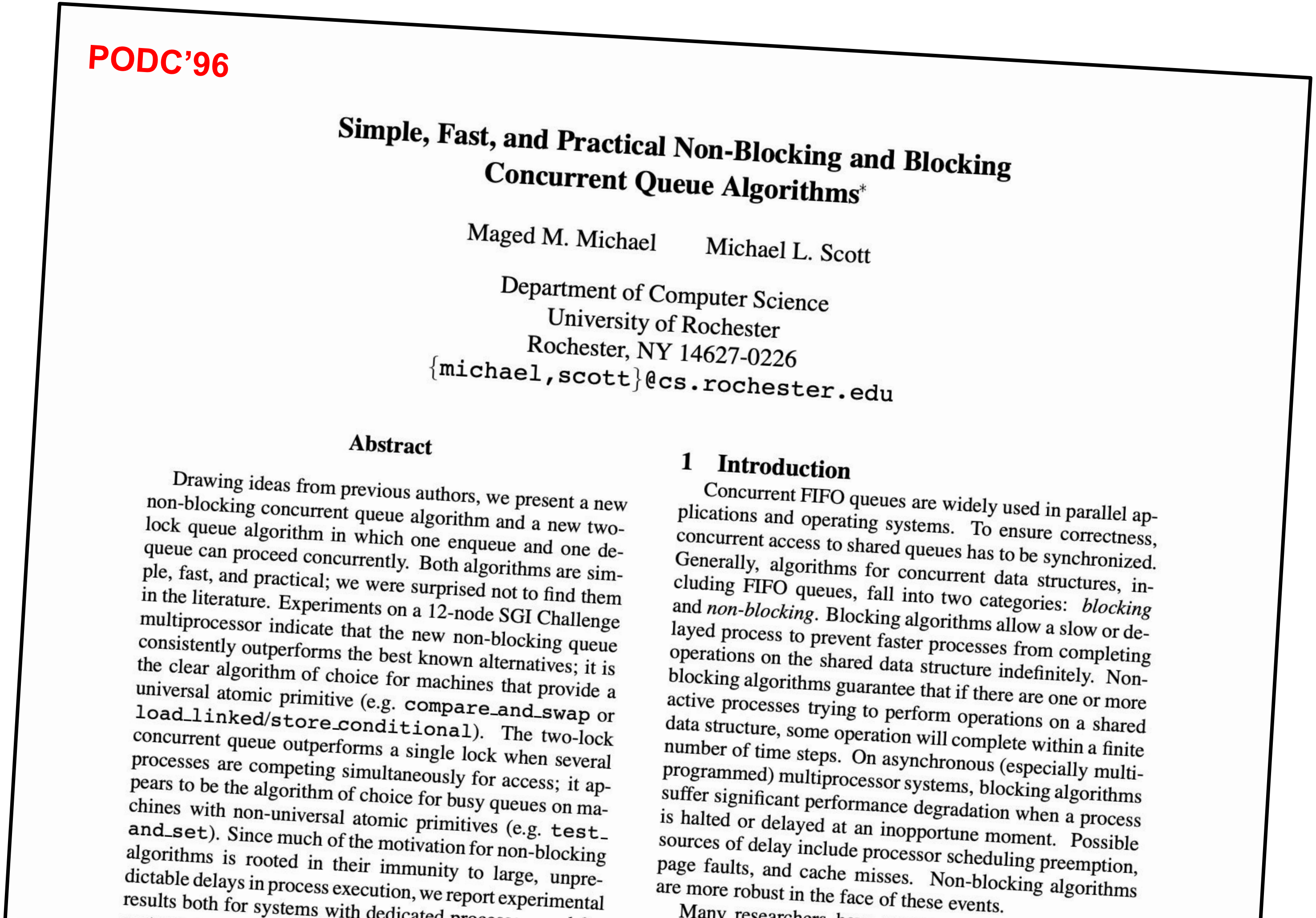
- **Lock-Freedom**: *there is* an operation that finishes in a bounded number of steps

- **Wait-Freedom:** *each* operation finishes in a bounded number of steps

Reads often can be wait-free

Michael-Scott Lock-Free Queue

PODC'96



Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms*

Maged M. Michael   Michael L. Scott

Department of Computer Science
University of Rochester
Rochester, NY 14627-0226
{michael,scott}@cs.rochester.edu

Abstract

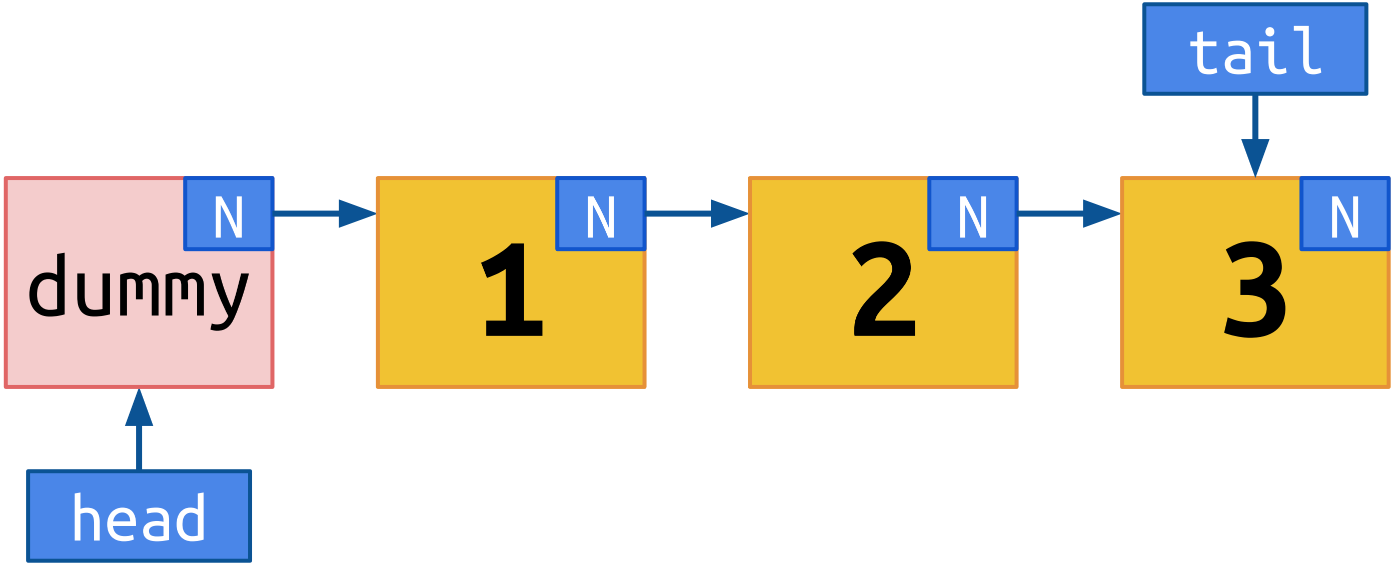
Drawing ideas from previous authors, we present a new non-blocking concurrent queue algorithm and a new two-lock queue algorithm in which one enqueue and one dequeue can proceed concurrently. Both algorithms are simple, fast, and practical; we were surprised not to find them in the literature. Experiments on a 12-node SGI Challenge multiprocessor indicate that the new non-blocking queue consistently outperforms the best known alternatives; it is the clear algorithm of choice for machines that provide a universal atomic primitive (e.g. `compare_and_swap` or `load_linked/store_conditional`). The two-lock concurrent queue outperforms a single lock when several processes are competing simultaneously for access; it appears to be the algorithm of choice for busy queues on machines with non-universal atomic primitives (e.g. `test_and_set`). Since much of the motivation for non-blocking algorithms is rooted in their immunity to large, unpredictable delays in process execution, we report experimental results both for systems with dedicated

1 Introduction

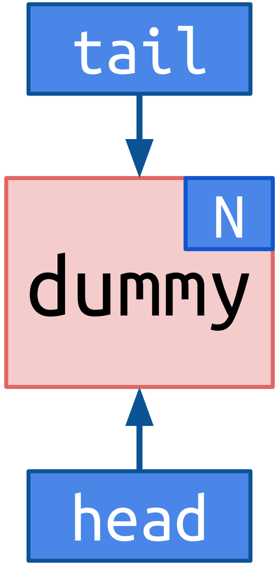
Concurrent FIFO queues are widely used in parallel applications and operating systems. To ensure correctness, concurrent access to shared queues has to be synchronized. Generally, algorithms for concurrent data structures, including FIFO queues, fall into two categories: *blocking* and *non-blocking*. Blocking algorithms allow a slow or delayed process to prevent faster processes from completing operations on the shared data structure indefinitely. Non-blocking algorithms guarantee that if there are one or more active processes trying to perform operations on a shared data structure, some operation will complete within a finite number of time steps. On asynchronous (especially multiprogrammed) multiprocessor systems, blocking algorithms suffer significant performance degradation when a process is halted or delayed at an inopportune moment. Possible sources of delay include processor scheduling preemption, page faults, and cache misses. Non-blocking algorithms are more robust in the face of these events.

Many researchers

MS Queue: The Structure

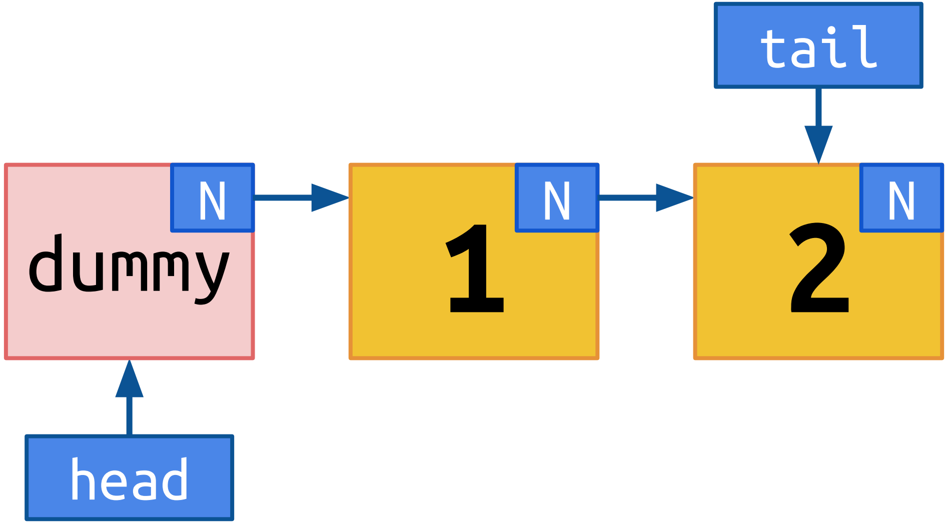


MS Queue: The Structure



Queue is empty ⇔ `head == tail`

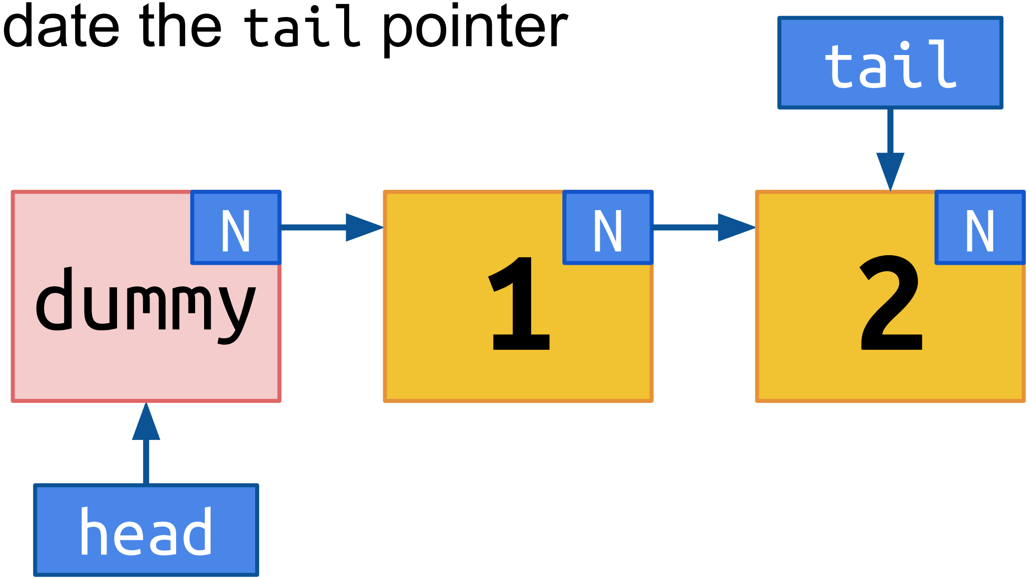
MS Queue: The Structure

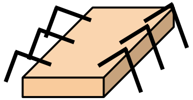


```
class MSQueueNode<E>(
   next: QueueNode<E>?,
   value: E
)

class MSQueue<E> {
   head: MSQueueNode<E>
   tail: MSQueueNode<E>

   init {
      dummy := QueueNode(null, null)
      head = tail = dummy
   }
}
```

MS Queue: Enqueue

0. Create a new node with the element
1. Update the `next` pointer of the current `tail`
2. Update the `tail` pointer

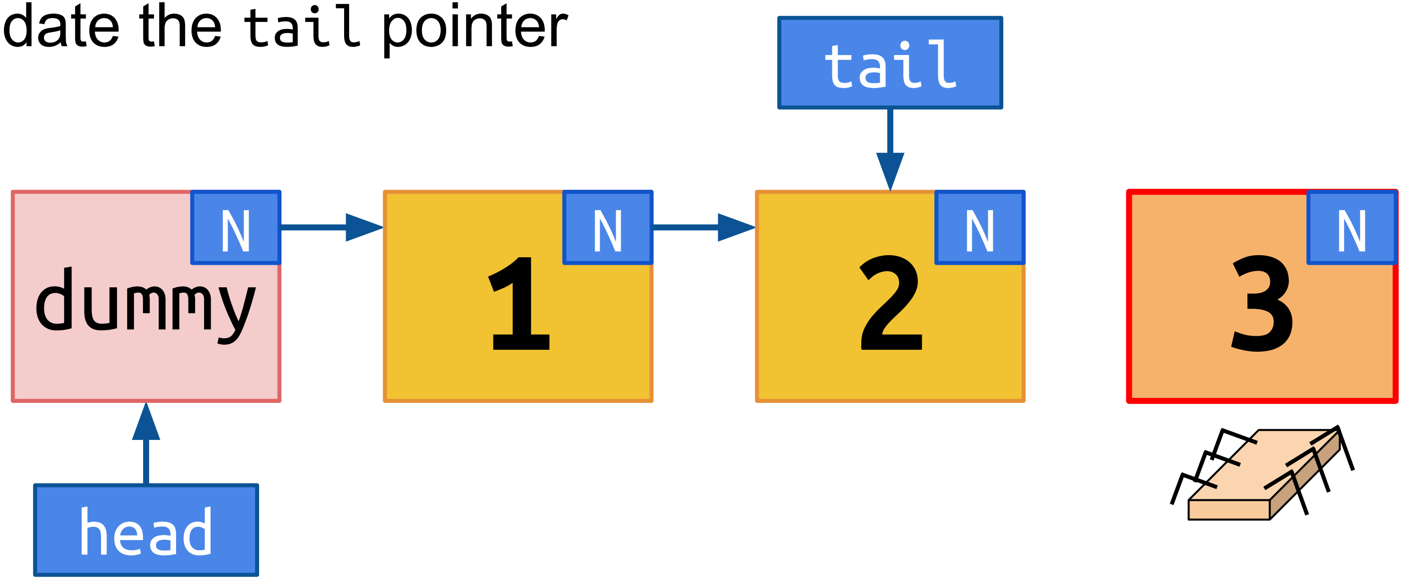




enqueue("3")

MS Queue: Enqueue

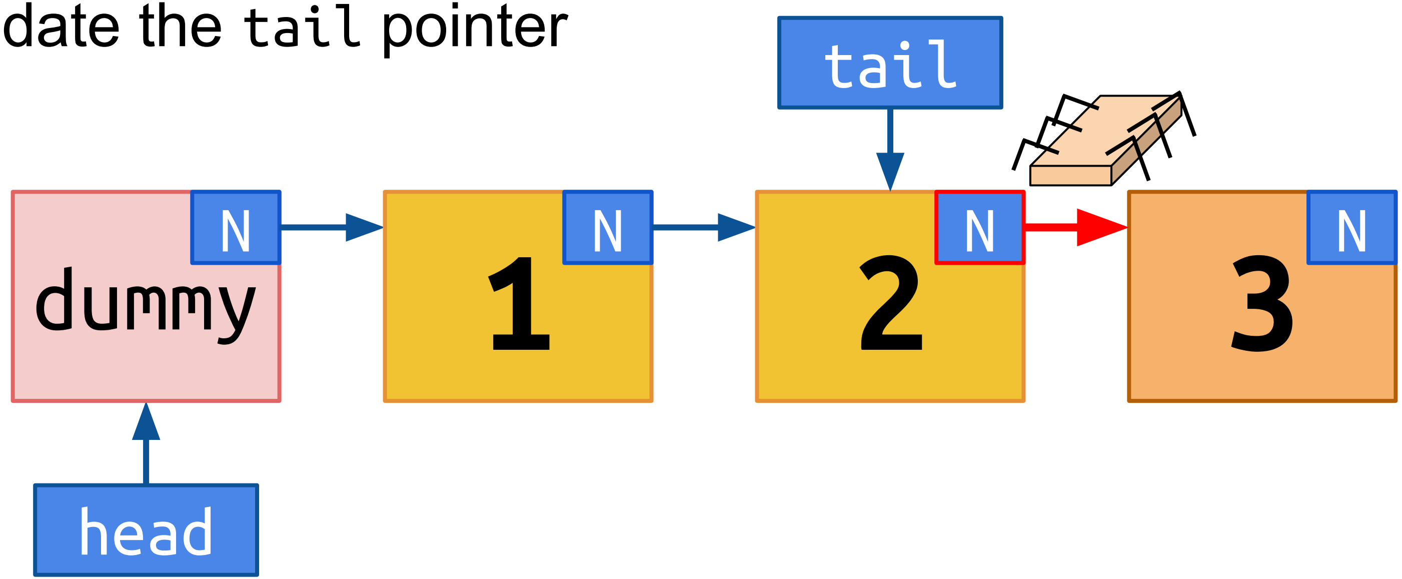
0. Create a new node with the element
1. Update the `next` pointer of the current `tail`
2. Update the `tail` pointer



enqueue("3")

MS Queue: Enqueue

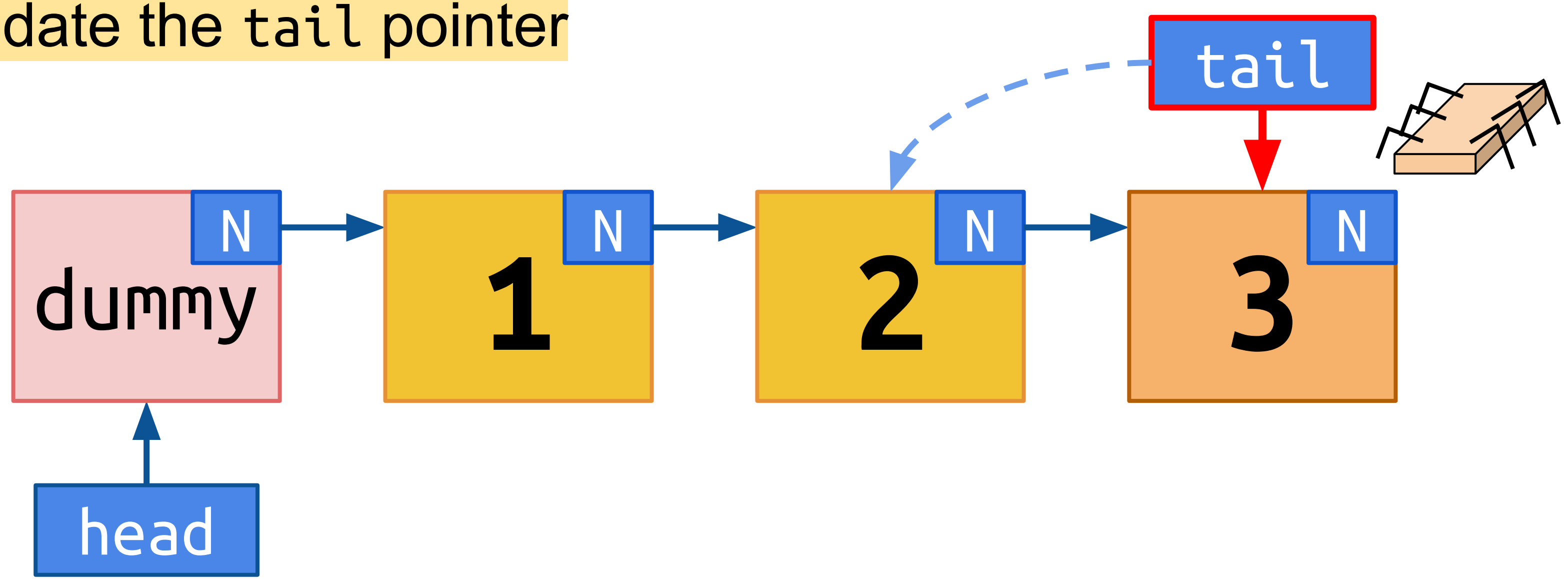
0. Create a new node with the element
1. Update the `next` pointer of the current `tail`
2. Update the `tail` pointer



enqueue("3")

MS Queue: Enqueue

0. Create a new node with the element
1. Update the `next` pointer of the current `tail`
2. Update the `tail` pointer

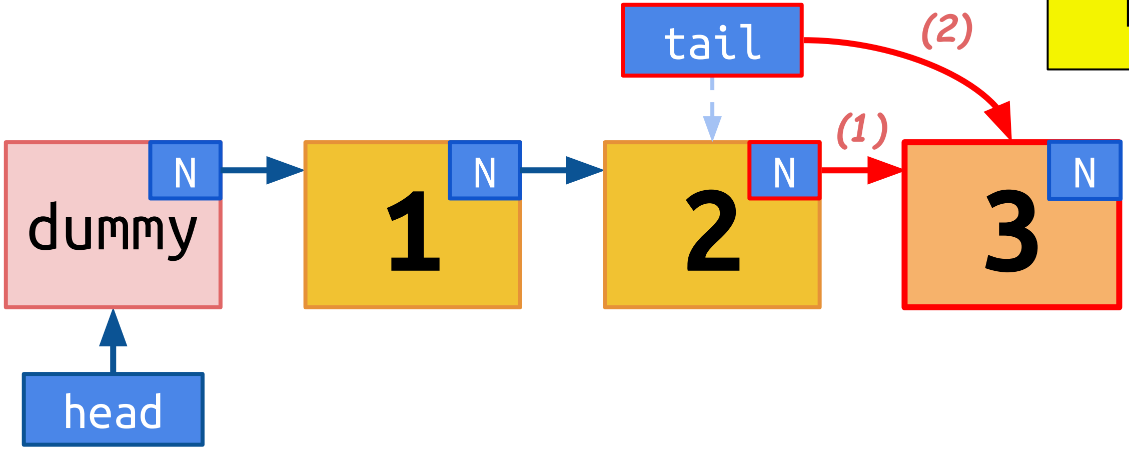


enqueue("3")

MS Queue: Enqueue

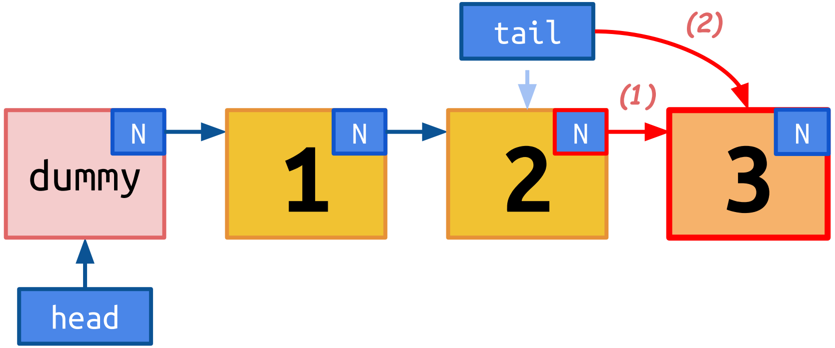
0. Create a new node with the element
1. Update the `next` pointer of the current `tail`
2. Update the `tail` pointer

(1) and (2) should be performed atomically



MS Queue: Enqueue

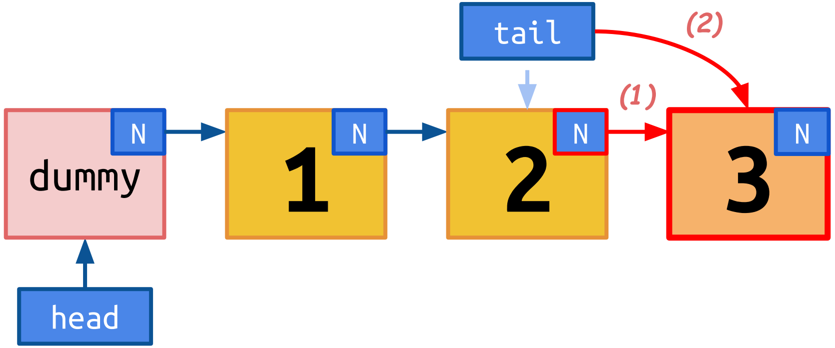
0. Create a new node with the element
1. Update the `next` pointer of the current `tail`
2. Update the `tail` pointer



```
fun enqueue(e: E) {
(0) node := MSQueueNode(e)
    cur_tail := tail
(1) cur_tail.next = node
(2) tail = node
}
```

MS Queue: Enqueue

0. Create a new node with the element
1. Update the `next` pointer of the current `tail`
2. Update the `tail` pointer

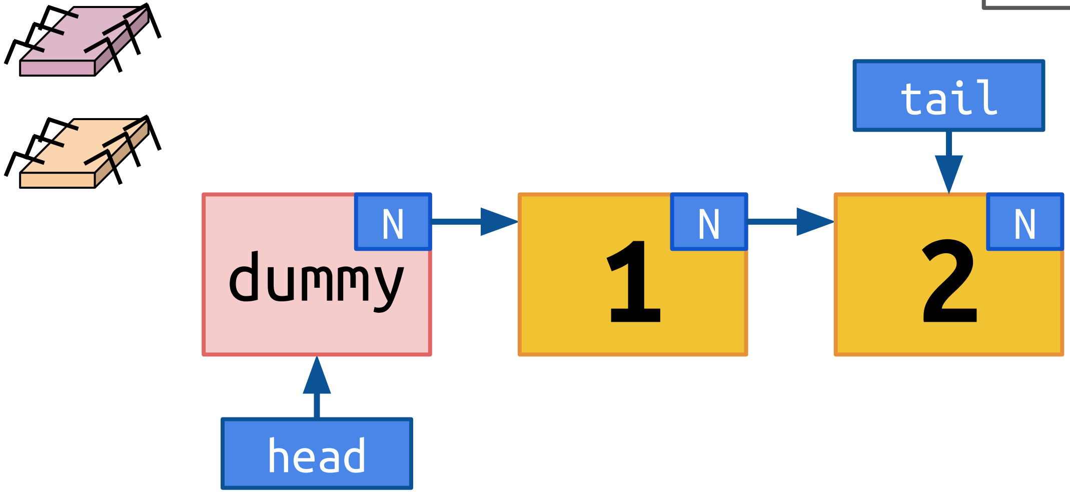


```
fun enqueue(e: E) = while (true) {
(0) node := MSQueueNode(e)
    cur_tail := tail
(1) if CAS(&cur_tail.next, null, node) {
  (2) tail = node
      return
    }
}
```

MS Queue: Enqueue

0. Create a new node with the element
1. Update the `next` pointer of the current `tail`
2. Update the `tail` pointer

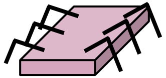
```
fun enqueue(e: E) = while (true) {
(0) node := MSQueueNode(e)
    cur_tail := tail
(1) if CAS(&cur_tail.next, null, node) {
    (2) tail = node
        return
    }
}
```

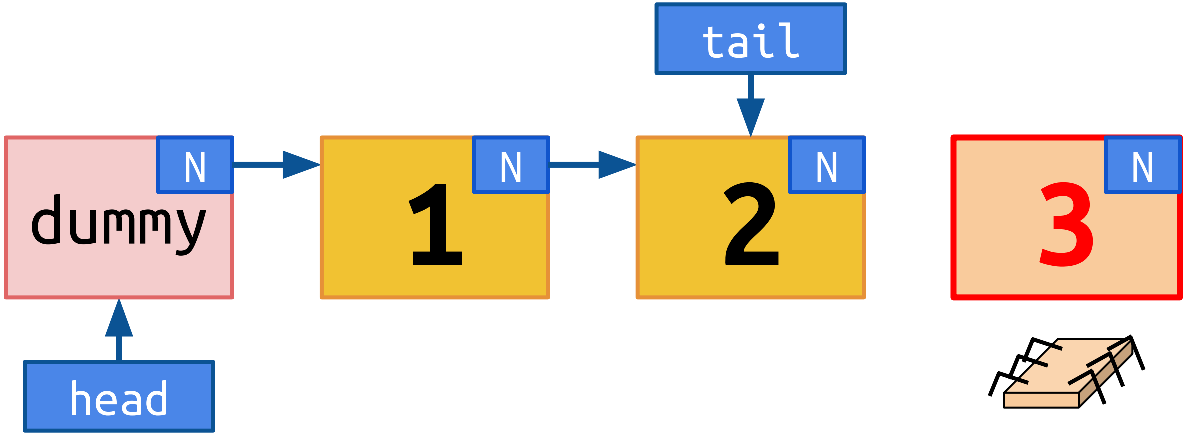


MS Queue: Enqueue

0. Create a new node with the element
1. Update the `next` pointer of the current `tail`
2. Update the `tail` pointer

```
    fun enqueue(e: E) = while (true) {
(0)   node := MSQueueNode(e)
      cur_tail := tail
(1)   if CAS(&cur_tail.next, null, node) {
    (2) tail = node
        return
      }
    }
```

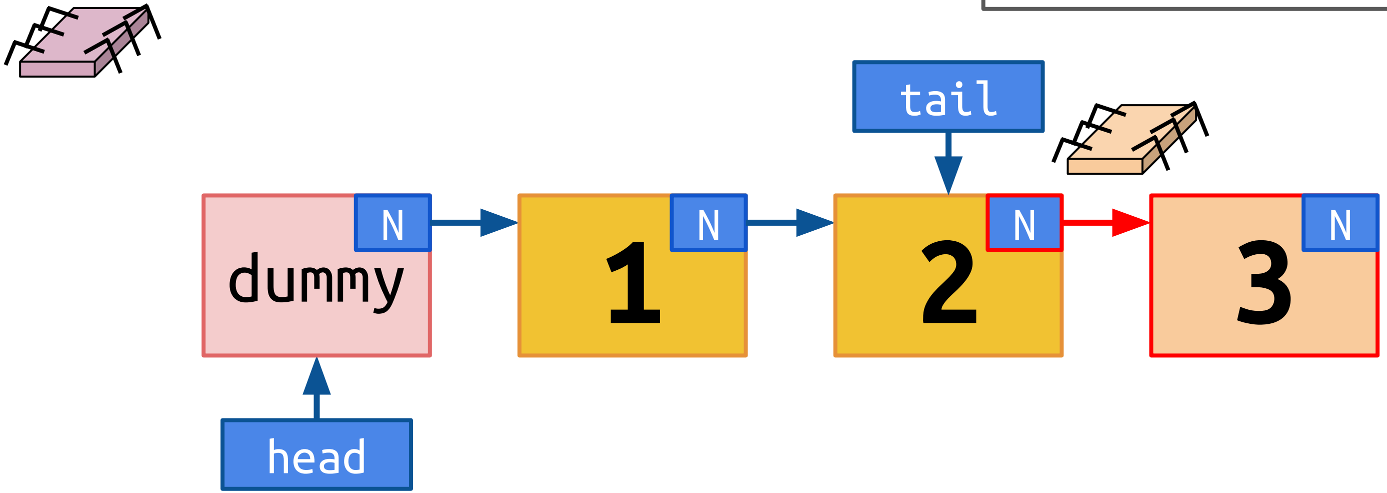




MS Queue: Enqueue

0. Create a new node with the element
1. Update the `next` pointer of the current `tail`
2. Update the `tail` pointer

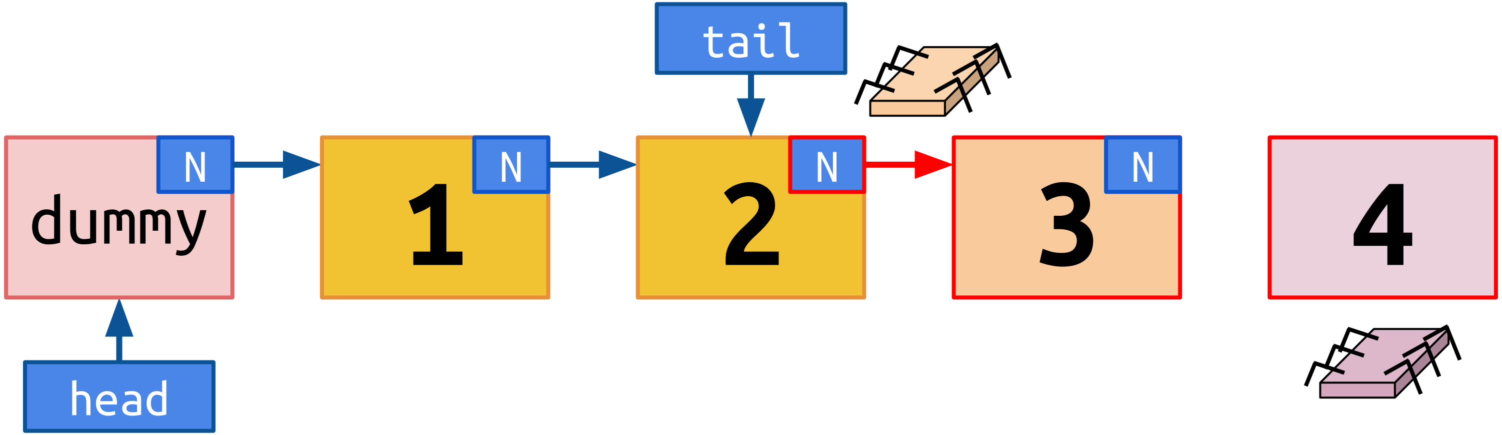
```
fun enqueue(e: E) = while (true) {
(0) node := MSQueueNode(e)
    cur_tail := tail
(1) if CAS(&cur_tail.next, null, node) {
    (2) tail = node
        return
    }
}
```



MS Queue: Enqueue

0. Create a new node with the element
1. Update the `next` pointer of the current `tail`
2. Update the `tail` pointer

```
fun enqueue(e: E) = while (true) {
(0) node := MSQueueNode(e)
    cur_tail := tail
(1) if CAS(&cur_tail.next, null, node) {
  (2) tail = node
      return
    }
}
```

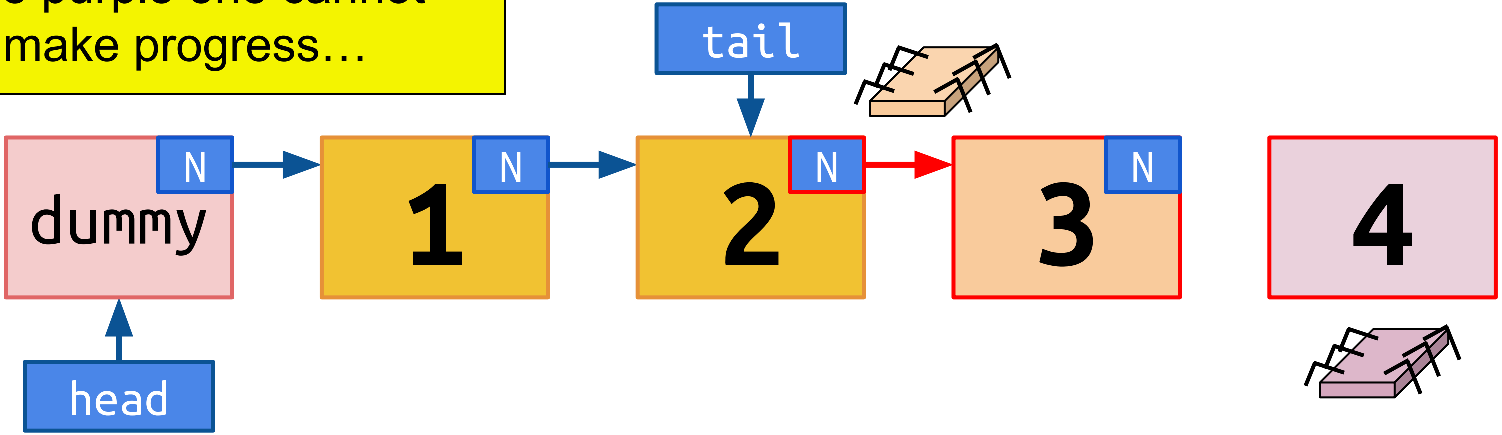


MS Queue: Enqueue

0. Create a new node with the element
1. Update the `next` pointer of the current `tail`
2. Update the `tail` pointer

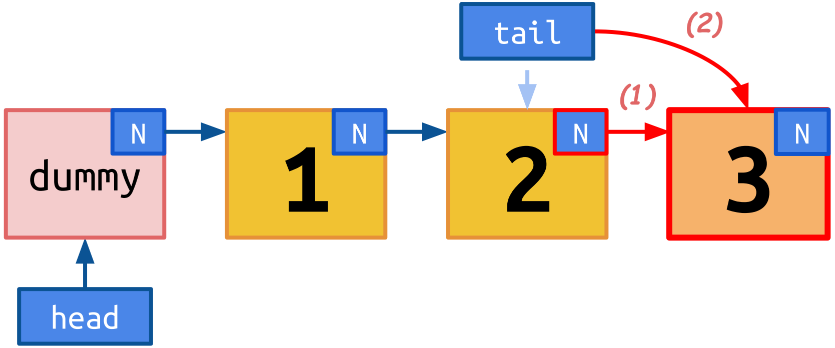
```
fun enqueue(e: E) = while (true) {
(0) node := MSQueueNode(e)
    cur_tail := tail
(1) if CAS(&cur_tail.next, null, node) {
    (2) tail = node
        return
    }
}
```

The purple one cannot make progress…



MS Queue: Enqueue

0. Create a new node with the element
1. Update the `next` pointer of the current `tail`
2. Update the `tail` pointer



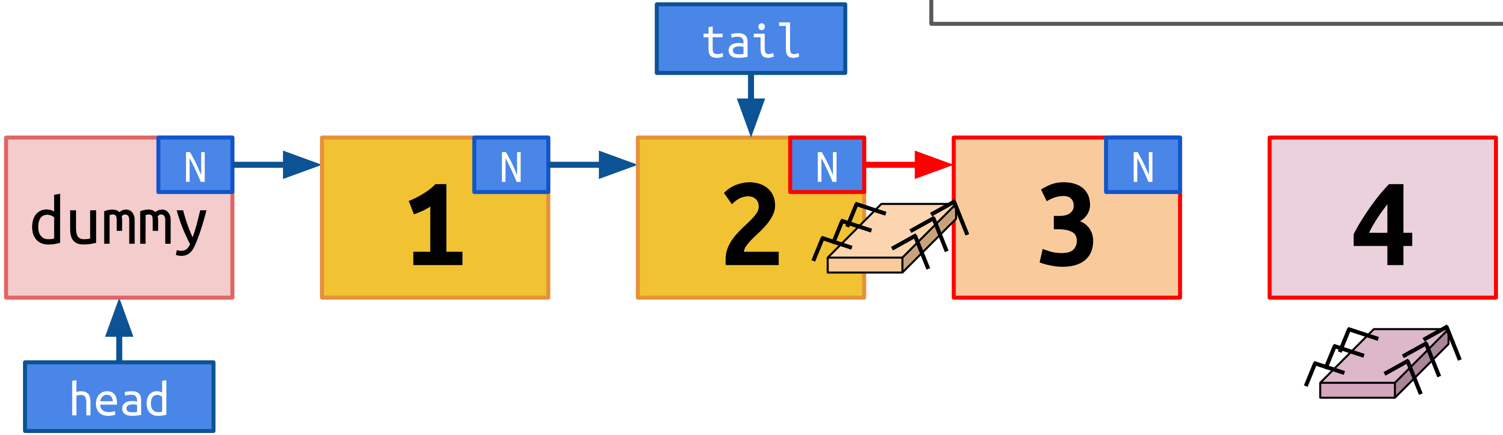
```
fun enqueue(e: E) = while (true) {
(0) node := MSQueueNode(e)
    cur_tail := tail
(1) if CAS(&cur_tail.next, null, node) {
      (2) CAS(&tail, cur_tail, node)
          return
    } else {
      (2) CAS(&tail, cur_tail, cur_tail.next)
    }
}
```

helping!

MS Queue: Enqueue

0. Create a new node with the element
1. Update the `next` pointer of the current `tail`
2. Update the `tail` pointer

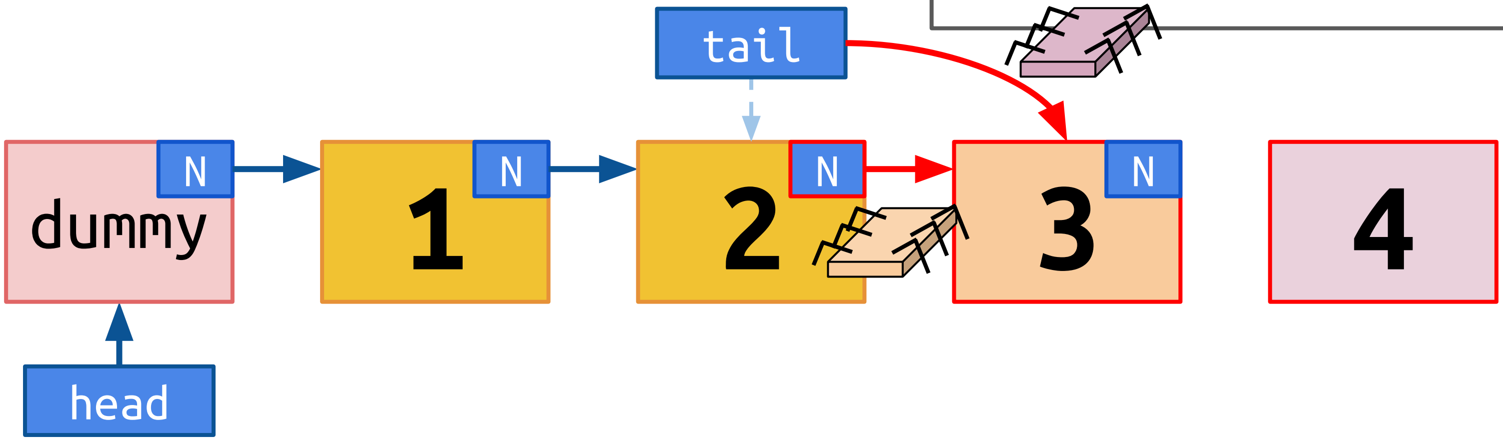
```
fun enqueue(e: E) = while (true) {
  node := MSQueueNode(e)
  cur_tail := tail
  if CAS(&cur_tail.next, null, node) {
    CAS(&tail, cur_tail, node)
    return
  } else {
    CAS(&tail, cur_tail, cur_tail.next)
  }
}
```



MS Queue: Enqueue

0. Create a new node with the element
1. Update the `next` pointer of the current `tail`
2. Update the `tail` pointer

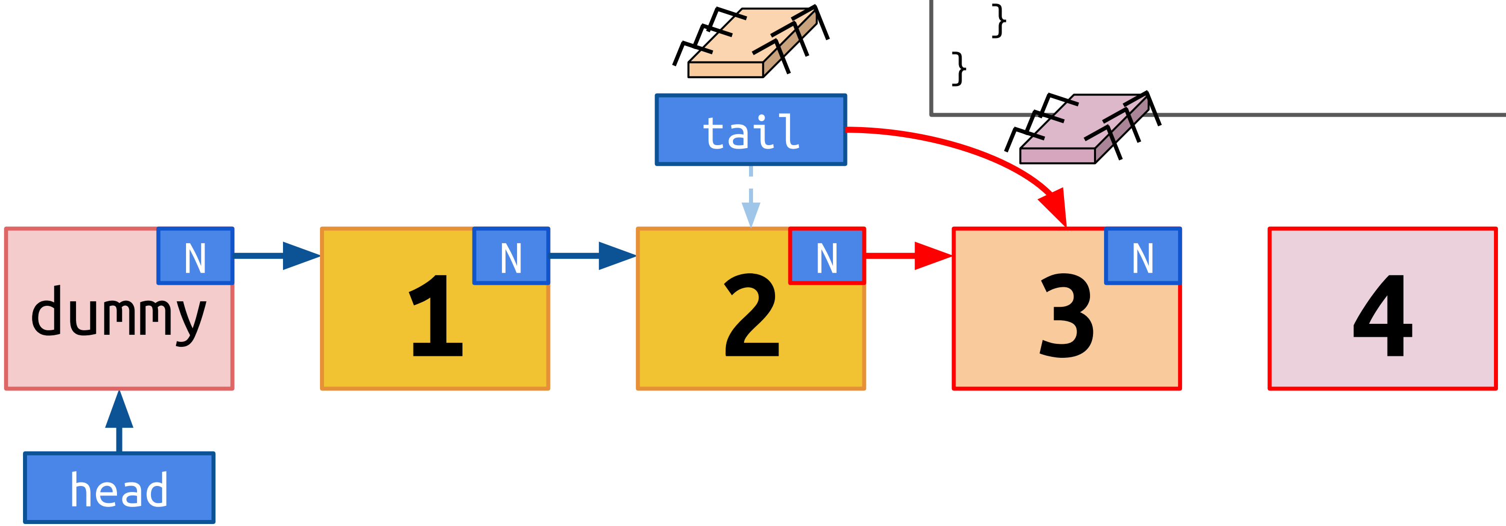
```
fun enqueue(e: E) = while (true) {
  node := MSQueueNode(e)
  cur_tail := tail
  if CAS(&cur_tail.next, null, node) {
    CAS(&tail, cur_tail, node)
    return
  } else {
    CAS(&tail, cur_tail, cur_tail.next)
  }
}
```



MS Queue: Enqueue

0. Create a new node with the element
1. Update the `next` pointer of the current `tail`
2. Update the `tail` pointer

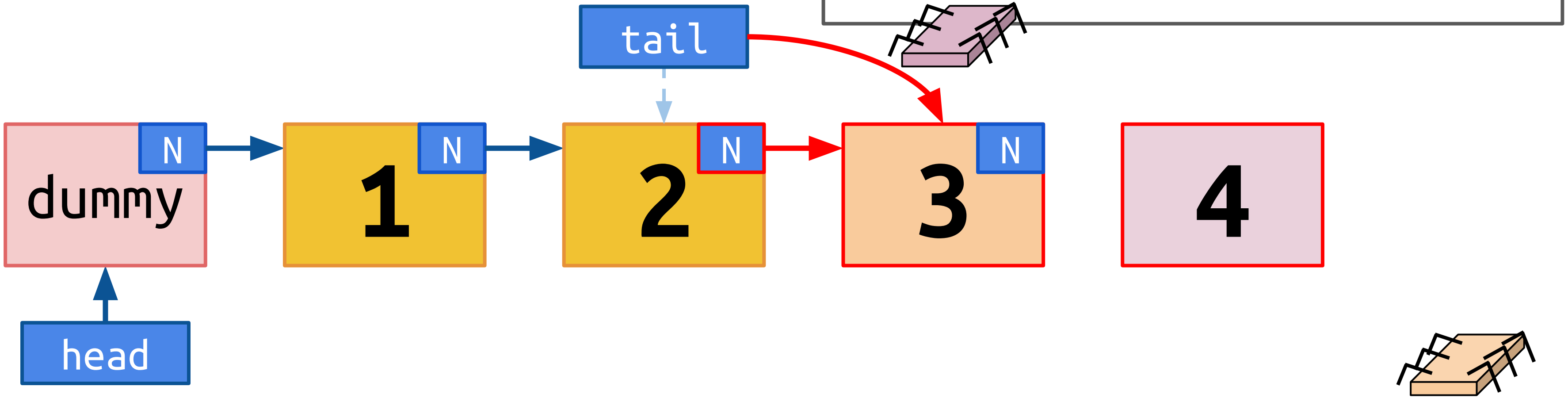
```
fun enqueue(e: E) = while (true) {
  node := MSQueueNode(e)
  cur_tail := tail
  if CAS(&cur_tail.next, null, node) {
    CAS(&tail, cur_tail, node)
    return
  } else {
    CAS(&tail, cur_tail, cur_tail.next)
  }
}
```



MS Queue: Enqueue

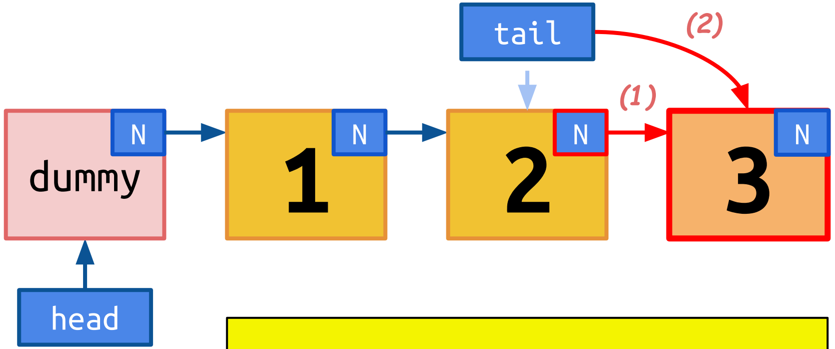
0. Create a new node with the element
1. Update the `next` pointer of the current `tail`
2. Update the `tail` pointer

```
fun enqueue(e: E) = while (true) {
  node := MSQueueNode(e)
  cur_tail := tail
  if CAS(&cur_tail.next, null, node) {
    CAS(&tail, cur_tail, node)
    return
  } else {
    CAS(&tail, cur_tail, cur_tail.next)
  }
}
```



MS Queue: Enqueue

0. Create a new node with the element
1. Update the `next` pointer of the current `tail`
2. Update the `tail` pointer

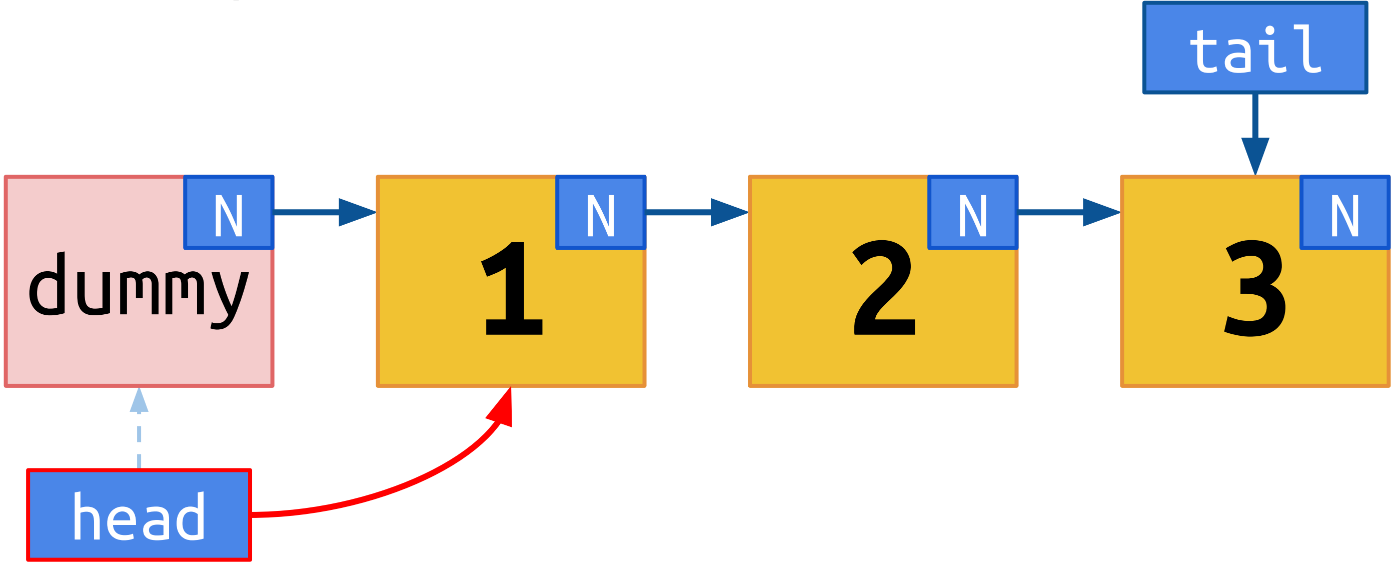


Is enqueue lock-free?

```
fun enqueue(e: E) = while (true) {
(0) node := MSQueueNode(e)
    cur_tail := tail
(1) if CAS(&cur_tail.next, null, node) {
  (2) CAS(&tail, cur_tail, node)
      return
    } else {
  (2) CAS(&tail, cur_tail, cur_tail.next)
    }
}
```

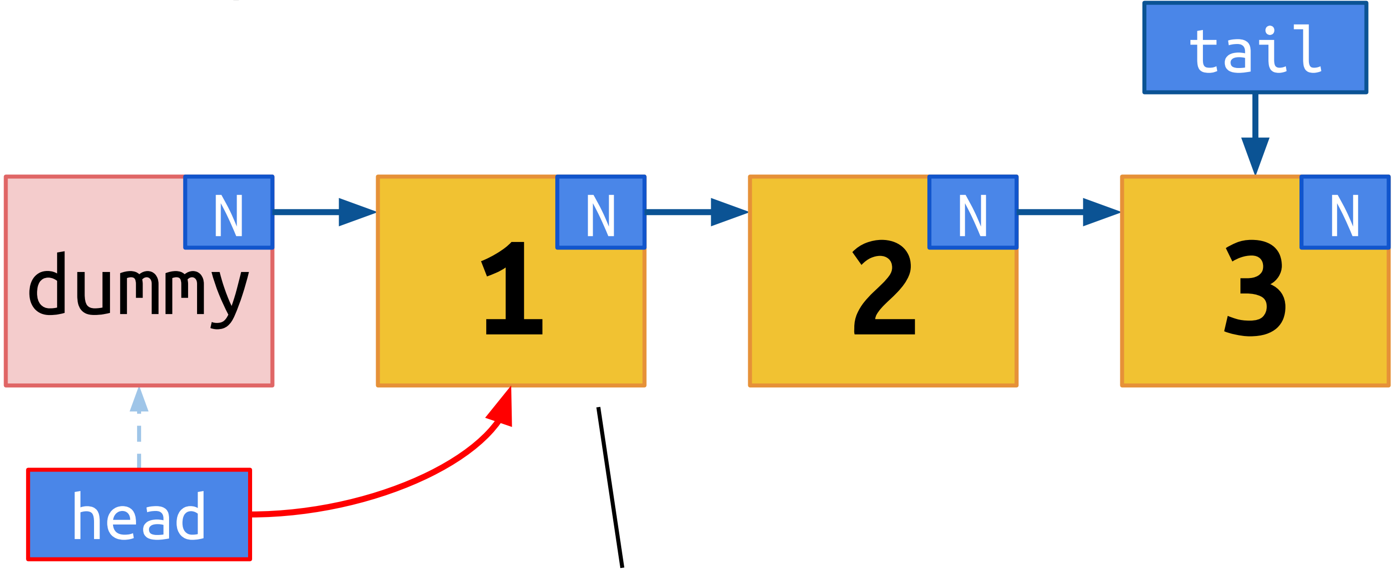
MS Queue: Dequeue

Move the `head` pointer forward



MS Queue: Dequeue

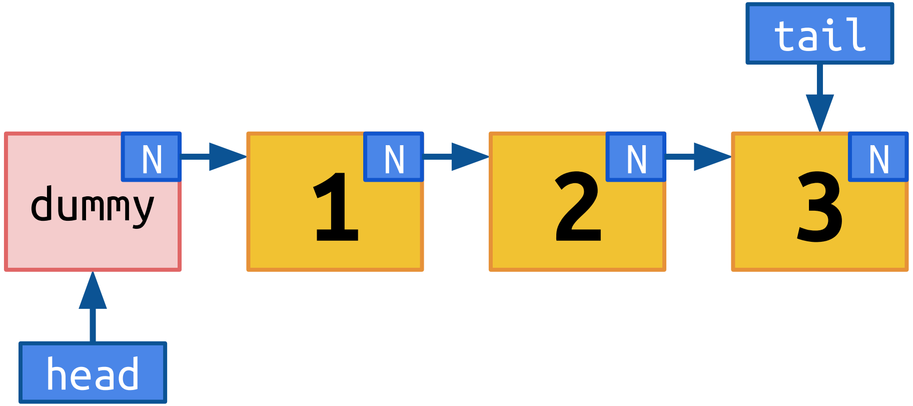
Move the `head` pointer forward



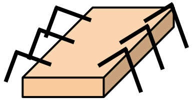
New “dummy” node

MS Queue: Dequeue

Move the `head` pointer forward

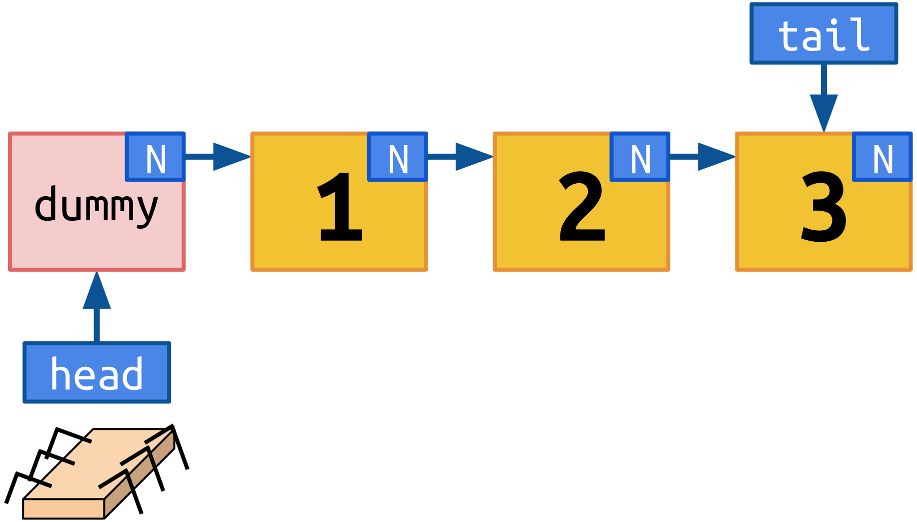


```
fun dequeue(): E {
  while (true) {
    cur_head := head
    cur_head_next := cur_head.next
    if cur_head_next === null:
      throw EmptyQueueException()
    if CAS(&head, cur_head, cur_head_next):
      return cur_head_next.value
  }
}
```



MS Queue: Dequeue

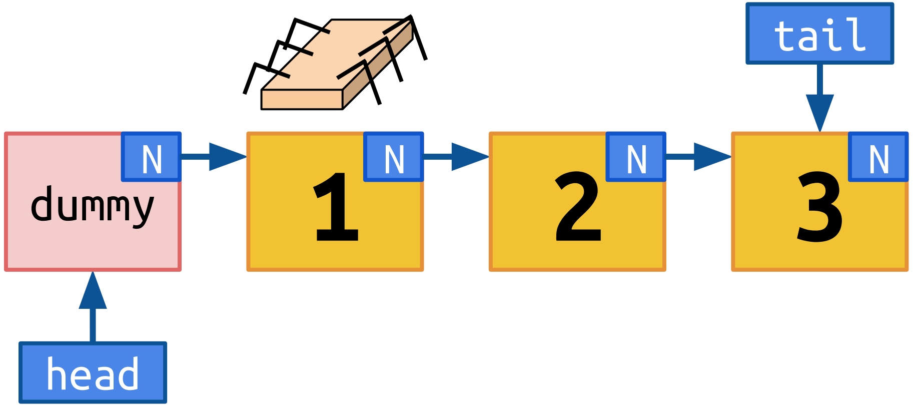
Move the `head` pointer forward



```
fun dequeue(): E {
  while (true) {
    cur_head := head
    cur_head_next := cur_head.next
    if cur_head_next === null:
      throw EmptyQueueException()
    if CAS(&head, cur_head, cur_head_next):
      return cur_head_next.value
  }
}
```

MS Queue: Dequeue

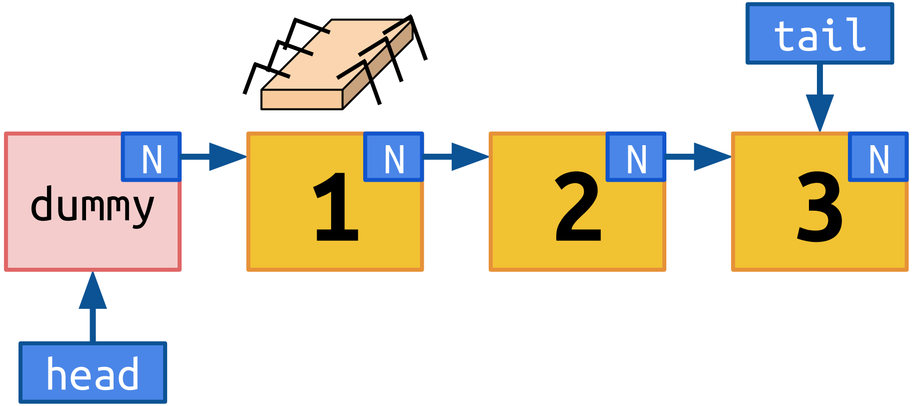
Move the head pointer forward



```
fun dequeue(): E {
  while (true) {
    cur_head := head
    cur_head_next := cur_head.next
    if cur_head_next === null:
      throw EmptyQueueException()
    if CAS(&head, cur_head, cur_head_next):
      return cur_head_next.value
  }
}
```

MS Queue: Dequeue

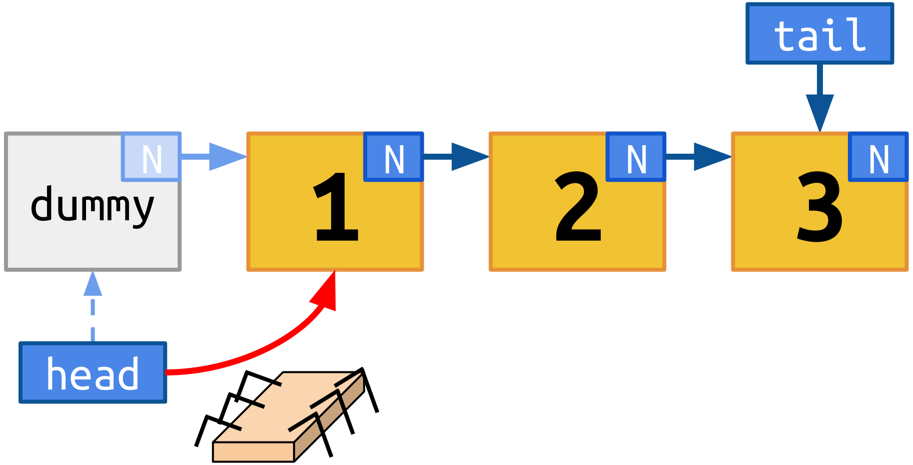
Move the `head` pointer forward



```
fun dequeue(): E {
  while (true) {
    cur_head := head
    cur_head_next := cur_head.next
    if cur_head_next === null:
      throw EmptyQueueException()
    if CAS(&head, cur_head, cur_head_next):
      return cur_head_next.value
  }
}
```

MS Queue: Dequeue

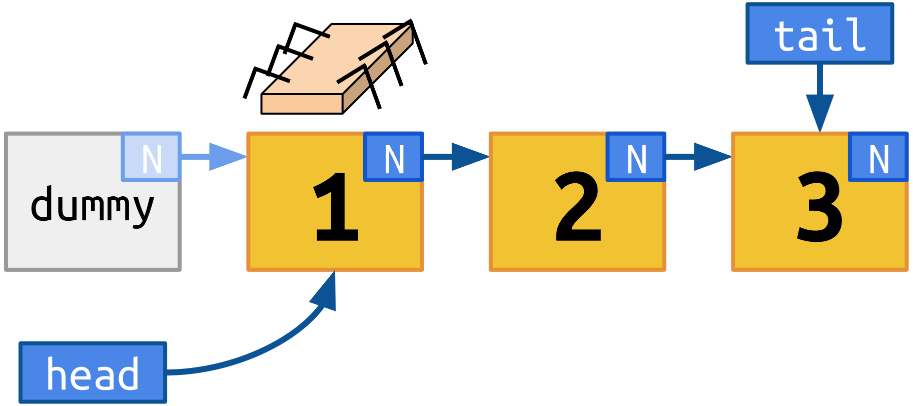
Move the `head` pointer forward



```
fun dequeue(): E {
  while (true) {
    cur_head := head
    cur_head_next := cur_head.next
    if cur_head_next === null:
      throw EmptyQueueException()
    if CAS(&head, cur_head, cur_head_next):
      return cur_head_next.value
  }
}
```

MS Queue: Dequeue

Move the `head` pointer forward



```
fun dequeue(): E {
  while (true) {
    cur_head := head
    cur_head_next := cur_head.next
    if cur_head_next === null:
      throw EmptyQueueException()
    if CAS(&head, cur_head, cur_head_next):
      return cur_head_next.value
  }
}
```

To Sum Up

- Coarse-Grained Locking
- Universal Lock-Free Construction

- Linearizability
- Lock- and Wait-Freedom

- Lock-Free Treiber Stack Algorithm
- Michael-Scott Queue Algorithm

Task Assignments

`github.com/ndkoval/Hydra2022`

Thank you!