Lock-free algorithms for Kotlin coroutines

It is all about scalability

Presented at SPTCC 2017

/Roman Elizarov @ JetBrains

Speaker: Roman Elizarov

- 16+ years experience
- Previously developed high-perf trading software @ Devexperts
- Teach concurrent & distributed programming @ St. Petersburg ITMO University
- Chief judge @ Northeastern European Region of ACM ICPC
- Now work on Kotlin @ JetBrains

Agenda

- Kotlin coroutines overview & motivation for lock-free algorithms
- Lock-free doubly linked list
- Lock-free multi-word compare-and-swap
- Combining them to get more complex atomic operations (without STM)

Kotlin basic facts

- Kotlin is a JVM language developed by JetBrains
- General purpose and statically-typed
- Object-oriented and functional paradigms
- Open source under Apache 2.0
- Reached version 1.0 in 2016
  - Compatibility commitment
  - Now at version 1.1
- Officially supported by Google on Android

Kotlin is …

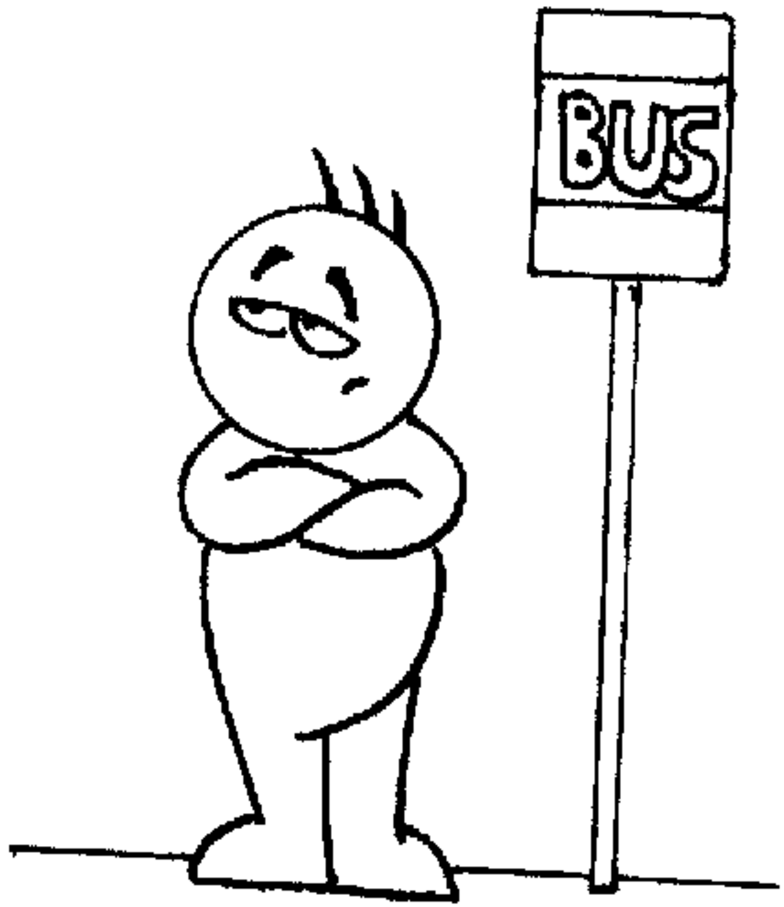
- Modern
- Concise
- Safe
- Extensible
- Pragmatic
- Fun to work with!

Kotlin is pragmatic

… and easy to learn

Coroutines

Asynchronous programming made easy



How do we write code that waits for something most of the time?

Blocking threads

Kotlin

```kotlin
fun postItem(item: Item) {
    val token = requestToken()
    val post = submitPost(token, item)
    processPost(post)
}
```

Callbacks

Kotlin

```
fun postItem(item: Item) {
    requestToken { token ->
        submitPost(token, item) { post ->
            processPost(post)
        }
    }
}
```

Futures/Promises/Rx

Kotlin

```
fun postItem(item: Item) {
    requestToken()
        .thenCompose { token ->
            submitPost(token, item)
        }
        .thenAccept { post ->
            processPost(post)
        }
}
```

Coroutines

Kotlin

```
fun postItem(item: Item) {
    launch(CommonPool) {
        val token = requestToken()
        val post = submitPost(token, item)
        processPost(post)
    }
}
```

CSP & Actor models

- A *style* of programming for modern systems
- Lots of concurrent tasks / jobs
  - Waiting most of the time
  - Communicating all the time

Share data by communicating

Kotlin coroutines primitives

- Jobs/Deferreds (futures)
  - join/await
- Channels
  - send & receive
  - synchronous & buffered channels
- Select/alternatives
  - Atomically wait on multiple events
- Cancellation
- Parent-child hierarchies

Implementation challenges

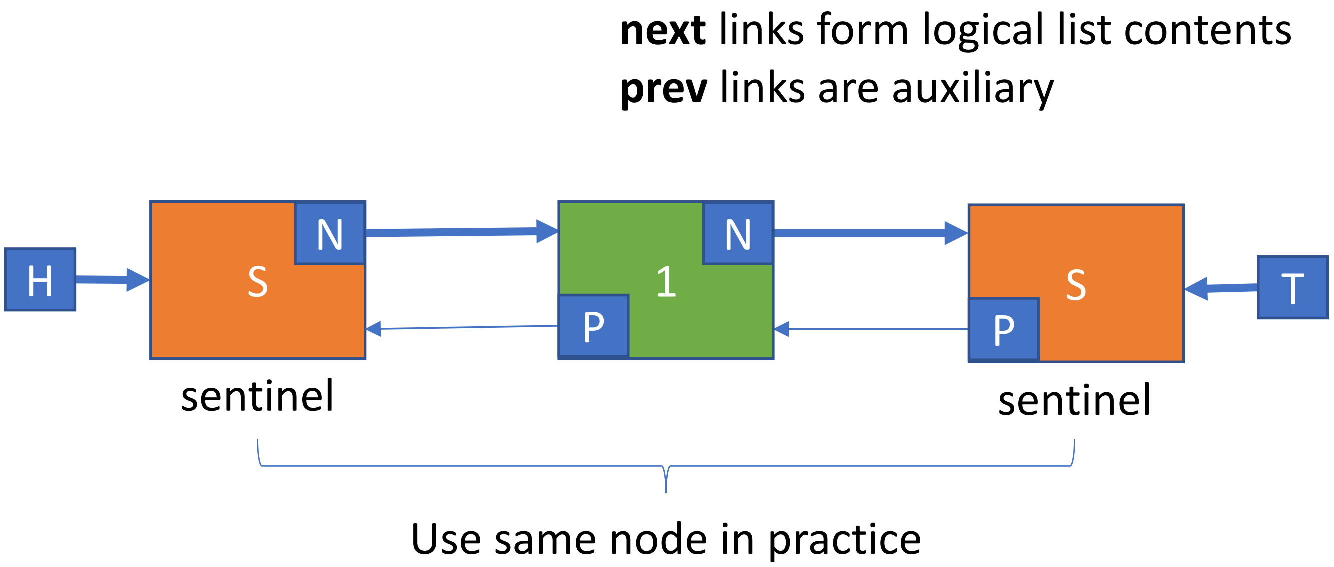
- Coroutines are like light-weight threads
- All the *low-level* scheduling & communication mechanisms have to *scale* to lots of coroutines

Lock-free algorithms

Building blocks

- Single-word CAS (that’s all we have on JVM)
- Automatic memory management (GC)
- *Practical* lock-free algorithms
  - Lock-Free and Practical **Doubly Linked List**-Based Deques Using Single-Word Compare-and-Swap by Sundell and Tsigas
  - A Practical **Multi-Word Compare-and-Swap** Operation by Timothy L. Harris, Keir Fraser and Ian A. Pratt.

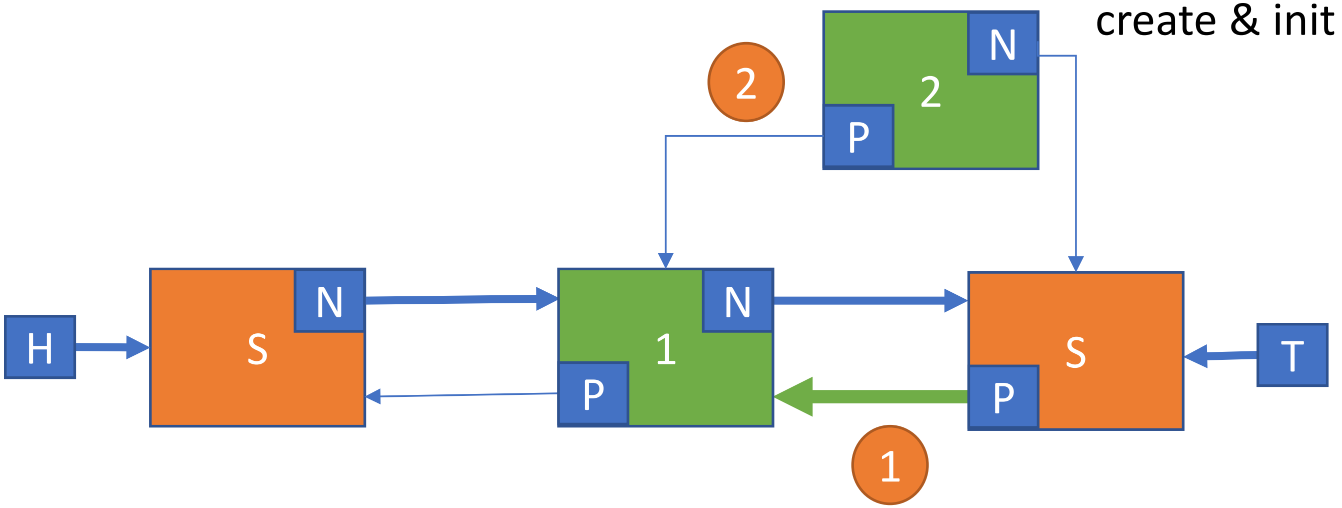
Doubly linked list



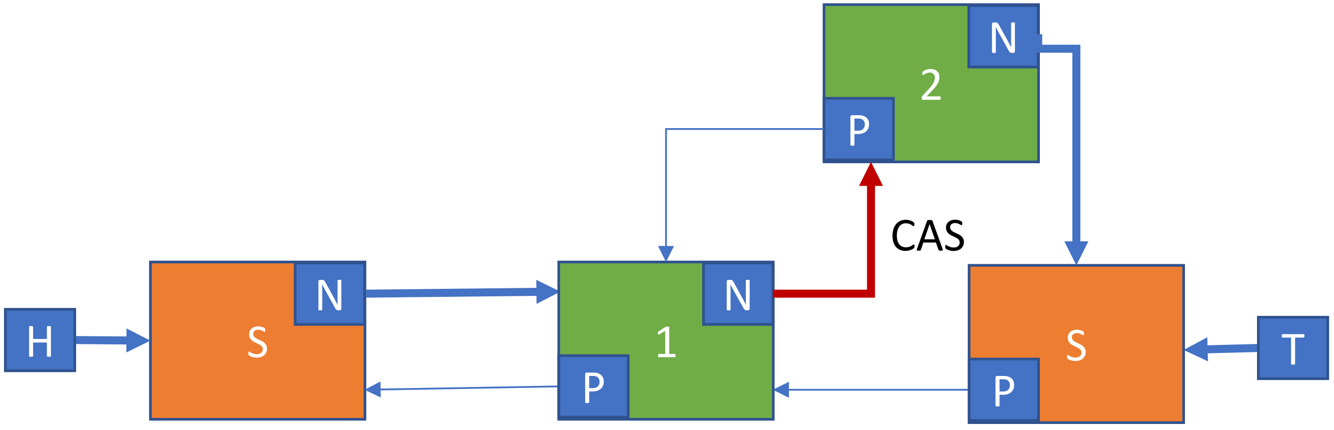
Insert

PushRight (like in queue)

Doubly linked list (insert 0)



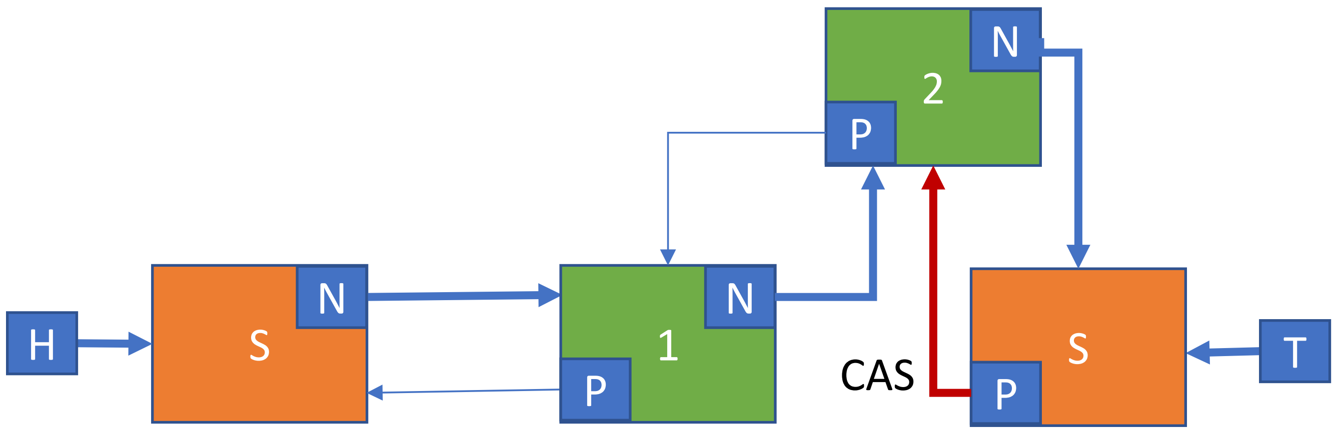
Doubly linked list (insert 1)



Retry insert on CAS failure

Doubly linked list (insert 2)

"finish insert"

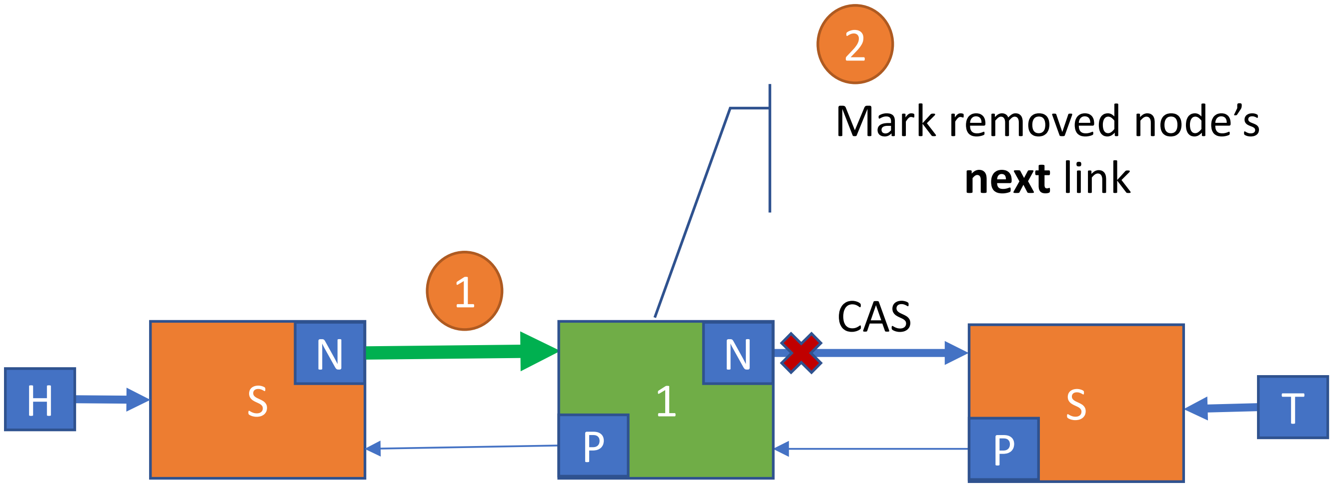


Ignore CAS failure

Remove

PopLeft (like in queue)

Doubly linked list (remove 1)



Retry remove on CAS failure

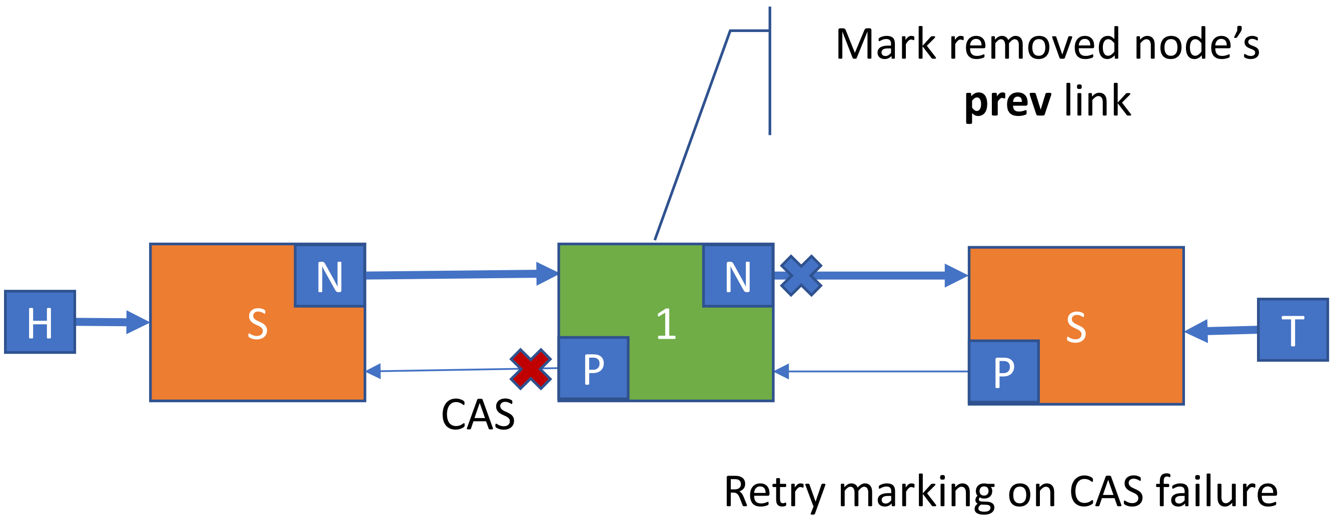
Use *wrapper* object for mark in practice

Don't use **AtomicMarkableReference**

Cache wrappers in pointed-to nodes

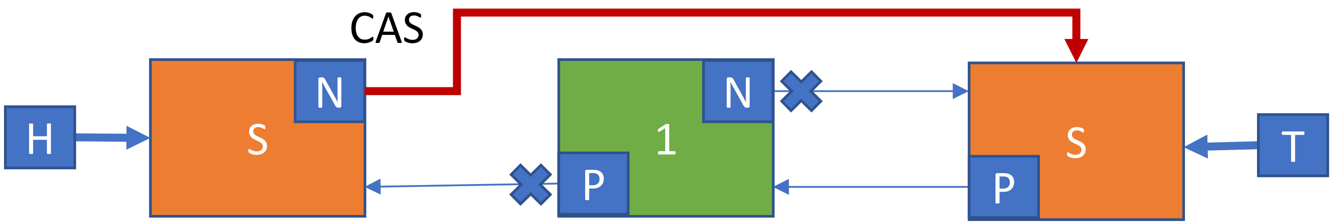
Doubly linked list (remove 2)

”finish remove”



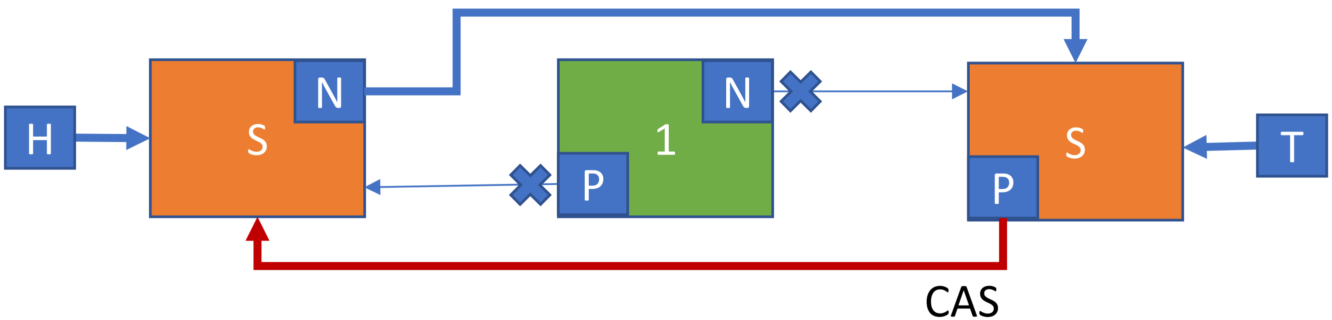
Doubly linked list (remove 3)

”help remove” – fixup **next** links



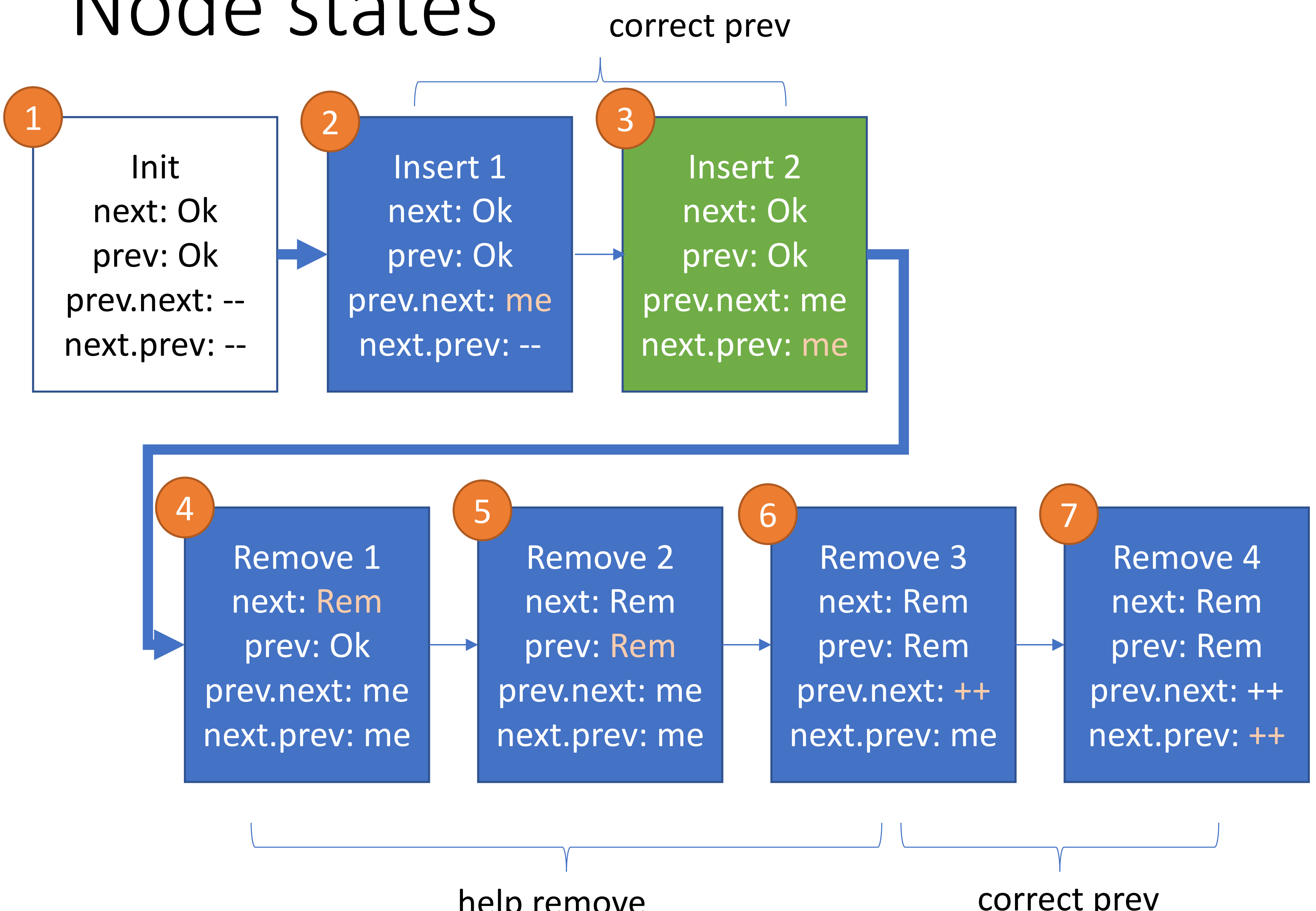
Doubly linked list (remove 4)

”correct prev” – fixup **prev** links



State transitions

Node states

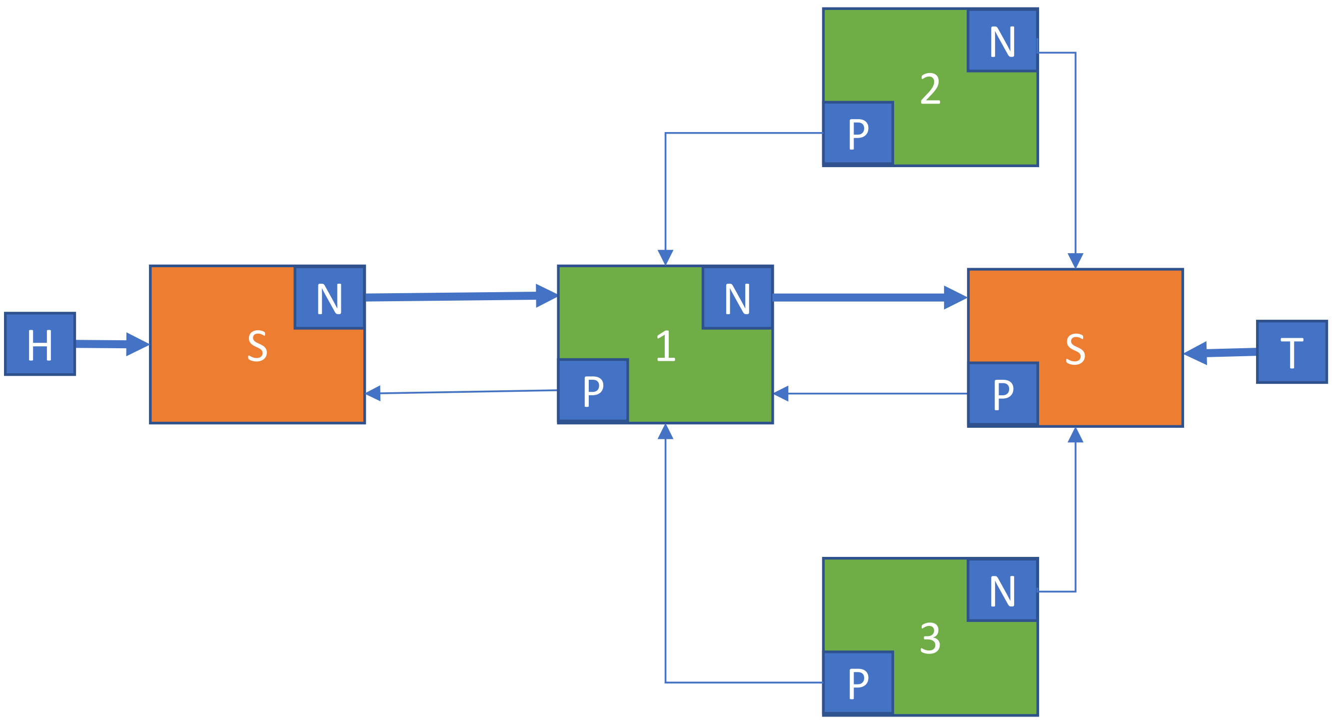

correct prev
1
Init
next: Ok
prev: Ok
prev.next: --
next.prev: --
2
Insert 1
next: Ok
prev: Ok
prev.next: me
next.prev: --
3
Insert 2
next: Ok
prev: Ok
prev.next: me
next.prev: me
4
Remove 1
next: Rem
prev: Ok
prev.next: me
next.prev: me
5
Remove 2
next: Rem
prev: Rem
prev.next: me
next.prev: me
6
Remove 3
next: Rem
prev: Rem
prev.next: ++
next.prev: me
7
Remove 4
next: Rem
prev: Rem
prev.next: ++
next.prev: ++
help remove
correct prev

Helping

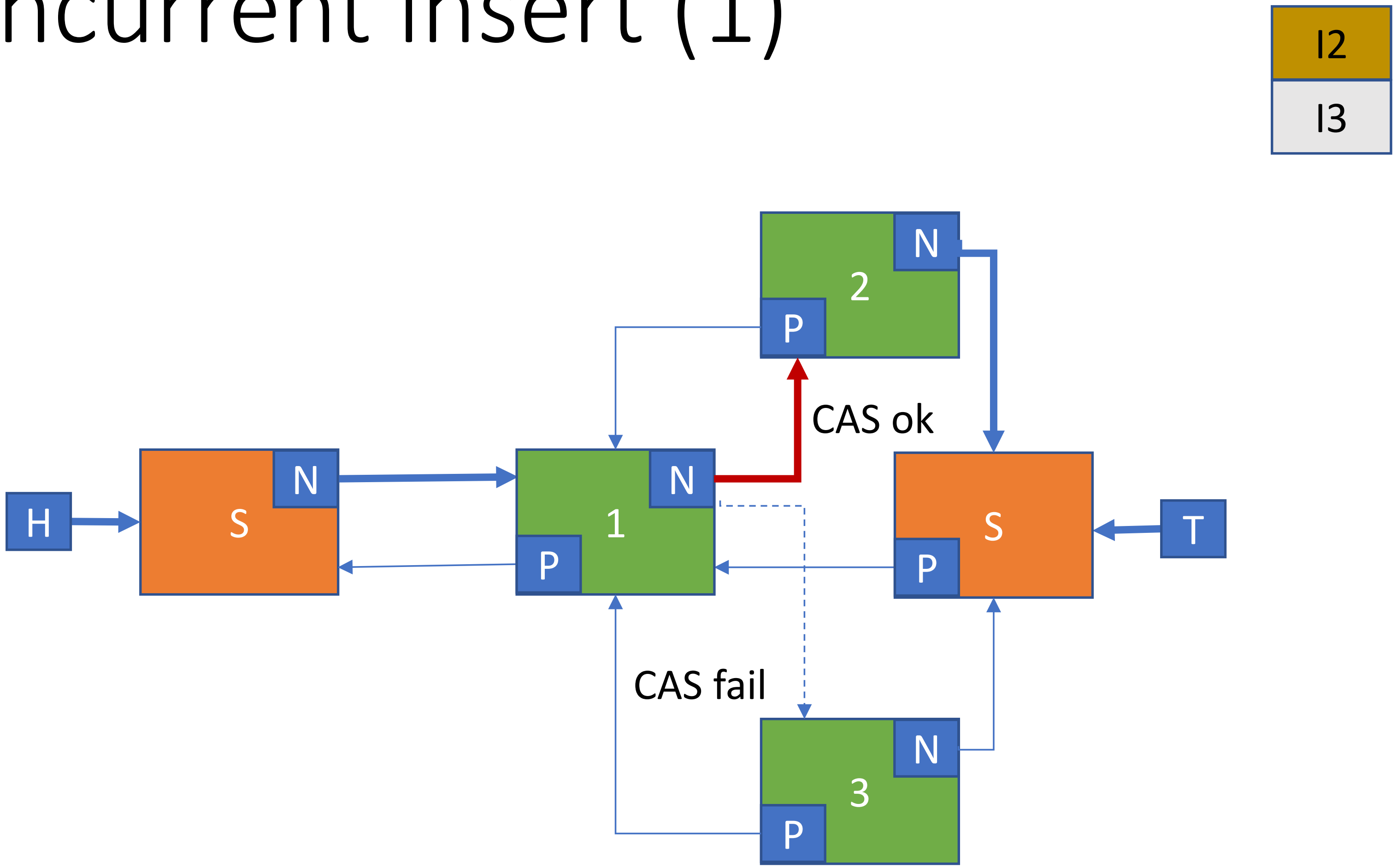
Concurrent insert

Concurrent insert (0)

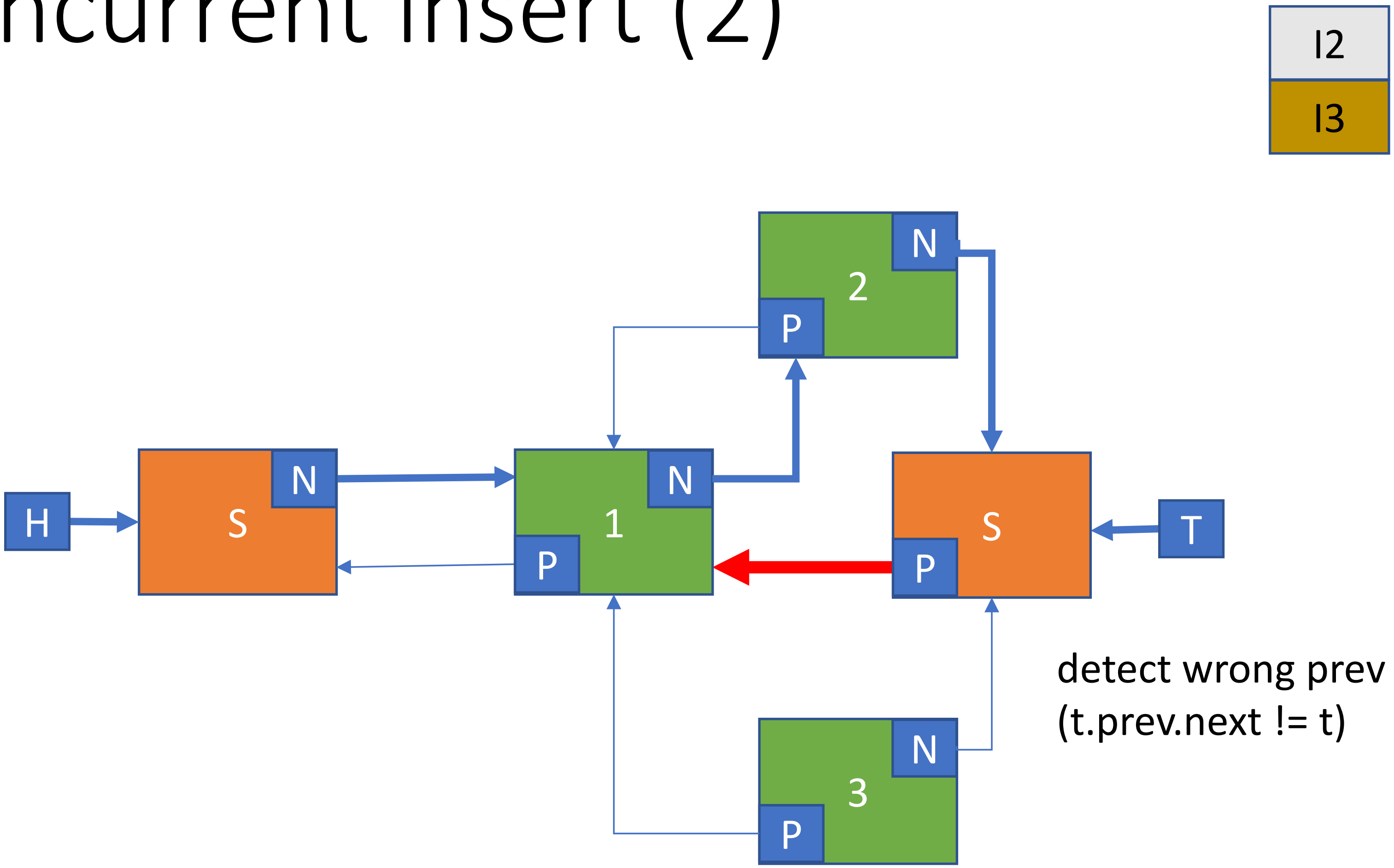
I2
I3



Concurrent insert (1)

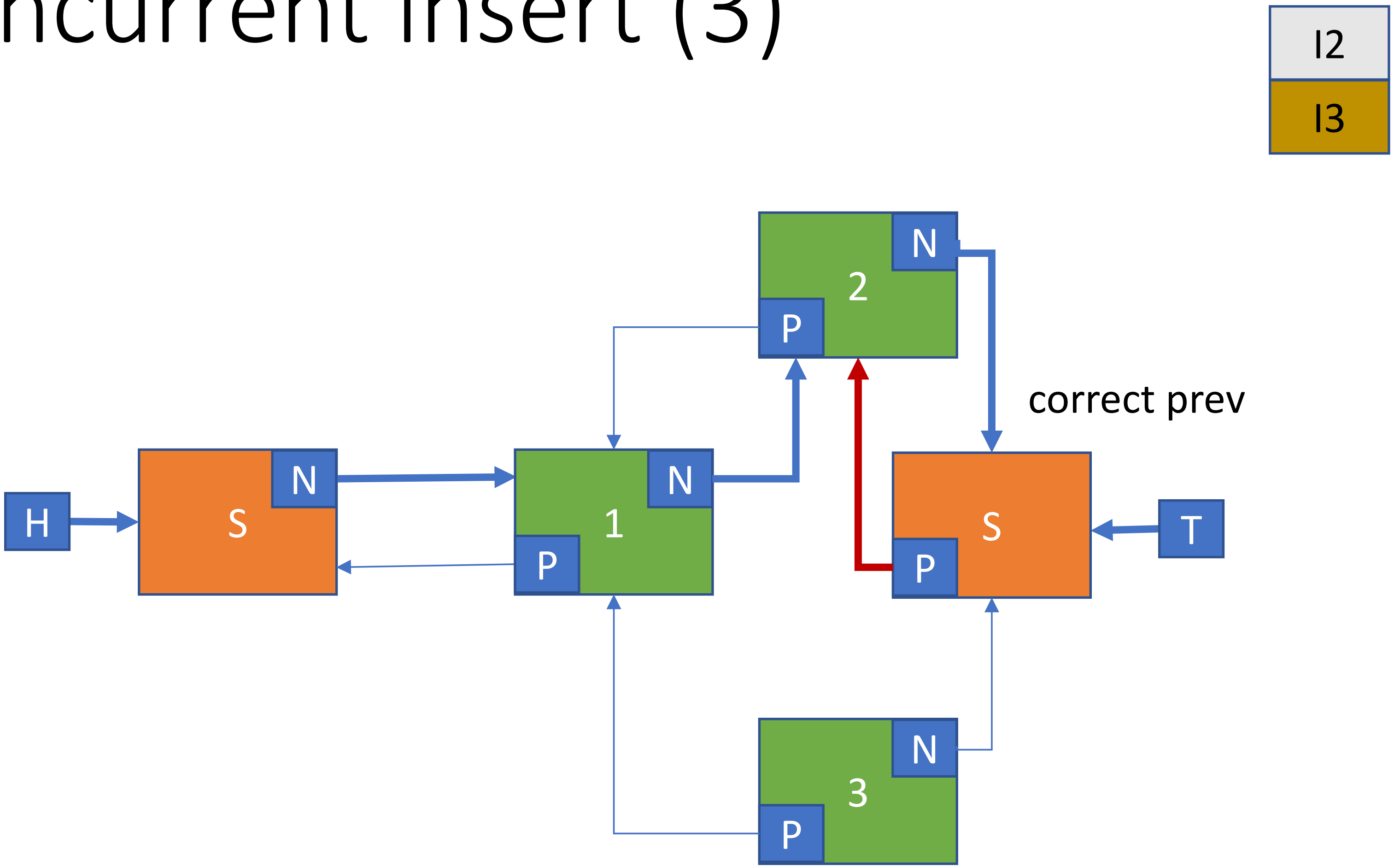


Concurrent insert (2)

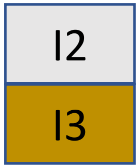


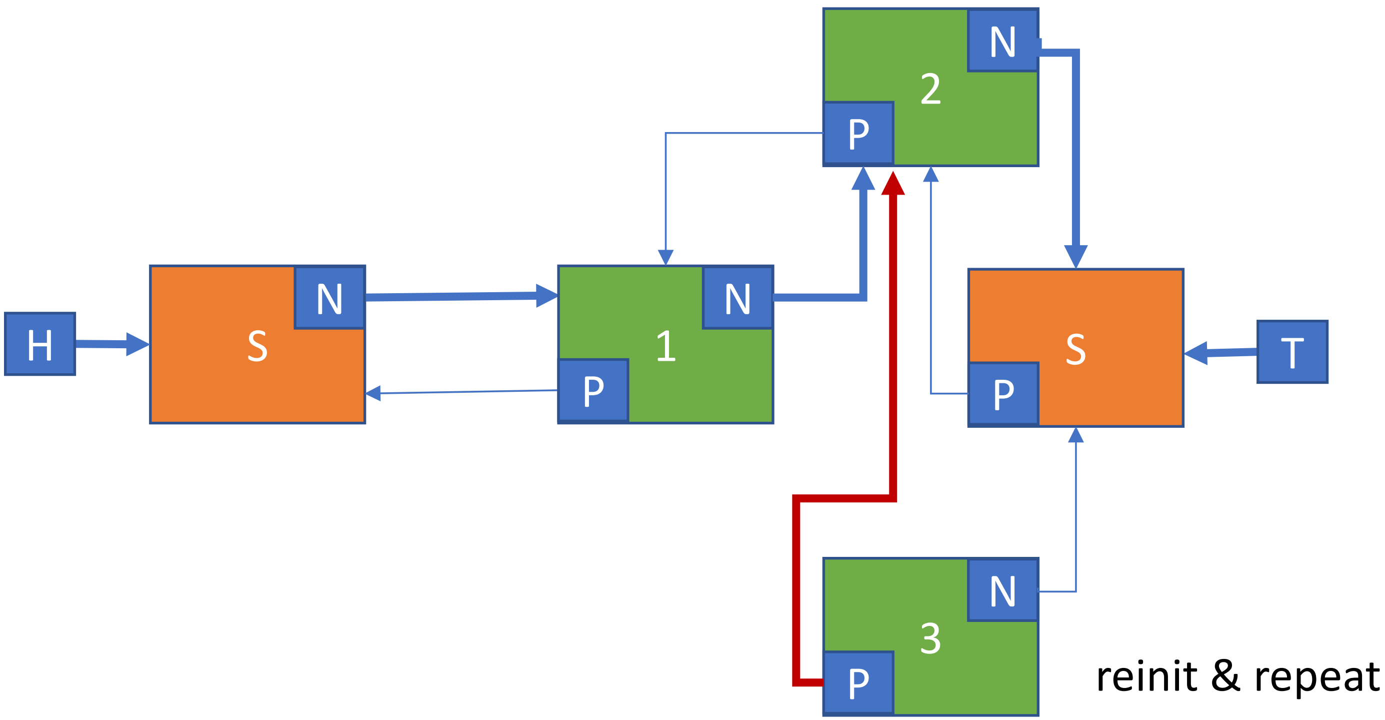
detect wrong prev
(t.prev.next != t)

Concurrent insert (3)



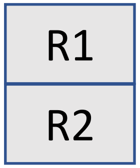
Concurrent insert (4)

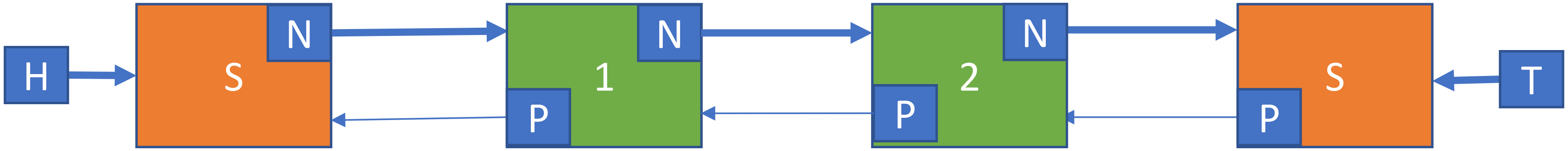




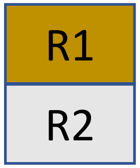
Concurrent remove

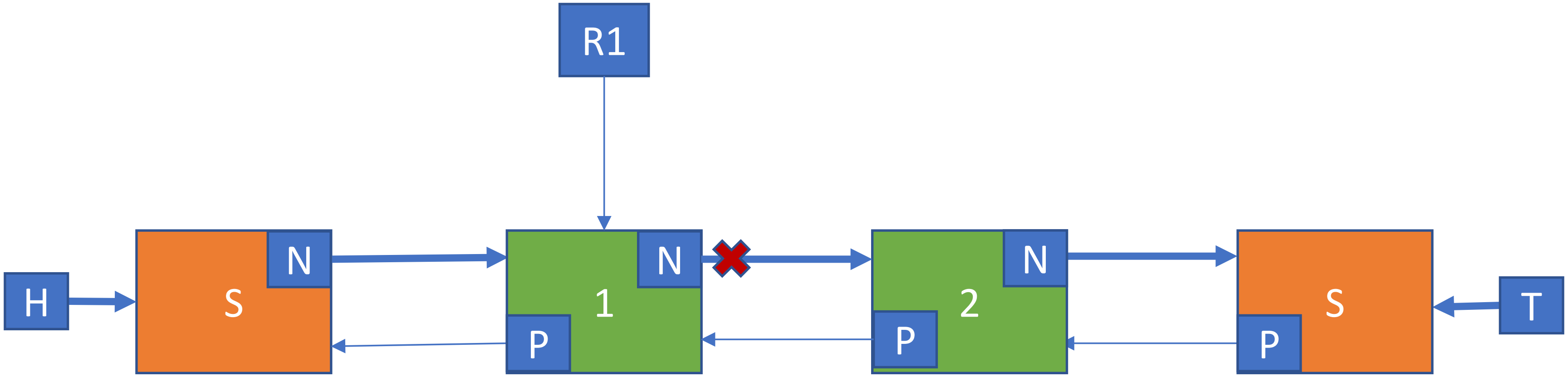
Concurrent remove (0)



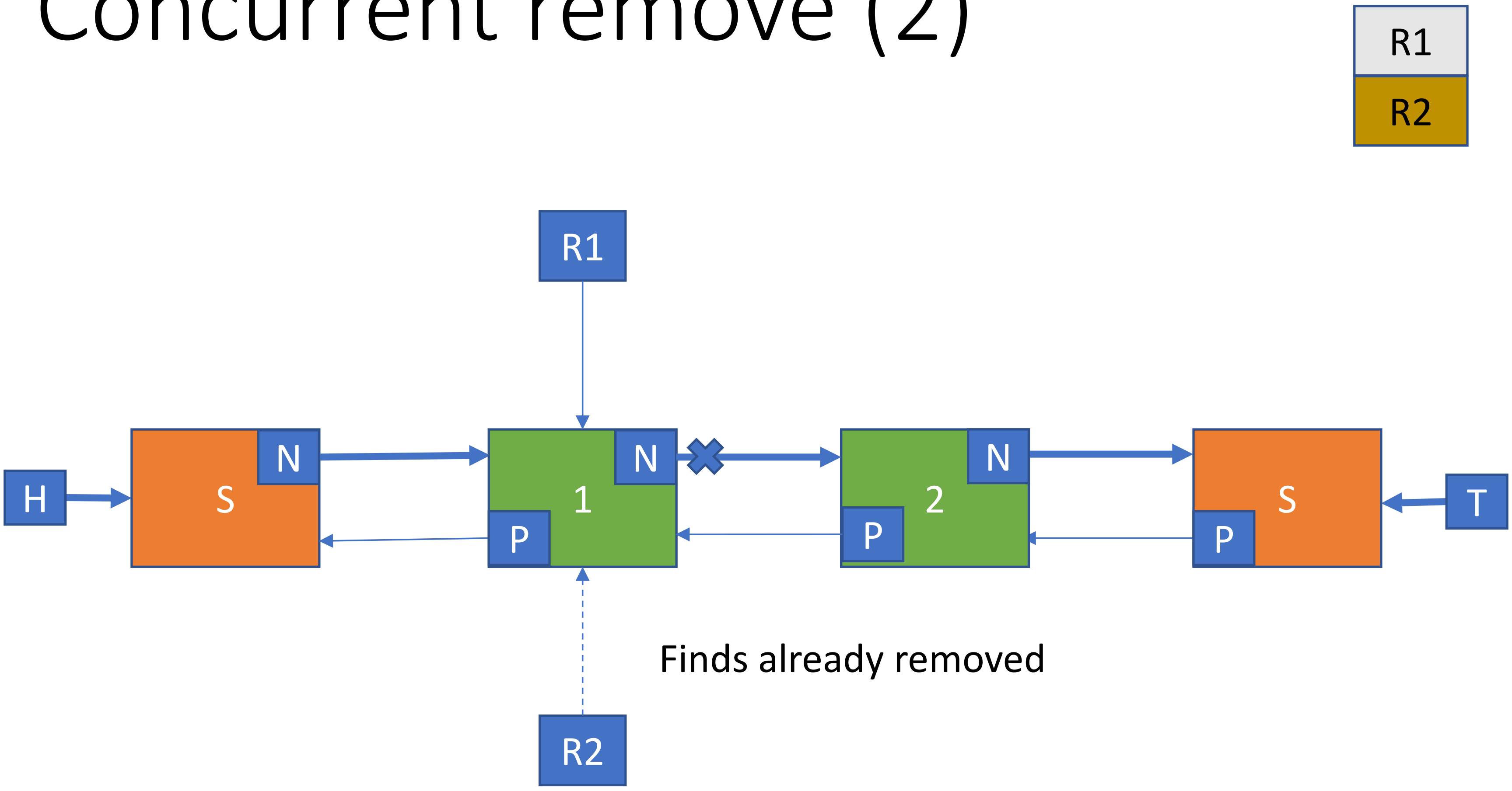


Concurrent remove (1)

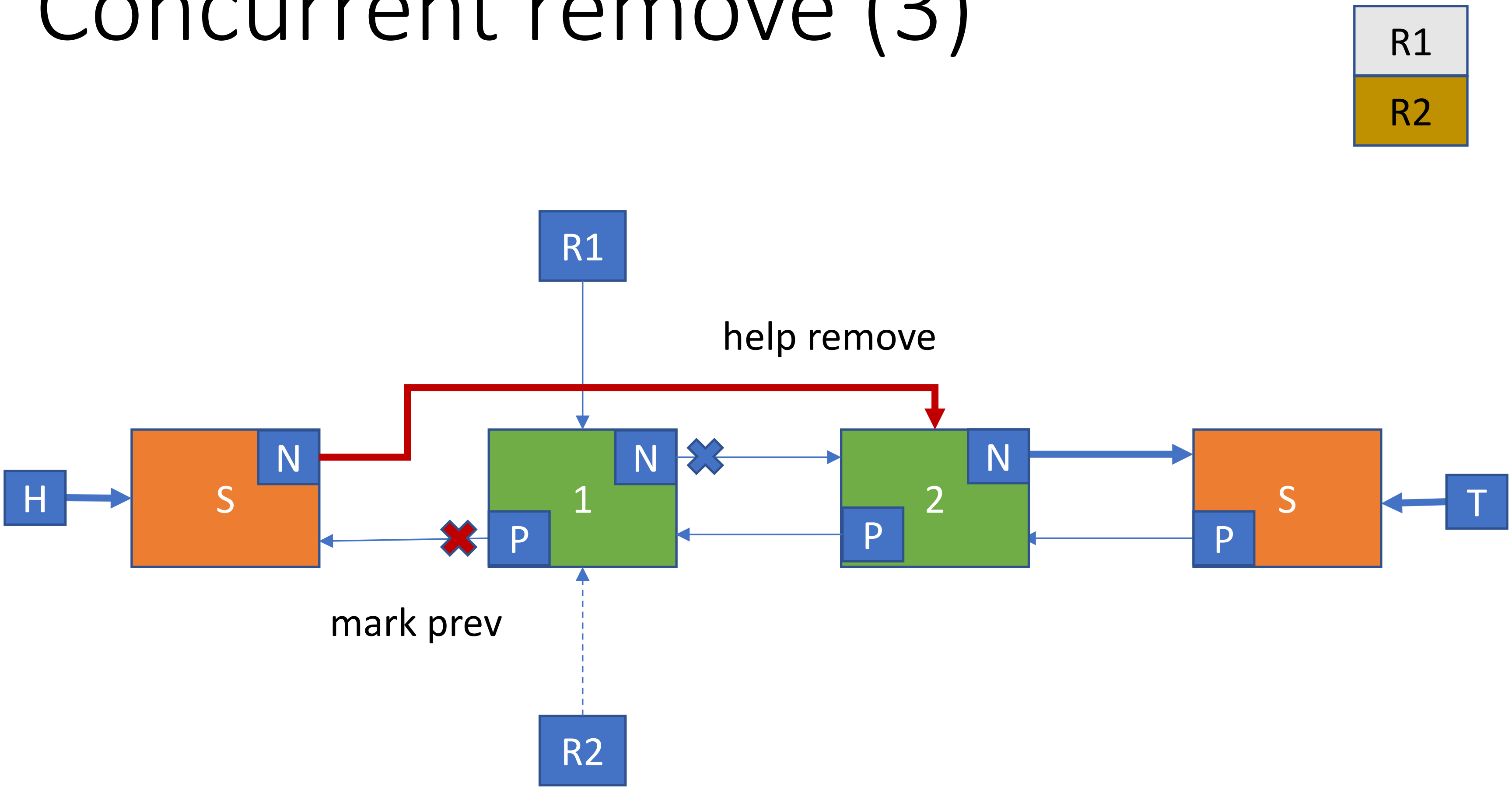




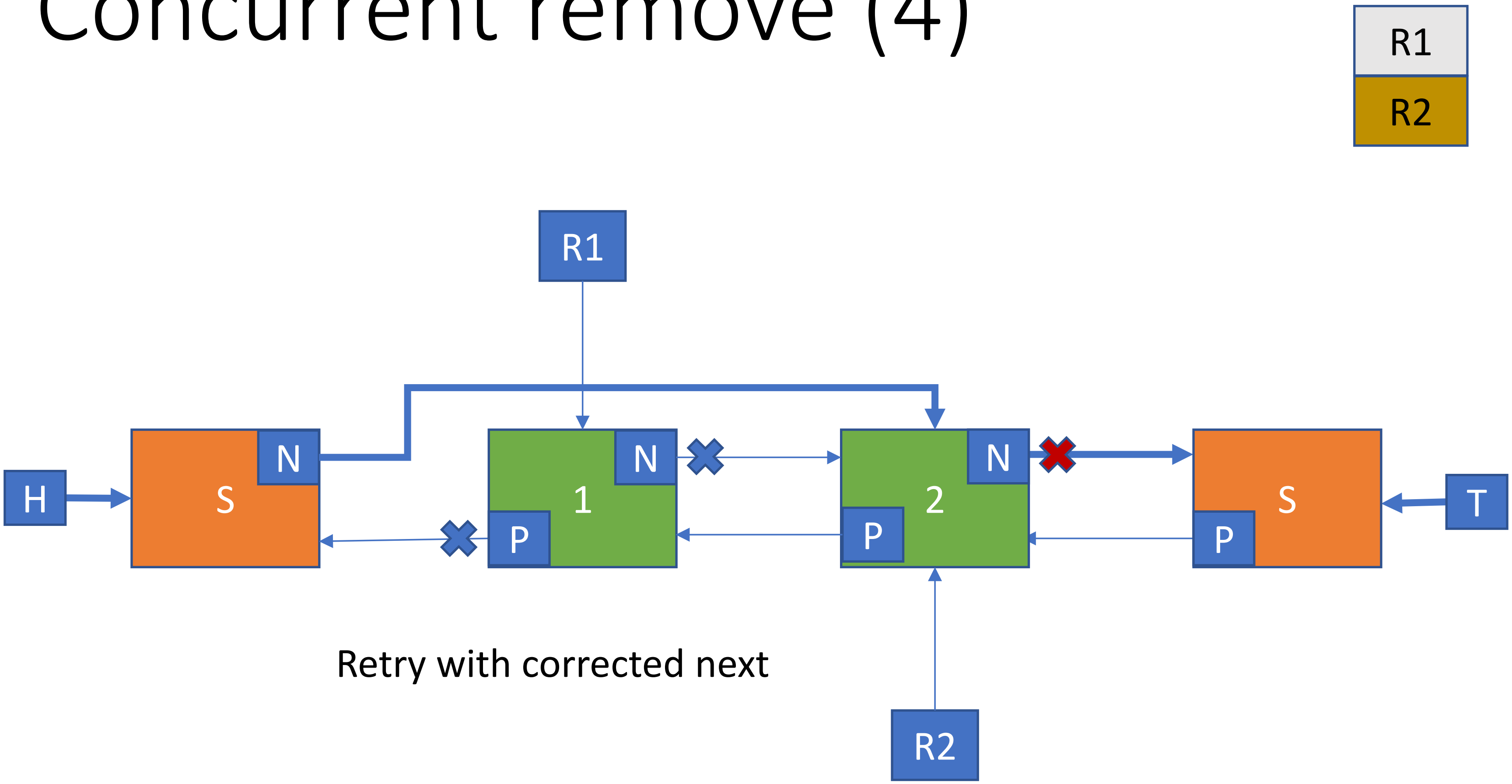
Concurrent remove (2)



Concurrent remove (3)

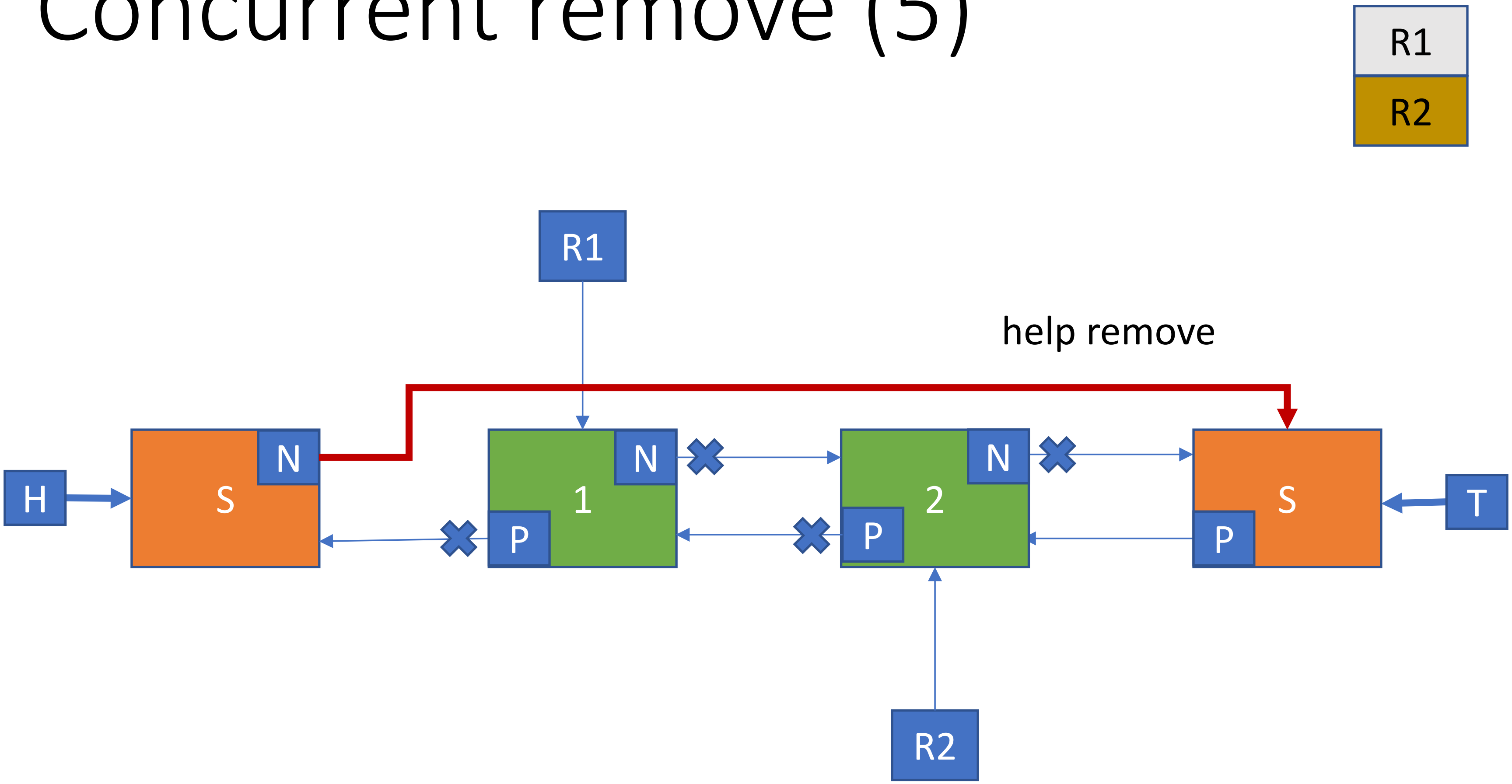


Concurrent remove (4)

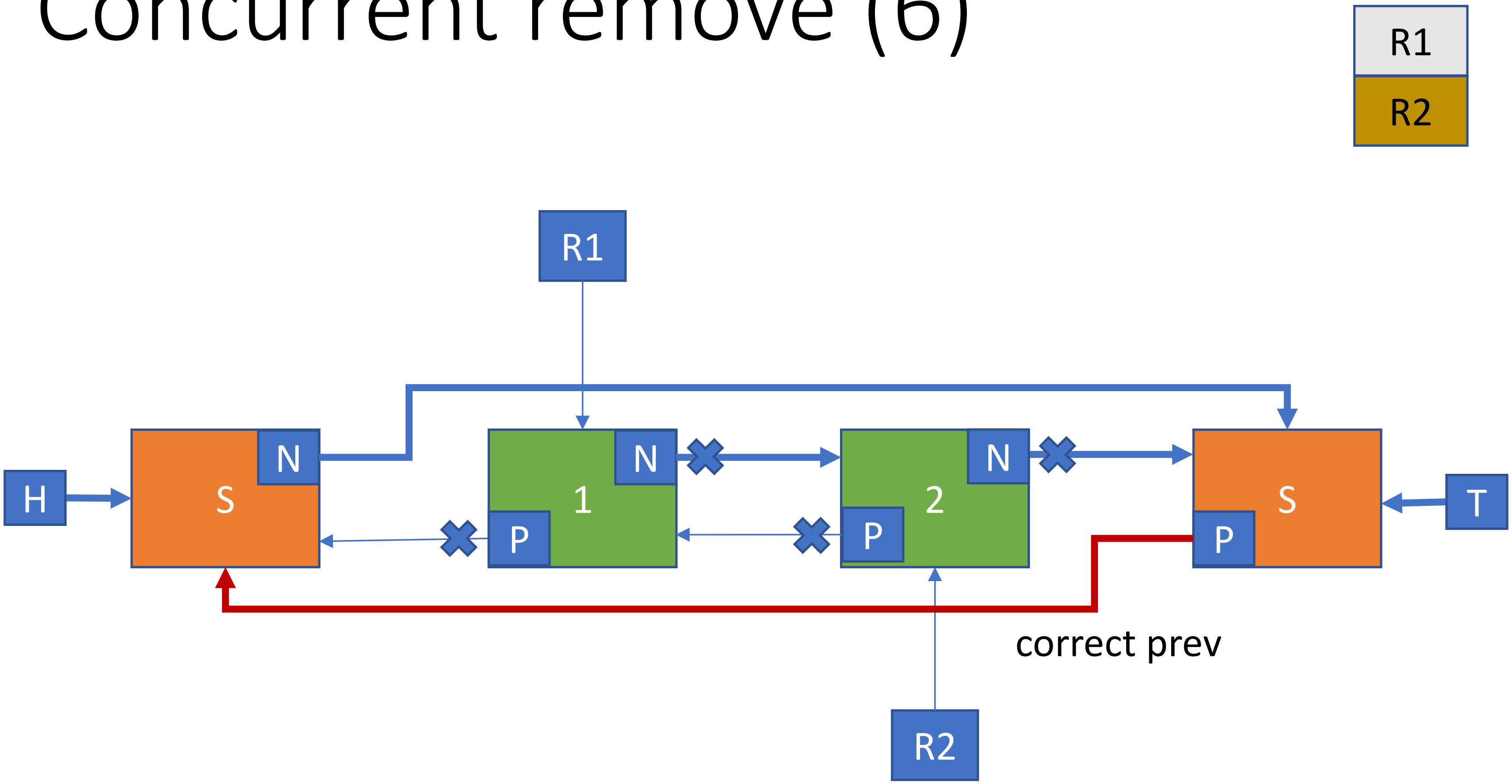


Retry with corrected next

Concurrent remove (5)



Concurrent remove (6)

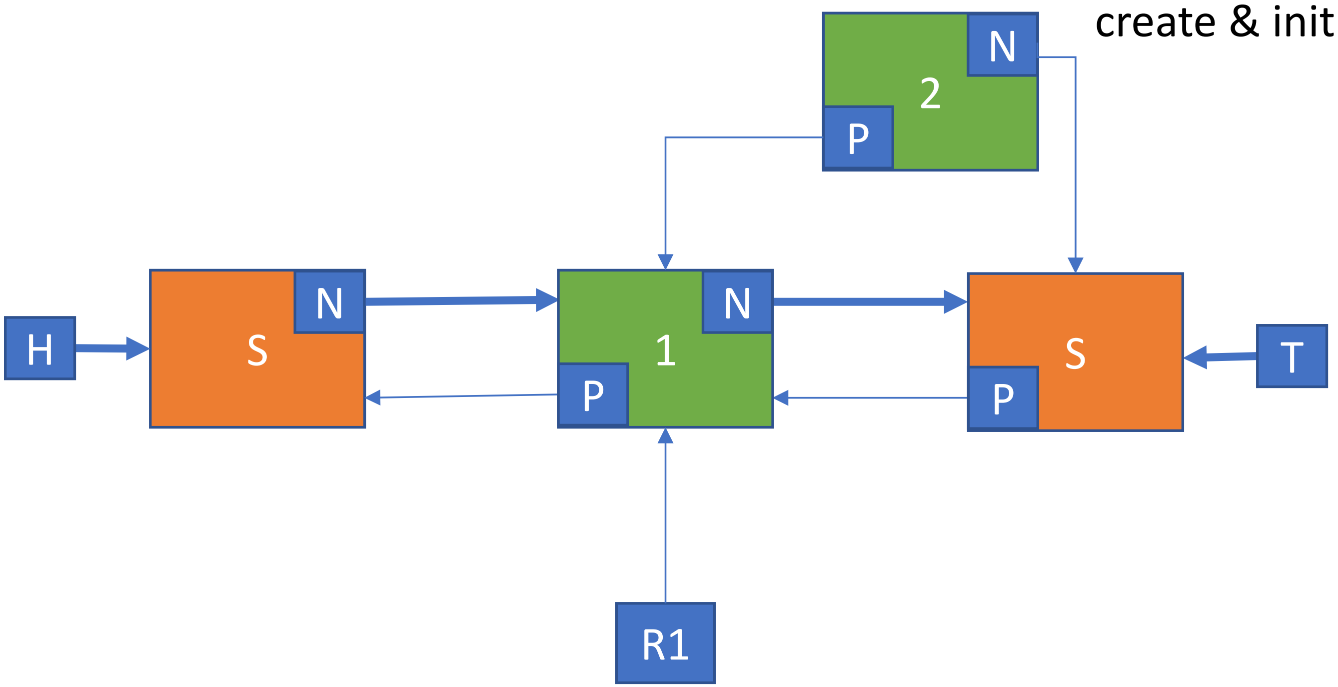


Concurrent remove & insert

When remove wins

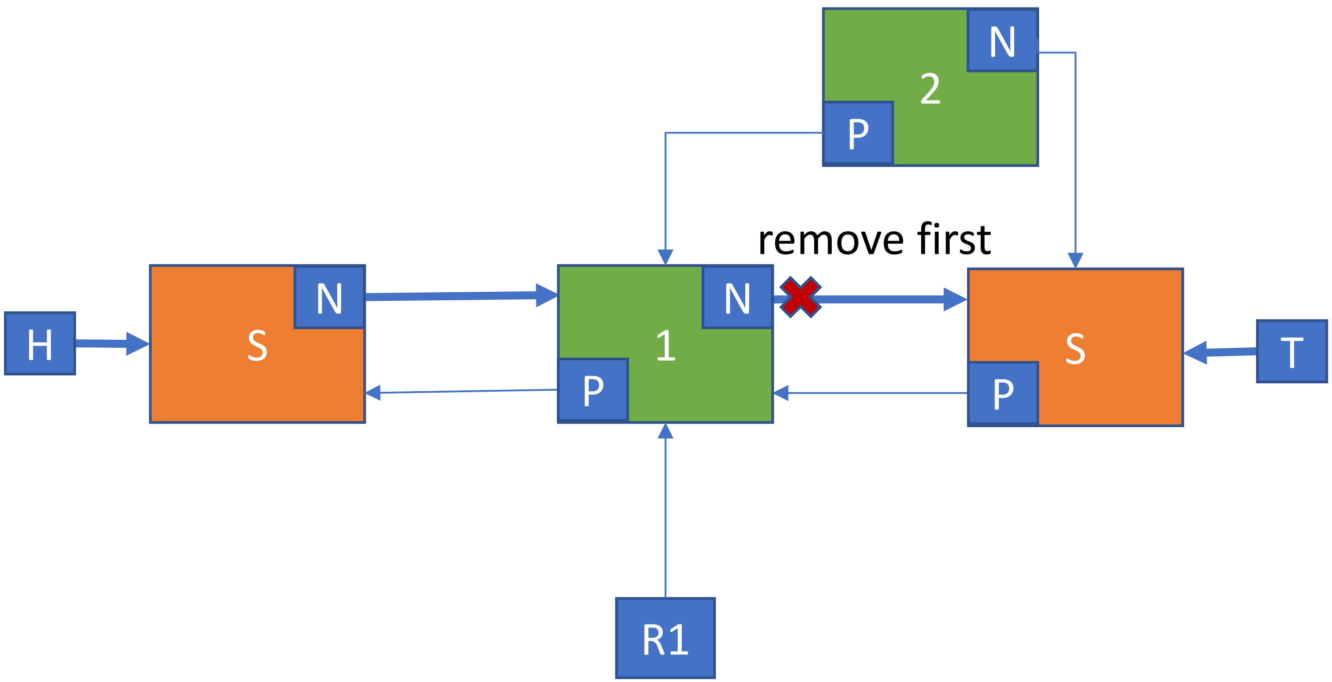
Concurrent remove & insert (0)

R1
I2

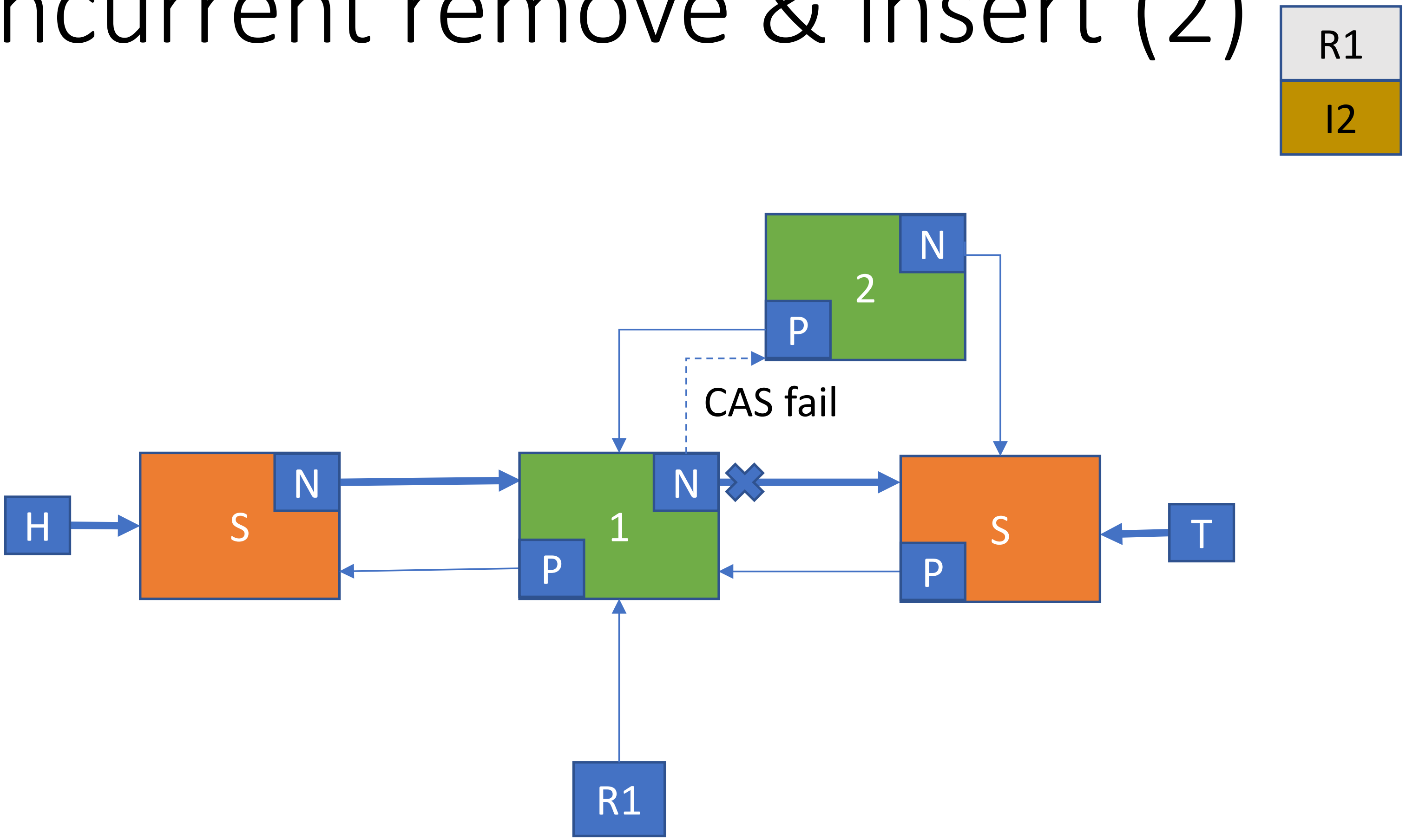


Concurrent remove & insert (1)

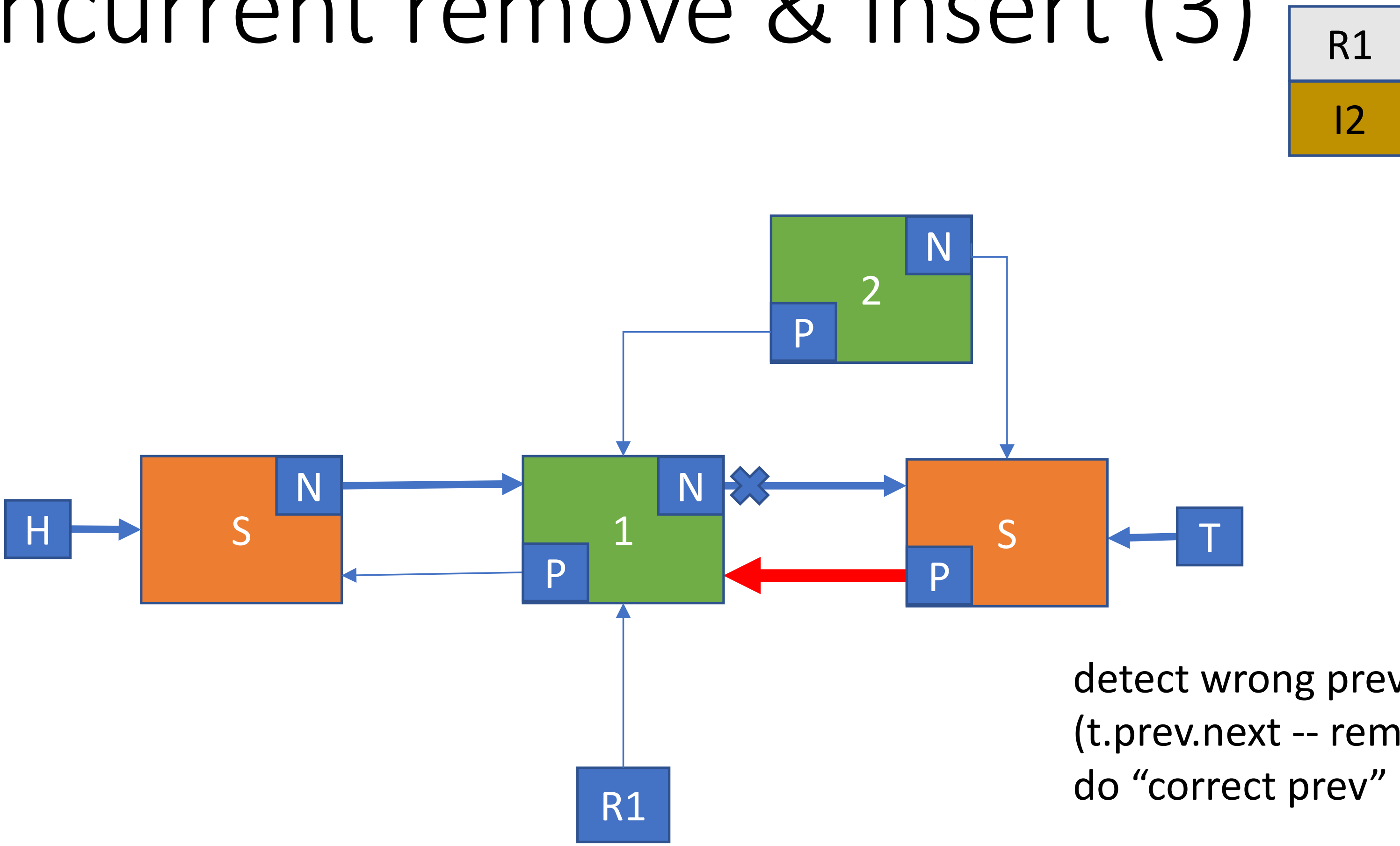
R1
I2



Concurrent remove & insert (2)

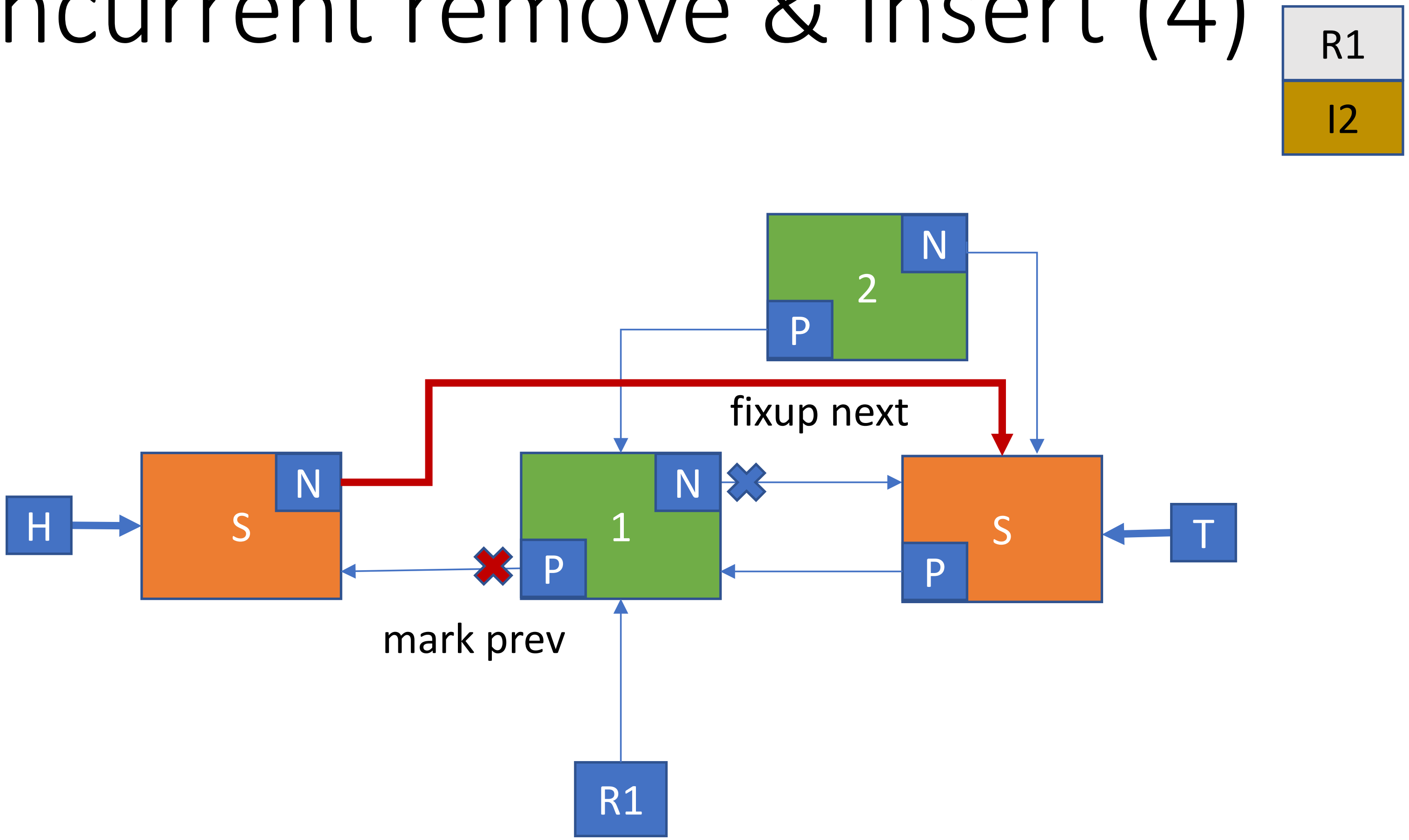


Concurrent remove & insert (3)

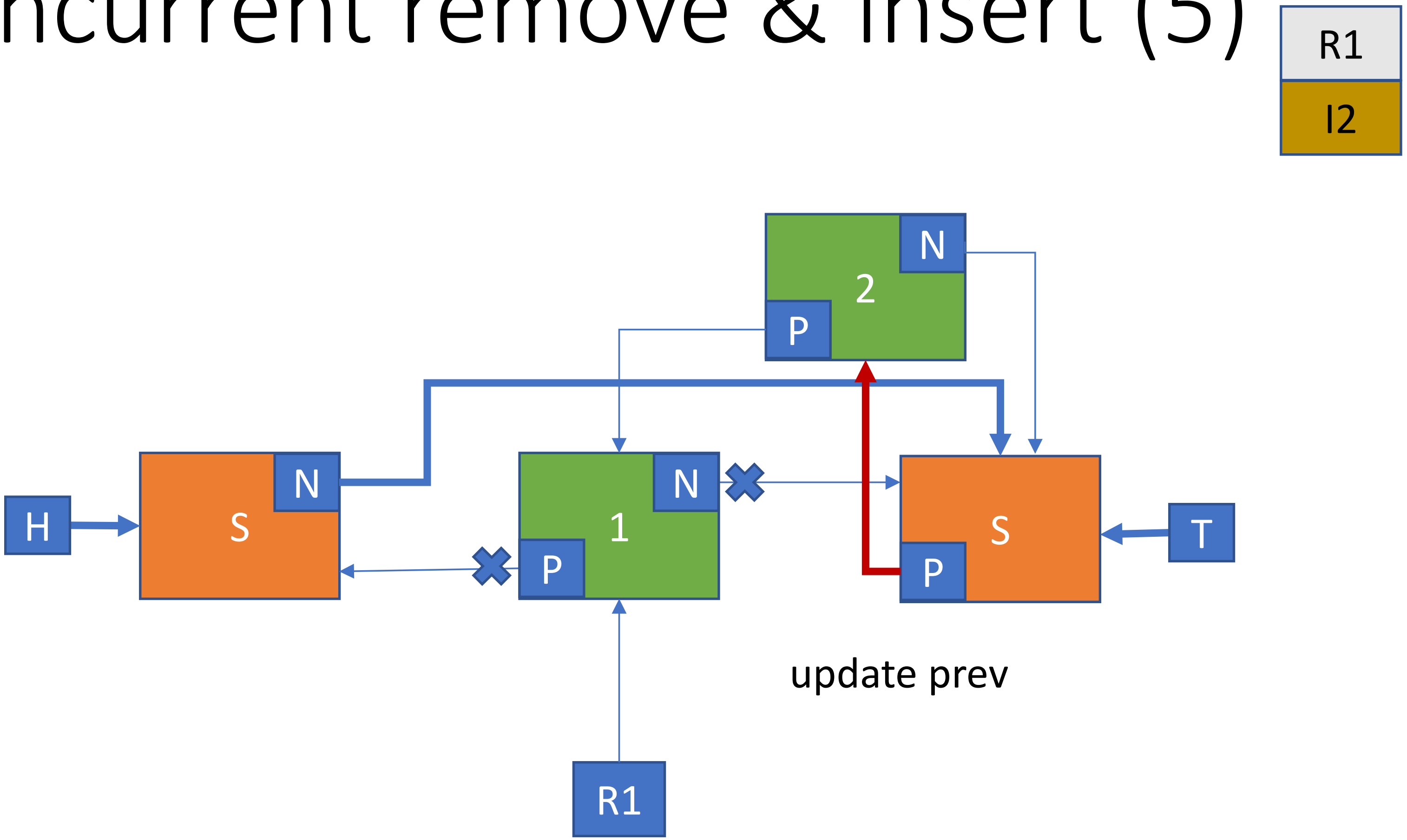


detect wrong prev
(t.prev.next -- removed)
do “correct prev”

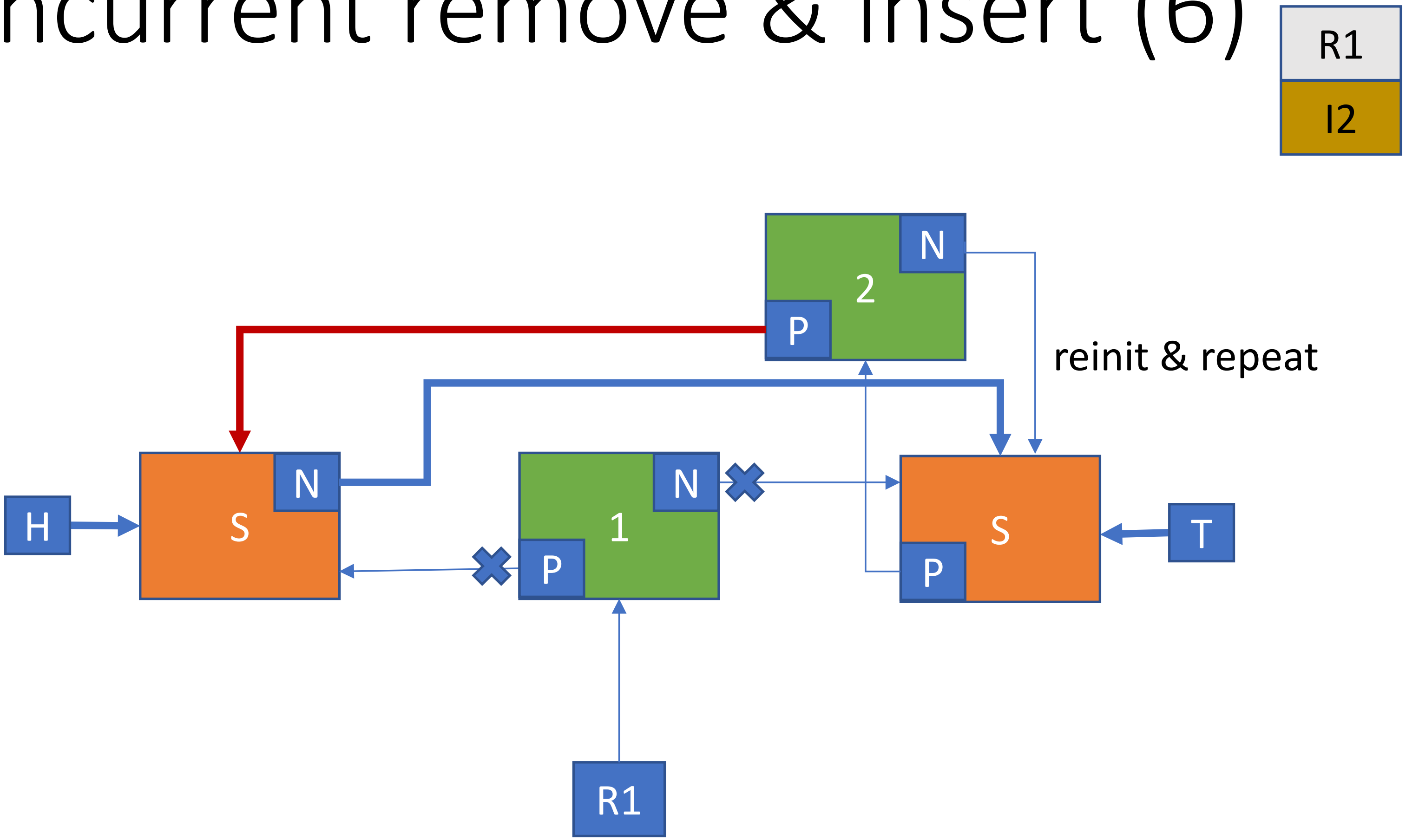
Concurrent remove & insert (4)



Concurrent remove & insert (5)



Concurrent remove & insert (6)

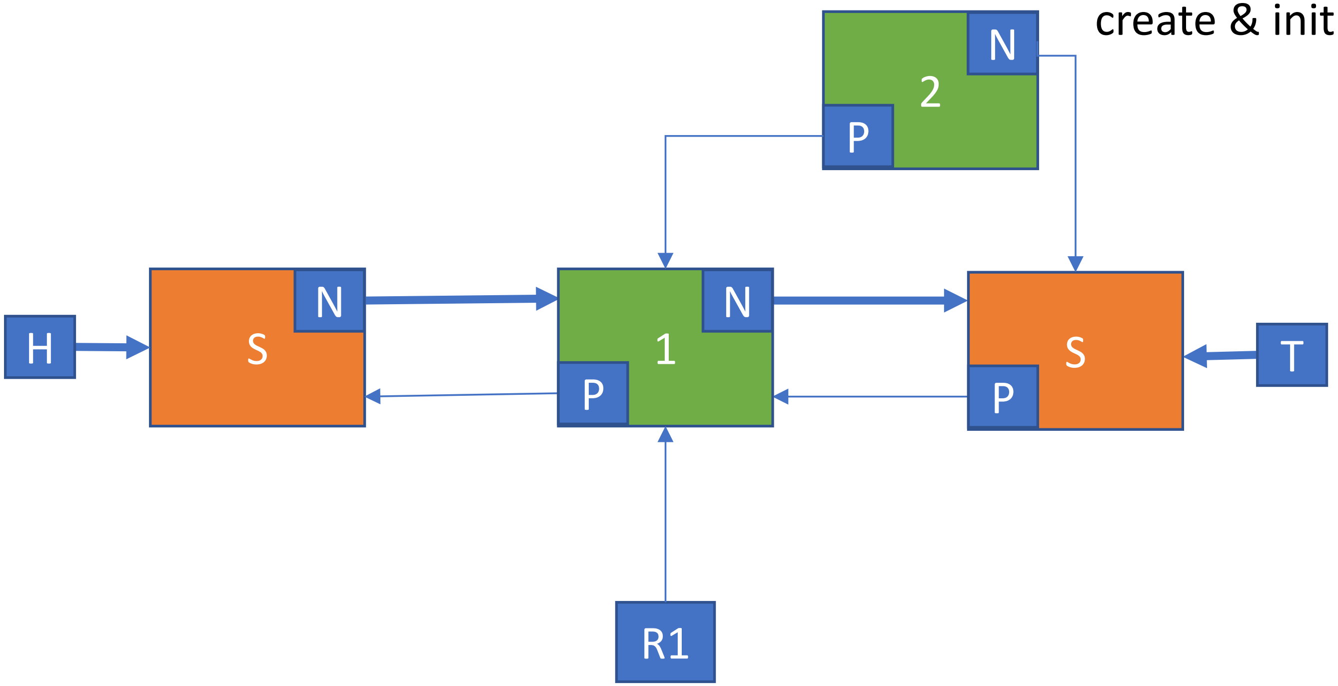


Concurrent remove & insert

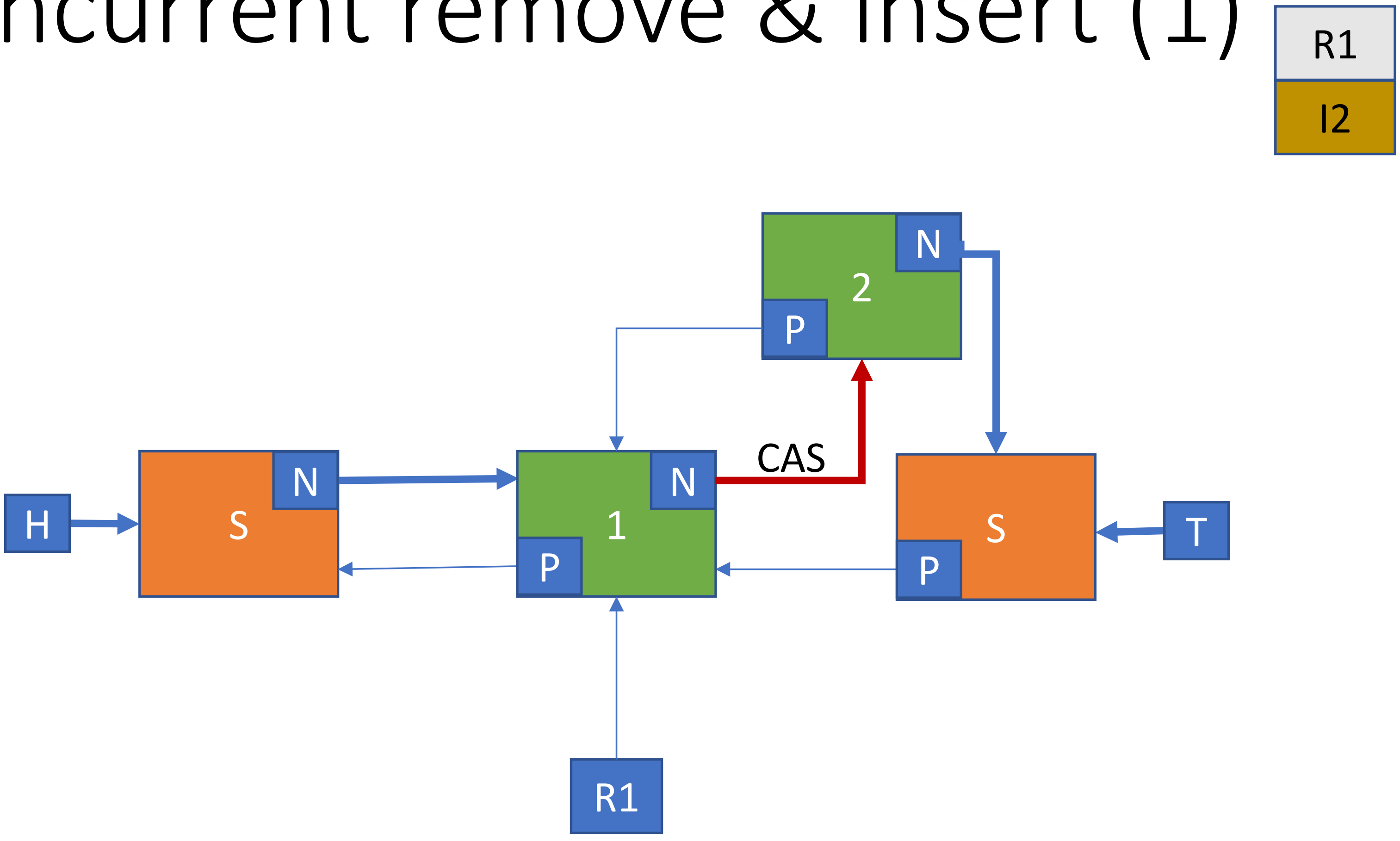
When insert wins

Concurrent remove & insert (0)

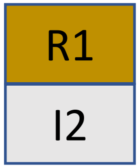
R1
I2

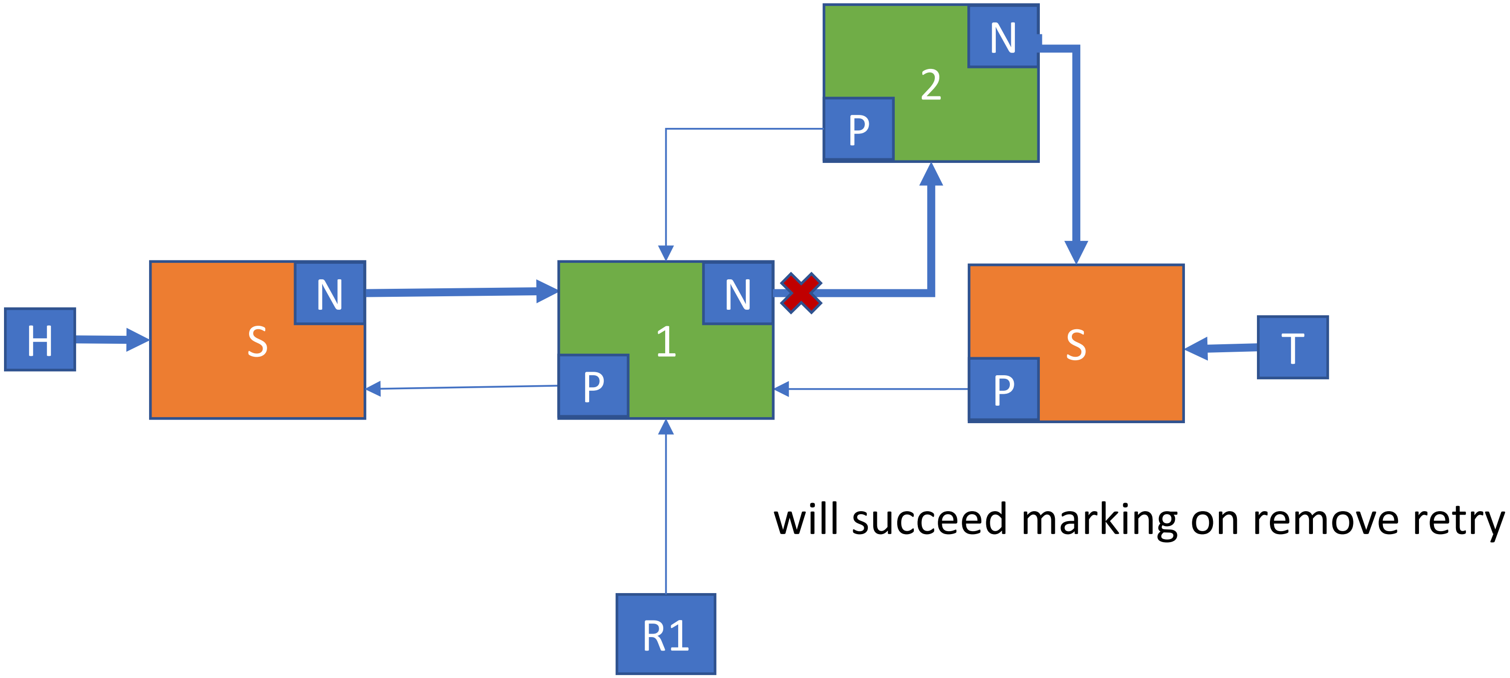


Concurrent remove & insert (1)

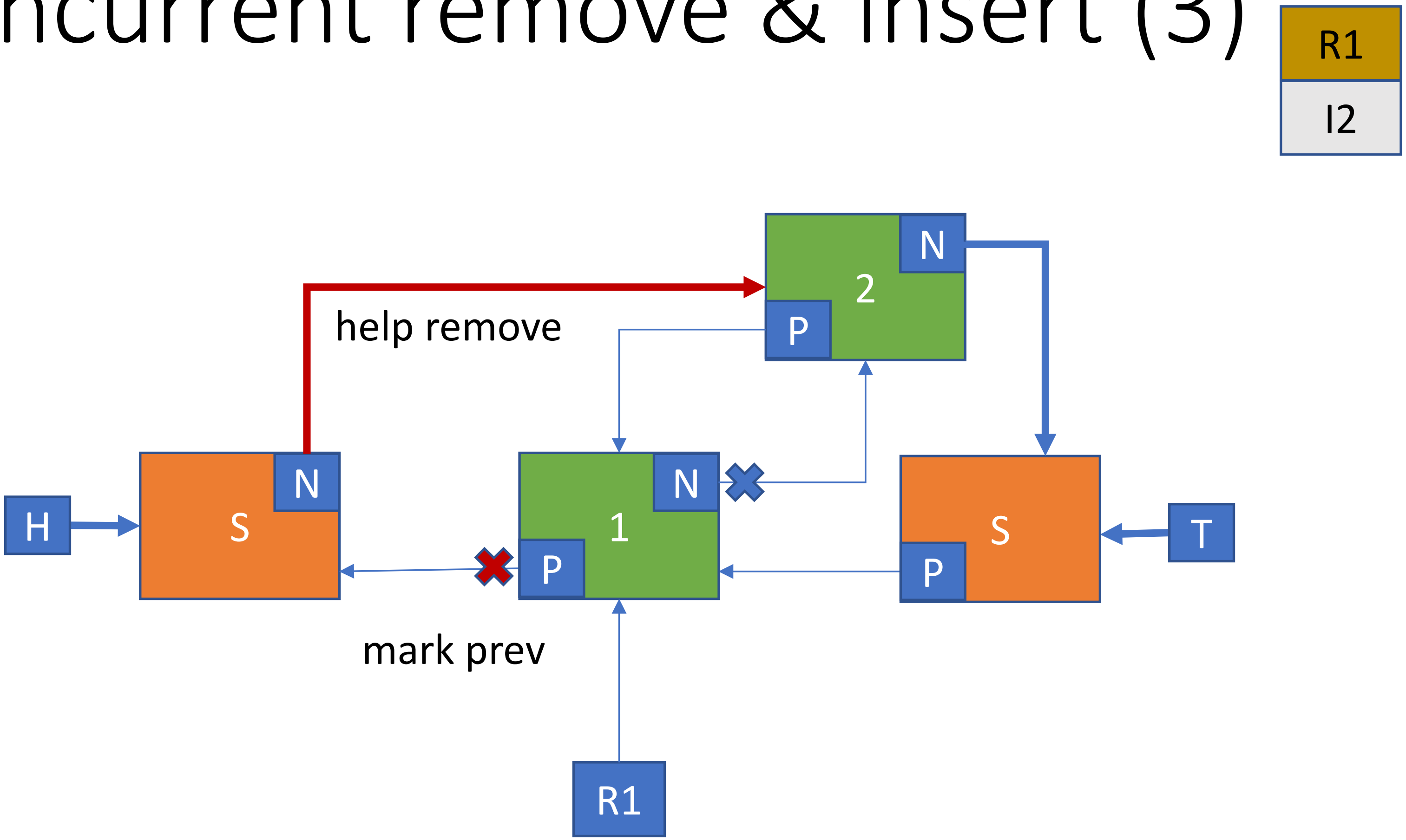


Concurrent remove & insert (2)

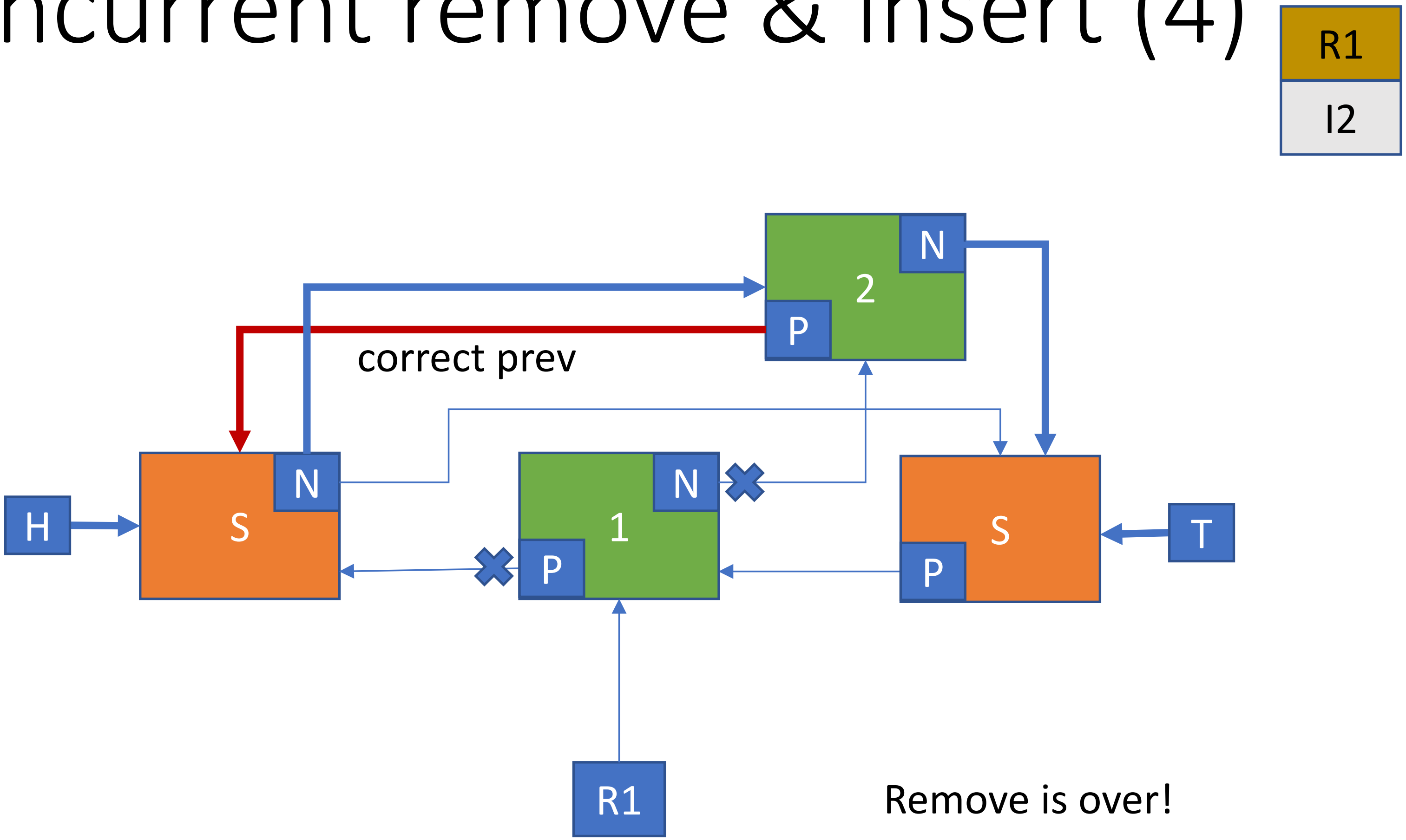




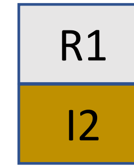
Concurrent remove & insert (3)

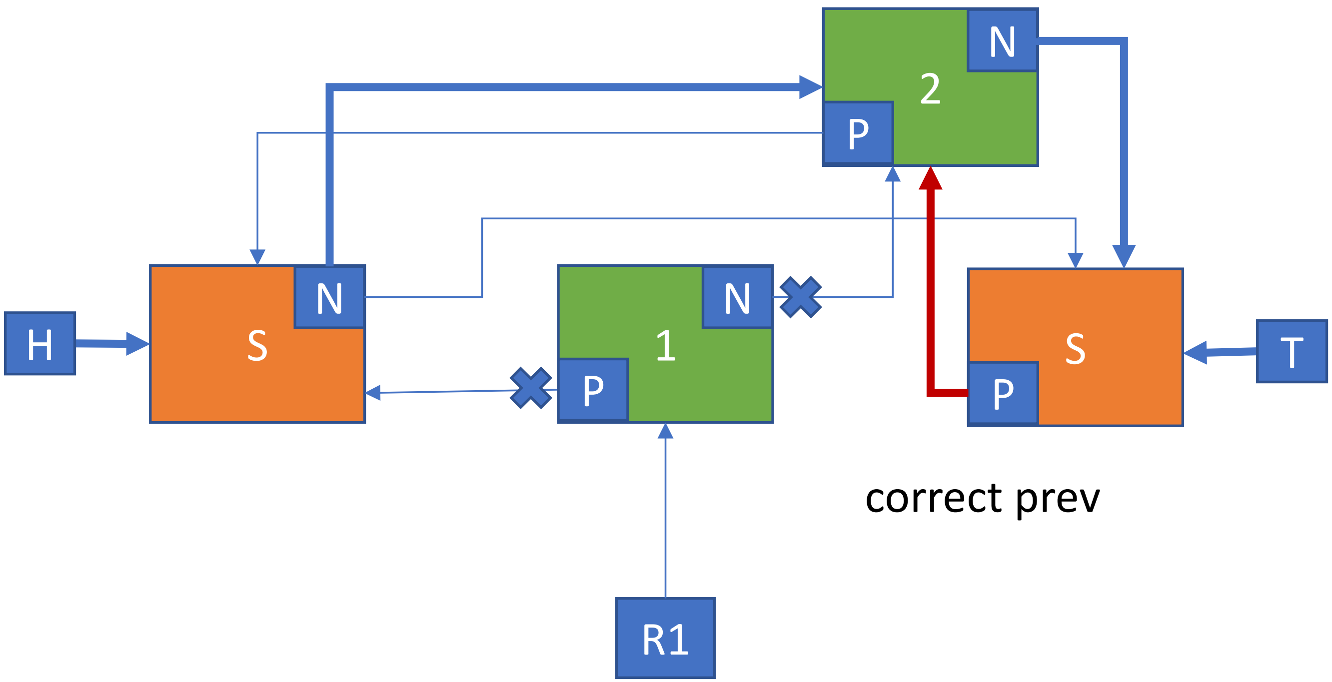


Concurrent remove & insert (4)



Concurrent remove & insert (5)





Takeaways

- A kind of algo you need a paper for
- Hard to improve w/o writing another paper
- **Good news:** stress tests uncover most impl bugs
- **Bad news:** when stress test fails, you up to long hours
- **More bad news:** hard to find bugs that violate lock-freedomness of algorithm

Summary: what we can do

- Insert items (at the end of the queue)
- Remove items (at the front of the queue)
- Traverse the list
- Remove items at arbitrary locations
  - In O(1)

Linearizability

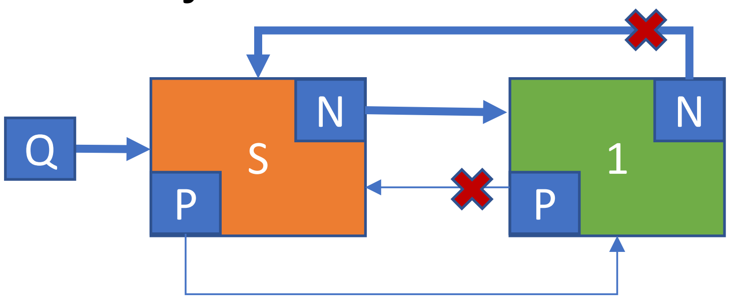
- Insert last
  - Linearizes at CAS of **next**
- Remove first / arbitrary
  - Success – at CAS of **next**
  - Fail – at read of **head.next**

More about algorithm

- Sundell & Tsigas algo supports *deque* operations
  - Can PushLeft & PopRight
- PopLeft is simple – read **head.next** & remove
- But cannot linearize them all at cas points
  - PushLeft, PushRight, PopRight - Ok
  - PopLeft linearizes at **head.next** read (!!!)

Summary of impl notes

- Use GC (drop all memory management details)
- Merge head & tail into a single sentinel node
  - Empty list is just one object (prev & next onto itself)
  - One item += one object



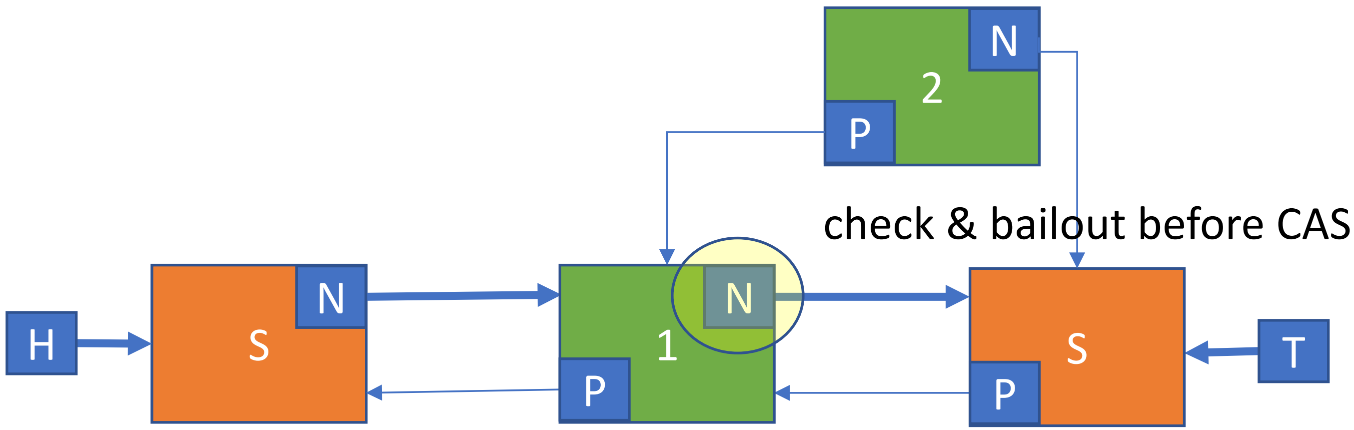
- Reuse “remove mark” objects
  - One-element lists reuse of ptrs to sentinel all the time
- Encapsulate!

Mods

More complex *atomic* operations

Basic mods (1)

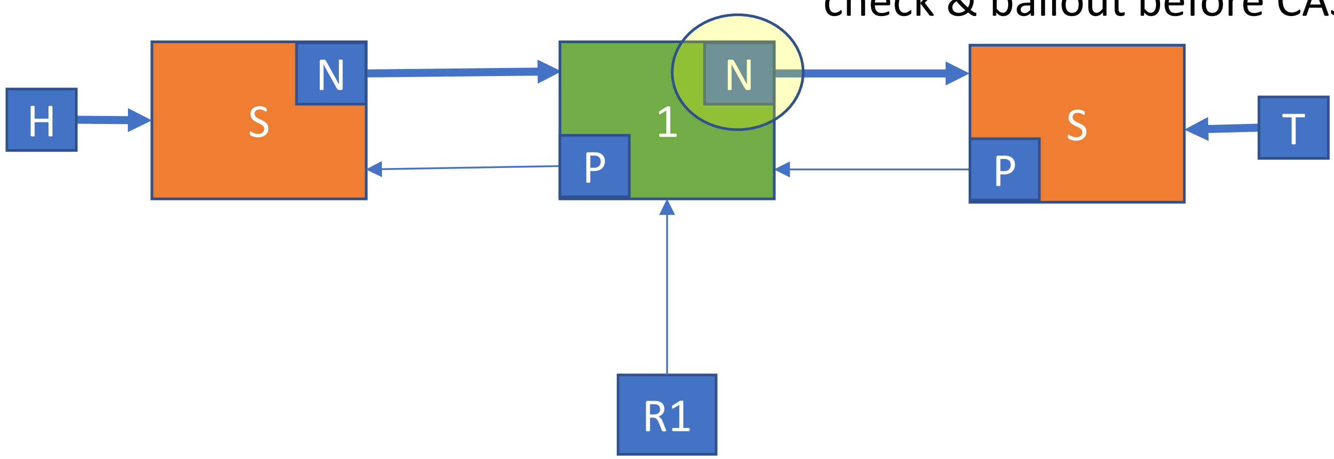
- Insert item conditionally on prev tail value



Basic mods (2)

- Remove head conditionally on prev head value

check & bailout before CAS



Practical use-case: synchronous channels

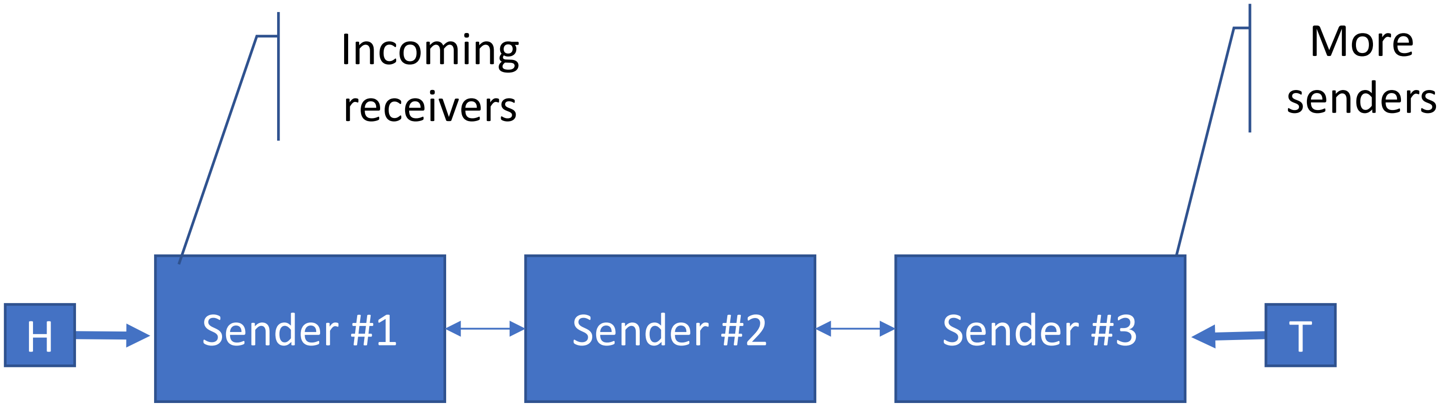
```
1   val channel = Channel<Int>()

    // coroutine #1
    for (x in 1..5) {
2       channel.send(x * x)
    }



    // coroutine #2
    repeat(5) {
3       println(channel.receive())
    }
```

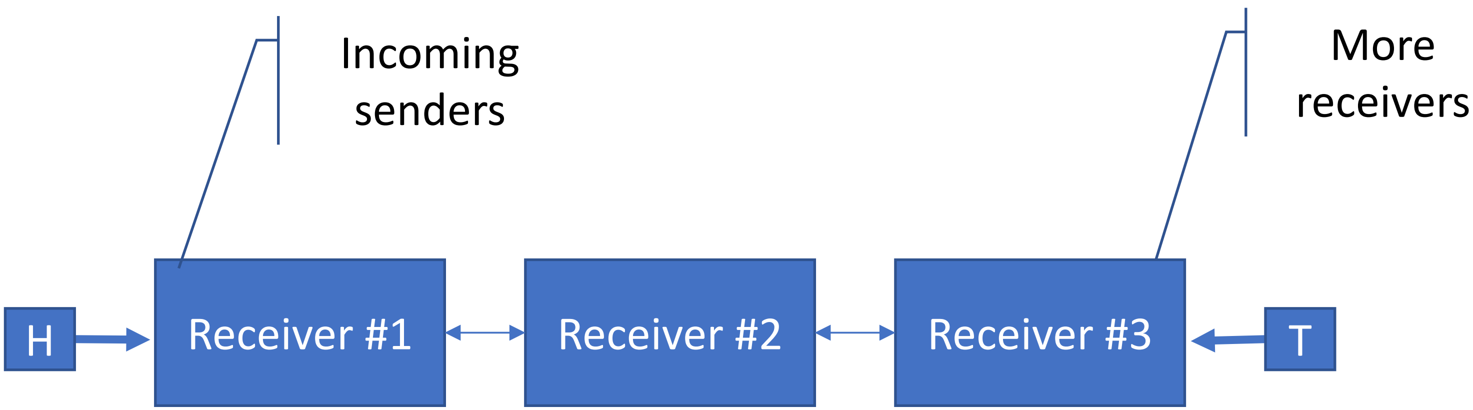
Senders wait



Receiver removes first if it is a sender node

Sender inserts last if it is not a receiver node

Receivers wait



Sender removes first if it is a receiver node

Receiver inserts last if it is not a sender node

Send function sketch

```
fun send(element: T) {
①   while (true) {
        // try to add sender, unless prev is receiver
②       if (enqueueSend(element)) break
        // try to remove first receiver
③       val receiver = removeFirstReceiver()
        if (receiver != null) {
④           receiver.resume(element) // resume receiver
            break
        }
    }
}
```

Channel use-case recap

- Uses insert/remove ops conditional on tail/head node
- Can abort (cancel) wait to receive/send at any time by using remove
  - Full removal -- no garbage is left
- Pretty efficient in practice
  - One item lists – one “garbage” object

Multi-word compare and swap (CASN)

Build even bigger atomic operations

Use-case: select expression

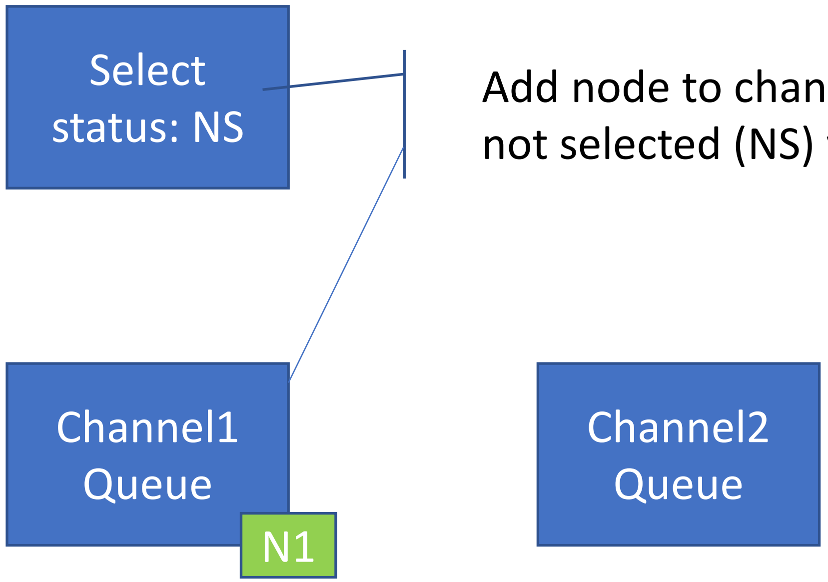
```kotlin
val channel1 = Channel<Int>()
val channel2 = Channel<Int>()
```

```kotlin
select {
    channel1.onReceive { e -> ... }
    channel2.onReceive { e -> ... }
}
```

Impl summary: register (1)

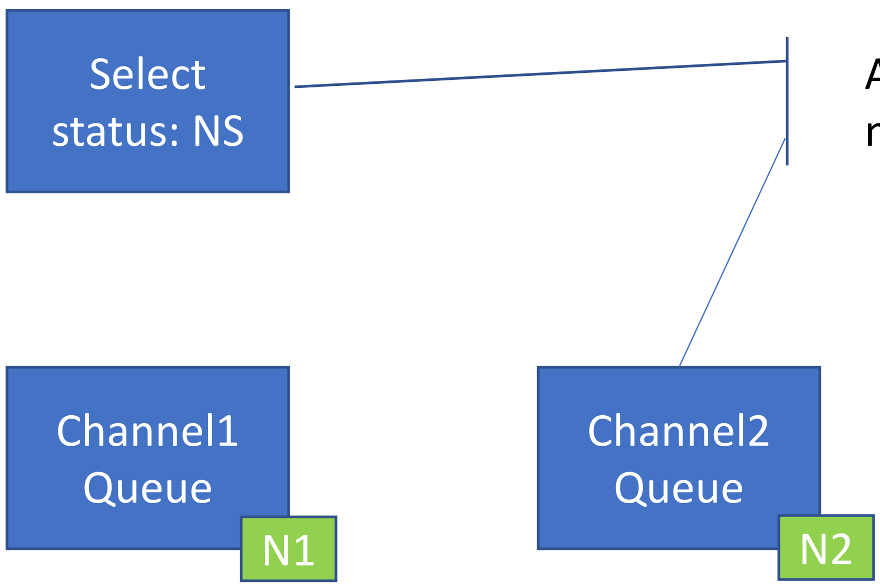
1. Not selected
2. Selected

Impl summary: register (2)



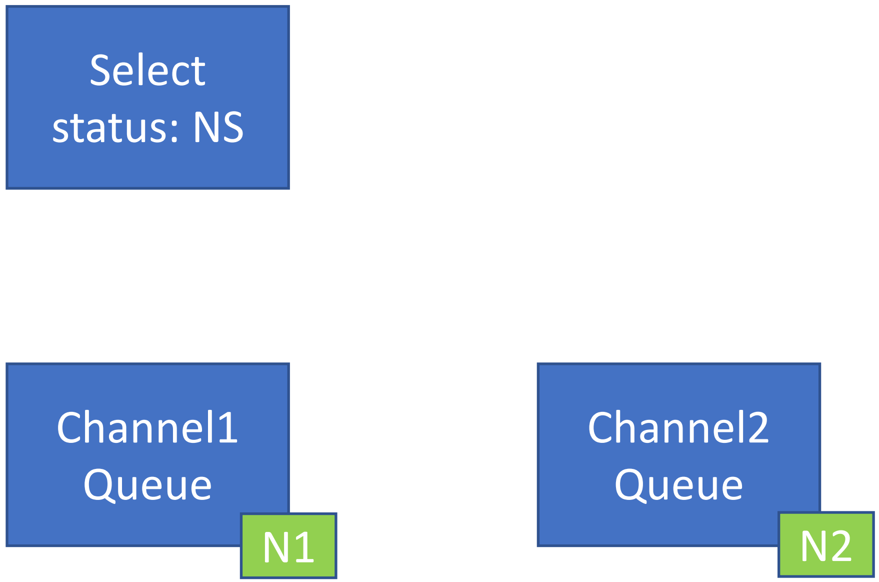
Add node to channel1 queue **if** not selected (NS) yet

Impl summary: register (3)

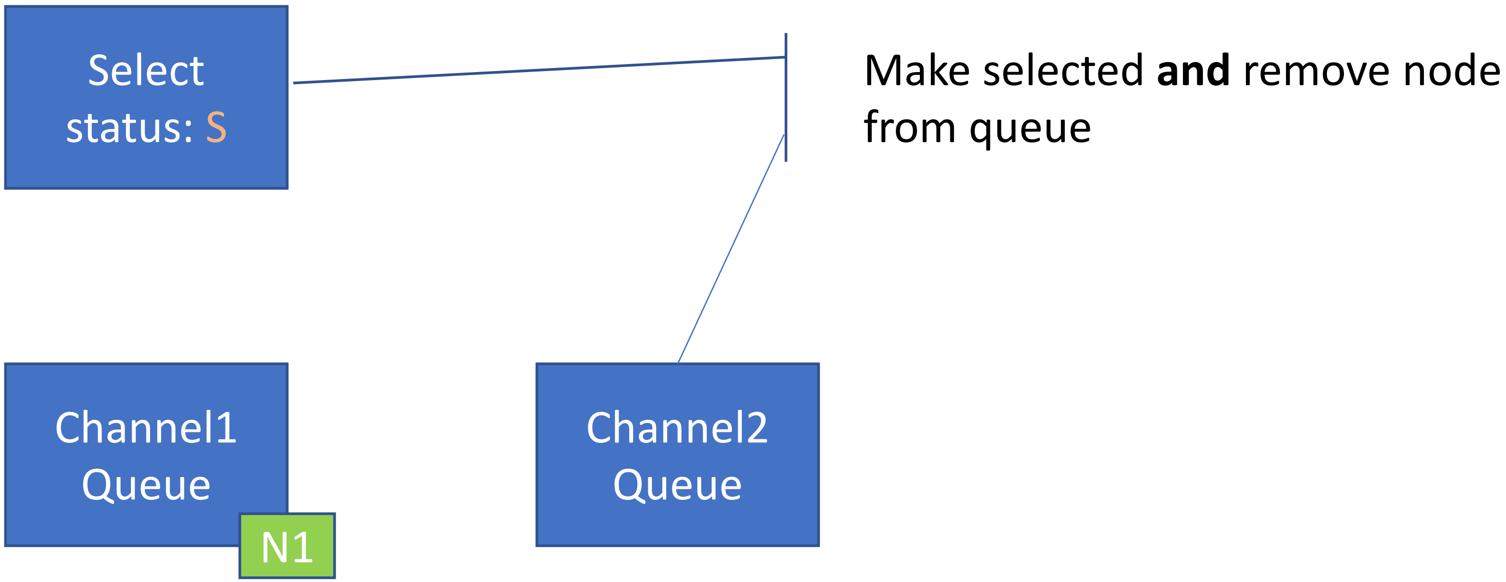


Add node to channel2 queue **if** not selected (NS) yet

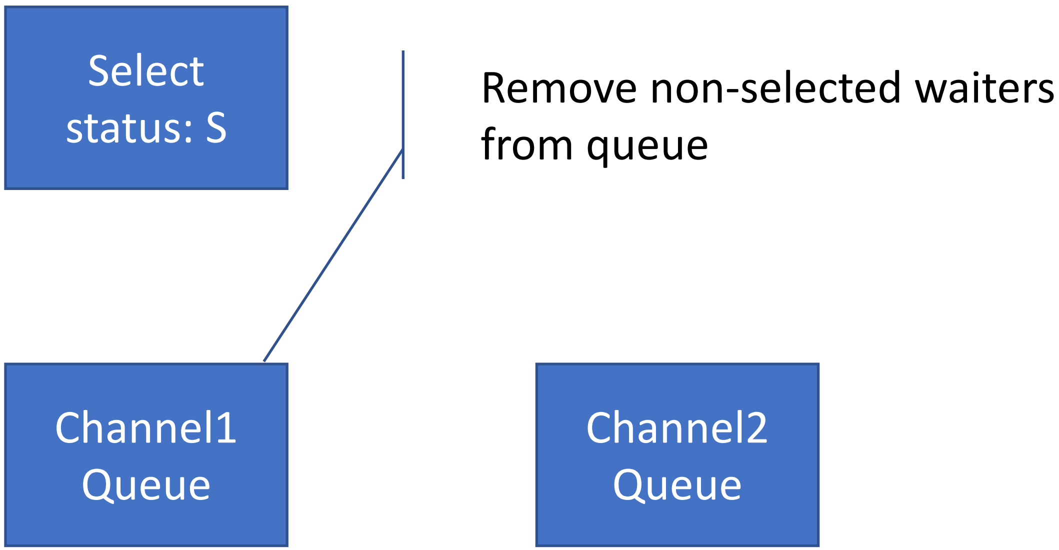
Impl summary: wait



Impl summary: select (resume)



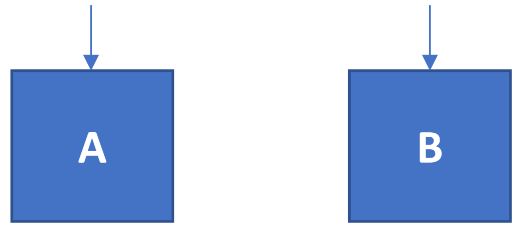
Impl summary: clean up rest



Double-Compare Single-Swap (DCSS)

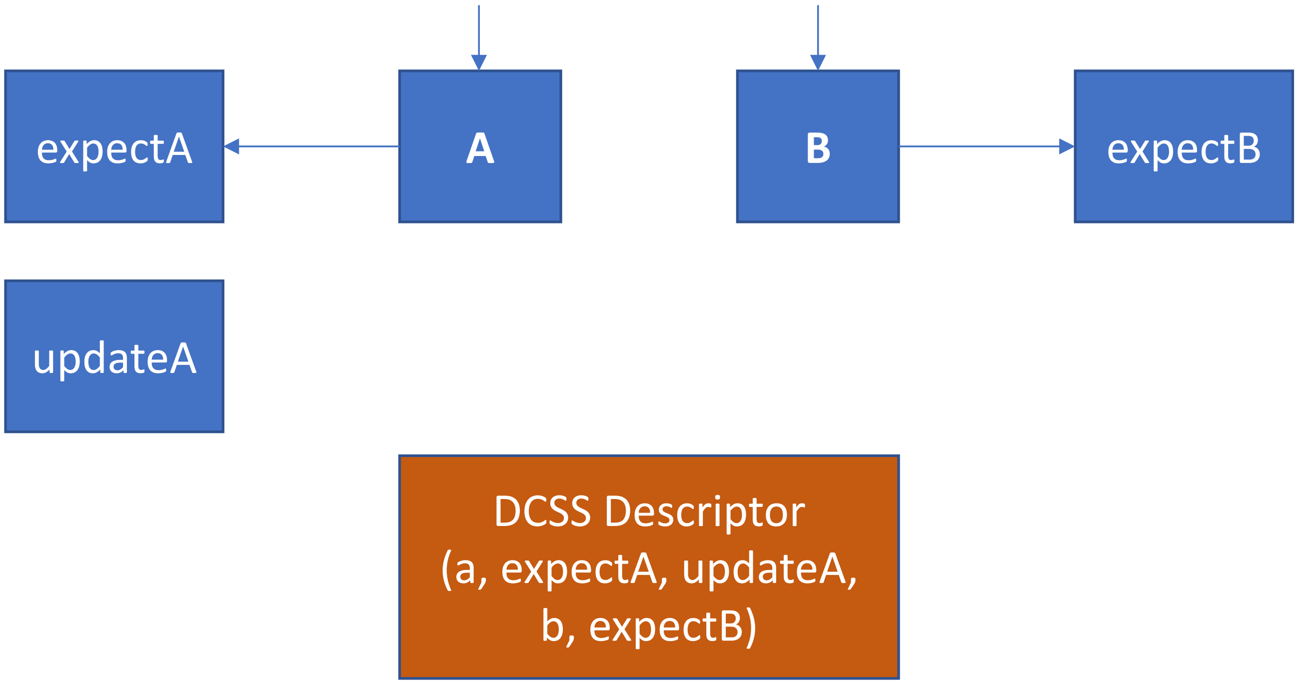
Building block for CASN

DCSS spec in pseudo-code

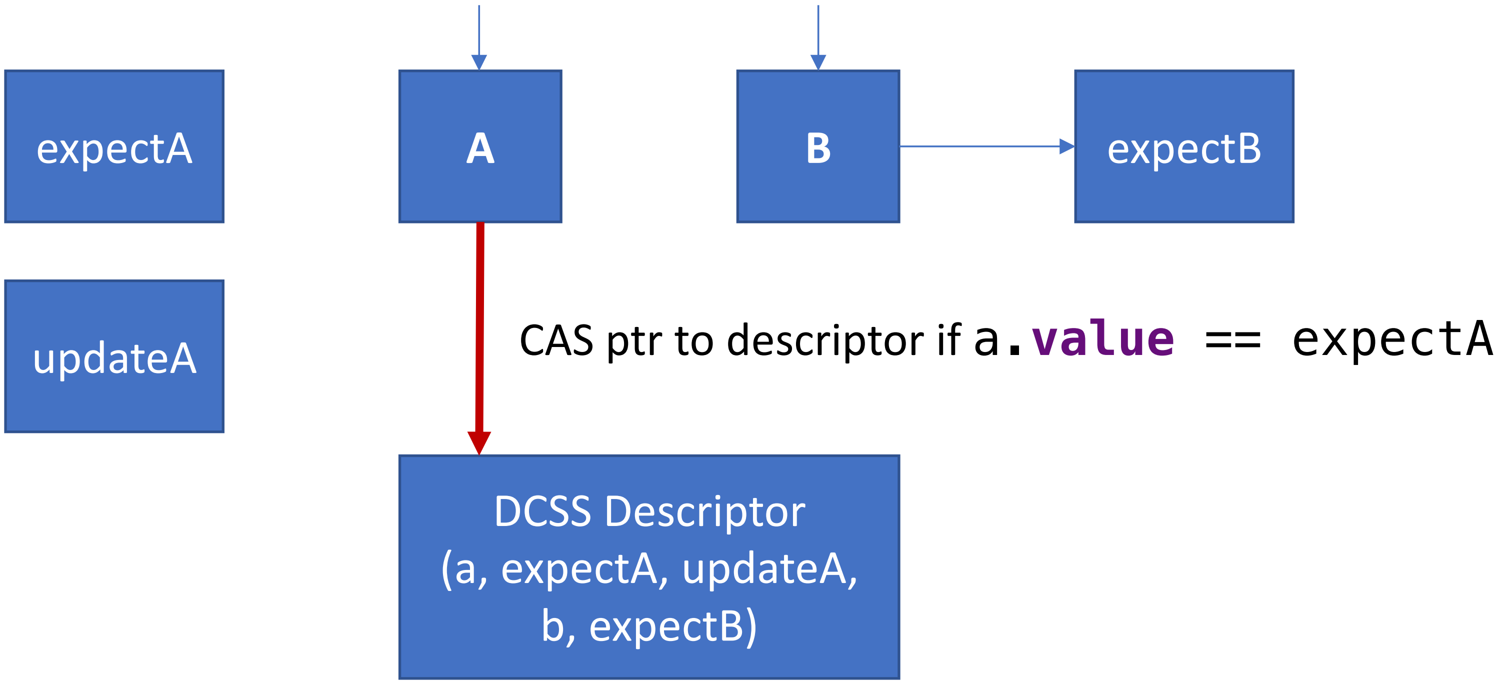



```
fun <A,B> dcss(
1       a: Ref<A>, expectA: A, updateA: A,
2       b: Ref<B>, expectB: B) =
            atomic {
3               if (a.value == expectA && b.value == expectB) {
4                   a.value = updateA
                }
            }
```

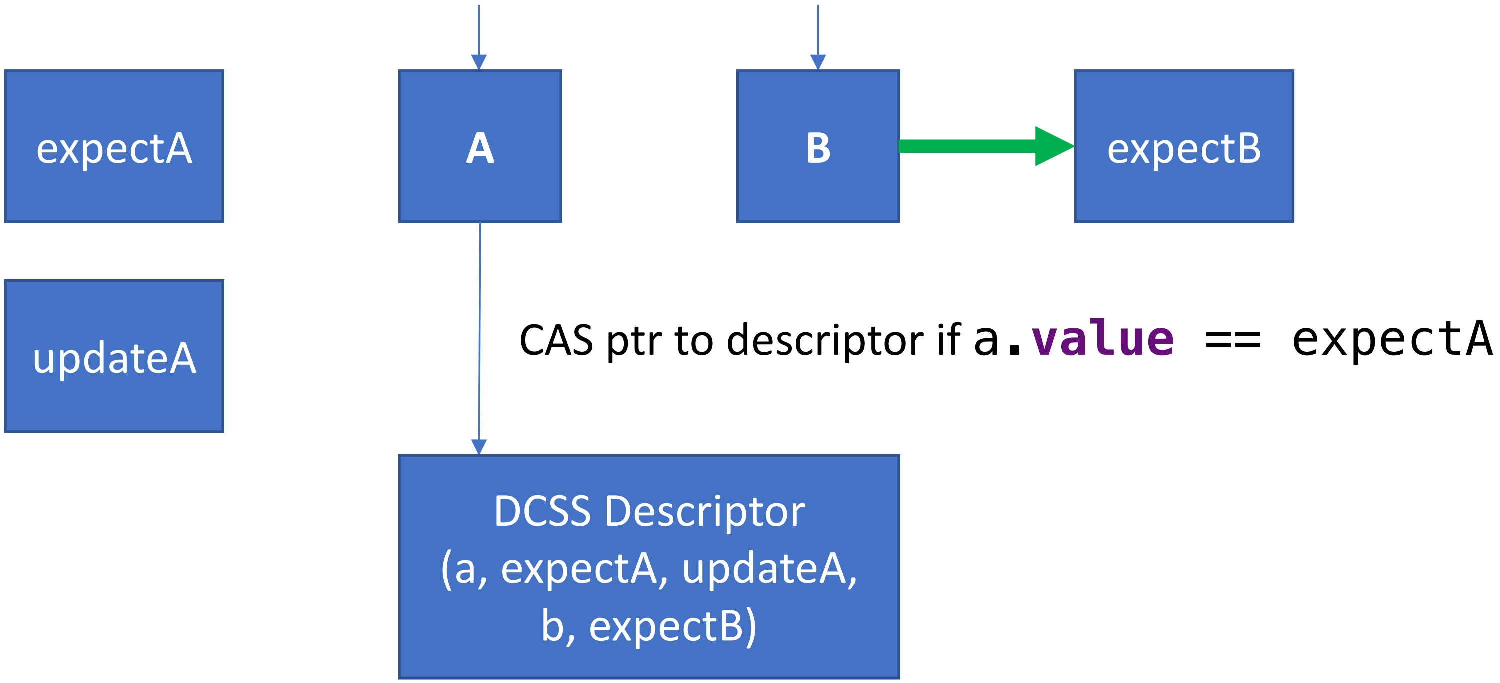
DCSS: init descriptor



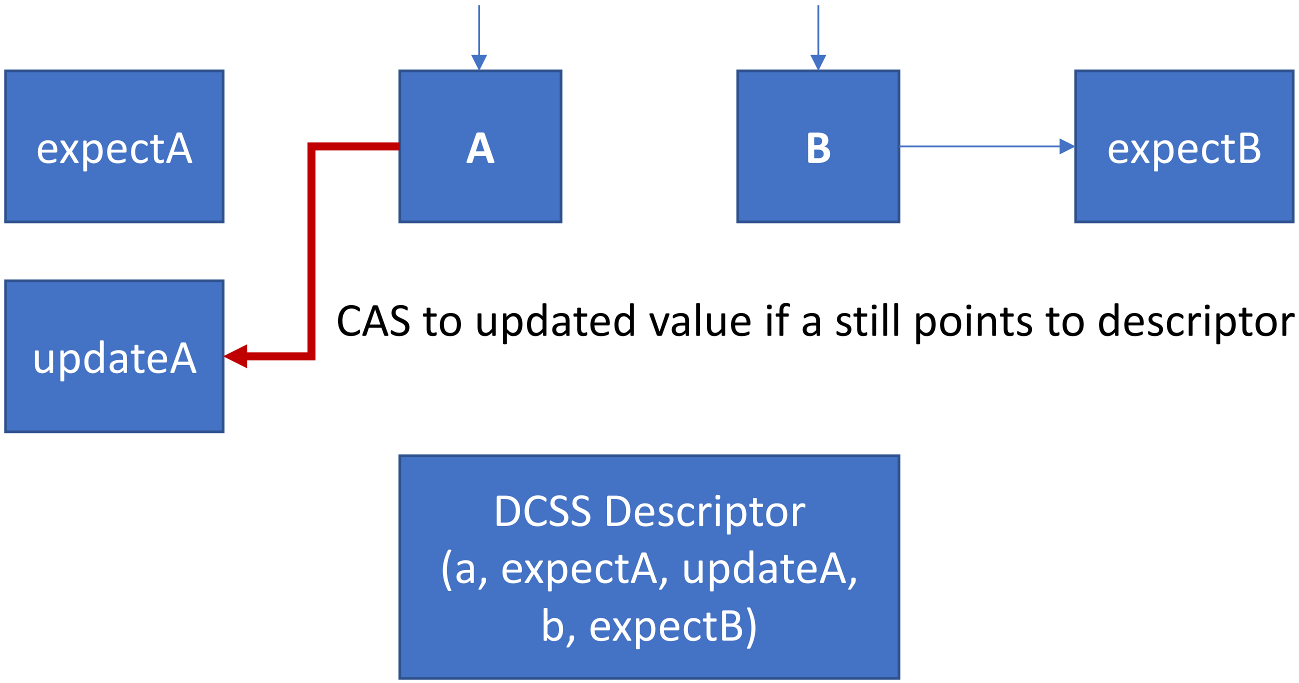
DCSS: prepare



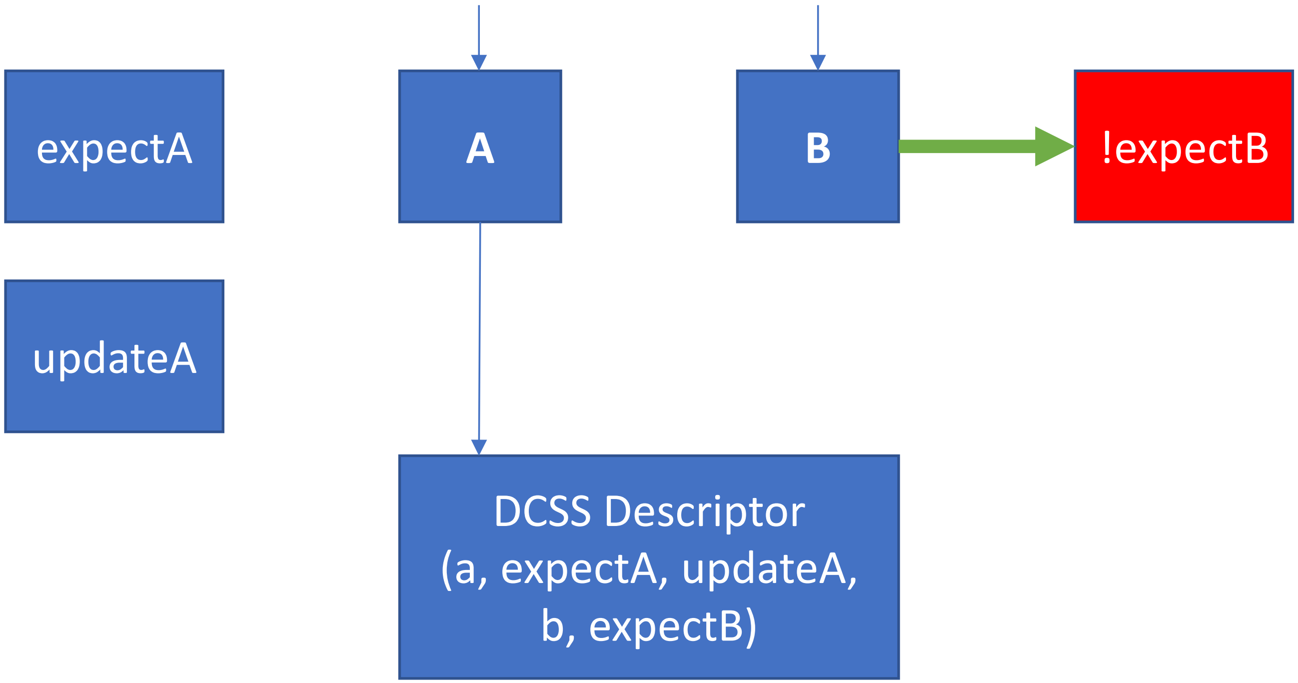
DCSS: read b.value



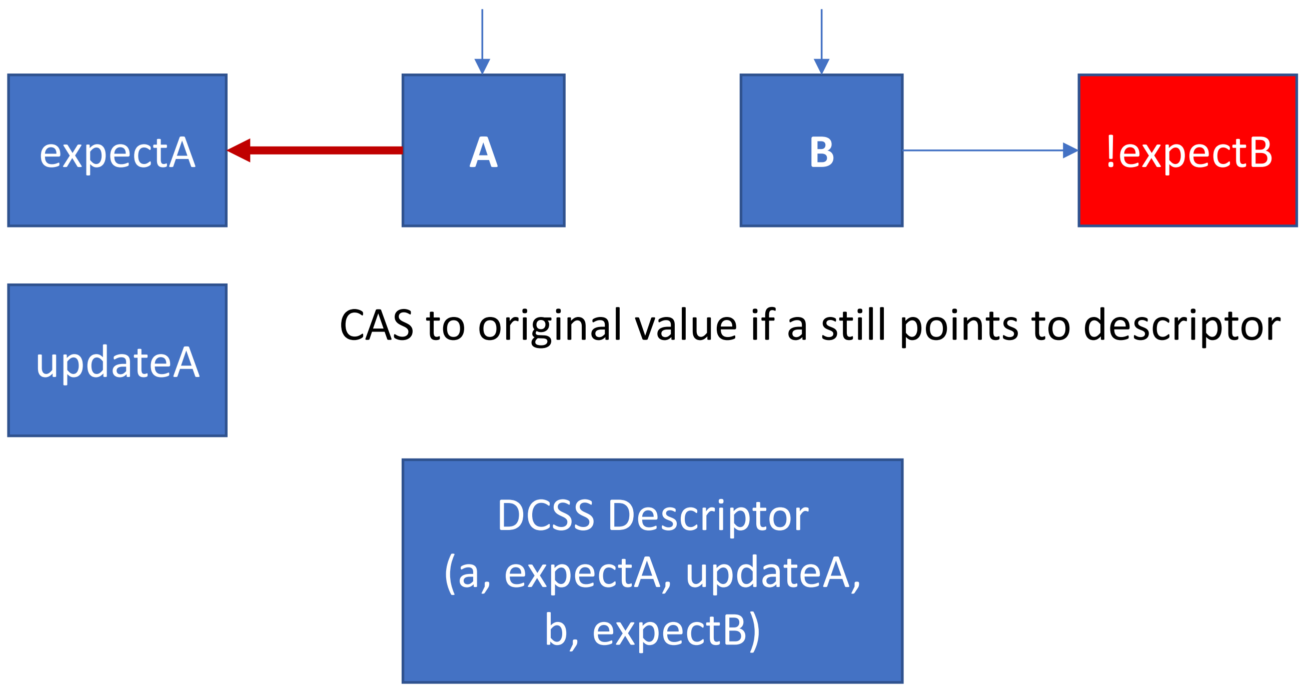
DCSS: complete (when success)



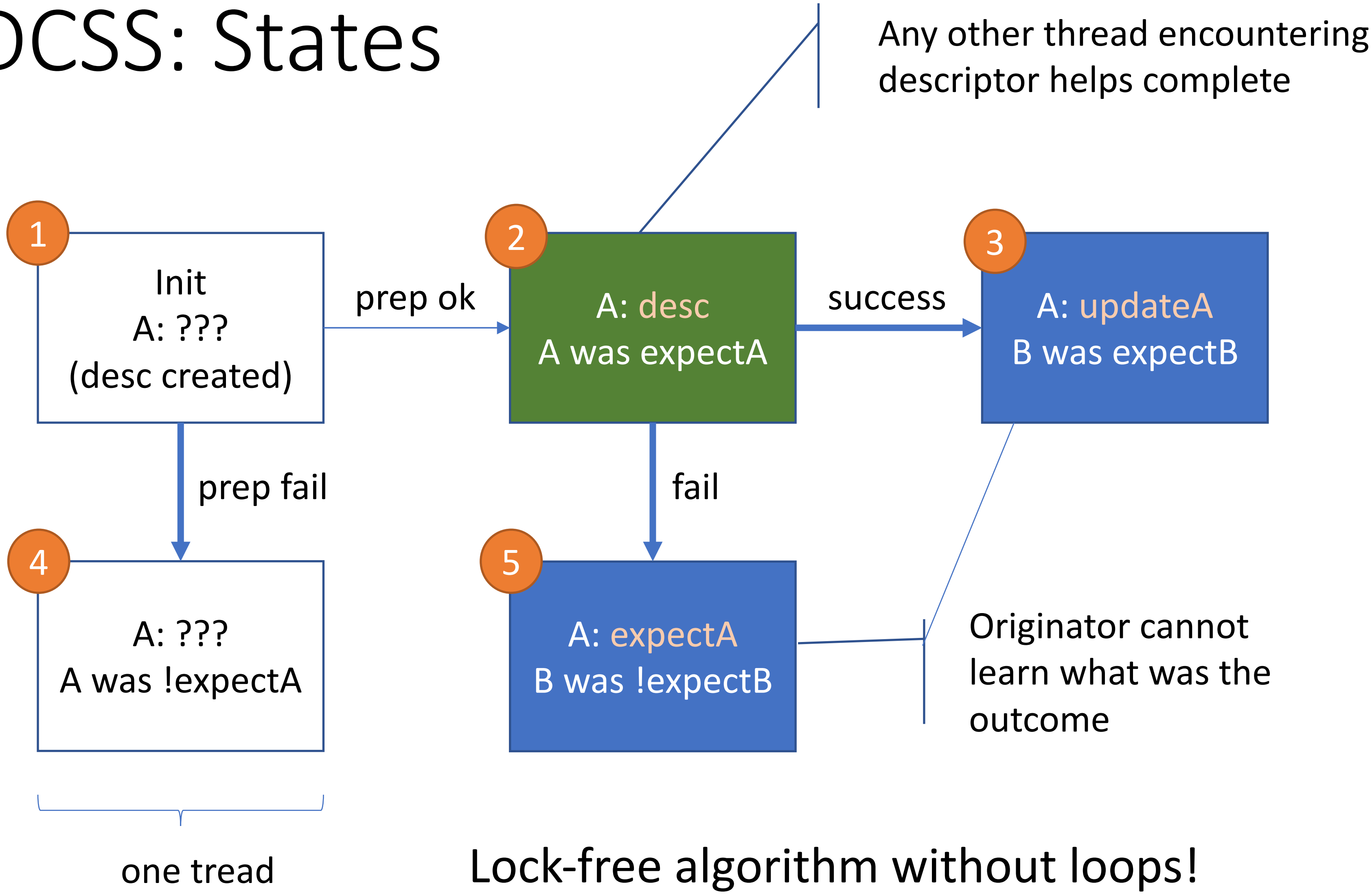
DCSS: complete (alternative)



DCSS: complete (when fail)



DCSS: States

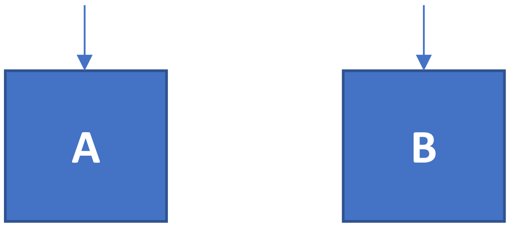


Lock-free algorithm without loops!

Caveats

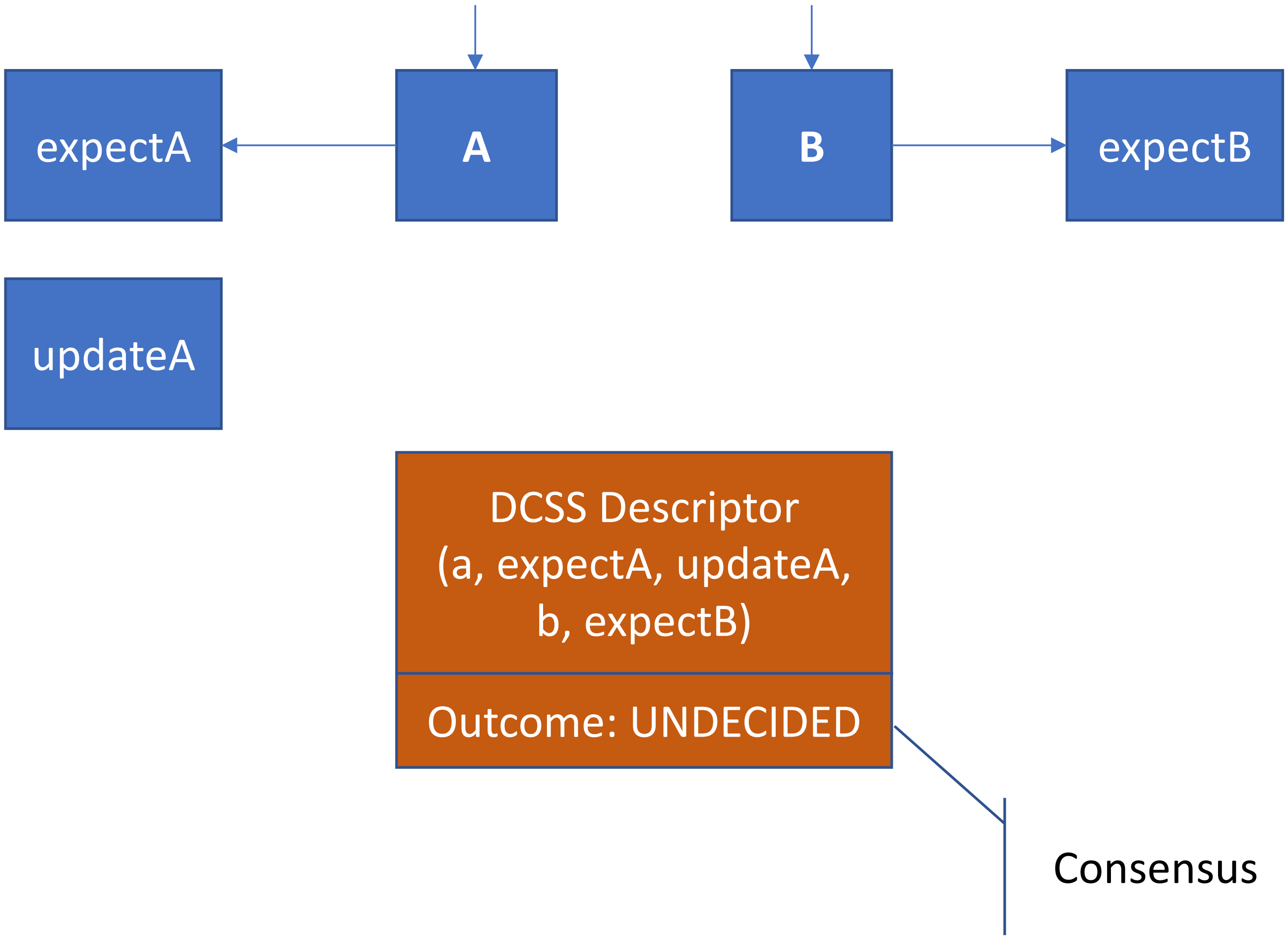
- A & B locations must be totally ordered
  - or risk stack-overflow while helping
- One way to look at it: Restricted DCSS (RDCSS)

DCSS Mod: learn outcome

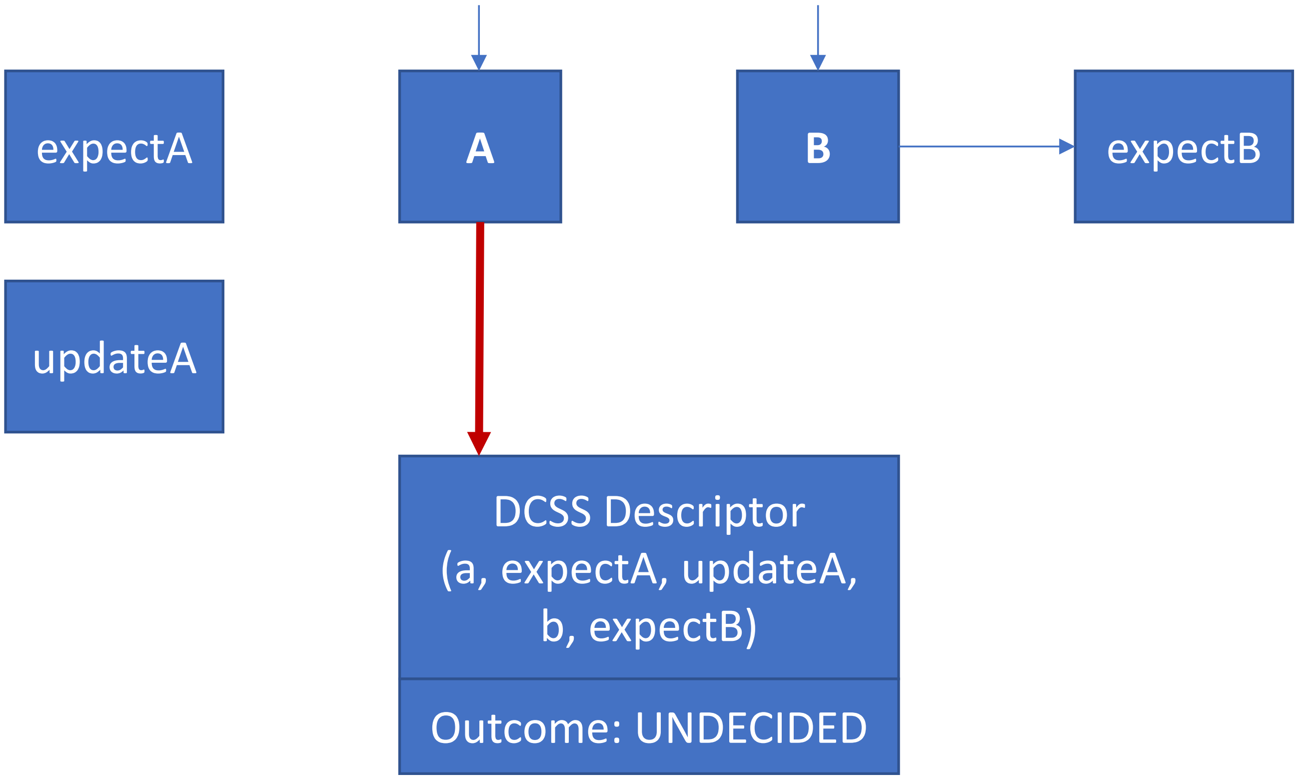



```
fun <A,B> dcssMod(
    a: Ref<A>, expectA: A, updateA: A,
    b: Ref<B>, expectB: B): Boolean =
        atomic {
            if (a.value == expectA && b.value == expectB) {
                a.value = updateA
                true
            } else
                false
        }
```

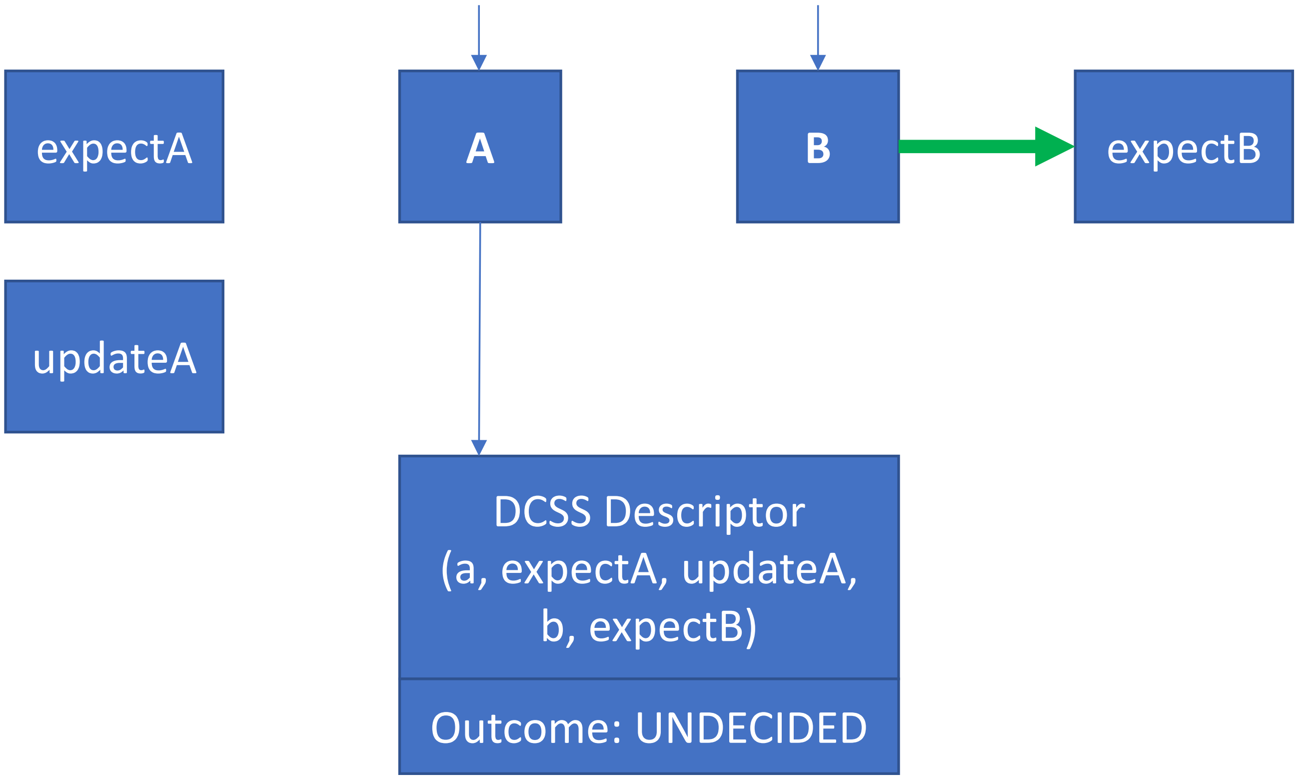
DCSS Mod: init descriptor



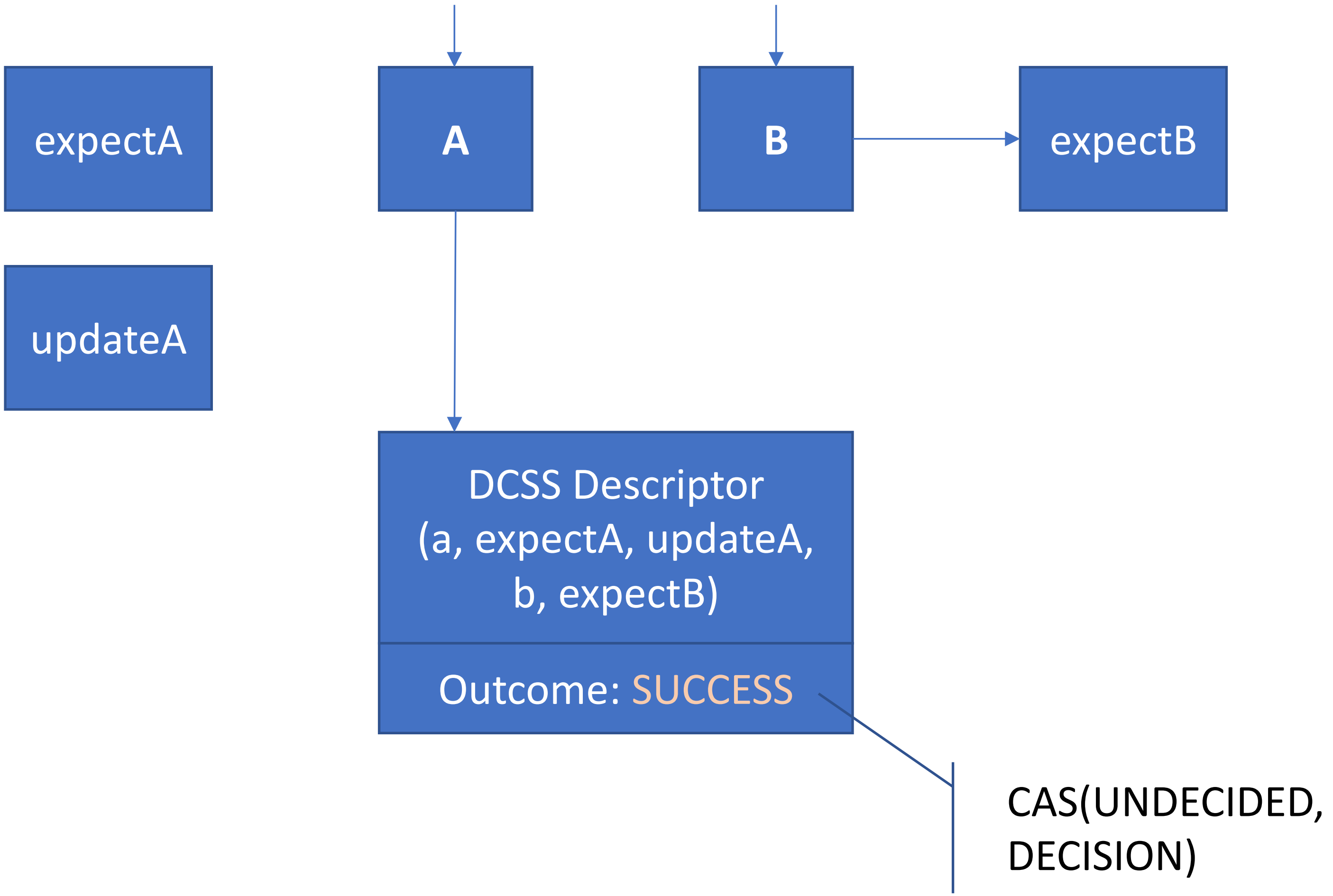
DCSS Mod: prepare



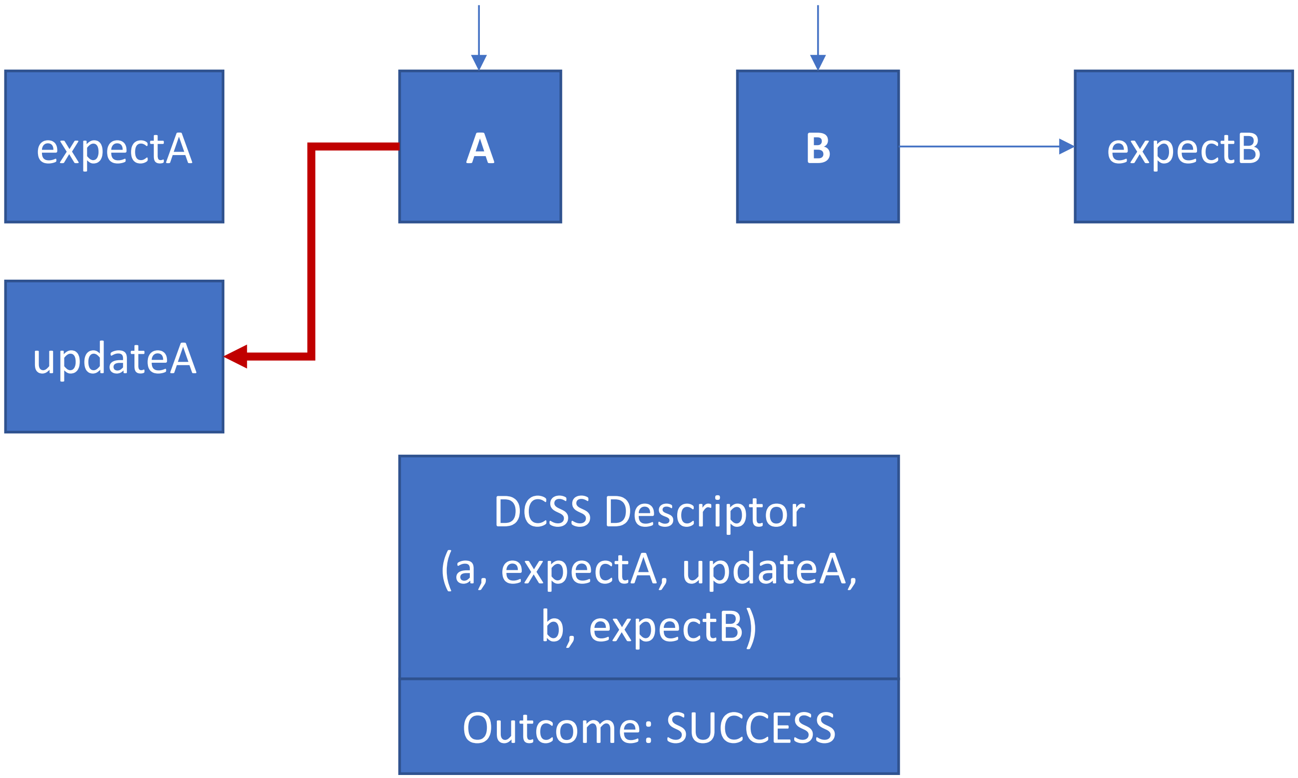
DCSS Mod: read b.value



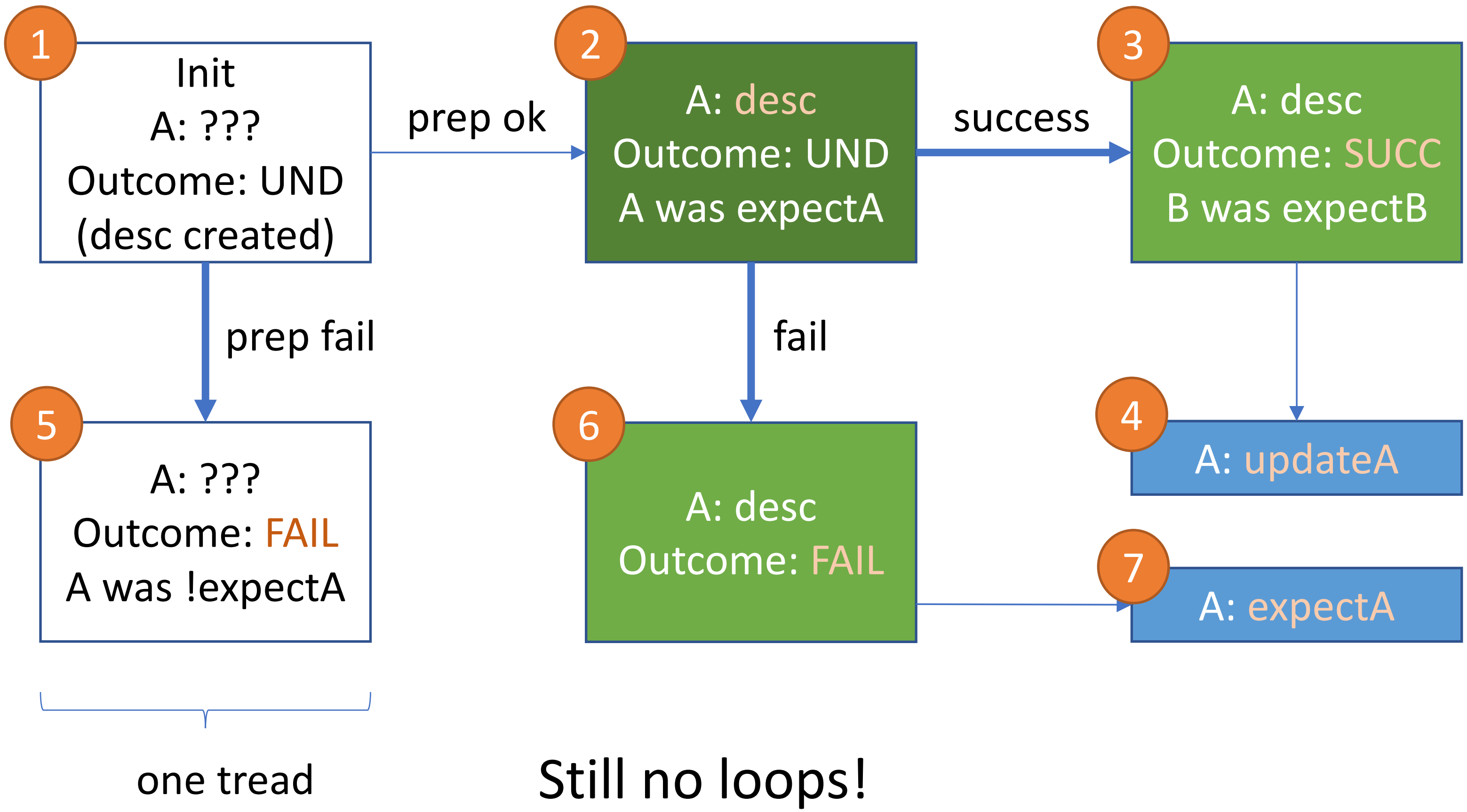
DCSS Mod: reach consensus



DCSS Mod: complete



DCSS Mod: States



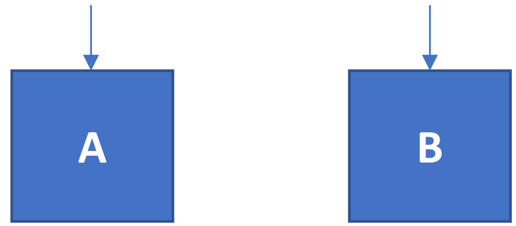
Still no loops!

Compare-And-Swap N-words (CASN)

The ultimate atomic update

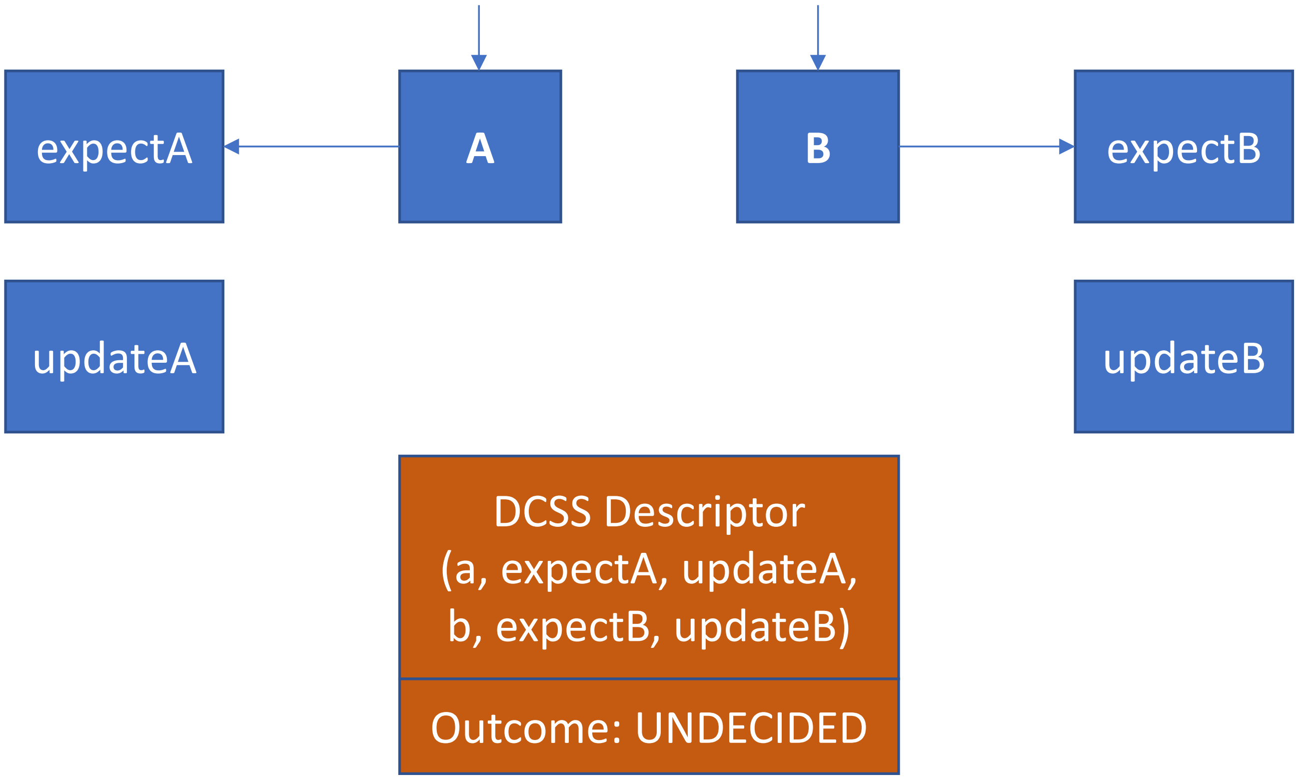
CASN spec in pseudo-code

For two words, for simplicity

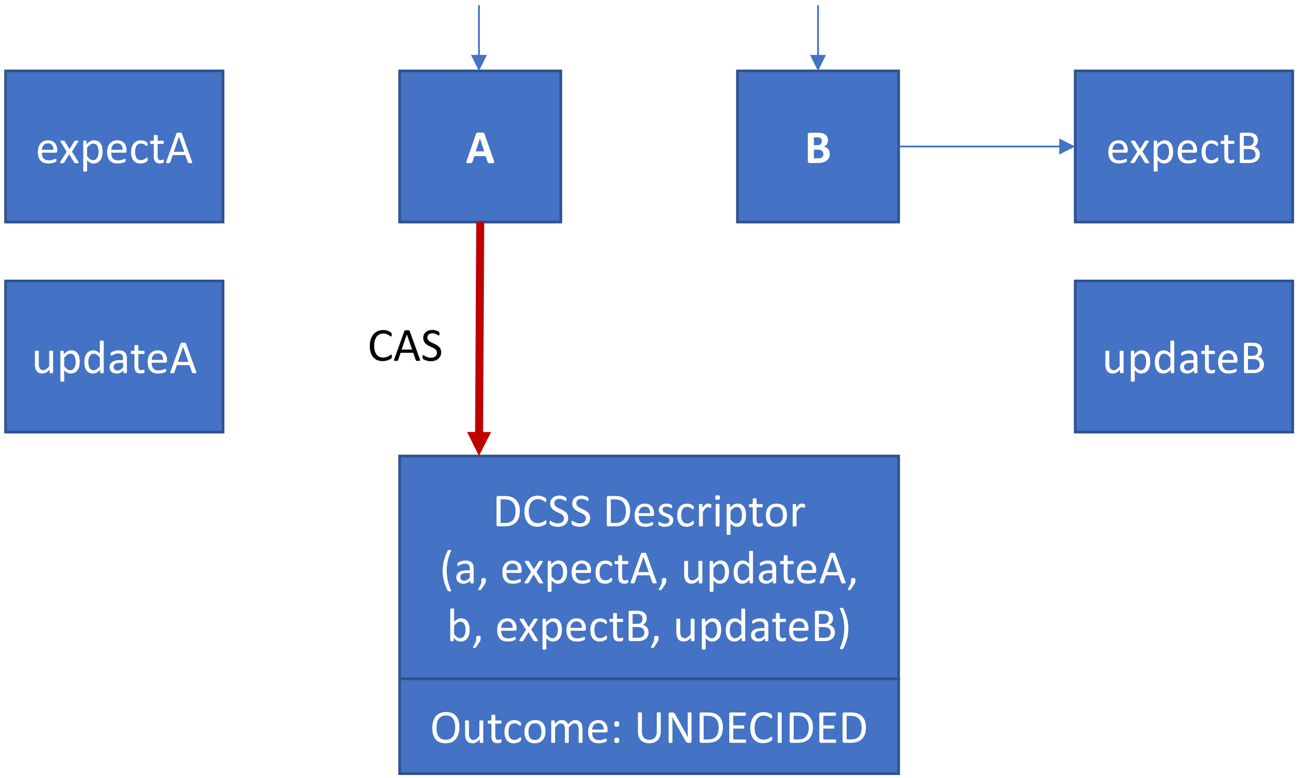



```
fun <A,B> cas2(
1       a: Ref<A>, expectA: A, updateA: A,
2       b: Ref<B>, expectB: B, updateB: B): Boolean =
            atomic {
3               if (a.value == expectA && b.value == expectB) {
                    a.value = updateA
4                   b.value = updateB
                    true
                } else
5                   false
            }
```

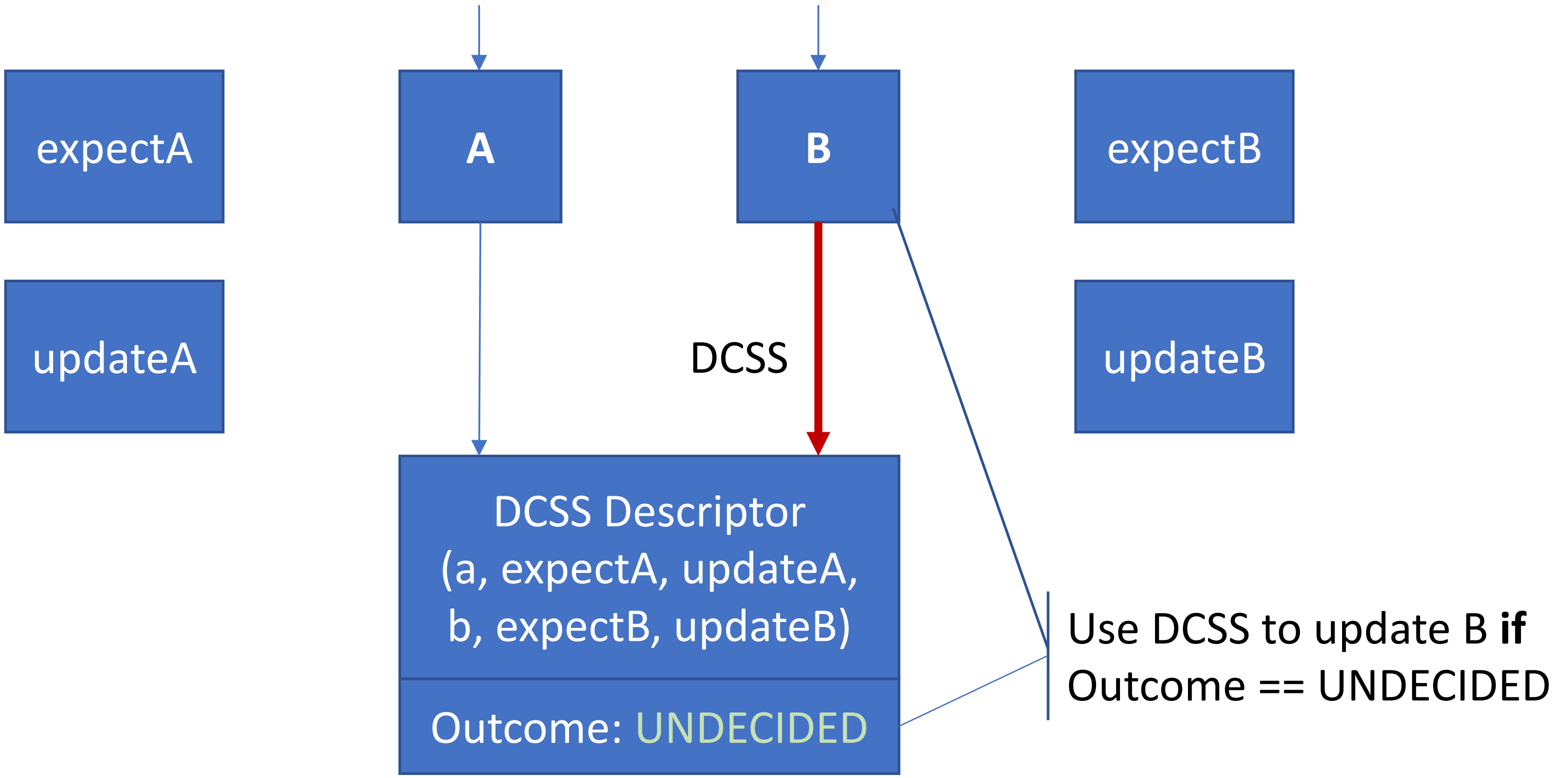
CASN: init descriptor



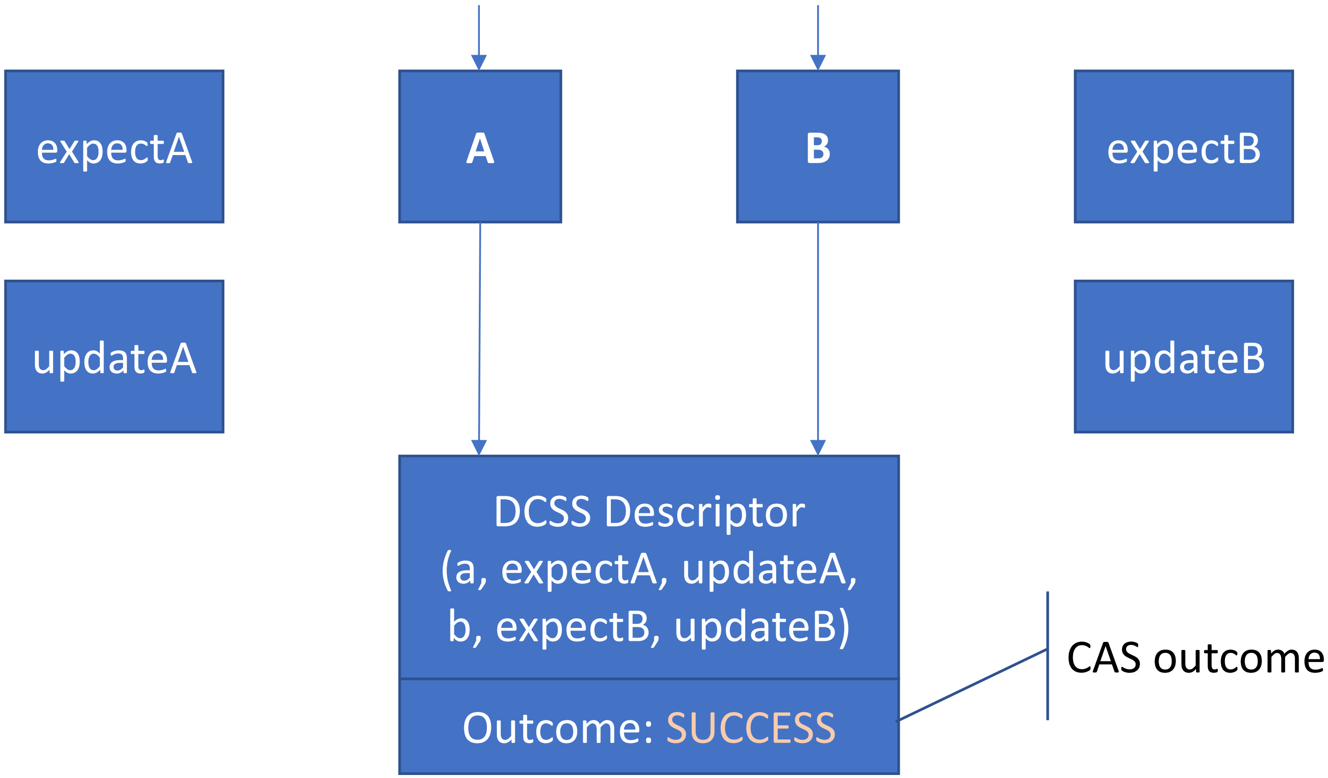
CASN: prepare (1)



CASN: prepare (2)

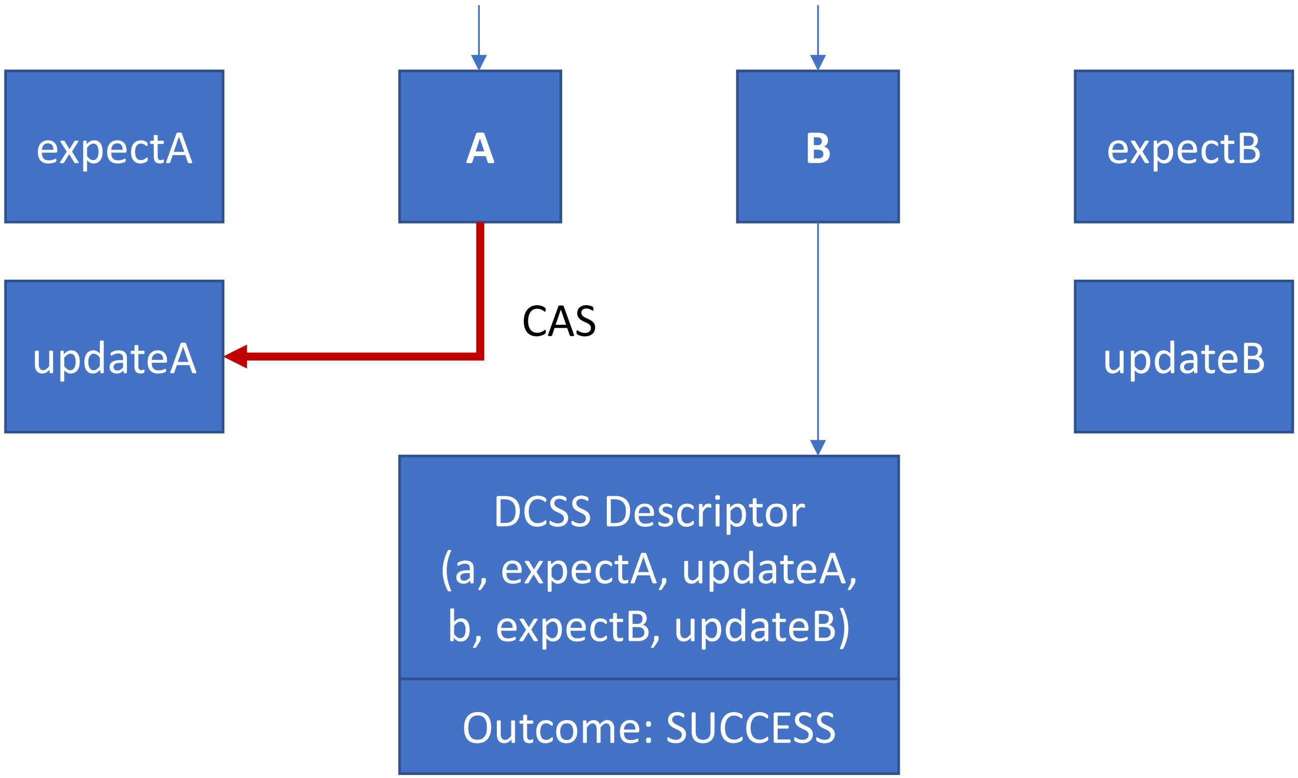


CASN: decide

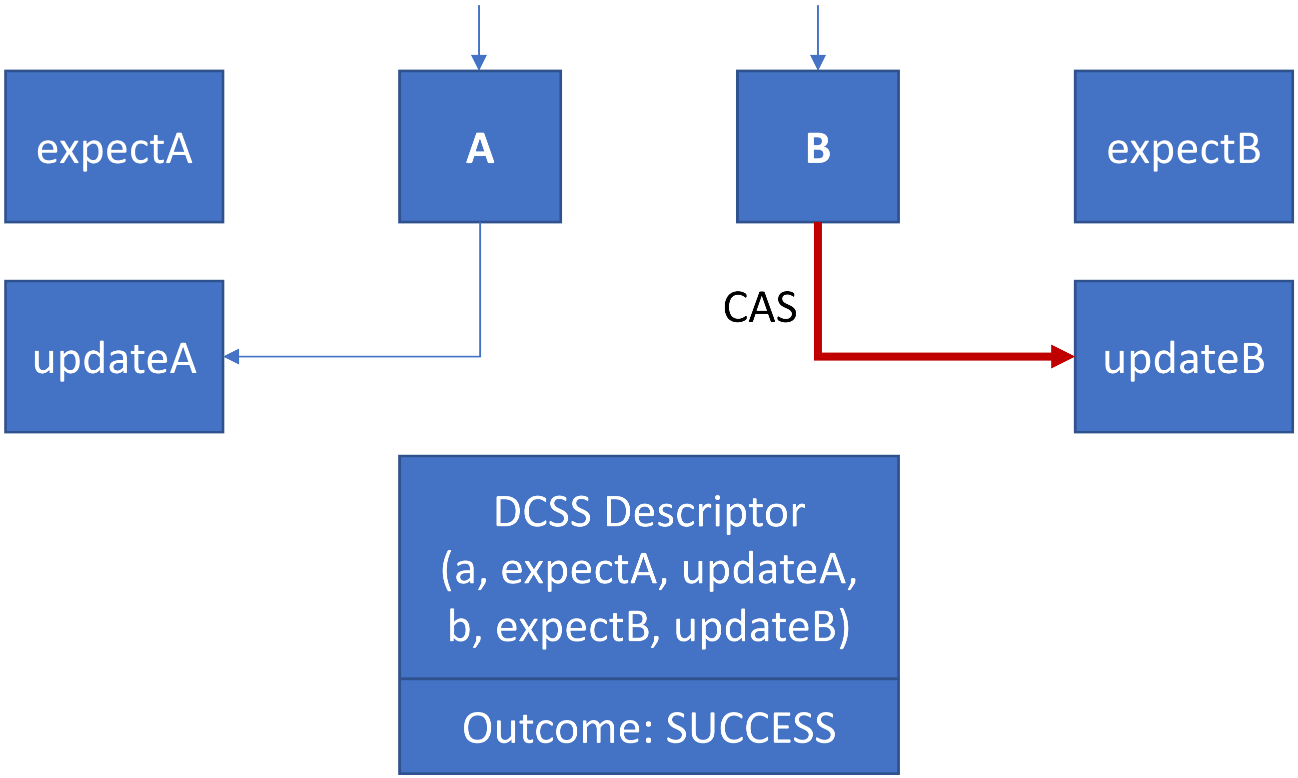


expectA
A
B
expectB
updateA
updateB
DCSS Descriptor
(a, expectA, updateA,
b, expectB, updateB)
Outcome: SUCCESS
CAS outcome

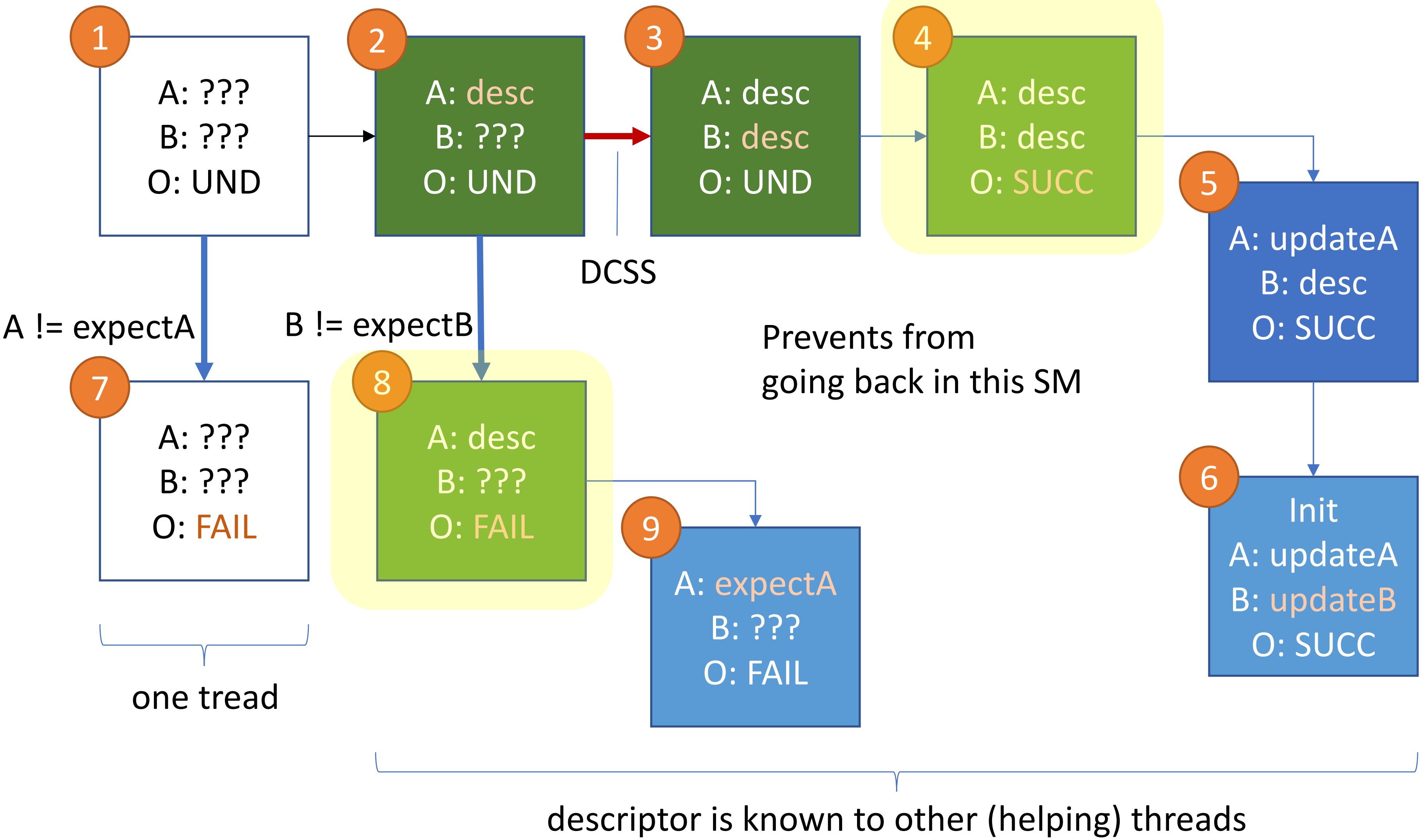
CASN: complete (1)



CASN: complete (2)



CASN: States



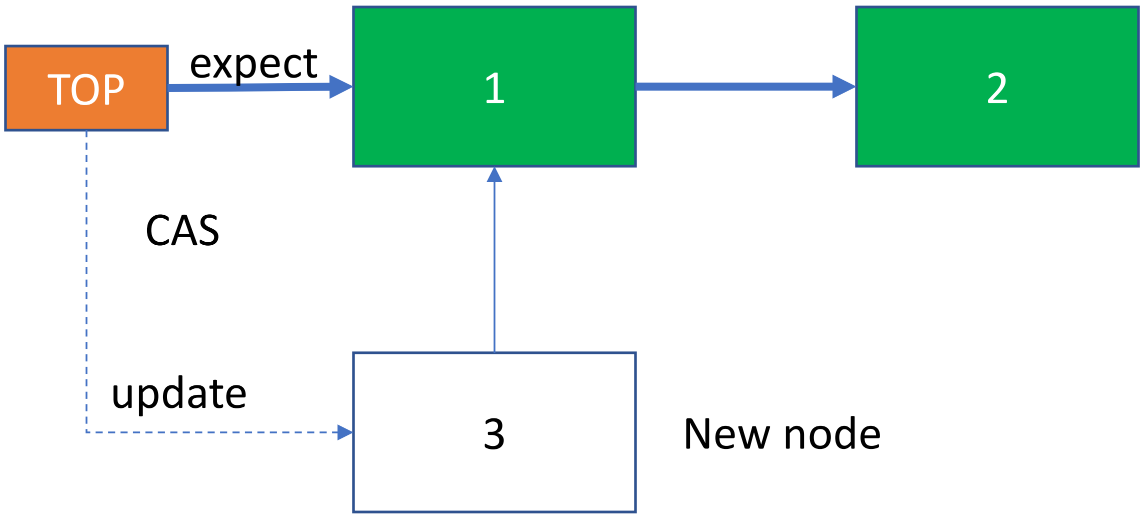
1
A: ???
B: ???
O: UND
2
A: desc
B: ???
O: UND
3
A: desc
B: desc
O: UND
4
A: desc
B: desc
O: SUCC
5
A: updateA
B: desc
O: SUCC
6
Init
A: updateA
B: updateB
O: SUCC
7
A: ???
B: ???
O: FAIL
8
A: desc
B: ???
O: FAIL
9
A: expectA
B: ???
O: FAIL
DCSS
A != expectA
B != expectB
Prevents from
going back in this SM
one tread
descriptor is known to other (helping) threads

Using it in practice

All the little things that matter

It is easy to combine multiple operations with DCSS/CASN that linearize on a **CAS** with a descriptor parameters that are **known in advance**

Trivial example: Treiber stack

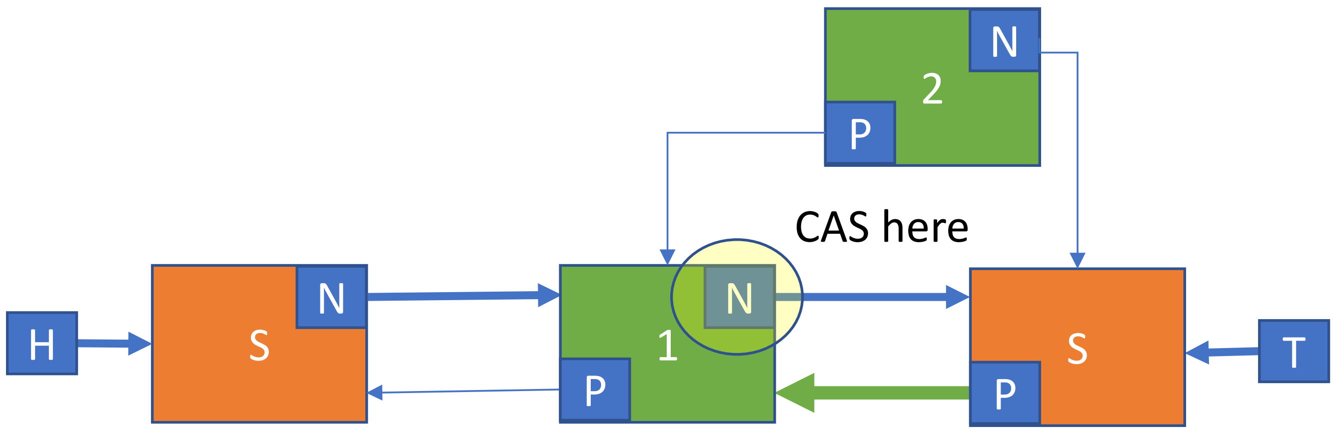


Let’s go deeper

Into unpublished territory

Doubly linked list: insert last

Doubly linked list: insert (0)



Operation Descriptor

A ref: ??? — ??? can fill in A before CAS & update on retry

expectA: Sentinel — We know expected value for CAS in advance

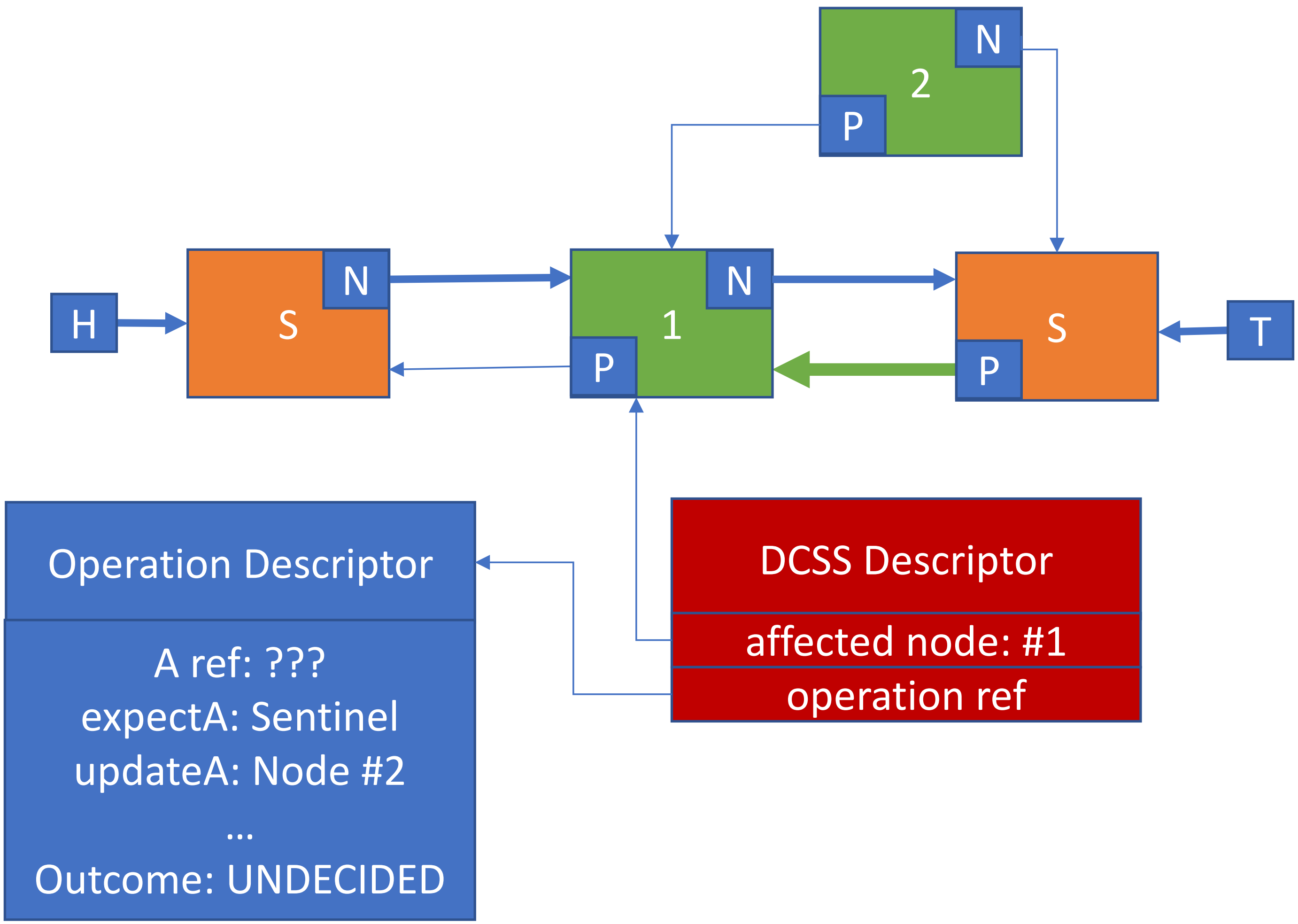
updateA: Node #2 — We know updated value for CAS in advance

…

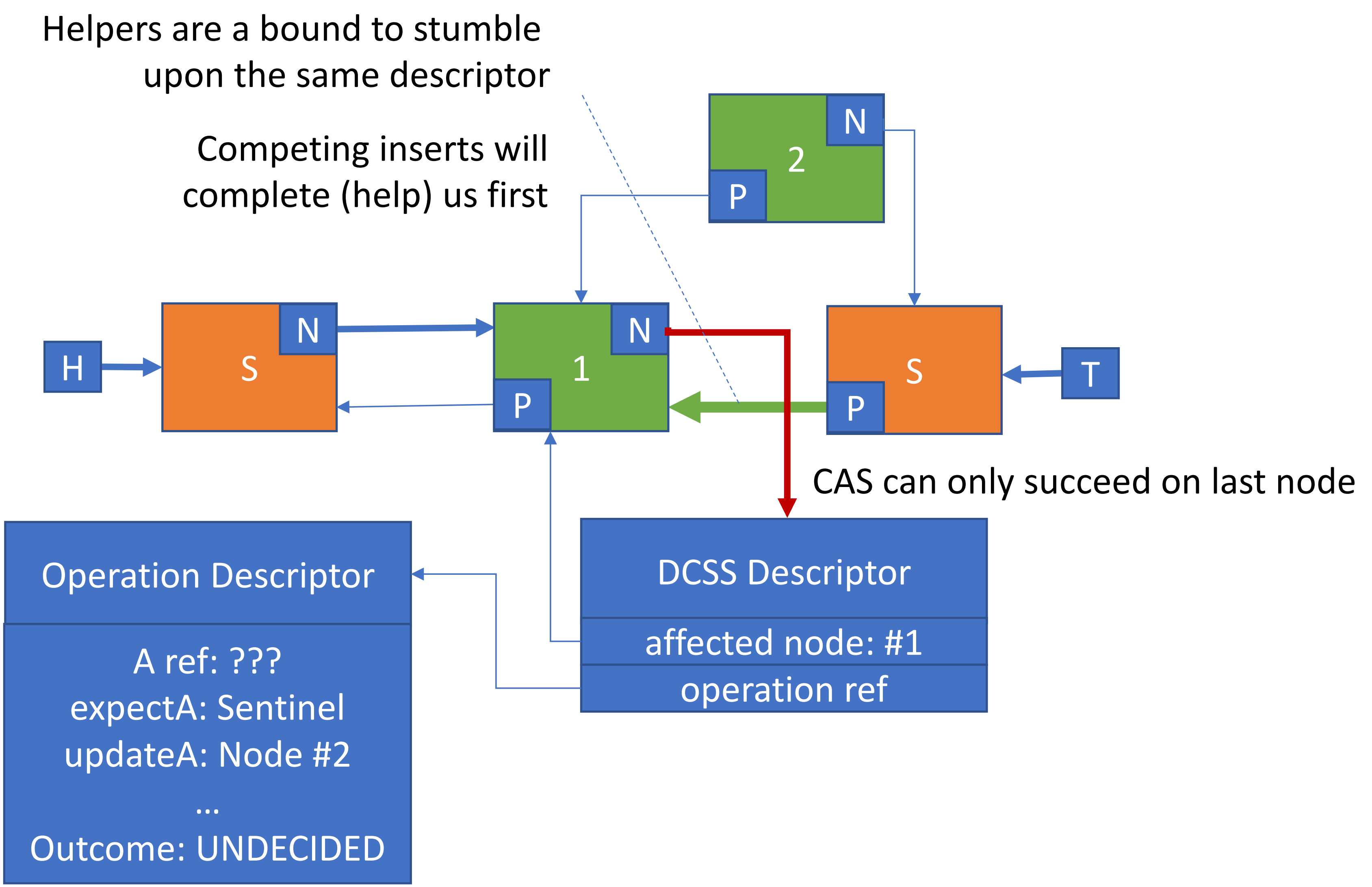
Outcome: UNDECIDED

DCSS here is needed (always!)

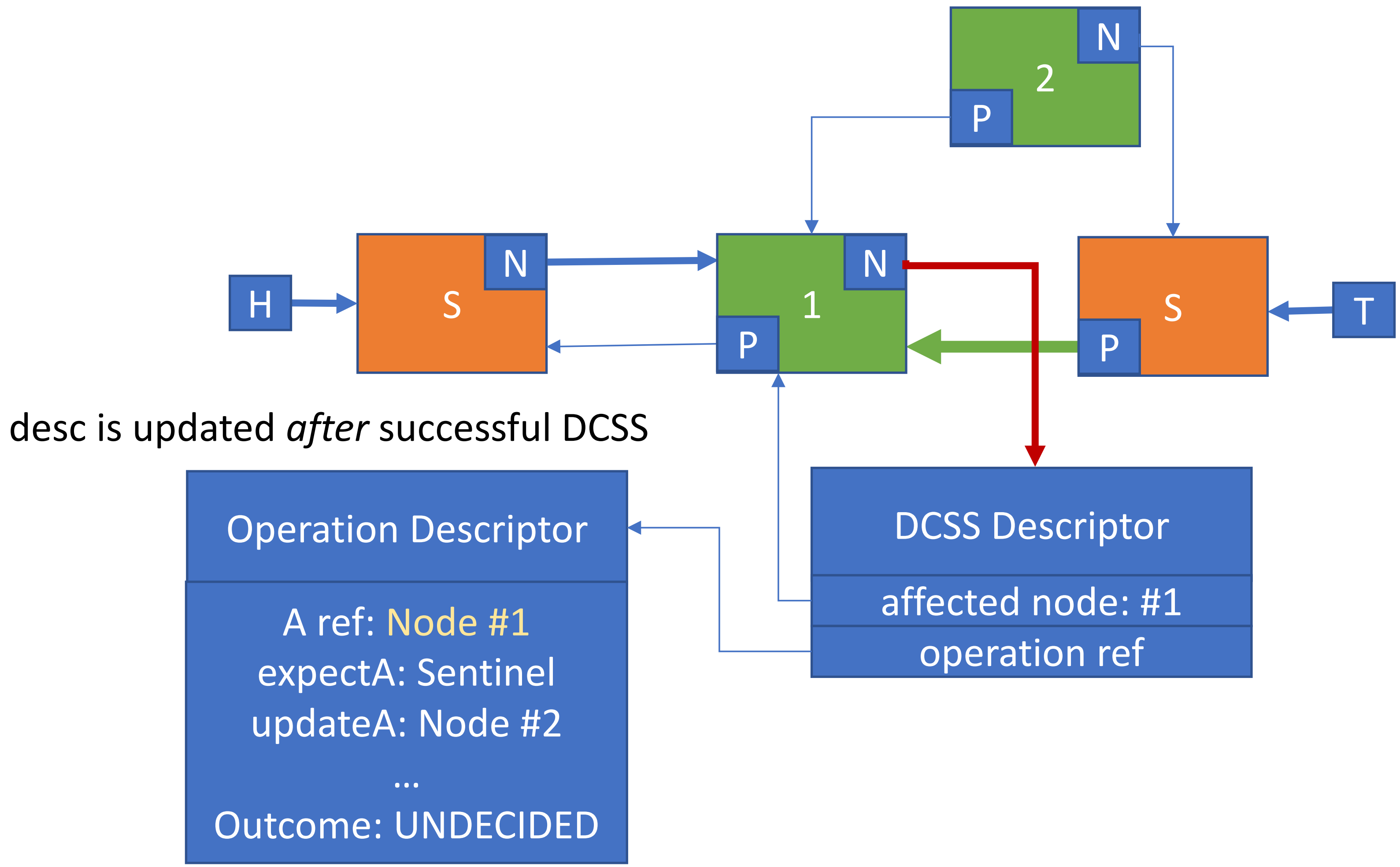
Doubly linked list: insert (1)



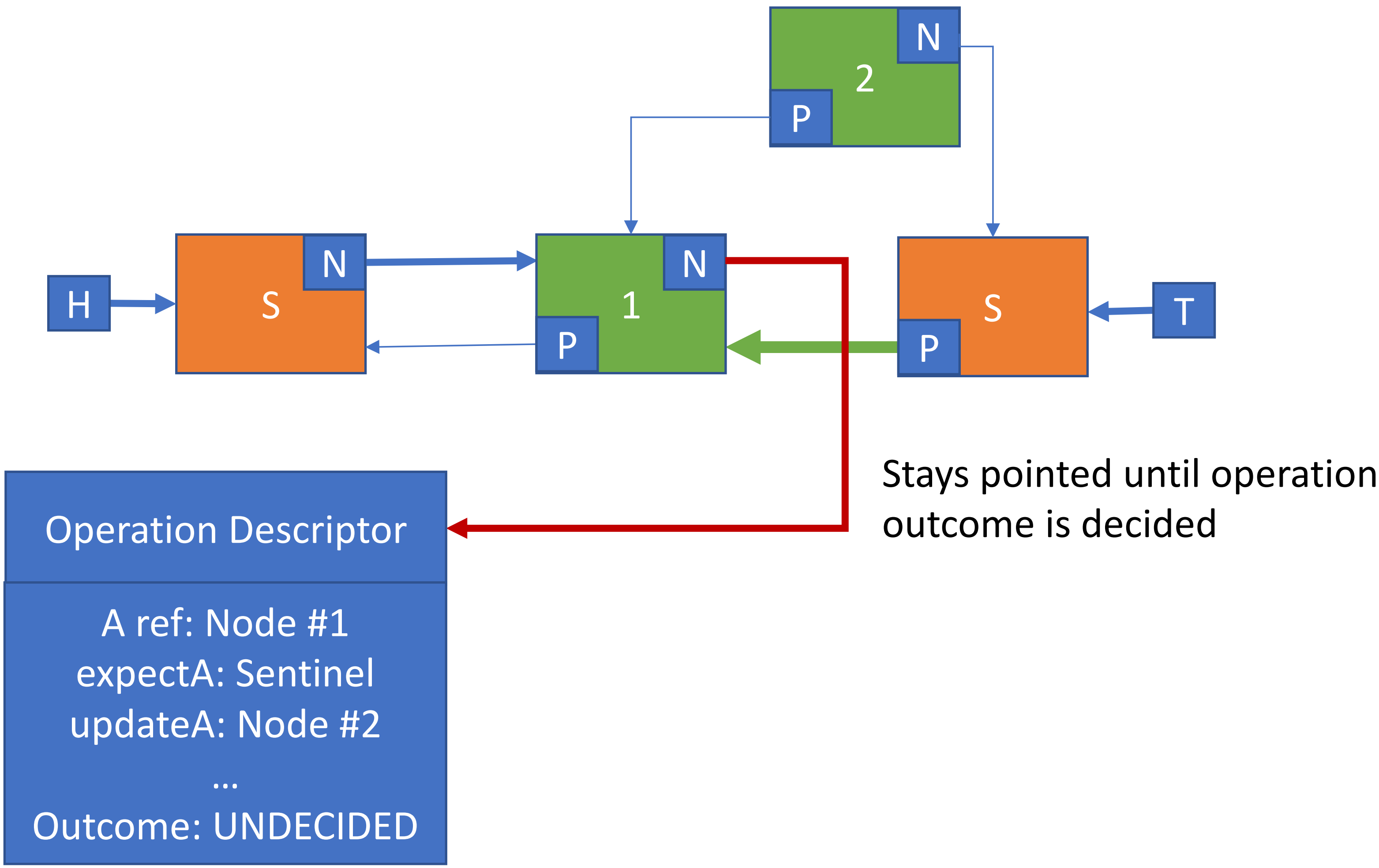
Doubly linked list: insert (2)



Doubly linked list: insert (3)

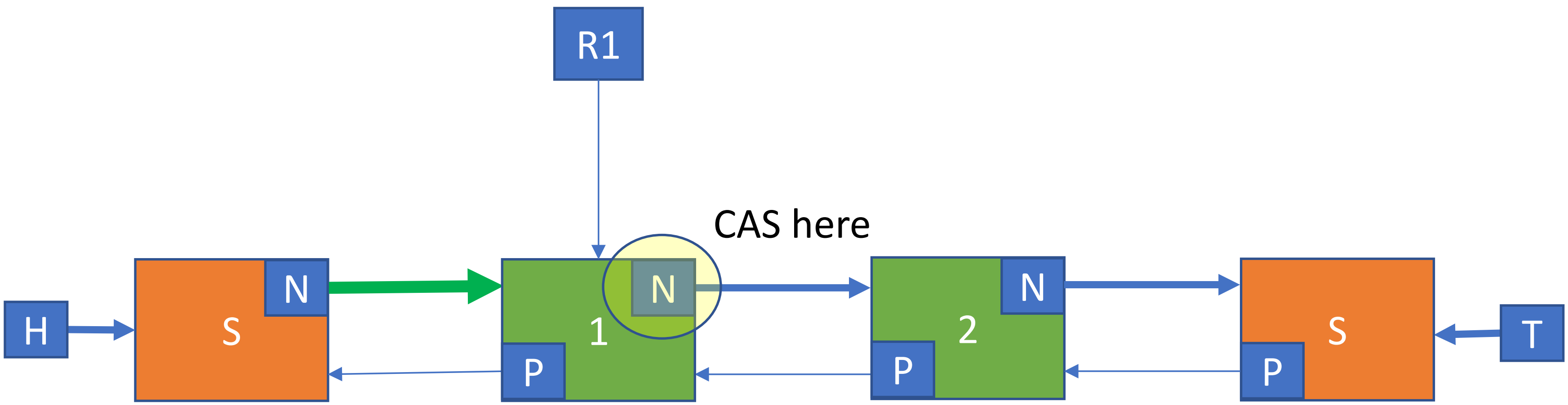


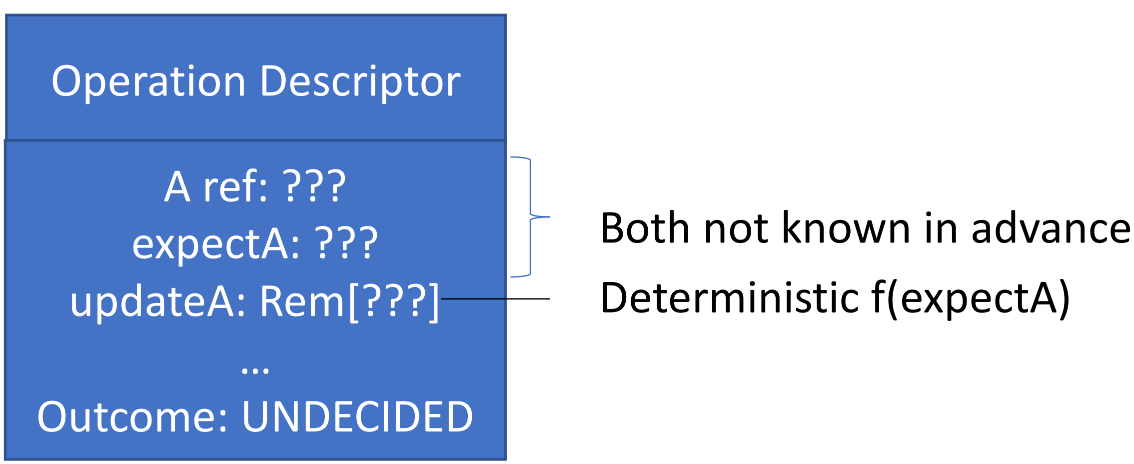
Doubly linked list: insert (4)



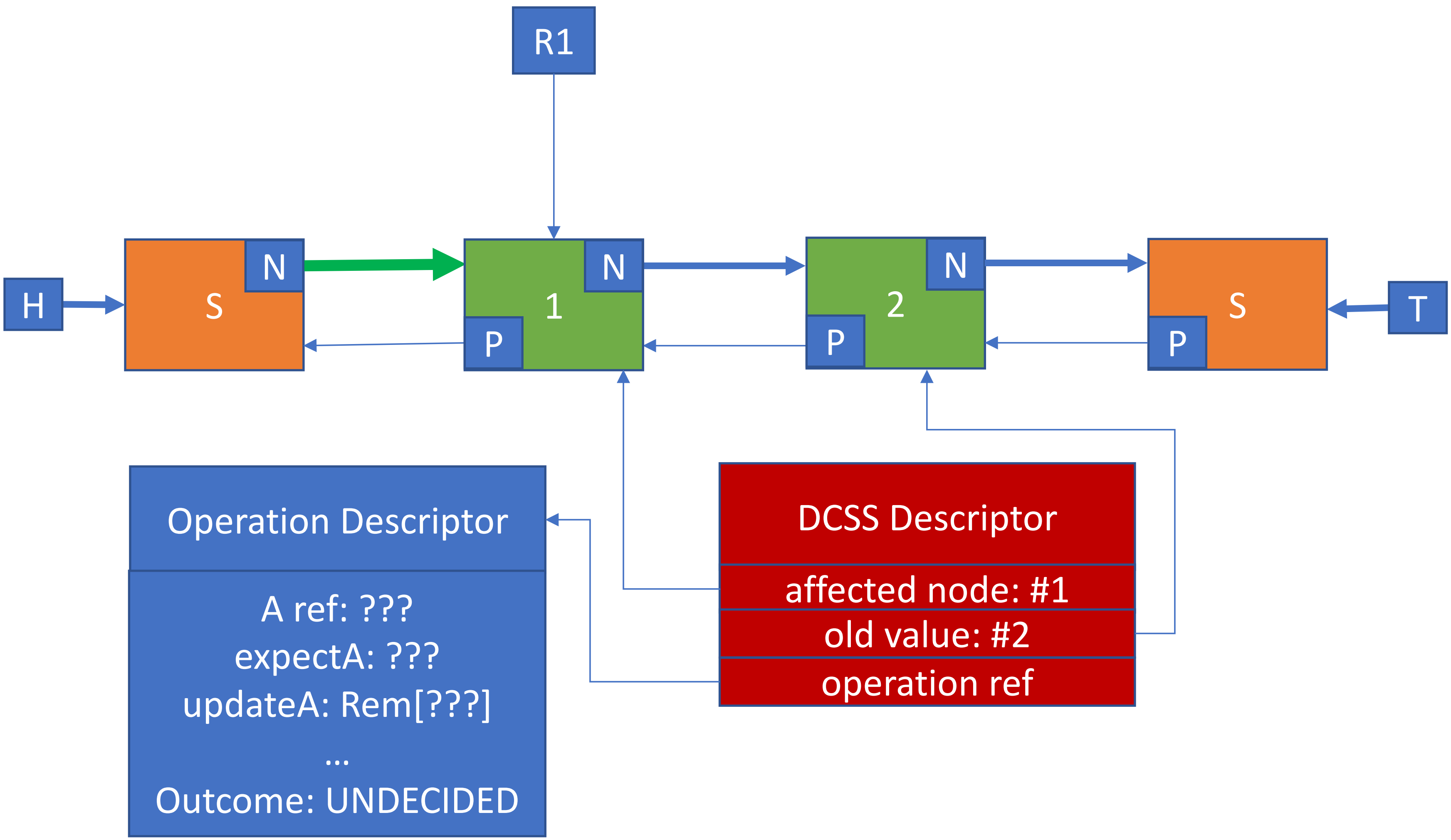
Doubly linked list: remove first

Doubly linked list: remove (0)

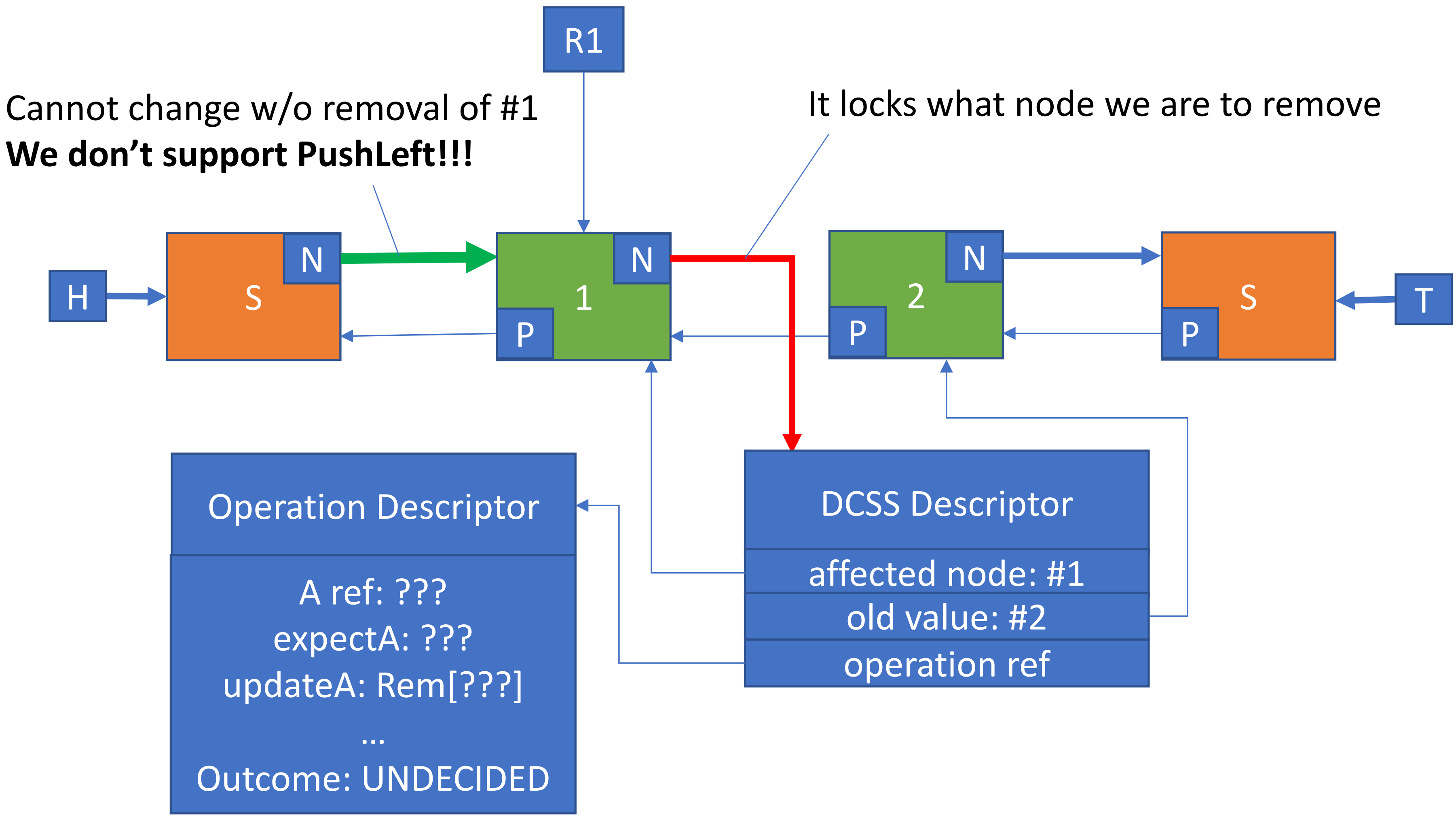




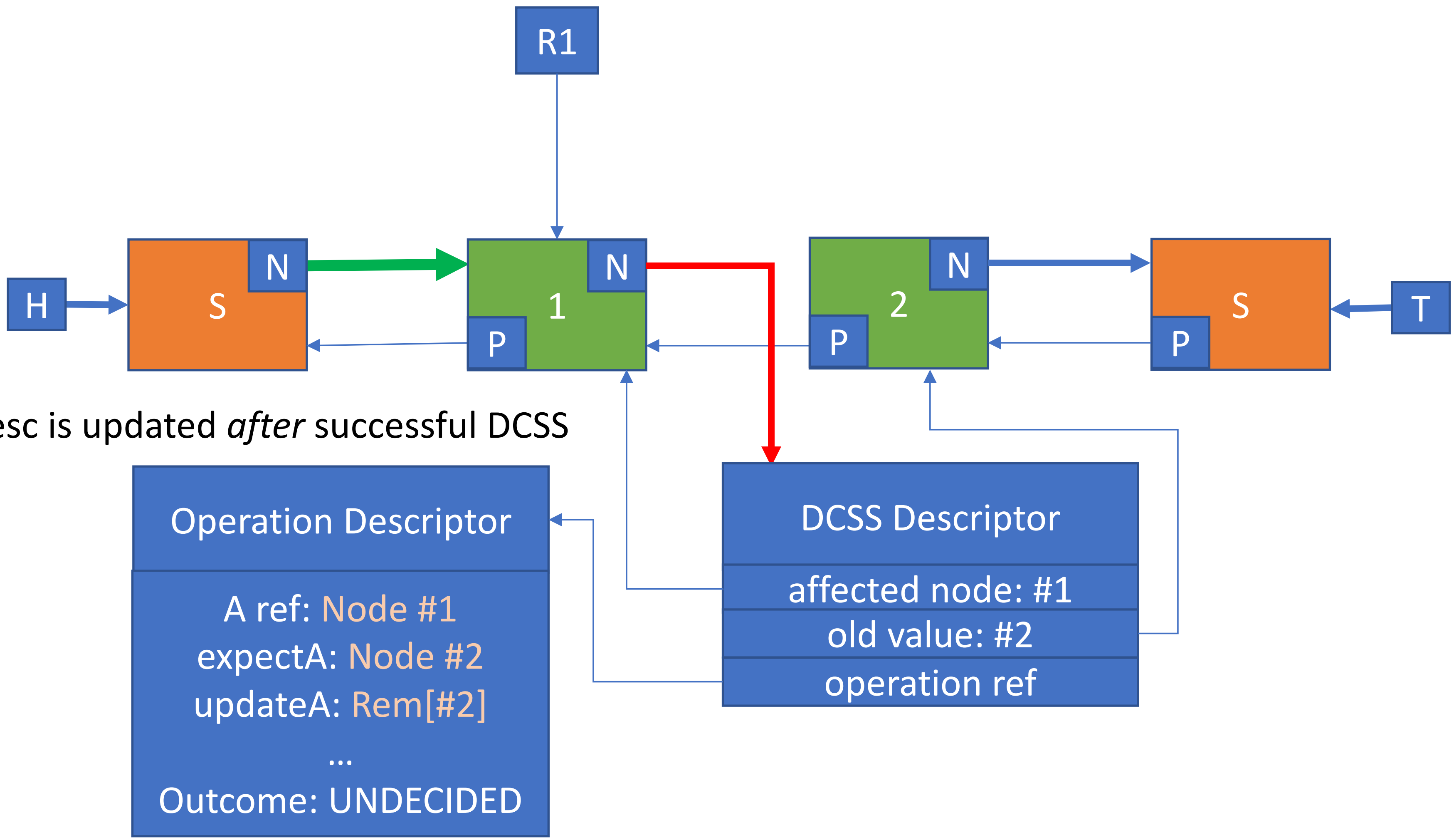
Doubly linked list: remove (1)



Doubly linked list: remove (2)

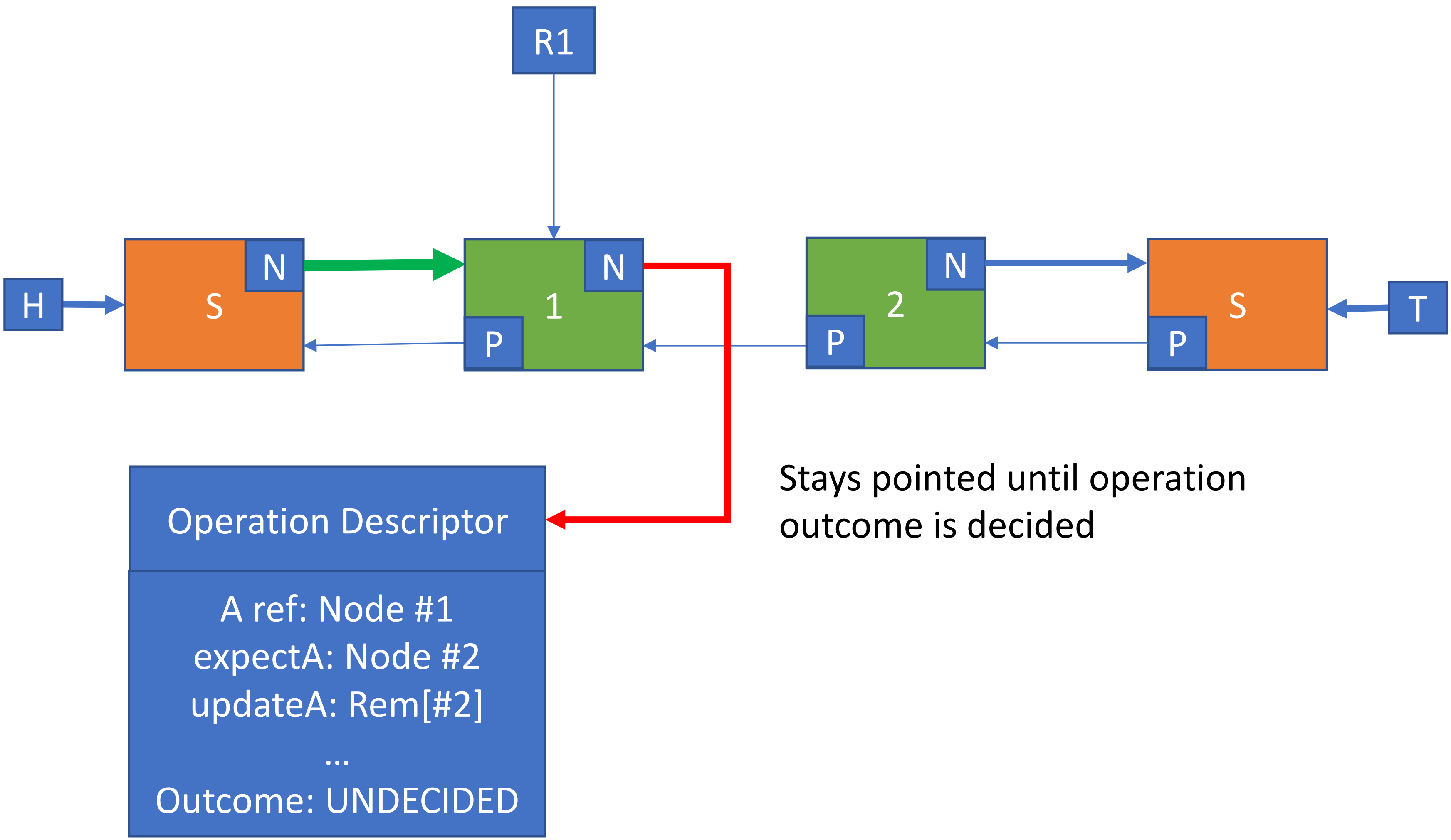


Doubly linked list: remove (3)



desc is updated *after* successful DCSS

Doubly linked list: remove (4)



Closing notes

- All we care about is **CAS** that linearizes operation
- Subsequent updates are *helper* moves
  - Invoke regular help/correct functions
- Perfect algorithm to combine with optional Hardware Transactional Memory (HTM)

Let’s enjoy what we’ve accomplished

References

- Kotlin language
  - http://kotlinlang.org
- Kotlin coroutines support library
  - http://github.com/kotlin/kotlinx.coroutines

Thank you

Any questions?

email me to **elizarov** at gmail

relizarov