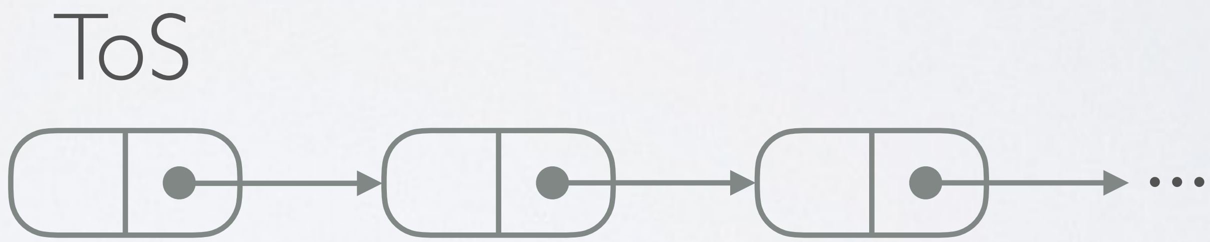
Thread-Modular Reasoning for Heap-Manipulating Programs: Exploiting Pointer Race Freedom

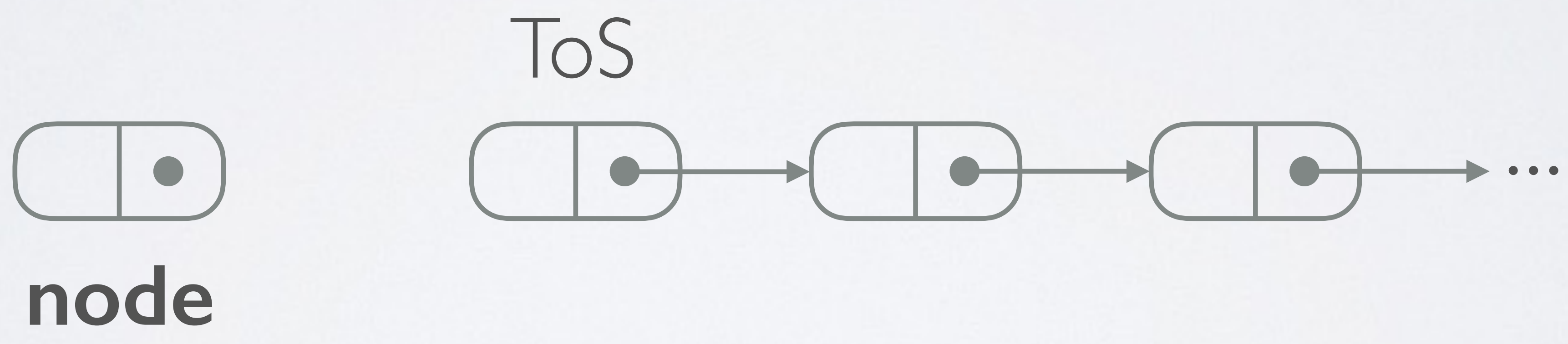
— Sebastian Wolff —

Treiber's stack



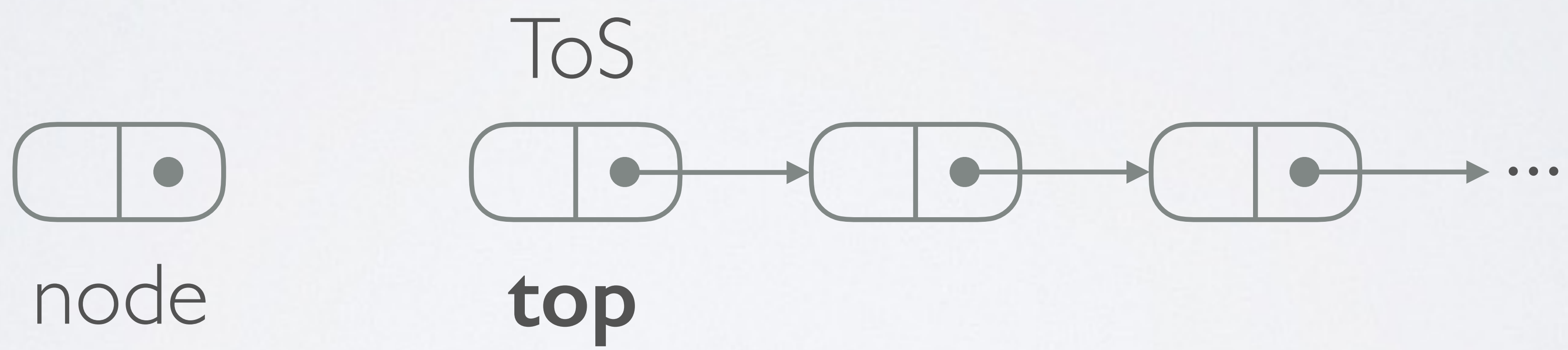
Treiber's stack

push: 1. allocate new node



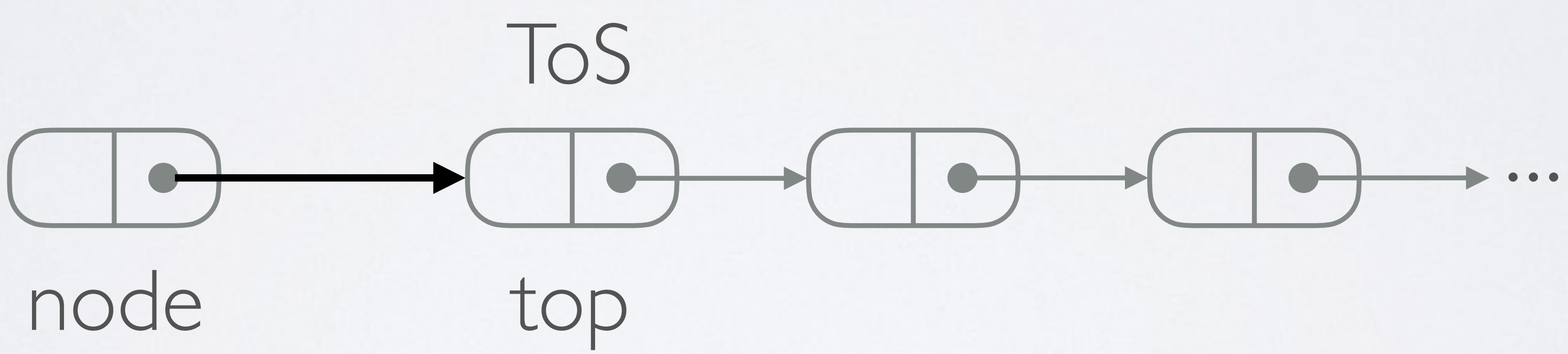
Treiber's stack

push: 2. read top of stack



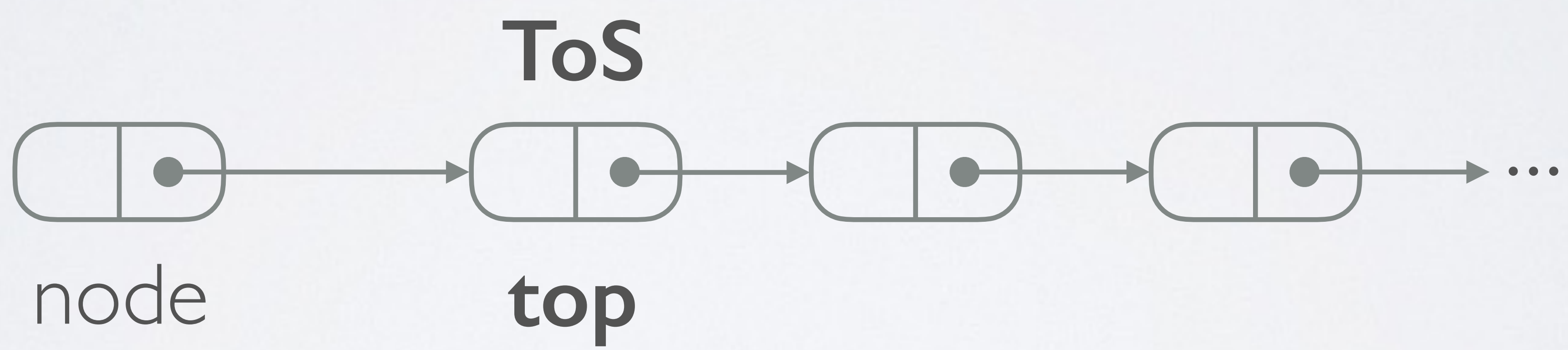
Treiber's stack

push: 3. connect new node



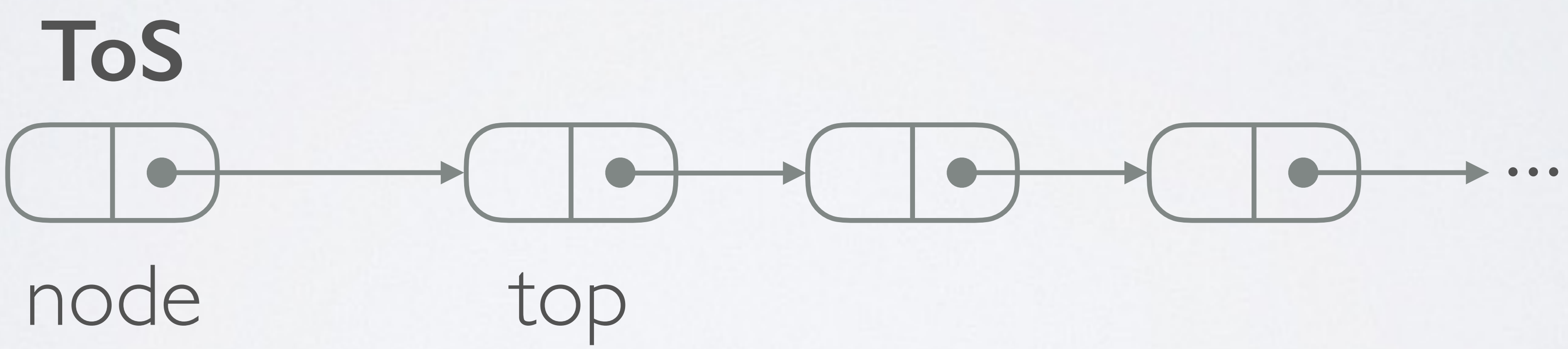
Treiber's stack

push: 4. move top of stack if consistent (`CAS`),
otherwise go back to step 2



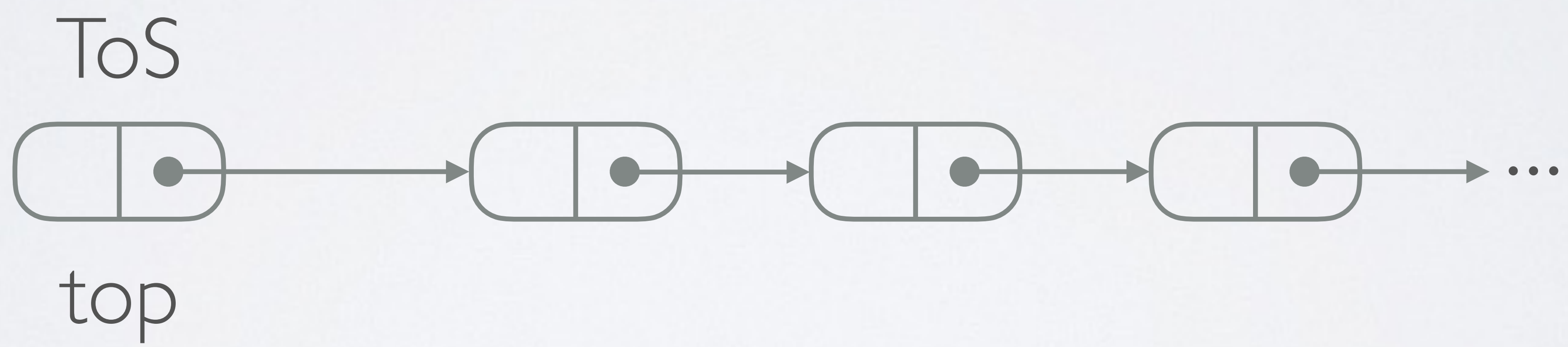
Treiber's stack

push: 4. move top of stack if consistent (`CAS`),
otherwise go back to step 2



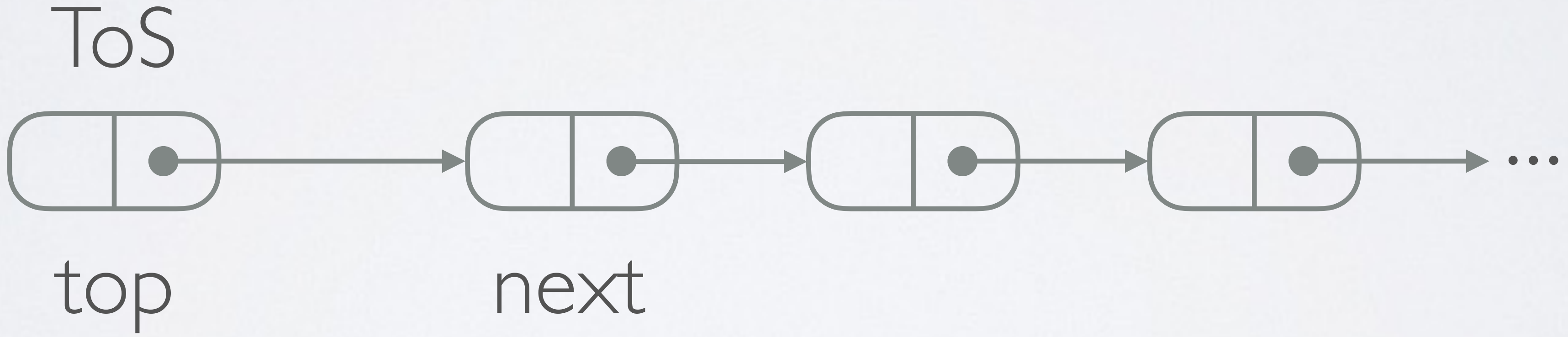
Treiber's stack

pop: 1. read top of stack



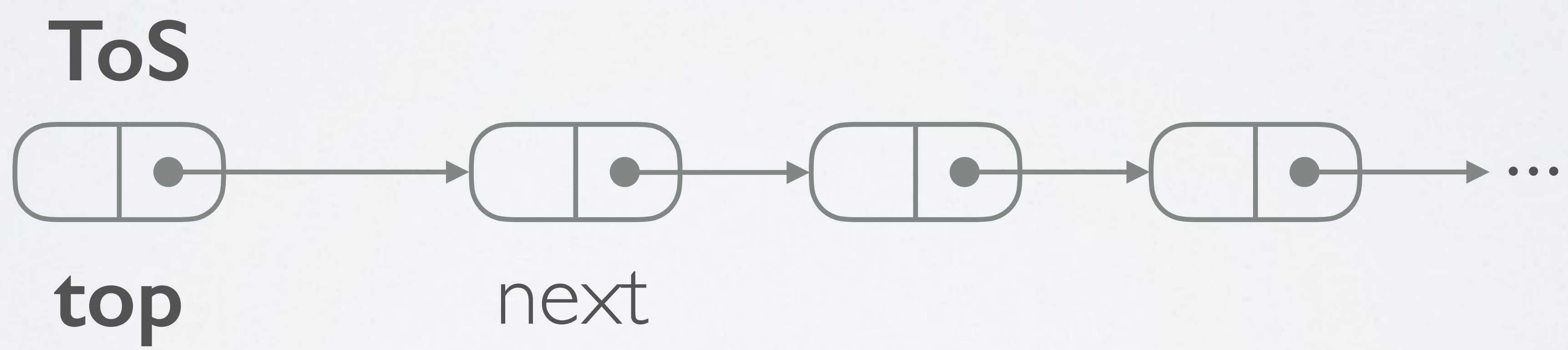
Treiber's stack

pop: 2. read the second topmost node



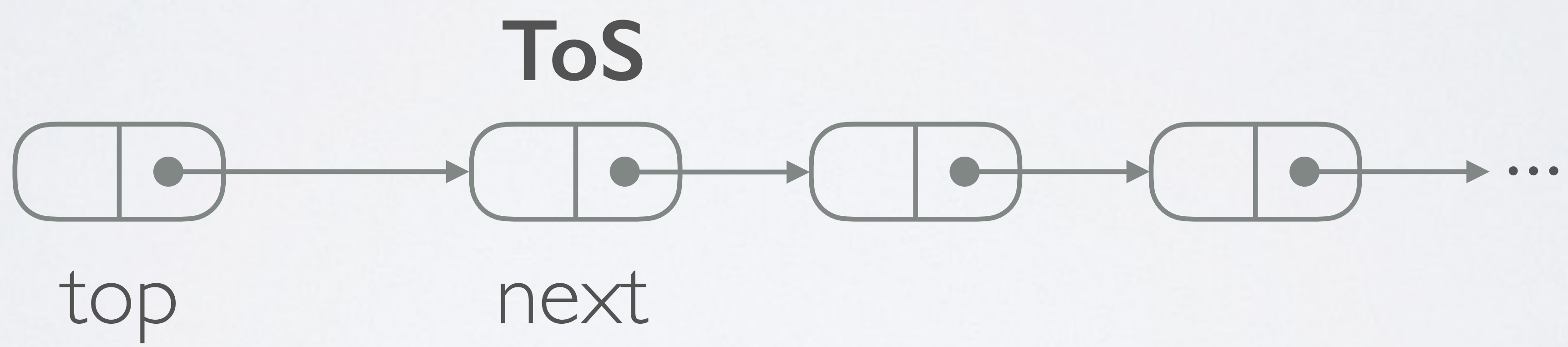
Treiber's stack

pop: 3. move top of stack if consistent (`CAS`),
go back to step 1 otherwise



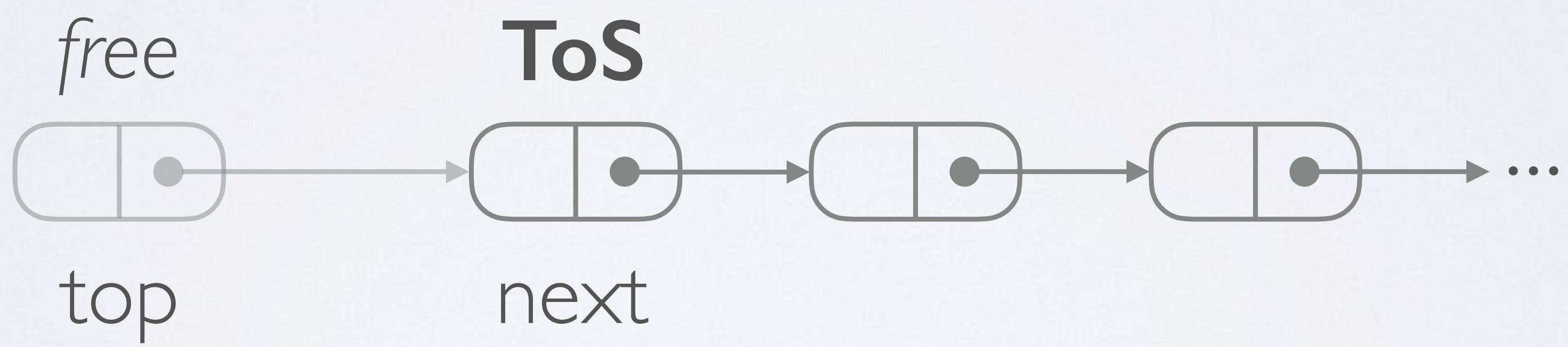
Treiber's stack

pop: 3. move top of stack if consistent (`CAS`),
go back to step 1 otherwise



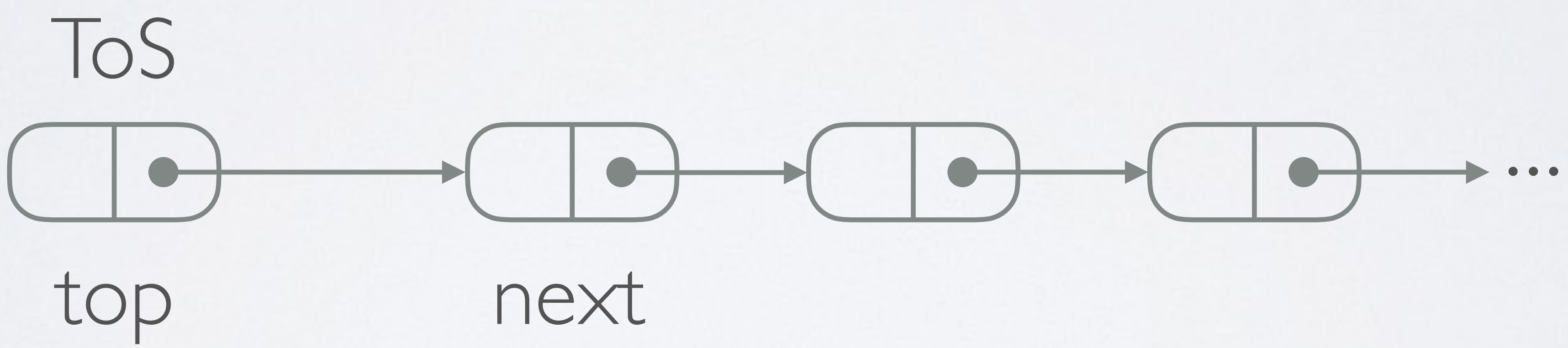
Treiber's stack

pop: 4. read out data, then free



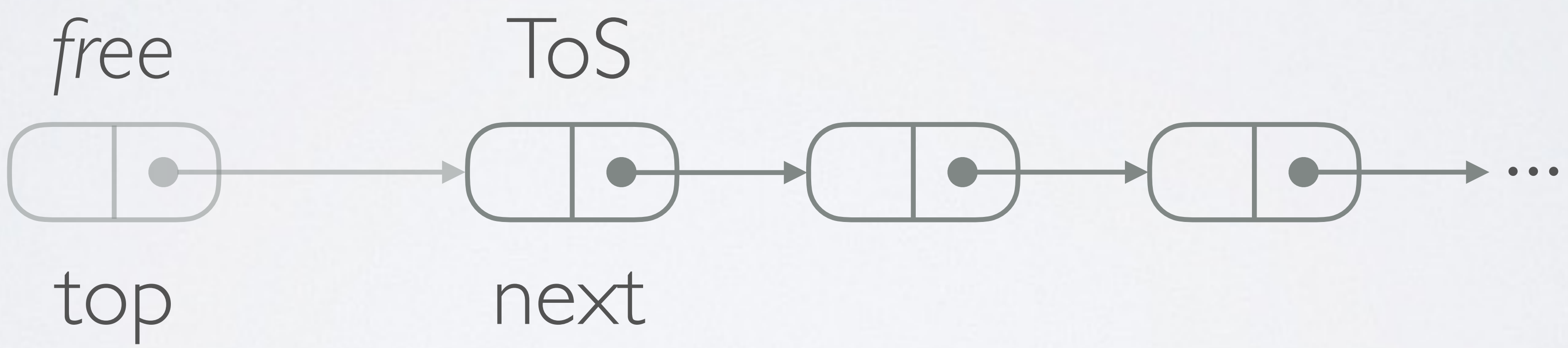
There is a bug

thread 1: is in step 3 of pop, but interrupted



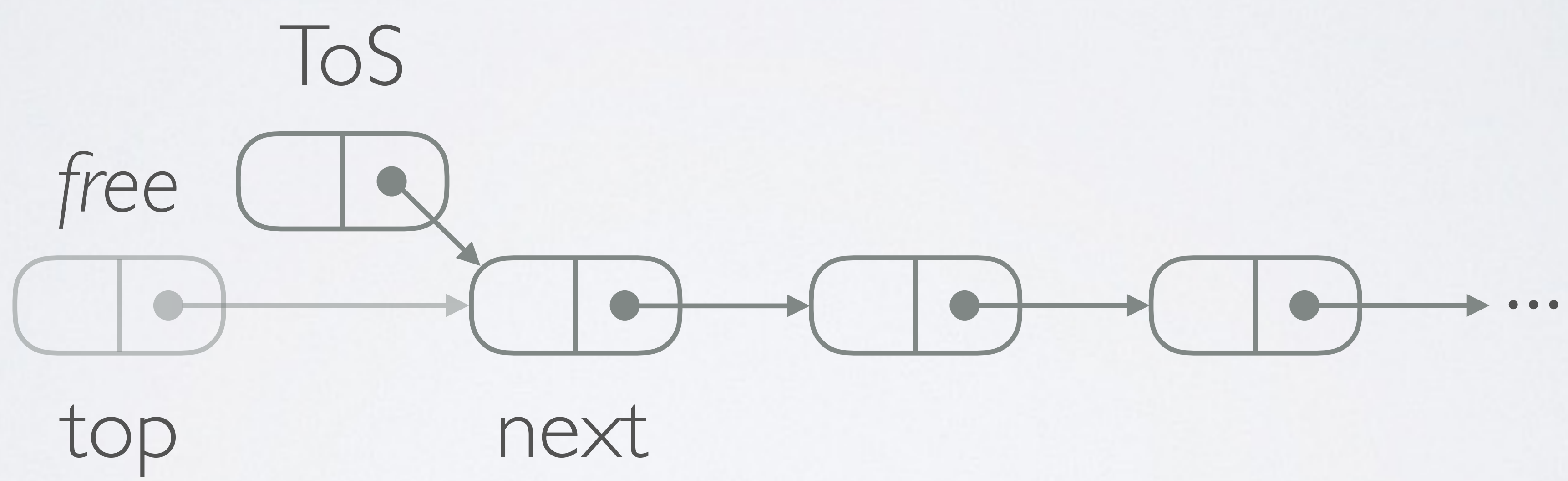
There is a bug

thread 2: pops



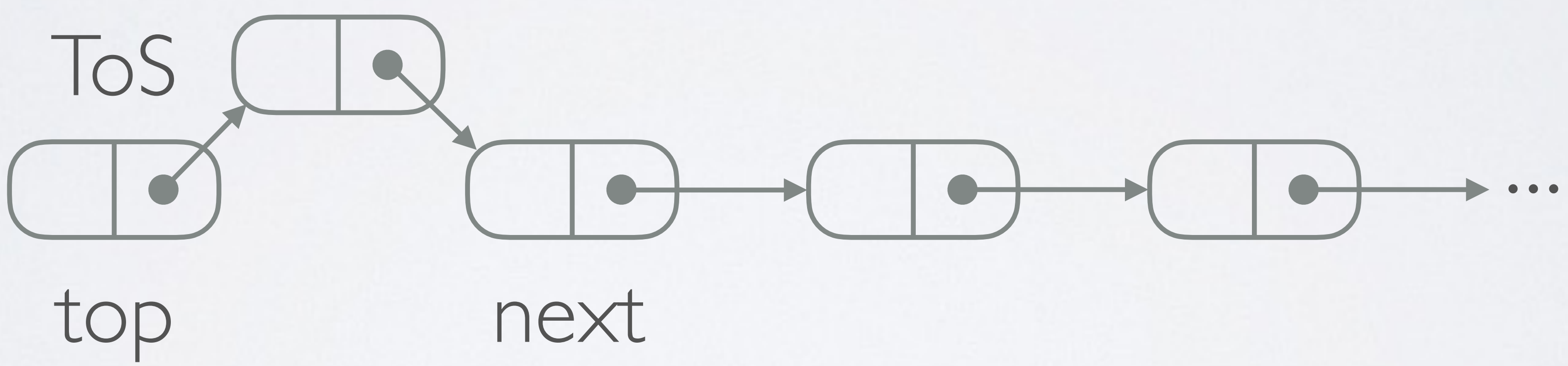
There is a bug

thread 2: pops, pushes



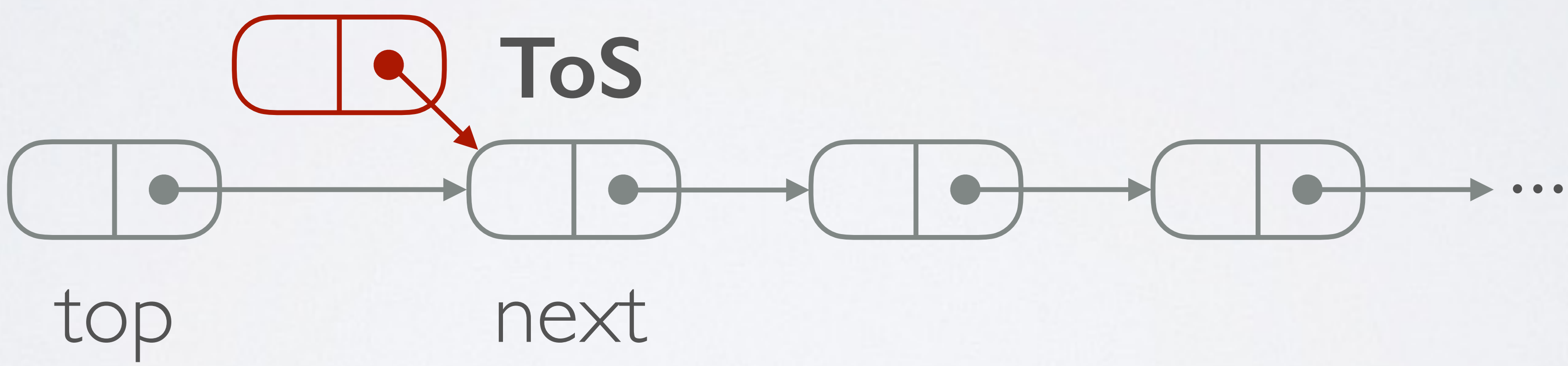
There is a bug

thread 2: pops, pushes, and pushes again



There is a bug

thread 1: continues and moves the top of stack



Analysis Requirements

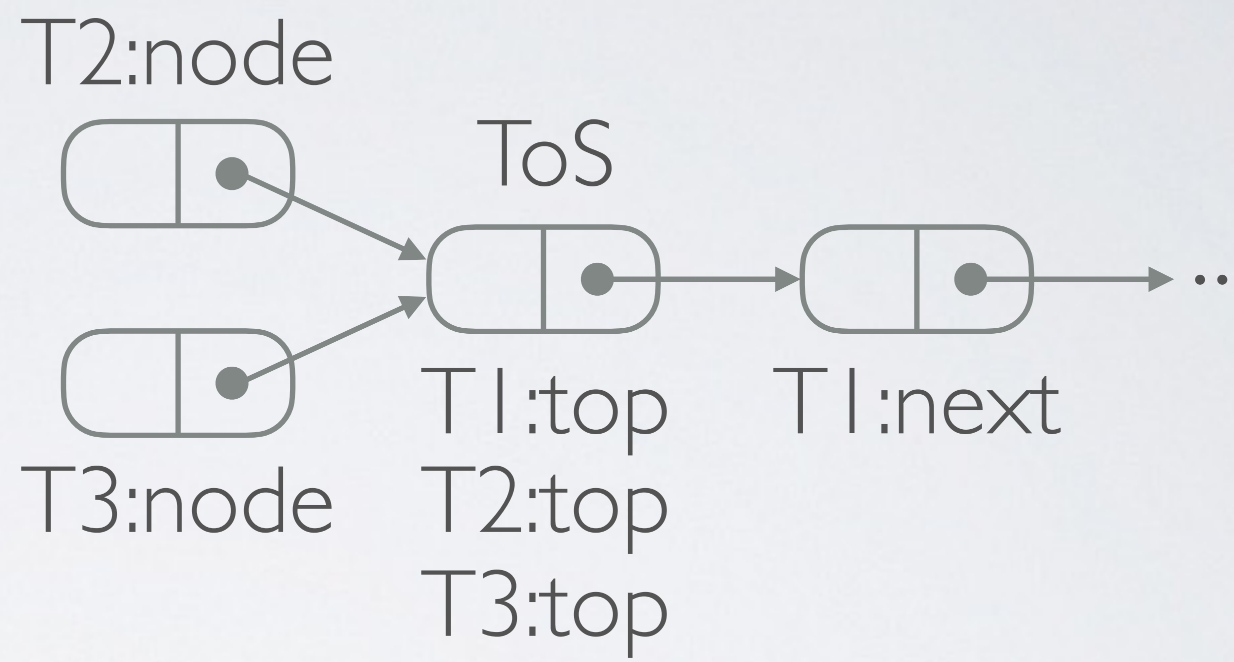
- prove correctness (linearisability)
- unbounded number of threads (library)
- unbounded heap
- low-level memory operations (C-like)
- scalability

Thread-Modular Reasoning

- abstract domain: set of views
  - single thread
  - heap reachable by that thread
  - relation among threads lost
- sequential + interference steps

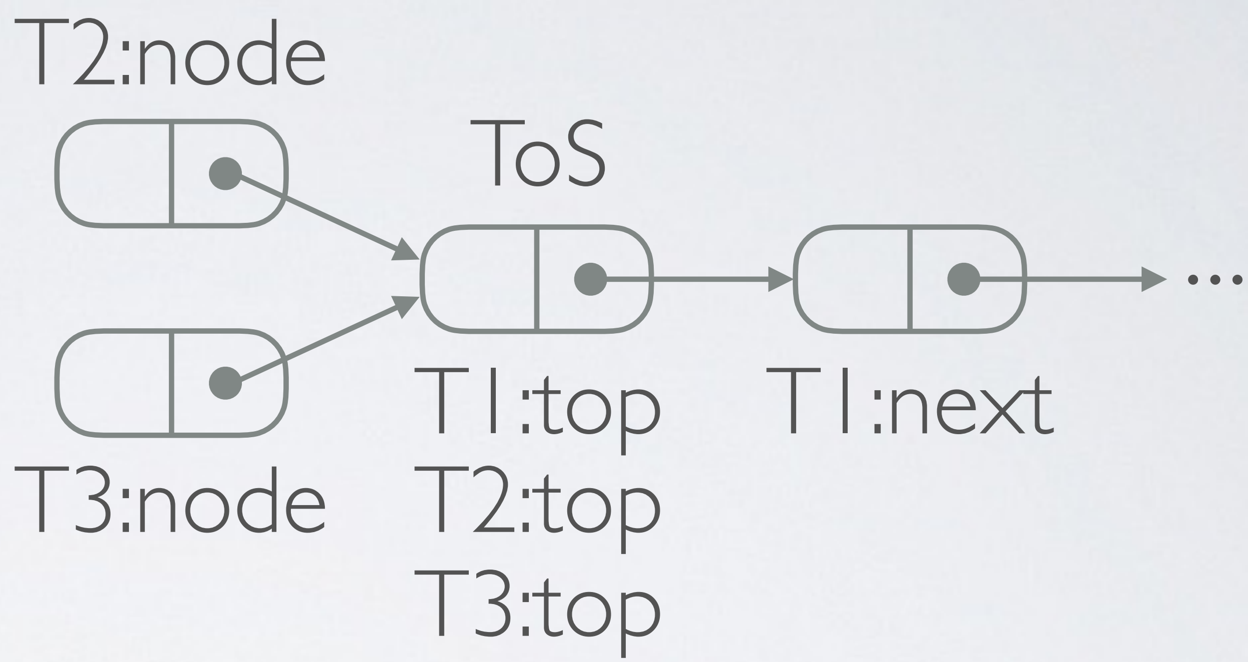
Example

T1.pc = pop:CAS
T2.pc = push:CAS
T3.pc = push:CAS

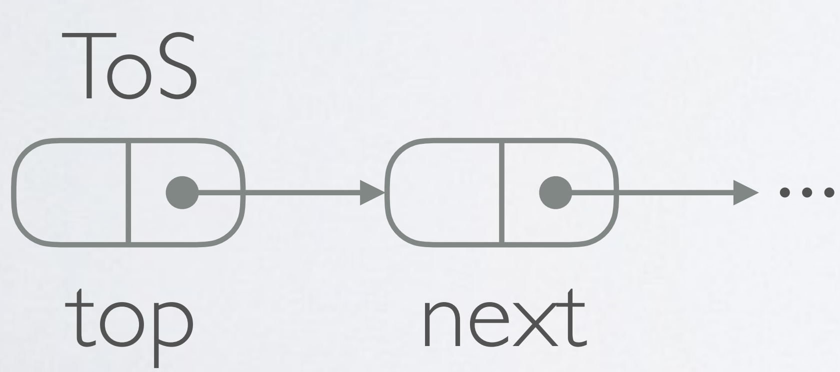


Example

T1.pc = pop:CAS
T2.pc = push:CAS
T3.pc = push:CAS

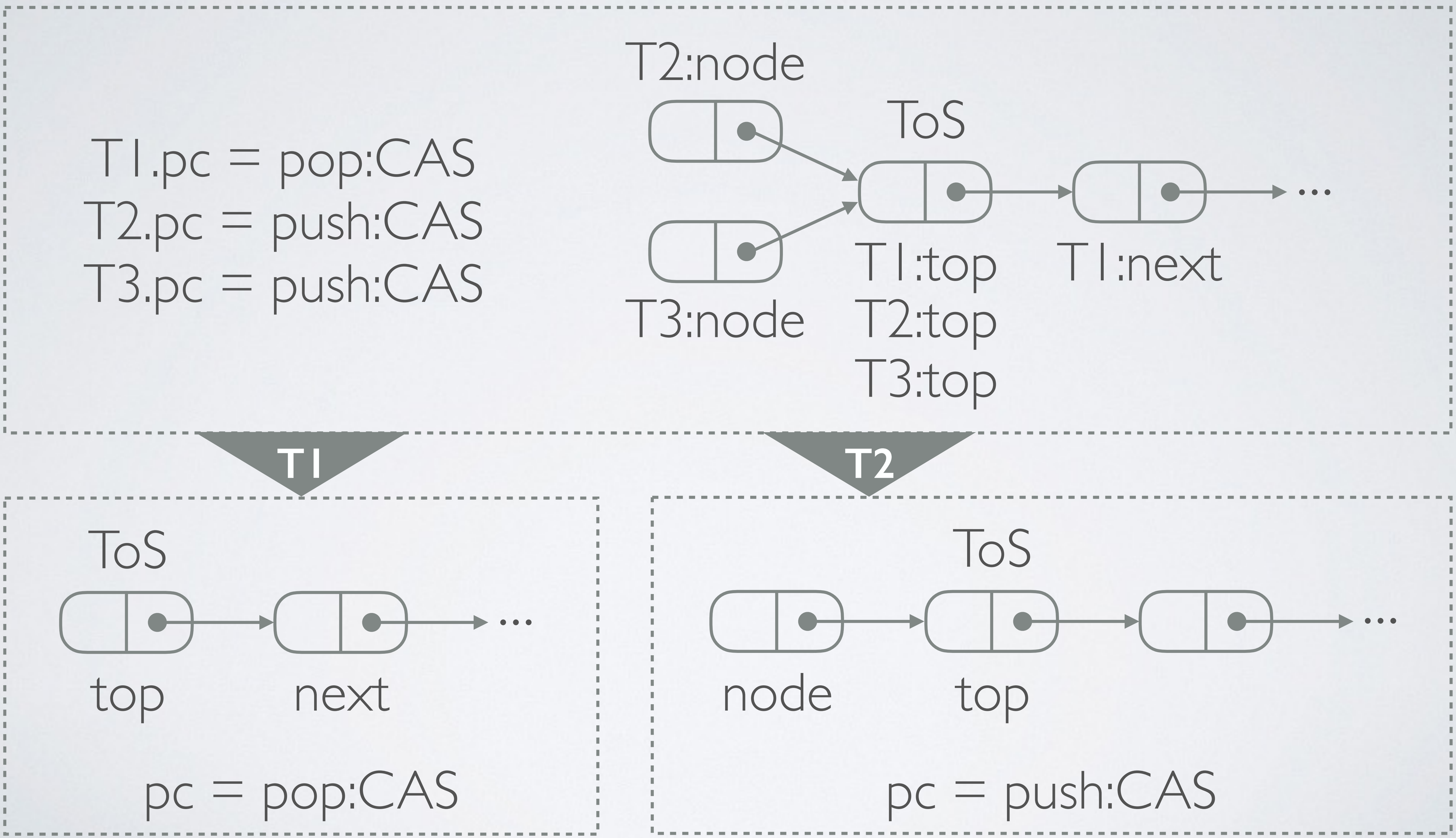


T1



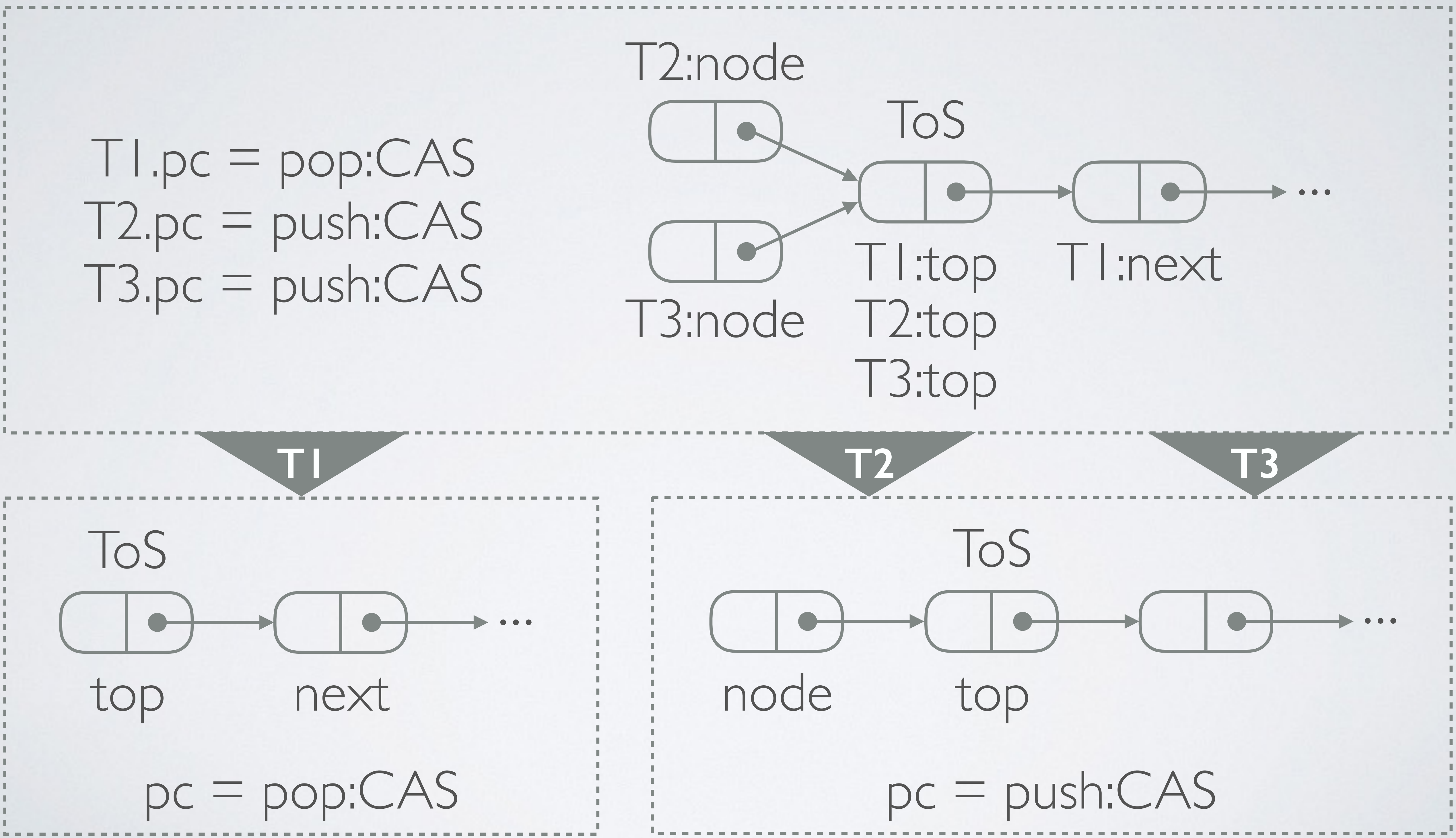
pc = pop:CAS

Example

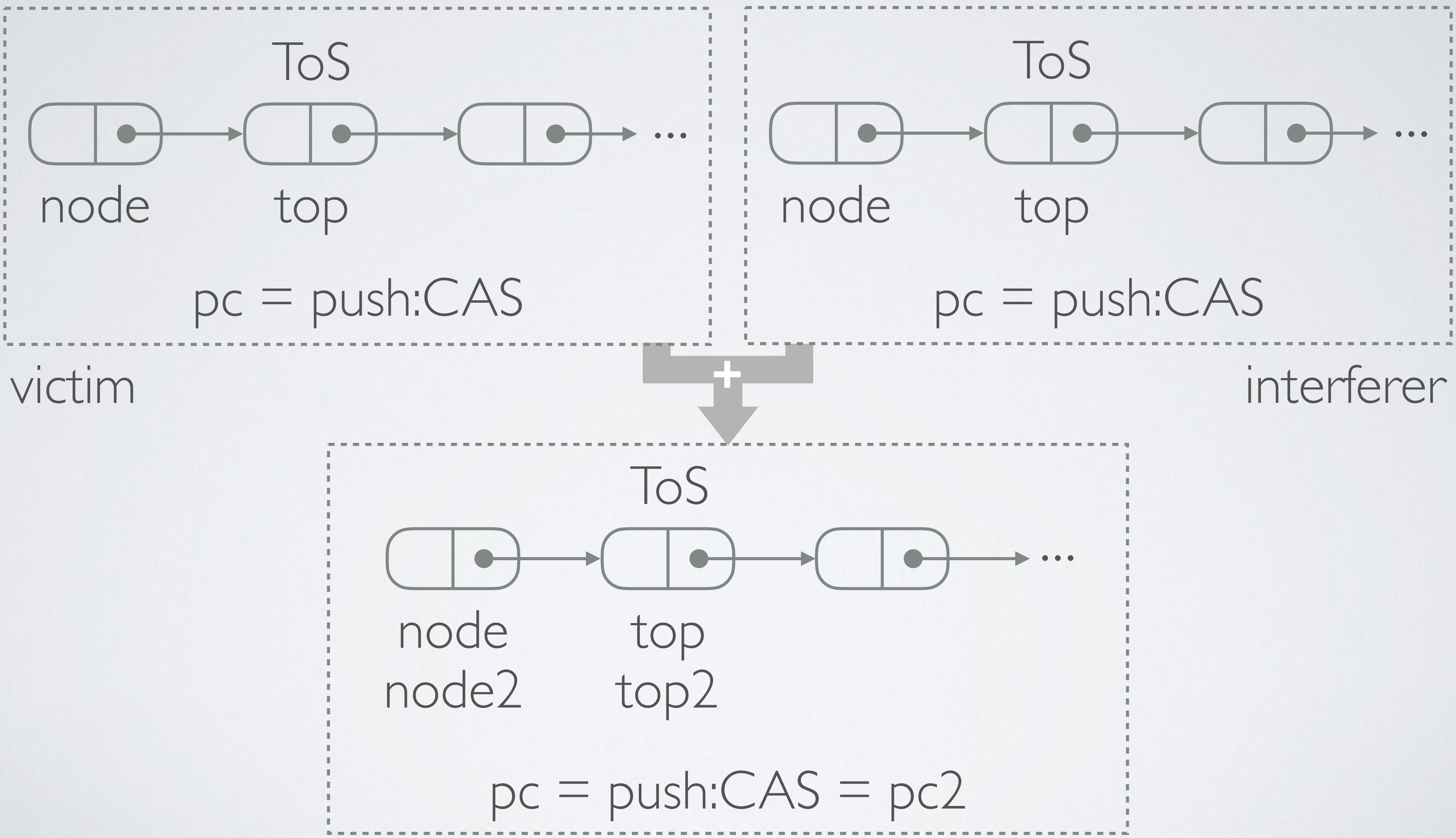


T1.pc = pop:CAS
T2.pc = push:CAS
T3.pc = push:CAS
T2:node
ToS
…
T1:top
T2:top
T3:top
T1:next
T3:node
T1
T2
ToS
…
top
next
pc = pop:CAS
ToS
…
node
top
pc = push:CAS

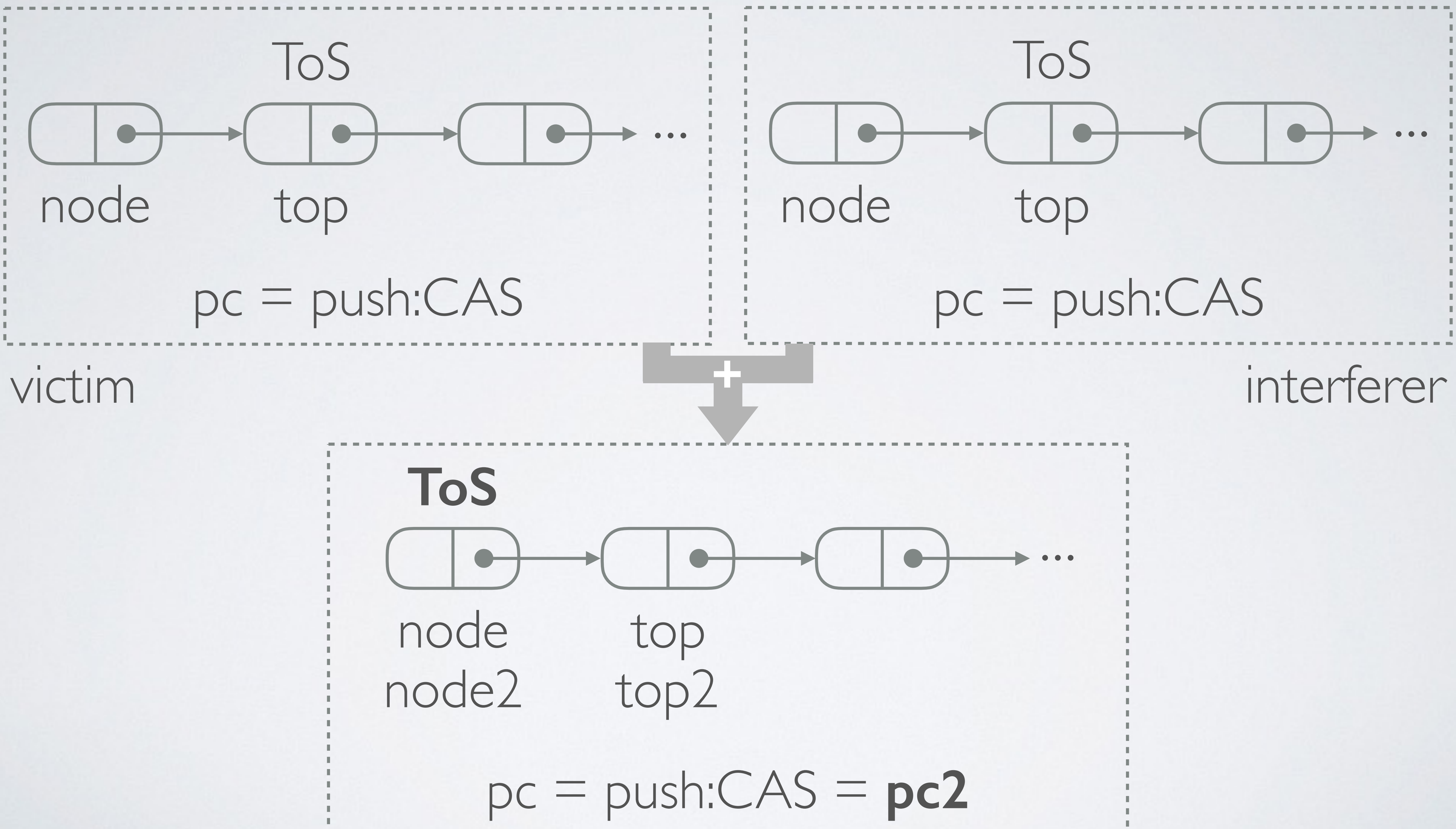
Example


T1.pc = pop:CAS
T2.pc = push:CAS
T3.pc = push:CAS
T2:node
ToS
...
T3:node
T1:top
T2:top
T3:top
T1:next
T1
T2
T3
ToS
...
top
next
pc = pop:CAS
ToS
...
node
top
pc = push:CAS

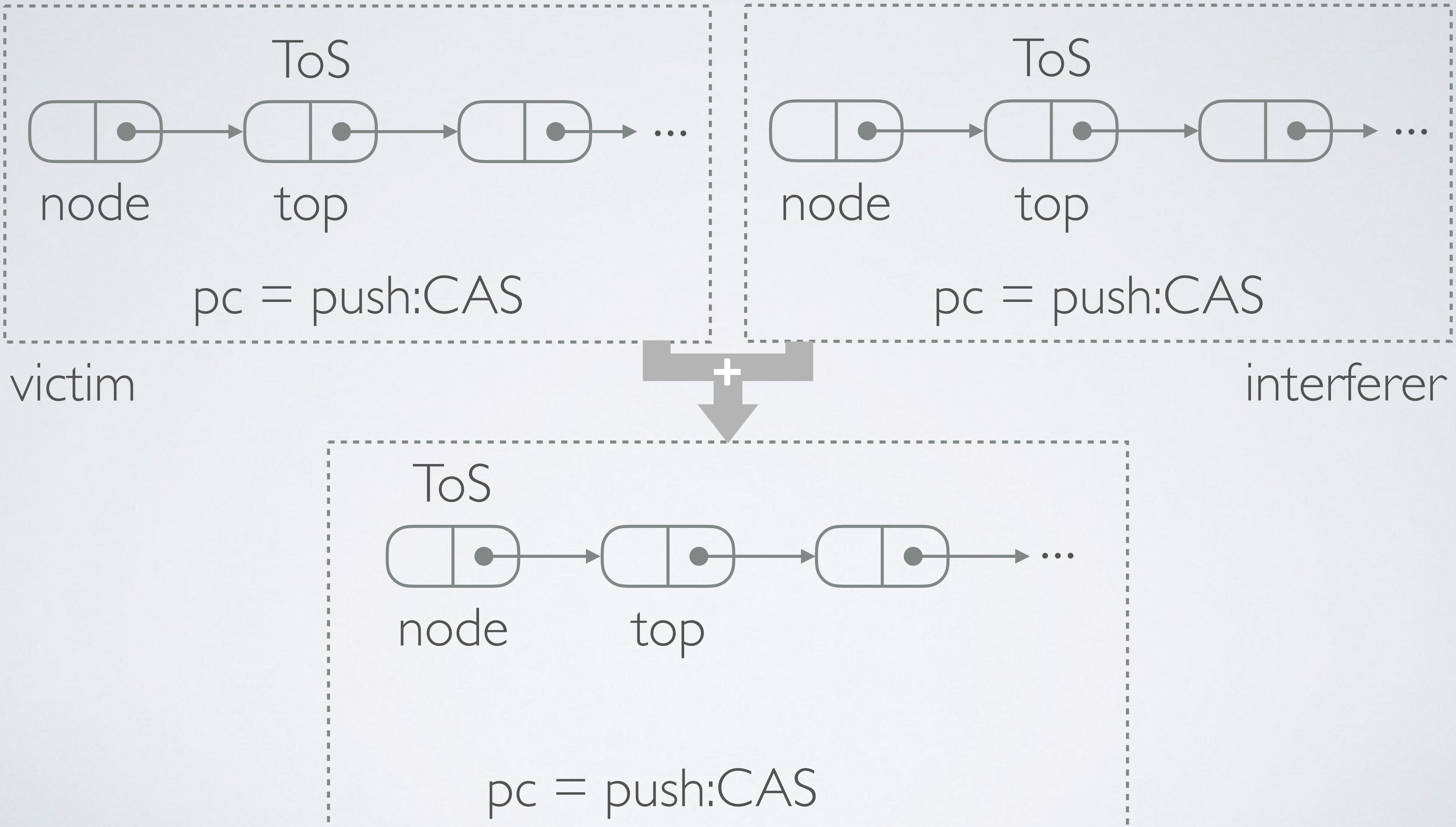
Interference



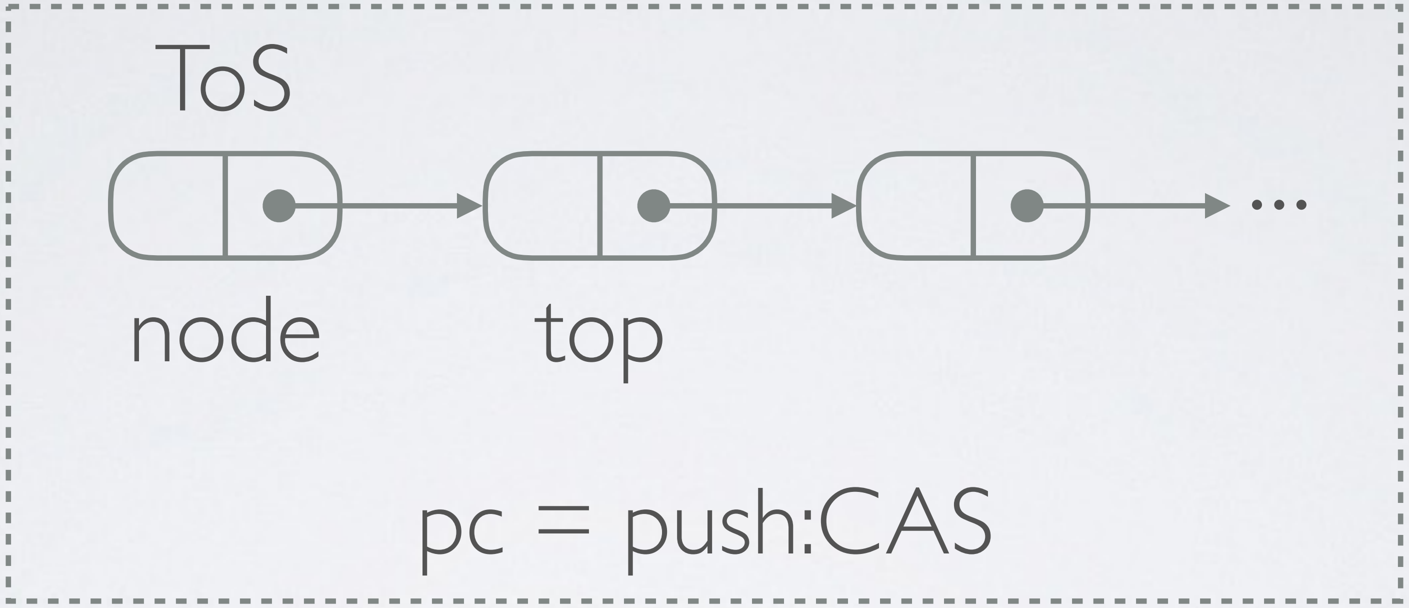
Interference



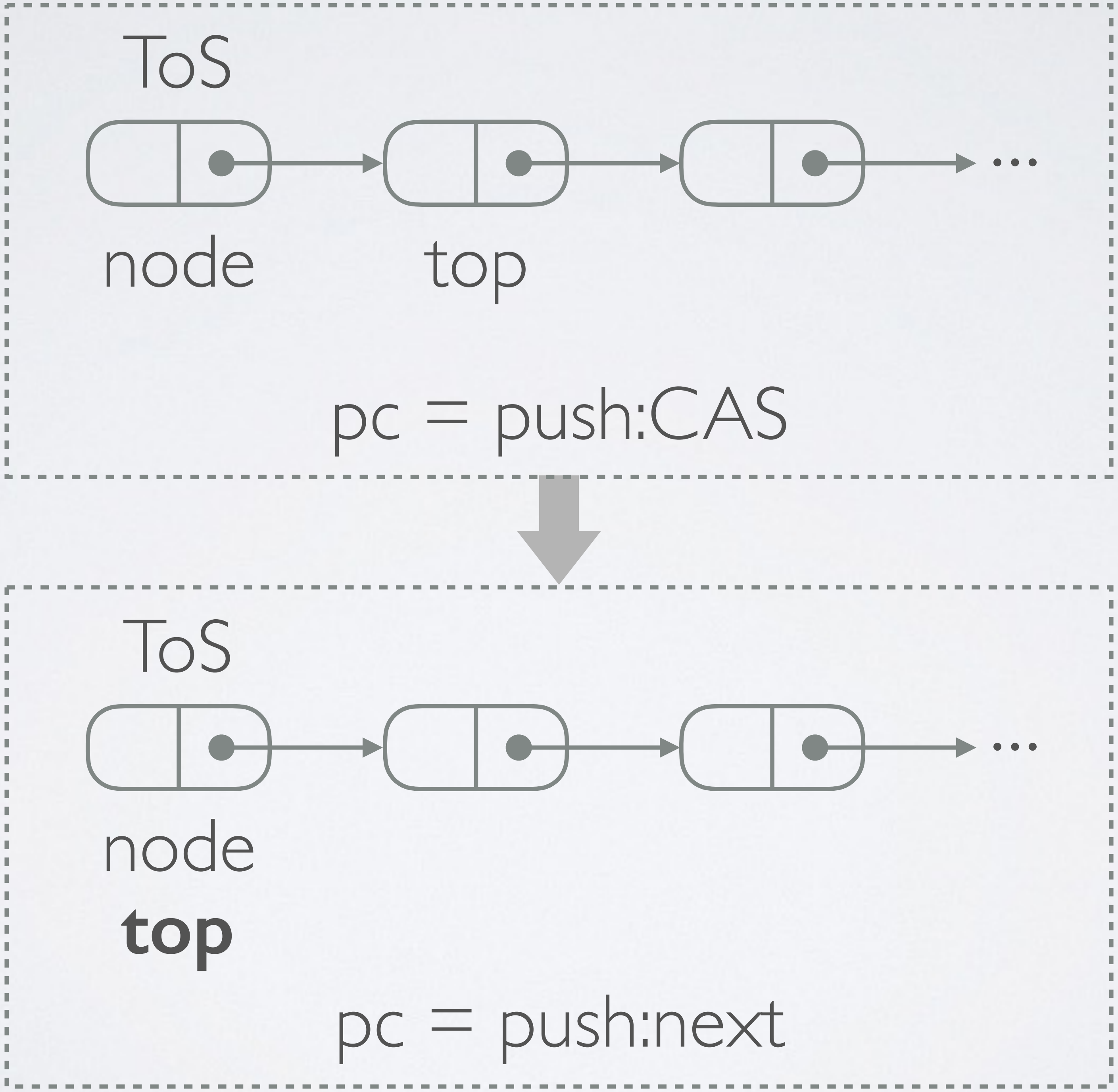
Interference



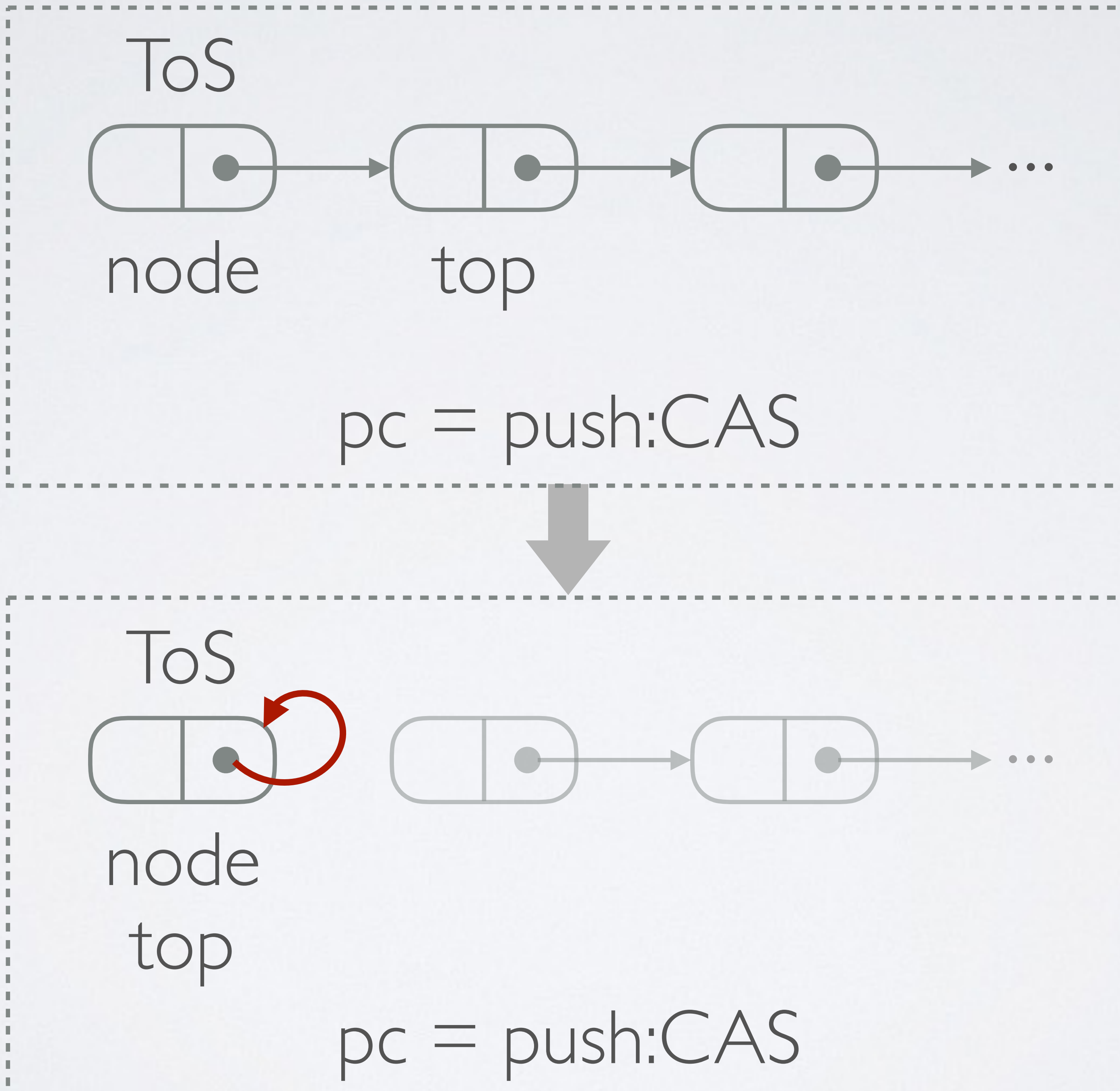
Sequential Steps



Sequential Steps



Sequential Steps



False Positives

- Problem: relation among *local* heaps is lost

  (due to thread modularity)

- Solution for GC: ownership

- Solution for MM: ??

  (key question)

False Positives

- Problem: relation among *local* heaps is lost

  (due to thread modularity)

- Solution for GC: ownership

- Solution for MM: **ownership!**

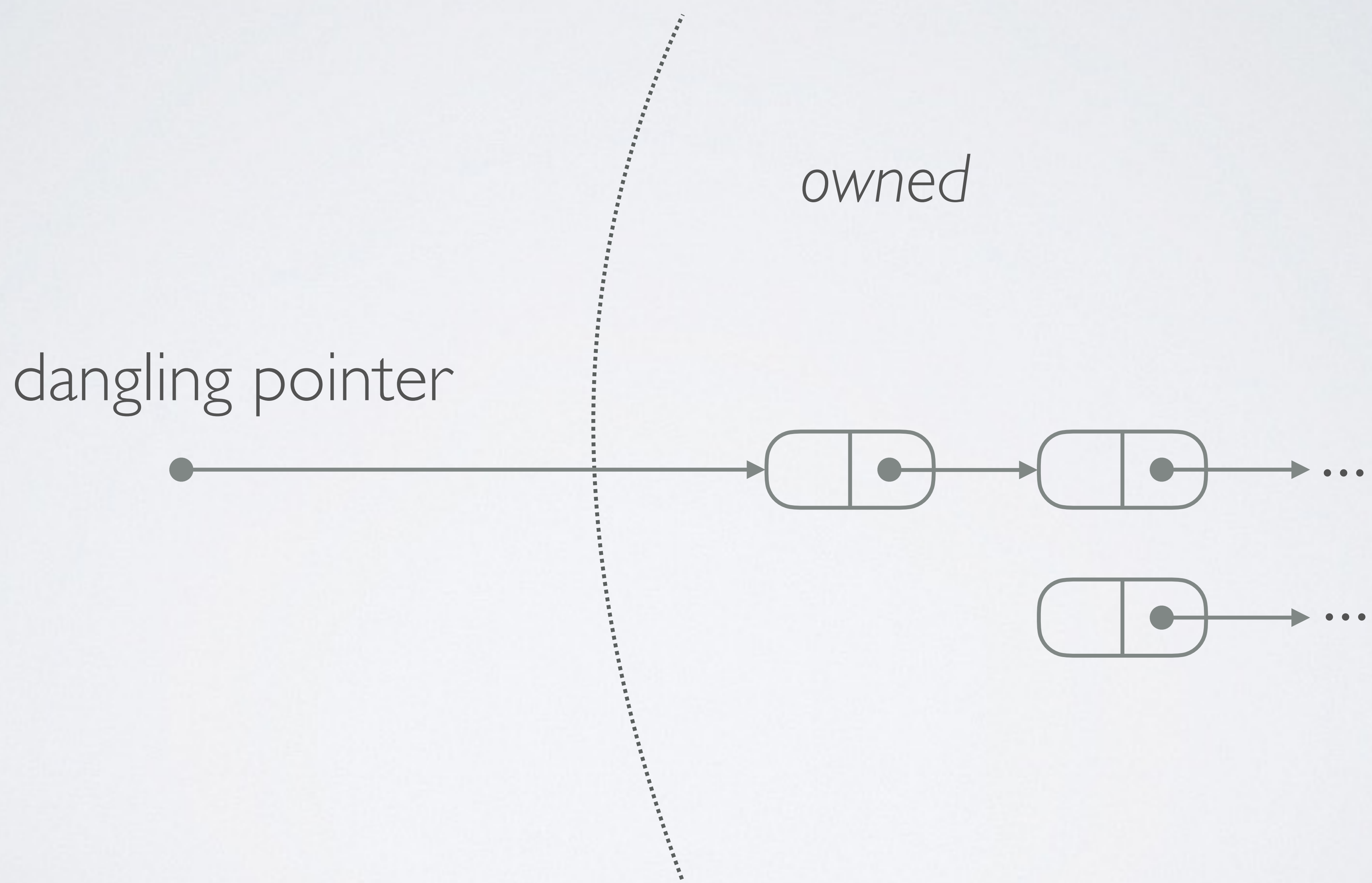
  (key question)

Ownership for MM

- *weak* ownership
  - granted upon allocation
  - removed upon publishing
  - dangling readers allowed

➡ write privilege

Ownership for MM



Pointer Races Freedom

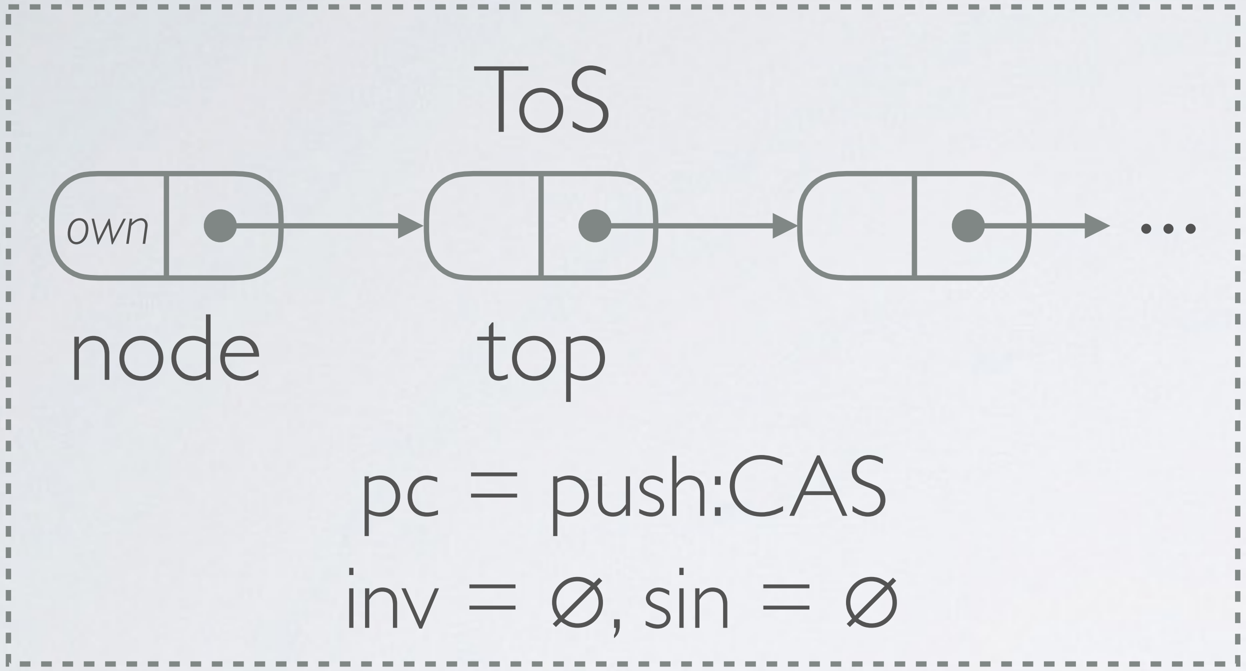
- establishes weak ownership for MM
- Pointer Race
  - writing through dangling pointers (invalid pointers)
  - following dangling pointers (strongly invalid pointers)

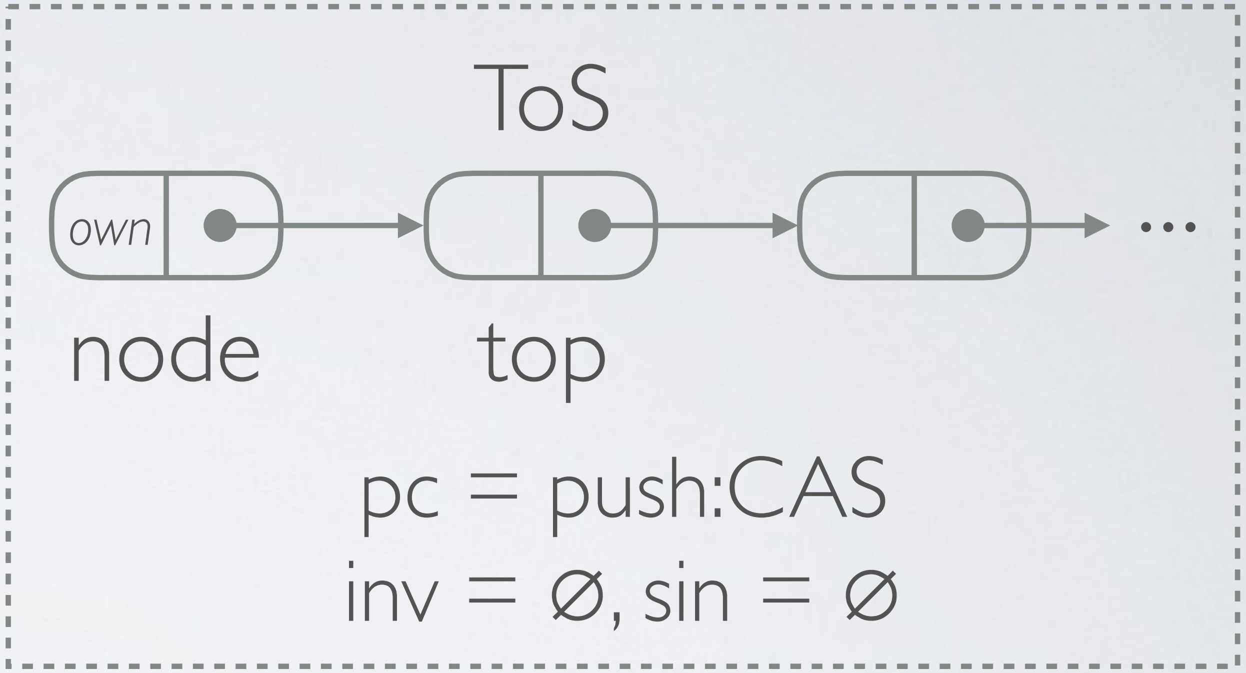
How to use PRF

- add validity and ownership information to views
- improve interference precision

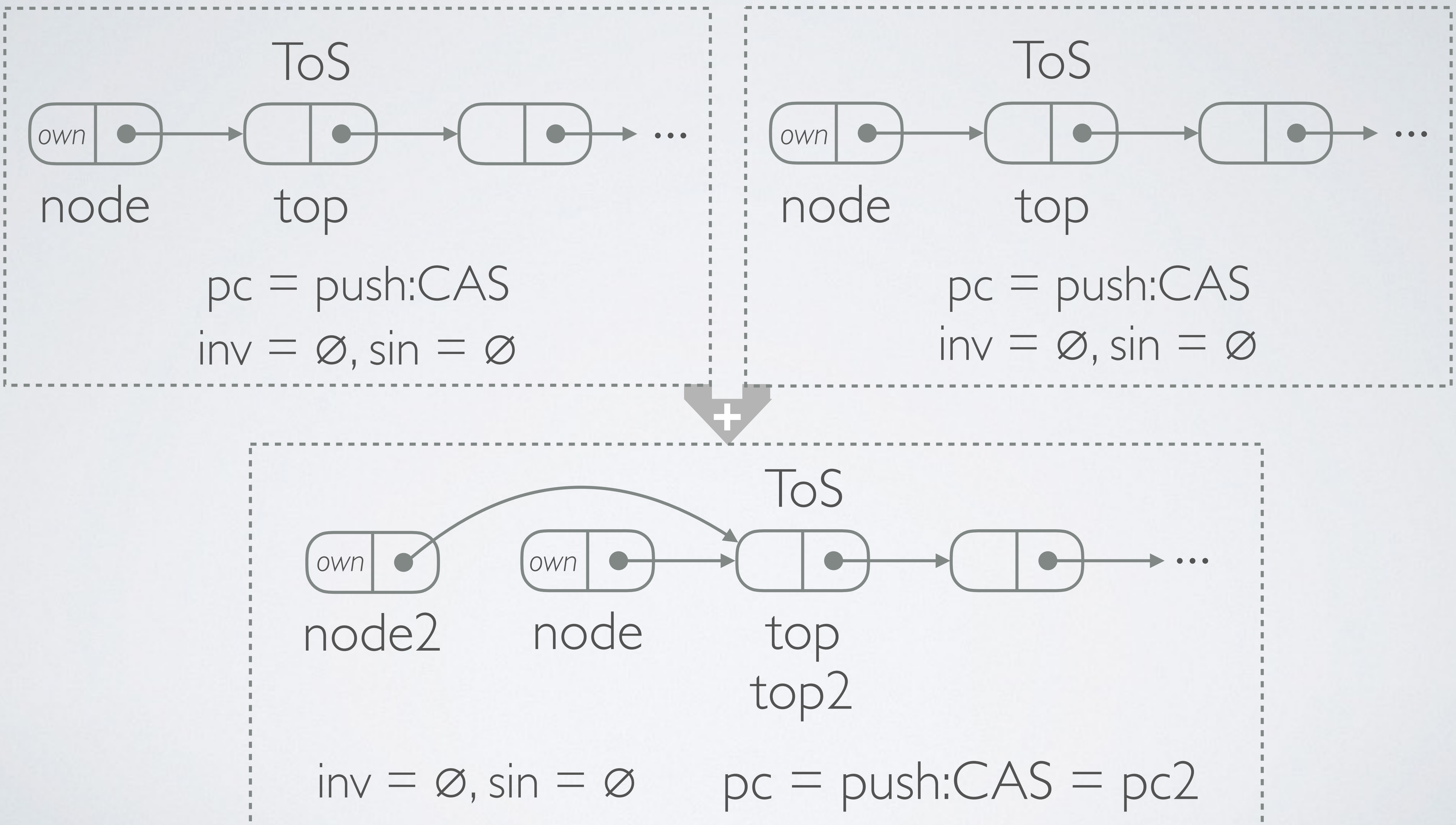
➡ dangling pointers:
- are allowed
- but invalid

Example

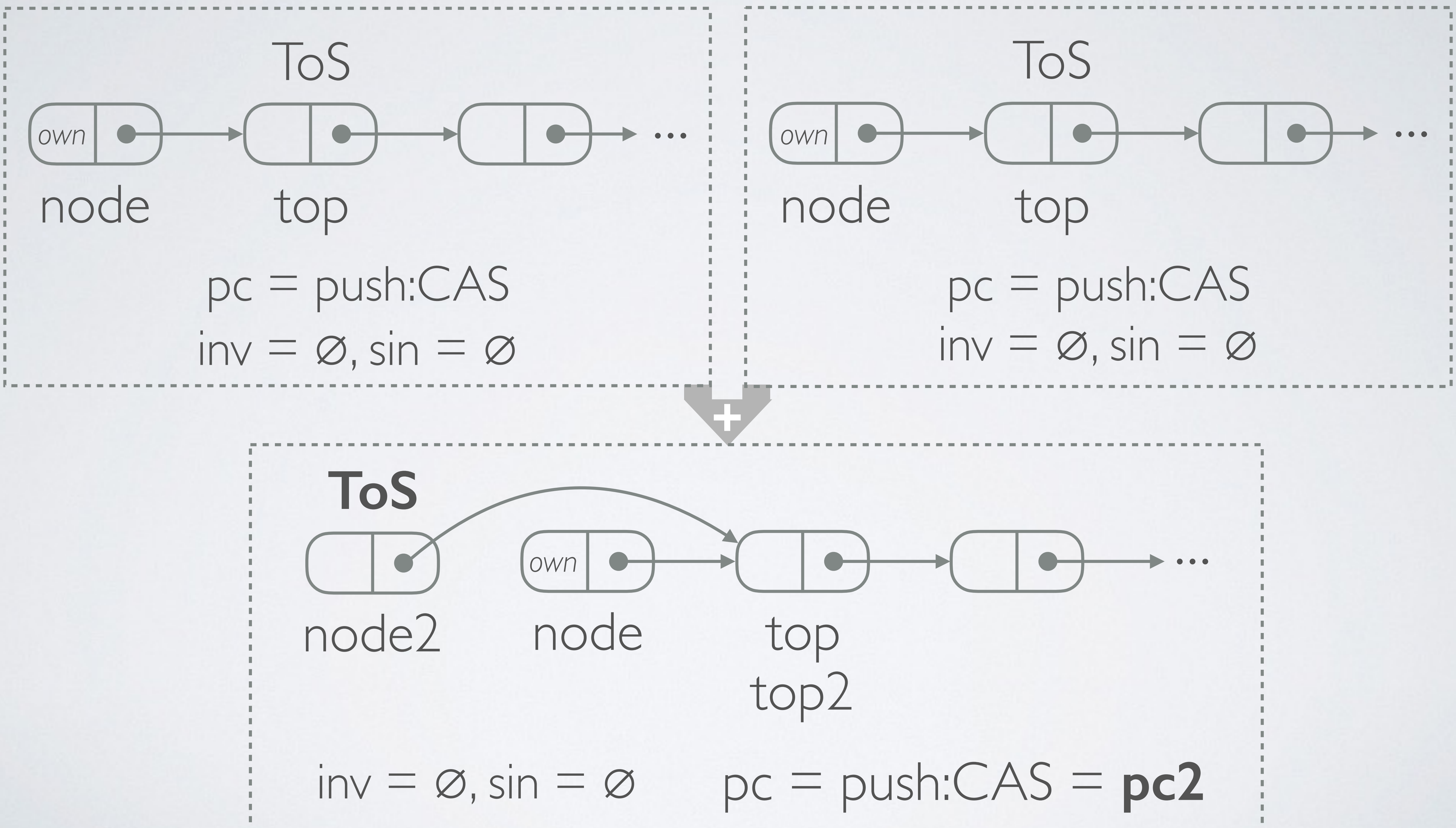

ToS
own
...
node
top
pc = push:CAS
inv = ∅, sin = ∅


ToS
own
...
node
top
pc = push:CAS
inv = ∅, sin = ∅

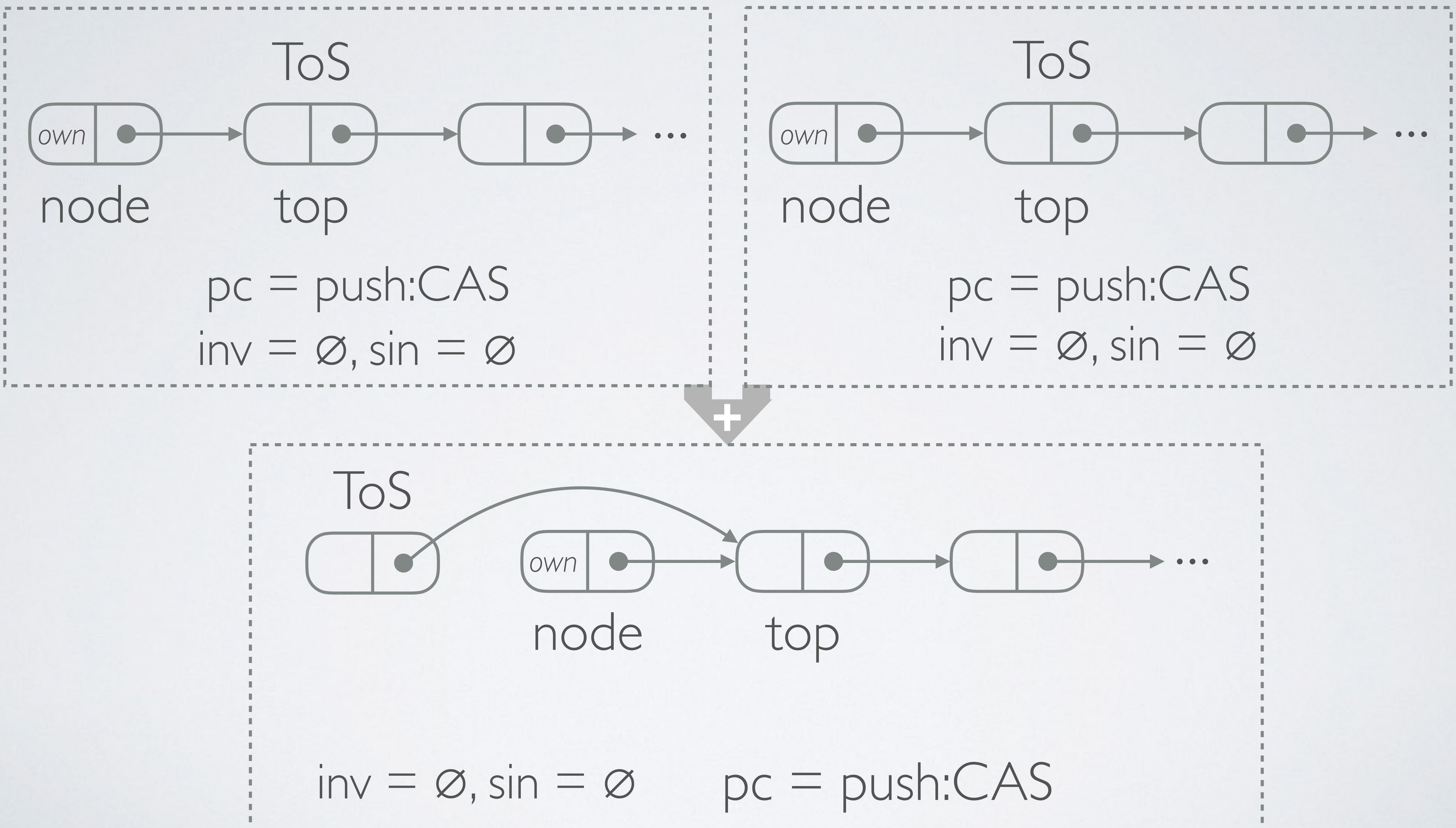
Example


ToS
own
node
top
pc = push:CAS
inv = ∅, sin = ∅
ToS
own
node
top
pc = push:CAS
inv = ∅, sin = ∅
+
ToS
own
own
node2
node
top
top2
inv = ∅, sin = ∅
pc = push:CAS = pc2

Example


ToS
own
node
top
...
pc = push:CAS
inv = ∅, sin = ∅
ToS
own
node
top
...
pc = push:CAS
inv = ∅, sin = ∅
+
ToS
own
node2
node
top
top2
...
inv = ∅, sin = ∅
pc = push:CAS = **pc2**

Example



ToS
own
node
top
pc = push:CAS
inv = ∅, sin = ∅
ToS
own
node
top
pc = push:CAS
inv = ∅, sin = ∅
+
ToS
own
node
top
inv = ∅, sin = ∅
pc = push:CAS

Prototype

- specification via finite observers (data independence)
- thread-modular framework
- shape analysis (heap abstraction)
- supports GC, MM, PRF
- checks linearisability
- ~5000 lines of code (C++)

Evaluation: Treiber's stack

	MM		PRF
runtime in seconds:	612	↓ :260	2.37
state space:	116776	↓ :150	744
sequential steps:	322328	↓ :120	2656
interference steps:	7913705	↓ :180	45815

— FIN —