Concurrent and parallel programming

Romolo Marotta

Correctness conditions
Progress conditions
Performance

Correctness conditions (incomplete) taxonomy

	Sequential Consistency	Linearizability	Serializability	Strict Serializability

Progress taxonomy

	Independent	**Dependent**	
	Non-blocking		**Blocking**
For everyone	Wait freedom	Obstruction freedom	Starvation freedom
For someone	Lock freedom		Deadlock freedom

- The Einsteinium of progress conditions: it does not exist in nature and (maybe) has no “commercial” value
- Clash freedom is a strictly weaker property than obstruction freedom

Speed-up according to Sun Ni

$$S_{Sun-Ni} = \frac{\alpha + (1-\alpha)G(p)}{\alpha + (1-\alpha)\frac{G(p)}{p}}$$

- If $G(p) = 1$

$$S_{Amdahl} = \frac{1}{\alpha + \frac{(1-\alpha)}{p}}$$

- If $G(p) = p$

$$S_{Gustafson} = \alpha + (1-\alpha)p$$

- In general $G(p) > p$ gives a higher scale-up

Concurrent Data Structures

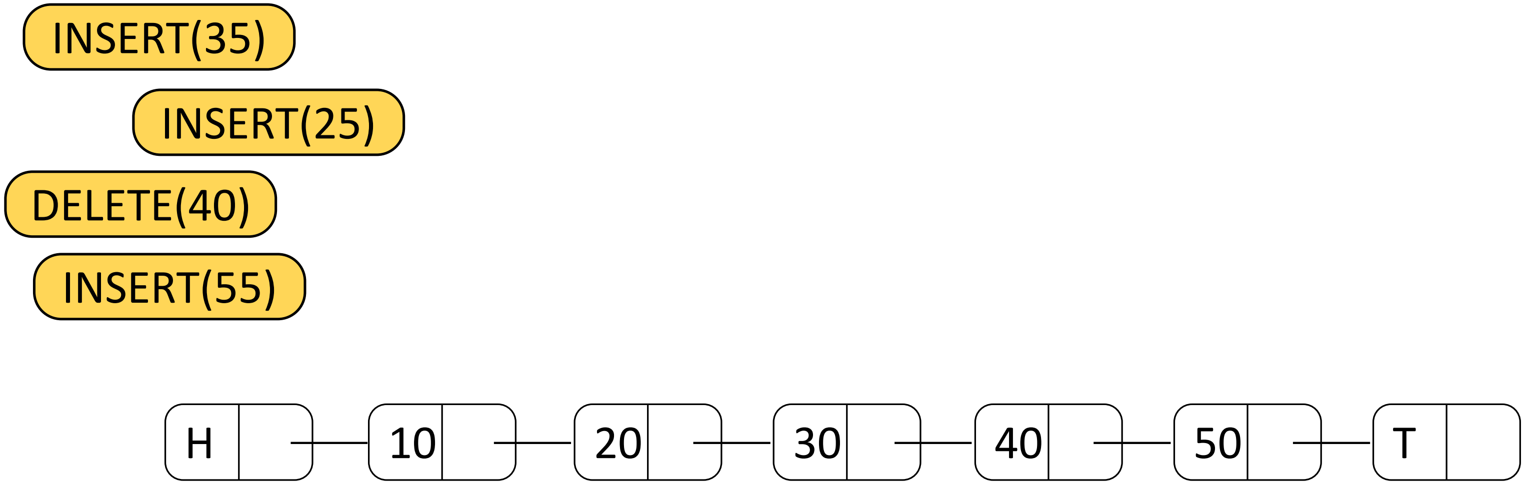
Concurrent Data Structures: sets

Concurrent data structures

- Developing data structures which can be concurrently accessed by multiple threads can significantly increase performance
- Result’s correctness must be guaranteed (recall linearizability)

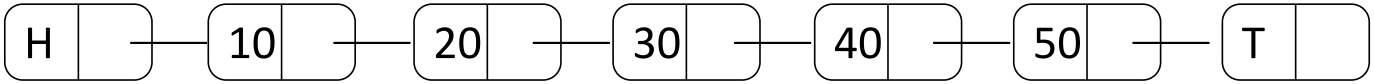
Set implementations

- Set methods:
    - `insert(k)`
    - `delete(k)`
    - ~~`find(k)`~~
- Implemented as an ordered linked list

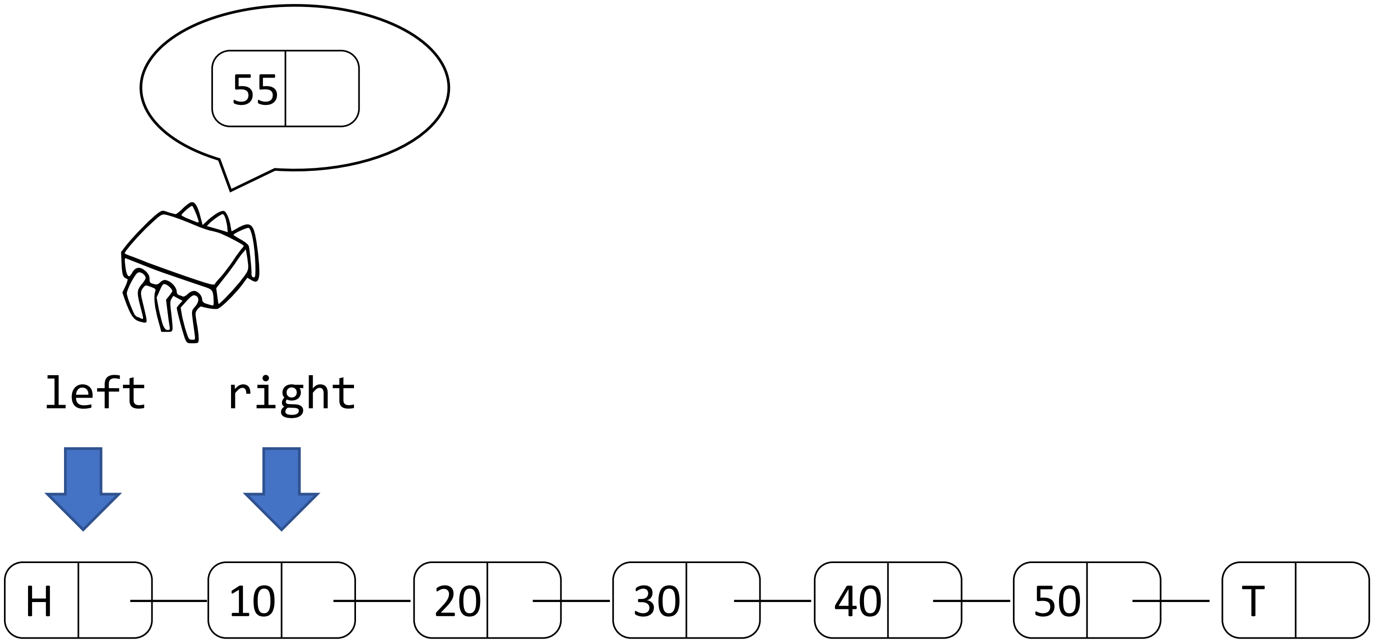


Insert algorithm

INSERT(55)



Insert algorithm

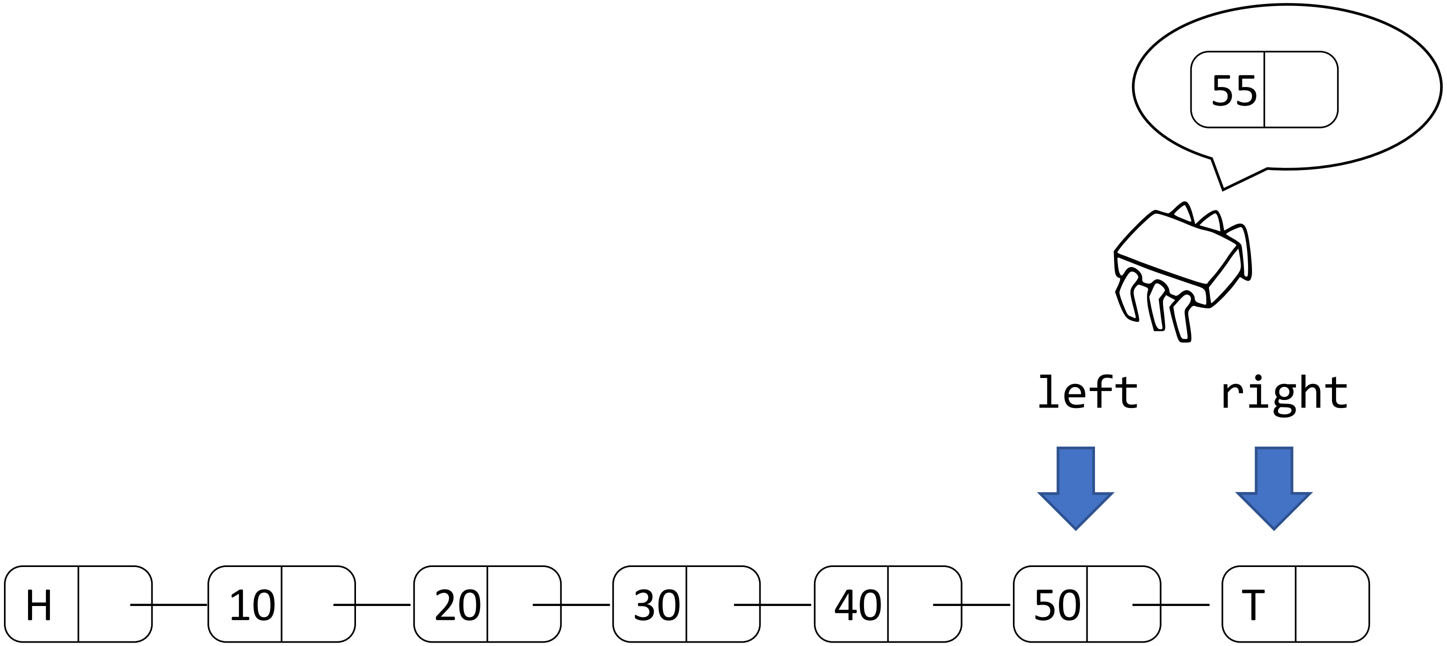


55

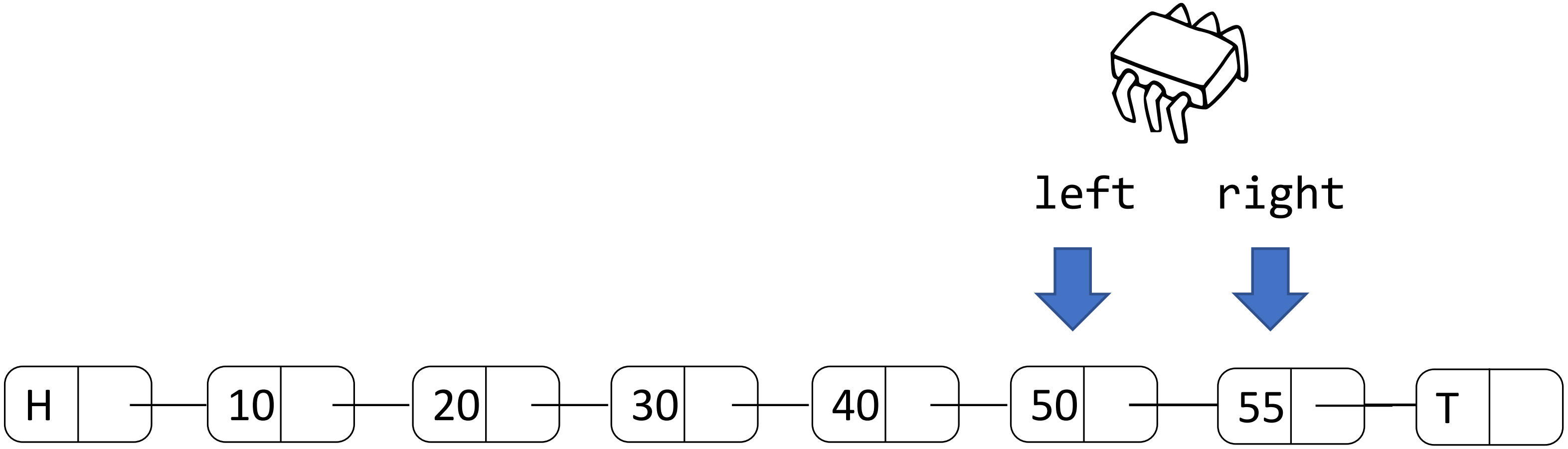
left right

H 10 20 30 40 50 T

Insert algorithm



Insert algorithm

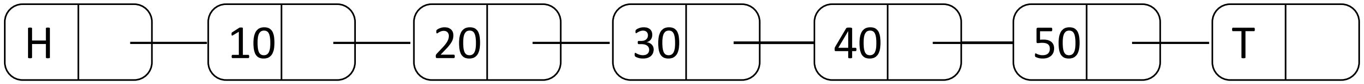


`left` `right`

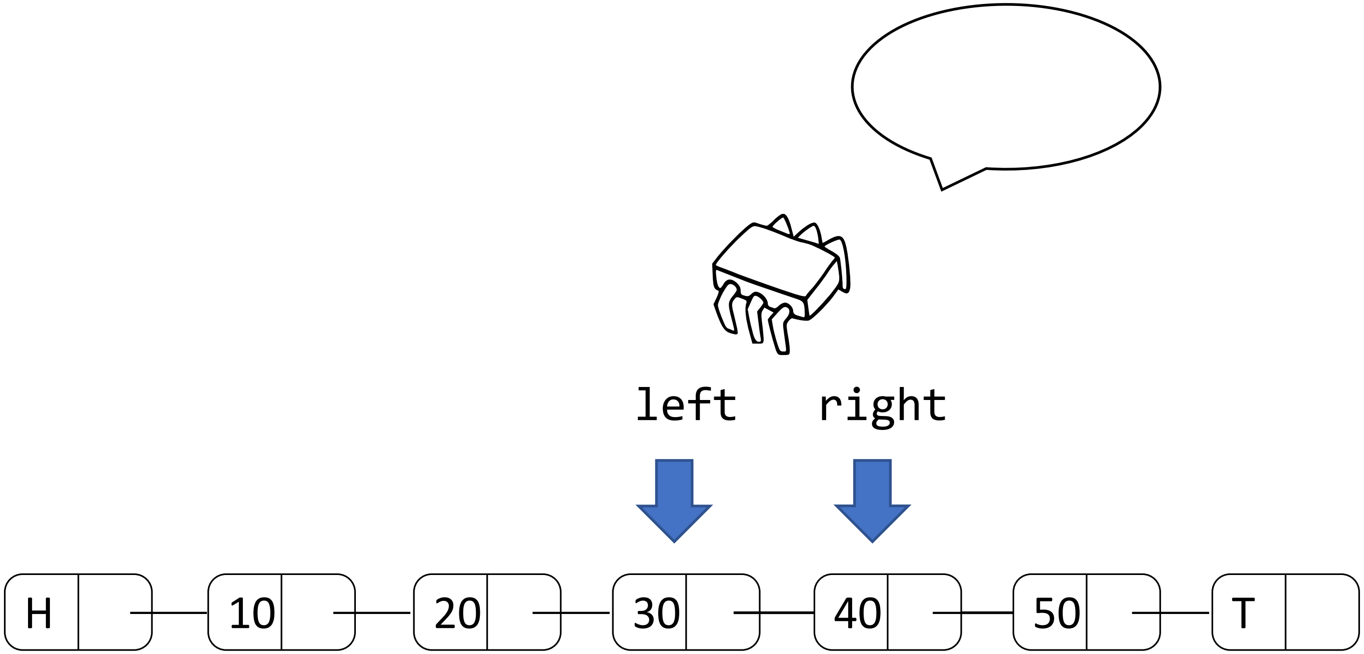
H 10 20 30 40 50 55 T

Delete algorithm

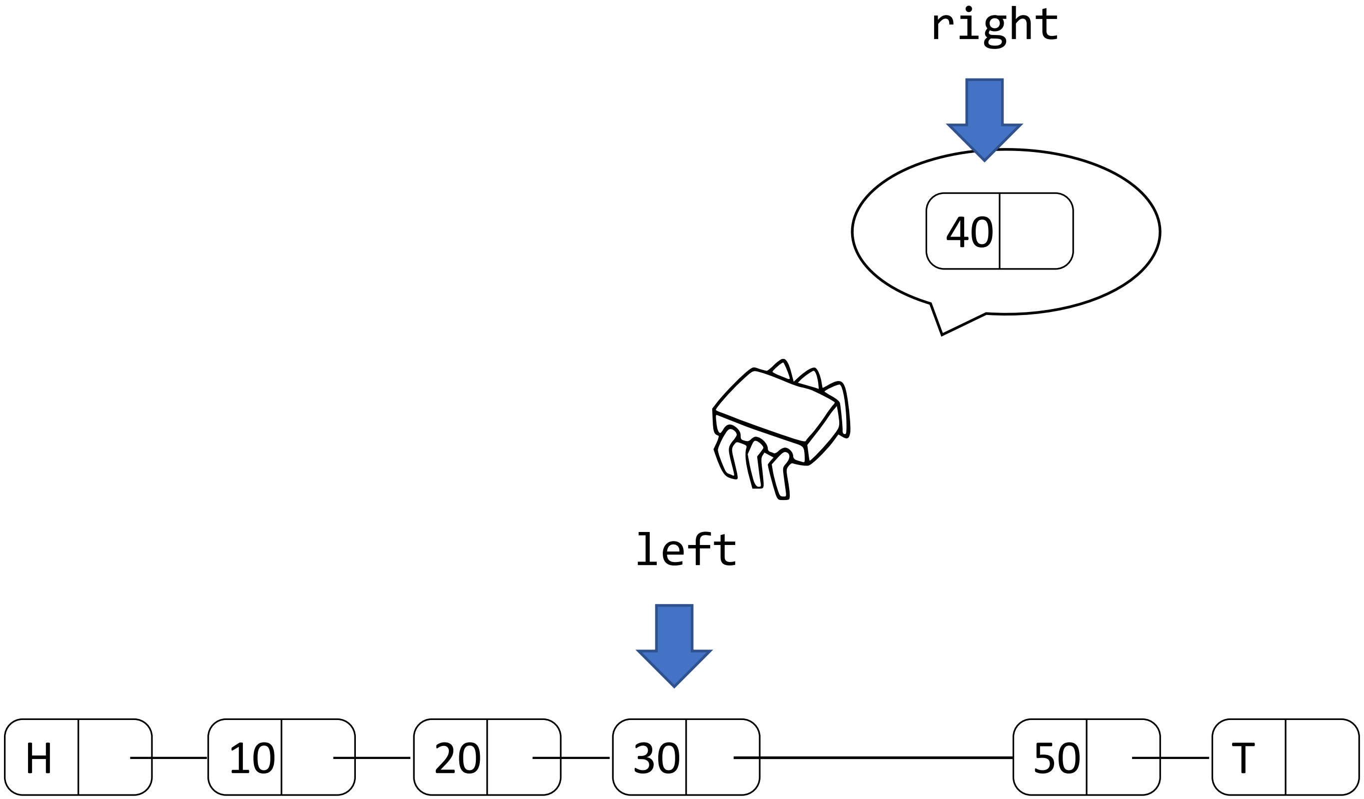
DELETE(40)



Delete algorithm



Delete algorithm



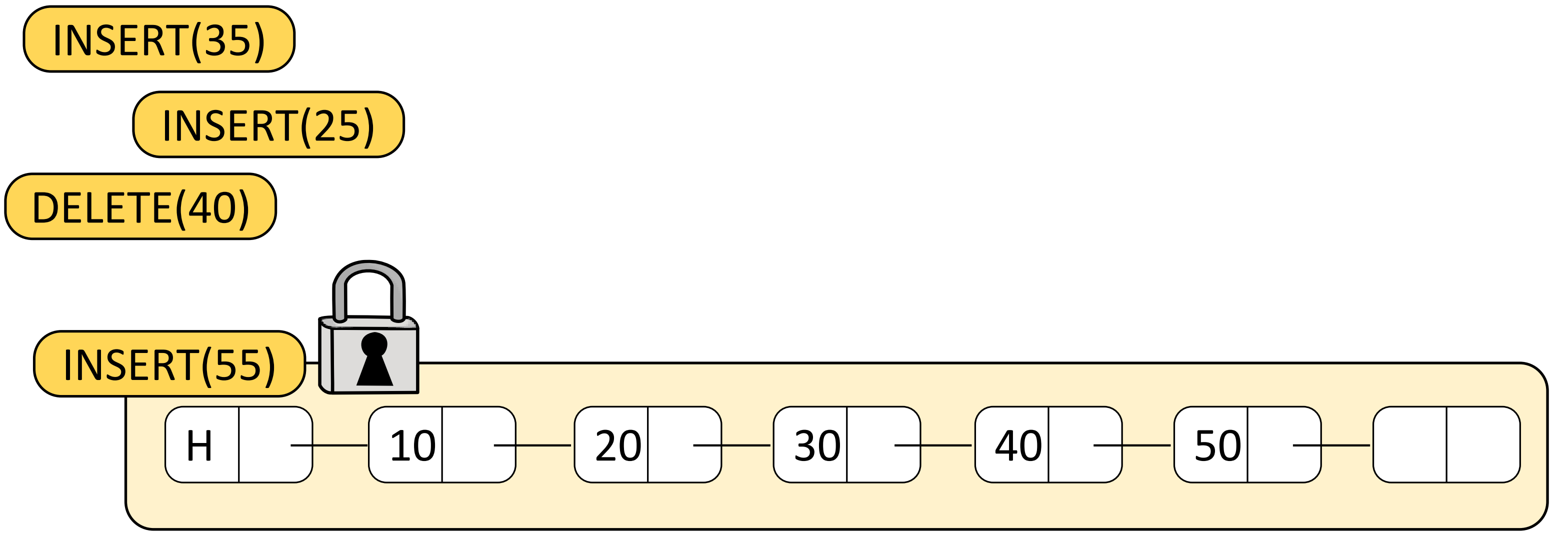
Sequential set implementation

```
bool do_operation(int k, int op_type){
  bool res = true;
  node *l,*r;

  l = search(k, &r);
  switch(op_type){
    case(INSERT):
      if(r->key == k)
        res = false;
      else
        l->next = new node(k,r);
      break;
    case(DELETE):
      if(r->key == k)
        l->next = r->next;
      else
        res = false;
      break;
  }


  return res;
}
```

```
node* search(int k, node **r){
  node *l, *r_next;
  l = set->head;

  *r = l->next;

  r_next = (*r)->next;
  while((*r)->key < k){

    l = *r;
    *r = r_next;

    r_next = (*r)->next;
  }
}
```

Concurrent set – Attempt 1

- PESSIMISTIC approach
- Synchronize via global lock



INSERT(35)

INSERT(25)

DELETE(40)

INSERT(55)

H | 10 | 20 | 30 | 40 | 50

Concurrent set – Attempt 1 (SRC)

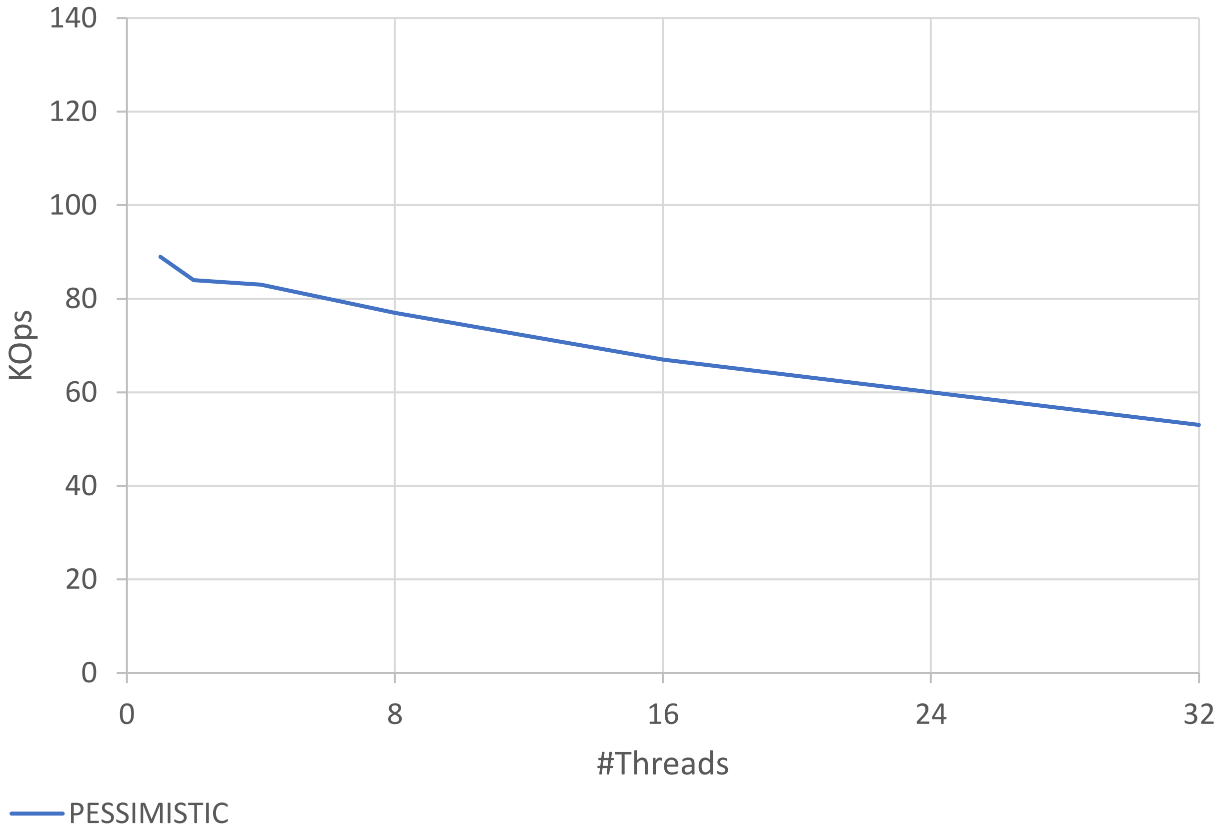
```
bool do_operation(int k, int op_type){
  bool res = true;
  node *l,*r;
  LOCK(&glock);
  l = search(k, &r);
  switch(op_type){
    case(INSERT):
      if(r->key == k)
        res = false;
      else
        l->next = new node(k,r);
      break;
    case(DELETE):
      if(r->key == k)
        l->next = r->next;
      else
        res = false;
      break;
  }
  UNLOCK(&glock);

  return res;
}
```

```
node* search(int k, node **r){
  node *l, *r_next;
  l = set->head;

  *r = l->next;

  r_next = (*r)->next;
  while((*r)->key < k){

    l = *r;
    *r = r_next;

    r_next = (*r)->next;
  }
}
```

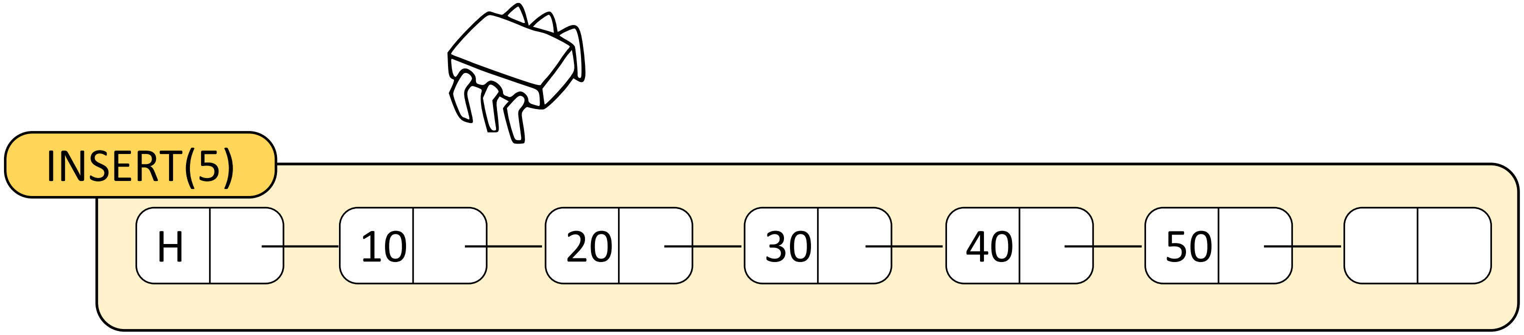
Concurrent set – Attempt 1

AMD Opteron 6128 – 32Cores

KeyRange = [0,6000] SetSize = 2400 Update=100%



Concurrent set – Attempt 1



INSERT(5)

H

10

20

30

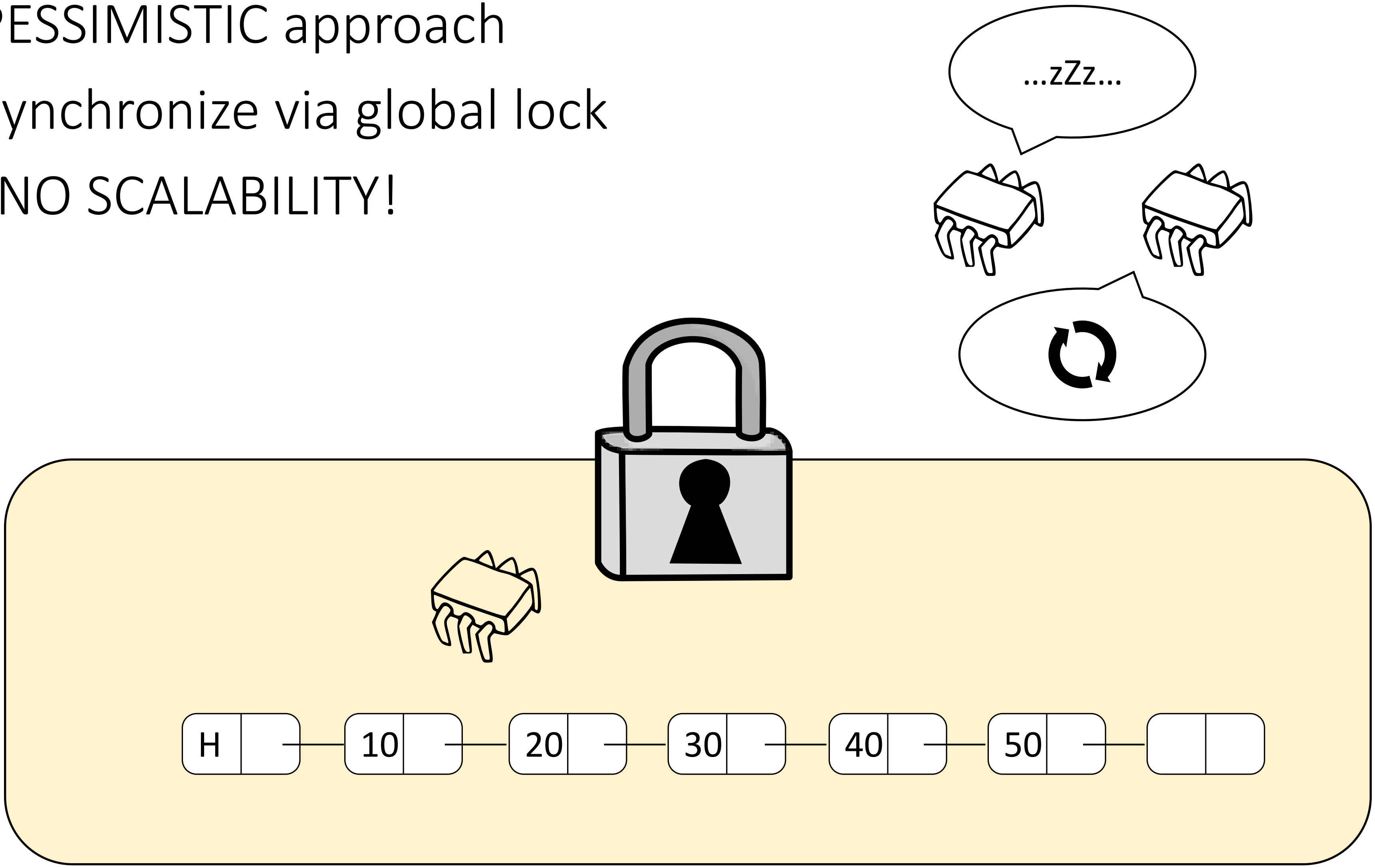
40

50

Concurrent set – Attempt 1

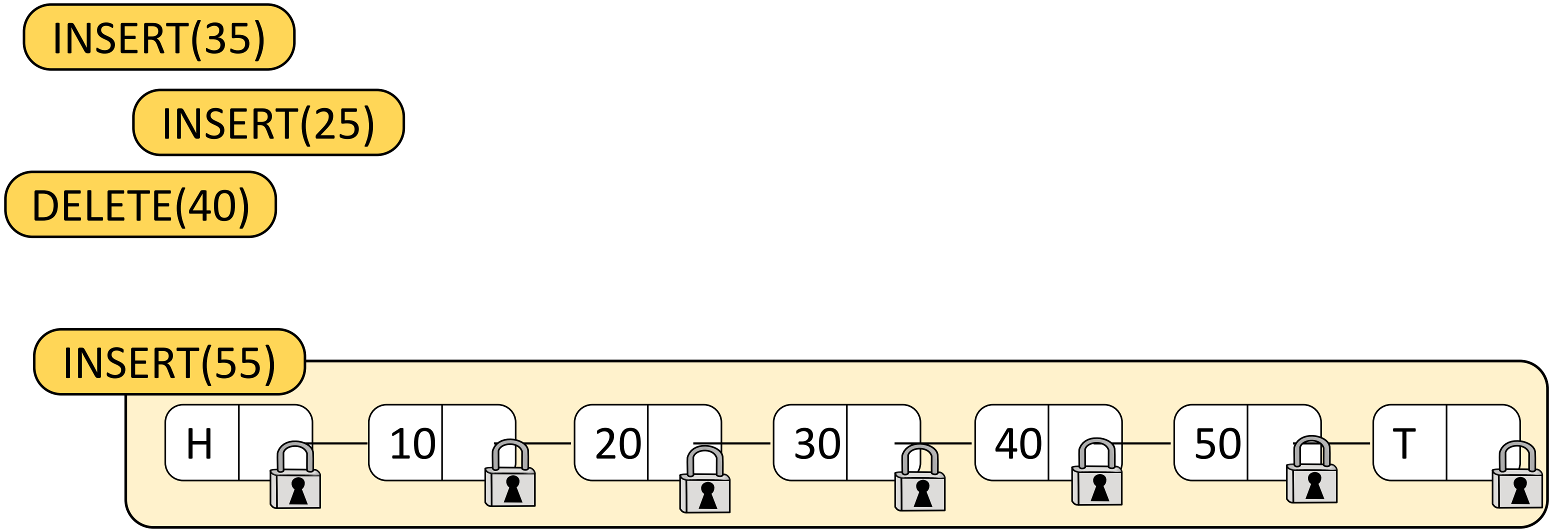
- PESSIMISTIC approach
- Synchronize via global lock

⇒NO SCALABILITY!



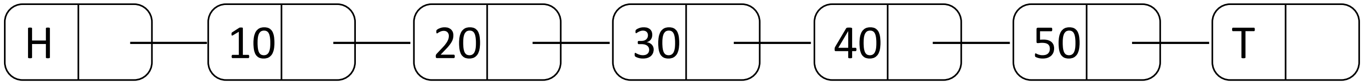
Concurrent set – Attempt 2

- Fine-grain approach
- Each node has its own lock
- Keep two locks at a time (lock coupling):
  - One on the current node
  - One on its predecessor



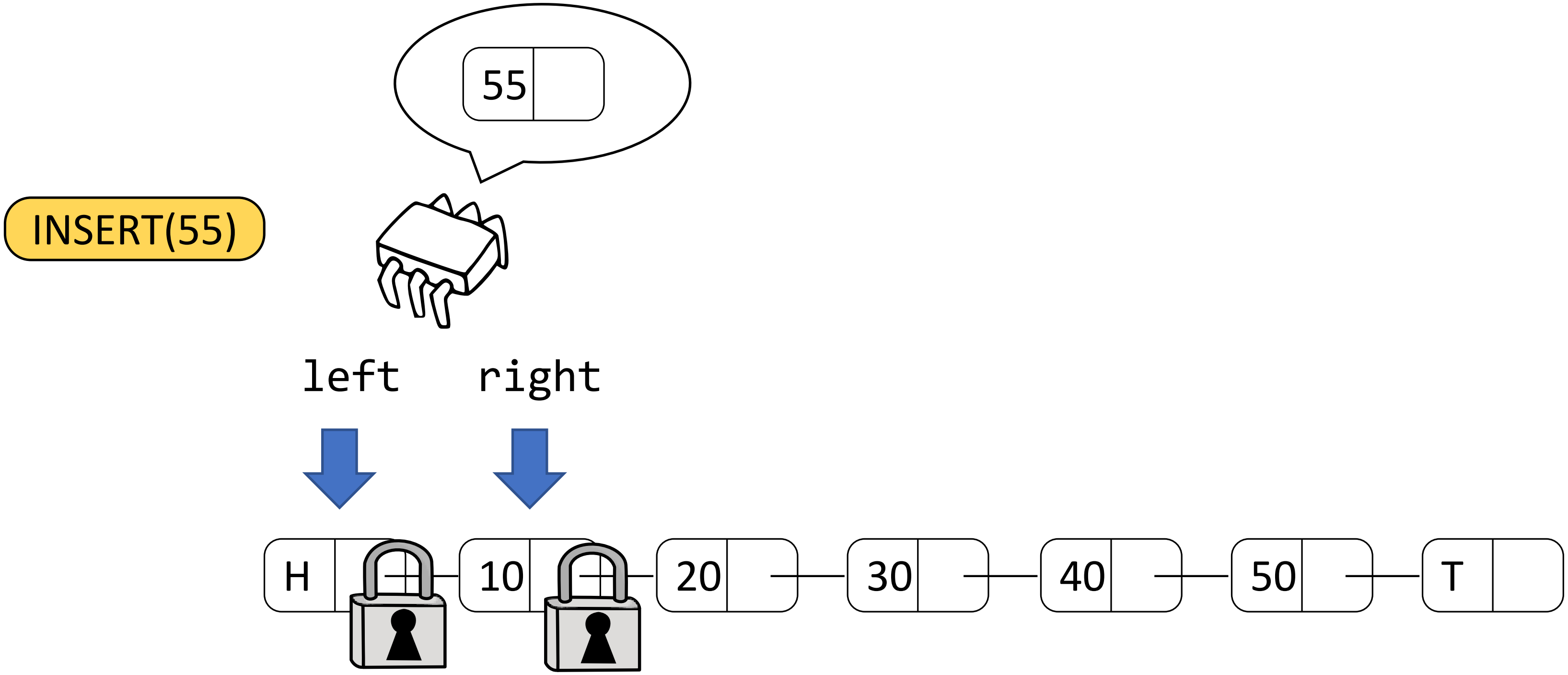
Search algorithm

INSERT(55)



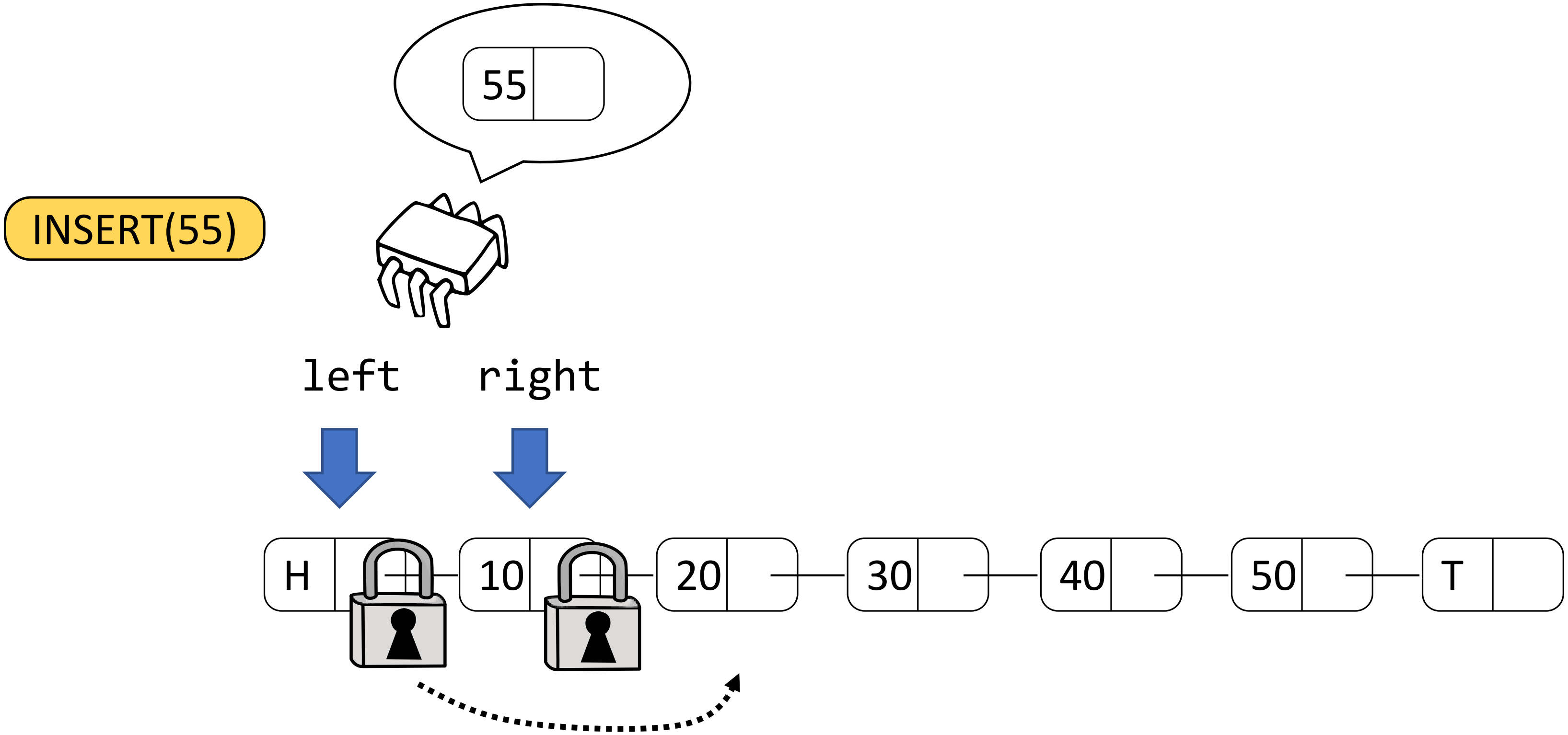
Search algorithm

- Keep two locks at a time (lock coupling):
  - One on the current node
  - One on its predecessor



Search algorithm

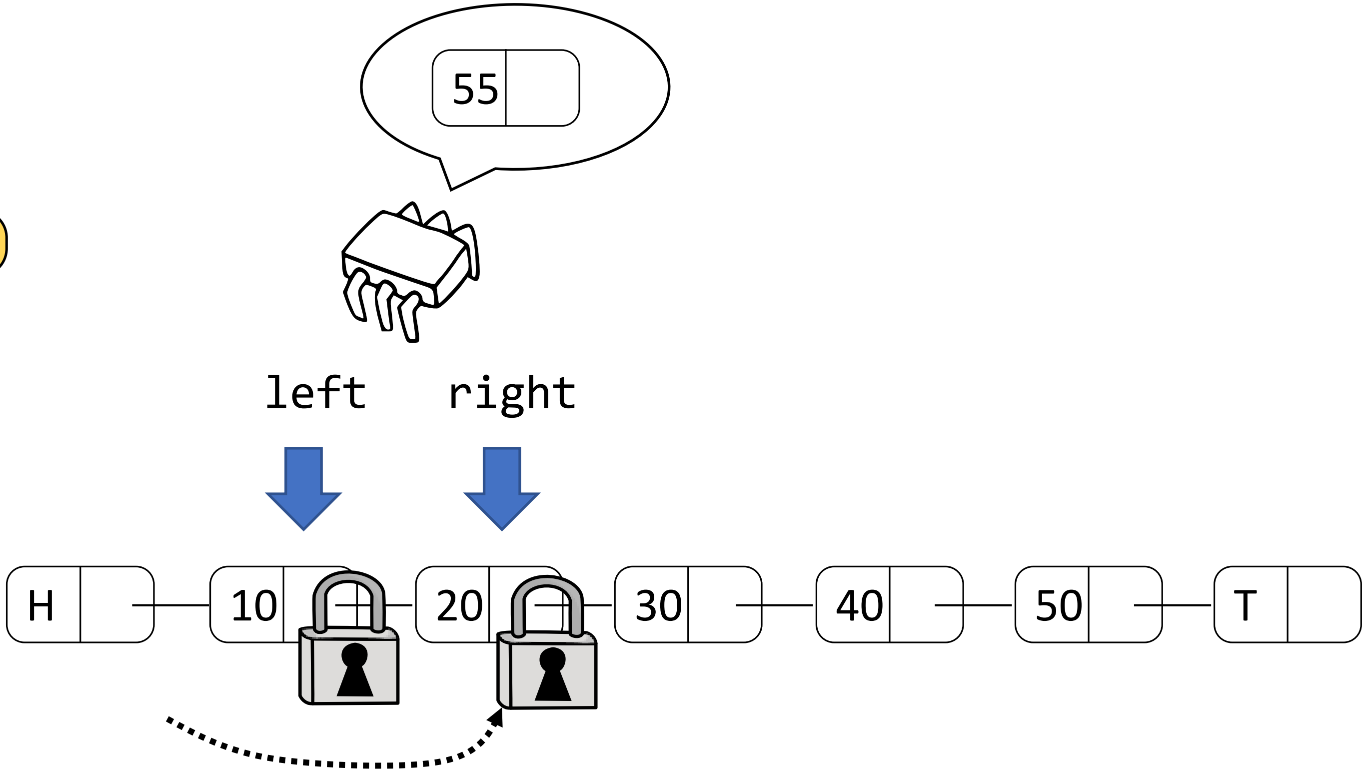
- Keep two locks at a time (lock coupling):
    - One on the current node
    - One on its predecessor



Search algorithm

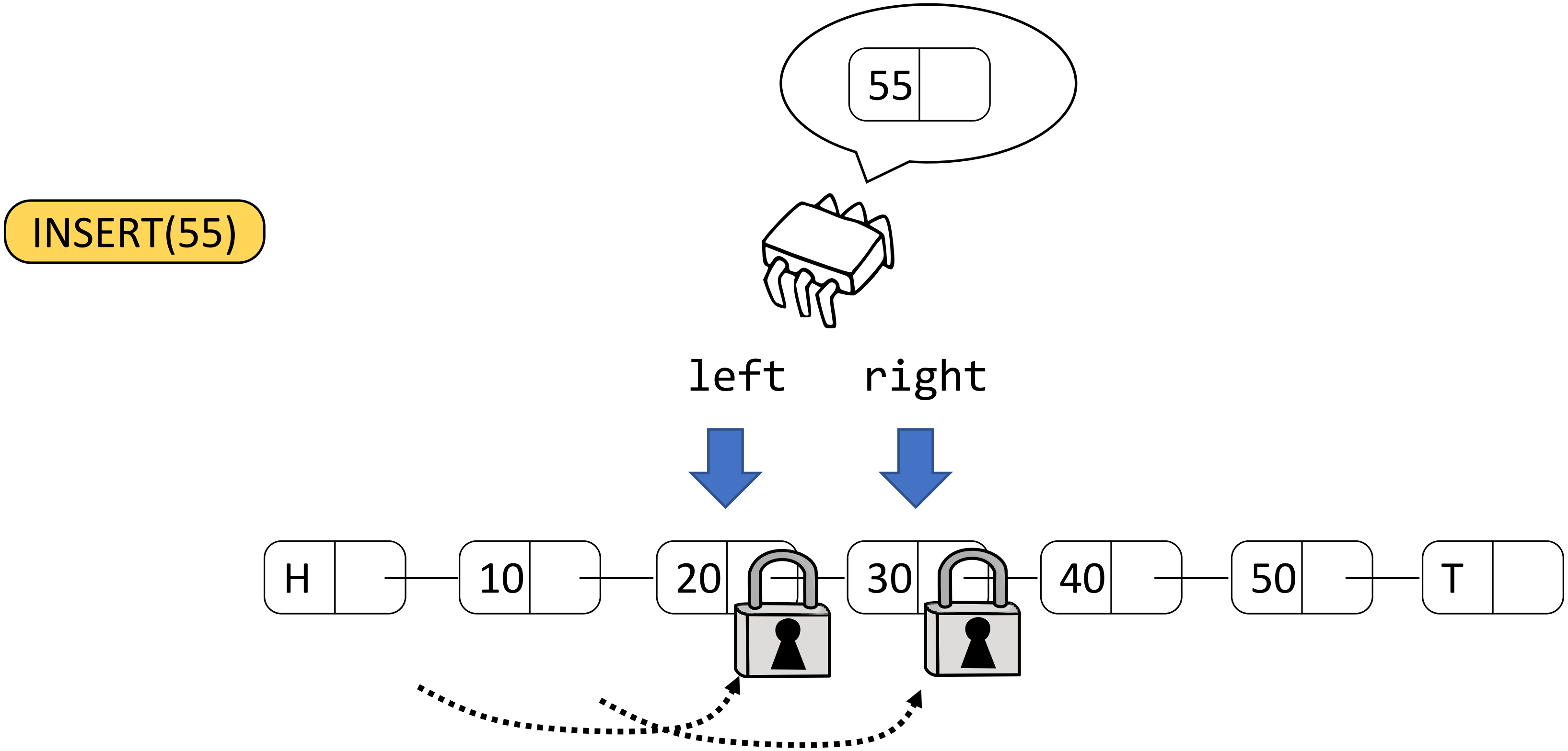
- Keep two locks at a time (lock coupling):
    - One on the current node
    - One on its predecessor

INSERT(55)



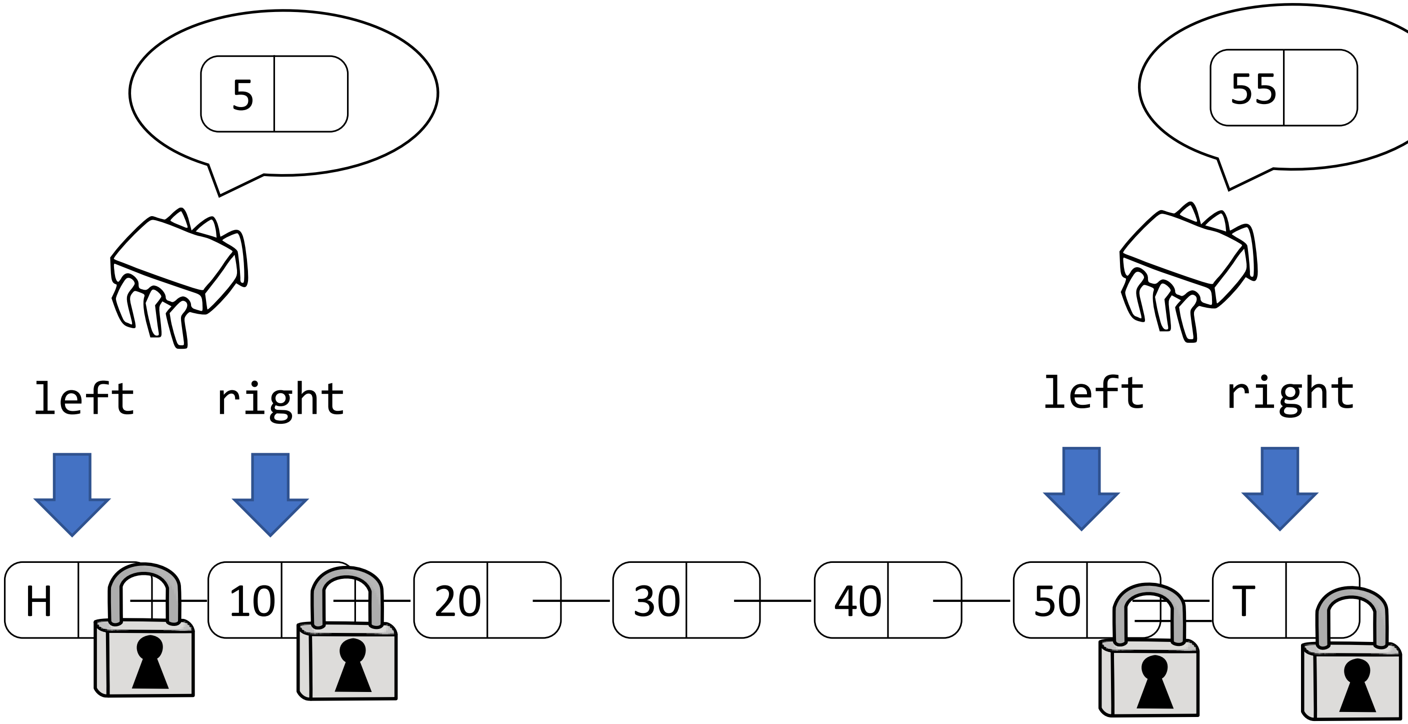
Search algorithm

- Keep two locks at a time (lock coupling):
  - One on the current node
  - One on its predecessor



Search algorithm

- Keep two locks at a time (lock coupling):
  - One on the current node
  - One on its predecessor
- Multiple threads access the data structure simultaneously



Concurrent set – Attempt 2 (SRC)

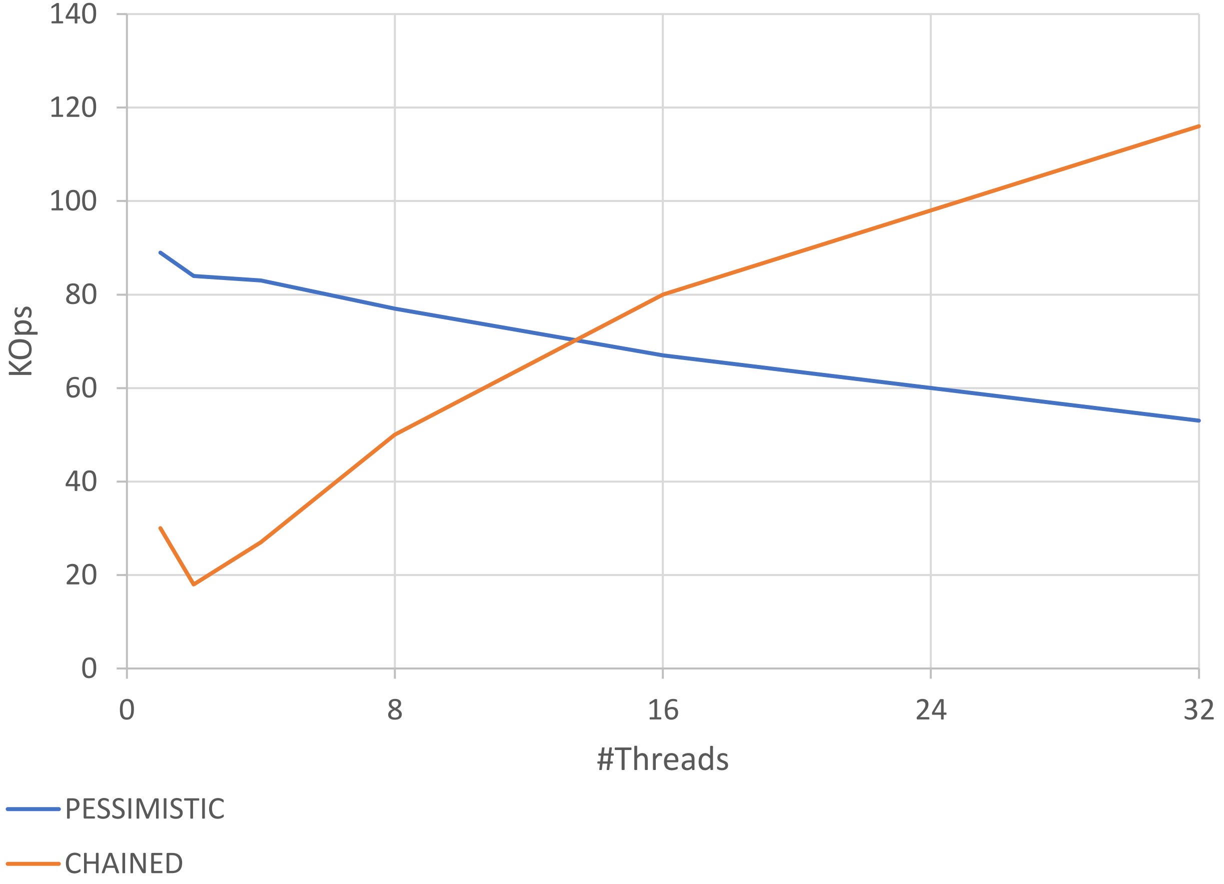
```
bool do_operation(int k, int op_type){
  bool res = true;
  node *l,*r;
  ~~LOCK(&glock);~~
  l = search(k, &r);
  switch(op_type){
    case(INSERT):
      if(r->key == k)
        res = false;
      else
        l->next = new node(k,r);
      break;
    case(DELETE):
      if(r->key == k)
        l->next = r->next;
      else
        res = false;
      break;
  }
  ~~UNLOCK(&glock);~~
  UNLOCK(&l->lock);
  UNLOCK(&r->lock);
 return res;
}
```

```
node* search(int k, node **r){
  node *l, *r_next;
  l = set->head;
  LOCK(&l->lock);
  *r = l->next;
  LOCK(&(*r)->lock);
  r_next = (*r)->next;
  while((*r)->key < k){
    UNLOCK(&l->lock);
    l = *r;
    *r = r_next;
    LOCK(&(*r)->lock);
    r_next = (*r)->next;
  }
}
```

Concurrent set – Attempt 2

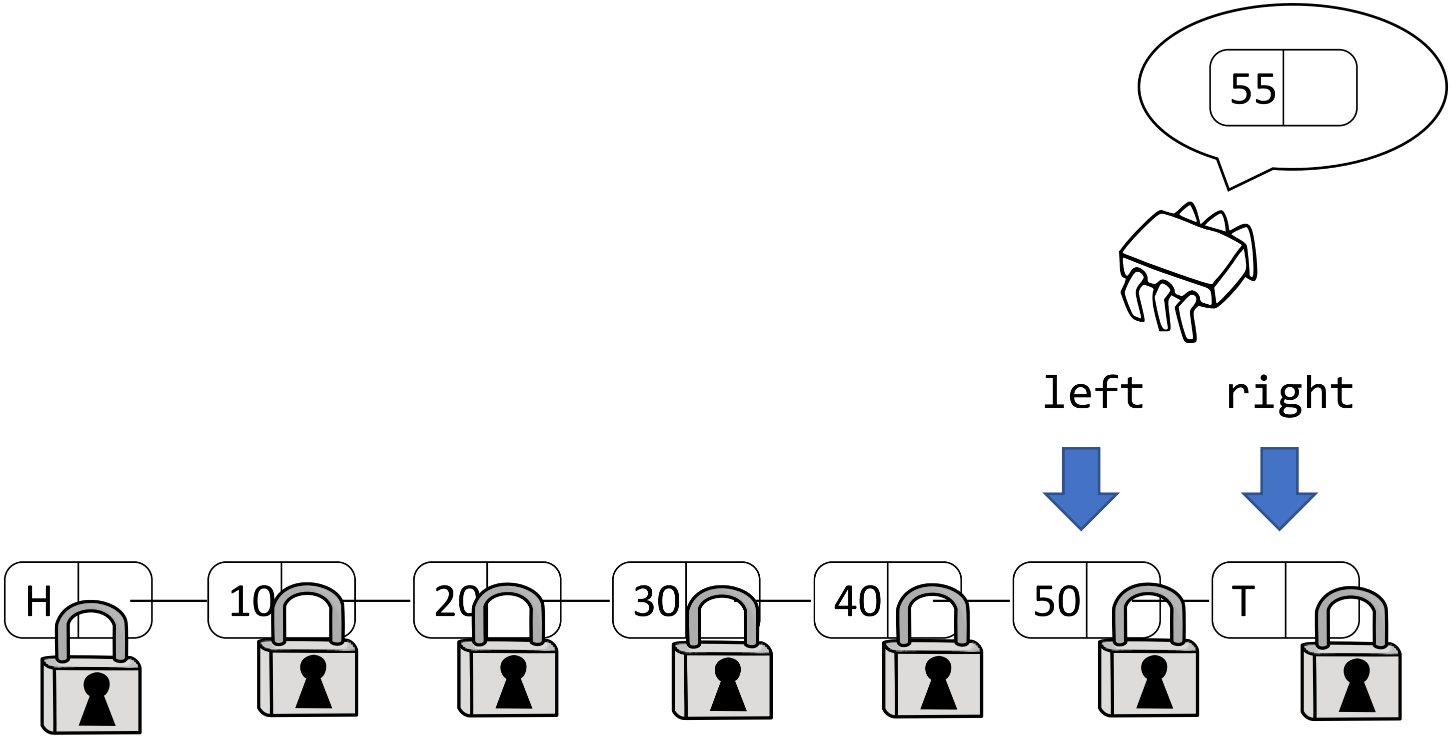
AMD Opteron 6128 – 32Cores

KeyRange = [0,6000]   SetSize = 2400   Update=100%



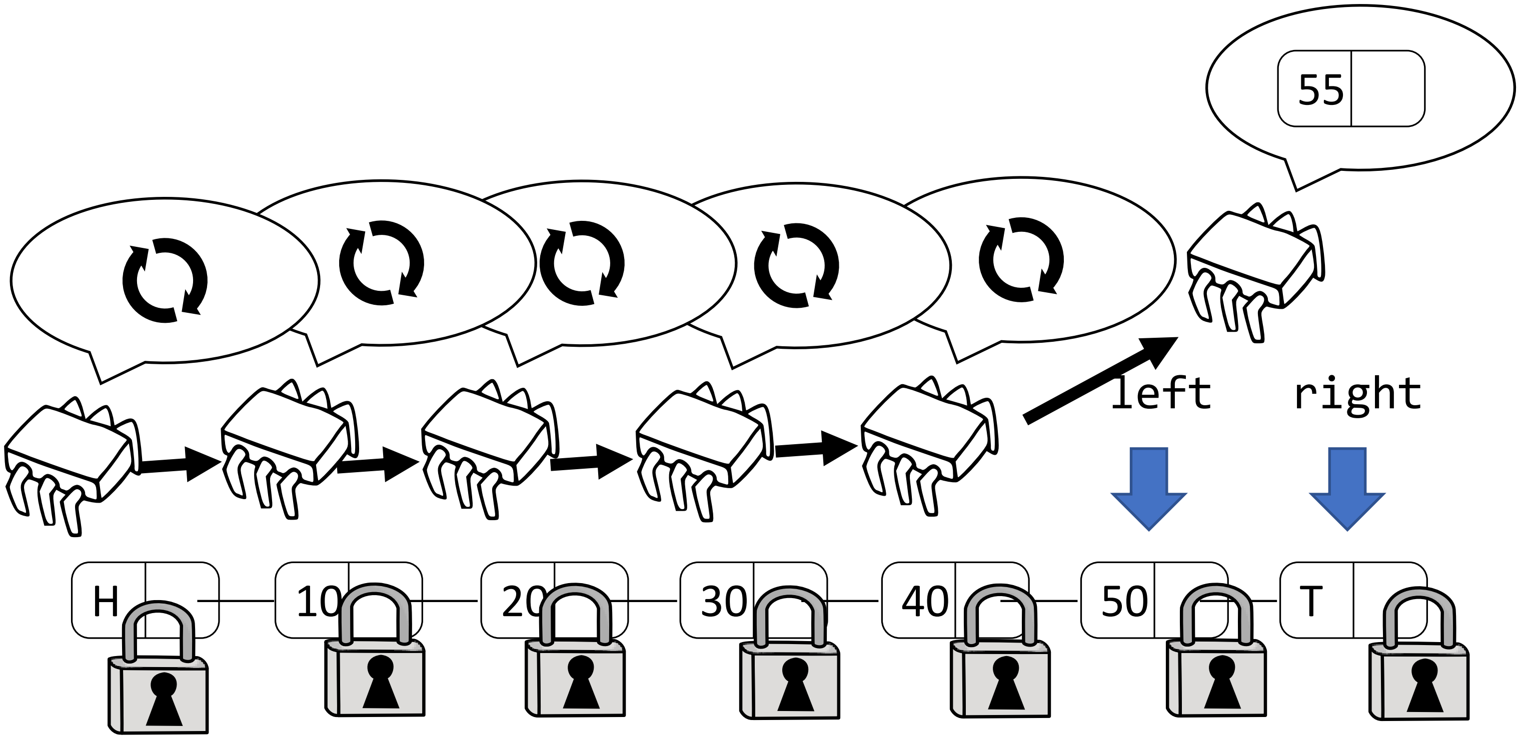
Search algorithm

- Allows an increased parallelism but…



Search algorithm

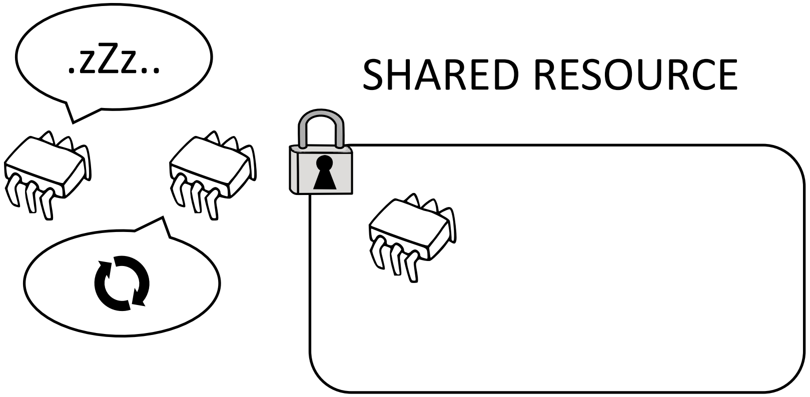
- Allows an increased parallelism but…
- High costs for lock handover



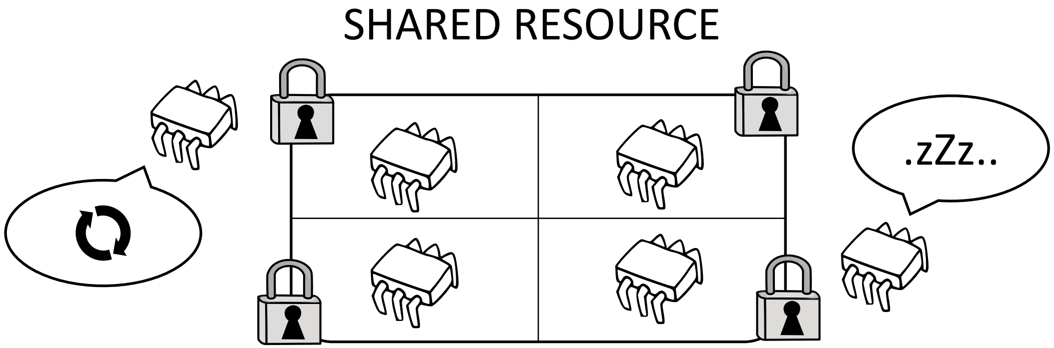
Recap

- Explored two blocking strategies:

1. Global (coarse-grain) lock

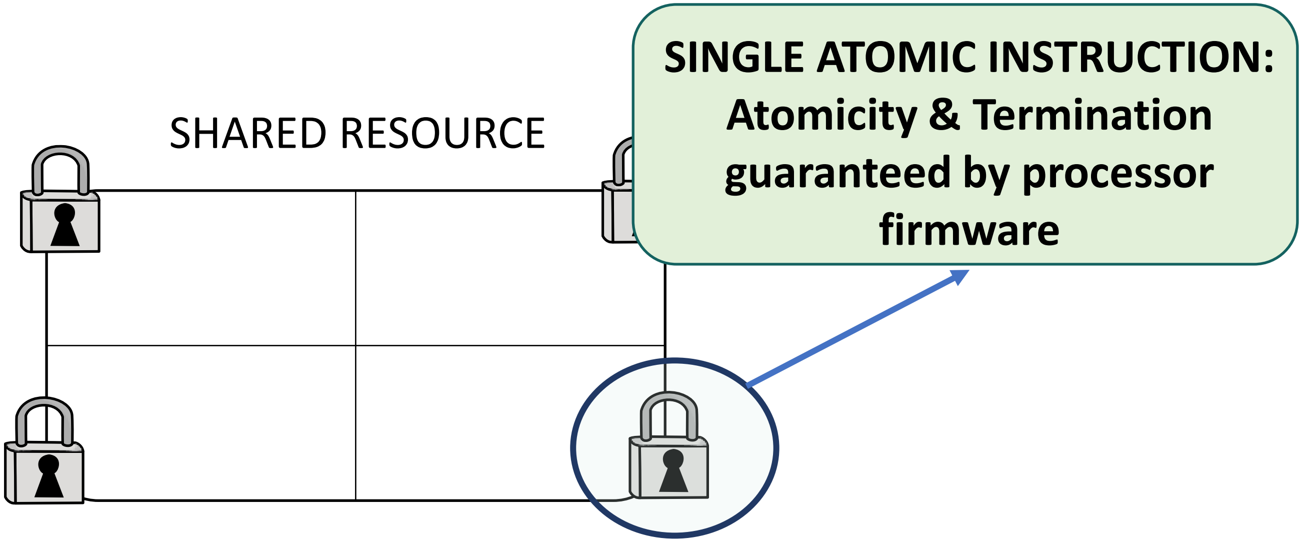


2. (Fine-grain) Lock coupling



Non-blocking algorithms

- We do not rely on locks for synchronization (they make our algorithm dependent on fairness)
- How ? **By ensuring that mutual exclusion regions terminate**
- How??



SINGLE ATOMIC INSTRUCTION:
Atomicity & Termination guaranteed by processor firmware

Read-Modify-Write

- RMW instructions allow to read memory and modify its content in an apparently instantaneous fashion.

```
RMW(MRegister *r, Function f){
  atomic{
    old = r;
    *r = f(r);
    return old;
  }
}
```

- Even conventional atomic Load and Store can be seen as RMW operations

Compare-And-Swap

- Compare-and-Swap (CAS) is an atomic instruction used in multithreading to achieve synchronization
  - It compares the contents of a memory area with a supplied value
  - If and only if they are the same
  - The contents of the memory area are updated with the new provided value
- Atomicity guarantees that the new value is computed based on up-to-date information
- If, in the meanwhile, the value has been updated by another thread, the update fails
- This instruction has been introduced in 1970 in the IBM 370 trying to limit as much as possible the use of spinlocks

Compare-And-Swap

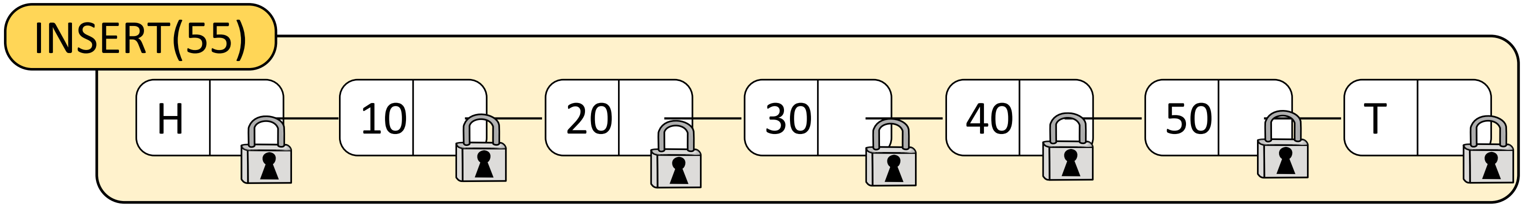
- RMW instructions allow to read memory and modify its content in an apparently instantaneous fashion.

```
CAS(Mregister *r, Value old_value, Value new_value f){
  atomic{
     Value res = *r;
     if(*r == old_value) *r = new_value;
     return res;
  }
}
```

- CAS is implemented by x86 architectures (see CMPXCHG)
- gcc offers the `__sync_val_compare_and_swap` builtin

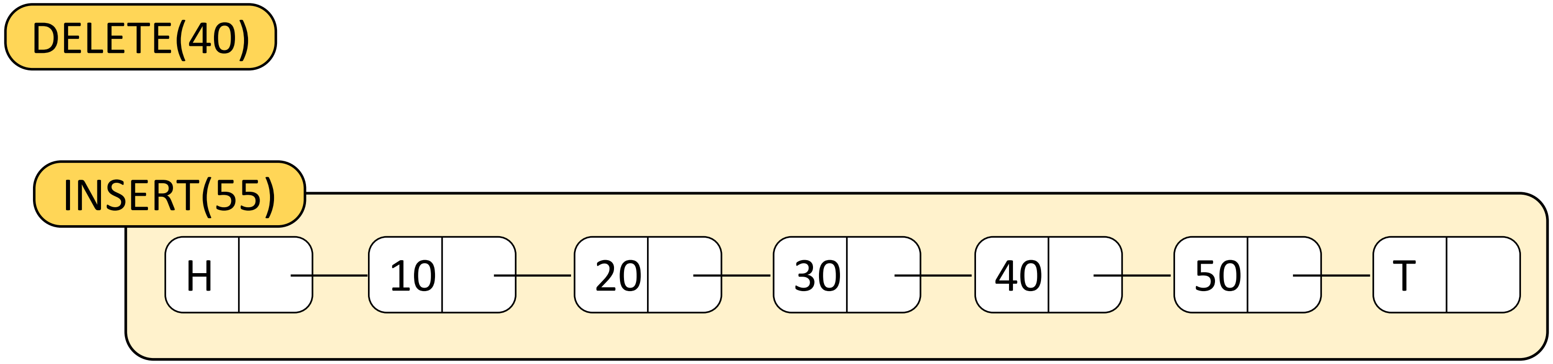
Concurrent set – Attempt 3

DELETE(40)



Concurrent set – Attempt 3

- NON-BLOCKING approach [Harris linked list]
- Search without acquiring any lock
- Apply updates with individual atomic instructions



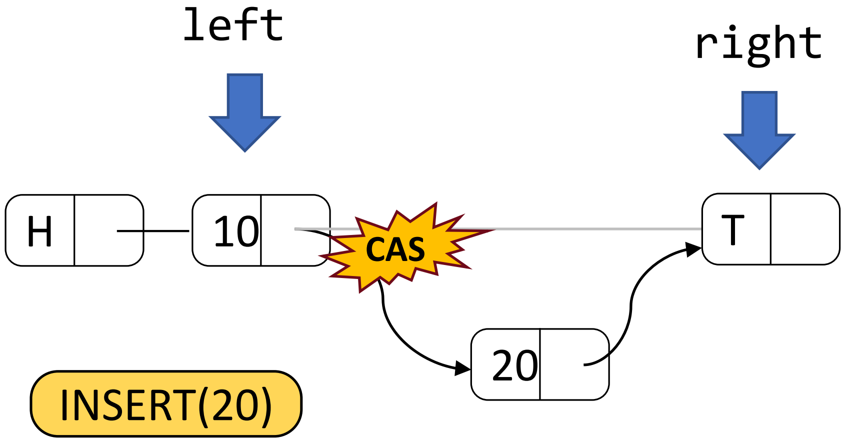
Non-blocking insert & delete algorithms

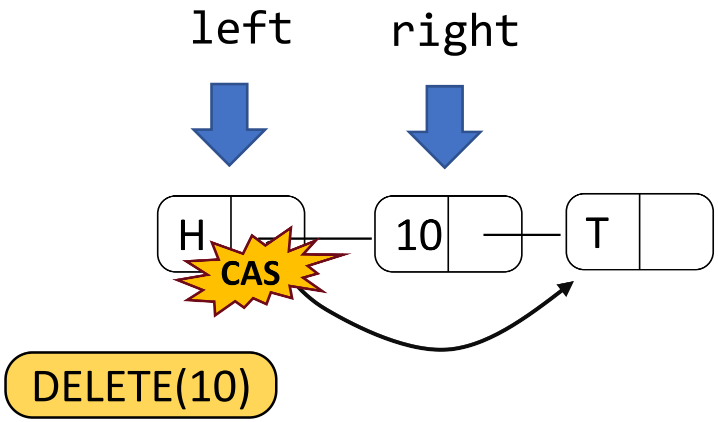
Insert:

1. Search left and right nodes
2. Insert the new item with a CAS
3. If CAS fails restart from 1

Delete:

1. Search left and right nodes
2. Disconnect the item with a CAS
3. If CAS fails restart from 1





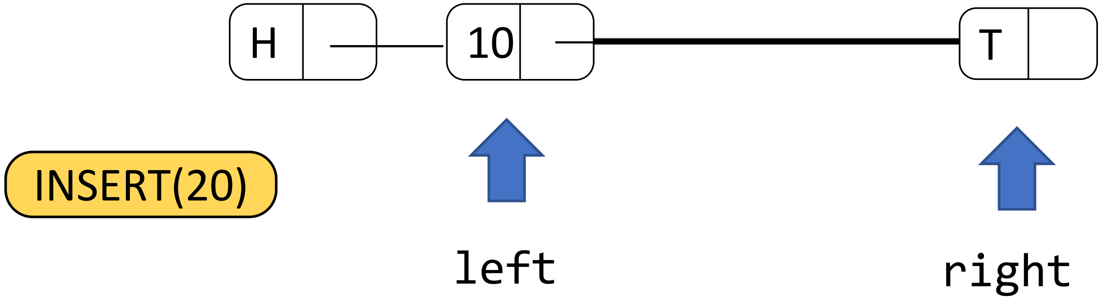
- Is it correct?

Incorrect delete algorithm

- Edge cases might lead to losing items!

Incorrect delete algorithm

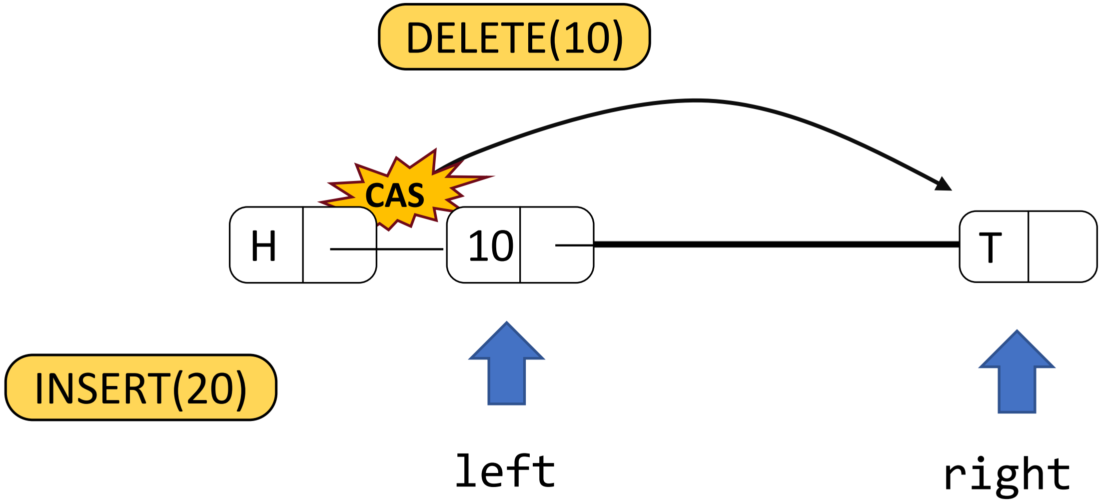
- Edge cases might lead to losing items!



1. Thread A gets left and right node and go to sleep
2. Thread B disconnects the node containing 10
3. Thread A wakes up and add 20 after 10
4. The new item is lost

Incorrect delete algorithm

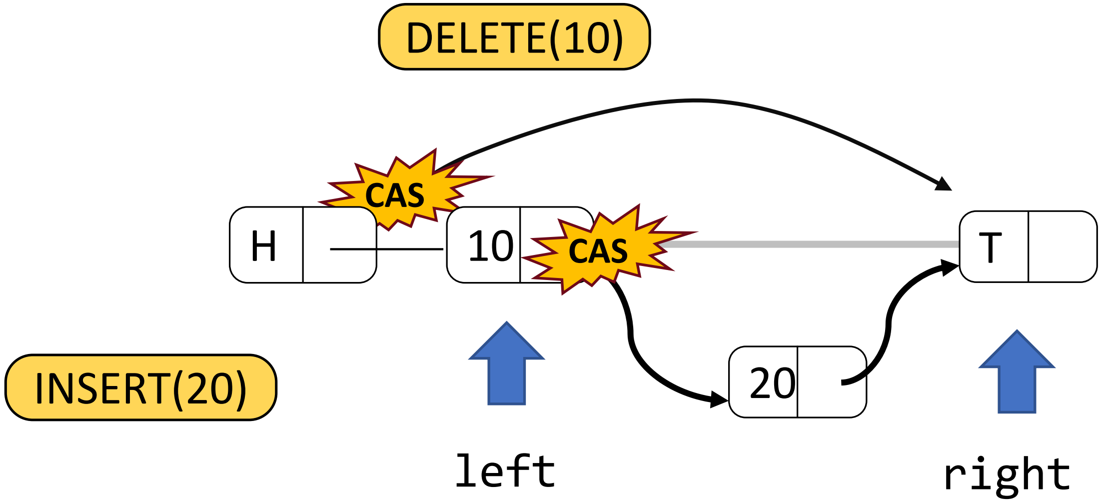
- Edge cases might lead to losing items!



1. Thread A gets left and right node and go to sleep
2. Thread B disconnects the node containing 10
3. Thread A wakes up and add 20 after 10
4. The new item is lost

Incorrect delete algorithm

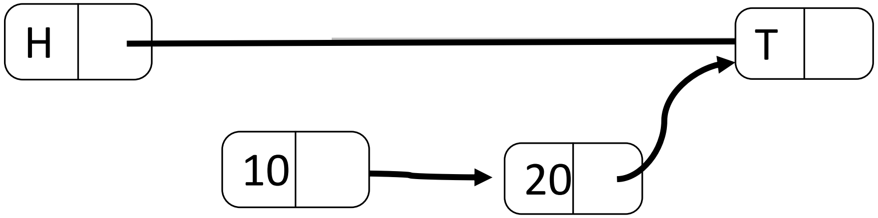
- Edge cases might lead to losing items!



1. Thread A gets left and right node and go to sleep
2. Thread B disconnects the node containing 10
3. Thread A wakes up and add 20 after 10
4. The new item is lost

Incorrect delete algorithm

- Edge cases might lead to losing items!

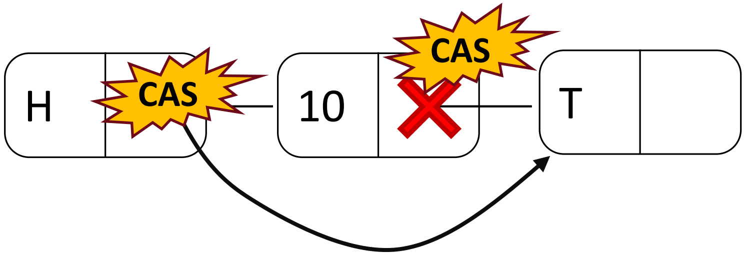


1. Thread A gets left and right node and go to sleep
2. Thread B disconnects the node containing 10
3. Thread A wakes up and add 20 after 10
4. The new item is lost

The correct delete algorithm

- Adopt logical deletion:

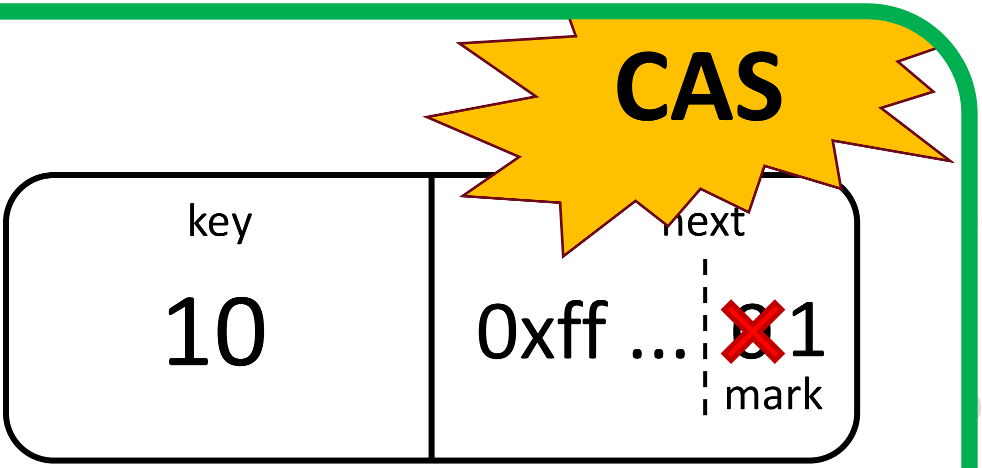
1. Get left and right node
2. Mark the item as deleted via CAS (*logical* deletion)
3. If CAS fails GOTO 1
4. Disconnect the item via CAS (*physical* deletion)
5. If CAS fails GOTO 4



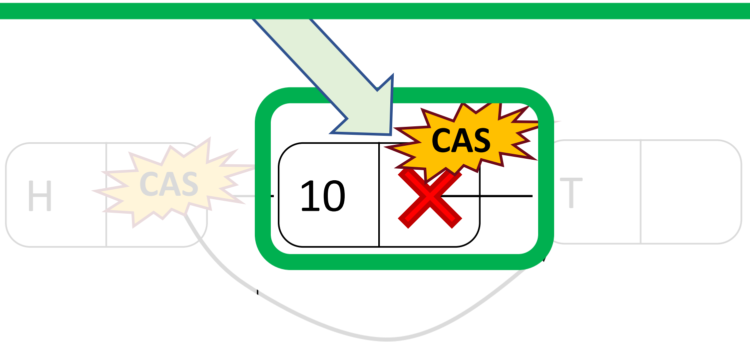
The correct delete algorithm

- Adopt l

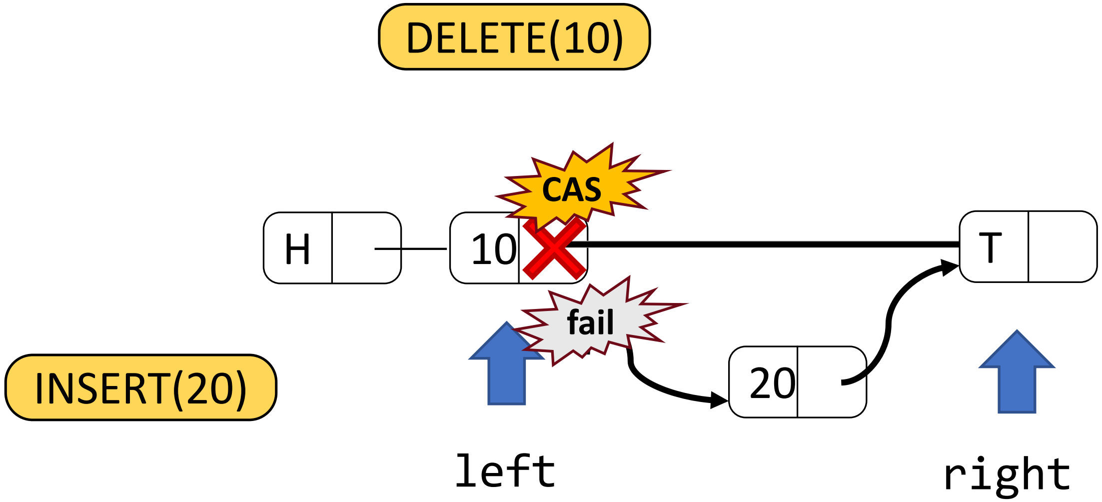
1. Get l
2. Marl etion)
3. If CA
4. Disc n)
5. If CA



- Typically memory objects are byte aligned
- The LSB is always 0! BIT STEALING!!!



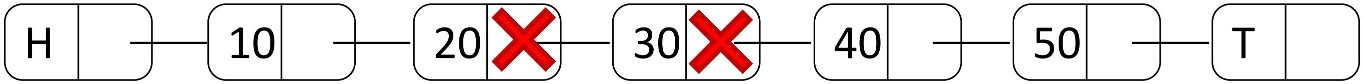
The correct delete algorithm



- Updates of the ”`next`” field by two opposite concurrent operations cannot both succeed
- What to do upon conflict (failed CAS)? RESTART FROM SCRATCH!!

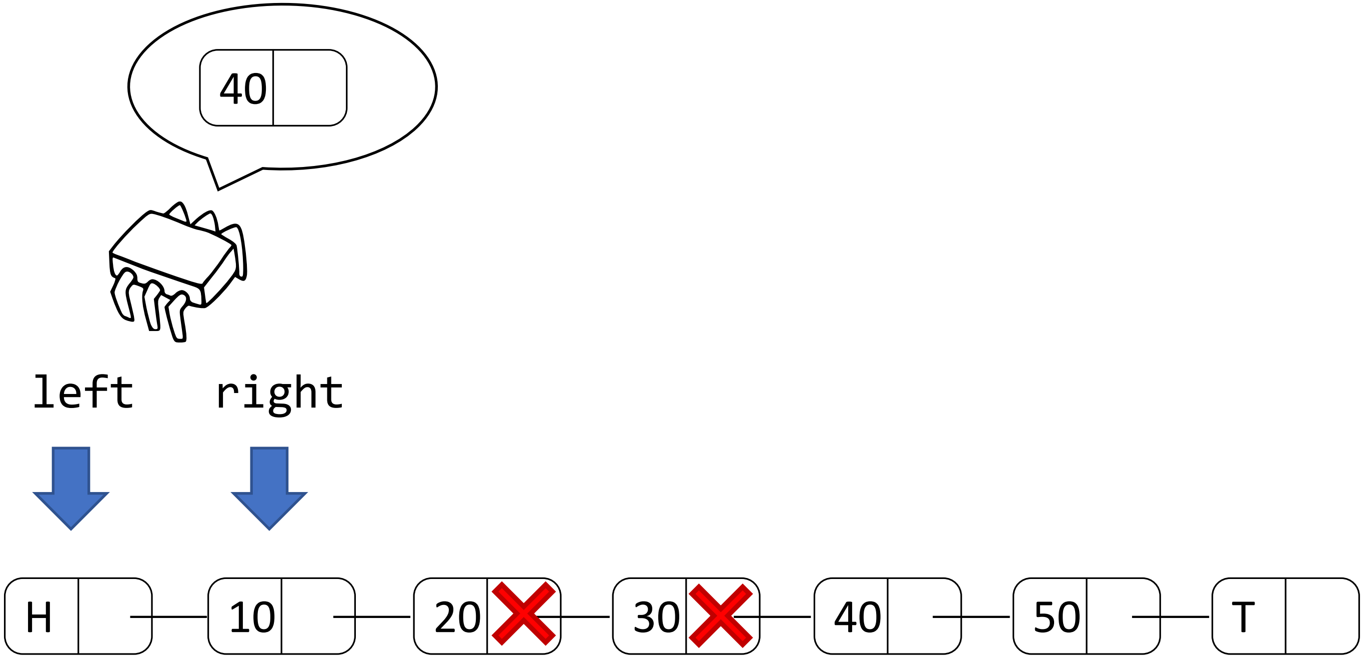
Non-blocking search

- The search returns two adjacent non-marked (left and right) nodes
- Housekeeping: disconnect logically delete items during searches



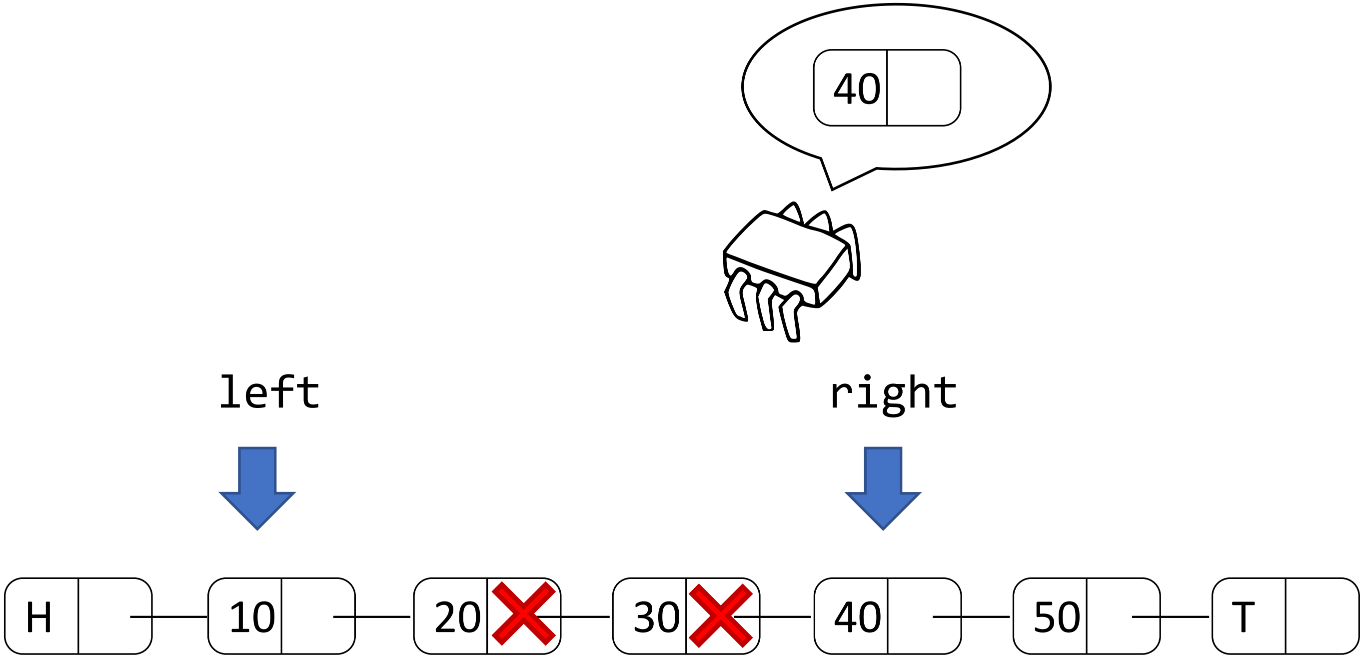
Non-blocking search

- The search returns two adjacent non-marked (left and right) nodes
- Housekeeping: disconnect logically delete items during searches



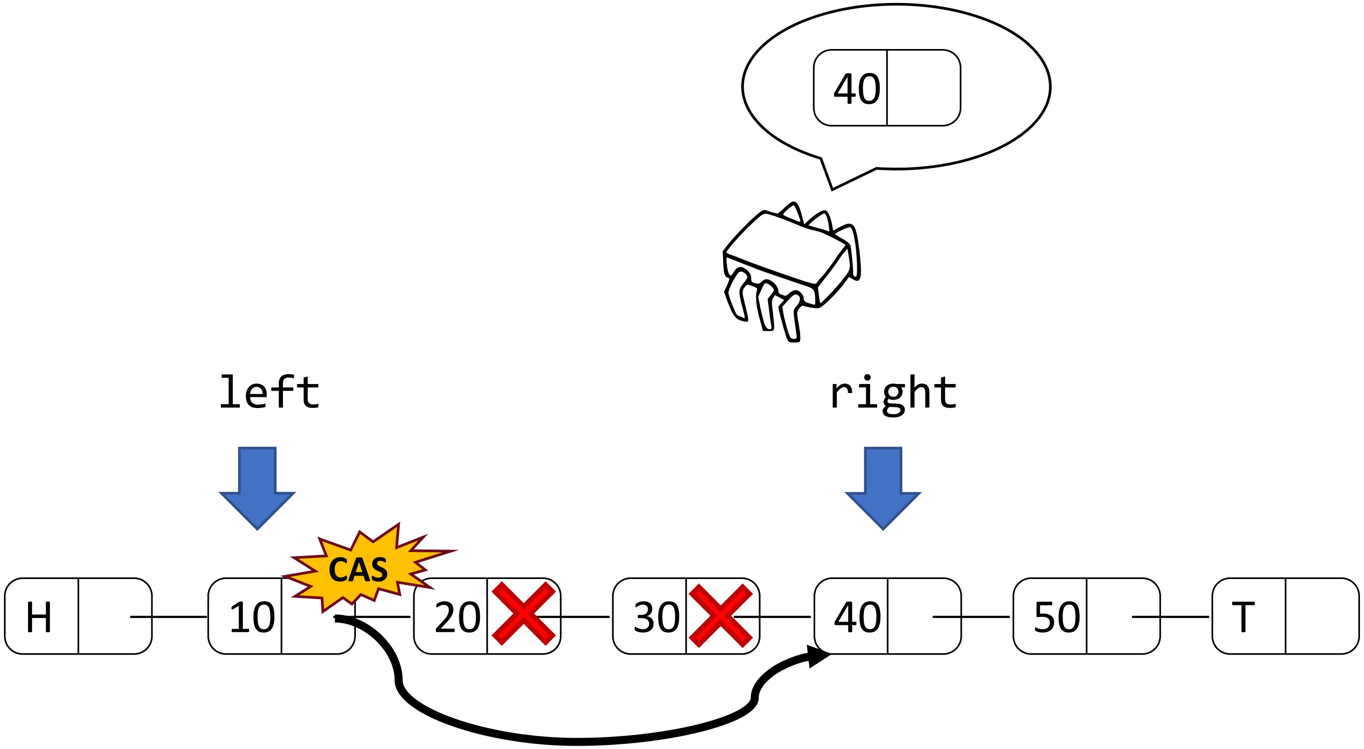
Non-blocking search

- The search returns two adjacent <u>non-marked</u> (left and right) nodes
- Housekeeping: disconnect logically delete items during searches



Non-blocking search

- The search returns two adjacent non-marked (left and right) nodes
- Housekeeping: disconnect logically delete items during searches



Concurrent set – Attempt 3 (SRC)

```
bool do_operation(int k, int op_type){
  node *l,*r, *n = new node(k);
  l = search(k, &r);                 /* get left and right node */
  switch(op_type){
    case(INSERT):
      if(r->key == k) return false;  /* key present in the set */
      n->next = r;
      l->next = n;                   /* insert the item         */


      break;
    case(DELETE):
      if(r->key != k) return false;  /* key not present         */
      l->next = r->next;             /* remove the key          */



      break;
  }
  return true;
}
```

Concurrent set – Attempt 3 (SRC)

```
1. bool do_operation(int k, int op_type){
2.   node *l,*r, *n = new node(k);
3.   l = search(k, &r);                  /* get left and right node */
4.   switch(op_type){
5.     case(INSERT):
6.       if(r->key == k) return false;   /* key present in the set */
7.       n->next = r;
8.       ~~l->next = n;~~                     /* insert the item       */
9.       if(!CAS(&l->next, r, n))
10.         goto 3;         /* insertion failed the item -> restart */
11.      break;
12.     case(DELETE):
13.      if(r->key != k) return false;   /* key not present        */
14.      ~~l->next = r->next;~~              /* remove the key         */
15.      if(is_marked_ref((l=r->next)) || !CAS(&r->next, l, mark(l)))
16.         goto 3;         /* insertion failed the item -> restart */
17.      search(k,&r);                   /* repeat search          */
18.      break;
19.  }
20.  return true;
21.}
```

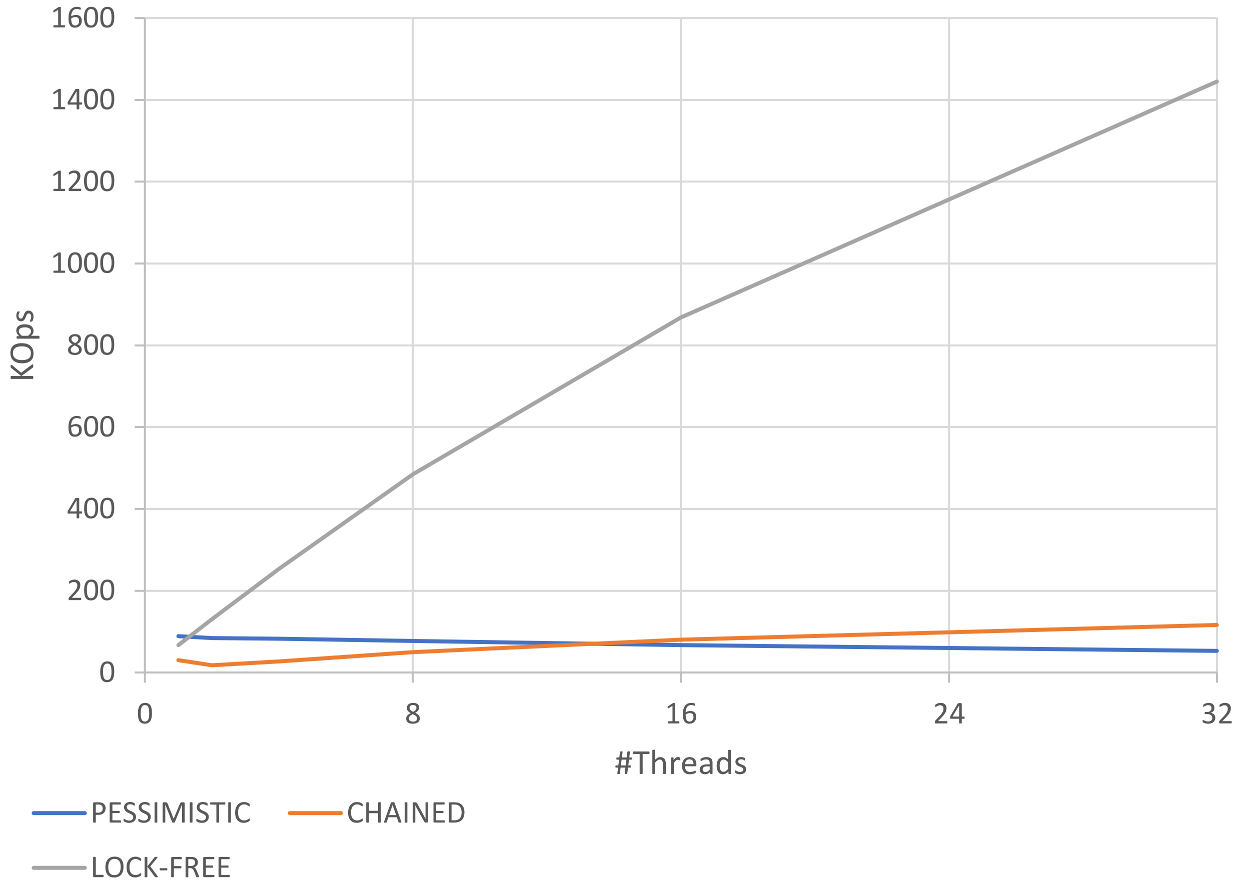
Concurrent set – Attempt 3 (SRC)

```
1. node* search(int k, node **r){
2.   node *l, *t, *t_next, *l_next;
3.   *t = set->head;
4.    t_next = t->head->next;
5.    while(1){                        /* FIND LEFT AND RIGHT NODE */
6.       if(!is_marked(t_next)){
7.          l = t;
8.          l_next = t_next;
9.       }
10.     t = get_unmarked_ref((t_next);
11.     t_next = t->next;
12.     if(!is_marked_ref(t_next) && t->key >= k) break;
13.  }
14.  *r = t;
15.  /* DEL MARKED NODES */
16.  if(l_next != *r && !CAS(&l->next, l_next, *r) goto 3;
17.  return l;
18.}
```

Concurrent set – Attempt 3

AMD Opteron 6128 – 32Cores

KeyRange = [0,6000] SetSize = 2400 Update=100%



Safety and liveness guarantees

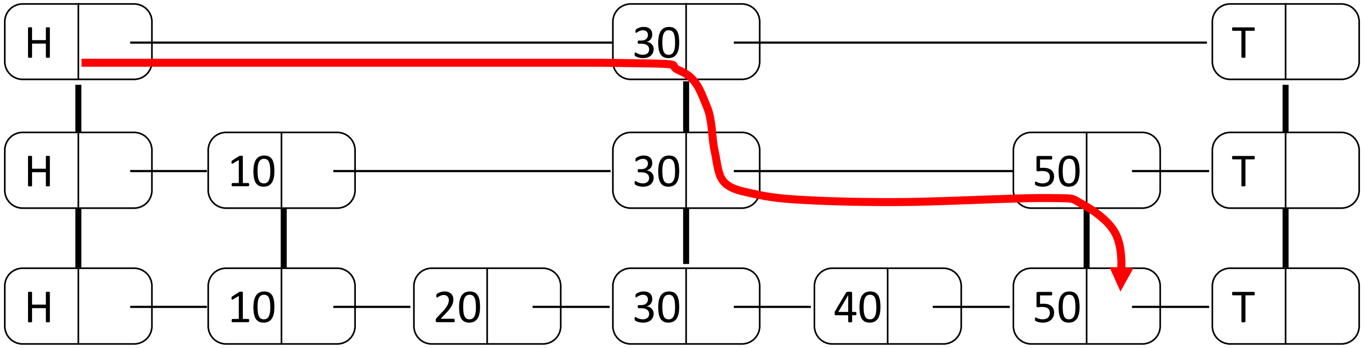
- The algorithm is NON-BLOCKING (LOCK-FREE):
  - If a thread A is stuck in a retry loop => a CAS fails each time
  - If a CAS fail, it is because of another CAS that was successfully executed by a thread B
  - Thread B is making progress
- The algorithm is LINEARIZABLE:
  - Each method execution take effect in an atomic point (a successful CAS) contained between its invocation and reply
  - The order obtained by using these points has been proved to be compliant with the Set semantic

Problems & Solutions

- Problems arise when re-using memory:
  - The CAS suffers from the ABA problem
  - We might reuse a node which is concurrently accessed by another thread (e.g. during a search)
- Solutions:
  1. Use a tag that changes every time a field has been update (even when overwritten with the same value)
     - Pros: easy to implement
     - Cons: ABA might still occur, but with low probability
  2. Adopt garbage collectors that enable safe memory reusage
     - Pros: solve all problems
     - Cons: Hard to implement efficiently

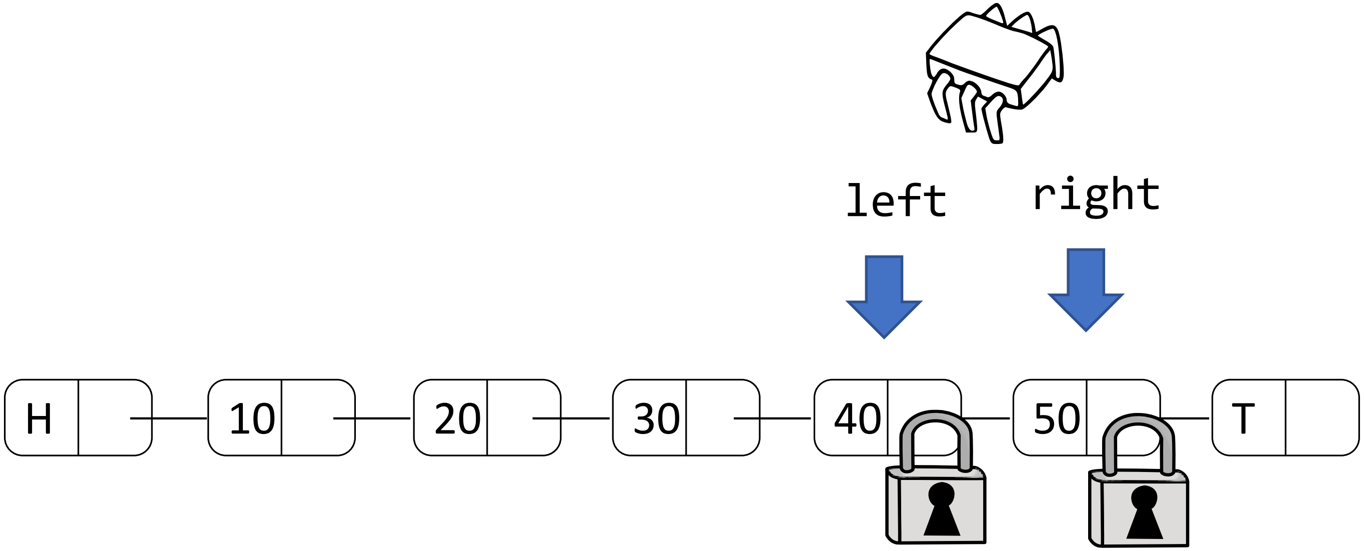
Can we do better?

- Starting from this “simple” set implementation we can build faster set implementations
  - Skip lists (O(logn))
  - Hash tables (O(1))
- Most of them are based on similar techniques:
  - use a linked list
  - build an index on top of it to accelerate look ups



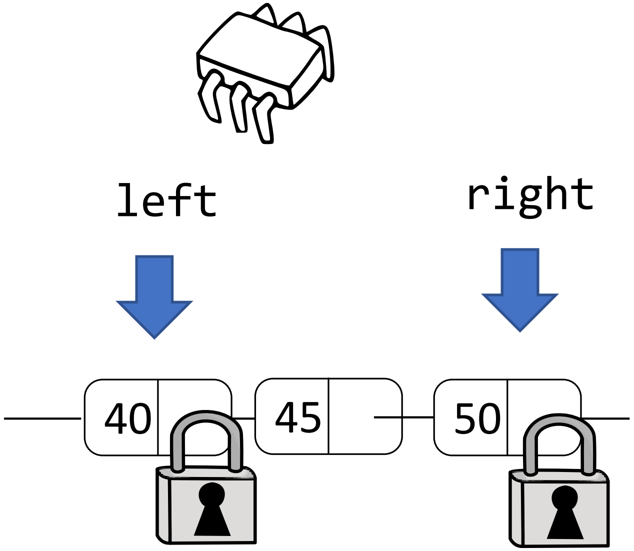
Lazy Linked List

- Wait-free search (no retry)
- Mark has its own memory field



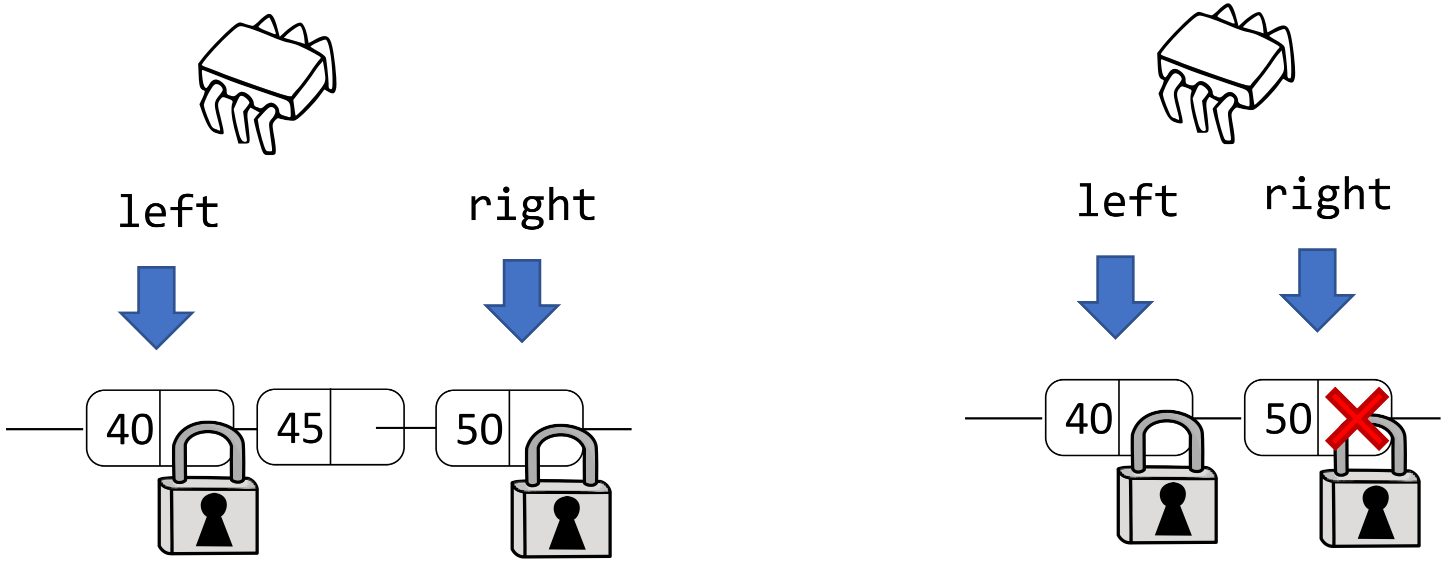
Lazy Linked List

- Wait-free search (no retry)
- Mark has its own memory field



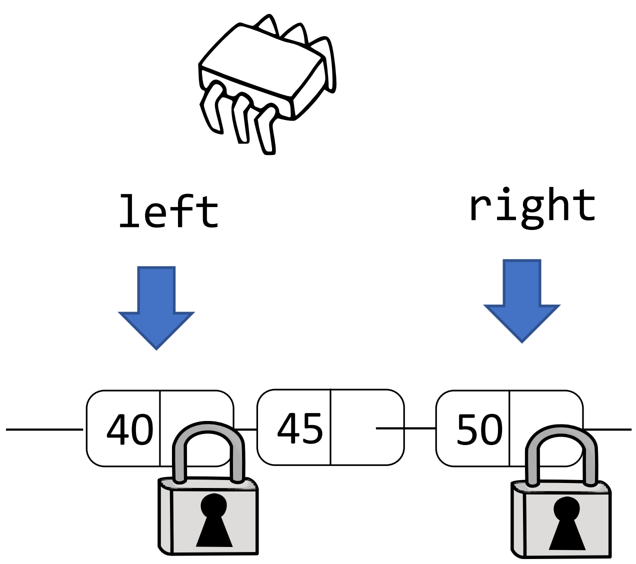
Lazy Linked List

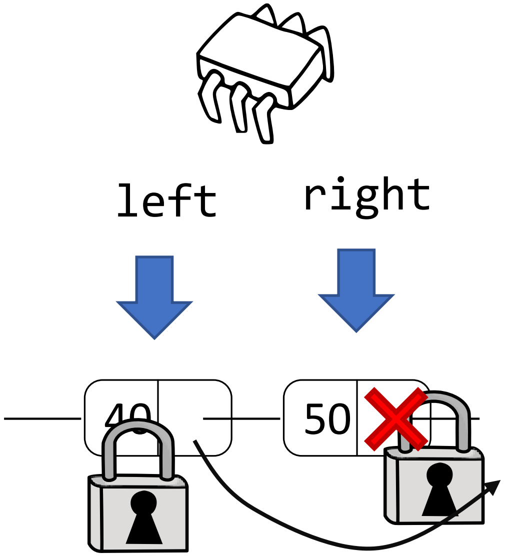
- Wait-free search (no retry)
- Mark has its own memory field



Lazy Linked List

- Wait-free search (no retry)
- Mark has its own memory field

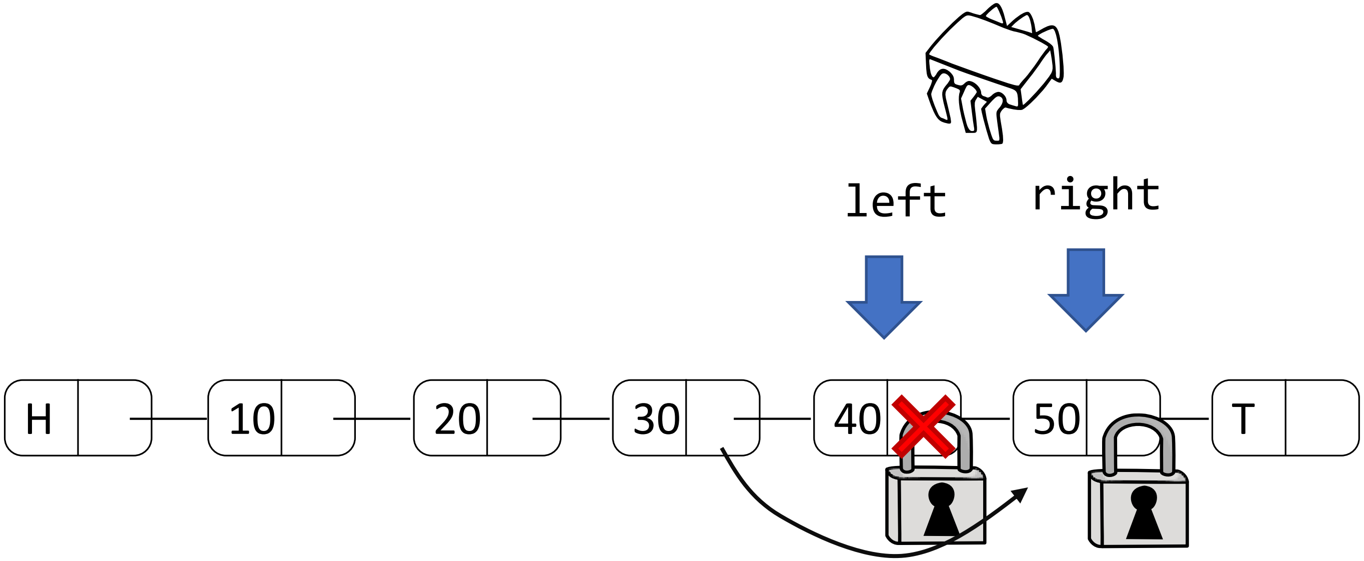




- Validate left and right before apply an update:
    - Left is unmarked
    - Right is unmarked

Lazy Linked List

- Wait-free search (no retry)
- Mark has its own memory field

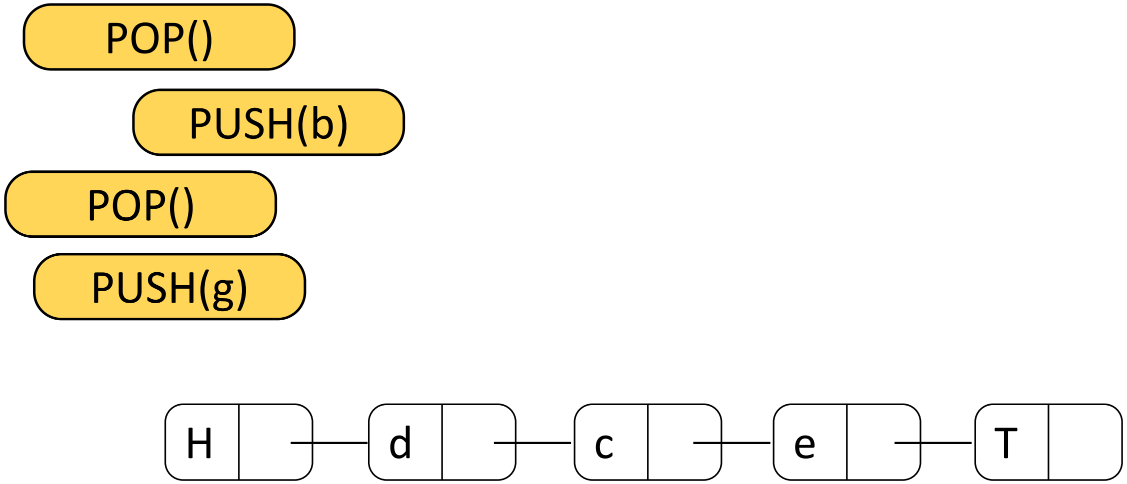


- Validate left and right before apply an update:
    - Left is unmarked
    - Right is unmarked

Concurrent Data Structures: **Non-blocking stacks**

Stack implementation

- Stack methods:
  - `push(v)`
  - `pop()`
- Implemented as a linked list



POP()

PUSH(b)

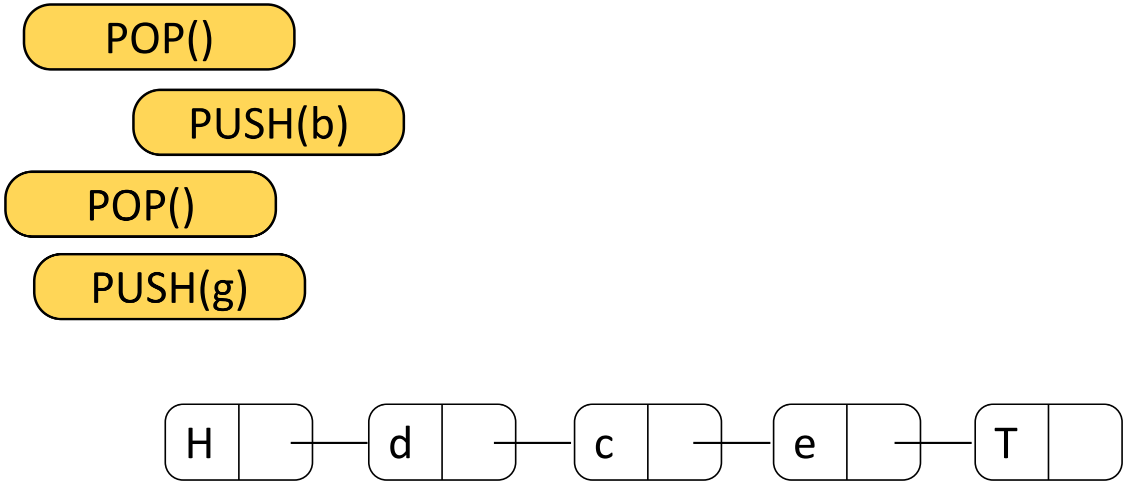
POP()

PUSH(g)

H — d — c — e — T

Concurrent stack implementations

- Resort to a global lock
    - Do not scale
- Resort to a non-blocking approach



POP()

PUSH(b)

POP()

PUSH(g)

H d c e T

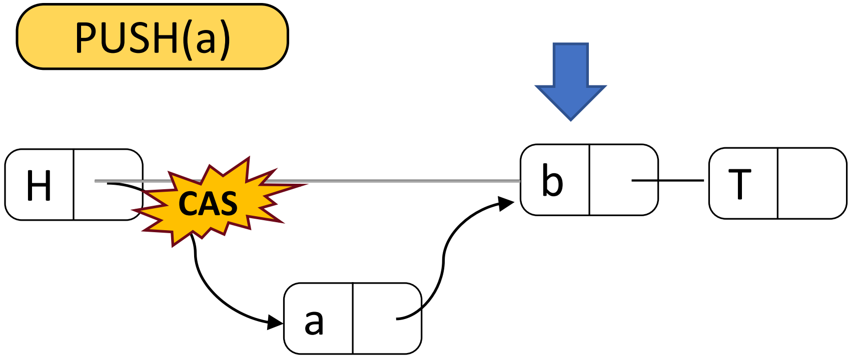
Non-blocking stack – Attempt 1 [Treiber]

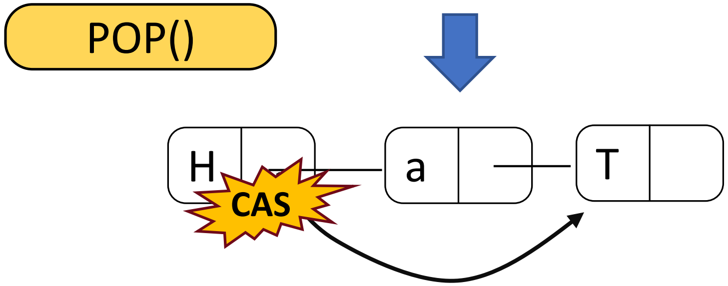
Push:

1. Get head next
2. Insert the new item with a CAS
3. If CAS fails, restart

Delete:

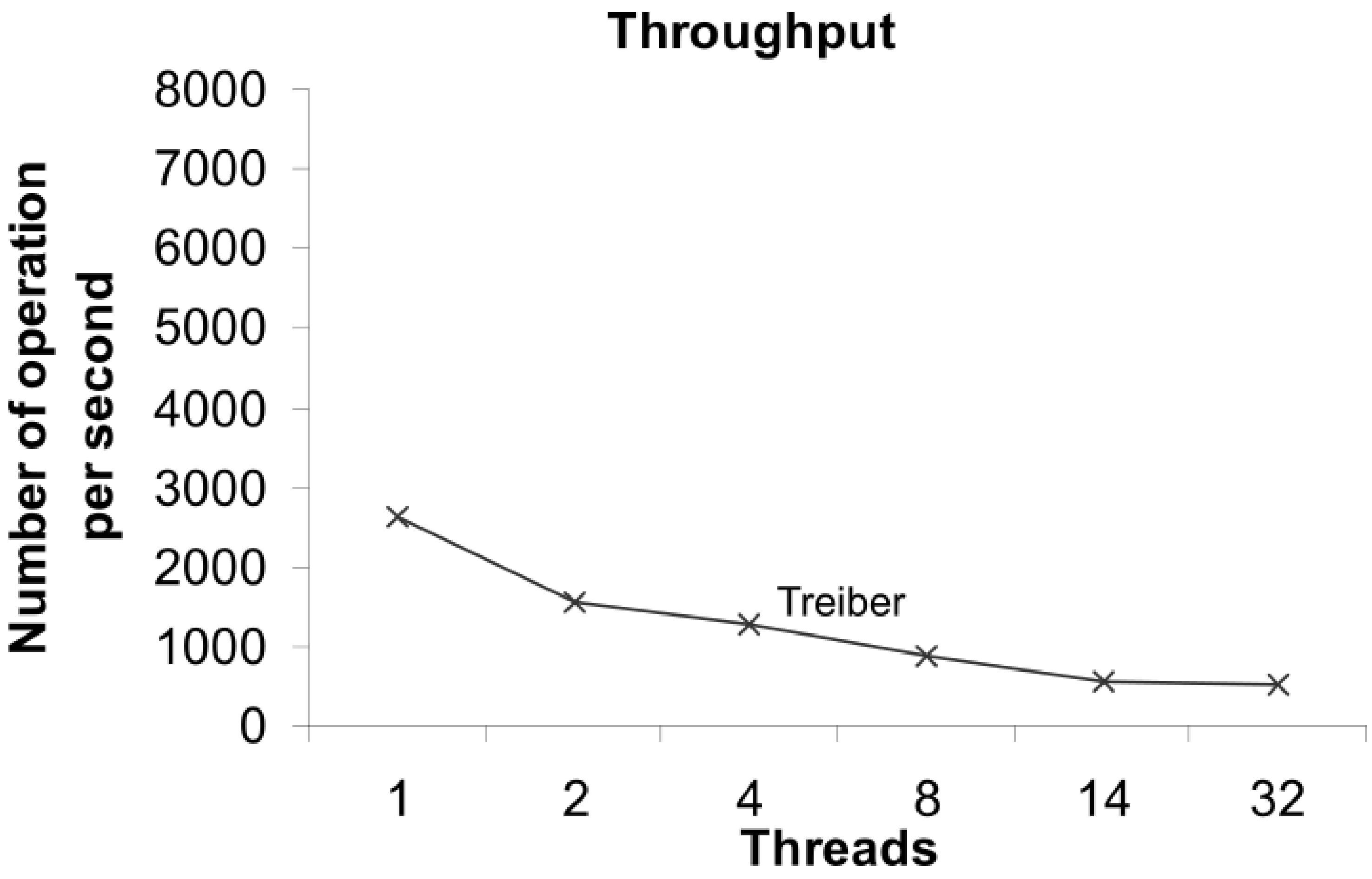
1. Get head next
2. Disconnect the item with a CAS
3. If CAS fails, restart





- Is it scalable?

Non-blocking stack – Attempt 1 [Treiber]



Non-blocking stack – Attempt 2 [Treiber+BO]

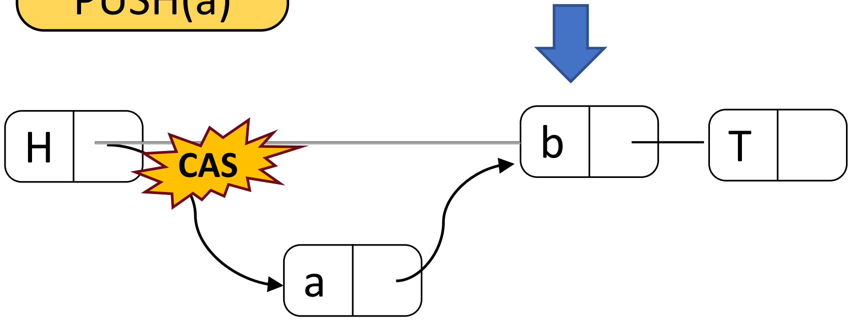
Push:

1. Get head next
2. Insert the new item with a CAS
3. If CAS fails, ~~restart~~ backoff and restart

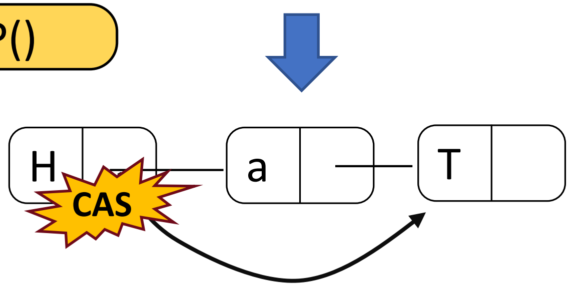
Delete:

1. Get head next
2. Disconnect the item with a CAS
3. If CAS fails, ~~restart~~ backoff and restart

PUSH(a)

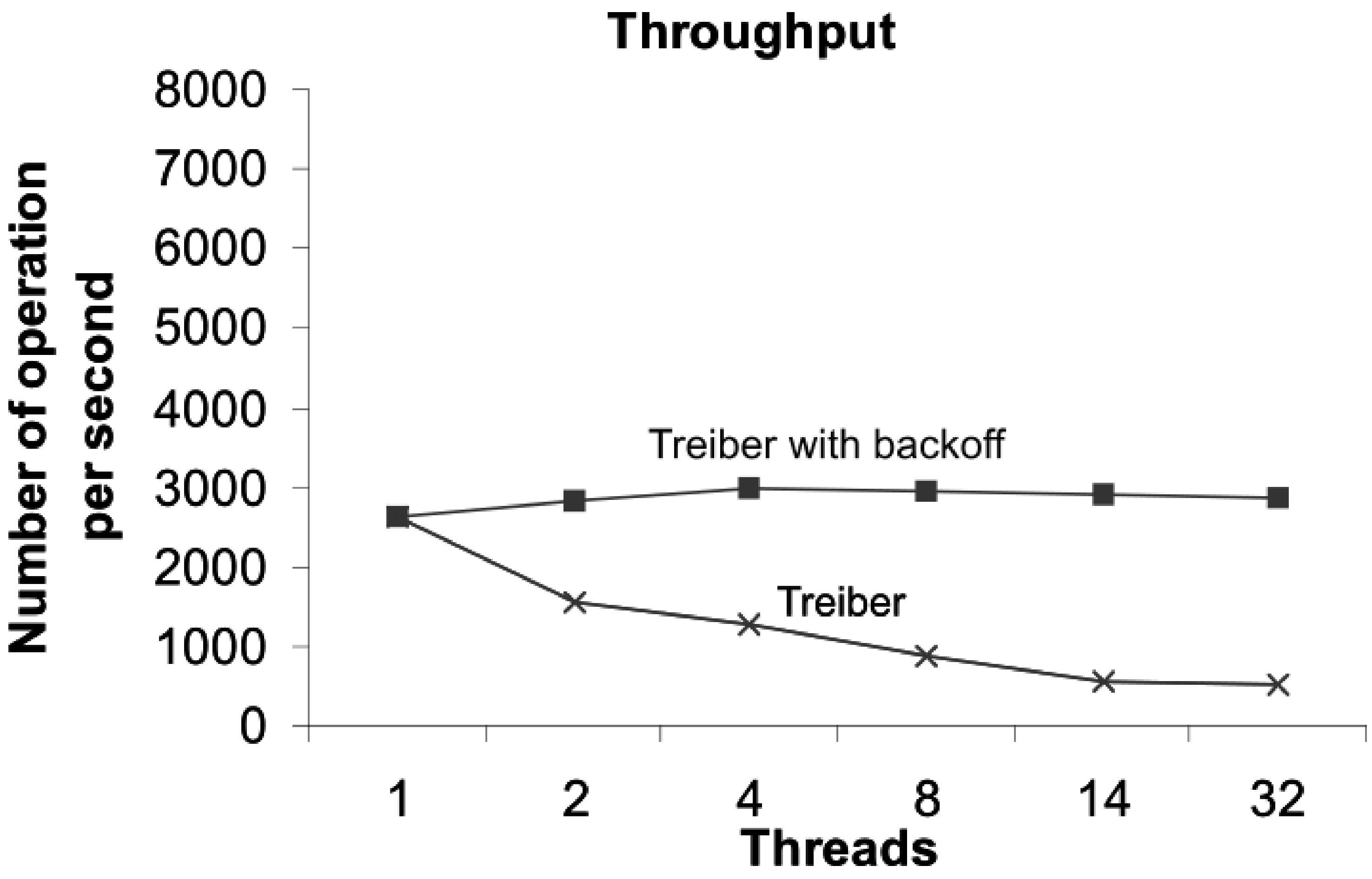


POP()



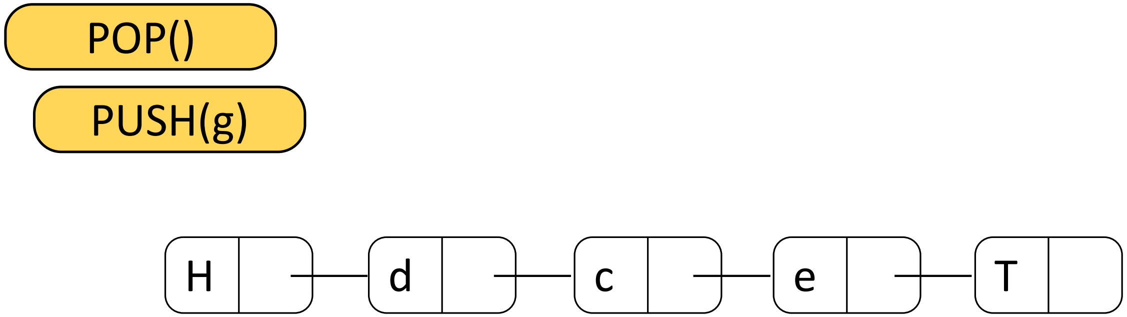
- Is it scalable?

Non-blocking stack – Attempt 2 [Treiber+BO]



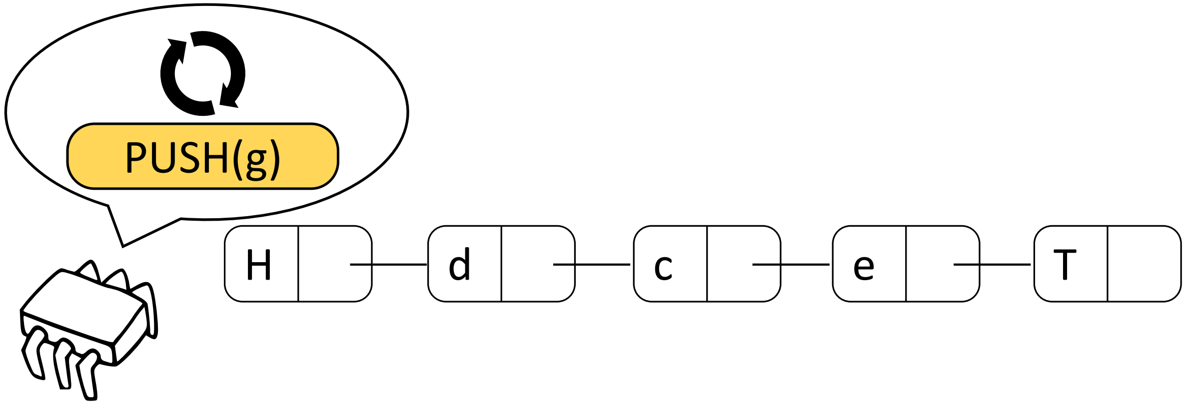
Concurrent stack implementations

- Resort to a global lock
    - Do not scale
- Resort to a naïve non-blocking approach
    - Do not scale
- Resort to a naïve non-blocking approach + Back off
    - Do not scale, but conflict resilient
- How achieve scalability? **Make back-off times useful**



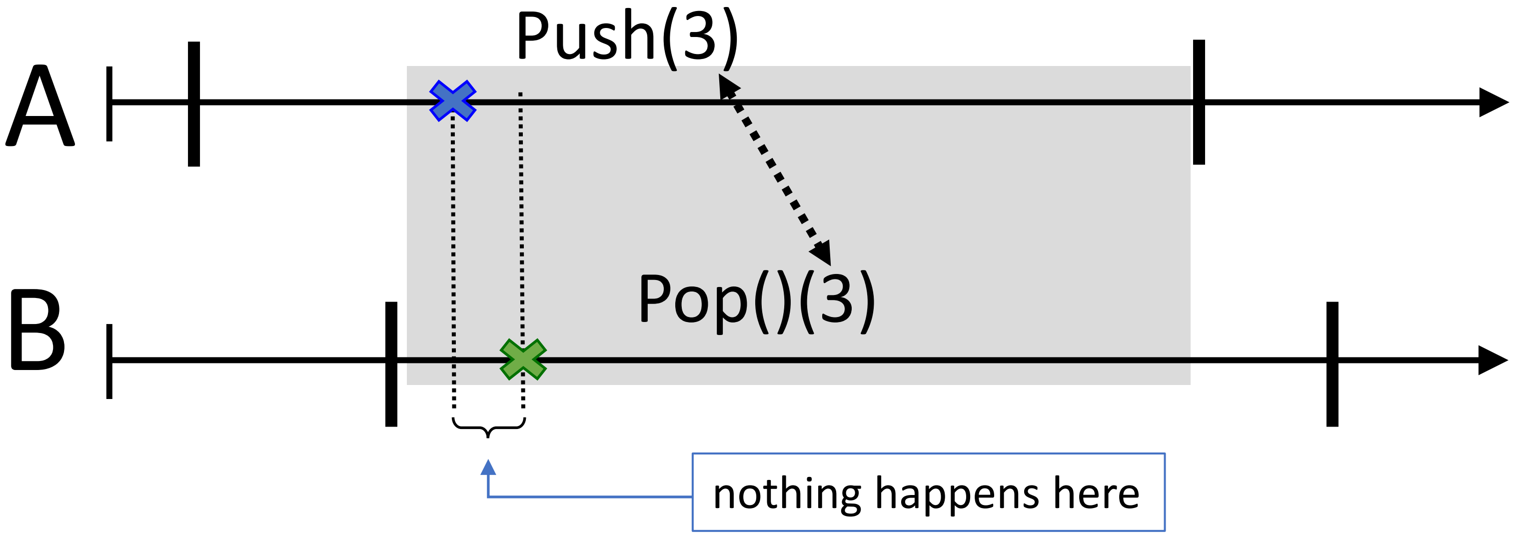
Non-blocking stack – Attempt 3

- How to take advantage of back-off times?



Observation

- Concurrent matching push/pop pairs are always linearizable



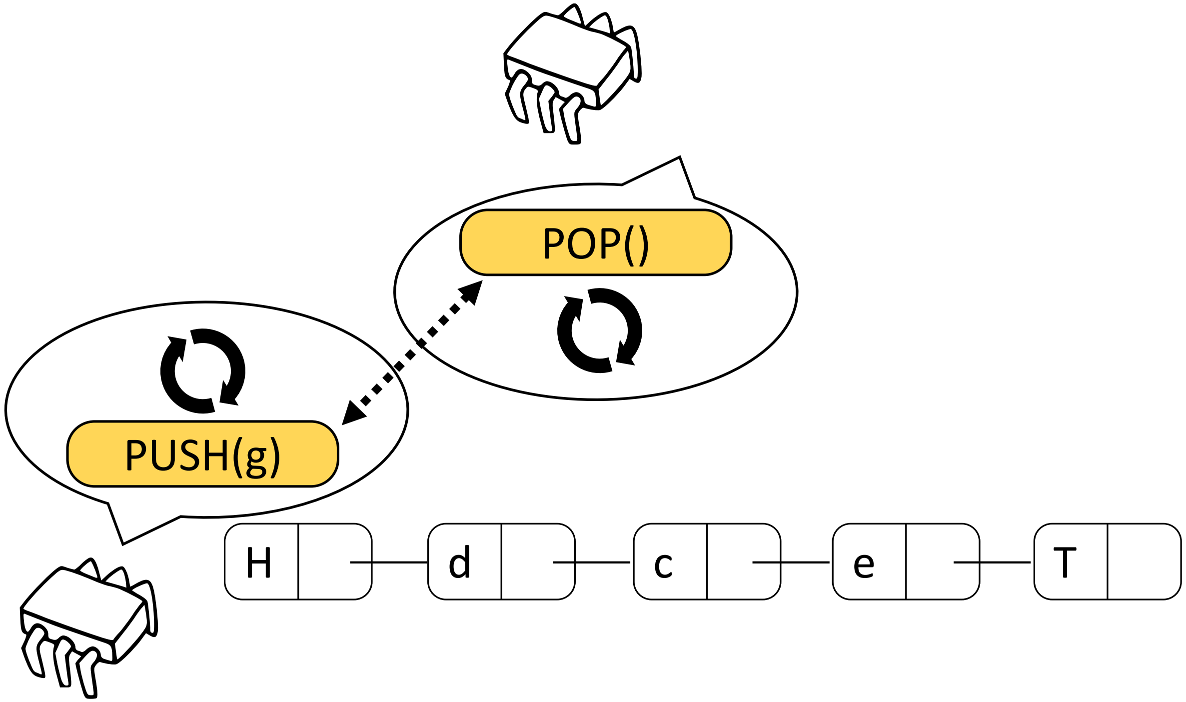
- A push A and a pop B are:
  - concurrent to each other
  - B returns the item inserted by A

⇒ we can always take two points such that:

- A is the last one to insert an item before A linearizes
- B appears to extract the last item inserted (by A)

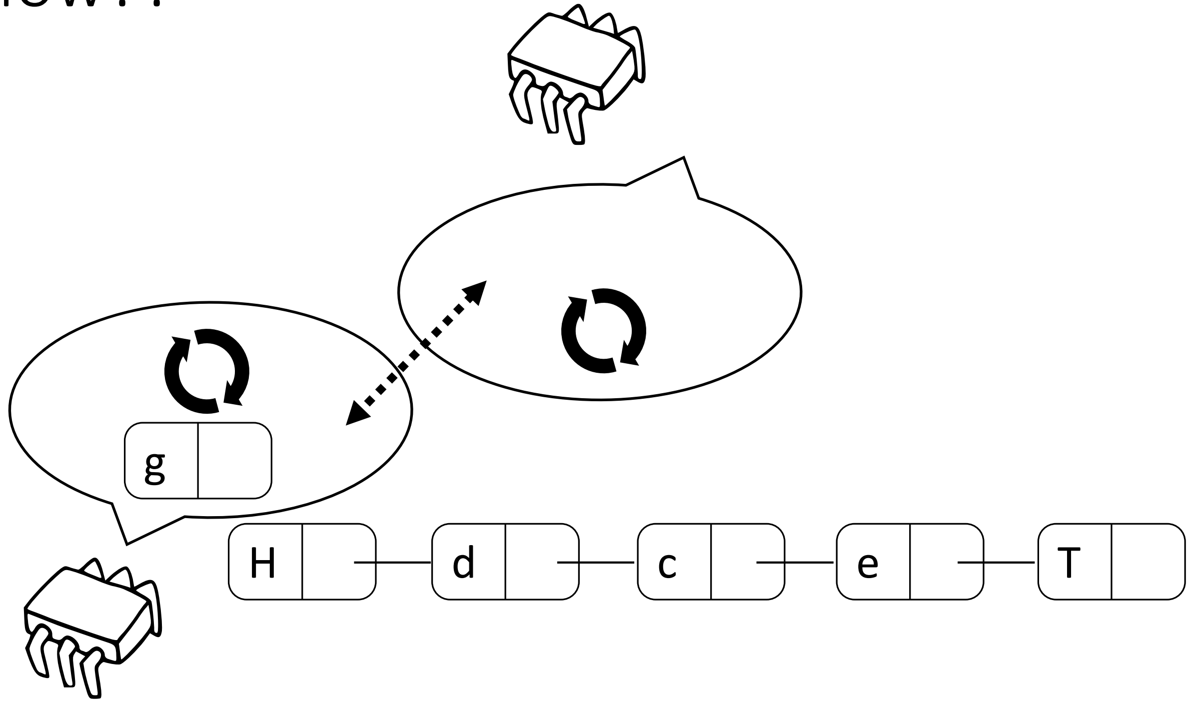
Non-blocking stack – Attempt 3

- How to take advantage of back-off times?
- Hope that an opposite operation arrives while waiting
- Match the two without interacting with the stack



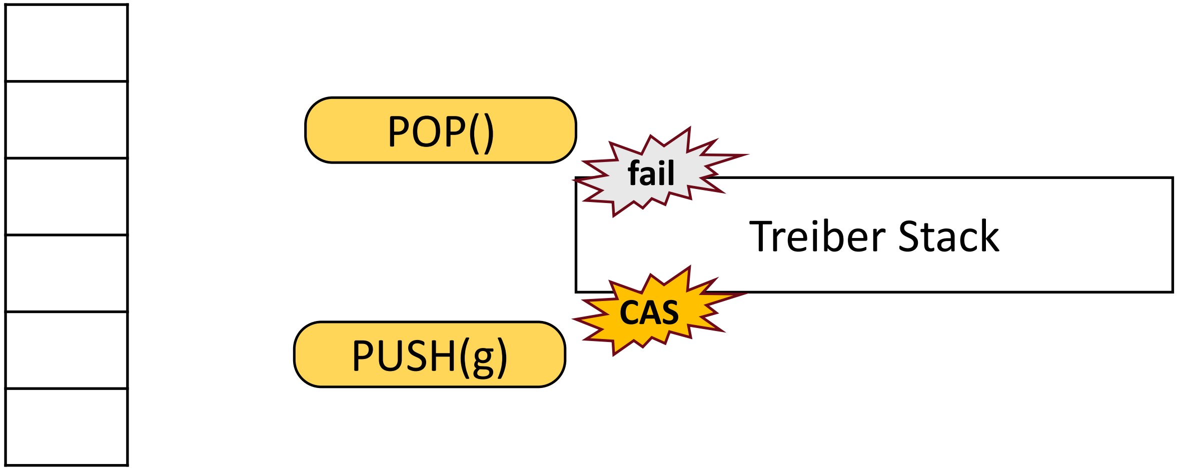
Non-blocking stack – Attempt 3

- How to take advantage of back-off times?
- Hope that an opposite operation arrives while waiting
- Match the two without interacting with the stack
- How??



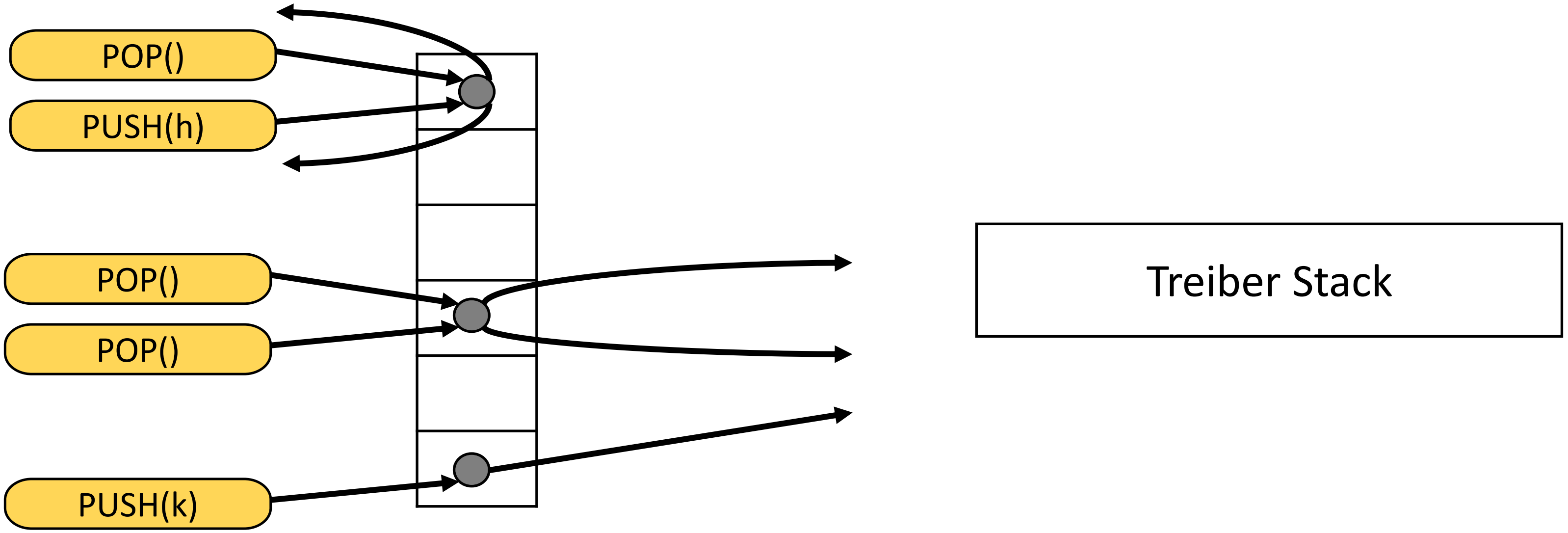
Non-blocking stack – Elimination stack

- Pair the Treiber stack with an array
- Algorithm:
  1. Update the original stack via CAS
  2. If CAS fails, publish the operation in a random cell of the array

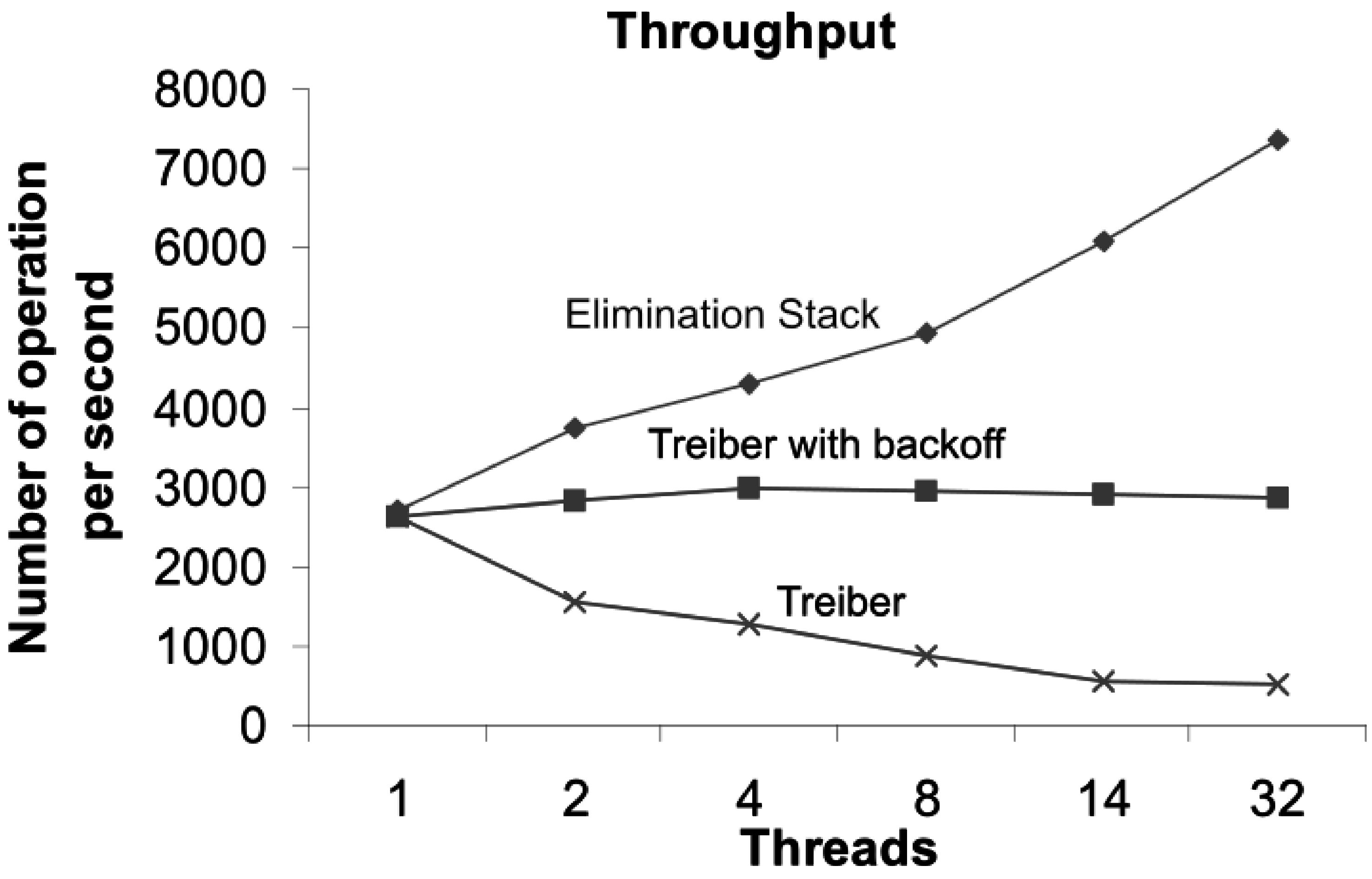


Non-blocking stack – Elimination stack

- Pair the Treiber stack with an array
- Algorithm:
  1. Update the original stack via CAS
  2. If CAS fails, publish the operation in a random cell of the array
  3. Wait for a matching operation
  4. If no matching op, GOTO 1



Non-blocking stack – Attempt 3

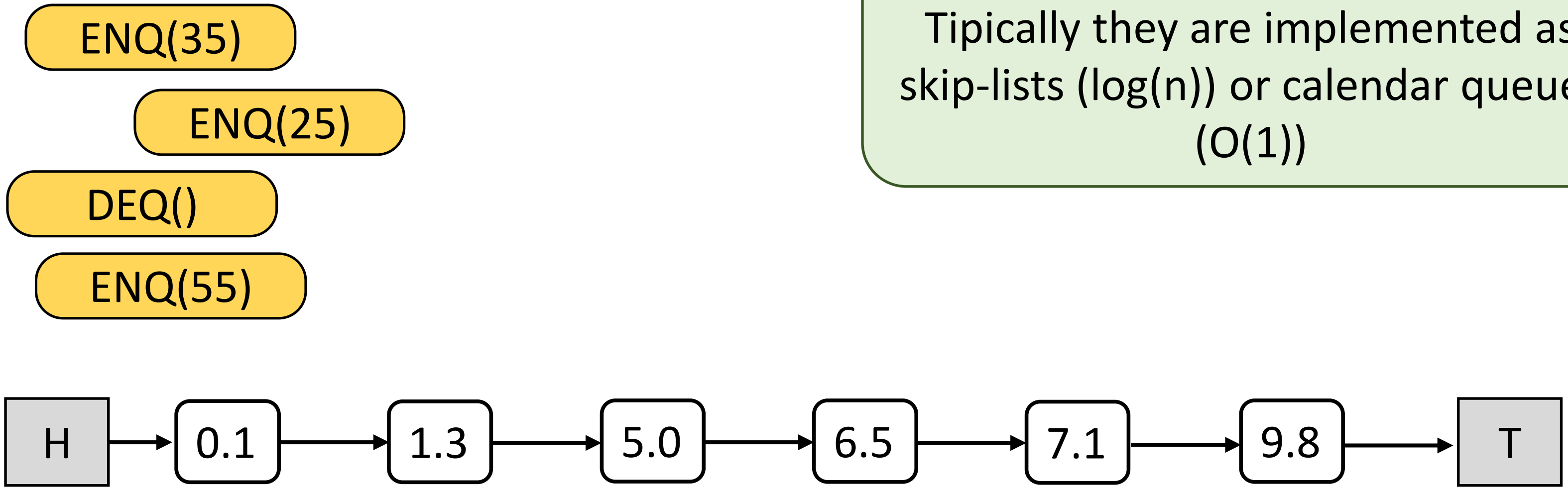


Concurrent Data Structures: Non-blocking priority queues

Priority queue implementations

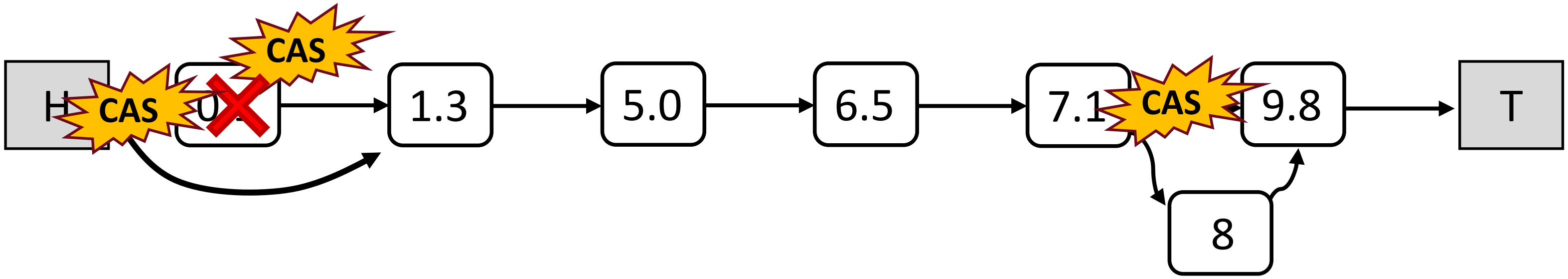
- Priority Queue methods:
  - `enqueue(k): adds a new item`
  - `dequeue(): returns and remove the highest priority item`
- Implemented as an ordered linked list

This is a huge simplification.
Tipically they are implemented as skip-lists (log(n)) or calendar queues (O(1))

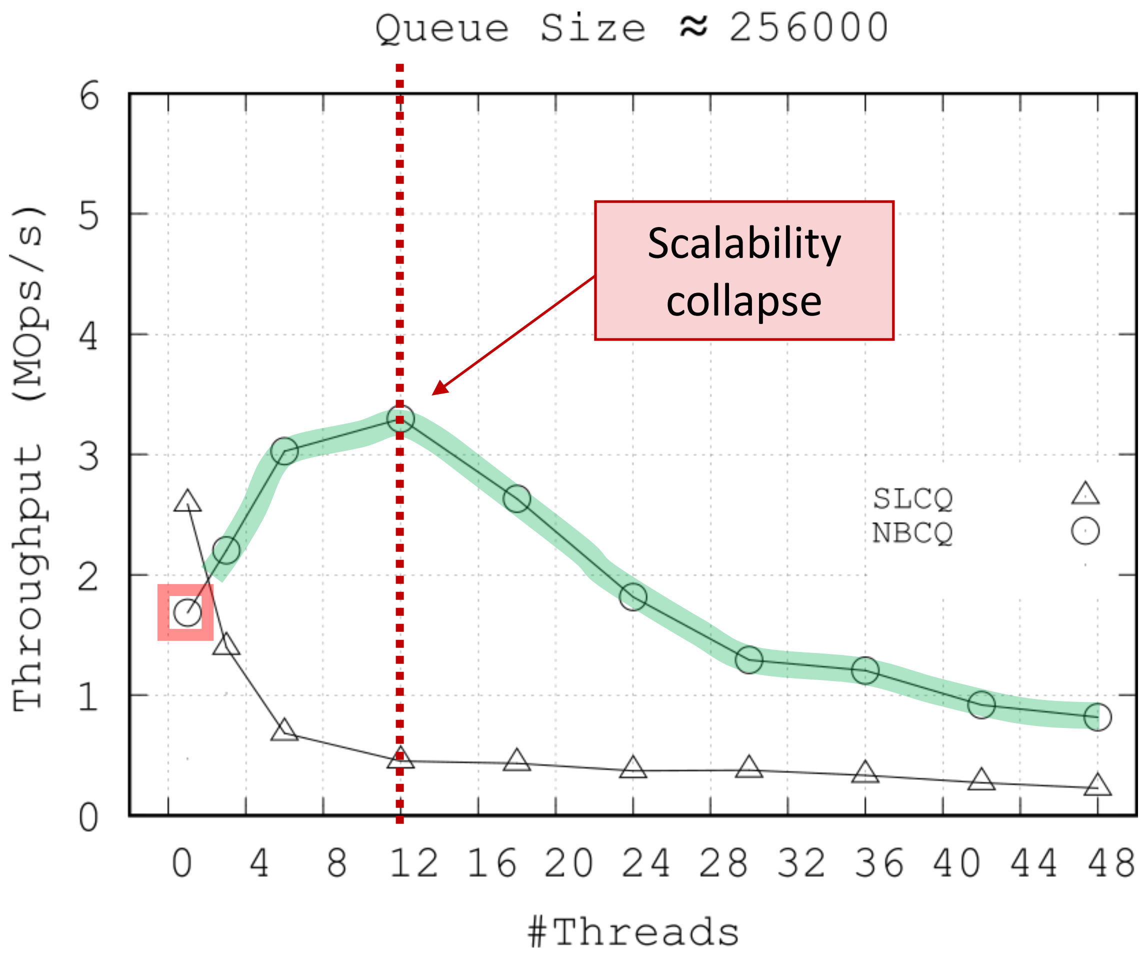


Priority queue – Attempt 1

- Enqueue: works as insertions in the non-blocking Set
  - Connect via CAS
- Dequeues: work as deletions in the non-blocking Set
  - Mark as logically deleted, but
  - DISCONNECT IMMEDIATELY
- Is it scalable?



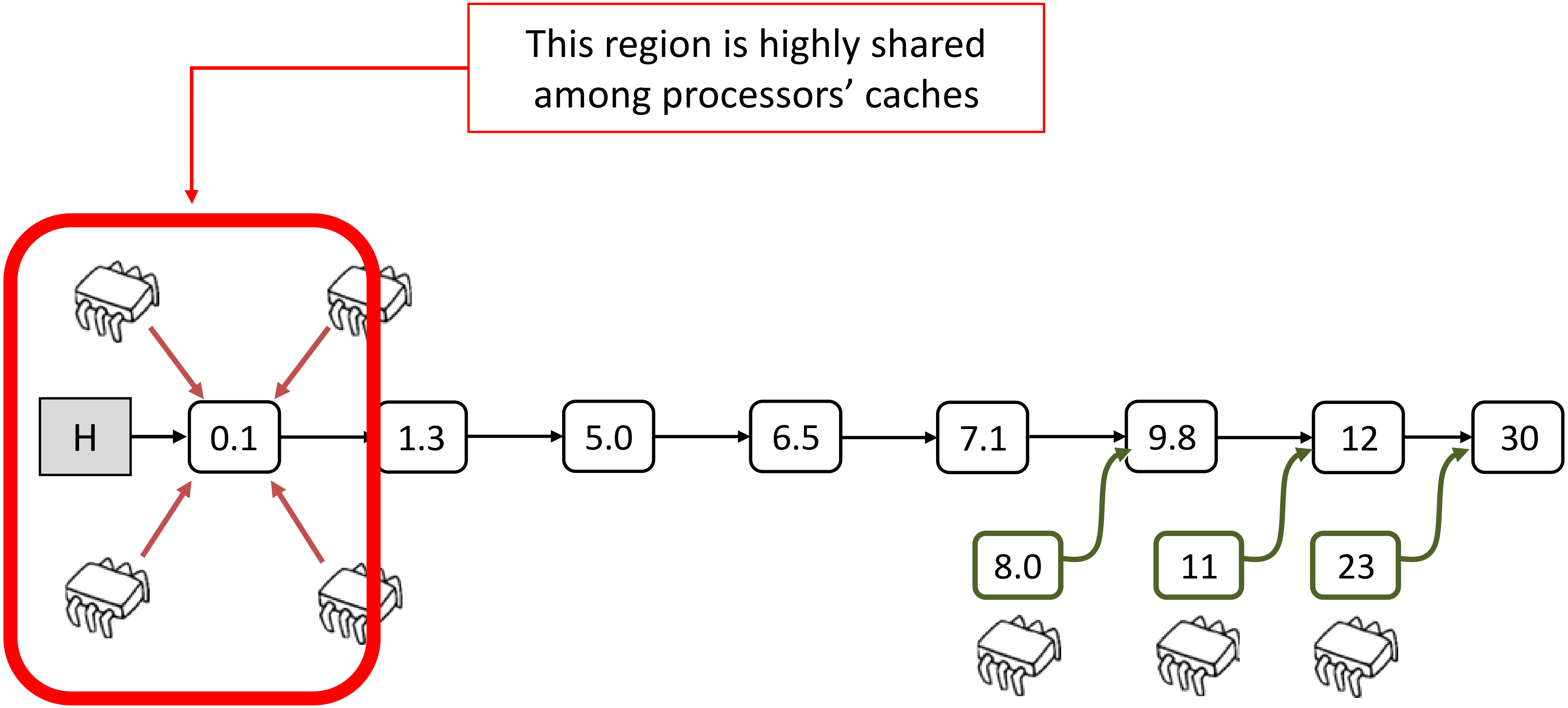
Priority queue – Attempt 1



Scalability collapse

Priority queues: an inherently “sequential” semantic

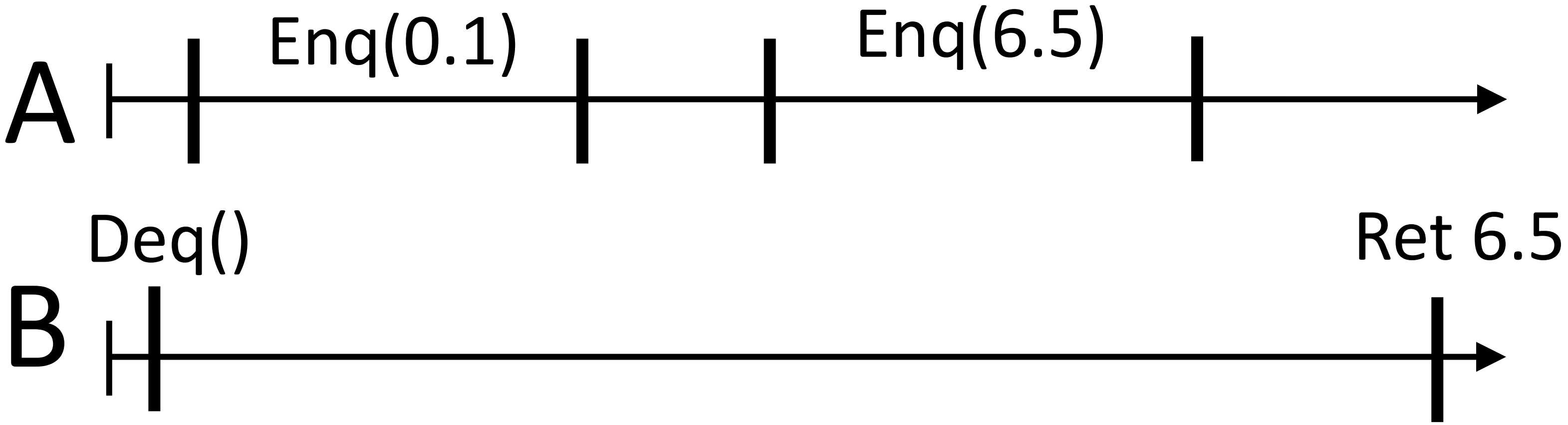
- Enqueue offers a high level of disjoint access parallelism
- Dequeues are prone to conflicts

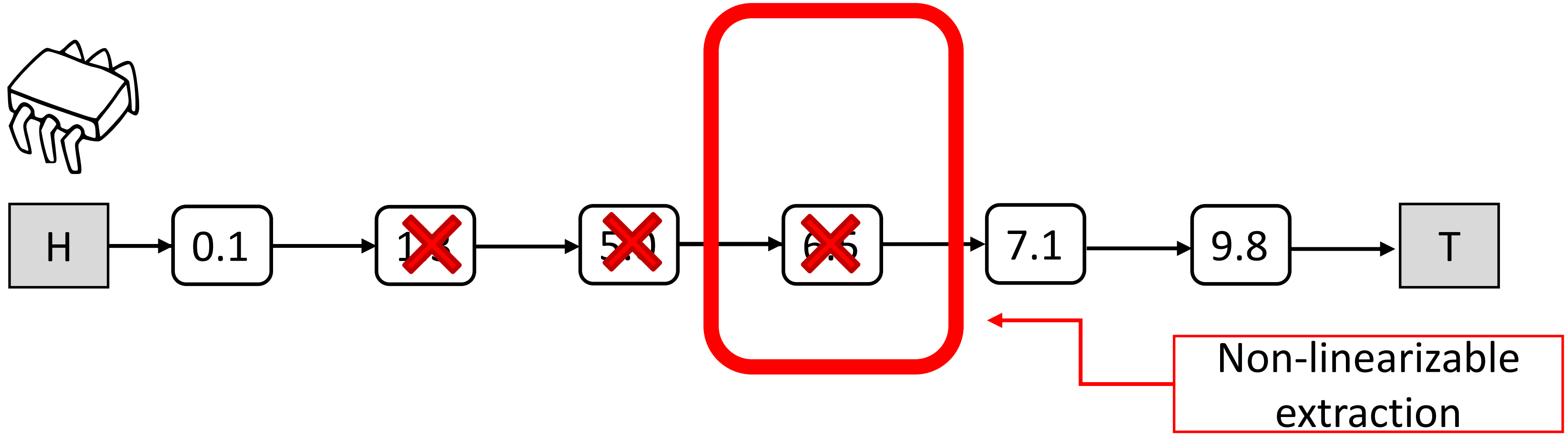


This region is highly shared among processors’ caches

Lazy deletion within priority queues

- If we use lazy deletion "as is", we might obtain non-linearizable extractions



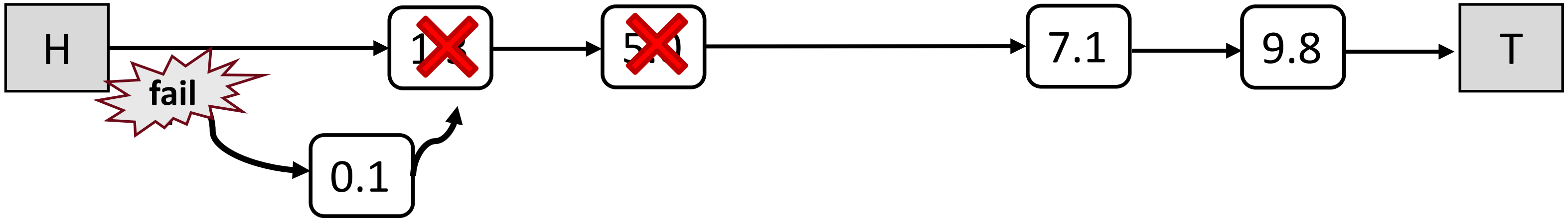


Non-linearizable extraction

Correct lazy deletion within priority queues

- To implement correct extractions with lazy deletions there are two main approaches

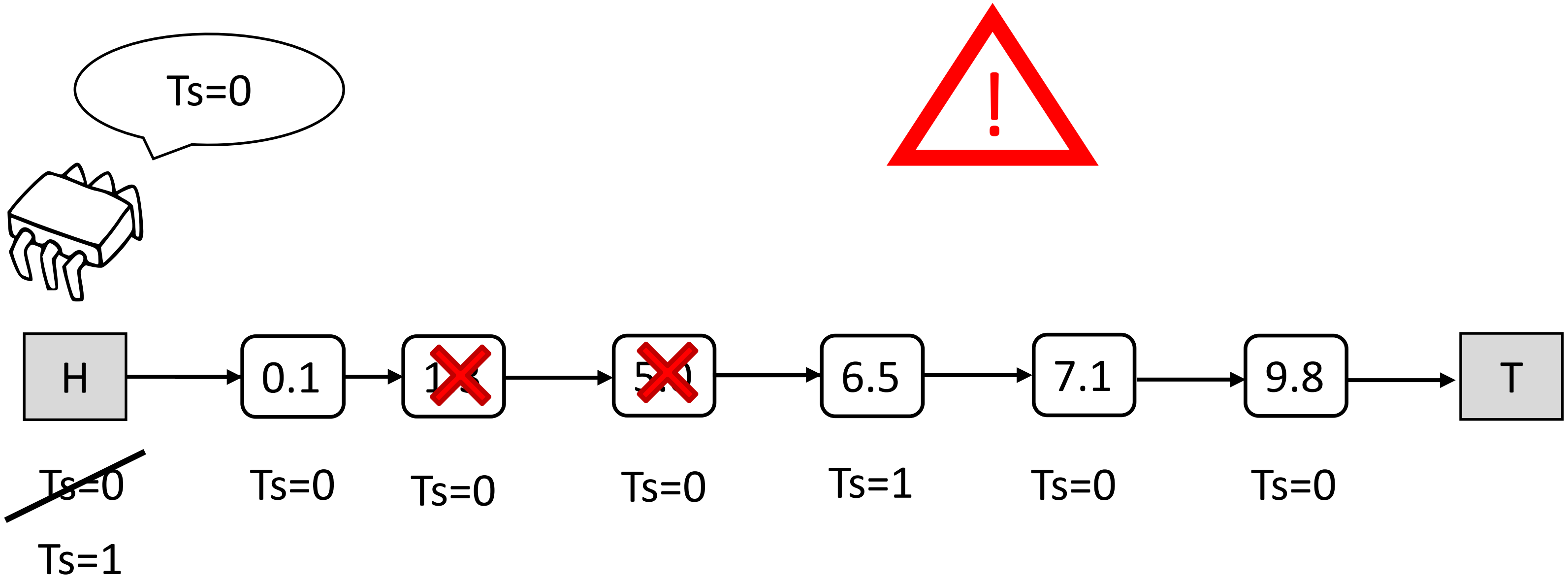
1. Move the logical mark of a node in the field “**`next`**” of its predecessor



Correct lazy deletion within priority queues

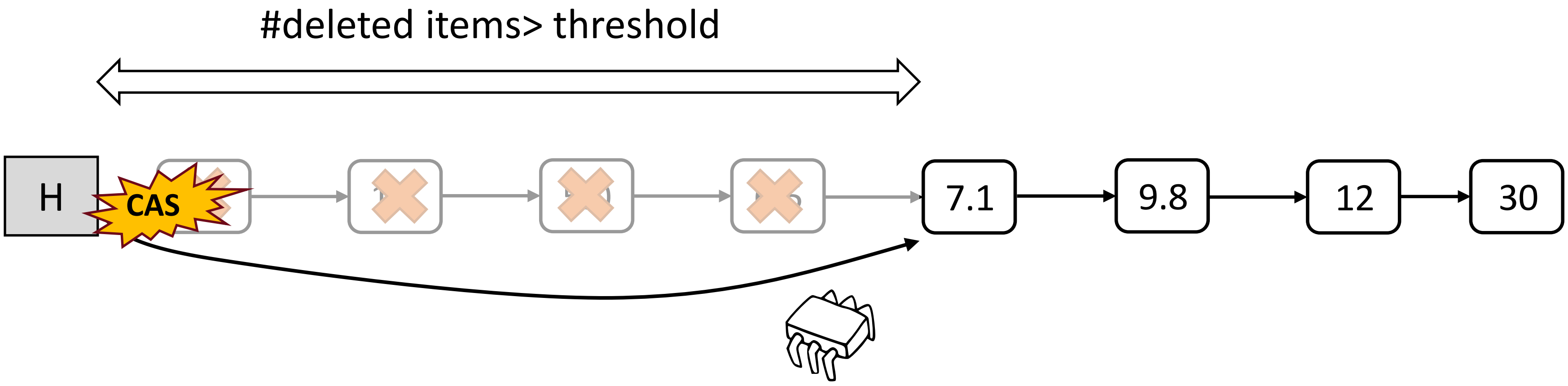
- To implement correct extractions with lazy deletions there are two main approaches

2. Use logical timestamps:
    - incremented each time a new minimum has been inserted
    - extract item compatible with the timestamp read at the beginning

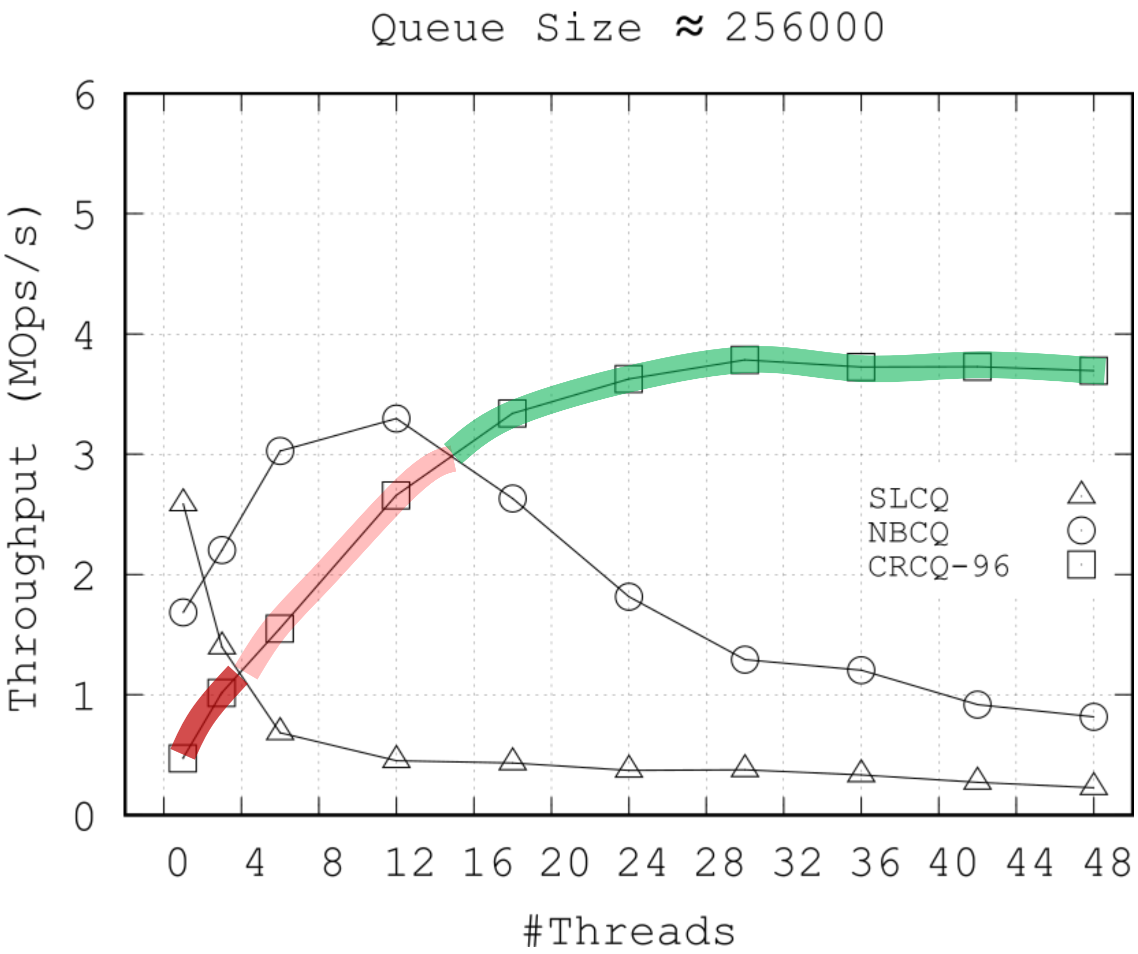


PQ – Attempt 2 - Introducing Conflict Resiliency

- Lazy deletion
- Skip logically deleted items ⇒ IT INCREASES THE NUMBER OF STEPS
- Periodic Housekeeping ⇒ EXPENSIVE IN TERMS OF IMPACT ON CACHE



Priority queue – Attempt 2



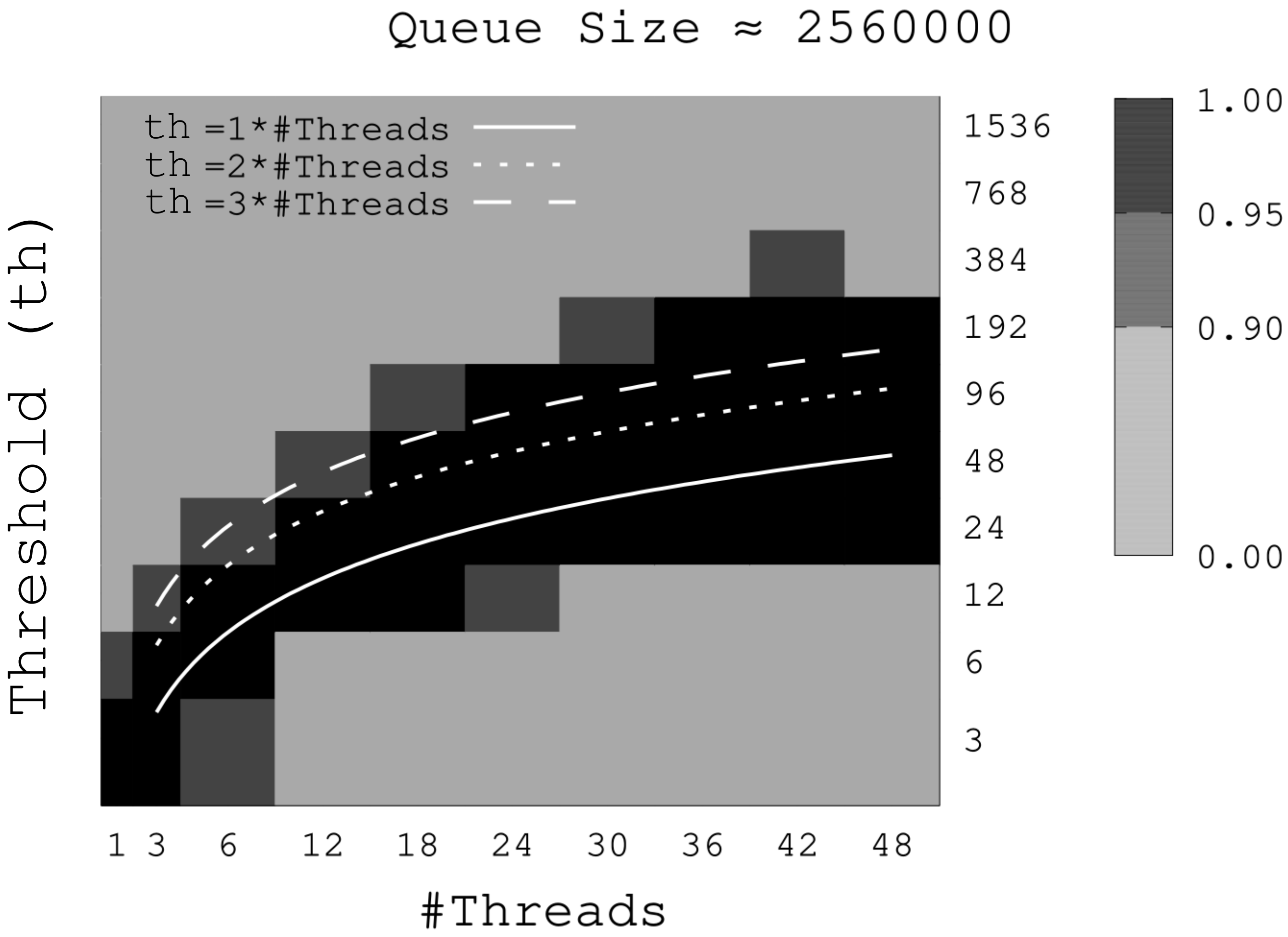
On the conflict resiliency trade off

- The number of steps per dequeue and costs of housekeeping are **dependent**:

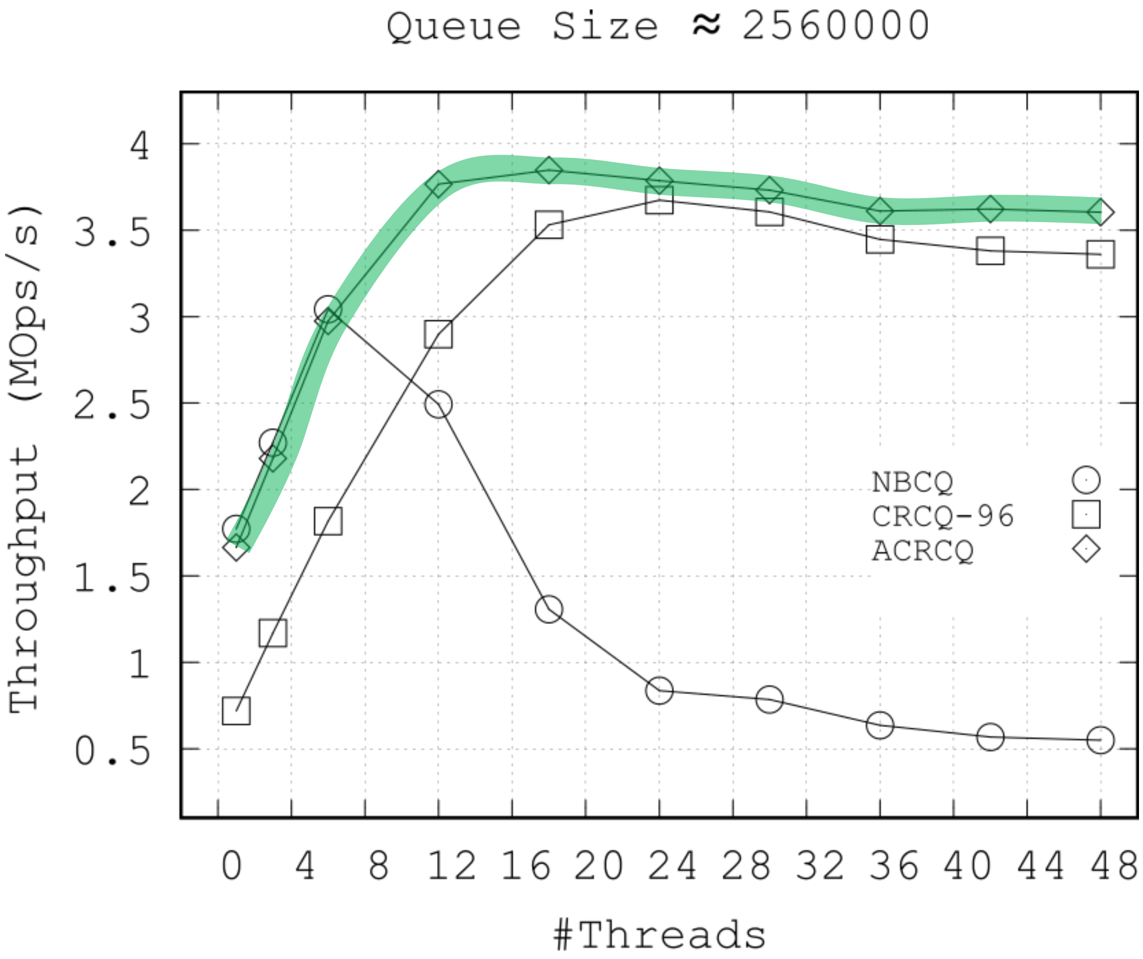
↑ THRESHOLD ⇒ ↑ READ LATENCY ☹ and ↓ RMW IMPACT ☺

↓ THRESHOLD ⇒ ↓ READ LATENCY ☺ and ↑ RMW IMPACT ☹

Conflict resiliency trade offs



Priority queues – Attempt 3



Open challenges

How to achieve scalability for priority queues?

- NO ANSWER for correct priority queue
- The research moved on looking for RELAXED SEMANTICS for priority queues
    - Enable scalability for extractions by removing an item which is “near” the minimum
- Explore orthogonal approaches by guaranteeing RELAXED CORRECTNESS CONDITIONS
    - K-linearizability
    - Quasi-linearizabilty
    - Quiescent consistency
    - Sequential consistency?
- Explore new hardware capabilities (e.g. HTM)

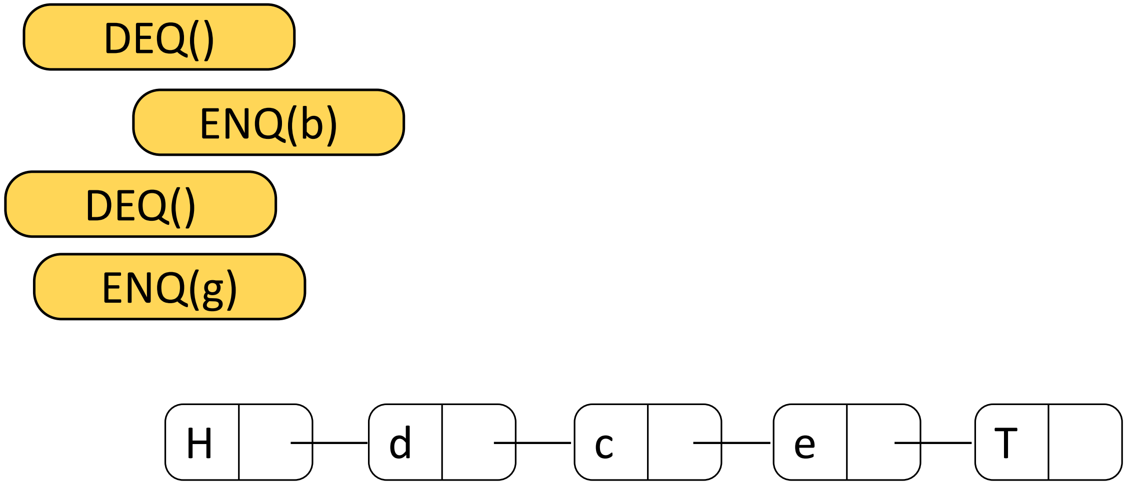
Why linearizable non-blocking algorithms?

- Performance is a good reason, but not the unique one
- The composition of linearizable algorithm is still linearizable
- Blocking algorithms (and their composition) might suffer from deadlocks, priority inversions and convoying
- The composition of non-blocking algorithms is non-blocking as a whole (progress property of individual algorithm might be hampered)
- Libraries should implement their algorithms in a non-blocking linearizable fashion
  - E.g. Java implements non-blocking concurrent data structure

Concurrent Data Structures: **FIFO queues**

FIFO queue implementation

- Queue methods:
  - `enqueue(v)`
  - `dequeue()`
- Implemented as a linked list



DEQ()

ENQ(b)

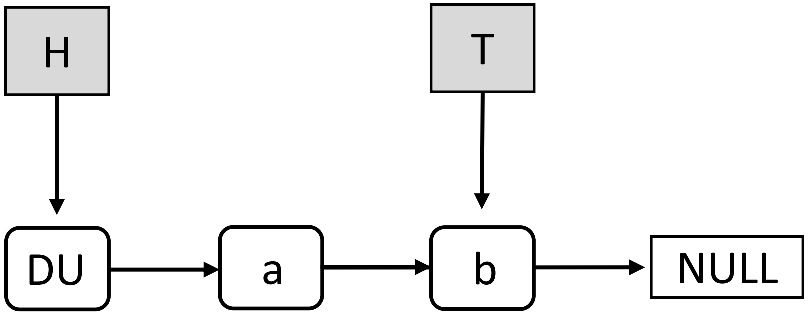
DEQ()

ENQ(g)

H d c e T

FIFO queue implementation

- Slightly different
- One dummy node, two pointers to access the data structure:
  - Head: points ALWAYS to a DUMMY node item
  - Tail: SHOULD point to the youngest item



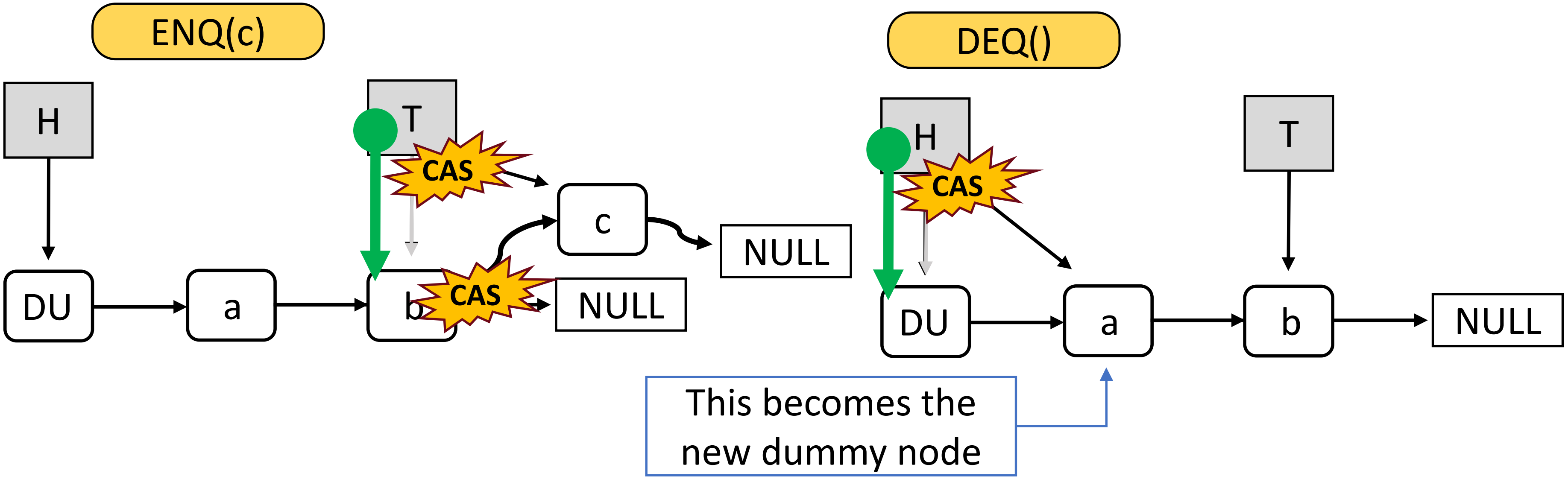
FIFO queue implementation

- Insert:

1. Get node pointed by tail
2. Scan until next is NULL
3. Try to insert with CAS
4. If KO goto 1
5. Else try to update Tail

- Dequeue:

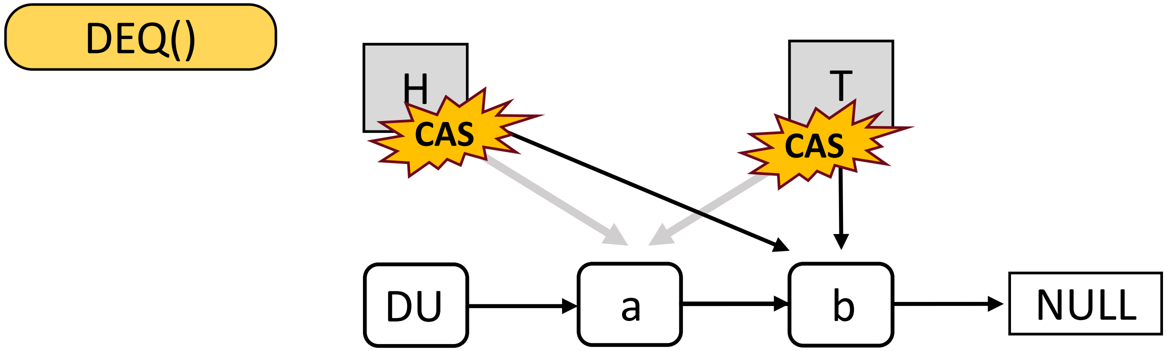
1. Get node pointed by head
2. Try to update head with its next
3. If KO goto 1



This becomes the new dummy node

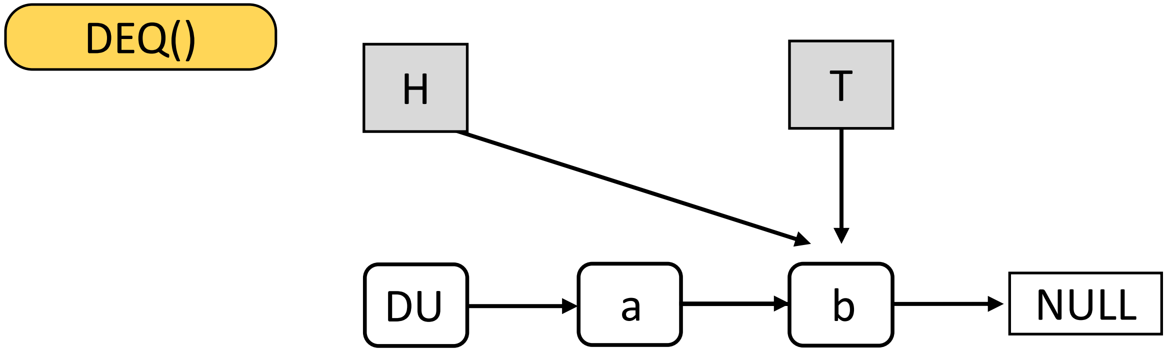
The whole story

- Since the insertion of a new item and the tail update are two separate RMW they might be inconsistent
- Also dequeuers might need to update tail before updating head
- This ensures that TAIL cannot go behind HEAD



Emptiness condition

- There is a NULL node after the one pointed by HEAD



Recommended readings

SET:

- *A pragmatic implementation of non-blocking linked-lists*
  T. L. Harris, International Symposium on Distributed Computing, 2001.
- *Fraser, K.: Practical Lock-Freedom. PhD thesis,*

STACK:

- *Systems programming: Coping with parallelism*
  R K Treiber, IBM Almaden Research Center, 1986.
- *A Scalable Lock-free Stack Algorithm*
  D. Hendler et al., SPAA’04.

PRIORITY QUEUE:

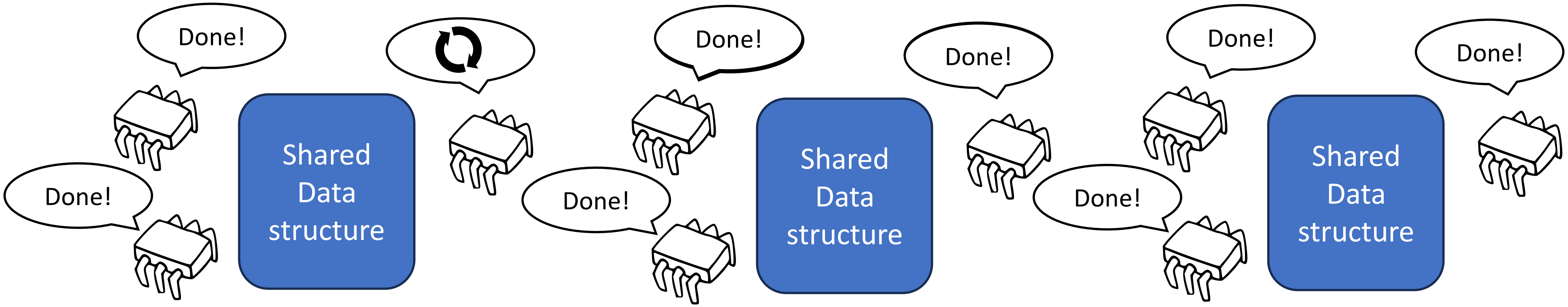
- *A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention*
  J. Lindén et al., ICPDS’2013
- *A Conflict-Resilient Lock-Free Calendar Queue for Scalable Share-Everything PDES Platforms*
  R. Marotta et al., PADS’2017
- *A Conflict-Resilient Lock-Free Linearizable Calendar Queue*
  R. Marotta et al., ACM TOPC (just accepted)

FIFO:

- *Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms*
  M. M. Michael et al., PODC '96

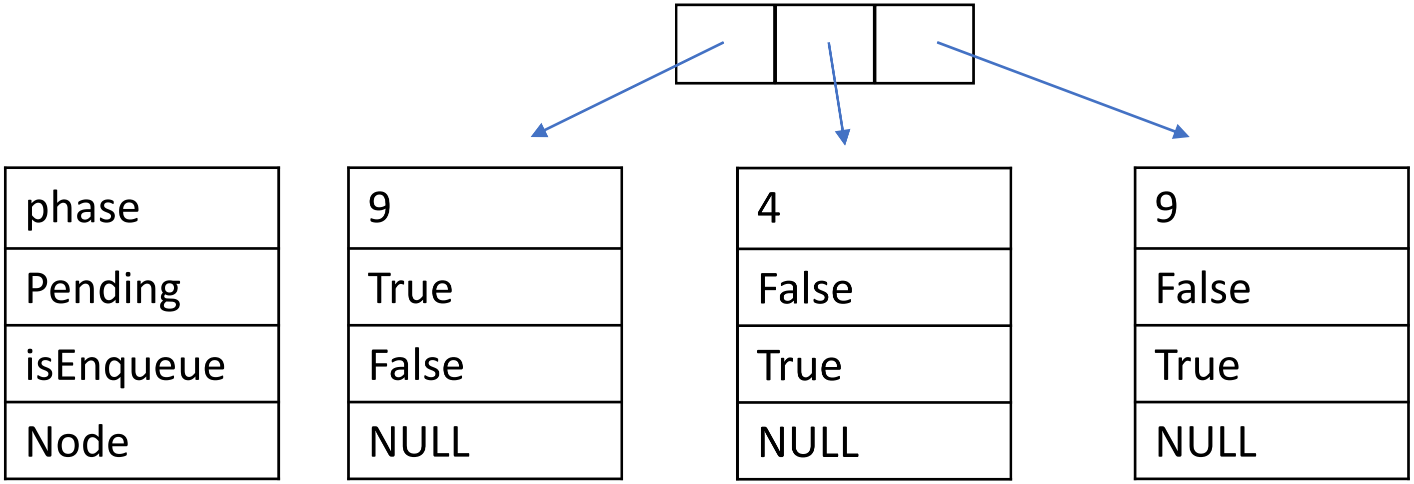
Wait-free FIFO queue

- What about a wait-free queue?
- Wait-free means that all method invocations are guaranteed to complete
- Can we modify the lock-free FIFO queue to achieve this?
- Lock-free means that some thread might starve
- If before starting any new operation we complete a pending operation, all method invocation complete eventually



Wait-free FIFO queue

- We need to be aware of pending calls



phase	9	4	9
Pending	True	False	False
isEnqueue	False	True	True
Node	NULL	NULL	NULL

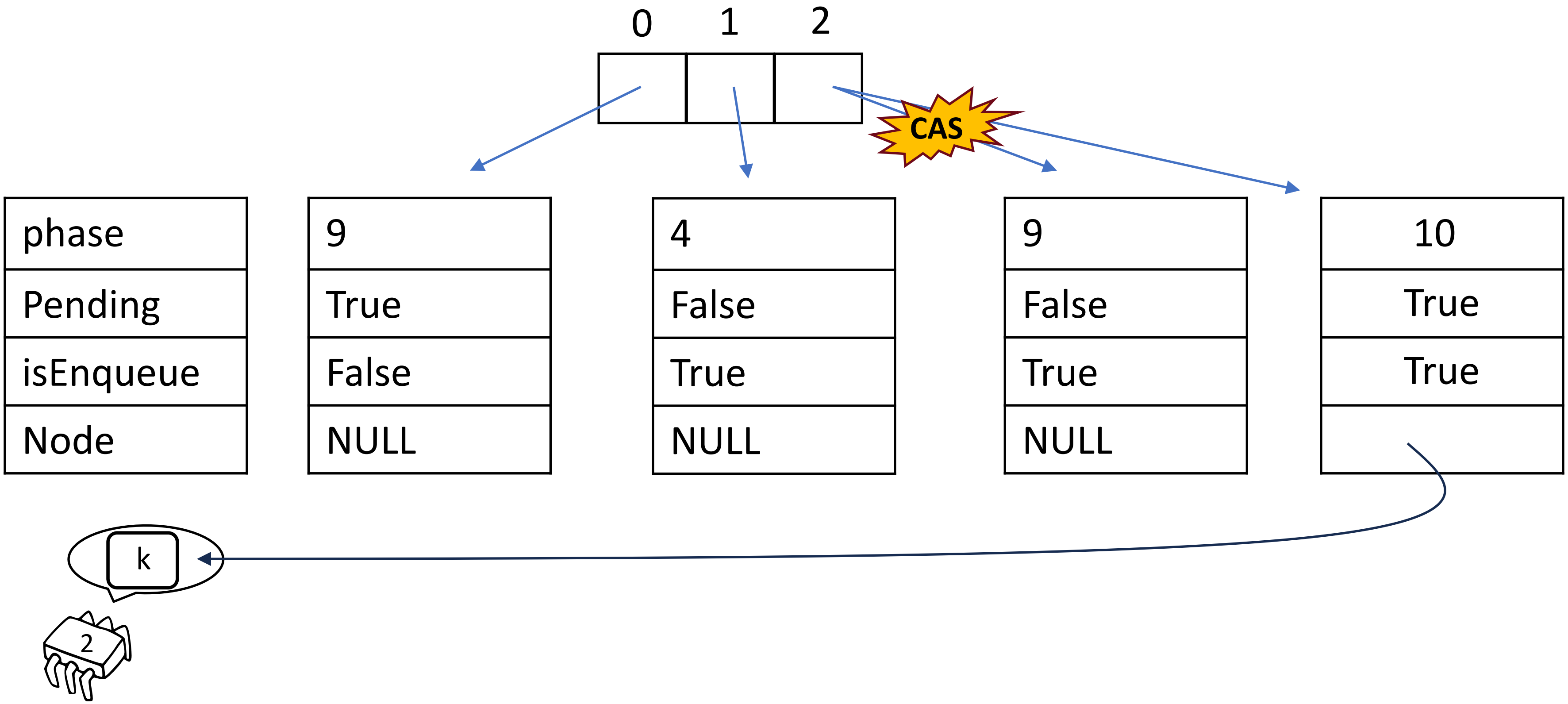
- Split operations on the linked list into 2 steps:
  1. Modify nodes for enqueue/dequeue (main step)
  2. Modify head/tail pointers (finishing step)

Wait-free FIFO queue

- Enqueue/Dequeue structure
  1. Publish op record
  2. Get the set **S** of all pending ops whose record has been previously or concurrently published
  3. Help any operation in **S**
  4. Do a finishing step

Wait-free FIFO queue

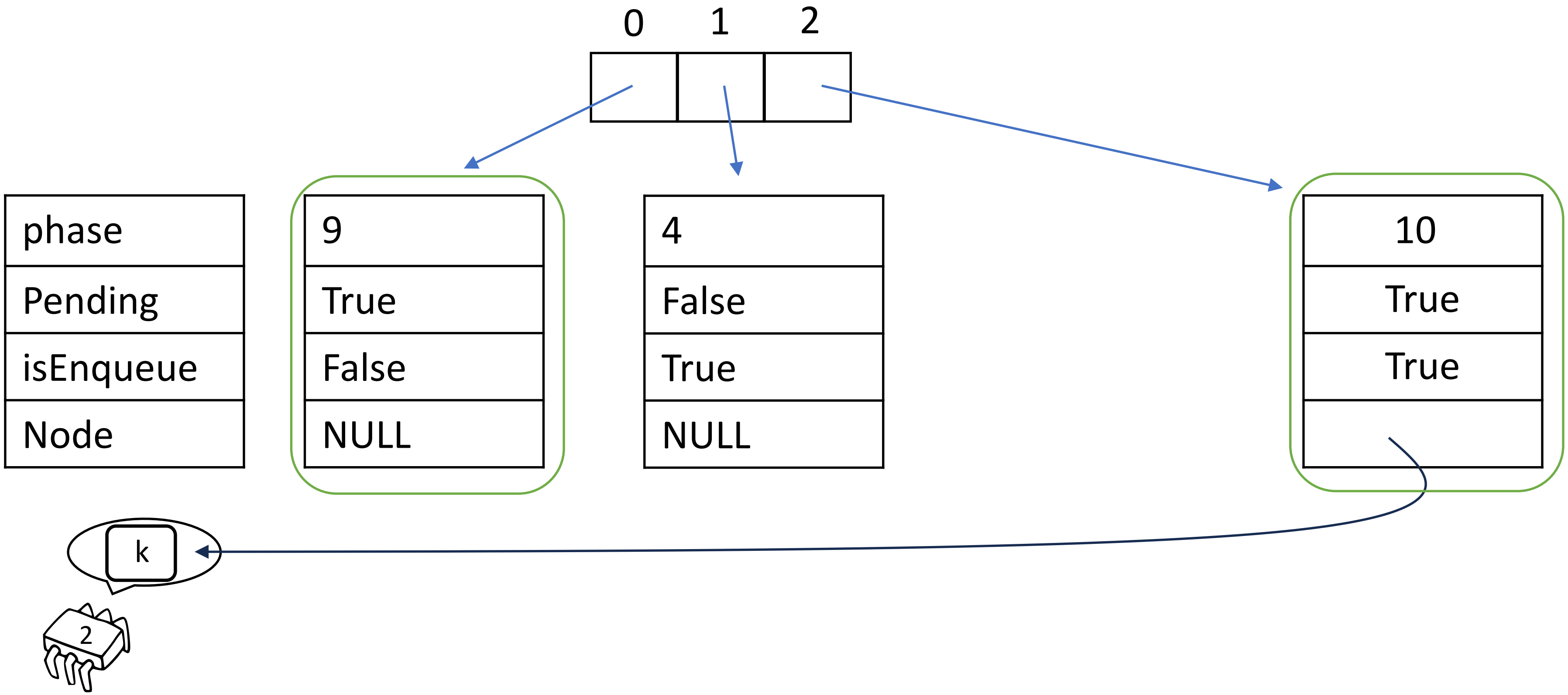
- Enqueue/Dequeue structure
  1. Publish op record



phase	9	4	9	10
Pending	True	False	False	True
isEnqueue	False	True	True	True
Node	NULL	NULL	NULL	

Wait-free FIFO queue

- Enqueue/Dequeue structure
  2. Get the set **S** of all pending ops whose record has been previously or concurrently published

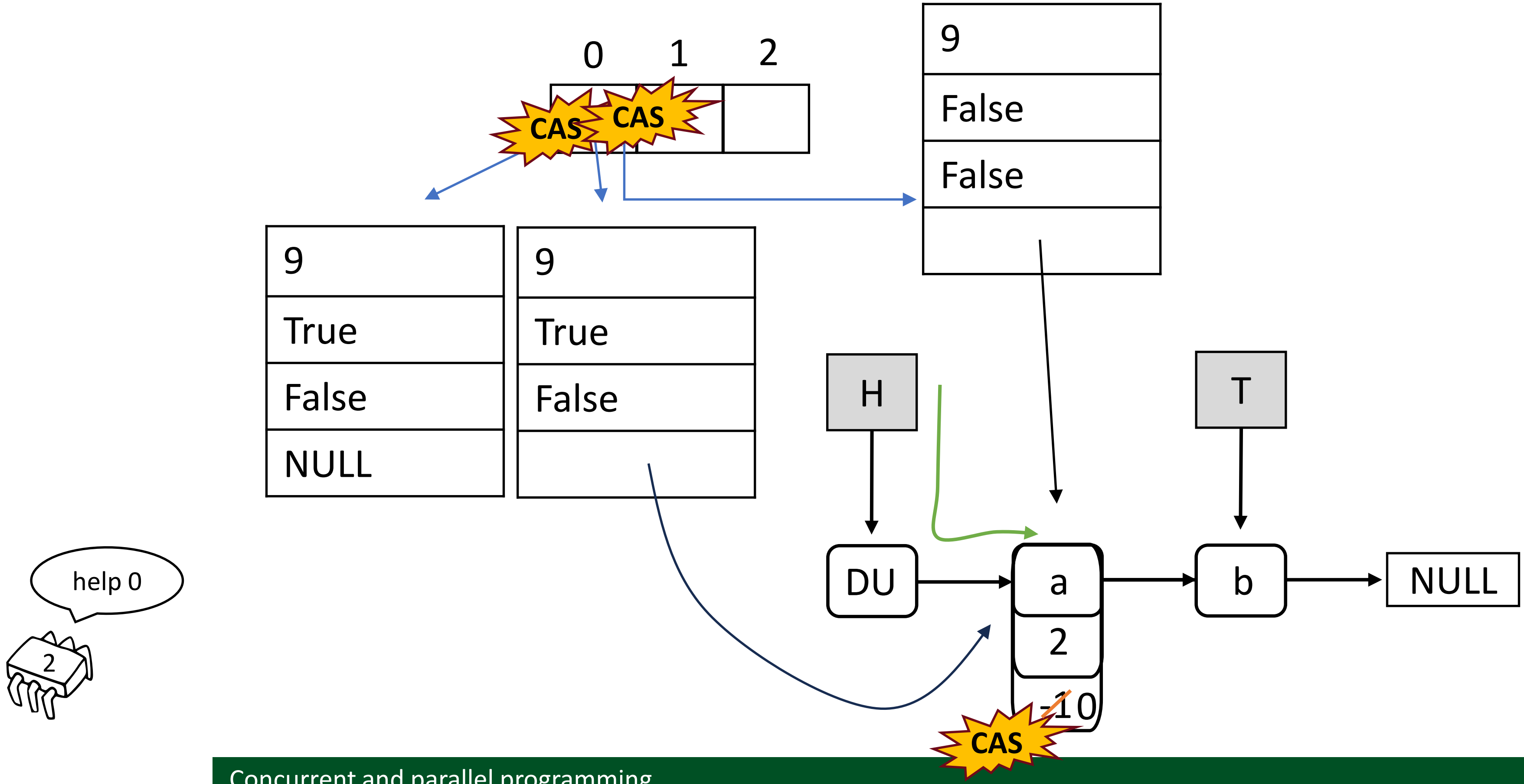


0 1 2

phase	9	4	10
Pending	True	False	True
isEnqueue	False	True	True
Node	NULL	NULL	

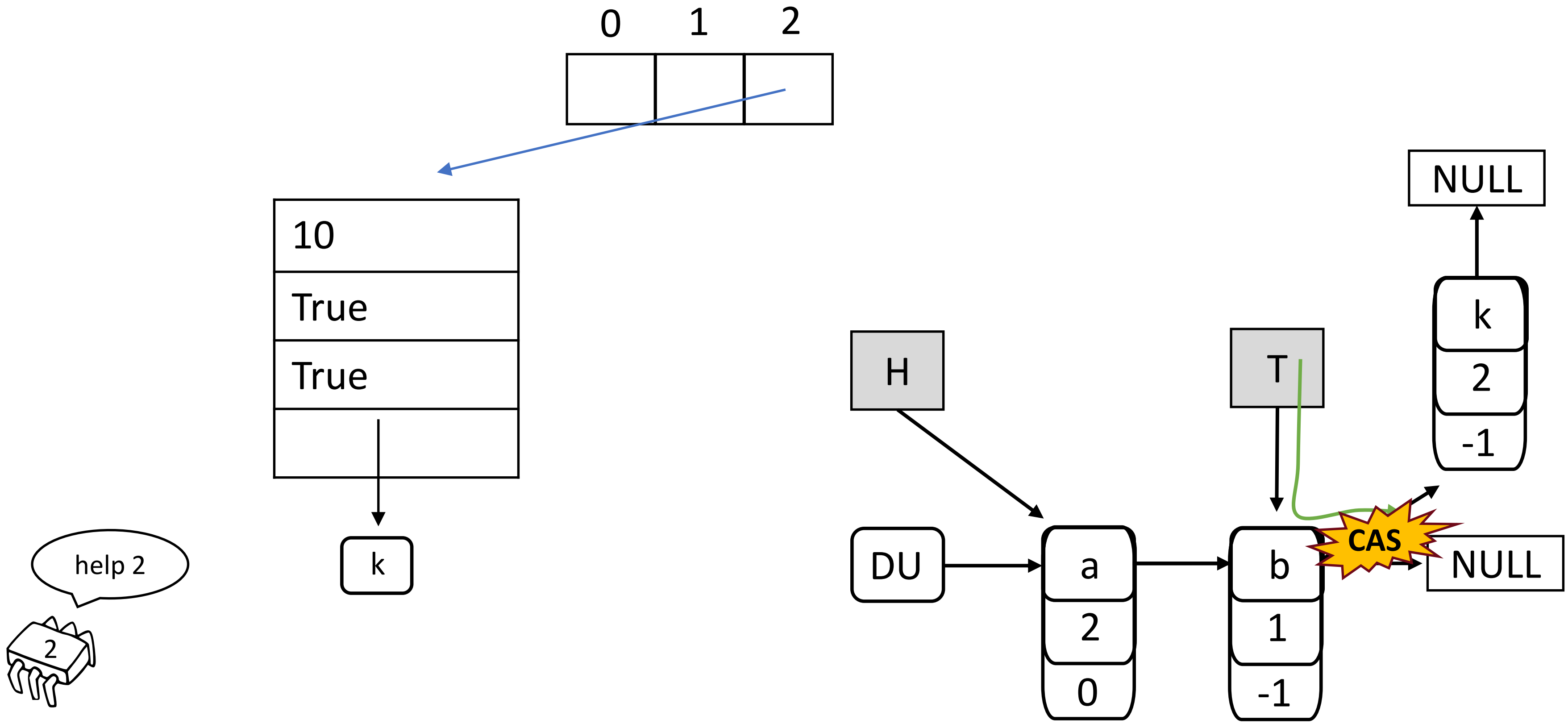
Wait-free FIFO queue

- Enqueue/Dequeue structure
  3. Help any operation in **S** (dequeue)
     a. Main step
     b. Finishing step



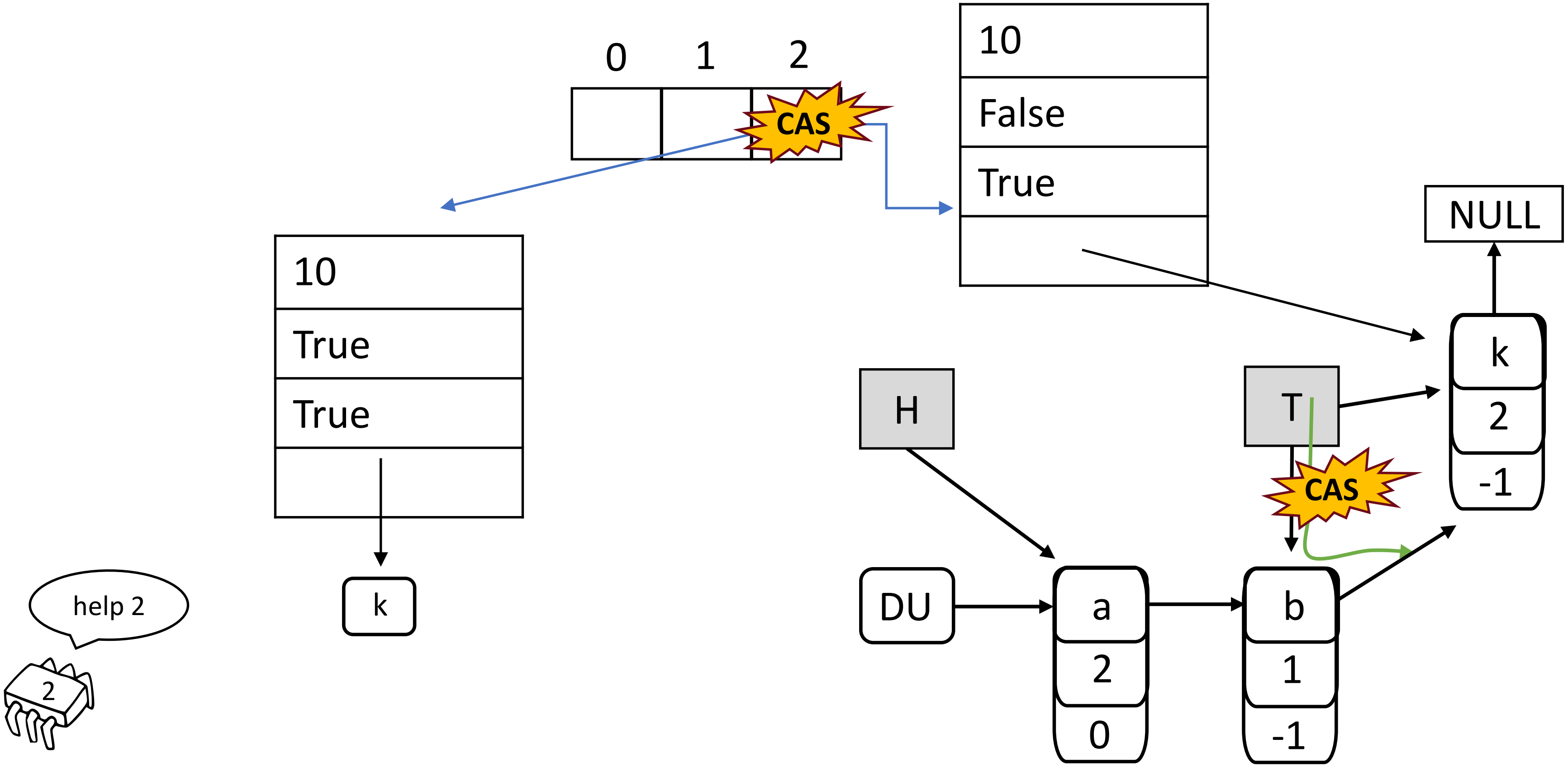
Wait-free FIFO queue

- Enqueue/Dequeue structure
  3. Help any operation in **S** (enqueue)
     a. Main step
     b. Finishing step



Wait-free FIFO queue

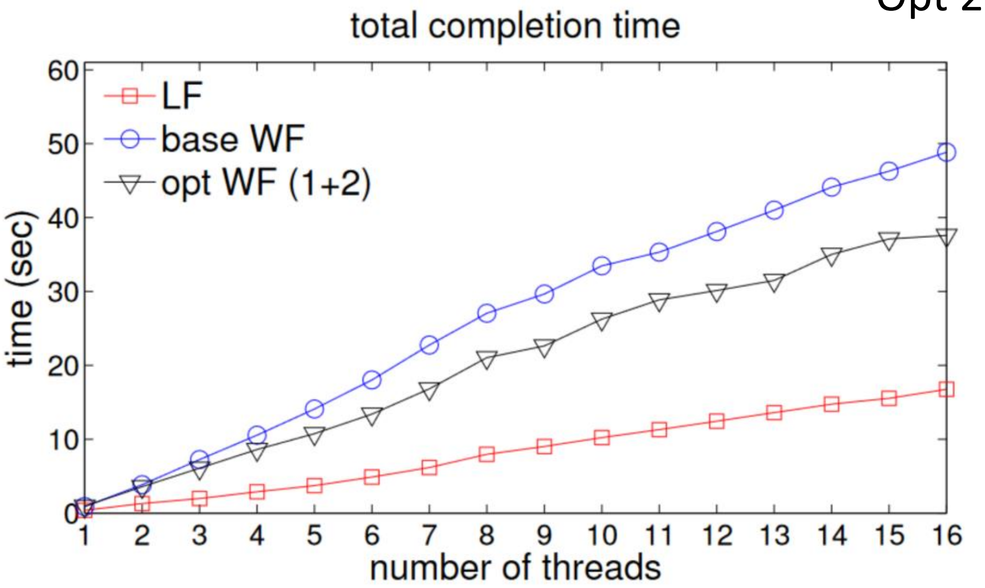
- Enqueue/Dequeue structure
  3. Help any operation in **S** (enqueue)
     a. Main step
     b. Finishing step



Wait-free FIFO queue

- Enqueue/Dequeue structure
  1. Publish op record
  2. Get the set **S** of all pending ops whose record has been previously or concurrently published
  3. Help any operation in **S**
  4. Do a finishing step

Opt 1: help only one pending op
Opt 2: use FAD to get phase num.



Fast Wait-free FIFO queue

- Try with lock-free approach
- If starving, back-off to wait-free implementation