Programmazione concorrente

Laurea Magistrale in Ingegneria Informatica
Università Tor Vergata
Docente: Romolo Marotta

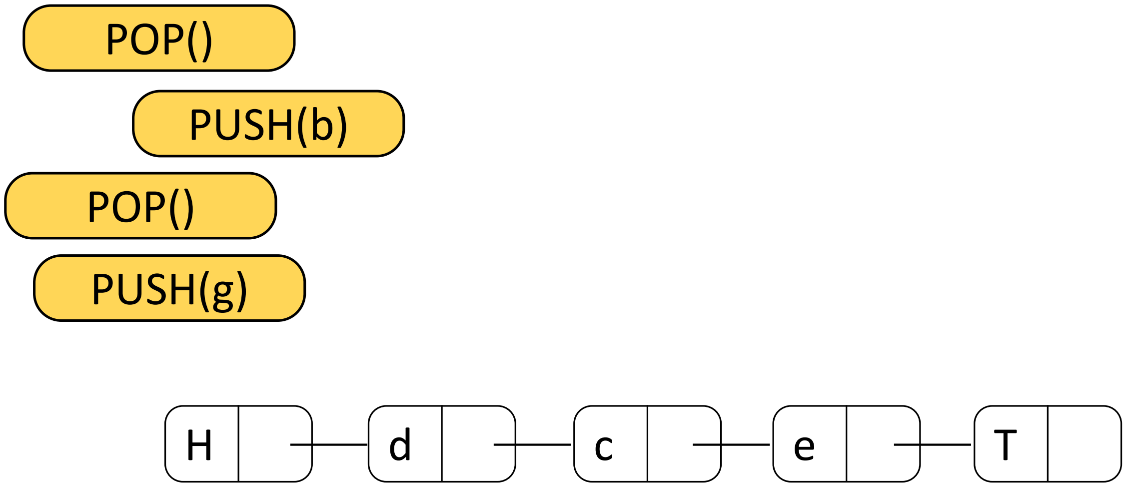
Concurrent data structures

1. Stack
2. Set
3. Priority queues
4. FIFO queues
5. MRSW Registers

Concurrent Data Structures: Stacks

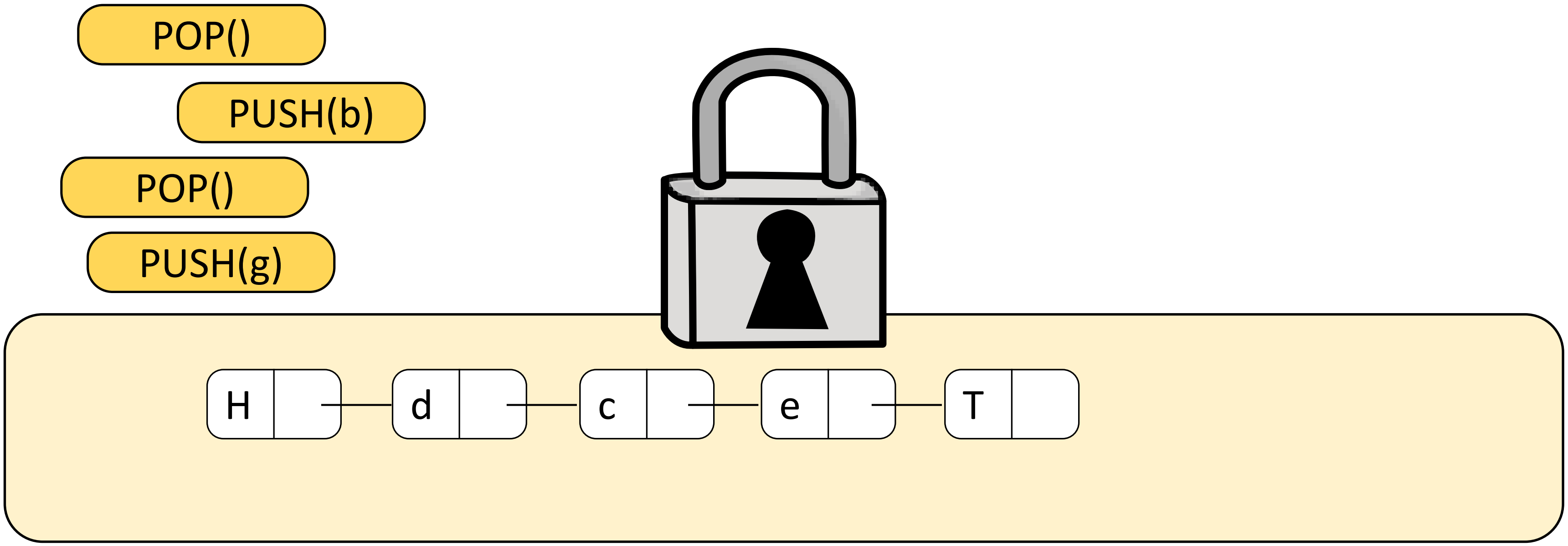
Stack implementation

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



Concurrent stack implementations

- Resort to a global lock



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

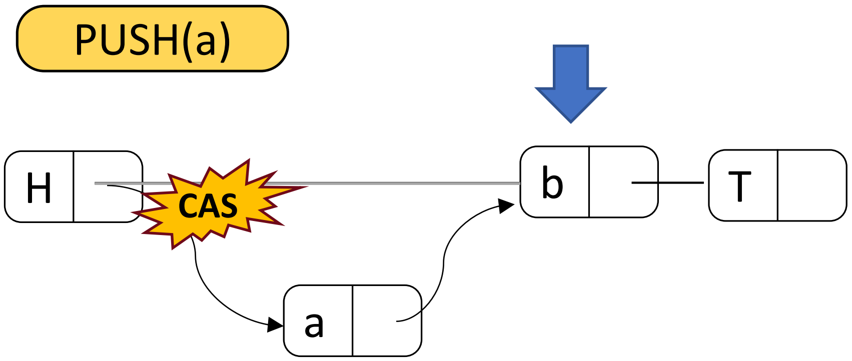
Attempt 1

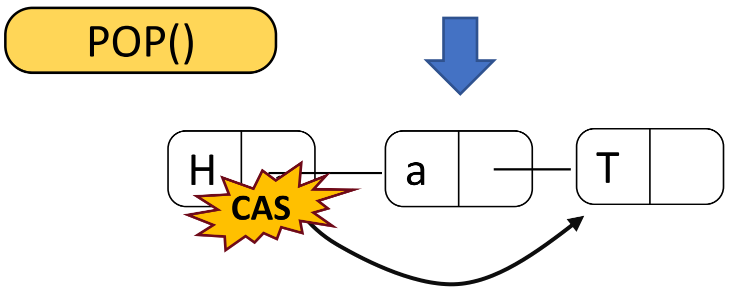
Push:

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

Delete:

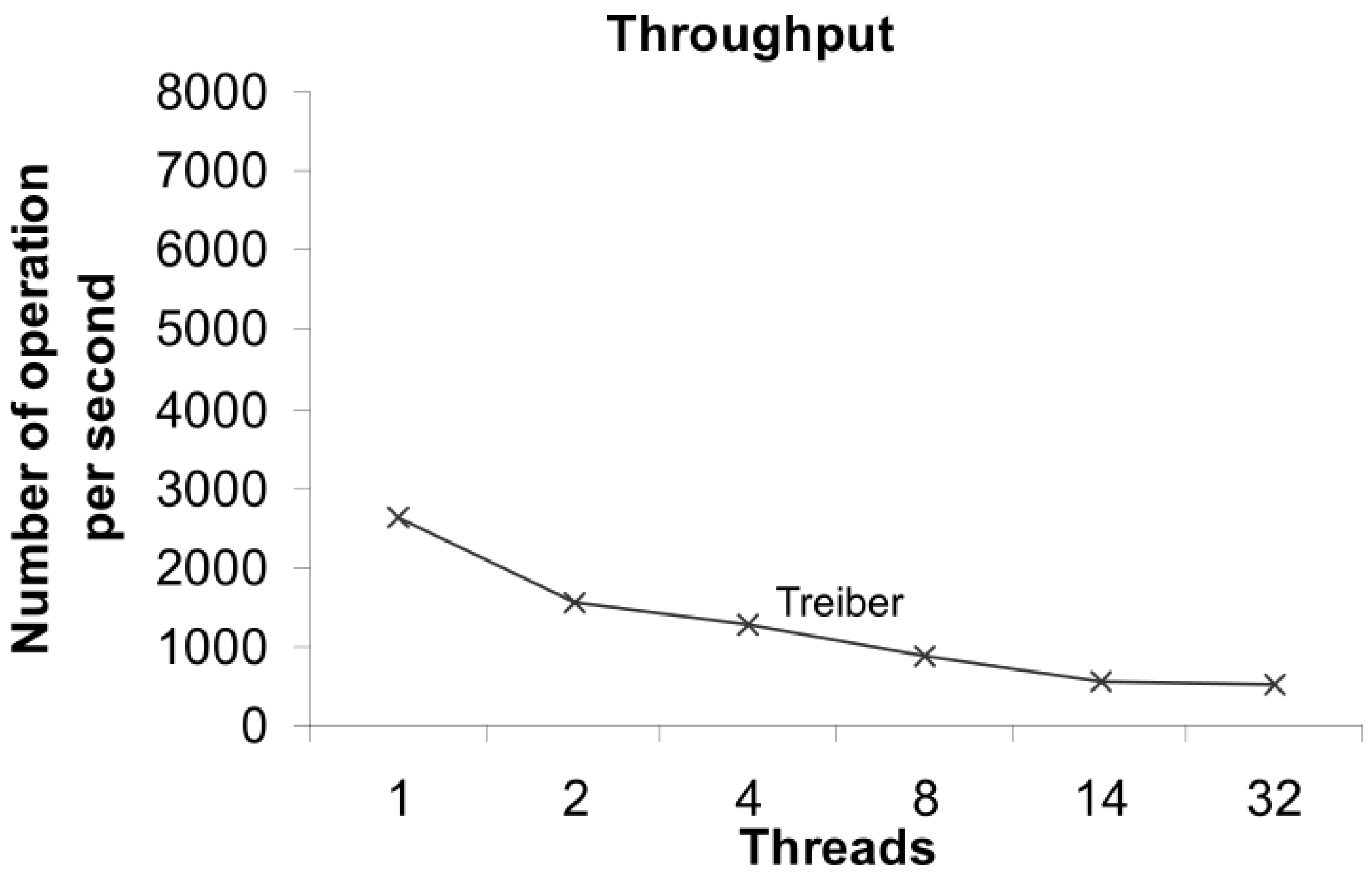
1. Get head next
2. Disconnect the item with a CAS
3. If CAS fails, restart





- Is it scalable?

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



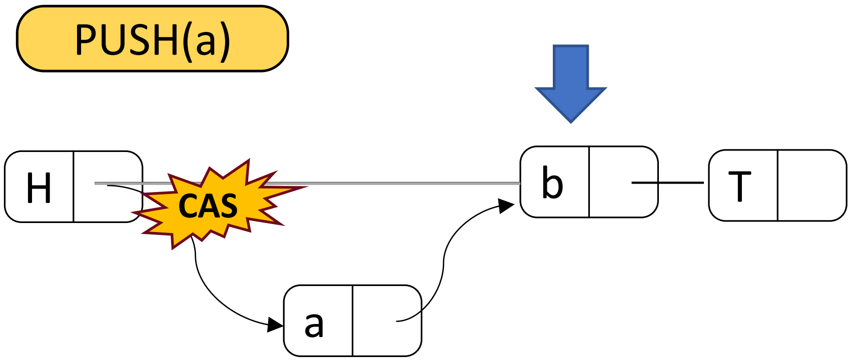
Non-blocking stack – Attempt 2 [Treiber+BO]

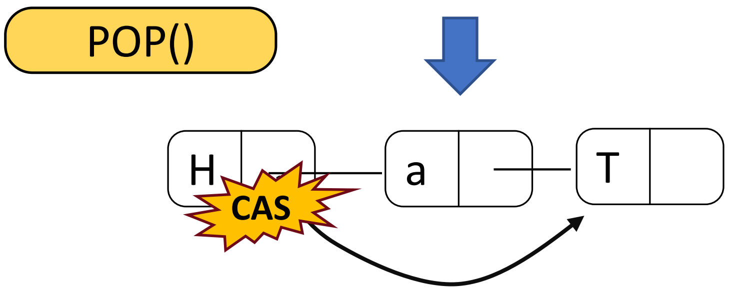
Push:

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

Delete:

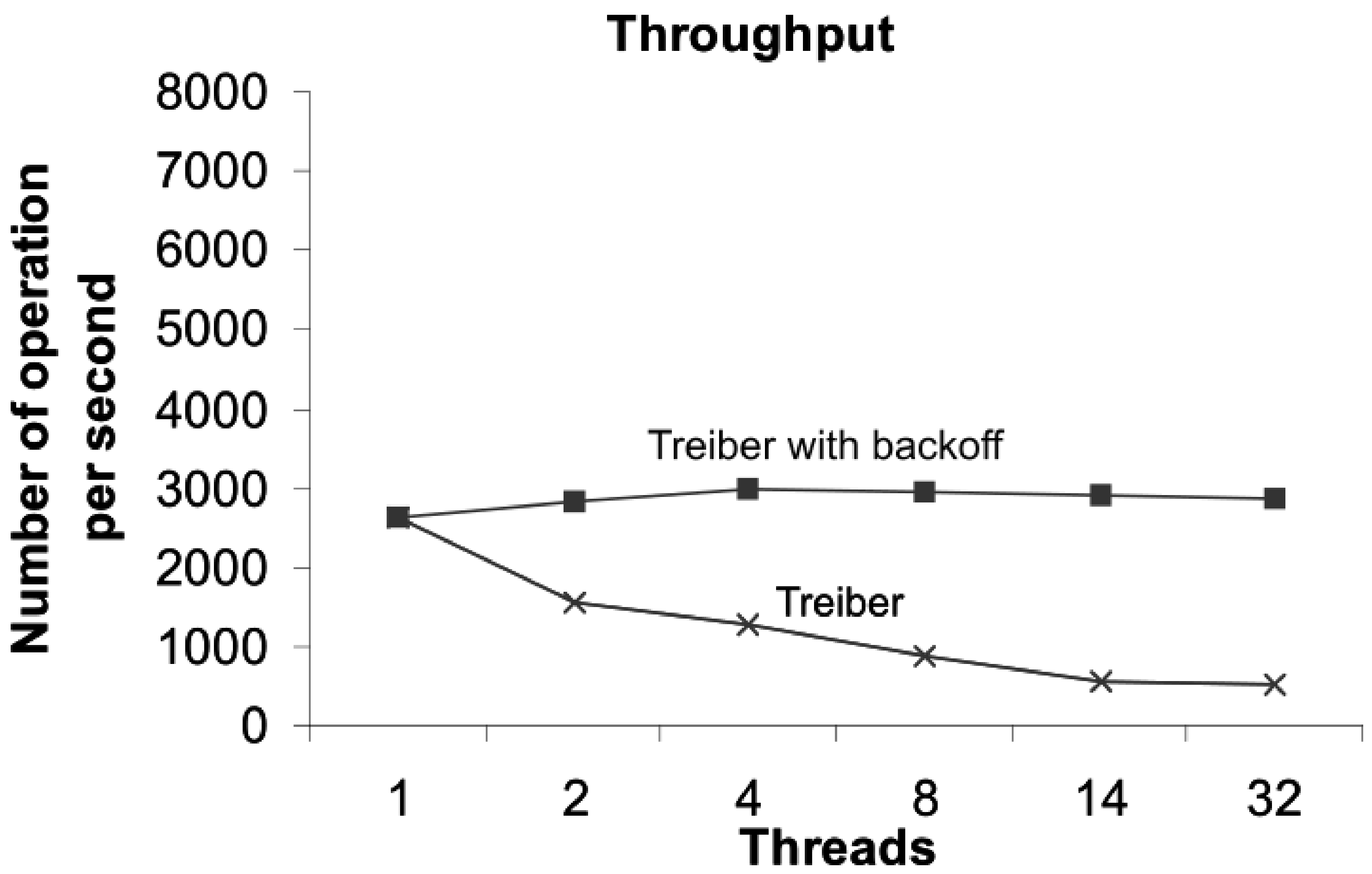
1. Get head next
2. Disconnect the item with a CAS
3. If CAS fails, ~~restart~~ backoff and restart





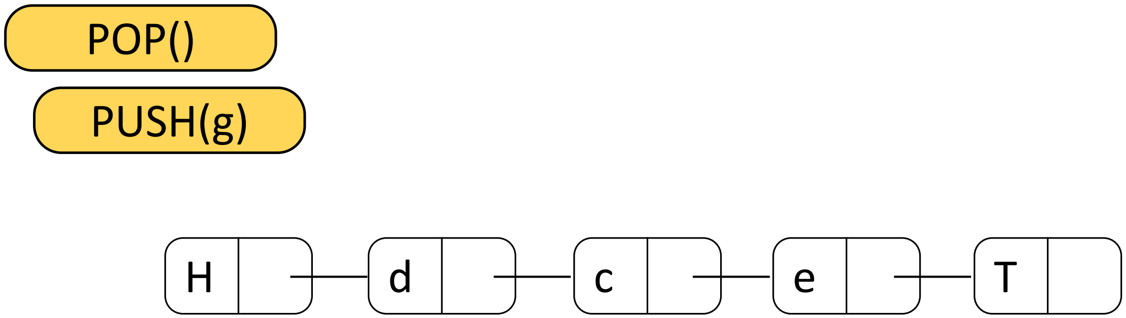
- Is it scalable?

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



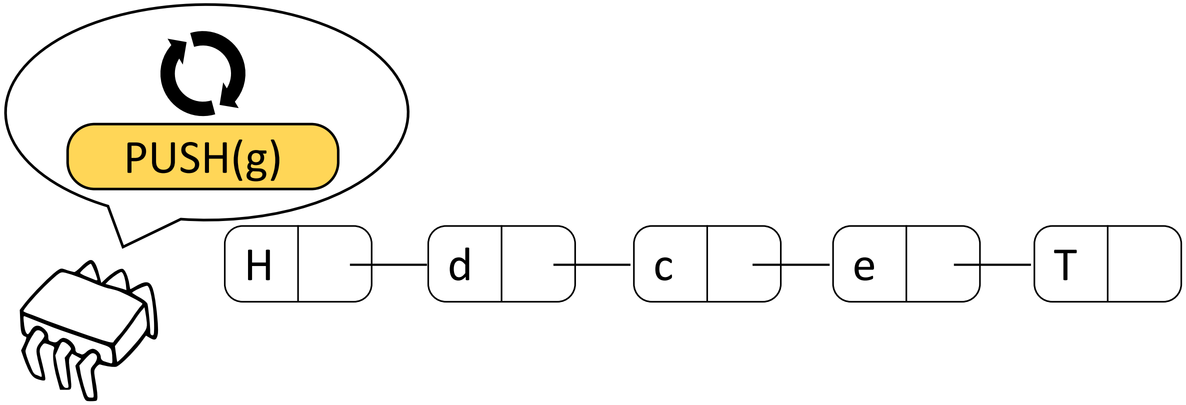
Concurrent stack implementations

- 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



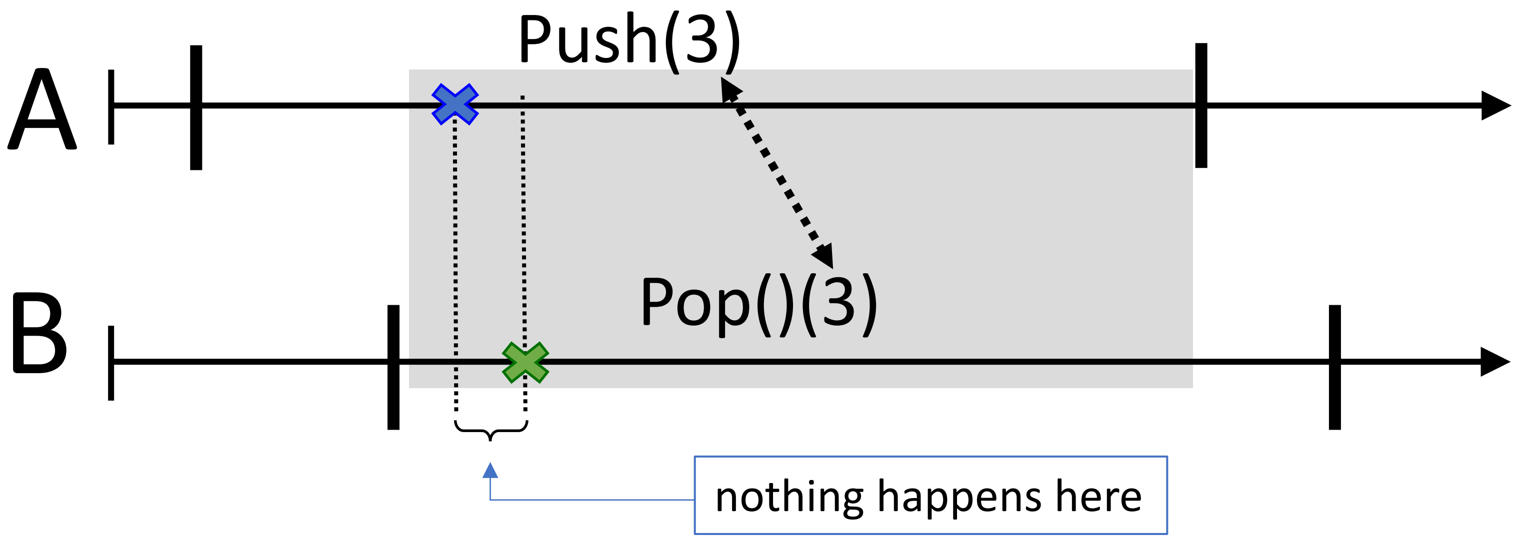
Non-blocking stack – Attempt 3

- How to take advantage of back-off times?



Observation

- Concurrent matching push/pop pairs are always linearizable



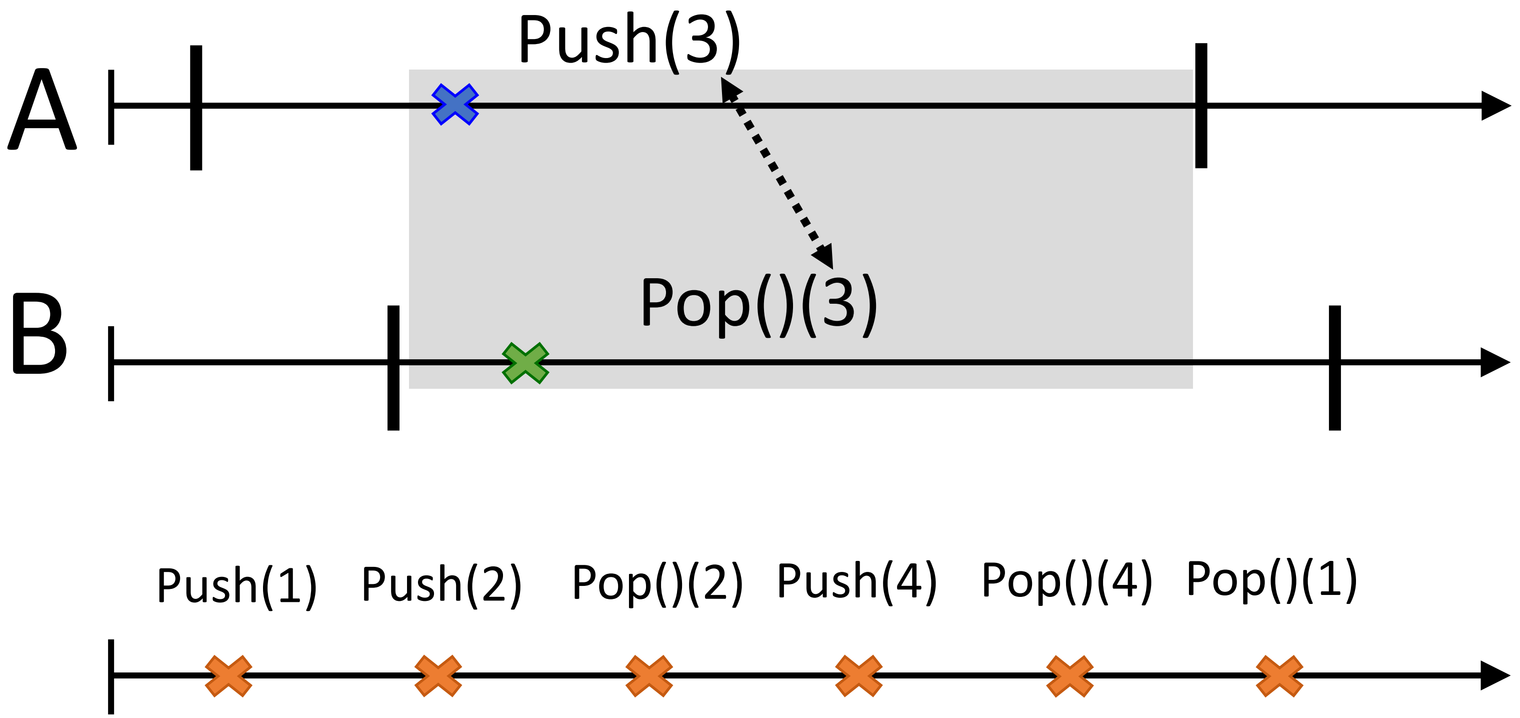
- A push A and a pop B are:
  - concurrent to each other
  - B returns the item inserted by A

⇒ 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)

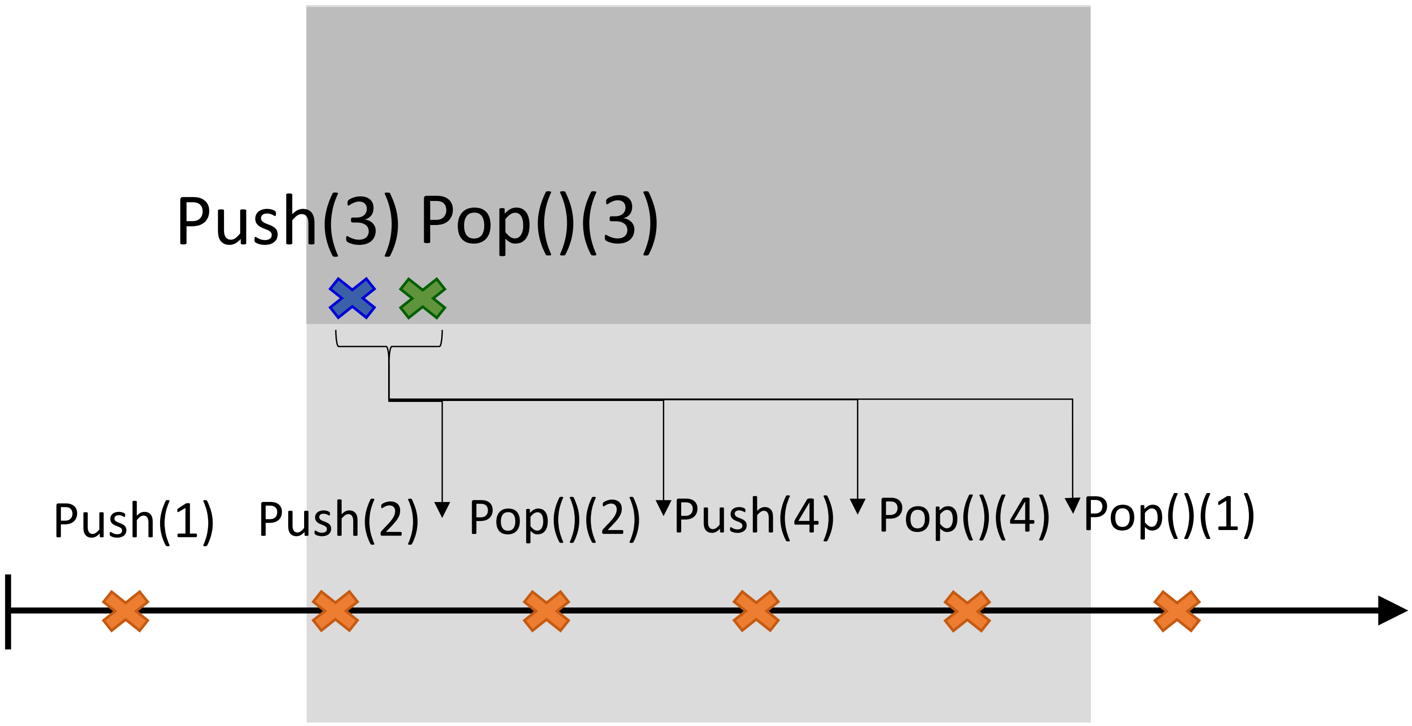
Observation

- Concurrent matching push/pop pairs are always linearizable



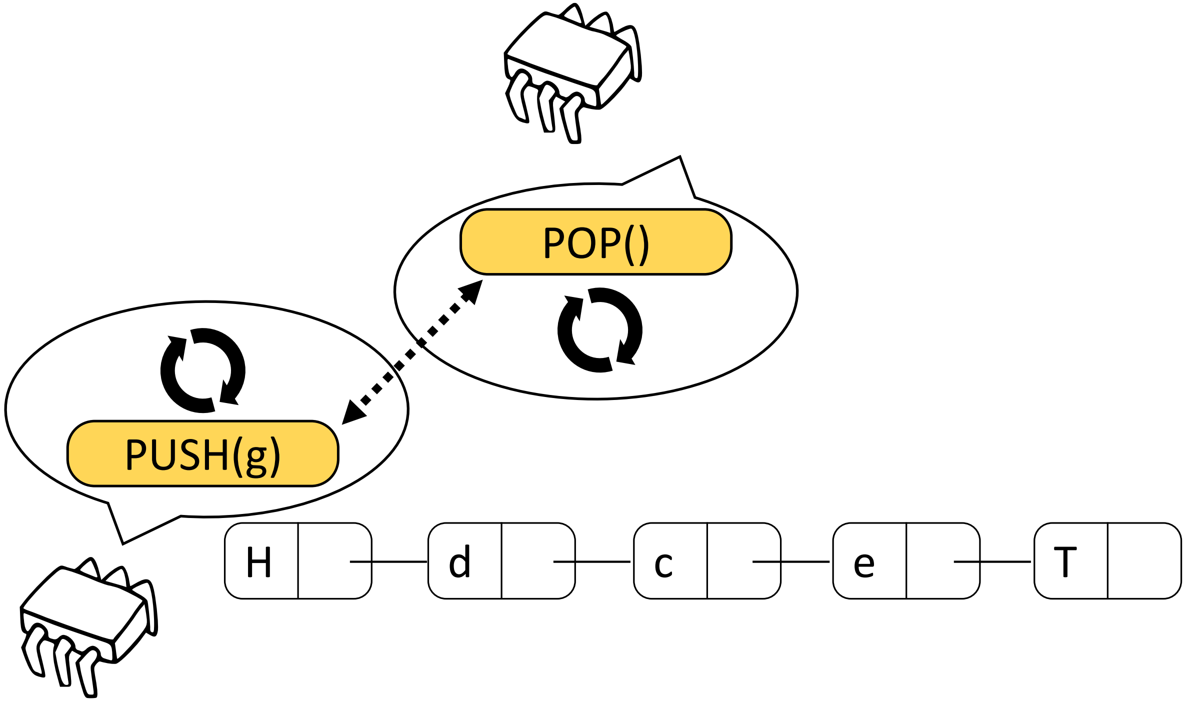
Observation

- Concurrent matching push/pop pairs are always linearizable



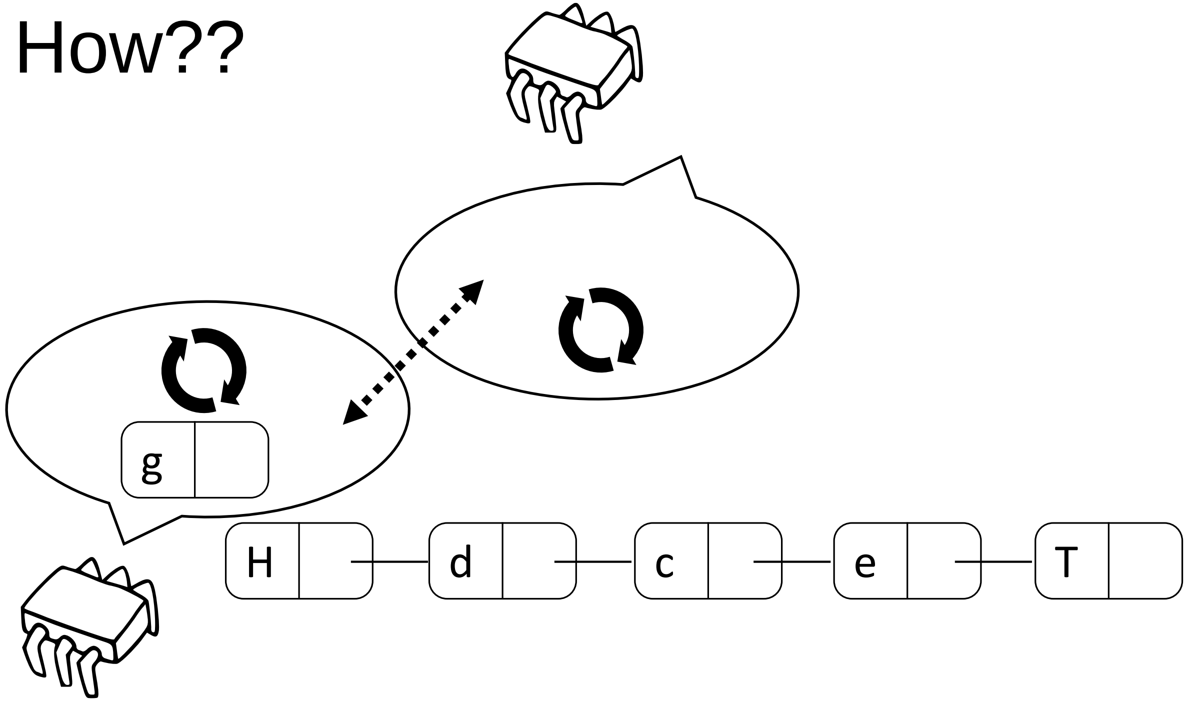
Non-blocking stack – Attempt 3

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



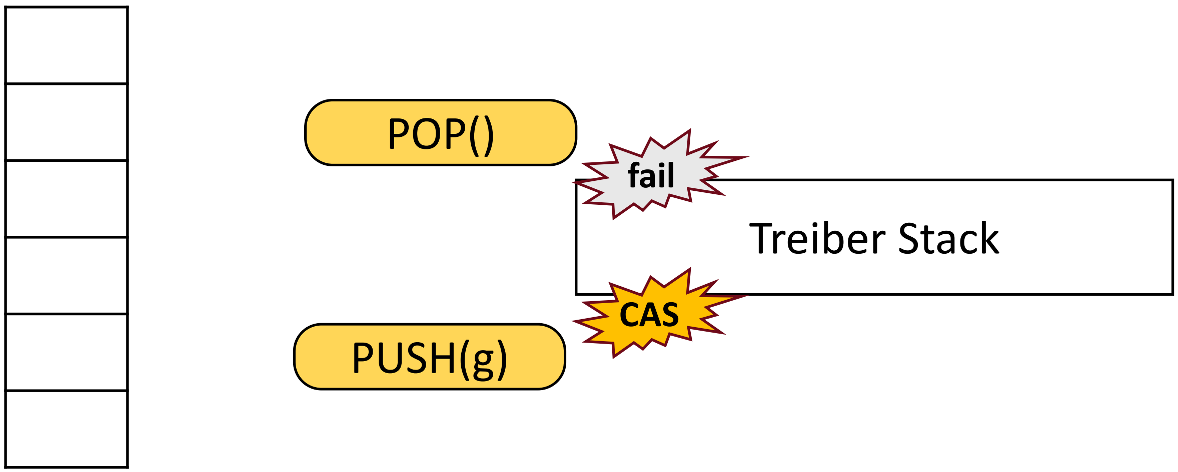
Non-blocking stack – Attempt 3

- 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??



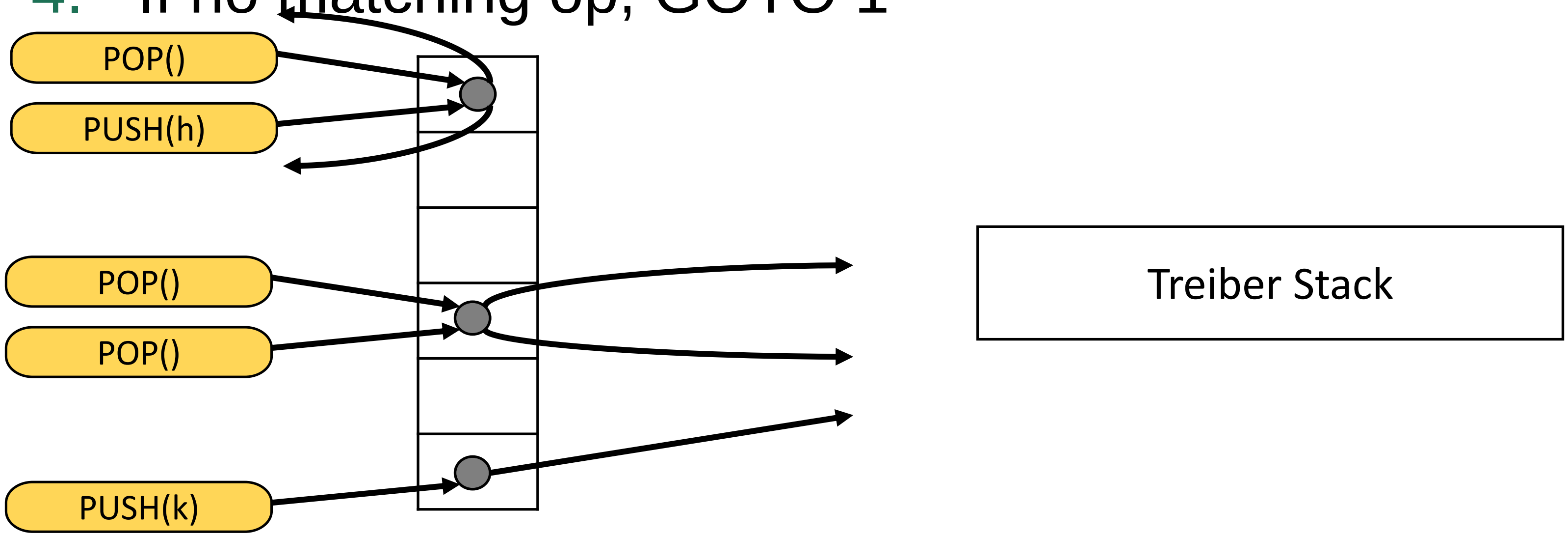
Non-blocking stack – Elimination stack

- 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

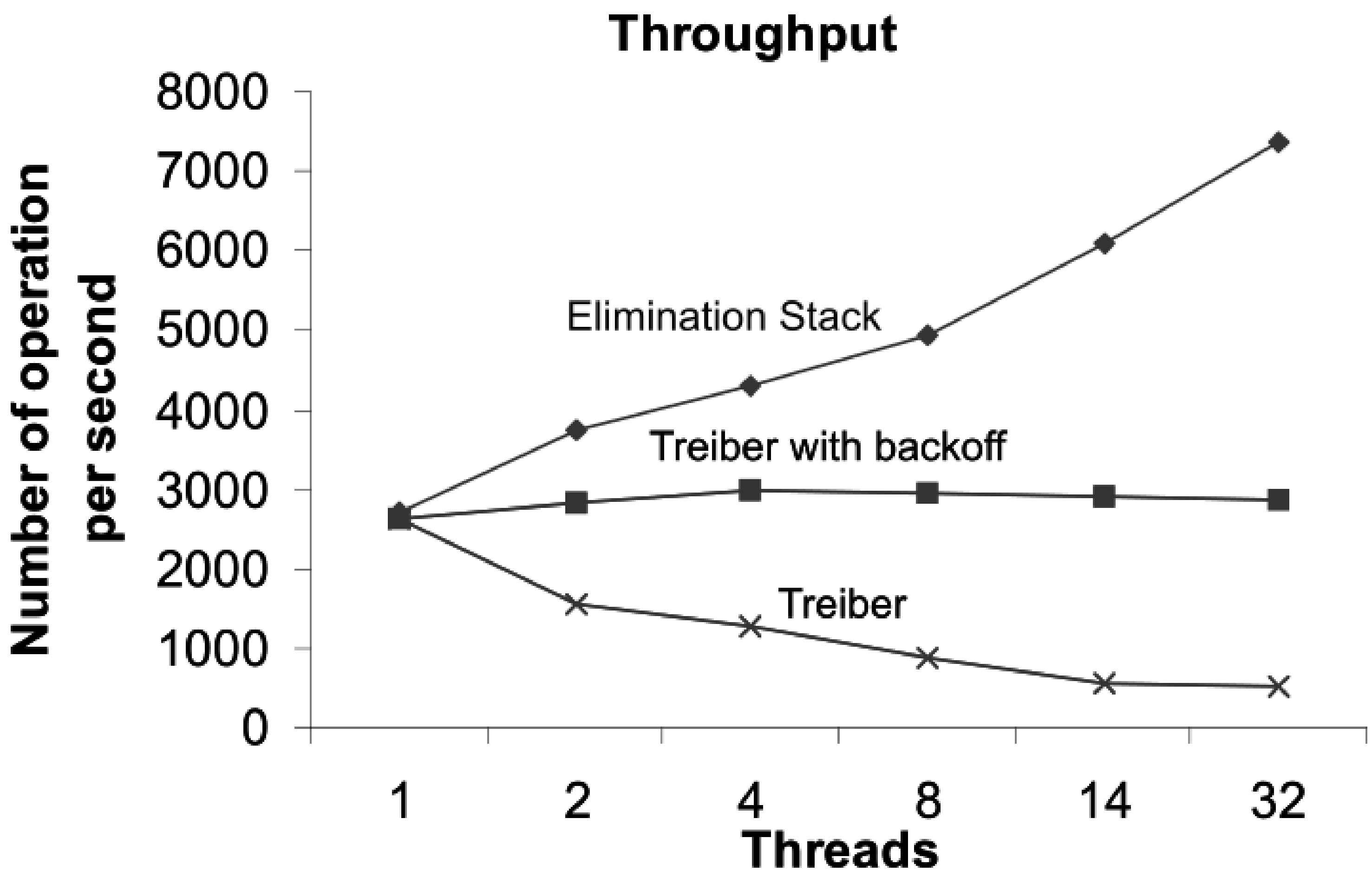


Non-blocking stack – Elimination stack

- 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



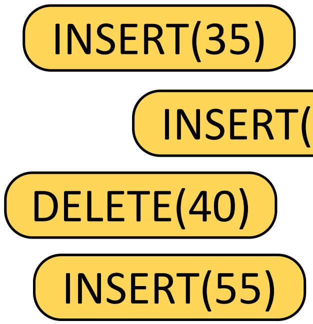
Non-blocking stack – Attempt 3

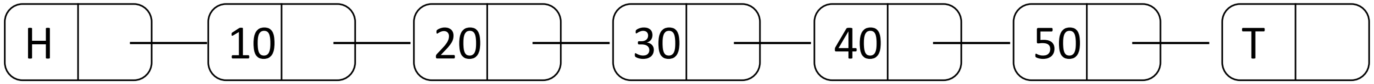


Concurrent Data Structures: Sets

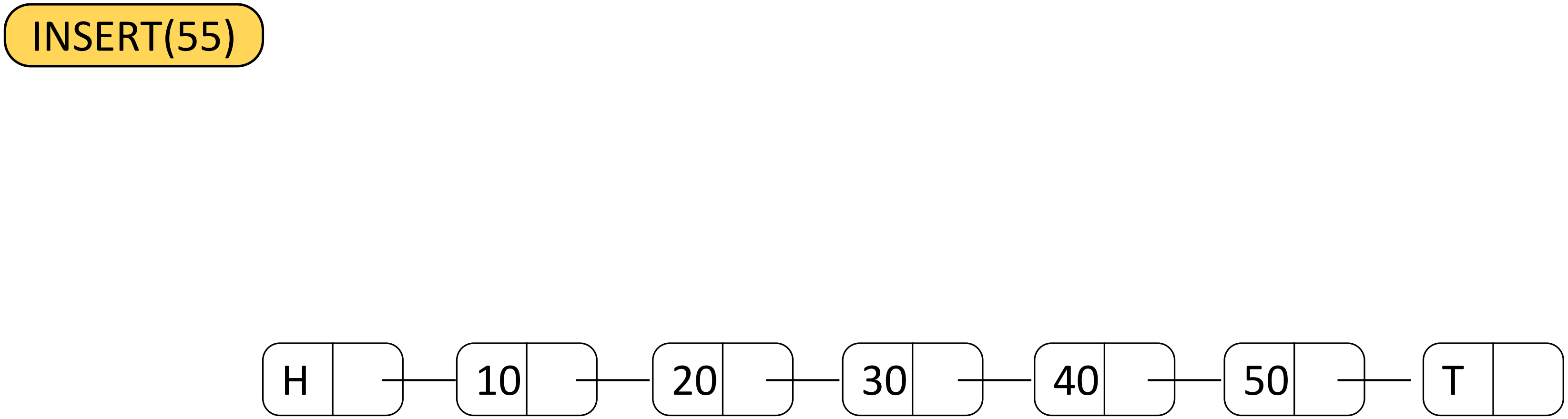
Set implementations

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

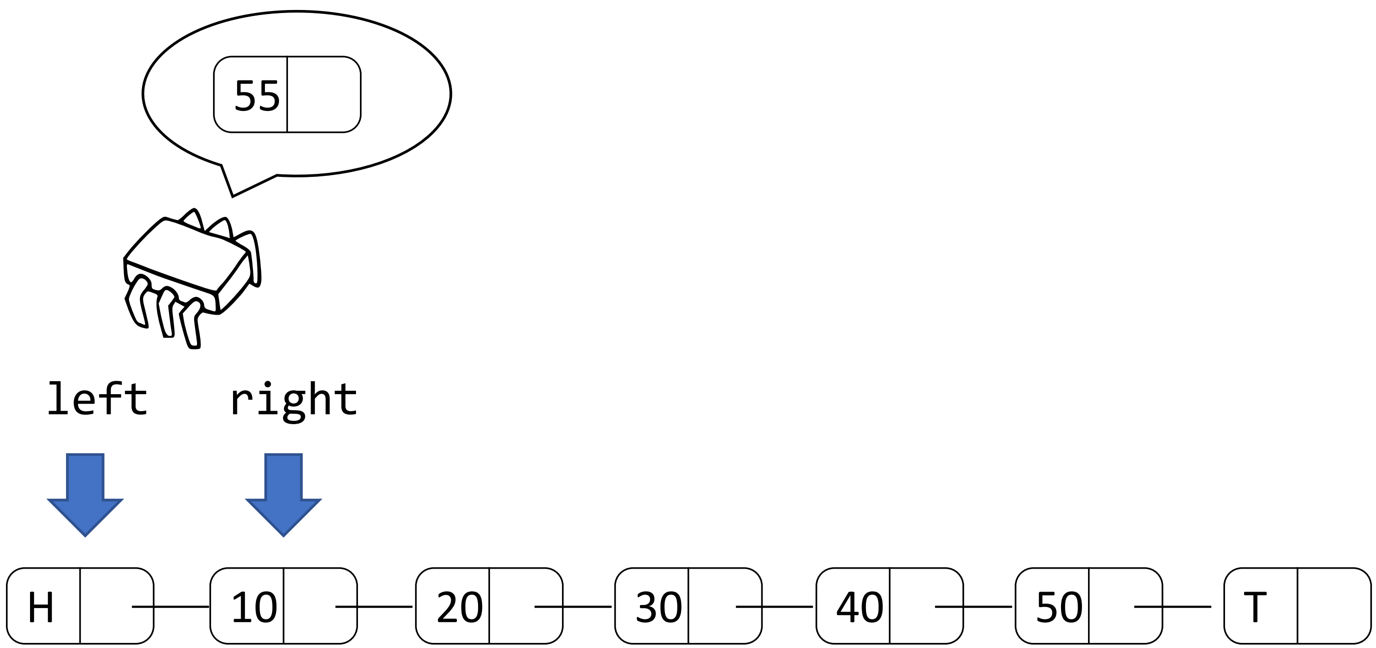




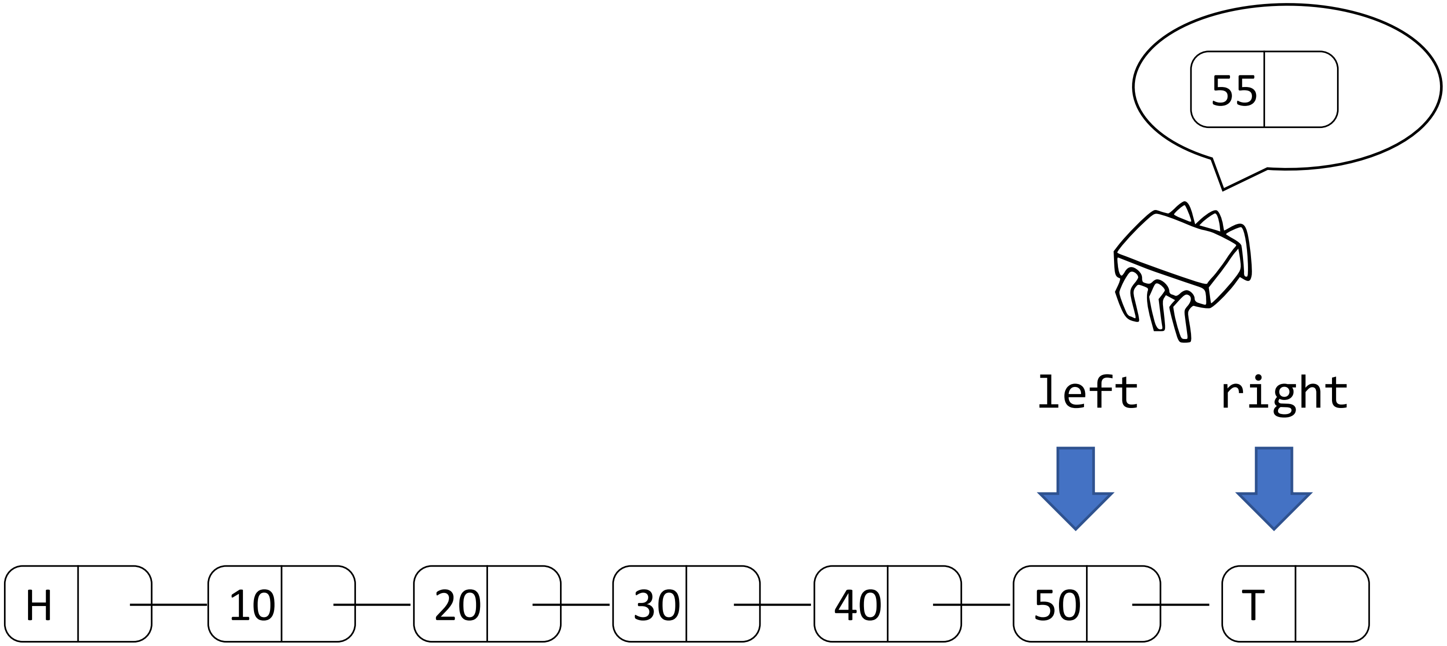
Insert algorithm



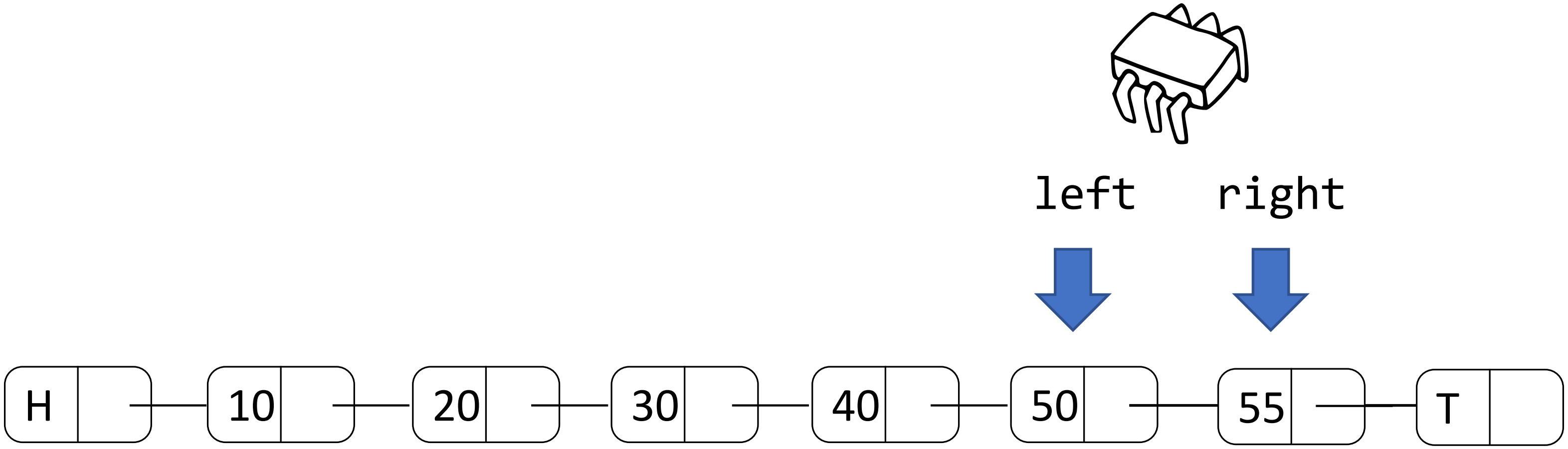
Insert algorithm



Insert algorithm

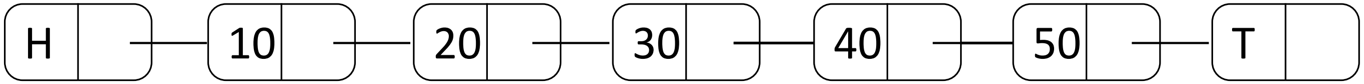


Insert algorithm

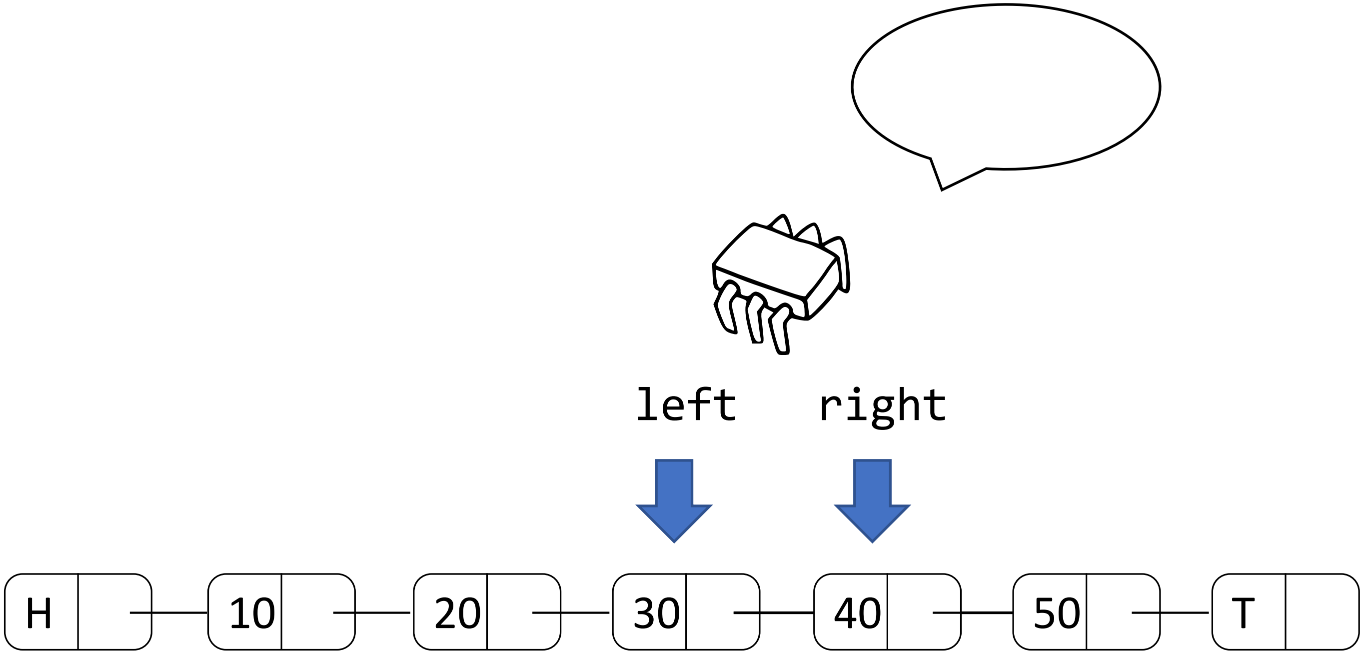


Delete algorithm

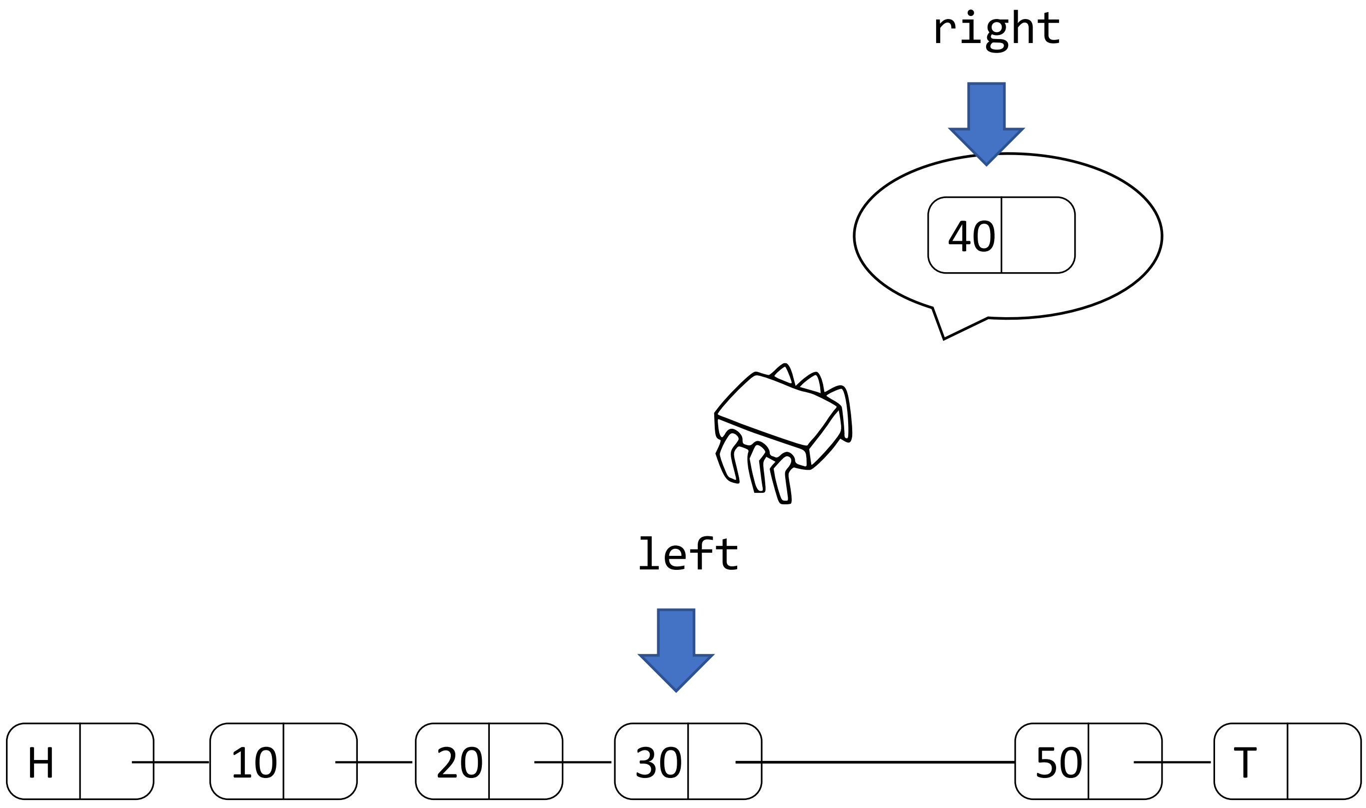
DELETE(40)



Delete algorithm



Delete algorithm



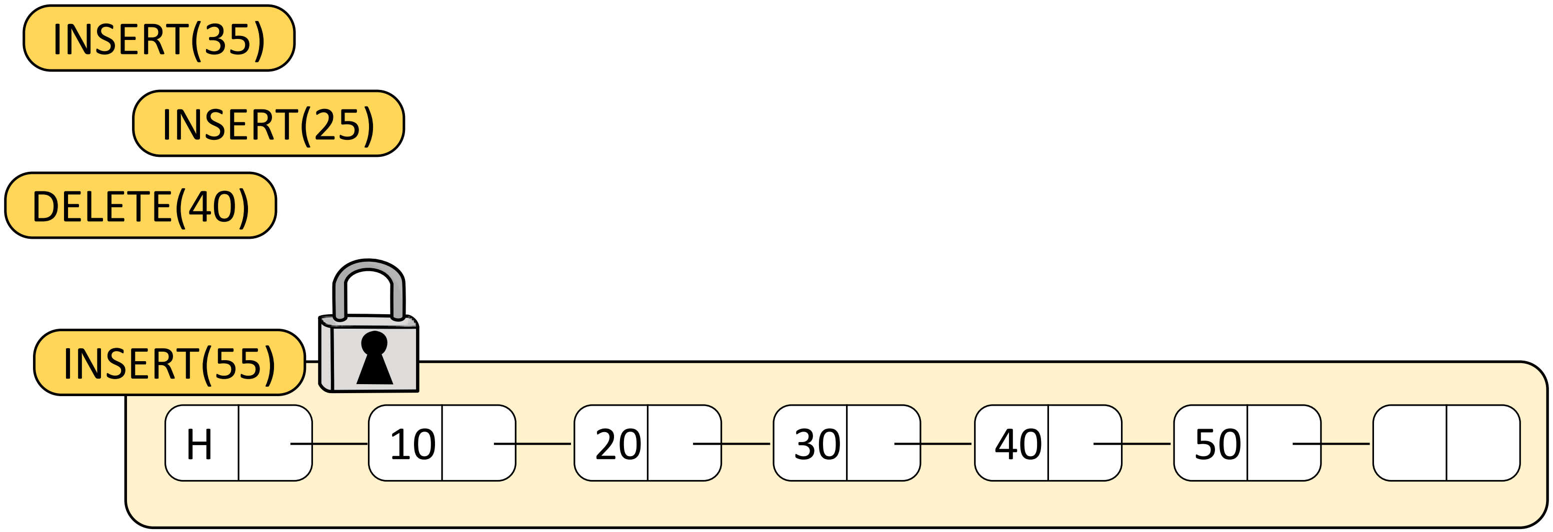
Sequential set implementation

```
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

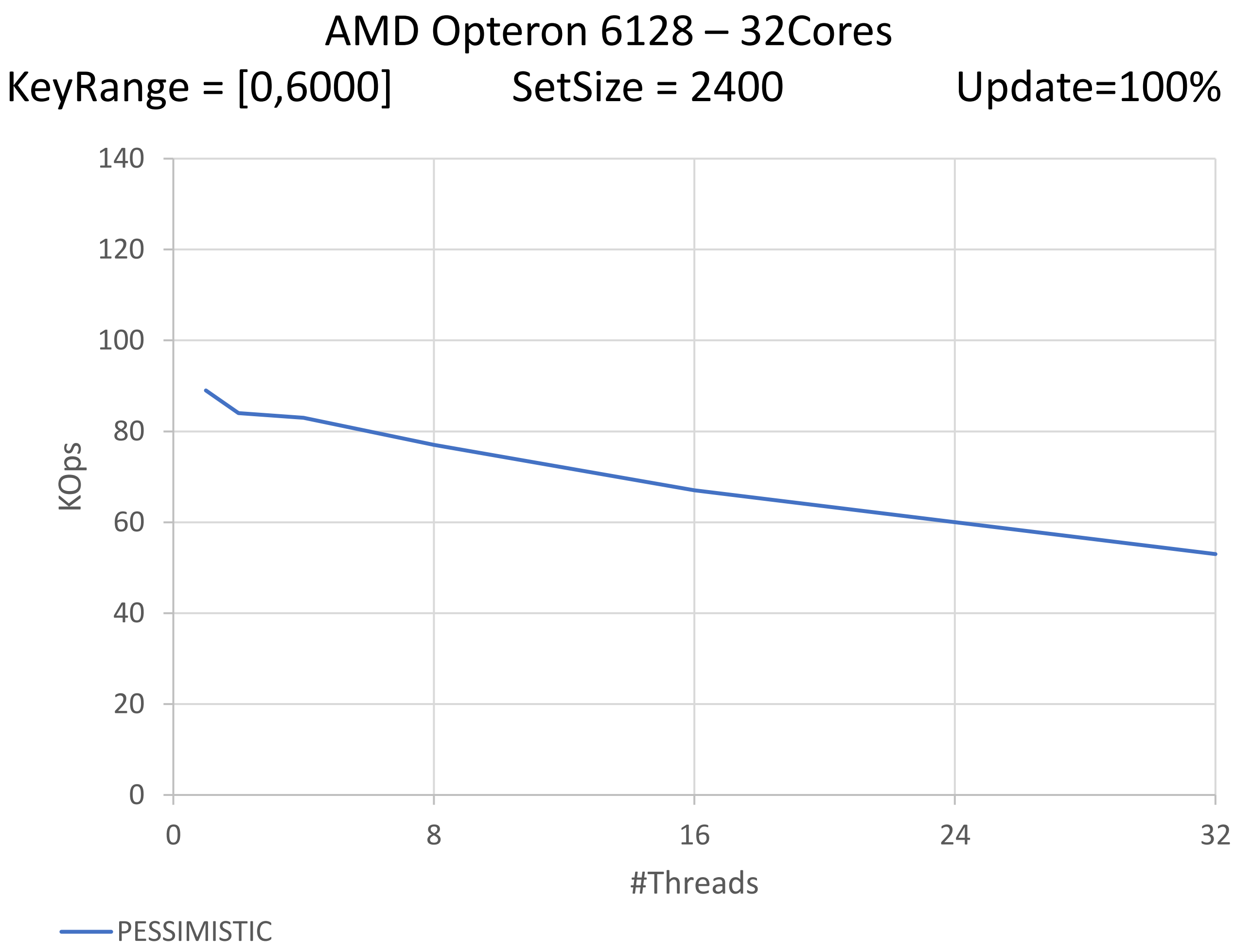


Concurrent set – Attempt 1 (SRC)

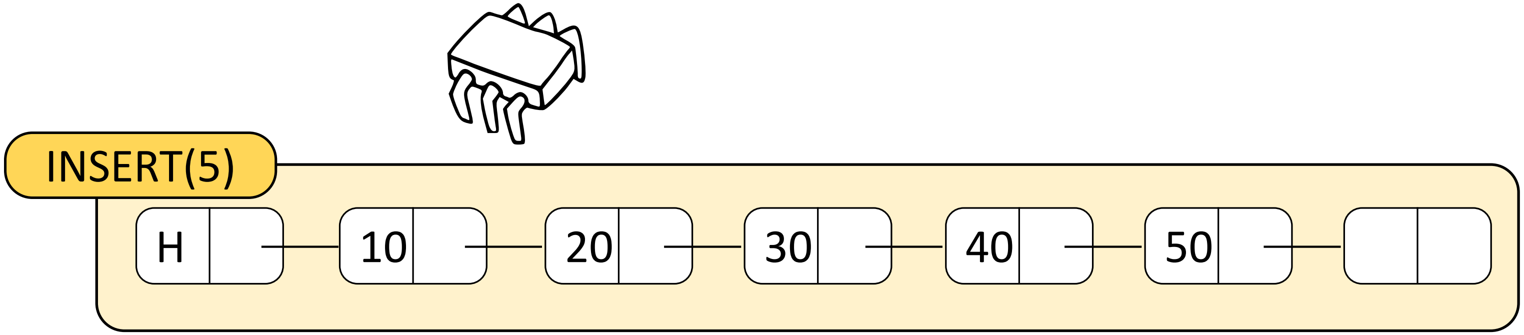
```
bool do_operation(int k, int op_type){
  bool res = true;
  node *l,*r;
  LOCK(&glock);
  l = search(k, &r);
  switch(op_type){
    case(INSERT):
      if(r->key == k)
        res = false;
      else
        l->next = new node(k,r);
      break;
    case(DELETE):
      if(r->key == k)
        l->next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



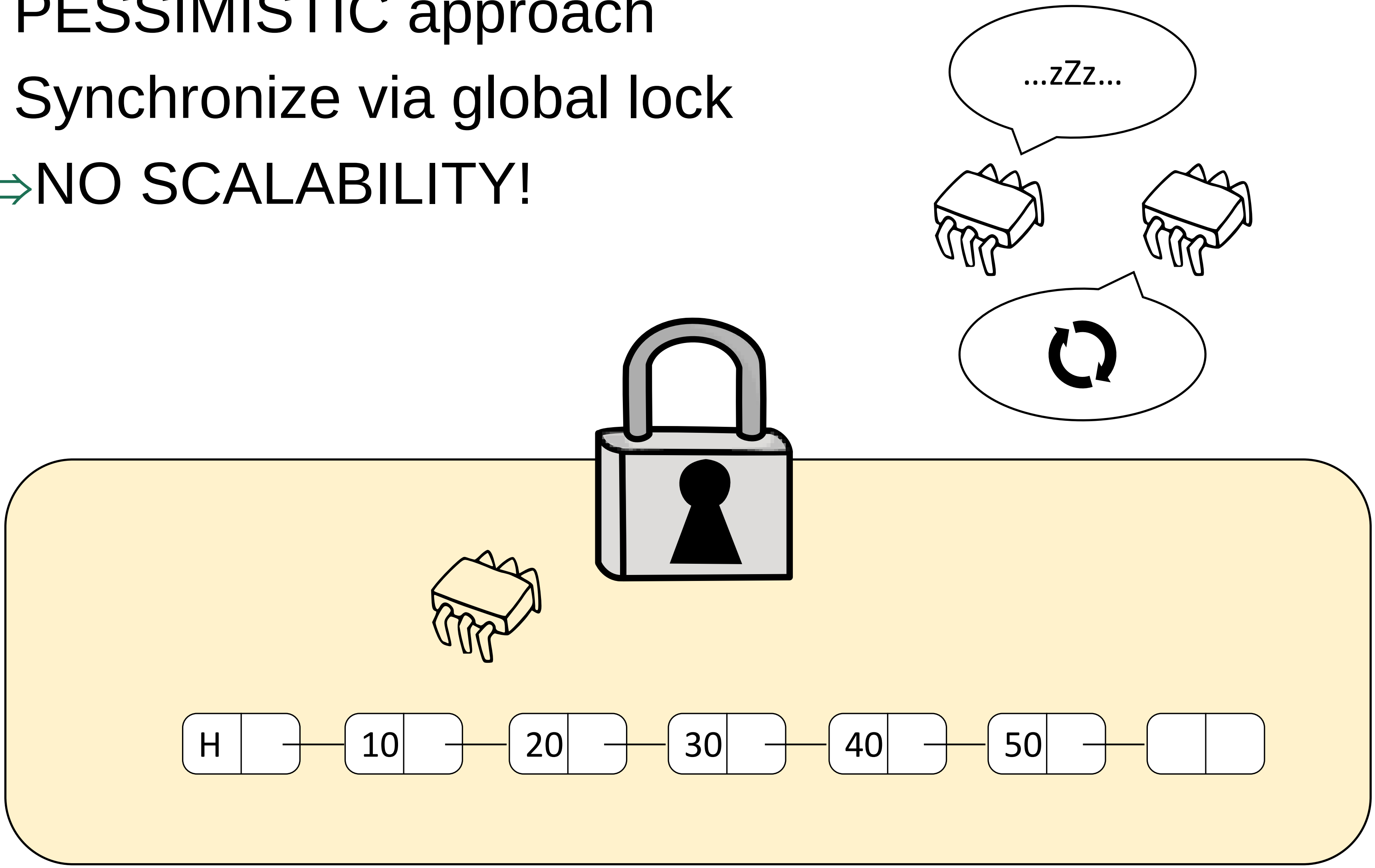
Concurrent set – Attempt 1



Concurrent set – Attempt 1

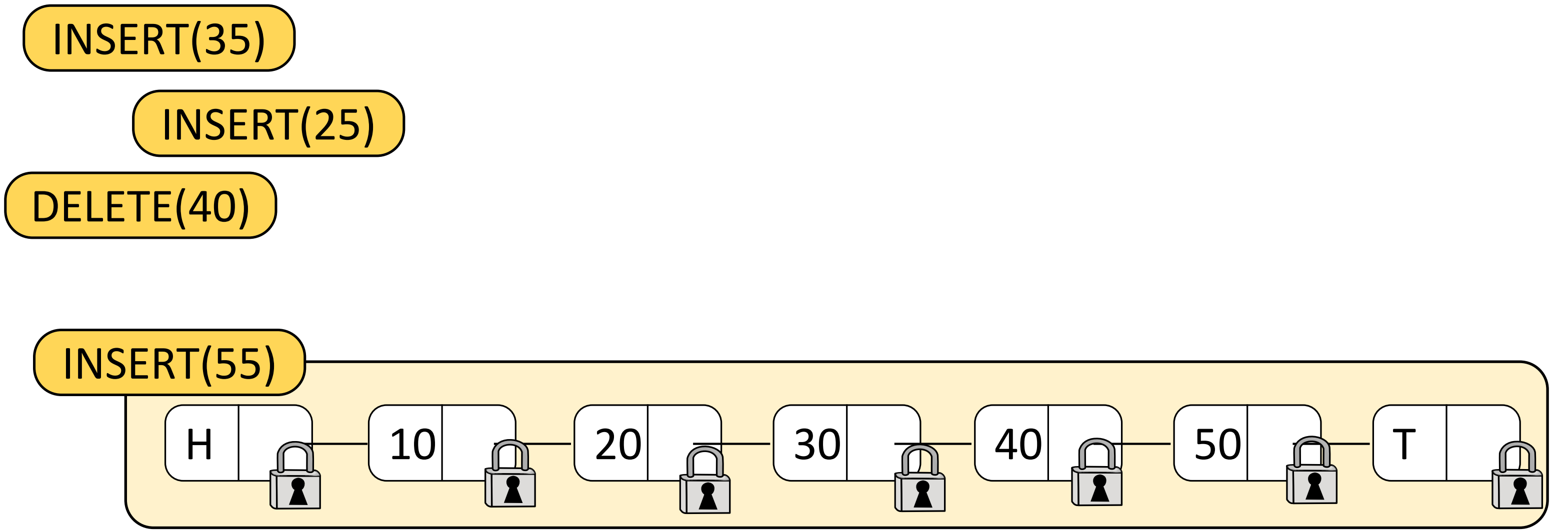
- PESSIMISTIC approach
- Synchronize via global lock

⇒NO SCALABILITY!



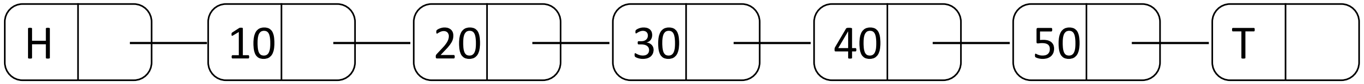
Concurrent set – Attempt 2

- 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



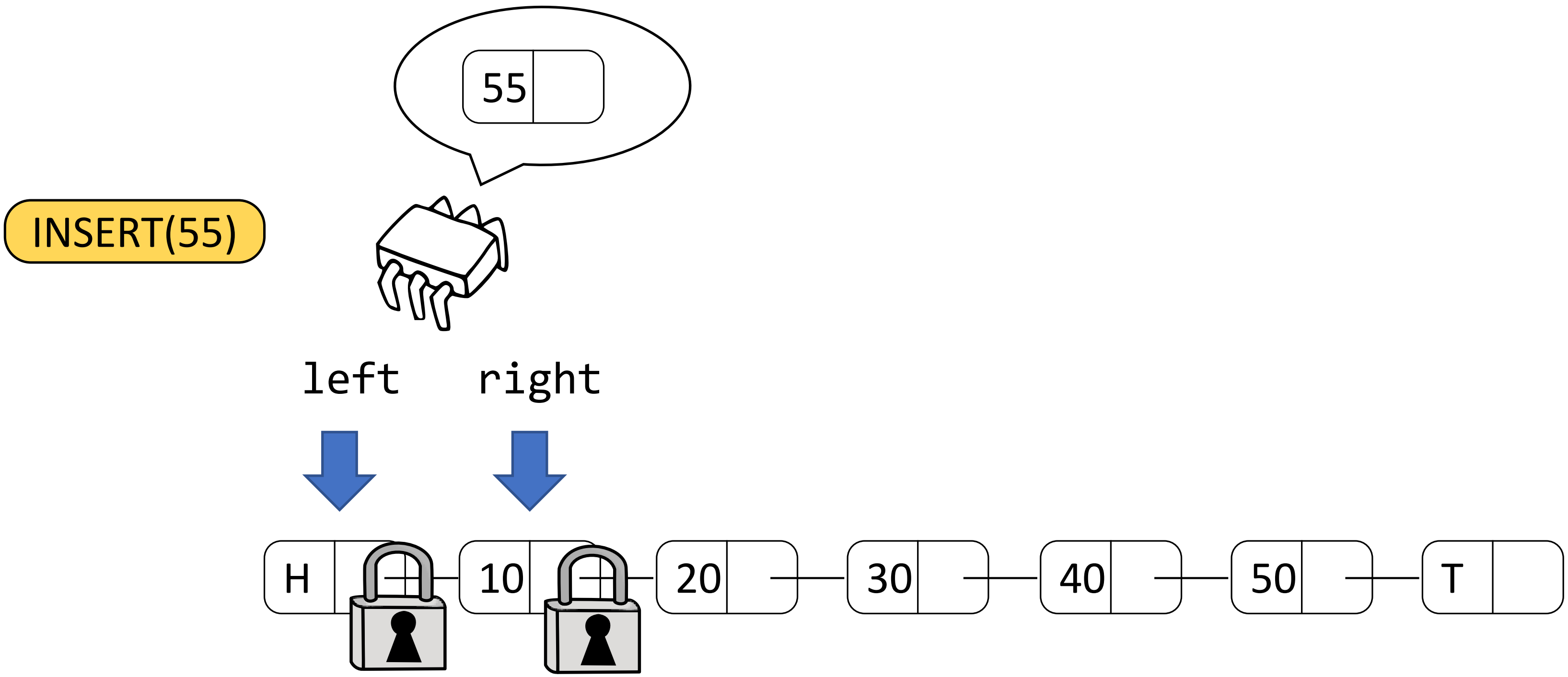
Search algorithm

INSERT(55)



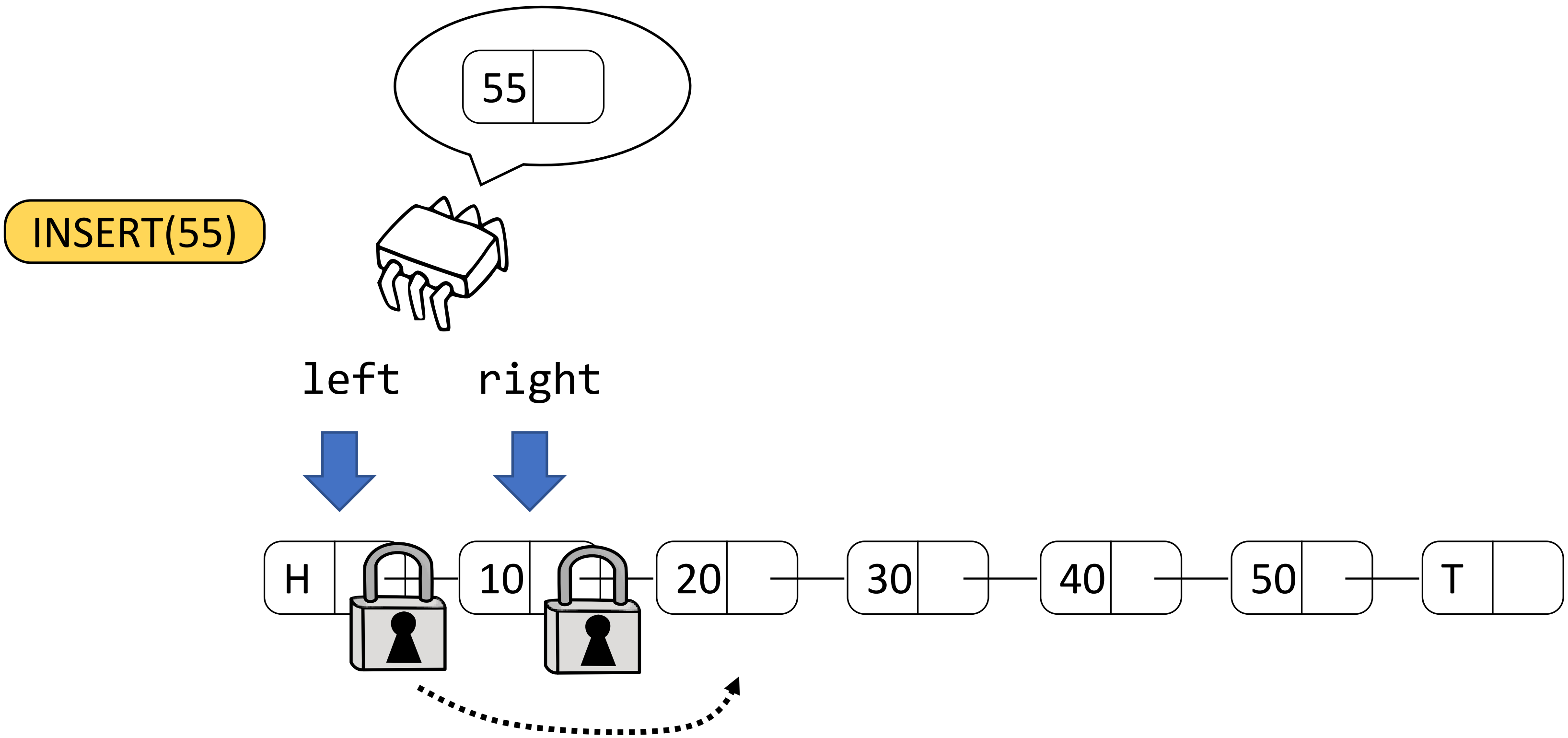
Search algorithm

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



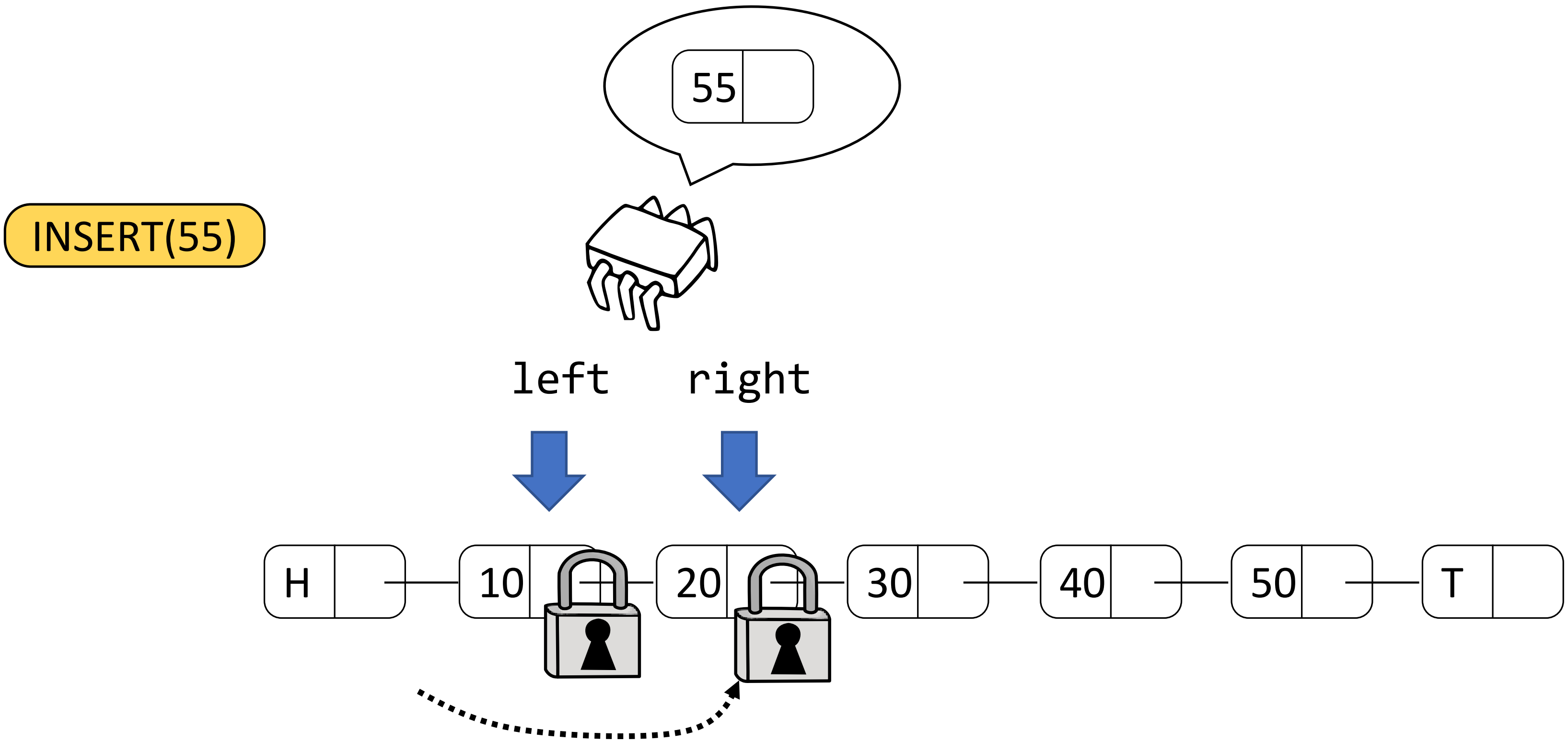
Search algorithm

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



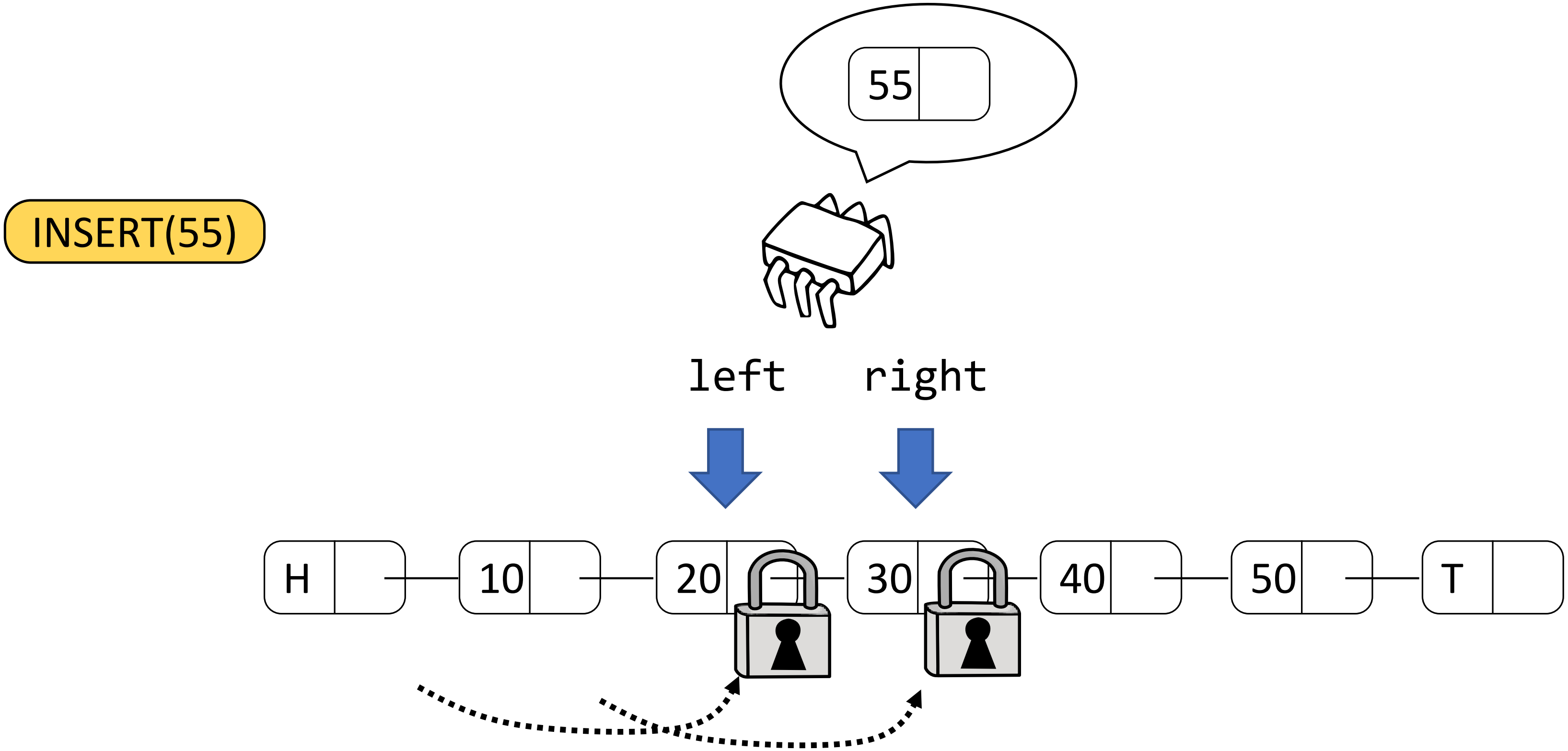
Search algorithm

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



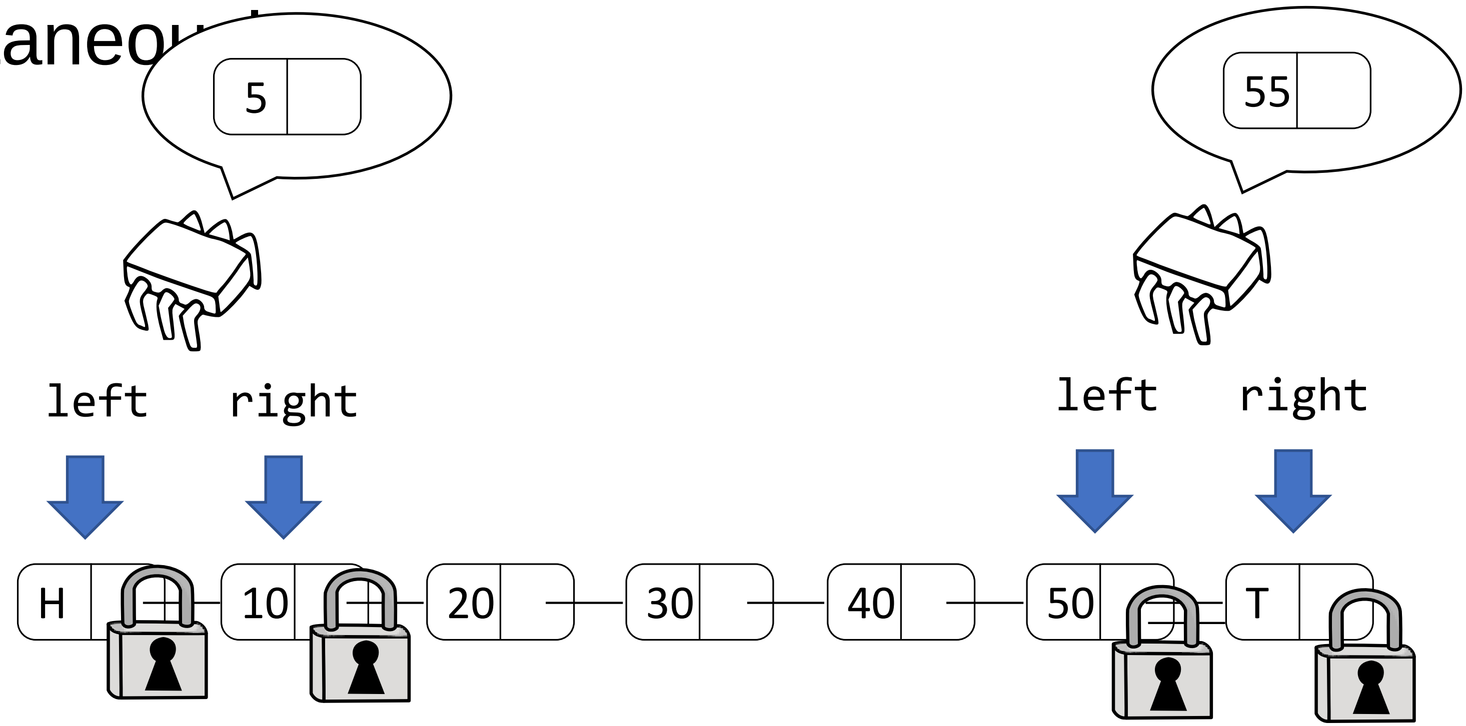
Search algorithm

- 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 simultaneou

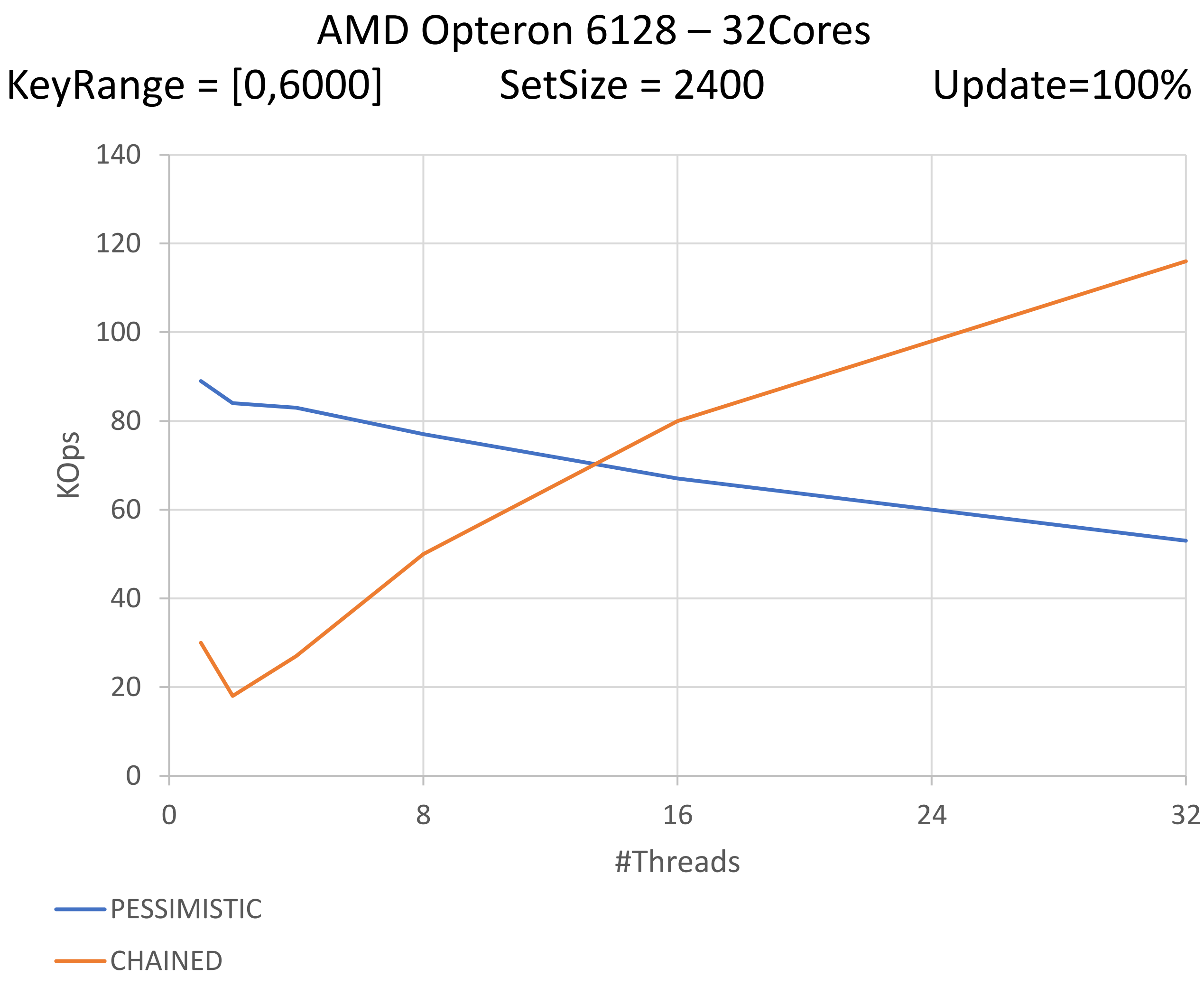


Concurrent set – Attempt 2 (SRC)

```
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;
}
```

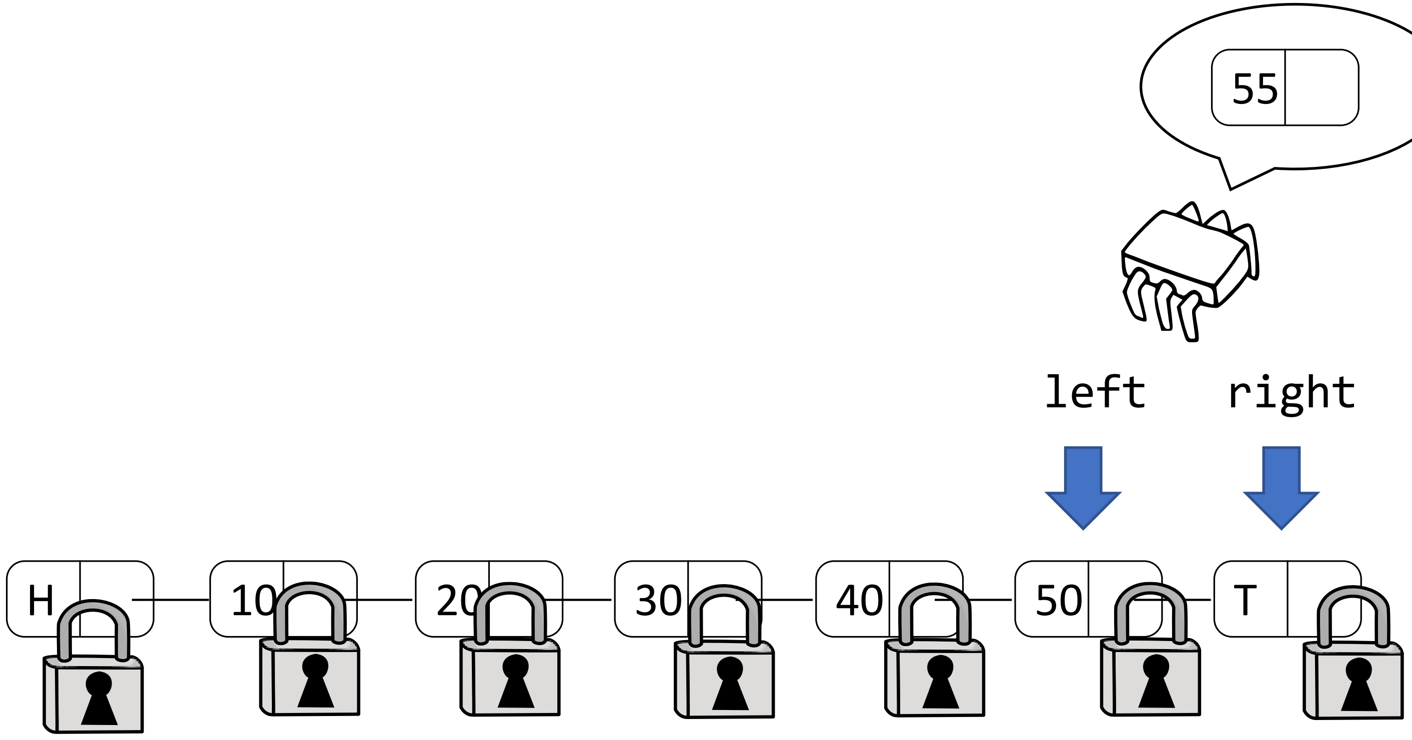
```
node* search(int k, node **r){
  node *l, *r_next;
  l = set->head;
  LOCK(&l->lock);
  *r = l->next;
  LOCK(&(*r)->lock);
  r_next = (*r)->next;
  while((*r)->key < k){
    UNLOCK(&l->lock);
    l = *r;
    *r = r_next;
    LOCK(&(*r)->lock);
    r_next = (*r)->next;
  }
}
```

Concurrent set – Attempt 2



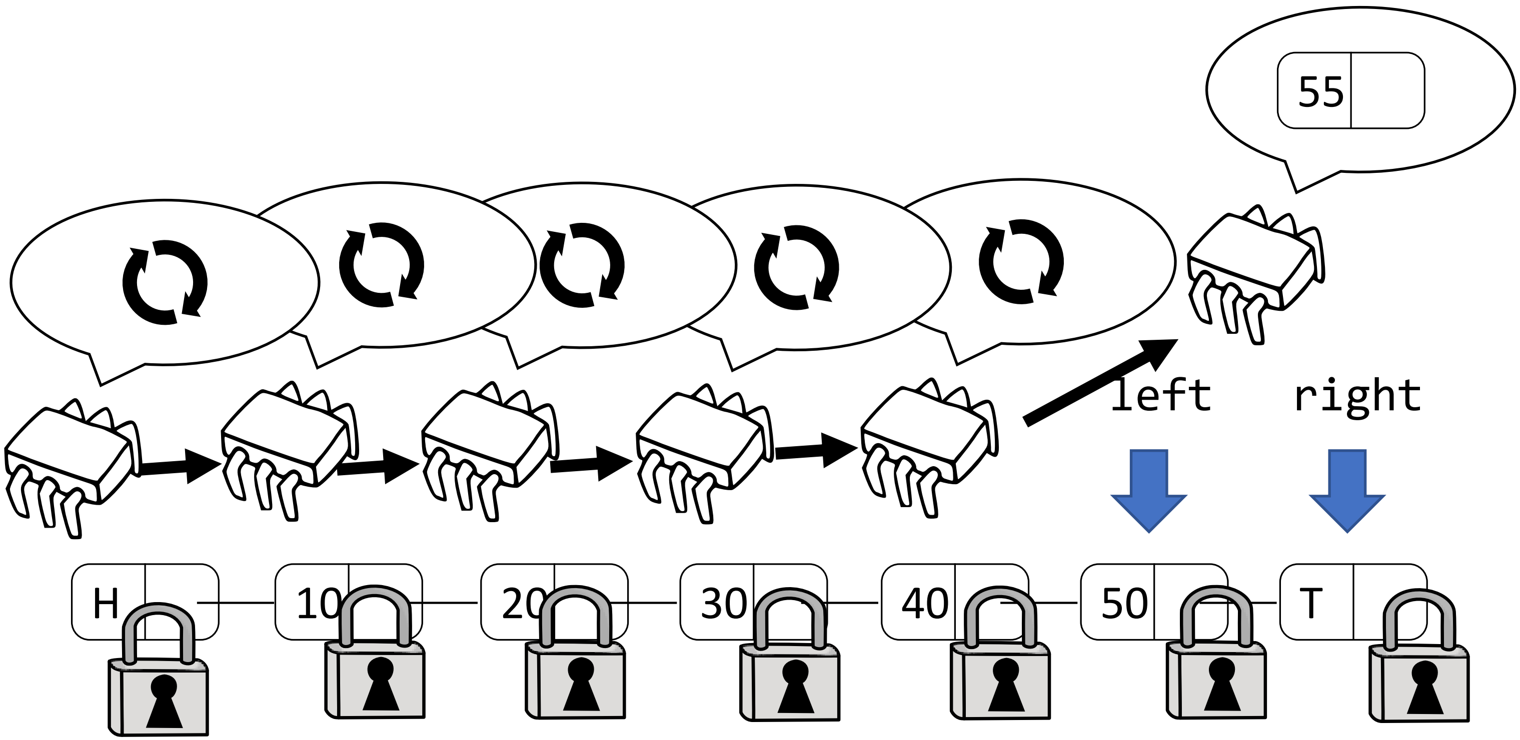
Search algorithm

- Allows an increased parallelism but…



Search algorithm

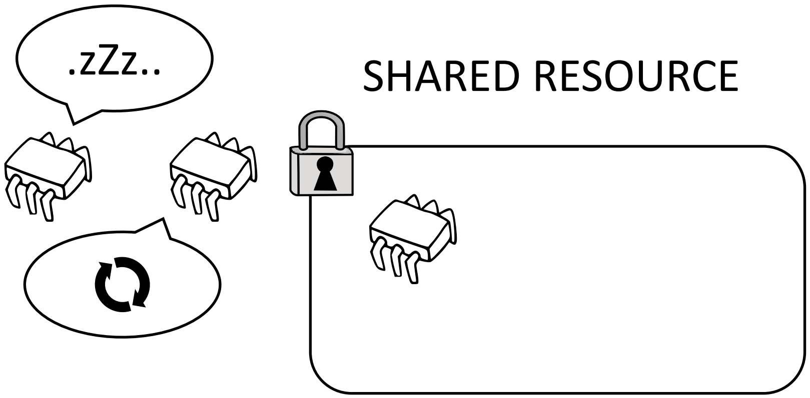
- Allows an increased parallelism but…
- High costs for lock handover



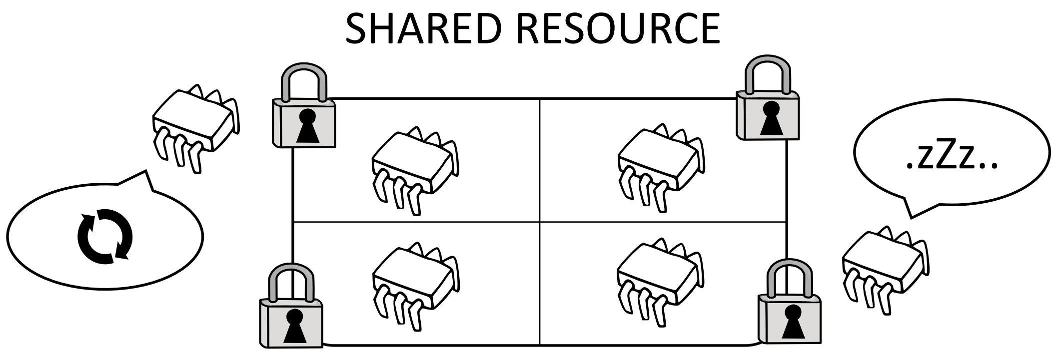
Recap

- Explored two blocking strategies:

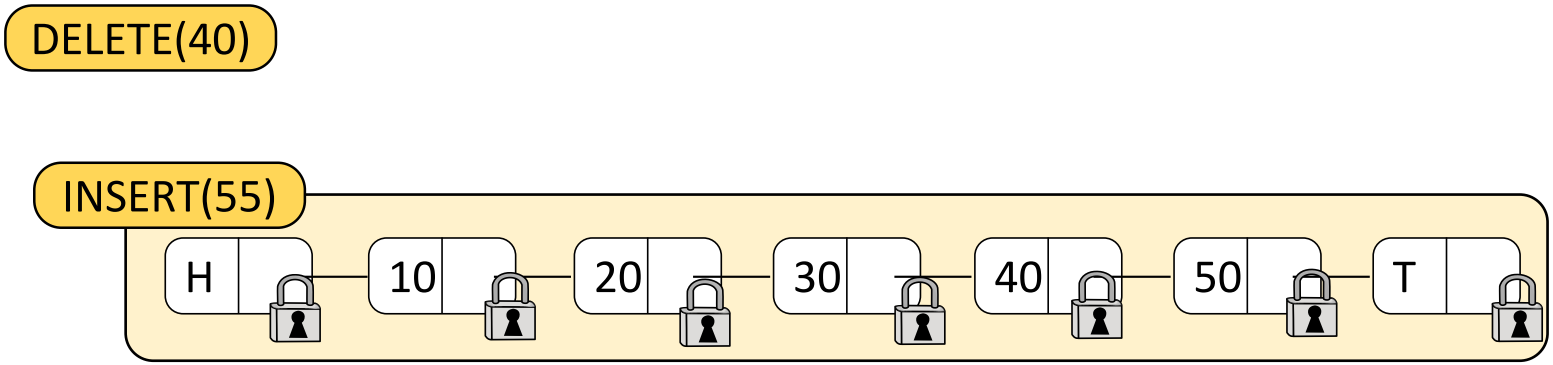
1. Global (coarse-grain) lock



2. (Fine-grain) Lock coupling

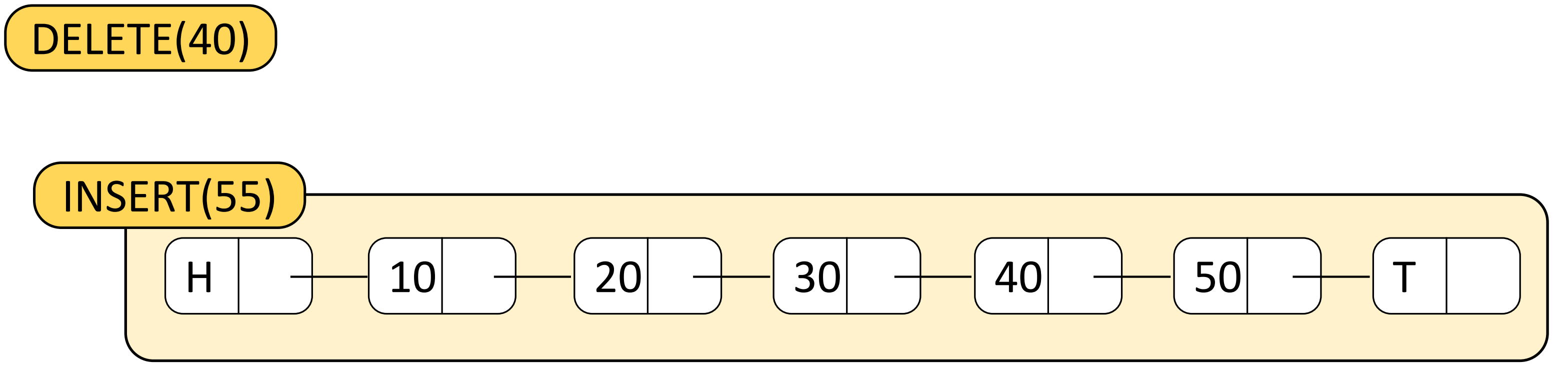


Concurrent set – Attempt 3



Concurrent set – Attempt 3

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



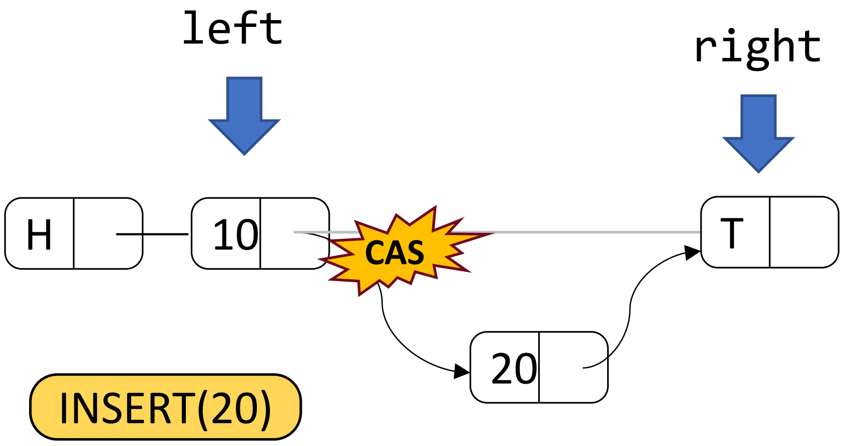
Non-blocking insert & delete algorithms

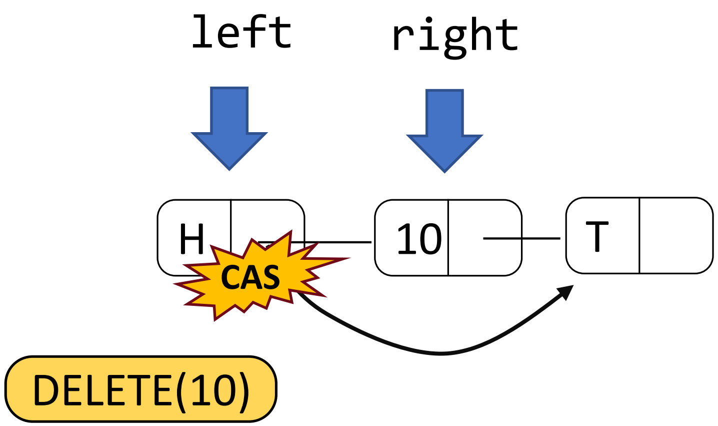
Insert:

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





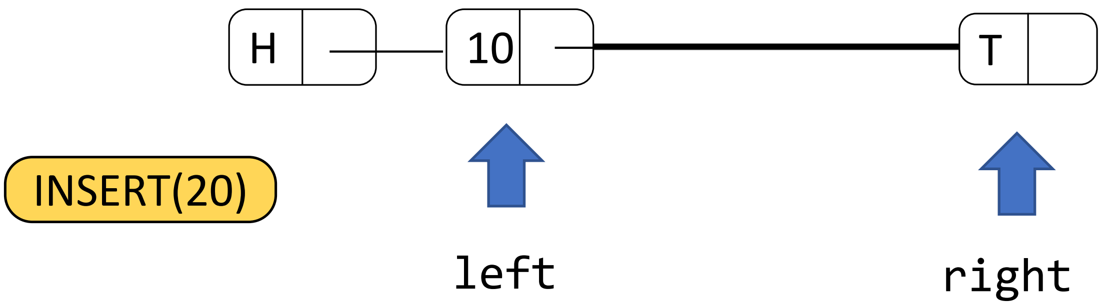
- Is it correct?

Incorrect delete algorithm

- Edge cases might lead to losing items!

Incorrect delete algorithm

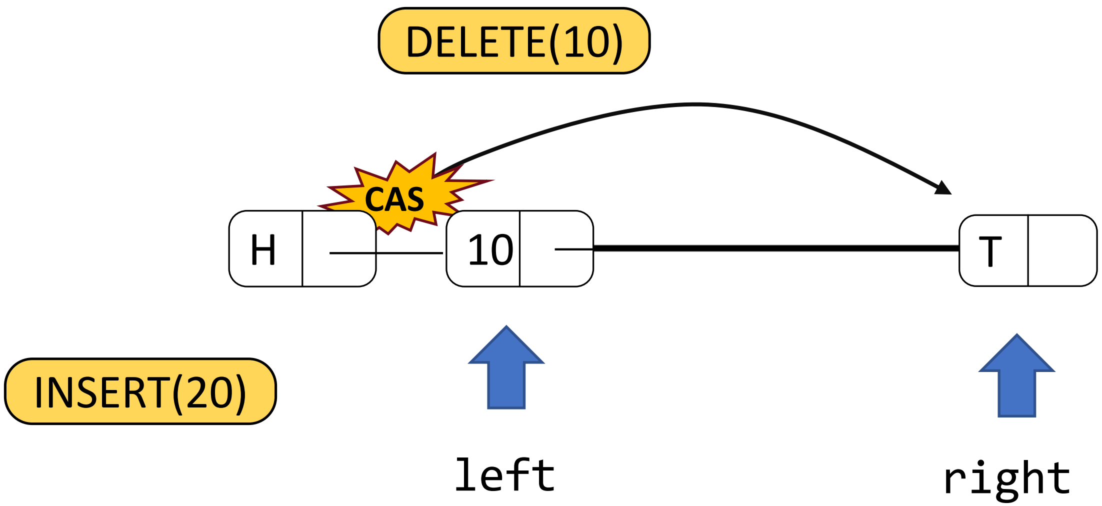
- Edge cases might lead to losing items!



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

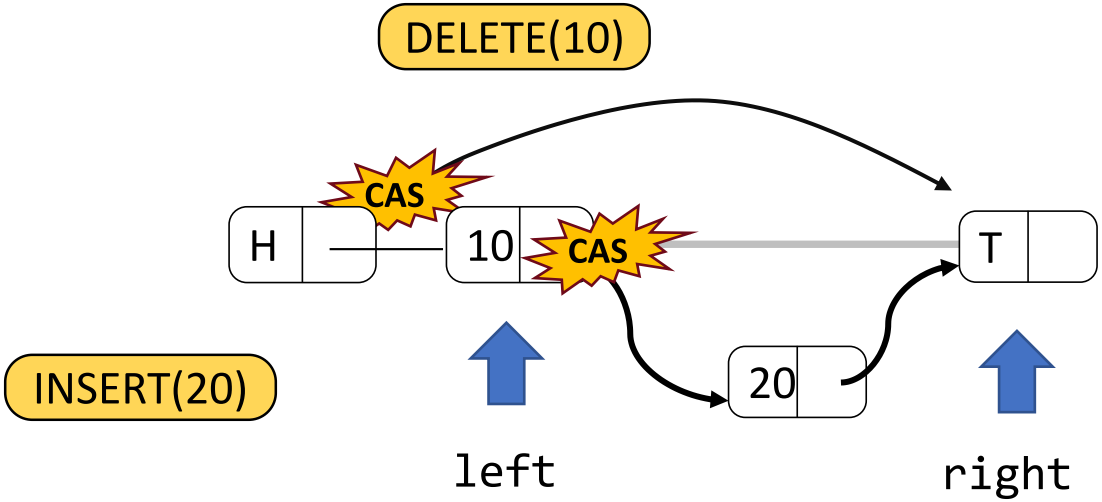
- Edge cases might lead to losing items!



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

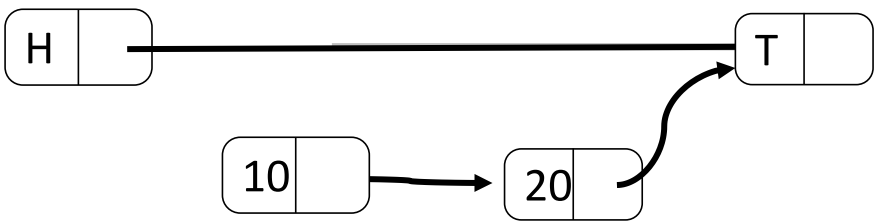
- Edge cases might lead to losing items!



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!

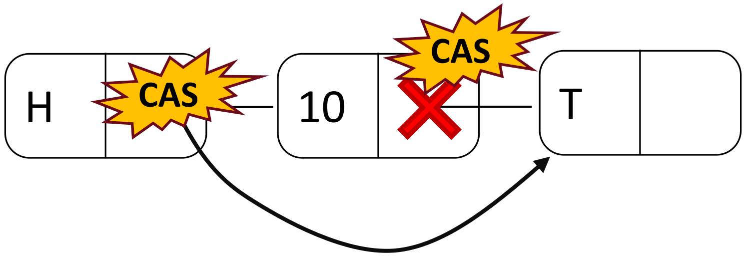


1. Thread A gets left and right node and go to sleep
2. Thread B disconnects the node containing 10
3. Thread A wakes up and add 20 after 10
4. The new item is lost

The correct delete algorithm

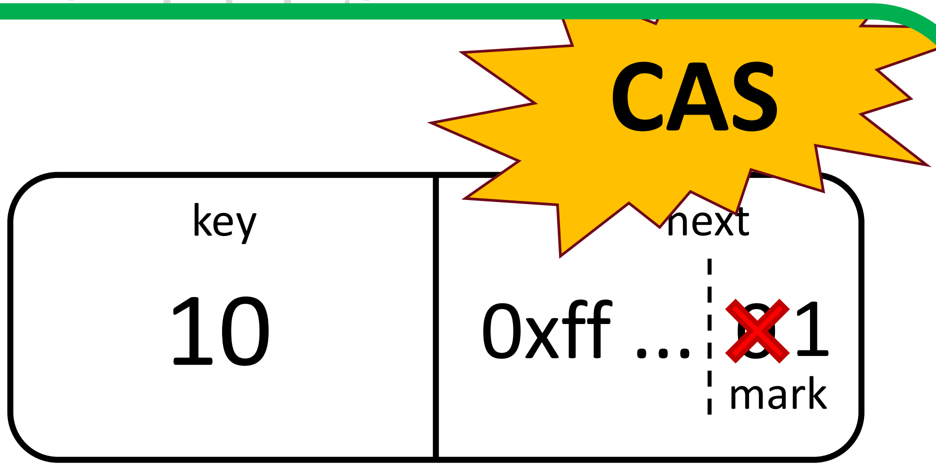
- Adopt logical deletion:

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

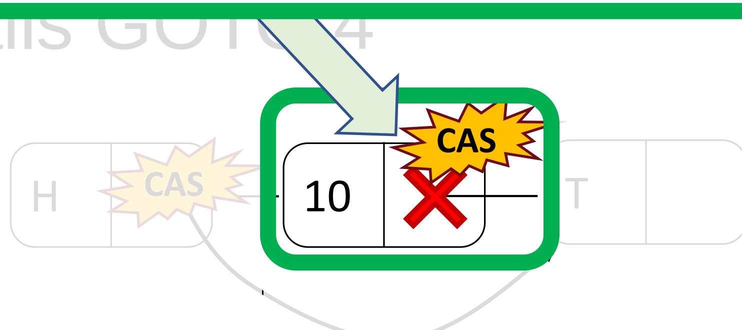


The correct delete algorithm

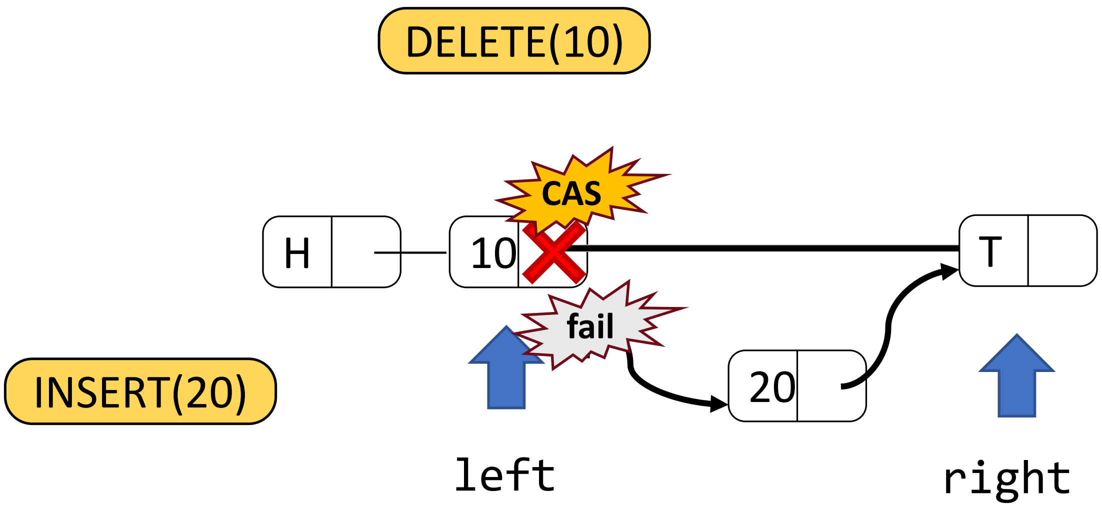
- Adopt ...
1. Ge...
2. Ma... *...ogical* de...
3. If C...
4. Di... *...cal* deletion)
5. If CAS fails GOTO 4



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



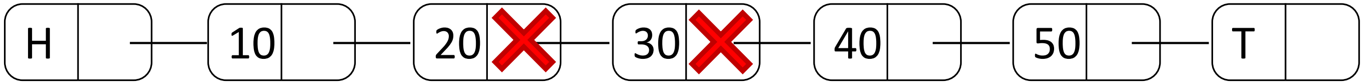
The correct delete algorithm



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

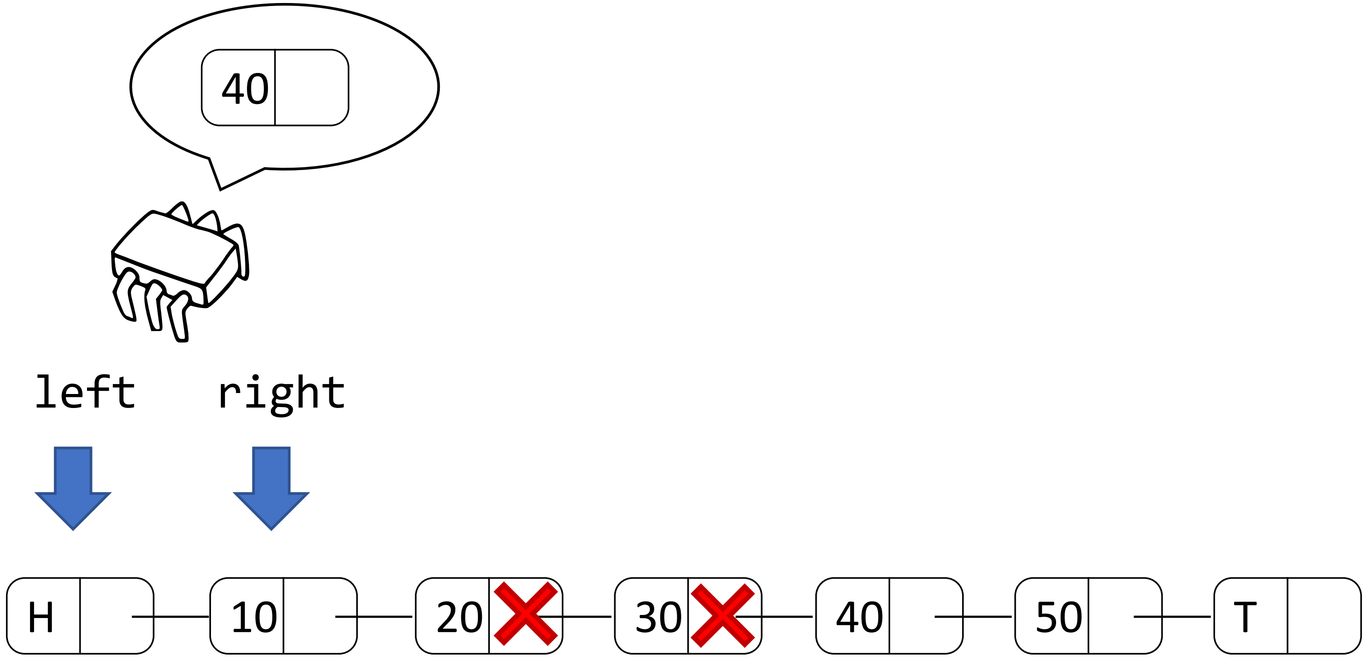
Non-blocking search

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



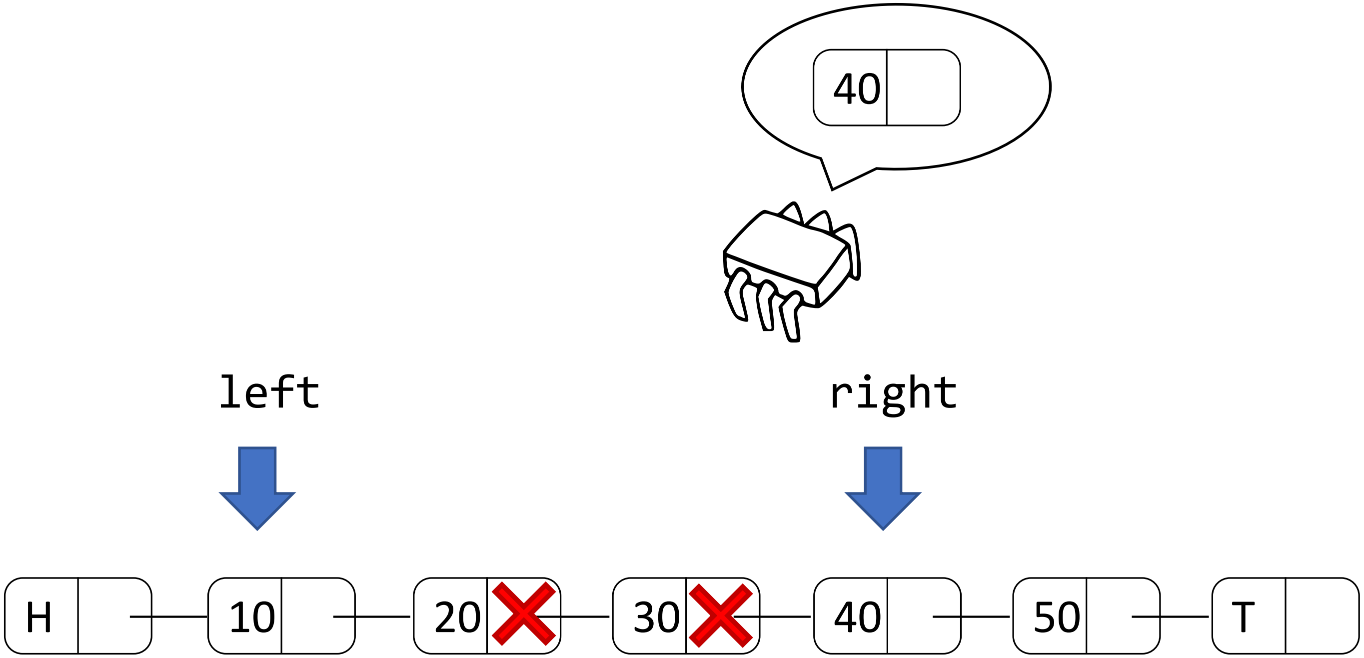
Non-blocking search

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



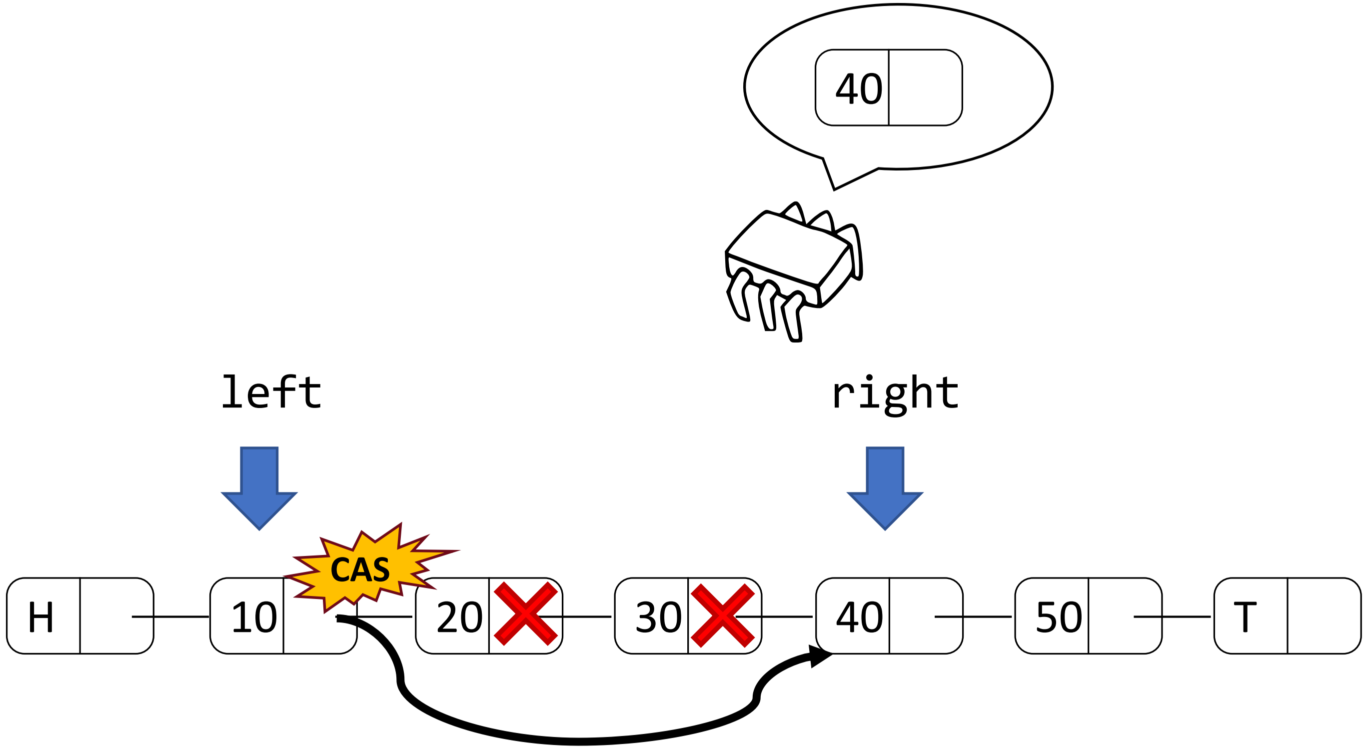
Non-blocking search

- 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;
}
```

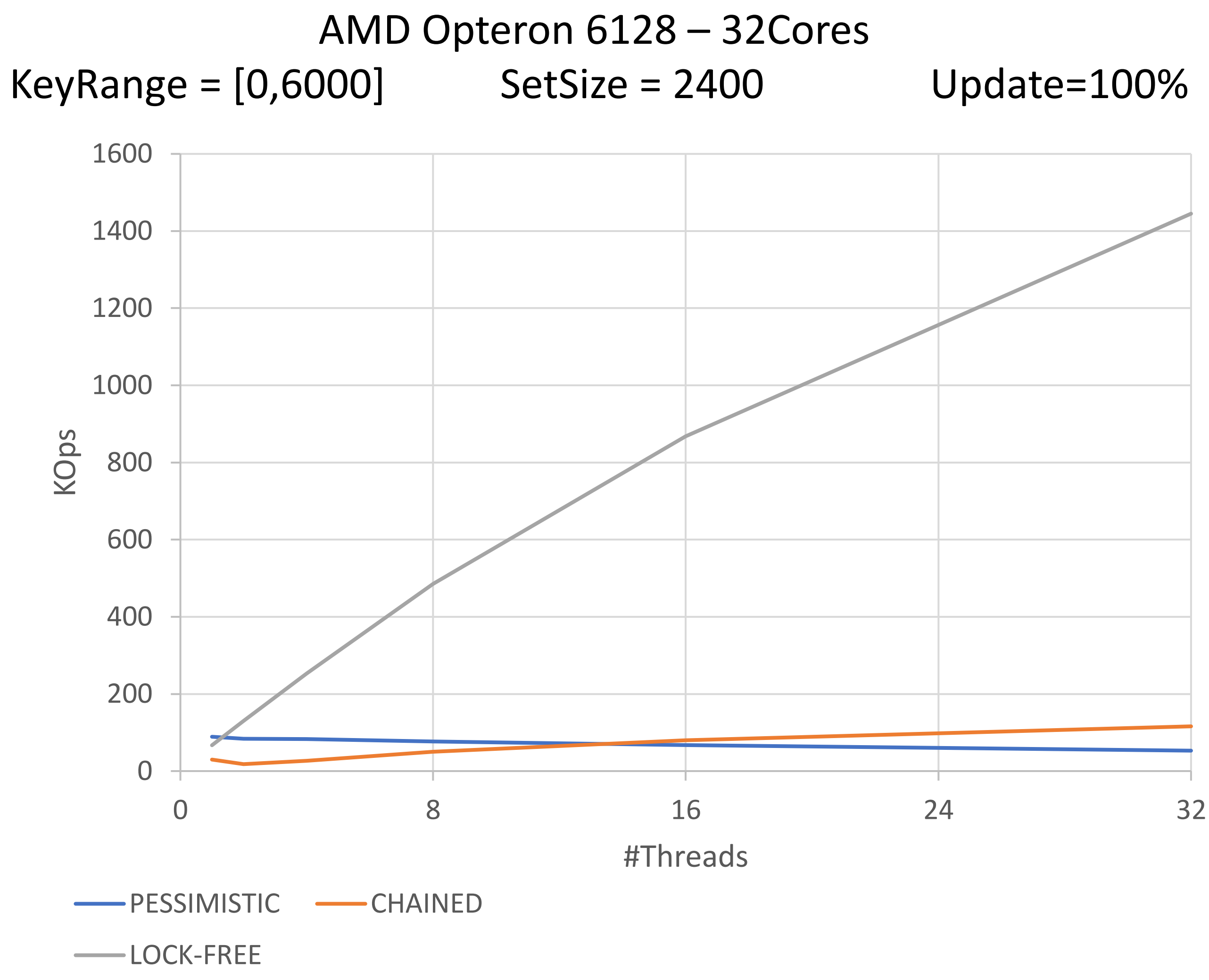
Concurrent set – Attempt 3 (SRC)

```
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       */
      if(!CAS(&l->next, r, n))
         goto 3;          /* insertion failed the item -> restart */
      break;
    case(DELETE):
      if(r->key != k) return false;  /* key not present        */
      l->next = r->next;                /* remove the key        */
      if(is_marked_ref((l=r->next)) || !CAS(&r->next, l, mark(l)))
         goto 3;          /* insertion failed the item -> restart */
      search(k,&r);                     /* repeat search         */
      break;
  }
  return true;
}
```

Concurrent set – Attempt 3 (SRC)

```
node* search(int k, node **r){
  node *l, *t, *t_next, *l_next;
  *t = set->head;
   t_next = t->head->next;
   while(1){                           /* FIND LEFT AND RIGHT NODE */
      if(!is_marked(t_next)){
         l = t;
         l_next = t_next;
      }
     t = get_unmarked_ref((t_next);
     t_next = t->next;
     if(!is_marked_ref(t_next) && t->key >= k) break;
  }
  *r = t;
  /* DEL MARKED NODES */
  if(l_next != *r && !CAS(&l->next, l_next, *r) goto 3;
  return l;
}
```

Concurrent set – Attempt 3



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

Progress (Lock freedom)

- Each method update method has two main steps
  - A search, which might end with a CAS
  - A CAS to insert delete a node

1. Suppose an update method is stuck in a search:
   - The key range is finite, so the number of node is finite
   - It continuously fails to disconnect marked nodes
   - It means that new nodes have been both inserted and marked!
     - Other threads have completed update methods
2. Suppose an updated method always fails its last step (insertion or marking)
   - Other threads have modified the target next pointer
   - If it is due to the disconnection of marked nodes, see point 1
   - If it is due to the updated step other methods have completed

Safety (Linearizability)

1. The search returns 2 adjacent nodes in an atomic point
   1. The read of next field of the left node
   2. The CAS that make left and right adjacent

- It is like that the search made a snapshot of interested key interval

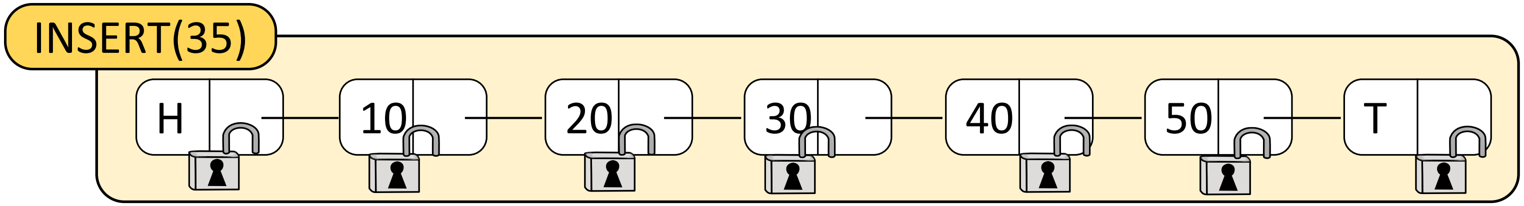
2. Find, unsuccessful delete and unsuccessful insert linearize with the search (1.1 or 1.2)
3. Insert linearizes with the successful CAS to connect a new node
4. Delete linearizes with the successful CAS to mark a node

Problems

- It is not possible to flip a bit of a reference on memory-managed languages (e.g. JAVA)
- How to solve?

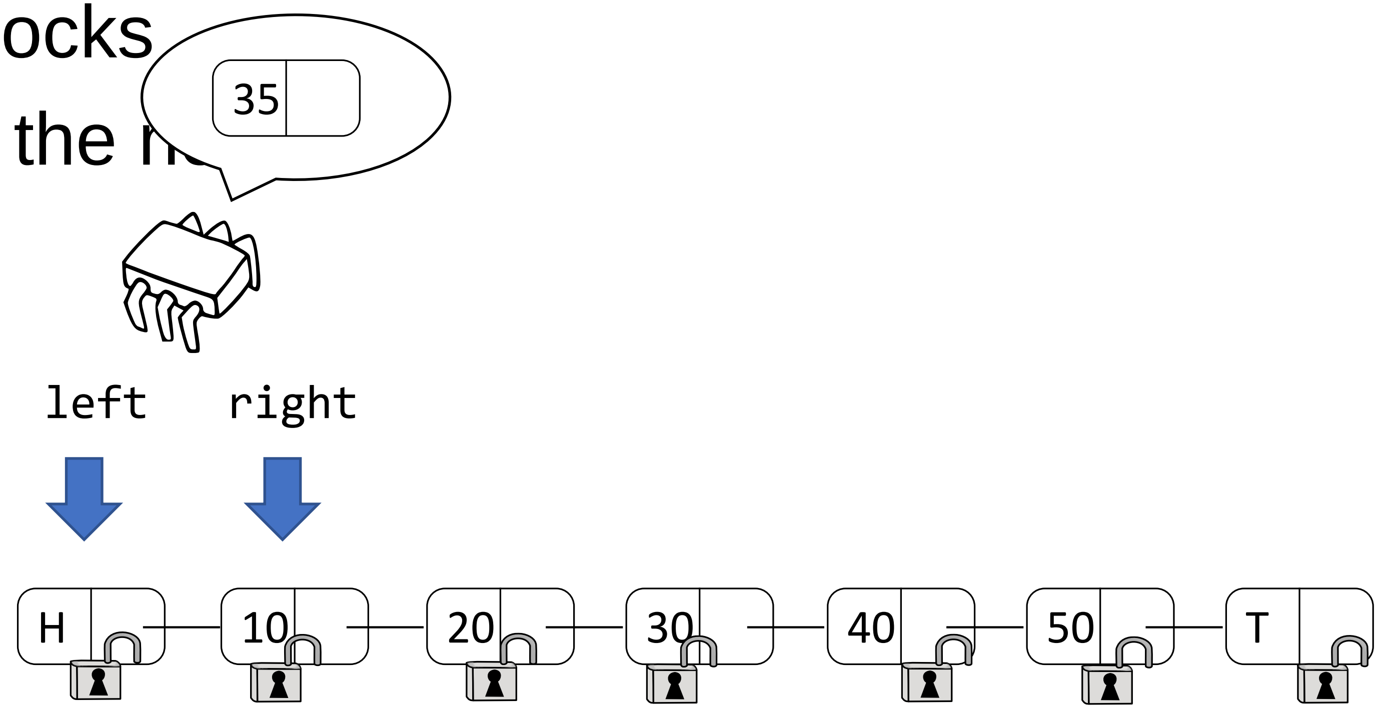
Locks + Optimism

- Use one lock per node
- Move "marked" to a dedicated field



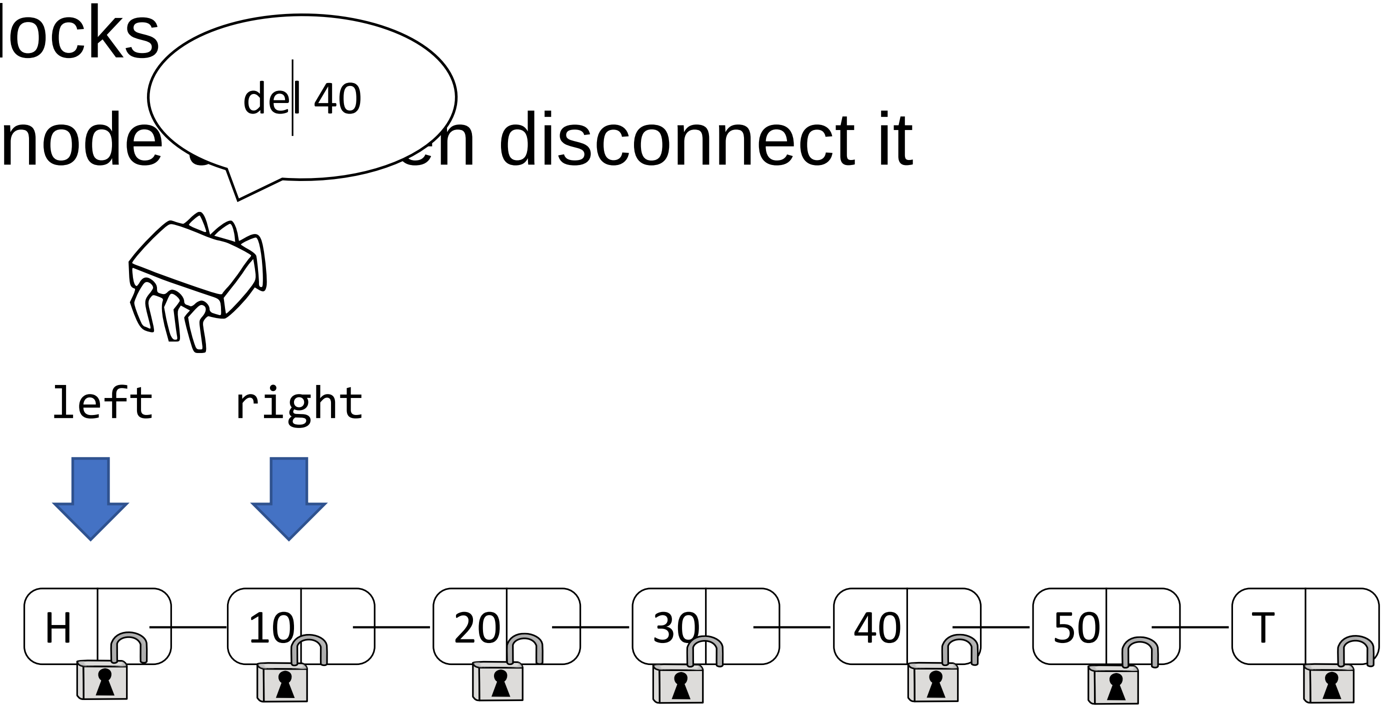
Locks + Optimism (insert)

- Use one lock per node
- Move "marked" to a dedicated field
- Find left and right without taking locks!
- Take locks
- Insert the n



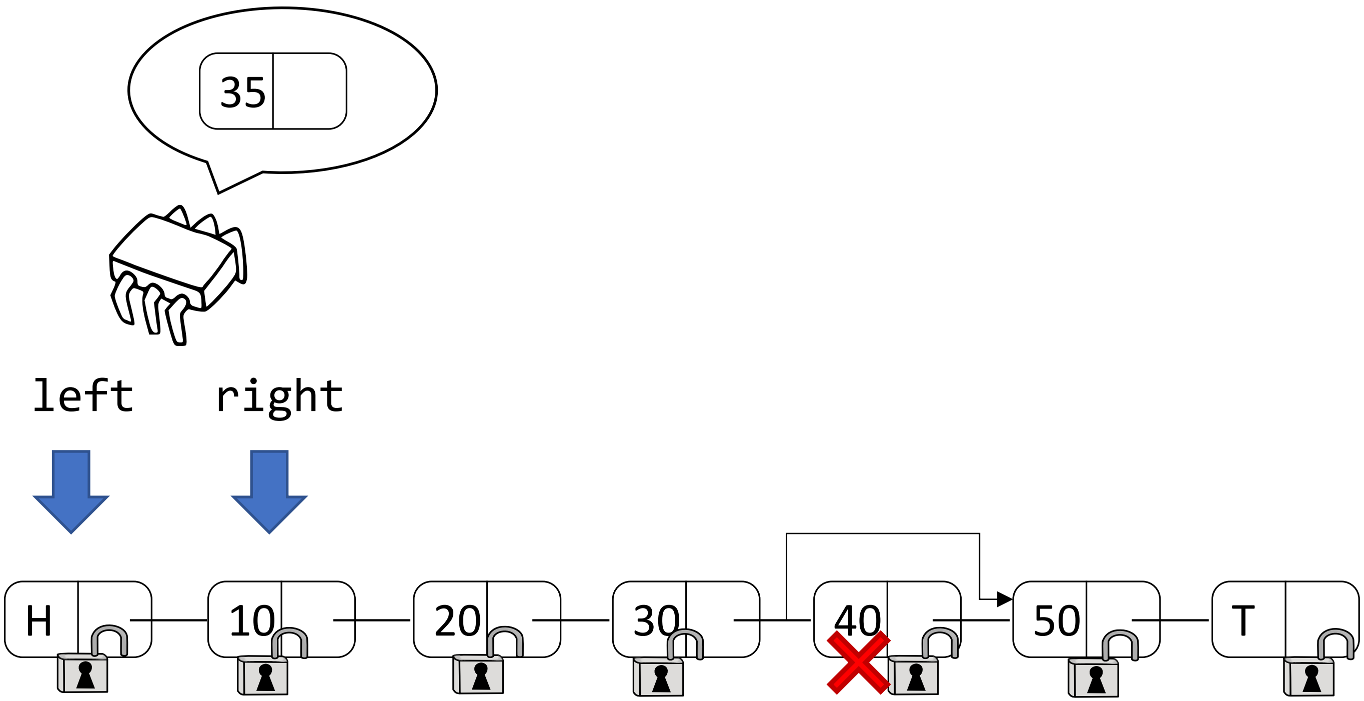
Locks + Optimism (delete)

- Use one lock per node
- Move “marked” to a dedicated field
- Find left and right without taking locks!
- Take locks
- Mark node … disconnect it



Locks + Optimism (delete)

- Why "optimistic"? Do work (search) and hope nothing wrong happens!
- What could go wrong?



Locks + Optimism (delete)

- Why “optimistic”? Do work (search) and hope nothing wrong happens!
- What could go wrong?
  - Left and/or right being marked
  - Left and right not adjacent
- How to solve?
- Validation of search results:
  - Left unmarked
  - Right unmarked
  - Left.next = right

Locks + Optimism (delete)

- Why “optimistic”? Do work (search) and hope nothing wrong happens!
- What could go wrong?
  - Left and/or right being marked
  - Left and right not adjacent
- How to solve?
- Validation of search results:
  - Left unmarked
  - Right unmarked
  - Left.next = right

Locks + Optimism = Lazy List

- What about correctness?
- What about progress?

Can we do better?

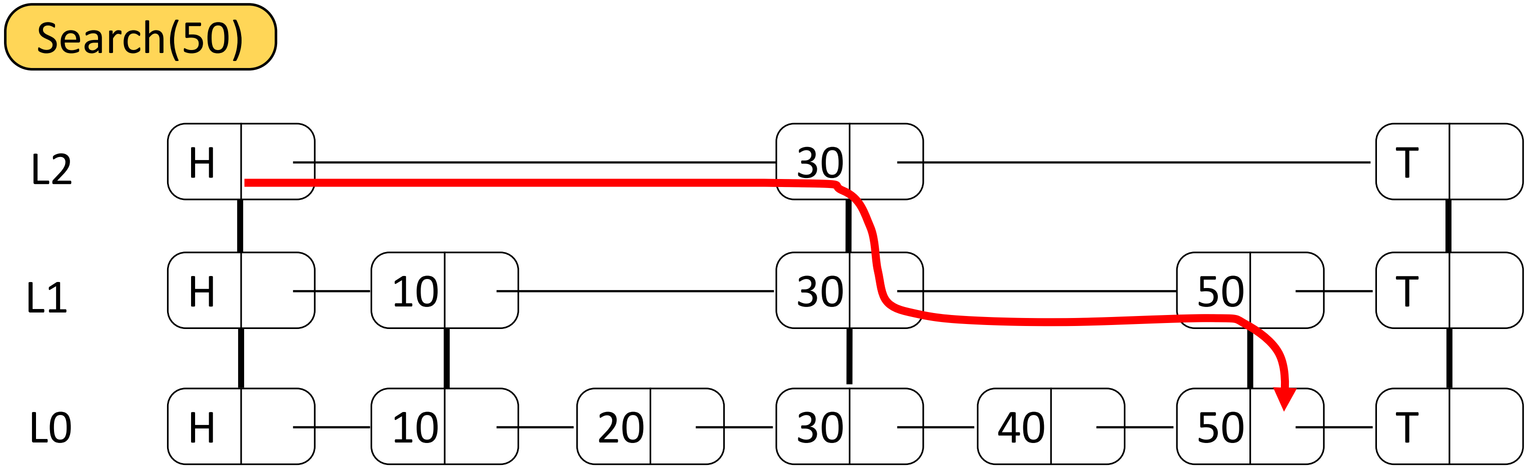
- Costs: O(n)
- Starting from scalable “simple” set implementation we can build faster set implementations
  - Hash table: O(1)
    - Array of buckets
    - Buckets are concurrent ordered-list based sets
- We know that a search in an ordered set could be more efficient O(log(n))
- How?

Skip list [Pugh 1990]

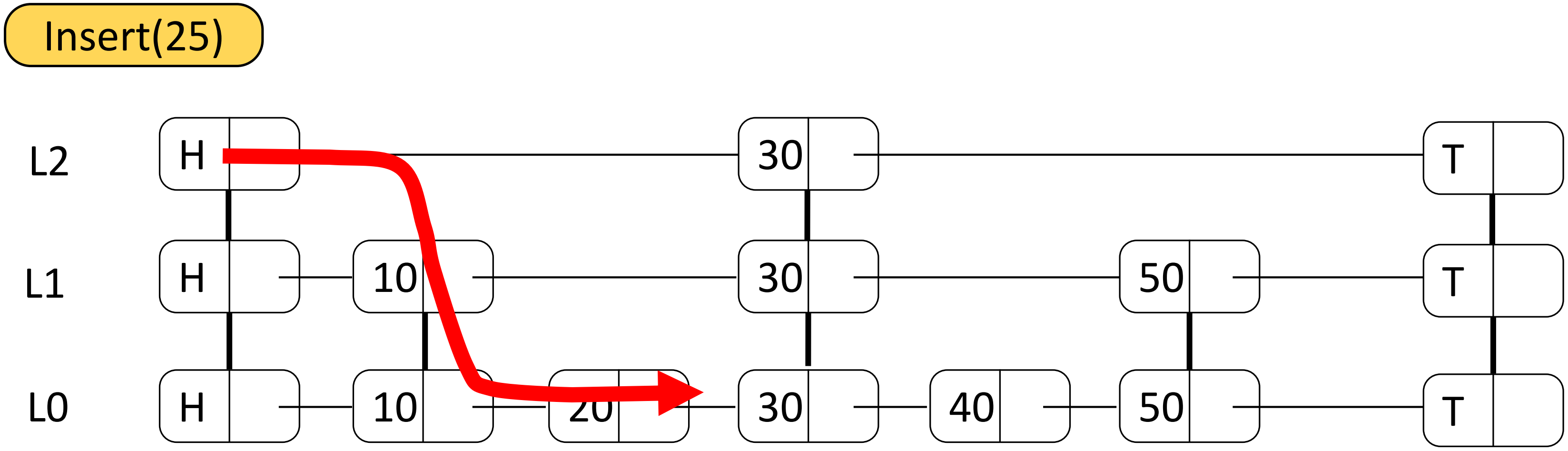
- Generalization of sorted linked lists
- Randomized data structure
- Costs: O(log(n))

- Idea:
  1. Maintain a core sorted linked list L0
  2. Use additional sorted linked lists Li such that:
     1. Li ⊂ Li-1
     2. |Li| ≈ |Li-1|/2
  3. Searches use lists in decreasing order

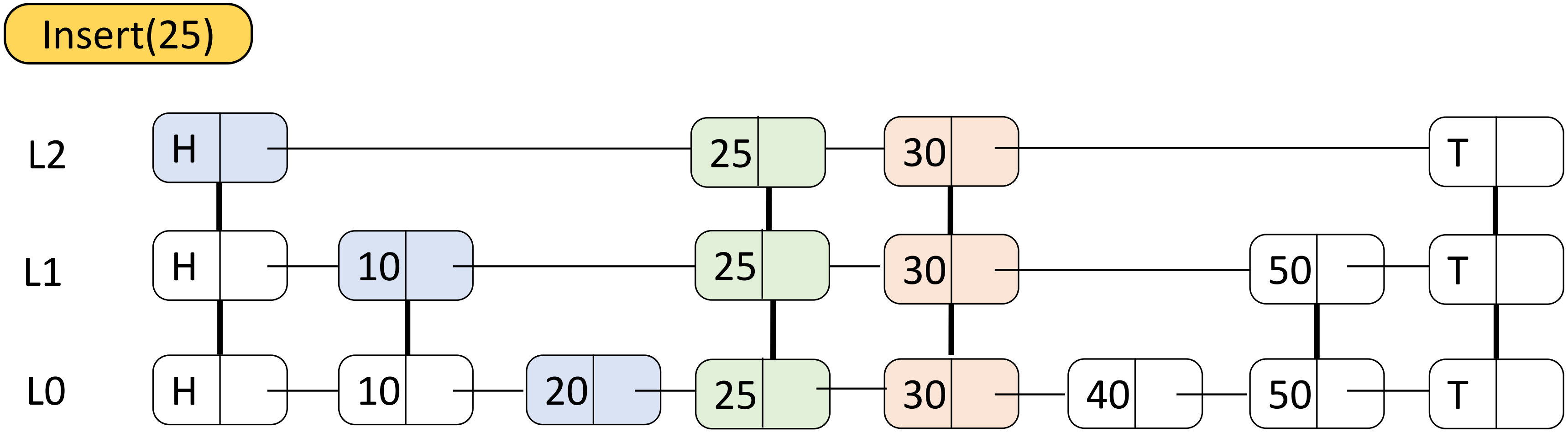
Skip list [Pugh 1990]



Skip list [Pugh 1990]



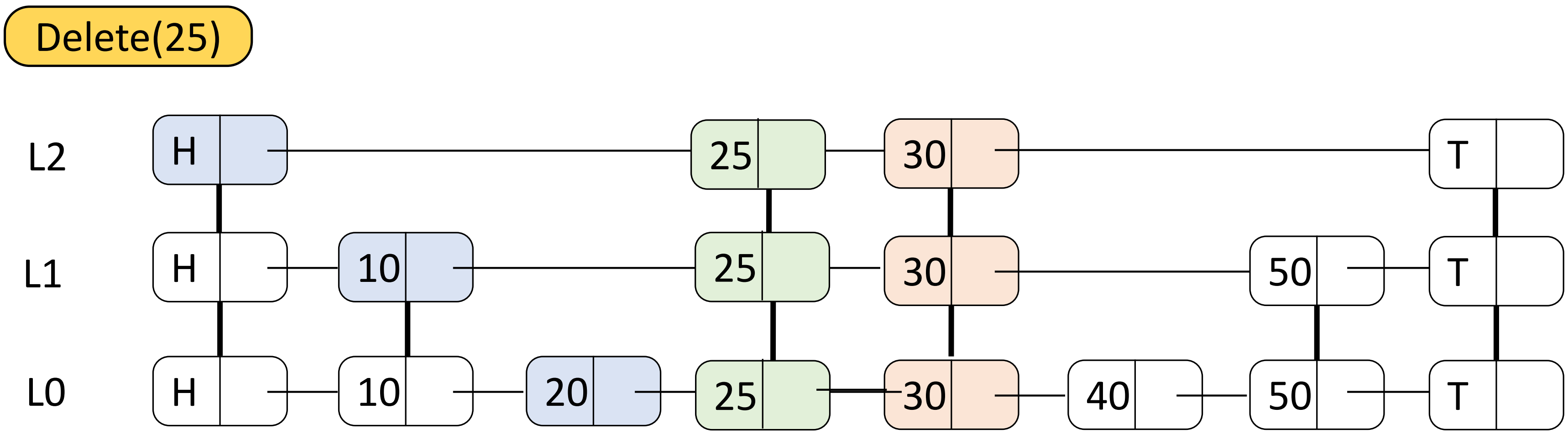
Skip list [Pugh 1990]



Should I insert 25 at L1? Flip a coin!

Should I insert 25 at L2? Flip a coin!

Skip list [Pugh 1990]



Skip list [Pugh 1990]

- How many (expected) keys for each level?
- L0 = N
- L1 = N/2
- L2 = N/4

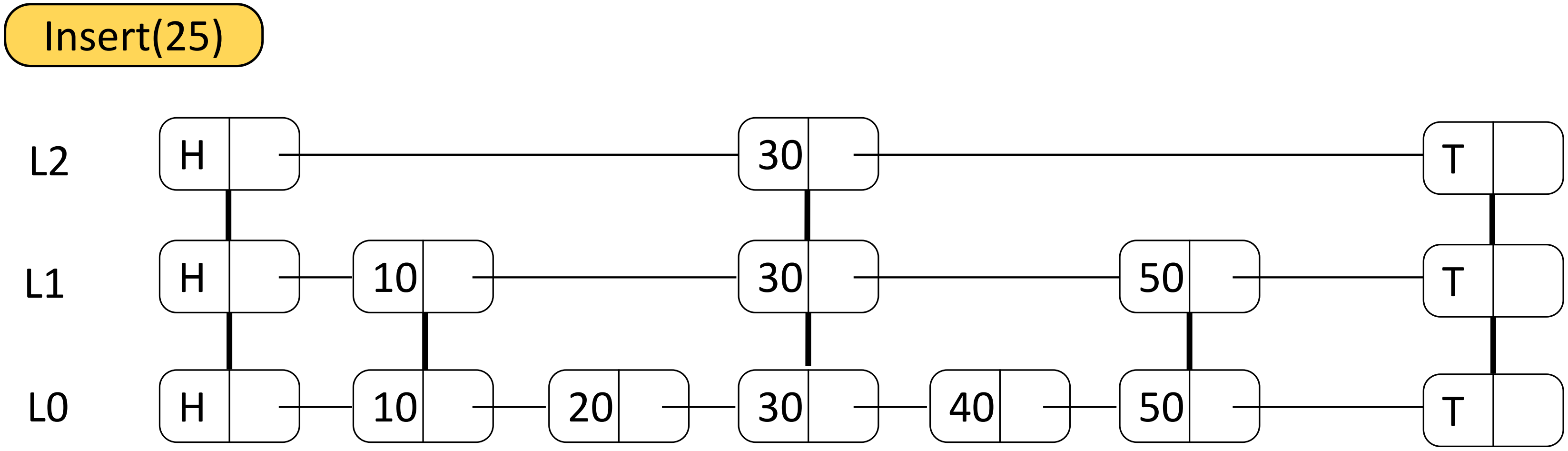
…

- L(logN) = 1

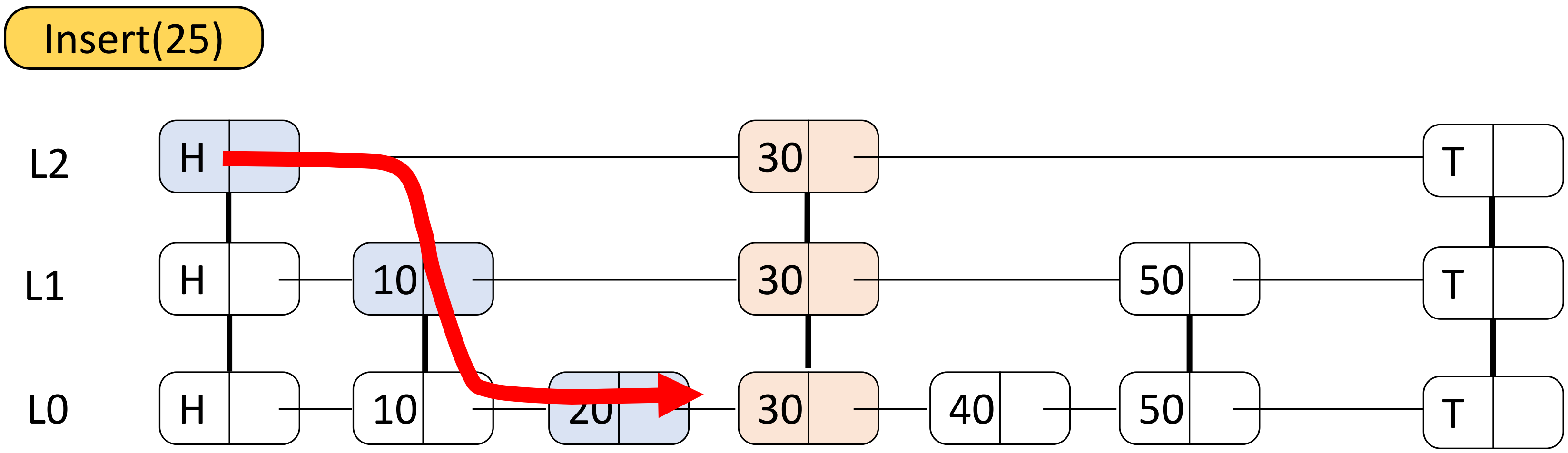
Skip list [Pugh 1990]

- How many steps per level?

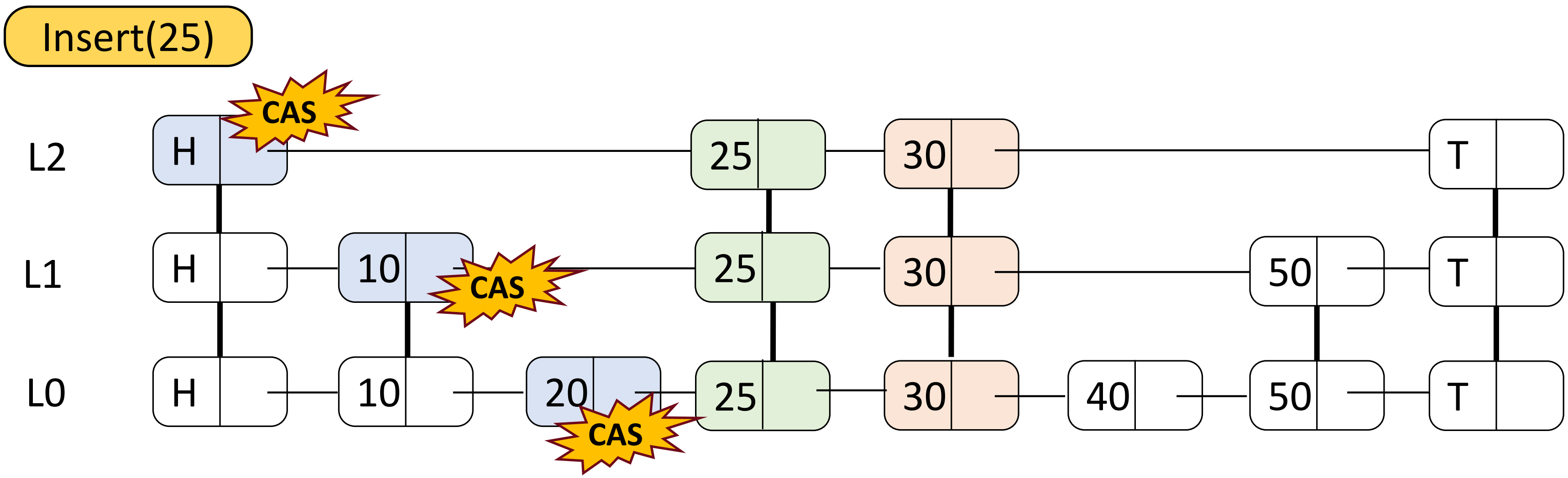
Non-blocking Skip list [Fraser2004]



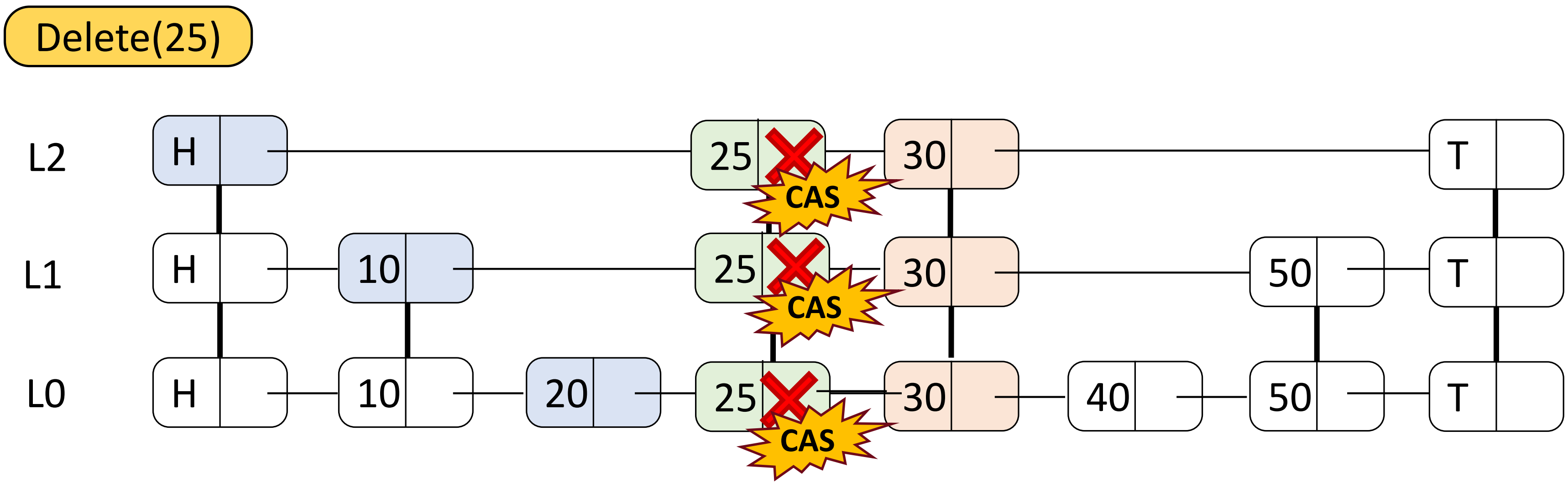
Non-blocking Skip list [Fraser2004]



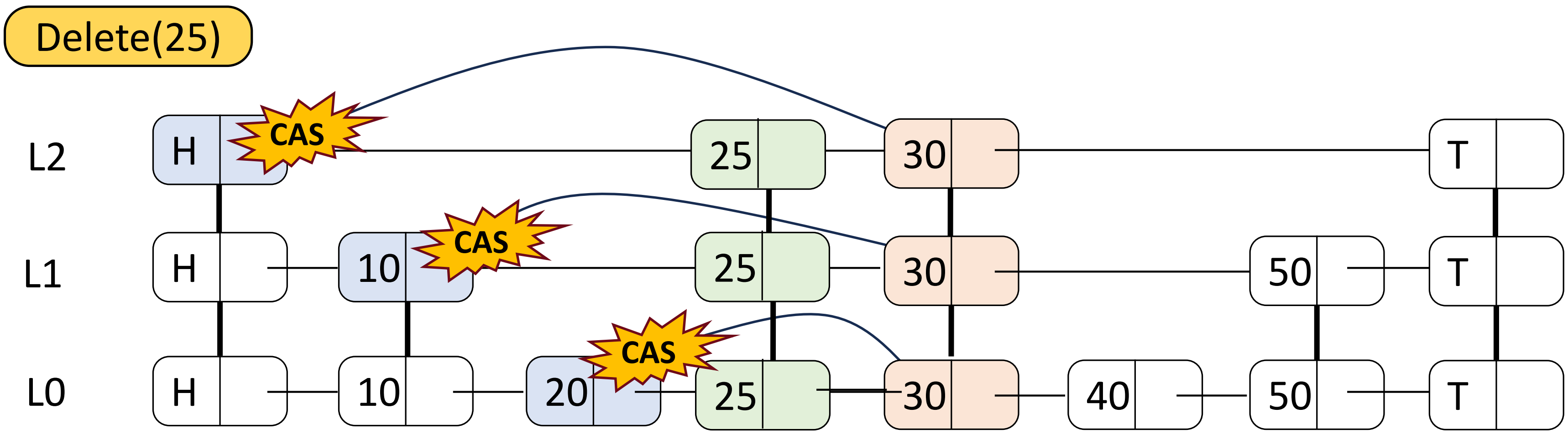
Non-blocking Skip list [Fraser2004]



Non-blocking Skip list [Fraser2004]



Non-blocking Skip list [Fraser2004]

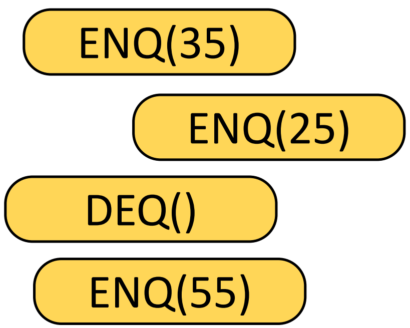


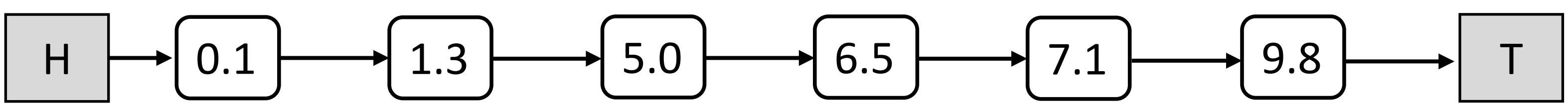
Concurrent Data Structures: **Priority queues**

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

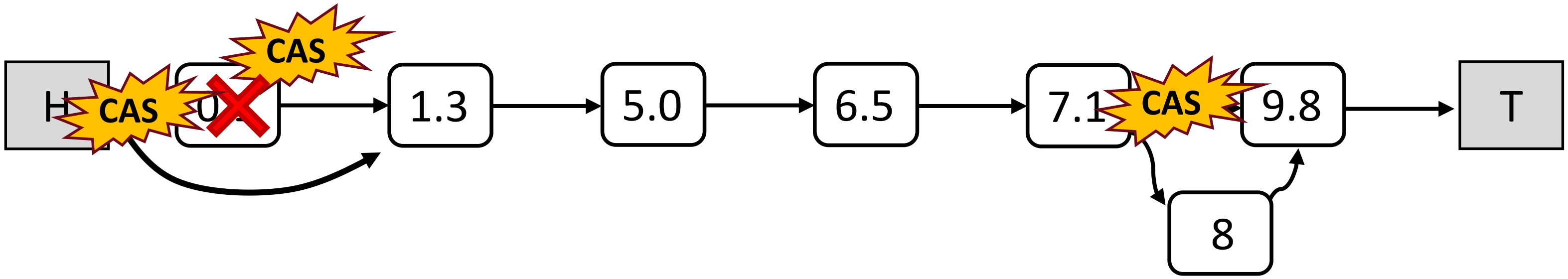
This is a huge simplification. Tipically they are implemented as skip-lists (log(n)) or calendar queues (O(1))



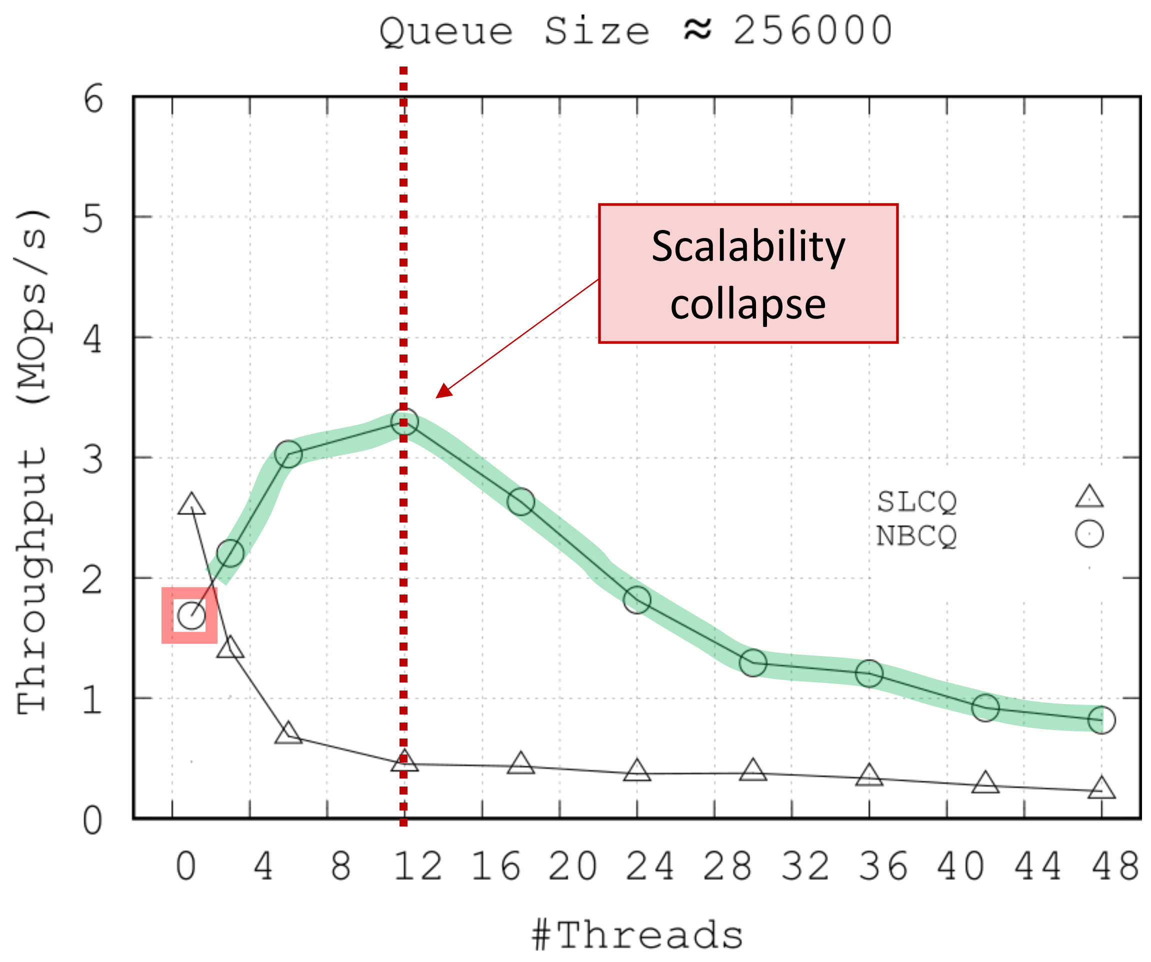


Priority queue – Attempt 1

- 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?

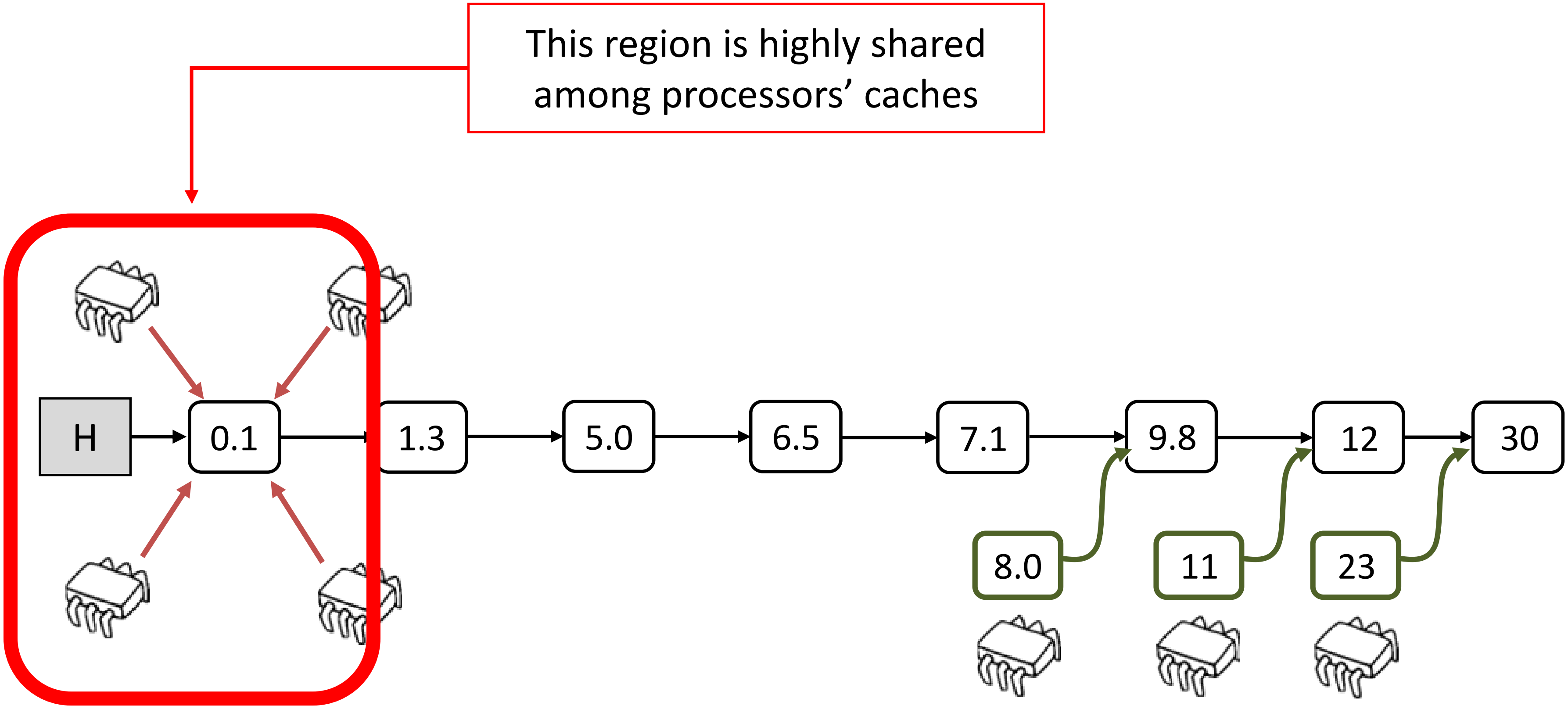


Priority queue – Attempt 1



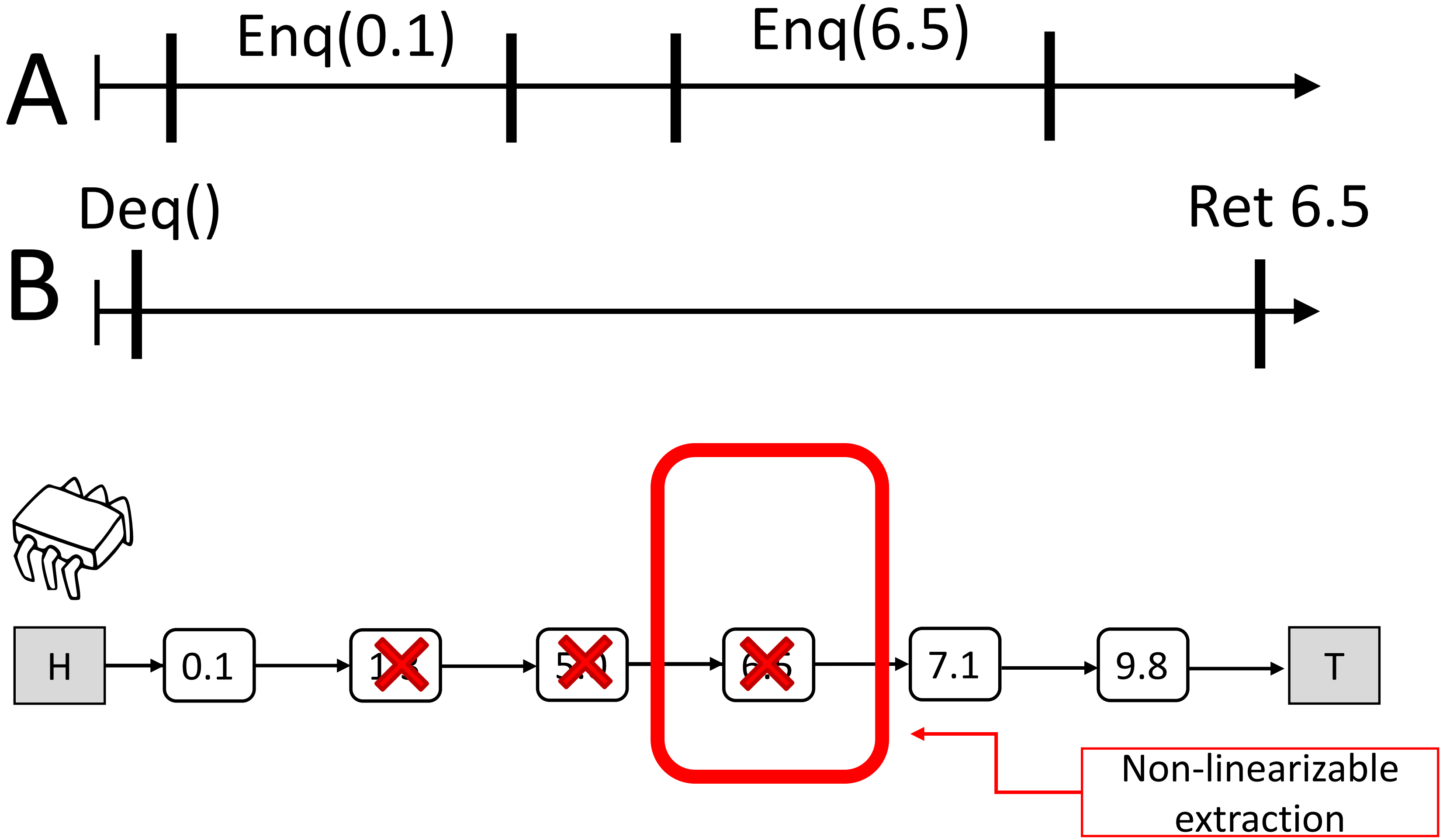
Priority queues: an inherently "sequential" semantic

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



Lazy deletion within priority queues

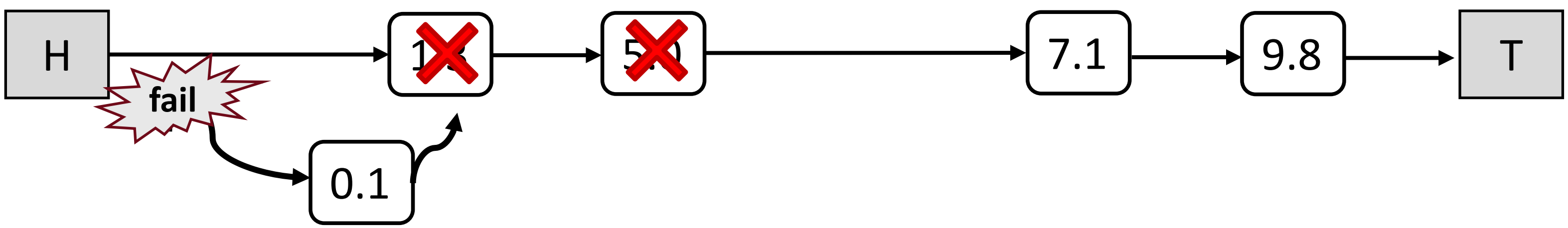
- If we use lazy deletion "as is", we might obtain non-linearizable extractions



Correct lazy deletion within priority queues

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

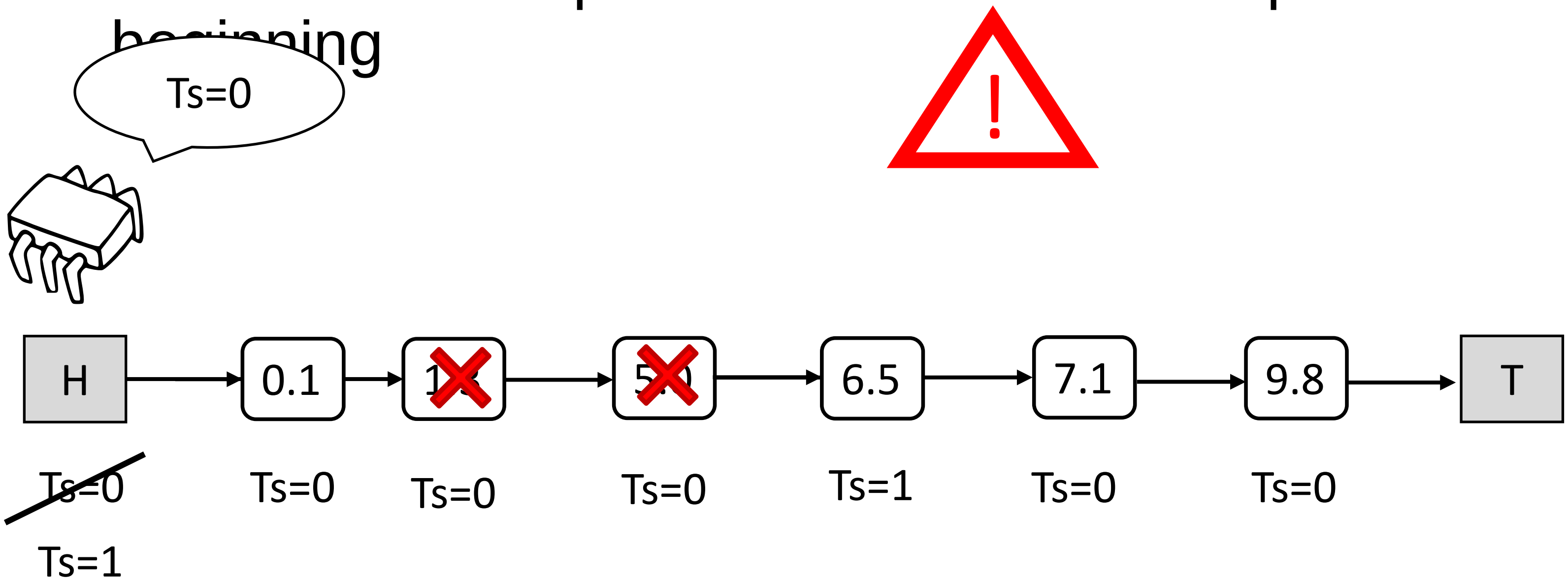
1. Move the logical mark of a node in the field "`next`" of its predecessor



Correct lazy deletion within priority queues

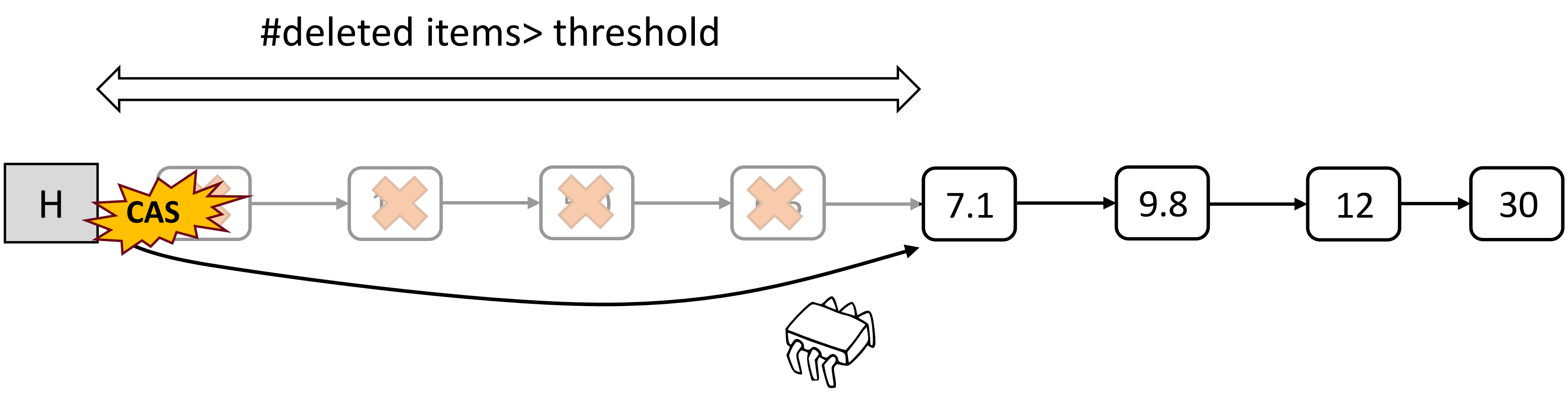
- To implement correct extractions with lazy deletions there are two main approaches

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

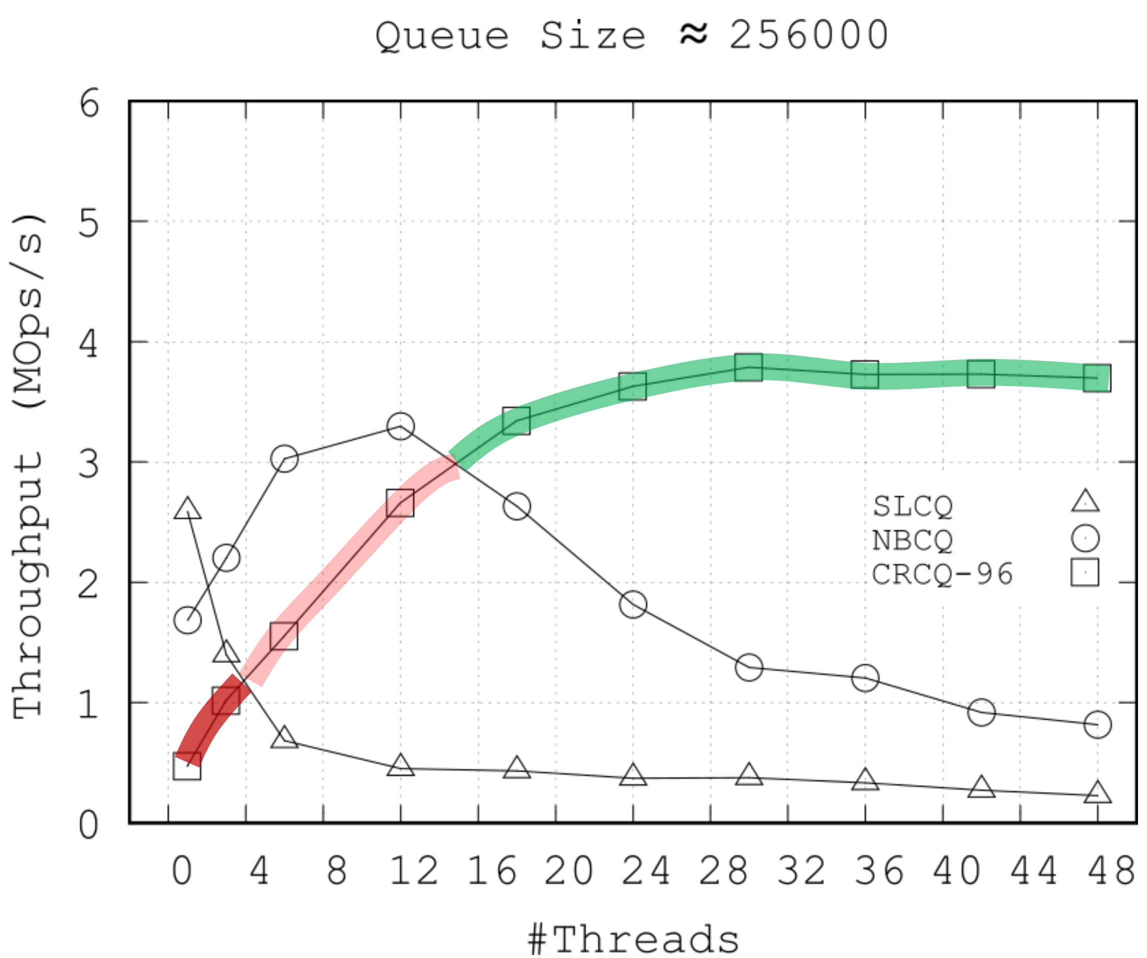


PQ – Attempt 2 - Introducing Conflict Resiliency

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



Priority queue – Attempt 2



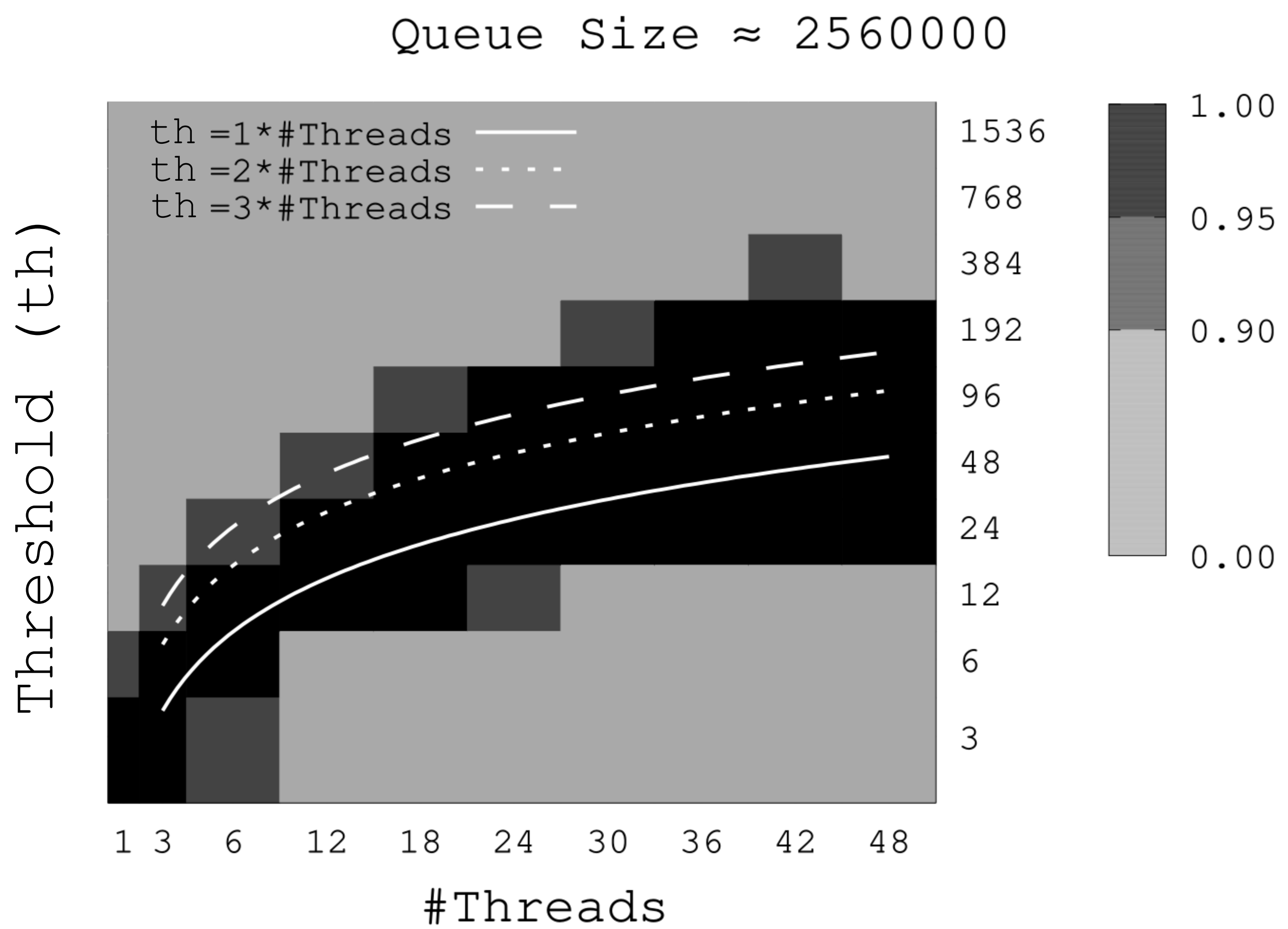
On the conflict resiliency trade off

- The number of steps per dequeue and costs of housekeeping are **<u>dependent</u>**:

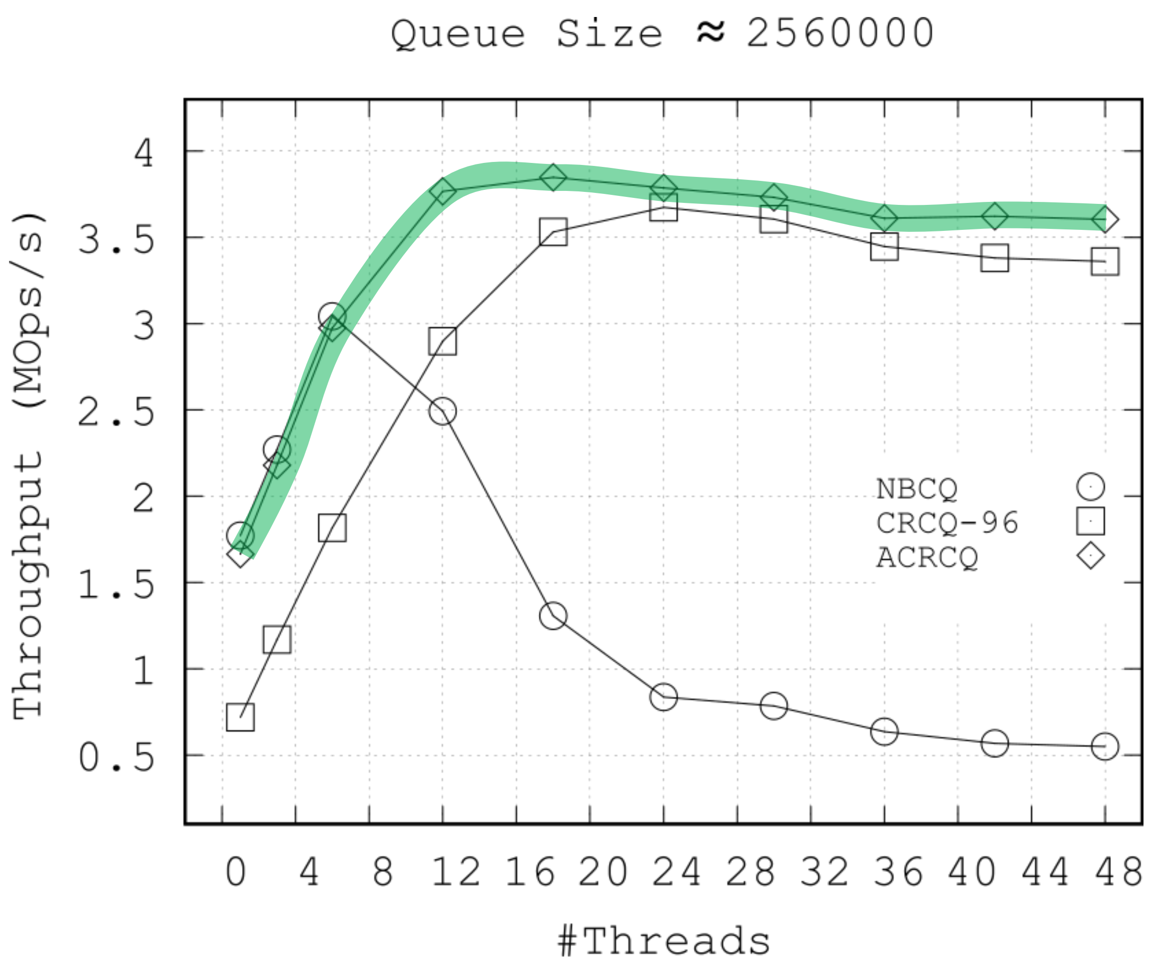
↑ **THRESHOLD** ⇒ ↑ **READ LATENCY** ☹ **and** ↓ **RMW IMPACT** ☺

↓ **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)

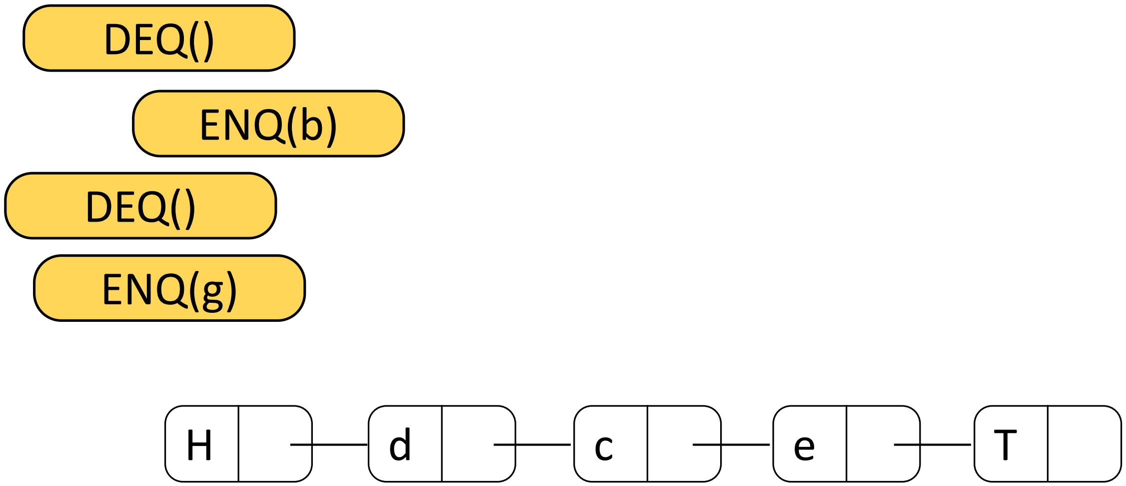
Why linearizable non-blocking algorithms?

- 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

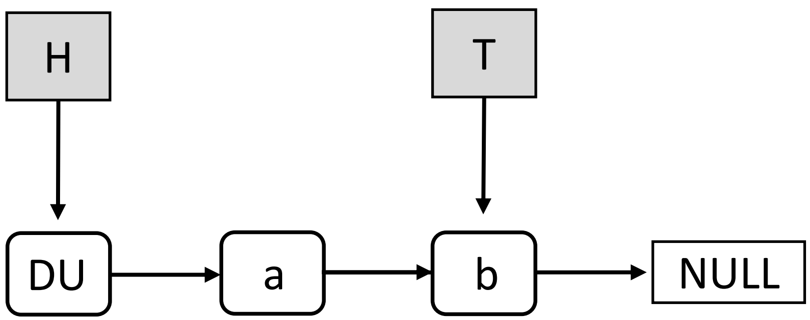
FIFO queue implementation

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



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



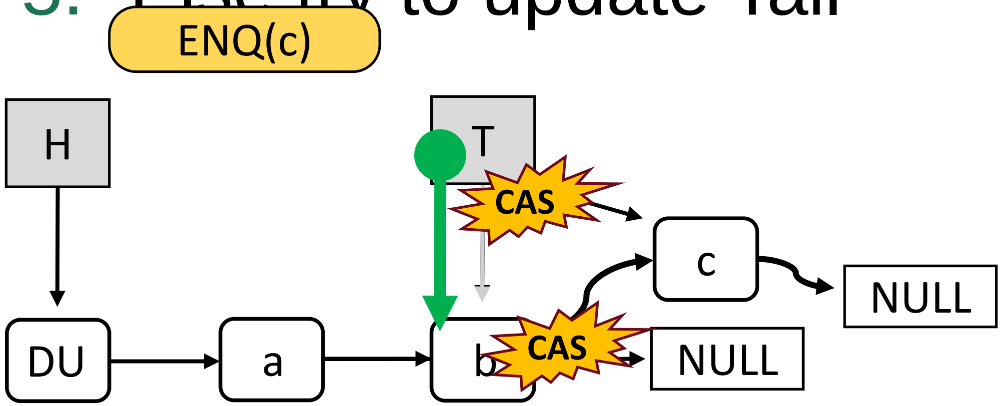
FIFO queue implementation

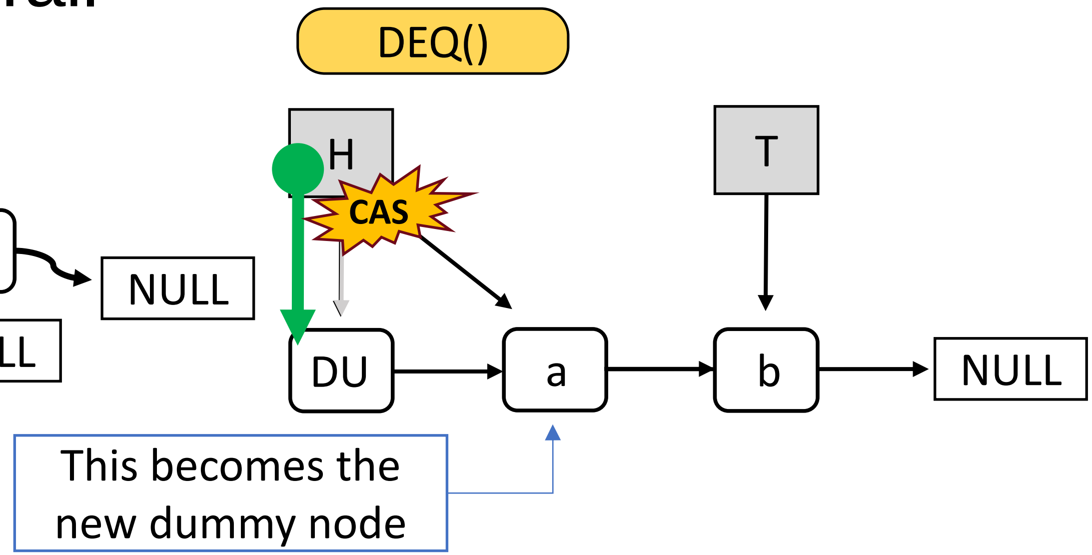
- Insert:

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:

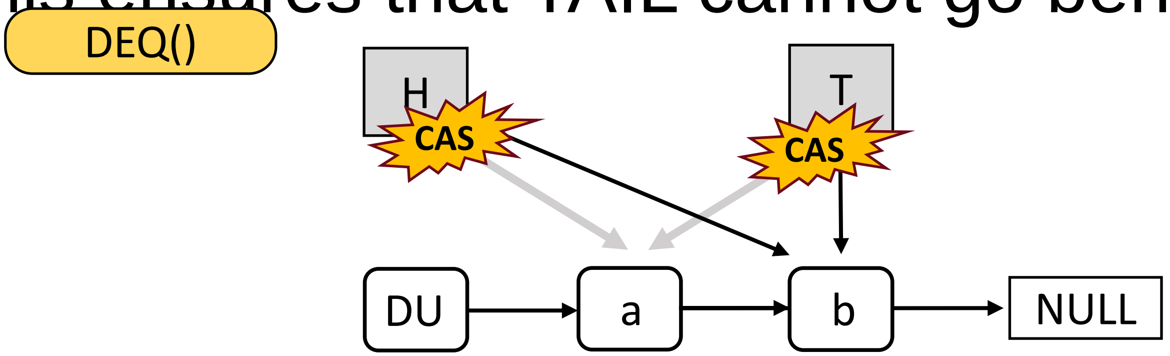
1. Get node pointed by head
2. Try to update head with its next
3. If KO goto 1





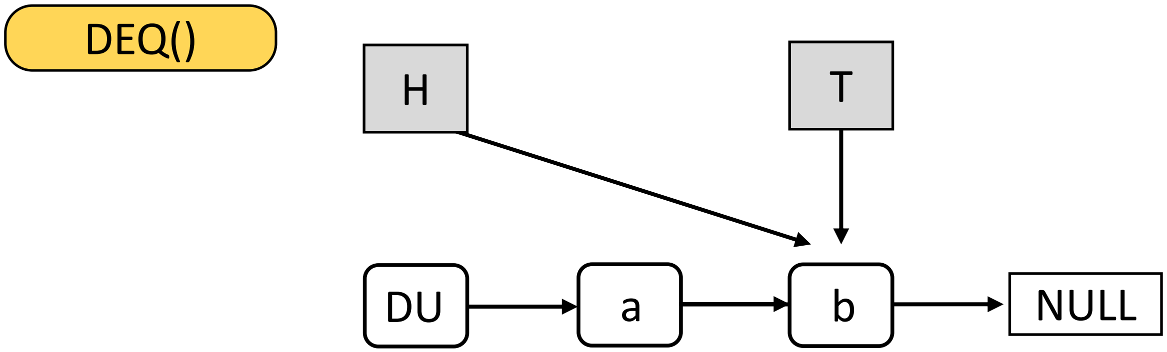
The whole story

- 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



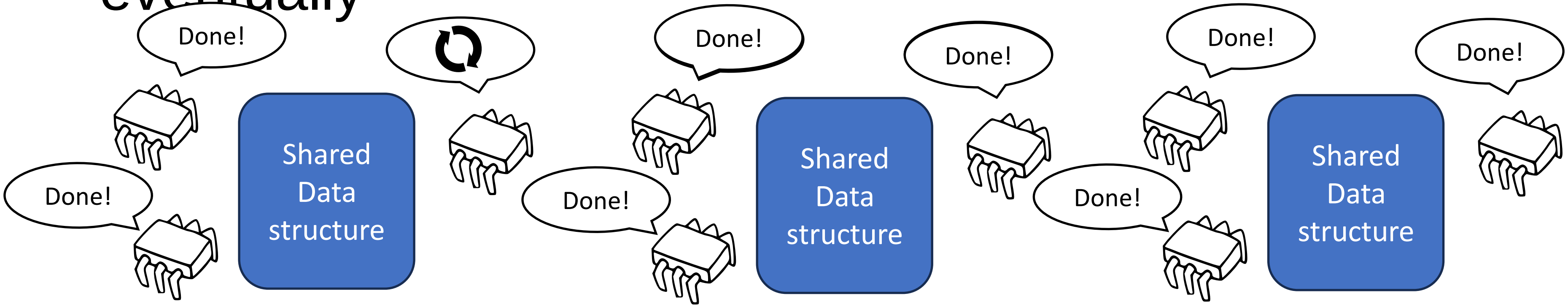
Emptiness condition

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



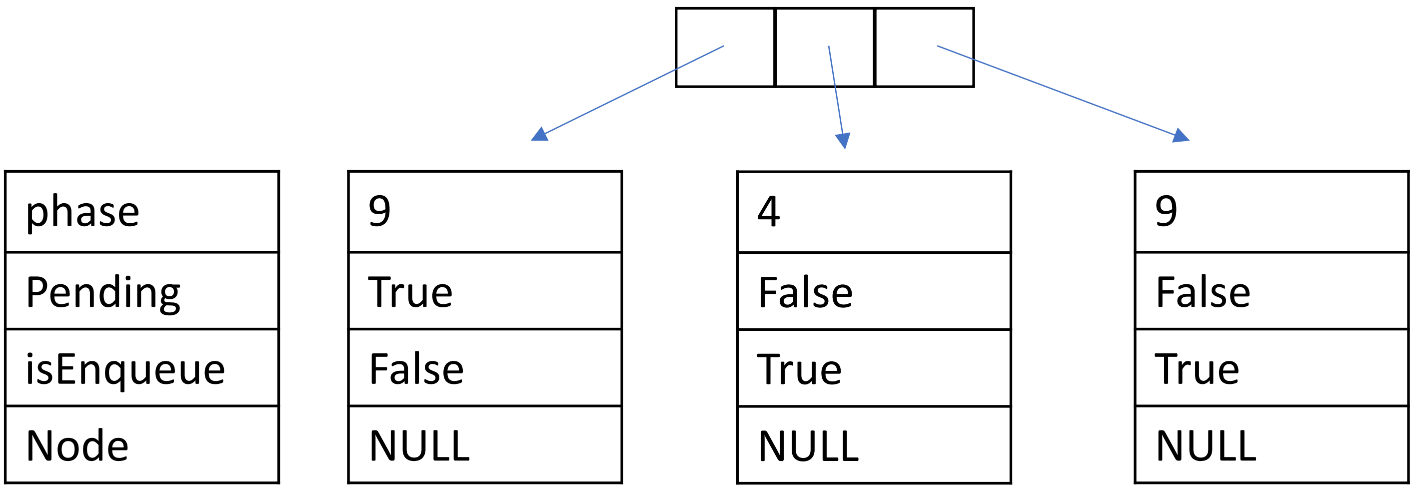
Wait-free FIFO queue

- 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



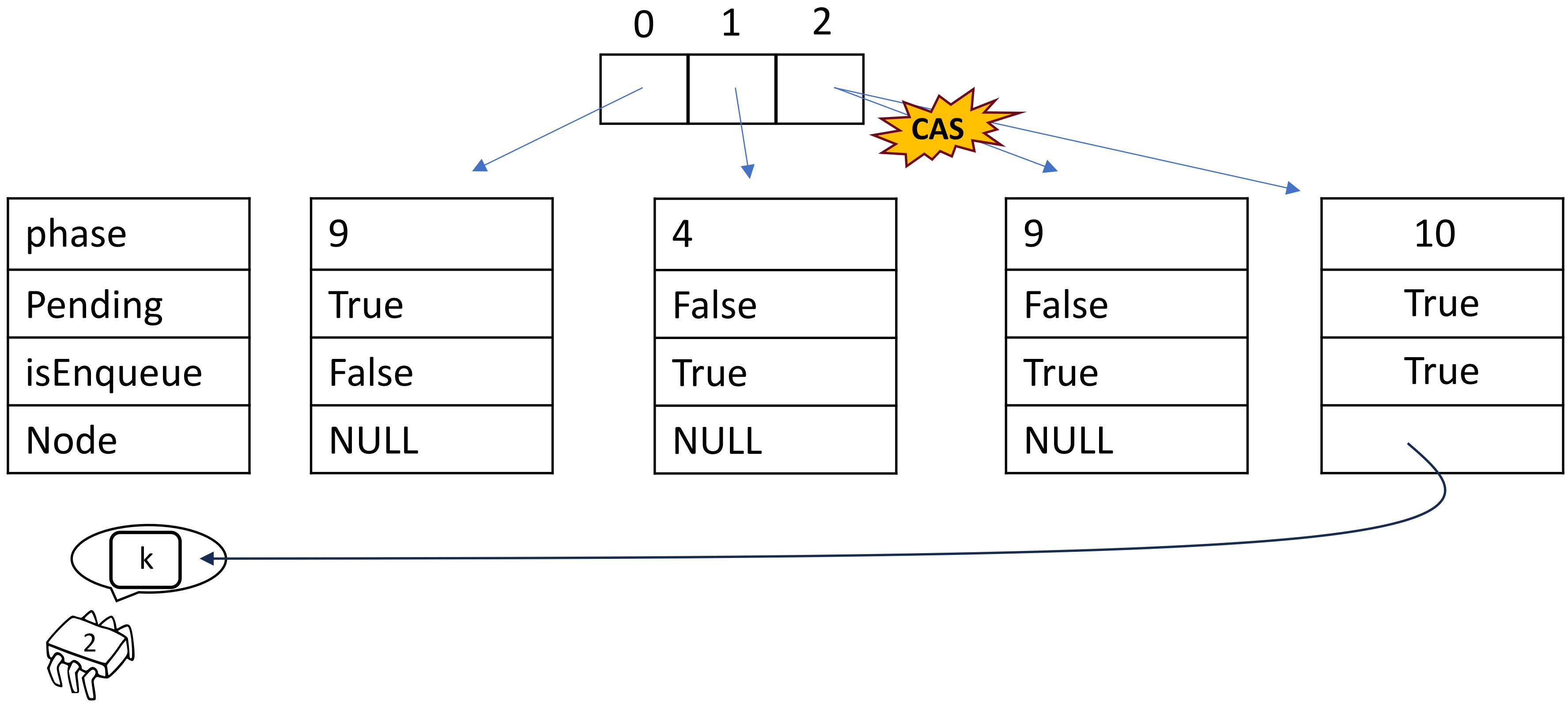
- Split operations on the linked list into 2 steps:
  1. Modify nodes for enqueue/dequeue (main step)
  2. Modify head/tail pointers (finishing step)

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

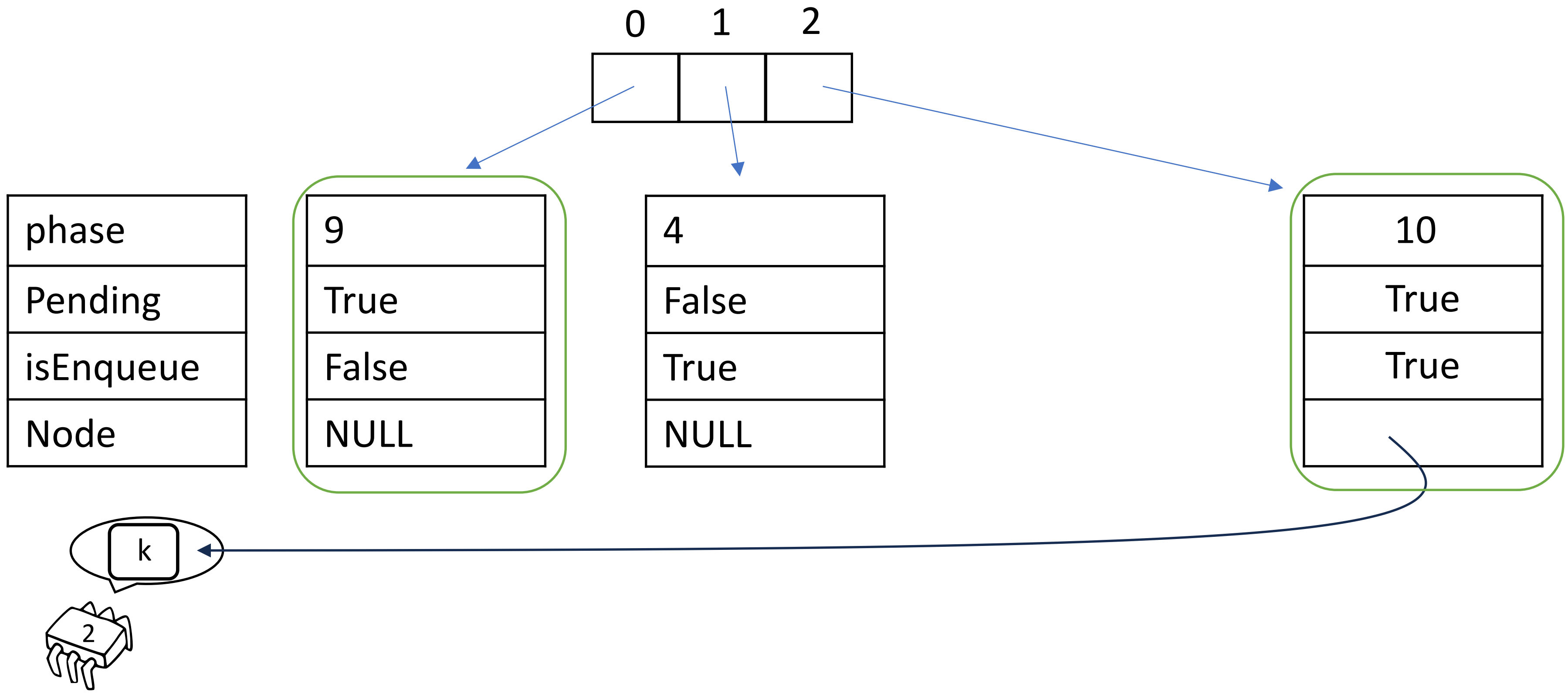
Wait-free FIFO queue

- Enqueue/Dequeue structure
  1. Publish op record



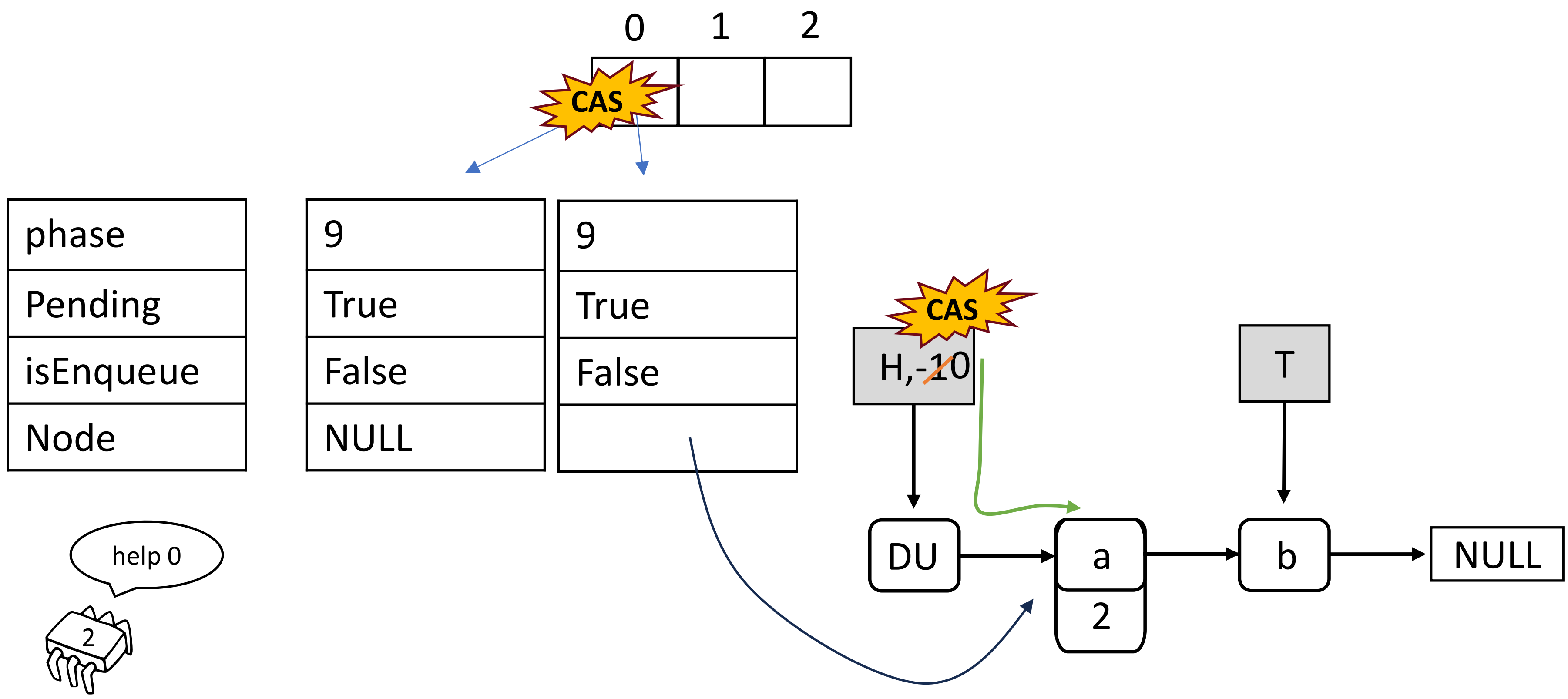
Wait-free FIFO queue

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



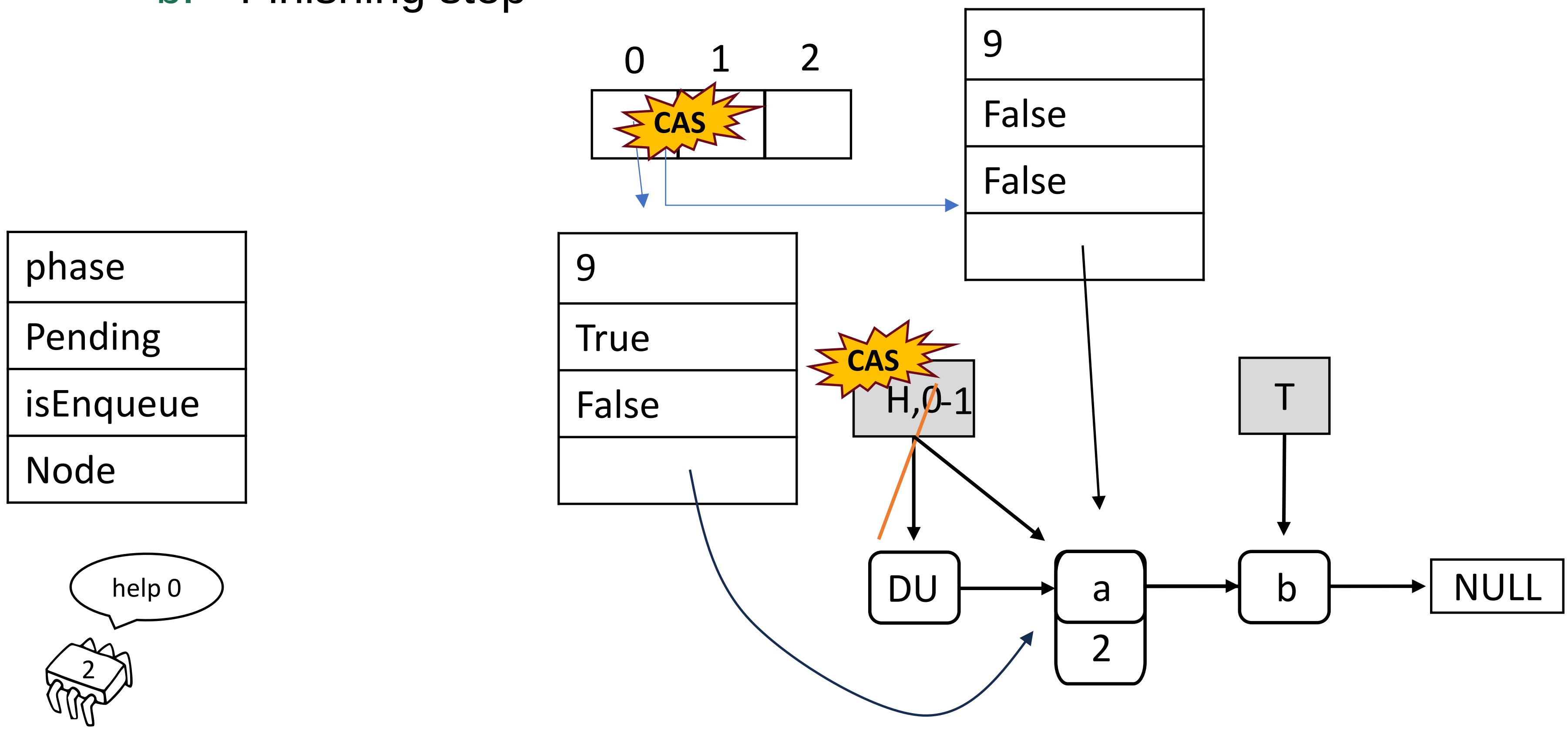
Wait-free FIFO queue

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



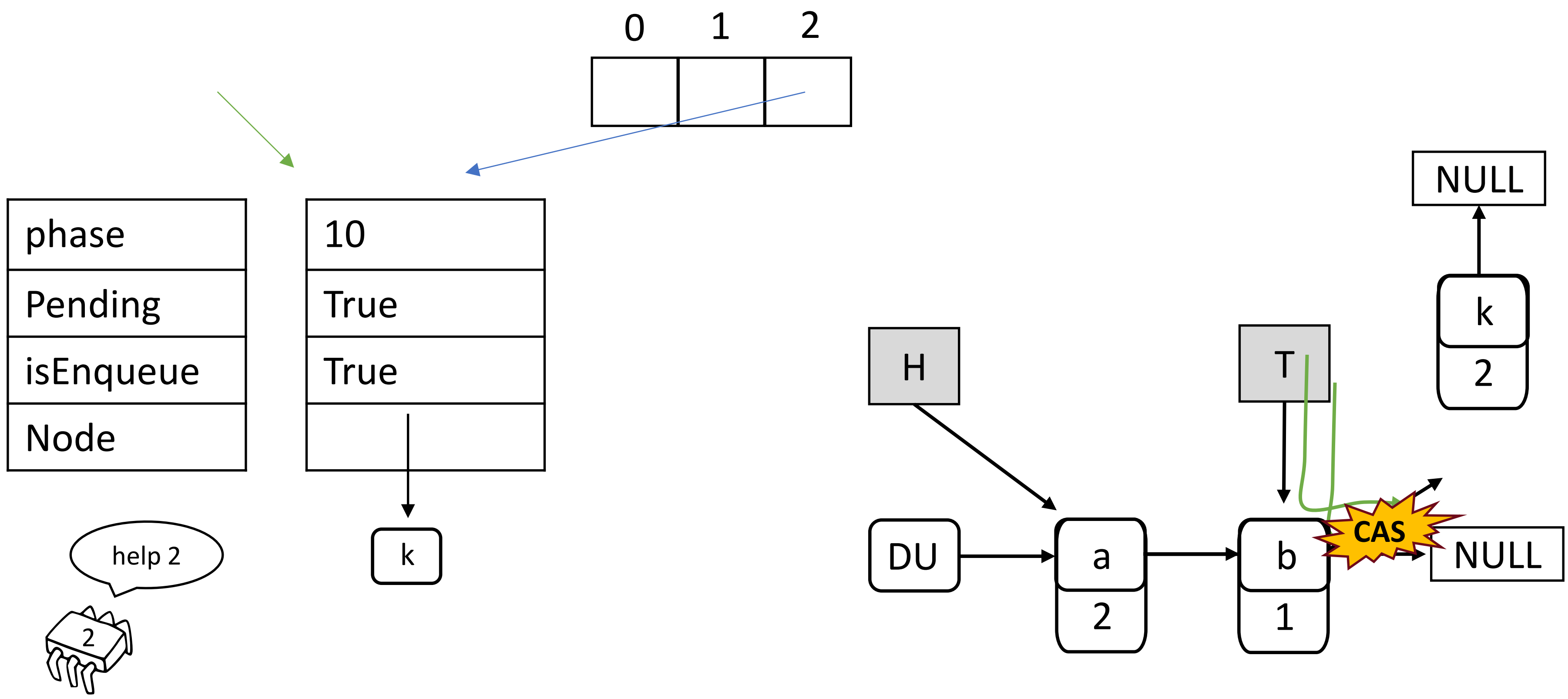
Wait-free FIFO queue

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



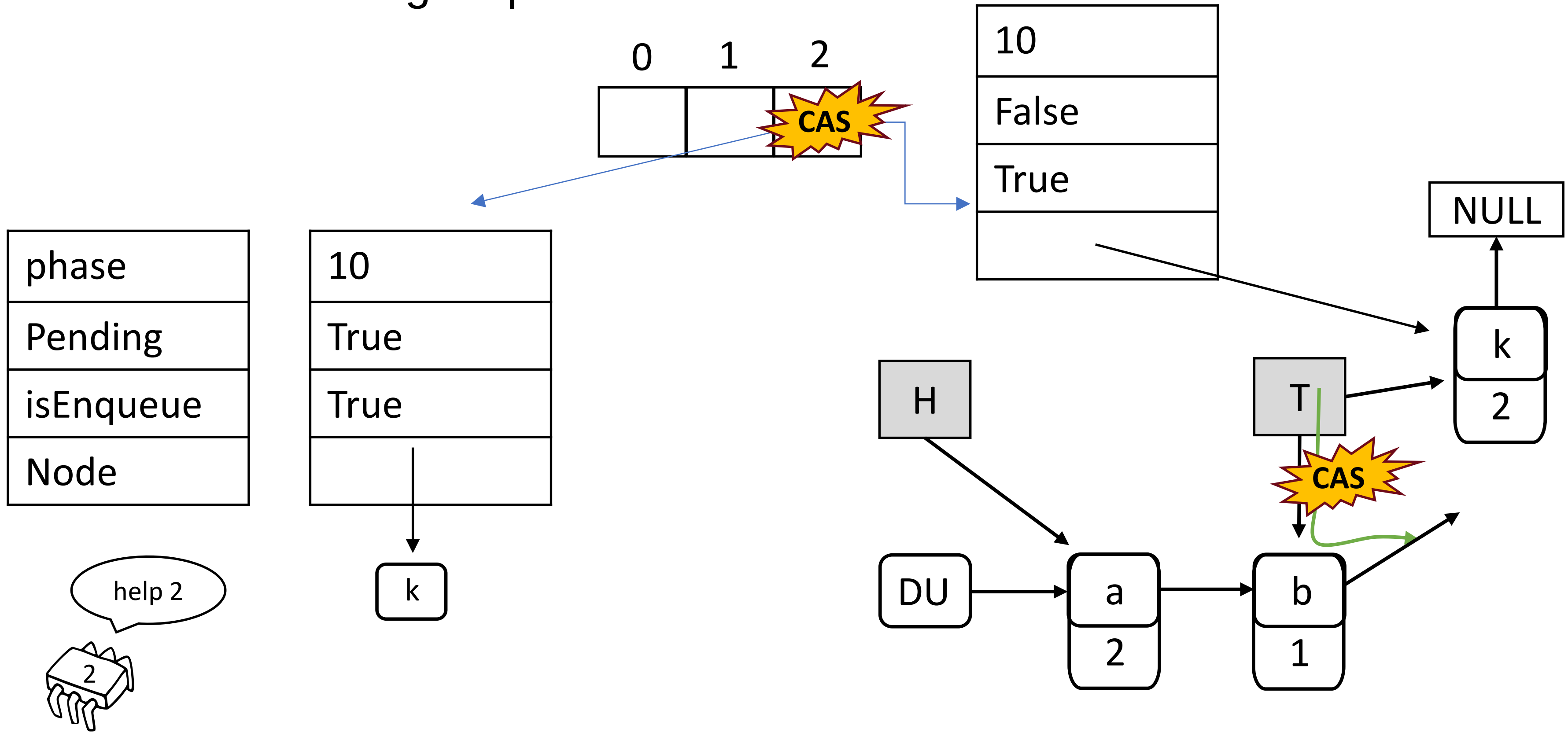
Wait-free FIFO queue

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



Wait-free FIFO queue

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

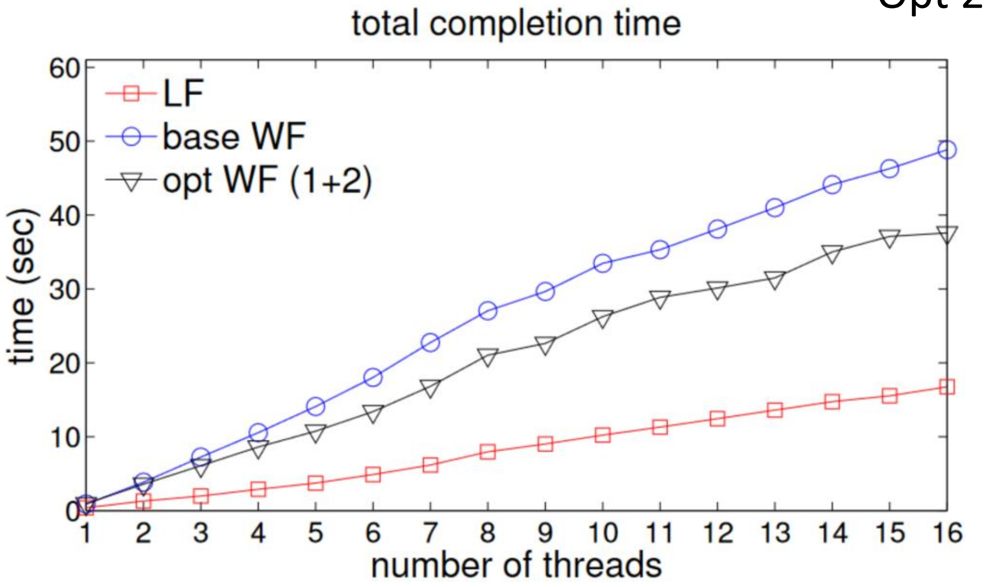


Wait-free FIFO queue

- 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

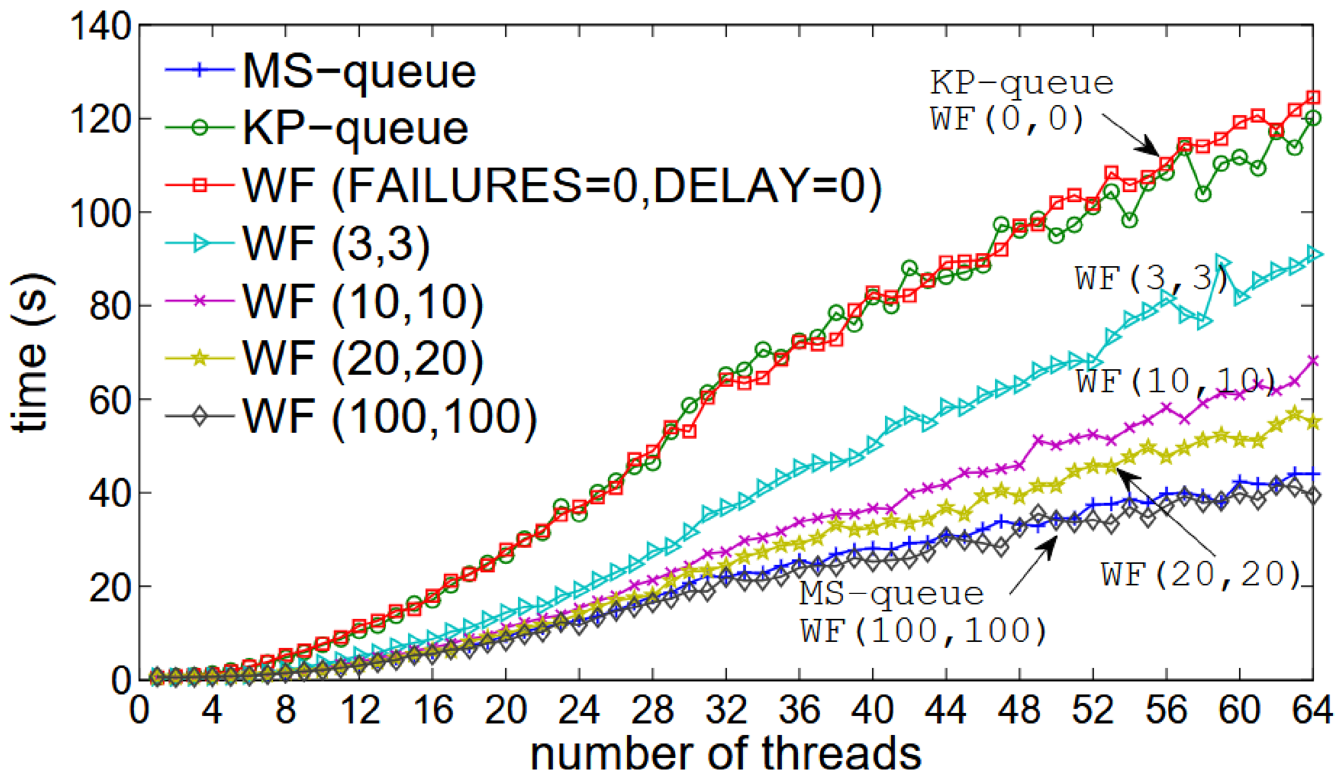
Opt 1: help only one pending op

Opt 2: use FAD to get phase num.



Fast Wait-free FIFO queue

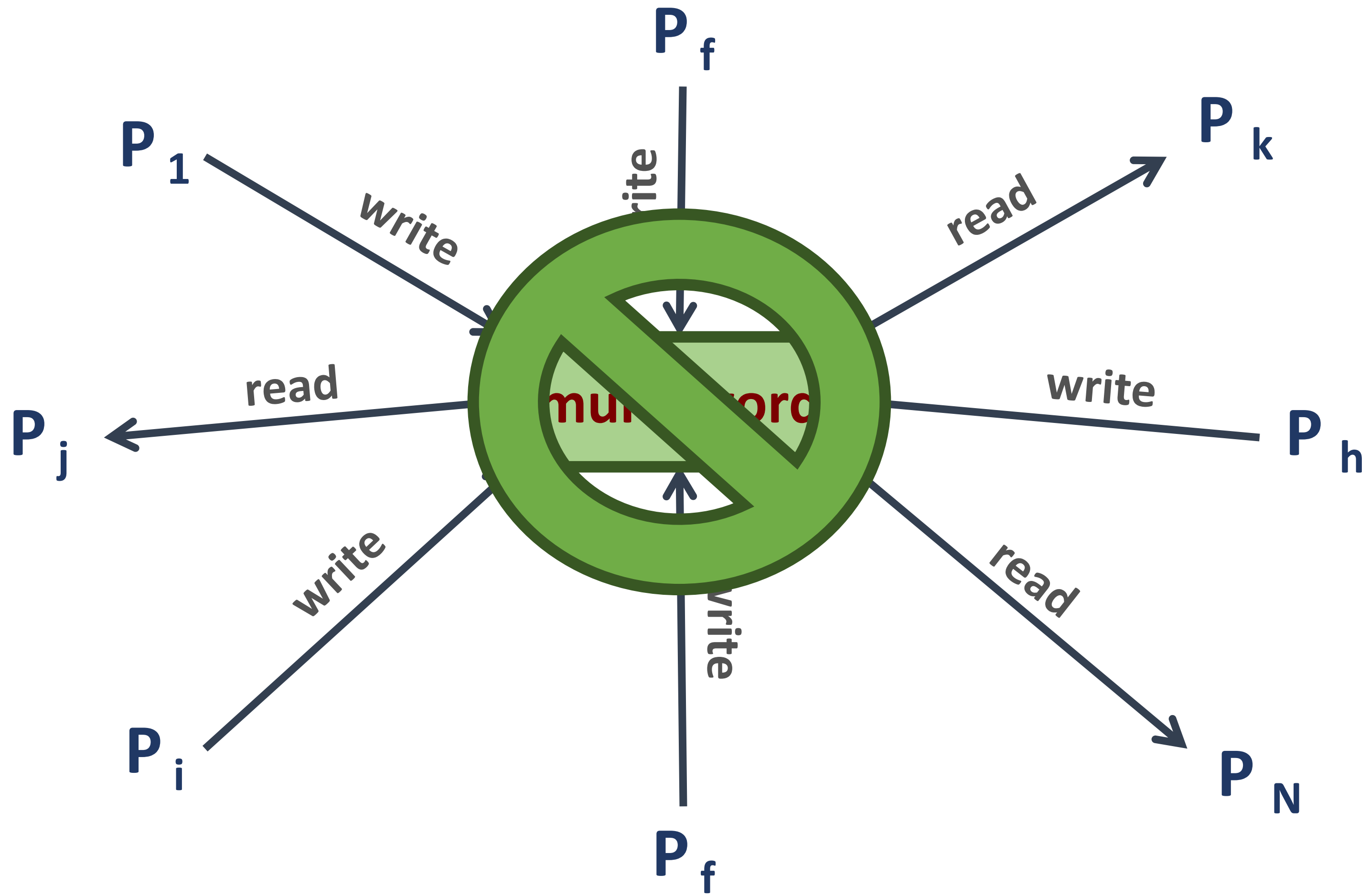
- Try with lock-free approach
- If starving, back-off to wait-free implementation



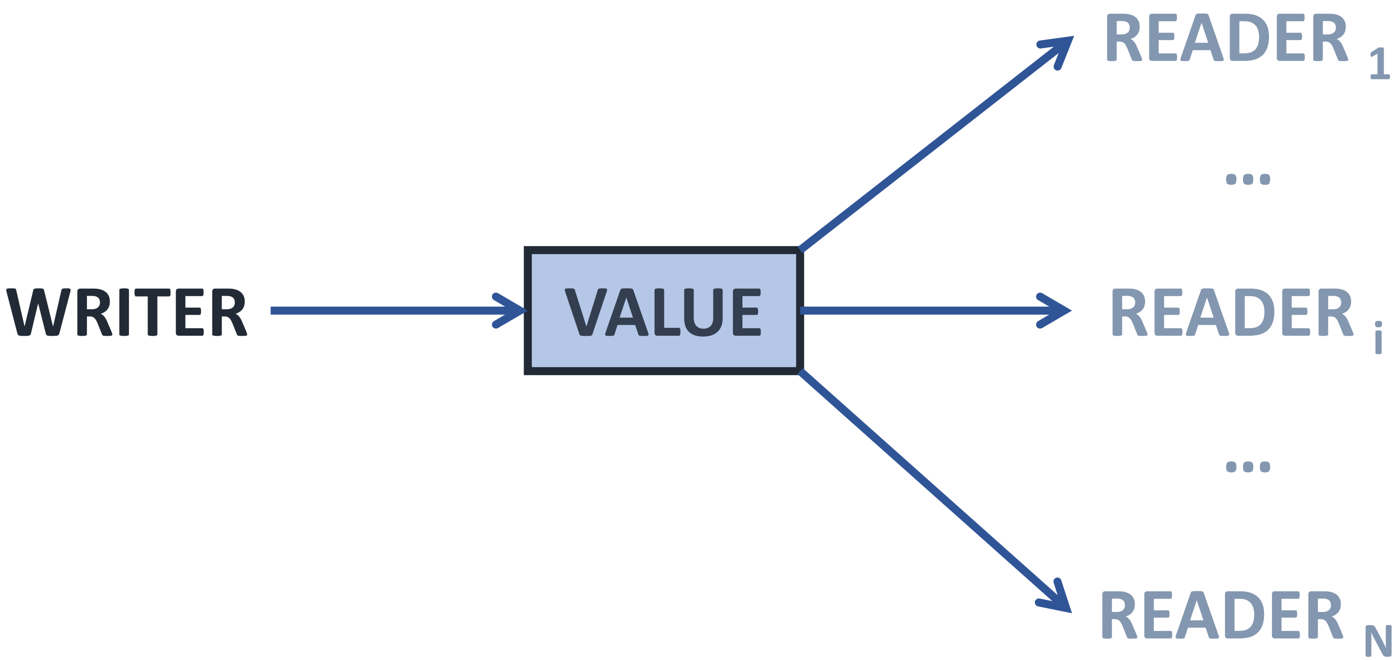
Concurrent Data Structures:

Atomic MRSW registers

Atomic MRSW Register



Atomic MRSW Register



Atomic MRSW Register - Peterson

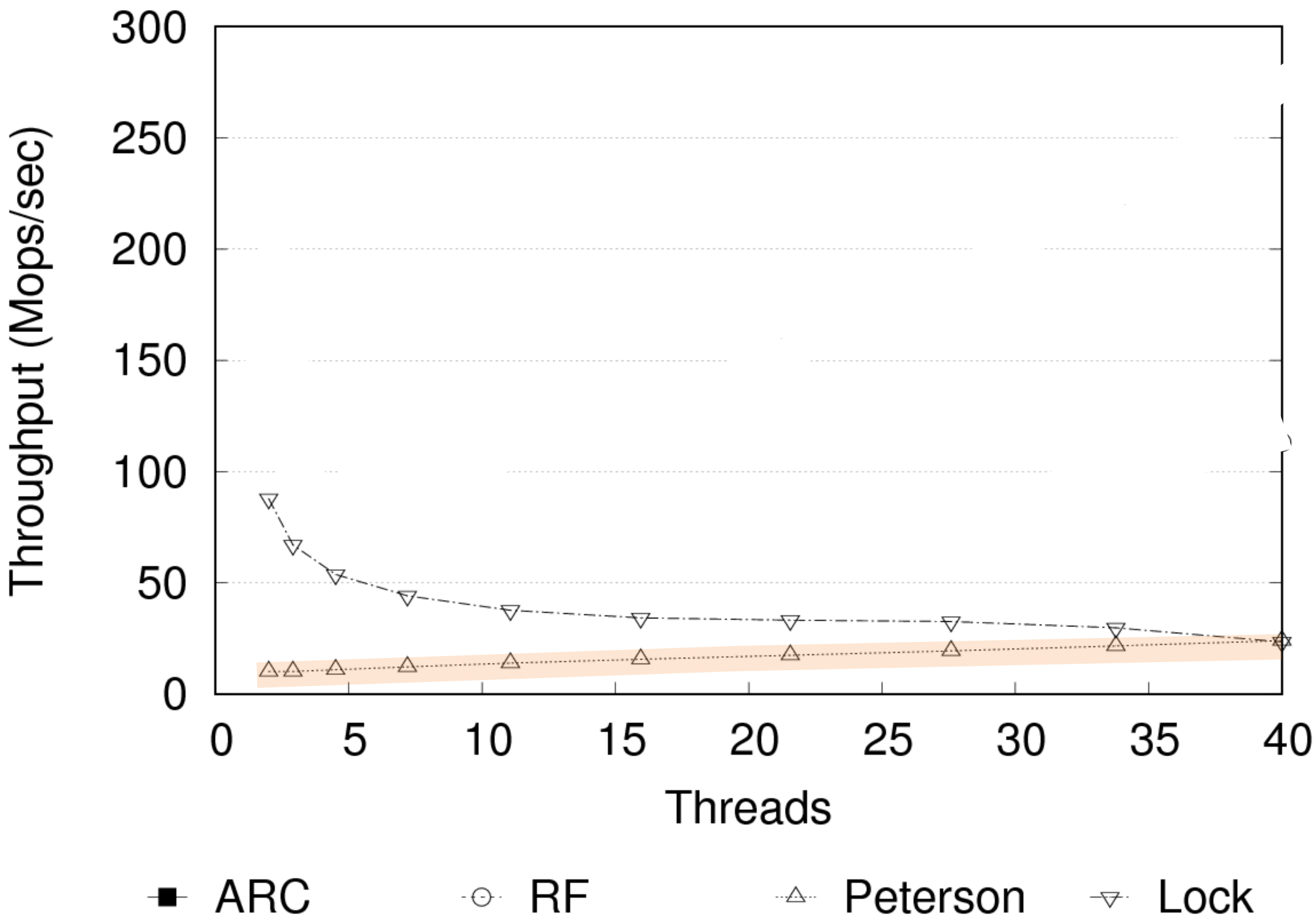
Variable	Type	Description
BUF1	Buffer	Main buffer.
BUF2	Buffer	Backup buffer.
COPYBUF	Array of n buffers	An individual backup buffer for each reader.

Write operation:

```
PW1   WFLAG := true;
PW2   write to BUF1;
PW3   SWITCH := !SWITCH;
PW4   WFLAG := false;
PW5   for (each reader r)
PW6       if (READING[r] != WRITING[r])
PW7           write to COPYBUF[r];
PW8           WRITING[r] := READING[r];
PW9   write to BUF2;
```

Read operation by reader r:

```
PR1   READING[r] := !WRITING[r];
PR2   flag1 := WFLAG;
PR3   sw1 := SWITCH;
PR4   read BUF1;
PR5   flag2 := WFLAG;
PR6   sw2 := SWITCH;
PR7   read BUF2;
PR8   if (READING[r] == WRITING[r])
PR9       return the value in COPYBUF[r];
PR10  else if ((sw1 != sw2) || flag1 || flag2)
PR11      return the value read from BUF2;
PR12  else
PR13      return the value read from BUF1;
```



300
250
200
150
100
50
0
Throughput (Mops/sec)
0
5
10
15
20
25
30
35
40
Threads
ARC
RF
Peterson
Lock

Atomic MRSW Register - Larsson

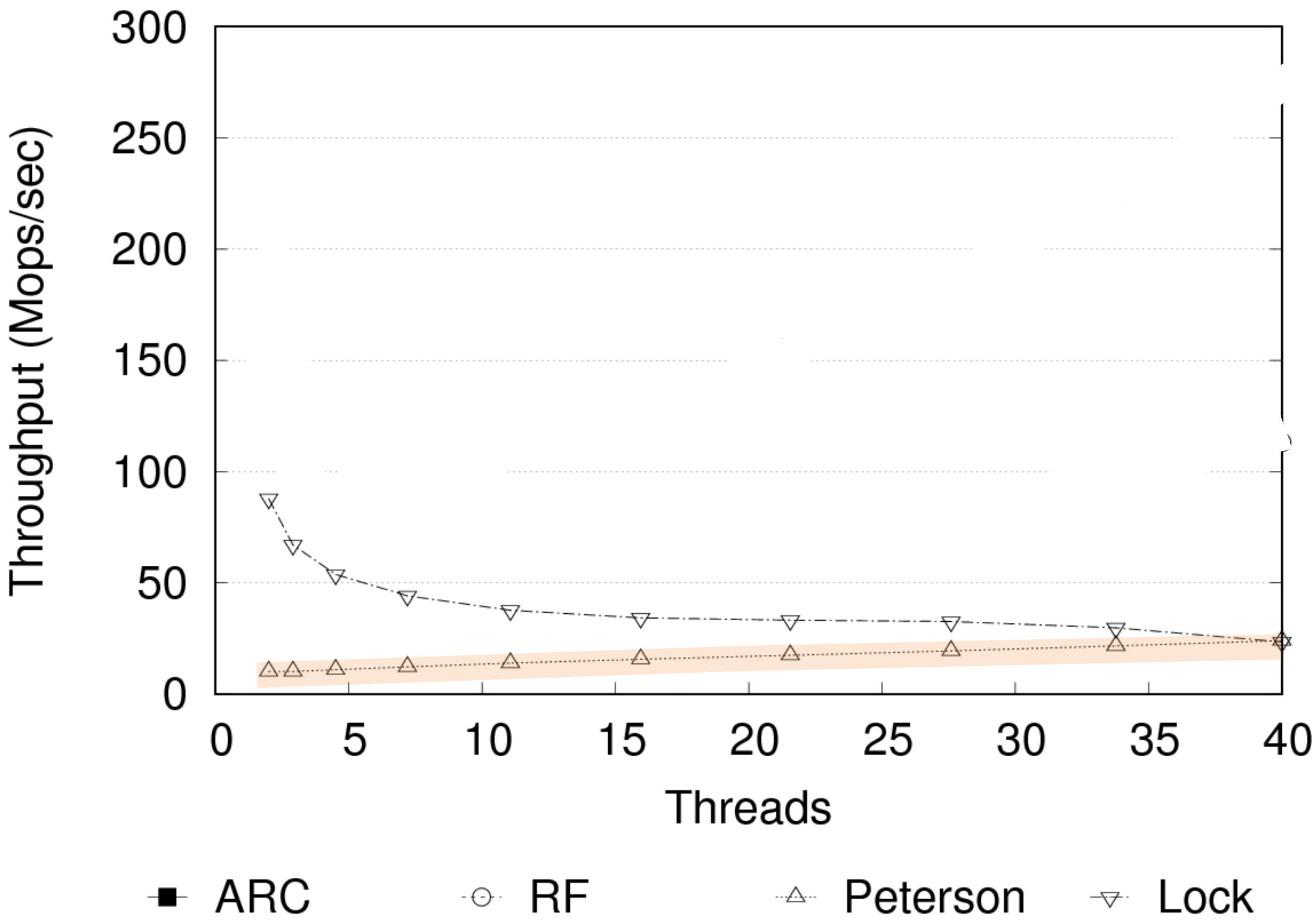
Variable	Type	Description
BUF$[n+2]$	Array of $n+2$ buffers	The buffers for the register value.

Read operation by reader r:

```
R1    readerbit := 1 << (r + PTRFIELDLEN);
R2    rsync := fetch_and_or(&SYNC, readerbit);
R3    rptr := rsync & PTRFIELD;
R4    read BUF[rptr]
```

Write operation:

```
W1    choose newwptr such that newwptr != oldwptr and
          newwptr != trace[r] for all r; /* oldwptr initialized to ⊥*/
W2    write BUF[newwptr];
W3    wsync := swap(&SYNC, 0 | newwptr); /* Clears all reading bits */
W4    oldwptr := newwptr;
      usedwptr := wsync & PTRFIELD;
W5    for each reader r
W6        if (wsync & (1 << (r + PTRFIELDLEN)))
W7            trace[r] := usedwptr;
```


Throughput (Mops/sec)
300
250
200
150
100
50
0
0
5
10
15
20
25
30
35
40
Threads
ARC
RF
Peterson
Lock

Atomic MRSW Register – ARC [Ianni]

Algorithm 1. Register Initialization

procedure INIT(*content*, *size*)
  for all $slot \in [0, N+1]$ **do**
    $register[slot].size \leftarrow 0$
    $register[slot].r\_start \leftarrow 0$
    $register[slot].r\_end \leftarrow 0$
  MEMCOPY $register[0].content, content, size$
  $register[0].size \leftarrow size$
  $current \leftarrow N$

Algorithm 2. The Atomic Register Read Operation

procedure READ
  $index \leftarrow current \gg 32$ ▷ **R1**
  if $last\_index = index$ **then**
    $entry \leftarrow register[last\_index]$
    return $\langle entry.content, entry.size \rangle$ ▷ **R2**
  ATOMICINC($register[last\_index].r\_end$) ▷ **R3**
  $tmp\_curr \leftarrow$ AtomicAddAndFetch ($current, 1$) ▷ **R4**
  $last\_index \leftarrow tmp\_curr \gg 32$ ▷ **R5**
  $entry \leftarrow register[last\_index]$
  return $\langle entry.content, entry.size \rangle$

Algorithm 3. The Atomic Register Write Operation

procedure WRITE(*content*, *size*)
  pick *slot* such that $slot \neq last\_slot \wedge$
  $register[slot].r\_start = register[slot].r\_end$ ▷ **W1**
  MEMCOPY($register[slot].content, content, size$)
  $register[slot].size \leftarrow size$
  $register[slot].r\_start \leftarrow 0$
  $register[slot].r\_end \leftarrow 0$
  $old\_curr \leftarrow$ ATOMICEXCHANGE ($current, slot \ll 32$) ▷ **W2**
  $old\_slot \leftarrow old\_curr \gg 32$
  $register[old\_slot].r\_start \leftarrow old\_curr \;\&\; (2^{32} - 1)$ ▷ **W3**
  $last\_slot \leftarrow slot$