Multi-Version Concurrent Data Structures

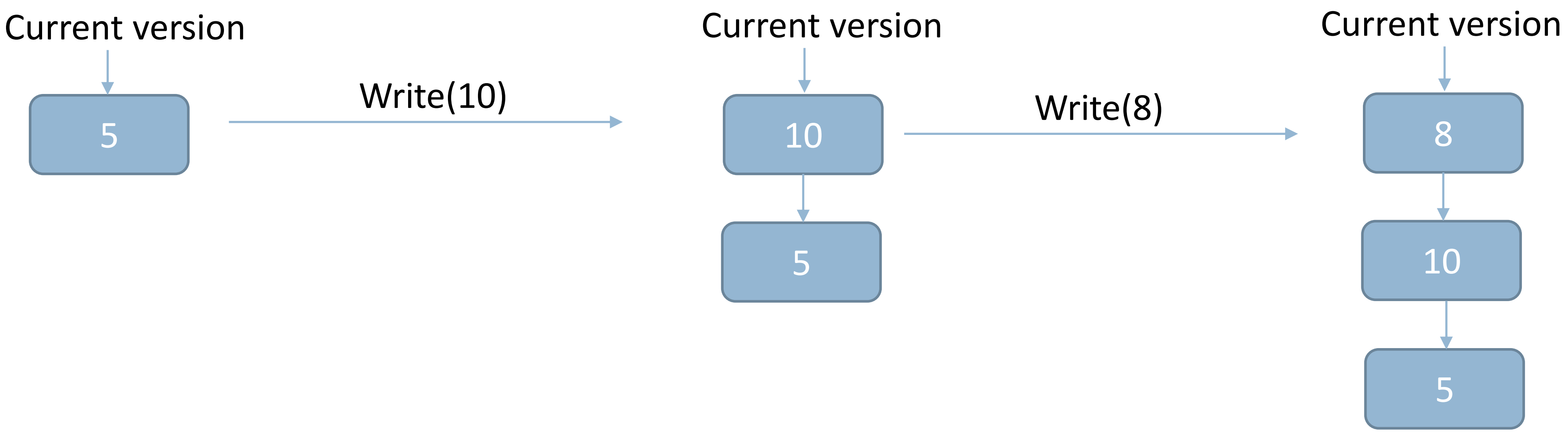
PANAGIOTA FATOUROU

University of Crete, Department of Computer Science

Foundation for Research and Technology - Hellas

School on the Practice and Theory of Distributed Computing, November 2023

Multiversion Objects



- A multiversion object maintains its previous versions,
so threads can have access to the history of the object (i.e., to its previous values).

Multiversioning

- Multiversioning is widely used:
  Database systems

- Software Transactional Memory
  [Fernandes et al. PPoPP’11] [Lu et al. DISC’13]

- **Concurrent data structures**
  [Fatourou et al. SPAA’19] **[Wei et al. PPoPP’21]**
  [Kobus et al. PPoPP’22] [Sheffi et al. OPODIS’22]

Why Multiversioning?

Many applications require querying large portions or multiple parts of the data structure.

Big-data applications use shared in-memory tree-based data indices
- Fast data retrieval
- Useful data analytics

Why multivesion Concurrent Data Structures?

Concurrent Data Structure built out of CAS objects → Take a snapshot of the data structure → Answer multi-point Queries using snapshot

Snapshot: Saves a read-only version of the state of the data structure at a single point in time. [An atomic view of the state of the data structure.]

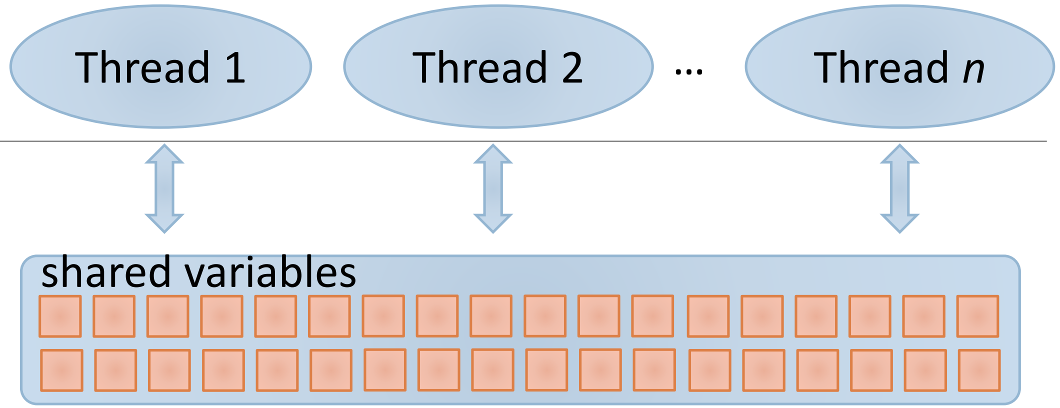
The vCAS technique

- Yuanhao Wei, Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun: *Constant-Time Snapshots with Applications to Concurrent Data Structures*, PPoPP 2021.

Background Knowledge

Model

- The system is asynchronous.
- Threads communicate by accessing shared variables.
- In addition to Read and Write, a thread may execute an atomic CAS instruction on a shared variable.
- Threads may fail by crashing.



ATOMIC boolean Compare&Swap(
    Variable V, Value v_{old}, Value v_{new}) {

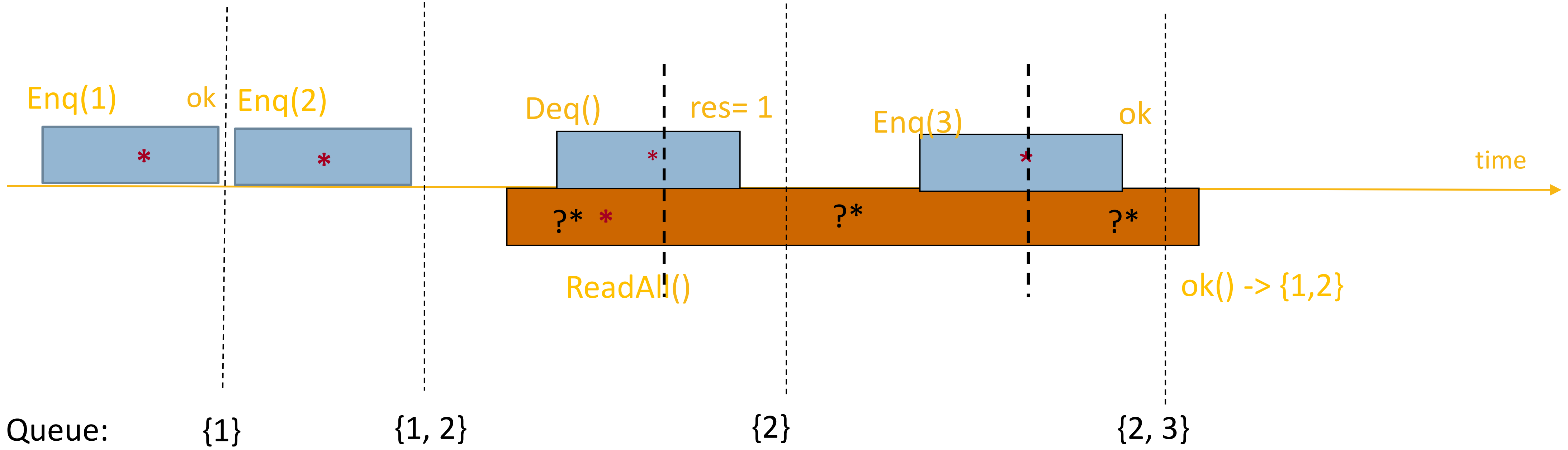
    if (V == v_{old}) { V = v_{new}; return TRUE; }
    return FALSE;
}

Correctness [Herlihy & Wing]

Linearizability

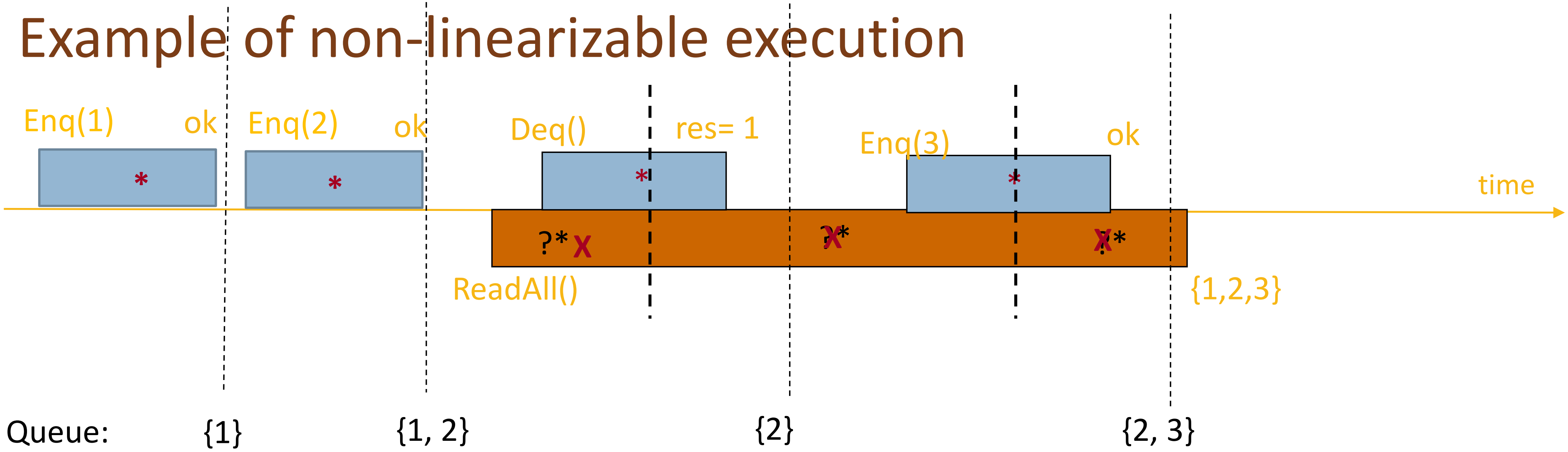
In every execution α, each operation should have the same response as if it has executed serially (or atomically) at some point in its execution interval. This point is called linearization point of the operation.

Linearizability: Queue supporting ReadAll()



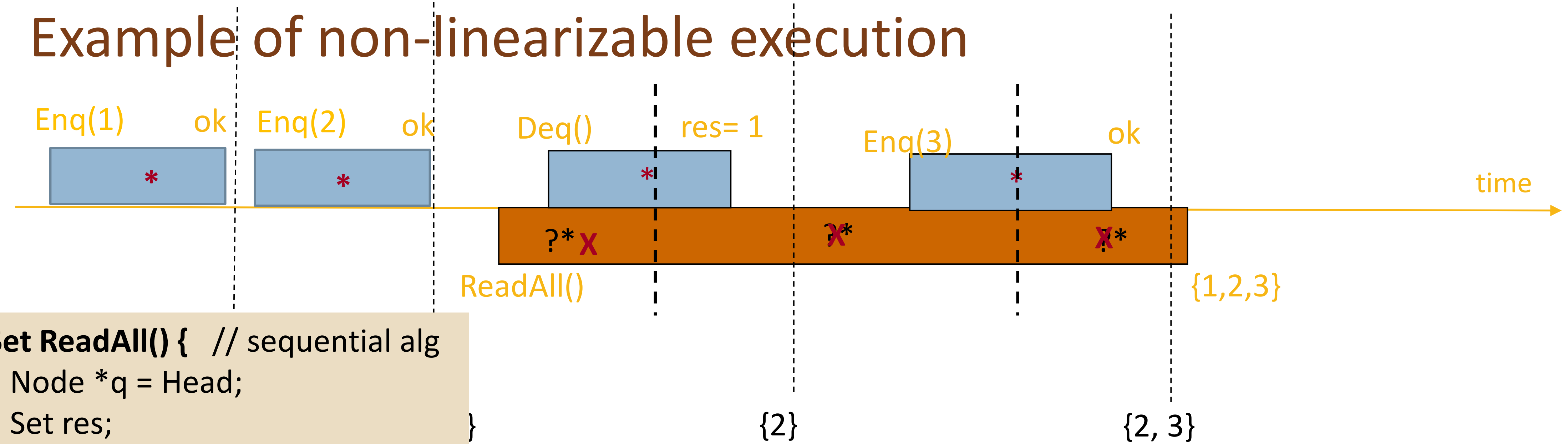
Linearizability: Queue supporting ReadAll()

Example of non-linearizable execution



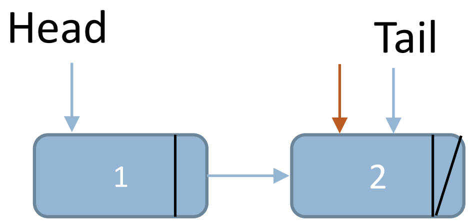
Linearizability: Queue supporting ReadAll()

Example of non-linearizable execution



Set ReadAll() { // sequential alg

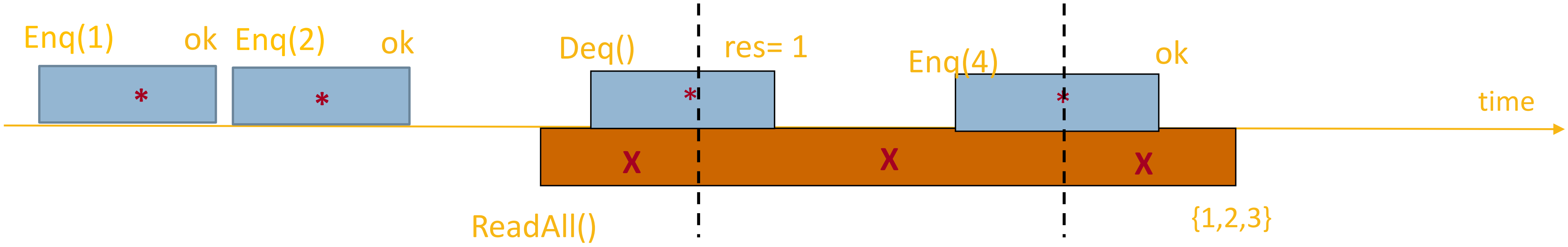
```
  Node *q = Head;
  Set res;
  while (q != NULL) {
      res = res ∪ {q->data};
      q = q-> next;  }
  return res;
}
```



res = {1,2}

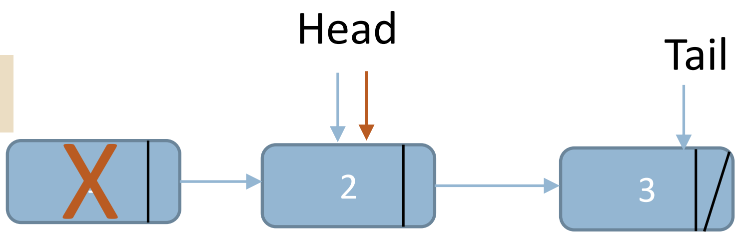
Linearizability: Queue supporting ReadAll()

Example of non-linearizable execution




```
Set ReadAll() {
  Node *q = Head;
  while (q != NULL) {
     res = res ∪ {q->data};
     q = q->next;  }
  return res;
}
```

res = {1,2,3}



Progress

Non-blocking Algorithms

Wait-Freedom

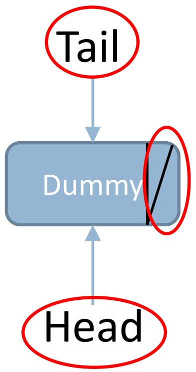
Every thread finishes the execution of its operation within a finite number of steps.

Lock-Freedom

Some thread finishes the execution of its operation within a finite number of steps.

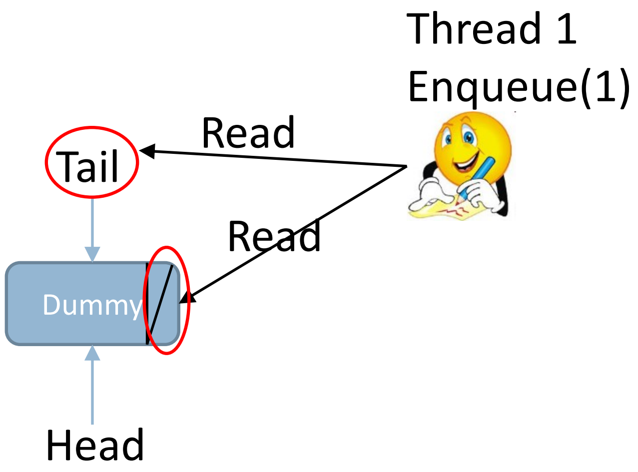
An Example of a Concurrent Queue Implementation

Michael & Scott Queue as an Example




```
struct node {
        T value ;                    // unmutable
        CAS Object next : struct node *;
}
CAS objects Head, Tail: struct node *;  // initially, both point
                                          to a dummy node
```

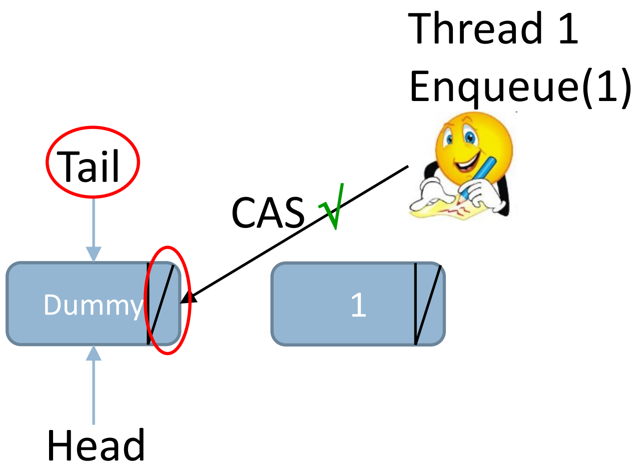
Michael & Scott Queue as an Example



struct node {
    T value ;   *// unmutable*
    CAS Object next : struct node *;
}

CAS objects Head, Tail: struct node *; *// initially, both point to a dummy node*

Michael & Scott Queue as an Example

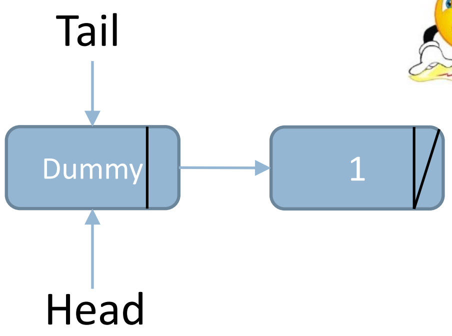


struct node {
    T value ;   *// unmutable*
    CAS Object next : struct node *;
}

CAS objects Head, Tail: struct node *; *// initially, both point to a dummy node*

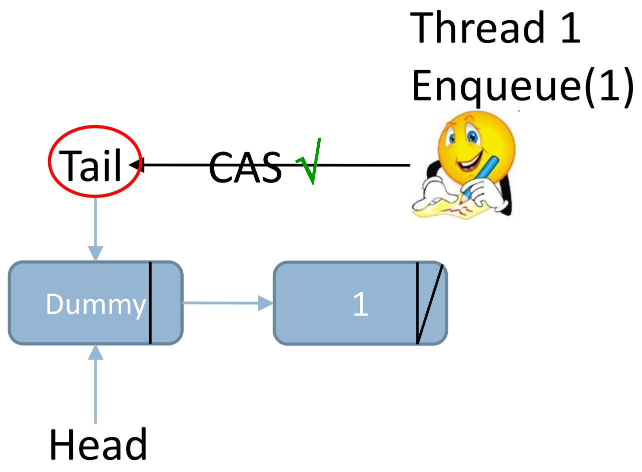
Michael & Scott Queue as an Example

Thread 1
Enqueue(1)




```
struct node {
        T value ;                    // unmutable
        CAS Object next : struct node *;
 }
CAS objects Head, Tail: struct node *;  // initially, both point
                                          to a dummy node
```

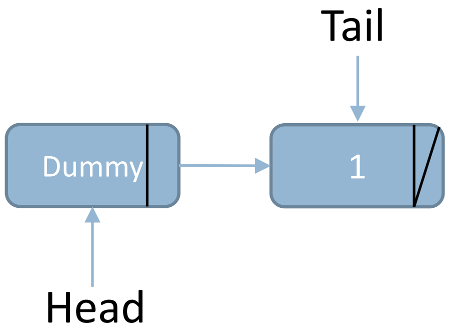
Michael & Scott Queue as an Example




```
struct node {
    T value ;                 // unmutable
    CAS Object next : struct node *;
}
```

CAS objects Head, Tail: struct node *; *// initially, both point to a dummy node*

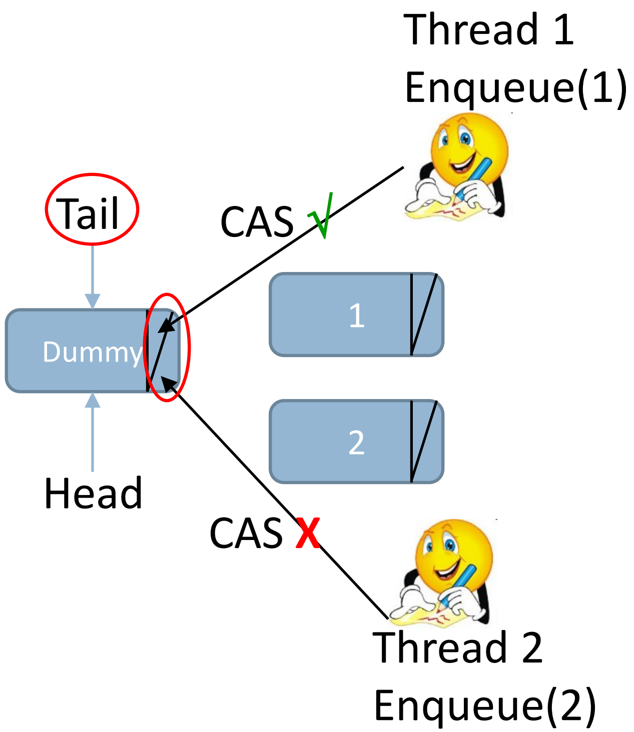
Michael & Scott Queue as an Example



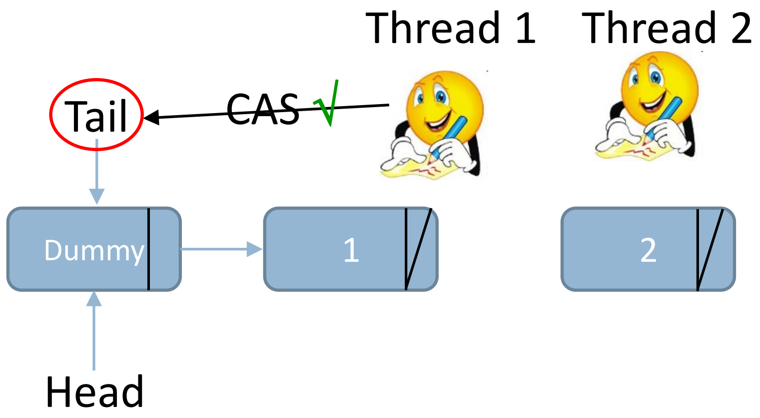

```
struct node {
        T value ;                    // unmutable
        CAS Object next : struct node *;
}
```

CAS objects Head, Tail: struct node *; *// initially, both point to a dummy node*

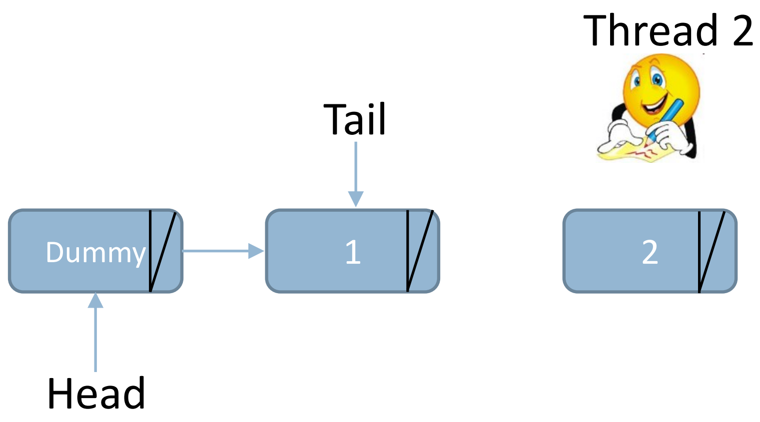
Michael & Scott Queue as an Example



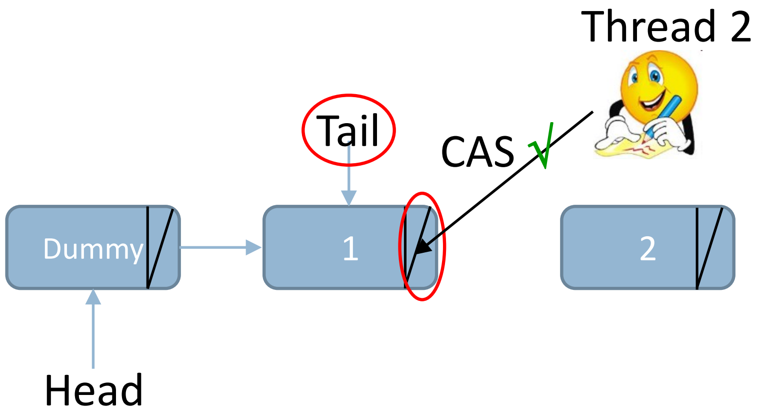
Michael & Scott Queue as an Example



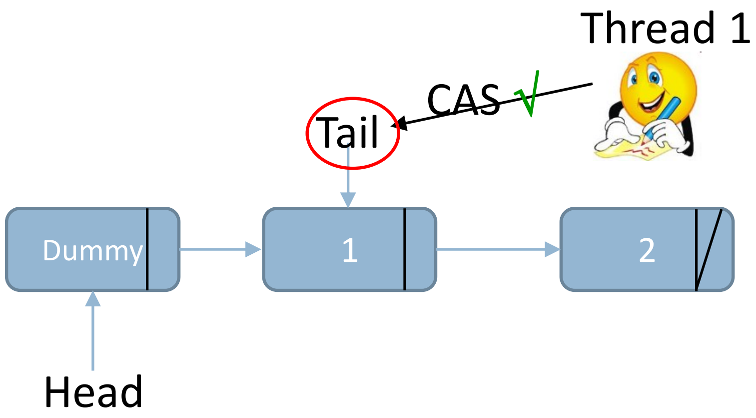
Michael & Scott Queue as an Example



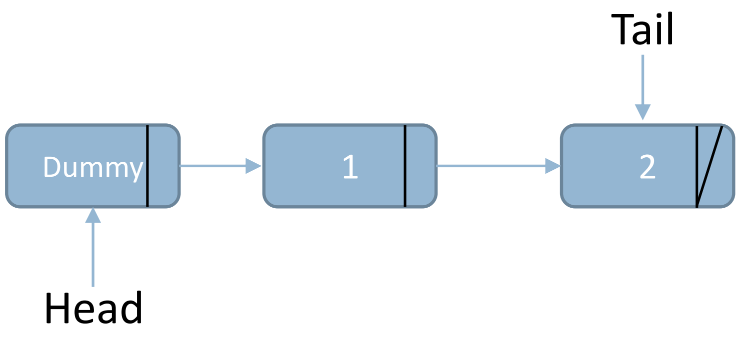
Michael & Scott Queue as an Example



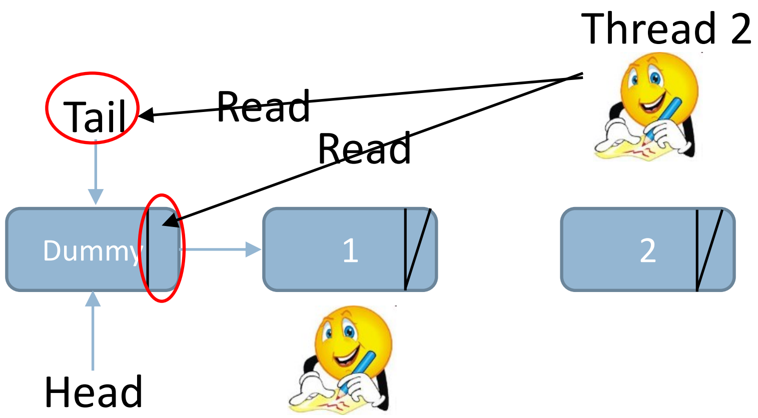
Michael & Scott Queue as an Example



Michael & Scott Queue as an Example

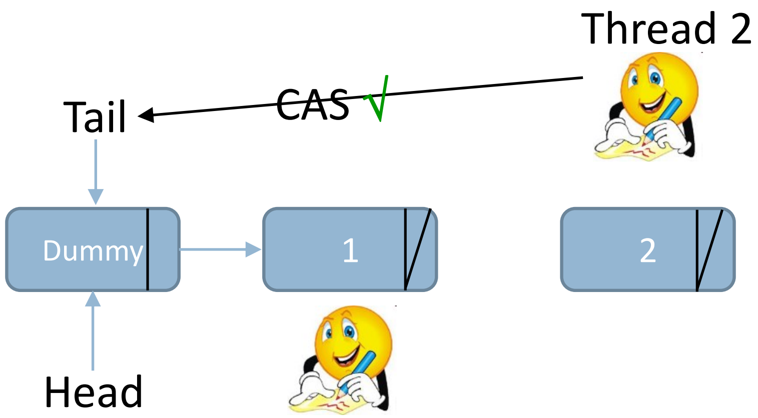


Michael & Scott Queue as an Example



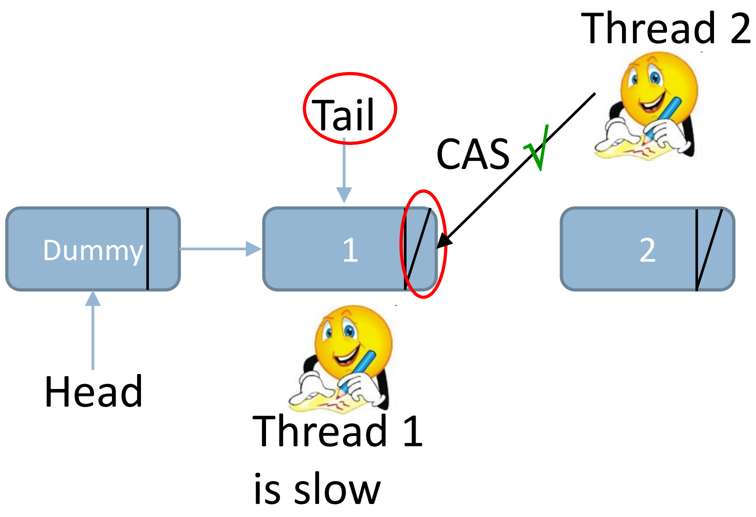
Thread 1 is slow

Michael & Scott Queue as an Example

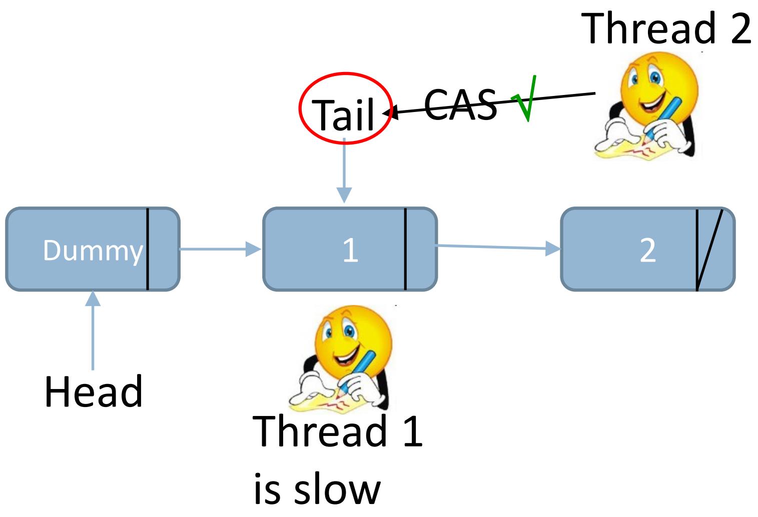


Thread 1 is slow

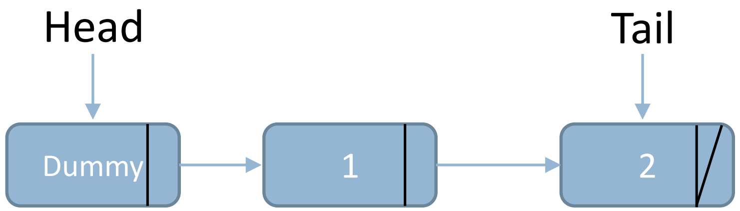
Michael & Scott Queue as an Example



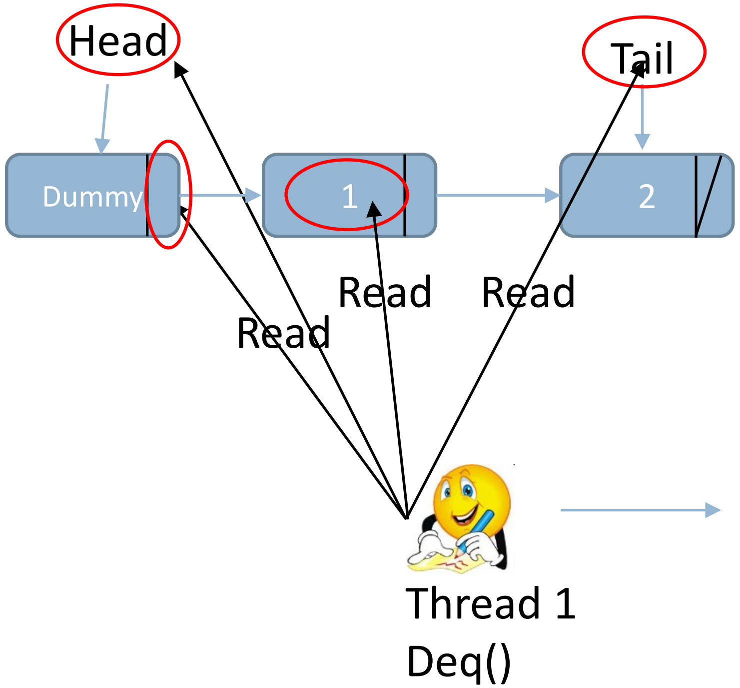
Michael & Scott Queue as an Example



Michael & Scott Queue as an Example

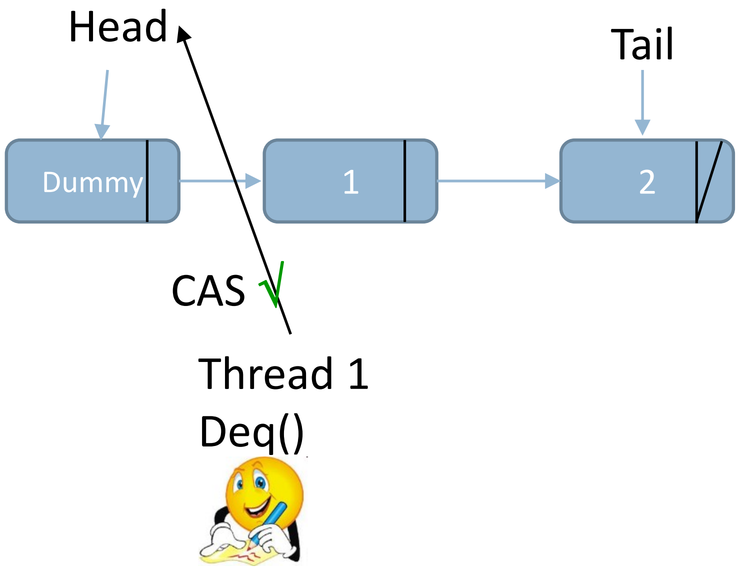


Michael & Scott Queue as an Example

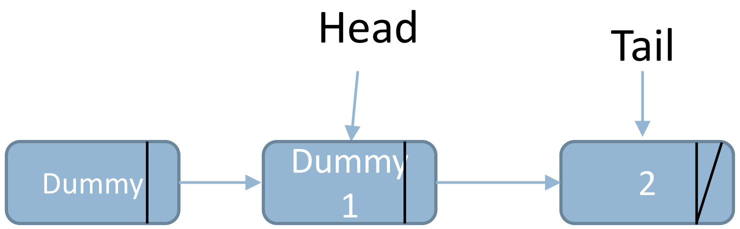


stores 1 as its return value

Michael & Scott Queue as an Example



Michael & Scott Queue as an Example



vCAS Technique

Simple, General, Efficient!

Lock-free Concurrent Data Structure → Take a snapshot of the Data Structure → Answer multi-point Queries using snapshot

Lock-free Queue [Michael & Scott'96]
Enqueue, Dequeue

→

Lock-free Snapshottable Queue

Enqueue
Dequeue
} Preserves parallelism and time bounds

Snapshot — O(1) time, a single CAS

Range Query
i-th element
All elements
} Wait-free, Linearizable

Works with many lock-free data structures, including:

- BST [Ellen,Fatourou, Ruppert, Breugel'10]
- Linked List [Harris'01]
- Chromatic Tree [BrownEllenRuppert'14]
- …

Overview of the VCAS Approach

CAS Object

Supports:

- Read
- CAS

Versioned CAS (vCAS) Object

Supports:

- vRead
- vCAS
- **readVersion**

Camera Object

Supports:

TakeSnapshot

Makes it possible for a thread to later read only the memory locations it needs from shared memory, knowning that all such reads will be atomic.

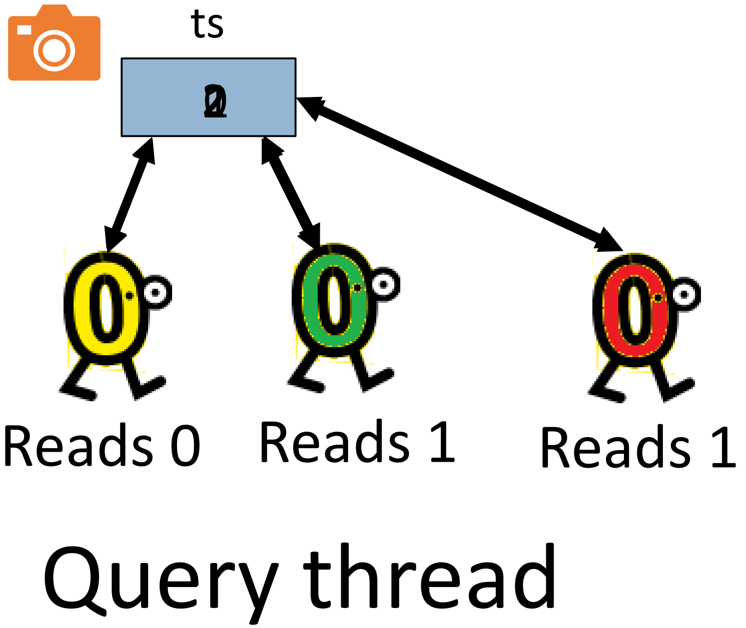
Time Complexity:

- vRead(X)
- vCAS(X, old, new)
- takeSnapshot()
- readVersion(X, S)

vRead(X), vCAS(X, old, new), takeSnapshot(): O(1) time, small constant

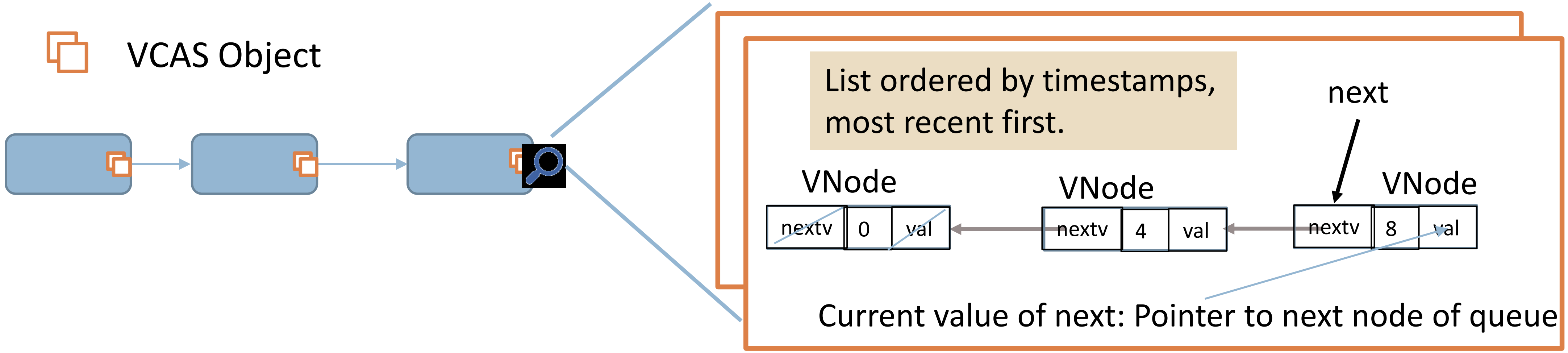
readVersion(X, S): wait-free

Supporting Multi-Point Queries



- Each query calls TakeSnapshot to get a timestamp.
- More than one queries may have the same timestamp.
- Each query attempts to atomically increment ts using CAS.
- Each version of a vCAS object has a timestamp, which has been read from ts.

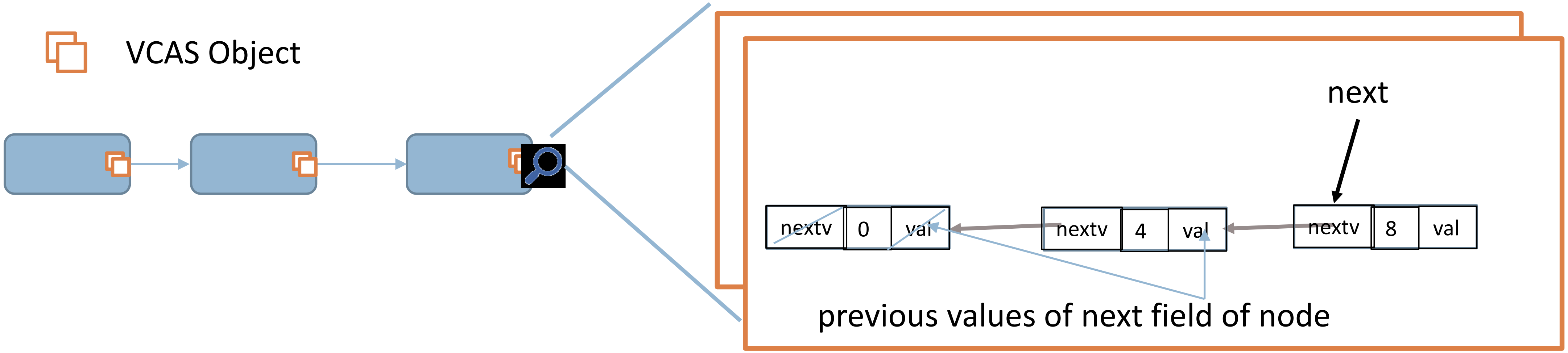
Versioned CAS Implementation



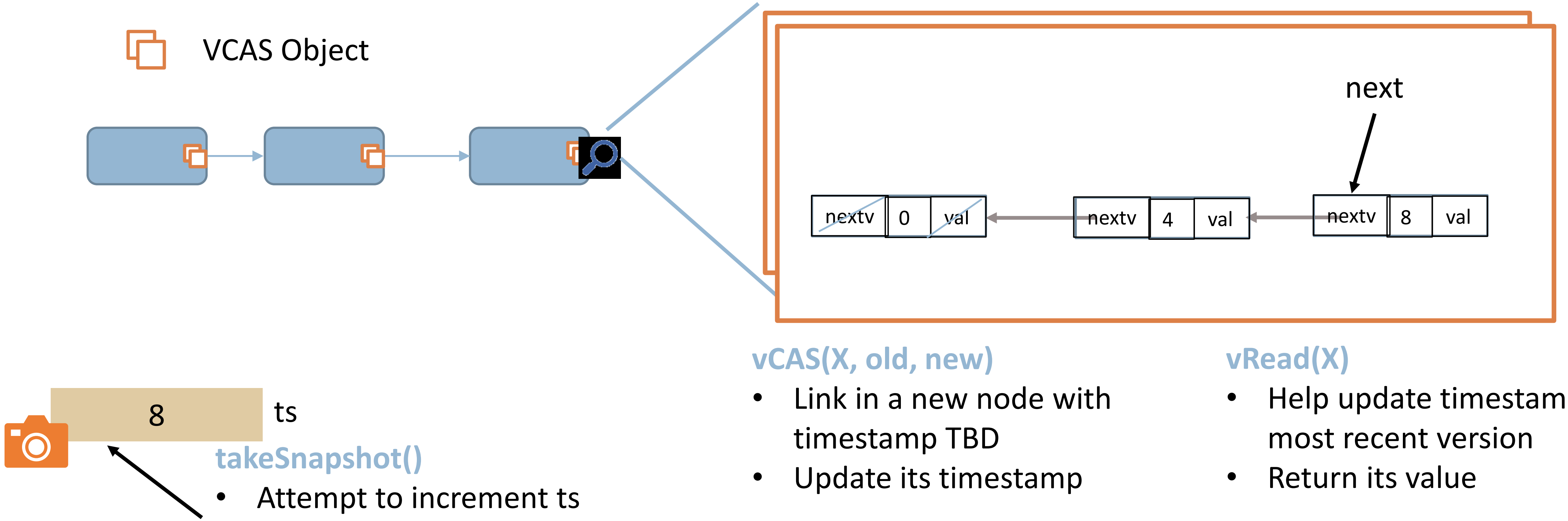
- VCAS objects are represented internally using version lists. The fields of a Vnode (i.e., a node of a version list) are:
  - val
  - ts
  - vnext

Versioned CAS Implementation

VCAS Object



Versioned CAS Implementation



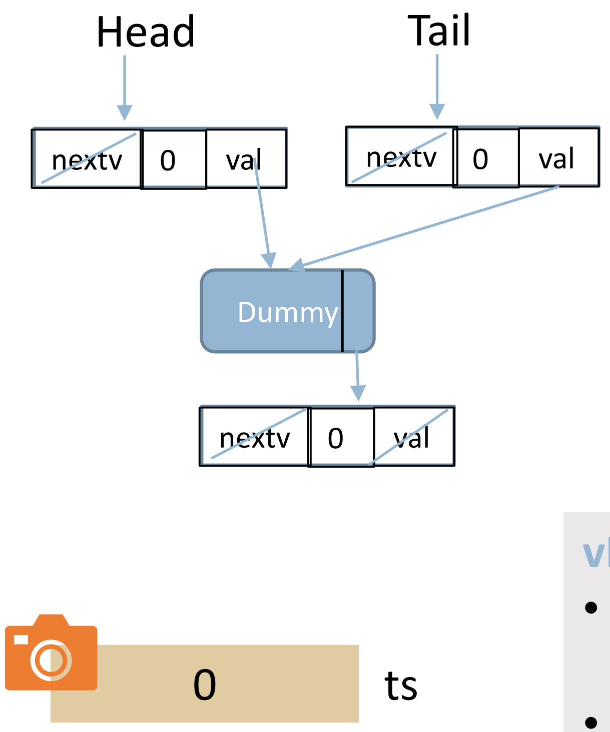
takeSnapshot()
- Attempt to increment ts using CAS
- Return its previous value

vCAS(X, old, new)
- Link in a new node with timestamp TBD
- Update its timestamp

vRead(X)
- Help update timestamp of most recent version
- Return its value

readVersion(X, *t*)
- Help update timestamp
- Find newest version with timestamp ≤ ***t***

Overview of the VCAS Approach: Michael & Scott Queue as an Example



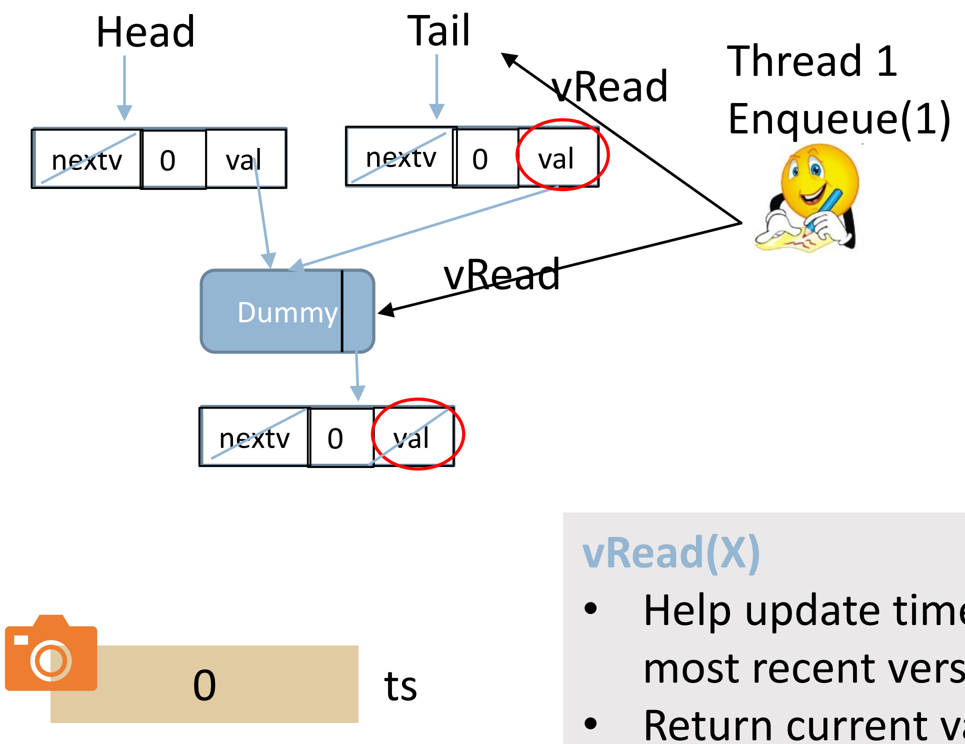
vRead(X)

- Help update timestamp of most recent version of X
- Return current value of X

```
struct node {
      T value ;
      CAS Object next : struct node *;
 }
CAS objects Head, Tail: struct node *;
```

```
struct node {
      T value ;
      vCAS Object next : struct node *;
 }
vCAS objects Head, Tail: struct node *;
```

Overview of the VCAS Approach: Michael & Scott Queue as an Example



vRead(X)

- Help update timestamp of most recent version of X
- Return current value of X

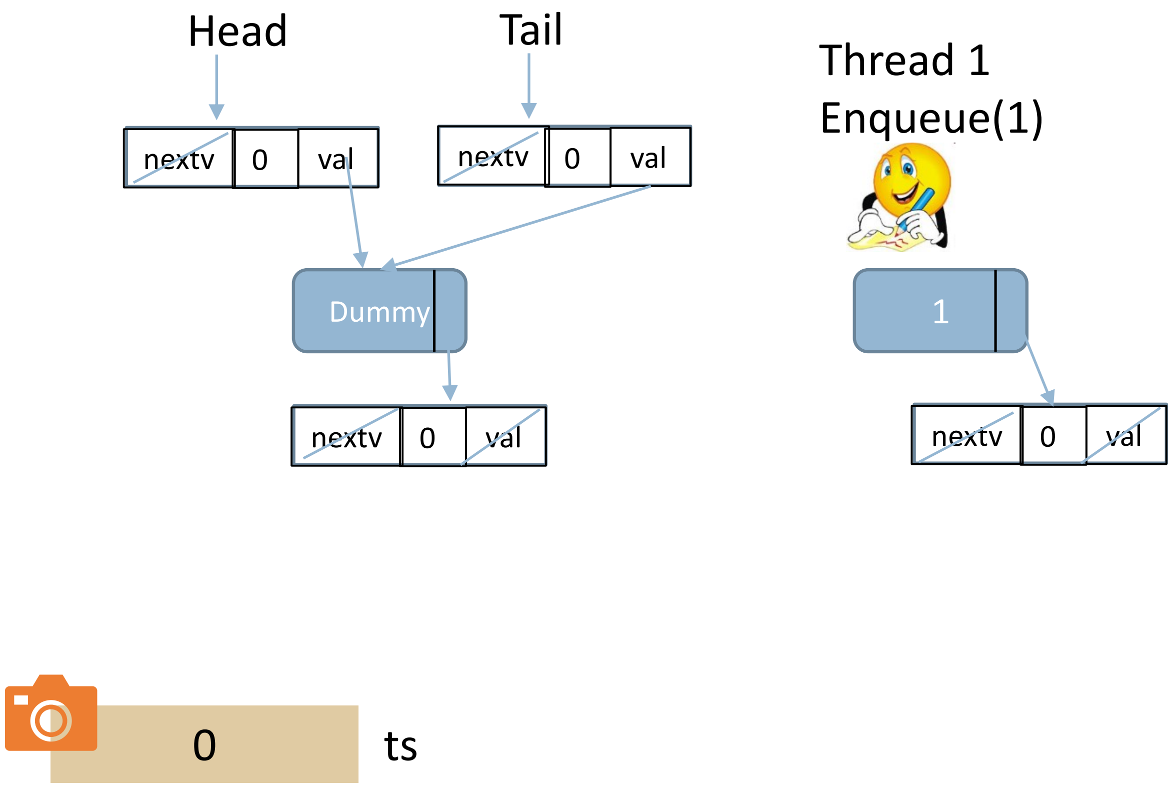
struct node {
    T value ;
    CAS Object next : struct node *;
}

CAS objects Head, Tail: struct node *;

struct node {
    T value ;
    vCAS Object next : struct node *;
}

vCAS objects Head, Tail: struct node *;

Overview of the VCAS Approach: Michael & Scott Queue as an Example



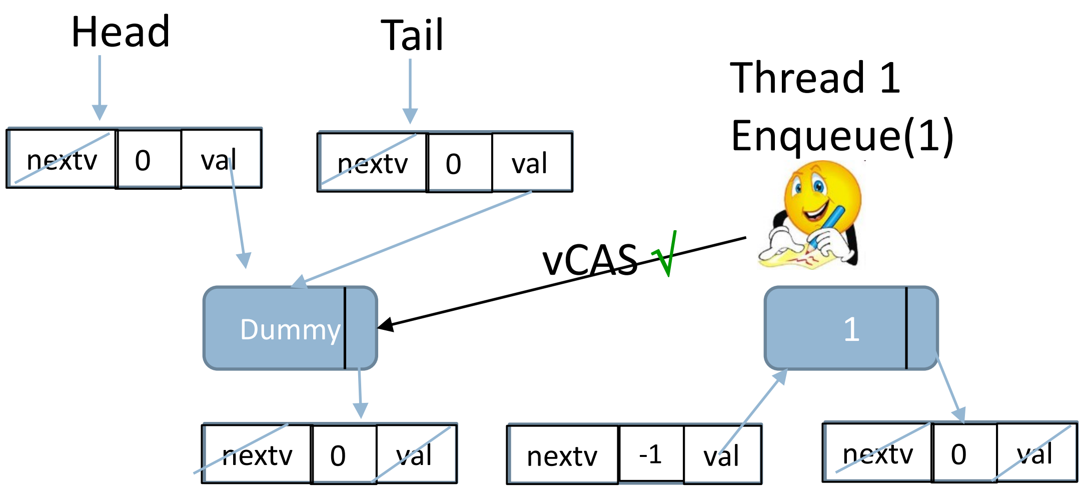
struct node {
    T value ;
    CAS Object next : struct node *;
}

CAS objects Head, Tail: struct node *;

struct node {
    T value ;
    vCAS Object next : struct node *;
}

vCAS objects Head, Tail: struct node *;

Overview of the VCAS Approach: Michael & Scott Queue as an Example



struct node {
T value ;
CAS Object next : struct node *;
}

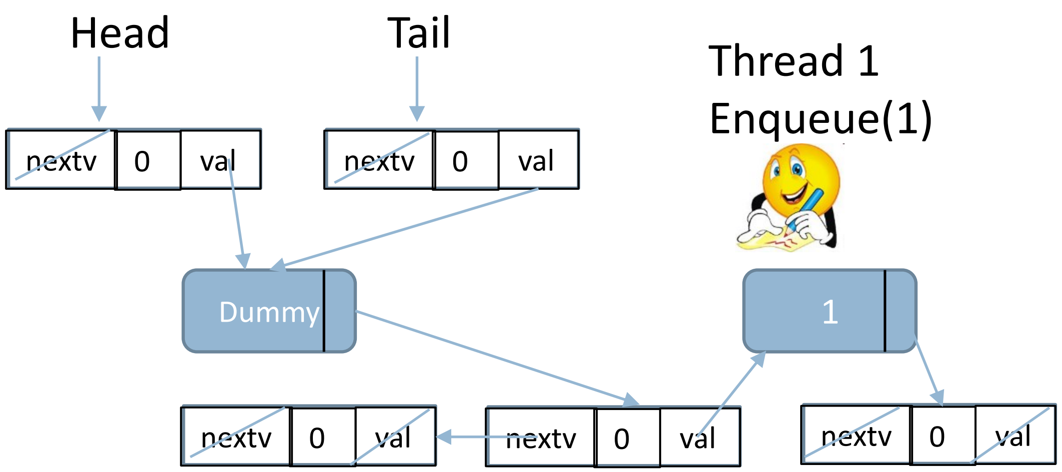
CAS objects Head, Tail: struct node *;

vCAS(X, old, new)
- Malloc() a new vNode with timestamp TBD (-1)
- Link it in the version list of the vCAS object
- Update its timestamp

struct node {
T value ;
vCAS Object next : struct node *;
}

vCAS objects Head, Tail: struct node *;

Overview of the VCAS Approach: Michael & Scott Queue as an Example



vCAS(X, old, new)

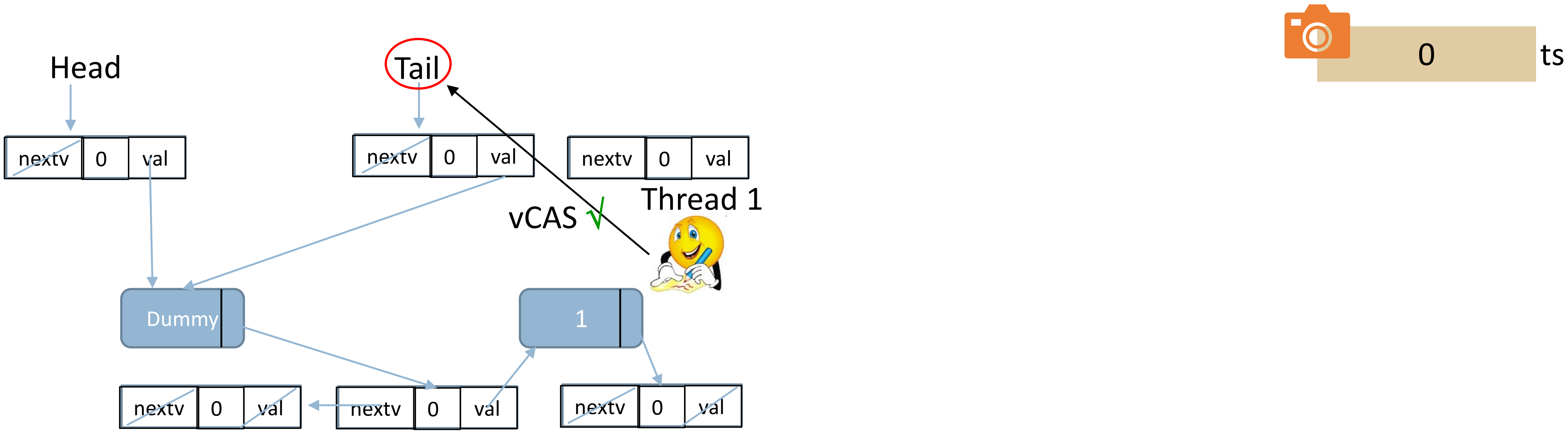
- Malloc() and link in a new vNode with timestamp TBD (-1)
- Make it the first node in the vlist of vCAS object
- Update its timestamp

0 ts

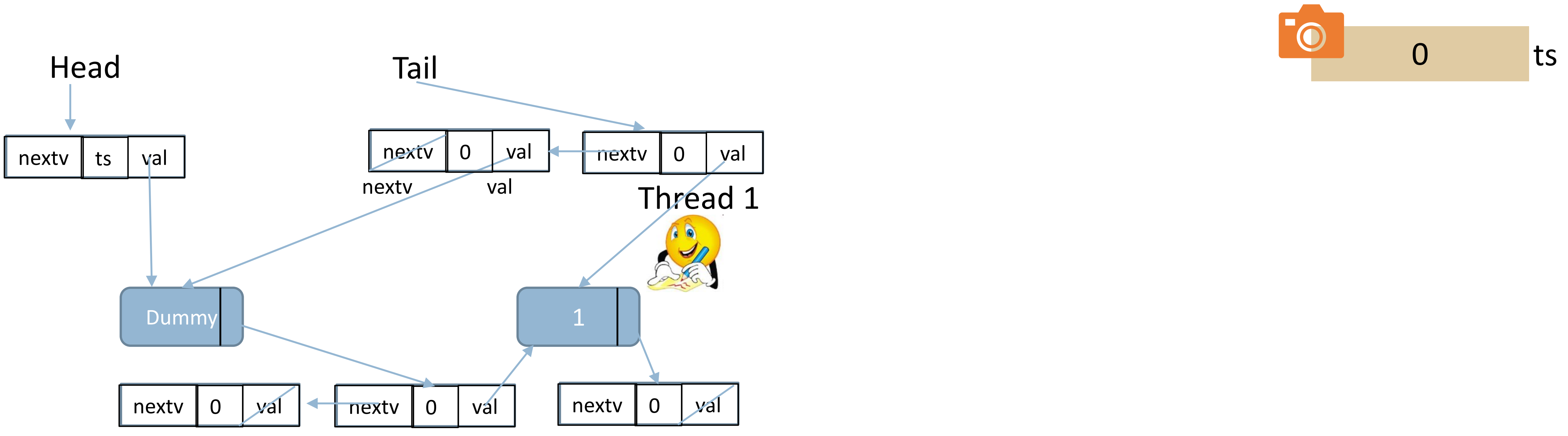
```
struct node {
    T value ;
    CAS Object next : struct node *;
}
CAS objects Head, Tail: struct node *;
```

```
struct node {
    T value ;
    vCAS Object next : struct node *;
}
vCAS objects Head, Tail: struct node *;
```

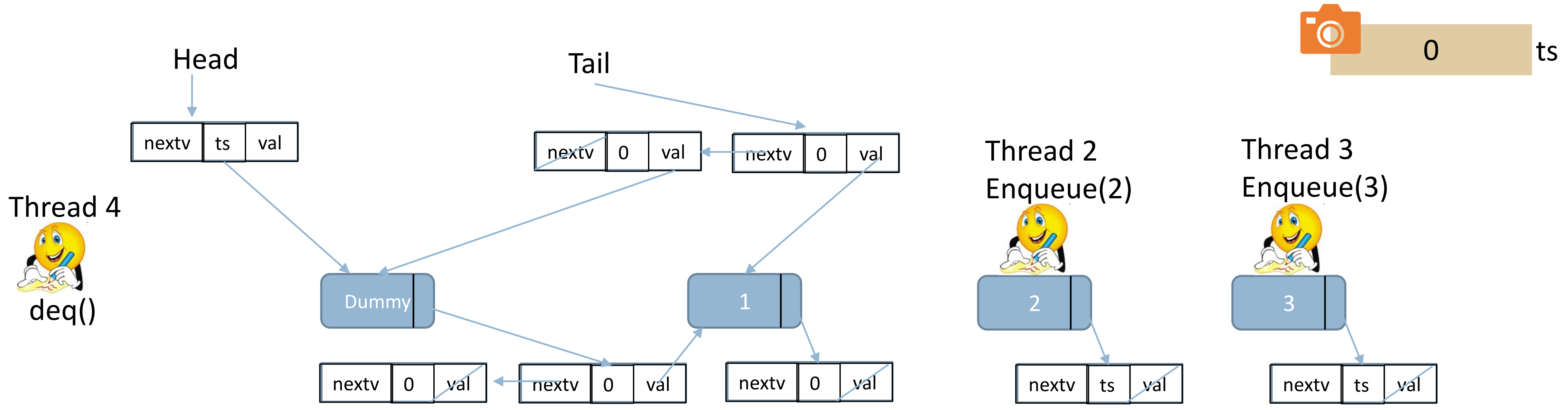
Overview of the VCAS Approach: Michael & Scott Queue as an Example



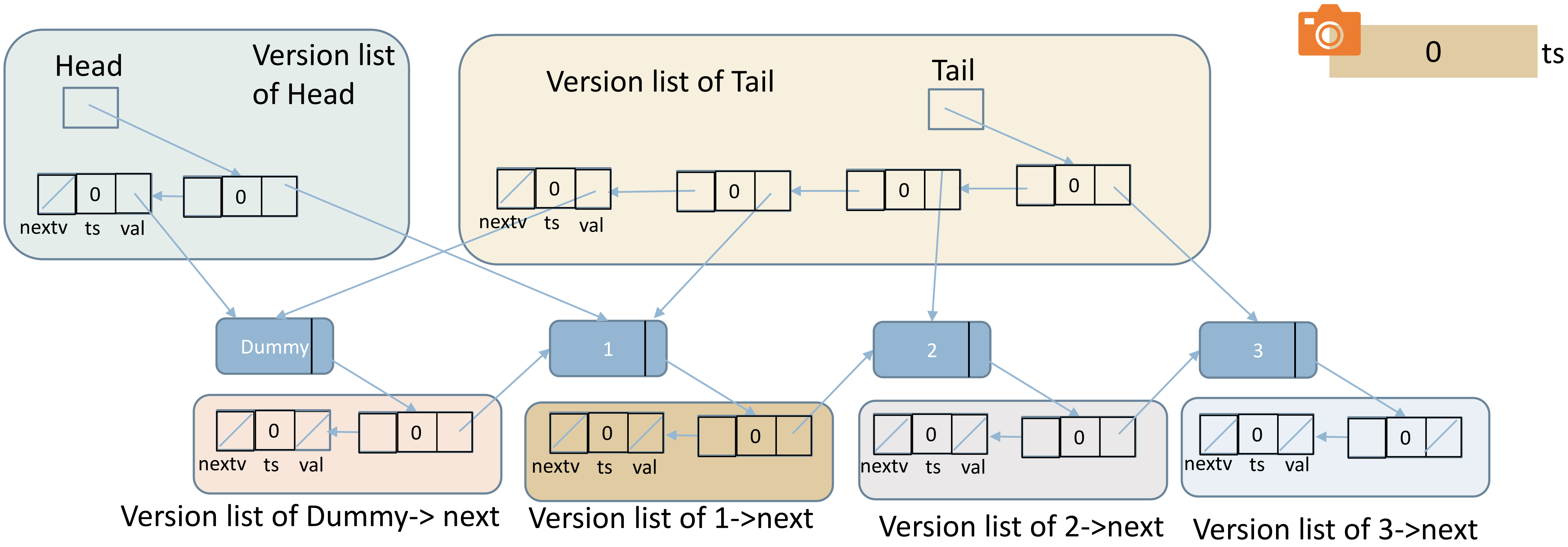
Overview of the VCAS Approach: Michael & Scott Queue as an Example



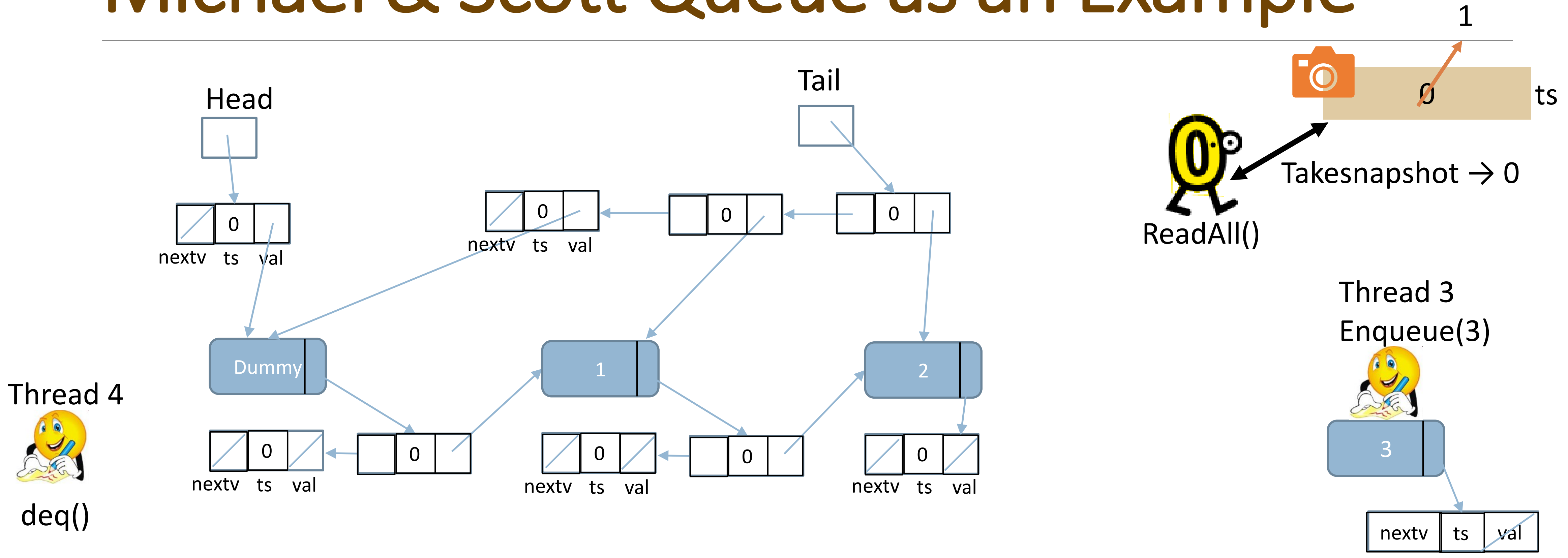
Overview of the VCAS Approach: Michael & Scott Queue as an Example



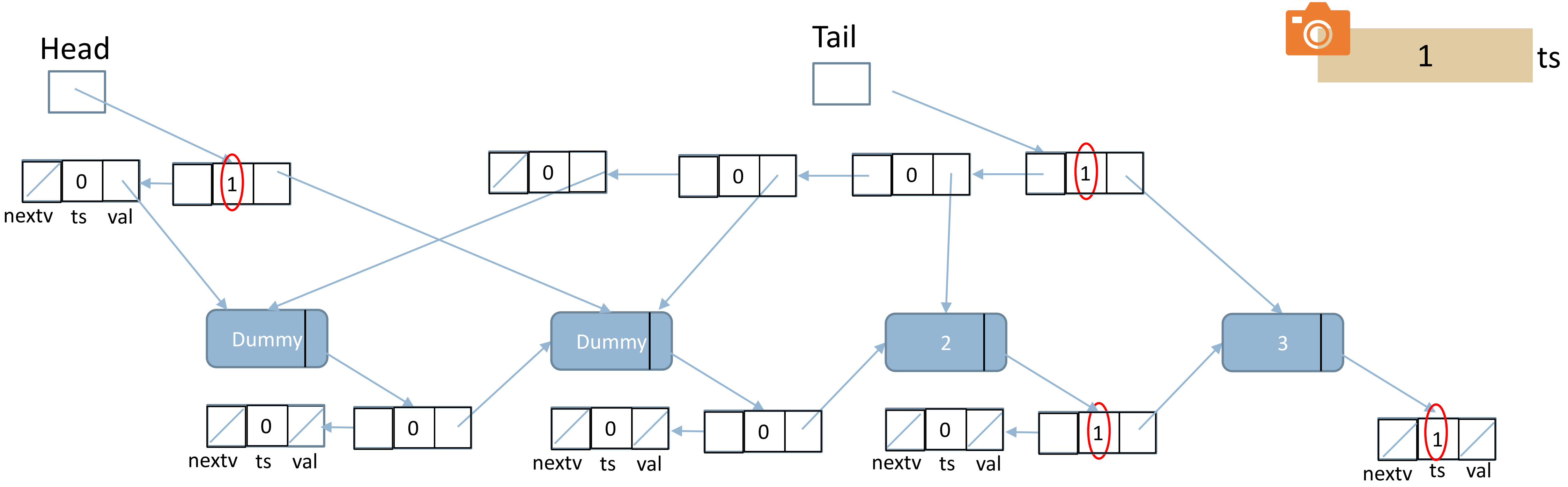
Overview of the VCAS Approach: Michael & Scott Queue as an Example



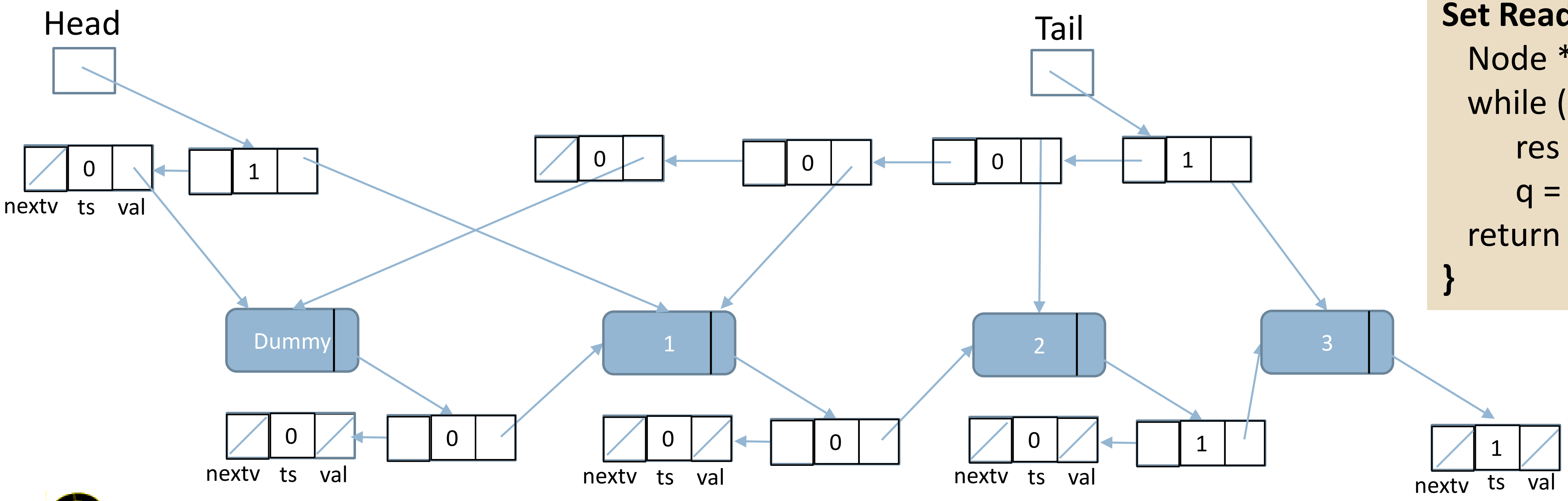
Multi-Point Queries: Michael & Scott Queue as an Example



Multi-Point Queries:
Michael & Scott Queue as an Example

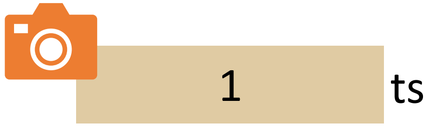


Multi-Point Queries: Michael & Scott Queue as an Example



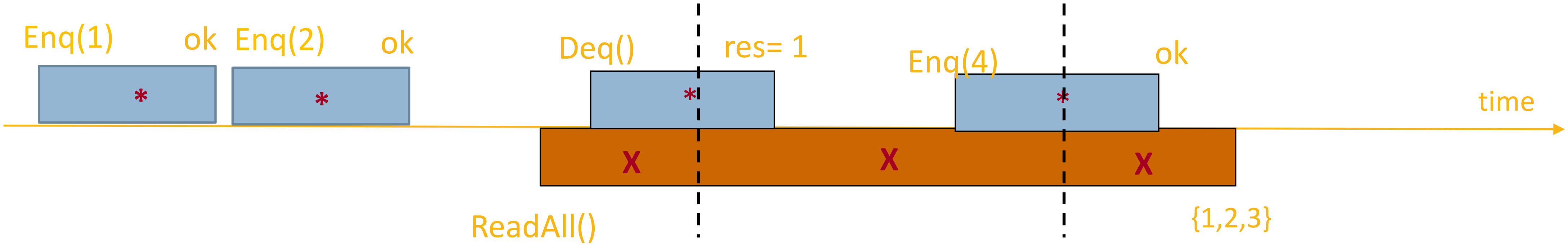

```
Set ReadAll() {
  Node *q = Head;
  while (q != NULL) {
      res = res ∪ {q->data};
      q = q-> next;  }
  return res;
}
```

- Executes Sequential Code, but…



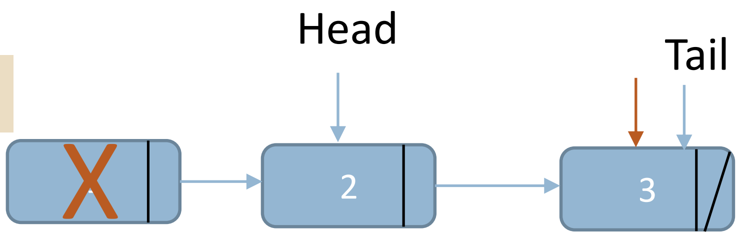
Linearizability: Queue supporting ReadAll()

Example of non-linearizable execution



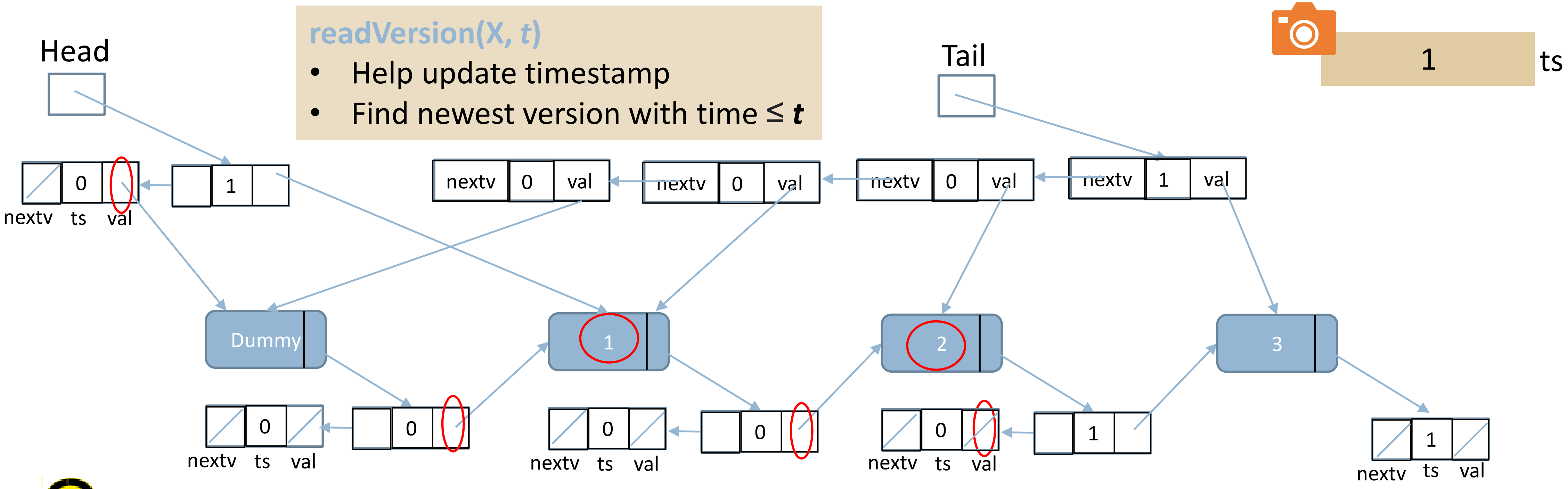

```
Set R adAll() {
   Nod  *q = H  ad;
   while (  !=  NULL) {
        res =   s ∪ {q->data};
        q =  ->   xt;  }
   retu   res;
}
```

res = {1,2,3}



Multi-Point Queries:
Michael & Scott Queue as an Example

readVersion(X, *t*)
- Help update timestamp
- Find newest version with time ≤ ***t***



ReadAll()

- Executes Sequential Code for ReadAll() but…
- Uses **ReadVersion(0)** to read the values of vCAS objects as it goes

It returns {1,2}

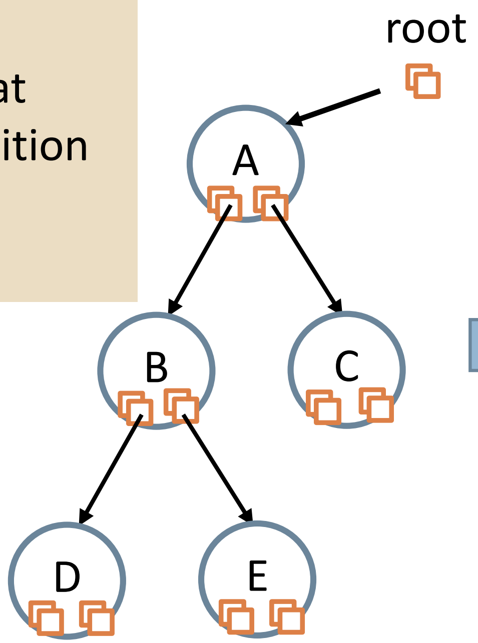
Overview of the VCAS Approach: Michael & Scott Queue as an Example

```
void enq(T value ) {
    NODE *next , *last ;
    NODE *p = newcell(NODE) ;
                    // p->value = value ; p->next = NULL;
    while (TRUE) {
            last = Tail ;
            next = last->next ;
            if (last != Tail) continue;
            if (next != NULL) {
                    CAS( Tail , last, next);
                    continue;
            }
            if (CAS(last->next , NULL , p)) break ;
    }
    CAS( Tail , last, p );
}
```

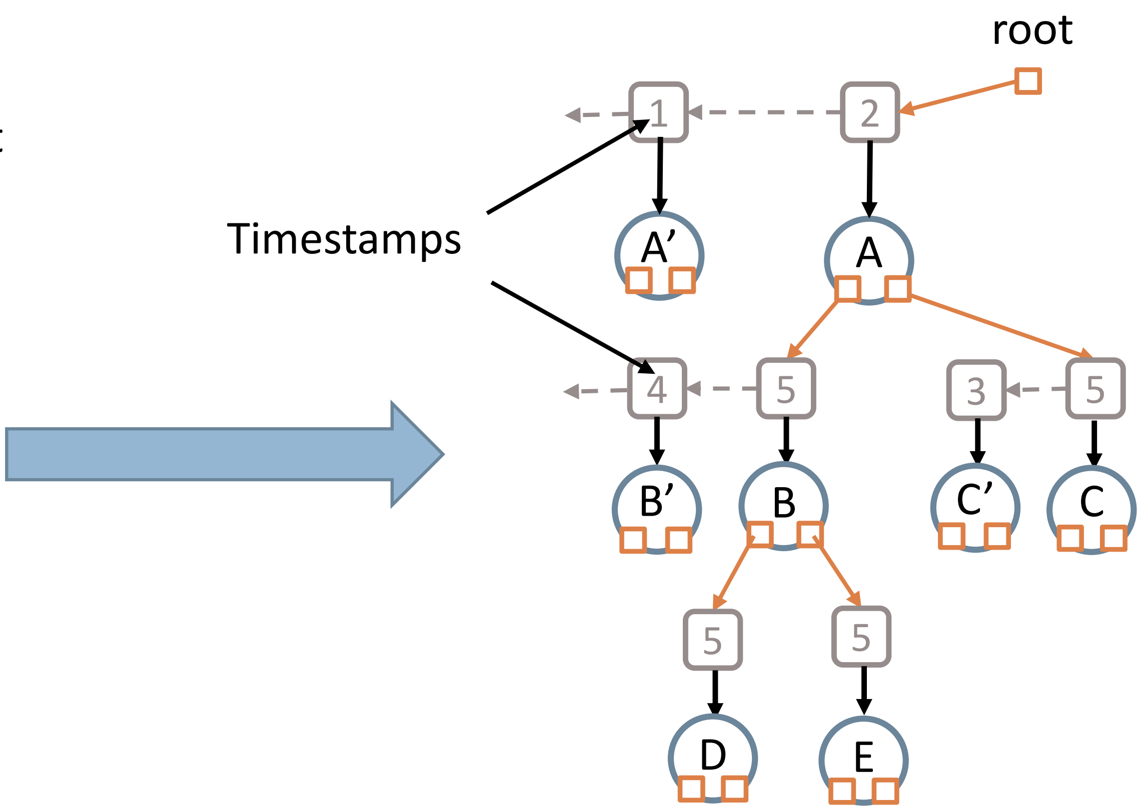
```
void enq(T value ) {
    NODE *next , *last ;
    NODE *p = new(NODE, value, NULL) ;

    while (TRUE) {
            last = vRead(Tail) ;
            next = vRead(last->next);
            if (last != vRead(Tail)) continue;
                if (next != NULL) {
                    vCAS(  Tail   , last, next);
                    continue;
            }
            if (vCAS(  last->next   , NULL , p)) break ;
    }
    vCAS(  Tail   , last, p );
}
```

Versioned CAS on BSTs

Examples of queries
- Range queries
- Tree height
- Smallest key that matches a condition
- K-successors
- Multi-lookup



Snapshottable BST



Expanding out VCAS objects

Comparison with Existing Techniques

The VCAS Approach, Wei et al.

LFCA [Winbland et al., SPAA'18]
KiWi [Basil et al., PPoPP'17]
PNB-BST [Fatourou et al., SPPA'19]
SnapTree [Bronson et al., PPoPP'10]

Epoch RQs [Arbel, Raviv et al., PPoPP'18]
SnapCollector [Petrank et al., PPoPP'13]

STM [Fernandez et al., PPoPP'11]

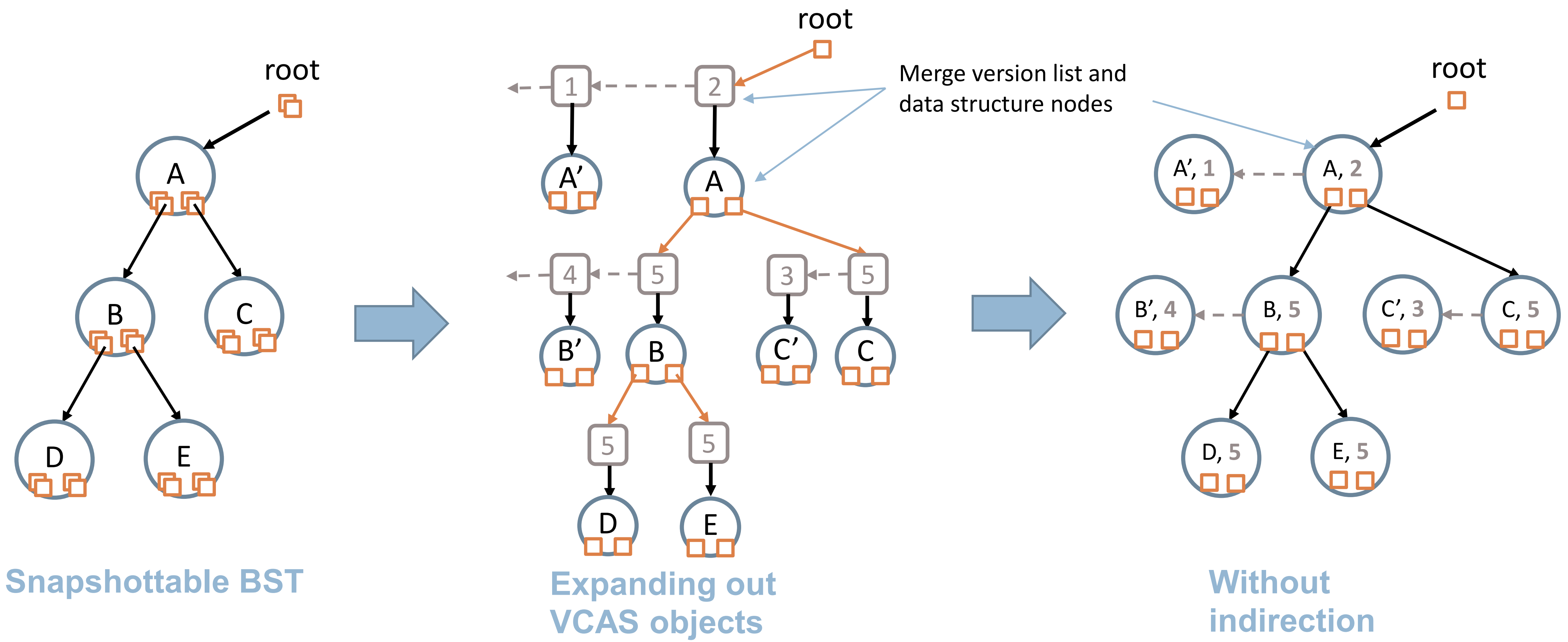
Efficiency

Generality

Practical Optimizations

- **Avoiding Indirection**
- Using exponential backoff to reduce contention when accessing the global timestamp
- Removing redundant versions from the version list
- Garbage collecting old versions

Avoiding Indirection



Experimental Evaluation

- Adding support for multi-point queries on top of existing concurrent lock-free data structures was very easy and required adding fewer than 150 lines of code (in C++).
- The vCAS approach adds very little overhead to the original data structure
- The vCAS approach (which is general-purpose) is often as fast as, or faster than, state-of-the-art lock-free data structures supporting range queries.

Summary of vCAS Technique

- vCAS is an approach for adding snapshotting and multi-point queries to existing concurrent data structures
  - **Easy-to-use**: simply replace CAS with Versioned CAS
  - **Efficient**: both theoretically and practically
  - **General**: supports a wide range of data structures and multi-point queries
- Code is available on GitHub: https://github.com/yuanhaow/vcaslib
- Full version (with full proof of correctness & DS characterization for supporting multi-point queries) is available on arxiv: https://arxiv.org/abs/2007.02372

Multi-Version Garbage Collection

ANY SYSTEM THAT MAINTAINS MULTIPLE VERSIONS OF EACH OBJECT NEEDS A WAY OF EFFICIENTLY RECLAIMING THEM!

Research Question

How do we garbage collect, efficiently, for multiversion data structures?

Ben-David, Blelloch, Fatourou, Ruppert, Wei, DISC 2021

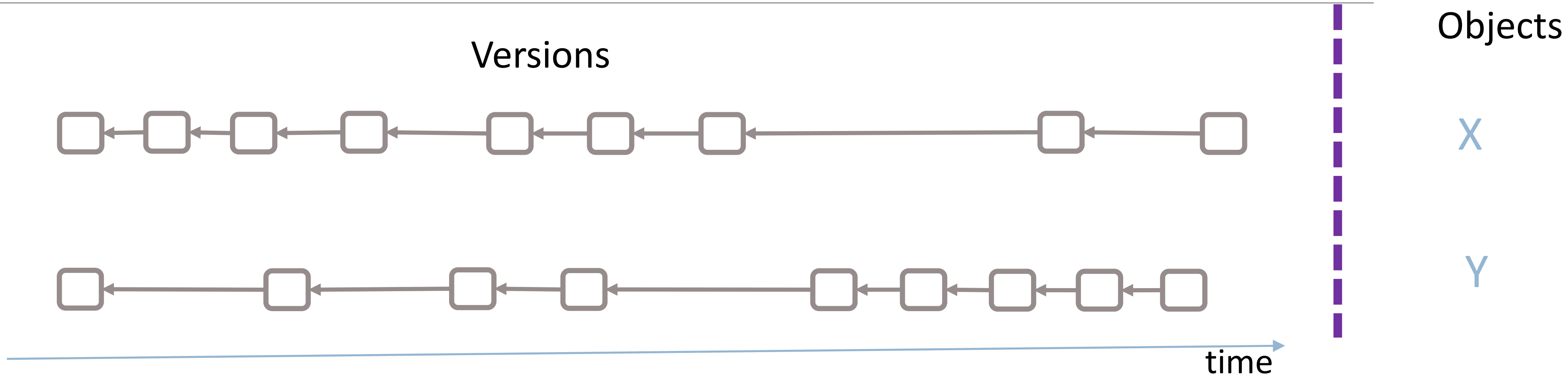
A general Multiversion Garbage Collection (GC) scheme with the following properties:

- **Progress:** wait-free
- **Time**: O(1) per reclaimed version, on average
- **Space**: constant factor more versions than needed, plus an additive term

Previous solutions either use:

- **unbounded space** [Fernandes et al., PPoPP'11] , or
- **O(P)** time per reclaimed version [Lu et al. DISC'13] [Böttcher et al., VLDB'19]
  - **P**: number of processes

Multiversion Garbage Collection (MVGC)

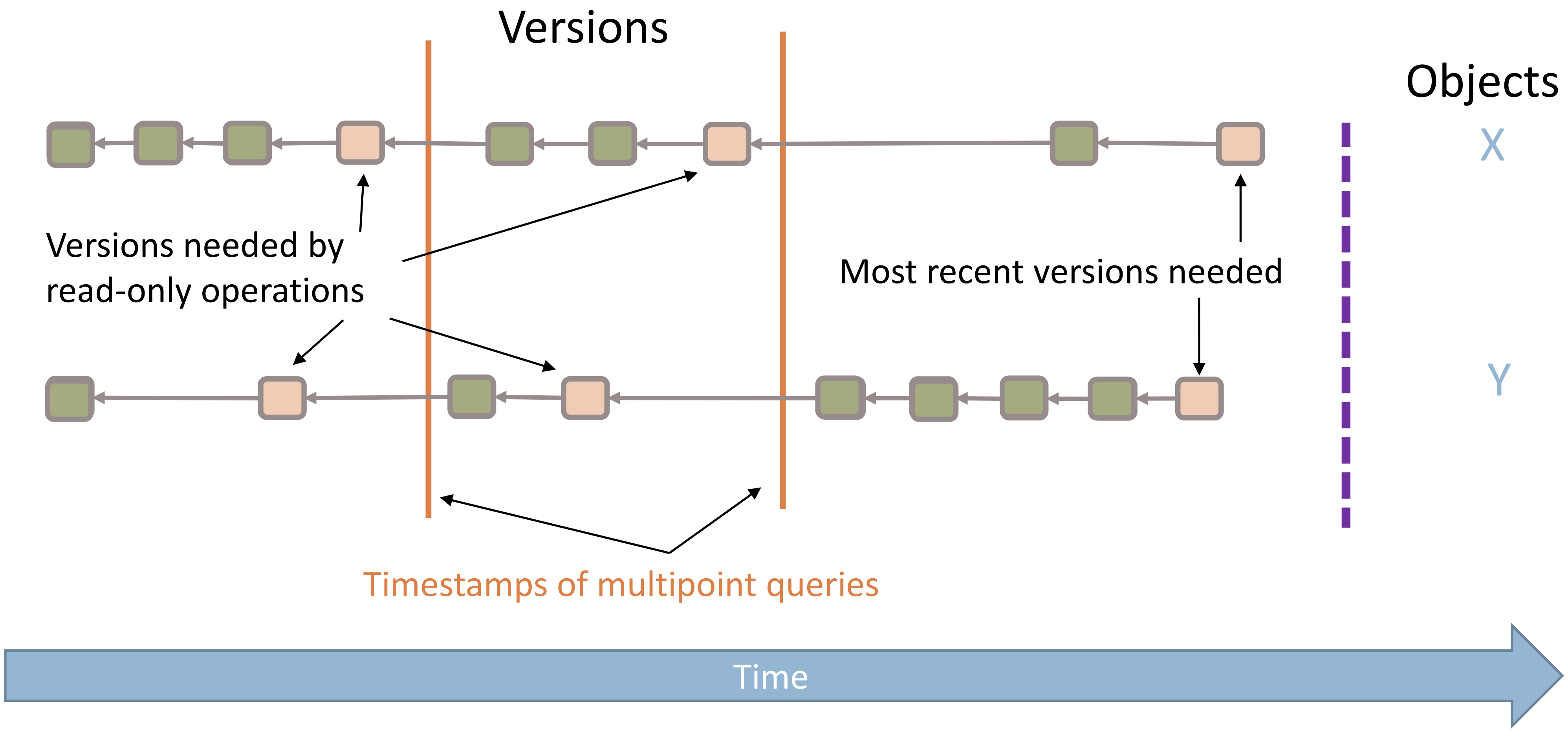


Maintaining all old versions ⇒ high memory usage

How do we identify which versions are not needed?

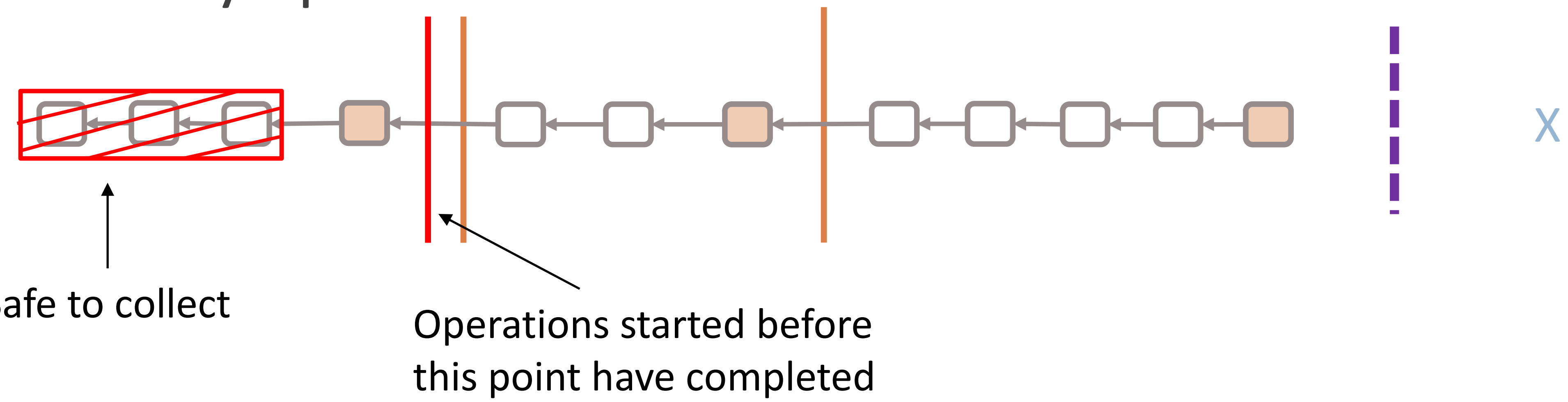
How do we safely reclaim them?

Which Versions are Needed?

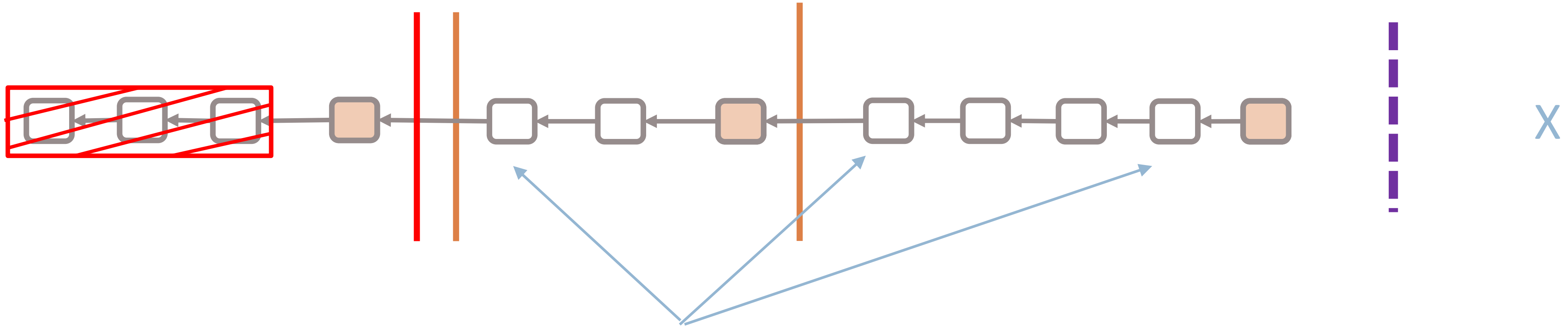


Related Work – Epoch-Based Solutions

- Reclaim versions overwritten before the start of the oldest read-only operation



Related Work – Epoch-Based Solutions



Cons: High space usage

- Unable to collect newer obsolete versions
- Particularly bad with long read-only operations
  - E.g. database scans, large range queries
- Paused process can lead to unbounded space usage

Pros: Fast, easy to implement

Related Work – Other Solutions

Techniques have been developed to address shortcomings of epoch-based solutions.

- GMV **[Lu et al. DISC'13]**, Hana **[Lee et al. SIGMOD'16]**, Steam **[Böttcher et al. VLDB'19]**
- Require Ω(P) time, on average, to collect each version in worst case executions.
  - P: number of processes
- Keep up to P times more versions than necessary

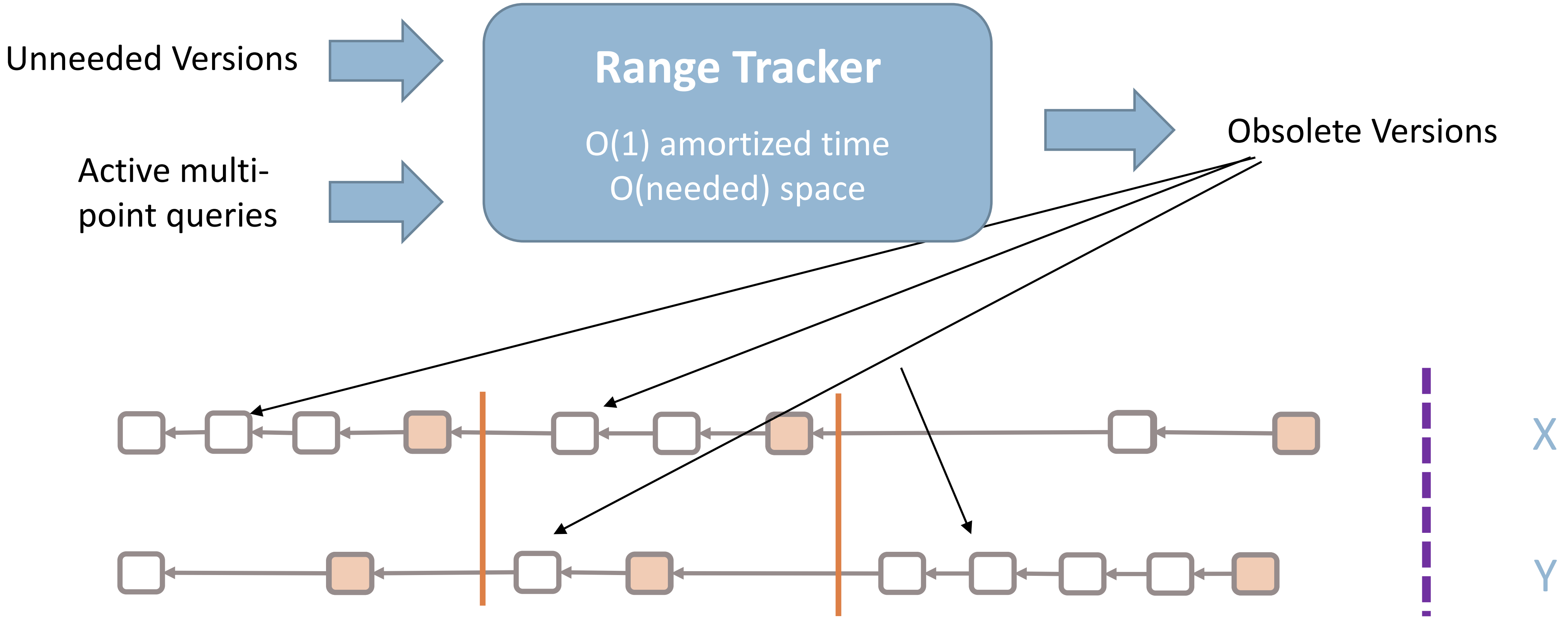
What is the problem to solve?

Step 1: Identify obsolete versions

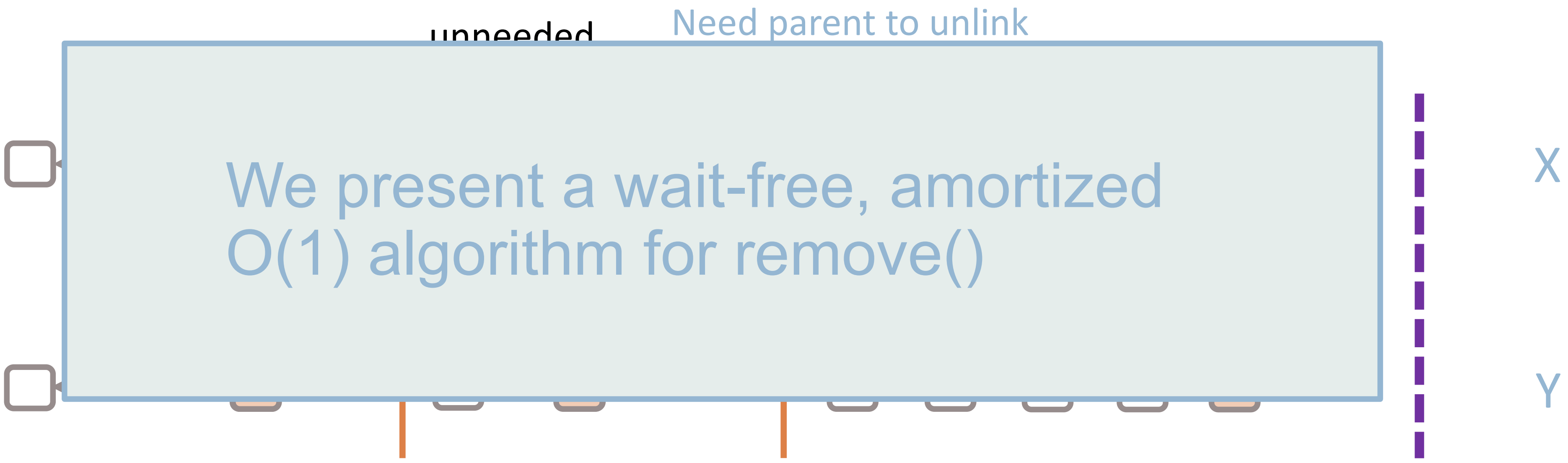
Step 2: Unlink from version list

Step 3: Reclaim memory of unlinked versions

Step 1: Identify obsolete versions

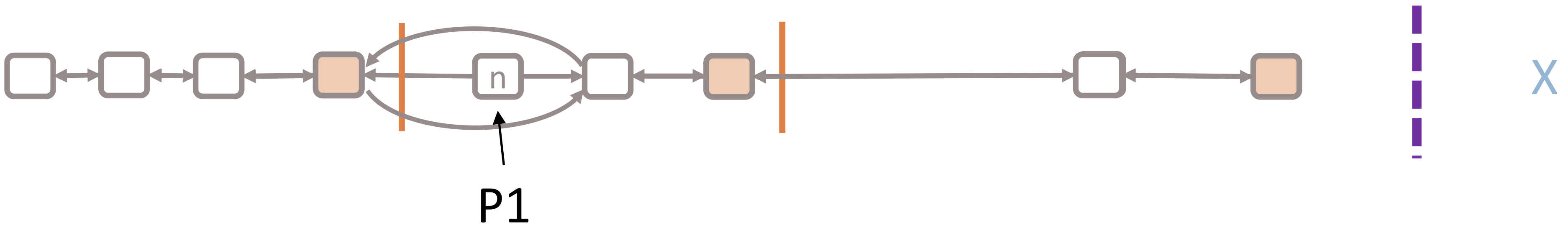


Step 2: Unlink from version list



We present a wait-free, amortized O(1) algorithm for remove()

Step 3: Reclaim memory of unlinked versions



- n is not safe to reclaim right away because a thread (P1) could be paused to access it
- Using Hazard Pointers (HP) or Concurrent Reference Counting (CRC) would solve this problem, but
    - HP sacrifices wait-freedom
    - CRC sacrifices space bounds
- Ben-David et al. presents a new safe reclamation scheme specifically for the doubly-linked version list implementation it provides

Overall Results

Time bounds:

- O(1) time, on average, to identify, remove, and reclaim a version
- Wait-free

Space bounds:

- Number of unreclaimed versions ∈ O(# required versions) + additive term

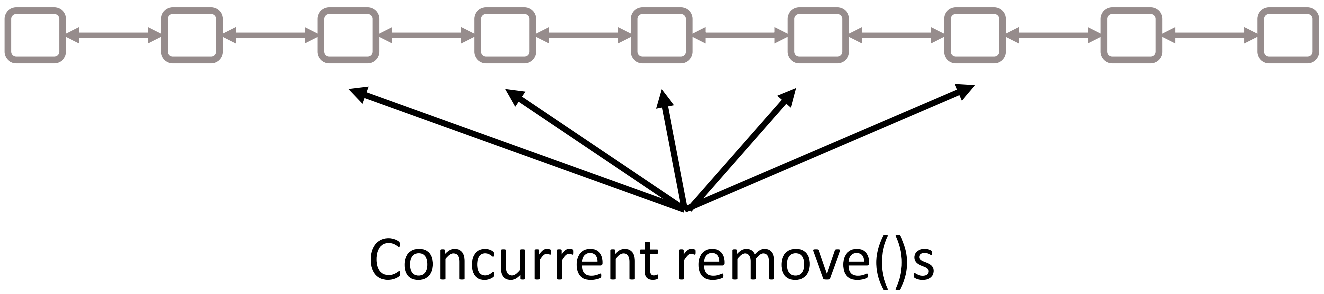
Full version (with proof of correctness) available on arxiv:
https://arxiv.org/abs/2108.02775

New MVGC Schemes

[Wei, Blelloch, Fatourou, Ruppert, PPoPP 2023]

Use range tracker to get good space efficiency

Time efficiency: BBF+ is over optimized for worst-case



- DL-RT: Range tracker + new doubly-linked version list
- SL-RT: Range tracker + new singly-linked version list

Results

Two new MVGC schemes:

- Fast and space efficient in practice
- Strong space bounds in theory

Full paper (with proofs of correctness) is available on arxiv: https://arxiv.org/abs/2212.13557

Code is available on GitHub: https://github.com/cmuparlay/ppopp23-mvgc

Conclusions

The vCAS Approach

- Simple, constant-time approach to take a snapshot of a collection of CAS objects.
- Technique to use snapshots to implement linearizable multi-point queries in many lock-free data structures.
- Adding snapshots to a CAS-based data structure preserves the data structures' asymptotic time bounds.
- Every read is completed within a finite number of instructions (i.e. it is wait-free).

Conclusions

- We present theoretically efficient solutions to the MVGC problem
- Developed new techniques for all 3 steps:
    1. Identify obsolete versions
    2. Unlink from version list
    3. Reclaim memory of unlinked versions
- The MVGC schemes:
    - Provide strong space and time bounds in theory.
    - Space and time efficient in practice.

This work was supported by the Hellenic Foundation for Research and Innovation (HFRI) under the “Second Call for HFRI Research Projects to support Faculty Members and Researchers” (project number: 3684, project acronym: PERSIST).

Thank You!

QUESTIONS?

faturu@csd.uoc.gr

www.ics.forth.gr/~faturu/

We are recruiting!