ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Concurrent Data Structures

Concurrent Algorithms 2017

Igor Zablotchi

(based in part on slides by Tudor David and Vasileios Trigonakis)

Data Structures (DSs)

- Constructs for efficiently storing and retrieving data
  - **Different types**: lists, hash tables, trees, queues, …
- Accessed through the DS interface
  - Depends on the DS type, but always includes
  - Store an element
  - Retrieve an element
- **Element**
  - Set: just one value
  - Map: key/value pair

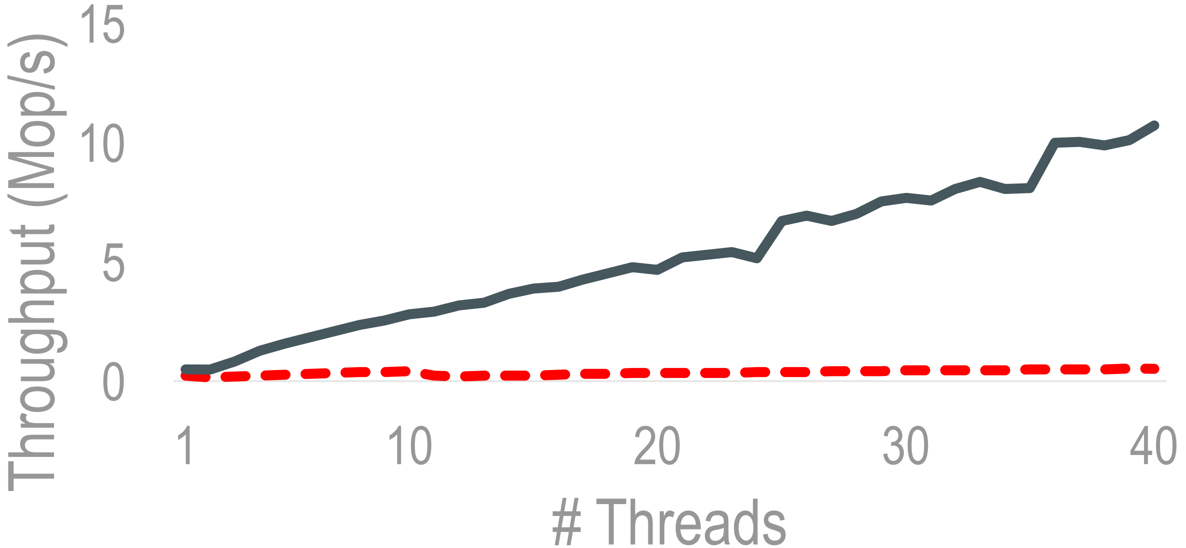
Concurrent Data Structures (CDSs)

- Concurrently accessed by multiple threads
  - Through the CDS interface → linearizable operations!

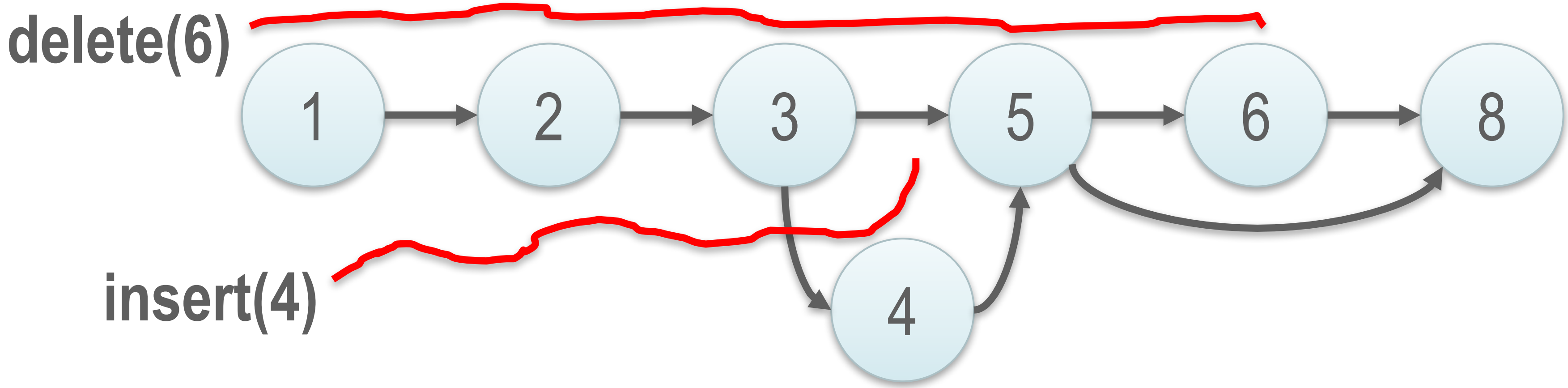
- Really important on multi-cores
- Used in most software systems

What do we care about in practice?

- Progress of individual operations - sometimes
- More often:
  - Number of operations per second (throughput)
  - The evolution of throughput as we increase the number of threads (scalability)



DS Example: Linked List



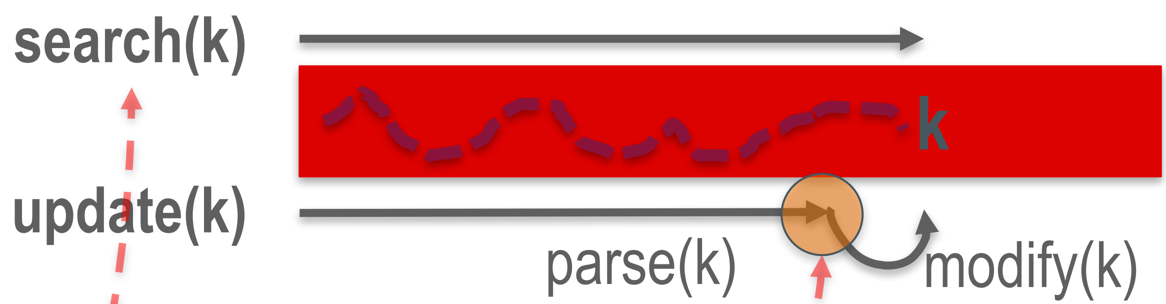
- A sequence of elements (nodes)
- **Interface**
  - search (aka contains)
  - insert
  - remove (aka delete)

```
struct node
{
  value_t value;
  struct node* next;
};
```

Search Data Structures

- Interface
  1. search
  2. insert
  3. remove

  updates



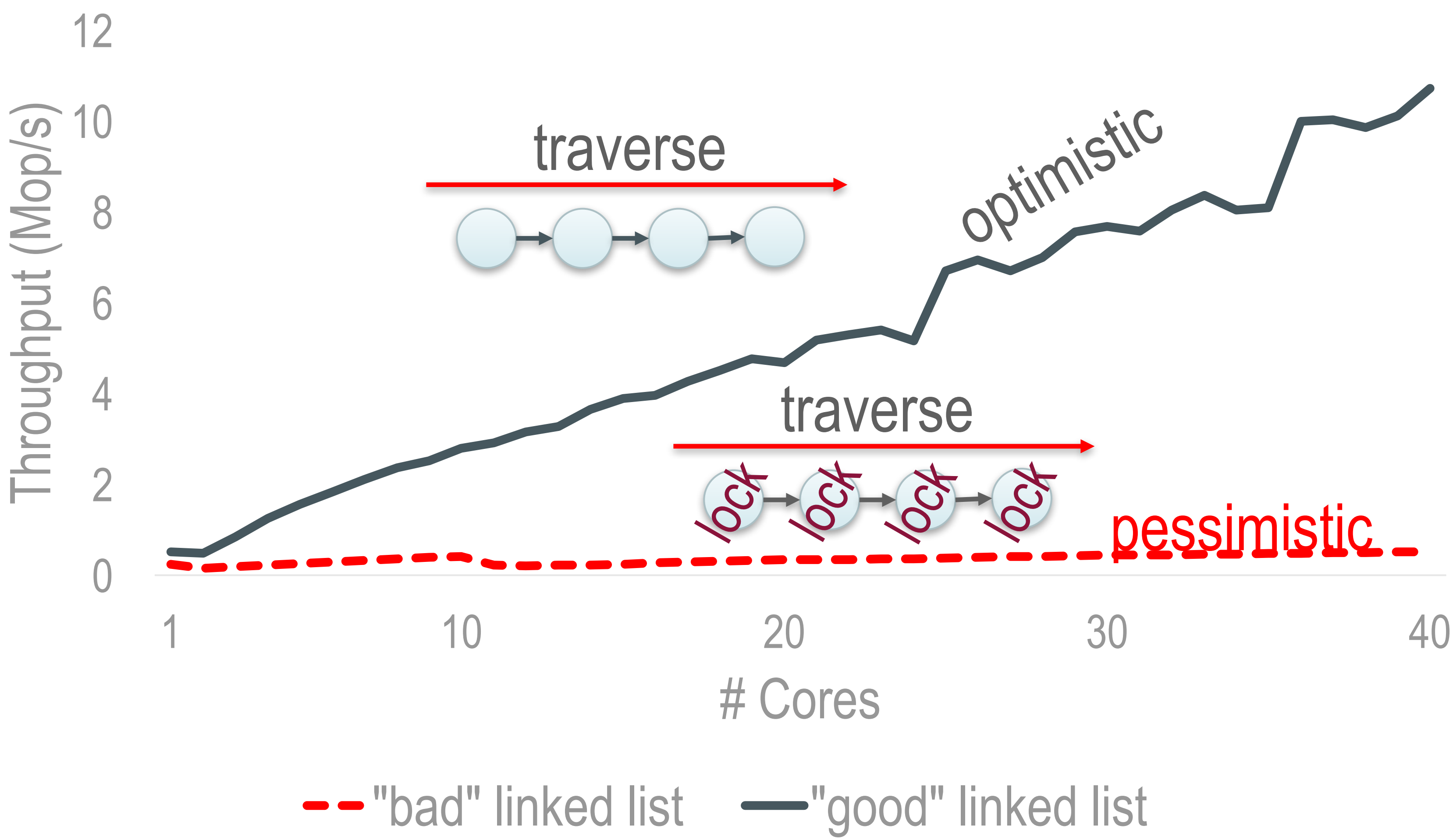
- Semantics
  1. read-only
  2. read-only
  3. read-only
  4. read-write

Concurrency Control

- How threads synchronize their writes to the shared memory (e.g., nodes)
  - Locks
  - CAS
  - Transactional memory

Optimistic vs. Pessimistic Concurrency

20-core Xeon
1024 elements



(**Lesson$_1$**) Optimistic concurrency is the only way to get scalability

Tools for Optimistic Concurrency Control (OCC)

- RCU: slow in the presence of updates
  - (also a memory reclamation scheme)
- STM: slow in general
- HTM: not ubiquitous, not very fast (yet)

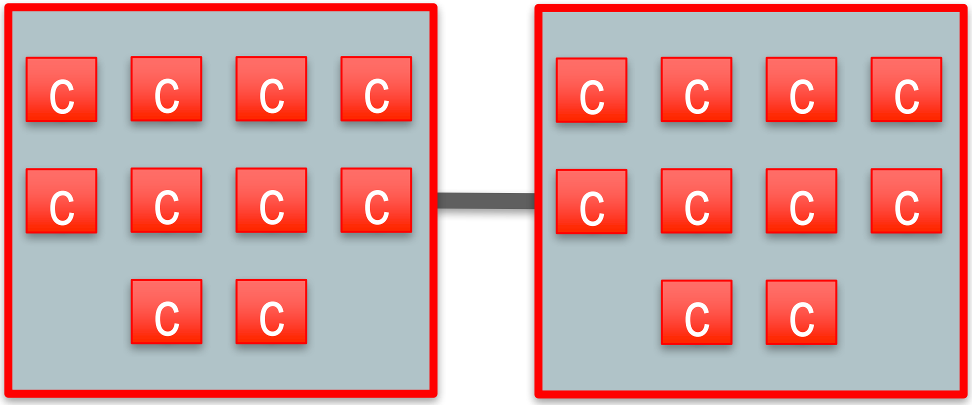
- Wait-free algorithms: slow in general
- (Optimistic) Lock-free algorithms: ☺
- Optimistic lock-based algorithms: ☺

We either need a lock-free or an optimistic lock-based algorithm

Parenthesis: Target platform

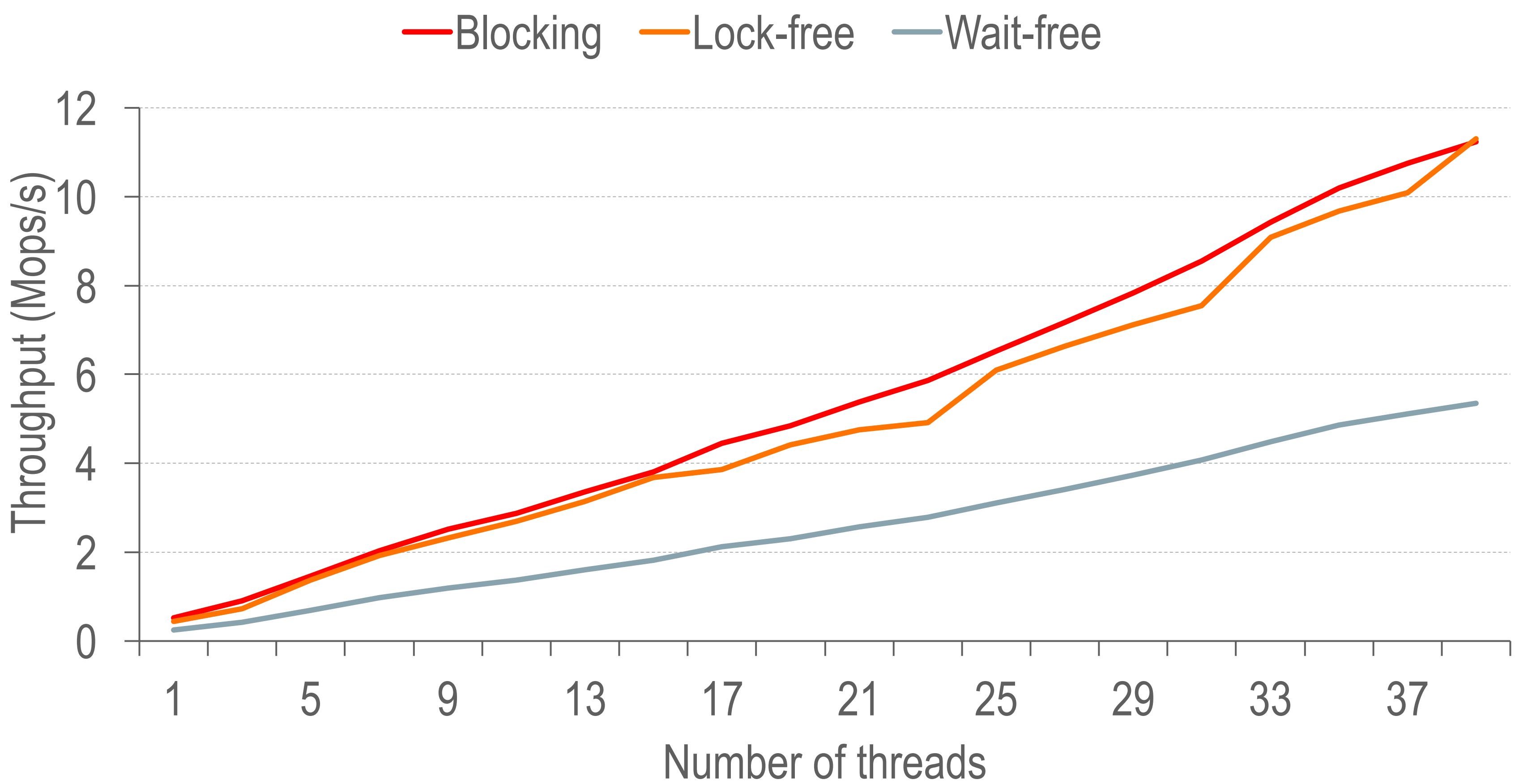
2-socket Intel Xeon E5-2680 v2 Ivy Bridge

- 20 cores @ 2.8 GHz, 40 hyper-threads
- 25 MB LLC (per socket)
- 256GB RAM



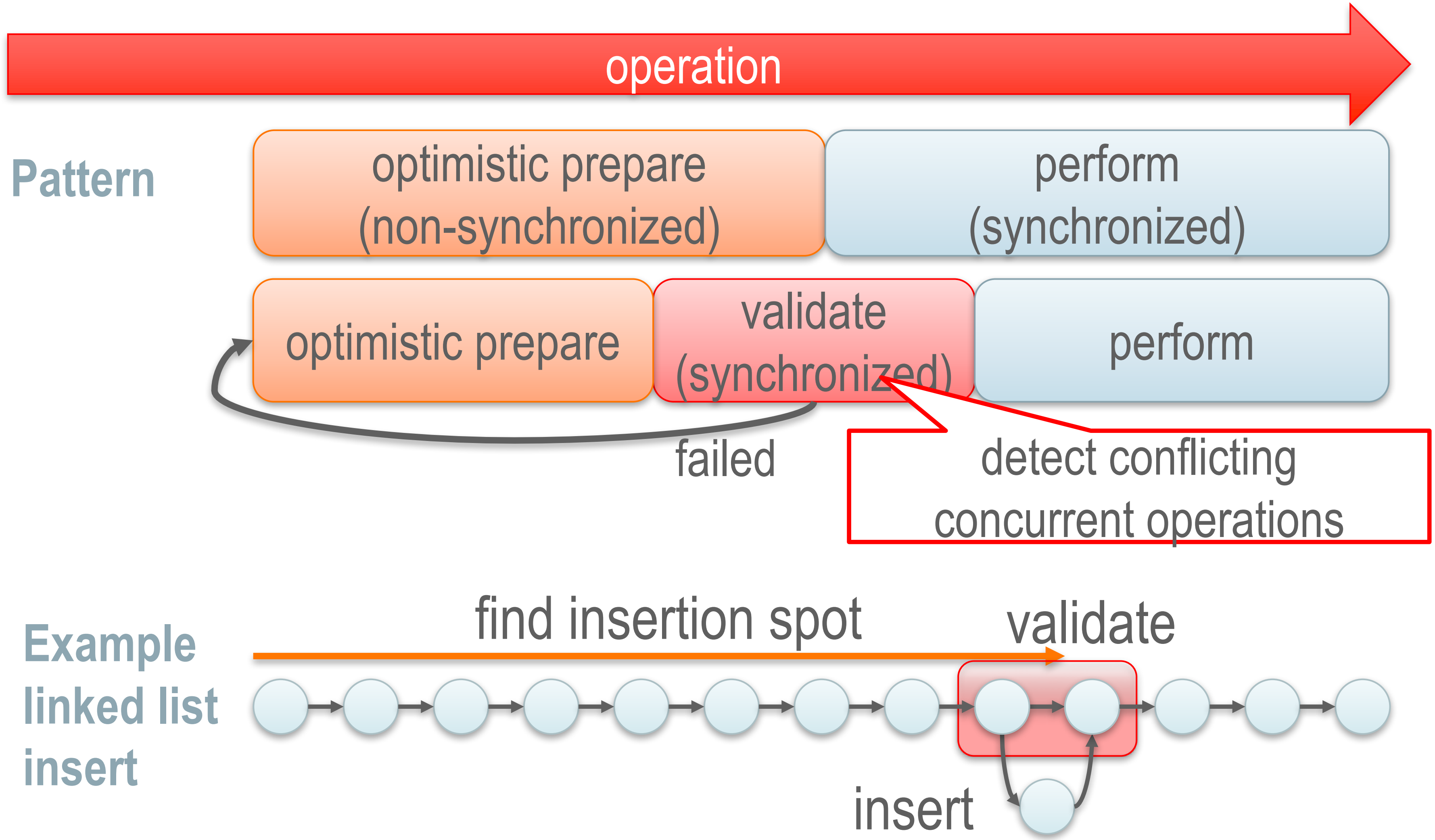
Concurrent Linked Lists – 5% Updates

1024 elements
5% updates



Wait-free algorithm is slow ☹

Optimistic Concurrency in Data Structures



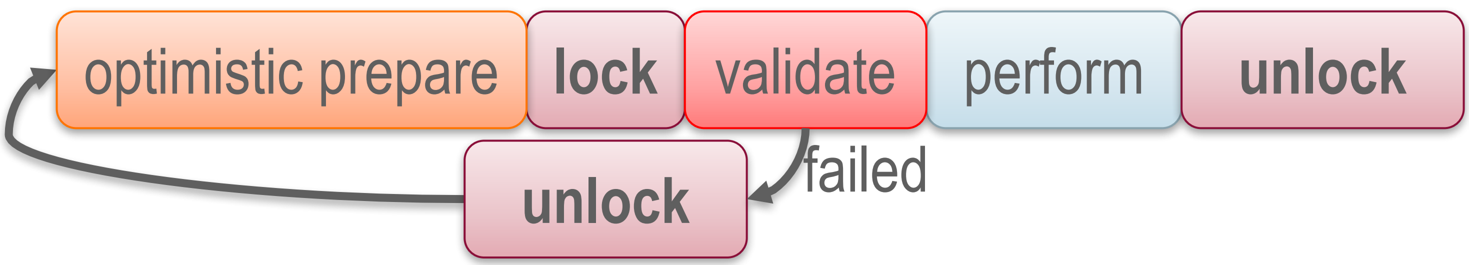
Validation plays a key role in concurrent data structures

Validation in Concurrent Data Structures

- Lock-free: atomic operations

  – marking pointers, flags, helping, …

- Lock-based: lock → validate



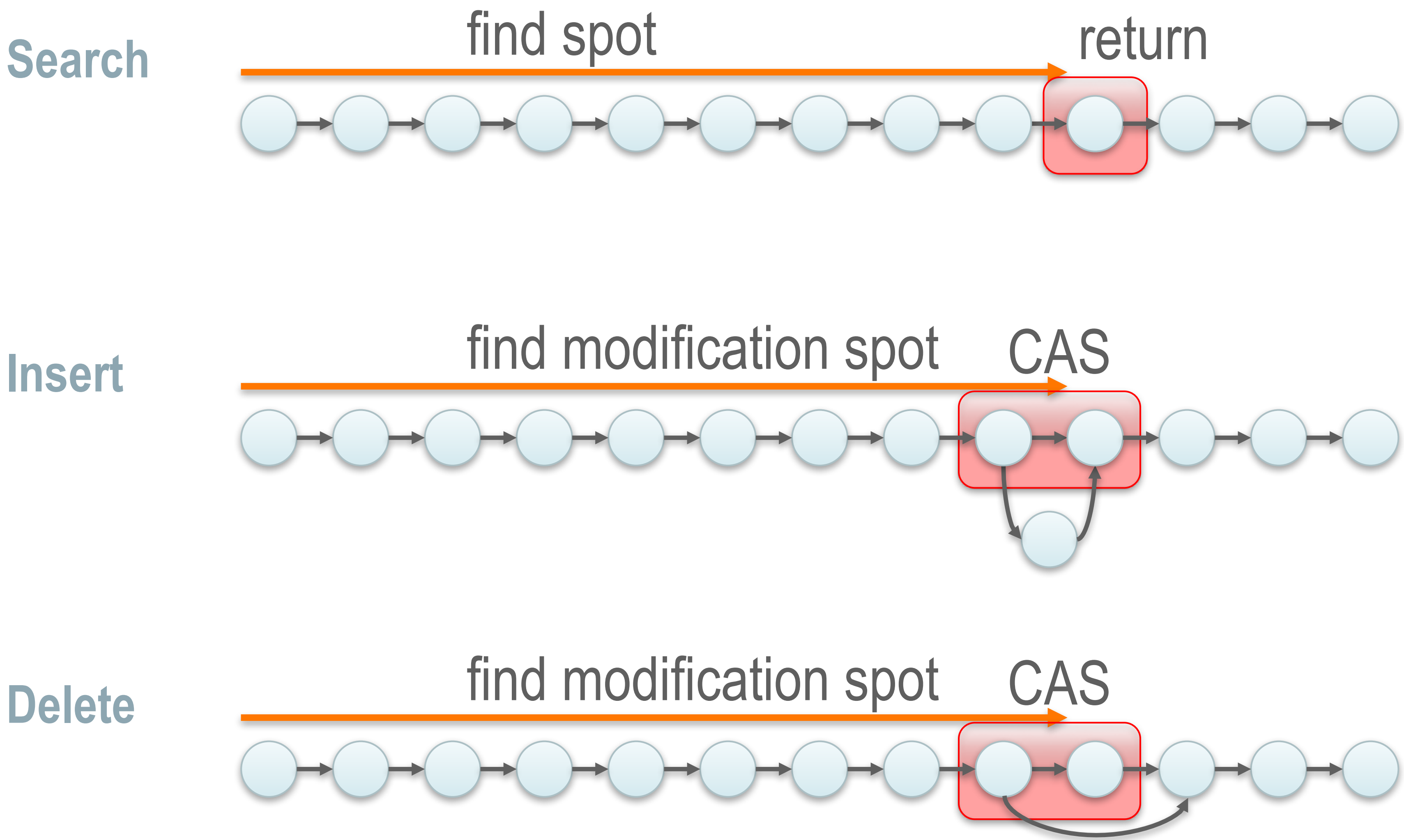
  – flags, pointer reversal, parsing twice, …

Validation is what differentiates algorithms

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

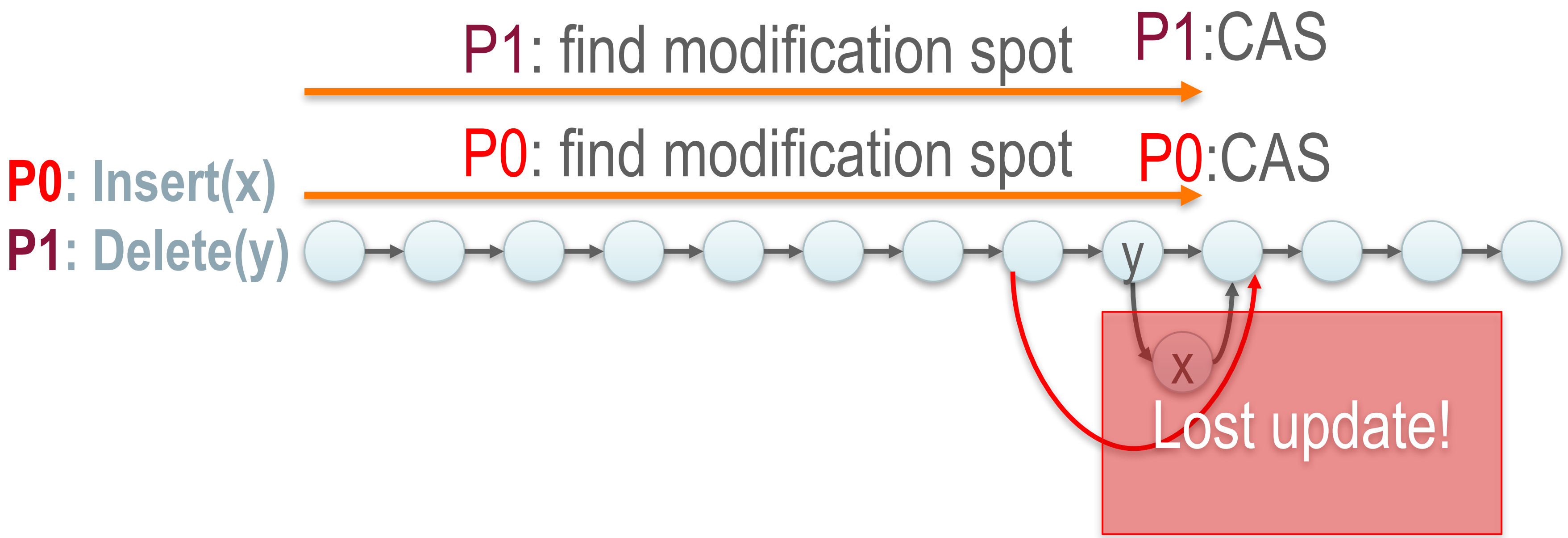
Let's design two concurrent linked lists: A **lock-free** and a **lock-based**

Lock-free Sorted Linked List: Naïve



Is this a correct (linearizable) linked list?

Lock-free Sorted Linked List: Naïve – Incorrect



- **What is the problem?**
  - Insert involves one existing node;
  - Delete involves two existing nodes

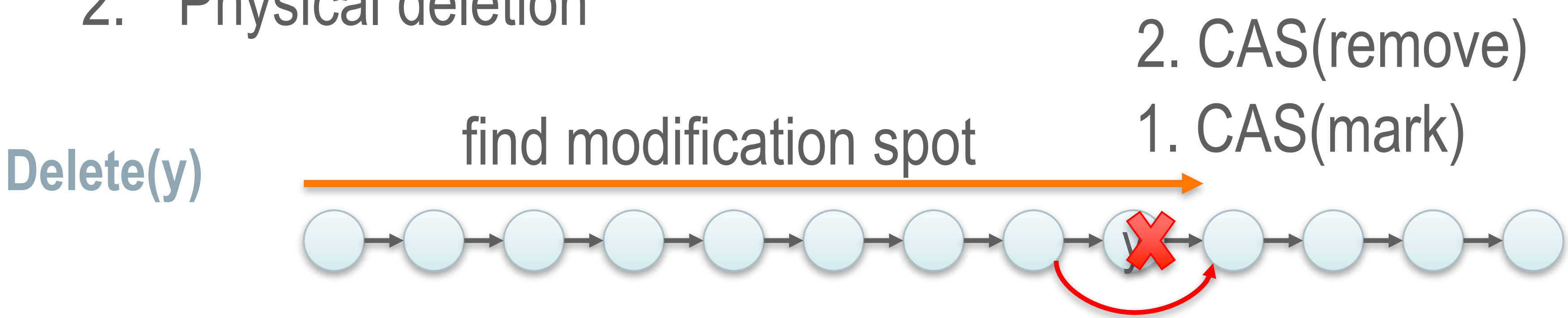
How can we fix the problem?

Lock-free Sorted Linked List: Fix

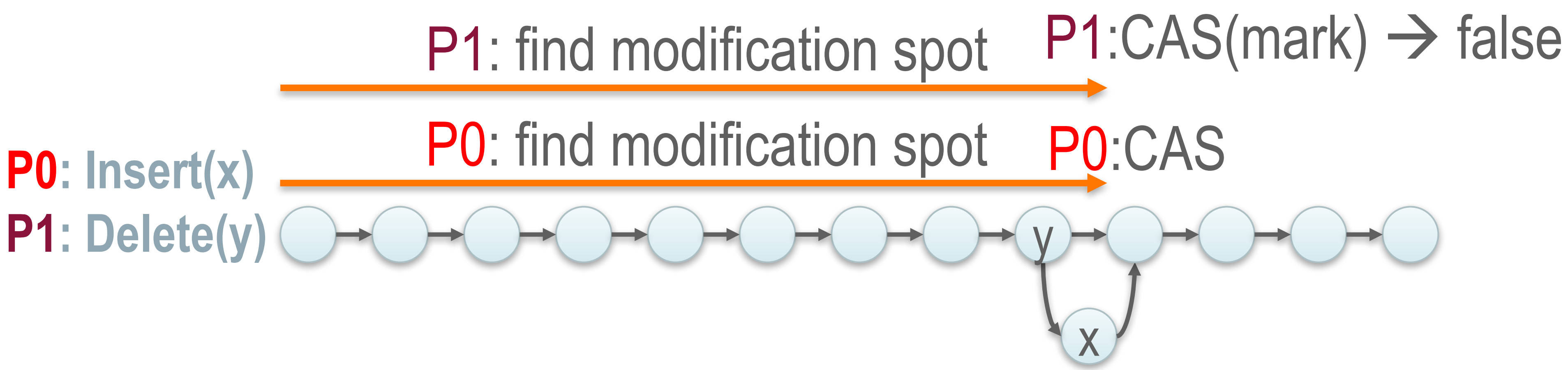
- **Idea**! To delete a node, make it unusable first…
  - **Mark it for deletion** so that
    1. You fail marking if someone changes next pointer;
    2. An insertion fails if the predecessor node is marked.

→ In other words: delete in two steps

1. Mark for deletion; and then
2. Physical deletion

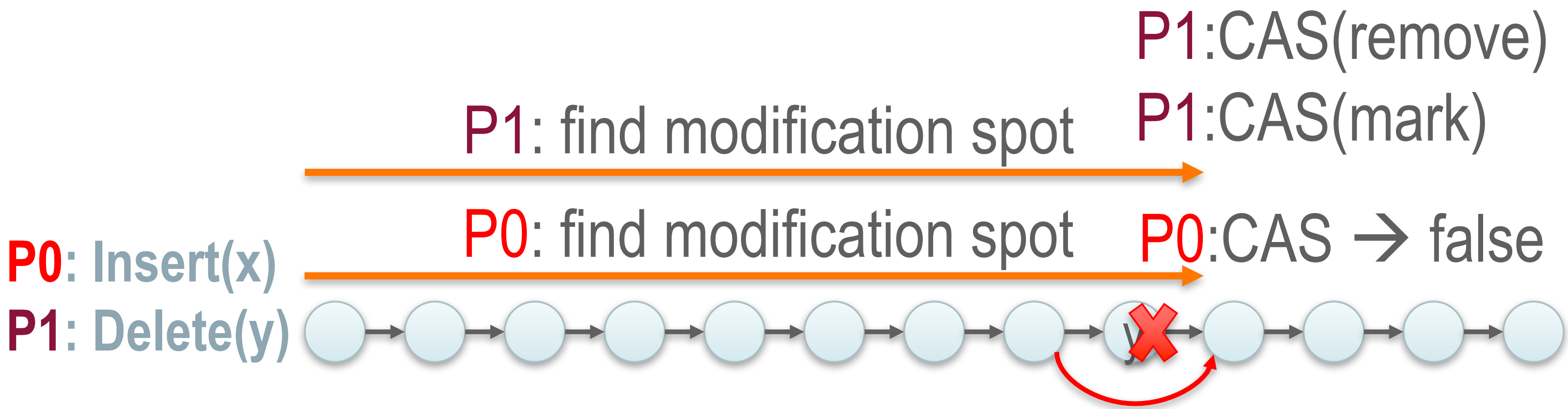


1. Failing Deletion (Marking)



- Upon failure → restart the operation
  - Restarting is part of "all" state-of-the-art-data structures

1. Failing Insertion due to Marked Node

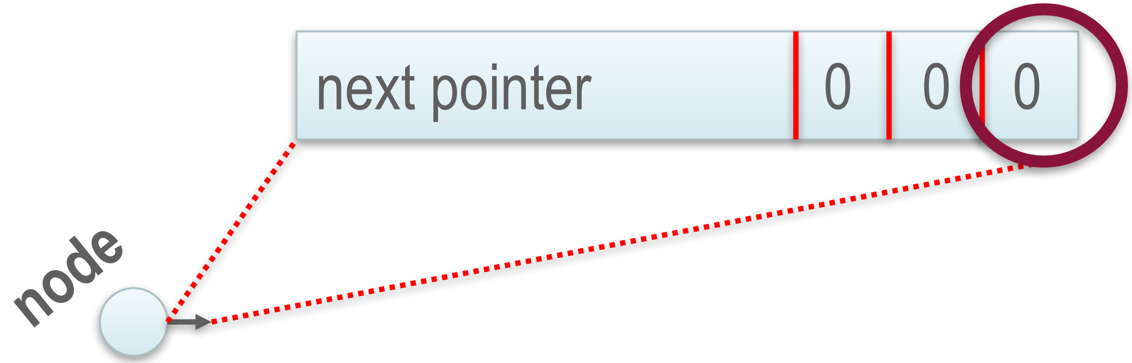


- Upon failure → restart the operation
  - Restarting is part of "all" state-of-the-art-data structures

How can we implement marking?

Implementing Marking (C Style)

- Pointers in 64 bit architectures
  - Word aligned - 8 bit aligned!




```
boolean mark(node_t* n)
    uintptr_t unmarked = n->next & ~0x1L;
    uintptr_t marked   = n->next | 0x1L;
    return CAS(&n->next, unmarked, marked) == unmarked;
```

Lock-free List: Putting Everything Together

- Traversal: traverse (requires unmarking nodes)
- Search: traverse
- Insert: traverse → CAS to insert
- Delete: traverse → CAS to mark → CAS to remove

- **Garbage (marked) nodes**
  - Cleanup while traversing (*helping* in this course's terms)

What happers if this CAS fails??

A pragmatic implementation of lock-free linked lists

What is not Perfect with the Lock-free List?

1. **Garbage nodes**
   - Increase path length; and
   - Increase complexity

     ```
     if (is_marked_node(n)) ...
     ```

2. **Unmarking every single pointer**
   - Increase complexity

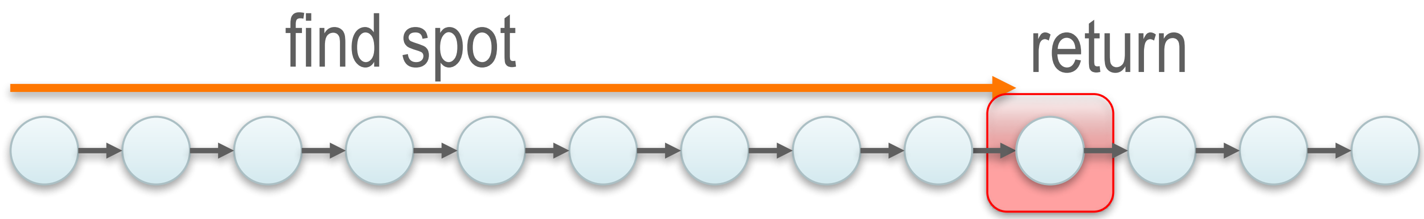
     ```
     curr = get_unmark_ref(curr->next)
     ```

Can we simplify the design with locks?

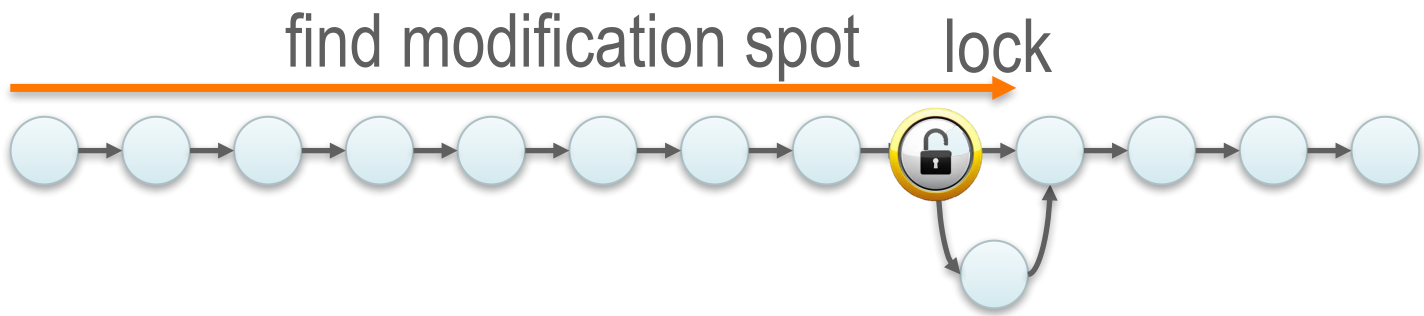
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Lock-based Sorted Linked List: Naïve

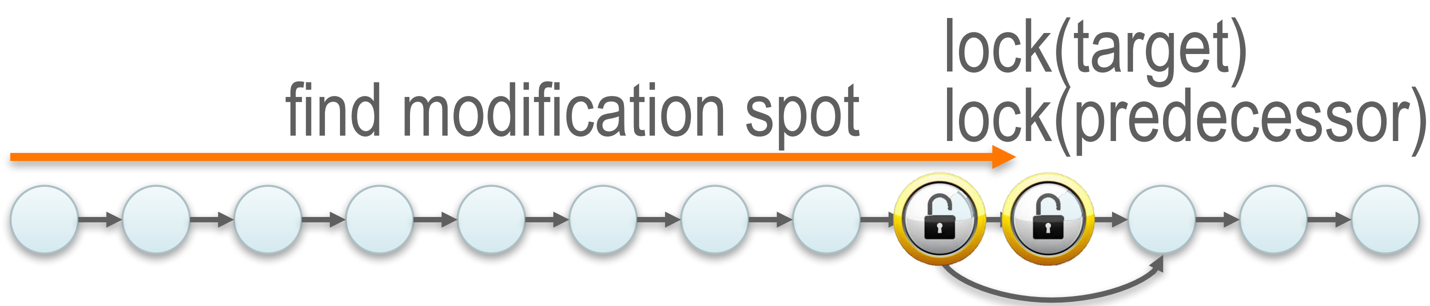
Search



Insert

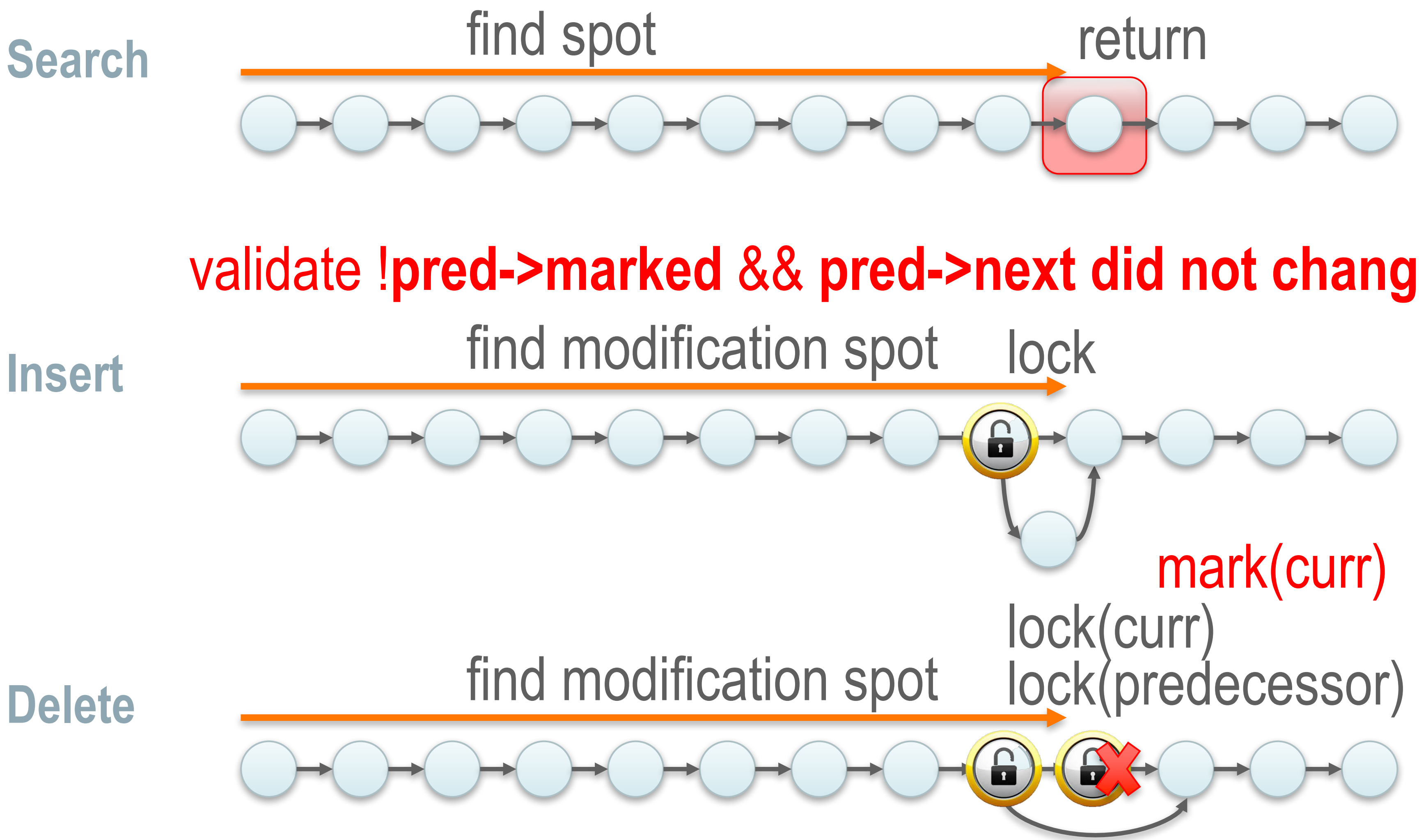


Delete



Is this a correct (linearizable) linked list?

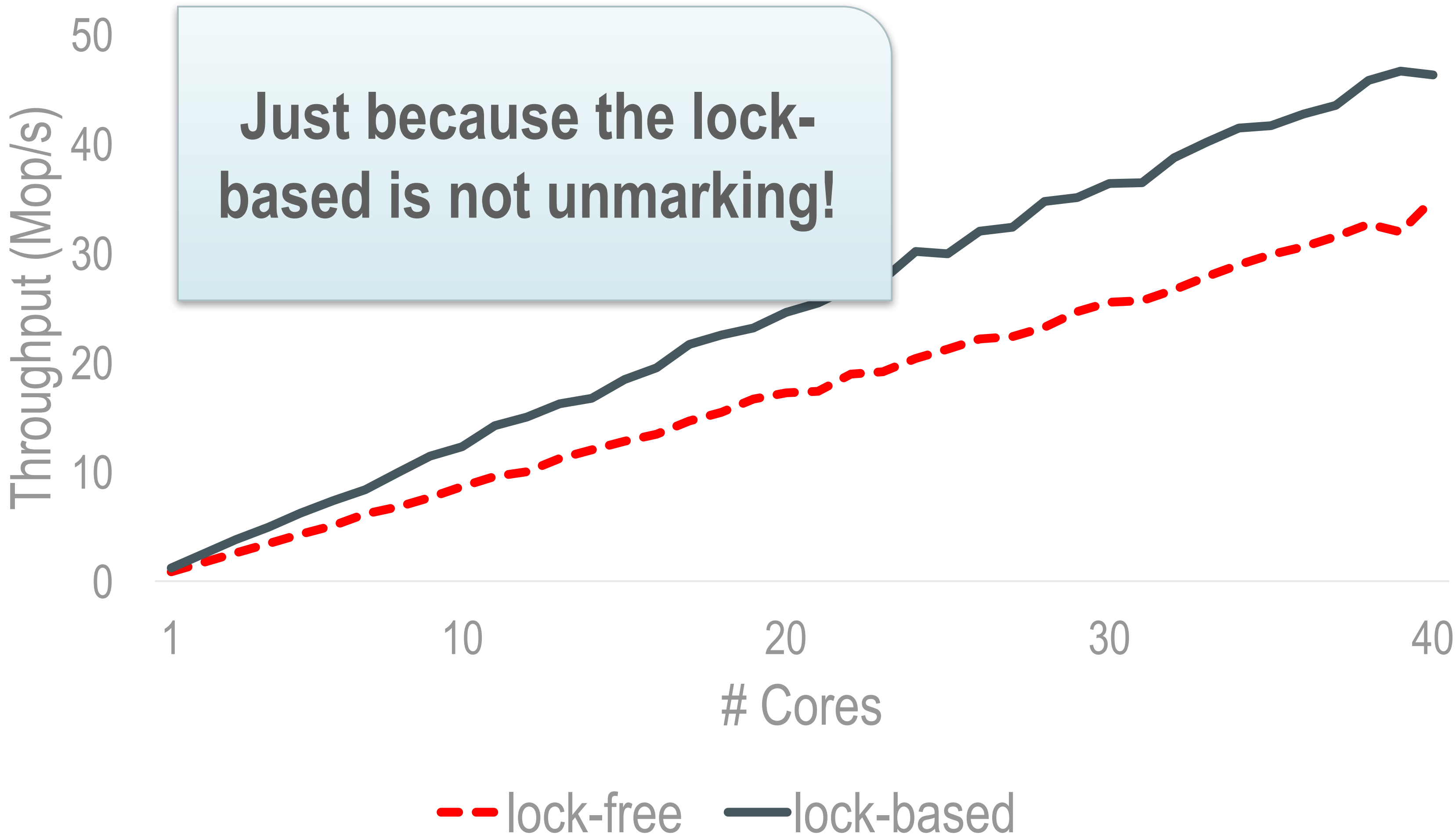
Lock-based List: Validate After Locking



!pred->marked && !curr->marked && pred->next did not change

Concurrent Linked Lists – 0% updates

1024 elements
0% updates



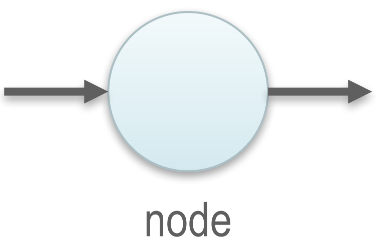
(**Lesson$_2$**) Sequential complexity matters → Simplicity ☺

Another DS Example: the Skiplist

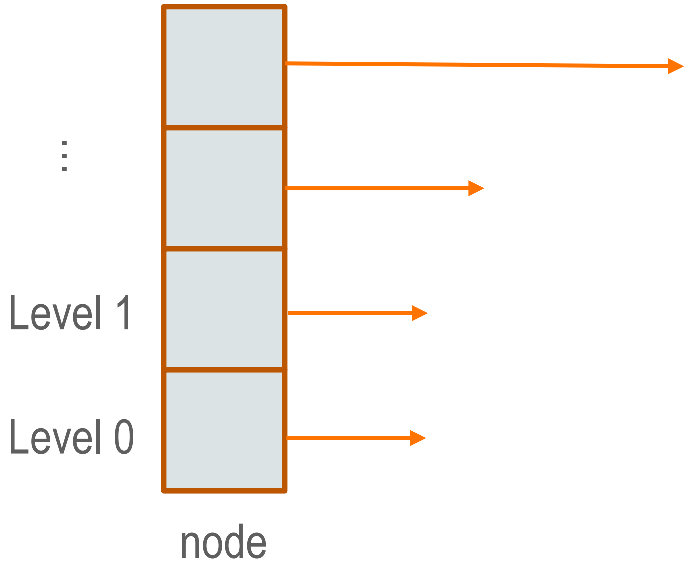
- The linked list is:
  - Easy to understand/design
  - But slow: O(n) for search, insert & remove
- A good alternative: the binary search tree (BST)
  - O(log(n)) search, insert & remove if balanced (else O(n))
  - Needs rebalancing: slow
- An even better alternative: the skiplist
  - O(log(n)) search, insert & remove
  - Builds on the simplicity of the linked list
  - No need to rebalance

Skiplist Overview

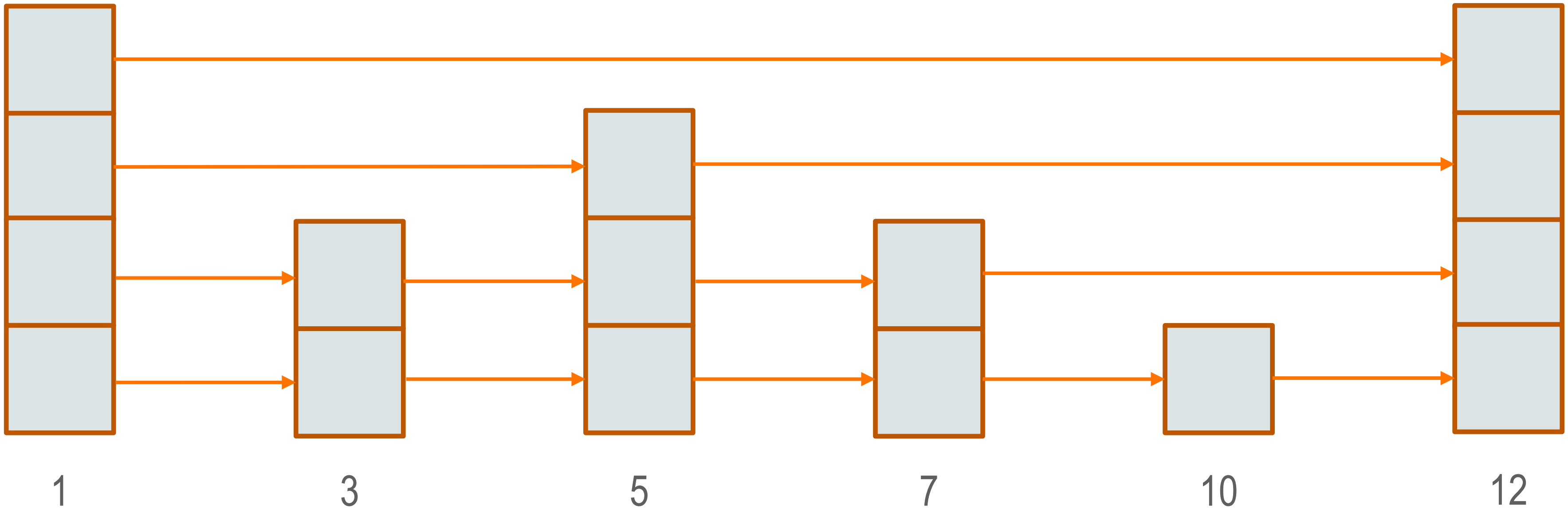
- Linked list:
  - One next pointer per node



- Skiplist:
  - Multiple levels of pointers per node



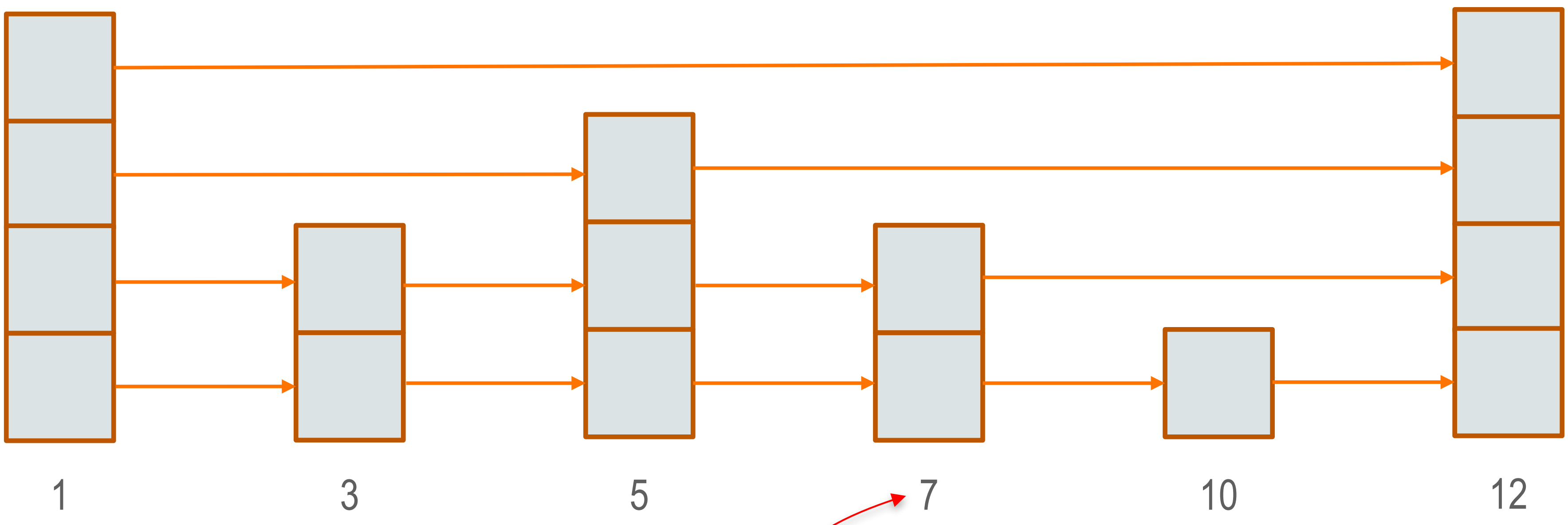
Skiplist Overview



Each node has a random number of levels

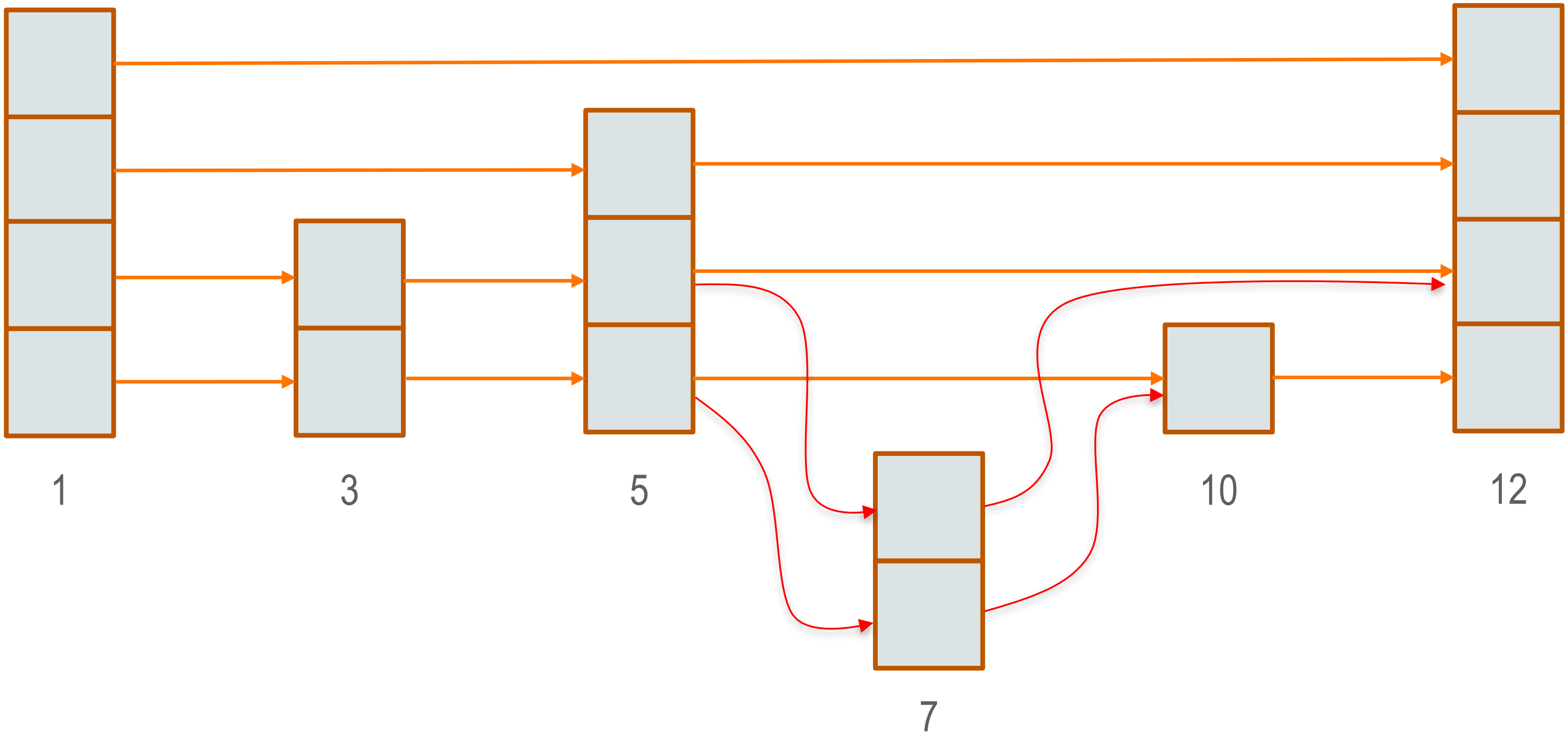
Higher levels are shortcuts for lower levels

Searching in a Skiplist



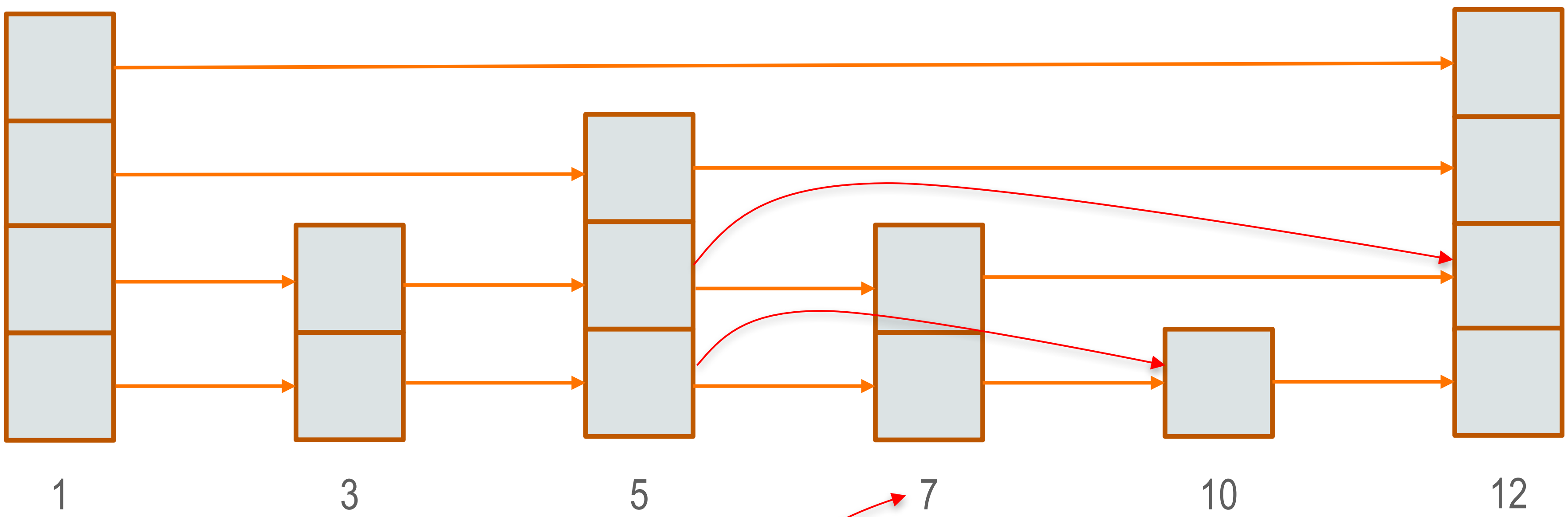
We’re searching for 7!

Inserting in a Skiplist (single-threaded)



We want to insert 7

Deleting from a Skiplist (single-threaded)

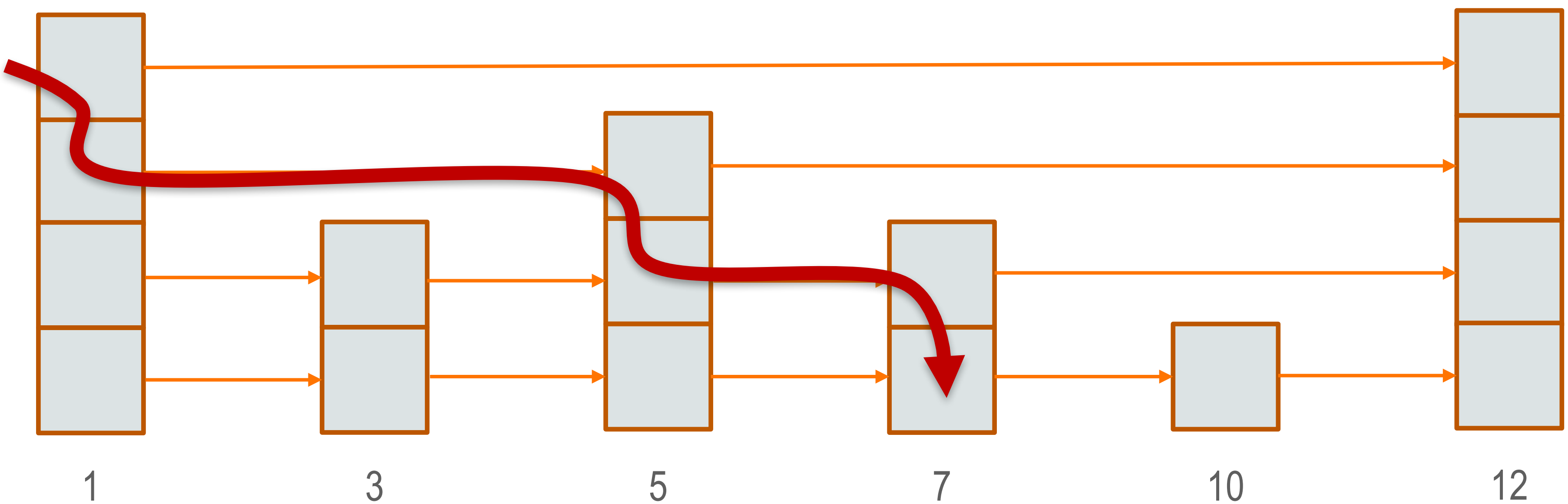


We want to delete 7

Let’s design a lock-free skiplist!

Lock-free Skiplist – Searches

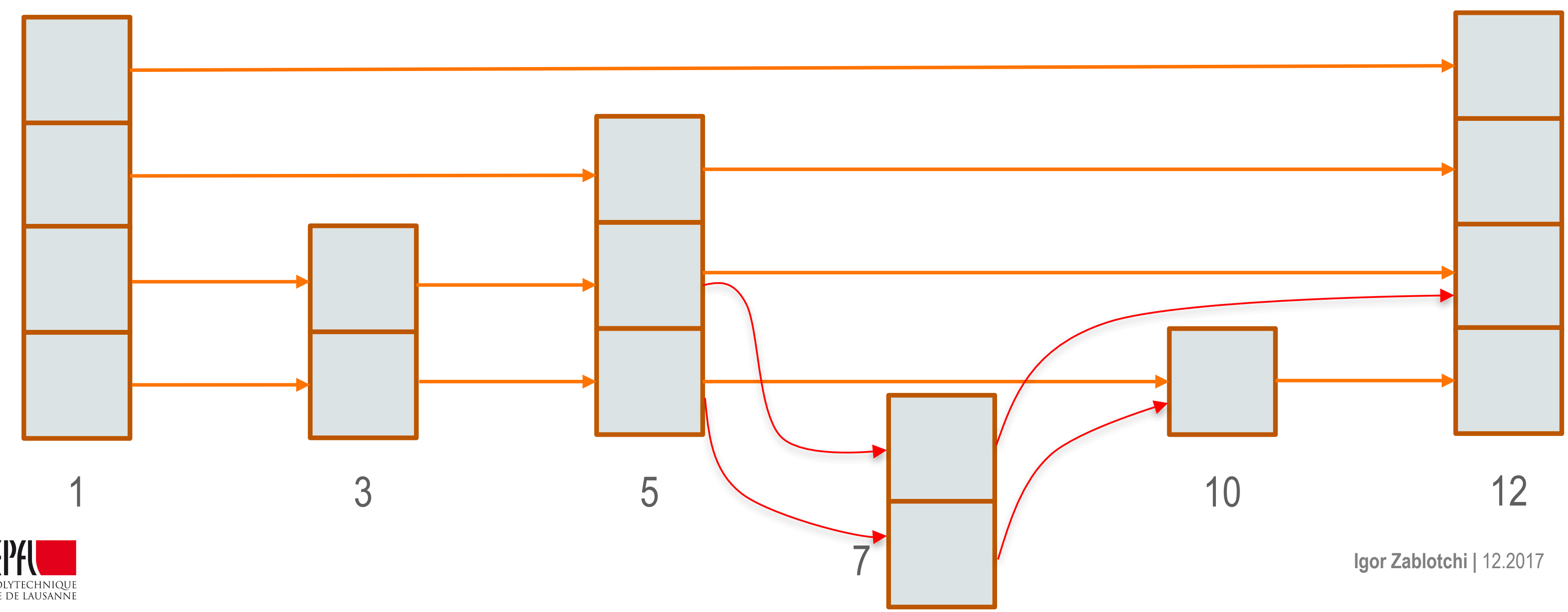
- Similar to the single-threaded case
- Search for the element on every level, starting with the topmost level
- Element is in the skiplist if present on level 0.



ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

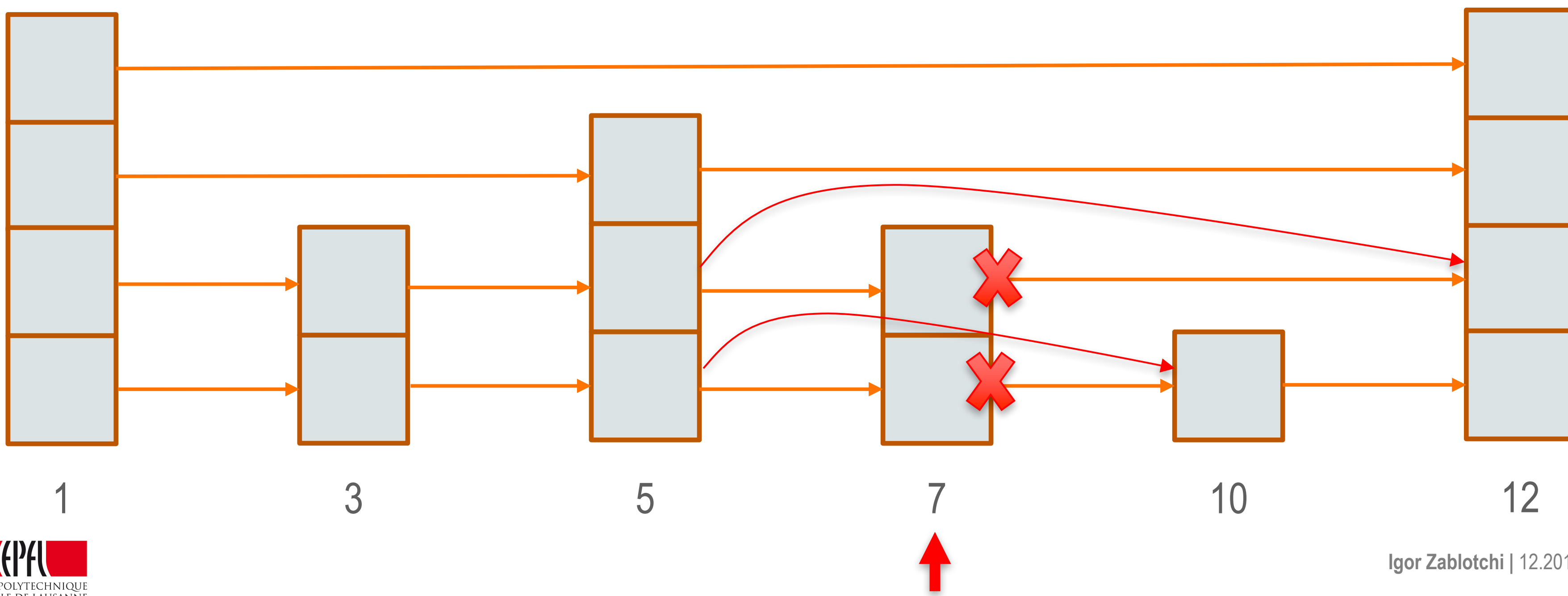
Lock-free Skiplist – Insert

- Randomly choose number of levels of new node
- Find predecessors and successors for new element
- Set element's next pointers to successors
- Atomically link element into level 0 (lin. point)
- Link element into higher levels, one by one



Lock-free Skiplist – Delete

- Find predecessors and successors for element
- Atomically mark element's next pointers one by one, starting from top
- Atomically mark bottom level next pointer (lin. point)
- Unlink marked node from all levels

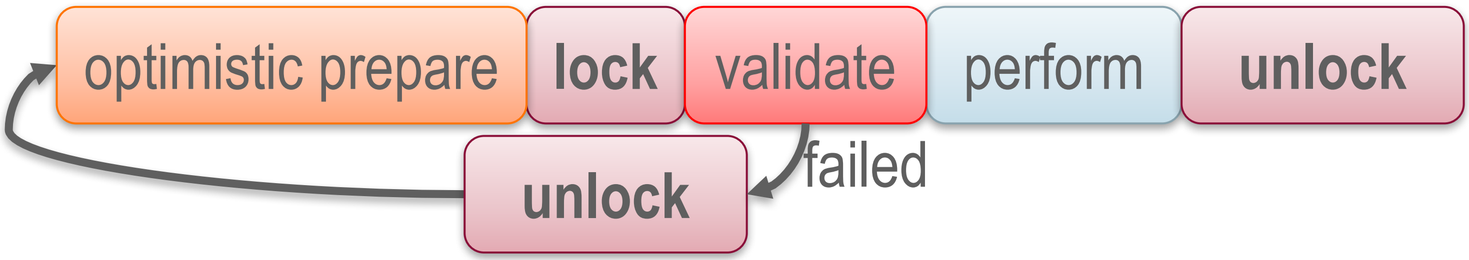


Optimistic Concurrency Control: Summary

- Lock-free: atomic operations

  – marking pointers, flags, helping, …

- Lock-based: lock → validate



  – flags, pointer reversal, parsing twice, …

Summary

- **Concurrent data structures are very important**
- **Optimistic concurrency necessary for scalability**
  - Only recently a lot of active work for CDSs

Word of caution: lock-based algorithms

- Search data structures ☺
- Queues, stacks, counters, ... ☹