Advanced Multiprocessor Programming

Jesper Larsson Träff
traff@par.tuwien.ac.at
Research Group Parallel Computing
Faculty of Informatics, Institute of Computer Engineering
Vienna University of Technology (TU Wien)

Abstract "pool" data stuctures

Pool supports operations on collections of items (that can appear more than once):

Insert (or: set): Put an item into the pool
Delete (or: get): Retrieve an item from the pool

Contains: Check if some item is in pool (not necessarily supported)

Note: Empty? (pool empty?) predicate not considered essential

Pool variants and properties

- Delete on an empty pool returns no element (exception, special value or behavior)
- Bounded vs. unbounded. A bounded pool has an associated capacity, and cannot contain more items than the capacity
- Partial vs. total. A total pool method always succeeds (returns value, or throws exception), a partial pool method may have to wait for some condition (non-empty, or non-full), by definition blocking

- Fairness: pool items are deleted according to some criterion

Queue: first-in, first-out fairness (FIFO)
Stack: last-in, first-out fairness (LIFO)
Priority Queue: fairness according to some order on item keys

Concurrent FIFO Queues (Chap. 10)

Instance of a pool with FIFO fairness. Items are maintained in order of insertion, the oldest item is returned on delete. Often the oldest and the most recent item are referred to as tail and head, respectively (aka: double-ended queue, deque)

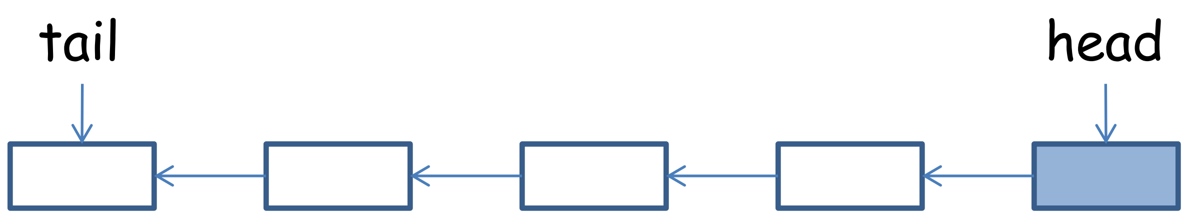
Queue operations:

- enq(x): add item to tail of queue
- y = deq(): return item from head of queue
- (but no operations on specific items, e.g. del(item))

(Semi-formal) Sequential semantics: State of queue described by (ordered) sequence of elements in queue

Our implementation: Linked list of items, plus sentinel element

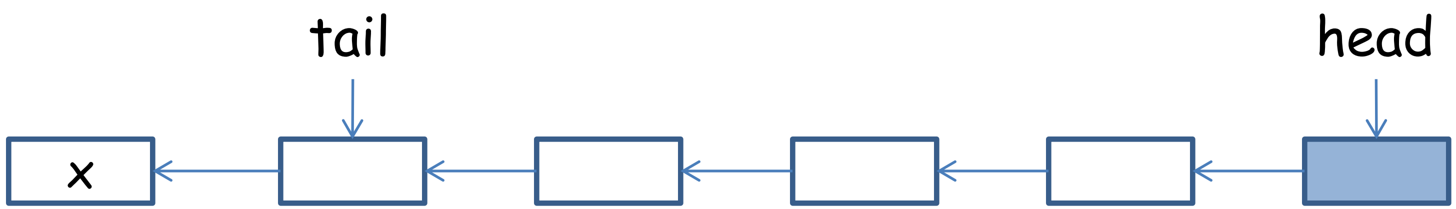
Sentinel: Logically invisible place-holder item (blue), avoids many special cases; here, sentinel is not a fixed element



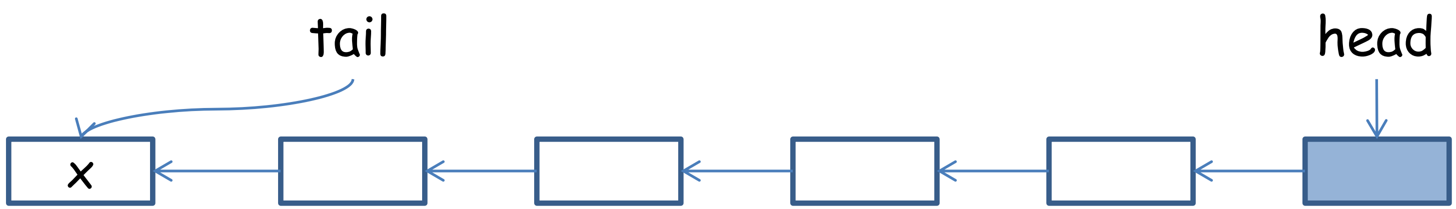
Head points to sentinel element. Queue is empty when sentinel has no successor (head.next=NULL)

Invariant: Item in queue, iff reachable from head

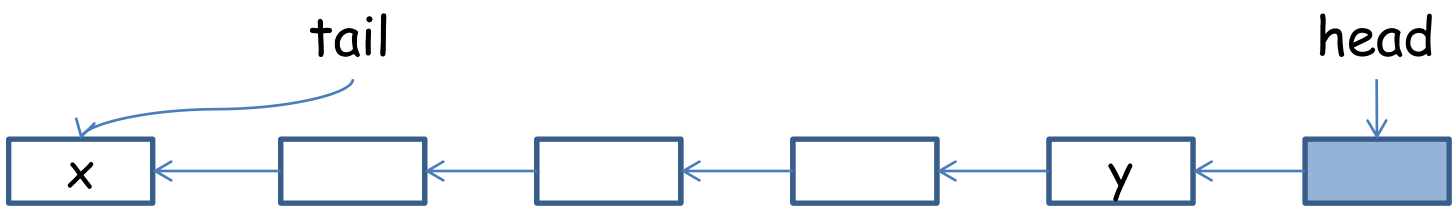
enq(x)



enq(x)

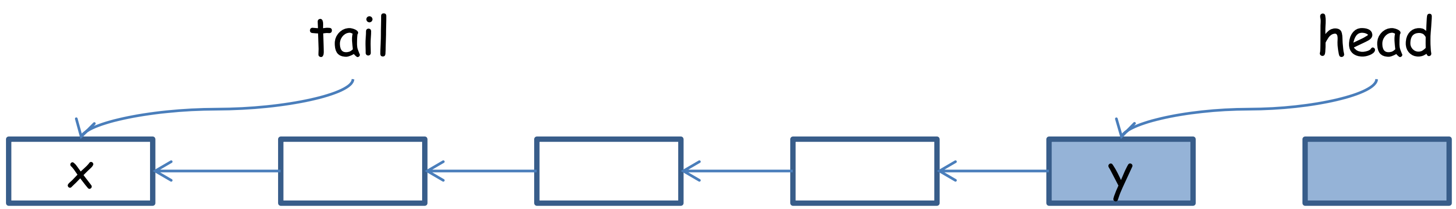


y = deq()



y = head.next.value;

y = deq()



y = head.next.value;

Memory management:
Recycle old sentinel

Concurrent queues

Enqueuers (threads) and dequeuers can work concurrently and independently, synchronization between enqueuers and dequeuers only needed when queue is close to empty (or full)

Example 1:
Bounded, partial queue with fine-grained locking

Use condition variables:
A thread having a lock may release the lock and wait until woken up by other thread; when woken up, it again has the lock (as the sole lock owner: no violation of CS)

`pthreads`: mutex and condition variables

Java implementation:

Queue class maintains capacity and current size of queue

enq(x): if queue is full (size==capacity) wait until other thread dequeues, insert at tail of list, increment size and wakeup waiting dequeuers if needed

deq(): if queue is empty wait until other thread enqueues, delete from head, decrement size and wakeup waiting enqueuers if needed

Linked list implementation. List could/should be replaced with circular array (head and tail indices, head=array[headidx%size])

```
class BoundedQueue {
  ReentrantLock enqlock, deqlock;
  Condition notEmpty, notFull;
  AtomicInt size; // current size of queue
  int capacity;
  volatile Node head, tail;
  public BoundedQueue (int C) {
    capacity = C; // set capacity
    head = new Node(null);
    tail = head;
    size = new AtomicInt(0);
    enqlock = new ReentrantLock();
    deqlock = new ReentrantLock();
    notFull = enqlock.newCondition();
    notEmpty = deqlock.newCondition();
  }
}
```

Condition variables

```
public void enq(T x) {
  boolean wakeup;
  enqlock.lock();
  try {
    while (size.get()==capacity) notFull.await();

    Node e = new Node(x);
    tail.next = e; tail = e;
    if (size.getAndIncrement()==0) wakeup = true;
  } finally enqlock.unlock();

  if (wakeup) {
    deqlock.lock();
    try {
      notEmpty.signalAll();
    } finally deqlock.unlock();
  }
}
```

Condition associated with lock: Acquire

```
public void deq() {
  T x;
  boolean wakeup;
  deqlock.lock();
  try {
    while (size.get()==0) notEmpty.await();
    x = head.next.value; head = head.next;
    if (size.getAndDecrement()==capacity) wakeup = true;
  } finally deqlock.unlock();

  if (wakeup) {
    enqlock.lock();
    try {
      notFull.signalAll();
    } finally enqlock.unlock();
  }
  return x;
}
```

Condition variable reminder:
When woken up, always recheck condition. It could have changed if wakeup is not performed in CS, on spurious wakeups, and if more threads are woken up

This implementation:
Contention on atomic size variable between enqueuers and dequeuers; can be remedied (idea: two counters, maintain upper/lower bound on number of elements)

Note:
Abstract queue's head and tail not always equals head and `tail`. The linearization point of enq(x) is the point where tail.next is set to the new item (not the point where tail is updated)

Synchronization constructs (reminder)

- Lock with condition variables (Java, pthreads, C++ threads,…)
- Monitor: Language construct, additional structure, safety guarantees that can be ensured by compiler
- Semaphore (binary, counting)

All three constructs equally powerful, either can be emulated in terms of the others

Counting semaphores are no more powerful than binary semaphores

Example 2:
Unbounded, total queue with (not so) fine-grained locking

- Liveness: Properties inherited from lock implementation, e.g. starvation free
- Progress: Blocking
- Correctness: Linearizable (enq(x) linearizes at the point where tail.next is set; deq() linearizes when head is updated to head.next)

```java
public void enq(T x) {
  enqlock.lock();
  try {
    Node e = new Node(x);
    tail.next = e; tail = e;
  } finally enqlock.unlock();
}
```

```java
public void deq() {
  T x;
  deqlock.lock();
  try {
    if (head.next==null)
      throw new EmptyException();
    x = head.next.value; head = head.next;
  } finally deqlock.unlock();
  return x;
}
```

Example 3:
Unbounded, total, lock-free queue

Idea:
Use CAS (Java: compareAndSet) to update list (pointers in C++)

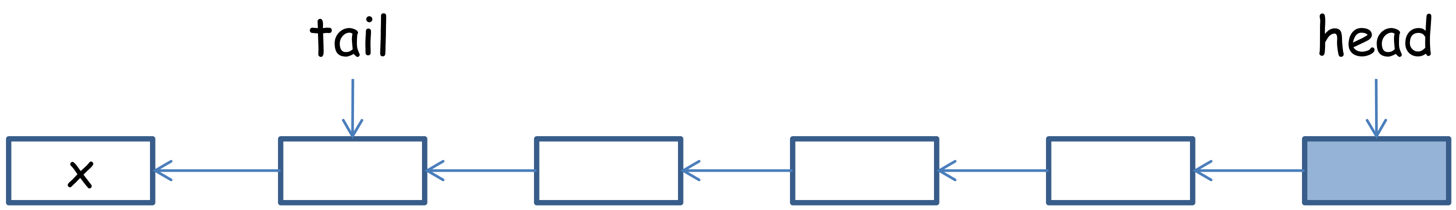
```
class Node {
  public T value;
  public AtomicReference<Node> next;
  public Node(T x) {
    this.value = x;
    next = new AtomicReference<Node>(null);
  }
}
```

Difficulties:

- enq(x) consists of two operations
  - Updating next-field of tail
  - Updating tail
- Other threads may encounter unfinished updates

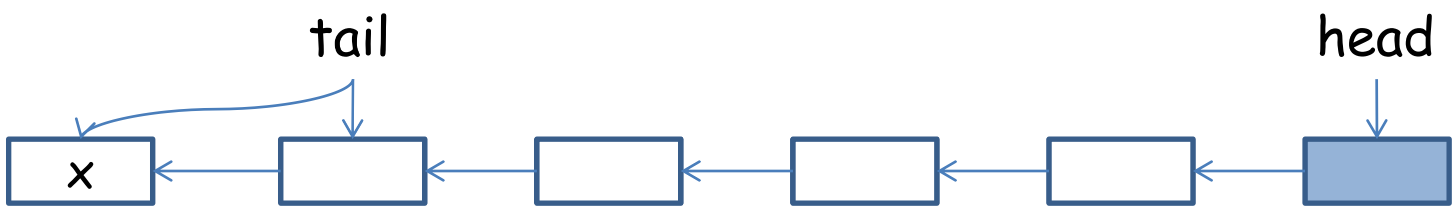
Solution: Make lazy with helper scheme

enq(x)



Try to update tail.next to x with CAS. Condition: tail.next must be **null**

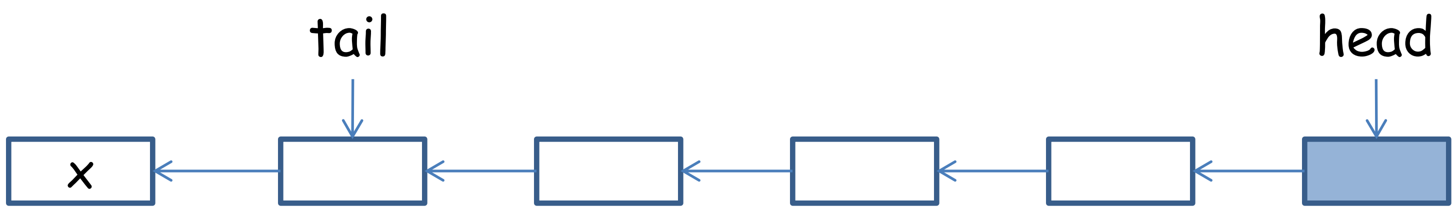
enq(x)



Try to update tail.next to x with CAS. Condition: tail.next must be **null**

Update tail to x with CAS

enq(x)

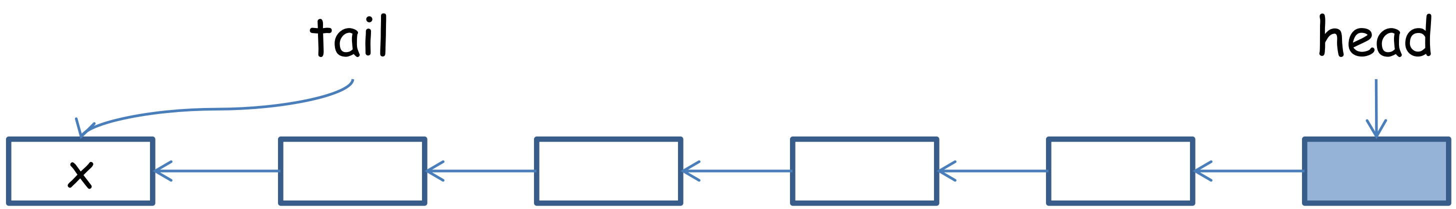


Try to update tail.next to x with CAS. Condition: tail.next must be **null**

If not, some other thread is performing a concurrent enq, help by advancing tail with CAS, try again

```
public void enq(T x) {
  Node node = new Node(x);
  while (true) {
    Node last = tail.get();
    Node next = last.next.get();
    if (next==null) {
      if (last.next.compareAndSet(null,node)) {
        tail.compareAndSet(last,node);
        return;
      }
    } else {
      tail.compareAndSet(last,next); // help
    }
  }
}
```

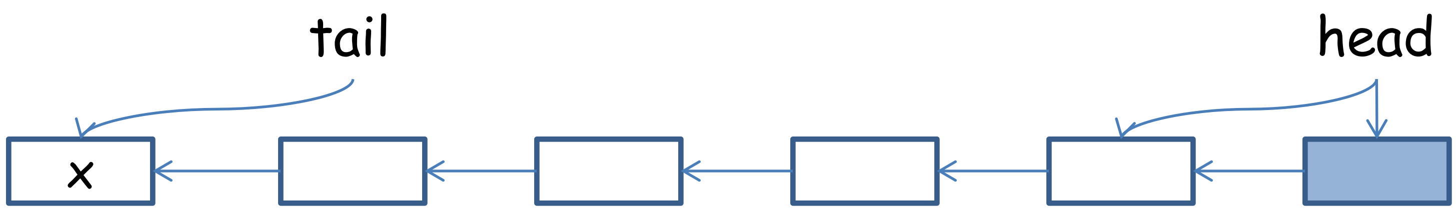
y = deq()



If queue is empty (head==tail, and no item reachable from head), throw exception

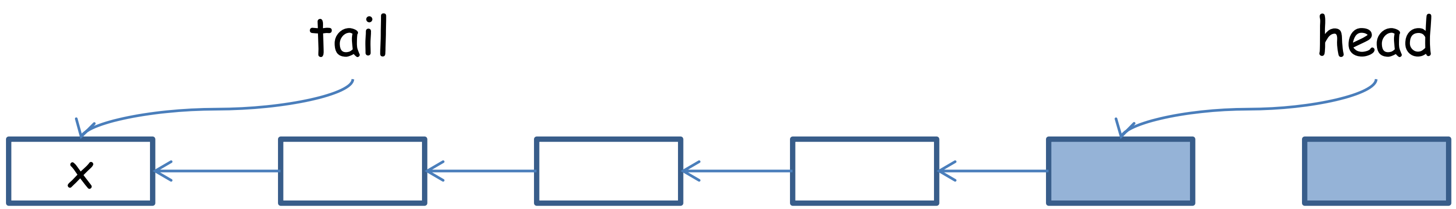
If there is an item reachable from head (by concurrent enq), help by advancing tail with CAS

y = deq()



Otherwise, return head.next, if head can be correctly advanced by CAS

y = deq()

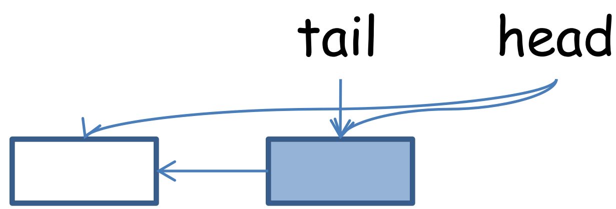


Otherwise, return head.next, if head can be correctly advanced by CAS

```
public void deq() {
  while (true) {
    Node first = head.get();
    Node last  = tail.get();
    Node next  = first.next.get();
    if (first==last) {
      if (next==null) throw new EmptyException();
      tail.compareAndSet(last,next); // help
    } else {
      T x = next.value;
      if (head.compareAndSet(first,next)) return x;
    }
  }
}
```

The need for help

1. Enqueuer has succeed in setting tail.next
2. Dequeuer updates head



3. Tail would refer to a removed node

Properties of lock-free queue

- Linearizable
  - enq: when CAS setting of next field succeeds
  - deq: when CAS updating of head succeeds

- Lock-free, but not wait-free (if CAS fails, some other thread must have succeeded)

Check:
Without helper scheme, enq would not be lock-free (other enq'er could go to sleep before updating tail, tail would not advance)

The ABA problem

Instead of giving old sentinel nodes back to memory allocator, they could be explicitly reused:

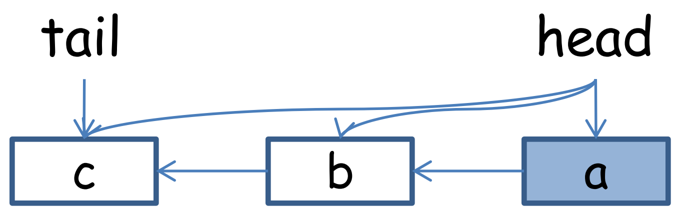
Keep thread-local free-lists, on deq link old sentinel node into free-list, on enq get free node from free-list
For load balance: Threads could steal free nodes from other nodes' free-lists…

Implicit garbage collection: Nice, but

- Expensive
- Not in every programming language (e.g., C++)
- Rarely (never?) lock- or wait-free

Recycling problem (example):

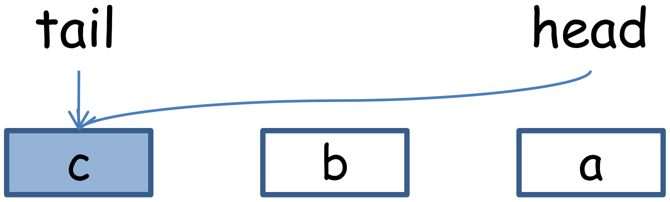
1. Thread A: reads head (==a) and head.next (==b), falls asleep just before CAS
2. Thread B: deq (b), CAS succeeds
3. Thread C: deq (c), CAS succeeds



4. Nodes a and b are put in free-lists

Recycling problem (example):

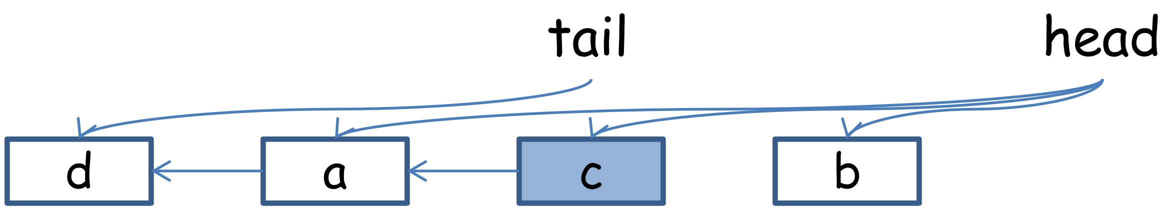
1. Thread A: reads head (==a) and head.next (==b), falls asleep just before CAS
2. Thread B: deq (b), CAS succeeds
3. Thread C: deq (c), CAS succeeds



4. Nodes a and b are put in free-lists

Recycling problem (example):

5. Node a is recycled and enqueued again, a new node d is enqueued
6. Node c is dequeued



7. Thread A wakes up: CAS on head (again a) succeeds (== a, but obsolete) sets head to head.next (==b)

Head now refers to a node not in the queue, but in the free-list of thread B (or somewhere else): Disaster!

The ABA problem:

- A value changes from a to b and back to a
- A CAS operations expects a, succeeds, although the a may no longer refer to the same entity (different semantics, different content)

Often a problem with compare and swap (CAS): Only a reference/pointer is compared, not what is referenced/pointed to

Put differently:
CAS is used to ensure that state (of the data structure) has not changed, but state is represented by a single word, and this word may have been changed back (A->B->A) even though state has changed

Note:
LL/SC (load-linked/store conditional) atomics does not have this problem (and may therefore be better for some algorithms).

SC fails if there has been a change on the address since LL.

But, LL/SC is not so commonly supported as CAS (exception: IBM Power; Risc-V)

Remedies:
- Multi-word CAS
- Transactional memory

Solution (hack) with time stamps

- Encode a time-stamp with the reference/pointer, steal some bits
- Increment time-stamp with every reference

Obvious problem:
What if there is not enough bits to steal from the pointer?

Need multi-word compare-and-swap (n-CAS, k-CAS)

Java: `AtomicStampedReference<T>` class

```
public boolean compareAndSet(T expectedReference,
                             T newReference,
                             int expectedStamp,
                             int newStamp)
public T get(int stamp[]);
```

C++:
Implementation by hand, steal upper bits from 64-bit pointer

x86 architecture:
64-bit addresses, currently only lower 48 bits used; 16 bits available for marks

```
public void deq() {
  int[] = lastX new int[];  // time stamps
  int[] = firstX new int[];
  int[] = nextX new int[];
  while (true) {
    Node first = head.get(firstX);
    Node last  = tail.get(lastX);
    Node next  = first.next.get(nextX);
    if (first==last) {
      if (next==null) throw new EmptyException();
      tail.compareAndSet(last,next,
                         lastX[0],lastX[0]+1);
    } else {
      T x = next.value;
      if (head.compareAndSet(first,next,
                             firstX[0],firstX[0]+1))
        return x;
    }
  }
}
```

Herlihy-Shavit book adopted the lock-free queue from

Maged M. Michael, Michael L. Scott: Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. PODC 1996: 267-275

Lock-free CAS (compare and swap) algorithms

CAS loop:

1. CAS to check that pointer (state) is as expected, update atomically
2. Retry on CAS failure

Lock-free if CAS failure means that other, competing thread has updated correctly and finished loop

On high contention, progress may be limited to a single/a few threads.
Possible remedy: Use atomic operation that always succeeds: FAA (fetch-and-add)

Adam Morrison, Yehuda Afek: Fast concurrent queues for x86 processors. PPOPP 2013: 103-112

Some work delegated to FAA, but also uses a double-word CAS (available on x86)

Chaoran Yang, John M. Mellor-Crummey: A wait-free queue as fast as fetch-and-add. PPOPP 2016: 16:1-16:13

Work-stealing dequeue: A special-purpose lock-free queue

Applications: Load-balancing by work-stealing (later), free-lists.

There is a collection of independent tasks to be executed:

- Each thread maintains own queue of tasks
- New tasks generated by thread are put at the bottom of queue
- When thread needs task, it is taken from the bottom
- If thread local queue empty, steal task from top of other queue

Asymmetric data structure:
Local access to bottom only, frequent; less frequent concurrent steal access to top; steal may spuriously fail (return null even if an item was present)

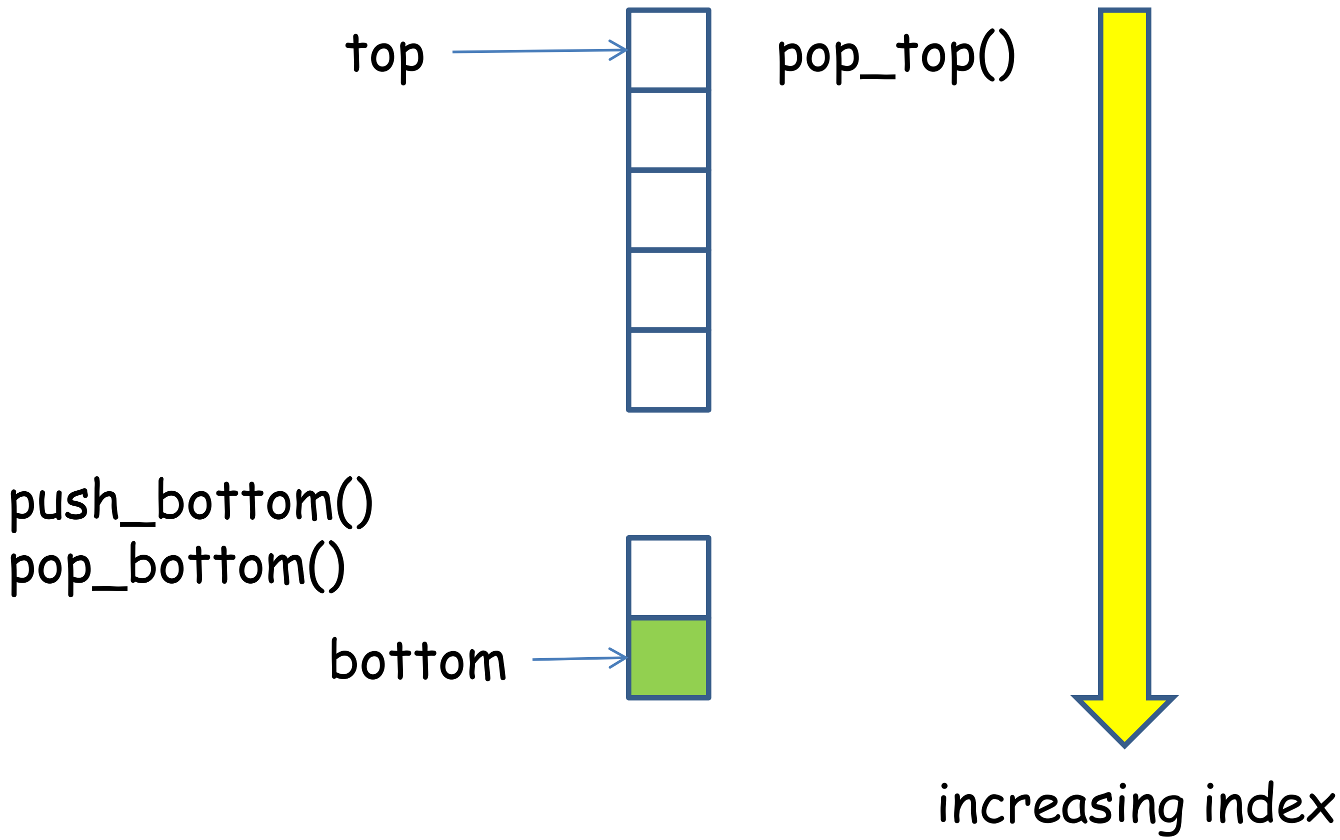
Operations:

push_bottom(x): Add element to bottom, only one thread (owner)
y = pop_bottom(): Remove element from bottom, only one thread
z = pop_top(): Remove element from top, many threads (thieves)

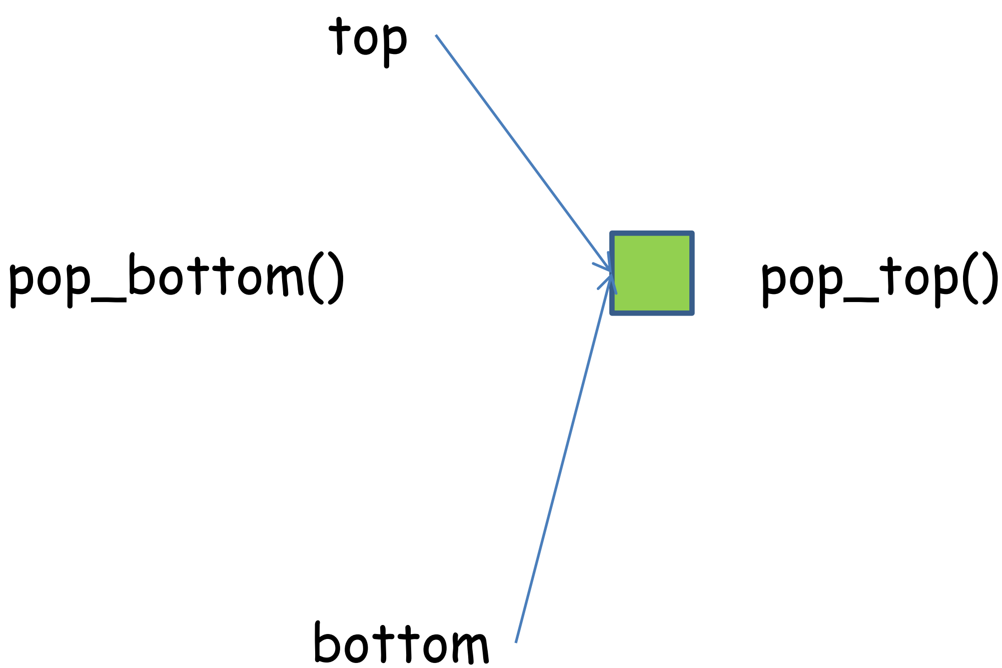
Further observations:

- Work-stealing queue is typically well-filled, no conflict between local and steal accesses. No atomics needed for local access, CAS for steal access
- One element remaining: both local access and steal must CAS

Unbounded array implementation



Unbounded array implementation



Lock-free, unbounded array implementation (in pseudo-C++)

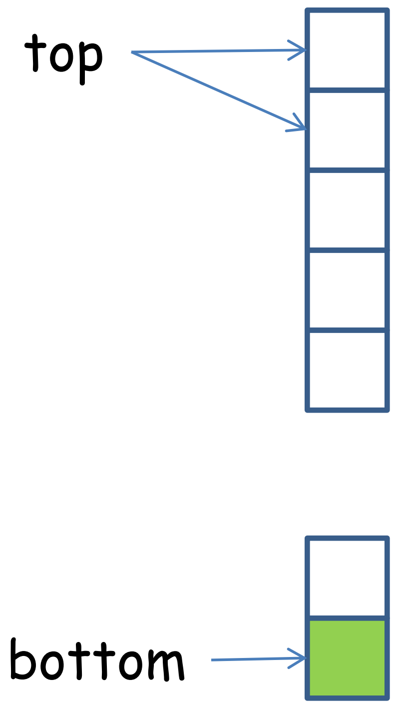
`atomic int bottom`: next free slot in array from bottom
`atomic int top`: current top element in array (only increasing)

Initially, `bottom = 0; top = 0;`

Dequeue is empty when `top>=bottom`

```
void push_bottom(T item) {
  data[bottom++] = item;
}

T pop_bottom() {
  if (bottom==top) return null_value<T>;

  T value = data[--bottom];
  int t = top;
  if (bottom>t) return value;

  if (bottom==t) {
    if (compare_exchange(top,t,t) return value;
    bottom++; // restore
  } else bottom = t;
  return null_value<T>;
}
```

```
T pop_top() {
  int t = top;
  if (bottom<=t) return null_value<T>;
  T value = data[t];

  if (compare_exchange(top,t,t+1) return value;
  else return null_value<T>;
}
```



B: pop_top()

1. Reads top
2. Reads data

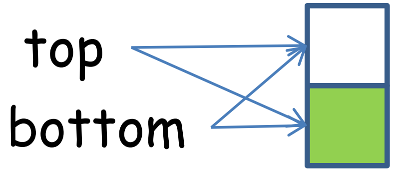
3. CAS succeeds
4. Return value

A: pop_top()

1. Reads top
2. Reads data

3. CAS fails
4. Return null

Spurious fail: The dequeue has plenty of elements



B: pop_bottom()

1. Reads data
2. Reads top

3. CAS fails
4. Restore bottom
5. Return null

A: pop_top()

1. Reads top
2. Reads data

3. CAS succeeds
4. Return value

B: pop_bottom()

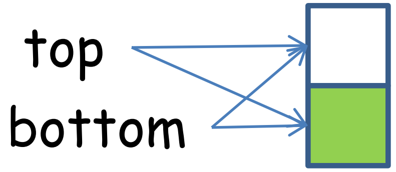
1. Reads data
2. Reads top

3. CAS succeeds
4. Return value

A: pop_top()

1. Reads top
2. Reads data

3. CAS succeeds
4. Return value



Wrong:
ABA problem! top has been updated by CAS (because bottom changed), but value has not changed

Properties:

- Suffers from ABA problem. Standard repair: Time stamps

- Local pop_bottom(): Decrement bottom, read top; when only one element remaining, CAS on top – fails if other thread modified top, but that means queue empty
- Steal pop_top(): Read top, read bottom, CAS on top. Only one thread may succeed before a decrement to bottom becomes visible

Linearization:

- push_bottom(): Increment of bottom
- pop_bottom():
  - Decrement of bottom, if top<bottom
  - CAS, if top==bottom
- pop_top(): CAS on top

Progress and liveness:

- push_bottom(): Wait-free
- pop_bottom():
  - wait-free, if top<bottom
  - if top==bottom: wait-free whether CAS fails or succeeds
- pop_top(): Wait-free, whether CAS fails or succeeds, but may fail spuriously (stealing therefore becomes lock-free)

Drawbacks:

1. Unbounded array!
2. Ignores ABA problem
3. (pop_top() may fail spuriously)

Nimar S. Arora, Robert D. Blumofe, C. Greg Plaxton: Thread Scheduling for Multiprogrammed Multiprocessors. Theory Comput. Syst. 34(2): 115-144 (2001)

Drawbacks. Fixes

1. Use array of size N, index with (bottom%N) and (top%N), when full either do not push_bottom(), or extend array. Recall array lock
2. Bounded time stamp, or hack
3. Not a problem for work-stealing context, could also try pop_top() some fixed number of times before selecting new victim

Nimar S. Arora, Robert D. Blumofe, C. Greg Plaxton: Thread Scheduling for Multiprogrammed Multiprocessors. Theory Comput. Syst. 34(2): 115-144 (2001)

David Chase, Yossi Lev: Dynamic circular work-stealing deque. SPAA 2005: 21-28

Bit stealing hack: Steal k (=8, =16?) bits from address

- Increment "time-stamp" on pop_bottom()
- Reset "time stamp" on pop_top()

```
T pop_bottom() {
  if (bottom==index_bits(top)) return null_value<T>;

  T value = data[--bottom];
  if (bottom>index_bits(top)) return value;

  int t = top;
  int new_t = inc_mark_bits(t);
  if (bottom==index_bits(t)) {
    if (compare_exchange(top,t,new_t) return value;
    bottom++;
  } else bottom = index_bits(t);
  return null_value<T>;
}
```

```
T pop_top() {
  int t = top;
  if (bottom<=index_bits(top)) return null_value<T>;
  T value = data[t];

  int new_t = index_bits(t); // reset “time stamp”
  if (compare_exchange(top,t,new_t+1) return value;
  else return null_value<T>;
}
```

```
#define index_bits(a) (a & 0xFFFFFF)
#define inc_mark_bits(a) \
((((a>>48)+1) & 0xFF)<<48) | index_bits(a))
```

Wait-free queues (lists, stacks)

Some form of help needed to make list and queues (and stacks) wait-free (instead of only lock-free).

Alex Kogan, Erez Petrank: Wait-free queues with multiple enqueuers and dequeuers. PPOPP 2011: 223-234
Chaoran Yang, John Mellor-Crummey: A wait-free queue as fast as fetch-and-add. PPOPP 2016: 16:1-16:13
Shahar Timnat, Anastasia Braginsky, Alex Kogan, Erez Petrank: Wait-Free Linked-Lists. OPODIS 2012: 330-344

Seep Goel, Pooja Aggarwal, Smruti R. Sarangi: A Wait-Free Stack. ICDCIT 2016: 43-55

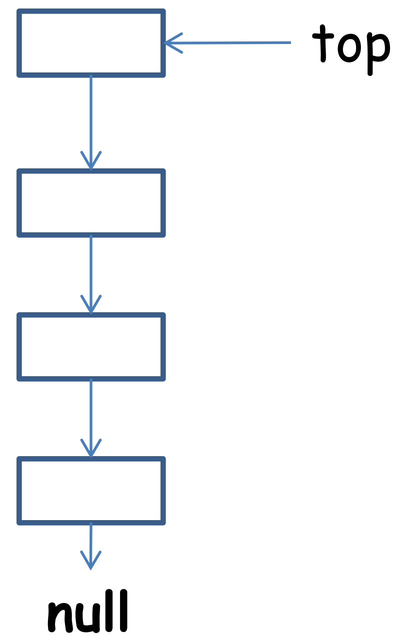
Concurrent LIFO stacks (Chap. 11)

Stack is a pool type with last-in, first-out (LIFO) fairness semantics

Operations:

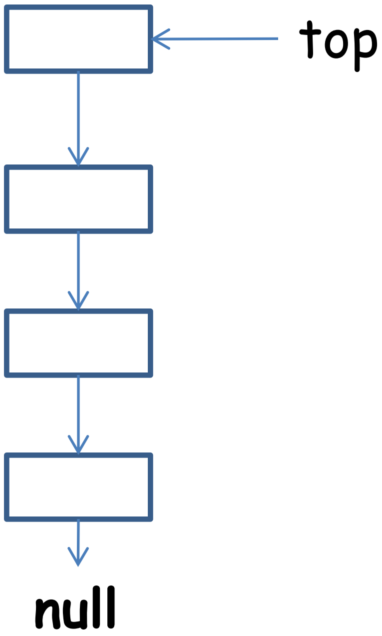
- push(x)
- x = pop()

The item returned by a pop() is the most recently pushed item



Implementation: Linked list with top reference/pointer

```
class Node {
  public T value;
  public Node next;
  public Node (T x) {
    value = x;
    next = null;
  }
}
```



Implementation:

- push(x): create new stack node, try to push with CAS. If CAS fails, there is contention on top, back off and try again
- y = pop(): remove top node with CAS. If CAS fails, there is contention on top, back off and try again

```
class LockFreeStack {
  AtomicReference<Node> top = new …
  static final int MIN_DELAY, MAX_DELAY;
  Backoff backoff =
    new Backoff(MIN_DELAY,MAX_DELAY);

  public void push(T x);

  public T pop();
}
```

Values for Backoff class, recall lock lecture

```
protected boolean try_push(Node node) {
  Node oldtop = top.get();
  node.next = oldtop;
  return top.compareAndSet(oldtop,node);
}

public void push(T x) {
  Node node = new Node(x);
  while (true) {
    if (try_push(node)) return;
    else backoff.backoff();
  }
}
```

```
protected boolean try_pop()
  throws EmptyException {
  Node oldtop = top.get();
  if (oldtop==null) throw new EmptyException();
  Node newtop = oldtop.next;
  if (top.compareAndSet(oldtop,newtop))
    return oldtop;
  else return null;
}

public T pop() throws EmptyException {
  while (true) {
    Node node = try_pop();
    if (node!=null) return node.value;
    else backoff.backoff();
  }
}
```

Properties of lock-free stack

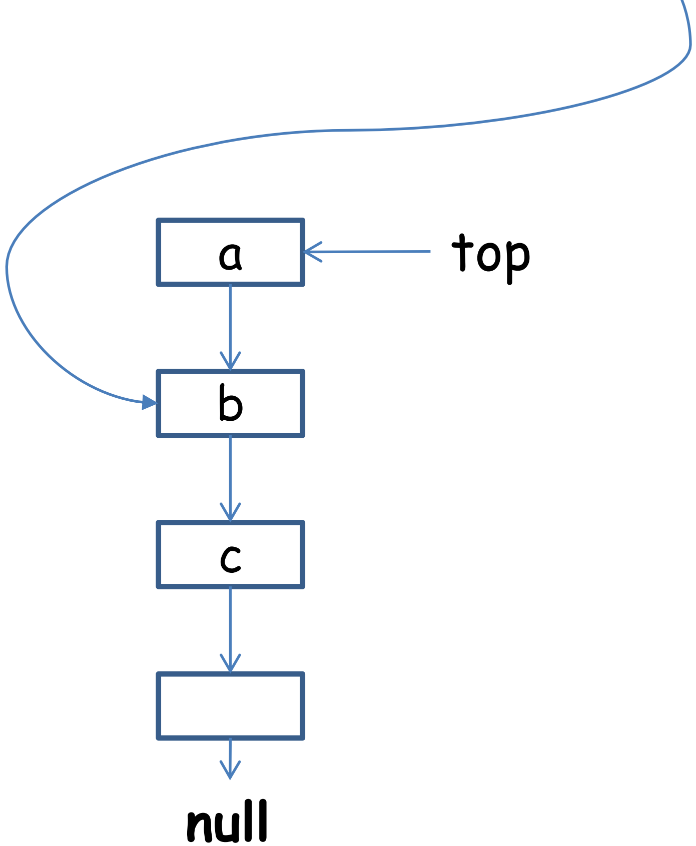
- Linearizable
  - push: successful CAS on top
  - pop: successful CAS on top, or read of null top-reference
- Lock-free, but not starvation free (if CAS fails, some other thread must have succeeded)

The Java implementation, relying on garbage collection does not have an ABA problem:
The push/pop interface passes values only (items), nodes are maintained (allocated) internally. The same address thus cannot reappear at one thread when some other threads references this address

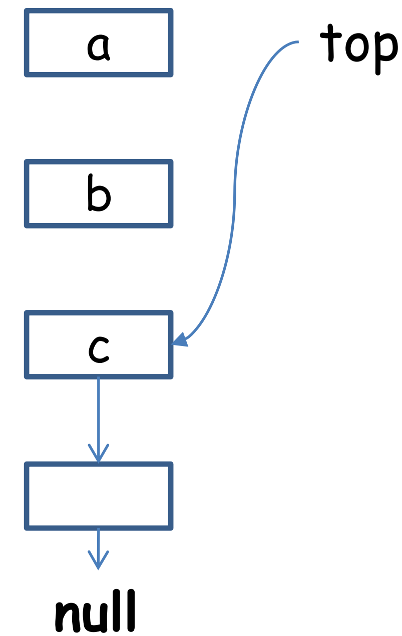
If pop'ed items can be reused, the stack has an ABA problem

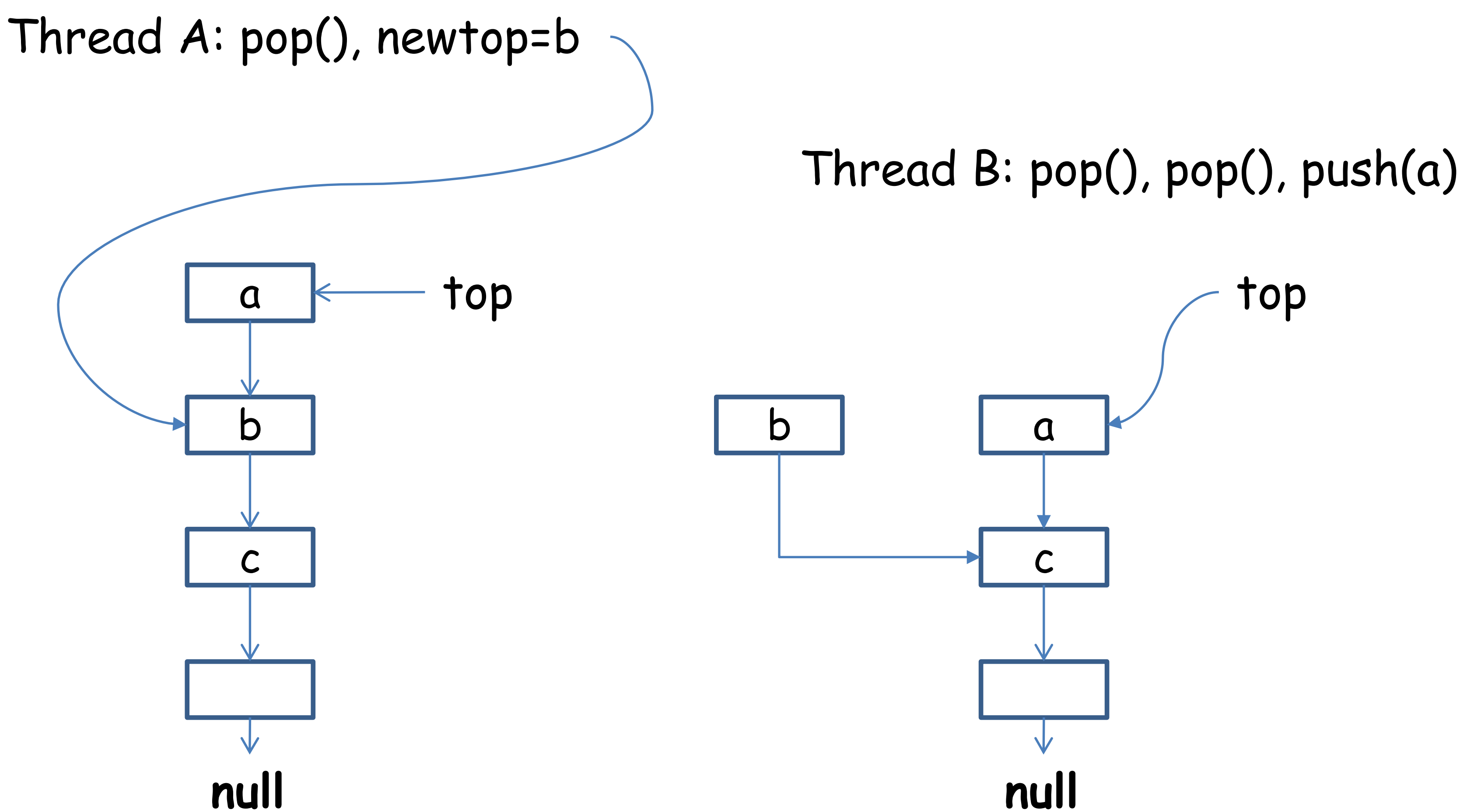
Thread A: pop(), newtop=b

Thread A delayed



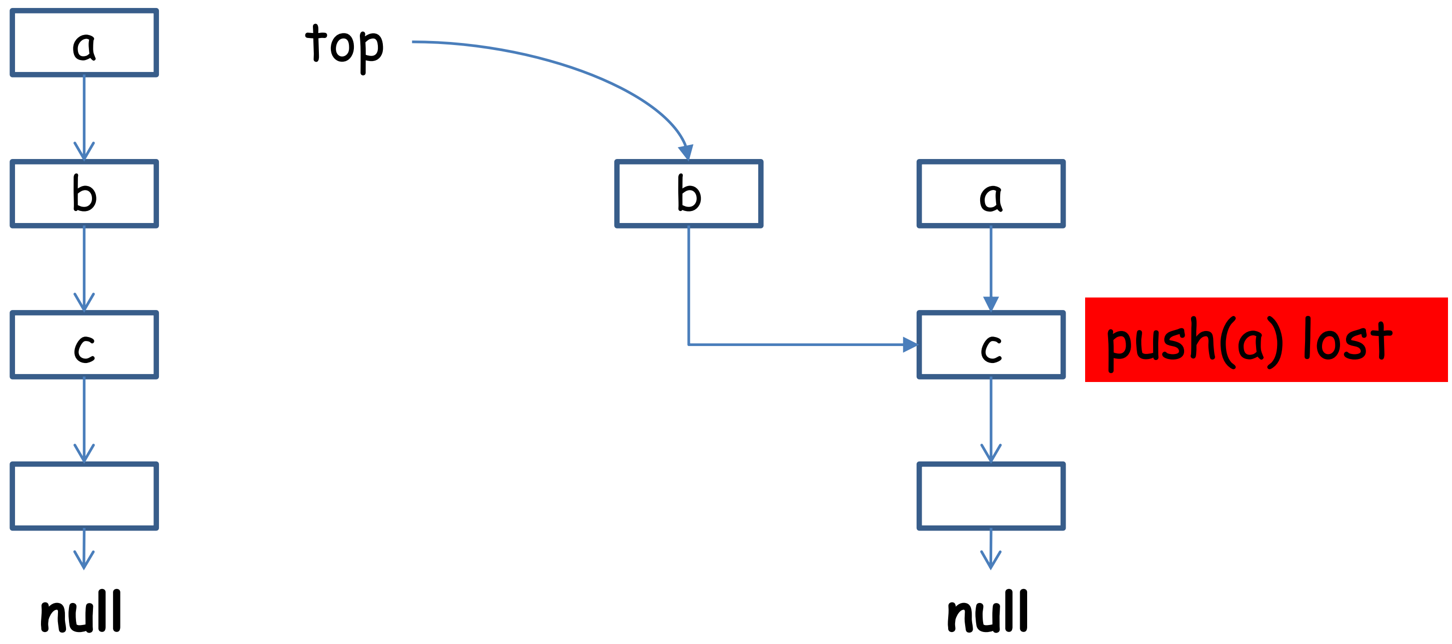
Thread B: pop(), pop()




Thread A: pop(), newtop=b
Thread B: pop(), pop(), push(a)
top
a
b
c
null
top
b
a
c
null

Thread A: pop(): newtop=b, CAS succeeds

Thread B: pop(), pop(), push(a)



Repair (in C): Claim some bits for time stamp

```
void *pop() {
  void *oldtop;
  do {
    oldtop = top;
    if (address_bits(oldtop)==NULL) return NULL;
    newtop = oldtop->next;
    set_mark_bits(newtop,get_mark_bits(oldtop)+1);
  while (!compare_exchange(top,oldtop,newtop);
  return oldtop;
}
```

Observation:
It suffices to increment the timestamp only on pop() (or on push())

Repair (in C): Claim some bits for time stamp

```
void push(void *newtop) {
  void *oldtop;
  do {
    oldtop = top;
    newtop->next = address_bits(oldtop);
    set_mark_bits(newtop,get_mark_bits(oldtop));
  while (!compare_exchange(top,oldtop,newtop);
  return oldtop;
}
```

R. Kent Treiber: Systems Programming: Coping with Parallelism. Technical Report 5118, IBM Almaden Research Center, 1986

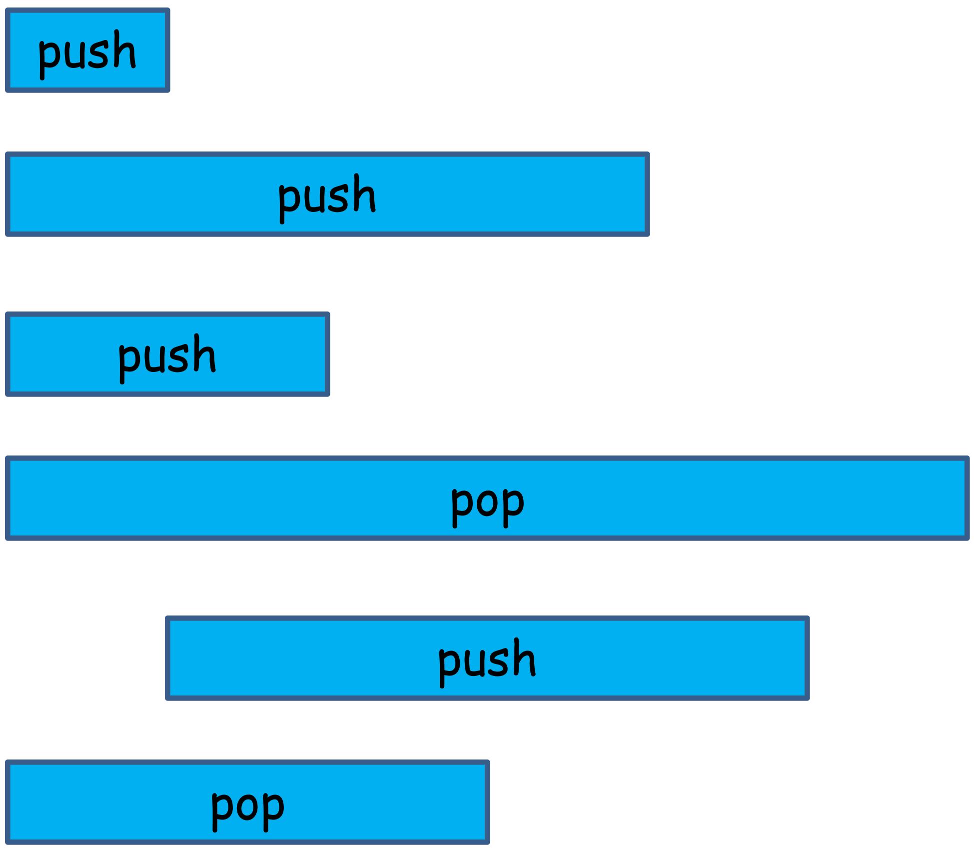
Problem with lock-free stack (and queue)

Access to top reference/pointer is a sequential bottleneck (even worse than the double-ended queue), all threads compete for the top of the stack, linearization enforces serialization

Possible to improve?

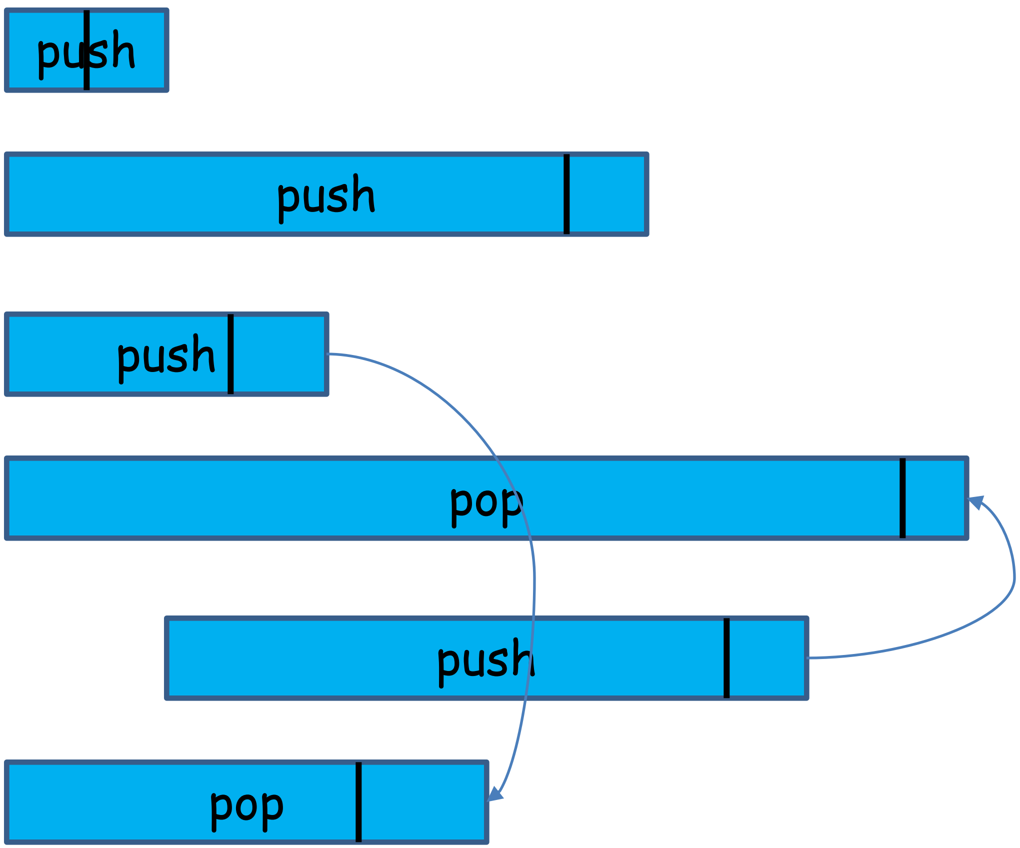
Highly sensitive to back off tuning

Stack linearization(*)

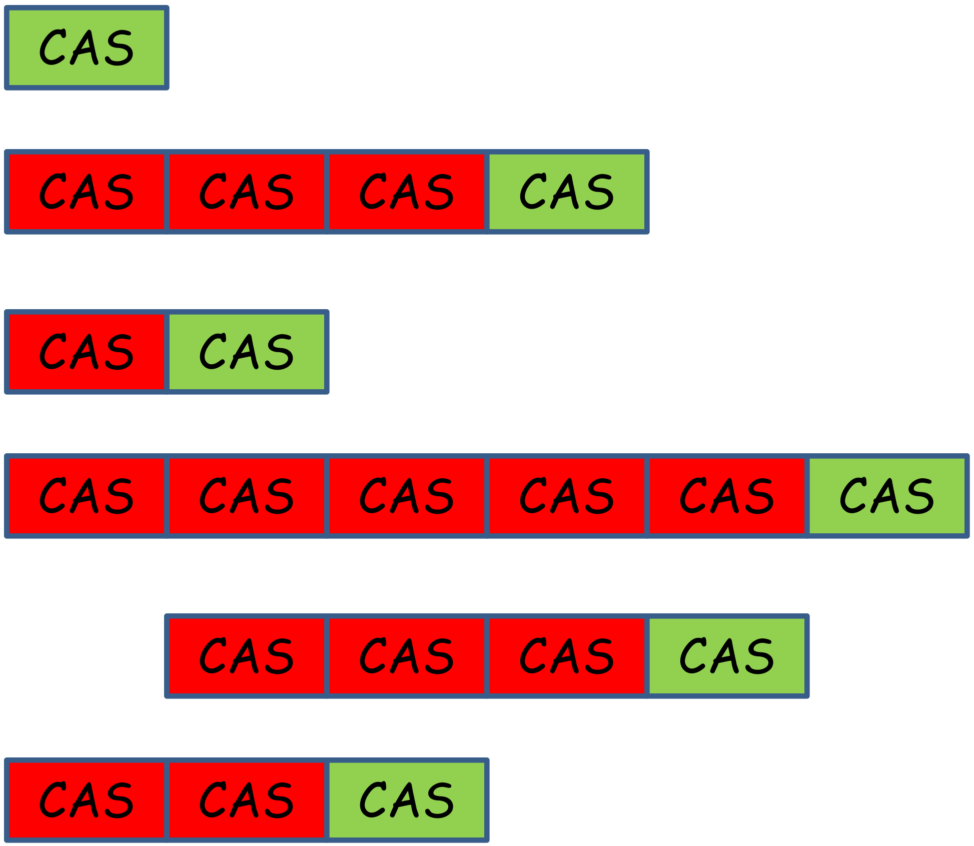


(*) Example due to Martin Wimmer

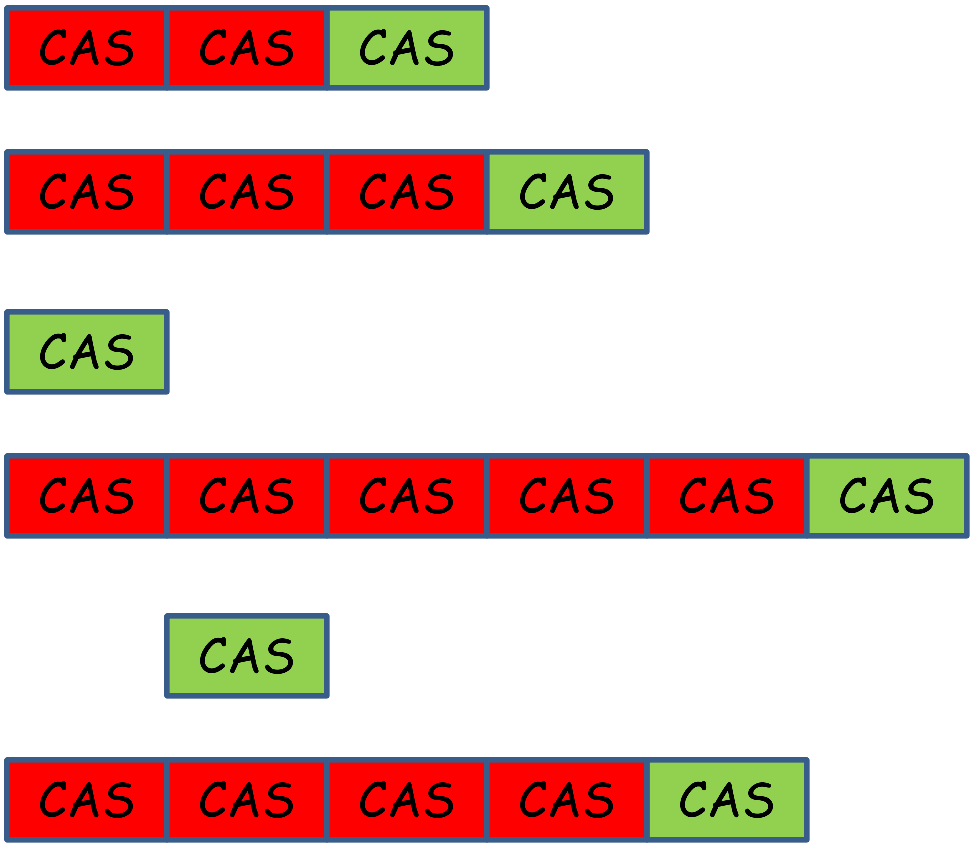
Stack linearization



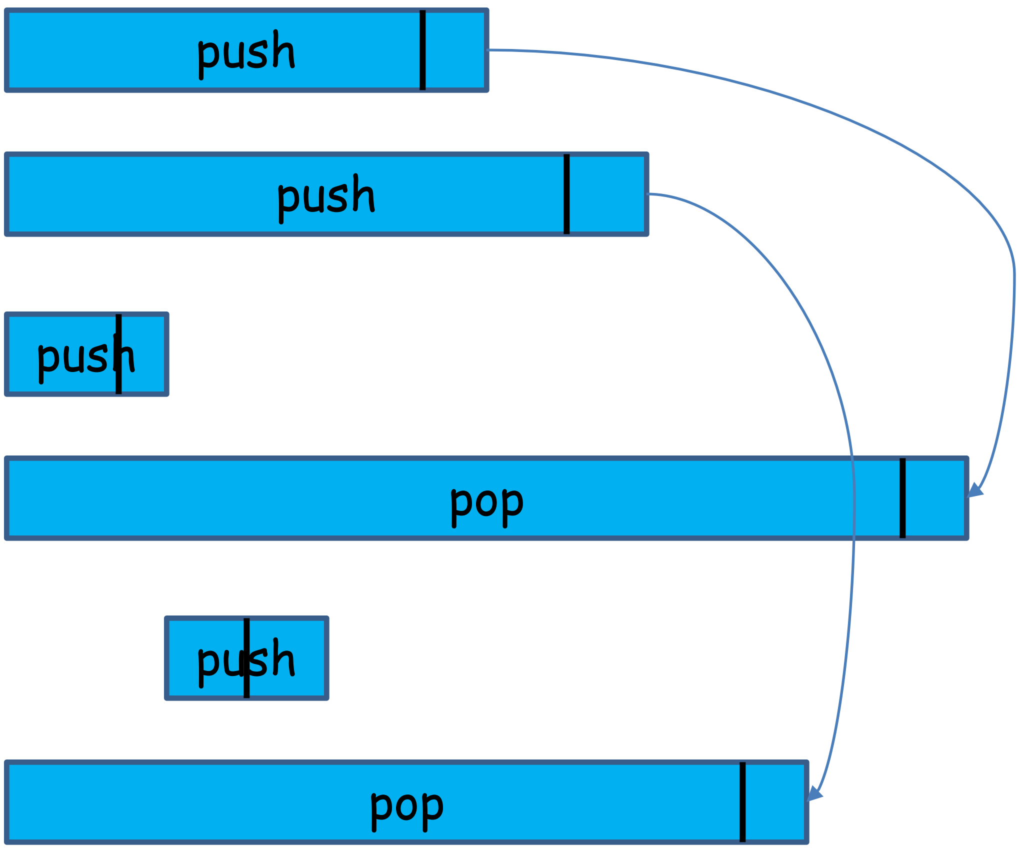
Stack linearization: Could have happened like this



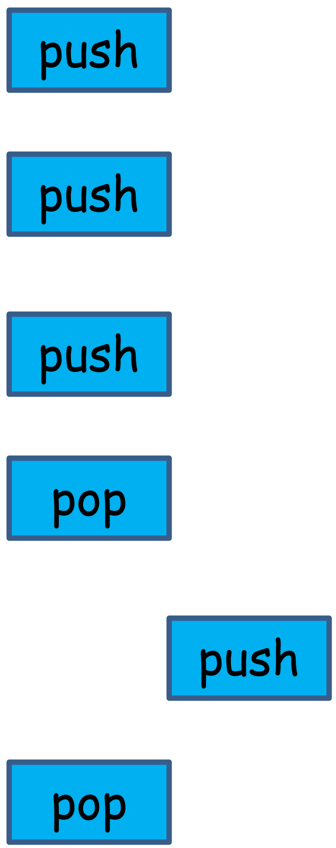
Stack linearization: But also differently…



Stack linearization: …with these real-time operations

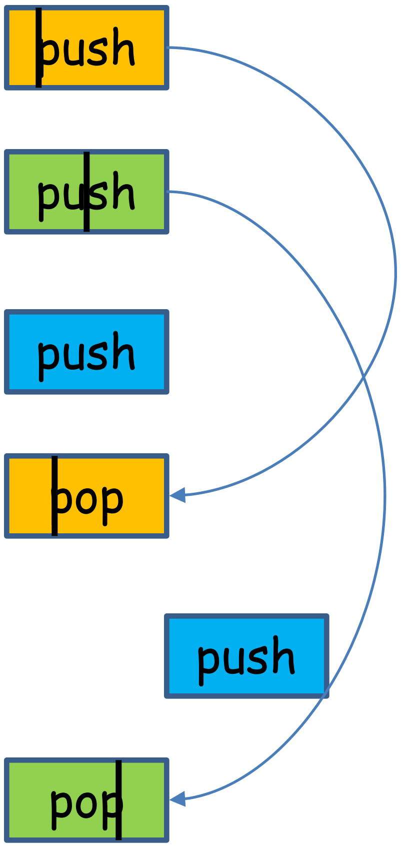


Can this history be linearized?



Yes: Two concurrent push and pop operations can be paired

Can this history be linearized?



Yes: Two concurrent push and pop operations can be paired

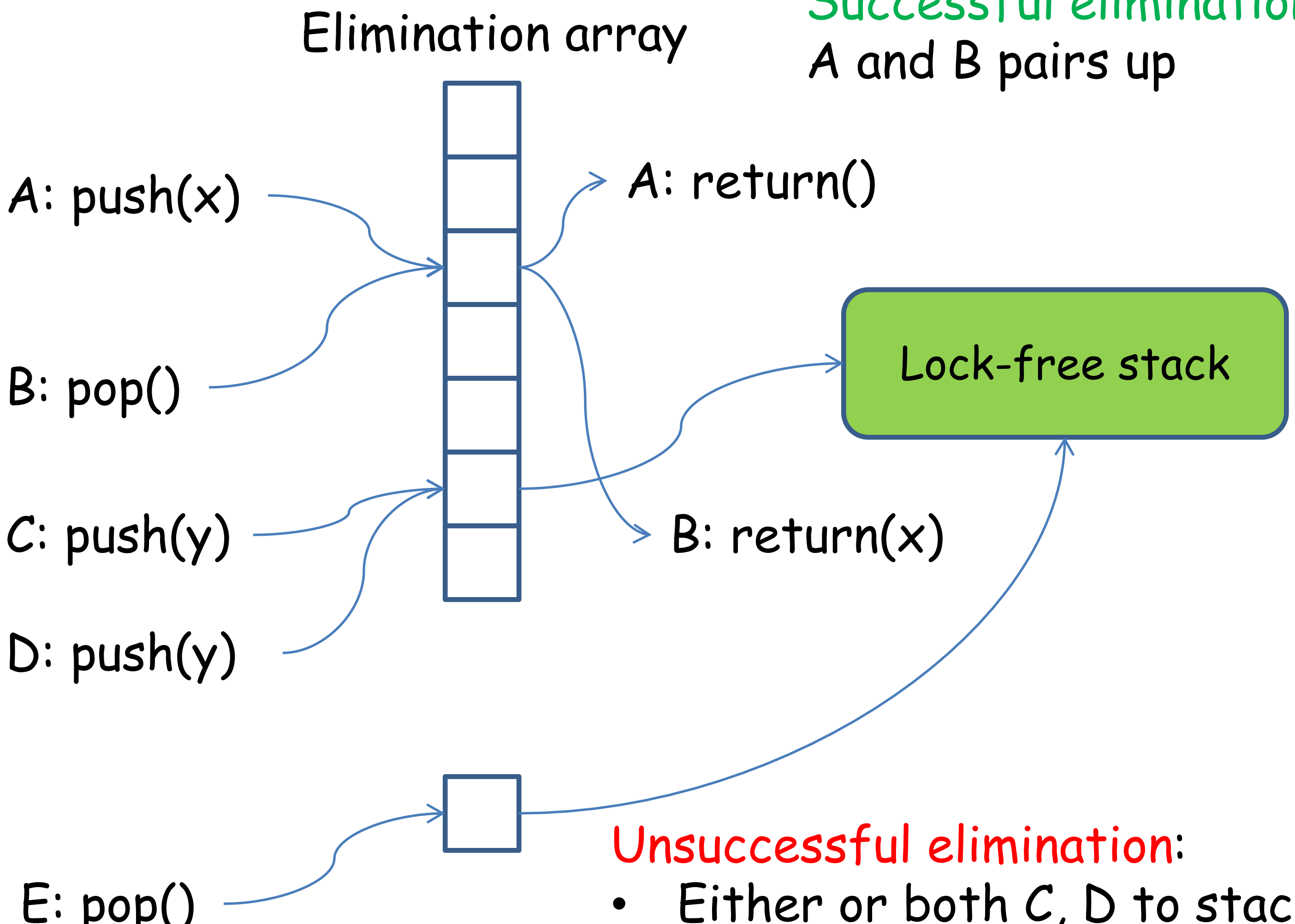
New idea: Elimination (backoff stack)

A push can be cancelled by an overlapping pop; this pair of operations do not have to access the stack at all, and are said to eliminate each other

Use elimination array (size: tuning parameter):

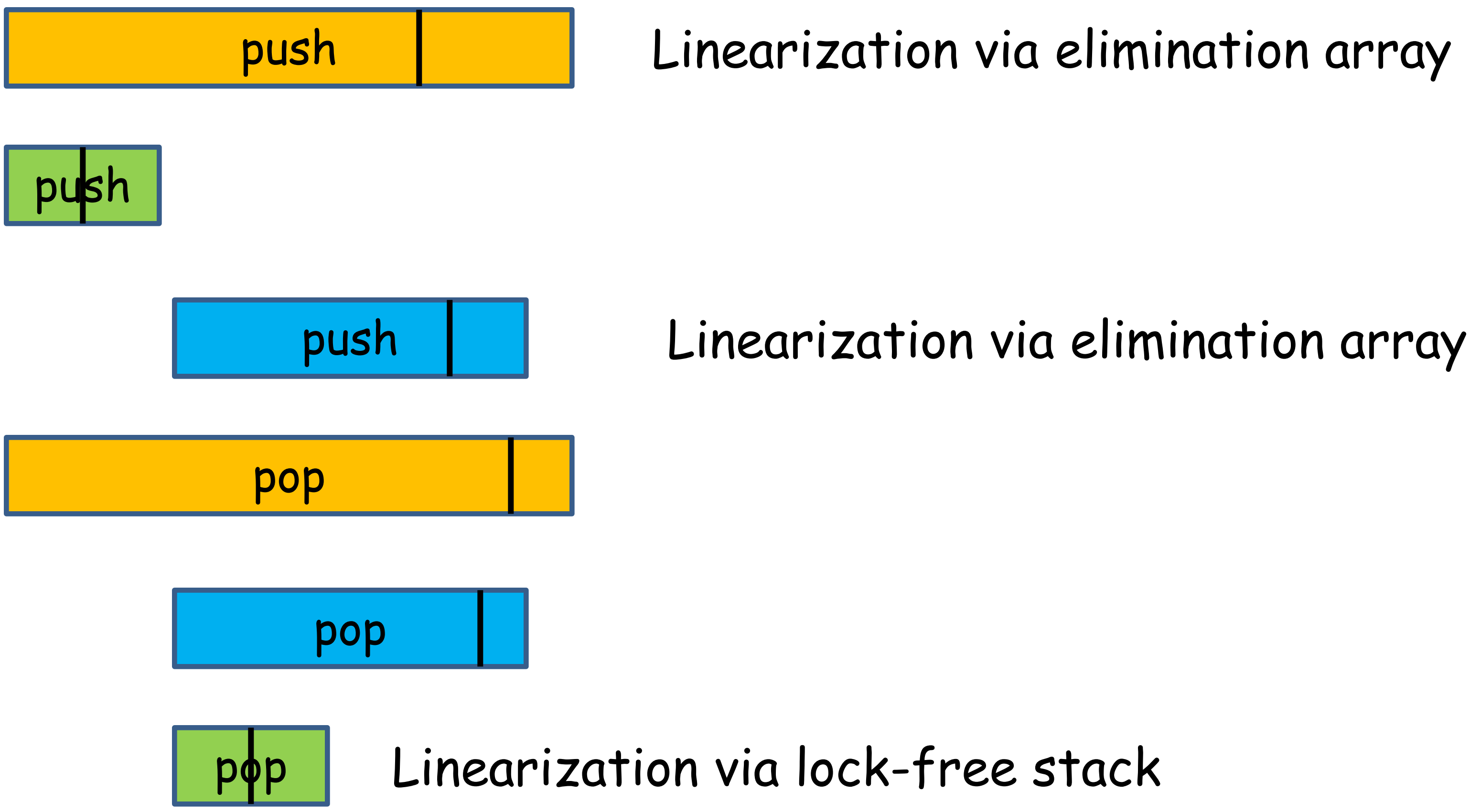
- push(x): choose random index, put x into index, wait for pop, try stack on time-out
- pop(): choose random index, wait for item at index, try stack on time-out

Successful elimination:
A and B pairs up

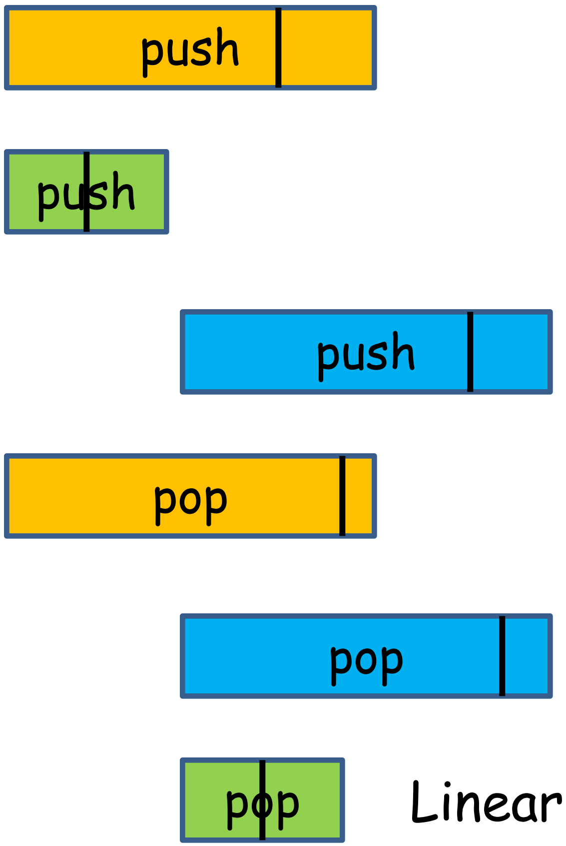


Unsuccessful elimination:
- Either or both C, D to stack
- E to stack

Linearization with elimination



Linearization with elimination (different history)



Linearization via elimination array

Linearization via elimination array

Linearization via lock-free stack

Implementation (straightforward, but tedious):

Elimination array of socalled exchangers each with a slot that allows exchange of some value between two threads using CAS. Then:

- Try stack first
- If unsuccessful, try elimination
- Backoff/time-out
- Repeat until success

Highly sensitive to tuning!

- Size of the elimination array (too large: no elimination; too small: to many conflicts)
- Back-off
- …

```
class LockFreeExchanger<T> {
  static final int EMPTY=…, WAITING=…, BUSY=…;
  AtomicStampedReference<T> slot =
    new AtomicStampedReference<T>(null,EMPTY);

  public exchange(T item, long timeout, TimeUnit unit)
  throws Timeoutexception {
    long timebound =
      System.nanotime()+unit.toNanos(timeout)
    int[] stamp = {EMPTY};

    while (true) {
      // try to exchange until timeout
      … (next slide)
    }
  }
}
```

```
// try to exchange until timeout
if (System.nanoTime()>timebound)
  throw new Timeoutexception();
  // timed out, no exchange
… (next slide)
```

Thread tries to grab slot with CAS. Three cases:

1. Slot is EMPTY, no one to exchange with, thread becomes WAITING until either timeout (change back to EMPTY) or someone shows up
2. Someone else is WAITING, change to BUSY and let other thread exchange
3. Already BUSY, slot cannot be used

```
T otheritem = slot.get(stamp); // read the slot
switch (stamp[0]) {
case EMPTY:
  if (slot.compareAndSet(otheritem,item,
                          EMPTY,WAITING)) {
    while (System.nanoTime()<timebound) {
      otheritem = slot.get(stamp);
      if (stamp[0]==BUSY) {
        slot.set(null,EMPTY);
        return otheritem;
      }
    }
  }
  if (slot.compareAndSet(item,null,WAITING,EMPTY))
    throw new TimeoutException();
  else {
    otheritem = slot.get(stamp);
    slot.set(null,EMPTY);
    return otheritem;
  }
```

```
case EMPTY:
  … (previous slide)
break;
case WAITING:
  if (slot.compareAndSet(otheritem,item,
                         WAITING,BUSY))
    return otheritem;
  break;
case BUSY:
  break;
default: // cannot be
}
```

```
class EliminationArray<T> {
  static final int duration=…; // tuning parameter
  LockFreeExchanger<T>[] exchanger;
  Random random;
  public EliminationArray(int capacity) {
    exchanger = (LockFreeExchanger<T>[])
      new LockFreeExchanger<T>();
    for (int i=0; i<capacity; i++)
      new LockFreeExchanger<T>();
    random = new Random();
  }
  public T visit(T value, int range)
  throws TimeoutException {
    int slot = random.nextInt(range);
    return (exchanger[slot].exchange(value,duration,
                                    TimeUnit.MILLISECONDS));
  }
}
```

```
public class EliminationBackoffStack<T> extends
LockFreeStack<T> {
  static final int capacity = …; //maximum capacity
  static final int range = … ; // here: fixed range
  // book provides for adaptive range selection
  EliminationArray<T> eliminationarray =
    new EliminationArray<T>(capacity);

  public void push(T value);

  public T pop();
}
```

```java
public void push(T x) {
  Node node = new Node(x);
  while (true) {
    if (try_push(node)) return;
    else try {
      T y = eliminationarray.visit(x,range);
      if (y==null) return; // successful elimination
    }
  }
}
```

On unsuccessful elimination (timeout): catch the exception, perhaps use this to adjust the range (book provides for such policy change). Also update range/policy on successful elimination

```java
public T pop() throws EmptyException {
  while (true) {
    Node node = try_pop();
    if (node!=null) return node.value;
    else try {
      T y = eliminationarray.visit(null,range);
      if (y!=null)
        return y; // successful elimination
    }
  }
}
```

On unsuccessful elimination (timeout): catch the exception, perhaps use this to adjust the range (book provides for such policy change). Also update range/policy on successful elimination

Linearization:

Any successful push()/pop() that completes via the lock-free stack, linearizes at the successful CAS in the stack

Any pair of eliminated push()/pop() operations, linearize at their elimination point. Such a pair leaves the stack unchanged, and can linearize anywhere within their concurrent invocations, pop before push (regardless of other, concurrent push() and/or pop() operations).

All push()/pop() operations complete either by successful elimination or via the lock free stack

Other uses of exchangers and elimination

William N. Scherer III, Doug Lea, Michael L. Scott: Scalable synchronous queues. Commun. ACM 52(5): 100-111 (2009) (*)

Irina Calciu, Hammurabi Mendes, Maurice Herlihy: The Adaptive Priority Queue with Elimination and Combining. DISC 2014: 406-420

Anastasia Braginsky, Nachshon Cohen, Erez Petrank: CBPQ: High Performance Lock-Free Priority Queue. Euro-Par 2016: 460-474

(*) Announced at PPoPP 2006

Are queues and stacks good, concurrent data structures?

Queues and stacks are sequential data structures… strong ordering/fairness guarantees (FIFO, LIFO, Priority, …)

Leads to serialization on concurrent updates

⇒ Little potential for parallelization, little that can be done locally by individual threads

Problem:
Linearization enforces sequential order. Is this too strict?

Relaxed linearization

Some applications do not require exact ordering:

- Element close to the top good enough (relaxed stack)
- k'th smallest element (Priority queue)
- …

As long as some bound on the deviation from a linearizable order can be given

Yehuda Afek, Guy Korland, Eitan Yanovsky: Quasi-Linearizability: Relaxed Consistency for Improved Concurrency. OPODIS 2010: 395-410
…
Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, Philippas Tsigas: The lock-free k-LSM relaxed priority queue. PPOPP 2015: 277-278

Some applications do not require exact ordering:

- Quiescent consistency gives more freedom for concurrent operations (might be easier to guarantee, see later)

But no "fairness" guarantees

Often used relaxed PQ example(s):
- SSSP a la Dijkstra is robust, an "incorrect" min element does not harm correctness, but may lead to unnecessary, duplicate, superfluous (speculative) work
- Concurrent tree search, e.g., branch-and-bound

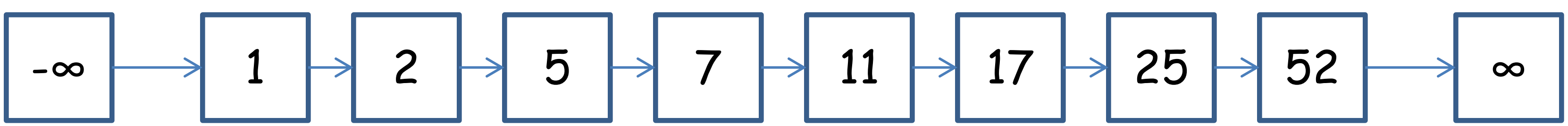
Skiplist (Chap. 14)

Efficient, randomized, list-based search structure (abstract set)

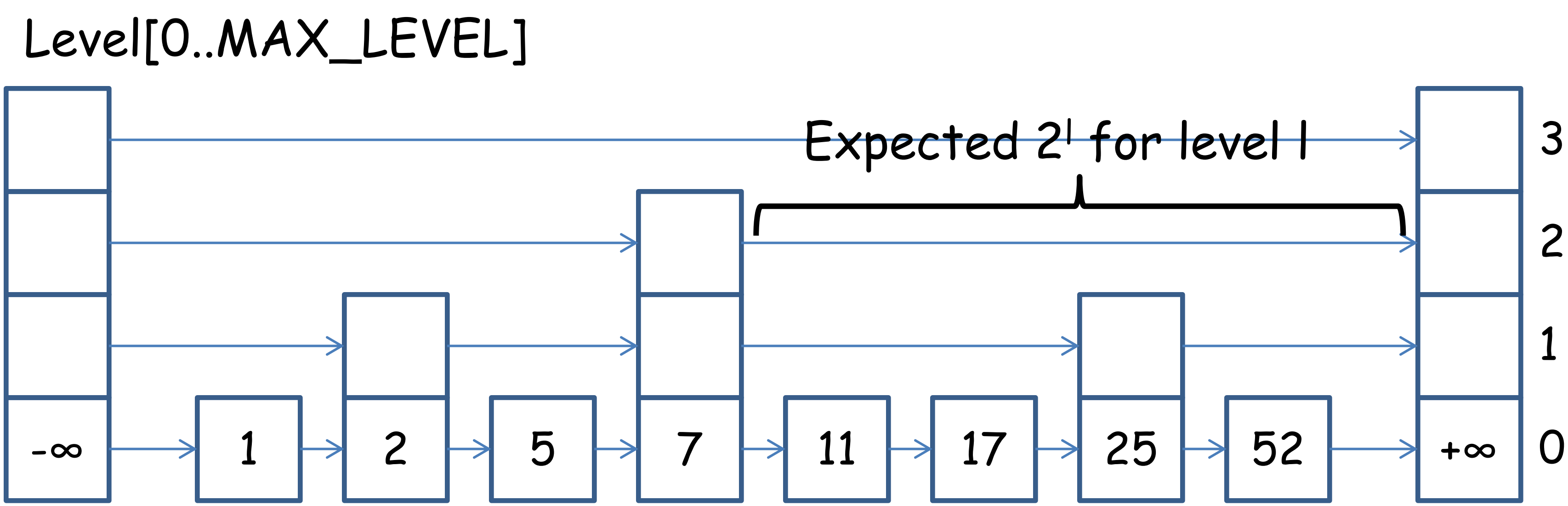
- O(log n) expected operations for n element skiplist
  - add(x),
  - y = remove()
  - contains(z)?
- O(n) space
- Compared to tree-based search structures: No need for rebalancing

William Pugh: Skip Lists: A Probabilistic Alternative to Balanced Trees. Comm. ACM 33(6): 668-676 (1990)

Ordered list (with sentinels at beginning and end), linear time



Skiplist: Multiple lists for fast navigation



Skiplist properties and invariants

- Collection of ordered lists, organized in levels 0, 1, …, MAX_LEVEL
- List at bottom level 0 represents set
- Ceiling(log n) levels for n element set
- Sublist property: List at level l, l>0, sublist of list at level l-1 (list at level l is a shortcut into the list at level l-1)
- Number of elements at level l is expected some constant fraction 1/f (often: f=2) of number of elements at level l-1
- Probability that an element appears at level l is $(1/f)^l$
- Expected space consumption is
  - $O(\sum_{0\leq l<MAX\_LEVEL} n\,(1/f)^l) = O(n)$

Why?

For the following, we do not distinguish between item and key

Skiplist operations, implementation

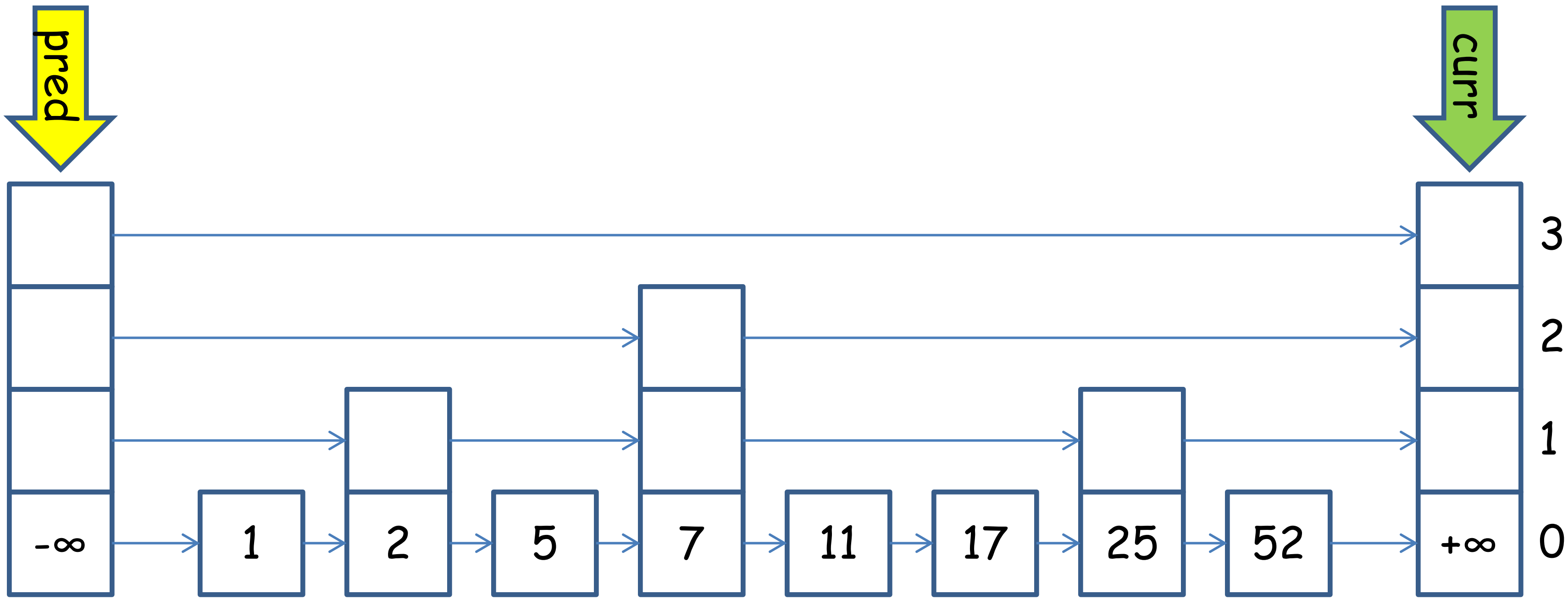
- find(x) and contains(x): Start at current maximum level l, search for elements pred and curr (== pred.next) such that pred.item<x≤curr.item, decrease l; until l==0. Found iff curr.item==x
- add(y): if not find(x), create new node with random number of levels k, insert in lists at all levels from 0 to k
- remove(z): if find(x), link out x from all lists from level 0 to level of x

All operations are O(log n), if the skiplist properties are fulfilled

Why?

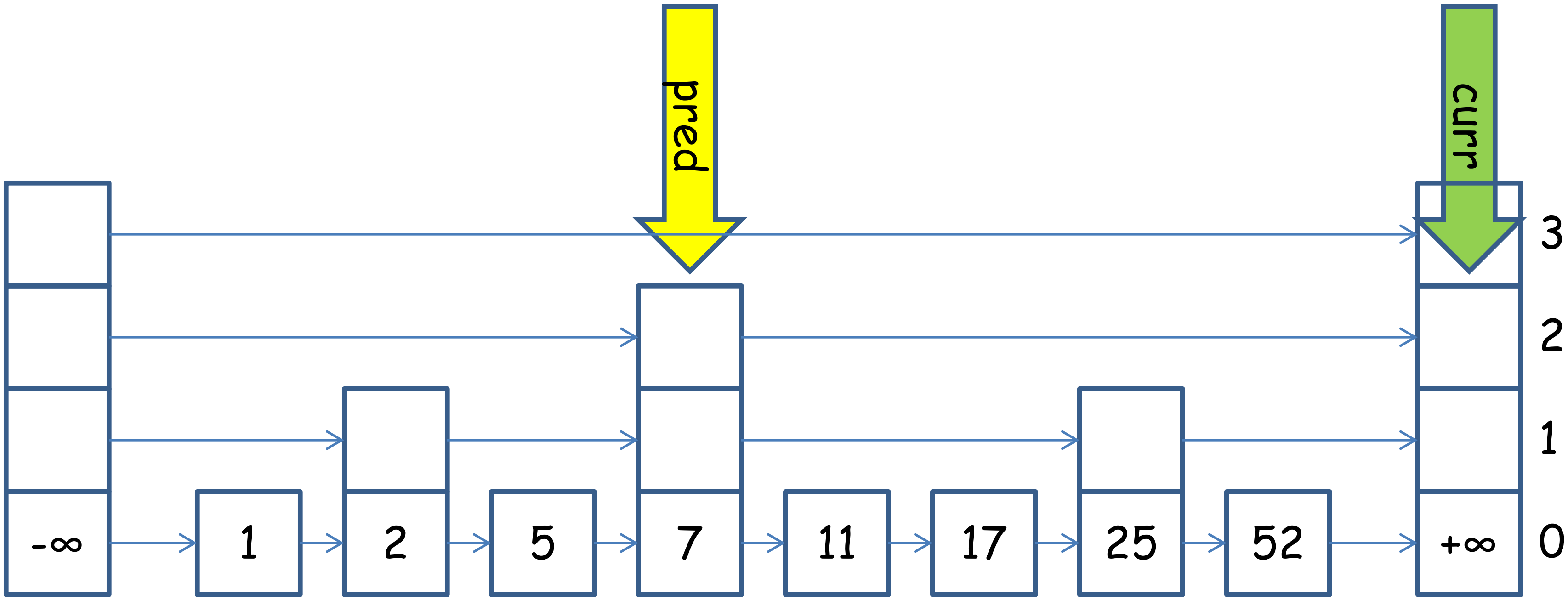
Example: find(14)

Set
preds[3] = pred; succs[3] = curr;



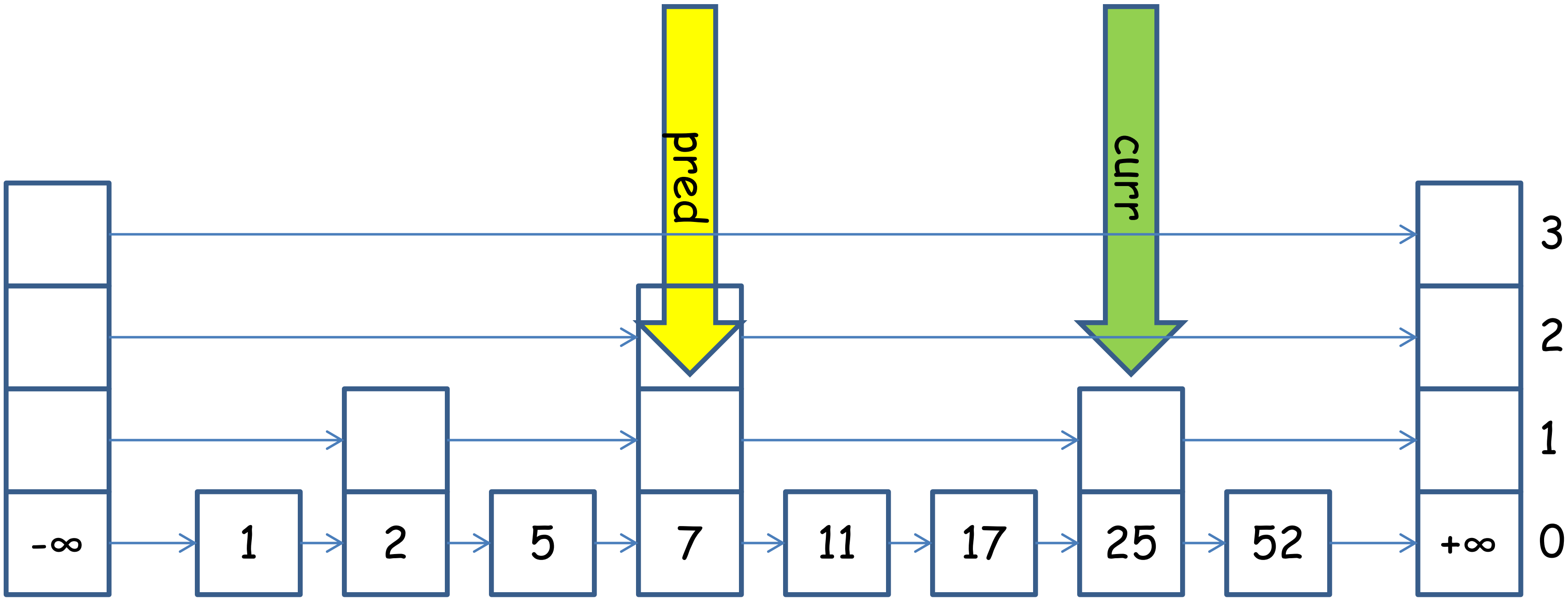
Example: find(14)

Set
preds[3] = pred; succs[3] = curr;
preds[2] = pred; succs[2] = curr;



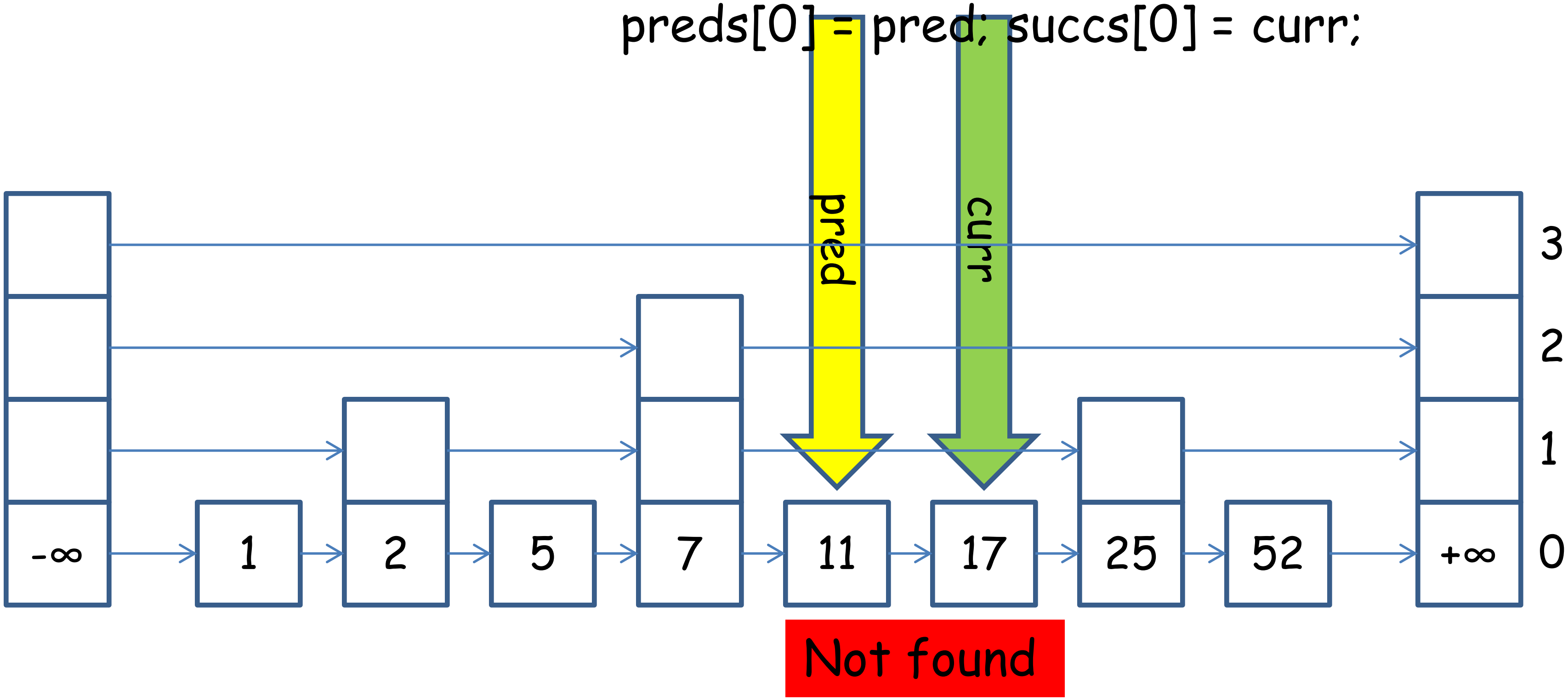
Example: find(14)

Set
preds[3] = pred; succs[3] = curr;
preds[2] = pred; succs[2] = curr;
preds[1] = pred; succs[1] = curr;



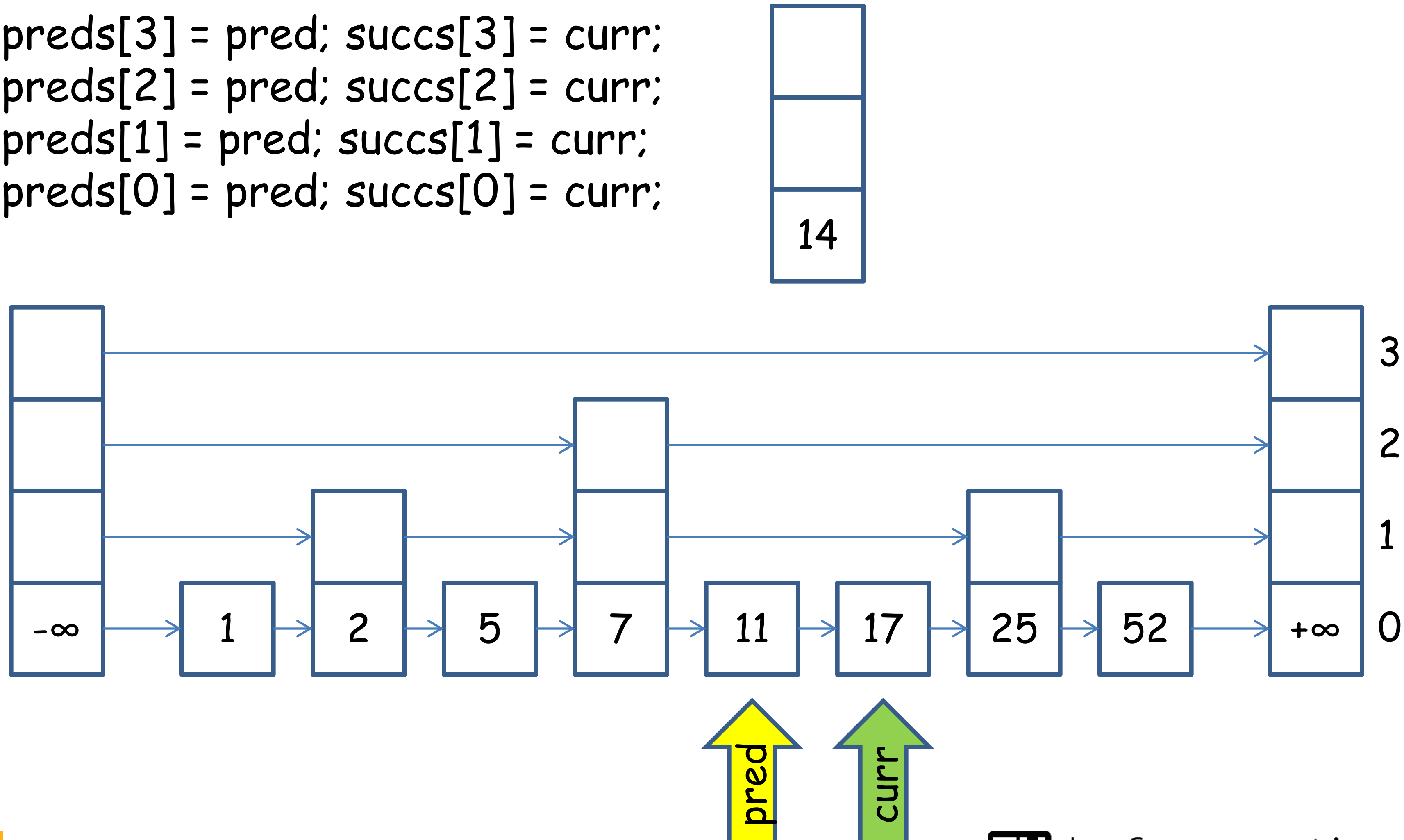
Example: find(14)

Set
preds[3] = pred; succs[3] = curr;
preds[2] = pred; succs[2] = curr;
preds[1] = pred; succs[1] = curr;
preds[0] = pred; succs[0] = curr;



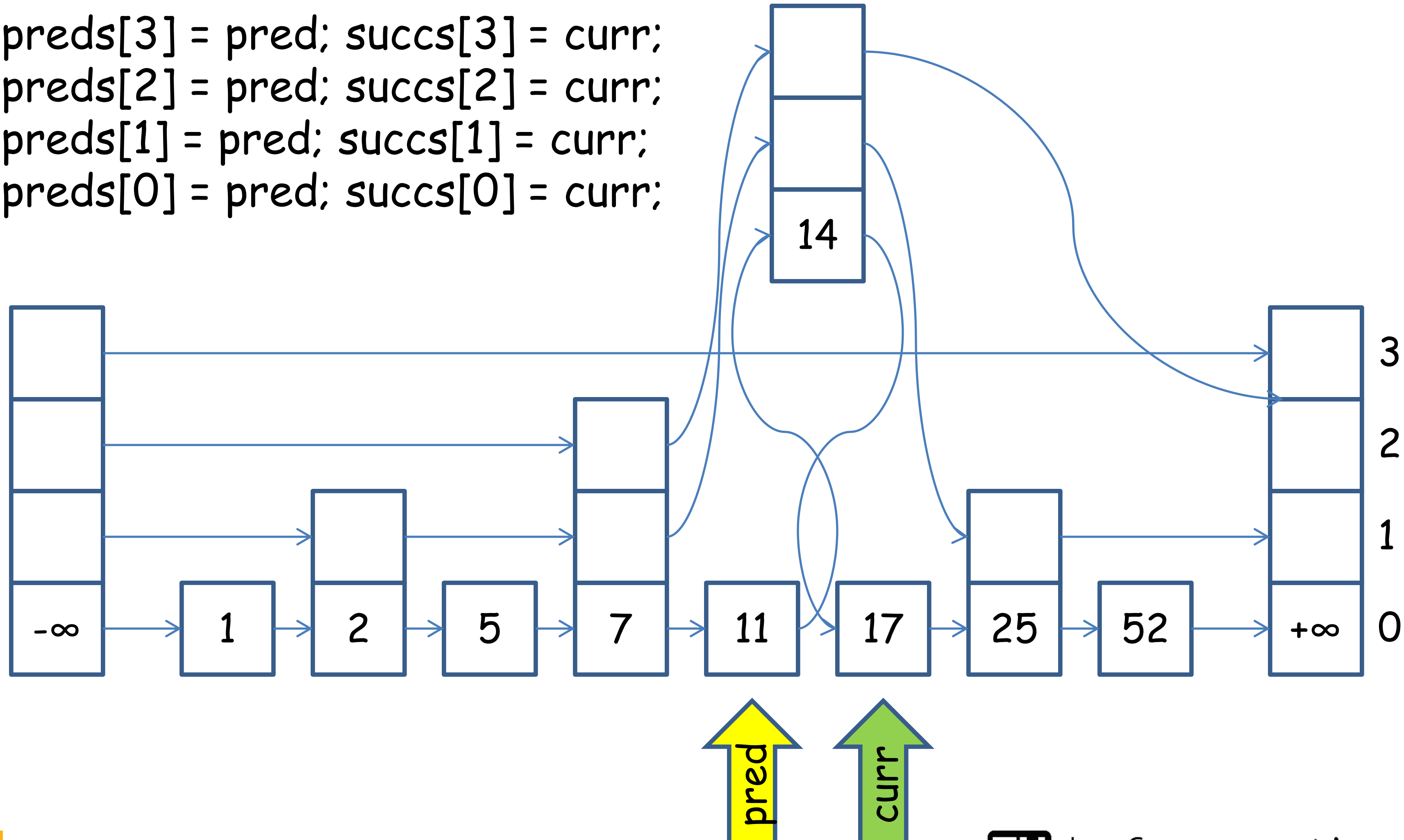
Example: add(14)

preds[3] = pred; succs[3] = curr;
preds[2] = pred; succs[2] = curr;
preds[1] = pred; succs[1] = curr;
preds[0] = pred; succs[0] = curr;



Example: add(14)

preds[3] = pred; succs[3] = curr;
preds[2] = pred; succs[2] = curr;
preds[1] = pred; succs[1] = curr;
preds[0] = pred; succs[0] = curr;



Lazy Skiplist

Idea:
Fine-grained locking, wait-free contains operation, remove elements lazily by setting marked flag: Each level is essentially as in the lazy list-based set algorithm.

To maintain skiplist property, insertion of new node is done by linking in from level 0 up to maximum level of new node, at deletion, links must be removed from maximum level of node down to 0

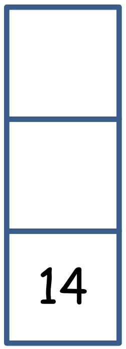
To maintain skiplist property, insertion of new node is done by linking in from level 0 up to maximum level of new node, at deletion, links must be removed from maximum level of node down to 0

In order have a wait-free contains() operation, locking only in add() and remove() operations, no lock in find().

Requires lock per node (note: reentrant/recursive lock convenient), marked flag per node, fully linked flag per node

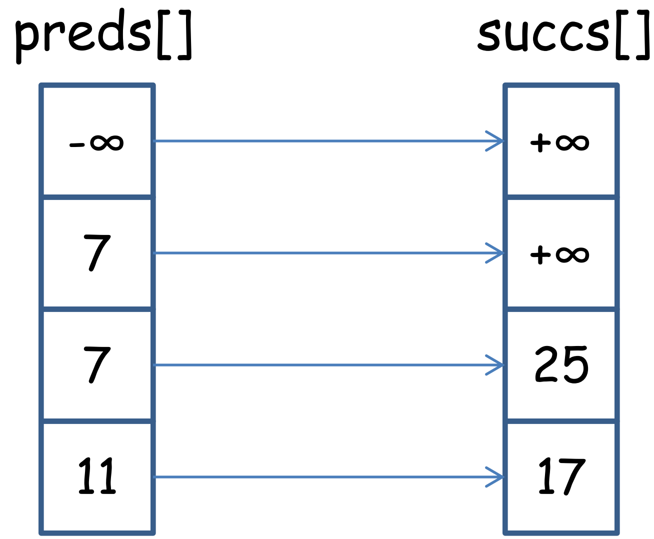
Invariant:
An element is in the skiplist iff it is unmarked and fully linked

The Shavit-Herlihy implementation



A (new) node in the skiplist with k levels is represented by an array of k next pointers

find(14) method:
Builds node arrays preds[] and succs[] with preds[l].item<14 and 14≤succs[l].item, for all l; returns first foundlevel from top such that item==succs[foundlevel].item



find(x): found if x==succs[foundlevel].item

add(x):
1. find(x)
2. If not marked, wait for fully linked
3. Lock predecessors (but only up to level of new element x) from bottom to top
4. Validate predecessors
5. Link in from bottom to top
6. Set fully linked flag

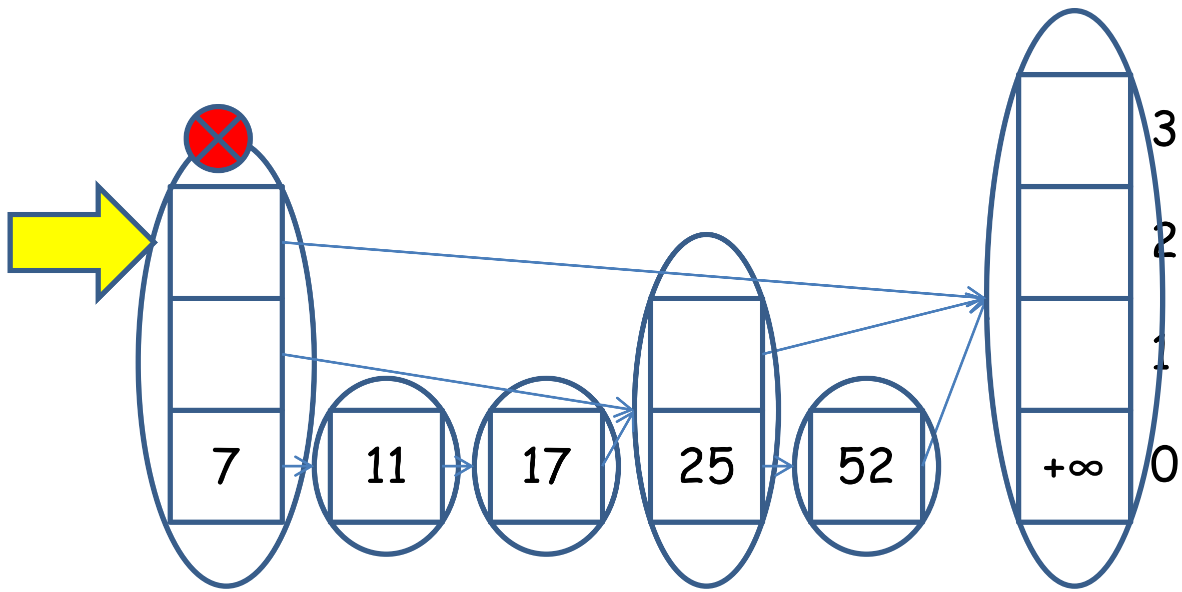
remove(y):
1. find(y)
2. Lock node
3. Set marked flag
4. Lock predecessors (only up to level of y), from bottom to top
5. Validate predecessors
6. Link out from top to bottom

```
private static final class Node<T> {
  final Lock lock = new ReentrantLock();
  final T item; // use as key… (sloppy)
  final Node<T>[] next;
  volatile boolean marked = false;
  volatile boolean fullylinked = false;
  private int k; // lastlevel for node

  public Node(T x, int levels) {
    item = x;
    next = new Node[levels+1];
    k = levels;
  }
  …
}
```

Reentrant (recursive) locks are needed. Why? See next slides…

Note:
- Next references are to Node's
- Locks are per node
- Flags (marked, fullylinked) are per node, not per level

k: last level for node



Reentrant (recursive) locks are needed (lock is per node)

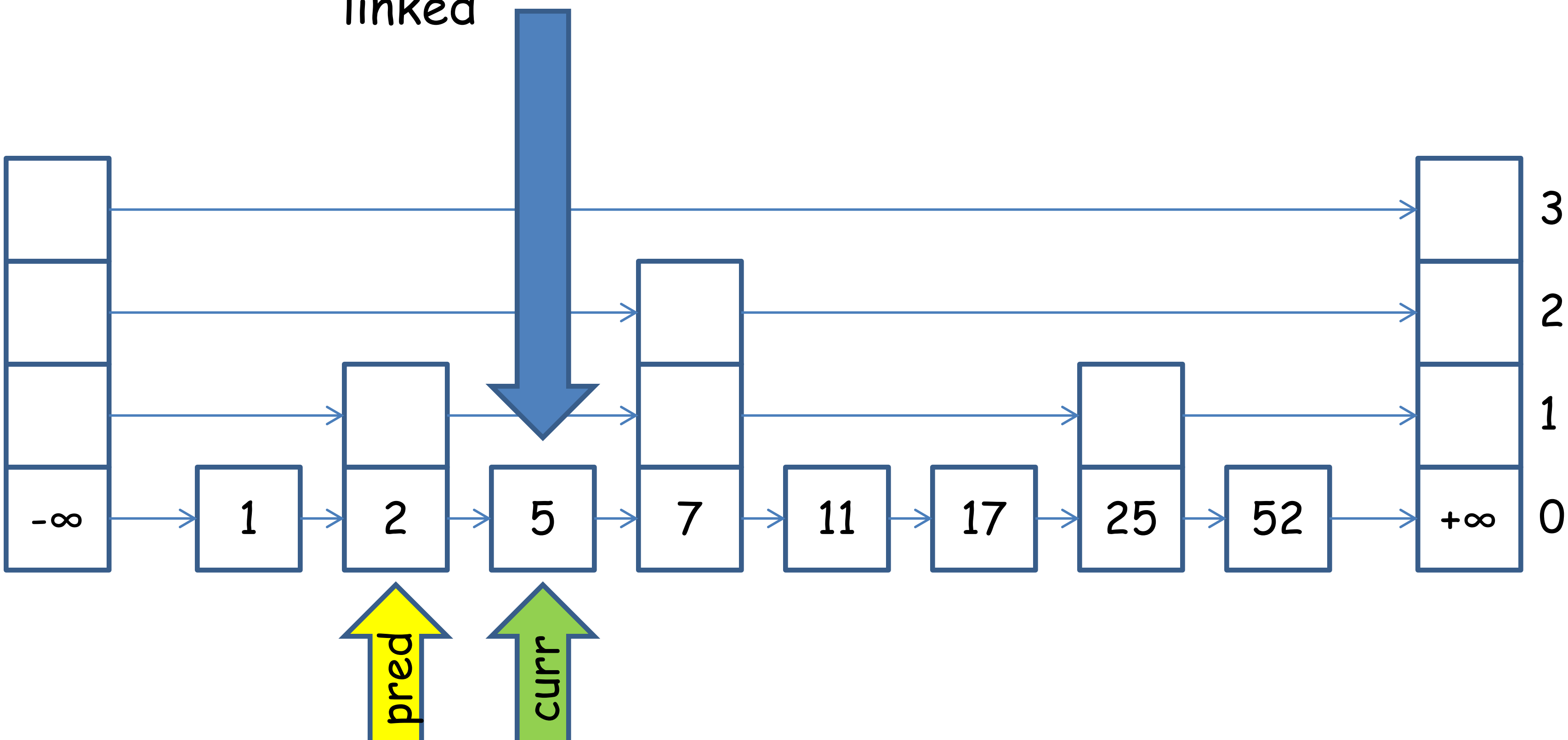
```
int find(T item, Node<T>[] preds, Node<T>[] succs) {
  int foundlevel = -1; // not found at any level≥0
  Node<T> pred = head;
  for (int l = MAX_LEVEL; l>=0; l--) {
    volatile Node<T> curr = pred.next[l];
    while (item>curr.item) {
      pred = curr; curr = curr.next[l];
    }
    if (foundlevel==-1&&item==curr.item)
      foundlevel = l; // found at level l
    preds[l] = pred; succs[l] = curr;
  }
  return foundlevel;
}
```

```
int contains(T item) {
  Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL+1];
  Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL+1];

  int foundlevel = find(x,preds,succs);
  return (foundlevel>=0 &&
          succs[foundlevel].fullylinked &&
          !succs[foundlevel].marked);
}
```

contains(x) is wait-free: No locks, no spinning

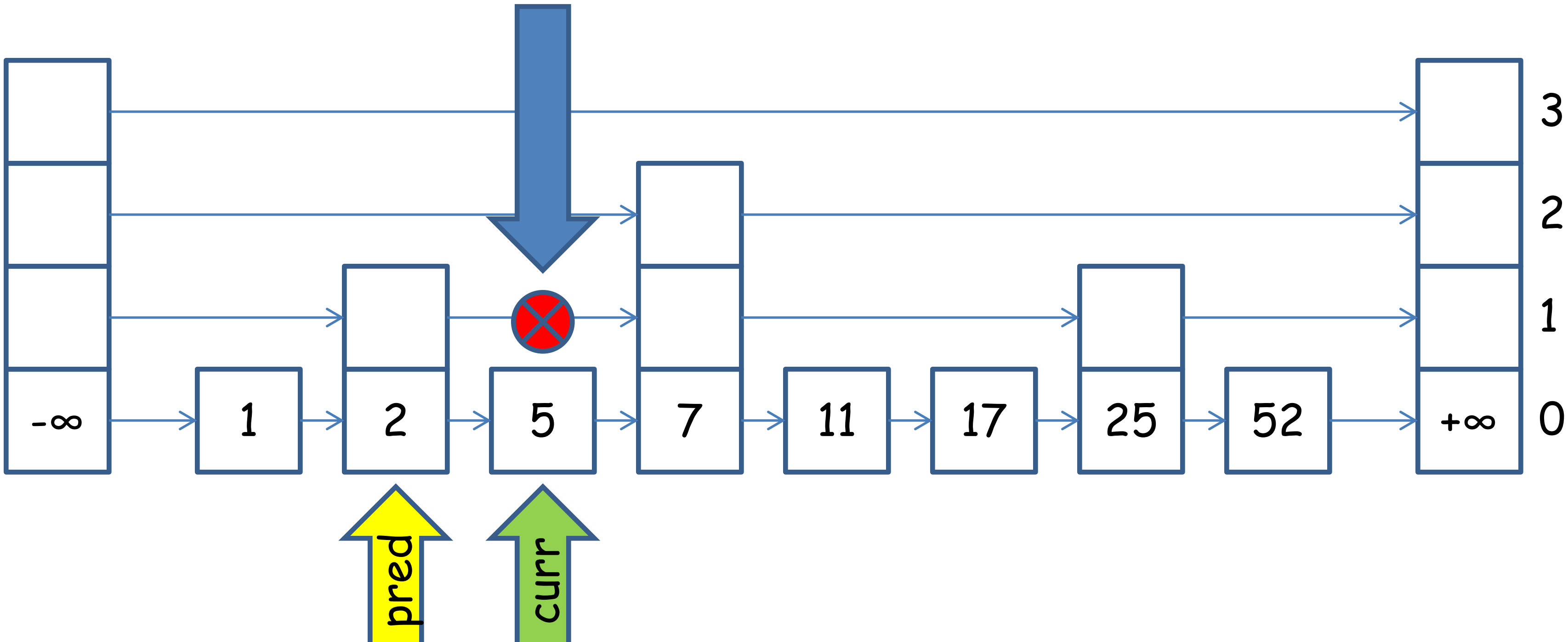
Example: remove(5)

"victim" for removal: Check unmarked and fully linked



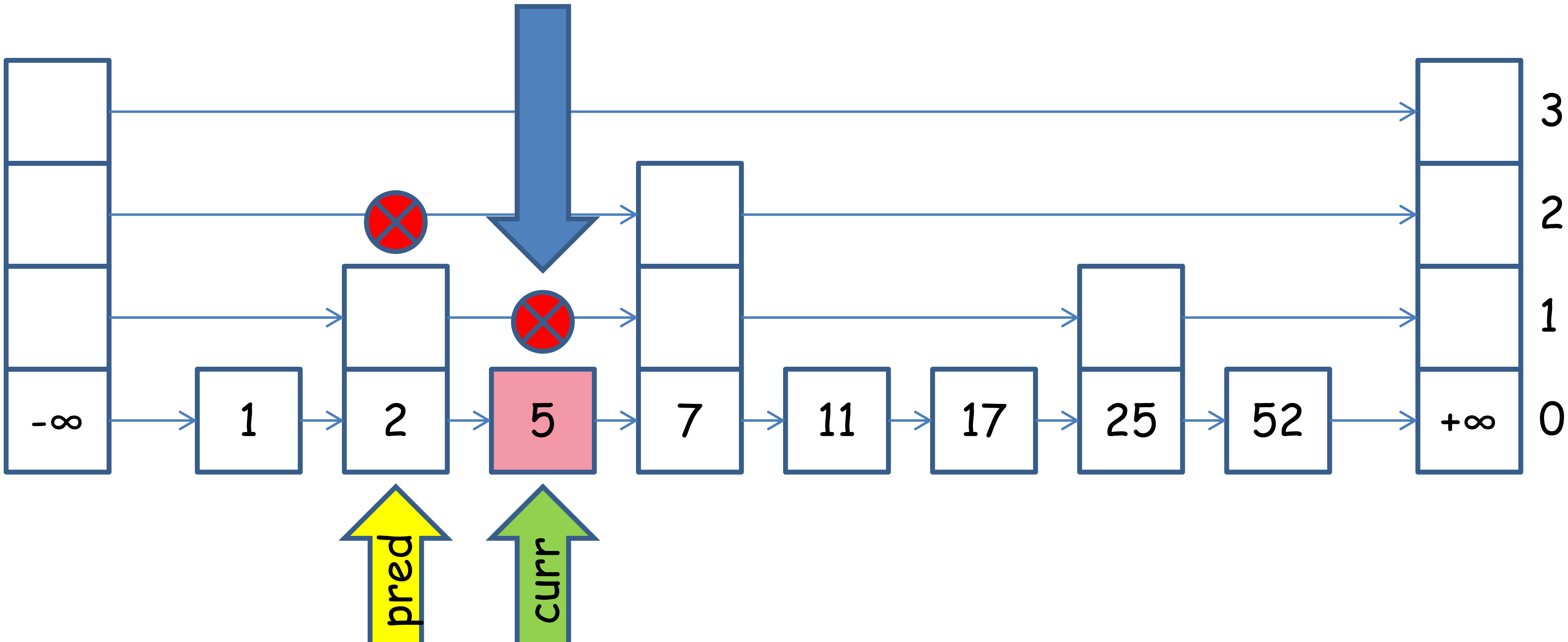
Example: remove(5)

"victim" for removal: check unmarked and fully linked
lock victim, mark, lock predecessors (k=0 for this victim)



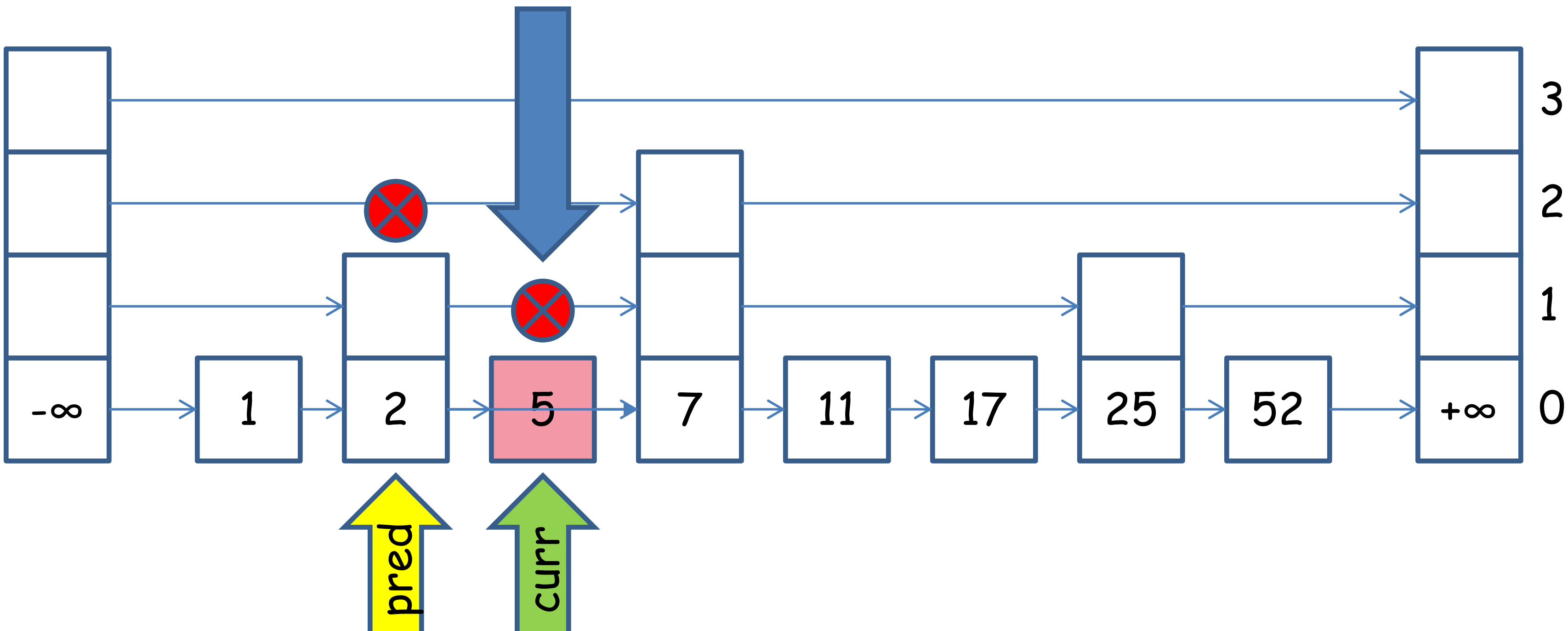
Example: remove(5)

"victim" for removal: check unmarked and fully linked
lock victim, mark, lock predecessors (k=0 for this victim)



Example: remove(5) successful

"victim" for removal: check unmarked and fully linked
lock victim, mark, lock predecessors
unlink, unlock



```
boolean remove(T y) {
  Node<T> victim = null;
  boolean marked = false; int k = -1;
  Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL+1];
  Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL+1];
  while (true) {
    int f = find(y,preds,succs);
    if (f>=0) victim = succs[f];
    if (marked||
        (f>=0&&victim.fullylinked&&victim.k==f&&
         !victim.marked)) {
      if (!marked) { // mark victim only once
        k = victim.k;
        victim.lock.lock();
        if (victim.marked) {
          victim.lock.unlock(); return false;
        }
        victim.marked = true; marked = true;
      }
      // now marked, try to link out … (next slide)
    } else return false;
} }
```

See next slide

?

Check for `victim.k==f` is an optimization:

If `victim` was found at a level `f<victim.k`, either victim is not yet fully linked (by concurrent add()) or is concurrently being unlinked by concurrent remove() operation, and validation would fail anyway

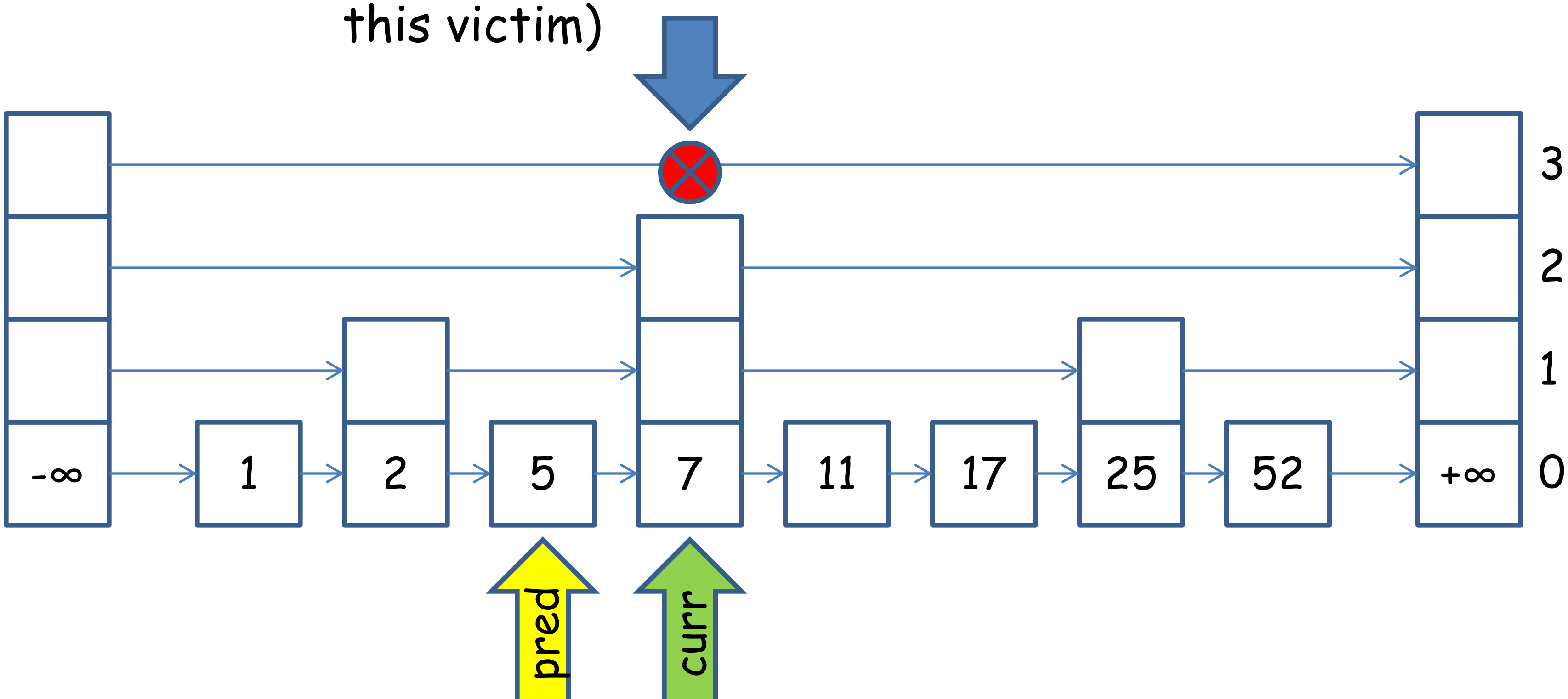
```
// try to link out (previous slide)
int highlock = -1;
try { // validate
  Node<T> pred, succ; boolean valid = true;
  for (l=0; valid&&(l<=k); l++) {
    pred = preds[l];
    pred.lock.lock();
    highlock = l;
    valid = !pred.marked&&pred.next[l]==victim;
  }
  if (!valid) continue;
  for (l=k; l>=0; l--)
    preds[l].next[l] = victim.next[l];
  victim.lock.unlock();
  return true;
} finally
  for (l=0; l<=highlock; l++) preds[l].lock.unlock();
```

Validation failed

Unlink downwards

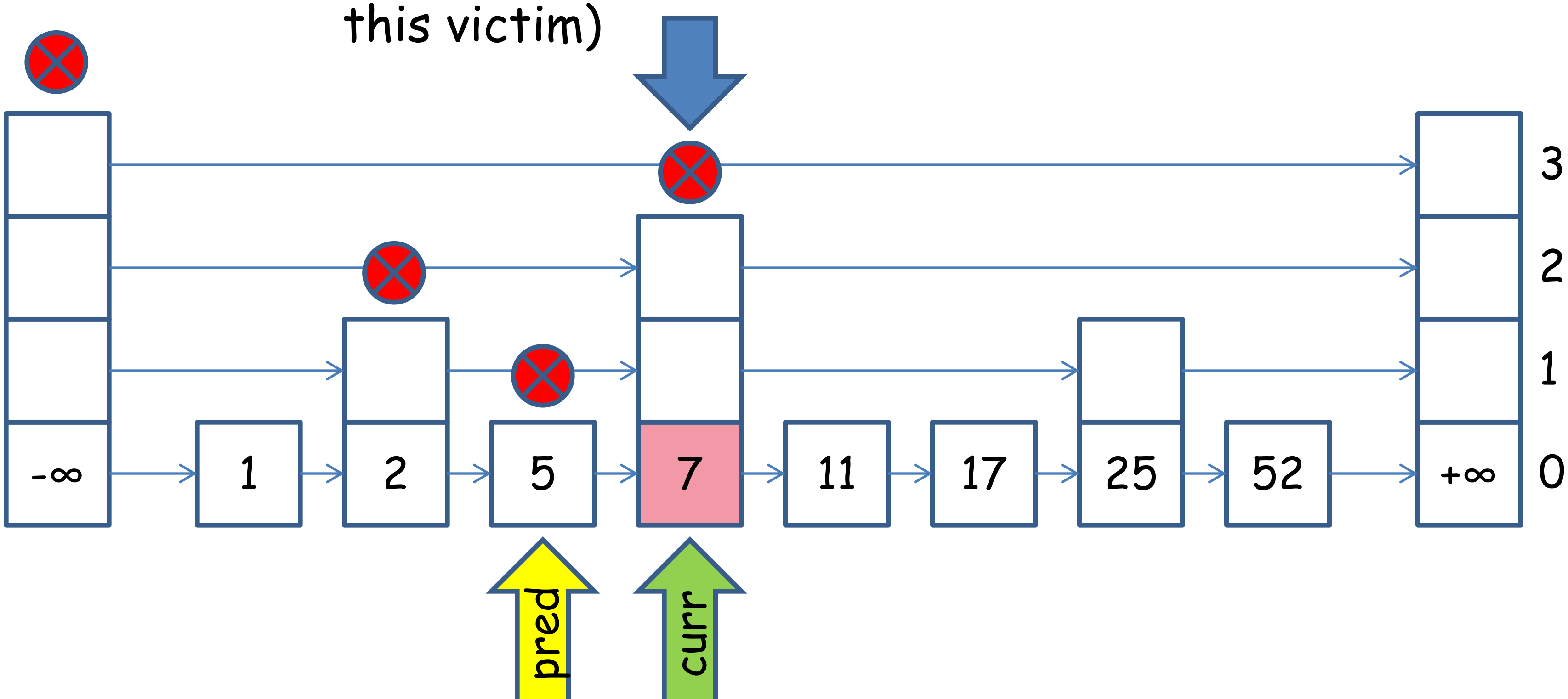
Example: remove(7)

"victim" for removal: check unmarked and fully linked
lock victim, mark, lock predecessors (k==2 for this victim)



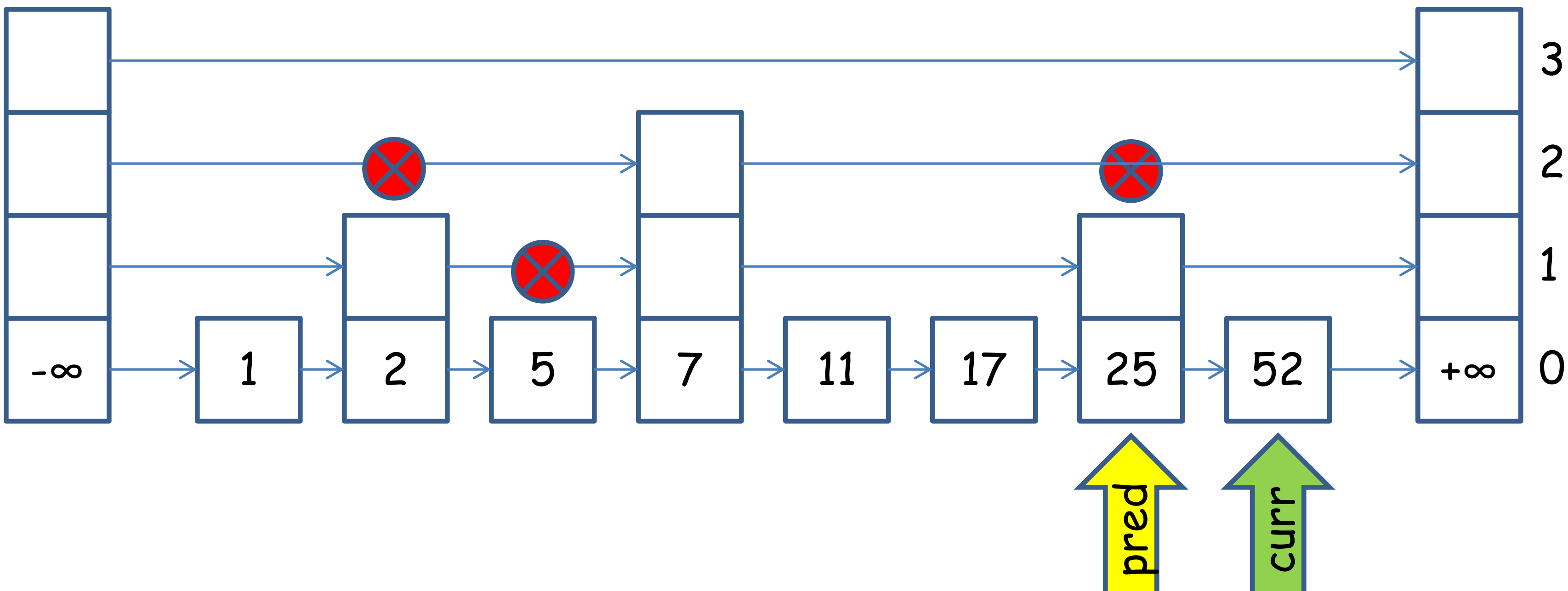
Example: remove(7)

"victim" for removal: check unmarked and fully linked
lock victim, mark, lock predecessors (k==2 for this victim)



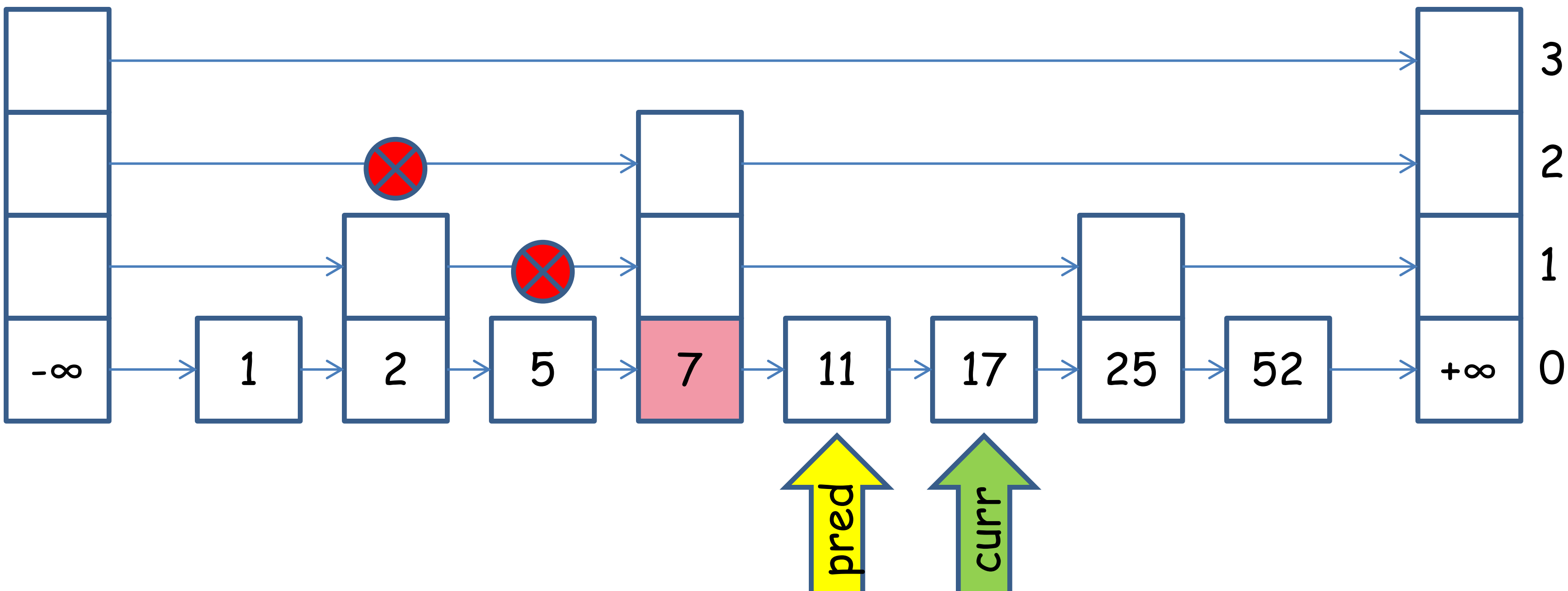
Example: remove(5), with concurrent add(40)

Find, lock and validate (predecessor and successor unmarked, pred.next==succ). Here: new element with k=0
Link in, bottom to top to maintain skiplist property



Example: remove(5), with concurrent, unsuccessful add(17)

Assume 17 not yet fully linked: Wait (spin) for node to become fully linked (what will happen if not?)



```
boolean add(T x) {
  int k = random_level(MAX_LEVELS);
  Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL+1];
  Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL+1];
  while (true) {
    int f = find(x,preds,succs);
    if (f>=0) {
      Node<T> Found = succs[f];
      if (!Found.marked)
        while (!Found.fullylinked) {} // spin
        return false;
      }
      continue;
    }
    // not in skiplist, try to add (next slide)
}
```

```
// try to add
highlock = -1; // highest level locked
try {
  Node<T> pred, succ;
  boolean valid = true;
  for (l=0; valid&&(l<=k); l++) { // validate
    pred = preds[l]; succ = succs[l];
    pred.lock.lock();
    highlock = l;
    valid =
     !pred.marked&&!succ.marked&&pred.next[l]==succ;
  }
  if (!valid) continue; // failed, retry
  Node<T> newNode = new Node(x,k); // allocate new
  for (l=0; l<=k; l++) { // link in upwards
    newNode.next[l] = succs[l];
    preds[l].next[l] = newNode;
  }
  newNode.fullylinked = true;
} finally
  for (l=0; l<highlock; l++) preds[l].lock.unlock();
```

Finding the right number of levels for new node

Number of levels for each node is chosen randomly, such that that the skiplist properties are preserved (constant fraction of nodes at next level)

```
int random_level(int maxlevel);
  int i = 0;
  for (i=0; i<maxlevel; i++)
    // choose random number in [0,1[ uniformly
    if (random(0,1)<1/f) break;
  return i;
}
```

With f=2, probability of k=0 is constant (1/2), probability that k=i, i≥1, is $2^{-(i+1)}$, probability that k is 31 is 2^{-32} (cf. Herlihy-Shavit)

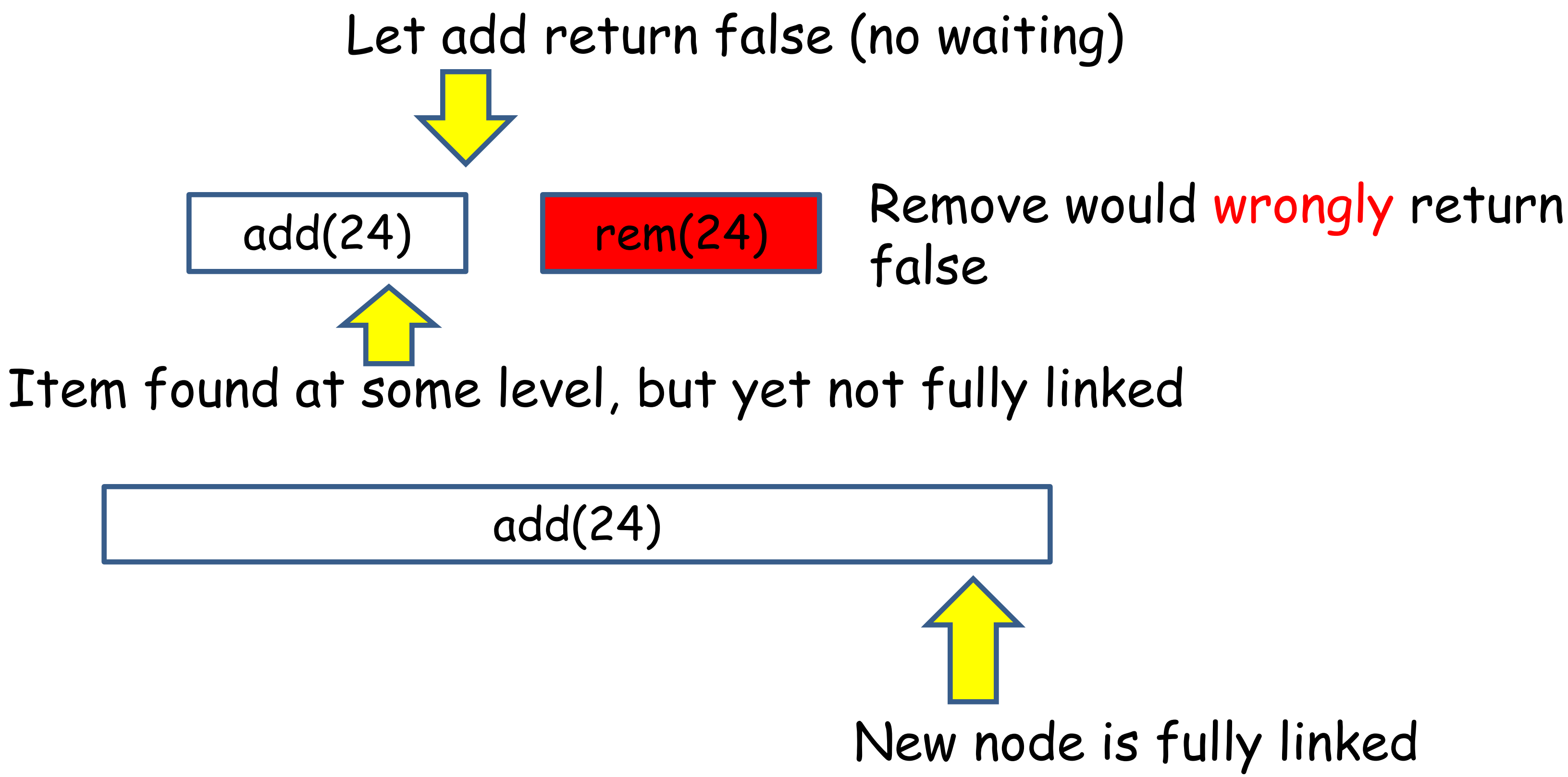
Properties and invariants:

Locks (add and remove) always acquired in same order, bottom to top: no deadlock

- Maintains skiplist property: List at level l is sublist of list at level l-1, lists are ordered (add sets links from bottom to top)
- An element is in the skiplist iff it is reachable, fully linked (linked on all levels) and not marked

When add(x) finds unmarked, but not fully linked node, it must wait for node to become fully linked (here: spin) to make operation linearizable while maintaining the skiplist property

Non-linearizable history when not waiting for fully linked



Correctness: All operations are linearizable

Unsuccessful add(x): Check for unmarked and fully linked succeeds
Successful add(x): After linking in the new node and setting it to fully linked

Unsuccessful remove(y): The found node is already marked (being deleted by concurrent thread), or not found, or not fully linked
Successful remove(y): Marking the found node

Successful contains(z): Predecessors next reference unmarked and fully linked
Unsuccessful contains(z): Node not found, or found marked, or before linearization of concurrent add (see lazy list)

Liveness:

Contains obviously wait-free

Add and remove not starvation free (even if locks are)

Lock-free Skiplist

Idea: Lock free lists at each level; need to maintain reference/pointer AND mark as one unit

Java: Each next field is an `AtomicMarkableReference<Node>`

Note:
Can update lists only one level at a time (single-word CAS), therefore not possible to maintain skiplist property: List at level l will no longer be a sublist of list at level l-1

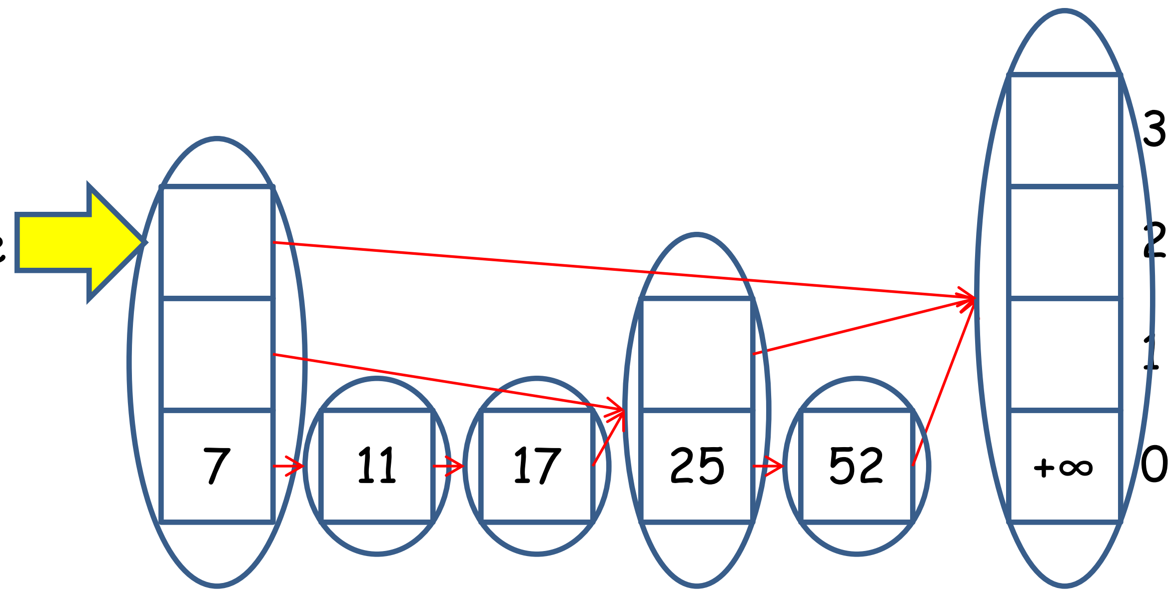
Instead:
List at level l is a shortcut into list at level l-1.
Property of abstract set: An element is in the set iff it is in the list at level 0

```
private static final class Node<T> {
  final T item; // use as key… (sloppy)
  final AtomicMarkableReference<Node<T>>[] next;
  private int k; // number of levels for node

  public Node(T x, int levels) {
    item = x;
    next = (AtomicMarkableReference<Node<T>>[])
      new AtomicMarkableReference[levels+1];
    k = levels;
    for (l=0; l<=k; l++) next[l] = new
      AtomicMarkableReference<Node<T>>(null,false);
  }
  …
}
```

Note:

- No locks, no fully linked flag for the nodes.
- Each reference to a next node is a marked reference, mark indicates that node link is logically deleted and should be skipped/short-cut: Node item is not present at this level

k: last level for node



find(x, preds, succs):

As in lock-free list, must physically link out marked nodes at each level (with CAS: compareAndSet in Java). Returns only whether item was found or not

To make contains(z) wait-free: Jump over marked nodes, as in lock-free list

```
boolean find(T x, Node<T>[] preds, Node<T>[] succs) {
  int b = 0;
  boolean[] marked = {false};
  Node<T> pred = null, curr = null, succ = null;
  retry:
    while (true) {
      pred = head;
      for (l=MAX_LEVEL; l>=0; l--) { // go down levels
        curr = pred.next[l].getReference();
        while (true) {
          succ = curr.next[l].get(marked);
          while (marked[0]) { // link out marked nodes
            if (!pred.next[l].
                   compareAndSet(curr,succ,false,false))
              continue retry;
            curr = pred.next[l].getReference();
            succ = curr.next[l].get(marked);
          }
```

```
            if (curr.item<x) {
              pred = curr; curr = succ;
            } else break;
          }
          preds[l] = pred; succs[l] = curr;
        }
        return (curr.item==x);
      }
    }
}
```

Linearization point when l==0

```
boolean contains(T item) {
  int bottom = 0;
  boolean[] marked = {false};
  Node<T> pred = head; curr = null; succ = null;
  for (int l=MAX_LEVEL; l>=bottom; l--) {
    curr = pred.next[l].getReference();
    while (true) {
      succ = curr.next[l].get(marked);
      while (marked[0]) {
        curr = curr.next[l].getReference();
        succ = curr.next[l].get(marked);
      }
      if (curr.item<item) {
        pred = curr; curr = succ;
      } else break;
    }
  }
  return (curr.item==item);
}
```

(sometimes) linearization point when l==0 add(x):

find(x), link in with CAS, on failure restart

remove(y):

find(y), attempt to mark from top to bottom (setting mark atomically), restart on failure. Find again to clean up lists


```
boolean add(T x) {
  int k = random      MAX_LEVELS); int b = 0;
  Node<T>[] pred      de<T>[]) new Node[MAX_LEVEL+1];
  Node<T>[] su        <T>[]) new Node[MAX_LEVEL+1];
  while (true) {
    if (find(x,preds,succs)) return false;
    else { // prepare new node
      Node<T> newNode = new Node(x,k);
      for (l=b; l<=k; l++) {
        Node<T> succ = succs[l];
        newNode.next[l].set(succ,false);
      }
      Node<T> pred = preds[b];
      Node<T> succ = succs[b];
      if (!pred.next[b].compareAndSet(succ,newNode,
                                       false,false))
        continue;
    }
    // remaining levels
  } }
```

Linearization on fail

References set early

Linearization on success

```
// remaining levels
for (l=b+1; l<=k; l++) {
  while (true) {
    pred = preds[l]; succ = succs[l];
    if (pred.next[l].compareAndSet(succ,newNode,
                                   false,false))
      break;
    find(x,preds,succ);
  }
}
return true;
```

```
boolean remove(T y) {
  int b = 0;
  Node<T>[] preds          e<T>[]) new Node[MAX_LEVEL+1];
  Node<T>[] succ           T>[]) new Node[MAX_LEVEL+1];
  while (true) {
    if (!find(y,preds,succs)) return false;
    else {
      // shortcut lists from k down to b+1
      Node<T> remNode = succs[b];
      for (l=remNode.k; l>=b+1; l--) {
        boolean marked[] = {false};
        succ = remNode.next[l].get(marked);
        while (!marked[0]) {
          remNode.next[l].compareAndSet(succ,succ,
                                        false,true);
          succ = remNode.next[l].get(marked);
        }
      }
      // level 0 list
} } }
```

Linearization on fail

```
// level 0 list
boolean[] marked = {false};
succ = remNode.next[b].get(marked);
while (true) {
  boolean done =
    remNode.next[b].compareAndSet(succ,succ,
                                  false,true);
  succ = succs[b].next[b].get(marked);
  if (done) {
    find(y,preds,succs); // clean up (optimization)
    return true;
  } else if (marked[0]) return false;
}
```

Linearization on success

```
// level 0 list
boolean[] marked = {false};
succ = remNode.next[b].get(marked);
while (true) {
  boolean done =
    remNode.next[b].compareAndSet(succ,succ,
                                  false,true);
  succ = succs[b].next[b].get(marked);
  if (done) {
    find(y,preds,succs); // clean              ion)
    return true;
  } else if (marked[0]) return fals
}
```

Linearization on success

Fail by concurrent remove

Key points (I)

add(x):
New node is linked in starting from bottom level 0, if CAS fails predecessor has changed (marked or new node inserted concurrently), sublist property maintained. When new node has been added, repeat until it has been linked at all levels; may lead to violation of sublist property if new node is concurrently removed by other thread

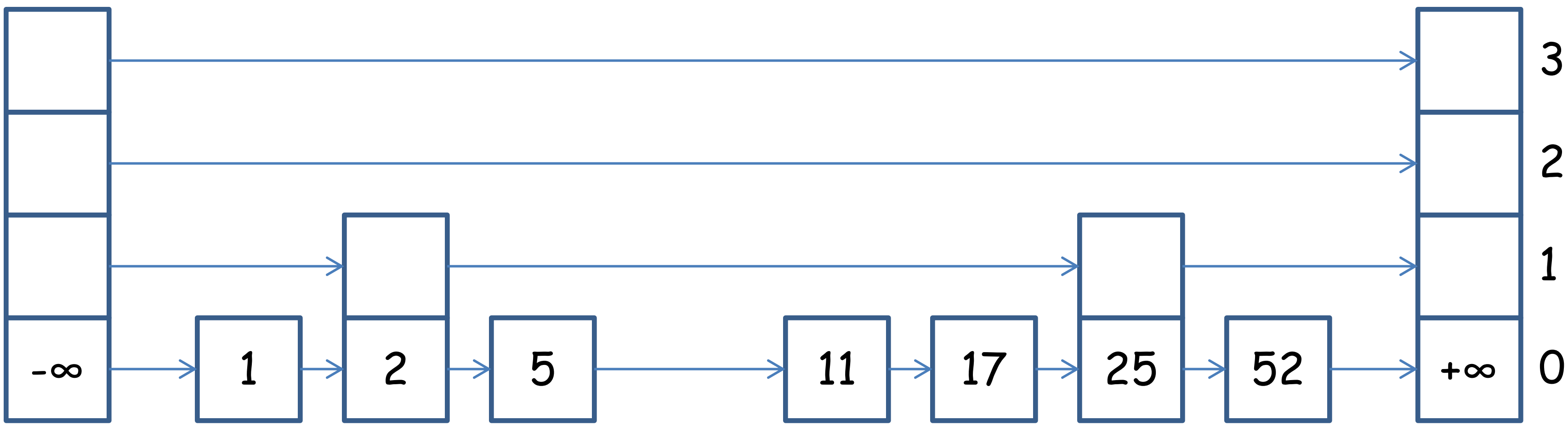
remove(y):
If found, node is marked starting from highest level, success if marking succeeds at bottom level 0; if CAS failure, node has been concurrently linked out by other thread. Node physically removed (linked out) by following find operation

Key points (II)

contains(x):
At each level, skip marked links, makes sure that item in set (at level 0) is eventually found

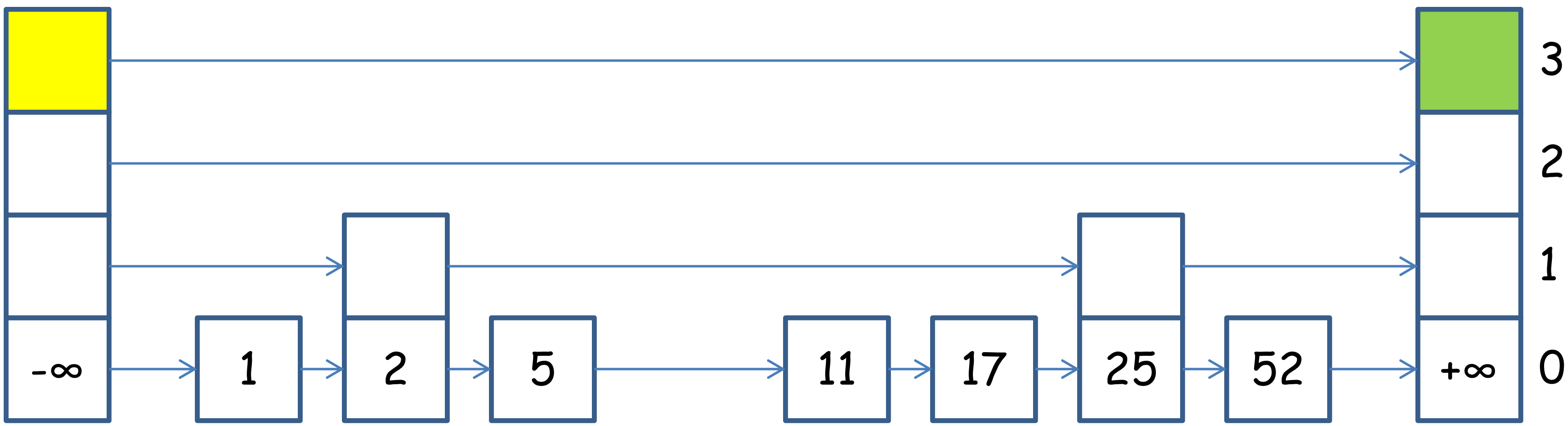
Example: add(7)

add(7): find



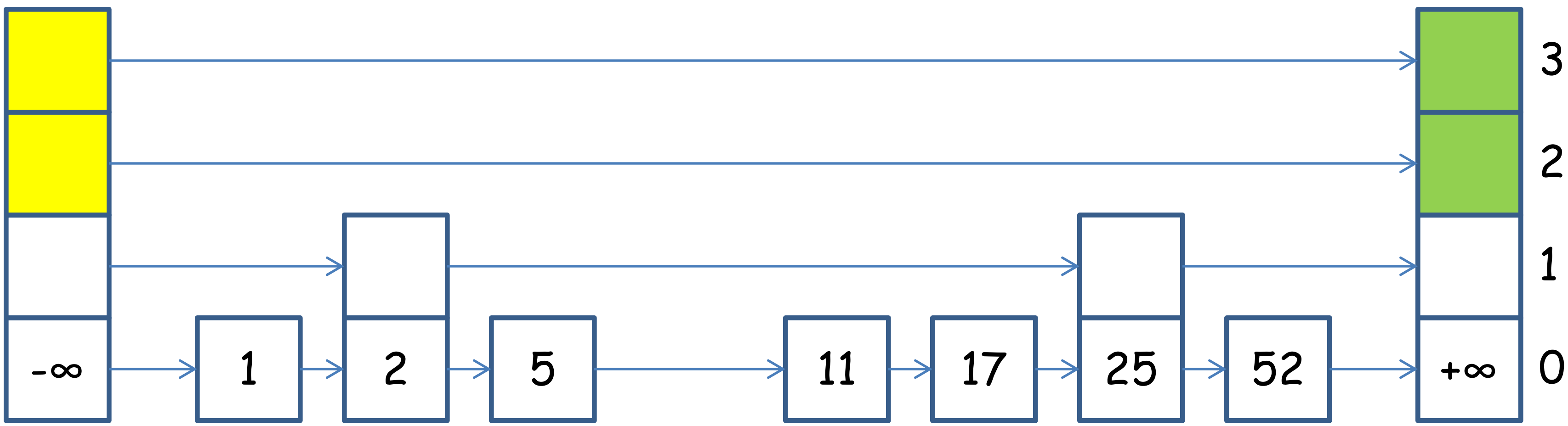
Example: add(7)

add(7): find



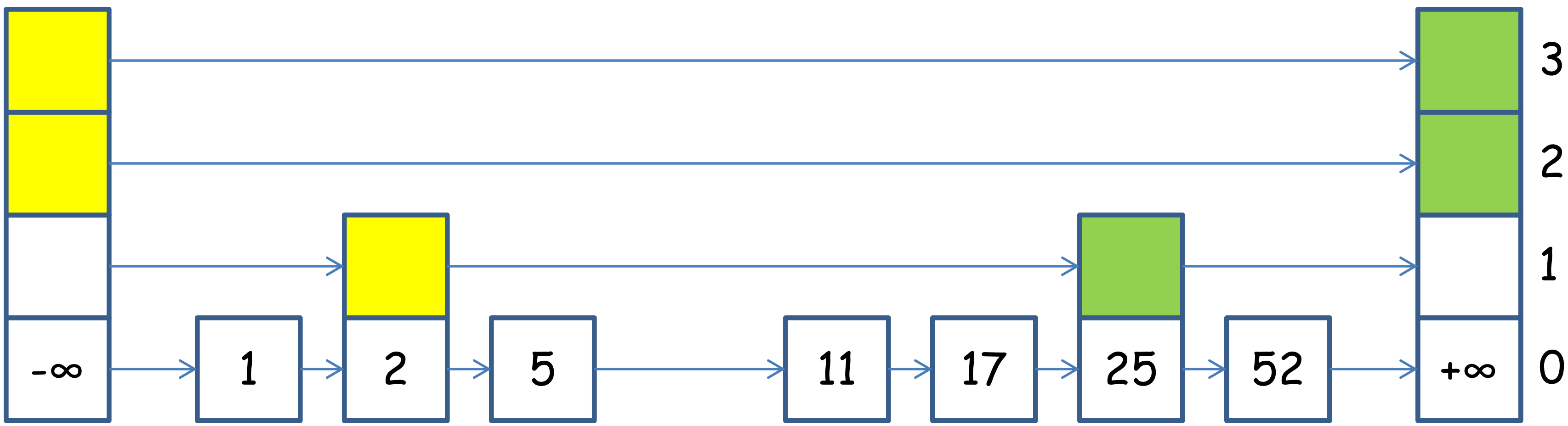
Example: add(7)

add(7): find



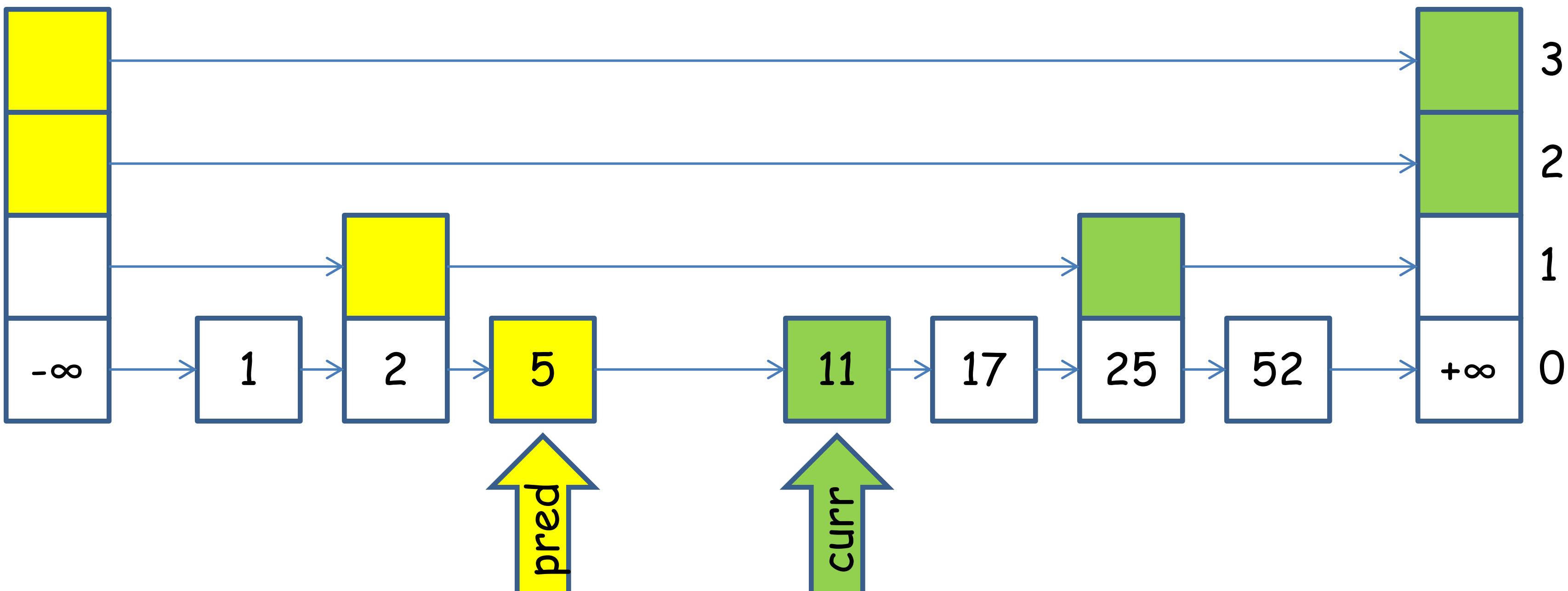
Example: add(7)

add(7): find

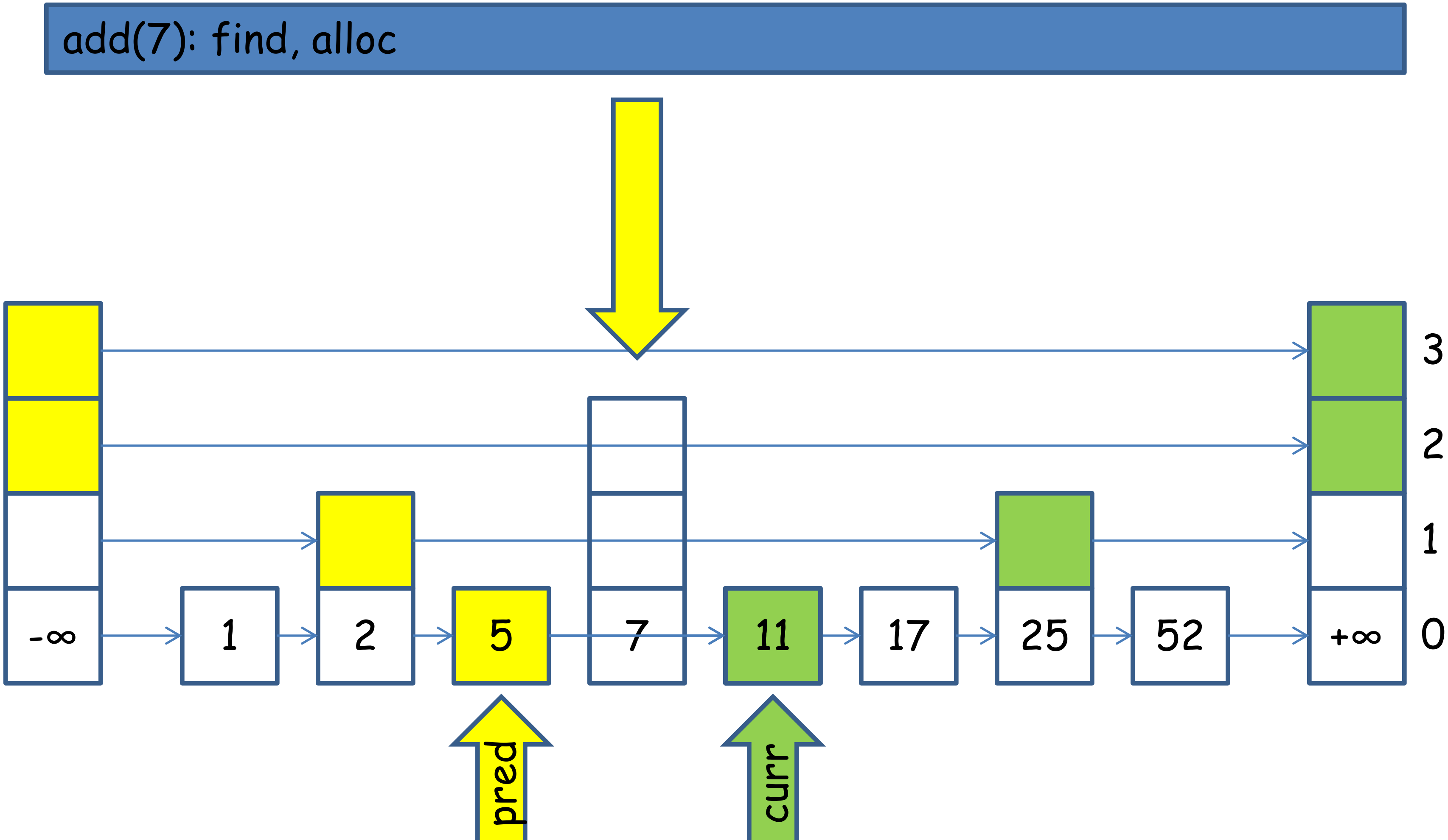


Example: add(7)

add(7): find

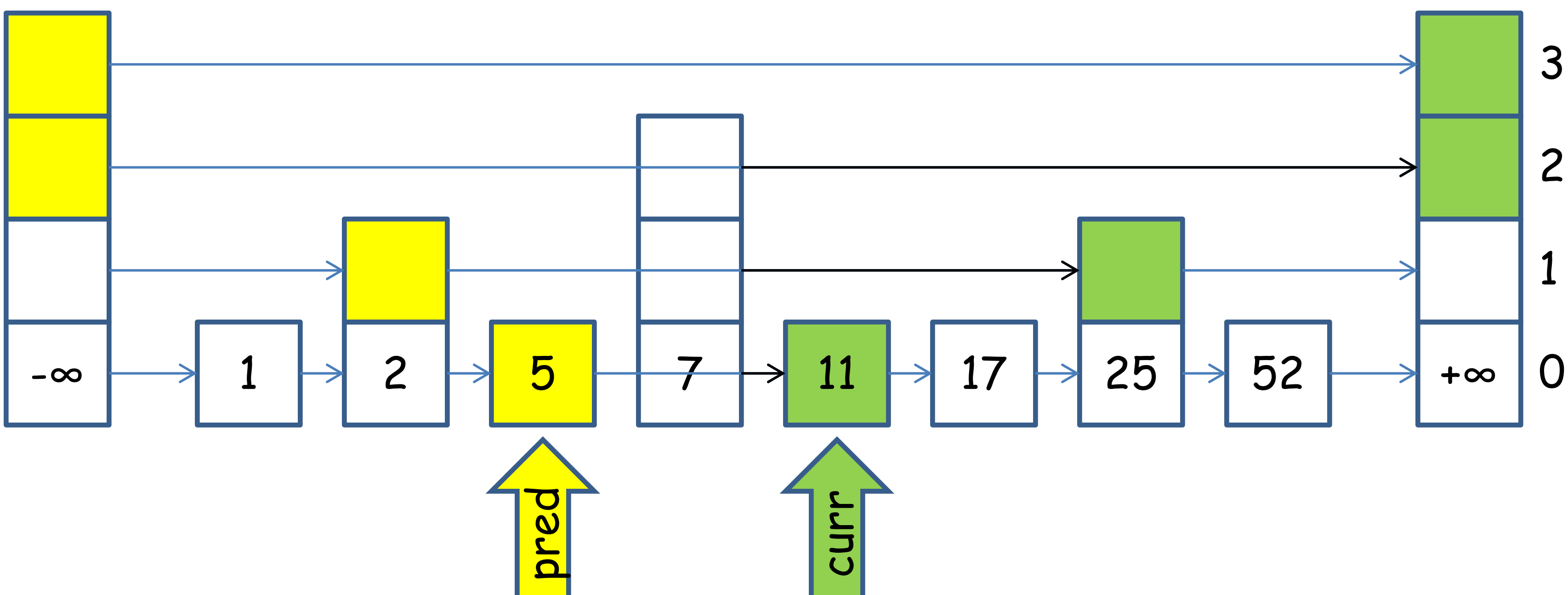


Example: add(7)



Example: add(7)

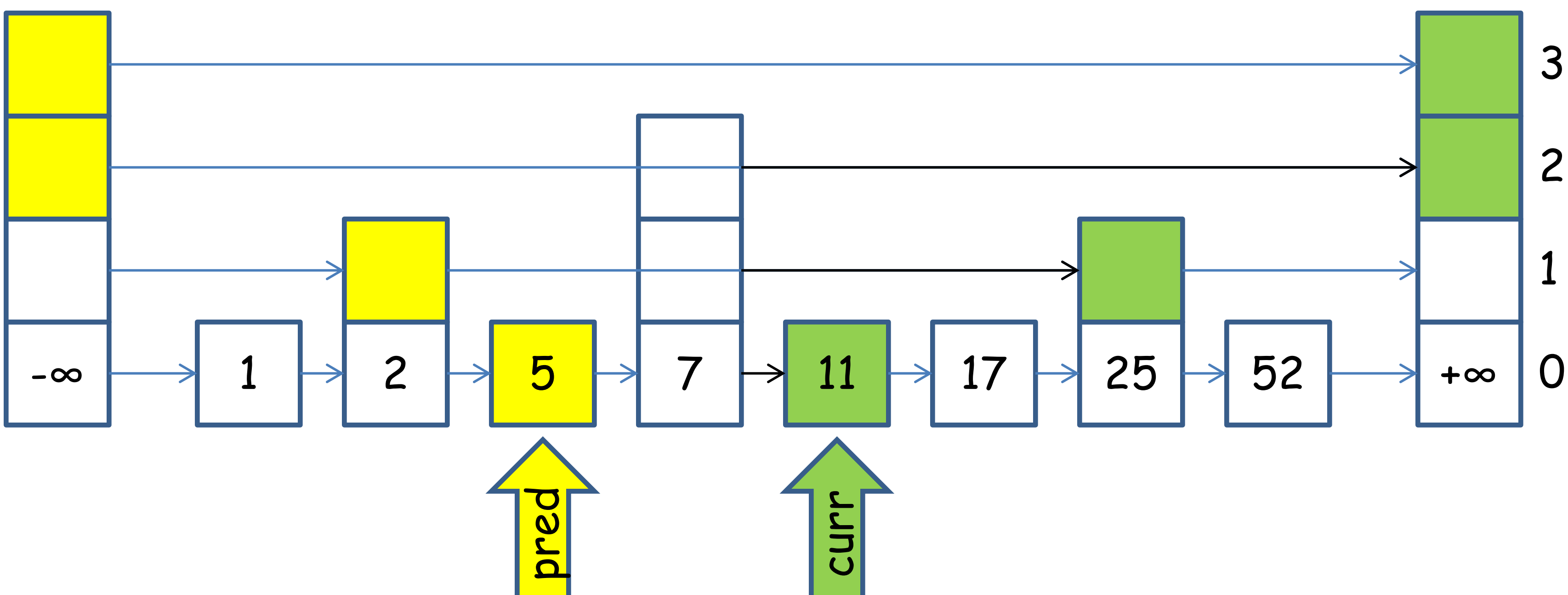
add(7): find, alloc, link



Linearization point of successful add

Example: add(7)

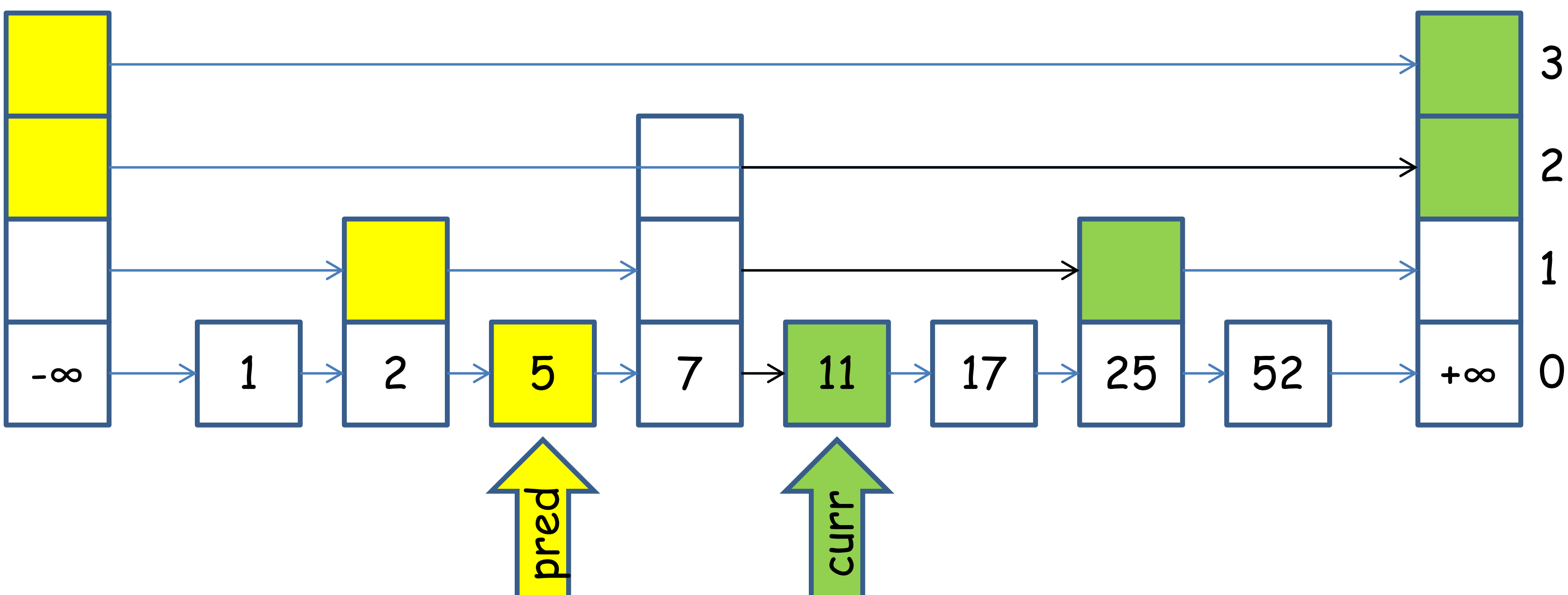
add(7): find, alloc, link, level 0



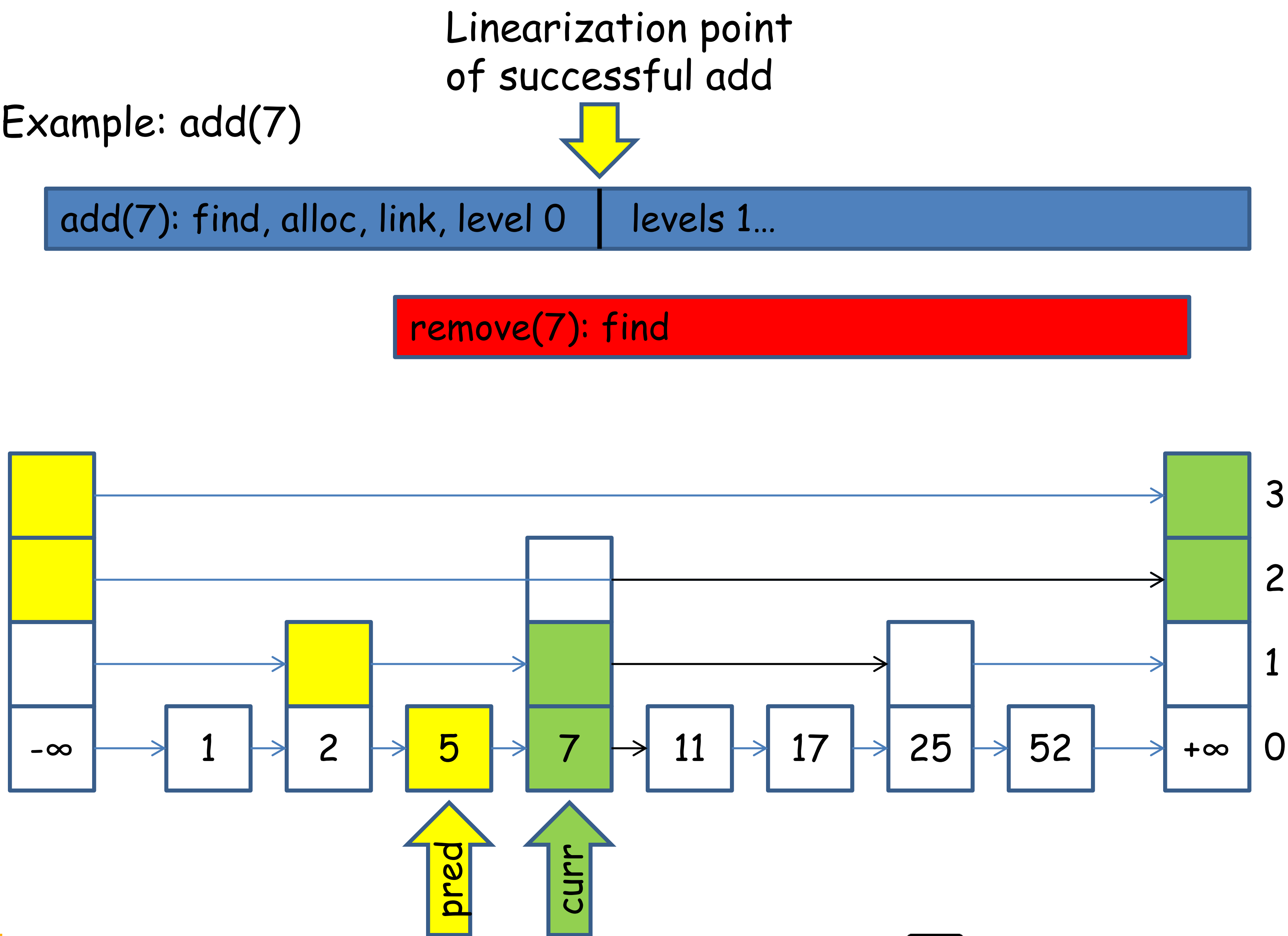
Linearization point
of successful add

Example: add(7)

add(7): find, alloc, link, level 0 | levels 1…



Example: add(7)

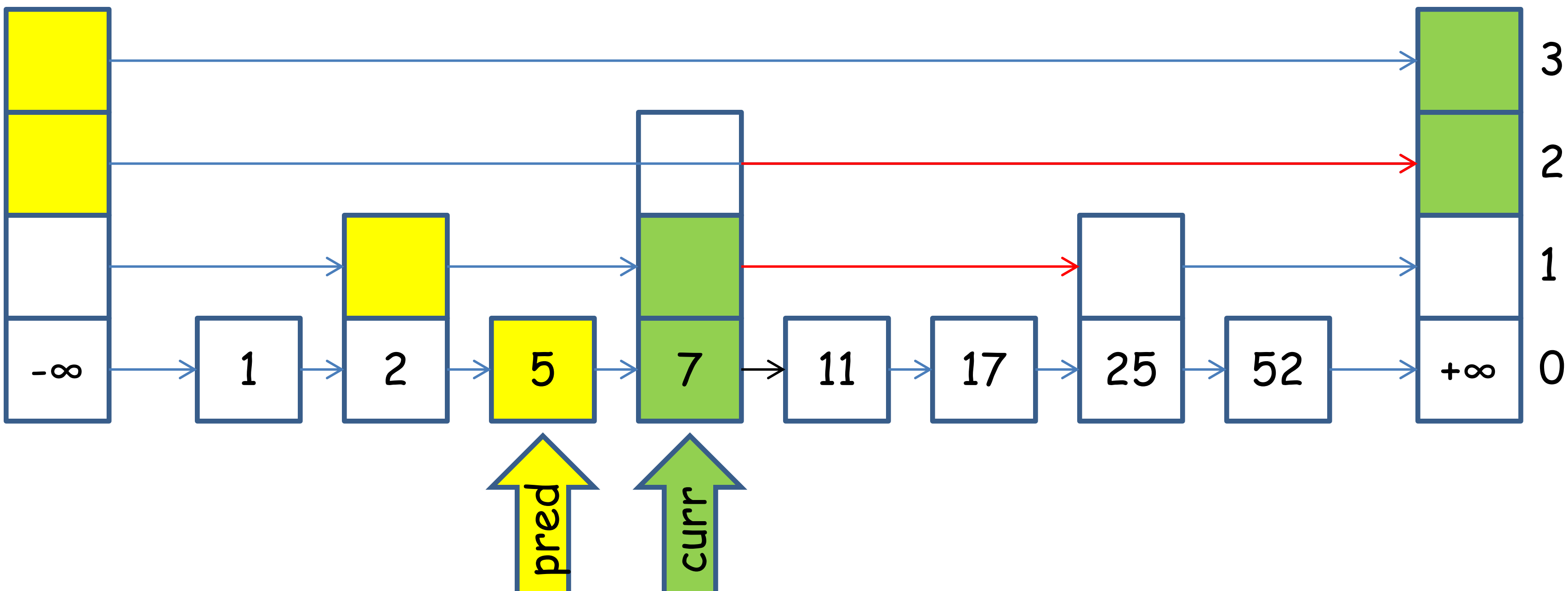


Linearization point
of successful add

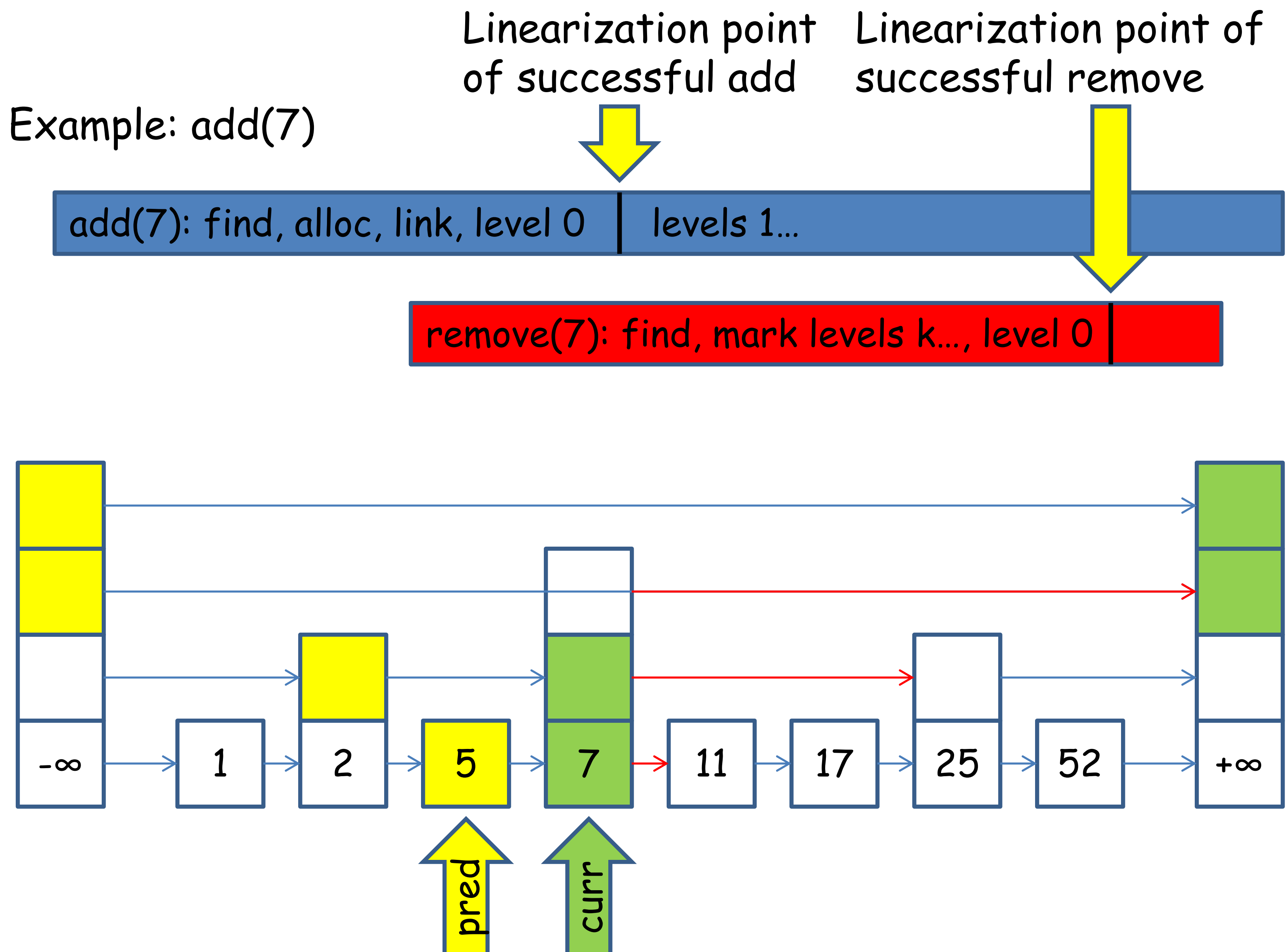
Example: add(7)

add(7): find, alloc, link, level 0 | levels 1…

remove(7): find, mark levels k…



Example: add(7)



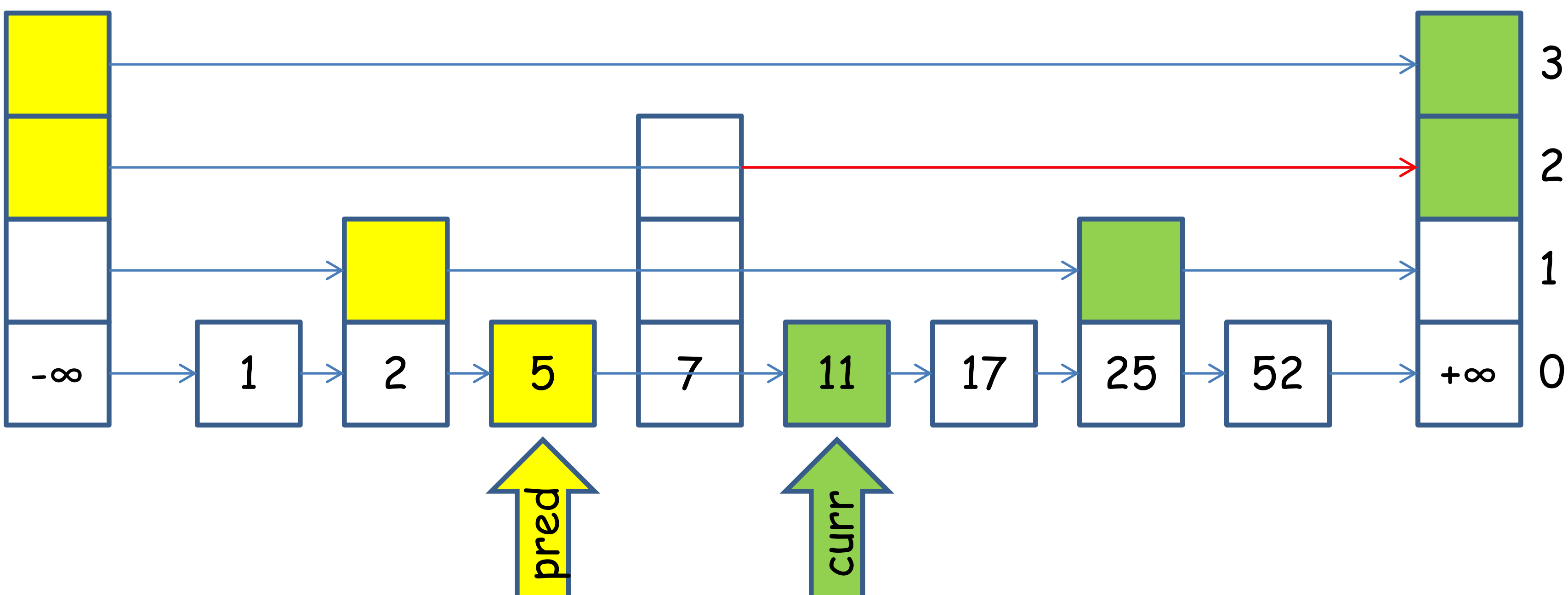
Linearization point of successful add

Linearization point of successful remove

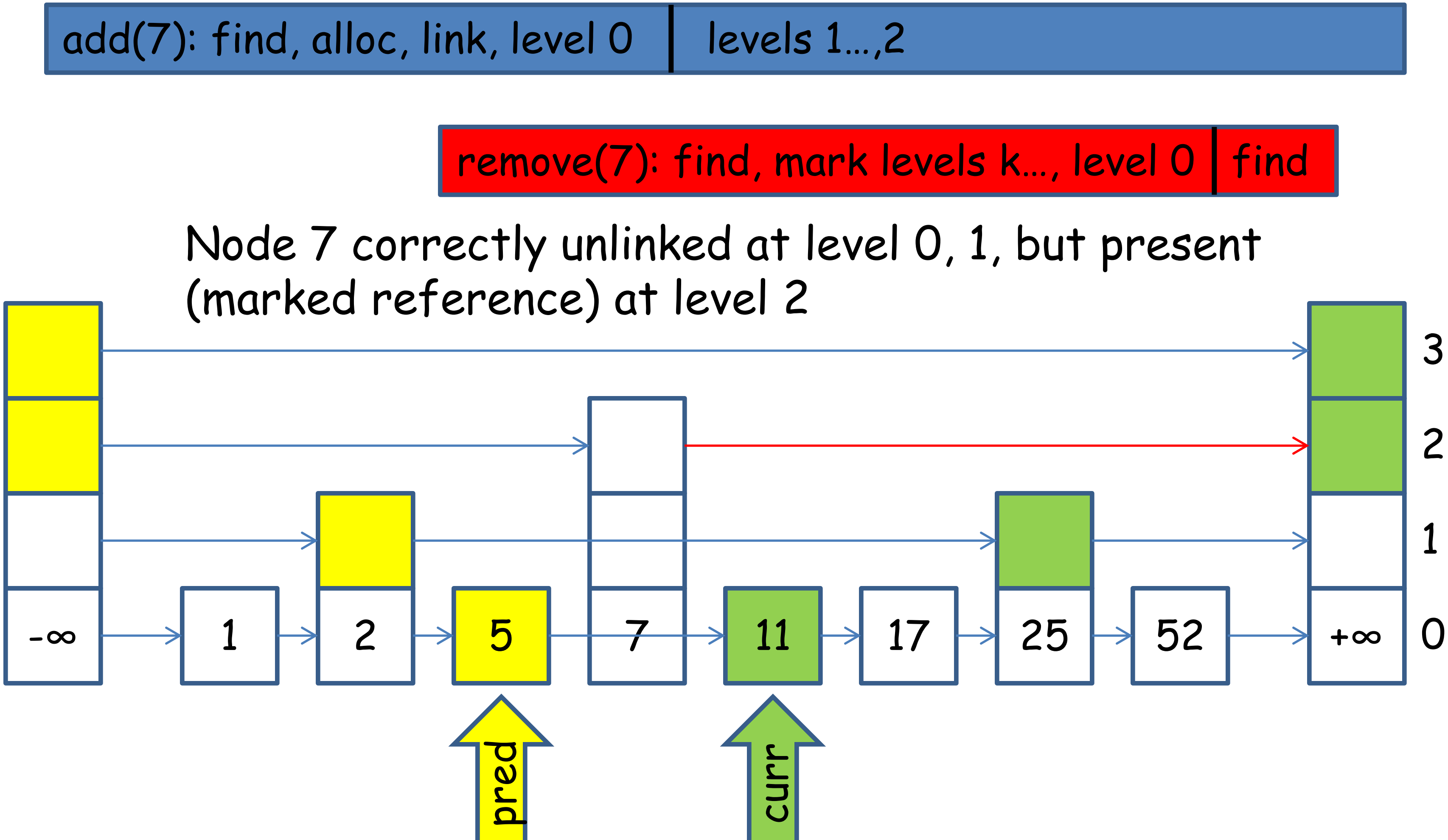
Example: add(7)

add(7): find, alloc, link, level 0 | levels 1…

remove(7): find, mark levels k…, level 0 | find



Example: add(7)



Correctness: All operations are linearizable

Successful add(x): The moment the element is in the list at level 0: successful CAS
Unsuccessful add(x): Linearization point of find

Successful remove(y): The point where the node containing the element is marked (CAS)
Unsuccessful remove(y): Not found (linearization point of find), or concurrent remove sets marked

find(z): node with item>=z found at level 0

contains(z): on overlapping add(z) call, either point where node is linked in at level 0, or if not found, before other thread links node in

Liveness:

contains() obviously wait-free

add() and remove()

- Lock-free
- But not starvation free

Lazy vs. lock-free skiplist

Skiplist data structure robust, it does not need the strict sublist property: Key observation for lock-free algorithm

Lock-free list in some way simpler than lazy, fine-grained locking based algorithm

Memory management: By garbage collection

Which performs better?

Another interesting algorithm for multi-linked data structures: Maintain property only for a subset of the links, other links may violate property, but serve as useful hints:

Håkan Sundell, Philippas Tsigas: Lock-free deques and doubly linked lists. J. Parallel Distrib. Comput. 68(7): 1008-1020 (2008)

List in one direction always correct (item in data structure iff it is reachable in this direction), links in other direction only approximate. Threads help each other to get other direction up to date

Priority-queue based on Skiplist (chap. 15)

Often needed sequential data structure (pool) with fairness guarantees based on key-value (here item = key):

- insert(key)
- key = delete_min()

Sometimes:

- decrease_key(item_reference,new_key)
- delete(item_reference)

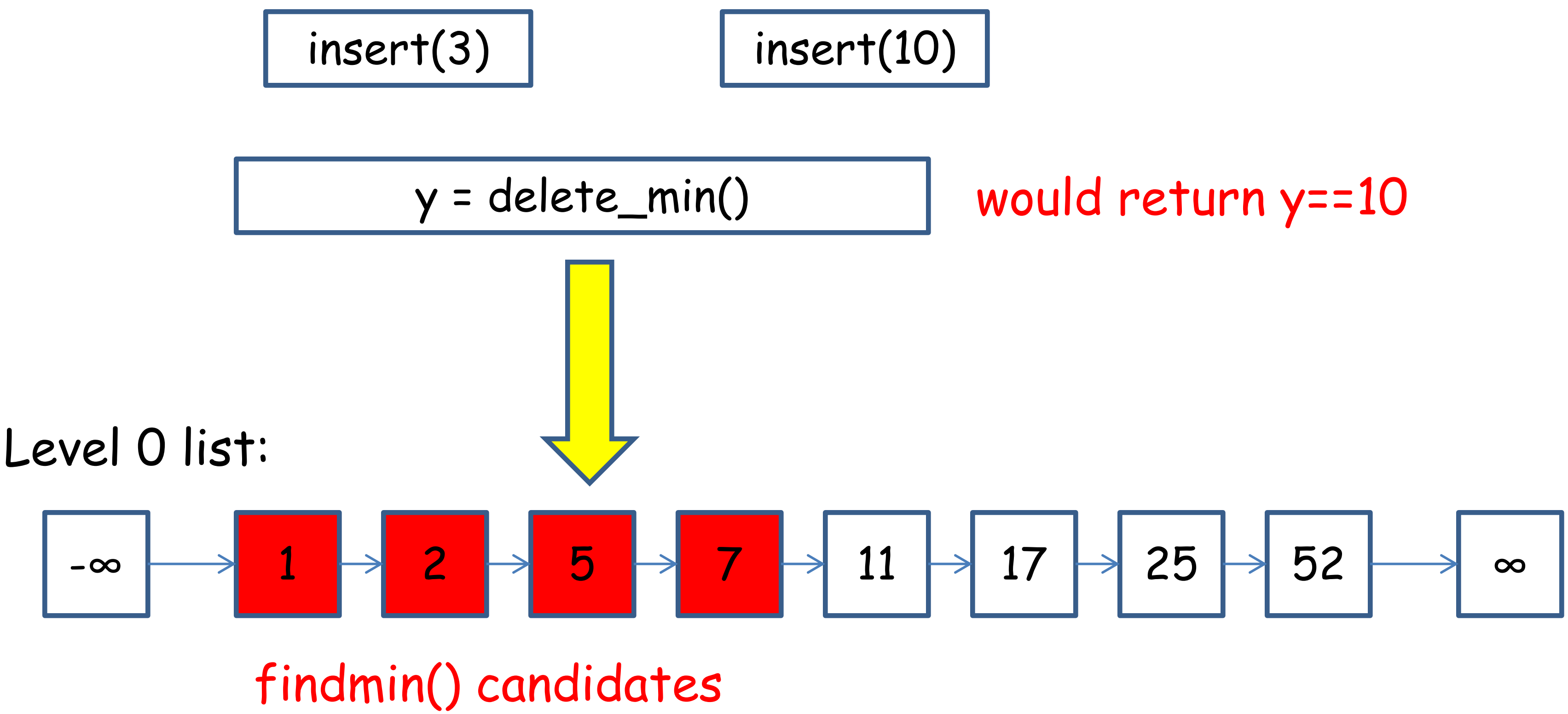
Skiplist based implementation (lock-free or lazy)

- Add atomic candidate flag to nodes
- delete_min() scans through level 0 list for first (=smallest key) non-candidate item, sets candidate flag (CAS), and deletes the node by calling skiplist remove()

```
public Node<T> findmincandidate() {
  Node<T> curr = null;
  curr = head.next[0].getReference();
  while (curr!=tail) {
    if (curr.candidate.compareAndSet(false,true))
      return curr;
    else curr = curr.next[0].getReference();
  }
  return null; // no candidates for delete_min
}
```

Note:
Could save some CAS operations by first reading the candidate flag (as in TTAS lock)

A linearizable execution cannot return key 10 before key 3 is returned



Properties of skiplist based priority queue

Progress guarantees (lock-freeness) inherited from skiplist

BUT:
- Not linearizable
- Instead, quiescently consistent

Alternative/better skiplist based priority queues

Nir Shavit, Itay Lotan: Skiplist-Based Concurrent Priority Queues. IPDPS 2000: 263-268
Håkan Sundell, Philippas Tsigas: Fast and lock-free concurrent priority queues for multi-thread systems. J. Parallel Distrib. Comput. 65(5): 609-627 (2005)
Jonatan Lindén, Bengt Jonsson: A Skiplist-Based Concurrent Priority Queue with Minimal Memory Contention. OPODIS 2013: 206-220

Relaxed priority queues

Dan Alistarh, Justin Kopinsky, Jerry Li, Nir Shavit: The SprayList: a scalable relaxed priority queue. PPOPP 2015: 11-20
Martin Wimmer, Jakob Gruber, Jesper Larsson Träff, Philippas Tsigas: The lock-free k-LSM relaxed priority queue. PPOPP 2015: 277-278
Tingzhe Zhou, Maged M. Michael, Michael F. Spear: A Practical, Scalable, Relaxed Priority Queue. ICPP 2019: 57:1-57:10

MultiQueue: A simple, randomized, relaxed priority queue

With p threads, use cp sequential priority queues, constant c>0 tuning parameter (each thread maintains c queues)

Idea:

- Insert: Choose queue randomly, try to lock, insert
- Delete minimum element: Choose two queues at random, try to lock the one with the smallest element, delete

Hamza Rihani, Peter Sanders, Roman Dementiev: Brief Announcement: MultiQueues: Simple Relaxed Concurrent Priority Queues. SPAA 2015: 80-82

```
boolean insert(T *x)
{
  int i; // queue index
  do {
    i = UniformRandom(0,cp);
  } while (!PQ[i].trylock())
  PQ[i].insert(x);
  PQ[i].unlock();
  return true;
}
```

```
boolean delete_min(T *x)
{
  int i, j; // two queue indices
  int k;
  do {
    i = UniformRandom(0,cp);
    j = UniformRandom(0,cp);
    // sample without locking
    if (PQ[i].findmin()<PQ[j].findmin())
      k = i; else k = j;
  } while (!PQ[k].trylock())
  PQ[k].delete_min(x);
  PQ[k].unlock();
  return true;
}
```

Proposition:
MultiQueue insert and delete_min are wait-free when $c>1$

Proof: Since at most p queues can be locked at any time, the success probability of finding an unlocked queue is constant

Corrollary:
MultiQueue insert and delete_min perform expected $O(1)$ trylock operations, and complete in time determined by the sequential priority queue operation

No strict bound for the relaxation (how far can the returned element of a delete_min operation be from the smallest element in queue at that time?), but two attempts is (much) better than one:

Yossi Azar, Andrei Z. Broder, Anna R. Karlin, Eli Upfal: Balanced Allocations. SIAM J. Comput. 29(1): 180-200 (1999)

Petra Berenbrink, Artur Czumaj, Angelika Steger, Berthold Vöcking: Balanced Allocations: The Heavily Loaded Case. SIAM J. Comput. 35(6): 1350-1385 (2006)

Michael Mitzenmacher: The Power of Two Choices in Randomized Load Balancing. IEEE Trans. Parallel Distrib. Syst. 12(10): 1094-1104 (2001)

New result

Dan Alistarh, Justin Kopinsky, Jerry Li, Giorgi Nadiradze:
The Power of Choice in Priority Scheduling. PODC 2017: 283-292

does give a tight bound on the relaxation quality (rank) of the MultiQueue

Benchmarking priority queues

Throughput benchmark often used in literature:

- Some mix of insertions and deletions (usually 50/50%)
- All threads perform same mix of operations
- Insert keys chosen uniformly from some (large) range, no dependency between deleted and inserted keys

Prefill queue, run for some interval (10 seconds), report total number of operations

Scalable PQ:
Throughput increases linearly with p (in some range)

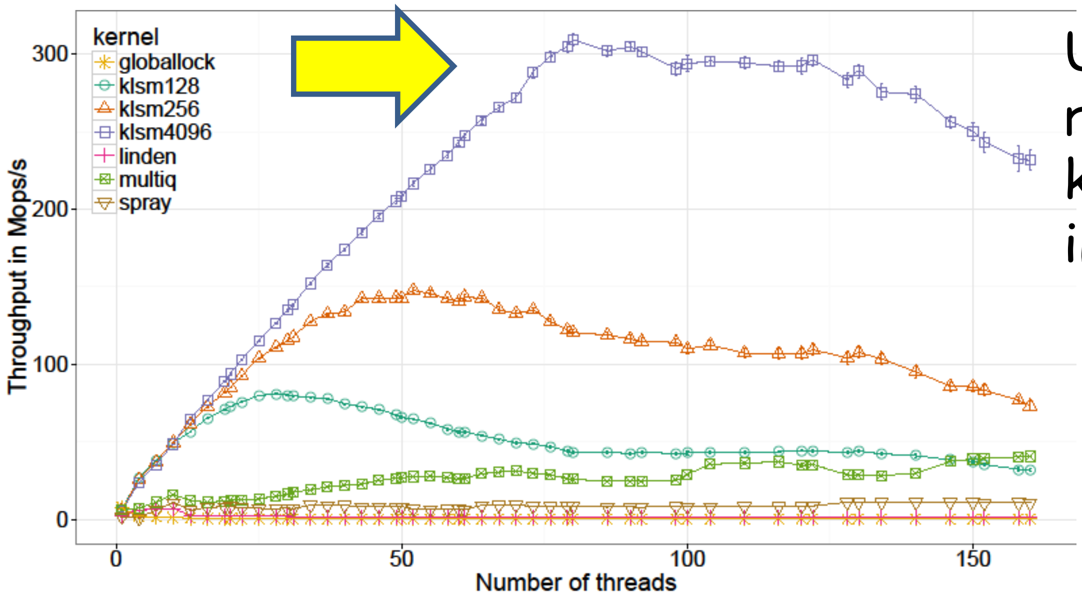
Question:

Is this a good benchmark?

Jakob Gruber, Jesper Larsson Träff, Martin Wimmer: Brief Announcement: Benchmarking Concurrent Priority Queues. Performance of k-LSM and Related Data Structures. SPAA 2016, to appear

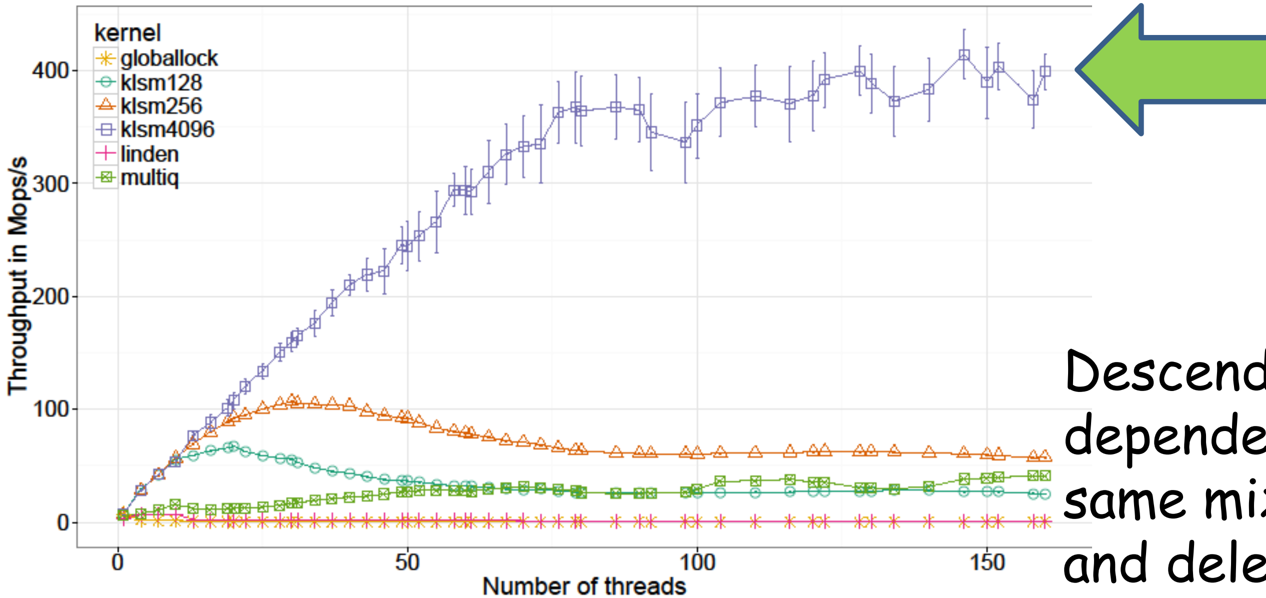
k-LSM:

- Linearizable, relaxed priority queue built on Log-Structured Merge trees, proposed by Martin Wimmer ("Variations for Shared Memory Systems", PhD thesis, TU Wien, 2014)
- Standalone C++ implementation by Jakob Gruber ("k-LSM: A relaxed, lock-free priority queue", MSc. Thesis, TU Wien, 2016)
- Hybrid of local and global LSM component, relaxation guarantee kp for chosen parameter k

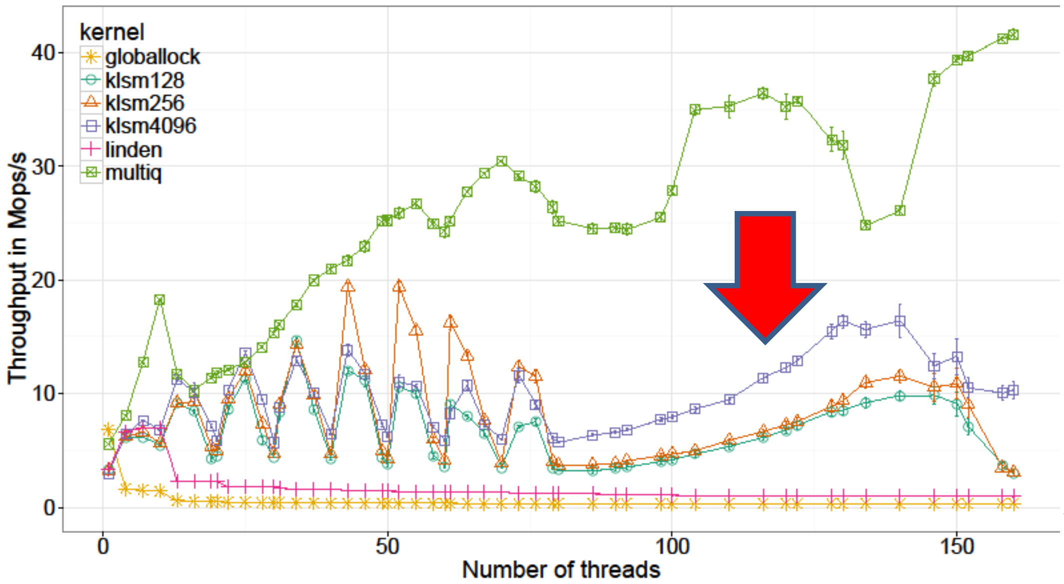


Uniformly drawn random 32-bit keys, same mix of insert and delete

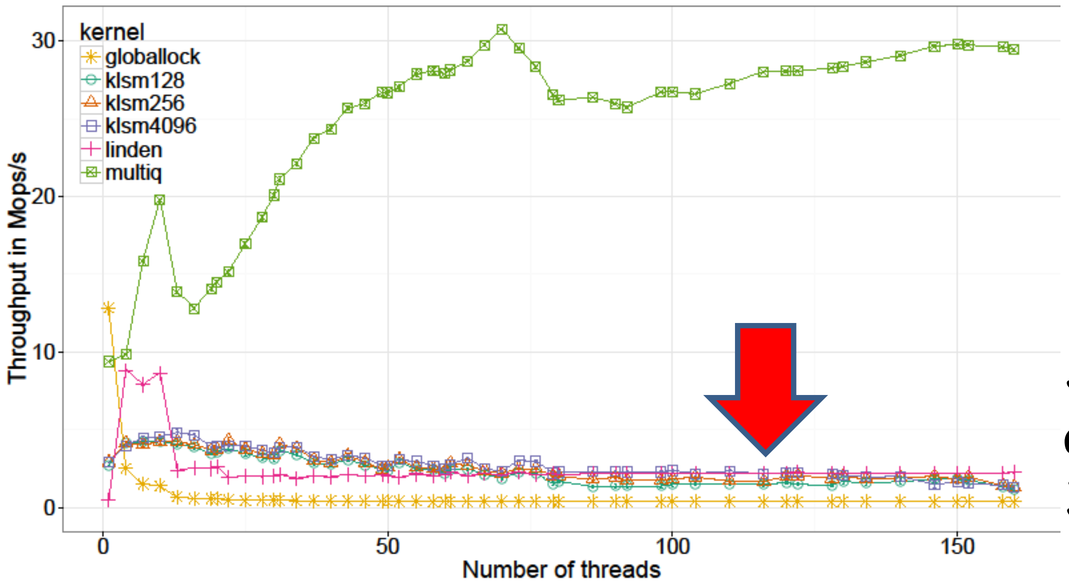
TUW "mars", 80-core Intel Xeon E7-8850, 2GHz; `gcc 5.2.1` with `-O3` and `-flto`; 30 repetitions. Also results on 48-core AMD, 64-core Sparc T5, 61-core Intel Xeon Phi



Descending dependent keys, same mix of insert and delete



Ascending dependent keys, same mix of insert and delete



Split mix of insert and delete, uniform 32-bit keys

- Many (most?) concurrent PQ's do not scale universally
- Locality important for performance/scalability
- Simple throughput benchmark can be very misleading
- Good to have: standard framework for PQ's