

**Lecture 17:**

# **Fine-grained synchronization & lock-free programming**

---

**Parallel Computer Architecture and Programming**  
**CMU 15-418/15-618, Fall 2024**

# Today's Topics

**Fine-grained Synchronization**

**Fine-grained Locking**

**Lock-free Programming**

# Locking Problem

**Locks can be big and expensive**

- **How many atomic operations does one lock require?**
- **How much data requires one lock?**

# Recall CUDA 7 atomic operations

```
int    atomicAdd(int* address, int val);
float  atomicAdd(float* address, float val);
int    atomicSub(int* address, int val);
int    atomicExch(int* address, int val);
float  atomicExch(float* address, float val);
int    atomicMin(int* address, int val);
int    atomicMax(int* address, int val);
unsigned int atomicInc(unsigned int* address, unsigned int val);
unsigned int atomicDec(unsigned int* address, unsigned int val);
int    atomicCAS(int* address, int compare, int val);
int    atomicAnd(int* address, int val); // bitwise
int    atomicOr(int* address, int val);  // bitwise
int    atomicXor(int* address, int val); // bitwise
```

**(omitting additional 64 bit and unsigned int versions)**

# **GCC Atomic Builtins**

**type \_\_sync\_fetch\_and\_add (type \*ptr, type value, ...)**

**type \_\_sync\_fetch\_and\_sub (type \*ptr, type value, ...)**

**type \_\_sync\_fetch\_and\_or (type \*ptr, type value, ...)**

**type \_\_sync\_fetch\_and\_and (type \*ptr, type value, ...)**

**type \_\_sync\_fetch\_and\_xor (type \*ptr, type value, ...)**

**type \_\_sync\_fetch\_and\_nand (type \*ptr, type value, ...)**

**type \_\_sync\_add\_and\_fetch (type \*ptr, type value, ...)**

**type \_\_sync\_sub\_and\_fetch (type \*ptr, type value, ...)**

**type \_\_sync\_or\_and\_fetch (type \*ptr, type value, ...)**

**type \_\_sync\_and\_and\_fetch (type \*ptr, type value, ...)**

**type \_\_sync\_xor\_and\_fetch (type \*ptr, type value, ...)**

**type \_\_sync\_nand\_and\_fetch (type \*ptr, type value, ...)**

**type can be (unsigned) char, short, int, or long**

# Implementing atomic fetch-and-op

```
// atomicCAS:  
// atomic compare and swap performs this logic atomically  
int atomicCAS(int* addr, int compare, int val) {  
    int old = *addr;  
    *addr = (old == compare) ? val : old;  
    return old;  
}
```

**Exercise: how can you build an atomic fetch+op out of atomicCAS()?**

- **try: atomic\_max()**

```
void atomic_max(int* addr, int x) {  
    int old = *addr;  
    int new = max(old, x);  
    while (atomicCAS(addr, old, new) != old) {  
        old = *addr;  
        new = max(old, x);  
    }  
}
```

**What about these operations?**

```
int atomic_increment(int* addr, int x);    // for signed values of x  
void lock(int* addr);
```

# C++ 11 `atomic<T>`

## Provides atomic read, write, read-modify-write of entire objects

- Atomicity may be implemented by mutex or efficiently by processor-supported atomic instructions (if T is a basic type)

## Provides memory ordering semantics for operations before and after atomic operations

- By default: sequential consistency
- See `std::memory_order` for more details

```
atomic<int> i;  
i++; // atomically increment i  
  
int a = i;  
// do stuff  
i.compare_exchange_strong(a, 10); // if i has same value as a, set i to 10  
bool b = i.is_lock_free();       // true if implementation of atomicity  
                                   // is lock free
```

**Will be useful if implementing the lock-free programming ideas in C++**

# How are the operations atomic?

## x86 Lock prefix

- **If the memory location is cached, then the cache retains that location until the operation completes**
- **If not:**
  - **On a bus, the processor uses the lock signal and holds the bus until the operation completes**
  - **On other designs, the processor (probably) NACKs any request for the cache line until the operation completes**

**N.B. Operations must be made on non-overlapping addresses**



# Locking more than one location

**Data structures are often larger than a single memory location**

- **How can an entire data structure be protected?**  
**E.g. 15213 Proxylab cache**

# Example: a sorted linked list

```
struct Node {
    int value;
    Node* next;
};
```

```
struct List {
    Node* head;
};
```

```
void insert(List* list, int value) {

    Node* n = new Node;
    n->value = value;

    // assume case of inserting before head of
    // of list is handled here (to keep slide simple)

    Node* prev = list->head;
    Node* cur = list->head->next;

    while (cur) {
        if (cur->value > value)
            break;

        prev = cur;
        cur = cur->next;
    }

    n->next = cur;
    prev->next = n;
}
```

## What can go wrong if multiple threads operate on the linked list simultaneously?

```
void delete(List* list, int value) {

    // assume case of deleting first element is
    // handled here (to keep slide simple)

    Node* prev = list->head;
    Node* cur = list->head->next;

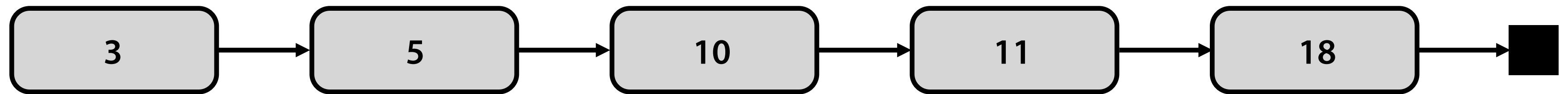
    while (cur) {
        if (cur->value == value) {
            prev->next = cur->next;
            delete cur;
            return;
        }

        prev = cur;
        cur = cur->next;
    }
}
```

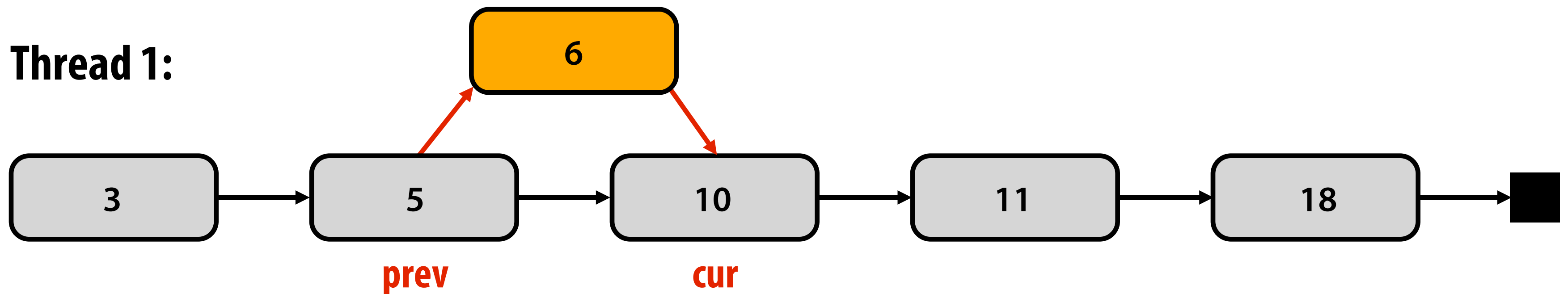
# Example: simultaneous insertion

Thread 1 attempts to insert 6

Thread 2 attempts to insert 7



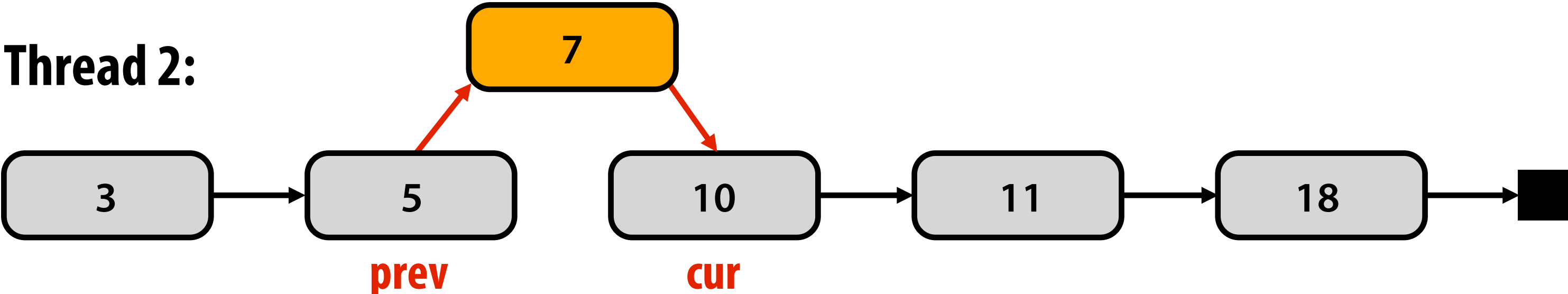
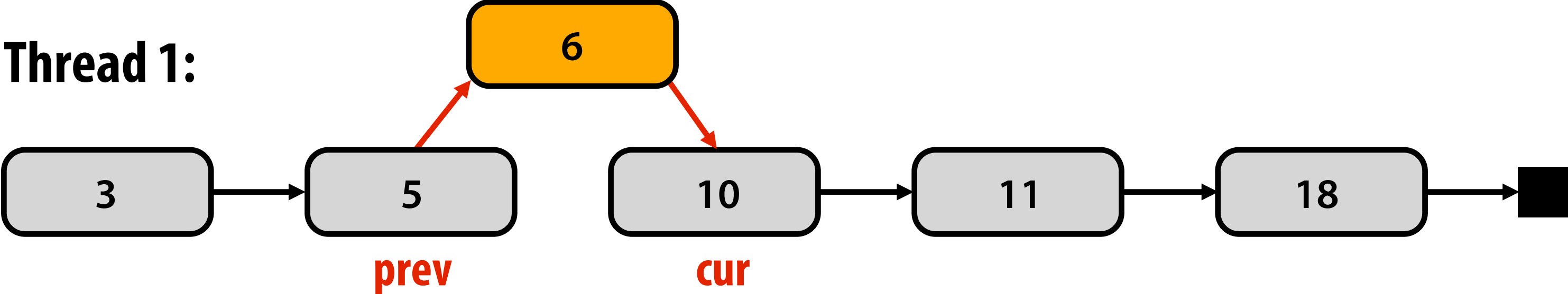
Thread 1:



# Example: simultaneous insertion

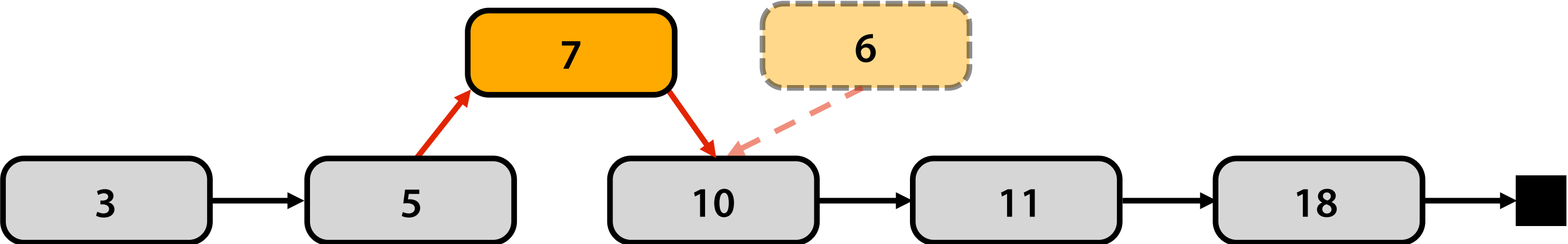
Thread 1 attempts to insert 6

Thread 2 attempts to insert 7



Thread 1 and thread 2 both compute same prev and cur.  
Result: one of the insertions gets lost!

**Result: (assuming thread 1 updates prev->next before thread 2)**



# Solution 1: protect the list with a single lock

```
struct Node {  
    int value;  
    Node* next;  
};
```

```
struct List {  
    Node* head;  
    Lock lock;  
};
```

← Per-list lock

```
void insert(List* list, int value) {  
  
    Node* n = new Node;  
    n->value = value;  
  
    lock(list->lock);  
  
    // assume case of inserting before head of  
    // of list is handled here (to keep slide simple)  
  
    Node* prev = list->head;  
    Node* cur = list->head->next;  
  
    while (cur) {  
        if (cur->value > value)  
            break;  
  
        prev = cur;  
        cur = cur->next;  
    }  
    n->next = cur;  
    prev->next = n;  
    unlock(list->lock);  
}
```

```
void delete(List* list, int value) {  
  
    lock(list->lock);  
  
    // assume case of deleting first element is  
    // handled here (to keep slide simple)  
  
    Node* prev = list->head;  
    Node* cur = list->head->next;  
  
    while (cur) {  
        if (cur->value == value) {  
            prev->next = cur->next;  
            delete cur;  
            unlock(list->lock);  
            return;  
        }  
  
        prev = cur;  
        cur = cur->next;  
    }  
    unlock(list->lock);  
}
```

# Single global lock per data structure

## Good:

- It is relatively simple to implement correct mutual exclusion for data structure operations (we just did it!)

## Bad:

- Operations on the data structure are serialized
- May limit parallel application performance

# Challenge: who can do better?

```
struct Node {
    int value;
    Node* next;
};
```

```
void insert(List* list, int value) {
```

```
    Node* n = new Node;
    n->value = value;
```

```
    // assume case of inserting before head of
    // of list is handled here (to keep slide simple)
```

```
    Node* prev = list->head;
    Node* cur = list->head->next;
```

```
    while (cur) {
        if (cur->value > value)
            break;
```

```
        prev = cur;
        cur = cur->next;
```

```
    }
```

```
    prev->next = n;
    n->next = cur;
```

```
}
```

```
struct List {
    Node* head;
};
```

```
void delete(List* list, int value) {
```

```
    // assume case of deleting first element is
    // handled here (to keep slide simple)
```

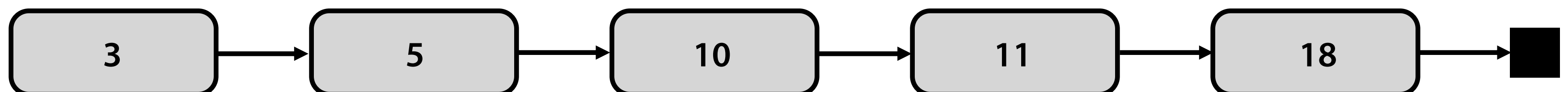
```
    Node* prev = list->head;
    Node* cur = list->head->next;
```

```
    while (cur) {
        if (cur->value == value) {
            prev->next = cur->next;
            delete cur;
            return;
        }
```

```
        prev = cur;
        cur = cur->next;
```

```
    }
```

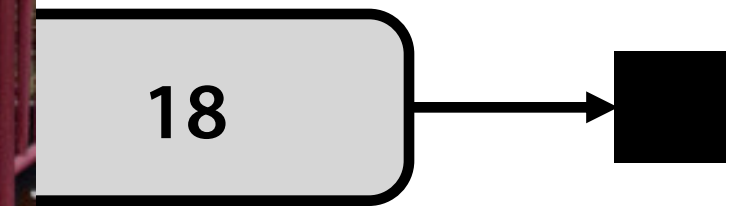
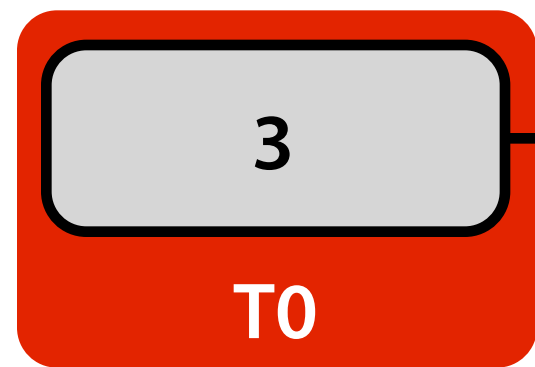
```
}
```





# Solution 2: "hand-over-hand" locking

Thread 0: delete(11)

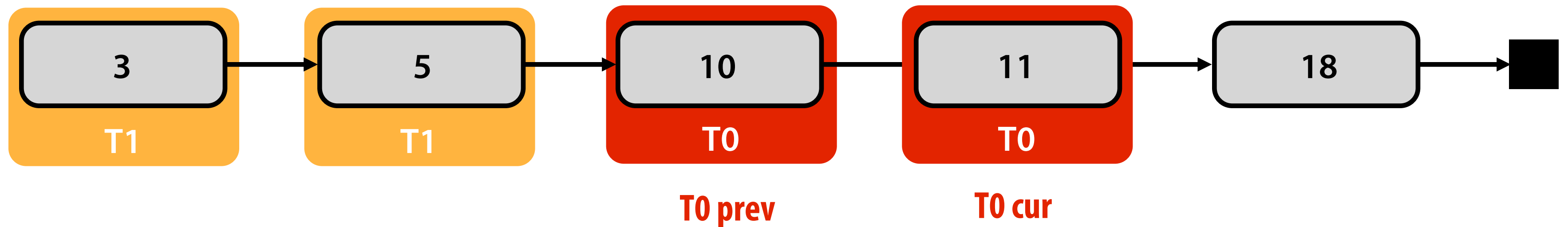




# Solution 2: "hand-over-hand" locking

Thread 0: delete(11)

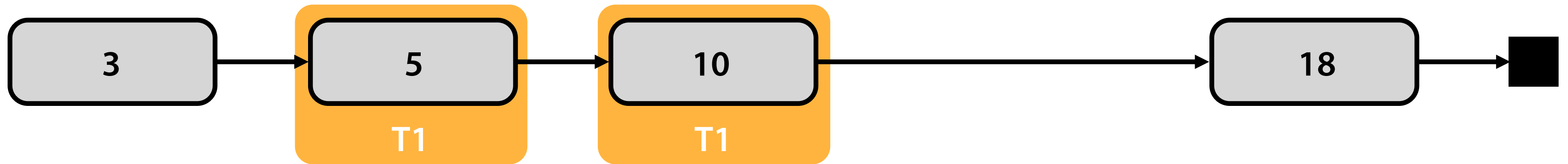
Thread 1: delete(10)



# Solution 2: "hand-over-hand" locking

Thread 0: delete(11)

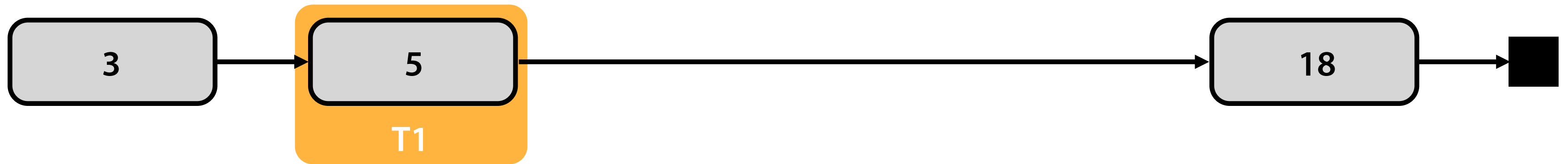
Thread 1: delete(10)



# Solution 2: "hand-over-hand" locking

Thread 0: delete(11)

Thread 1: delete(10)



# Solution 2: fine-grained locking

```
struct Node {
    int value;
    Node* next;
    Lock* lock;
};

struct List {
    Node* head;
    Lock* lock;
};

void insert(List* list, int value) {

    Node* n = new Node;
    n->value = value;

    // assume case of insert before head handled
    // here (to keep slide simple)

    Node* prev, *cur;

    lock(list->lock);
    prev = list->head;
    cur = list->head->next;

    lock(prev->lock);
    unlock(list->lock);
    if (cur) lock(cur->lock);

    while (cur) {
        if (cur->value > value)
            break;

        Node* old_prev = prev;
        prev = cur;
        cur = cur->next;
        unlock(old_prev->lock);
        if (cur) lock(cur->lock);
    }

    n->next = cur;
    prev->next = n;

    unlock(prev->lock);
    if (cur) unlock(cur->lock);
}
```

**Challenge to students: there is way to further improve the implementation of insert(). What is it?**

```
void delete(List* list, int value) {

    // assume case of delete head handled here
    // (to keep slide simple)

    Node* prev, *cur;

    lock(list->lock);
    prev = list->head;
    cur = list->head->next;

    lock(prev->lock);
    unlock(list->lock);
    if (cur) lock(cur->lock)

    while (cur) {
        if (cur->value == value) {
            prev->next = cur->next;
            unlock(prev->lock);
            unlock(cur->lock);
            delete cur;
            return;
        }

        Node* old_prev = prev;
        prev = cur;
        cur = cur->next;
        unlock(old_prev->lock);
        if (cur) lock(cur->lock);
    }
    unlock(prev->lock);
}
```

# Fine-grained locking

## Goal: enable parallelism in data structure operations

- Reduces contention for global data structure lock
- In previous linked-list example: a single monolithic lock is overly conservative (operations on different parts of the linked list can proceed in parallel)

## Challenge: tricky to ensure correctness

- Determining when mutual exclusion is required
- Deadlock? (how do you immediately know the earlier linked-list code is deadlock free?)
- Livelock?

## Costs?

- Overhead of taking a lock each traversal step (extra instructions + traversal now involves memory writes)
- Extra storage cost (a lock per node)
- What is a middle-ground solution that trades off some parallelism for reduced overhead? (hint: similar issue to selection of task granularity)

# Practice exercise

## Implement a fine-grained locking implementation of a binary search tree supporting insert and delete

```
struct Tree {  
    Node* root;  
};
```

```
struct Node {  
    int value;  
    Node* left;  
    Node* right;  
};
```

```
void insert(Tree* tree, int value);  
void delete(Tree* tree, int value);
```

# Lock-free data structures

# Blocking algorithms/data structures

**A blocking algorithm allows one thread to prevent other threads from completing operations on a shared data structure indefinitely**

## **Example:**

- Thread 0 takes a lock on a node in our linked list
- Thread 0 is swapped out by the OS, or crashes, or is just really slow (takes a page fault), etc.
- Now, no other threads can complete operations on the data structure (although thread 0 is not actively making progress modifying it)

**An algorithm that uses locks is blocking regardless of whether the lock implementation uses spinning or pre-emption**



# Lock-free algorithms

**Non-blocking algorithms are lock-free if some thread is guaranteed to make progress (“systemwide progress”)**

- In lock-free case, it is not possible to preempt one of the threads at an inopportune time and prevent progress by rest of system
- Note: this definition does not prevent starvation of any one thread

# Single reader, single writer bounded queue \*

```
struct Queue {
    int data[N];
    int head;    // head of queue
    int tail;    // next free element
};

void init(Queue* q) {
    q->head = q->tail = 0;
}

// return false if queue is full
bool push(Queue* q, int value) {
    // queue is full if tail is element before head
    if (q->tail == MOD_N(q->head - 1))
        return false;

    q->data[q->tail] = value;
    q->tail = MOD_N(q->tail + 1);
    return true;
}

// returns false if queue is empty
bool pop(Queue* q, int* value) {
    // if not empty
    if (q->head != q->tail) {
        *value = q->data[q->head];
        q->head = MOD_N(q->head + 1);
        return true;
    }
    return false;
}
```

- Only two threads (one producer, one consumer) accessing queue at the same time**  
**Threads never synchronize or wait on each other**
- **When queue is empty (pop fails), when it is full (push fails)**

\* Assume a sequentially consistent memory system for now  
(or the presence of appropriate memory fences, or C++ 11 atomic<>)

# Single reader, single writer unbounded queue \*

Source: Dr. Dobbs Journal

```
struct Node {
    Node* next;
    int value;
};

struct Queue {
    Node* head;
    Node* tail;
    Node* reclaim;
};

void init(Queue* q) {
    q->head = q->tail = q->reclaim = new Node;
}
```

```
void push(Queue* q, int value) {

    Node* n = new Node;
    n->next = NULL;
    n->value = value;

    q->tail->next = n;
    q->tail = q->tail->next;

    while (q->reclaim != q->head) {
        Node* tmp = q->reclaim;
        q->reclaim = q->reclaim->next;
        delete tmp;
    }
}

// returns false if queue is empty
bool pop(Queue* q, int* value) {

    if (q->head != q->tail) {
        *value = q->head->next->value;
        q->head = q->head->next;
        return true;
    }
    return false;
}
```

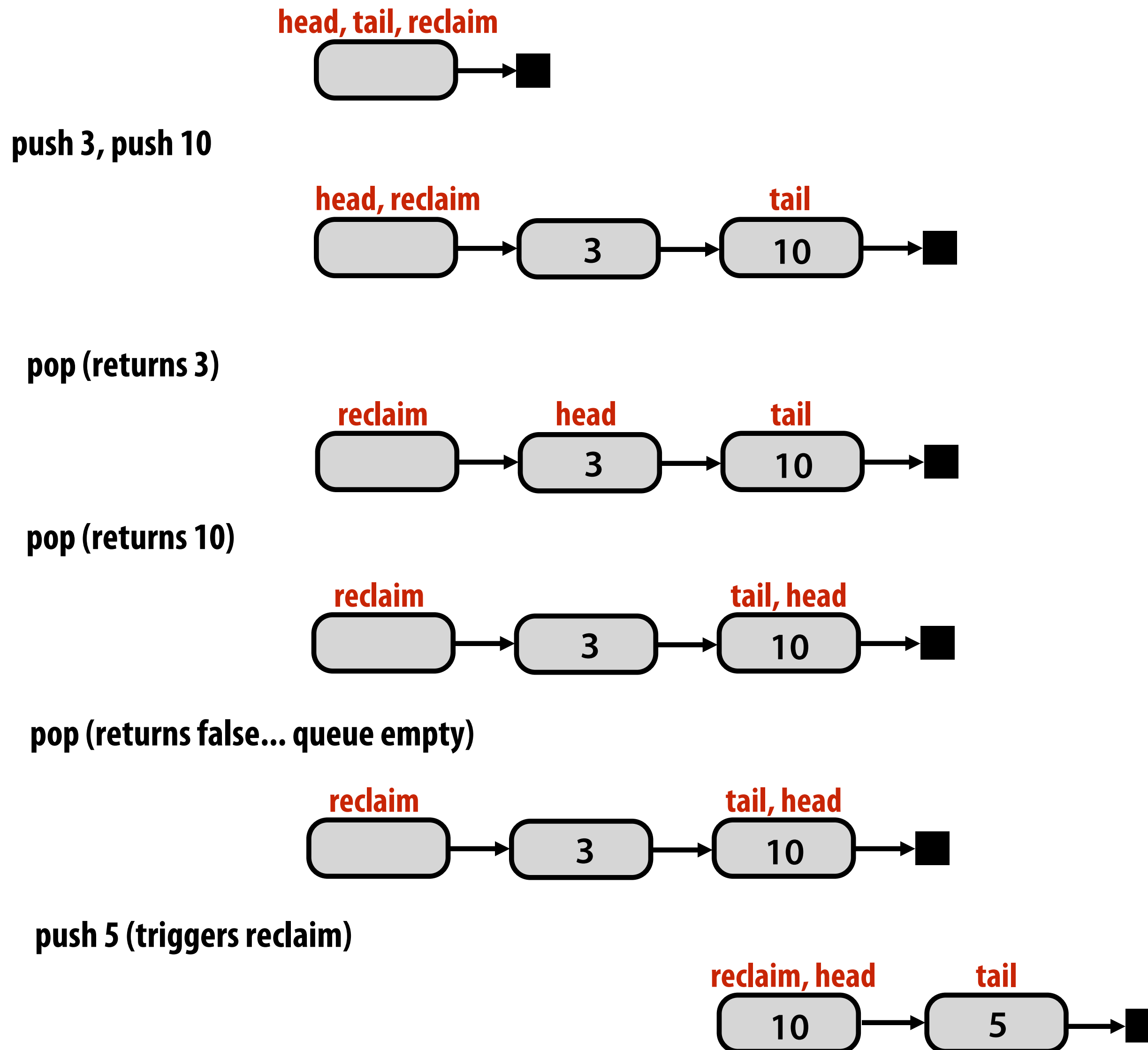
**Tail points to last element added**

**Head points to element BEFORE head of queue**

**Allocation and deletion performed by the same thread (producer)**

\* Assume a sequentially consistent memory system for now  
(or the presence of appropriate memory fences, or C++ 11 atomic<>)

# Single reader, single writer unbounded queue



# Lock-free stack (first try)

```
struct Node {
    Node* next;
    int value;
};

struct Stack {
    Node* top;
};

void init(Stack* s) {
    s->top = NULL;
}

void push(Stack* s, Node* n) {
    while (1) {
        Node* old_top = s->top;
        n->next = old_top;
        if (compare_and_swap(&s->top, old_top, n) == old_top)
            return;
    }
}

Node* pop(Stack* s) {
    while (1) {
        Node* old_top = s->top;
        if (old_top == NULL)
            return NULL;
        Node* new_top = old_top->next;
        if (compare_and_swap(&s->top, old_top, new_top) == old_top)
            return old_top;
    }
}
```

**Main idea: as long as no other thread has modified the stack, a thread's modification can proceed. Note difference from fine-grained locks example earlier: before, implementation locked a part of a data-structure for fine-grained access. Here, threads do not hold lock on data-structure at all.**

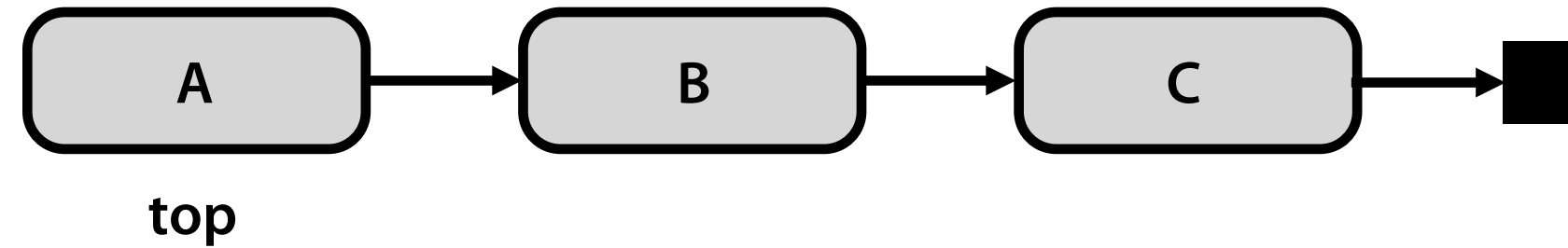
\* Assume a sequentially consistent memory system for now  
(or the presence of appropriate memory fences, or C++ 11 atomic<>)

# The ABA problem

A, B, C, and D are stack node addresses.

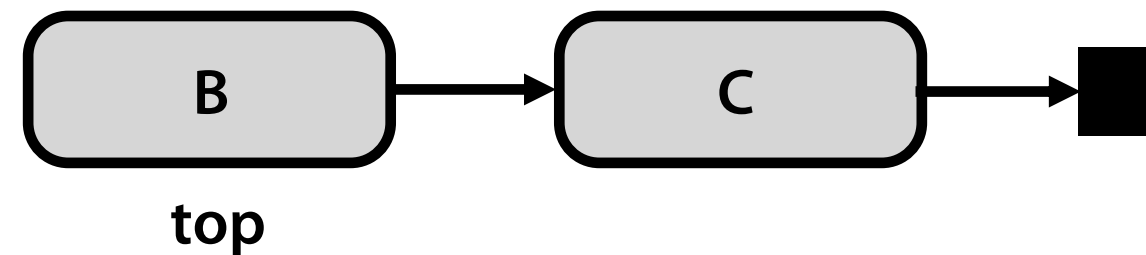
Thread 0

Thread 1

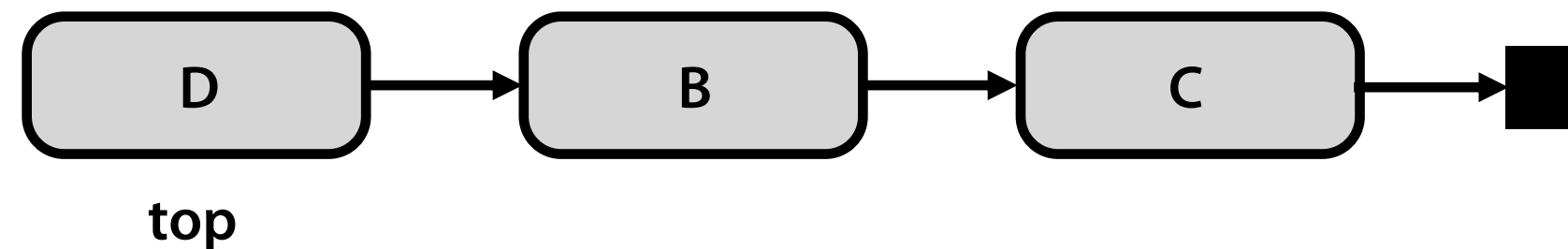


begin pop() (local variable: old\_top = A, new\_top = B)

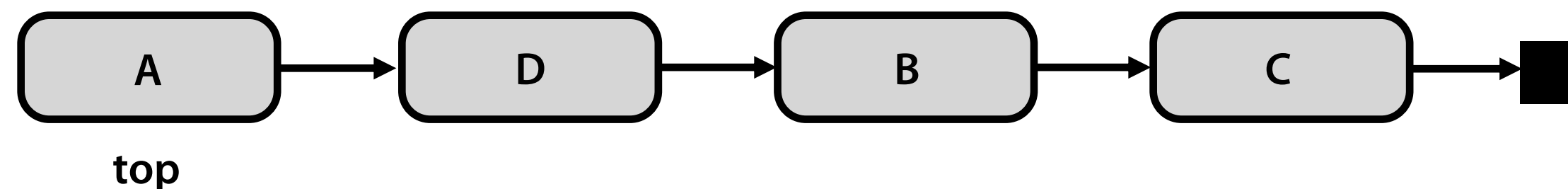
begin pop() (local variable old\_top == A)  
complete pop() (returns A)



begin push(D)  
complete push(D)

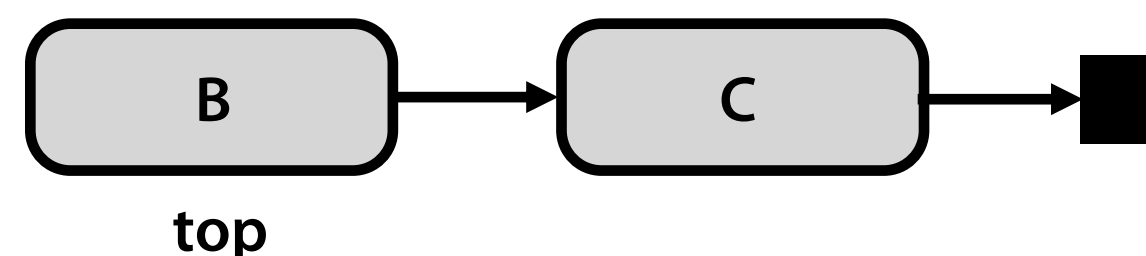


modify node A: e.g., set value = 42  
begin push(A)  
complete push(A)



CAS succeeds (sets top to B!)  
complete pop() (returns A)

**Stack structure is corrupted! (lost D)**



time

# Lock-free stack using counter for ABA soln

```
struct Node {
    Node* next;
    int value;
};


struct Stack {
    Node* top;
    int pop_count;
};

void init(Stack* s) {
    s->top = NULL;
}

void push(Stack* s, Node* n) {
    while (1) {
        Node* old_top = s->top;
        n->next = old_top;
        if (compare_and_swap(&s->top, old_top, n) == old_top)
            return;
    }
}

Node* pop(Stack* s) {
    while (1) {
        int pop_count = s->pop_count;
        Node* top = s->top;
        if (top == NULL)
            return NULL;
        Node* new_top = top->next;
        if (double_compare_and_swap(&s->top, top, new_top,
                                   &s->pop_count, pop_count, pop_count+1))
            return top;
    }
}
```

test to see if either have changed (in this example: return true if no changes)



**Maintain counter of pop operations**

**Requires machine to support “double compare and swap” (DCAS) or doubleword CAS**

**Could also solve ABA problem with node allocation and/or element reuse policies**

# Compare and swap on x86

## **x86 supports a “wide” compare-and-swap instruction**

- **Not quite the “double compare-and-swap” used in the code on the previous slide**
- **But could simply ensure the stack’s count and top fields are contiguous in memory to use the 64-bit wide single compare-and-swap instruction below.**

## **cmpxchg8b**

- **“compare and exchange eight bytes”**
- **Can be used for compare-and-swap of two 32-bit values**

## **cmpxchg16b**

- **“compare and exchange 16 bytes”**
- **Can be used for compare-and-swap of two 64-bit values**



# Another problem: referencing freed memory

```
struct Node {
    Node* next;
    int value;
};

struct Stack {
    Node* top;
    int pop_count;
};

void init(Stack* s) {
    s->top = NULL;
}

void push(Stack* s, int value) {
    Node* n = new Node;
    n->value = value;
    while (1) {
        Node* old_top = s->top;
        n->next = old_top;
        if (compare_and_swap(&s->top, old_top, n) == old_top)
            return;
    }
}

int pop(Stack* s) {
    while (1) {
        Stack old;
        old.pop_count = s->pop_count;
        old.top = s->top;

        if (old.top == NULL)
            return NULL;

        Stack new_stack;
        new_stack.top = old.top->next;
        new_stack.pop_count = old.pop_count+1;

        if (doubleword_compare_and_swap(&s, &old, new_stack))
            int value = top->value;
            delete top;
            return value;
    }
}
```

top might have been freed at this point  
by the thread that popped it.



# Hazard pointer: avoid freeing nodes until its determined all other threads do not hold reference to node

```
struct Node {
    Node* next;
    int value;
};

struct Stack {
    Node* top;
    int pop_count;
};

// per thread ptr (node that cannot
// be deleted since the thread is
// accessing it)
Node* hazard;

// per-thread list of nodes thread
// must delete
Node* retireList;
int  retireListSize;

// delete nodes if possible
void retire(Node* ptr) {
    push(retireList, ptr);
    retireListSize++;

    if (retireListSize > THRESHOLD)
        for (each node n in retireList) {
            if (n not pointed to by any
                thread's hazard pointer) {
                remove n from list
                delete n;
            }
        }
}
```

```
void init(Stack* s) {
    s->top = NULL;
}

void push(Stack* s, int value) {
    Node* n = new Node;
    n->value = value;
    while (1) {
        Node* old_top = s->top;
        n->next = old_top;
        if (compare_and_swap(&s->top, old_top, n) == old_top)
            return;
    }
}

int pop(Stack* s) {
    while (1) {
        Stack old;
        old.pop_count = s->pop_count;
        old.top = s->top;

        if (old.top == NULL) return NULL;

        hazard = old.top;
        Stack new_stack;
        new_stack.top = old.top->next;
        new_stack.pop_count = old.pop_count+1;

        if (doubleword_compare_and_swap(&s, &old, new_stack))
        {
            int value = old.top->value;
            retire(old.top);
            return value;
        }
        hazard = NULL;
    }
}
```

# Lock-free linked list insertion \*

```
struct Node {
    int value;
    Node* next;
};

struct List {
    Node* head;
};

// insert new node after specified node
void insert_after(List* list, Node* after, int value) {

    Node* n = new Node;
    n->value = value;

    // assume case of insert into empty list handled
    // here (keep code on slide simple for class discussion)

    Node* prev = list->head;

    while (prev->next) {
        if (prev == after) {
            while (1) {
                Node* old_next = prev->next;
                n->next = old_next;
                if (compare_and_swap(&prev->next, old_next, n) == old_next)
                    return;
            }
        }

        prev = prev->next;
    }
}
```

**Compared to fine-grained  
locking implementation:**

**No overhead of taking locks  
No per-node storage overhead**

\* For simplicity, this slide assumes the \*only\* operation on the list is insert

# Lock-free linked list deletion

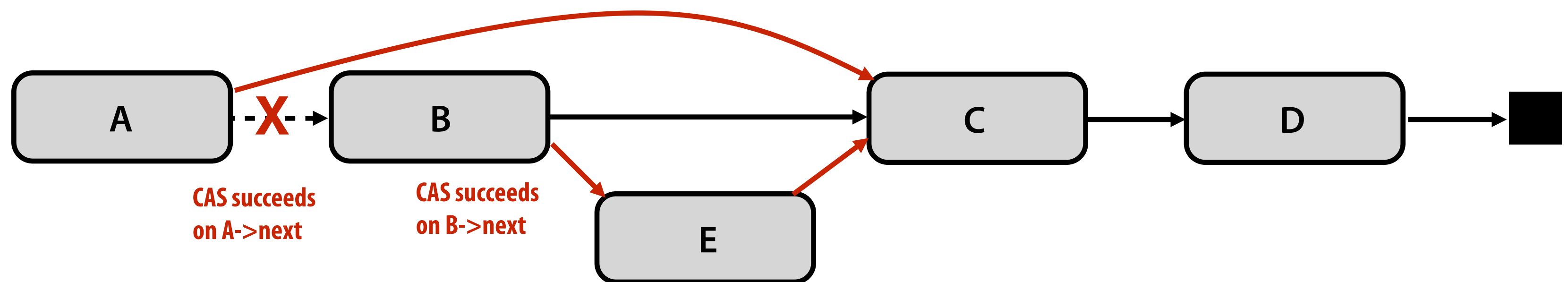
Supporting lock-free deletion significantly complicates data-structure

Consider case where B is deleted simultaneously with successful insertion of E after B.

B now points to E, but B is not in the list!

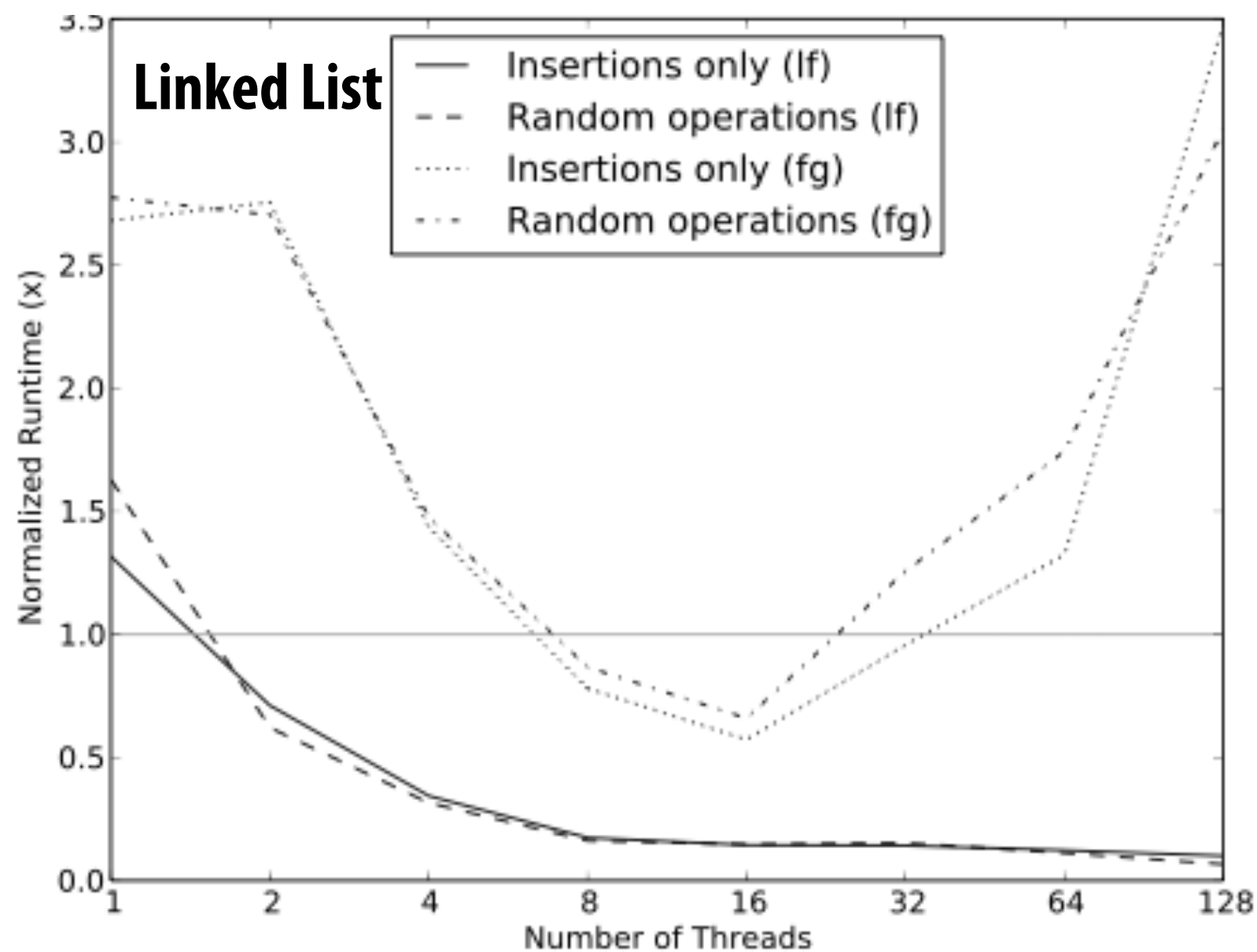
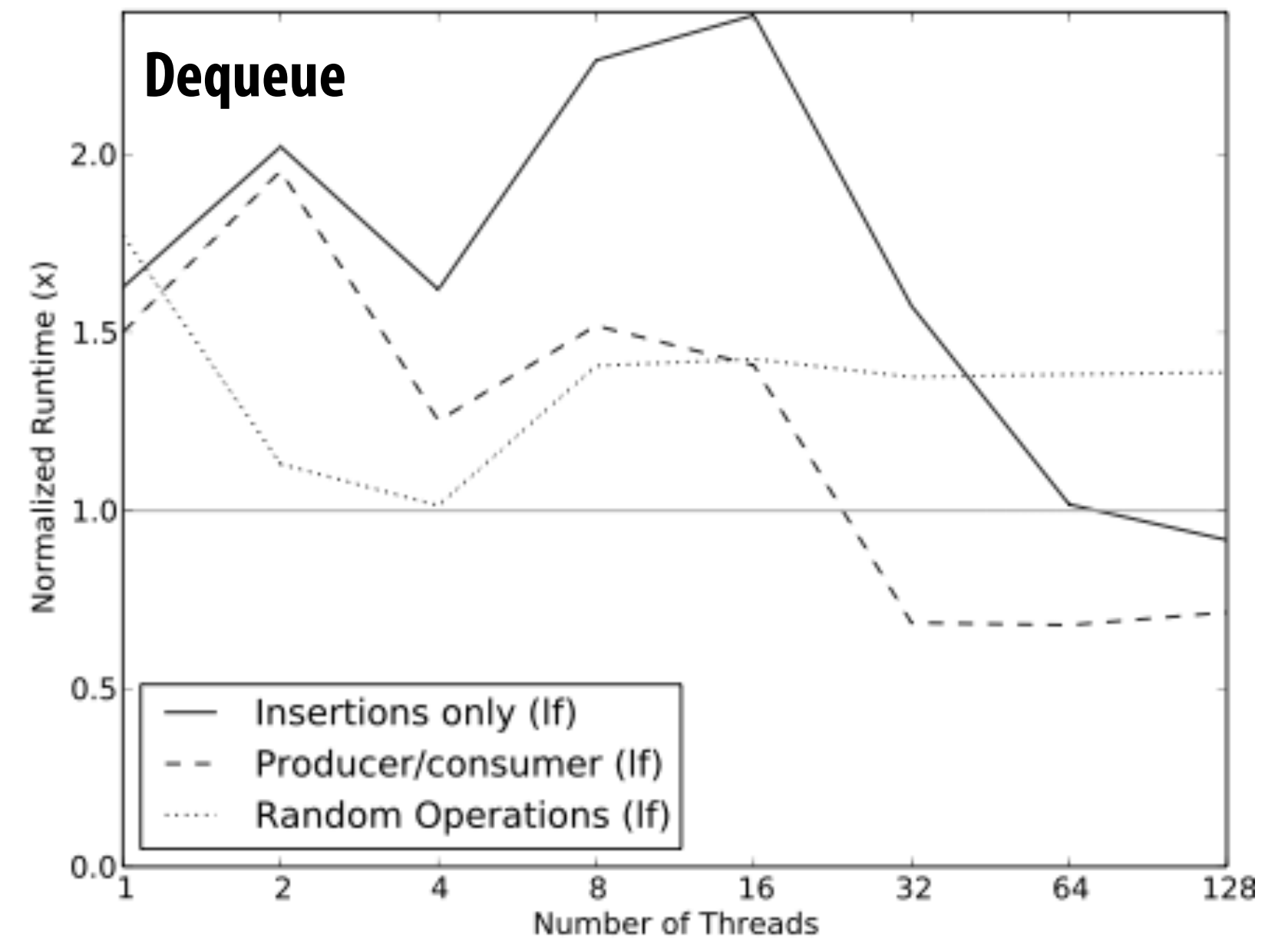
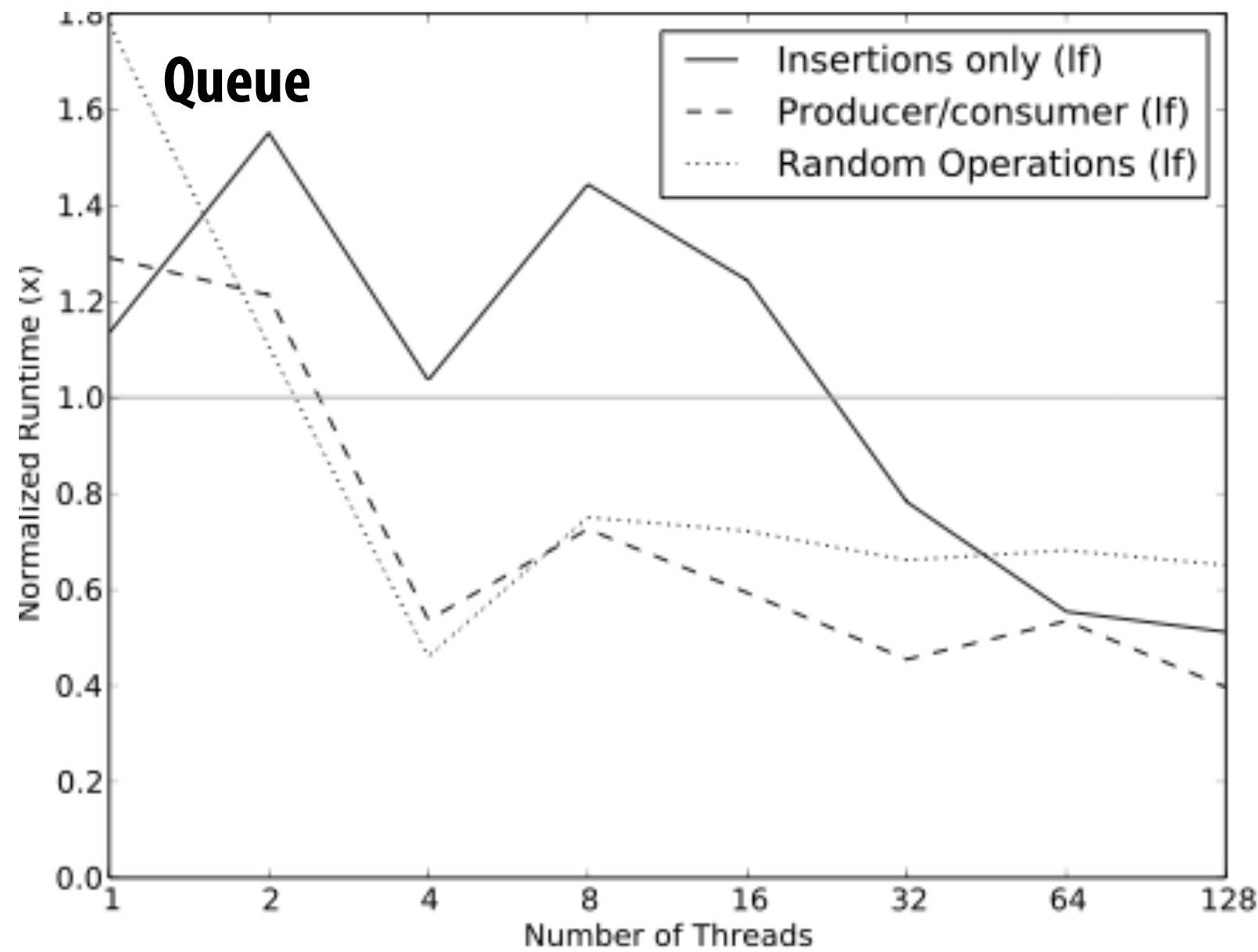
For the curious:

- Harris 2001. A Pragmatic Implementation of Non-blocking Linked-Lists
- Fomitchev 2004. Lock-free linked lists and skip lists



# Lock-free vs. locks performance comparison

Lock-free algorithm run time normalized to run time of using pthread mutex locks



If = "lock free"  
fg = "fine grained lock"

Source: Hunt 2011. Characterizing the Performance and Energy Efficiency of Lock-Free Data Structures

# **In practice: why lock free data-structures?**

**When optimizing parallel programs in this class you often assume that only your program is using the machine**

- **Because you care about performance**
- **Typical assumption in scientific computing, graphics, data analytics, etc.**

**In these cases, well written code with locks can be as fast (or faster) than lock-free code**

**But there are situations where code with locks can suffer from tricky performance problems**

- **Multi-programmed situations where page faults, pre-emption, etc. can occur while thread is in a critical section**
- **Creates problems like priority inversion, convoying, crashing in critical section, etc. that are often discussed in OS classes**

# Summary

**Use fine-grained locking to reduce contention (maximize parallelism) in operations on shared data structures**

- But fine-granularity can increase code complexity (errors) and increase execution overhead

**Lock-free data structures: non-blocking solution to avoid overheads due to locks**

- But can be tricky to implement (ensuring correctness in a lock-free setting has its own overheads)
- Still requires appropriate memory fences on modern relaxed consistency hardware

**Note: a lock-free design does not eliminate contention**

- Compare-and-swap can fail under heavy contention, requiring spins

# More reading

**Michael and Scott 1996. Simple, Fast and Practical Non-Blocking and Blocking Concurrent Queue Algorithms**

- **Multiple reader/writer lock-free queue**

**Harris 2001. A Pragmatic Implementation of Non-Blocking Linked-Lists**

**Many good blog posts and articles on the web:**

- **<http://www.drdobbs.com/cpp/lock-free-code-a-false-sense-of-security/210600279>**
- **<http://developers.memsql.com/blog/common-pitfalls-in-writing-lock-free-algorithms/>**

**Often students like to implement lock-free data structures for projects**

- **Linked list, skip-list based maps (Java's ConcurrentSkipListMap), list-based sets, etc.**