

# Concurrent List-Based Sets

fine-grained, optimistic and lazy synchronization

SLR206, P1

# Implementing a scalable concurrent data structure?

- What *is* a concurrent data structure?
  - ✓ Sequential type
  - ✓ Wait-free
  - ✓ Linearizable
- What is scalable?
  - ✓ **Throughput**: the number of complete operations per time unit
  - ✓ **Workload**: concurrent operations applied
  - ✓ Throughput scales with the growing **workload** (ideally)
- Typically, better **concurrency** translates to better performance
  - ✓ The “number” of accepted concurrent **schedules**

동시성

# Example: **set** type

A **set** abstraction stores a set of integers (no duplicates) and exports operations:

- `insert(x)` – adds `x` to the set and returns true if and only if `x` is not in the set
- `remove(x)` – removes `x` from the set and returns true if and only if `x` is in the set
- `contains(x)` – returns true if and only if `x` is in the set

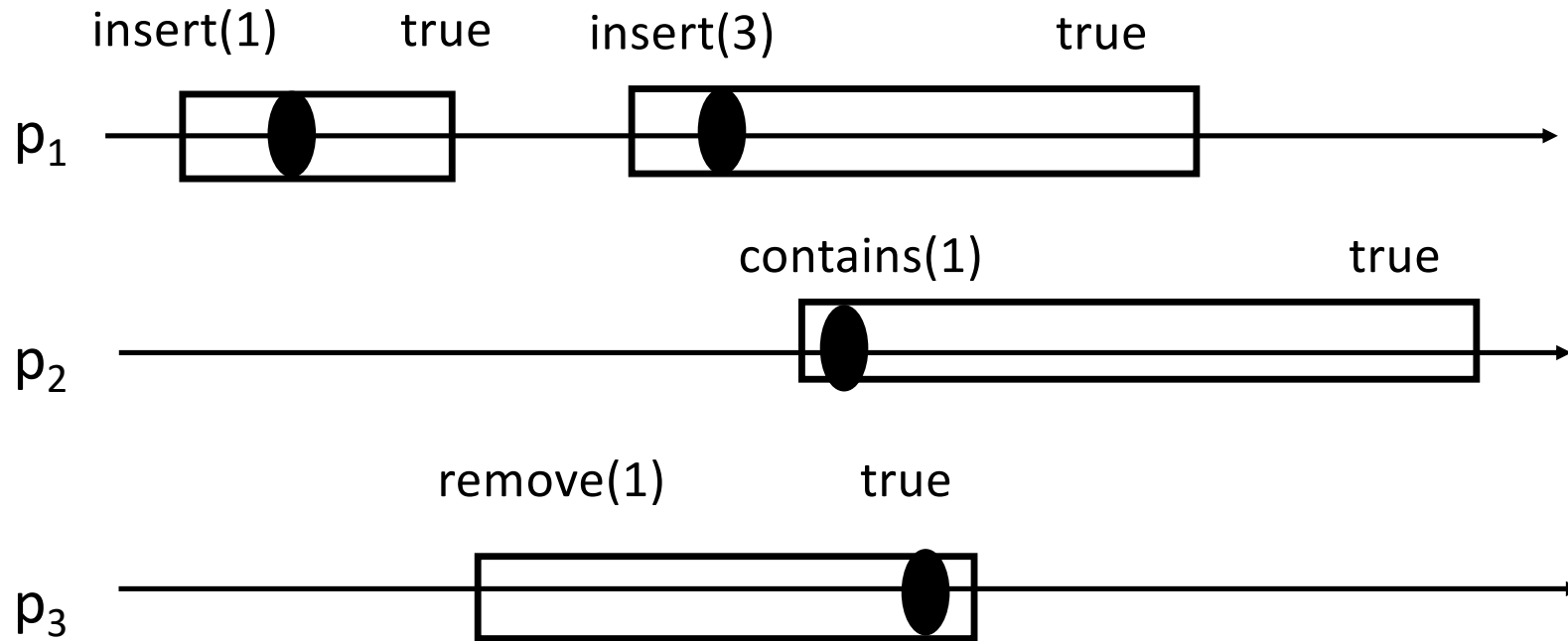
# Sequential list-based set

Implementing a set using a **sorted linked list**:

- To **locate**  $x$ , search starting from the head  $curr$  points to the first node storing  $x' \geq x$ ,  $prev$  points to its predecessor
- To **remove**  $x$  (if  $x' = x$ ), point  $prev.next$  to  $curr.next$
- To **insert**  $x$  (if  $x' > x$ ), set  $prev.next$  to the new node storing  $x$  and pointing to  $curr$



# Linearizable histories



The history is **equivalent** to a **legal sequential** history on a set (**real-time order** preserved)

# Linked-list for Set: sequential implementation

```
/* The node of an integer list. At creation, default pointer
   is null */
public class Node{
    Node(int item){key=item;next=null;}
    public int key;
    public Node next;}

public class SetList{

    private Node head;

    public SetList(){
        head = new Node(Integer.MIN_VALUE);
        head.next = new Node(Integer.MAX_VALUE);
    }
}
```

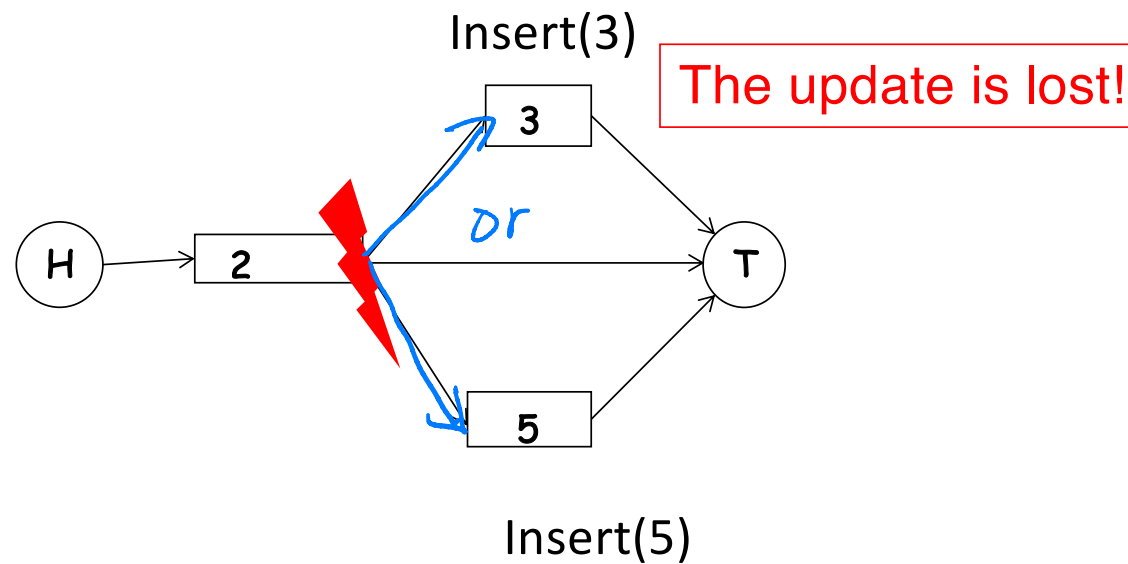
# Linked-list for Set: sequential implementation

```
public boolean insert(int item){
    Node pred=head;
    Node curr=head.next;
    while (curr.key < item){
        pred = curr;
        curr = pred.next;}
    if (curr.key==item)
        {return false;}
    else{
        Node node = new
        Node(item);
        node.next=curr;
        pred.next=node;
        return true;
    } }
```

```
public boolean contains(int item){
    Node pred=head;
    Node curr=head.next;
    while (curr.key < item){
        pred = curr;
        curr = pred.next;}
    if (curr.key==item)
        {return true;}
    else {return false;}}
```

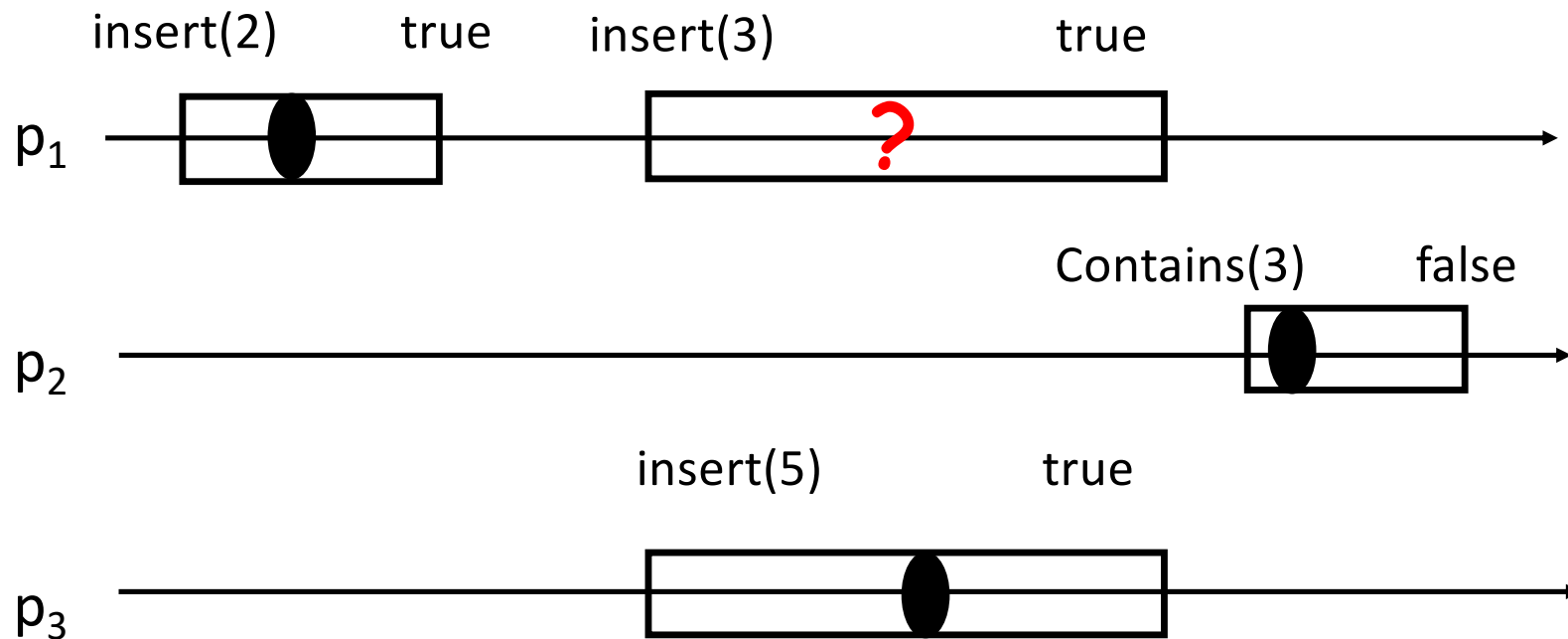
```
public boolean remove(int item){
    Node pred=head;
    Node curr=head.next;
    while (curr.key < item){
        pred = curr;
        curr = pred.next;}
    if (curr.key==item)
        {pred.next=curr.next;
        return true;}
    else {return false;}}
```

As is?



The extension with contains(3)  
is not linearizable!





Need to **protect** the list elements:  
locks, transactional memory...

# Concurrent reasoning?

- How to show that an implementation is correct (linearizable)?
- Invariants: *true* initially, no transition can render it *false*
  - ✓ E.g., the **object representation** “makes sense”
- (Sorted) list-based sets:
  - ✓ *head* and *tail* are **sentinels**
  - ✓ nodes are **sorted** and keys are **unique**
  - ✓ (the structure can be produced **sequentially**)

# Progress guarantees?

- Locks are used to protect list elements (assuming **cooperation**):
  - ✓ Deadlock-freedom: at least one process makes progress (completes all its operations)
  - ✓ Starvation-freedom: every process makes progress
- Nonblocking approaches:
  - ✓ Wait-free: every operation completes in a finite number of steps
  - ✓ Lock-free: some operation completes in a finite number of steps

use 1 lock

## Coarse grained solution

```
public class CoarseList{

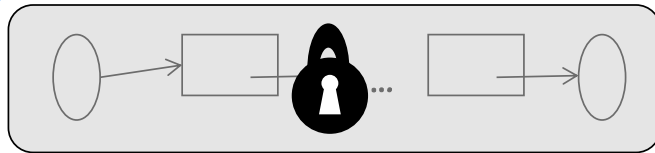
    private Node head;
    private Lock lock = new ReentrantLock();

    public boolean insert(int item){
        lock.lock();
        Node pred=head;
        try {
            Node curr=head.next;
            while (curr.key < item){
                pred = curr;
                curr = pred.next;
            }
            if (curr.key==item){return false;}
            Node node = new Node(item);
            node.next=curr;
            pred.next=node;
            return true;
        } finally{
            lock.unlock();
        }
    }
}
```

- Same progress guarantees as lock
  - ✓ ReentrantLock – starvation-free
- Good for low contention
- Sub-optimal for moderate to high contention: operations run sequentially

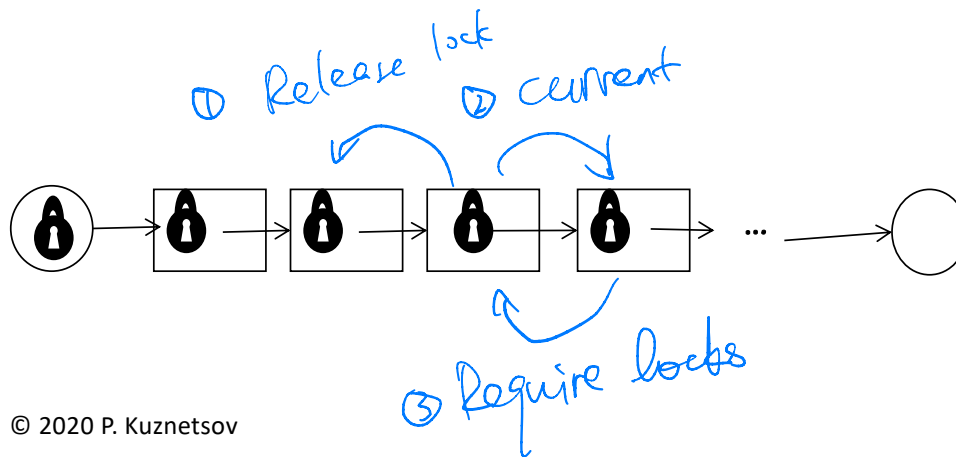
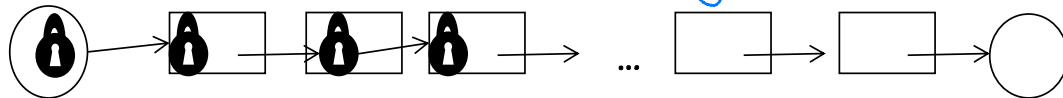
# Locking schemes for a linked-list

link-base



Coarse-grained locking

2-phase locking



Hand-over-hand locking

# Fine-grained solution: hand-over-hand

```
public boolean insert(int item)
    head.lock();
    Node pred=head;
    Node curr=pred.next;
    try {
        curr.lock();
        try {
            while (curr.key < item){
                pred.unlock();
                pred = curr;
                curr = pred.next;
                curr.lock()
            }
            if (curr.key==item){
                return false;}
            Node node = new Node(item);
            node.next=curr;
            pred.next=node;
            return true;
        } finally{
            curr.unlock();
        }
    } finally{
        pred.unlock();
    }
}
```

```
public boolean remove(int item){
    head.lock();
    Node pred=head;
    Node curr=pred.next;
    try {
        curr.lock();
        try {
            while (curr.key < item){
                pred.unlock();
                pred = curr;
                curr = pred.next;
                curr.lock()
            }
            if (curr.key==item){
                pred.next=curr.next;
                return true;}
            return false;
        } finally{
            curr.unlock();
        }
    } finally{
        pred.unlock();
    }
}
```

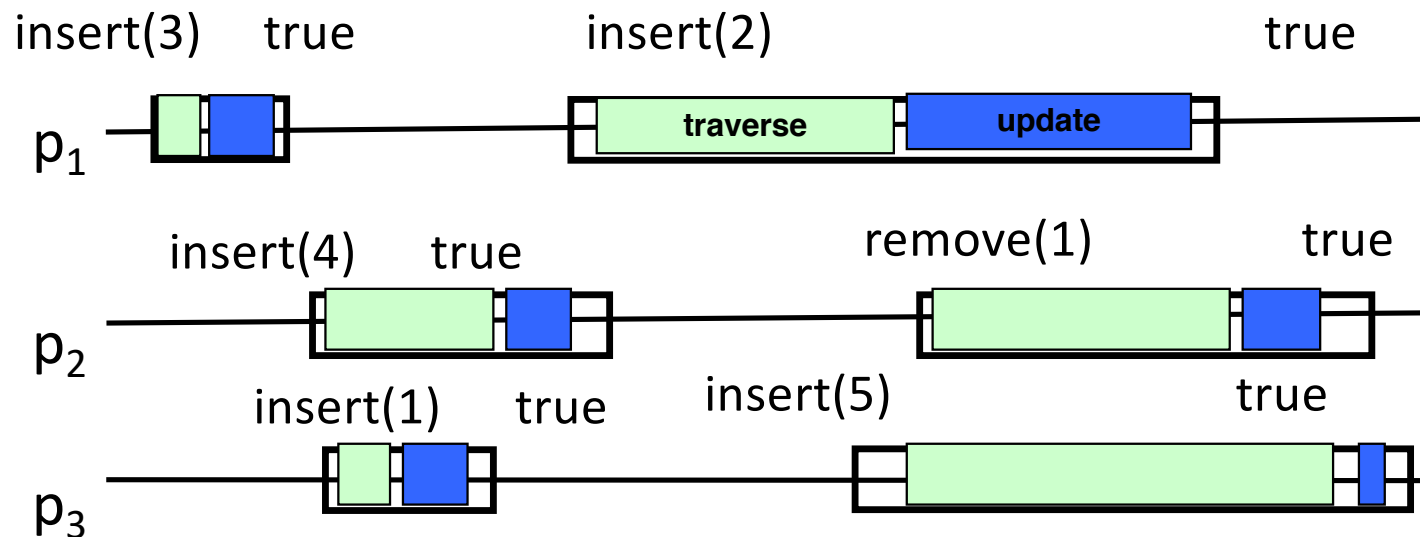
# Hand-over-hand: concurrency limitations

```
public boolean contains(int item){
    head.lock();
    Node pred=head;
    Node curr=pred.next;
    try {
        curr.lock();
        try {
            while (curr.key < item){
                pred.unlock();
                pred = curr;
                curr = pred.next;
                curr.lock()
            }
            return (curr.key==item);
        } finally{
            curr.unlock();
        }
    } finally{
        pred.unlock();
    }
}
```

- More concurrency:
  - ✓ An operation working on a “high” node does not obstruct ones working on “low” nodes

# Hand-over-hand: linearization

- Every complete operation is linearized within the critical section (between **locks** and **unlocks**)
- No update concerning **pred or any subsequent node** concurrently occurs: pred remains reachable as long as it is locked





# Hand-over-hand: progress

- **Starvation-freedom** (assuming starvation-free locks)
  - ✓ Operations acquire locks in **the order of growing items**: no deadlock possible
  - ✓ Every lock acquisition eventually completes
  - ✓ Traverse for item eventually reaches a node with  $\text{item}' \geq \text{item}$
  - ✓ **Why?**
- **But!** Operations concerning disjoint nodes may obstruct each other
  - ✓ E.g. `insert(2)` obstructs `insert(5)`, when applied to `{3,4}`
- **Optimistic** algorithm?
  - ✓ No locks on the traverse path

## Quiz 2.1: hand-over-hand

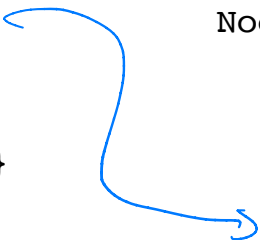
- Check if **contains** requires locking
  - ✓ What if **contains** traversed the list without lock acquisition?
- What if traverse (in remove, insert) checks the value in **curr** before locking it (only holds lock on **pred** when traverse terminates)?
- Can we just use one lock at a time? *No*
- Prove starvation-freedom (assuming starvation-free locks)
  - ✓ Can an operation be blocked (delayed forever) by infinitely many concurrent inserts?

# Optimistic: wait-free traversal plus validation

```
private boolean validate(Node pred, Node
curr) {
    Node node=head;
    while (node.key <= pred.key){
        if (node==pred){
            return pred.next==curr;}
        node=node.next;
    }
    return false;
}
```

Validation necessary for  
updates?

```
public boolean remove(int item)
    while (true){
        Node pred=head;
        Node curr=pred.next;
        while (curr.key<item){
            pred=curr;
            curr=curr.next;
        }
        pred.lock(); curr.lock();
        try {
            if
            (validate(pred,curr)){
                if (curr.key==item) {
                    pred.next=curr.next;
                    return true;
                }
                return false; }
        } finally{
            pred.unlock();
            curr.unlock();
        }
    }
```



# Optimistic: wait-free traversal plus validation

```
public boolean insert(int item){
    while (true){
        Node pred=head;
        Node curr=pred.next;
        while (curr.key<item){
            pred=curr;
            curr=curr.next;
        }
        pred.lock(); curr.lock();
        try {
            if (validate(pred,curr)){
                if (curr.key==item) {
                    return false;
                }
                Node node = new Node(item);
                node.next=curr;
                pred.next=node;
                return true; }
        } finally{
            pred.unlock();
            curr.unlock();}
    }
}
```

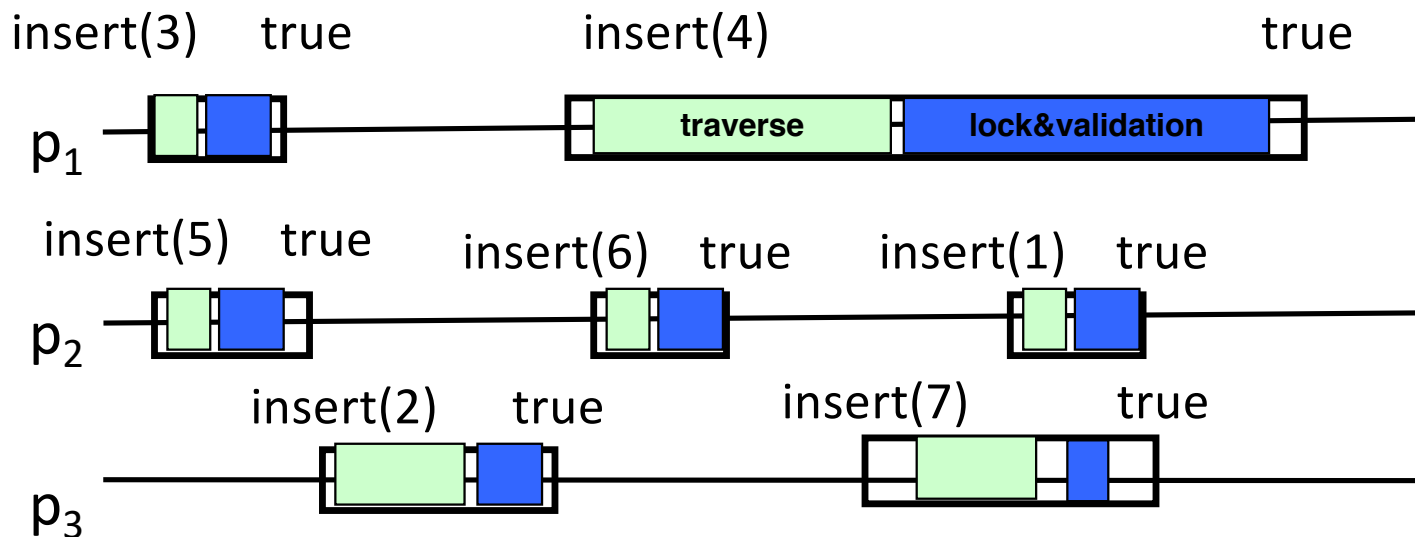
© 2020 P. Kuznetsov

```
public boolean contains(int item) {
    while (true){
        Node pred=head;
        Node curr=pred.next;
        while (curr.key<item){
            pred=curr;
            curr=curr.next;
        }
        pred.lock(); curr.lock();
        try {
            if (validate(pred,curr)){
                return (curr.key==item);
            }
        } finally{
            pred.unlock();
            curr.unlock();}
    }
}
```

- contains grabs locks
- updates re-traverse even if no contention.

# Optimistic: linearization

- Every complete operation is linearized within the critical section (between **locks** and **unlocks**)
- No update concerning **pred** and **curr** can take place concurrently
- And validation in the CS ensures that **pred**->**curr** are still reachable (possibly via a new path)



## Quiz 2.2: optimistic

- Show that validation is **necessary** for updates
  - ✓ Hint: consider an algorithm without validation and show that an update can get **lost** because of a series of concurrent removes
- Is validation necessary for contains?
- Show that the algorithm is **not** starvation-free (even if all locks are)

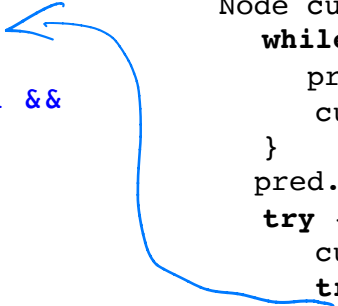
# Lazy synchronization: logical removals and wait-free contains

```
private boolean validate(Node pred, Node
    curr) {

    return !pred.marked && !curr.marked &&
        pred.next==curr;
}
```

- remove first marks the node for deletion and then physically removes it
- contains returns true iff the node is reachable and not marked
- A node is in the set iff it is an unmarked reachable node

```
public boolean remove(int item)
    while (true){
        Node pred=head;
        Node curr=pred.next;
        while (curr.key<item){
            pred=curr;
            curr=curr.next;
        }
        pred.lock();
        try {
            curr.lock();
            try {
                if (validate(pred,curr)){
                    if (curr.key!=item){
                        return false;}
                    curr.marked=true;
                    pred.next=curr.next;
                    return true; }
            } finally{
                curr.unlock(); }
        } finally{
            pred.unlock();}
    }
}
```



# Lazy synchronization: wait-free contains

```
public boolean insert(int item){
    while (true){
        Node pred=head;
        Node curr=pred.next;
        while (curr.key<item){
            pred=curr;
            curr=curr.next;
        }
        pred.lock();
        try {
            curr.lock();
            try {
                if (validate(pred,curr)){
                    if (curr.key==item) {
                        return false;
                    }
                }
                Node node = new Node(item);
                node.next=curr;
                pred.next=node;
                return true; }
            } finally{
                curr.unlock(); }
        } finally{
            pred.unlock();}
    }
}
```

```
public boolean contains(int item){
    Node curr=head;
    while (curr.key<item){
        curr=curr.next;
    }
    return (curr.key==item)&& !curr.marked ;
}
```



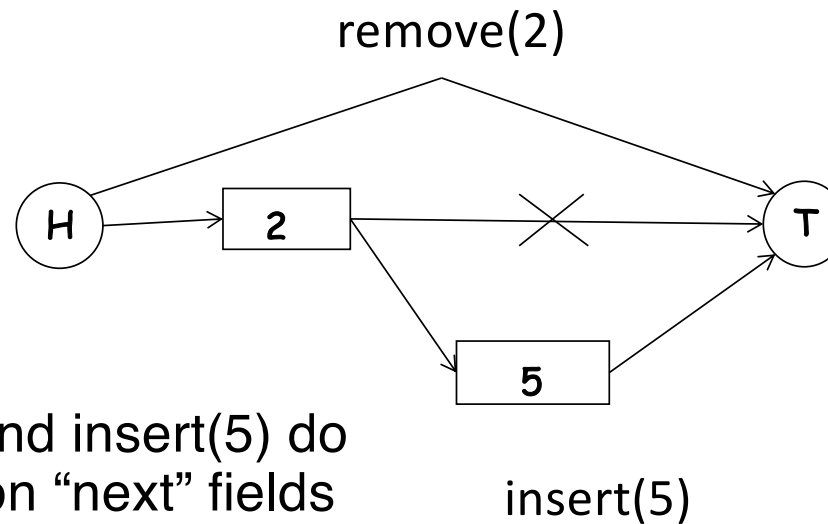
## Quiz 2.3: lazy

- Show that all conditions in the validation check are **necessary**  
Hint: consider concurrent **removes** on two consecutive nodes, or a **remove** concurrent to an **insert** of a preceding node
- Is the check **!curr.marked** <sup>Not necessary?</sup> necessary in **contains**?
- Determine linearization points for all operations:
  - ✓ **insert**(successful or not)
  - ✓ **remove** (successful or not)
  - ✓ **contains** (successful or not)Hint: for an unsuccessful **contains**(x), linearization point may vary depending on the presence of a concurrent **insert**(x)

# From locks to nonblocking

- Lazy [Heller et al.]: best of the class?
  - ✓ contains **wait-free**
  - ✓ add and remove are only **deadlock-free**
- Can we make updates lock-free?
  - ✓ Wait-free for contains
- Replace read and update of curr.next with **CAS**?
  - ✓ Not that easy: may need to **atomically update the reference and check the logical deletion mark**
  - ✓ AtomicMarkableReference in java, bit stealing in C++
  - ✓ **Maintain reference to the next item and logical deletion mark “together”**

## Why AMR or bit stealing?



- `remove(2)` and `insert(5)` do not conflict on “next” fields
- `insert(5)` is lost!
- non-coupled logical deletion checks do not prevent “lost updates”

# Nonblocking synchronization [Harris 2003]: lock-free updates and wait-free contains

```
public boolean remove(int item)
{
    ...
    while (true){
        \\ traverse with physical
        \\ removal of marked nodes
        \\ determine pred and curr

        if (curr.key!=item)){
            return false;}
        Node succ=curr.next.getReference();
        snip =
        curr.next.compareAndSet(succ,succ,
                               false,true);
        if (!snip) continue;
        pred.next.compareAndSet(curr,succ,
                               false,false);
        \\ just try once
        return true;
    }
}
```

- Even lazier: a successful **remove** does not **always** unlink the node, but marks it for deletion
- Updates unlink nodes marked for deletion by previous removes
- Remove first tests if curr.next stores the expected reference and, if yes, logically marks curr (restart if no)
- Then it uses CAS on two fields: succeeds only if the reference and mark do not change

# Nonblocking synchronization [Harris 2003]: lock-free updates and wait-free contains

```
public boolean insert(int item)
```

```
...
```

```
while (true){
```

```
    \\ traverse with physical
```

```
    \\ removal of marked nodes
```

```
    \\ determine pred and curr
```

```
    if (curr.key==item){
```

```
        return false;
```

```
    Node node=new Node();
```

```
    node.next = new AtomicMarkableReference(curr, false);
```

```
    if (pred.next.compareAndSet(curr, node, false, false)
```

```
    {
```

```
        return true;
```

```
    }
```

```
}
```

```
}
```

Insert atomically updates the markable reference pred.next with a reference to a new node, making sure sure that pred is not removed meanwhile

- More details in [Herlihy and Shavit, Chapter 9.8]

# Conventional synchronization

- Locks are hard to use efficiently
- Nonblocking implementations with CAS have inherent (hardware) limitations
- Multiple operations cannot be easily **composed**

What can we do about it?

# Transactions?

```
public class TxnList{

    private Node head;

    public boolean add(int item){
        atomic {
            Node pred=head;
            Node curr=head.next;
            while (curr.key < item){
                pred = curr;
                curr = pred.next;
            }
            if (curr.key==item){return false;}
            Node node = new Node(item);
            node.next=curr;
            pred.next=node;
            return true;
        }
    }
}
```

# Transactional memory

- A transaction `atomic { ... }` **commits** or **aborts**
- Committed transactions **serialize**:
  - ✓ Constitute a sequential execution
- Aborted transactions “never happened”
  - ✓ Can affect other aborted ones?
- A correct sequential program implies a correct concurrent one
- Composition is easy:

```
atomic{  
    x=q0.deq( );  
    q1.enq(x);  
}
```



# So what is better?

It depends on:

- the data structure (some are more concurrency-friendly than others, cf. queues vs. lists)
- workload (high update-rate vs. read-dominated)
- Programming skills
- TM inherent costs

	blocking dependent	Non blocking dependent	Non blocking independent
Global	SF	OF	WF
Local	DF		LF

- Project (in teams): list-based sets in java
  - ✓ What is better on what workload?
  - ✓ SynchroBench: <https://github.com/gramoli/synchrobench>
  - ✓ Compare Coarse-grained, HOH, Optimistic, Lazy
  - ✓ Various update ratios, scales, list sizes
  - ✓ Use a multiprocessor!

