Lock Freedom

Chris Rossbach and Calvin Lin

cs380p

Outline

Agenda

- Non-blocking Synchronization

Acknowledgements:

- *https://www.cl.cam.ac.uk/teaching/1718/R204/slides-tharris-2-lock-free.pptx*
- *http://concurrencyfreaks.blogspot.com/2013/05/lock-free-and-wait-free-definition-and.html*

Non-Blocking Synchronization

Non-Blocking Synchronization

Locks: a litany of problems

Non-Blocking Synchronization

Locks: a litany of problems
Deadlock

Non-Blocking Synchronization

Locks: a litany of problems
Deadlock
Priority inversion

Non-Blocking Synchronization

Locks: a litany of problems
Deadlock
Priority inversion
Convoys

Non-Blocking Synchronization

Locks: a litany of problems

Deadlock

Priority inversion

Convoys

Fault Isolation

Non-Blocking Synchronization

Locks: a litany of problems
Deadlock
Priority inversion
Convoys
Fault Isolation
Preemption Tolerance

Non-Blocking Synchronization

Locks: a litany of problems
Deadlock
Priority inversion
Convoys
Fault Isolation
Preemption Tolerance
Performance

Non-Blocking Synchronization

Locks: a litany of problems
Deadlock
Priority inversion
Convoys
Fault Isolation
Preemption Tolerance
Performance

Solution: don’t use locks

Non-Blocking Synchronization

Locks: a litany of problems
Deadlock
Priority inversion
Convoys
Fault Isolation
Preemption Tolerance
Performance

Lock-free programming

Lock-free programming

Subset of a broader class: ***Non-blocking Synchronization***

Lock-free programming

Subset of a broader class: ***Non-blocking Synchronization***
Thread-safe access shared mutable state without mutual exclusion

Lock-free programming

Subset of a broader class: ***Non-blocking Synchronization***
Thread-safe access shared mutable state without mutual exclusion
Possible without HW support

- e.g. Lamport’s Concurrent Buffer
- …but not really practical wo HW

Lock-free programming

Subset of a broader class: ***Non-blocking Synchronization***
Thread-safe access shared mutable state without mutual exclusion
Possible without HW support
- e.g. Lamport’s Concurrent Buffer
- …but not really practical wo HW

Built on atomic instructions like CAS + clever algorithmic tricks

Lock-free programming

Subset of a broader class: ***Non-blocking Synchronization***
Thread-safe access shared mutable state without mutual exclusion
Possible without HW support
- e.g. Lamport's Concurrent Buffer
- …but not really practical wo HW

Built on atomic instructions like CAS + clever algorithmic tricks
Lock-free *algorithms* are hard, so

Lock-free programming

Subset of a broader class: ***Non-blocking Synchronization***

Thread-safe access shared mutable state without mutual exclusion

Possible without HW support

- e.g. Lamport's Concurrent Buffer
- …but not really practical wo HW

Built on atomic instructions like CAS + clever algorithmic tricks

Lock-free *algorithms* are hard, so

General approach: encapsulate lock-free algorithms in data structures

- Queue, list, hash-table, skip list, etc.
- New LF data structure → research result

Basic List Append

Basic List Append

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

Basic List Append

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);
    if (*head_ref == NULL) {
        *head_ref = new_node;
        return;
    }
    while (last->next != NULL)
        last = last->next;
    last->next = new_node;
}
```

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

Basic List Append

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);
    if (*head_ref == NULL) {
        *head_ref = new_node;
        return;
    }
    while (last->next != NULL)
        last = last->next;
    last->next = new_node;
}
```

- Is this thread safe?

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

Basic List Append

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);
    if (*head_ref == NULL) {
        *head_ref = new_node;
        return;
    }
    while (last->next != NULL)
        last = last->next;
    last->next = new_node;
}
```

- Is this thread safe?
- What can go wrong?

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

Example: List Append

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

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);
    lock();
    if (*head_ref == NULL) {
        *head_ref = new_node;
    } else {
        while (last->next != NULL)
            last = last->next;
        last->next = new_node;
    }
    unlock();
}
```

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

Example: List Append

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);

    if (*head_ref == NULL) {
        *head_ref = new_node;
    } else {
        while (last->next != NULL)
            last = last->next;
        last->next = new_node;
    }

}
```

Example: List Append

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

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);

    if (*head_ref == NULL) {
        *head_ref = new_node;
    } else {
        while (last->next != NULL)
            last = last->next;
        last->next = new_node;
    }

}
```

Example: List Append

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

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);

    if (*head_ref == NULL) {
        *head_ref = new_node;
    } else {
        while (last->next != NULL)
            last = last->next;
        last->next = new_node;
    }

}
```

property do the locks enforce?

Example: List Append

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

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);

    if (*head_ref == NULL) {
        *head_ref = new_node;
    } else {
        while (last->next != NULL)
            last = last->next;
        last->next = new_node;
    }

}
```

- property do the locks enforce?
- What does the mutual exclusion ensure?

Example: List Append

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

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);

    if (*head_ref == NULL) {
        *head_ref = new_node;
    } else {
        while (last->next != NULL)
            last = last->next;
        last->next = new_node;
    }

}
```

- property do the locks enforce?
- What does the mutual exclusion ensure?
- Can we ensure consistent view (invariants hold) sans mutual exclusion?

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

Example: List Append

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data, head_ref);

    if (*head_ref == NULL) {
        *head_ref = new_node;
    } else {
        while (last->next != NULL)
            last = last->next;
        last->next = new_node;
    }

}
```

- property do the locks enforce?
- What does the mutual exclusion ensure?
- Can we ensure consistent view (invariants hold) sans mutual exclusion?
- Key insight: allow inconsistent view and fix it up algorithmically

Example: List Append

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

```
void append(Node** head_ref, int new_data) {
    Node* new_node = mknode(new_data);
    new_node->next = NULL;
    while(TRUE) {
        Node * last = *head_ref;
        if(last == NULL) {
            if(cas(head_ref, new_node, NULL))
                break;
        }
        while(last->next != NULL)
            last = last->next;
        if(cas(&last->next, new_node, NULL))
            break;
    }
}
```

ef);

ual exclusion?

- Key insight: allow inconsistent view and fix it up algorithmically

Example: SP-SC Queue

```
next(x):
    if(x == Q_size-1) return 0;
    else return x+1;
```

```
Q_get(data):
    t = Q_tail;
    while(t == Q_head)
      ;
    data = Q_buf[t];
    Q_tail = next(t);
```

```
Q_put(data):
    h = Q_head;
    while(next(h) == Q_tail)
      ;
    Q_buf[h] = data;
    Q_head = next(h);
```

- Single-producer single-consumer
- Why/when does this work?

Example: SP-SC Queue

```
next(x):
    if(x == Q_size-1) return 0;
    else return x+1;

Q_get(data):
    t = Q_tail;
    while(t == Q_head)
      ;
    data = Q_buf[t];
    Q_tail = next(t);

Q_put(data):
    h = Q_head;
    while(next(h) == Q_tail)
      ;
    Q_buf[h] = data;
    Q_head = next(h);
```

- Single-producer single-consumer
- Why/when does this work?

1. Q_head is last write in Q_put, so Q_get never gets "ahead".
2. *single* p,c only (as advertised)
3. Requires fence before setting Q head
4. Devil in the details of "wait"
5. No lock → "optimistic"

Lock-Free Stack

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

```
void push(int t) {
    Node* node = new Node(t);
    do {
        node->next = head;
    } while (!cas(&head, node, node->next));
}

bool pop(int& t) {
   Node* current = head;
   while(current) {
       if(cas(&head, current->next, current)) {
           t = current->data;
           return true;
       }
       current = head;
   }
   return false;
}
```

Lock-Free Stack

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

```
void push(int t) {
     Node* node = new Node(t);
     do {
          node->next = head;
     } while (!cas(&head, node, node->next));
}

bool pop(int& t) {
    Node* current = head;
    while(current) {
        if(cas(&head, current->next, current)) {
            t = current->data;
            return true;
        }
        current = head;
    }
    return false;
}
```

- Why does is it work?

Lock-Free Stack

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

```
void push(int t) {
     Node* node = new Node(t);
     do {
          node->next = head;
     } while (!cas(&head, node, node->next));
}

bool pop(int& t) {
    Node* current = head;
    while(current) {
        if(cas(&head, current->next, current)) {
            t = current->data; // problem?
            return true;
        }
        current = head;
    }
    return false;
}
```

- Why does is it work?

Lock-Free Stack

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

```
void push(int t) {
    Node* node = new Node(t);
    do {
        node->next = head;
    } while (!cas(&head, node, node->next));
}

bool pop(int& t) {
    Node* current = head;
    while(current) {
        if(cas(&head, current->next, current)) {
            t = current->data; // problem?
            return true;
        }
        current = head;
    }
    return false;
}
```

- Why does is it work?
- Does it enforce all invariants?

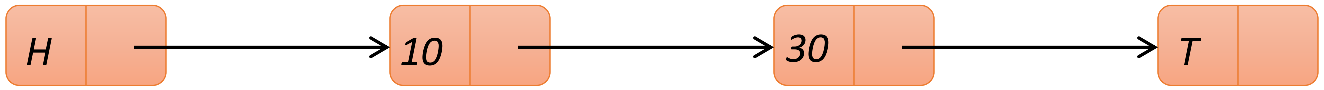
ABA Problem

- Thread 1 observes shared variable → 'A'
- Thread 1 calculates using that value
- Thread 2 changes variable to B
  - if Thread 1 wakes up now and tries to CAS, CAS fails and Thread 1 retries
- Instead, Thread 2 changes variable back to A!
  - Very bad if the variables are pointers
- Anyone see a work-around?

- Keep update count → DCAS
- Avoid re-using memory
- Multi-CAS support → HTM

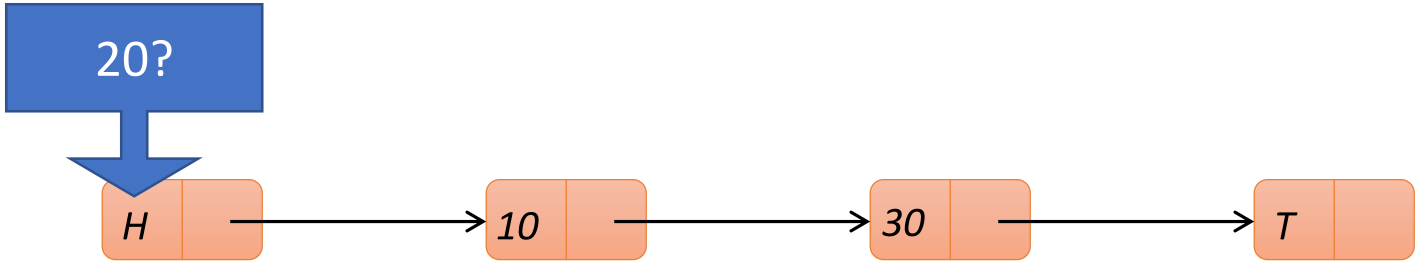
Correctness: Searching a sorted list

- find(20):



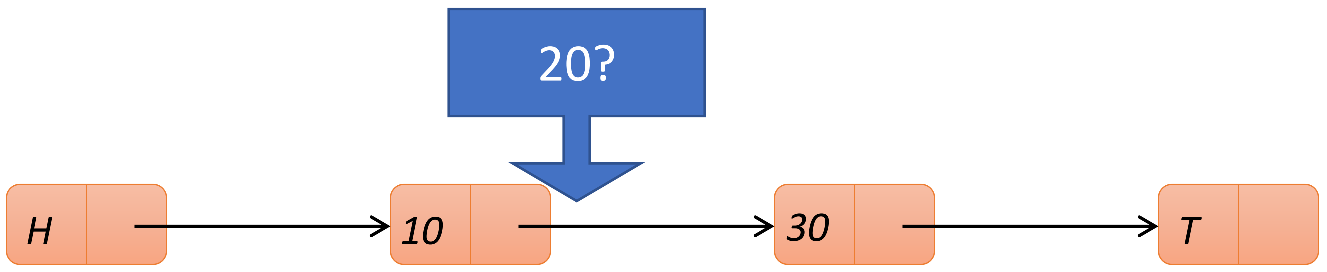
Correctness: Searching a sorted list

- find(20):



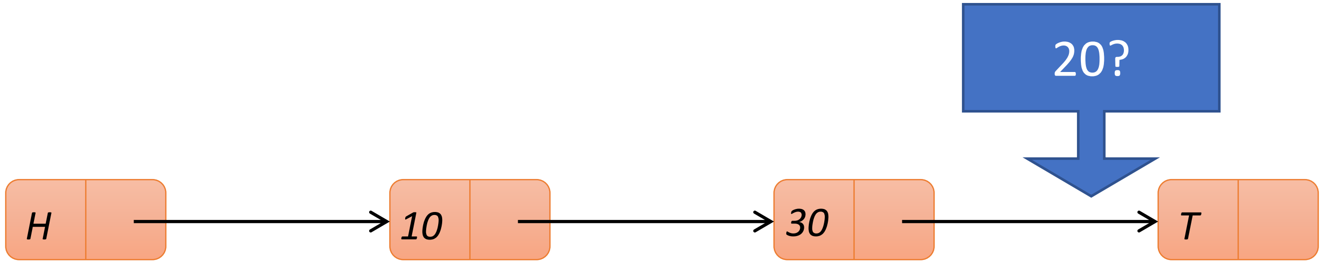
Correctness: Searching a sorted list

- find(20):



Correctness: Searching a sorted list

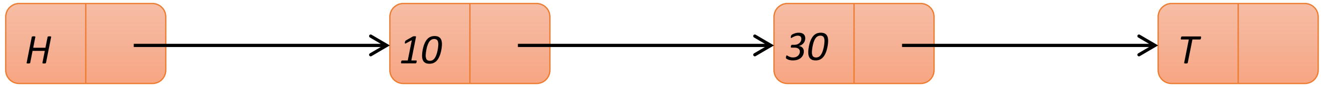
- find(20):



find(20) -> false

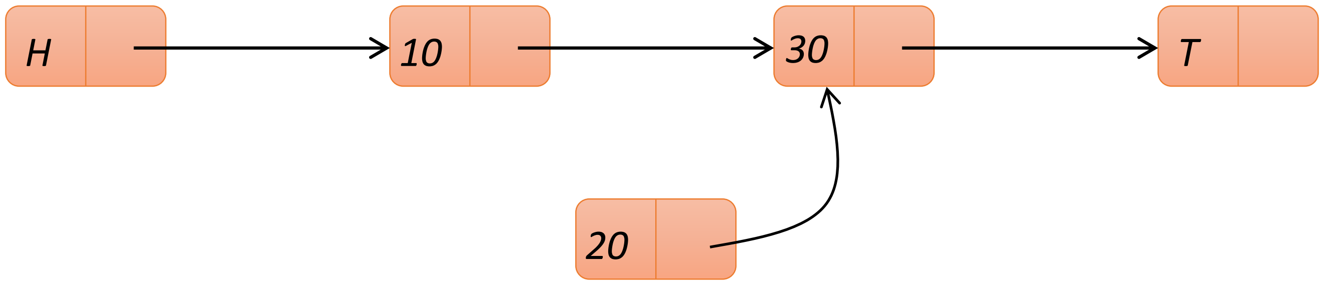
Inserting an item with CAS

- insert(20):



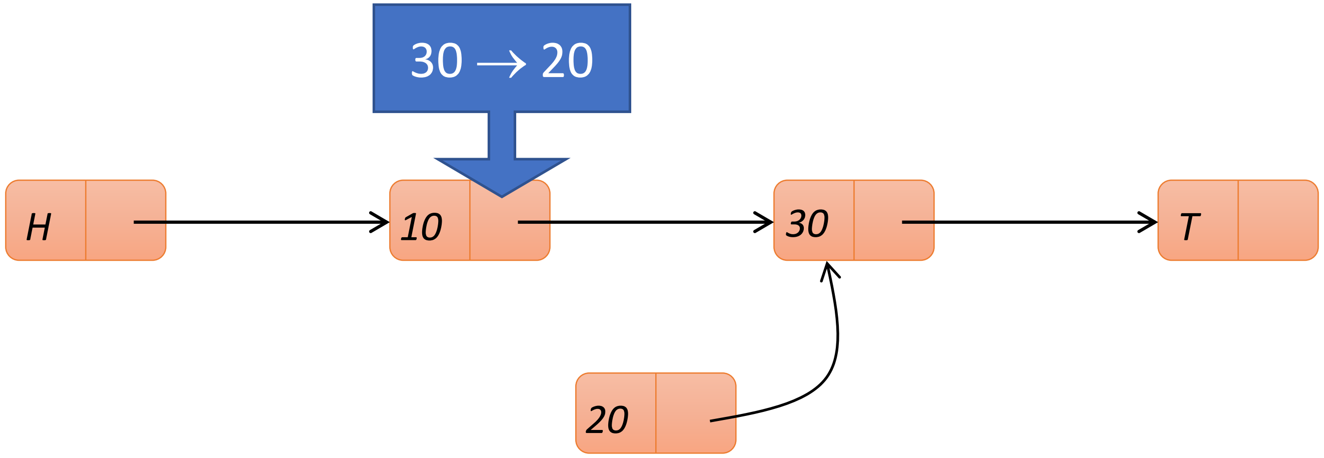
Inserting an item with CAS

- insert(20):



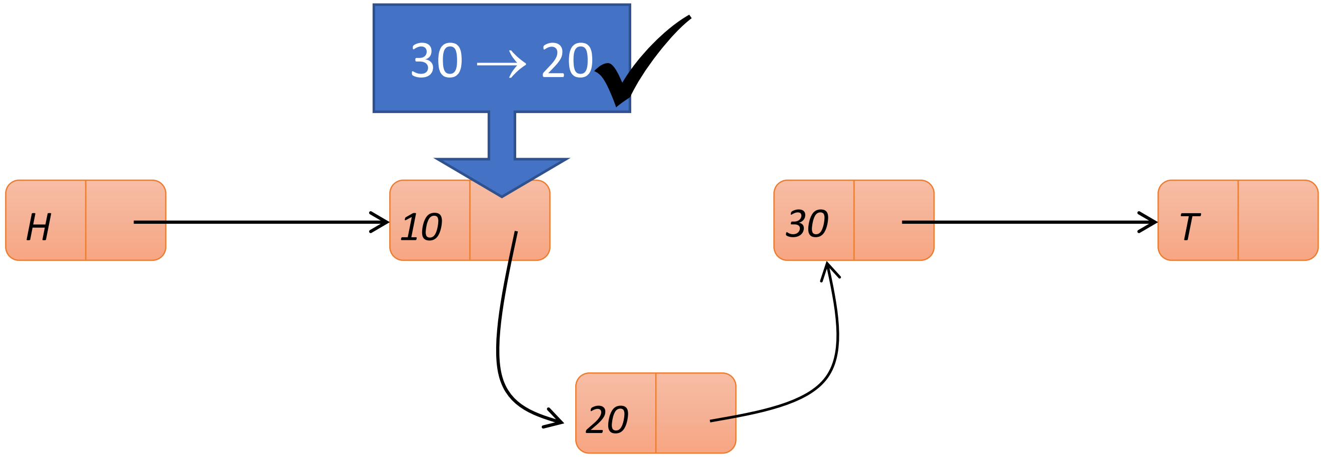
Inserting an item with CAS

- insert(20):



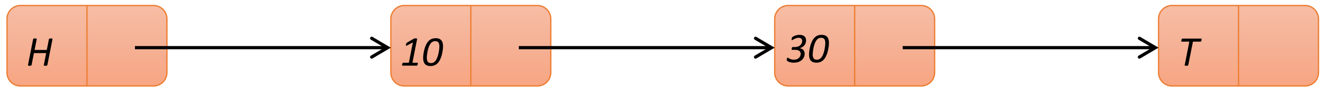
Inserting an item with CAS

- insert(20):



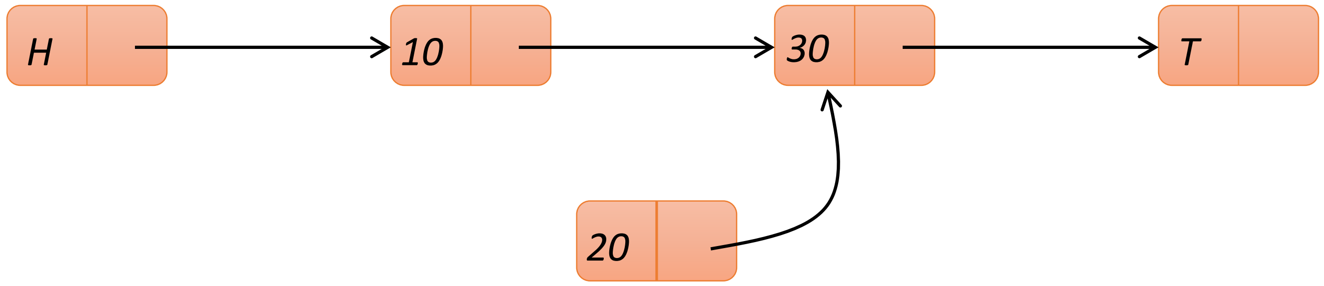
insert(20) -> true

Inserting an item with CAS



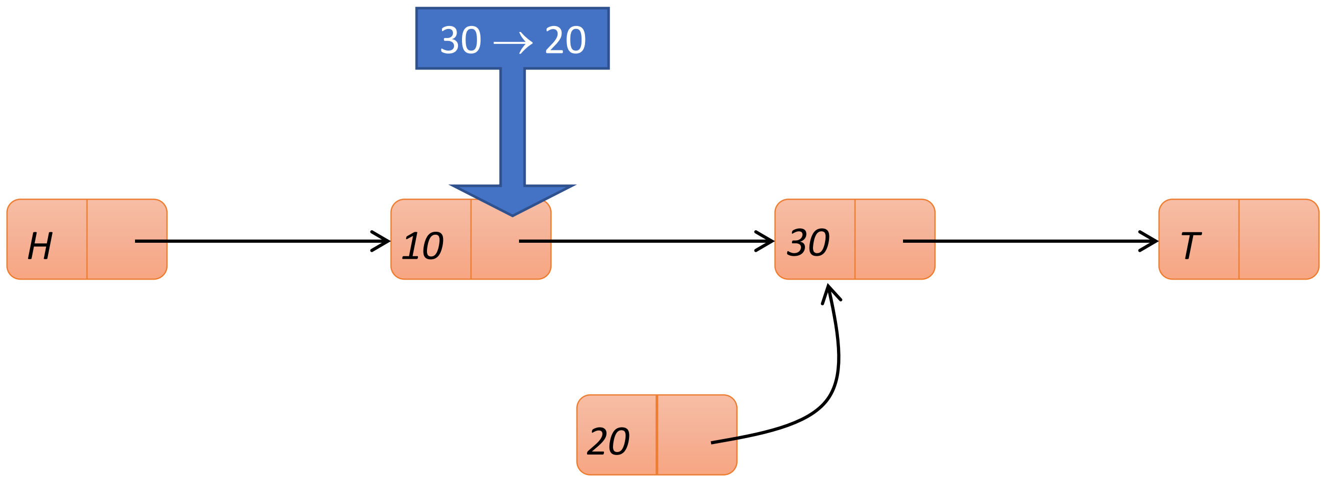
Inserting an item with CAS

- insert(20):



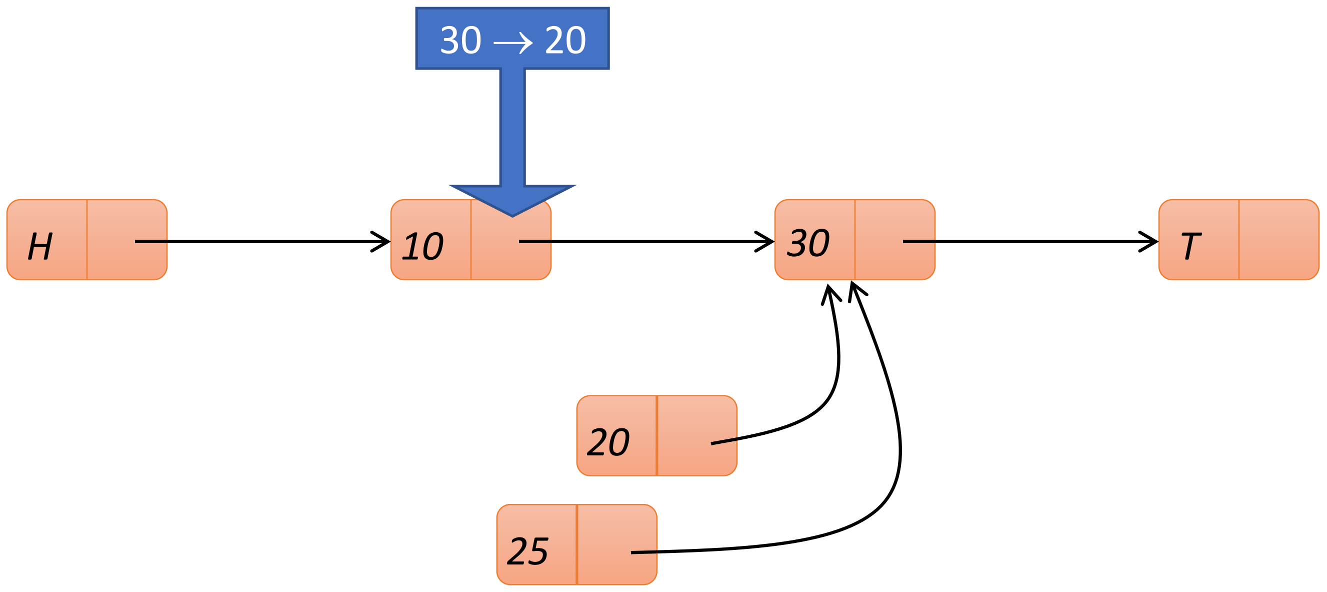
Inserting an item with CAS

- insert(20):



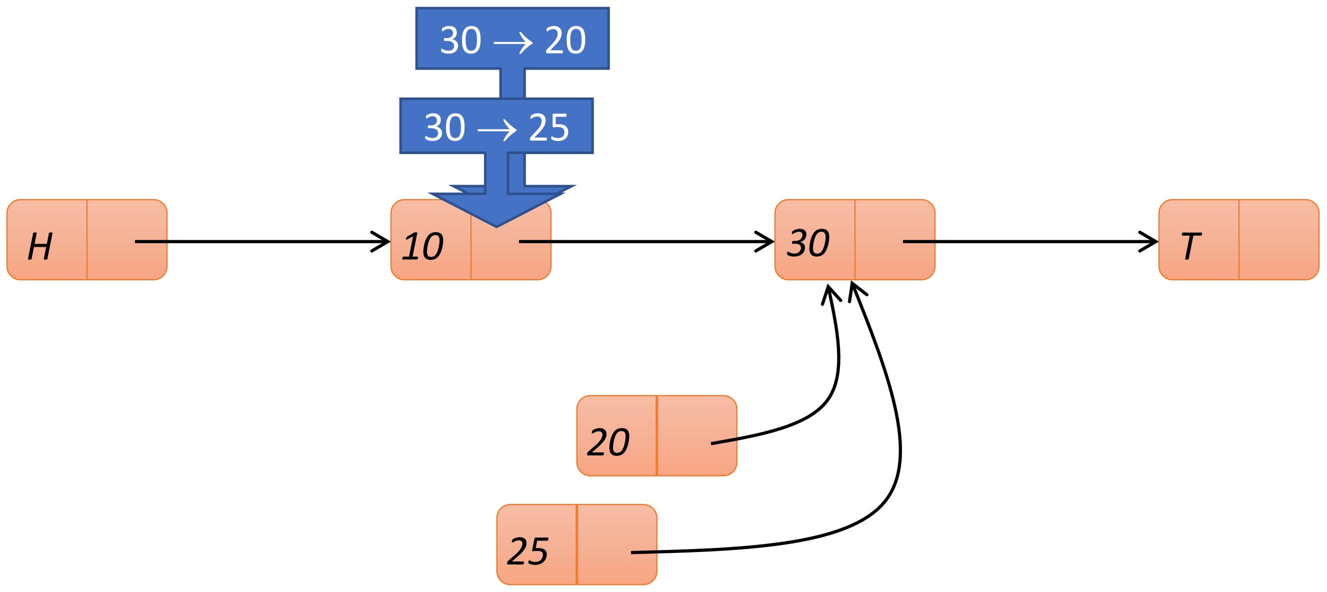
Inserting an item with CAS

- insert(20):
- insert(25):



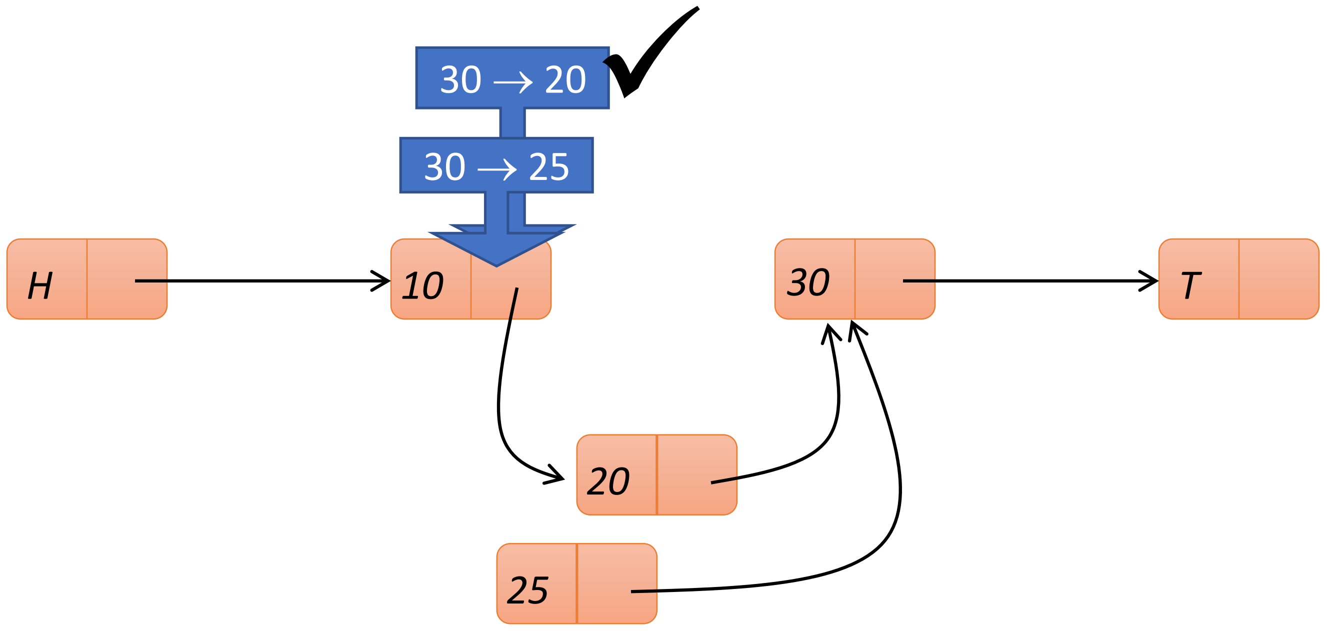
Inserting an item with CAS

- insert(20):
- insert(25):



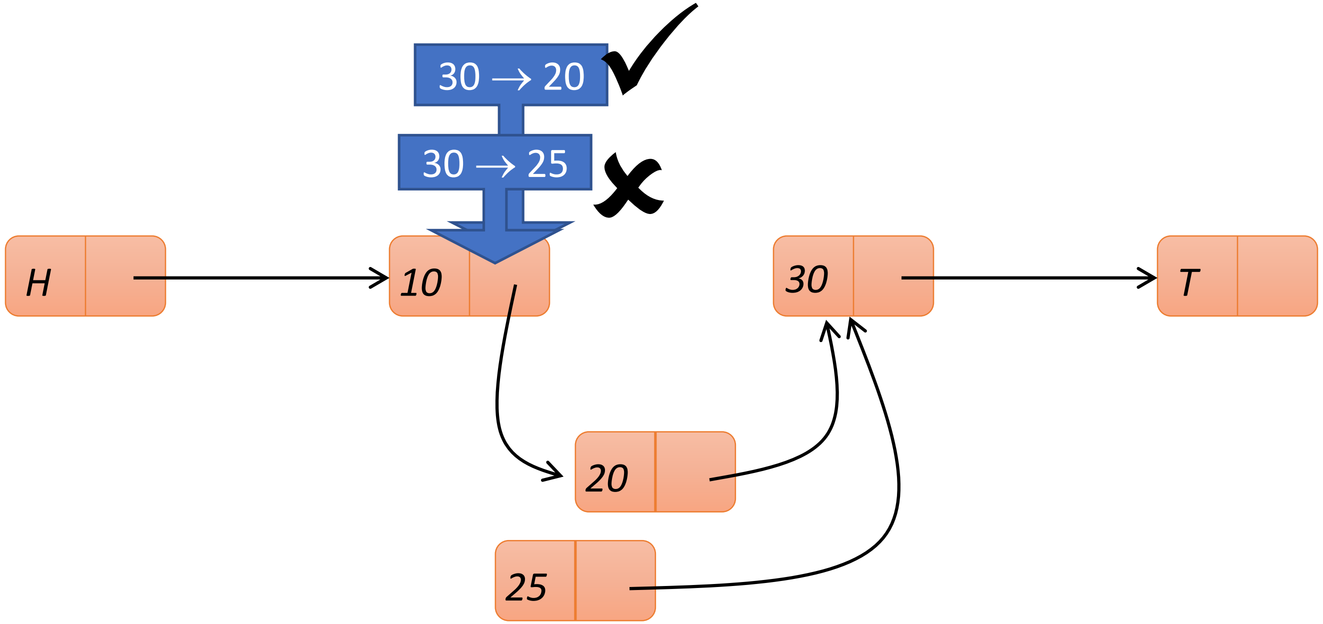
Inserting an item with CAS

- insert(20):
- insert(25):



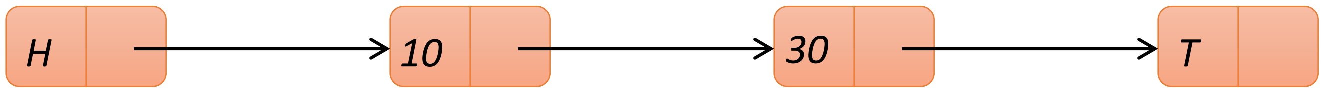
Inserting an item with CAS

- insert(20):
- insert(25):



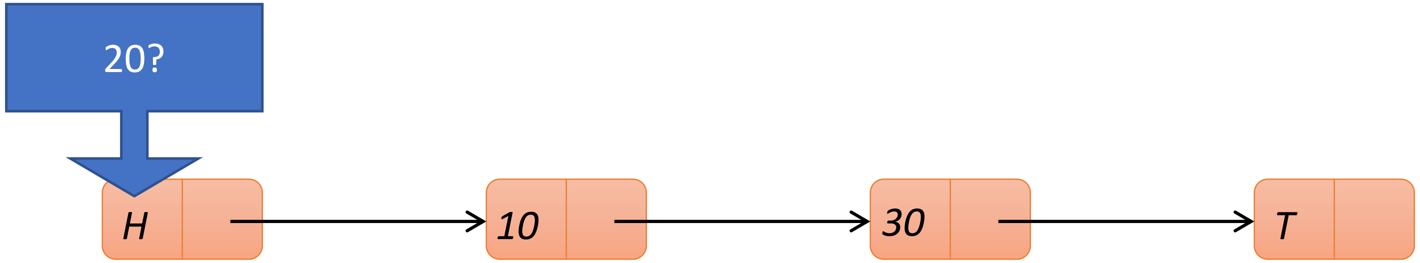
Searching and finding together

- find(20)



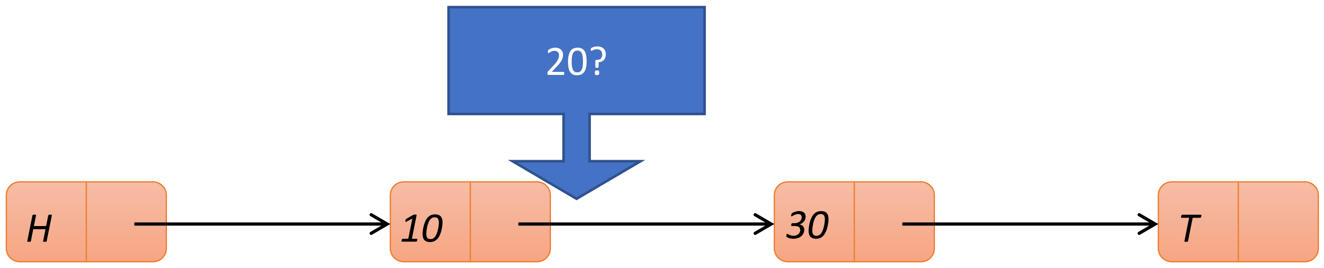
Searching and finding together

- find(20)



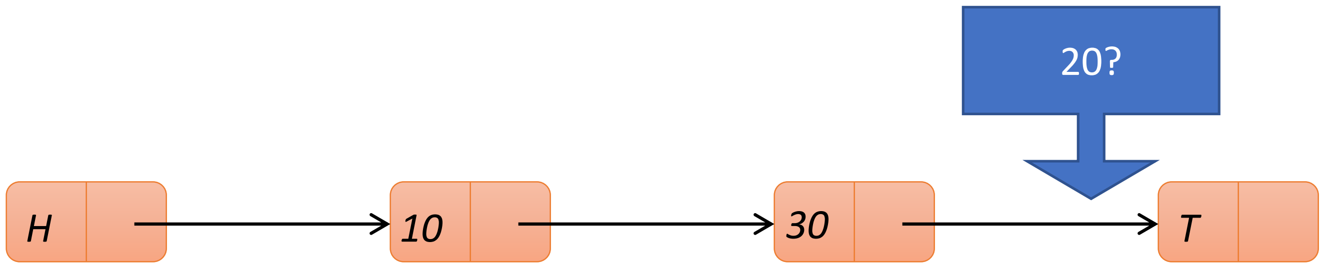
Searching and finding together

- find(20)



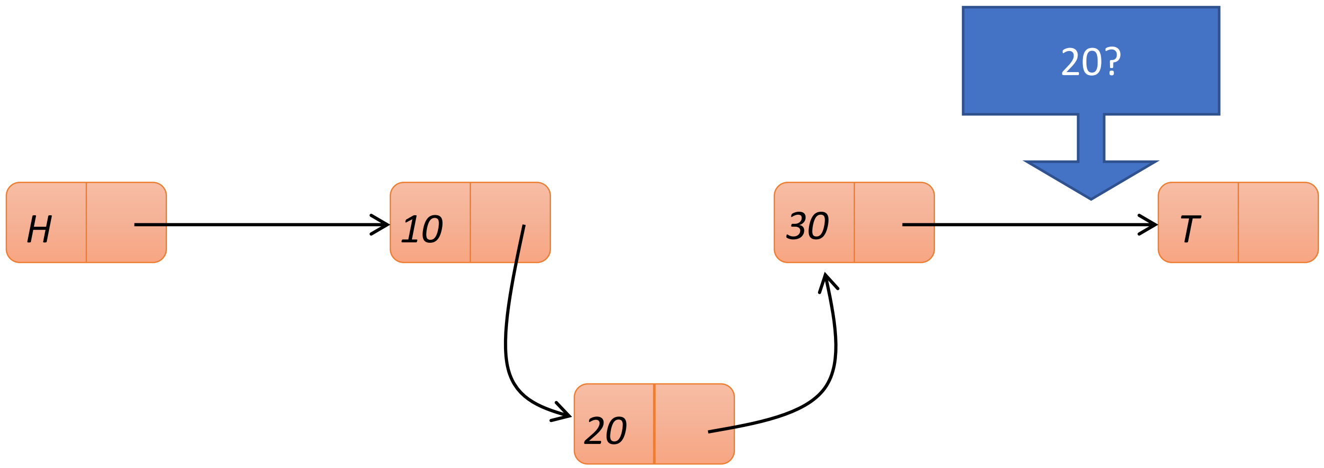
Searching and finding together

- find(20)



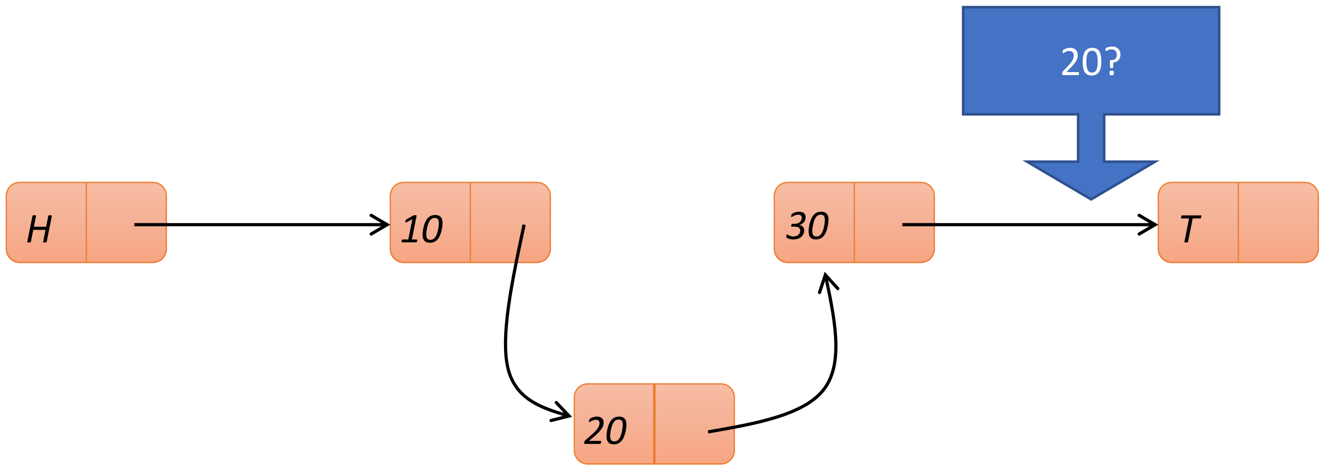
Searching and finding together

- find(20)
- insert(20) -> true



Searching and finding together

- find(20) -> false
- insert(20) -> true



Searching and finding together

- find(20) -> false
- insert(20) -> true

This thread saw 20 was not in the set...

...but this thread succeeded in putting it in!

- Is this a correct implementation?
- Should the programmer be surprised if this happens?
- What about more complicated mixes of operations?

Correctness criteria

Informally:

Look at the behavior of the data structure

- what operations are called on it
- what their results are

If behavior is indistinguishable from atomic calls to a sequential implementation then the concurrent implementation is correct.

Sequential history

- No overlapping invocations

Sequential history

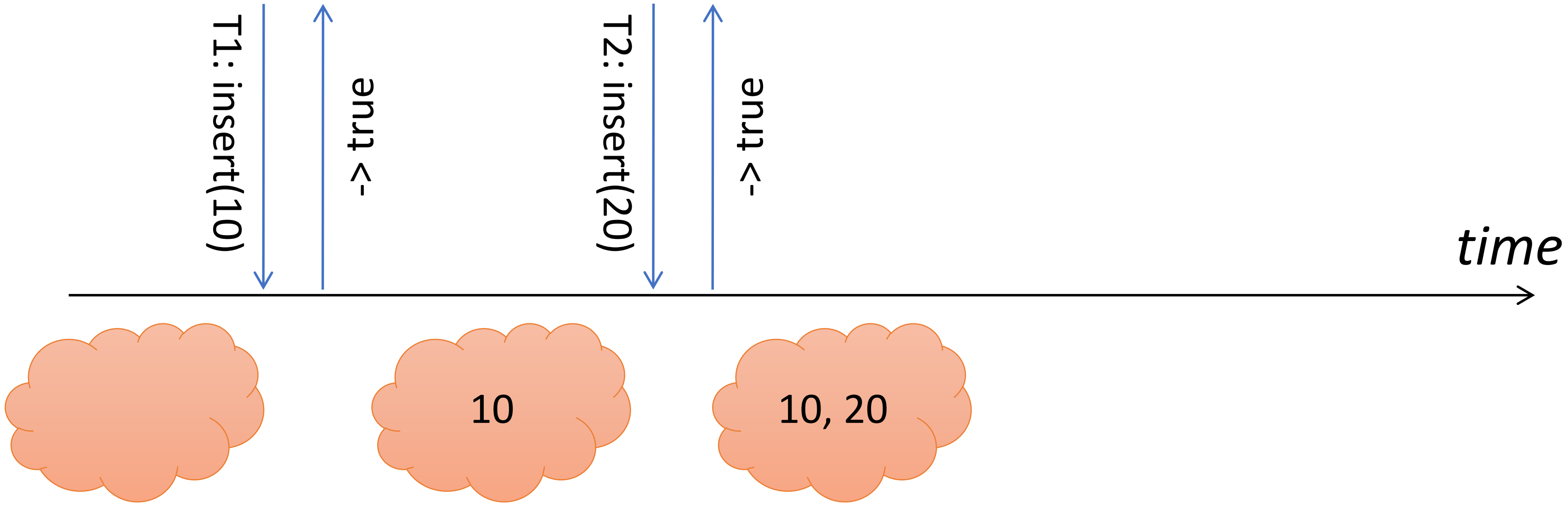
- No overlapping invocations

Sequential history

- No overlapping invocations

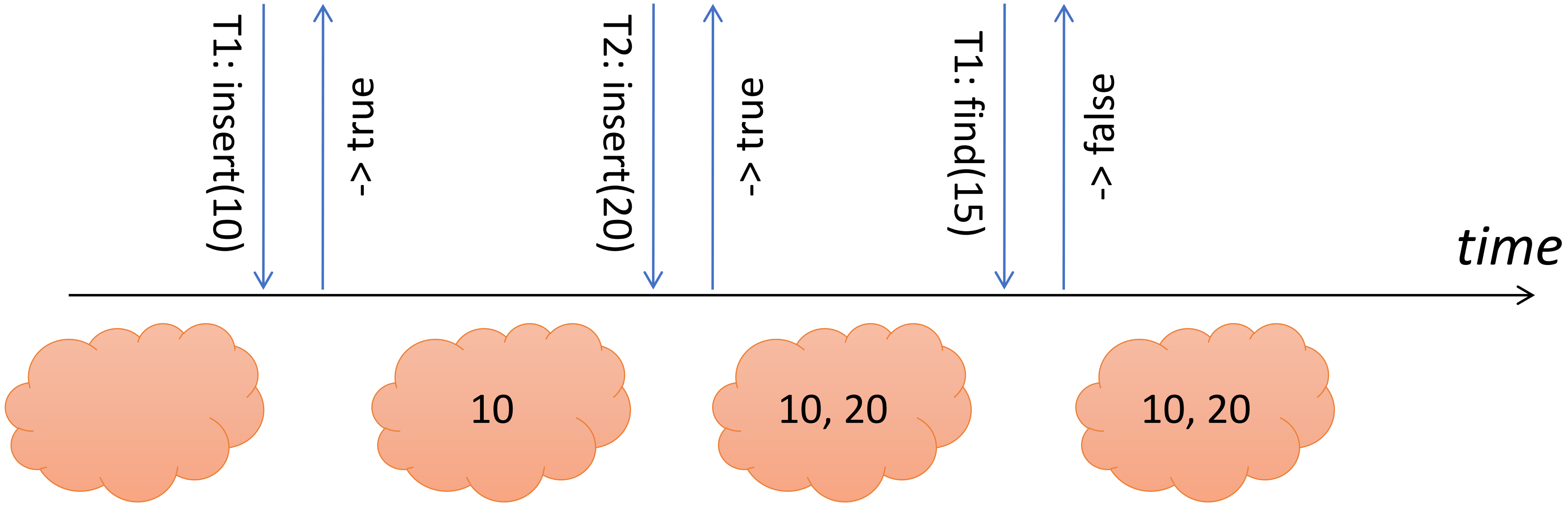
Sequential history

- No overlapping invocations



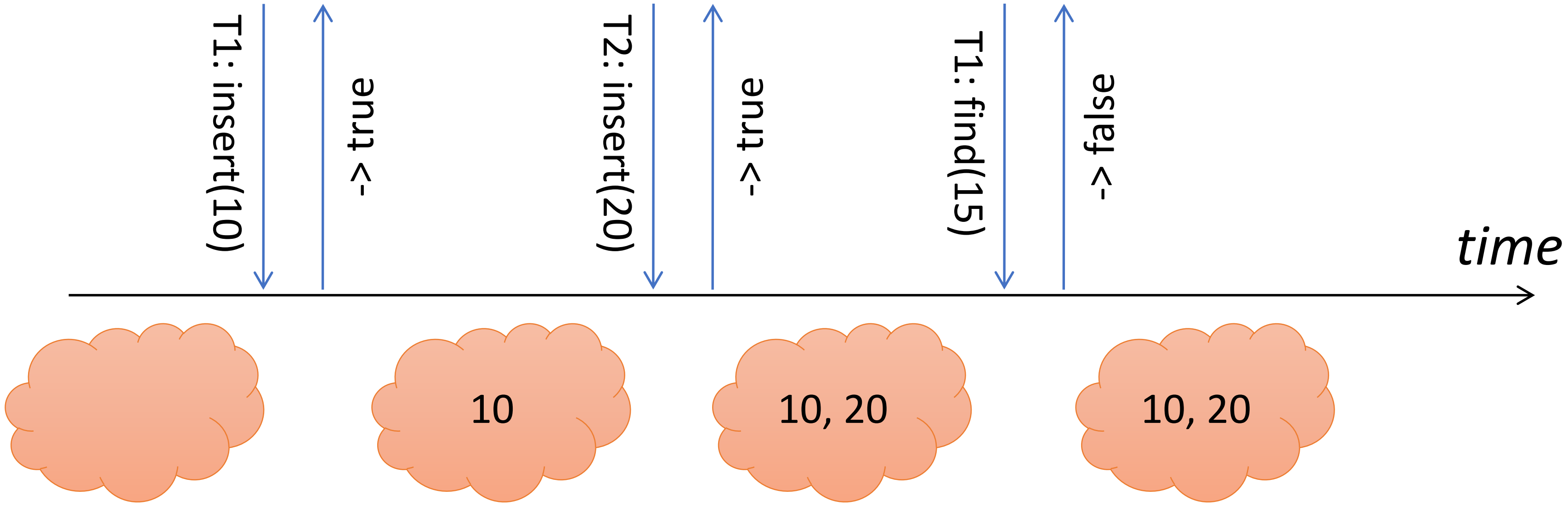
Sequential history

- No overlapping invocations



Sequential history

- No overlapping invocations

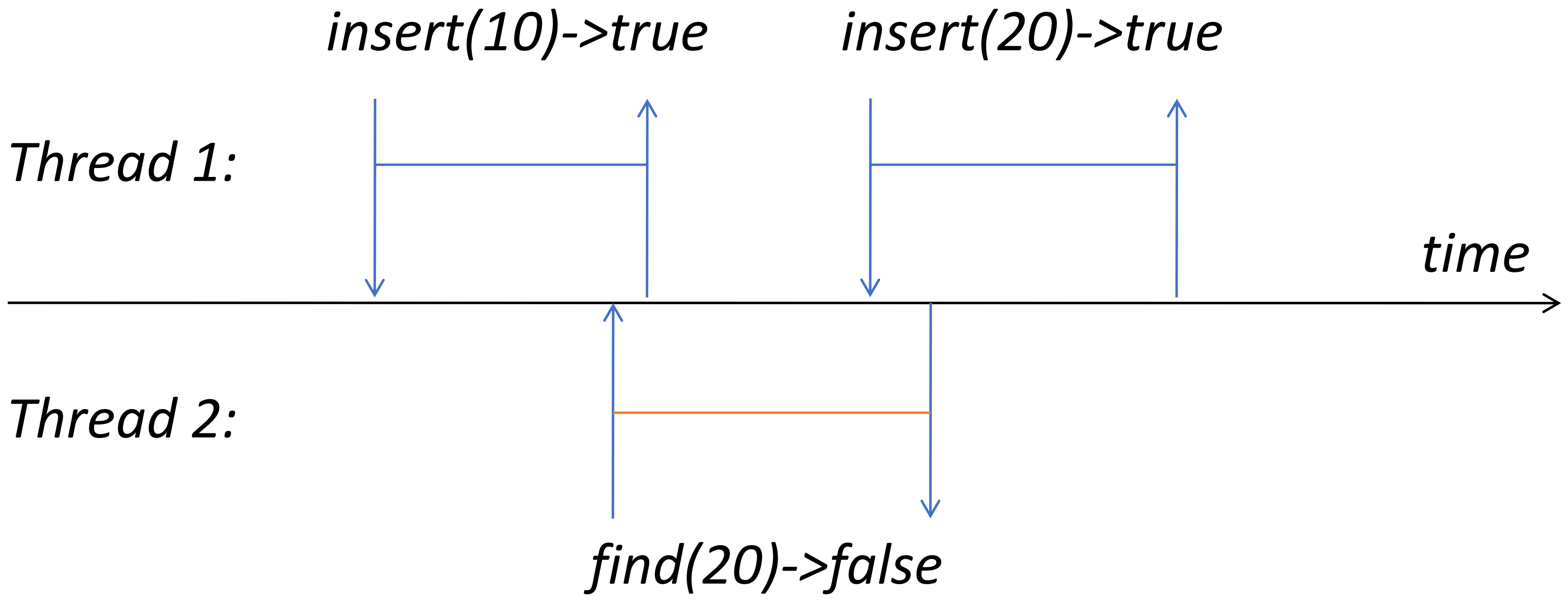


Linearizability: concurrent behaviour should be similar

- even when threads can see intermediate state
- Recall: mutual exclusion precludes overlap

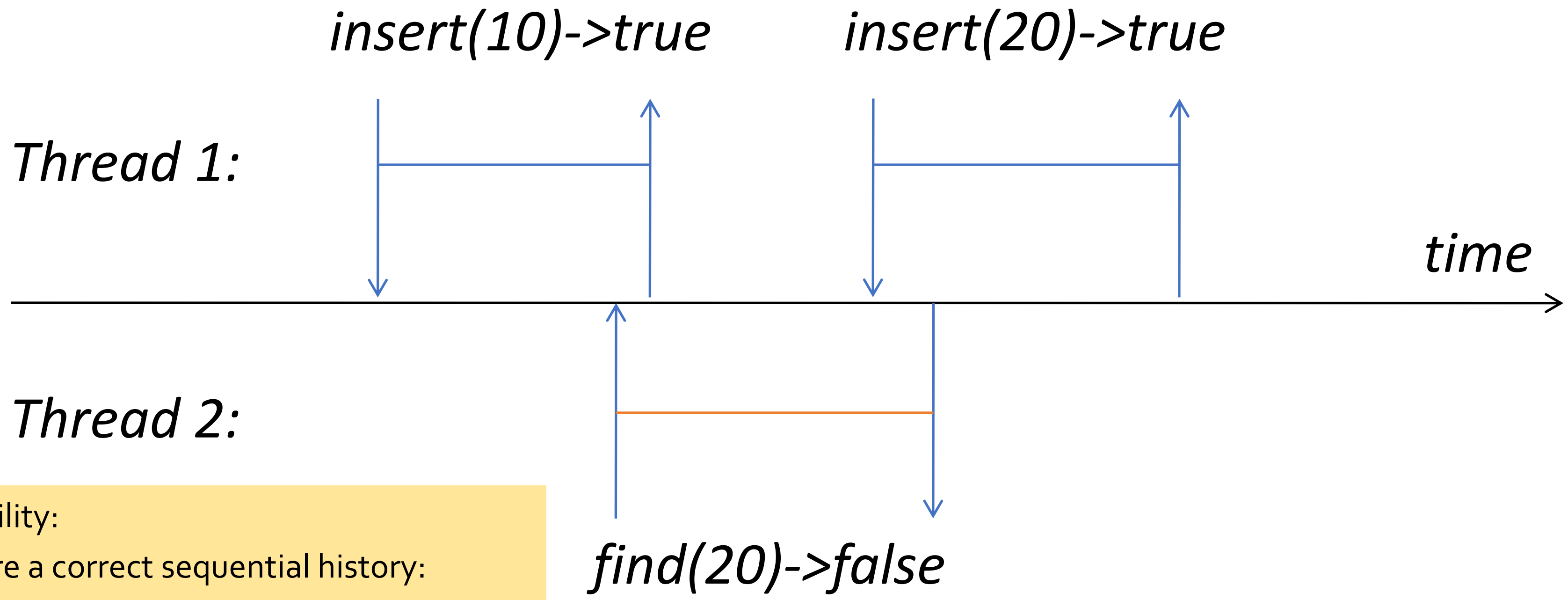
Concurrent history

Allow *overlapping* invocations



Concurrent history

Allow *overlapping* invocations

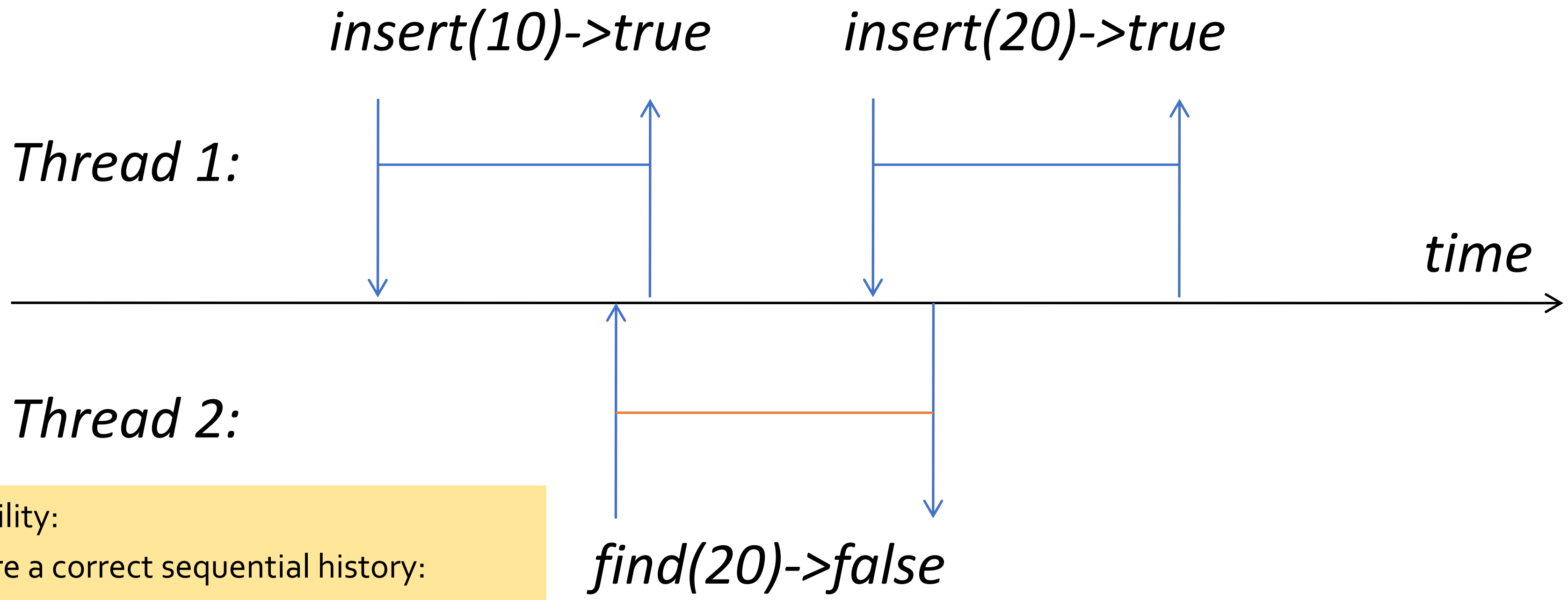


Linearizability:

- Is there a correct sequential history:
  - Same results as the concurrent one
  - Consistent with the timing of the invocations/responses?
  - Start/end impose ordering constraints

Concurrent history

Allow *overlapping* invocations

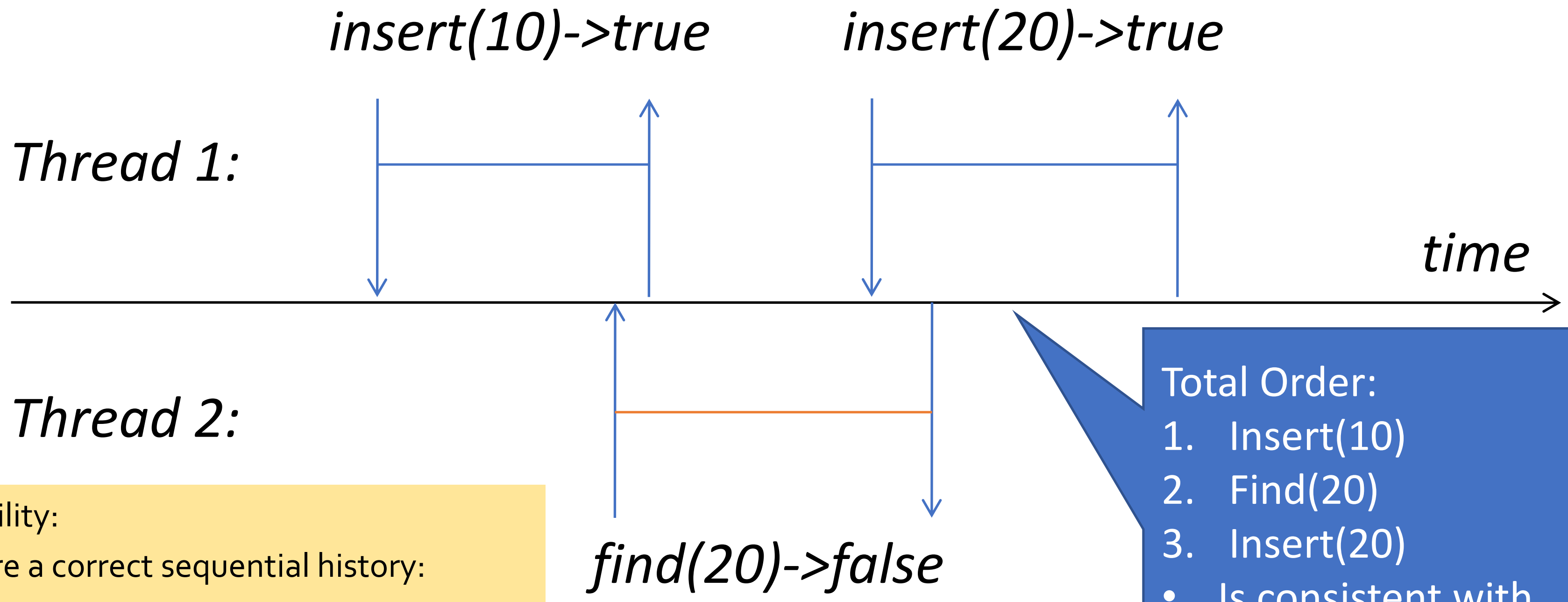


Linearizability:

- Is there a correct sequential history:
  - Same results as the concurrent one
  - Consistent with the timing of the invocations/responses?
  - Start/end impose ordering constraints

Concurrent history

Allow *overlapping* invocations



Total Order:

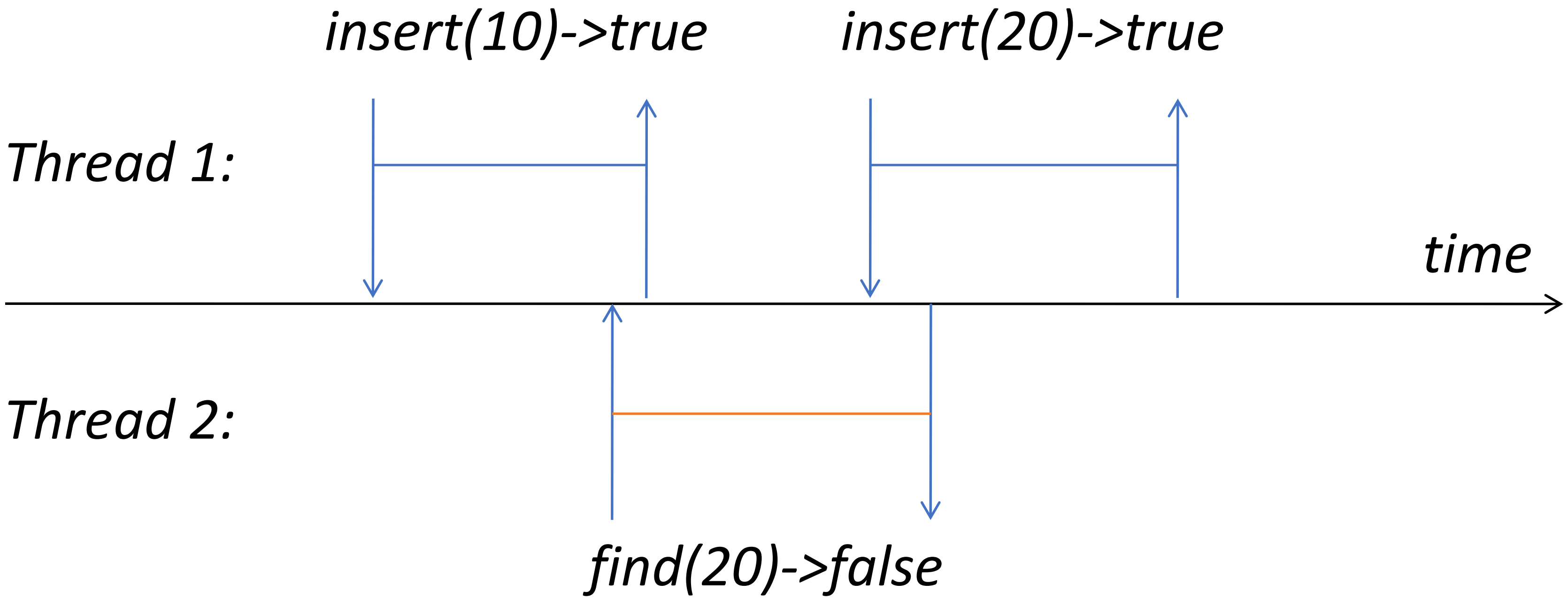
1. Insert(10)
2. Find(20)
3. Insert(20)

- Is consistent with real-time order
- 2, 3 overlap, but return order OK

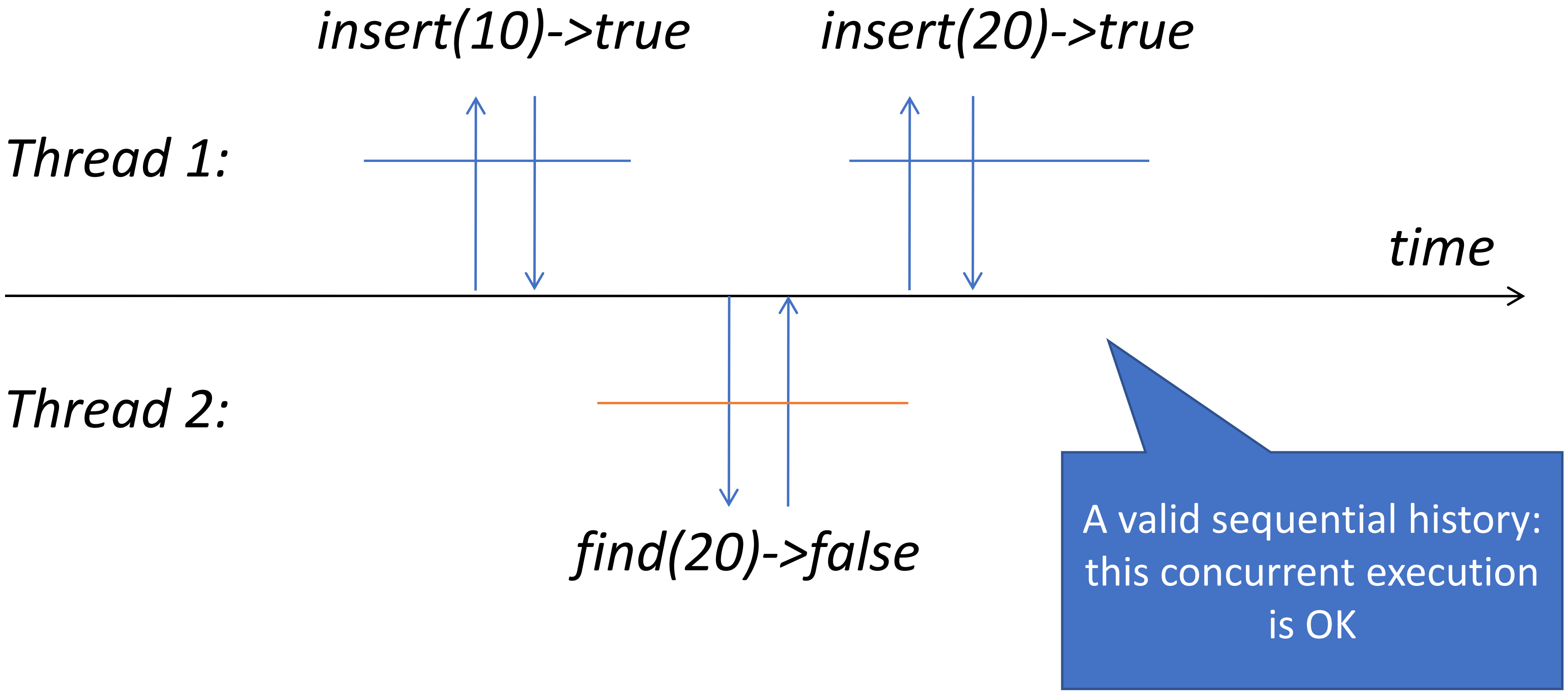
Linearizability:

- Is there a correct sequential history:
  - Same results as the concurrent one
  - Consistent with the timing of the invocations/responses?
  - Start/end impose ordering constraints

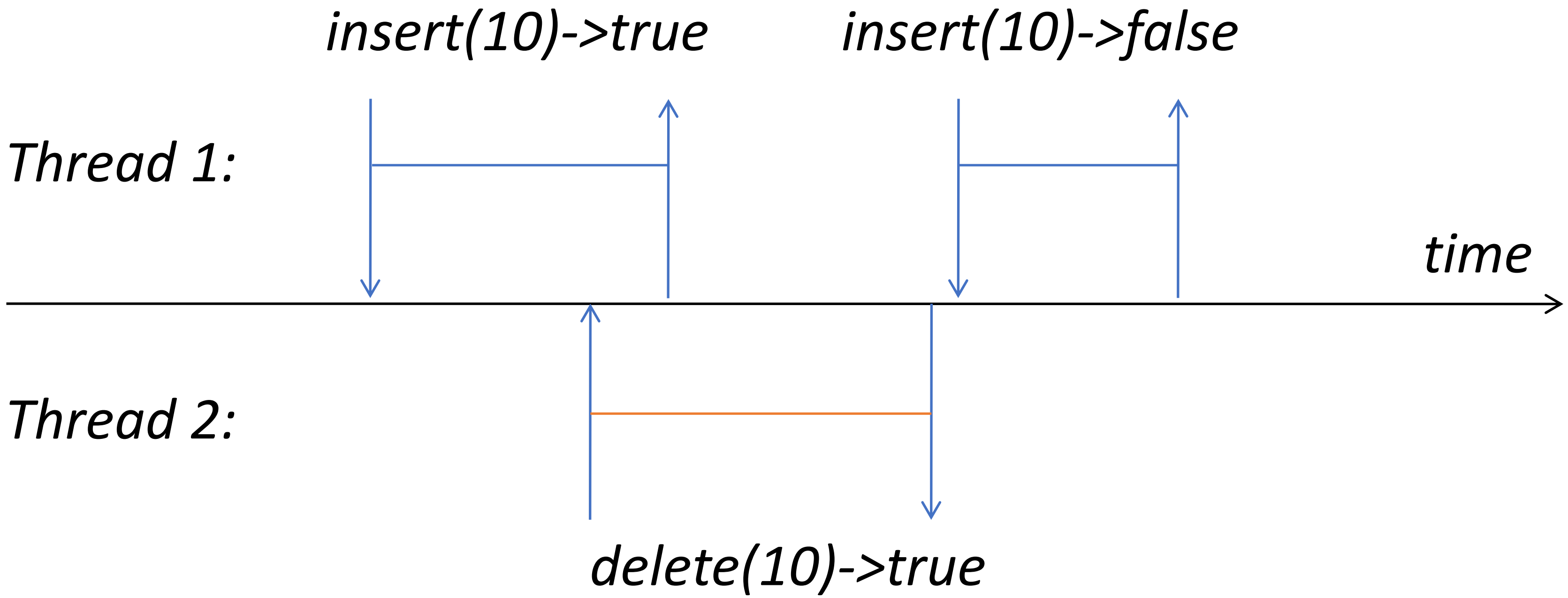
Example: linearizable



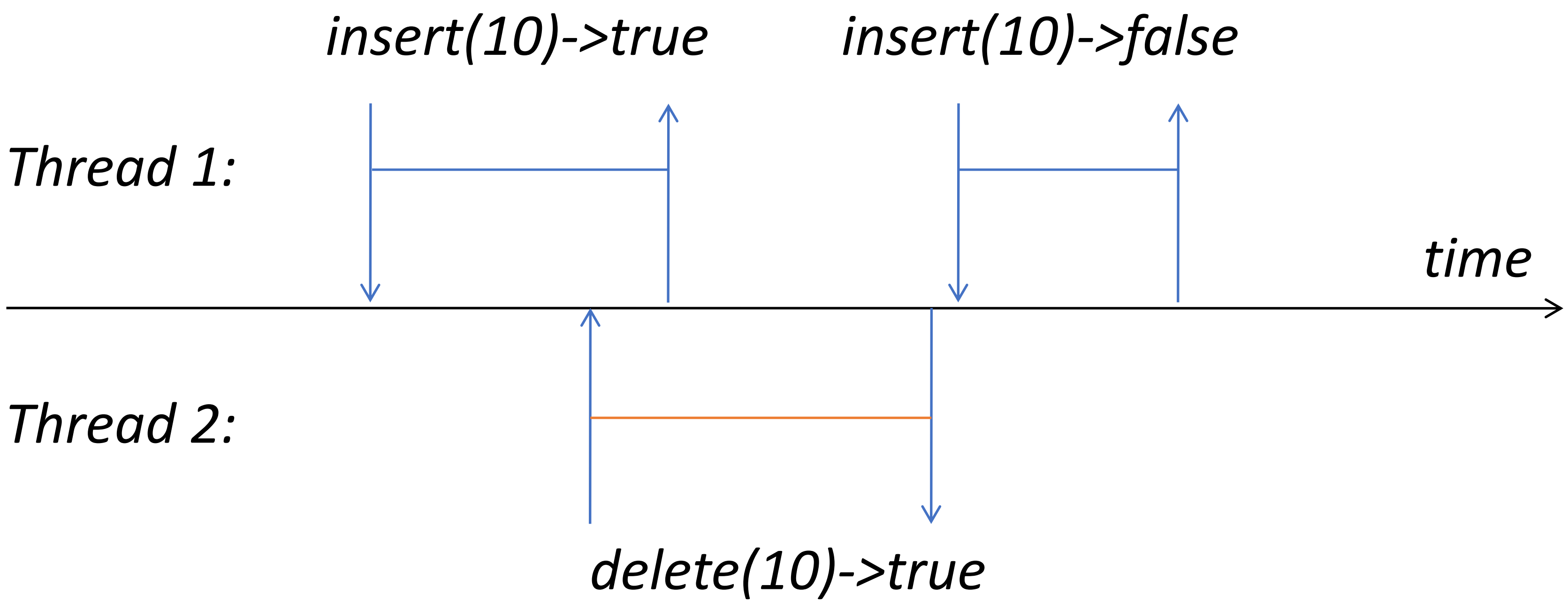
Example: linearizable



Example: not linearizable

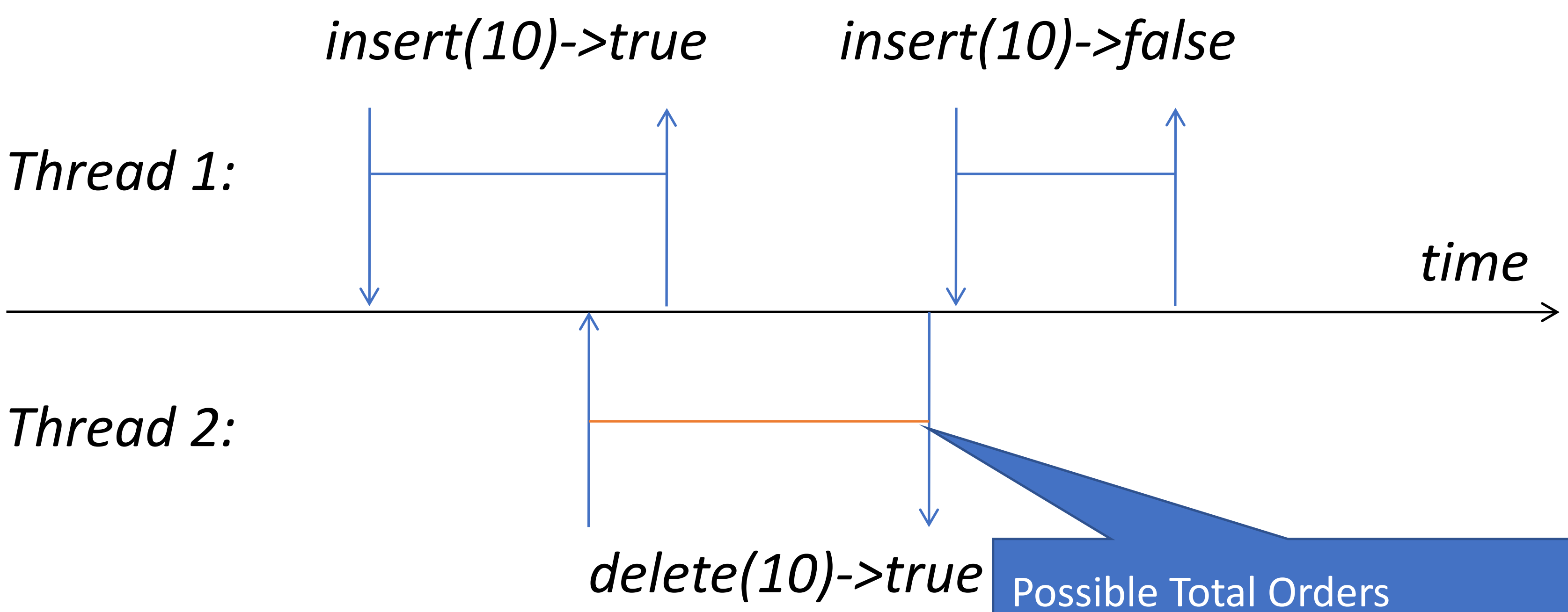


Example: not linearizable



Why is this one NOT OK?

Example: not linearizable



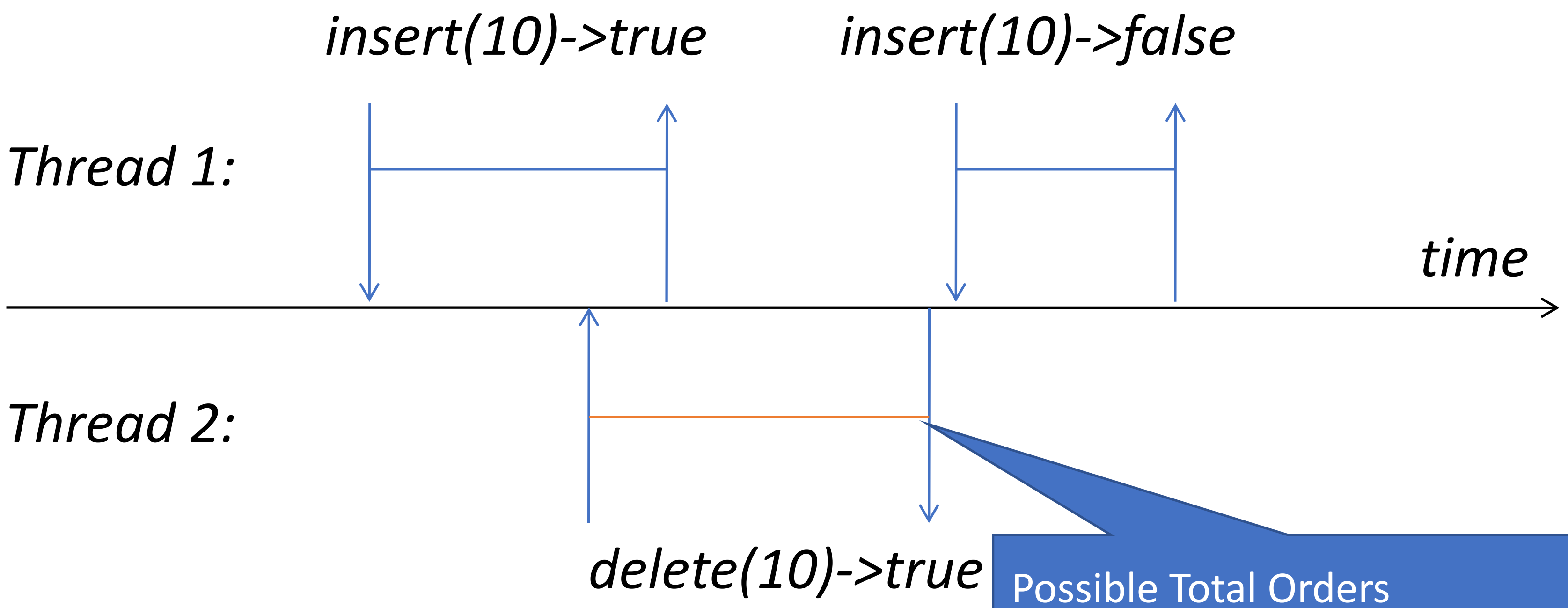
Possible Total Orders

1. Insert(10)	1. Delete(10)
2. Delete(10)	2. Insert(10)
3. Insert(10)	3. Insert(10)

- Both consistent with real-time order
- 1, 2 overlap, but 3 doesn't

Why is this one NOT OK?

Example: not linearizable



Possible Total Orders

1. Insert(10)
2. Delete(10)
3. Insert(10)

1. Delete(10)
2. Insert(10)
3. Insert(10)

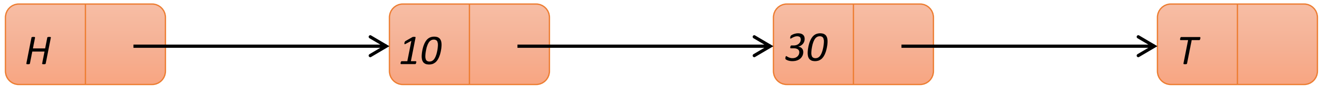
- Both consistent with real-time order
- 1, 2 overlap, but 3 doesn't

Why is this one NOT OK?

How can things like this happen?

Example Revisited

- find(20)

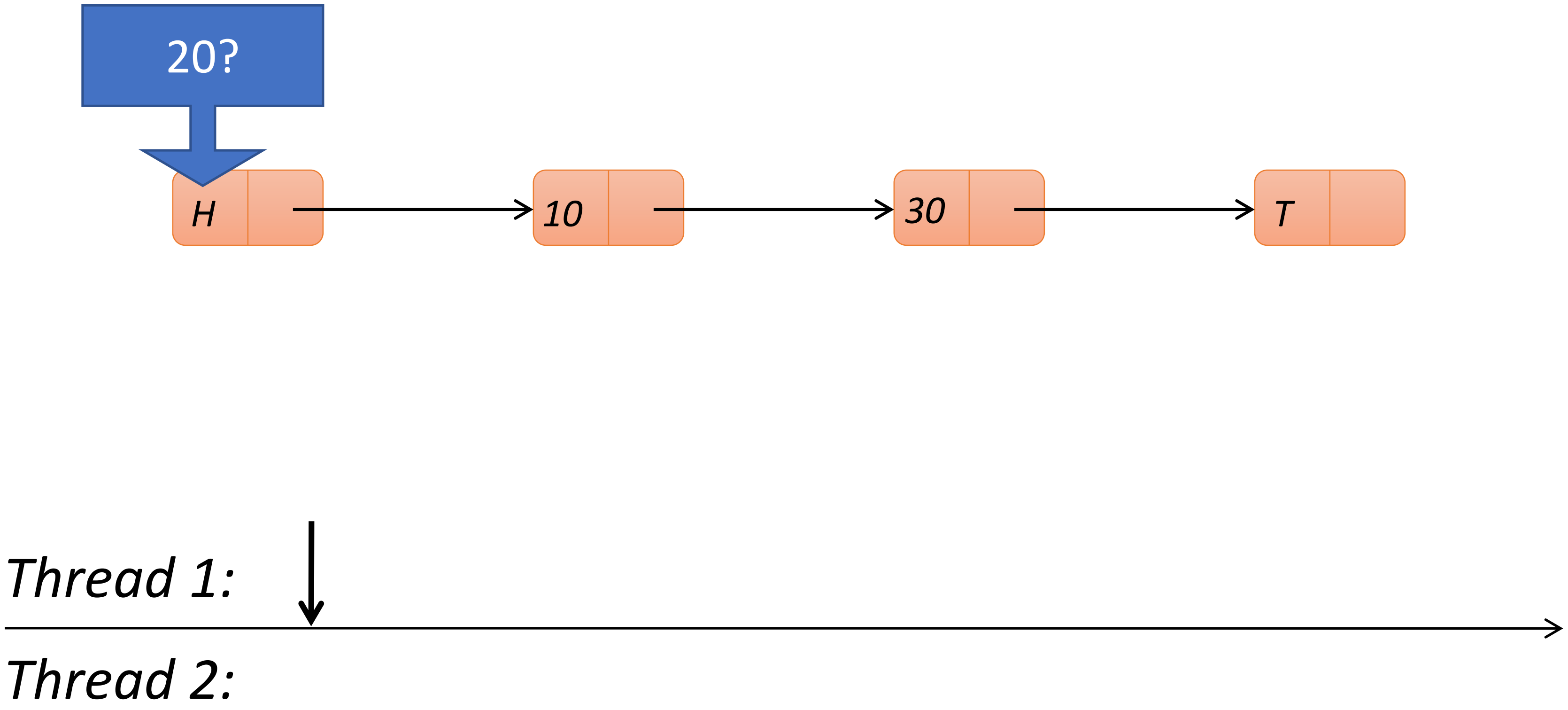


Thread 1:

Thread 2:

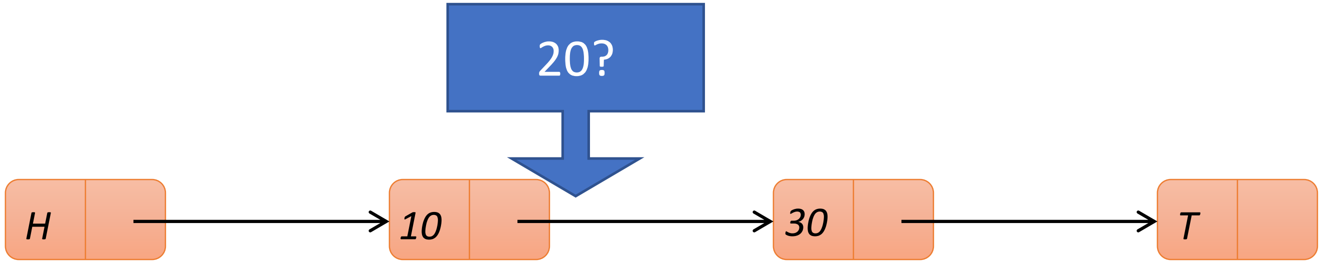
Example Revisited

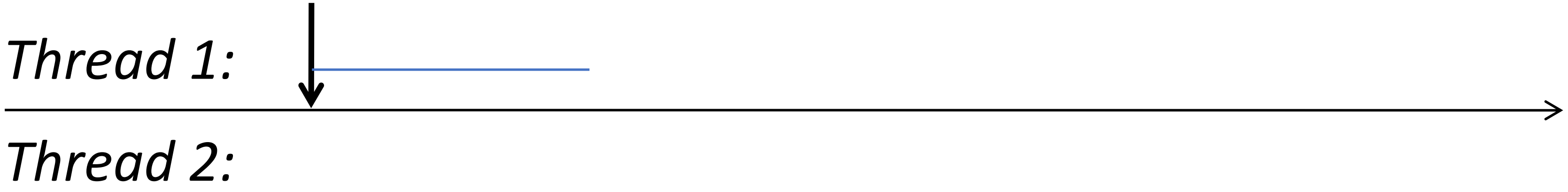
- find(20)



Example Revisited

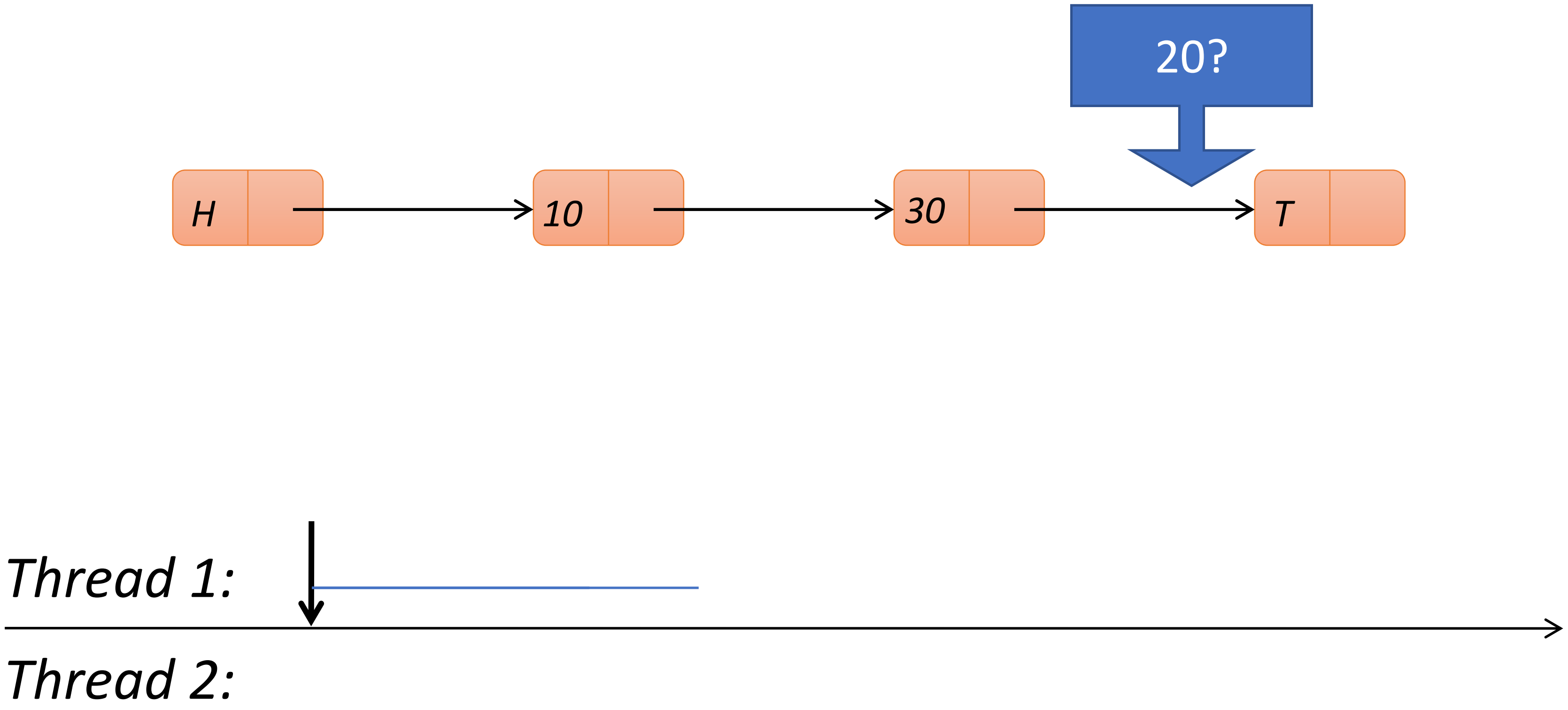
- find(20)





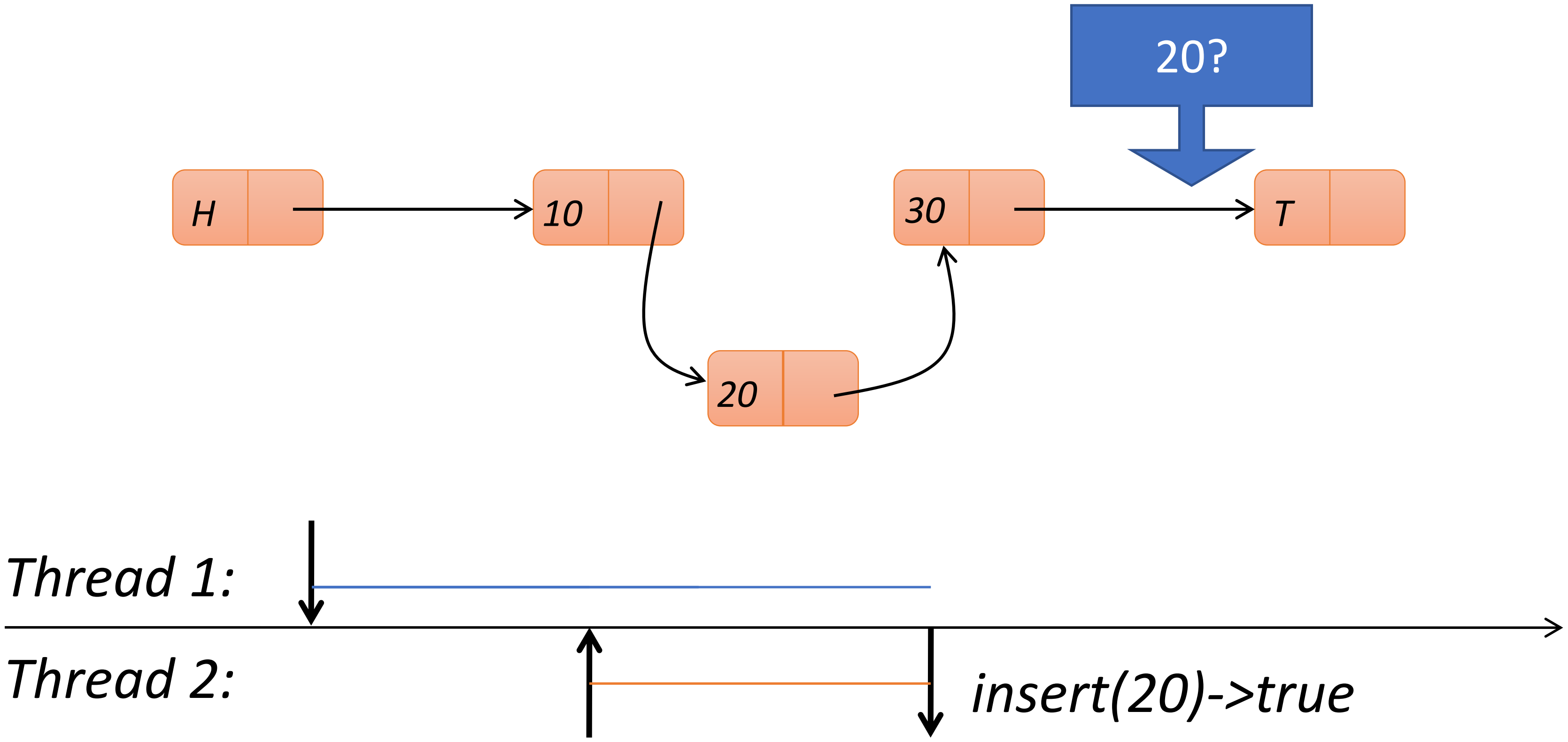
Example Revisited

- find(20)



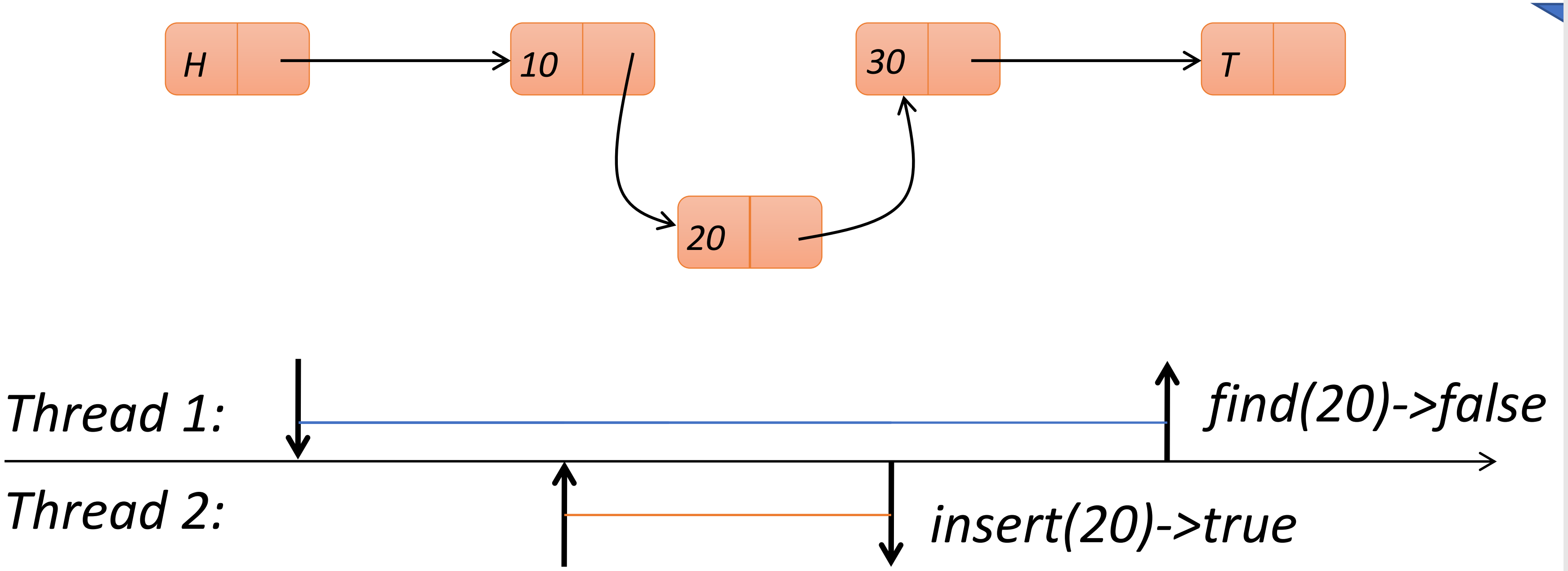
Example Revisited

- find(20)
- insert(20) -> true



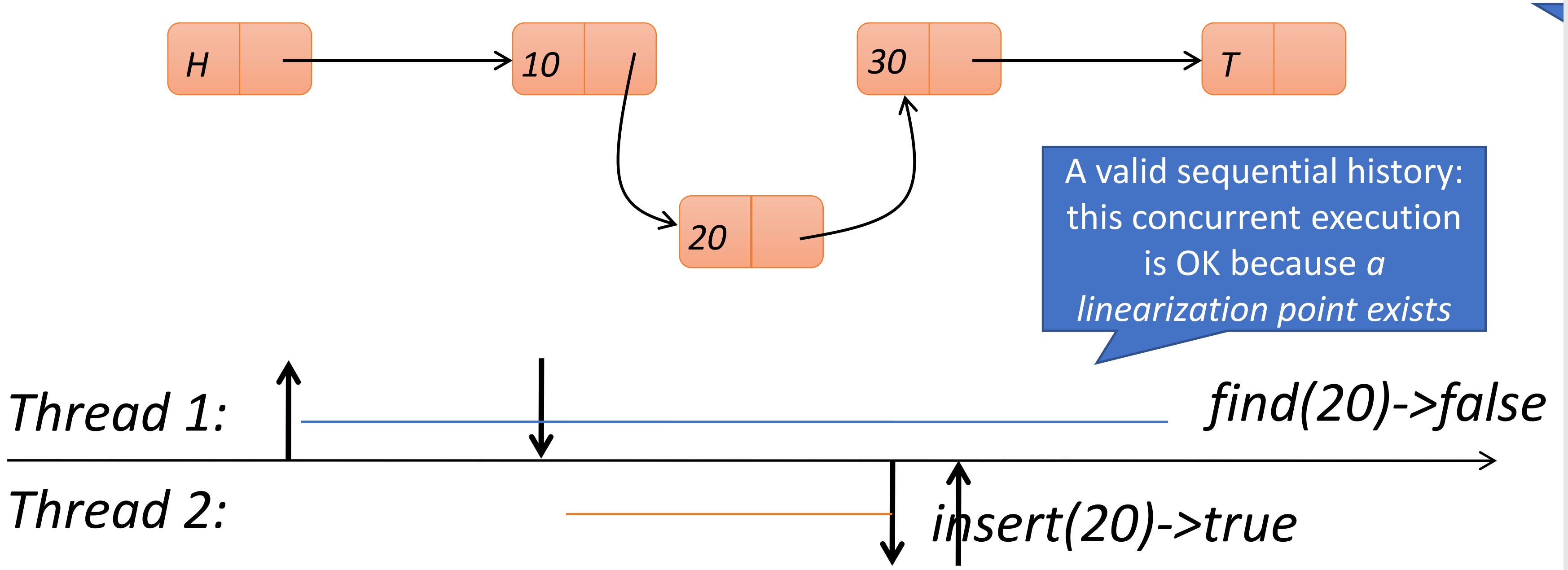
Example Revisited

- find(20) -> false
- insert(20) -> true



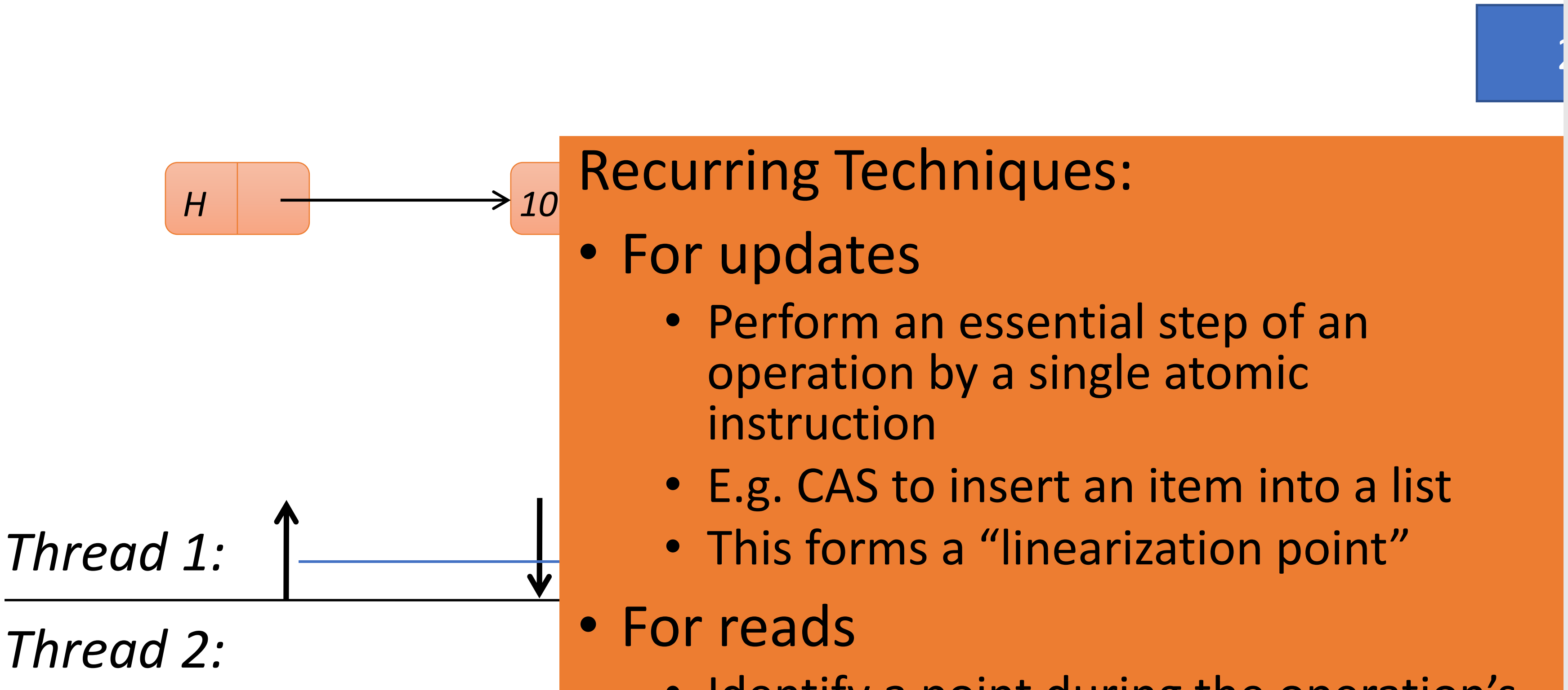
Example Revisited

- find(20) -> false
- insert(20) -> true



Example Revisited

- find(20) -> false
- insert(20) -> true



Recurring Techniques:

- For updates
  - Perform an essential step of an operation by a single atomic instruction
  - E.g. CAS to insert an item into a list
  - This forms a “linearization point”
- For reads
  - Identify a point during the operation’s execution when the result is valid
  - Not always a specific instruction

Formal Properties

Formal Properties

- Wait-free
  - A thread finishes its own operation if it continues executing steps
  - Strong: everyone eventually finishes

Formal Properties

- Wait-free
  - A thread finishes its own operation if it continues executing steps
  - Strong: everyone eventually finishes
- Lock-free
  - Some thread finishes its operation if threads continue taking steps
  - Weaker: some forward progress guaranateed, but admits unfairness, live-lock, etc.

Formal Properties

- Wait-free
  - A thread finishes its own operation if it continues executing steps
  - Strong: everyone eventually finishes
- Lock-free
  - Some thread finishes its operation if threads continue taking steps
  - Weaker: some forward progress guaranateed, but admits unfairness, live-lock, etc.
- Obstruction-free
  - A thread finishes its own operation if it runs in isolation
  - Very weak. Means if you remove contention, someone finishes

Formal Properties

- Wait-free
  - A thread finishes its own operation if it continues executing steps
  - Strong: everyone eventually finishes
- Lock-free
  - Some thread finishes its operation if threads continue taking steps

ranateed, but admits unfairness,

f it runs in isolation

ntention, someone finishes

Blocking
1. Blocking
2. Starvation-Free

Obstruction-Free

3. Obstruction-Free

Lock-Free

4. Lock-Free (LF)

Wait-Free

5. Wait-Free (WF)
6. Wait-Free Bounded (WFB)
7. Wait-Free Population Oblivious (WFPO)

stronger

Formal Properties

- Wait-free
  - A thread finishes its own operation if it continues executing steps
  - Strong: everyone eventually finishes
- Lock-free
  - Some thread finishes its operation if threads continue taking steps

ranat

f it ru

ntent

Blocking

1. Blocking
2. Starvation-Free

Obstruction-Free

3. Obstruction-Free

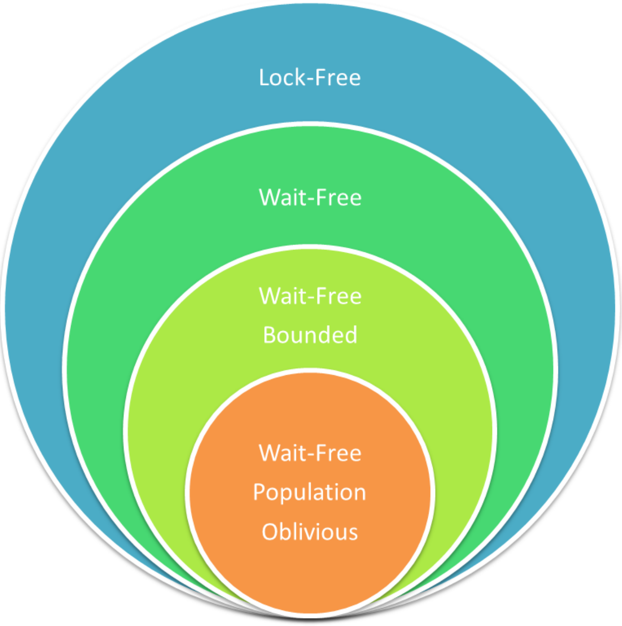
Lock-Free

4. Lock-Free (LF)

Wait-Free

5. Wait-Free (WF)
6. Wait-Free Bounded (WFB)
7. Wait-Free Population Oblivious (WFPO)

stronger



Linearizability Properties

Linearizability Properties

- non-blocking
  - one method is never forced to wait to sync with another.

Linearizability Properties

- non-blocking
  - one method is never forced to wait to sync with another.
- local property:
  - a system is linearizable iff each individual object is linearizable.
  - gives us **composability**.
  -

Linearizability Properties

- non-blocking
  - one method is never forced to wait to sync with another.
- local property:
  - a system is linearizable iff each individual object is linearizable.
  - gives us **composability**.
- Why is it important?
  - Serializability is not composable.

Linearizability Properties

- non-blocking
  - one method is never forced to wait to sync with another.
- local property:
  - a system is linearizable iff each individual object is linearizable.
  - gives us **composability**.
- Why is it important?
  - Serializability is not composable.

Composability again!

Practical difficulties:

- Key-value mapping
- Population count
- Iteration
- Resizing the bucket array

Practical difficulties:

- Key-va
- Popu
- Itera
- Resi

Options to consider when implementing a “difficult” operation:

Practical difficulties:

- Key-va
- Popu
- Itera
- Resi

Options to consider when implementing a “difficult” operation:

Relax the semantics
(e.g., non-exact count, or non-linearizable count)

Practical difficulties:

- Key-va
- Popu
- Itera
- Resi

Options to consider when implementing a "difficult" operation:

Relax the semantics
(e.g., non-exact count, or non-linearizable count)

Fall back to a simple implementation if permitted
(e.g., lock the whole table for resize)

Practical difficulties:

- Key-va
- Popu
- Itera
- Resi

Options to consider when implementing a "difficult" operation:

Relax the semantics
(e.g., non-exact count, or non-linearizable count)

Fall back to a simple implementation if permitted
(e.g., lock the whole table for resize)

Design a clever implementation
(e.g., split-ordered lists)

Practical difficulties:

- Key-va
- Popu
- Itera
- Resi

Options to consider when implementing a “difficult” operation:

Relax the semantics
(e.g., non-exact count, or non-linearizable count)

Fall back to a simple implementation if permitted
(e.g., lock the whole table for resize)

Design a clever implementation
(e.g., split-ordered lists)

Use a different data structure
(e.g., skip lists)

Summary

Lock free data structures can be super-fast

Based on clever algorithmic tricks and HW atomics

Corner cases often hard to get right

Good tool for the toolbox, use conservatively.

Backups…

Formal Properties

Formal Properties

- Wait-free

Formal Properties

- Wait-free
  - A thread finishes its own operation if it continues executing steps

Formal Properties

- Wait-free
  - A thread finishes its own operation if it continues executing steps
  - Strong: everyone eventually finishes

Formal Properties

- Wait-free
  - A thread finishes its own operation if it continues executing steps
  - Strong: everyone eventually finishes
- Lock-free

Formal Properties

- Wait-free
  - A thread finishes its own operation if it continues executing steps
  - Strong: everyone eventually finishes
- Lock-free
  - Some thread finishes its operation if threads continue taking steps

Formal Properties

- Wait-free
  - A thread finishes its own operation if it continues executing steps
  - Strong: everyone eventually finishes
- Lock-free
  - Some thread finishes its operation if threads continue taking steps
  - Weaker: some forward progress guaranateed, but admits unfairness, live-lock, etc.

Formal Properties

- Wait-free
  - A thread finishes its own operation if it continues executing steps
  - Strong: everyone eventually finishes
- Lock-free
  - Some thread finishes its operation if threads continue taking steps
  - Weaker: some forward progress guaranateed, but admits unfairness, live-lock, etc.
- Obstruction-free

Formal Properties

- Wait-free
  - A thread finishes its own operation if it continues executing steps
  - Strong: everyone eventually finishes
- Lock-free
  - Some thread finishes its operation if threads continue taking steps
  - Weaker: some forward progress guaranateed, but admits unfairness, live-lock, etc.
- Obstruction-free
  - A thread finishes its own operation if it runs in isolation

Formal Properties

- Wait-free
  - A thread finishes its own operation if it continues executing steps
  - Strong: everyone eventually finishes
- Lock-free
  - Some thread finishes its operation if threads continue taking steps
  - Weaker: some forward progress guaranateed, but admits unfairness, live-lock, etc.
- Obstruction-free
  - A thread finishes its own operation if it runs in isolation
  - Very weak. Means if you remove contention, someone finishes