Fast Parallel Programming: Lock Freedom

cs378h

Today

Questions?

Administrivia

- Project presentations?

Agenda:

- Lock Freedom

Review: Sequential Consistency

- weaker than strict/strong consistency
  - All operations are executed in *some* sequential order
  - each process issues operations in program order
    - Any valid interleaving is allowed
    - All agree on the same interleaving
    - Each process preserves its program order

P1:	W(x)a				
P2:		W(x)b			
P3:			R(x)b		R(x)a
P4:				R(x)b	R(x)a

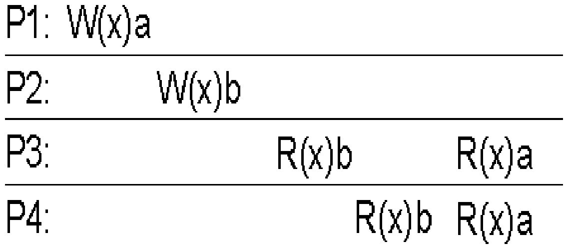
(a)

P1:	W(x)a				
P2:		W(x)b			
P3:			R(x)b		R(x)a
P4:				R(x)a	R(x)b

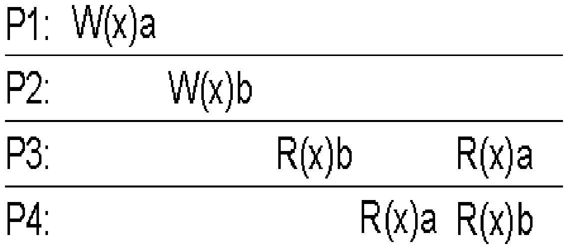
(b)

Review: Sequential Consistency

- weaker than strict/strong consistency
  - All operations are executed in *some* sequential order
  - each process issues operations in program order
    - Any valid interleaving is allowed
    - All agree on the same interleaving
    - Each process preserves its program order



)

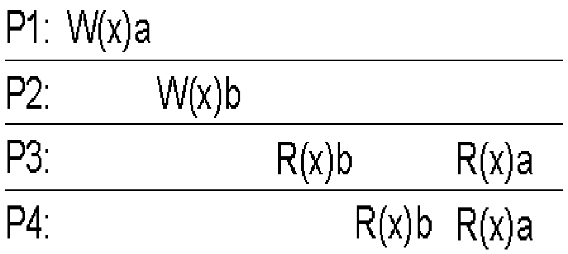


(b)

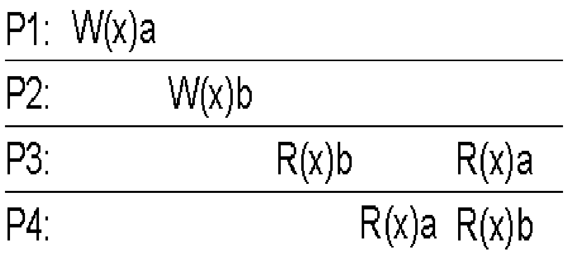
- ***Why is this weaker than strict/strong?***

Review: Sequential Consistency

- weaker than strict/strong consistency
  - All operations are executed in *some* sequential order
  - each process issues operations in program order
    - Any valid interleaving is allowed
    - All agree on the same interleaving
    - Each process preserves its program order



)



(b)

- ***Why is this weaker than strict/strong?***
- ***Nothing is said about "most recent write"***

Review: Causal consistency

Review: Causal consistency

- Causally related writes seen by all processes in same order.

Review: Causal consistency

- Causally related writes seen by all processes in same order.
  - *Causally?*

Review: Causal consistency

- Causally related writes seen
  - *Causally?*

Causal:

If a write produces a value that causes another write, they are causally related

```
X = 1
if(X > 0) {
    Y = 1
}
```

Causal consistency → all see X=1, Y=1 in same order

Review: Causal consistency

- Causally related writes seen by all processes in same order.
  - *Causally?*

Review: Causal consistency

- Causally related writes seen by all processes in same order.
  - *Causally?*
  - *Concurrent* writes may be seen in different orders on different machines

Review: Causal consistency

- Causally related writes seen by all processes in same order.
  - *Causally?*
  - *Concurrent* writes may be seen in different orders on different machines

P1:	W(x)a				
P2:		R(x)a	W(x)b		
P3:				R(x)b	R(x)a
P4:				R(x)a	R(x)b

(a)

Review: Causal consistency

- Causally related writes seen by all processes in same order.
  - *Causally?*
  - *Concurrent* writes may be seen in different orders on different machines

P1:	W(x)a				
P2:		R(x)a	W(x)b		
P3:				R(x)b	R(x)a
P4:				R(x)a	R(x)b

(a)

Not permitted

Review: Causal consistency

- Causally related writes seen by all processes in same order.
  - *Causally?*
  - *Concurrent* writes may be seen in different orders on different machines

P1:	W(x)a				
P2:		R(x)a	W(x)b		
P3:				R(x)b	R(x)a
P4:				R(x)a	R(x)b

(a)

P1:	W(x)a			
P2:		W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

(b)

Not permitted

Review: Causal consistency

- Causally related writes seen by all processes in same order.
  - *Causally?*
  - *Concurrent* writes may be seen in different orders on different machines

P1:	W(x)a				
P2:		R(x)a	W(x)b		
P3:				R(x)b	R(x)a
P4:				R(x)a	R(x)b

(a)

Not permitted

P1:	W(x)a			
P2:		W(x)b		
P3:			R(x)b	R(x)a
P4:			R(x)a	R(x)b

(b)

Permitted

Review: Linearizability

Review: Linearizability

- Assumes sequential consistency *and*
  - If TS(x) < TS(y) then OP(x) should precede OP(y) in the sequence
  - Stronger than sequential consistency
  - Difference between linearizability and serializability?
    - Granularity: reads/writes versus transactions

Review: Linearizability

- Assumes sequential consistency *and*
  - If TS(x) < TS(y) then OP(x) should precede OP(y) in the sequence
  - Stronger than sequential consistency
  - Difference between linearizability and serializability?
    - Granularity: reads/writes versus transactions
- Example:
  - Stay tuned…relevant for lock free data structures
  - Importantly: *a property of concurrent objects*

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

```
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;
       return;
    }
    while (last->next != NULL)
        last = last->next;
    last->next = new_node;
}
```

Basic 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;
       return;
    }
    while (last->next != NULL)
        last = last->next;
    last->next = new_node;
}
```

- Is this thread safe?

Basic 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;
       return;
    }
    while (last->next != NULL)
        last = last->next;
    last->next = new_node;
}
```

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

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();
}
```

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;
    }

}
```

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;
    }

}
```

- What 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;
    }

}
```

- What 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;
    }

}
```

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

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;
    }

}
```

- What 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
 data;
uct 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;
    }
}
```

e?

nsure?

- Can we ensure consistent view (invariants hold) sans mutual 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);
```

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"

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

Optimistic Synchronization: MP-SC

```
AddWrap(x,n):
    x += n;
    if(x >= Qsize) x -= Qsize
    return x;

SpaceLeft(h):
    t = Q_tail;
    if(h >= t)  return t-h-1+Q_size;
    else  return t-h-1;

Q_put(data,N):
    do {
        h = Q_head;
        h1 = AddWrap(h,N);
    } while(Spaceleft(h) >= N
          && cas(Q_head,h,h1) == FAIL);
    for(i=0; i<N; i++) {
        Q_buf[ AddWrap(h,i) ] = data[i];
        Q_flag[ AddWrap(h,i) ] = 1;
    }
```

- Where is the “optimism” here?
- Why/when does this work?

Optimistic Synchronization: MP-SC

```
AddWrap(x,n):
    x += n;
    if(x >= Qsize) x -= Qsize
    return x;

SpaceLeft(h):
    t = Q_tail;
    if(h >= t)  return t-h-1+Q_size;
    else  return t-h-1;

Q_put(data,N):
    do {
        h = Q_head;
        h1 = AddWrap(h,N);
    } while(Spaceleft(h) >= N
            && cas(Q_head,h,h1) == FAIL);
    for(i=0; i<N; i++) {
        Q_buf[ AddWrap(h,i) ] = data[i];
        Q_flag[ AddWrap(h,i) ] = 1;
    }
```

1. CAS used to reserve space
2. Q_flags is last write in Q_put, acting as atomic commit
3. *single* c only
4. Requires fence between Q_buf and Q_flag set
5. We don't get to see Q_get code

- Where is the "optimism" here?
- Why/when does this 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;
            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;
}
```

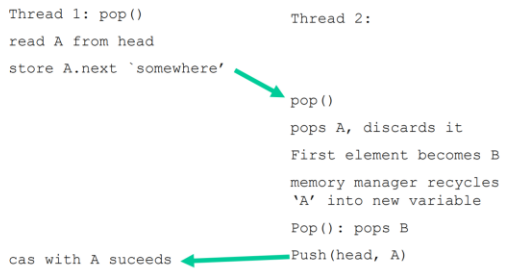
- Why does is it work?
- Does it enforce all invariants?

Lock-Free Stack: ABA Problem

Thread 1: pop()	Thread 2:
read A from head	
store A.next `somewhere’	
	pop()
	pops A, discards it
	First element becomes B
	memory manager recycles ‘A’ into new variable
	Pop(): pops B
cas with A suceeds	Push(head, A)

Lock-Free Stack: ABA Problem

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



Lock-Free Stack: ABA Problem

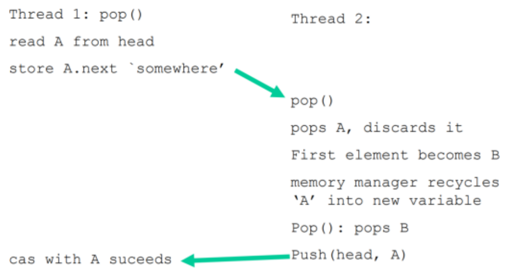
```
Node* pop() {
    Node* current = head;
    while(current) {
        if(cas(&head, current->next, current))
            return current;
        current = head;
    }
    return false;
}
```

Thread 1: pop()	Thread 2:
read A from head	
store A.next `somewhere’	
→	pop()
	pops A, discards it
	First element becomes B
	memory manager recycles ‘A’ into new variable
	Pop(): pops B
cas with A suceeds ←	Push(head, A)

Lock-Free Stack: ABA Problem

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

```
Node* pop() {
    Node* current = head;
    while(current) {
```



Lock-Free Stack: ABA Problem

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

```
Node* pop() {
    Node* current = head;
    while(current) {
```

```
Node * node = pop();
delete node;
node = new Node(blah_blah);
push(node);
```

Thread 1: pop()	Thread 2:
read A from head	
store A.next `somewhere' →	
	pop()
	pops A, discards it
	First element becomes B
	memory manager recycles 'A' into new variable
	Pop(): pops B
cas with A suceeds ←	Push(head, A)

Lock-Free Stack: ABA Problem

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

```
Node* pop() {
    Node* current = head;
    while(current) {



        if(cas(&head, current->next, current))
            return current;
        current = head;
    }
    return false;
}
```

```
Node * node = pop();
delete node;
node = new Node(blah_blah);
push(node);
```

Thread 1: pop()
read A from head
store A.next `somewhere'

Thread 2:
pop()
pops A, discards it
First element becomes B
memory manager recycles 'A' into new variable
Pop(): pops B
Push(head, A)

cas with A suceeds

Lock-Free Stack: ABA Problem

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

```
Node* pop() {
    Node* current = head;
    while(

        if(cas(&head, current->next, current))
            return current;
        current = head;
    }
    return false;
}
```

```
Node * node = pop();
delete node;
                Node(blah_blah);
```

Thread 1: pop()	Thread 2:
read A from head	
store A.next `somewhere'	
	pop()
	pops A, discards it
	First element becomes B
	memory manager recycles 'A' into new variable
	Pop(): pops B
cas with A suceeds	Push(head, A)

Lock-Free Stack: ABA Problem

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

```
Node* pop() {
    Node* current = head;
    while

        if(cas(&head, current->next,
            return current;
        current = head;
    }
    return false;
}
```

```
Node * node = pop();
delete node;
            Node(blah_blah);
```

Thread 1: pop()	Thread 2:
read A from head	
store A.next `somewhere’ →	
	pop()
	pops A, discards it
	First element becomes B
	memory manager recycles ‘A’ into new variable
	Pop(): pops B
cas with A suceeds ←	Push(head, A)

Lock-Free Stack: ABA Problem

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

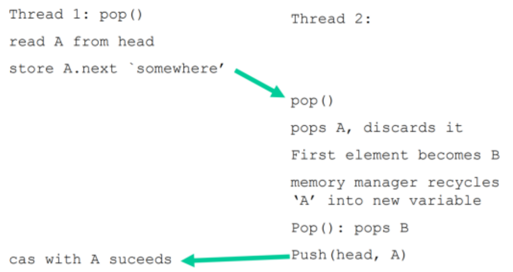
```
Node* pop() {
    Node* current = head;
    while

        if(cas(&head, current->next,
            return current;
        current = head;
    }
    return false;
}
```

```
Node * node = pop();
delete node;
              Node(blah_blah);
```

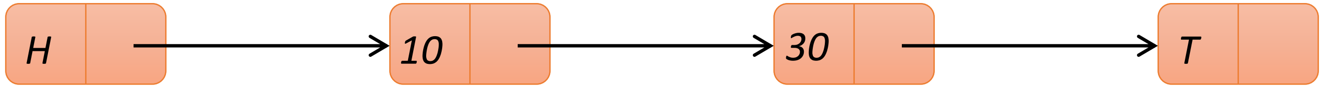
Fixes?

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



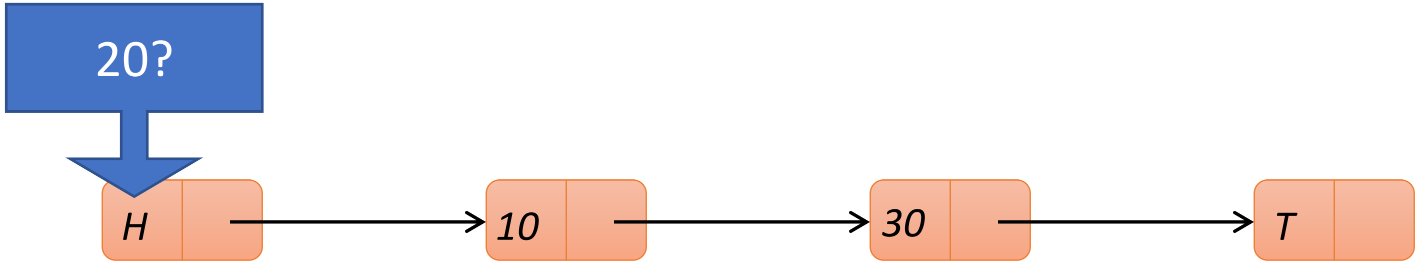
Correctness: Searching a sorted list

- find(20):



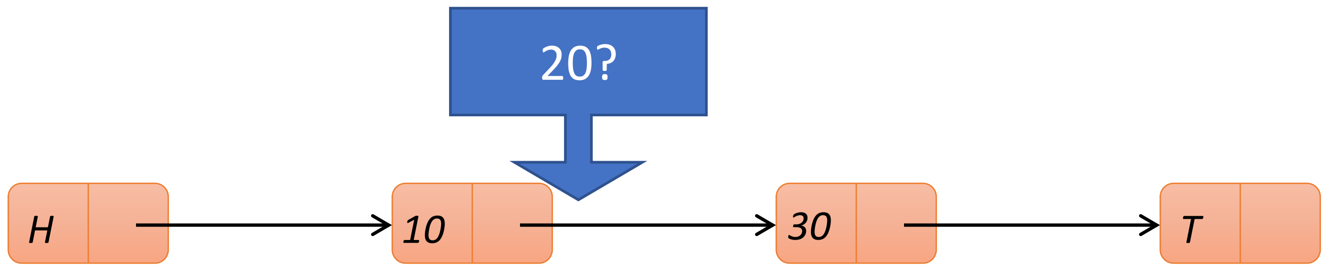
Correctness: Searching a sorted list

- find(20):



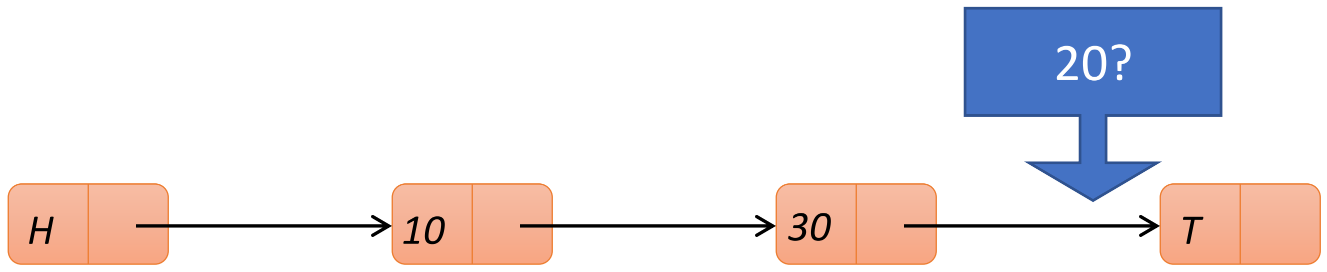
Correctness: Searching a sorted list

- find(20):



Correctness: Searching a sorted list

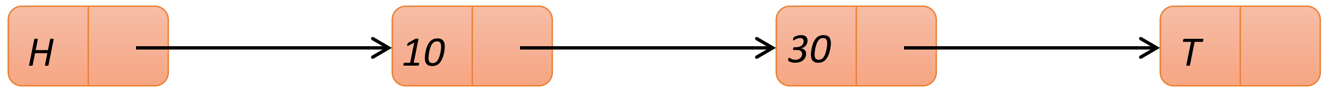
- find(20):



find(20) -> false

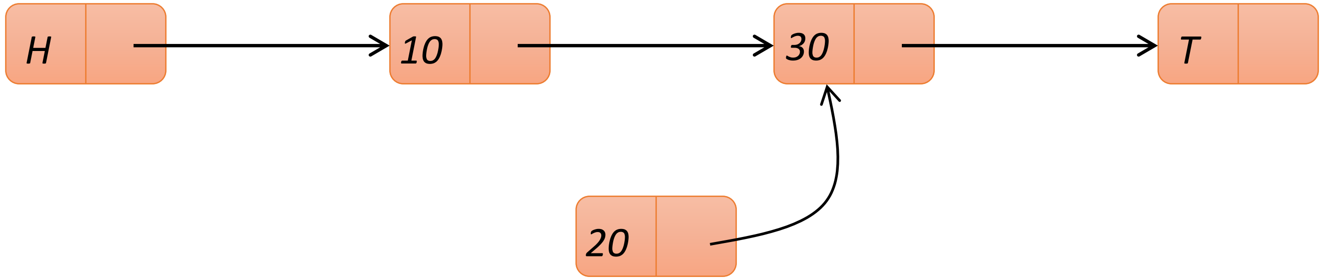
Inserting an item with CAS

- insert(20):



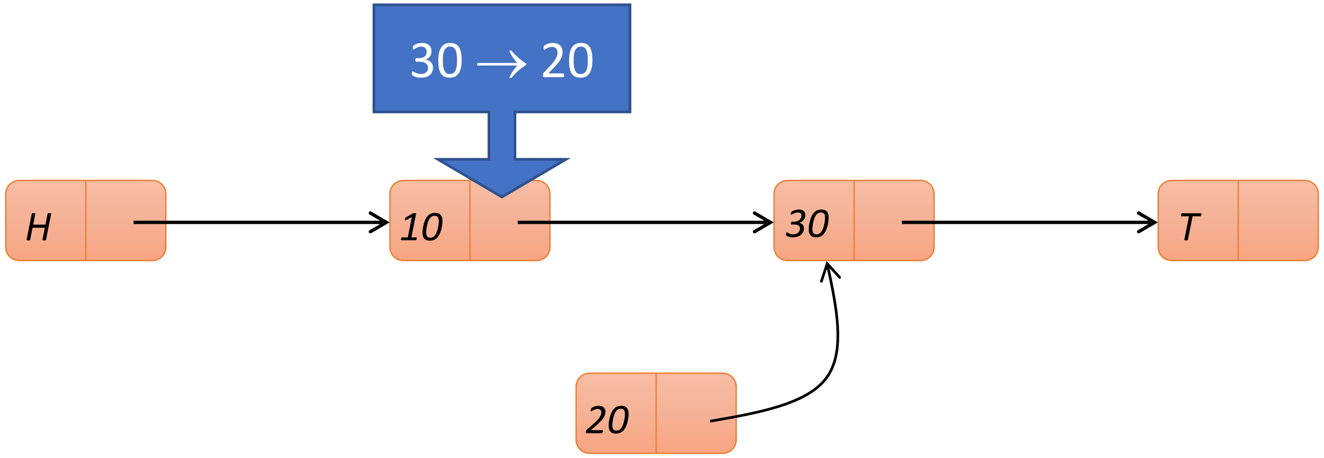
Inserting an item with CAS

- insert(20):



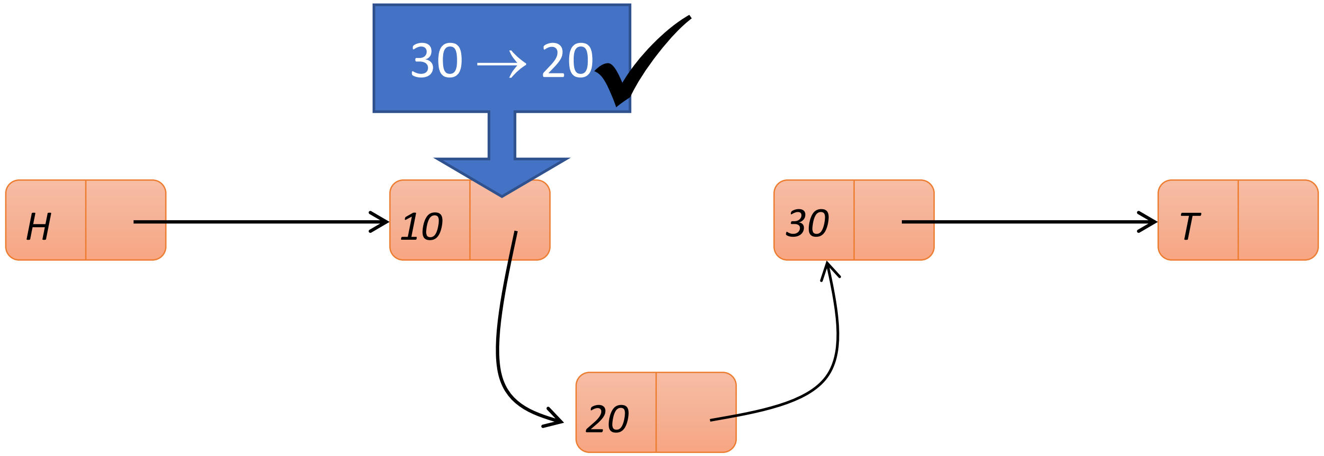
Inserting an item with CAS

- insert(20):



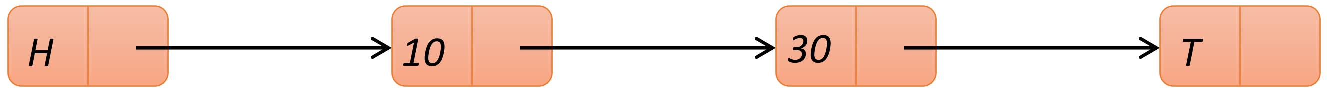
Inserting an item with CAS

- insert(20):



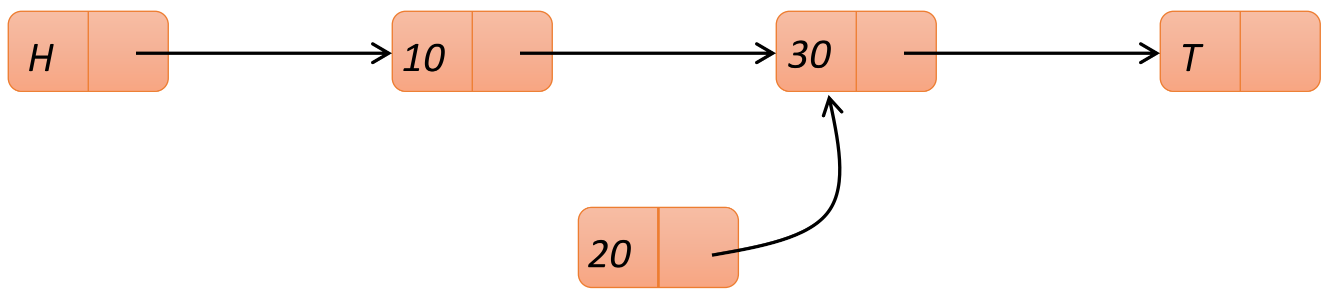
insert(20) -> true

Inserting an item with CAS



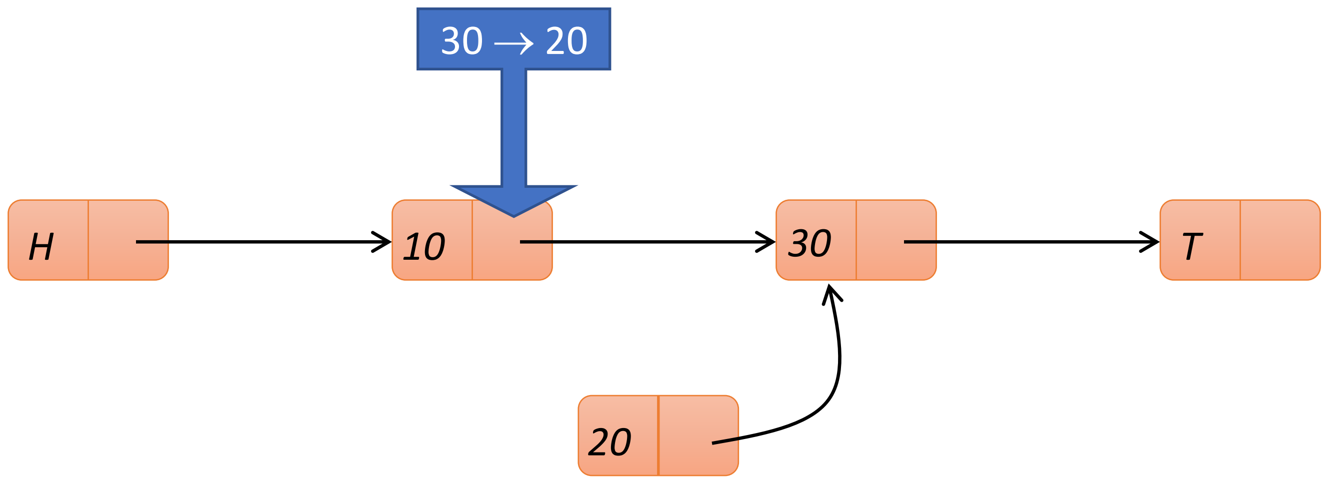
Inserting an item with CAS

- insert(20):



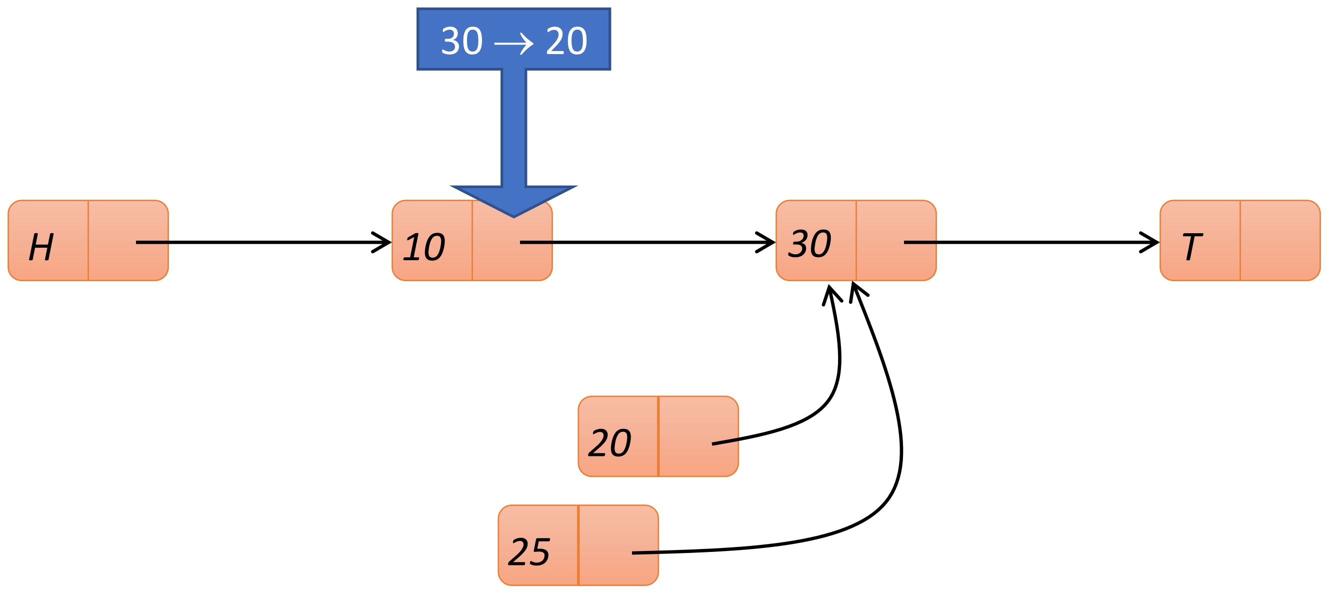
Inserting an item with CAS

- insert(20):



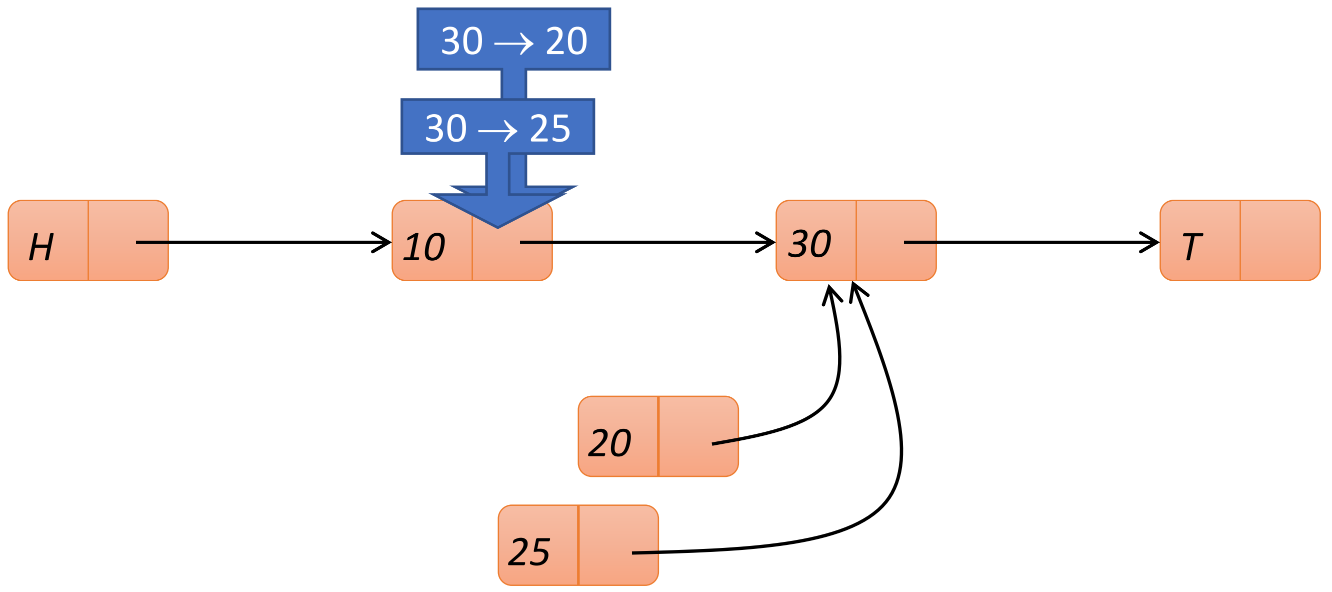
Inserting an item with CAS

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



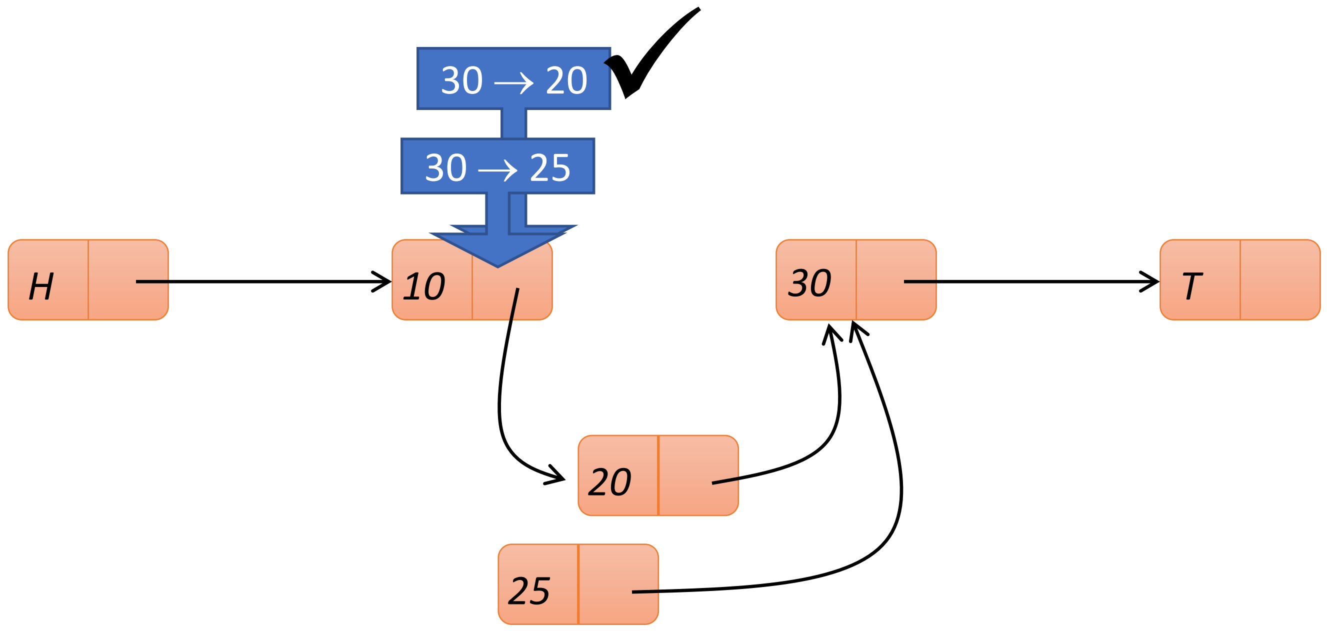
Inserting an item with CAS

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



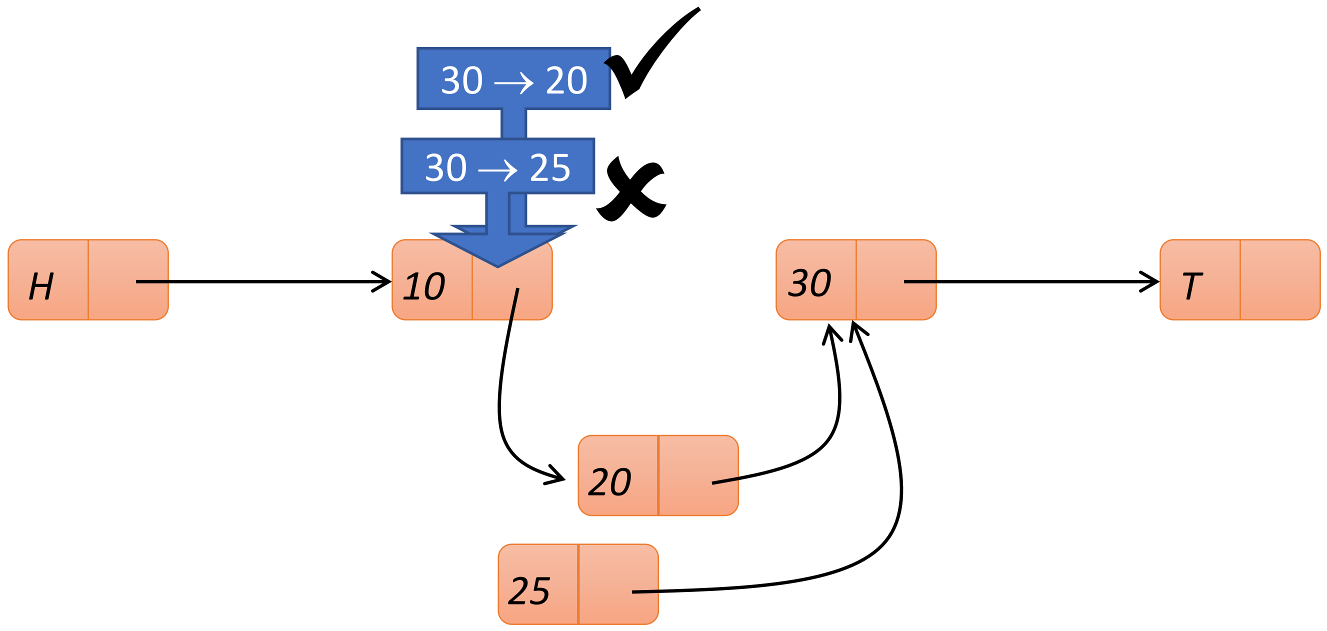
Inserting an item with CAS

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



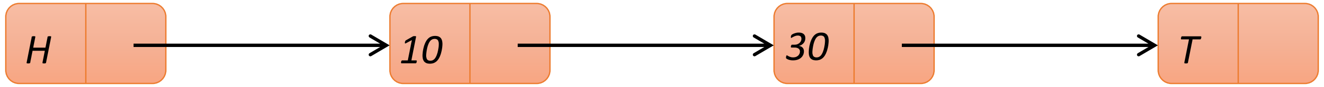
Inserting an item with CAS

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



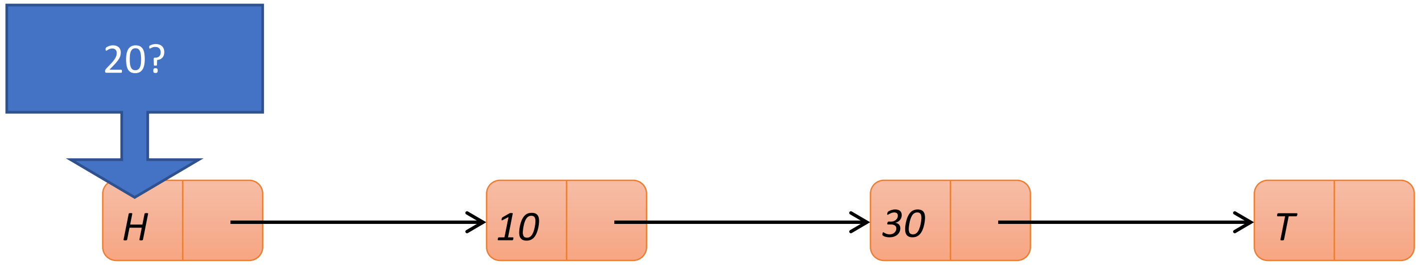
Searching and finding together

- find(20)



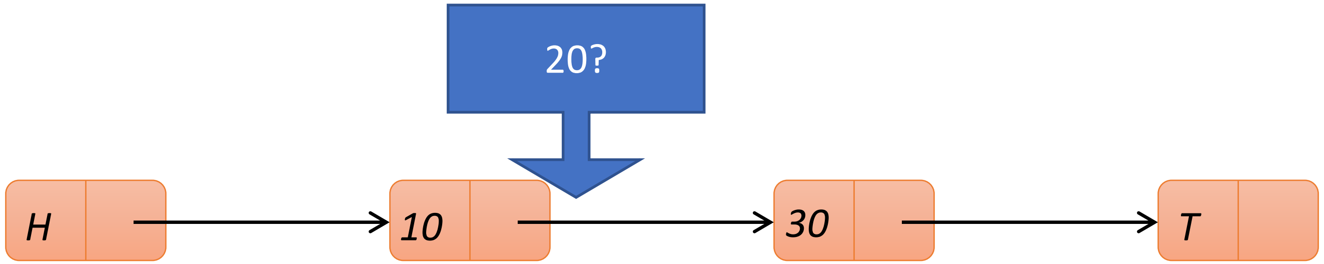
Searching and finding together

- find(20)



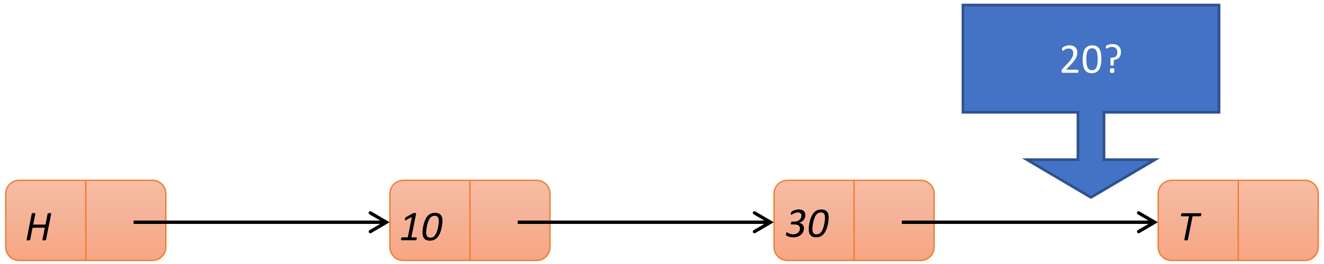
Searching and finding together

- find(20)



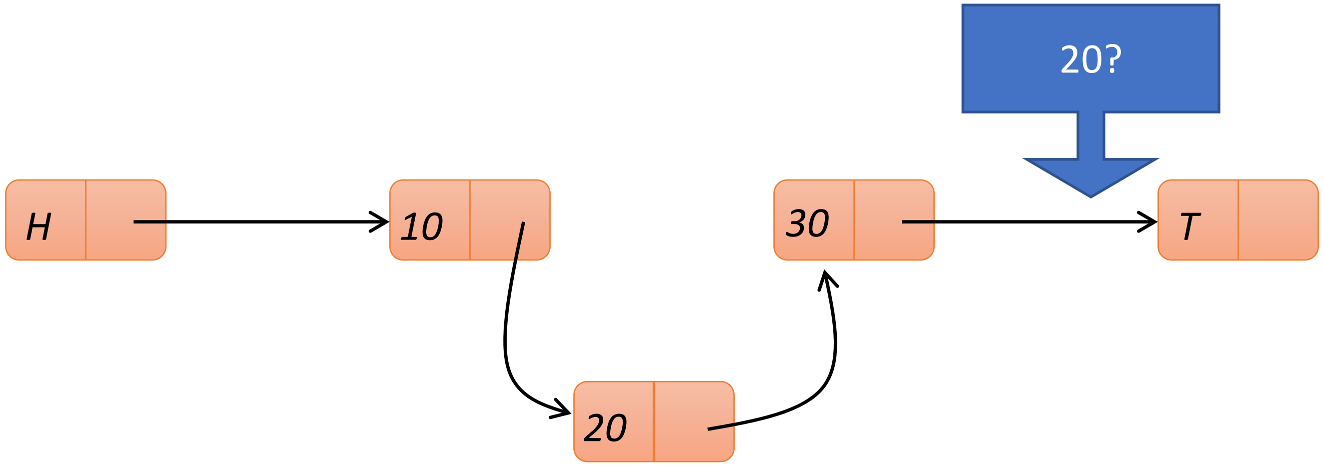
Searching and finding together

- find(20)



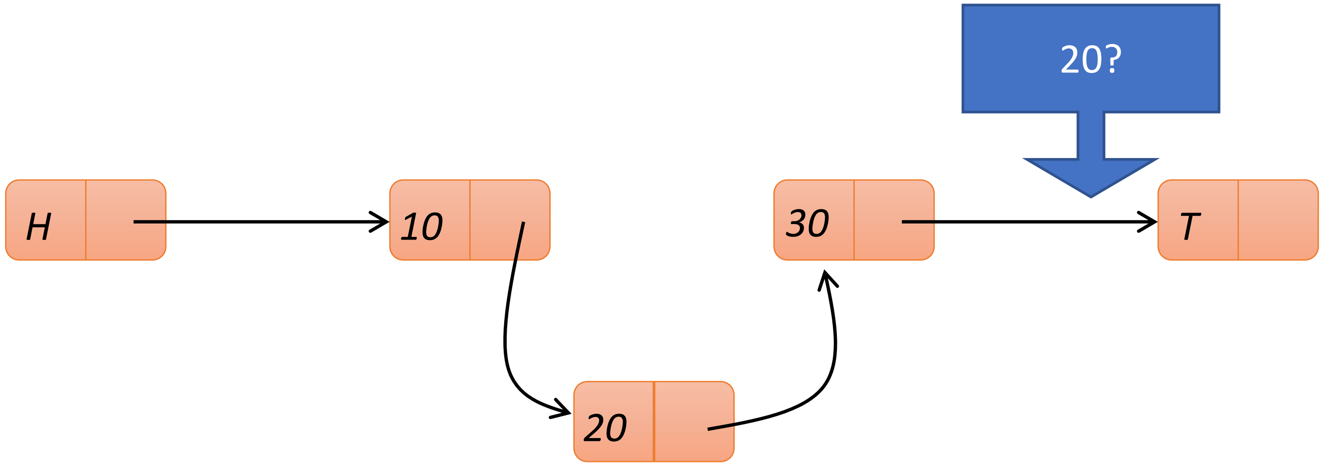
Searching and finding together

- 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 behaviour of the data structure

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

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

Sequential history

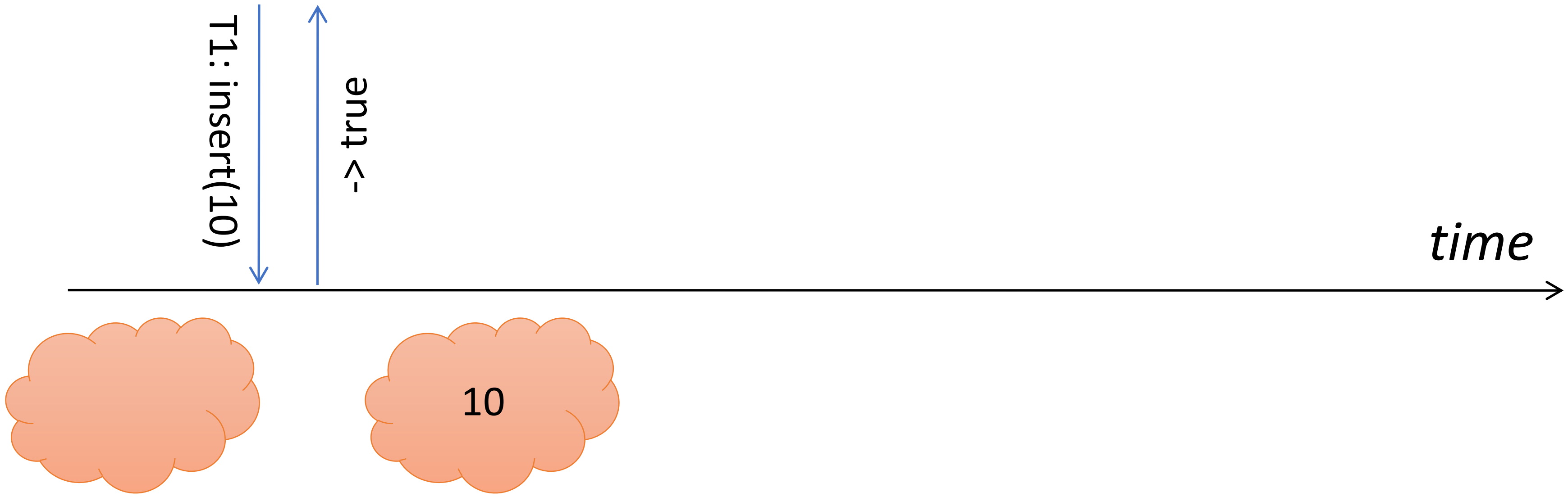
- No overlapping invocations

Sequential history

- No overlapping invocations

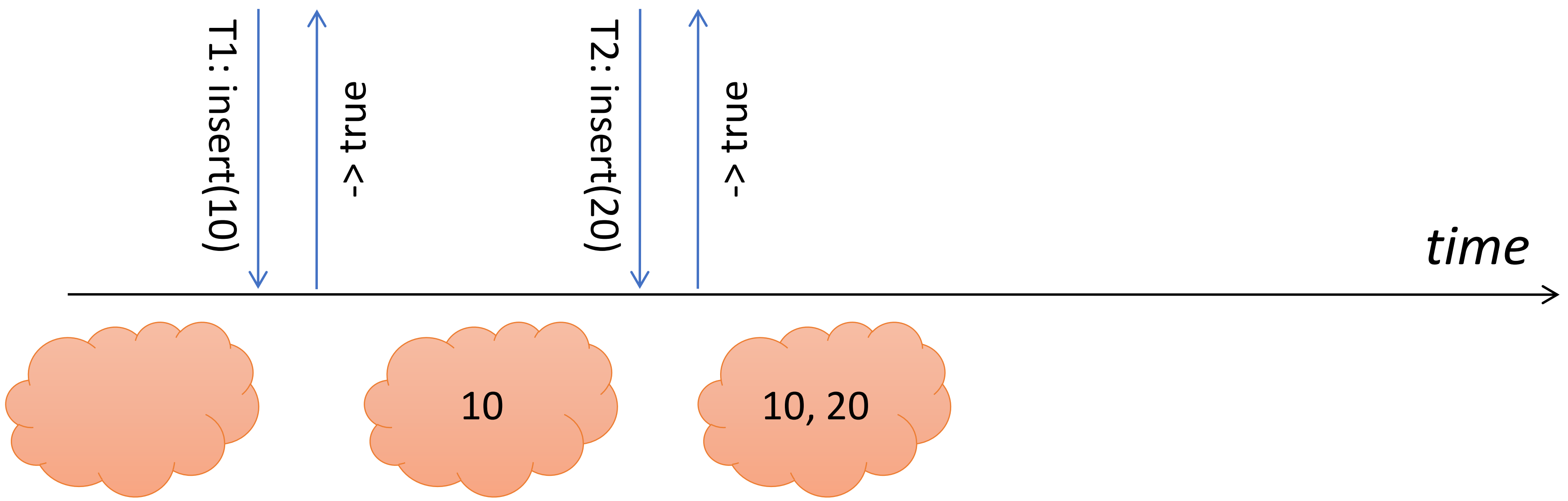
Sequential history

- No overlapping invocations



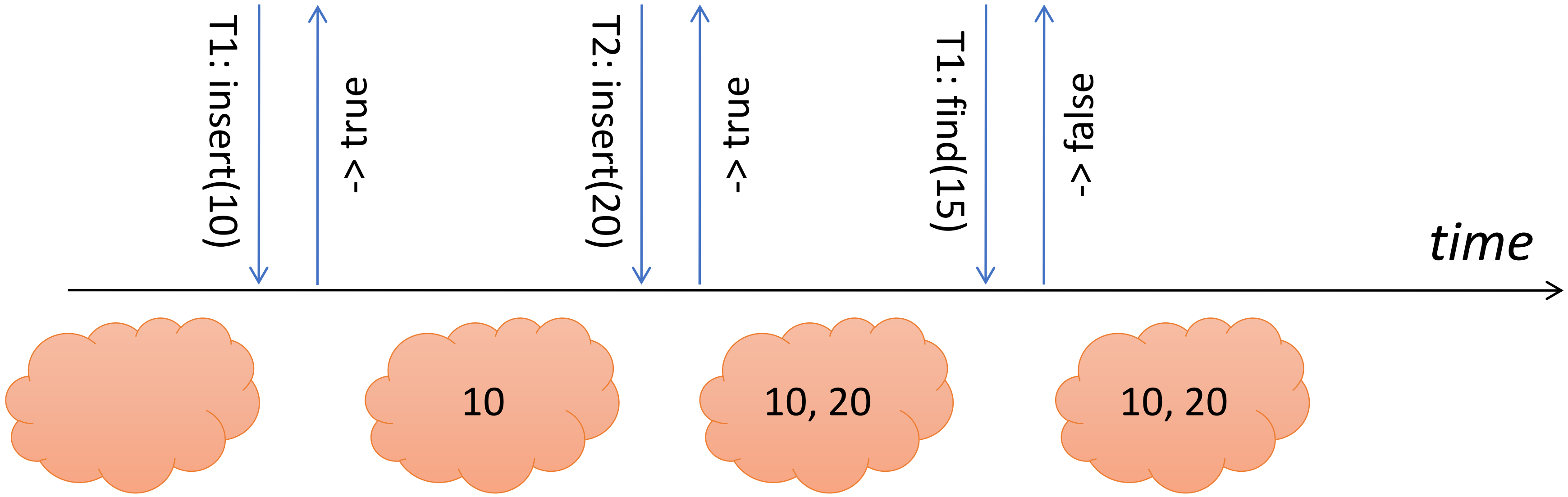
Sequential history

- No overlapping invocations



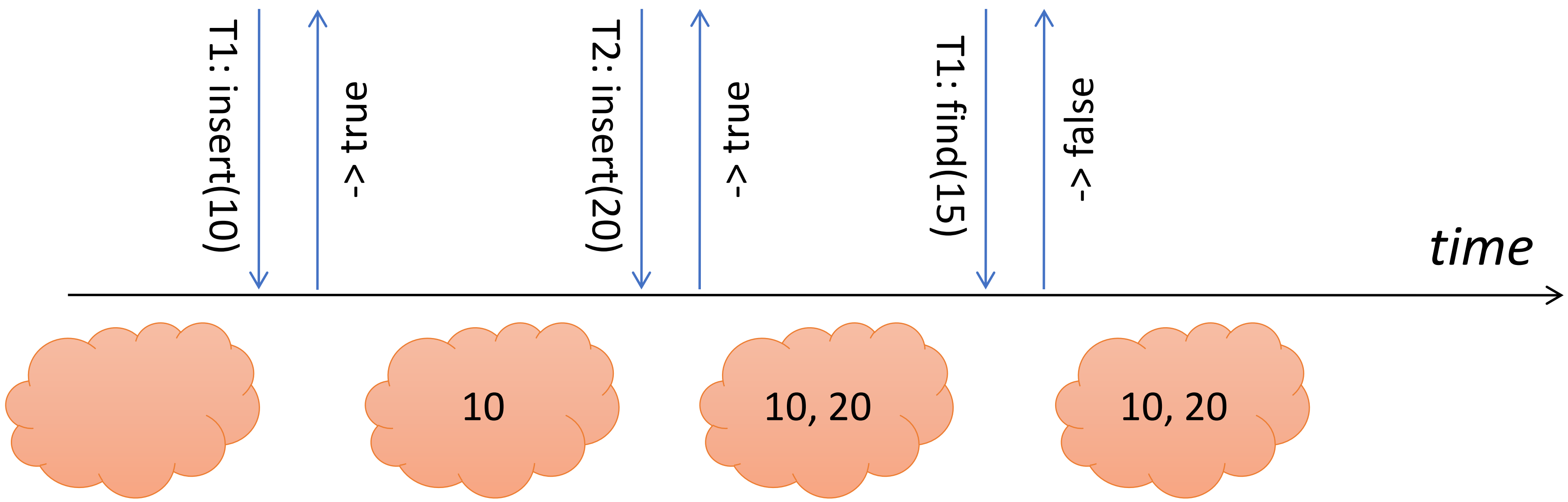
Sequential history

- No overlapping invocations



Sequential history

- No overlapping invocations

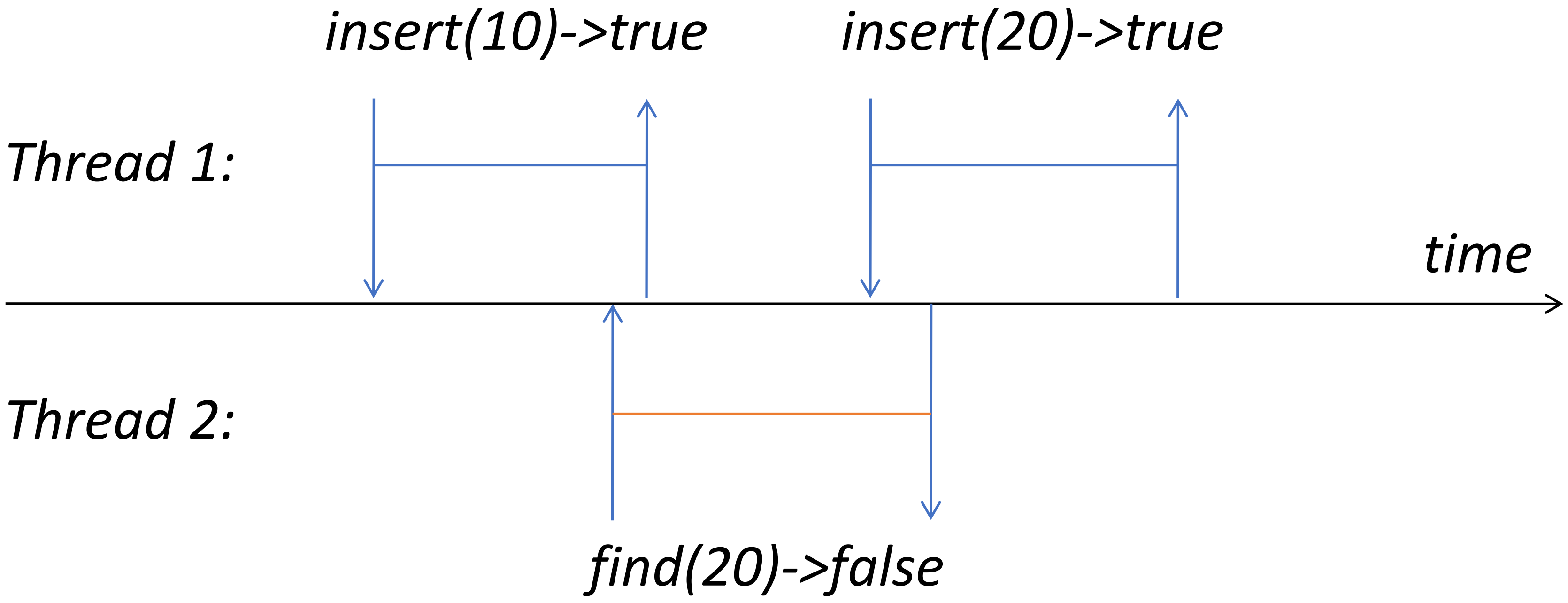


Linearizability: concurrent behaviour should be similar

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

Concurrent history

Allow *overlapping* invocations

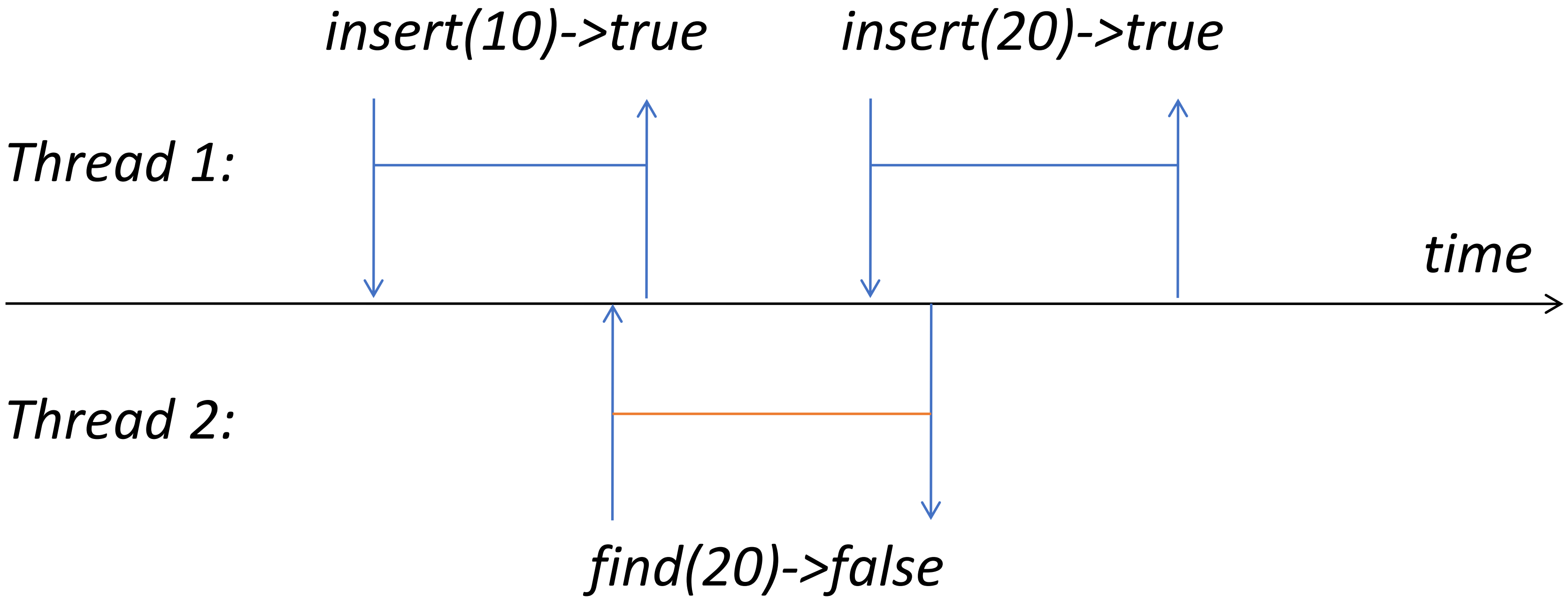


Concurrent history

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

Allow *overlapping* invocations

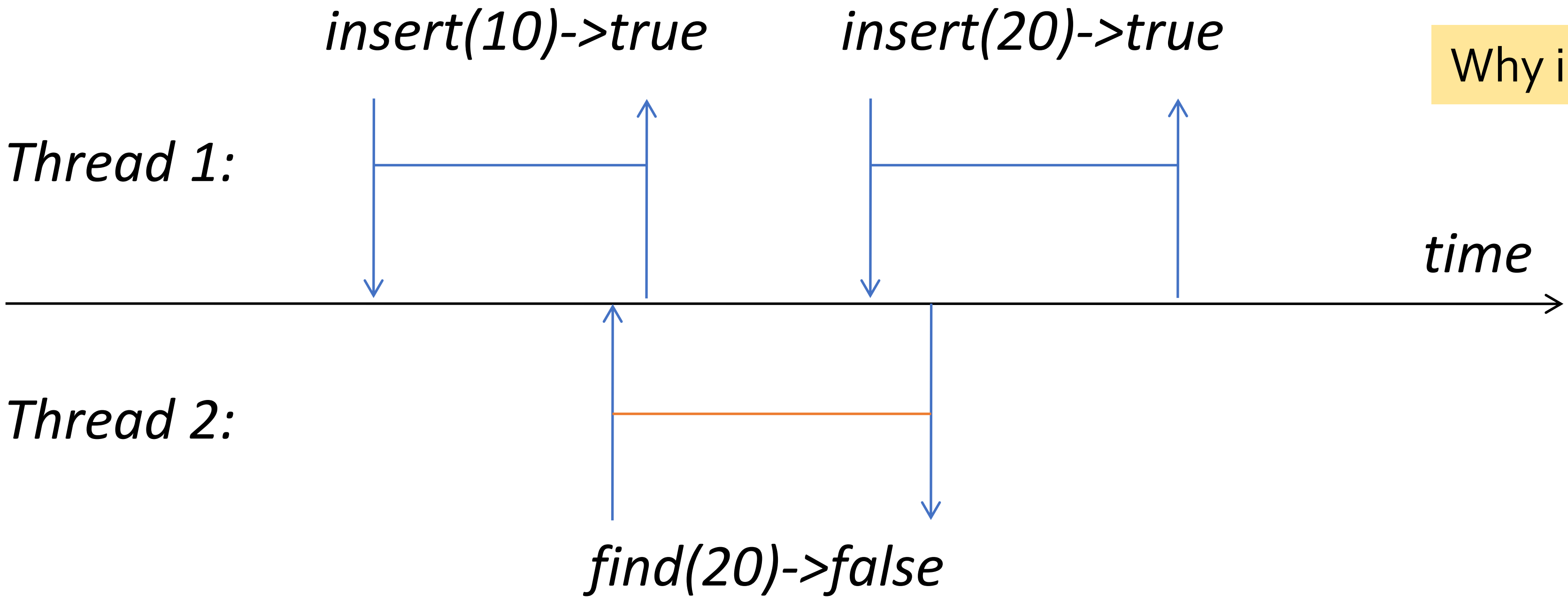


Concurrent history

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



Why is this one OK?

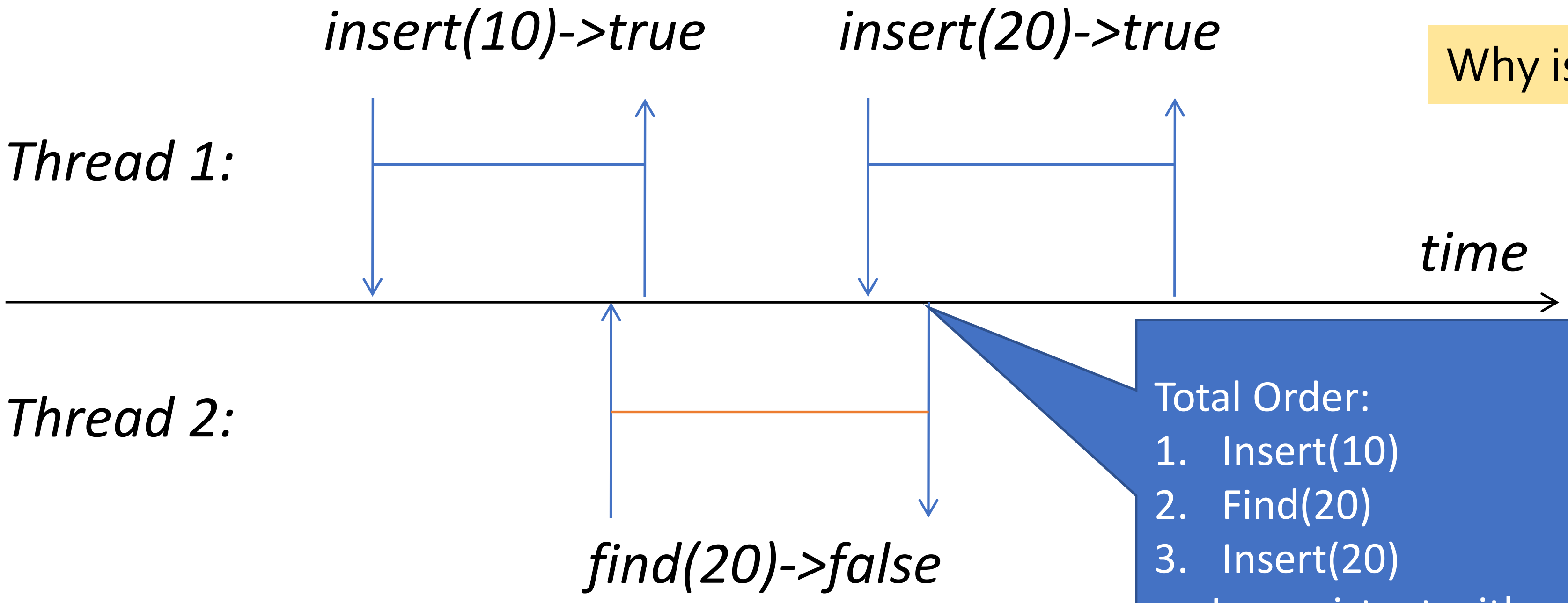
Concurrent history

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

Allow *overlapping* invocations

Why is this one OK?

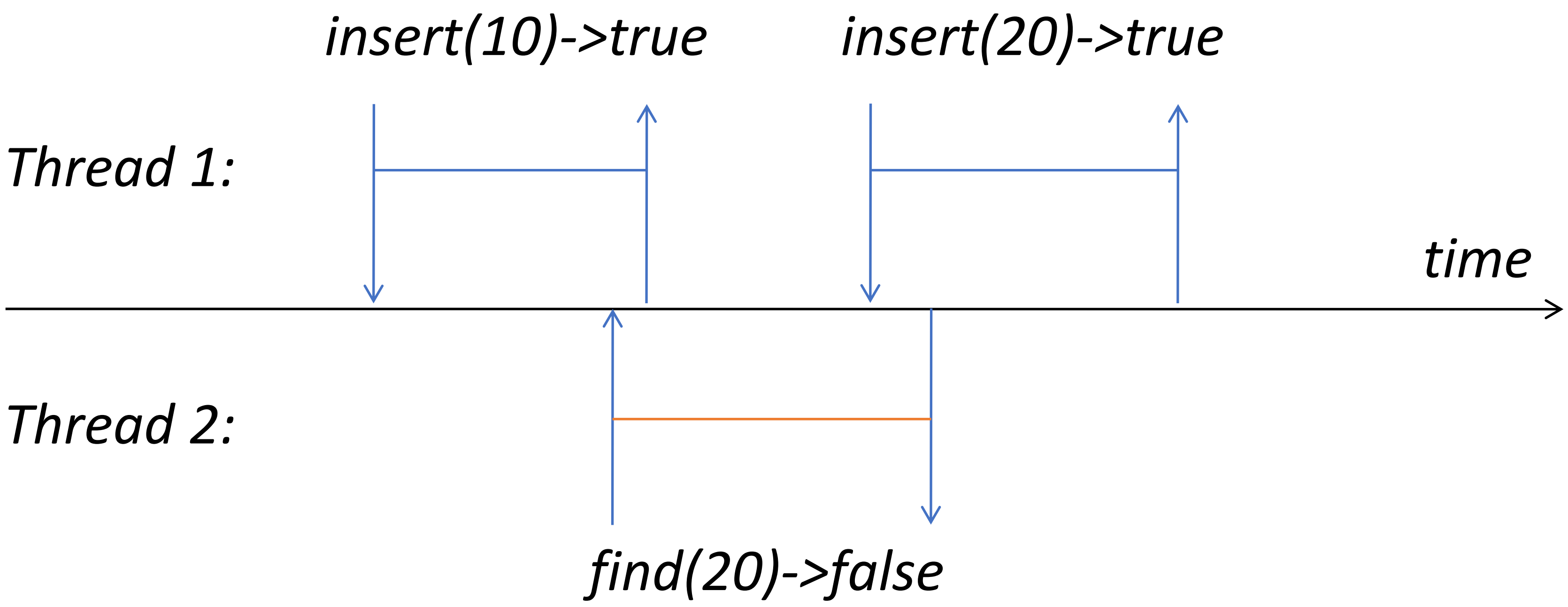


Total Order:

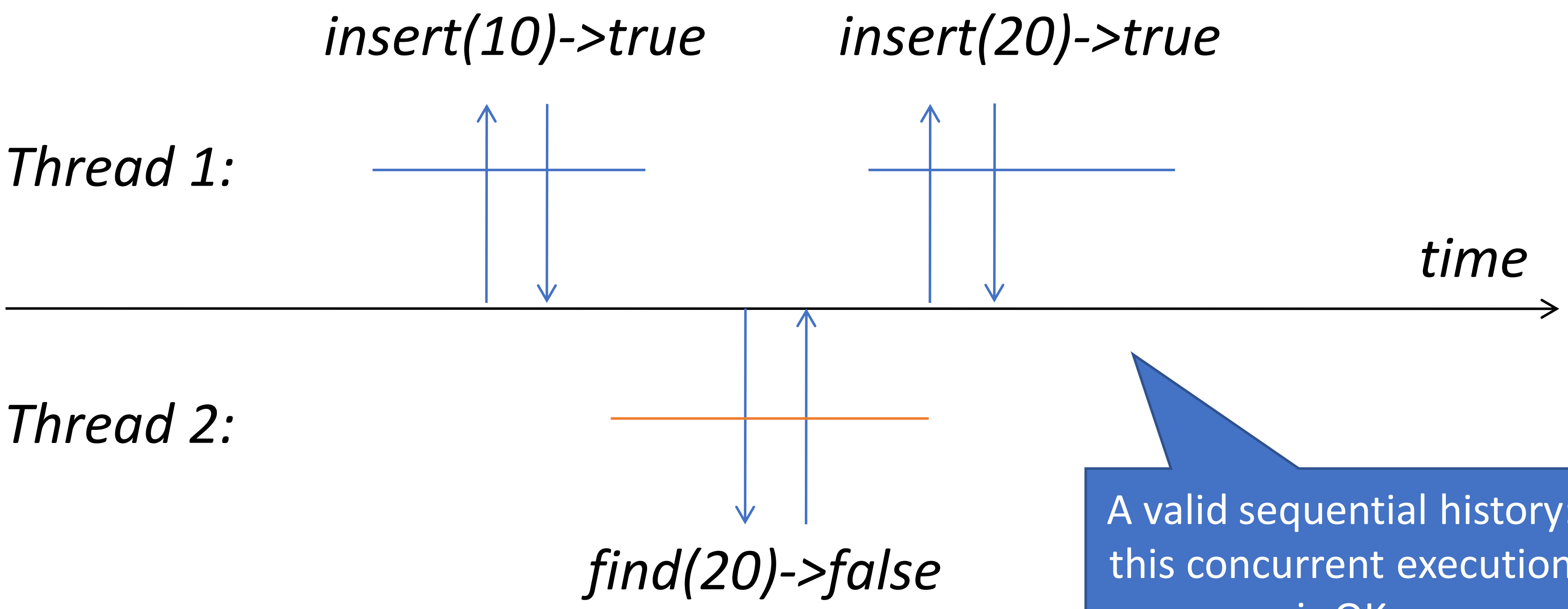
1. Insert(10)
2. Find(20)
3. Insert(20)

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

Example: linearizable



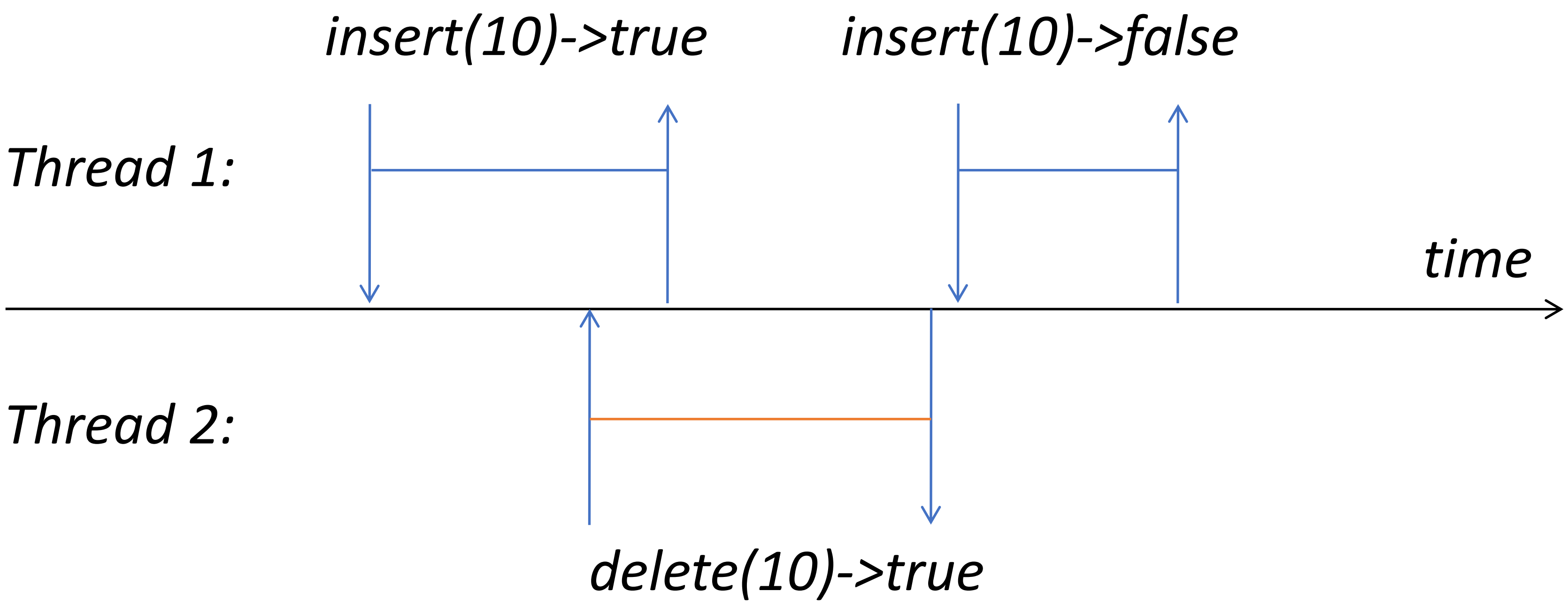
Example: linearizable



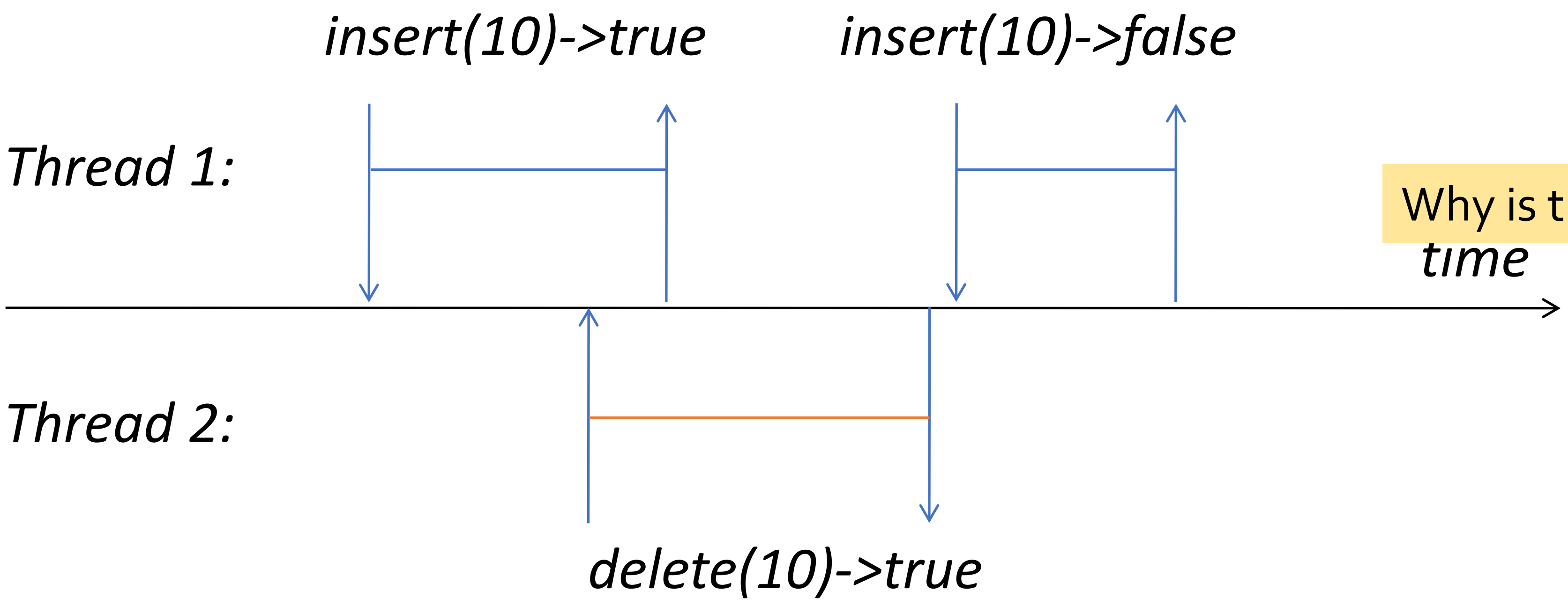
A valid sequential history: this concurrent execution is OK

Note: linearization point

Example: not linearizable

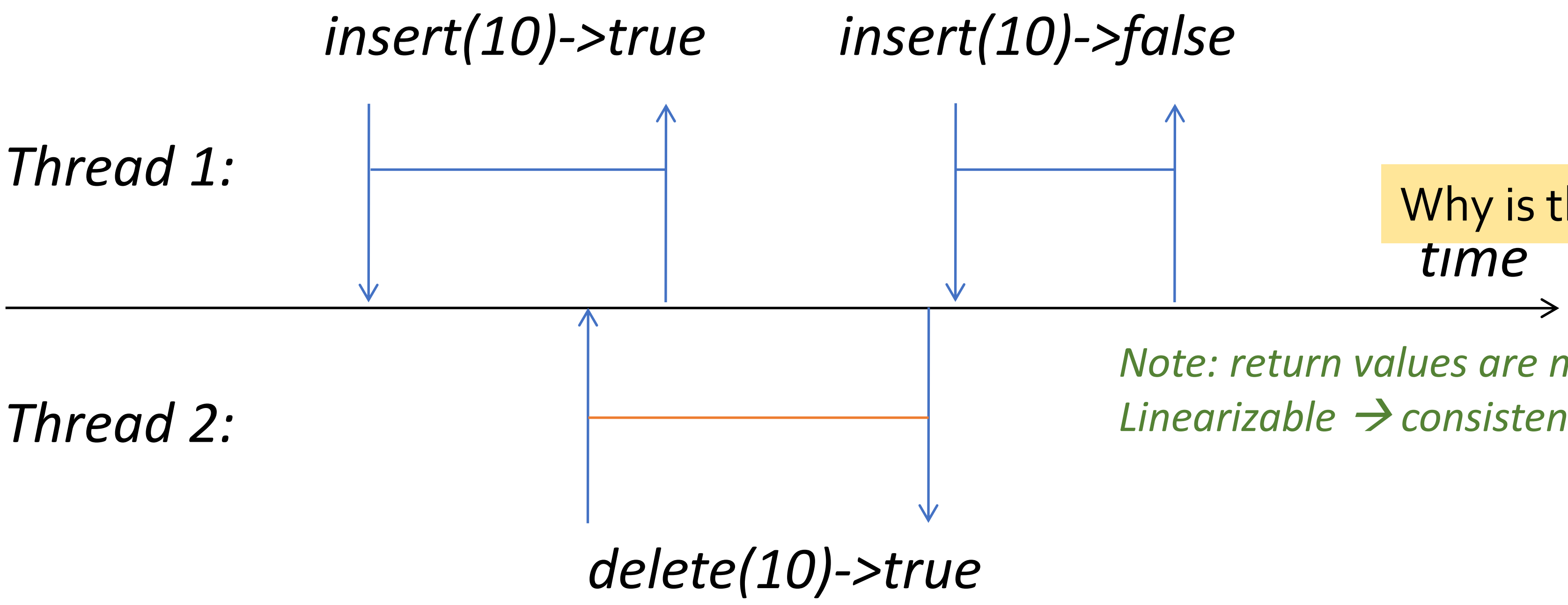


Example: not linearizable



Why is this one NOT OK?

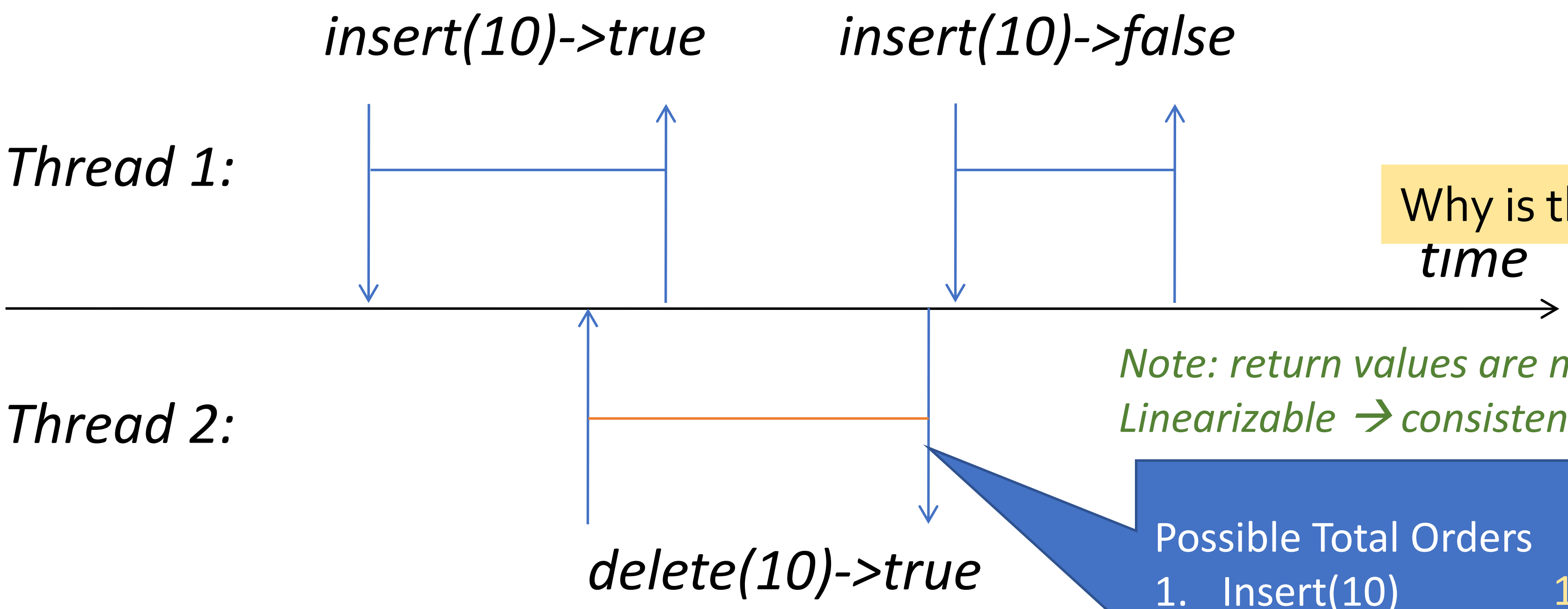
Example: not linearizable



Why is this one NOT OK?

Note: return values are meaningful!
Linearizable → consistent with return values

Example: not linearizable



Why is this one NOT OK?

Note: return values are meaningful!
Linearizable → consistent with return values

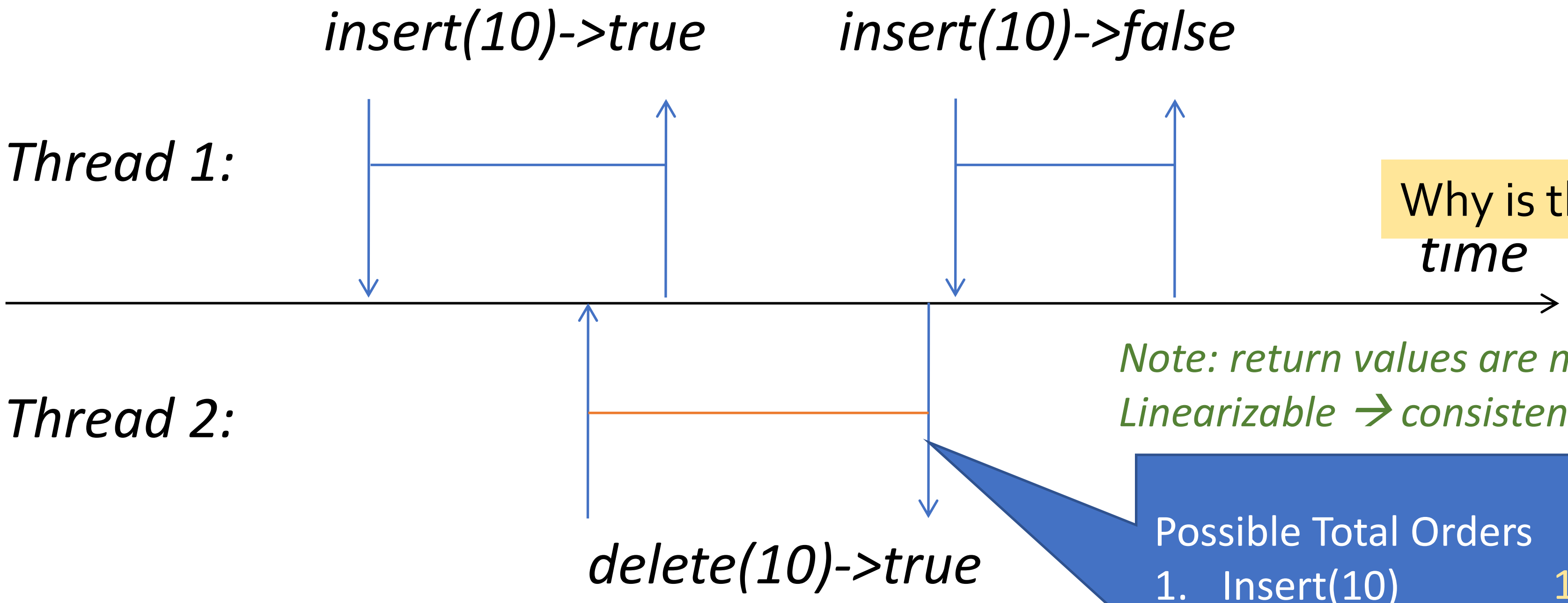
Possible Total Orders

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

Example: not linearizable



Why is this one NOT OK?

Note: return values are meaningful!
Linearizable → consistent with return values

Possible Total Orders

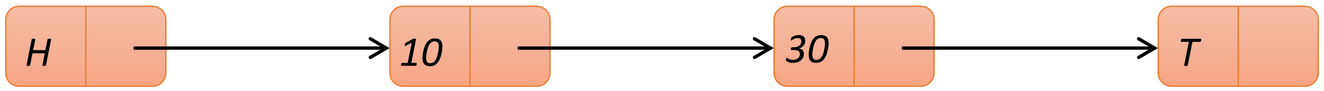
1. Insert(10)	1. Delete(10)
2. Delete(10)	2. Insert(10)
3. Insert(10)	3. Insert(10)

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

How can things like this happen?

Example Revisited

- find(20)

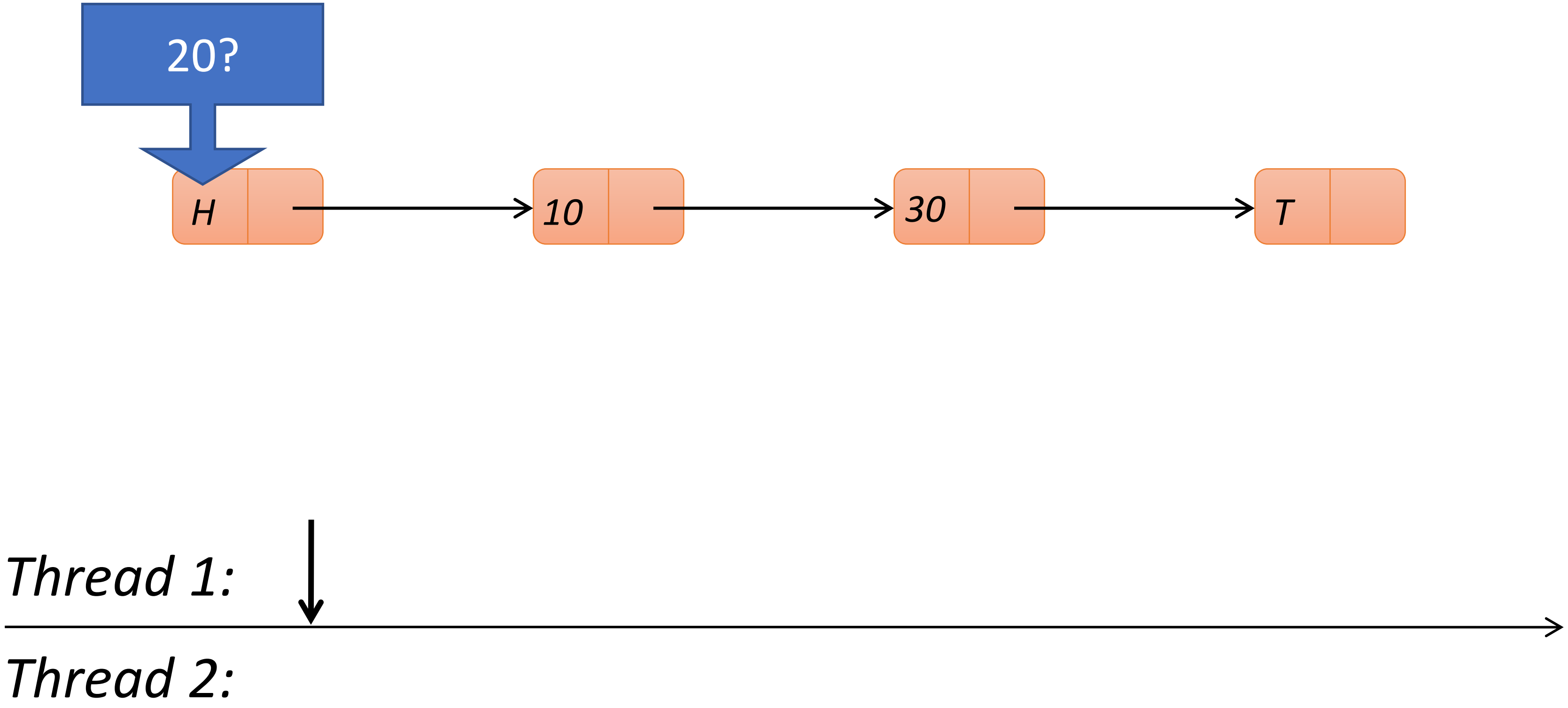


Thread 1:

Thread 2:

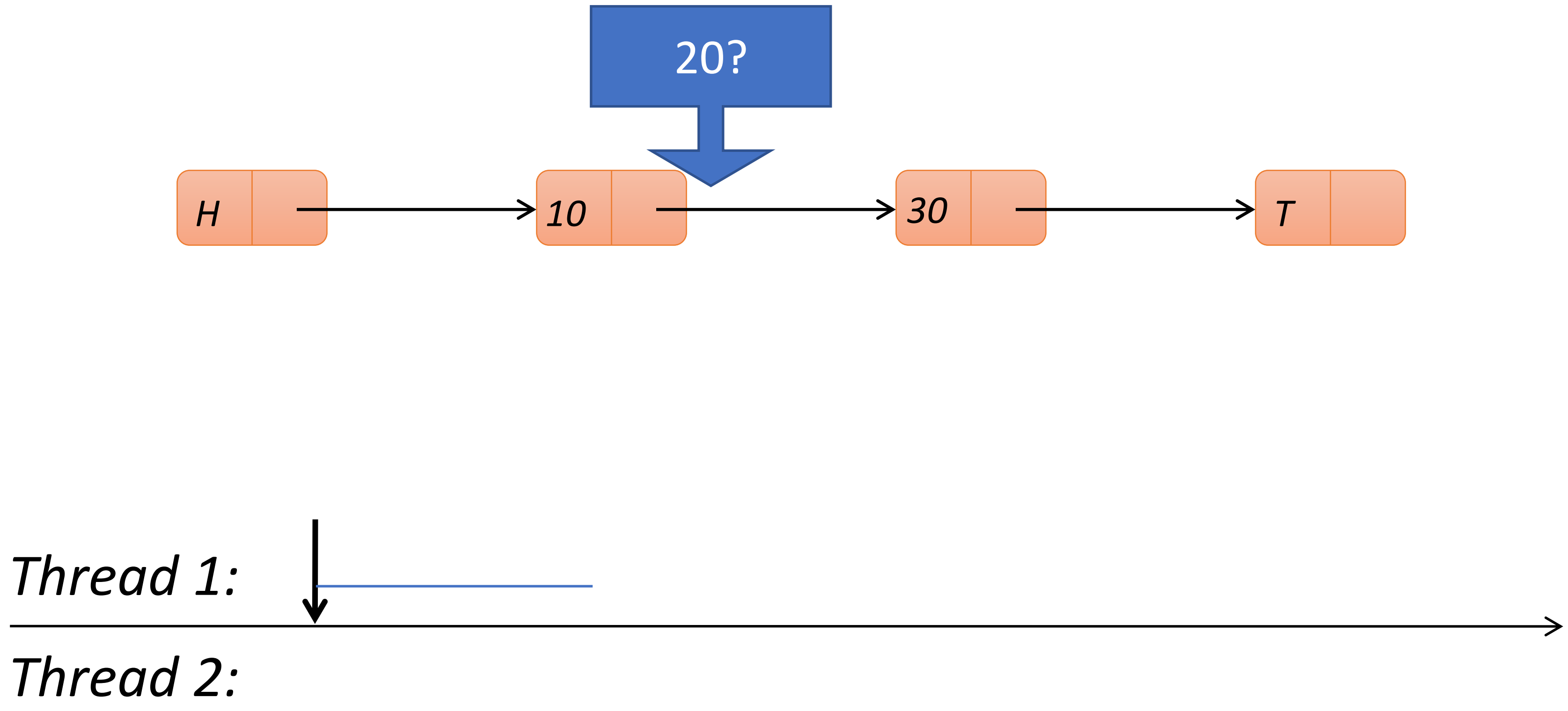
Example Revisited

- find(20)



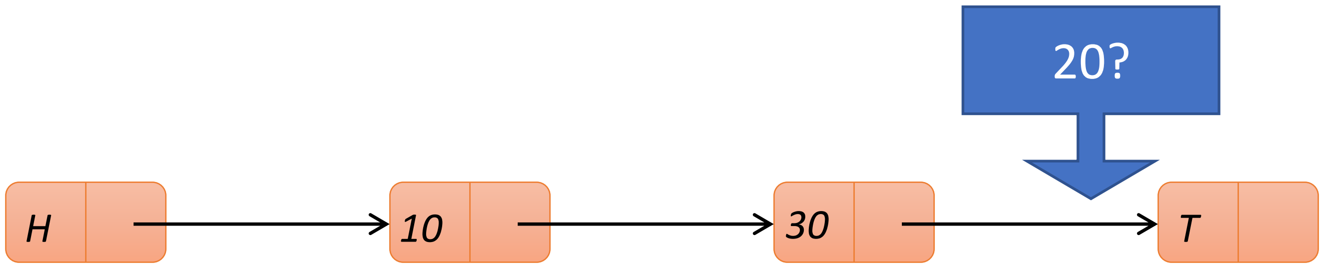
Example Revisited

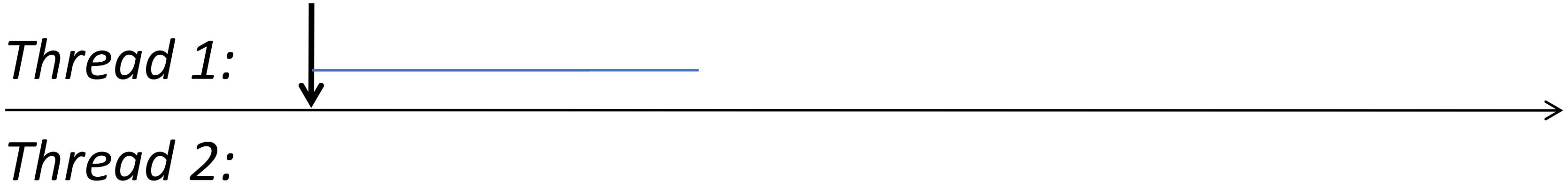
- find(20)



Example Revisited

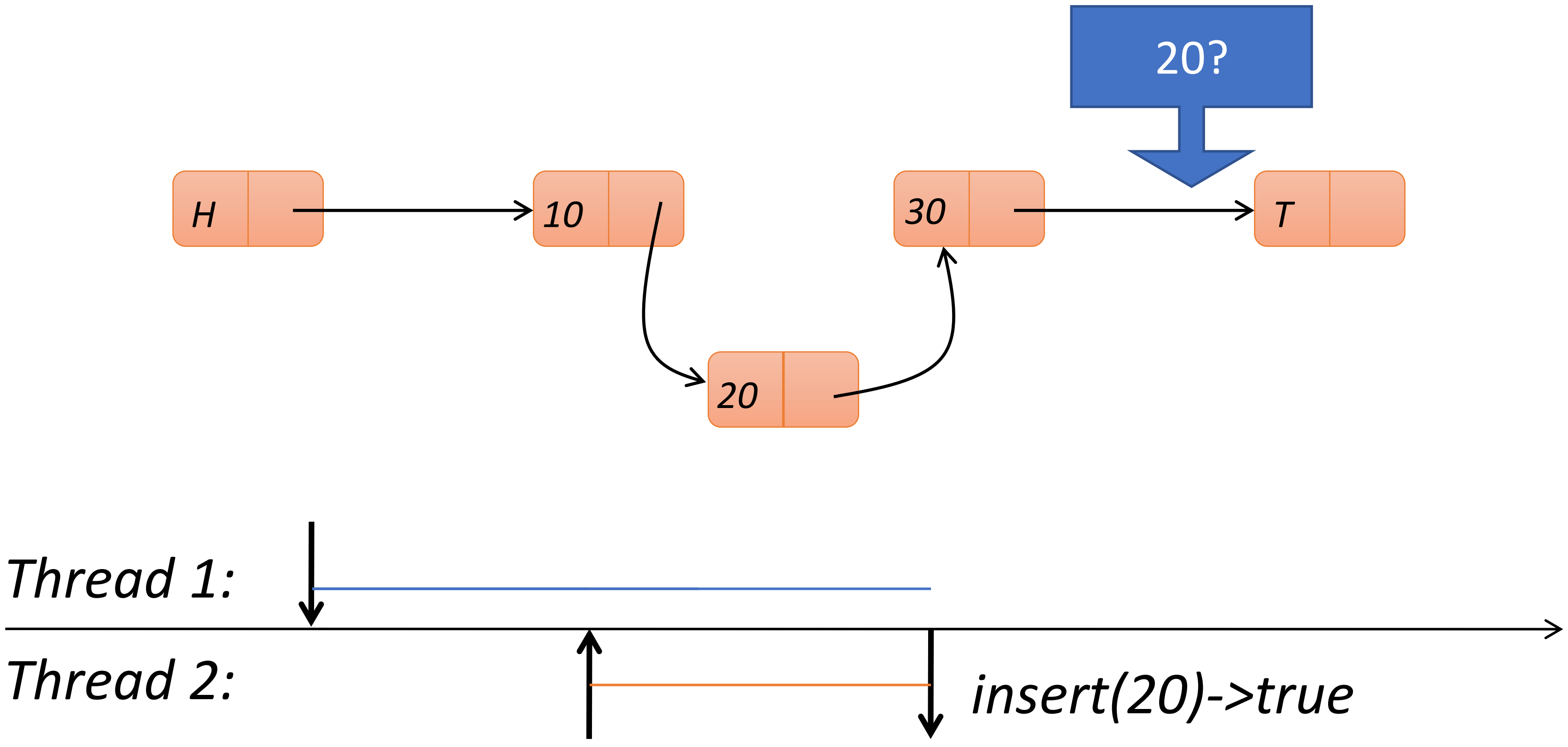
- find(20)





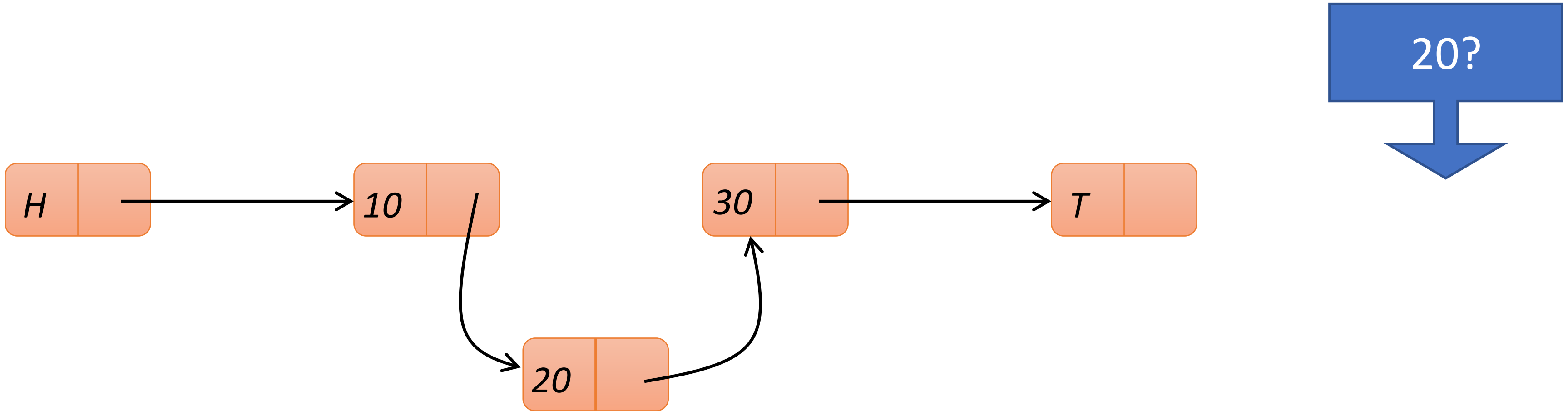
Example Revisited

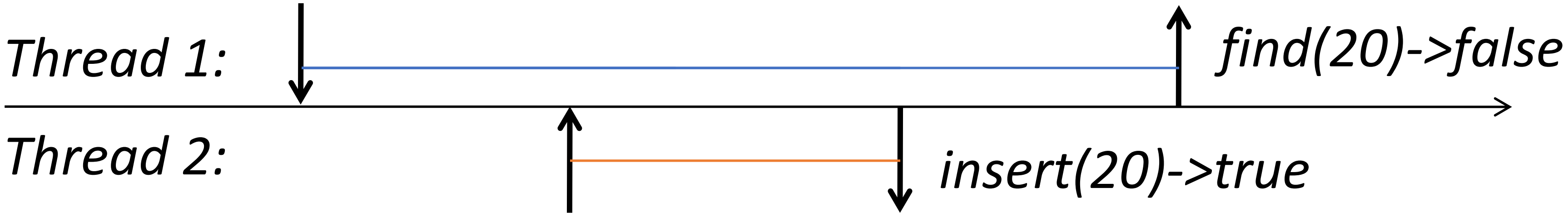
- find(20)
- insert(20) –> true



Example Revisited

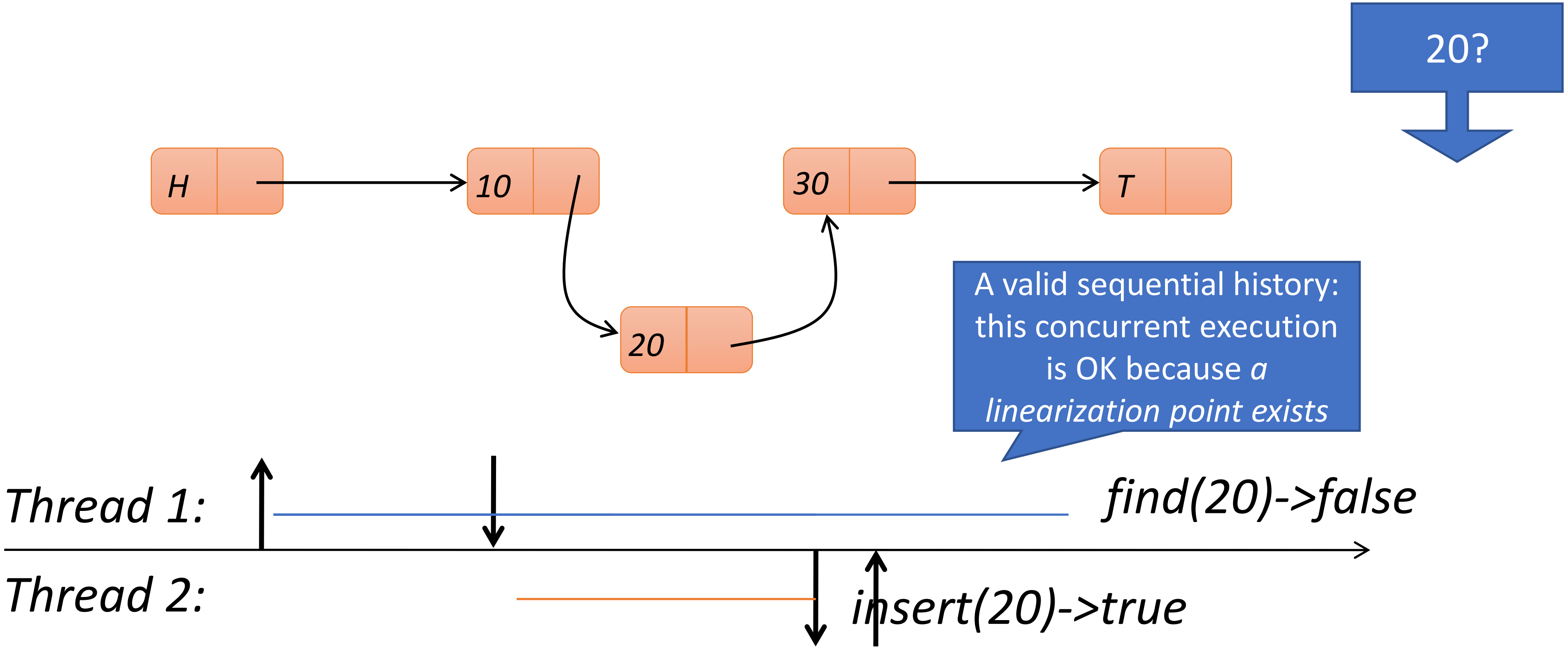
- find(20) -> false
- insert(20) -> true





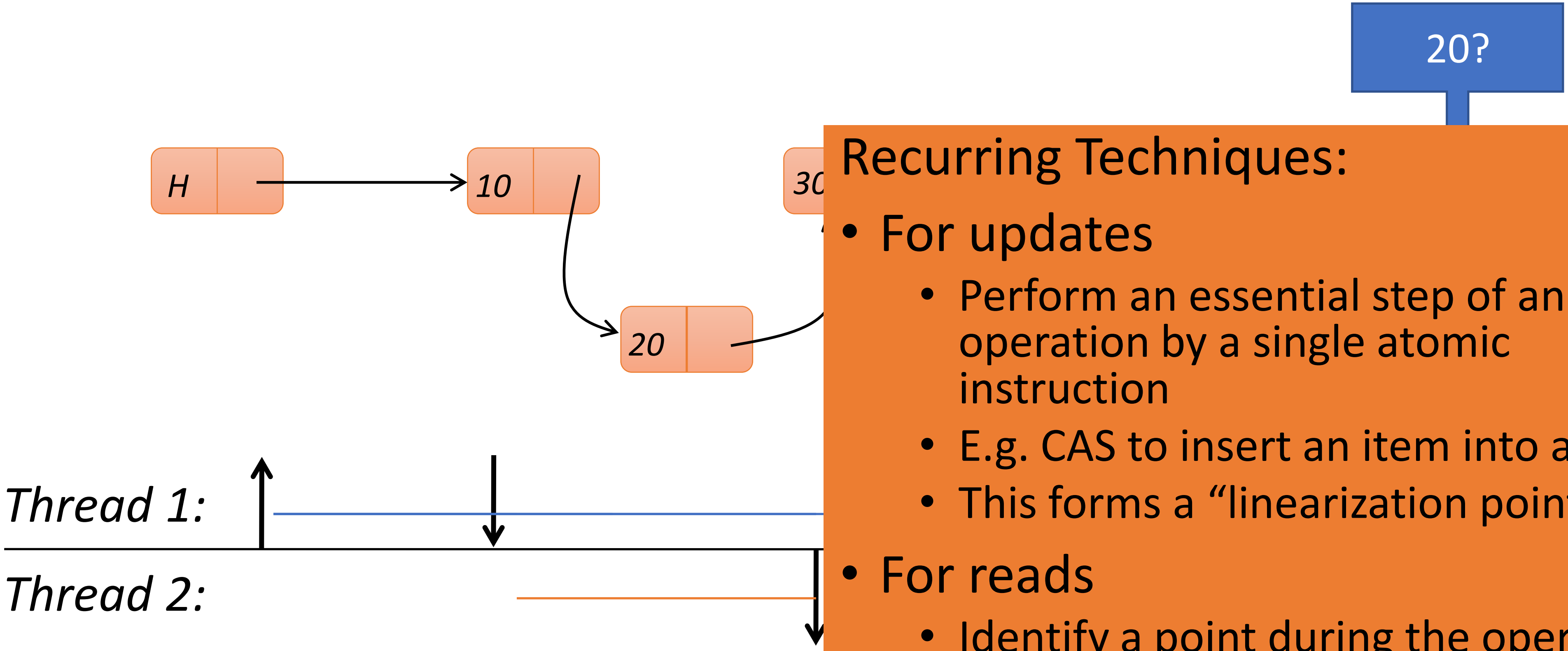
Example Revisited

- 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

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

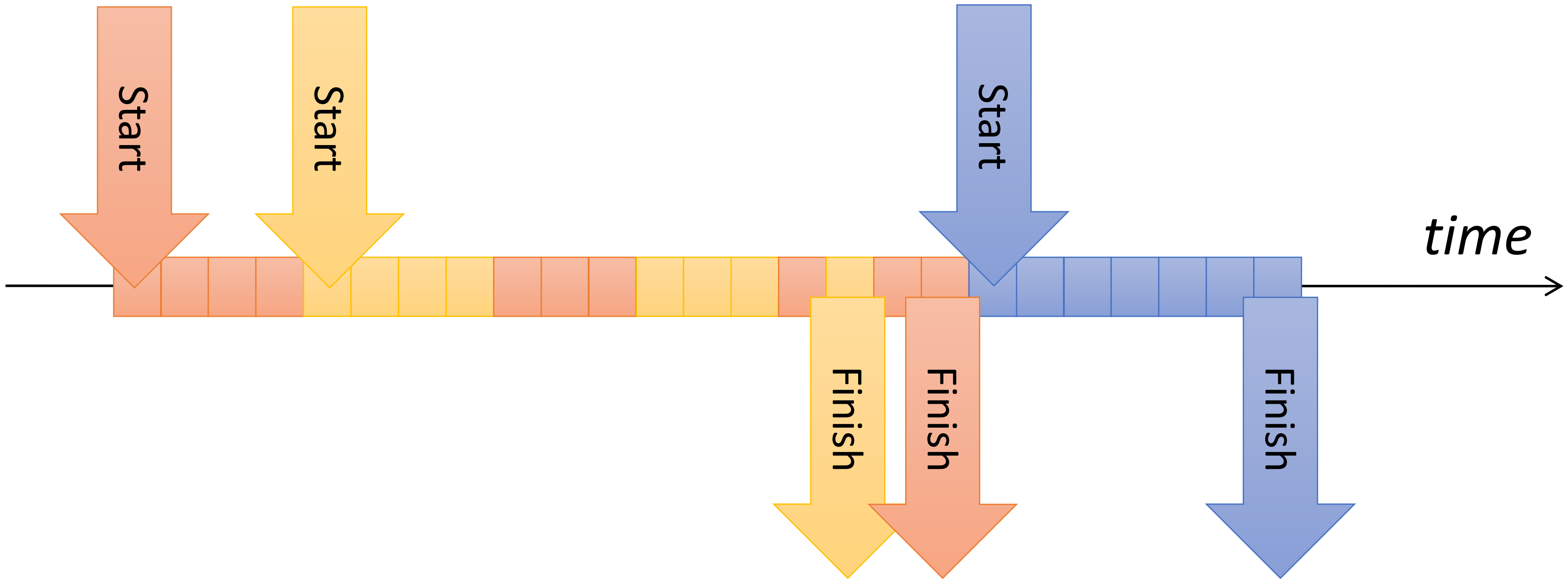
- Wait-free
  - A thread finishes its own operation if it continues executing steps
  - Strong: everyone eventually finishes
- Lock-free
  - Some thread finishes its operation if threads continue taking steps
  - Weaker: some forward progress guaranateed, but admits unfairness, live-lock, etc.
- Obstruction-free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

Wait-free

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

Wait-free

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

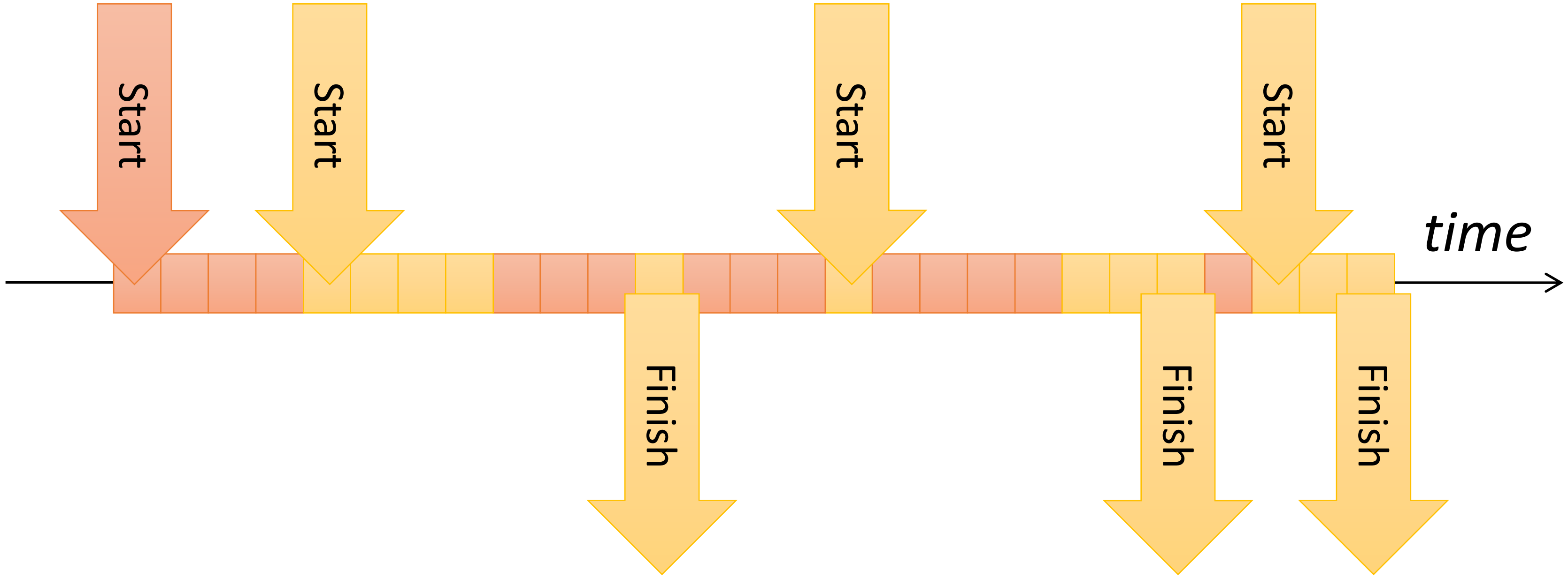


Lock-free

- Some thread finishes its operation if threads continue taking steps

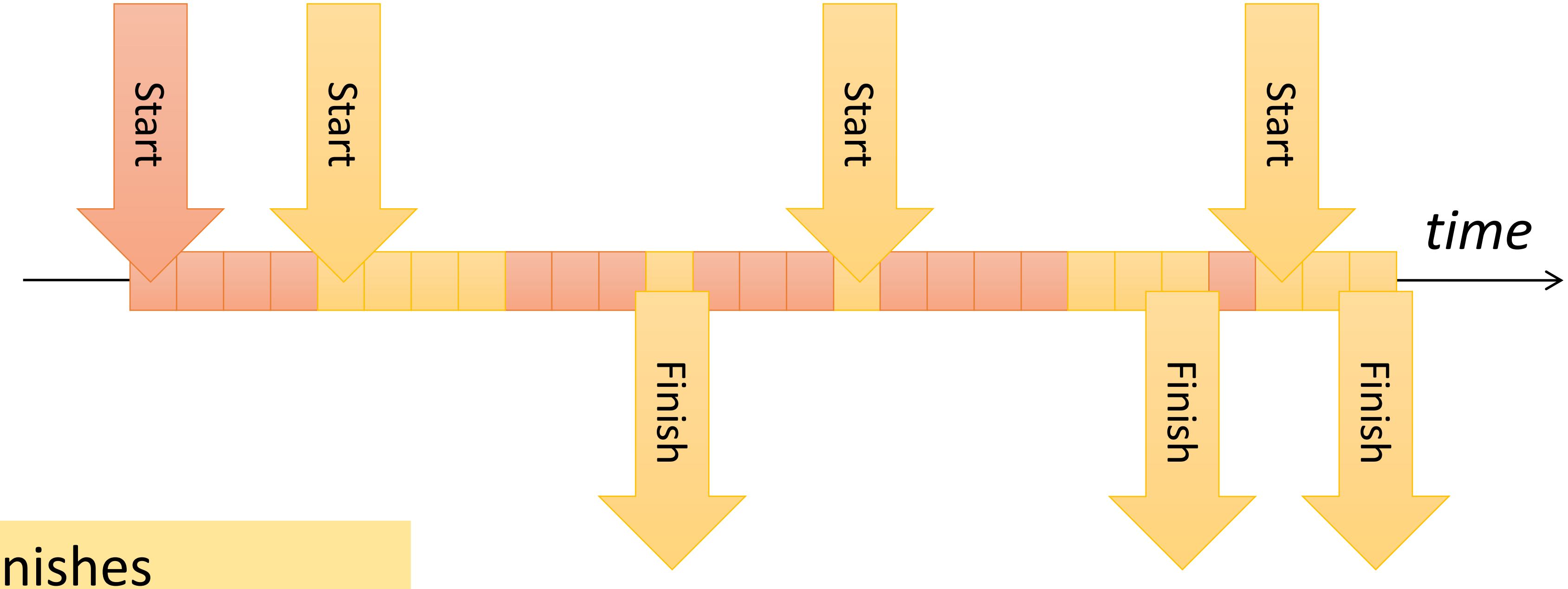
Lock-free

- Some thread finishes its operation if threads continue taking steps



Lock-free

- Some thread finishes its operation if threads continue taking steps



- Red never finishes
- Orange does
- Still lock-free

Obstruction-free

Obstruction-free

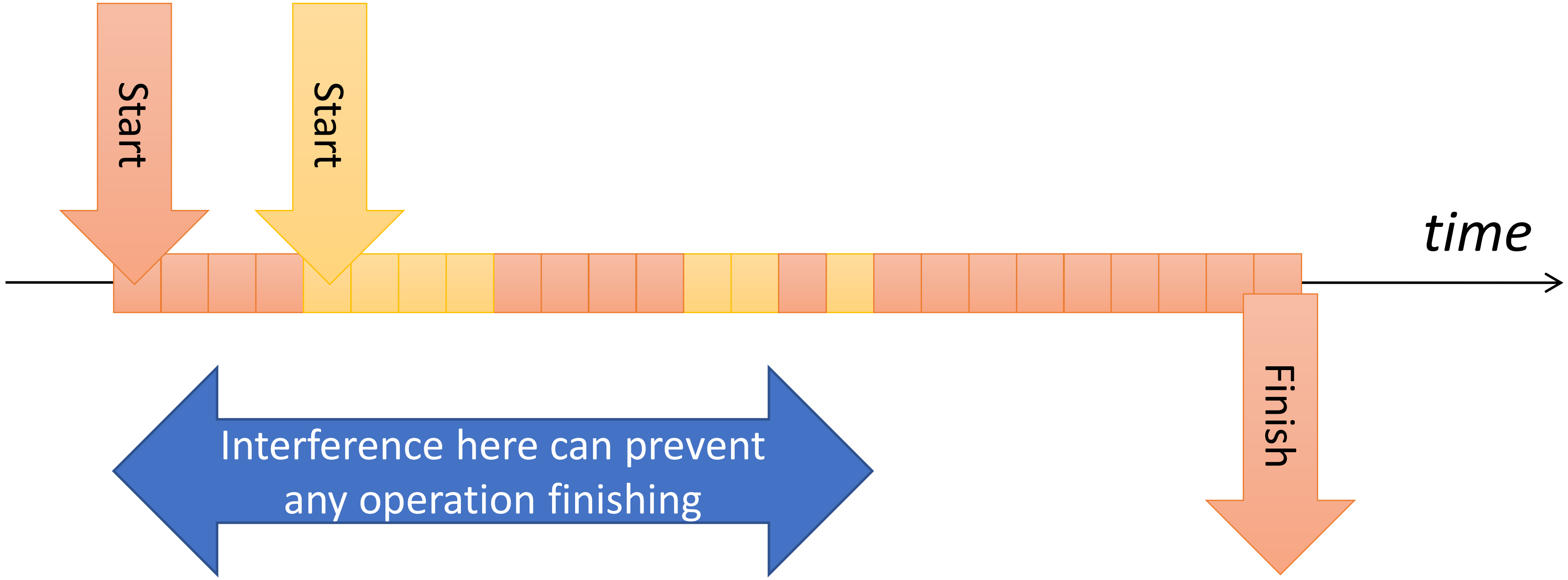
- A thread finishes its own operation if it runs in isolation

Obstruction-free

- A thread finishes its own operation if it runs in isolation
- *Meaning, if you de-schedule contenders*

Obstruction-free

- A thread finishes its own operation if it runs in isolation
- *Meaning, if you de-schedule contenders*



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 continue
  - 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

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

Blocking
1. Blocking
2. Starvation-Free

Obstruction-Free

3. Obstruction-Free

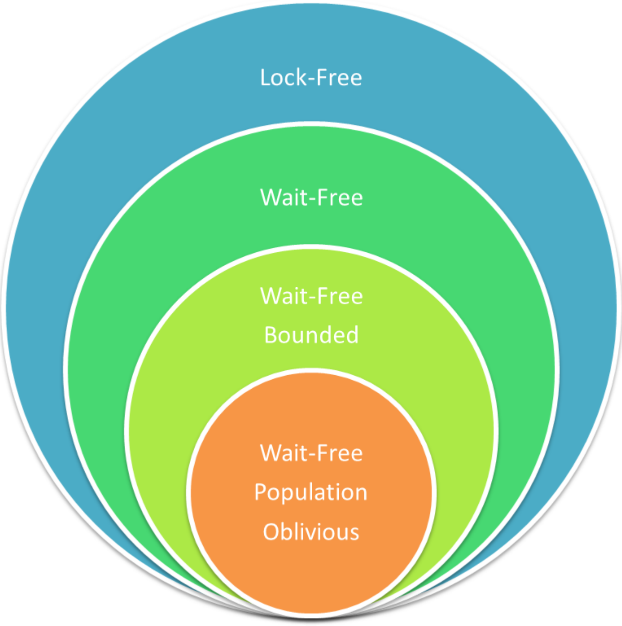
Lock-Free

4. Lock-Free (LF)

Wait-Free

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

stronger



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.

Huh? Composable?

Composability

Composability

```
T * list::remove(Obj key){
  LOCK(this);
  tmp = __do_remove(key);
  UNLOCK(this);
  return tmp;
}
```

Composability

```
T * list::remove(Obj key){
  LOCK(this);
  tmp = __do_remove(key);
  UNLOCK(this);
  return tmp;
}

void list::insert(Obj key, T * val){
  LOCK(this);
  __do_insert(key, val);
  UNLOCK(this);
}
```

Composability

```
T * list::remove(Obj key){
  LOCK(this);
  tmp = __do_remove(key);
  UNLOCK(this);
  return tmp;
}
```

```
void list::insert(Obj key, T * val){
  LOCK(this);
  __do_insert(key, val);
  UNLOCK(this);
}
```

```
void move(list s, list d, Obj key){
  tmp = s.remove(key);
  d.insert(key, tmp);
}
```

Composability

Thread-safe?

```
T * list::remove(Obj key){
  LOCK(this);
  tmp = __do_remove(key);
  UNLOCK(this);
  return tmp;
}

void list::insert(Obj key, T * val){
  LOCK(this);
  __do_insert(key, val);
  UNLOCK(this);
}
```

```
void move(list s, list d, Obj key){
  tmp = s.remove(key);
  d.insert(key, tmp);
}
```

Composability

```
T * list::remove(Obj key){
  LOCK(this);
  tmp = __do_remove(key);
  UNLOCK(this);
  return tmp;
}
```

```
void list::insert(Obj key, T * val){
  LOCK(this);
  __do_insert(key, val);
  UNLOCK(this);
}
```

```
void move(list s, list d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}
```

Composability

```
T * list::remove(Obj key){
  LOCK(this);
  tmp = __do_remove(key);
  UNLOCK(this);
  return tmp;
}
```

```
void list::insert(Obj key, T * val){
  LOCK(this);
  __do_insert(key, val);
  UNLOCK(this);
}
```

```
void move(list s, list d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}
```

- Lock-based code doesn't compose

Composability

```
T * list::remove(Obj key){
  LOCK(this);
  tmp = __do_remove(key);
  UNLOCK(this);
  return tmp;
}

void list::insert(Obj key, T * val){
  LOCK(this);
  __do_insert(key, val);
  UNLOCK(this);
}
```

```
void move(list s, list d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}
```

- Lock-based code doesn't compose
- If list were a linearizable concurrent data structure, composition OK

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.
  - Core hypotheses:
    - structuring all as concurrent objects buys composability
    - structuring all as concurrent objects is tractable/possible

Practical difficulties:

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

Practical difficulties:

- Key-value ma
- Population co
- Iteration
- Resizing the b

Options to consider when implementing a “difficult” operation:

Practical difficulties:

- Key-value ma
- Population co
- Iteration
- Resizing the b

Options to consider when implementing a "difficult" operation:

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

Practical difficulties:

- Key-value ma
- Population c
- Iteration
- Resizing the

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-value ma
- Population co
- Iteration
- Resizing the b

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-value ma
- Population co
- Iteration
- Resizing the b

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)