Concurrency Control Techniques

Outline

- Purpose of Concurrency Control
- Two-Phase locking
- Limitations of Concurrency Control
- Timestamp based concurrency control
- Multi-Version Concurrency Control
- Multiple granularity locks
- Index locking

Purpose of Concurrency Control

- To enforce Isolation (through mutual exclusion) among conflicting transactions.
- To preserve database consistency through consistency preserving execution of transactions.
- To resolve read-write and write-write conflicts.

- Example:
  - In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

Lock-Based Concurrency Control

- DBMS should ensure that only serializable and recoverable schedules are allowed
  - No actions of committed transactions are lost while undoing aborted transactions
- Uses a locking protocol
- Lock: a bookkeeping object associated with each “object”
  - different granularity
- Locking protocol:
  - a set of rules to be followed by each transaction

Two-Phase Locking Techniques

- Locking is an operation which secures
  - (a) permission to Read
  - (b) permission to Write a data item for a transaction.
- Example:
  - Lock (X). Data item X is locked in behalf of the requesting transaction.
- Unlocking is an operation which removes these permissions from the data item.
- Example:
  - Unlock (X): Data item X is made available to all other transactions.
- Lock and Unlock are Atomic operations.

Two Locks Modes

- Two locks modes:
  - (a) shared (read) (b) exclusive (write).
- Shared mode: shared lock (X)
  - More than one transaction can apply share lock on X for reading its value but no write lock can be applied on X by any other transaction.
- Exclusive mode: Write lock (X)
  - Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.
- Conflict matrix

	Read	Write
Read	Y	N
Write	N	N

Locking Rules

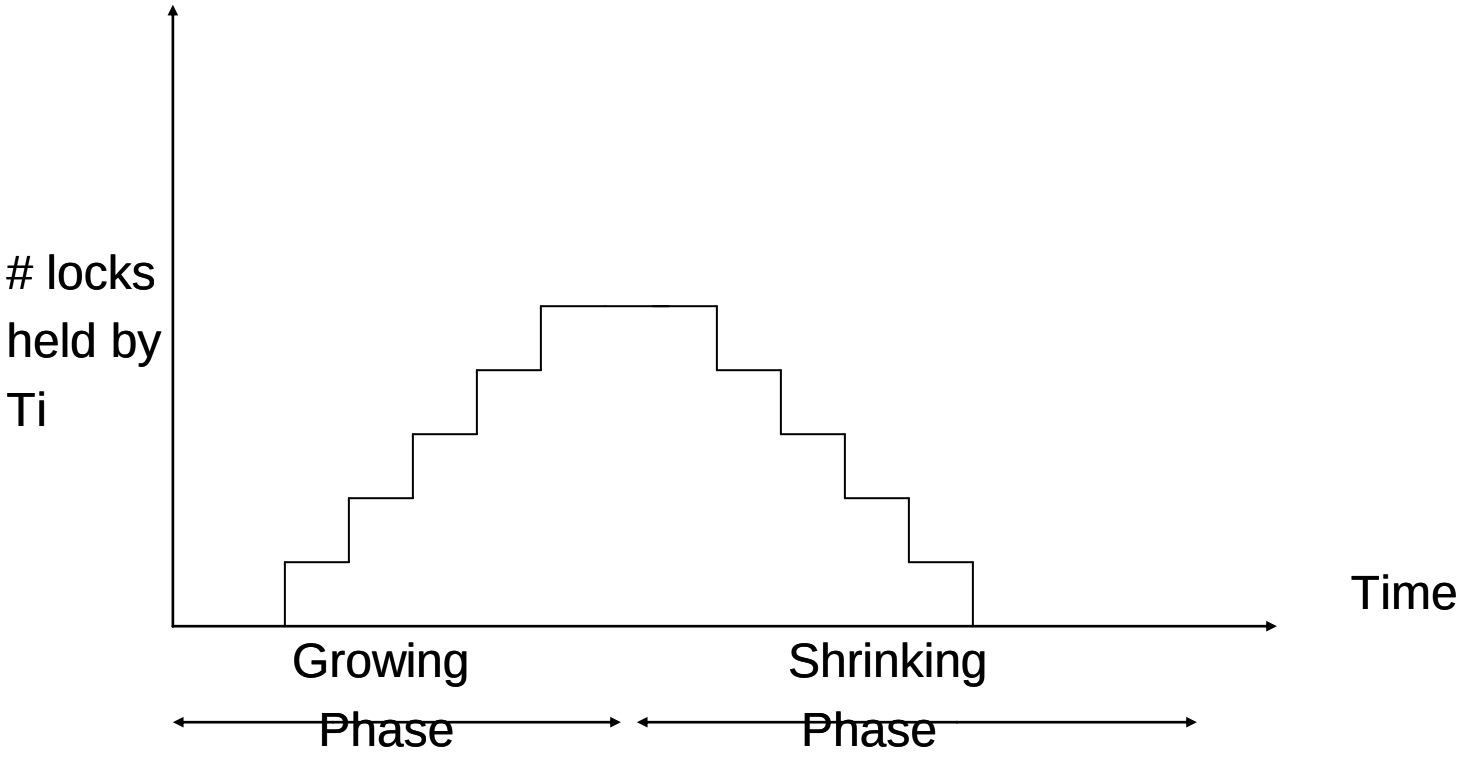
- T must issue read_lock(X) or write_lock(X) before any read_item(X) op is performed in T
- T must issue write_lock(X) before any write_item(X) op is performed in T
- T must issue unlock(X) after all read_item(X) and write_item(X) ops are completed in T
- T will not issue a read_lock(X) if it already holds a read lock or write lock on X (may be relaxed)
- T will not issue a write_lock(X) if it already holds a read lock or write lock on X (may be relaxed)

Lock Conversions

- Sometimes beneficial to relax locking rules 4 and 5
- Upgrade read lock on X to a write lock (by issuing a write_lock(X))
  - Only possible if T is the only transaction holding a read lock on X
- Downgrade a write lock by issuing a read_lock(X)
- Must be noted in lock table

2P Locking Technique Algorithm

- **Two Phases:**
  - (a) Locking (Growing)
  - (b) Unlocking (Shrinking).
- **Locking (Growing) Phase:**
  - A transaction applies locks (read or write) on desired data items one at a time.
- **Unlocking (Shrinking) Phase:**
  - A transaction unlocks its locked data items one at a time.
- **Requirement:**
  - For a transaction these two phases must be mutually exclusively, that is, during locking phase unlocking phase must not start and during unlocking phase locking phase must not begin.


locks
held by
Ti
Time
Growing
Phase
Shrinking
Phase

2P Locking Technique Example

T1

```
read_lock (Y);
write_lock (X);
read_item (Y);
read_item (X);
X:=X+Y;
write_item (X);
unlock (Y);
unlock (X);
```

T2

```
read_lock (X);
Write_lock (Y);
read_item (X);
read_item (Y);
Y:=X+Y;
write_item (Y);
unlock (X);
unlock (Y);
```

2P Locking Technique Example

T1	T2
read_lock (Y);	
read_item (Y);	
unlock (Y);	
	read_lock (X);
	read_item (X);
	unlock (X);
	write_lock (Y);
	read_item (Y);
	Y:=X+Y;
	write_item (Y);
	unlock (Y);
write_lock (X);	
read_item (X);	
X:=X+Y;	
write_item (X);	
unlock (X);	

Result

Nonserializable because it. violated two-phase policy.

2P Locking Technique Example

T'1

```
read_lock (Y);
read_item (Y);
write_lock (X);
unlock (Y);
read_item (X);
X:=X+Y;
write_item (X);
unlock (X);
```

T'2

```
read_lock (X);
read_item (X);
Write_lock (Y);
unlock (X);
read_item (Y);
Y:=X+Y;
write_item (Y);
unlock (Y);
```

T1 and T2 follow two-phase policy but they are subject to deadlock, which must be dealt with.

Limitations of Concurrency Control

- **Deadlock**

T'1	**T'2**
read_lock (Y);	
read_item (Y);	
	read_lock (X);
	read_item (X);
write_lock (X);	
(waits for X)	write_lock (Y);
	(waits for Y)

T1 and T2 did follow two-phase policy but they are deadlock

- Deadlock (T'1 and T'2)

Deadlock Detection

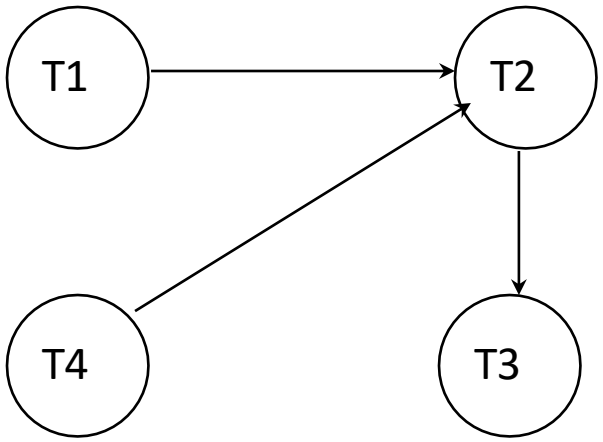
1. Create a waits-for graph: (example on next slide)
   - Nodes are transactions
   - There is an edge from T_i to T_j if T_i is waiting for T_j to release a lock
- Periodically check for cycles in the waits-for graph
- Abort a transaction on a cycle and release its locks, proceed with the other transactions
  - several choices
  - one with the fewest locks
  - one has done the least work/farthest from completion
  - if being repeatedly restarted, should be favored at some point

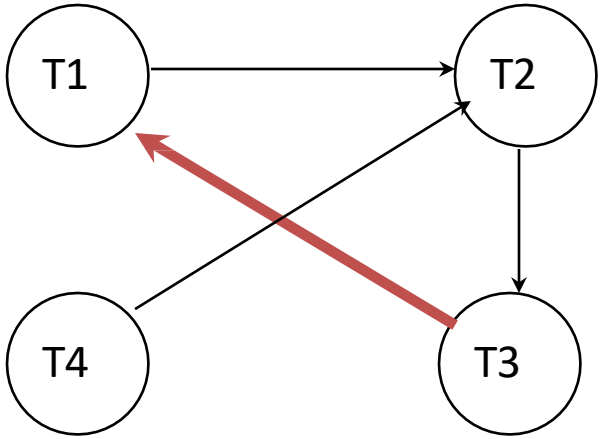
2. Use timeout, if long delay, assume (pessimistically) a deadlock

Deadlock Detection

Example:

T1: RL(A), R(A), RL(B)
T2: WL(B),W(B) WL(C)
T3: RL(C), R(C) WL(A)
T4: WL(B)





Deadlock Prevention

- Assign priorities based on timestamps
- Assume T_i wants a lock that T_j holds. Two policies are possible:
  - Wait-Die: It T_i has higher priority, T_i waits for T_j; otherwise T_i aborts
  - Wound-wait: If T_i has higher priority, T_j aborts; otherwise T_i waits
- Convince yourself that no cycle is possible
- If a transaction re-starts, make sure it has its original timestamp
  - each transaction will be the oldest one and have the highest priority at some point

- A variant of strict 2PL, conservative 2PL, works too
  - acquire all locks it ever needs before a transaction starts
  - no deadlock but high overhead and poor performance, so not used in practice

Starvation

- Starvation occurs when a particular transaction consistently waits or restarted and never gets a chance to proceed further.
- In a deadlock resolution it is possible that the same transaction may consistently be selected as victim and rolled-back.
- This limitation is inherent in all priority based scheduling mechanisms.
- In Wound-Wait scheme a younger transaction may always be wounded (aborted) by a long running older transaction which may create starvation.

2P Locking Technique Algorithms

Two-Phase Locking Techniques:

- **Conservative**:
  - Prevents deadlock by locking all desired data items before transaction begins execution.
- **Basic**:
  - Transaction locks data items incrementally. This may cause deadlock which is dealt with.
- **Strict**:
  - A more stricter version of Basic algorithm where unlocking is performed after a transaction terminates (commits or aborts and rolled-back). This is the most commonly used two-phase locking algorithm.

Example: Strict 2PL

```
T1:  R(A), W(A),                                        R(B), W(B), Commit
T2:                  R(A), W(A), R(B), W(B), Commit
```

- WR conflict (dirty read)
- Strict 2PL does not allow this

```
T1:  WL(A), R(A), W(A),
T2:                        HAS TO WAIT FOR LOCK ON A
```

```
T1:  WL(A), R(A), W(A), WL(B), R(B), W(B), C
T2:                                          WL(A), R(A), W(A), WL(B), R(B), W(B), C
```

Example: Strict 2PL

T1:	RL(A), R(A),	WL(C), R(C), W(C), C
T2:	RL(A), R(A), WL(B), R(B), W(B), C	

- Strict 2PL allows interleaving

Strict 2PL and Conflict Serializability

- Strict 2PL allows only schedules whose precedence graph is acyclic
- Can never allow cycles as the write locks are being held by one transaction
- However, it is sufficient but not necessary for serializability

Timestamp based concurrency control algorithm

- **Timestamp (TS)**
  - A monotonically increasing variable (integer) indicating the age of an operation or a transaction. A larger timestamp value indicates a more recent event or operation.
  - Timestamp based algorithm uses timestamp to serialize the execution of concurrent transactions.
  - TS is unique identifier assigned to each transaction
  - if T1 starts before T2, then TS(T1) < TS(T2) (older has smaller timestamp value)
  - *Wait-die* and *wound-wait* schemes

Wait-Die Scheme

- Assume T_i tries to lock X which is locked by T_j
- If $TS(T_i) < TS(T_j)$ (T_i older than T_j), then T_i is allowed to wait
- Otherwise, T_i younger than T_j, abort T_i (T_i dies) and restart later with SAME timestamp
- Older transaction is allowed to wait on younger transaction
- Younger transaction requesting an item held by older transaction is aborted and restarted

Wound-Wait Scheme

- Assume T_i tries to lock X which is locked by T_j
- If $TS(T_i) < TS(T_j)$ (T_i older than T_j), abort T_j (T_i wounds T_j) and restart later with SAME timestamp
- Otherwise, T_i younger than T_j, T_i is allowed to wait
- Younger transaction is allowed to wait on older transaction
- Older transaction requesting item held by younger transaction preempts younger one by aborting it
- Both schemes abort younger transaction that may be involved in deadlock
- Both deadlock free but may cause needless aborts

Cautious Waiting

- *Waiting schemes* (require no timestamps)
- No waiting: if transaction cannot obtain lock, aborted immediately and restarted after time *t*
  - → needless restarts
- Cautious waiting:
  - Suppose T_i tries to lock item X which is locked by T_j
  - If T_j is not blocked, T_i is blocked and allowed to wait
  - Otherwise abort T_i
  - Cautious waiting is deadlock-free

Timestamp Ordering CC

Main Idea:

- Give each object O
  - a read-timestamp RT(O), and
  - a write-timestamp WT(O)

- Give each transaction T
  - a timestamp TS(T) when it begins

- If
  - action ai of Ti conflicts with action aj of Tj,
  - and TS(Ti) < TS(Tj)
- then
  - ai must occur before aj
- Otherwise, abort and restart violating transaction

Request for a read: $R_T(X)$

1. If TS(T) >= WT(X)
   - last written by a previous transaction –- *OK (i.e. "physically realizable")*
   - If C(X) is true –- *check if previous transaction has committed*
     - Grant the read request by T
     - if TS(T) > RT(X)
       - set RT(X) = TS(T)
   - If C(X) is false
     - Delay T until C(X) becomes true, or the transaction that wrote X aborts
2. If TS(T) < WT(X)
   - write is not realizable -- already *written by a later trans.*
   - Abort (or, Rollback) T --*i.e. abort and restart with a larger timestamp*

Request for a write: $W_T(X)$

1. If TS(T) >= RT(X) and TS(T) >= WT(X)
   - last written/read by a previous transaction – *OK*
   - Grant the write request by T
     - write the new value of X
   - Set WT(X) = TS(T)
   - Set C(X) = false -- *T not committed yet*
2. If TS(T) >= RT(X) but TS(T)< WT(X)
   - write is still realizable –-*but already a later value in X*
   - If C(X) is true
     - previous writer of X has committed
     - simply ignore the write request by T
     - but allow T to proceed without making changes to the database
   - If C(X) is false
     - Delay T until C(X) becomes true, or the transaction that wrote X aborts
- If TS(T) < RT(X)
  - write is not realizable -- *already read by a later transaction*
  - Abort (or, Rollback) T

Example

- Three transactions T1 (TS = 200), T2 (TS = 150), T3 (TS = 175)
- Three objects A, B, C
  - initially all have RT = WT = 0, C = 1 (i.e. true)
- Sequence of actions
  - $R_1(B)$, $R_2(A)$, $R_3(C)$, $W_1(B)$, $W_1(A)$, $W_2(C)$, $W_3(A)$
- Q. What is the state of the database at the end if the timestamp-based CC protocol is followed
  - i.e. report the RT, WT, C

Initial condition and Steps

Step	T1	T2	T3	A	B	C
	200	150	175	RT = 0, WT = 0, C = 1	RT = 0, WT = 0, C = 1	RT = 0, WT = 0, C = 1
1	$R_1(B)$					
2		$R_2(A)$				
3			$R_3(C)$			
4	$W_1(B)$					
5	$W_1(A)$					
6		$W_2(C)$				
7			$W_3(A)$			

After Step 1

WT of B is <= TS(T_1)
C = 1
Read OK.

Step	T1	T2	T3	A	B	C
	200	150	175	RT = 0, WT = 0, C = 1	RT = 200, WT = 0, C = 1	RT = 0, WT = 0, C = 1
1	$R_1(B)$				RT=200	
2		$R_2(A)$				
3			$R_3(C)$			
4	$W_1(B)$					
5	$W_1(A)$					
6		$W_2(C)$				
7			$W_3(A)$			

After Step 2

WT of A is <= TS(T_2)
C = 1
Read OK.

Step	T1	T2	T3	A	B	C
	200	150	175	RT = 150, WT = 0, C = 1	RT = 200, WT = 0, C = 1	RT = 0, WT = 0, C = 1
1	$R_1(B)$				RT=200	
2		$R_2(A)$		RT=150		
3			$R_3(C)$			
4	$W_1(B)$					
5	$W_1(A)$					
6		$W_2(C)$				
7			$W_3(A)$			

After Step 3

WT of C is <= TS(T_3)
C = 1
Read OK.

Step	T1	T2	T3	A	B	C
	200	150	175	RT = 150, WT = 0, C = 1	RT = 200, WT = 0, C = 1	RT = 175, WT = 0, C = 1
1	$R_1(B)$				RT=200	
2		$R_2(A)$		RT=150		
3			$R_3(C)$			RT=175
4	$W_1(B)$					
5	$W_1(A)$					
6		$W_2(C)$				
7			$W_3(A)$			

After Step 4

WT & RT of B is <= TS(T_1)
Write OK.

Step	T1	T2	T3	A	B	C
	200	150	175	RT = 150, WT = 0, C = 1	RT = 200, WT = 200 C = 0	RT = 175, WT = 0, C = 1
1	$R_1(B)$				RT=200	
2		$R_2(A)$		RT=150		
3			$R_3(C)$			RT=175
4	$W_1(B)$				WT=200 C=0	
5	$W_1(A)$					
6		$W_2(C)$				
7			$W_3(A)$			

After Step 5

RT & WT of A <= TS(T_1)
Write ok.

Step	T1	T2	T3	A	B	C
	200	150	175	RT = 150 WT = 200 C = 0	RT = 200 WT = 200 C = 0	RT = 175 WT = 0 C = 1
1	$R_1(B)$				RT=200	
2		$R_2(A)$		RT=150		
3			$R_3(C)$			RT=175
4	$W_1(B)$				WT=20 0 C=0	
5	$W_1(A)$			WT=200 C=0		
6		$W_2(C)$				
7			$W_3(A)$			

After Step 6

RT(C) = 175 < 150 = TS(T_2)
Abort T_2

Step	T1	T2	T3	A	B	C
	200	150	175	RT = 150 WT = 200 C = 0	RT = 200 WT = 200 C = 0	RT = 175 WT = 0 C = 1
1	$R_1(B)$				RT=200	
2		$R_2(A)$		RT=150		
3			$R_3(C)$			RT=175
4	$W_1(B)$				WT=200 C=0	
5	$W_1(A)$			WT=200 C=0		
6		$W_2(C)$ Abort				
7			$W_3(A)$			

After Step 7

RT(A) <= $TS(T_3)$ – write ok
WT(A) > $TS(T_3)$ and C(A) = 0
Delay T_3

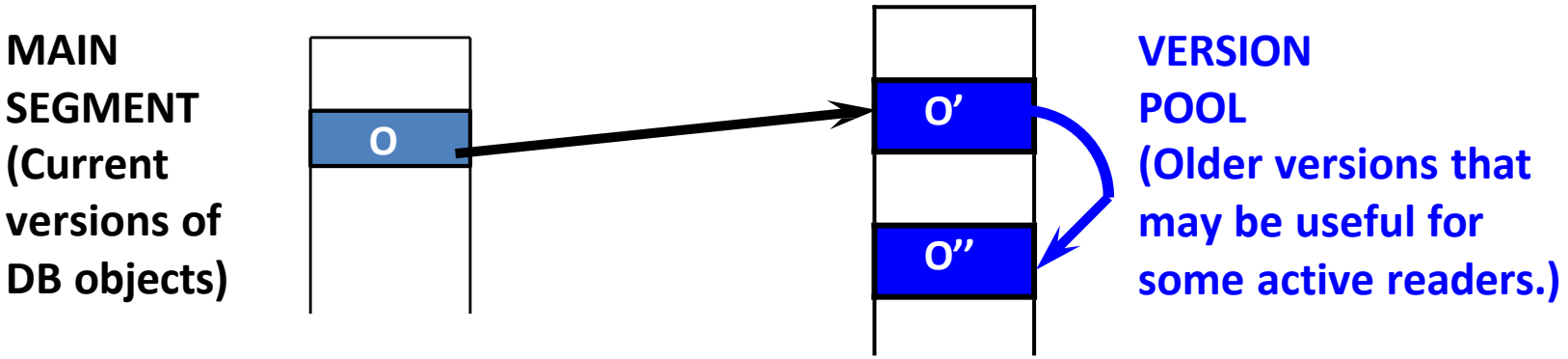
Step	T1	T2	T3	A	B	C
	200	150	175	RT = 150 WT = 200 C = 0	RT = 200 WT = 200 C = 0	RT = 175 WT = 0 C = 1
1	$R_1(B)$				RT=200	
2		$R_2(A)$		RT=150		
3			$R_3(C)$			RT=175
4	$W_1(B)$				WT=200 C=0	
5	$W_1(A)$			WT=200 C=0		
6		$W_2(C)$ Abort				
7			$W_3(A)$ Delay			

Multiversion Timestamp CC

- Multiversion CC
  - another way of using timestamps
  - ensures that a transaction never has to be restarted (aborted) to read an object
    - unlike timestamp-based CC
- The idea is to make several copies of each DB object
  - each copy of each object has a write timestamp
- Ti reads the most recent version whose timestamp precedes TS(Ti)

Multiversion Timestamp CC

- Idea: Let writers make a "new" copy while readers use an appropriate "old" copy:



Readers are always allowed to proceed

- But may be "blocked" until writer commits.

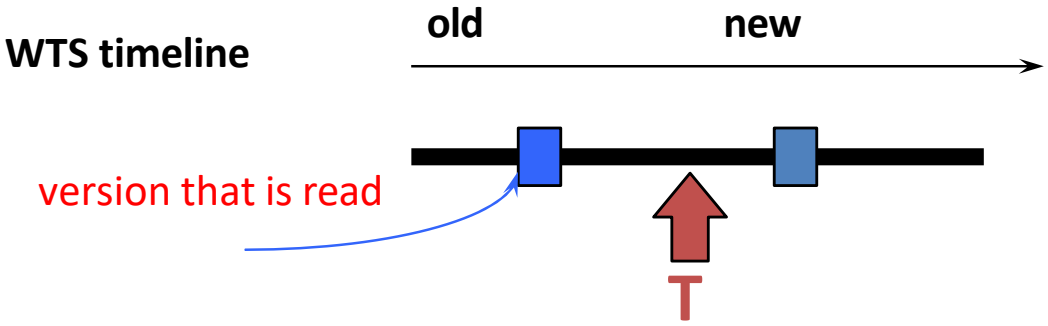
Multiversion CC (Contd.)

- Each version of an object has
  - its writer's TS as its WT, and
  - the timestamp of the transaction that most recently read this version as its RT
- Versions are chained backward
  - we can discard versions that are "too old to be of interest"
- Each transaction is classified as Reader or Writer.
  - Writer may write some object; Reader never will
  - Transaction declares whether it is a Reader when it begins

Reader Transaction

- For each object to be read:
  - Finds newest version with WT < TS(T)
  - Starts with current version in the main segment and chains backward through earlier versions
  - Update RT if necessary (i.e. if TS(T) > RT, then RT = TS(T))

- Assuming that some version of every object exists from the beginning of time, Reader transactions are never restarted
  - However, might block until writer of the appropriate version commits



Writer Transaction

- To read an object, follows reader protocol
- To write an object:
  - must make sure that the object has not been read by a ”later” transaction
  - Finds newest version V s.t. WT(V) <= TS(T).
- If RT(V) <= TS(T)
  - T makes a copy CV of V, with a pointer to V,
    with WT(CV) = TS(T), RT(CV) = TS(T)
  - Write is buffered until T commits; other transactions can see TS values but can’t read version CV
- Else
  - reject write

Example

- Four transactions T1 (TS = 150), T2 (TS = 200), T3 (TS = 175), T4(TS = 225)
- One object A
  - Initial version is A_0
- Sequence of actions
  - $R_1(A)$, $W_1(A)$, $R_2(A)$, $W_2(A)$, $R_3(A)$, $R_4(A)$
- Q. What is the state of the database at the end if the multiversion CC protocol is followed

Initial condition and Steps

A_0 existed before the transactions started

Step	T1	T2	T3	T4	A_0		
	150	200	175	225	RT=0, WT=0		
1	$R_1(A)$						
2	$W_1(A)$						
3		$R_2(A)$					
4		$W_2(A)$					
5			$R_3(A)$				
6				$R_4(A)$			

After Step 1

A_0 is the newest version with WT <= TS(T_1)
Read A_0

Step	T1	T2	T3	T4	A_0		
	150	200	175	225	RT=0, WT=0		
1	$R_1(A)$				Read RT = 150		
2	$W_1(A)$						
3		$R_2(A)$					
4		$W_2(A)$					
5			$R_3(A)$				
6				$R_4(A)$			

After Step 2

- A_0 is the newest version with WT <= TS(T_1)
- RT(A_0) <= TS(T_1)
- Create a new version A_{150}
- Set its WT, RT to TS(T_1) = 150 (A_{150} named accordingly)

Step	T1	T2	T3	T4	A_0	A_{150}	
	150	200	175	225	RT=150 WT=0	RT=150 WT=150	
1	$R_1(A)$				Read RT = 150		
2	$W_1(A)$					Create RT=150 WT=150	
3		$R_2(A)$					
4		$W_2(A)$					
5			$R_3(A)$				
6				$R_4(A)$			

After Step 3

- A_{150} is the newest version with WT <= TS(T_2)
- Read A_{150}
- Update RT

Step	T1	T2	T3	T4	A_0	A_{150}	
	150	200	175	225	RT=150 WT=0	RT=200 WT=150	
1	$R_1(A)$				Read		
2	$W_1(A)$					Create RT=150 WT=150	
3		$R_2(A)$				Read RT=200	
4		$W_2(A)$					
5			$R_3(A)$				
6				$R_4(A)$			

After Step 4

- A_{150} is the newest version with WT <= $TS(T_2)$
- $RT(A_{150})$ <= $TS(T_2)$
- Create a new version A_{200}
- Set its WT, RT to $TS(T_2)$ = 200 (A_{200} named accordingly)

Step	T1	T2	T3	T4	A_0	A_{150}	A_{200}
	150	200	175	225	RT=150 WT=0	RT=200 WT=150	RT=200 WT=200
1	$R_1(A)$				Read		
2	$W_1(A)$					Create RT=150 WT=150	
3		$R_2(A)$				Read RT=200	
4		$W_2(A)$					Create RT=200 WT=200
5			$R_3(A)$				
6				$R_4(A)$			

After Step 5

- A_{150} is the newest version with WT <= TS(T_3)
- Read A_{150}
- DO NOT Update RT

Step	T1	T2	T3	T4	A_0	A_{150}	A_{200}
	150	200	175	225	RT=150 WT=0	RT=200 WT=150	RT=200 WT=200
1	$R_1(A)$				Read		
2	$W_1(A)$					Create RT=150 WT=150	
3		$R_2(A)$				Read RT=200	
4		$W_2(A)$					Create RT=200 WT=200
5			$R_3(A)$			Read	
6				$R_4(A)$			

After Step 6

- A_{200} is the newest version with WT <= TS(T_4)
- Read A_{200}
- Update RT

Step	T1	T2	T3	T4	A_0	A_{150}	A_{200}
	150	200	175	225	RT=150 WT=0	RT=200 WT=150	RT=225 WT=200
1	$R_1(A)$				Read		
2	$W_1(A)$					Create RT=150 WT=150	
3		$R_2(A)$				Read RT=200	
4		$W_2(A)$					Create RT=200 WT=200
5			$R_3(A)$			Read	
6				$R_4(A)$			Read RT=225

Multi-version technique based on timestamp ordering

To ensure serializability, the following two rules are used.

1. If transaction T issues write_item (X) and version i of X has the highest write_TS(Xi) of all versions of X that is also less than or equal to TS(T), and read _TS(Xi) > TS(T) (T is older than read_TS), then abort and roll-back T; otherwise create a new version Xi and read_TS(X) = write_TS(Xj) = TS(T).

2. If transaction T issues read_item (X), find the version i of X that has the highest write_TS(Xi) of all versions of X that is also less than or equal to TS(T), then return the value of Xi to T, and set the value of read_TS(Xi) to the largest of TS(T) and the current read_TS(Xi).

Rule 2 guarantees that a read will never be rejected.

Multi-version Two-Phase Locking Using Certify Locks

- Concept
  - Allow a transaction T' to read a data item X while it is write locked by a conflicting transaction T.
  - This is accomplished by maintaining two versions of each data item X where one version must always have been written by some committed transaction. This means a write operation always creates a new version of X.

Multi-version Two-Phase Locking Using Certify Locks

- Steps
  - X is the committed version of a data item.
  - T creates a second version X' after obtaining a write lock on X.
  - Other transactions continue to read X.
  - T is ready to commit so it obtains a certify lock on X'.
  - The committed version X becomes X'.
  - T releases its certify lock on X', which is X now.

Compatibility tables for

	Read	Write
Read	yes	no
Write	no	no

read/write locking scheme

	Read	Write	Certify
Read	yes	yes	no
Write	yes	no	no
Certify	no	no	no

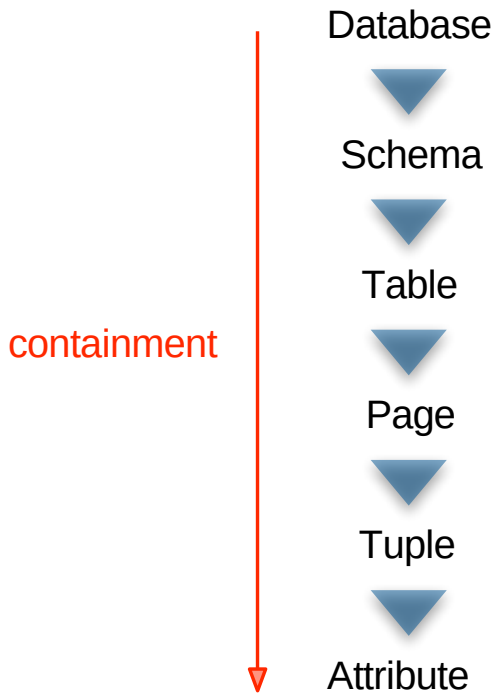
read/write/certify locking scheme

Multi-version Two-Phase Locking Using Certify Locks

- Note:
  - In multiversion 2PL read and write operations from conflicting transactions can be processed concurrently.
  - This improves concurrency but it may delay transaction commit because of obtaining certify locks on all its writes. It avoids cascading abort but like strict two phase locking scheme conflicting transactions may get deadlocked.

Multiple granularity locks

- *What* should we *lock*?
  Tuples, pages, tables, ...
- But there is an *implicit containment*
- *Idea*: *lock* DB objects *hierarchically*



The phantom problem

- If we *relax* the *assumption* that the *DB* is a *fixed collection* of objects, *even Strict 2PL* will *not assure serialisability*!
  - *T1 locks all pages* containing *sailor records* with *rating* = 1, and *finds oldest sailor* (say, *age* = 71)
  - Next, *T2 inserts* a *new sailor*: *rating* = 1, *age* = 96
  - *T2* also *deletes oldest sailor* with *rating* = 2 (and, say, age=80), and *commits*
  - *T1* now *locks all pages* containing *sailor records* with *rating* = 2, and *finds oldest* (say, age=63)
- *No lock conflicts*, *but also no consistent DB state* where T1 is "correct"!

The problem

- *T1 implicitly assumes* that it has *locked the* set of *all sailor* records with *rating* = 1
  - The *assumption* only *holds if no sailor* records are *added while T1* is *executing*!
  - We *need* some *mechanism* to *enforce* this *assumption*
    - *Index locking*
    - *Predicate locking*
- The *example shows* that *conflict serialisability guarantees serialisability* only *if* the set of *objects* is *fixed*!

Hierarchical locks and new locking modes

- *Allow transactions* to *lock* at *each level* of the *hierarchy*
- Introduce *"intention" locks*: *IS* and *IX*
  - *Before locking* an item, a *transaction must introduce intention locks* on *all* the *item's ancestors* in the *hierarchy*
  - *Release locks* in *reverse order*
- One *extra lock*: *SIX* — "share, with intention to write"

Compatibility matrix

held lock

wanted lock	NL	IS	IX	SIX	S	X
NL	Y	Y	Y	Y	Y	Y
IS	Y	Y	Y	Y	Y	N
IX	Y	Y	Y	N	N	N
SIX	Y	Y	N	N	N	N
S	Y	Y	N	N	Y	N
X	Y	N	N	N	N	N

In more detail

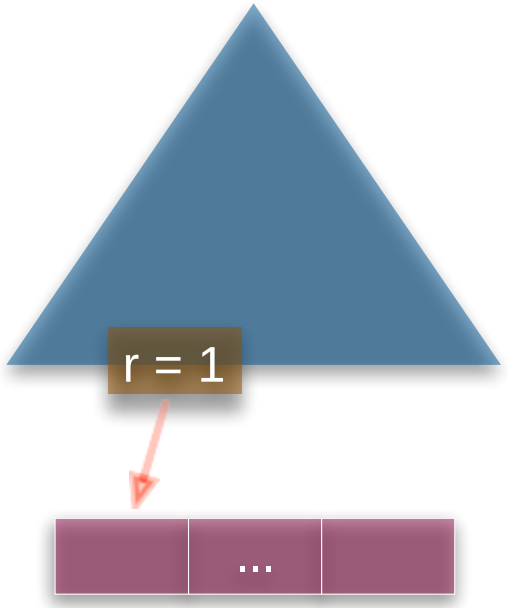
- *Each transaction starts* from the *root* of the *hierarchy*
- To *obtain S* or *IS lock on* a *node*, *must hold IS* or *IX* on *parent node*
  - What if a transaction holds SIX on parent? S on parent?
- To *obtain X* or *IX* or *SIX* on a *node*, *must hold IX* or *SIX* on *parent node*
- *Must release* locks in *bottom-up order*

A few examples

- *T1 scans R*, and *updates* a few *tuples*
  - *T1* gets an *SIX lock* on *R*, then *repeatedly* gets an *S lock* on *tuples of R*, and *occasionally upgrades* to *X* on the *tuples*
- *T2 uses an index* to *read* only *part of R*
  - *T2* gets an *IS lock* on *R*, and *repeatedly* gets an *S lock* on *tuples of R*
- *T3 reads all of R*
  - *T3* gets an *S lock* on the *entire relation*
  - *Or*, it gets an *IS lock* on *R*, *escalating* to *S lock* on every tuple

Index locking

- *If* there is an *index* on the *rating field*, *T1* should *lock* the *index page* containing the *data entries* with *rating* = 1
  - *If* there are *no records* with *rating* = 1, *T1 must lock* the *index page* where such a *data entry would be*, *if* it *existed*!
- *If* there is *no suitable index*, *T1 must lock all pages*, and *lock* the *file/table* to *prevent* new *pages* from being *added*, to *ensure* that *no* new *records* with *rating* = 1 are *added*



Predicate locking

- *Grant lock* on all *records* that *satisfy* some *logical predicate*, *e.g.*, *salary* > 2000
  - *Index locking* is a *special case* of *predicate locking* for which an *index supports* efficient *implementation* of the *predicate lock*
- *In general*, *predicate locking* imposes a *lot of locking overhead*

B+tree locking

- *How* can we *efficiently lock* a *particular node*?
  - This is *entirely different* than *multiple granularity locking* (why?)
- One *solution*: *ignore* the *tree structure*, just *lock pages* while *traversing* the tree, following *2PL*
  - *Terrible performance*
  - *Root* node (and many *higher level nodes*) become *bottlenecks* because *every tree access* begins at the *root*

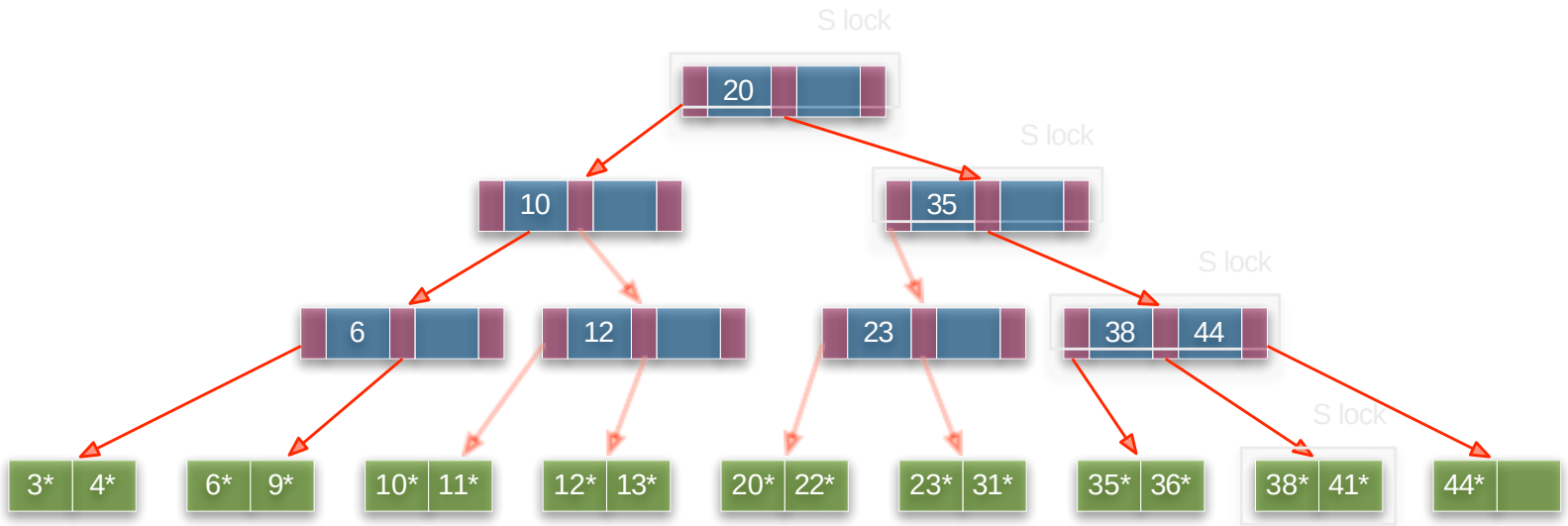
Key observations

- *Higher levels* of the tree *only direct searches* to leaf pages
- For *insertions*, a *node* on a *path* from the *root* to a modified *leaf* must be *locked* (in *X mode*, of course), *only if* a *split* can *propagate up* to it *from* the *modified leaf* (similar point holds for deletions)
- We can *exploit* these *observations* to design *efficient locking protocols* that *guarantee serialisability* even though they *violate 2PL*

The basic algorithm

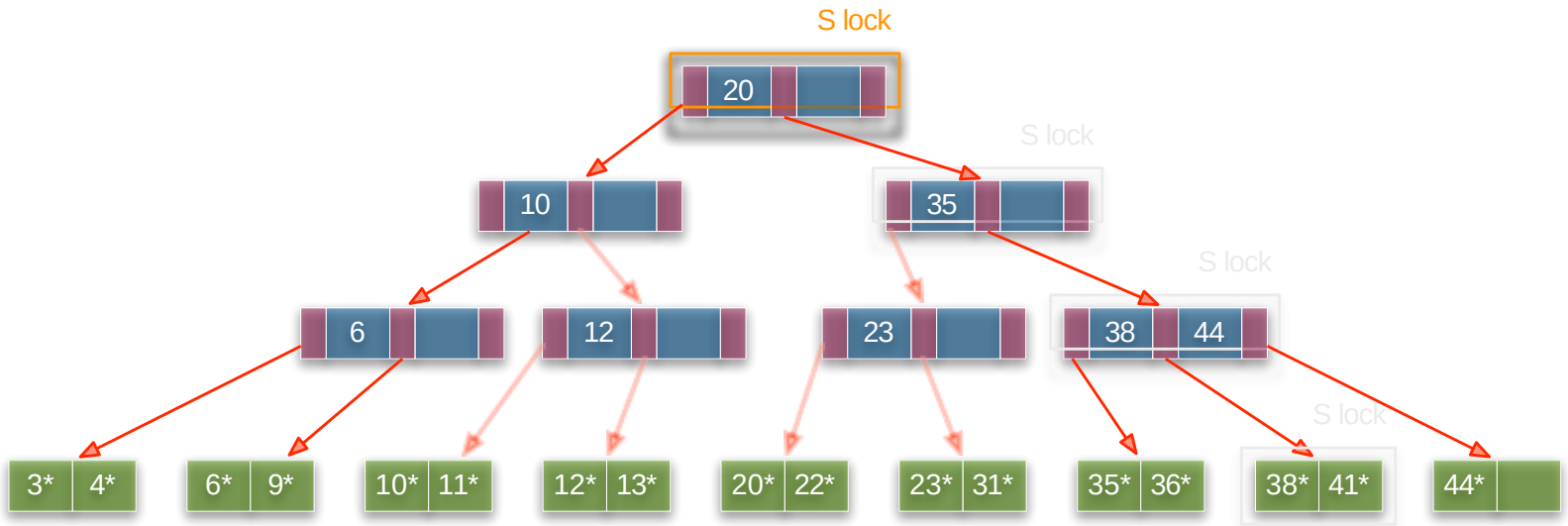
- *Search*: *start* at *root* and *descend*; repeatedly, *S lock child* then *unlock parent*
- *Insert/Delete*: *start* at *root* and *descend*, obtaining *X locks as needed*; once *child* is *locked*, *check* if it is *safe*:
  - *Safe node*: a *node* such that *changes* will *not propagate* up *beyond* this *node*
    - *Insertion*: *node* is *not full*
    - *Deletion*: *node* is *not half-empty*
  - If *child* is *safe*, *release* all *locks* on *ancestors*

Example: search 38*



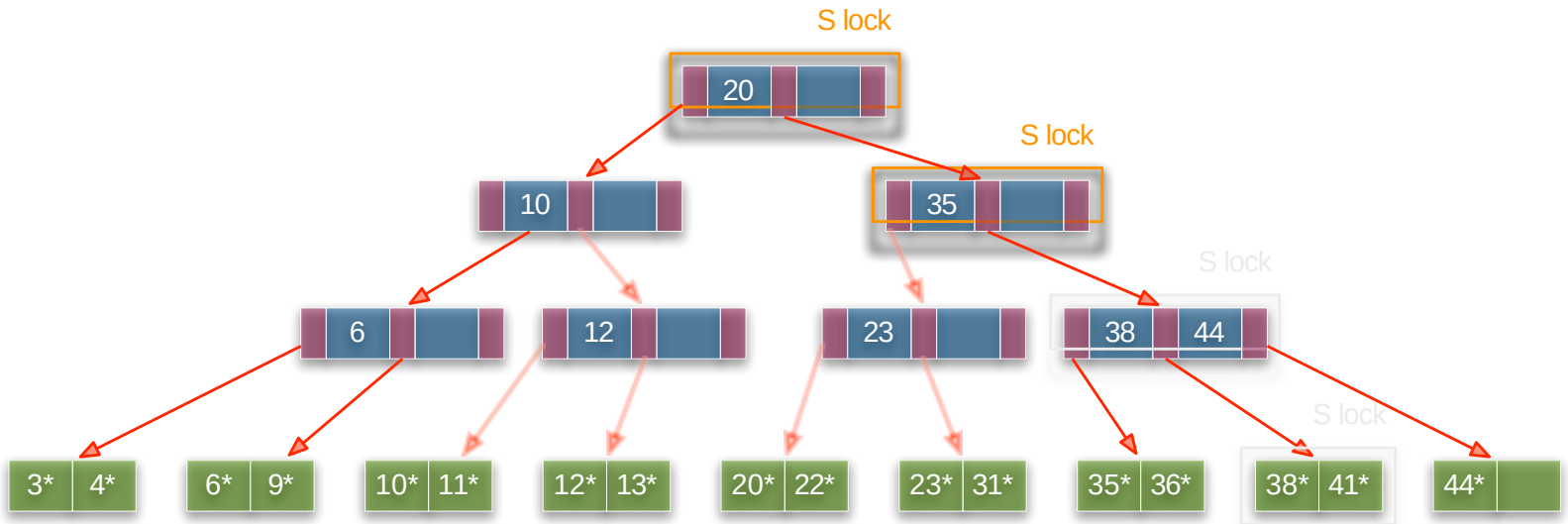
Obtain and *release* *S-locks level-by-level*

Example: search 38*



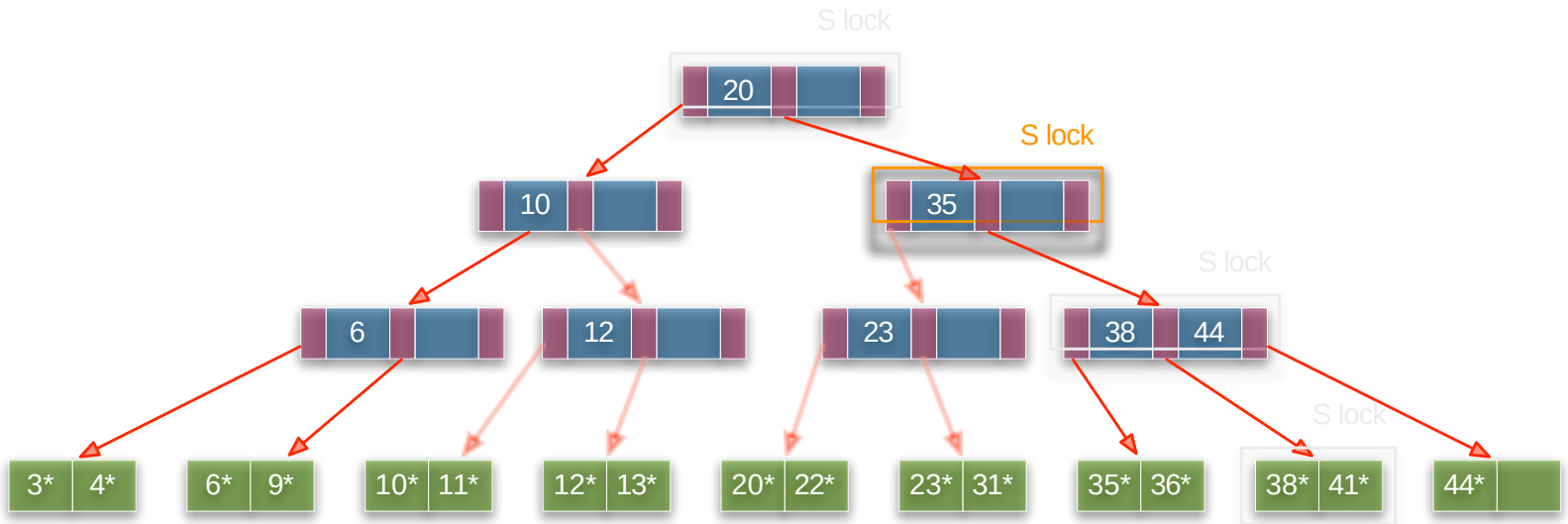
Obtain and *release* *S-locks* *level-by-level*

Example: search 38*



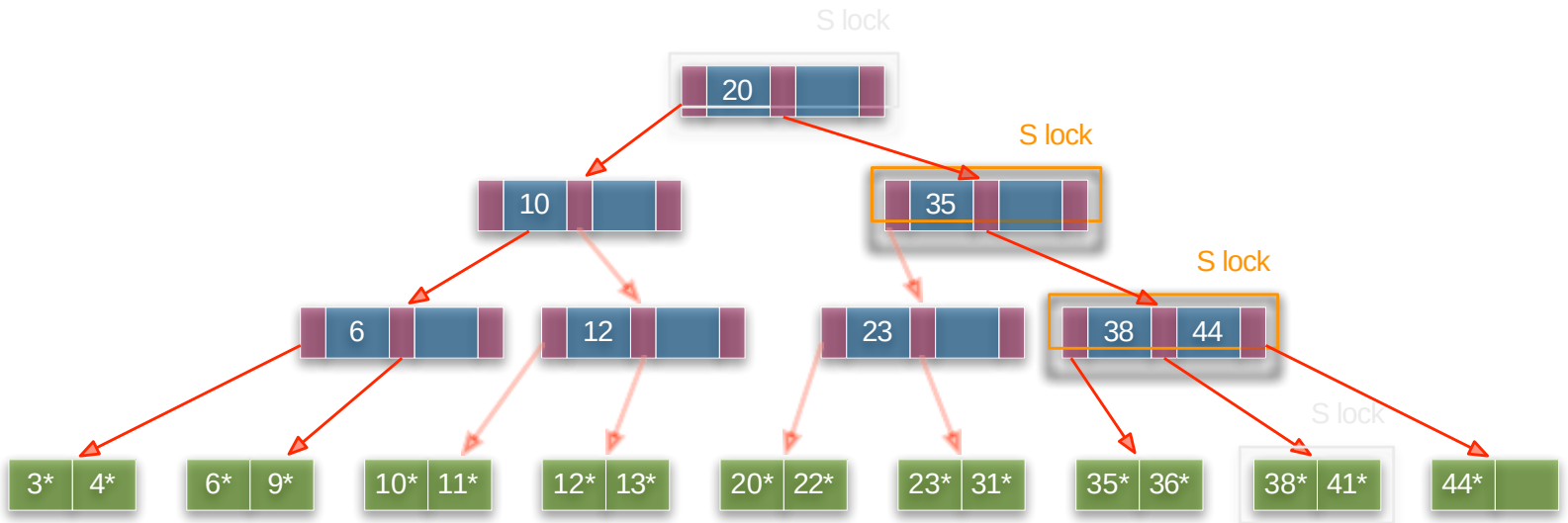
Obtain and *release* *S-locks* *level-by-level*

Example: search 38*



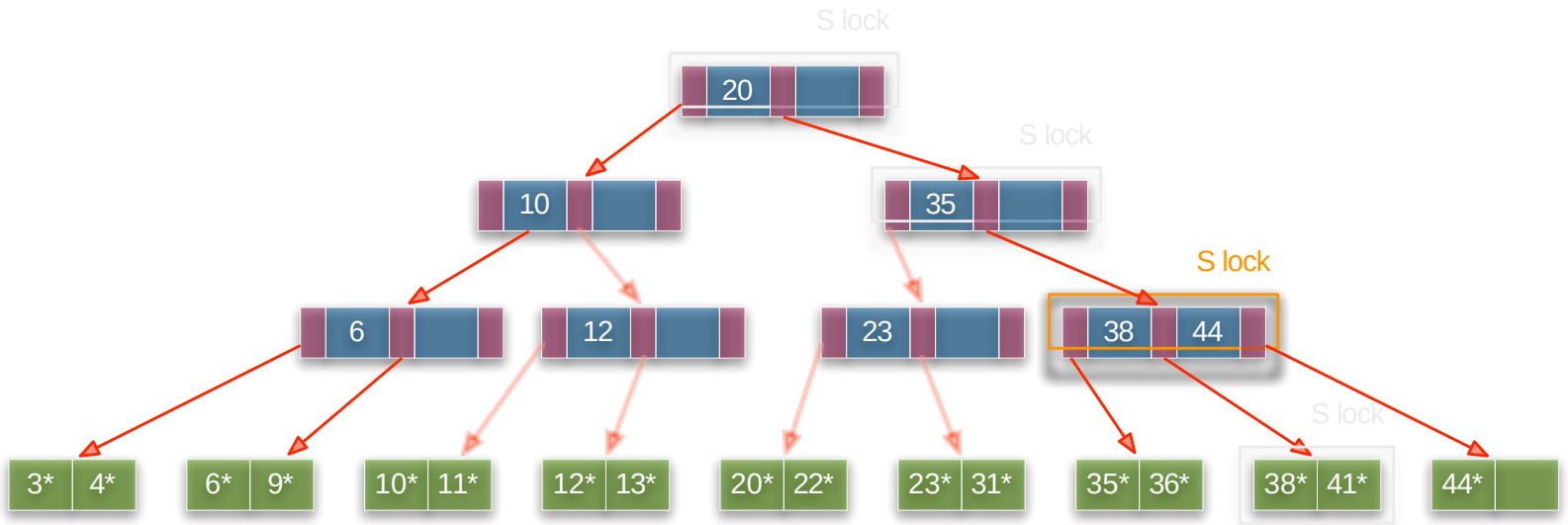
Obtain and *release* *S-locks level-by-level*

Example: search 38*



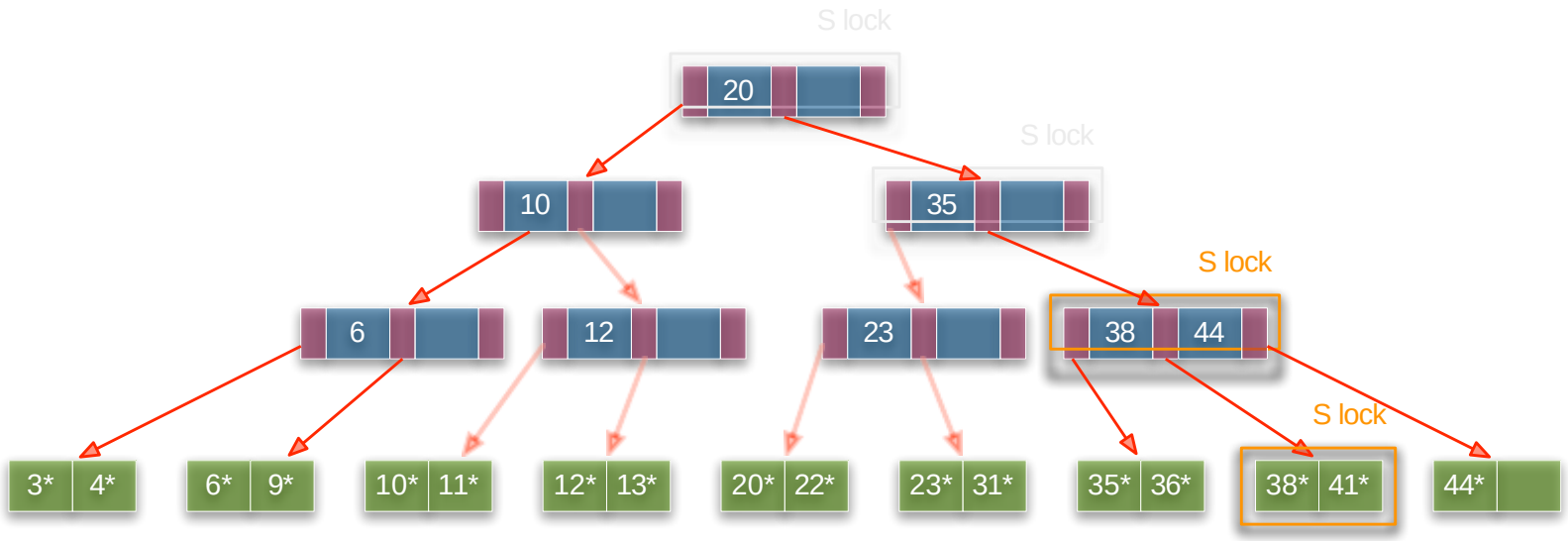
Obtain and *release* *S-locks level-by-level*

Example: search 38*



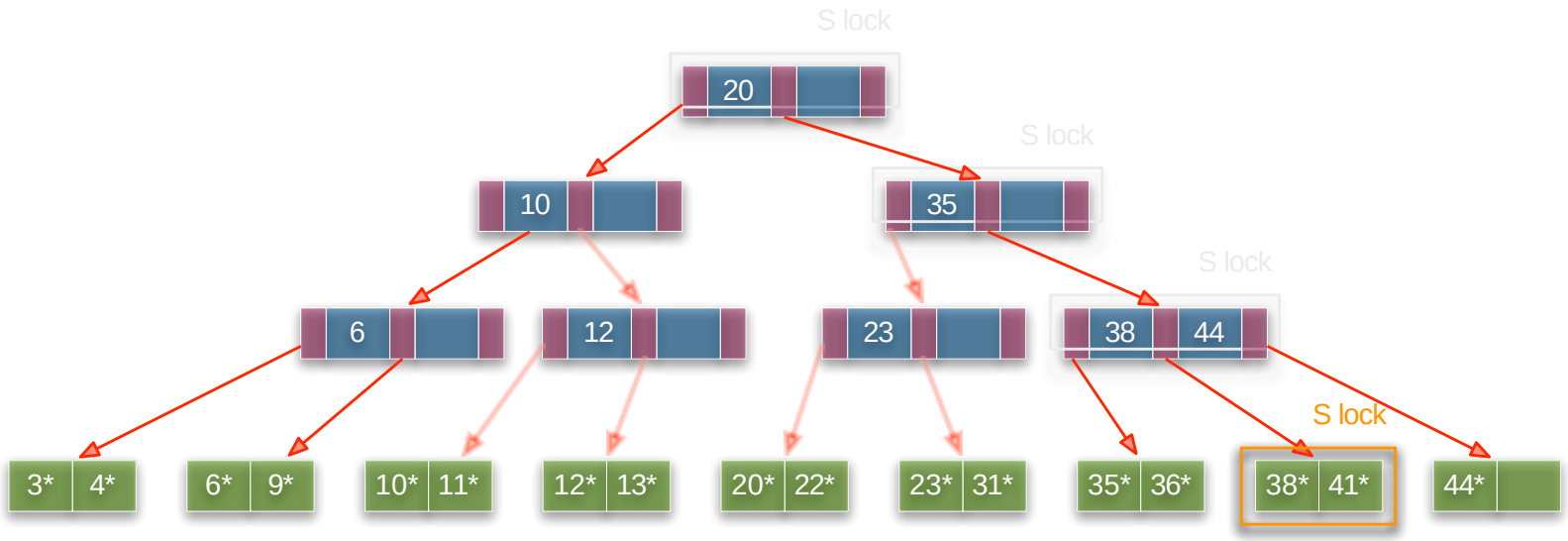
Obtain and *release* *S-locks* *level-by-level*

Example: search 38*



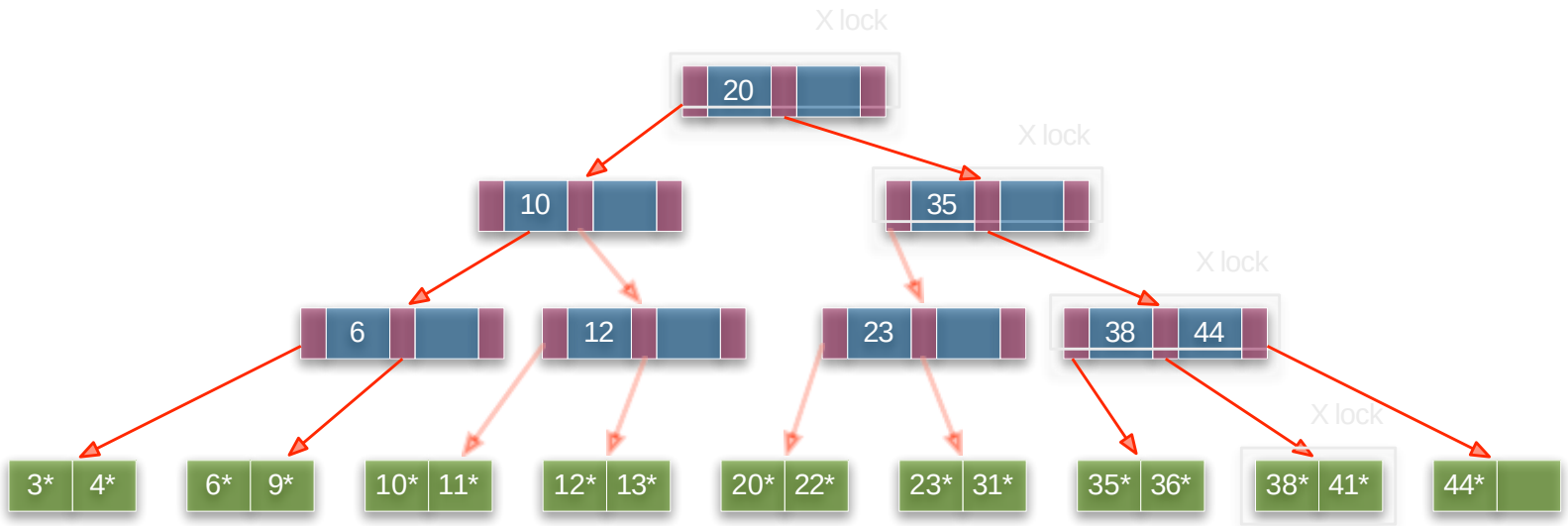
Obtain and *release* *S-locks level-by-level*

Example: search 38*



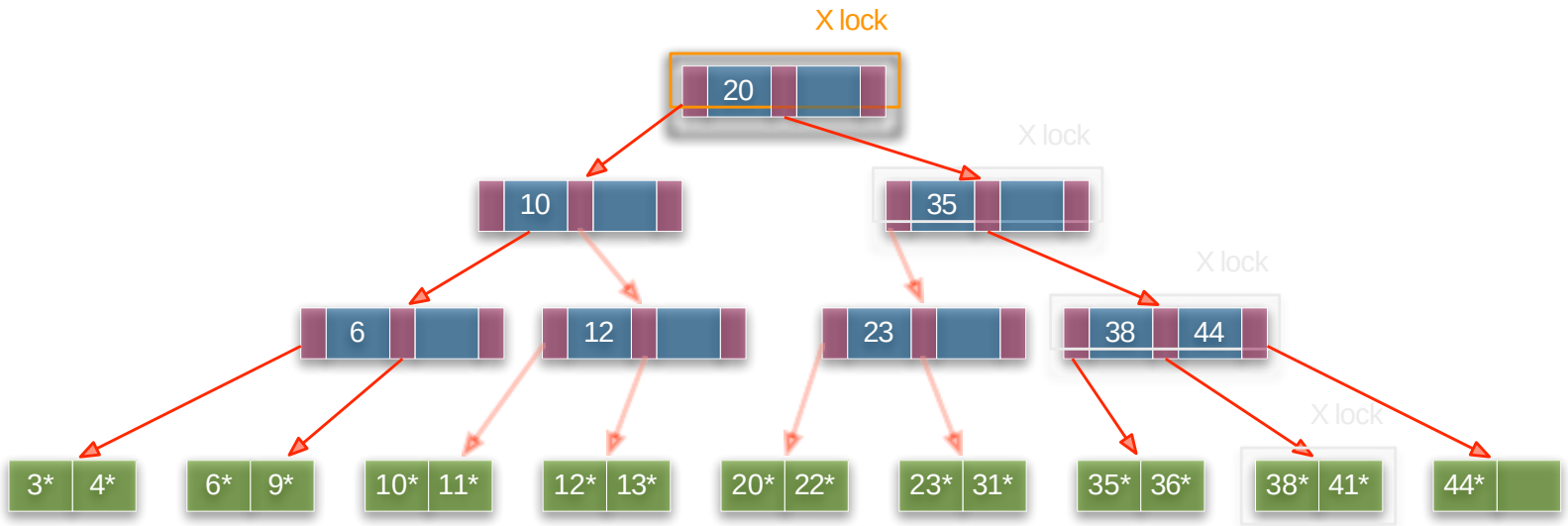
Obtain and *release* *S-locks level-by-level*

Example: delete 38*



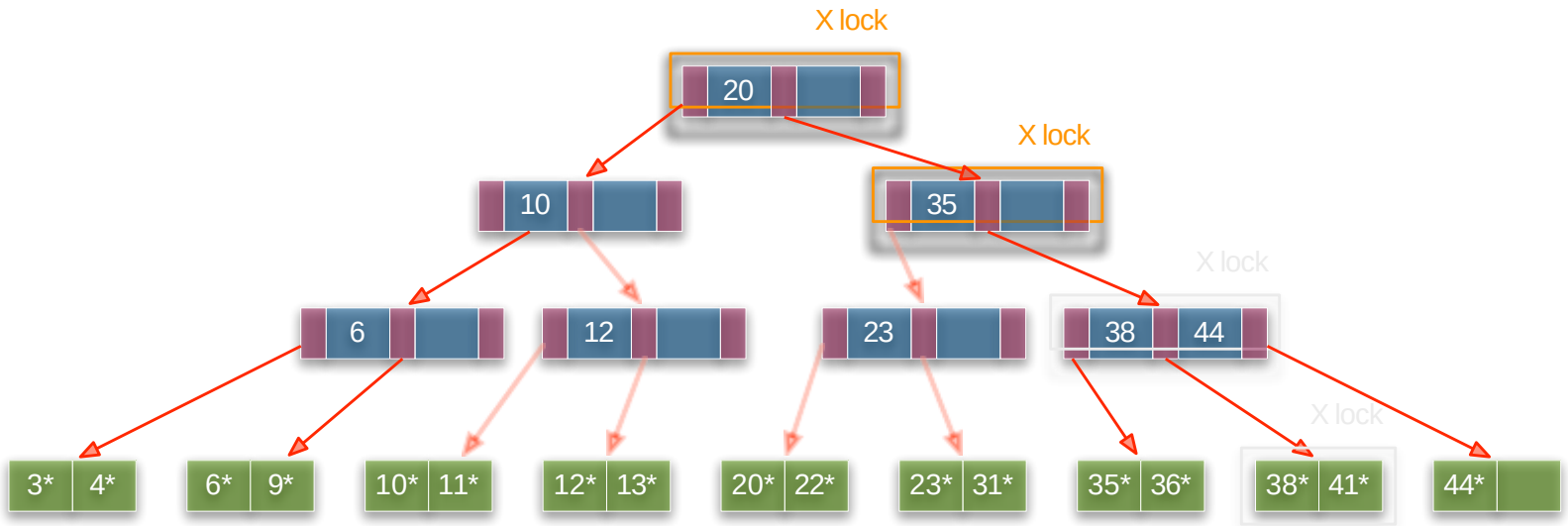
Obtain X-locks while *descending*; *release them top-down* once the node is *designated safe*

Example: delete 38*



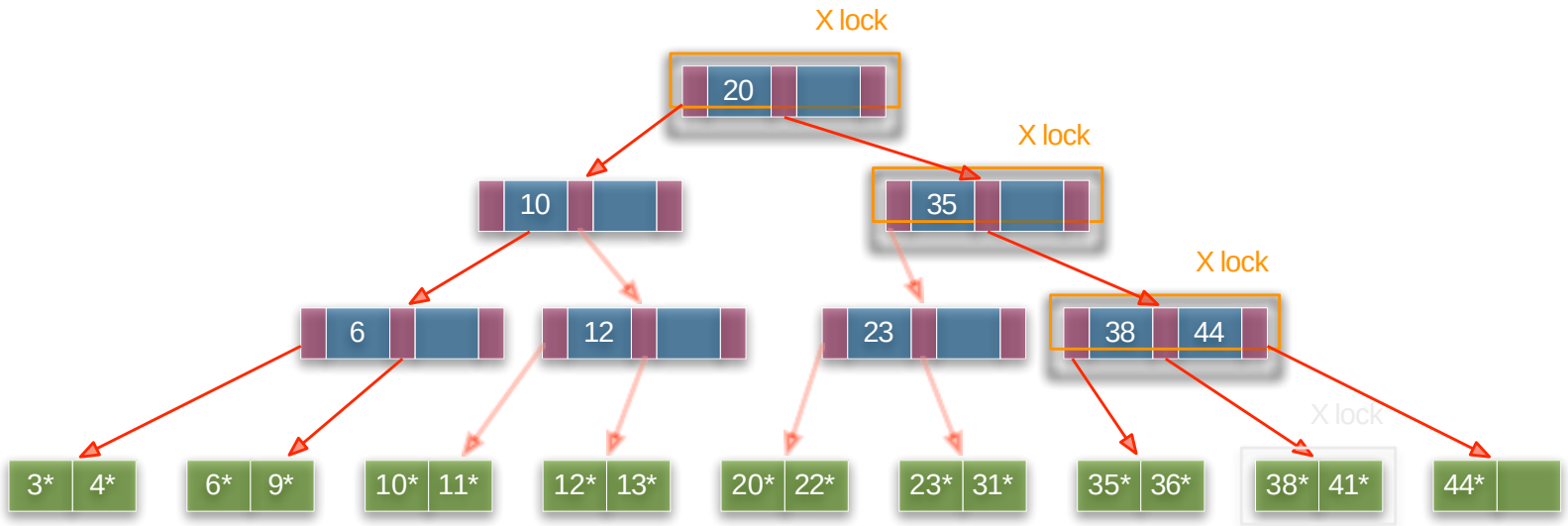
Obtain X-locks while *descending*; *release them top-down* once the node is *designated safe*

Example: delete 38*



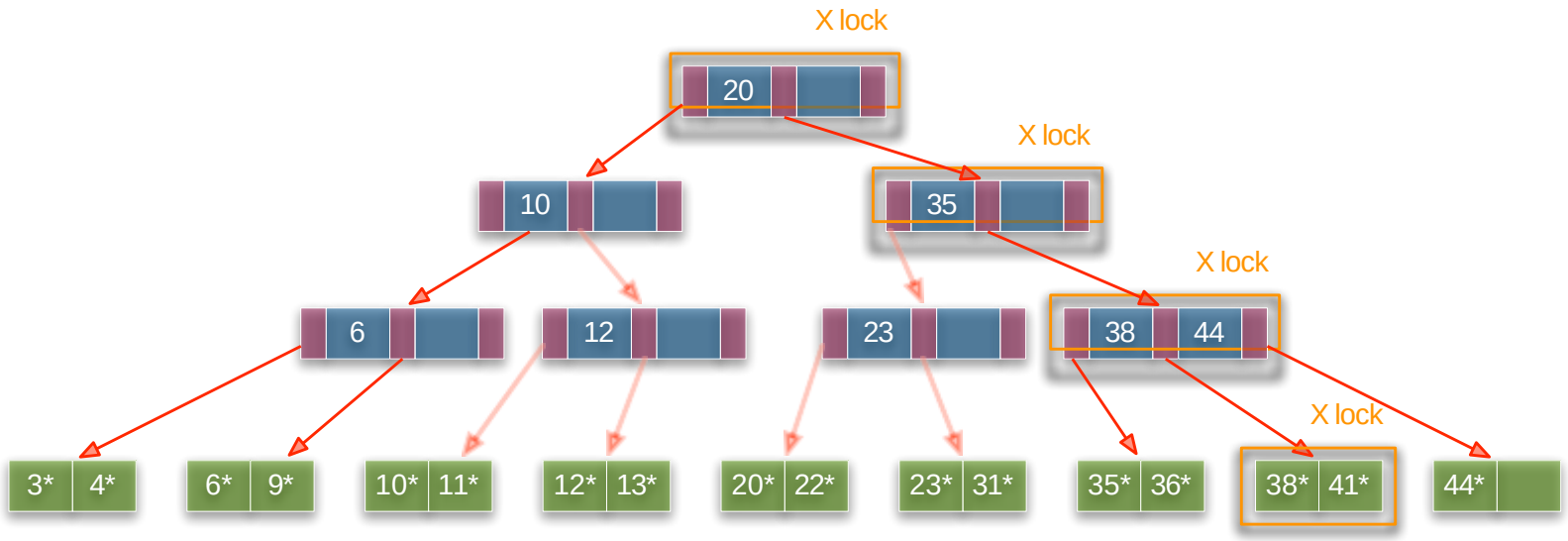
Obtain X-locks while *descending*; *release them top-down* once the node is *designated safe*

Example: delete 38*



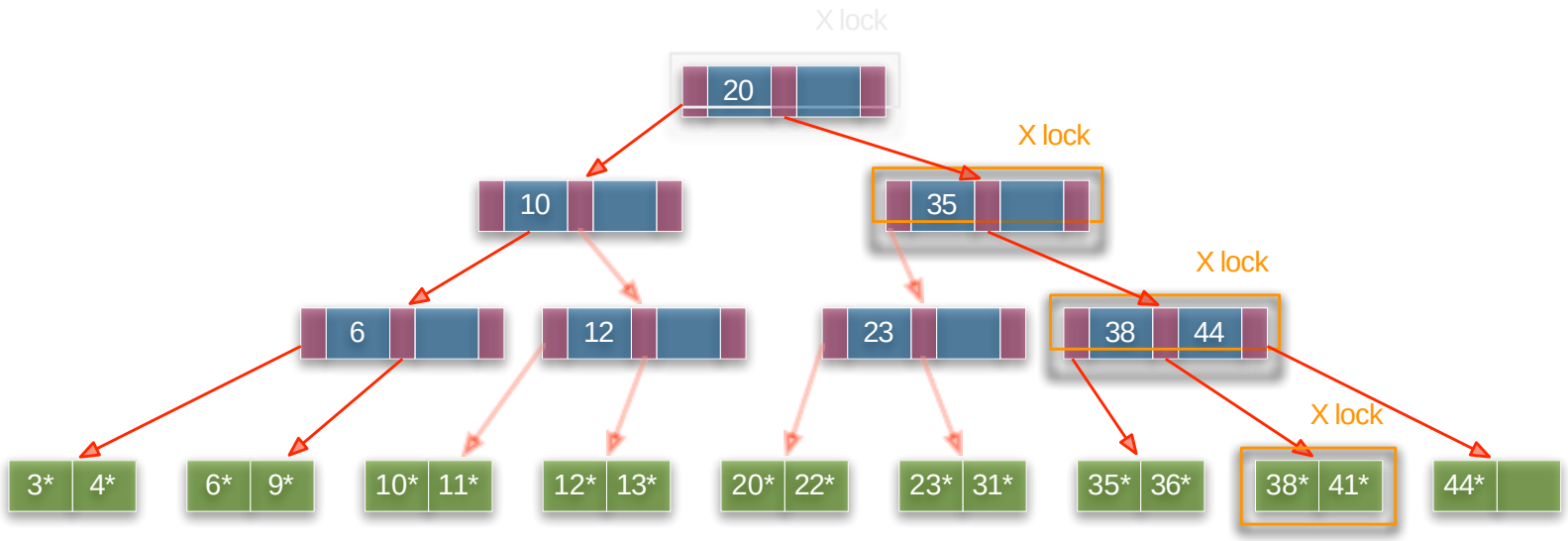
Obtain X-locks while *descending*; *release them top-down* once the node is *designated safe*

Example: delete 38*



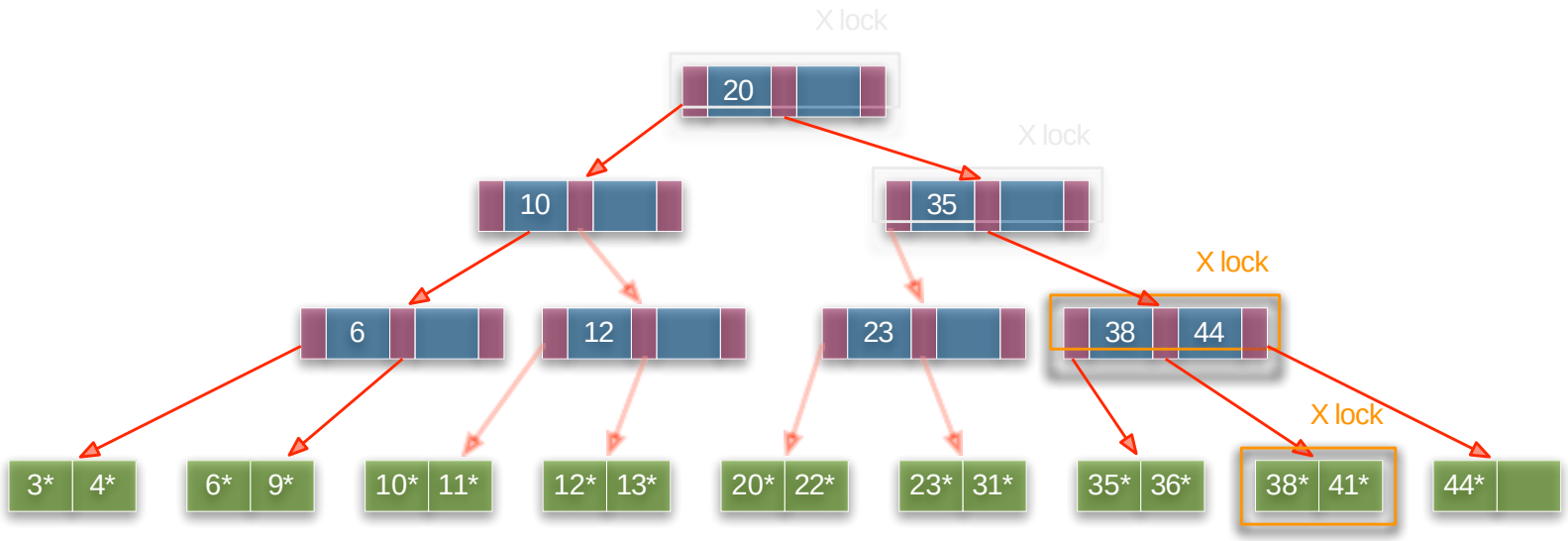
Obtain X-locks while *descending*; *release them top-down* once the node is *designated safe*

Example: delete 38*



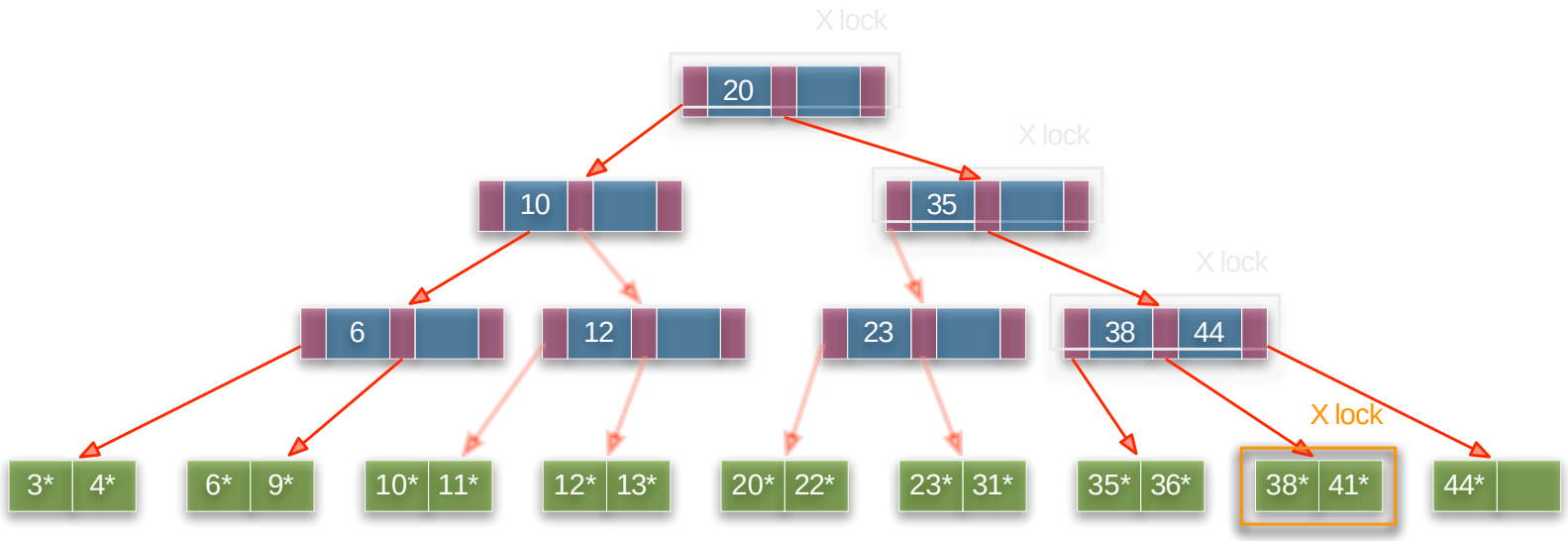
Obtain X-locks while *descending*; *release them top-down* once the node is *designated safe*

Example: delete 38*



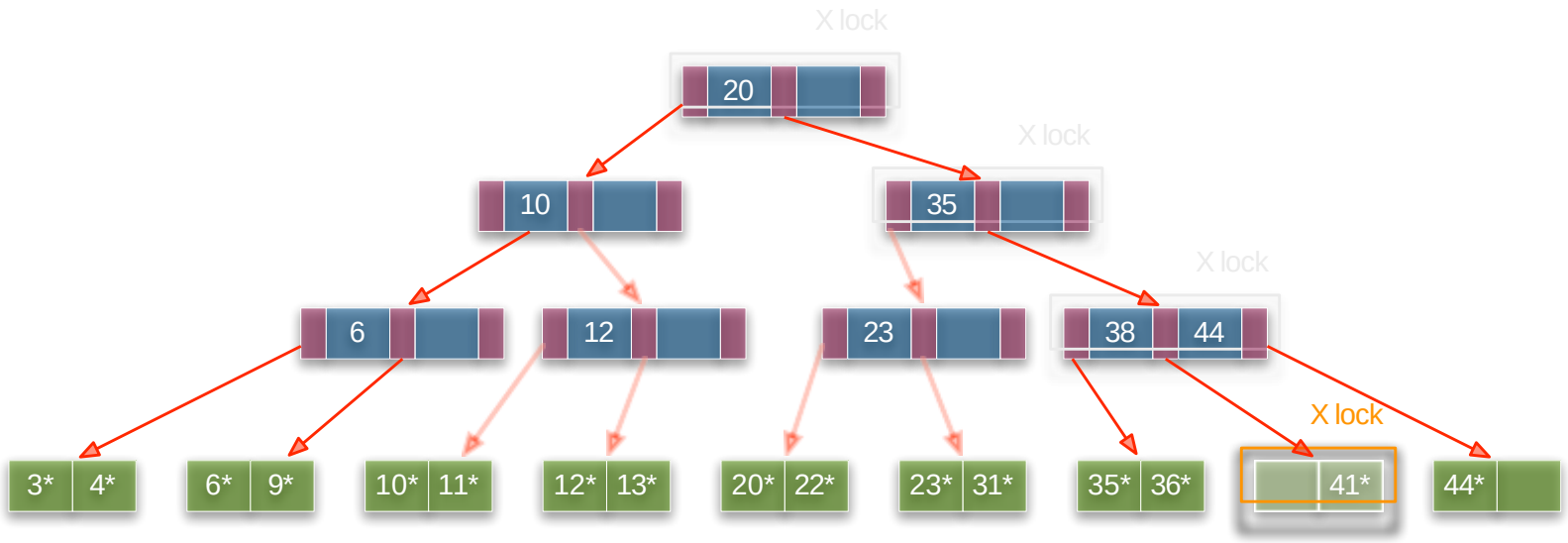
Obtain X-locks while *descending*; *release them top-down* once the node is *designated safe*

Example: delete 38*



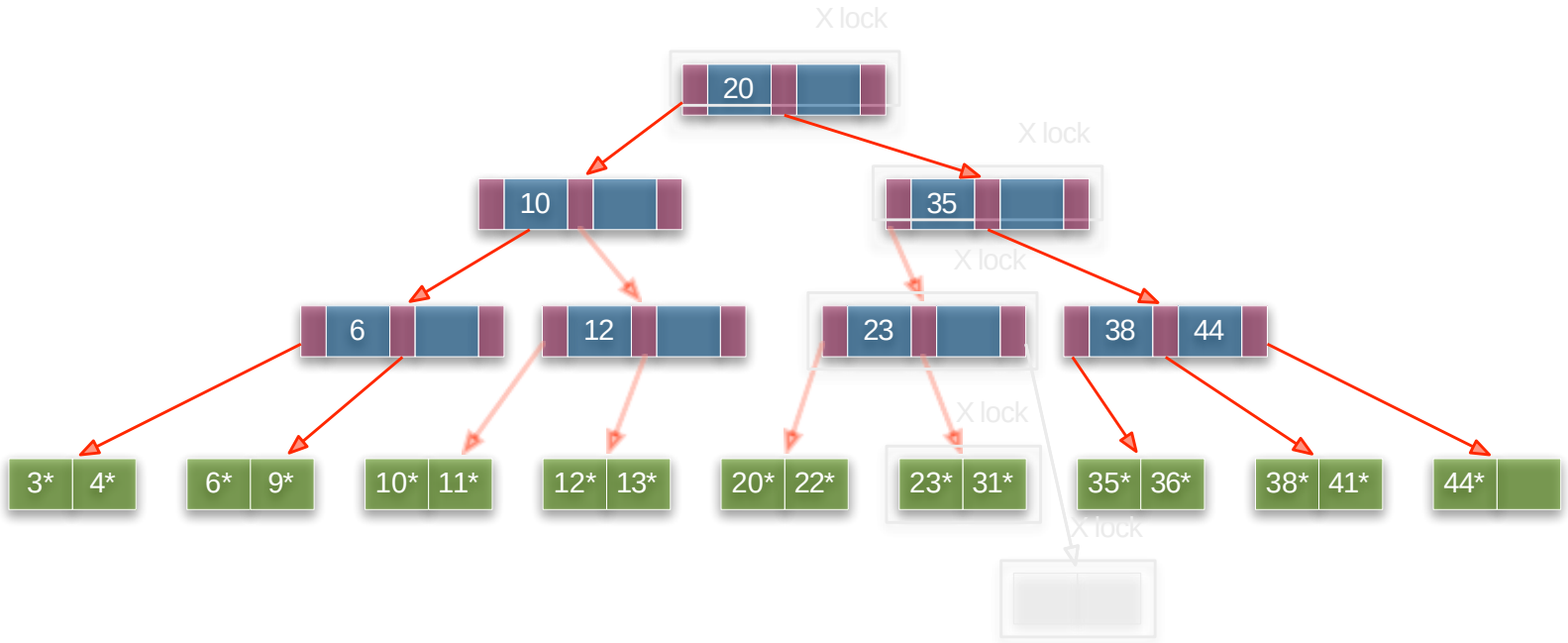
Obtain X-locks while *descending*; *release them top-down* once the node is *designated safe*

Example: delete 38*



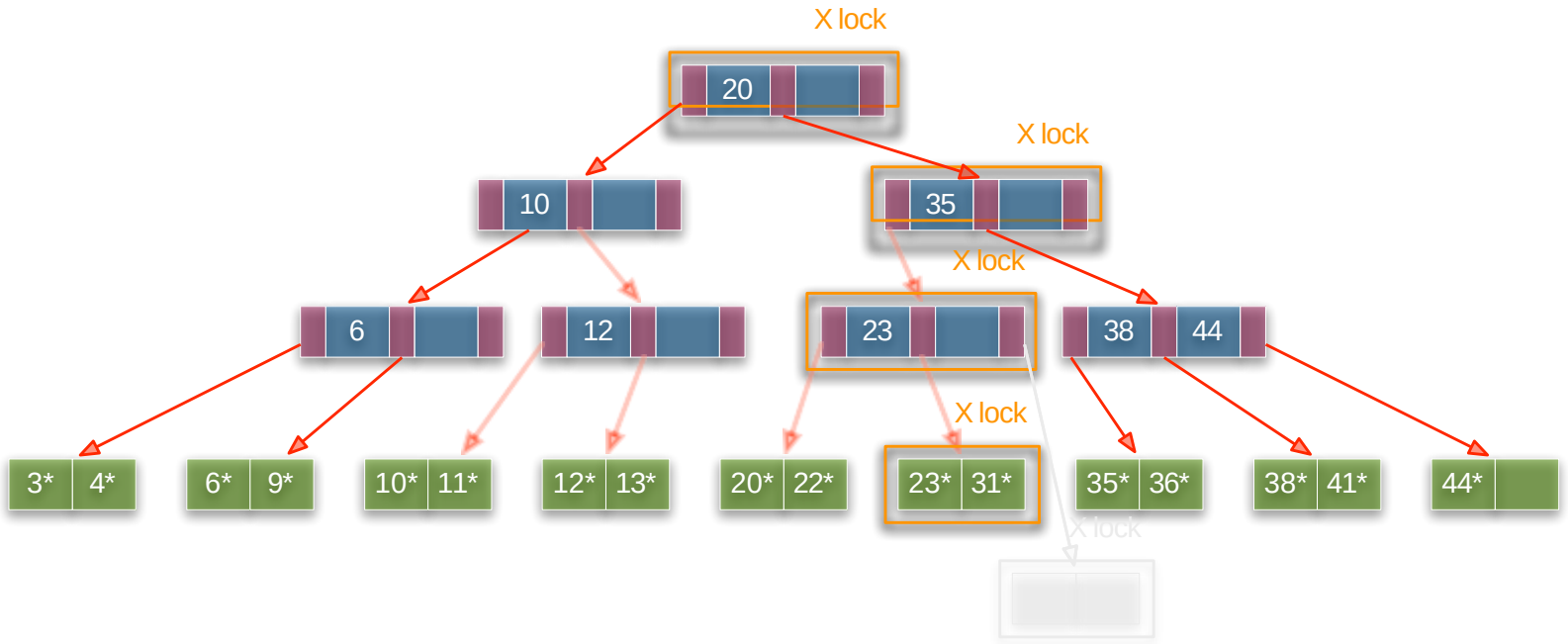
Obtain X-locks while *descending*; *release them top-down* once the node is *designated safe*

Example: insert 25*



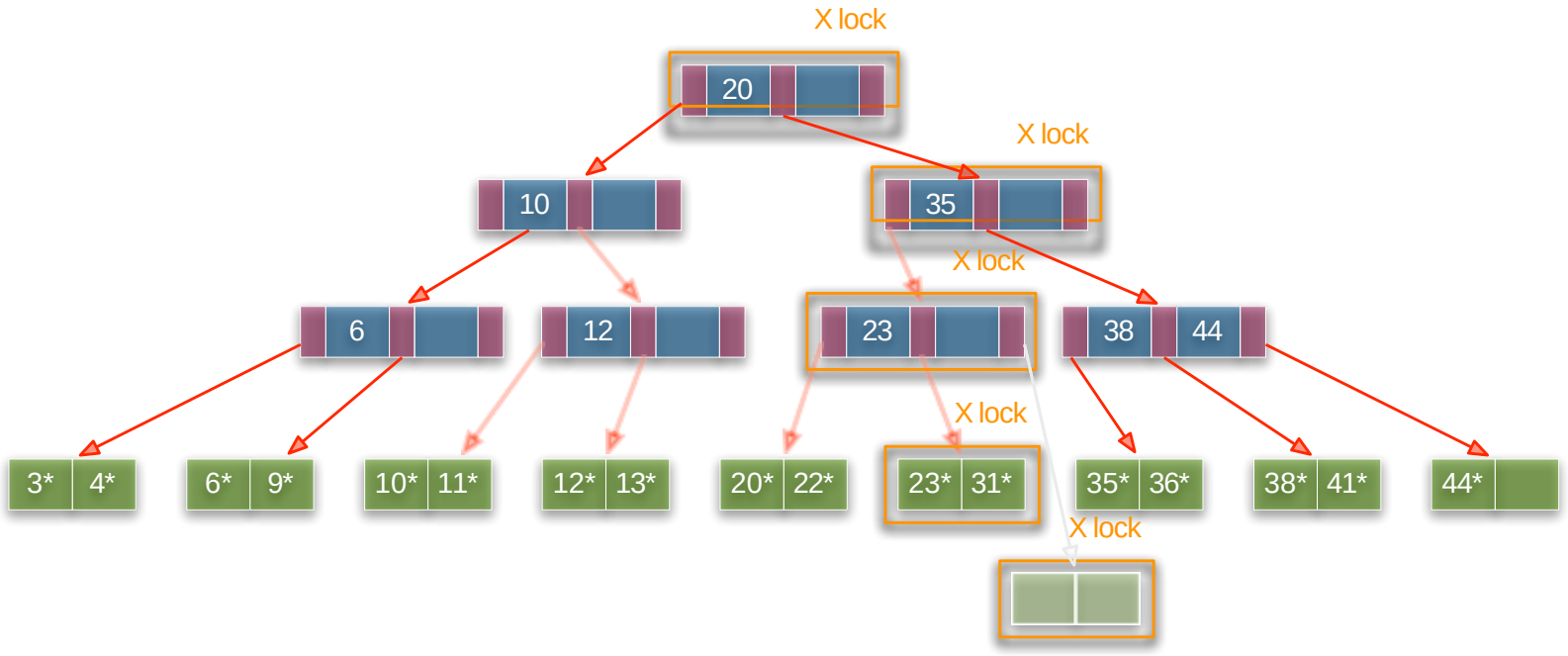
Obtain X-locks while *descending*; *leaf-node is not safe* so *create* a *new* one and *lock it in X-mode*; first release *locks on leaves* and then the rest *top-down*

Example: insert 25*



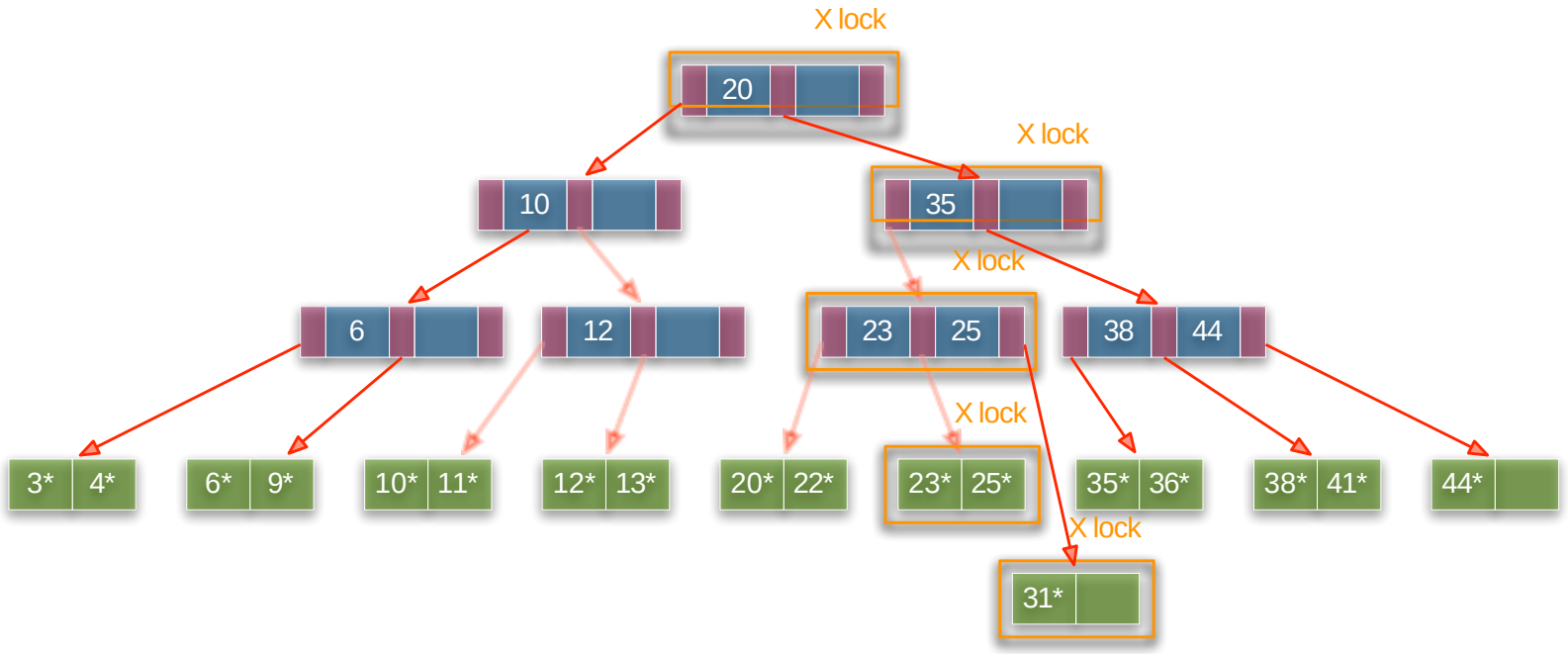
Obtain X-locks while *descending*; *leaf-node is not safe* so *create* a *new* one and *lock it in X-mode*; first release *locks on leaves* and then the rest *top-down*

Example: insert 25*



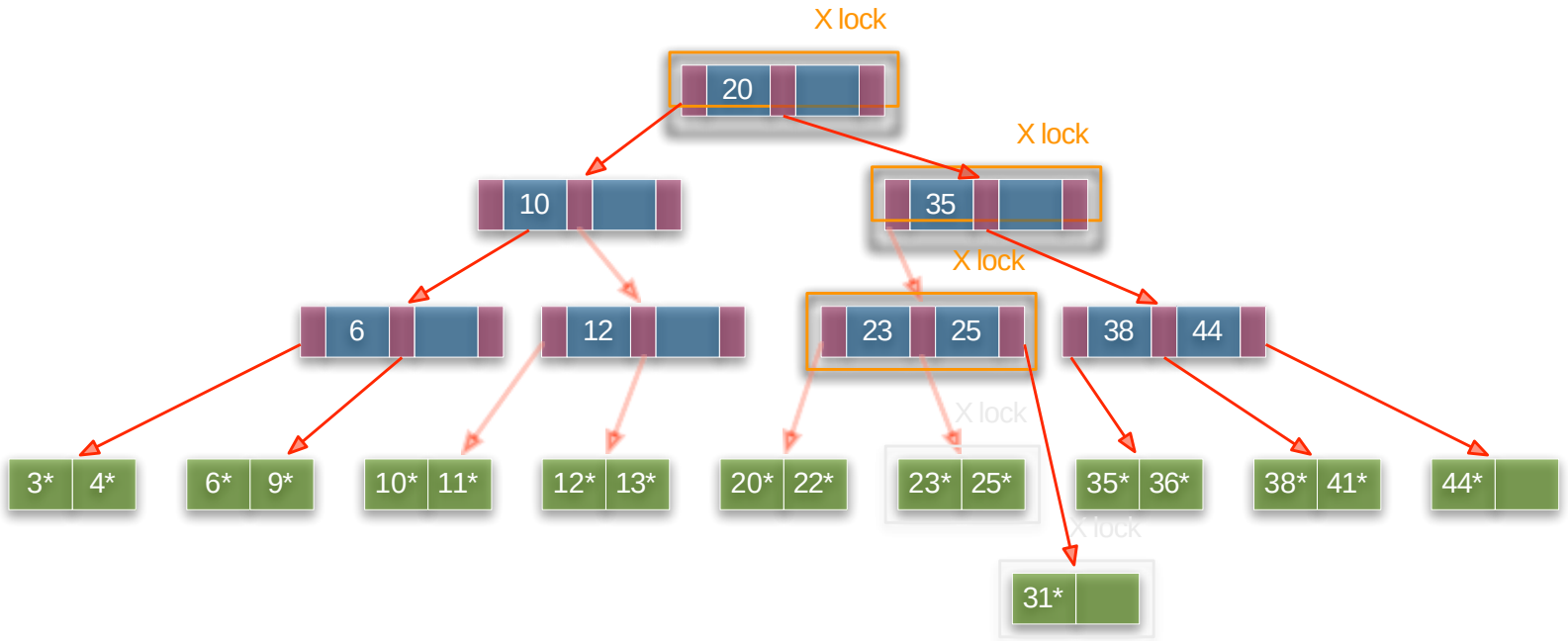
Obtain X-locks while *descending*; *leaf-node is not safe* so *create* a *new* one and *lock it in X-mode*; first release *locks on leaves* and then the rest *top-down*

Example: insert 25*



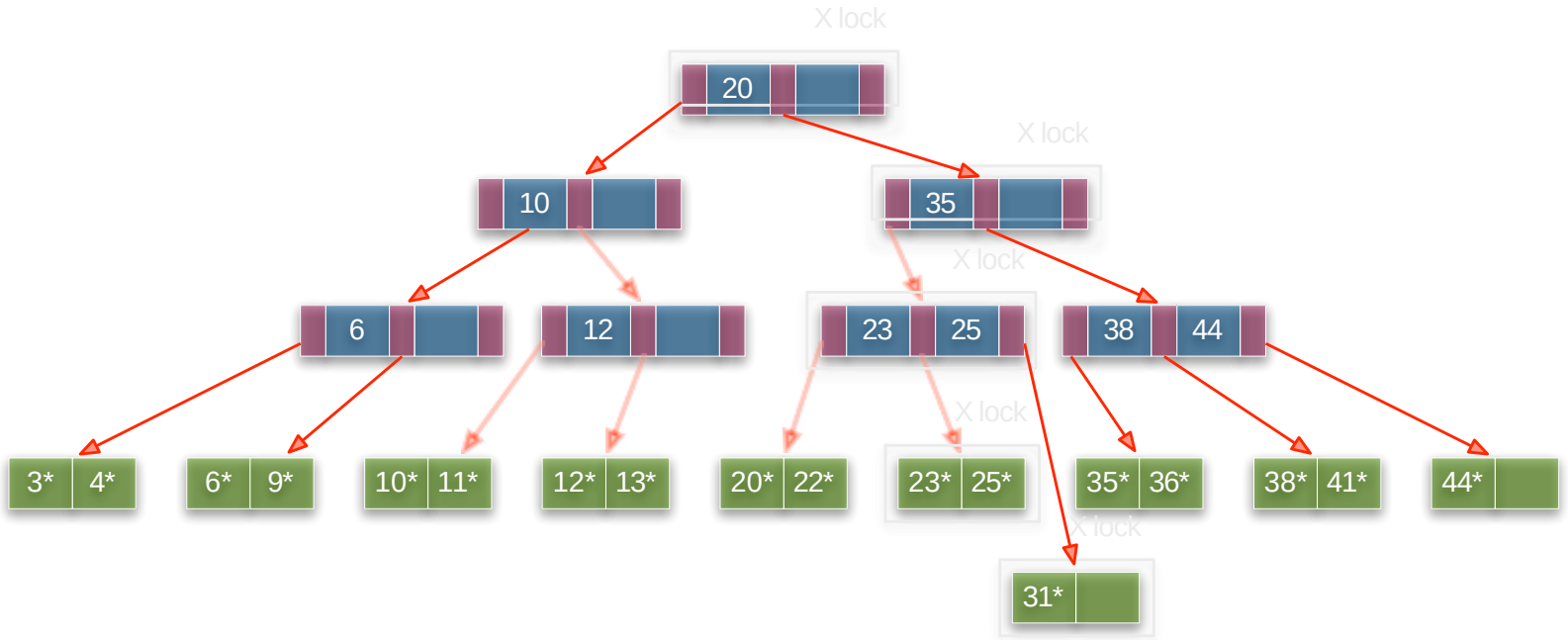
Obtain X-locks while *descending*; *leaf-node is not safe* so *create* a *new* one and *lock it in X-mode*; first release *locks on leaves* and then the rest *top-down*

Example: insert 25*



Obtain X-locks while *descending*; *leaf-node is not safe* so *create* a *new* one and *lock it in X-mode*; first release *locks on leaves* and then the rest *top-down*

Example: insert 25*

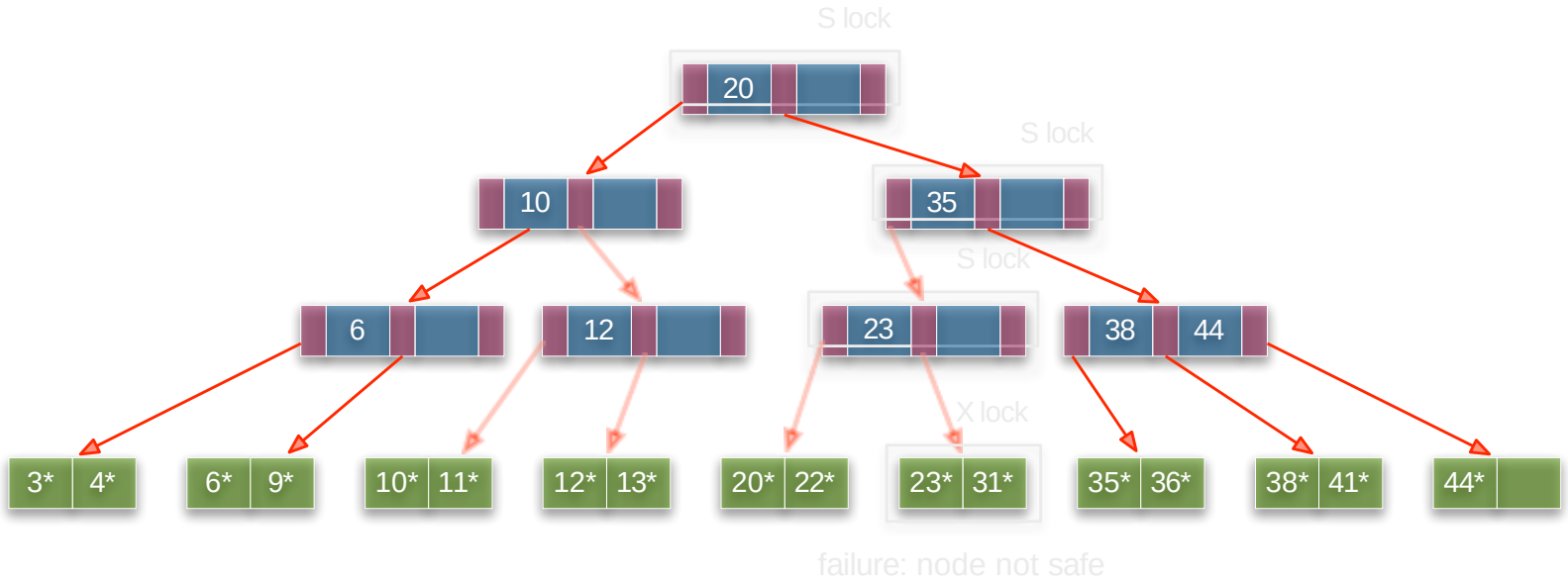


Obtain X-locks while *descending*; *leaf-node is not safe* so *create* a *new* one and *lock it in X-mode*; first release *locks on leaves* and then the rest *top-down*

Optimistic B+tree locking

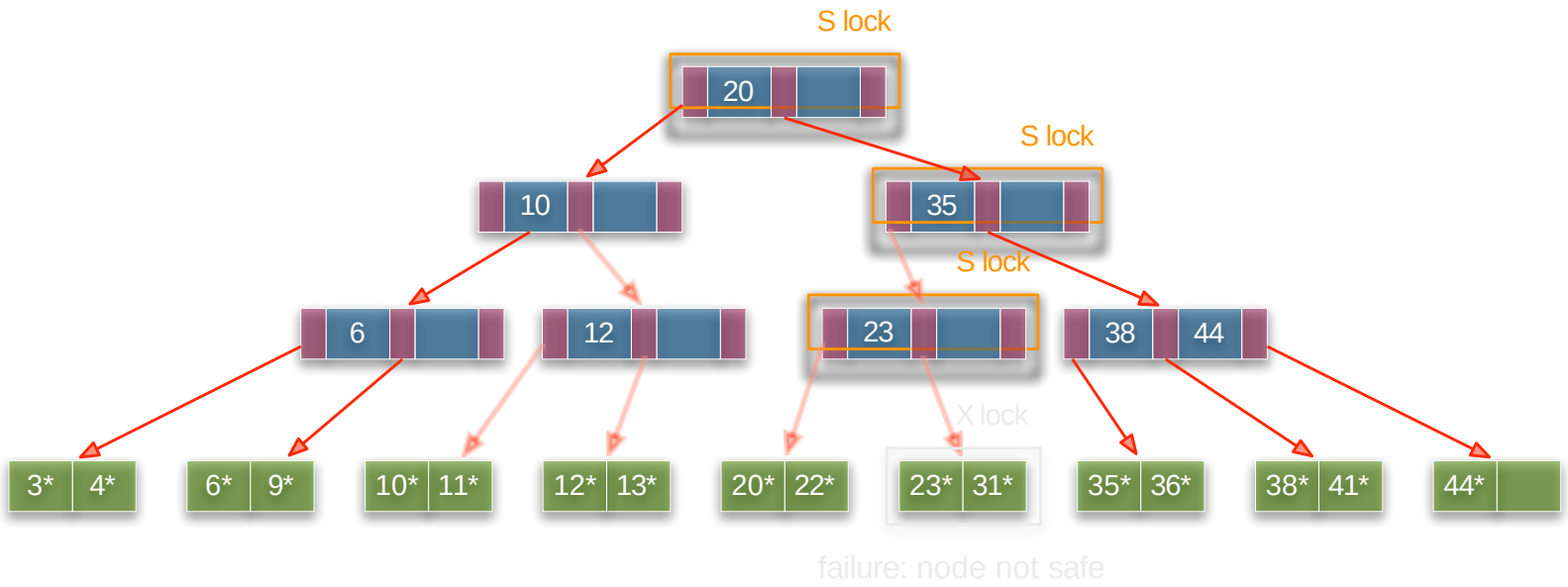
- *Search*: *as before*
- *Insert/delete*: set *locks as* if for *search*, get to the leaf, and *set X lock on the leaf*
  - *If* the *leaf is not safe*, *release* all locks, and *restart* transaction, *using previous insert/delete protocol*
- *"Gambles"* that only *leaf node* will be *modified*; *if not*, *S locks* set on the first pass to leaf are *wasteful*
  - *In practice*, *better* than previous algorithm

Example: insert 25*



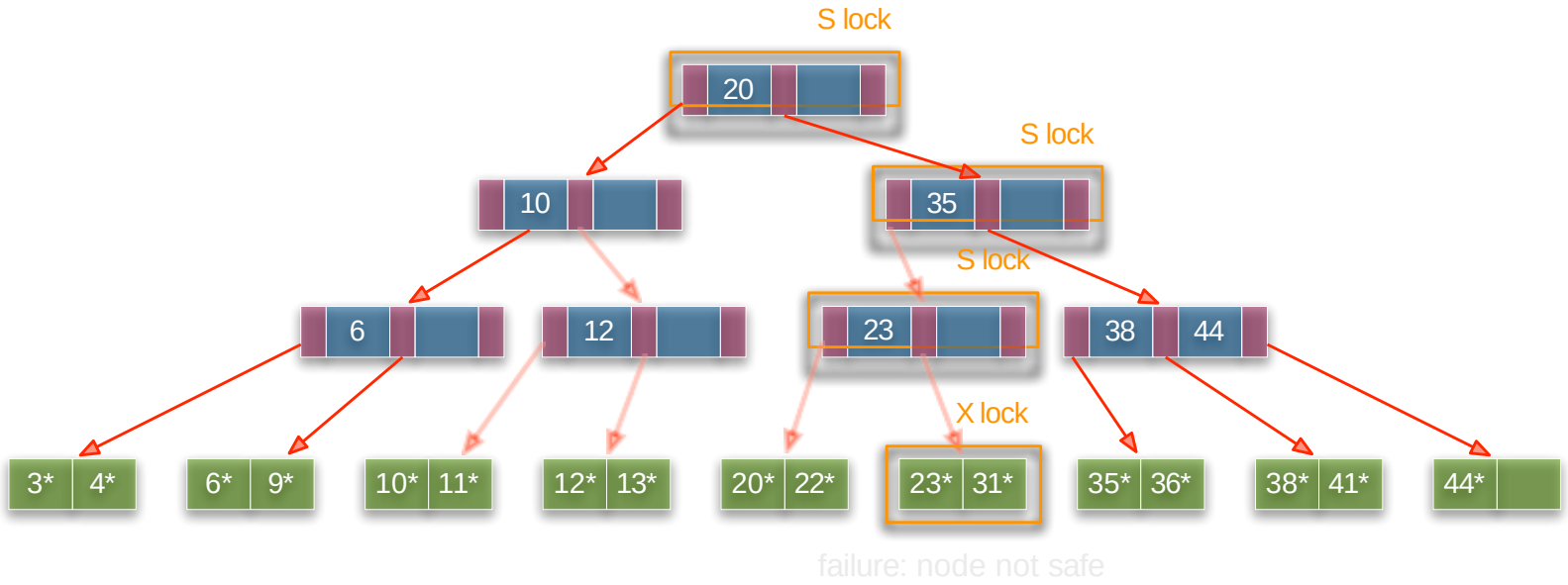
Obtain S-locks while *descending*, and *X-lock* at leaf; the leaf is *not safe*, so *abort*, *release all locks* and *restart* using the *previous algorithm*

Example: insert 25*



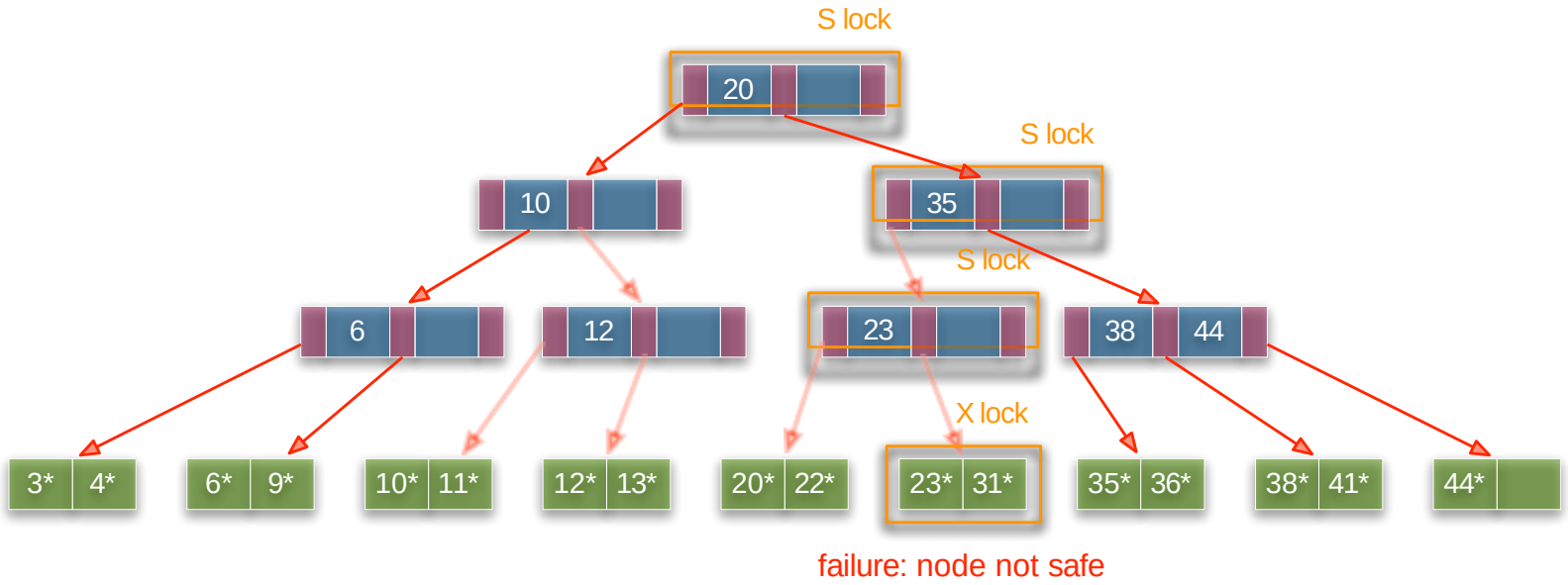
Obtain S-locks while *descending*, and *X-lock* at leaf; the leaf is *not safe*, so *abort*, *release all locks* and *restart* using the *previous algorithm*

Example: insert 25*



Obtain S-locks while *descending*, and *X-lock* at leaf; the leaf is *not safe*, so *abort*, *release all locks* and *restart* using the *previous algorithm*

Example: insert 25*

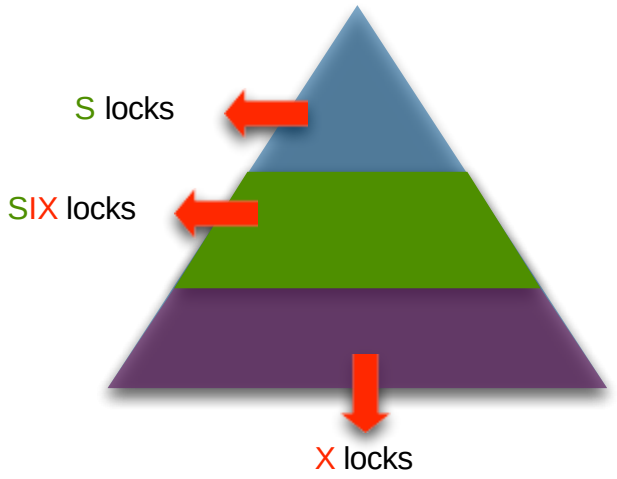


Obtain S-locks while *descending*, and *X-lock* at leaf; the leaf is *not safe*, so *abort*, *release all locks* and *restart* using the *previous algorithm*

Even better algorithm

- *Search*: *as before*
- *Insert/delete*: use *original insert/delete* protocol, but *set IX locks instead of X* locks at all nodes
  - *Once leaf* is *locked*, *convert* all *IX locks to X locks top-down*: *i.e.*, starting from the unsafe node nearest to root
  - *Top-down reduces* chances of *deadlock*
    - Remember, this is *not the same* as *multiple granularity locking*!

Hybrid approach



- The *likelihood* that we will *need* an *X lock decreases* as we *move up* the *tree*
- Set *S locks* at *high levels*, *SIX locks* at *middle levels*, *X locks* at *low levels*

Summary

- Purpose of Concurrency Control
- Two-Phase locking
- Limitations of Concurrency Control
- Timestamp based concurrency control
- Multi-Version Concurrency Control
- Multiple granularity locks
- Index locking

Multiversion Practice

Does multiversion timestamp order ever abort?
Assume, TS(Ti) = i

W0(X)R1(X)W1(X)R2(X)R3(X)W3(X)W2(X)

Under what circumstances will it abort?