Trees: part 2

Petr Kurapov

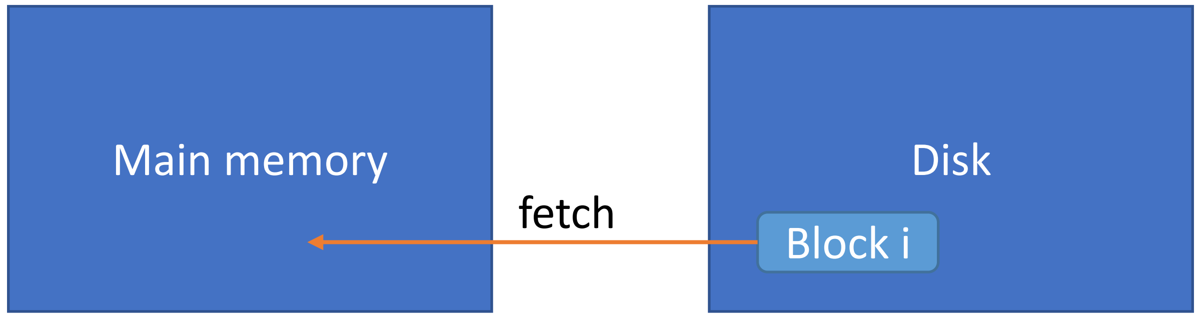
Fall 2024

Agenda

- B-tree
- B+ tree
- Concurrent tree access

Motivation

- Data is organized in blocks, “low” number of pages present in the main memory – not able to store all the data in it



Id	Col1	Col2	Col3	…
0				
1				
2				
3				

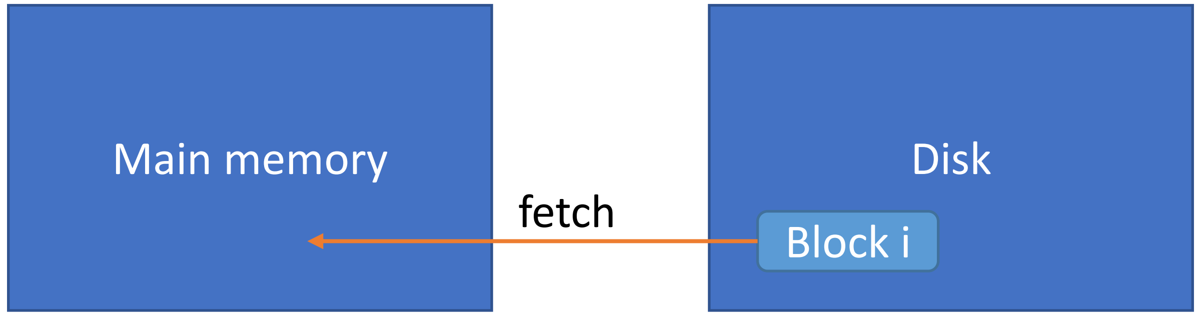
Latency Comparison Numbers (~2012)

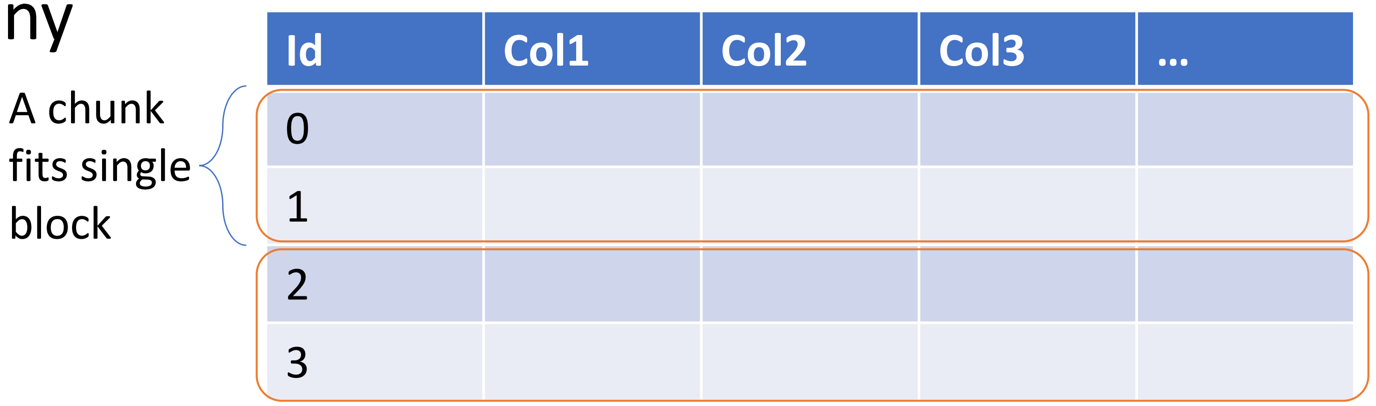
L1 cache reference	0.5 ns			
Branch mispredict	5 ns			
L2 cache reference	7 ns			14x L1 cache
Mutex lock/unlock	25 ns			
Main memory reference	100 ns			20x L2 cache, 200x L1 cache
Compress 1K bytes with Zippy	3,000 ns	3 us		
Send 1K bytes over 1 Gbps network	10,000 ns	10 us		
Read 4K randomly from SSD*	150,000 ns	150 us		~1GB/sec SSD
Read 1 MB sequentially from memory	250,000 ns	250 us		
Round trip within same datacenter	500,000 ns	500 us		
Read 1 MB sequentially from SSD*	1,000,000 ns	1,000 us	1 ms	~1GB/sec SSD, 4X memory
Disk seek	10,000,000 ns	10,000 us	10 ms	20x datacenter roundtrip
Read 1 MB sequentially from disk	20,000,000 ns	20,000 us	20 ms	80x memory, 20X SSD
Send packet CA->Netherlands->CA	150,000,000 ns	150,000 us	150 ms	

https://gist.github.com/jboner/2841832

Motivation

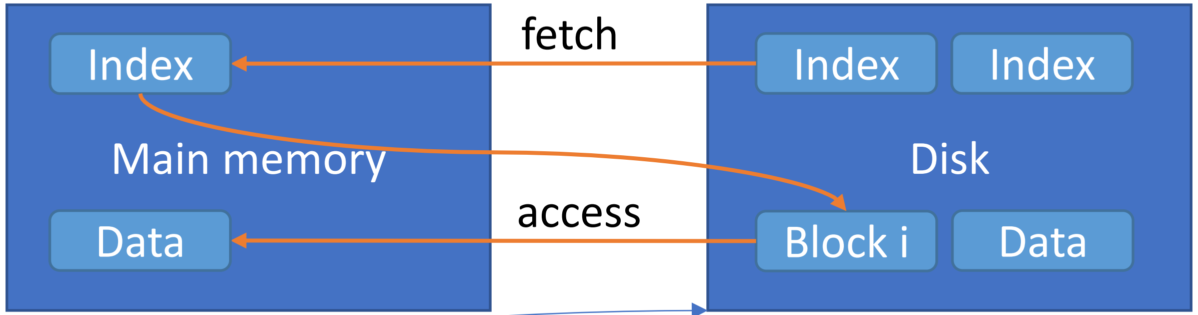
- Data is organized in blocks, “low” number of pages present in the main memory – not able to store all the data in it
- i.e. row-store
- Sequential table scan – many accesses (# of chunks)





Motivation

- i.e. row-store
- Sequential table scan – many accesses (# of chunks)
- Use index to minimize disk accesses



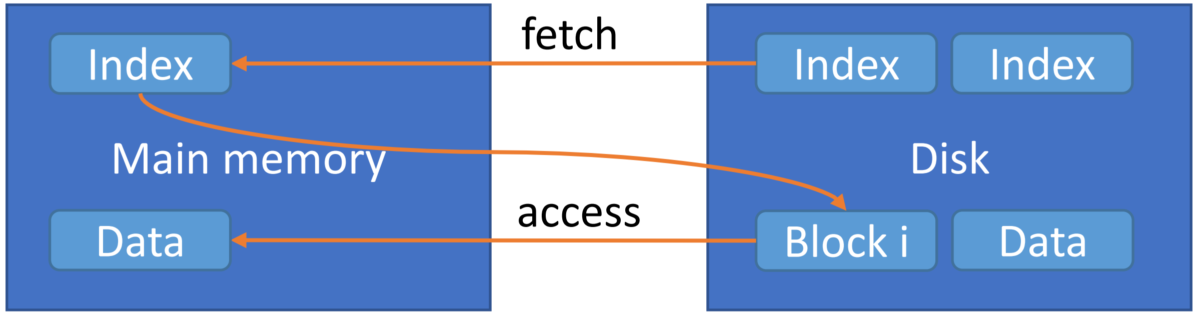
Id	BlockId
0	X
1	Y

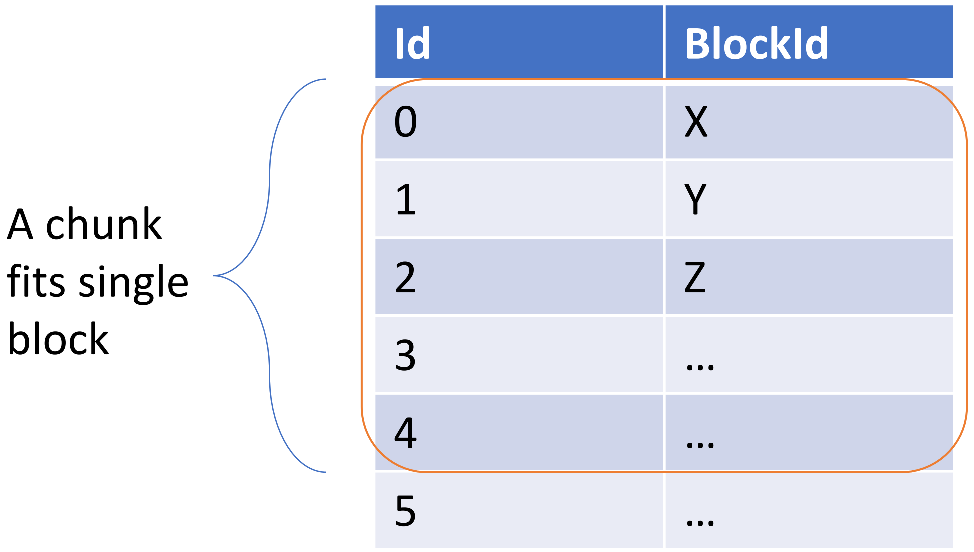
A chunk fits single block

Id	Col1	Col2	Col3	…
0				
1				
2				
3				

Motivation

- i.e. row-store
- Sequential table scan – many accesses (# of chunks)
- Use index to minimize disk accesses
- As index grows, fetching it becomes expensive
- Tree indexing!

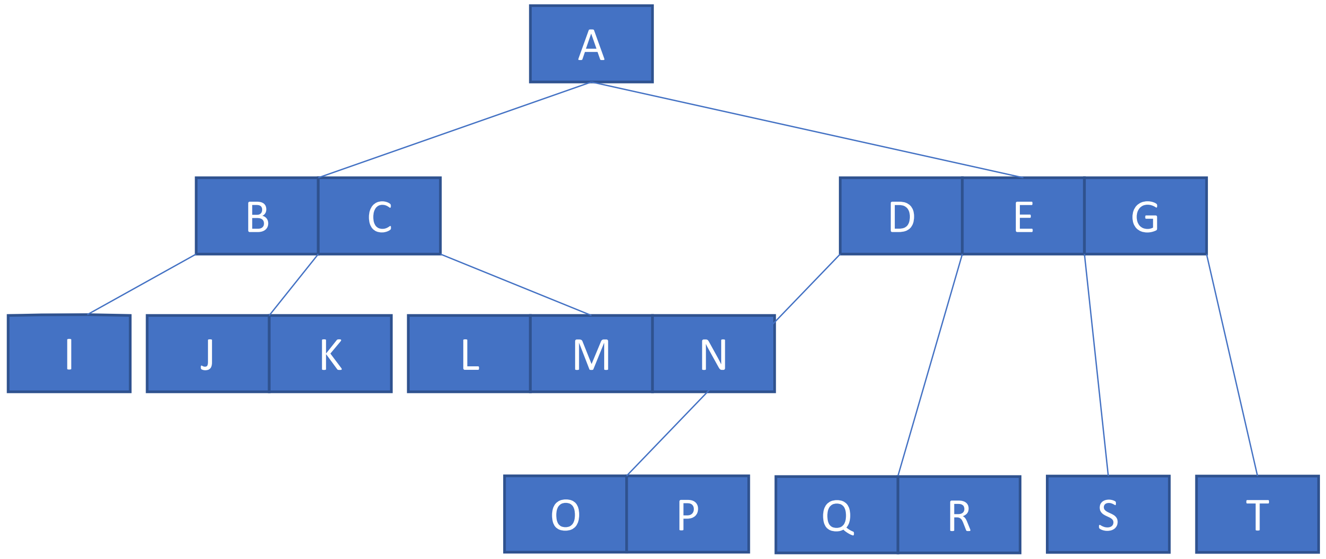



A chunk fits single block

Id	BlockId
0	X
1	Y
2	Z
3	…
4	…
5	…

B(+)-tree

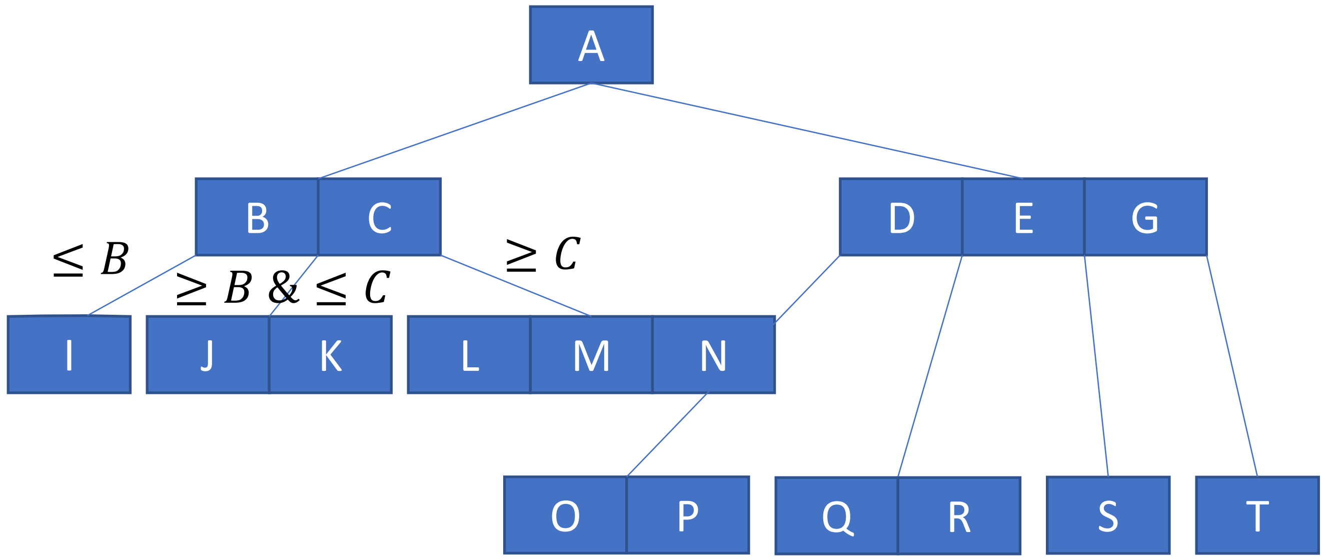
- Self-balancing m-way search tree
- Perfectly balanced: same height on every path from root
- Every node, except for root are at least half full
- Keys are sorted in a node



D.Comer, The ubiquitous B-Tree. http://carlosproal.com/ir/papers/p121-comer.pdf

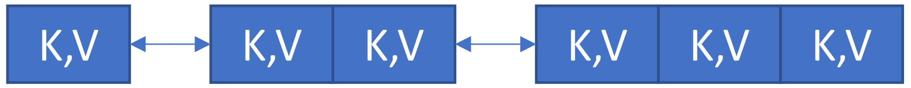
B(+)-tree

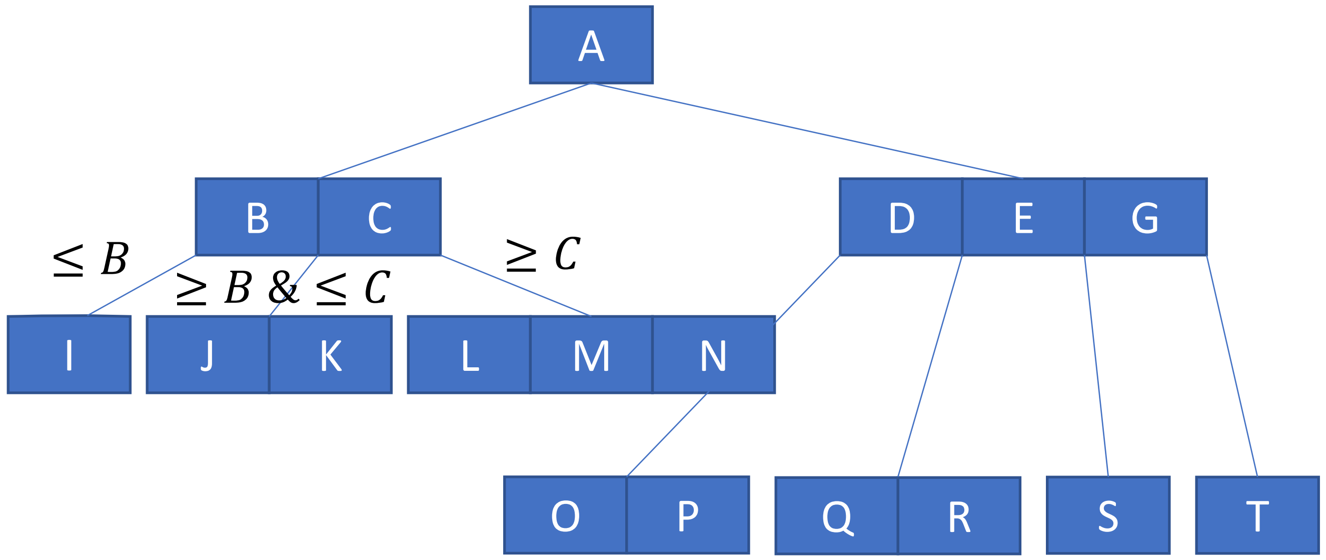
- Self-balancing m-way search tree
- Perfectly balanced: same height on every path from root
- Every node, except for root are at least half full
- Keys are sorted in a node



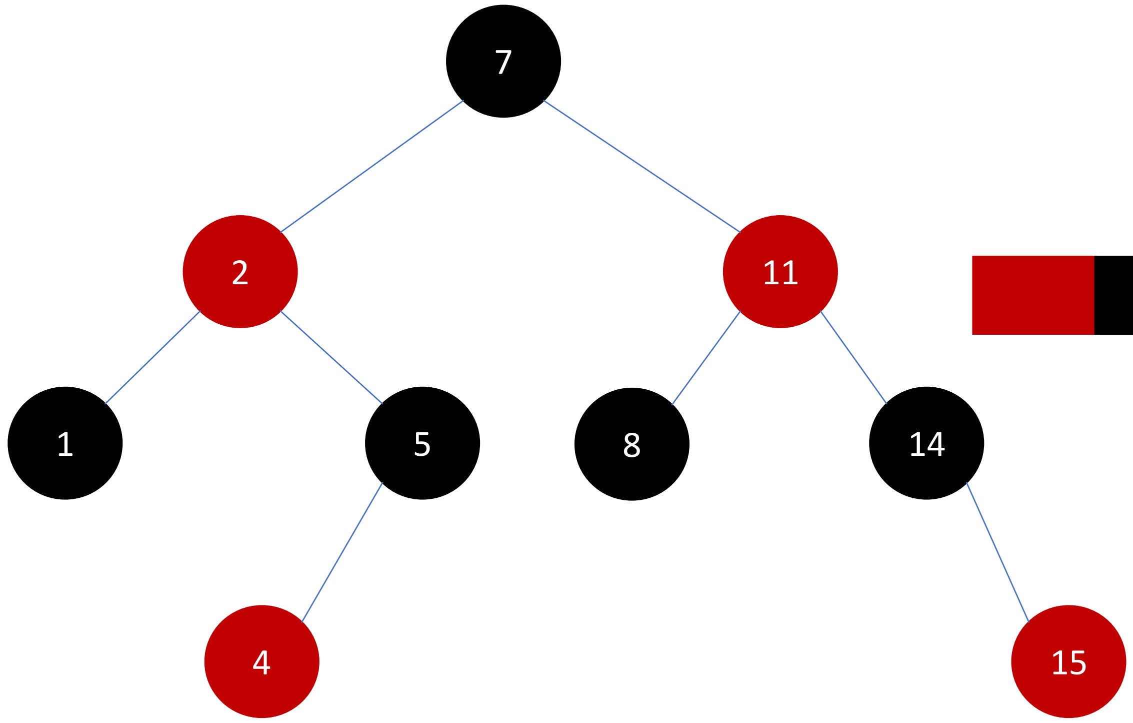
B(+)-tree

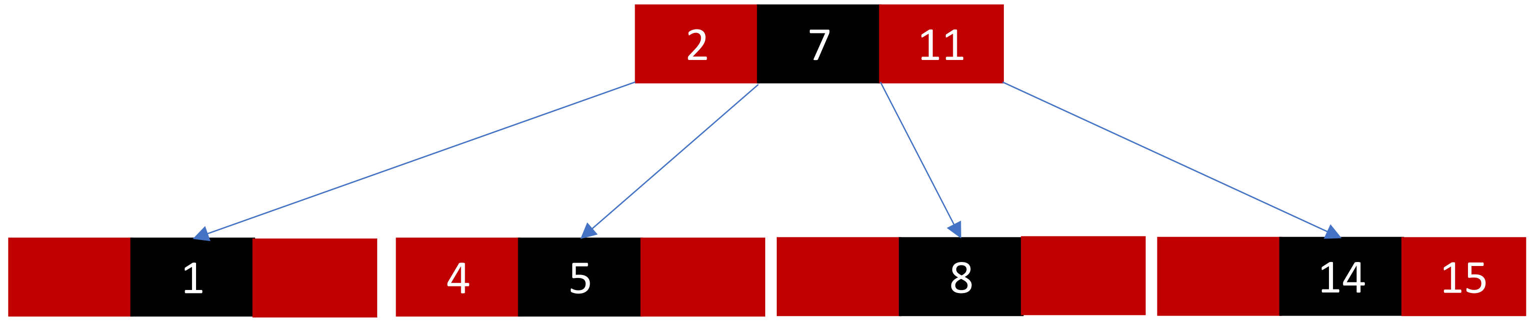
- Leaf nodes for B+ tree contain pointers/tuples they index
- Connected into linked list





Relation to red-black tree





B(+)-tree

B-tree

- Less space required
- Updates are more expensive for multiple thread access: rebalancing requires more latches

B+ tree

- All values are in leaf nodes
- Keys are duplicated
- Can preserve key in non-leaf node upon key removal

B(+)-tree: Insertion

- Find appropriate leaf node for the new element
- If enough space – insert, preserving order
- If not, split the node in two and move middle element to parent

B tree grows “upward”

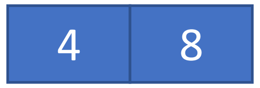
B(+)-tree: Insertion

- Find appropriate leaf node for the new element
- If enough space – insert, preserving order
- If not, split the node in two and move middle element to parent

B tree grows “upward”

B(+)-tree: Insertion

- Find appropriate leaf node for the new element
- If enough space – insert, preserving order
- If not, split the node in two and move middle element to parent



B tree grows “upward”

B(+)-tree: Insertion

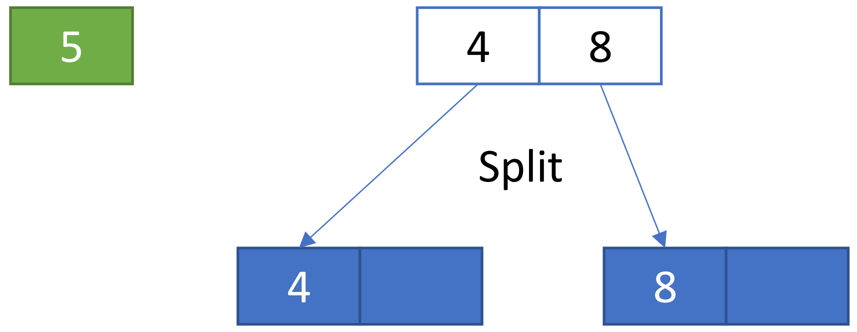
- Find appropriate leaf node for the new element
- If enough space – insert, preserving order
- If not, split the node in two and move middle element to parent

B tree grows “upward”

B(+)-tree: Insertion

- Find appropriate leaf node for the new element
- If enough space – insert, preserving order
- If not, split the node in two and move middle element to parent

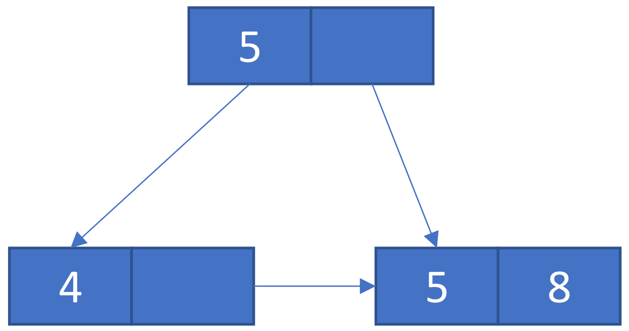
B tree grows "upward"



B(+)-tree: Insertion

- Find appropriate leaf node for the new element
- If enough space – insert, preserving order
- If not, split the node in two and move middle element to parent

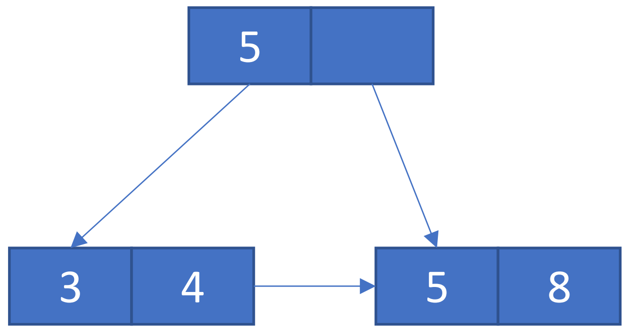
B tree grows “upward”



B(+)-tree: Insertion

- Find appropriate leaf node for the new element
- If enough space – insert, preserving order
- If not, split the node in two and move middle element to parent

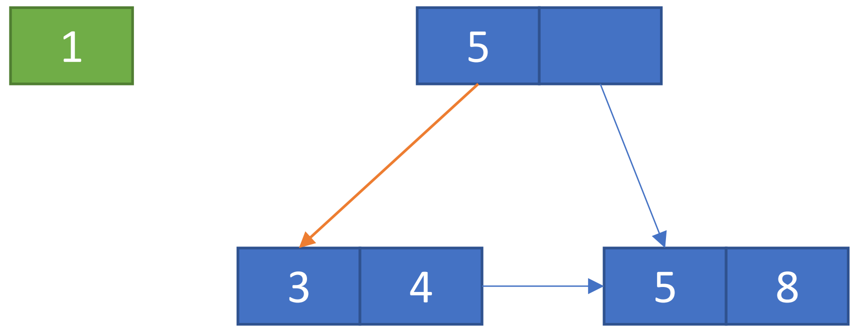
B tree grows “upward”



B(+)-tree: Insertion

- Find appropriate leaf node for the new element
- If enough space – insert, preserving order
- If not, split the node in two and move middle element to parent

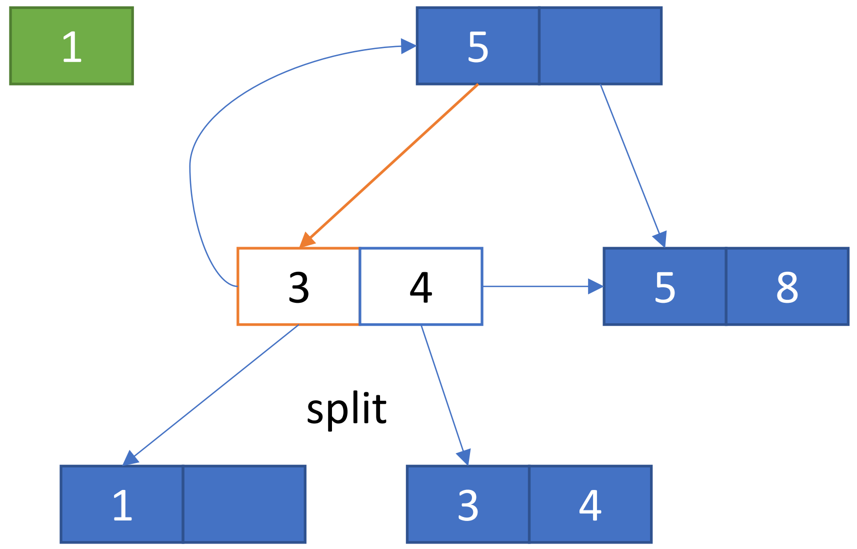
B tree grows “upward”



B(+)-tree: Insertion

- Find appropriate leaf node for the new element
- If enough space – insert, preserving order
- If not, split the node in two and move middle element to parent

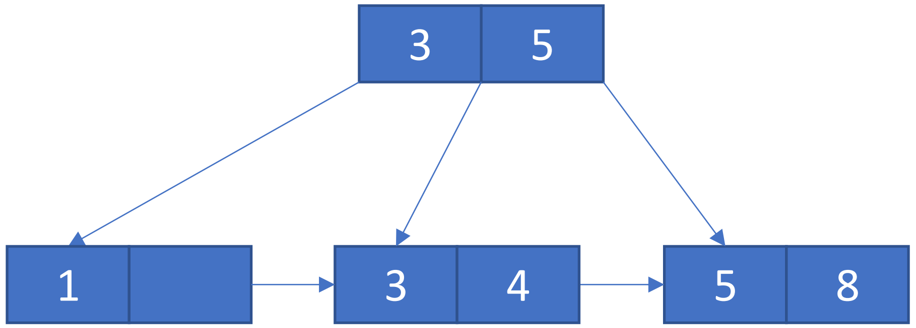
B tree grows “upward”



B(+)-tree: Insertion

- Find appropriate leaf node for the new element
- If enough space – insert, preserving order
- If not, split the node in two and move middle element to parent

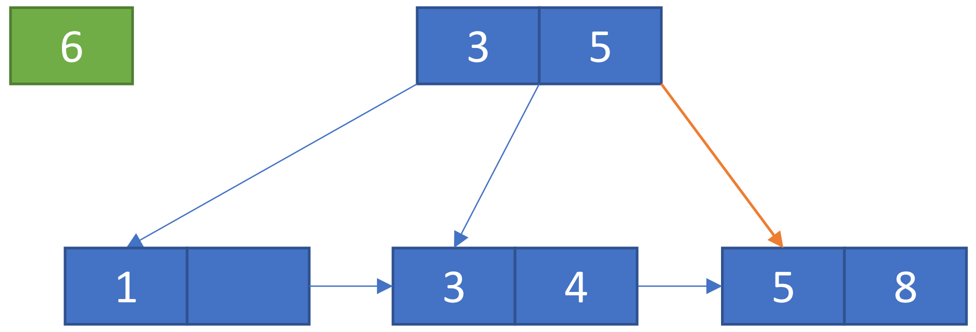
B tree grows “upward”



B(+)-tree: Insertion

- Find appropriate leaf node for the new element
- If enough space – insert, preserving order
- If not, split the node in two and move middle element to parent

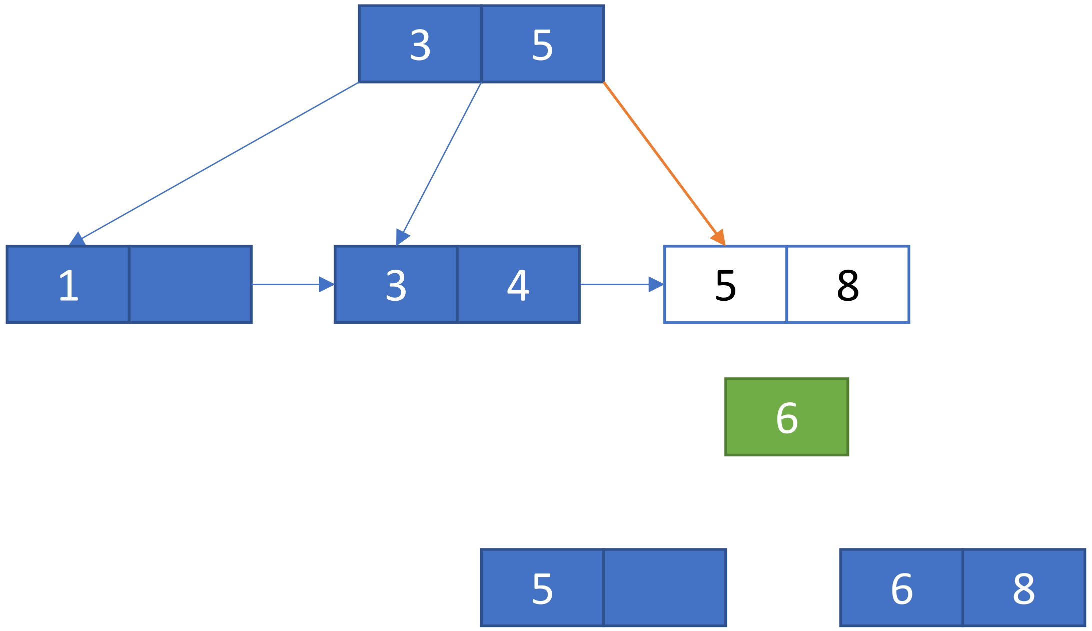
B tree grows “upward”



B(+)-tree: Insertion

- Find appropriate leaf node for the new element
- If enough space – insert, preserving order
- If not, split the node in two and move middle element to parent

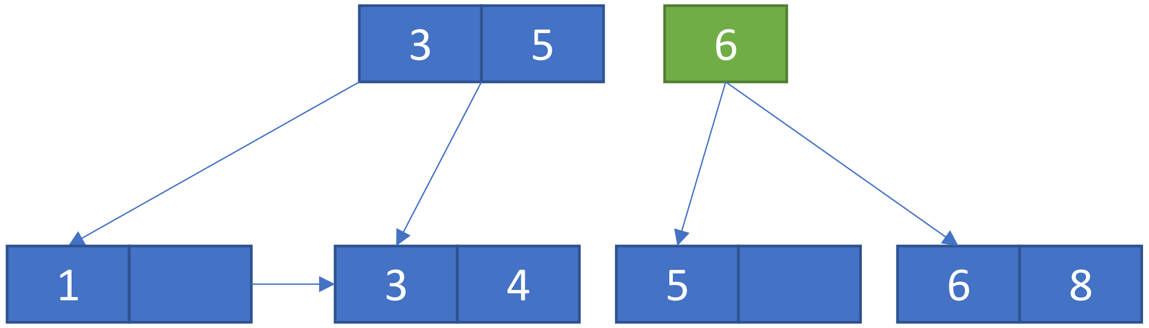
B tree grows “upward”



B(+)-tree: Insertion

- Find appropriate leaf node for the new element
- If enough space – insert, preserving order
- If not, split the node in two and move middle element to parent

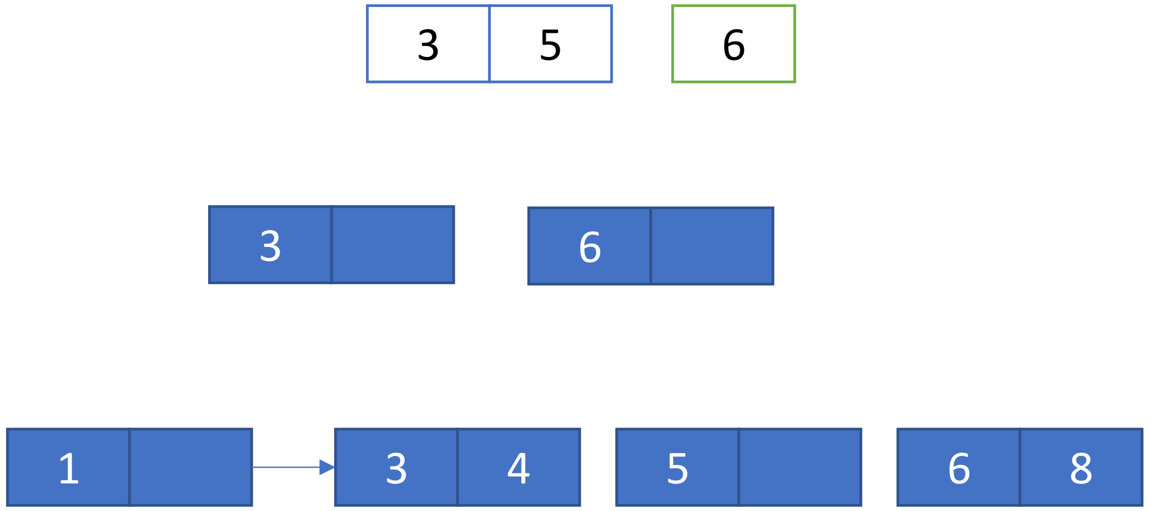
B tree grows “upward”



B(+)-tree: Insertion

- Find appropriate leaf node for the new element
- If enough space – insert, preserving order
- If not, split the node in two and move middle element to parent

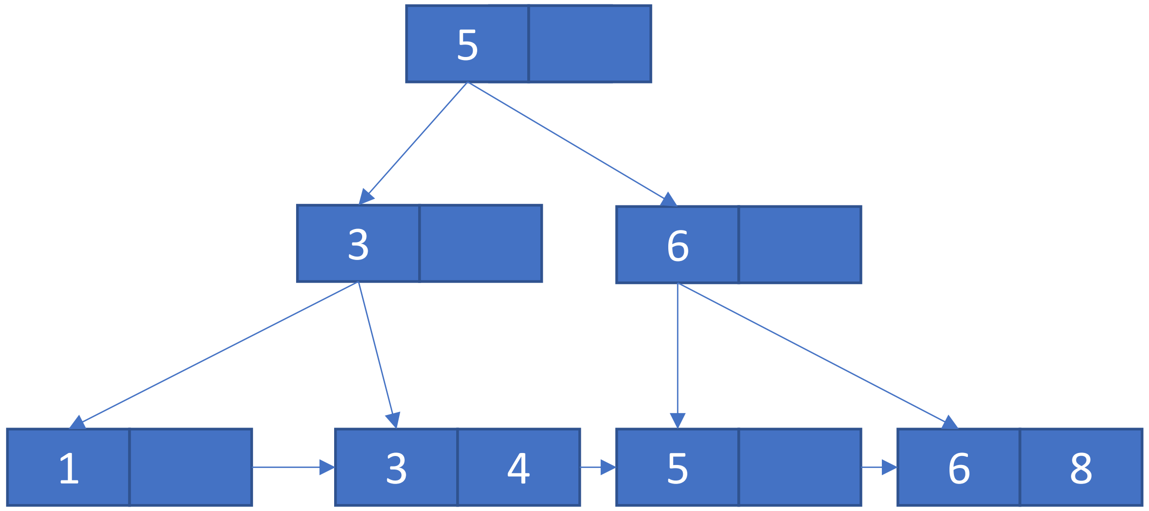
B tree grows “upward”



B(+)-tree: Insertion

- Find appropriate leaf node for the new element
- If enough space – insert, preserving order
- If not, split the node in two and move middle element to parent

B tree grows “upward”

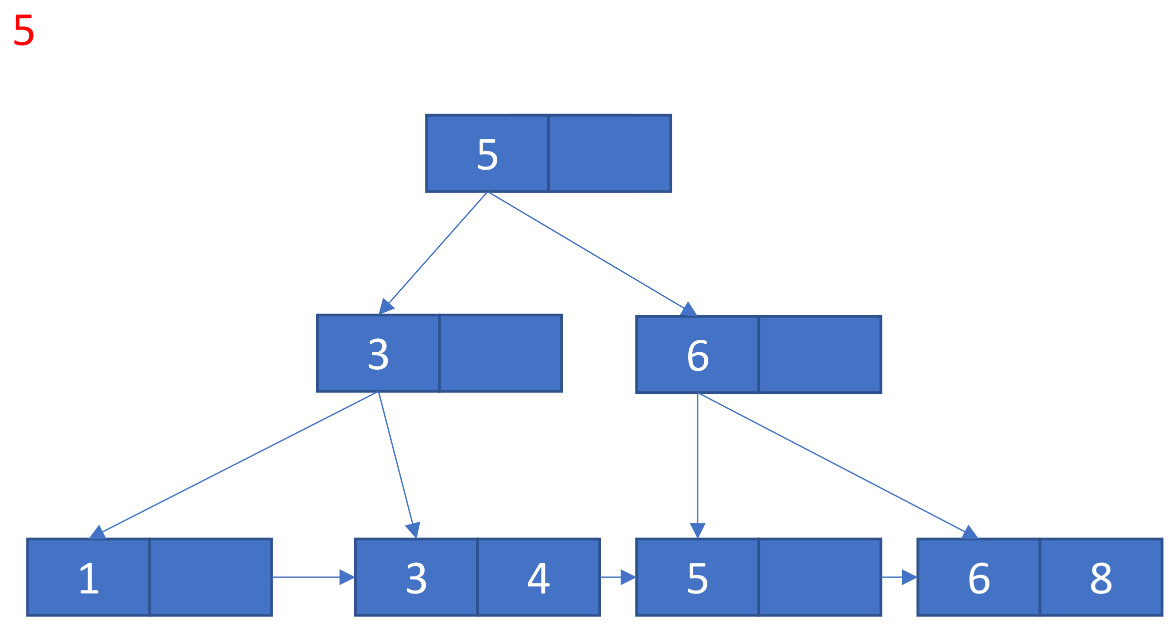


B(+)-tree: Deletion

- Find leaf node
- If deletion doesn’t make node less than half full – done
- If not, redistribute
- If unable, merge

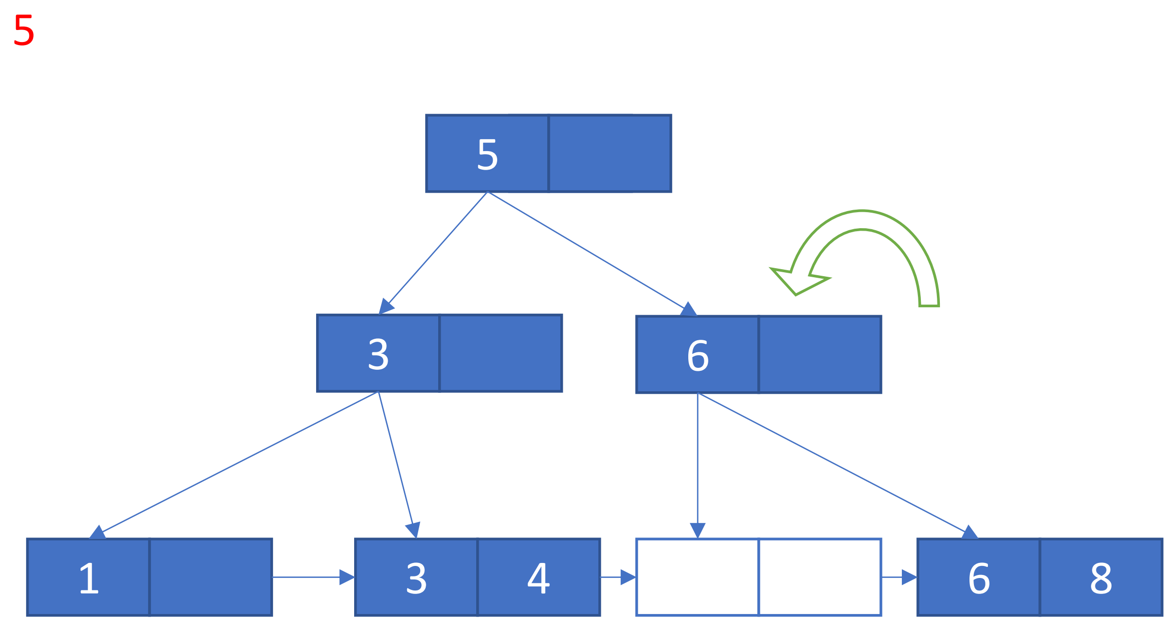
B(+)-tree: Deletion

- Find leaf node
- If deletion doesn't make node less than half full – done
- If not, redistribute
- If unable, merge



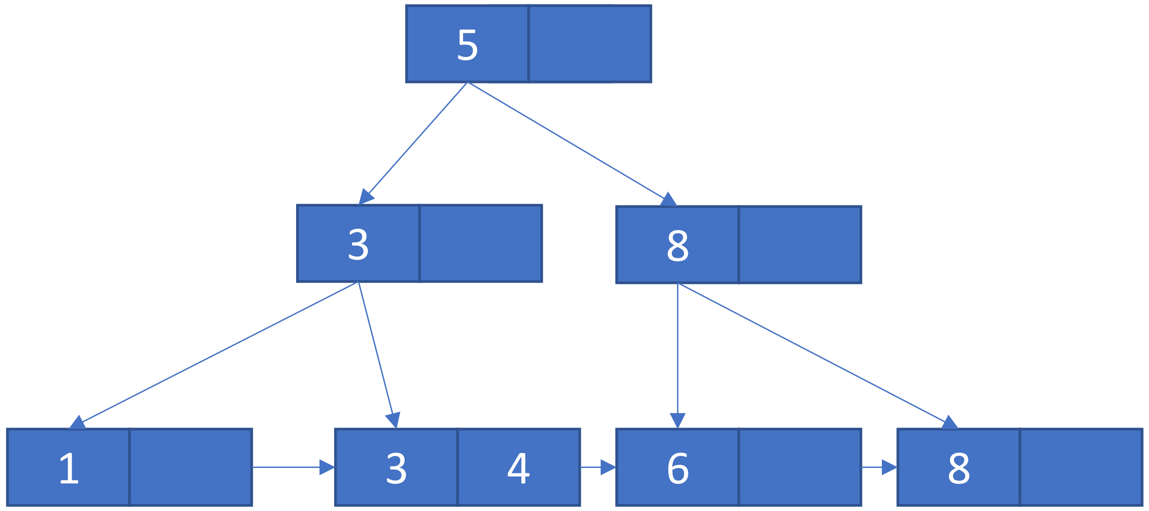
B(+)-tree: Deletion

- Find leaf node
- If deletion doesn't make node less than half full – done
- If not, redistribute
- If unable, merge



B(+)-tree: Deletion

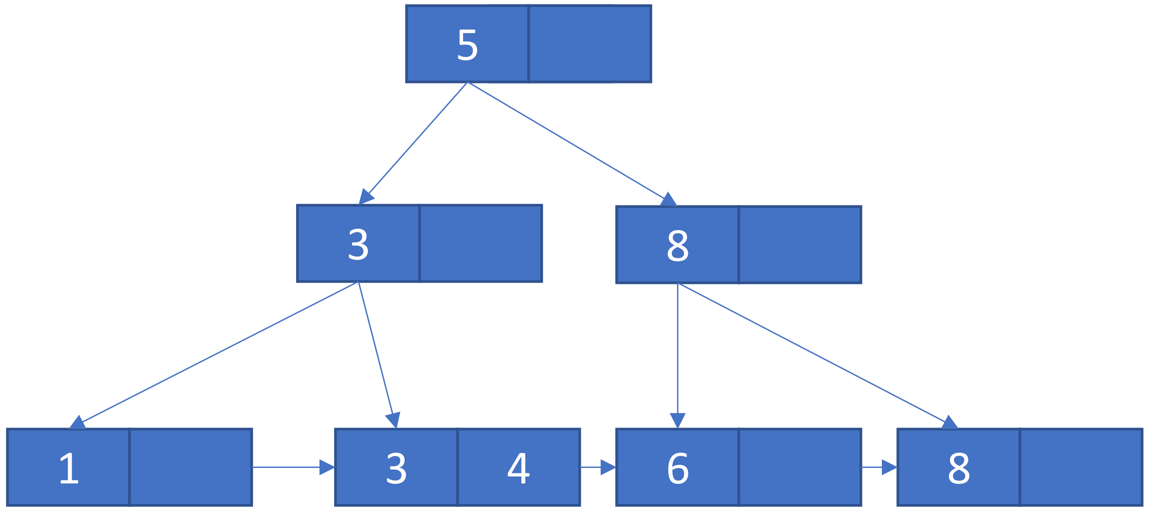
- Find leaf node
- If deletion doesn't make node less than half full – done
- If not, redistribute
- If unable, merge



B(+)-tree: Deletion

- Find leaf node
- If deletion doesn't make node less than half full – done
- If not, redistribute
- If unable, merge

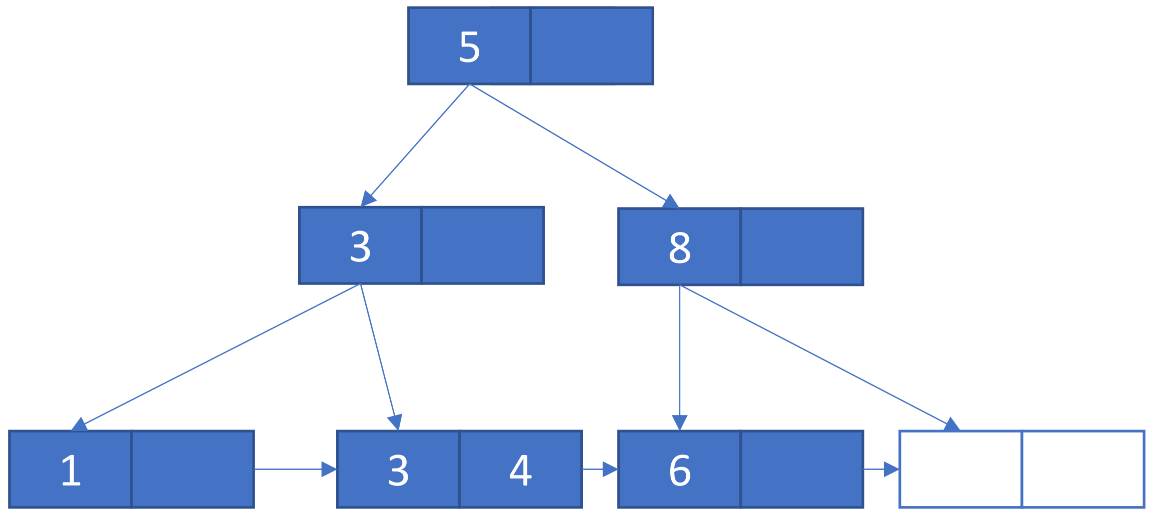
8



B(+)-tree: Deletion

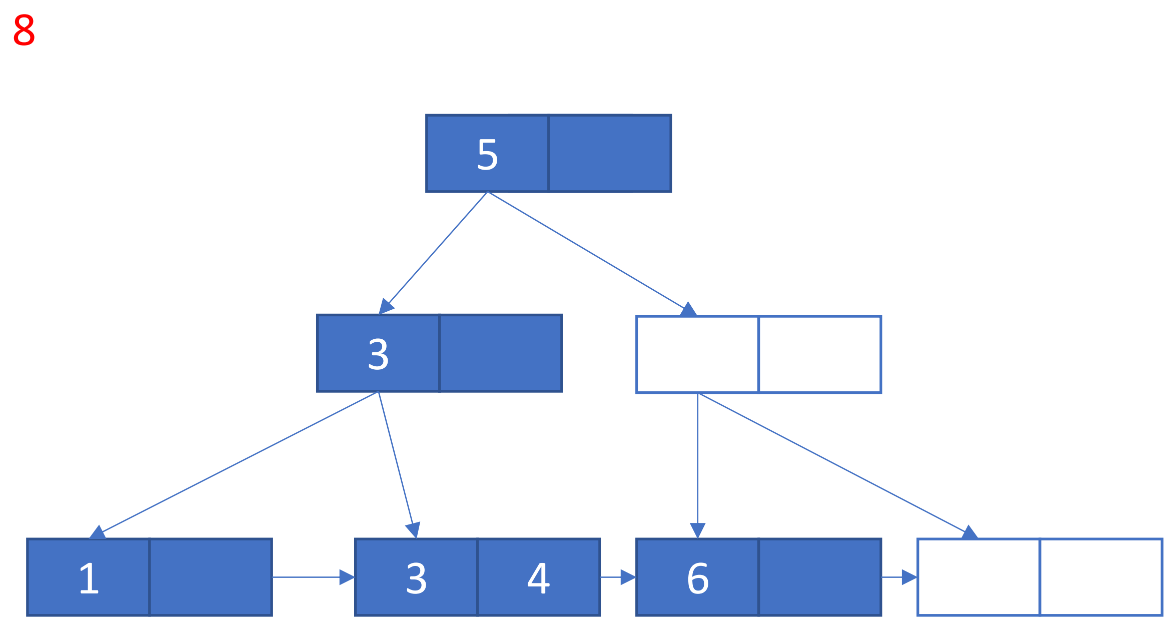
- Find leaf node
- If deletion doesn't make node less than half full – done
- If not, redistribute
- If unable, merge

8



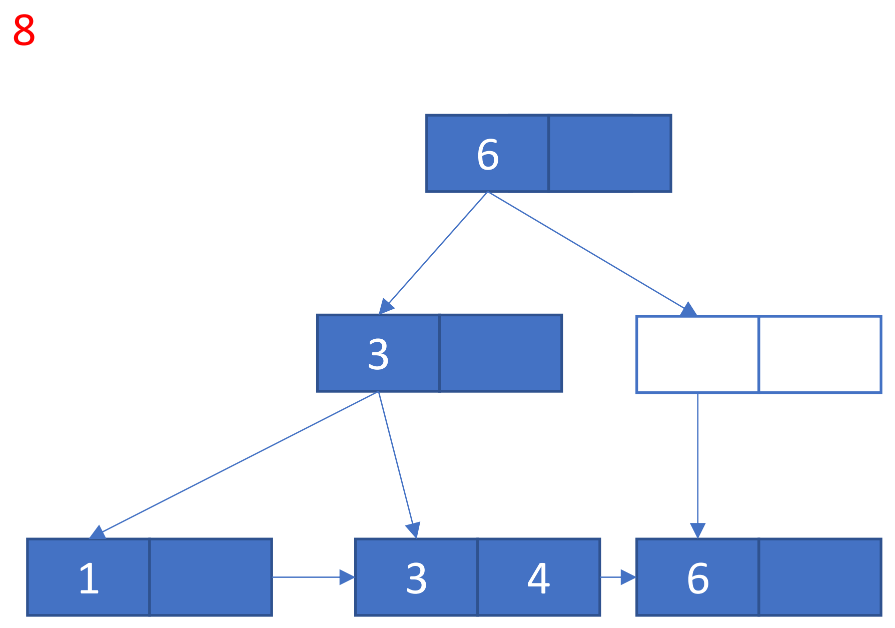
B(+)-tree: Deletion

- Find leaf node
- If deletion doesn't make node less than half full – done
- If not, redistribute
- If unable, merge



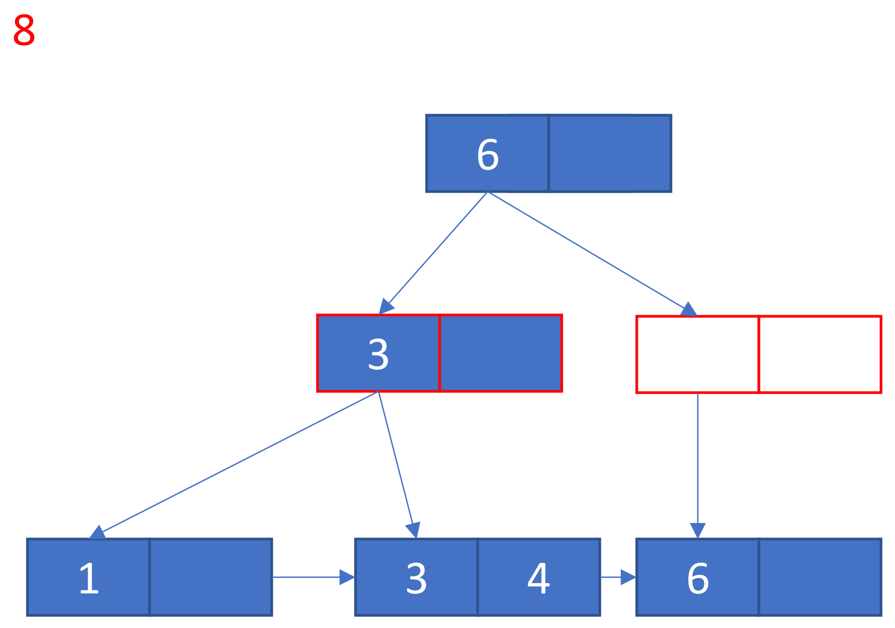
B(+)-tree: Deletion

- Find leaf node
- If deletion doesn't make node less than half full – done
- If not, redistribute
- If unable, merge



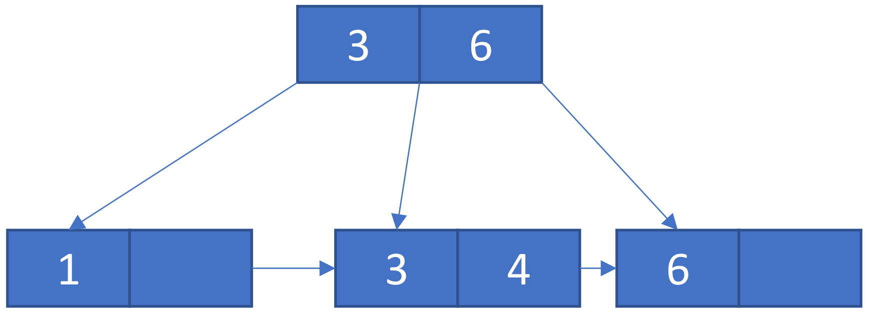
B(+)-tree: Deletion

- Find leaf node
- If deletion doesn't make node less than half full – done
- If not, redistribute
- If unable, merge



B(+)-tree: Deletion

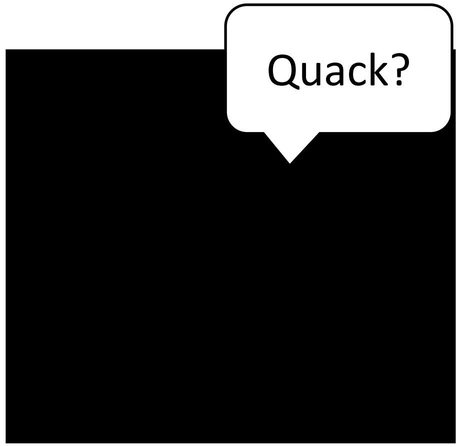
- Find leaf node
- If deletion doesn't make node less than half full – done
- If not, redistribute
- If unable, merge



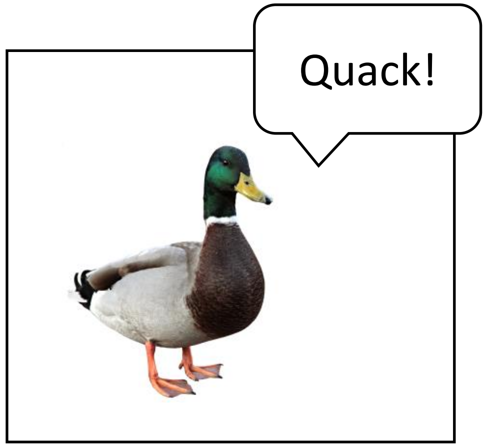
Concurrency control

- A *latch* protects a critical section of an in-memory data structure operation
- Read latch would allow multiple threads to read the value concurrently
- Write latch allows exclusive access only

- Logical correctness – quacks like a duck
- Physical correctness – the duck is quaking



Logical correctness



Physical correctness

Concurrency control

- A *latch* protects a critical section of an in-memory data structure operation
- Read latch would allow multiple threads to read the value concurrently
- Write latch allows exclusive access only

Implementation:

- Blocking mutex
  - std::mutex
  - Fast user-space locking (futex) – compare-and-swap + OS fallback

Concurrency control

- A *latch* protects a critical section of an in-memory data structure operation
- Read latch would allow multiple threads to read the value concurrently
- Write latch allows exclusive access only

Implementation:

- Blocking mutex
  - std::mutex
  - Fast user-space locking (futex)
- Spinlock
  - std::atomic

Concurrency control

- A *latch* protects a critical section of an in-memory data structure operation
- Read latch would allow multiple threads to read the value concurrently
- Write latch allows exclusive access only

Implementation:

- Blocking mutex
  - std::mutex
  - Fast user-space locking (futex)
- Spinlock
  - std::atomic
- Read-write latch
  - Allow multiple readers storing a # of accessors
  - Policy based

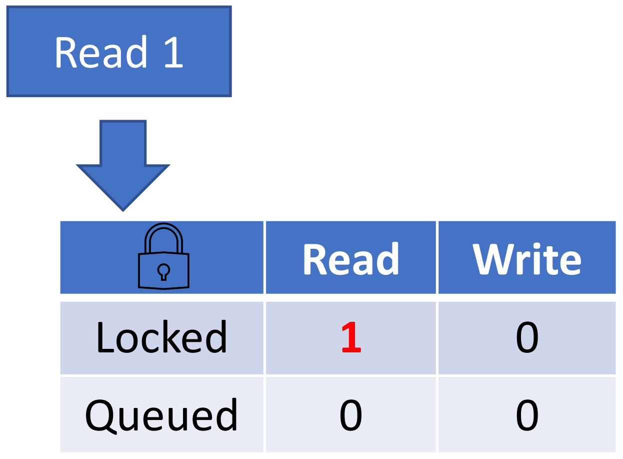
Concurrency control

🔒	Read	Write
Locked	0	0
Queued	0	0

Implementation:

- Blocking mutex
  - std::mutex
  - Fast user-space locking (futex)
- Spinlock
  - std::atomic
- Read-write latch
  - Allow multiple readers storing a # of accessors
  - Policy based

Concurrency control

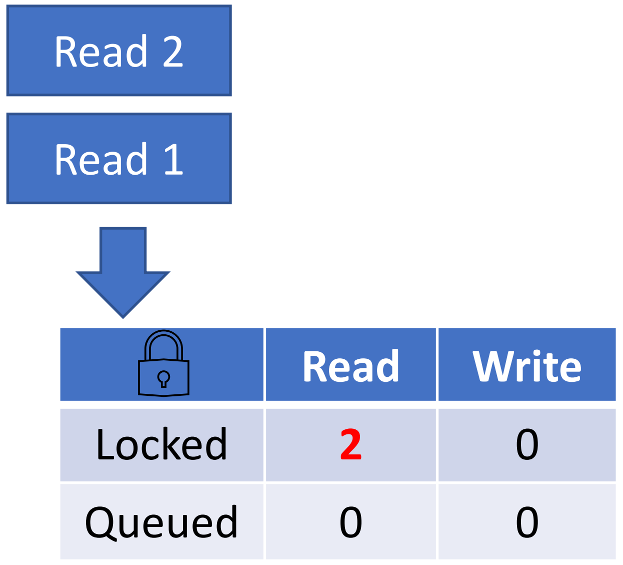


🔒	Read	Write
Locked	1	0
Queued	0	0

Implementation:

- Blocking mutex
  - std::mutex
  - Fast user-space locking (futex)
- Spinlock
  - std::atomic
- Read-write latch
  - Allow multiple readers storing a # of accessors
  - Policy based

Concurrency control

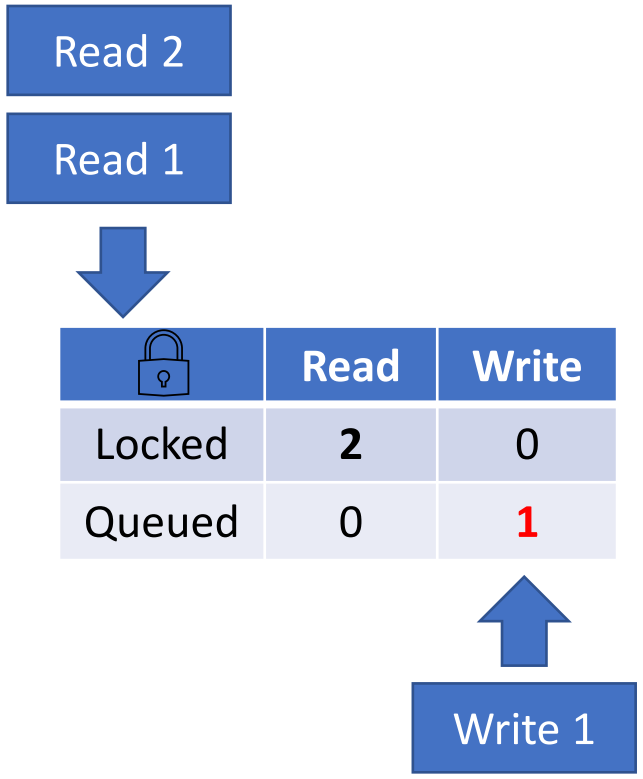


🔒	Read	Write
Locked	2	0
Queued	0	0

Implementation:

- Blocking mutex
  - std::mutex
  - Fast user-space locking (futex)
- Spinlock
  - std::atomic
- Read-write latch
  - Allow multiple readers storing a # of accessors
  - Policy based

Concurrency control

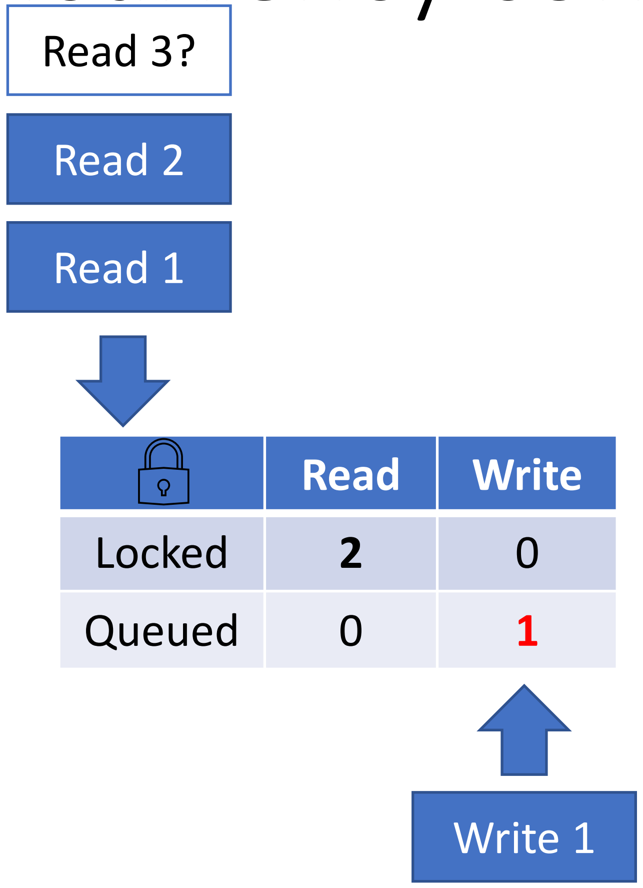


🔒	Read	Write
Locked	**2**	0
Queued	0	**1**

Implementation:

- Blocking mutex
  - std::mutex
  - Fast user-space locking (futex)
- Spinlock
  - std::atomic
- Read-write latch
  - Allow multiple readers storing a # of accessors
  - Policy based

Concurrency control



🔒	Read	Write
Locked	**2**	0
Queued	0	**1**

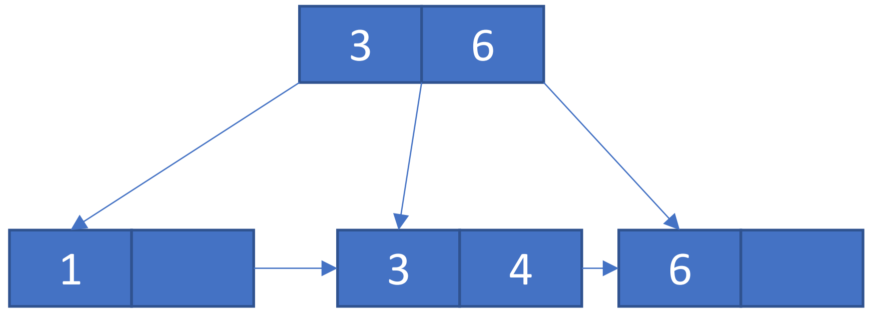
Implementation:

- Blocking mutex
  - std::mutex
  - Fast user-space locking (futex)
- Spinlock
  - std::atomic
- Read-write latch
  - Allow multiple readers storing a # of accessors
  - Policy based

B+ tree: multithreading

- Allow multiple threads to do reads and writes simultaneously
- Potential issues:
  - Concurrent writes on the same object
  - Read on split/merged data

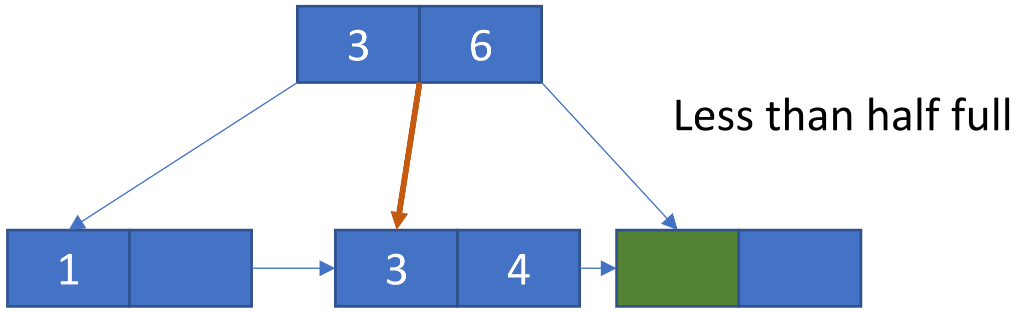
- Thread 1: delete 6
- Thread 2: find 4



B+ tree: multithreading

- Allow multiple threads to do reads and writes simultaneously
- Potential issues:
  - Concurrent writes on the same object
  - Read on split/merged data

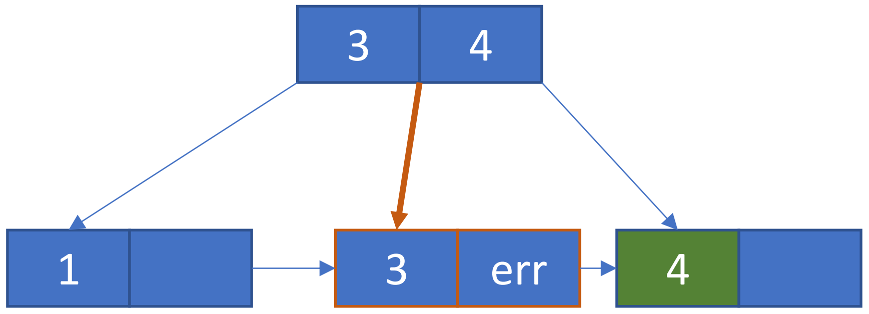
- Thread 1: delete 6 (stalled)
- Thread 2: find 4



B+ tree: multithreading

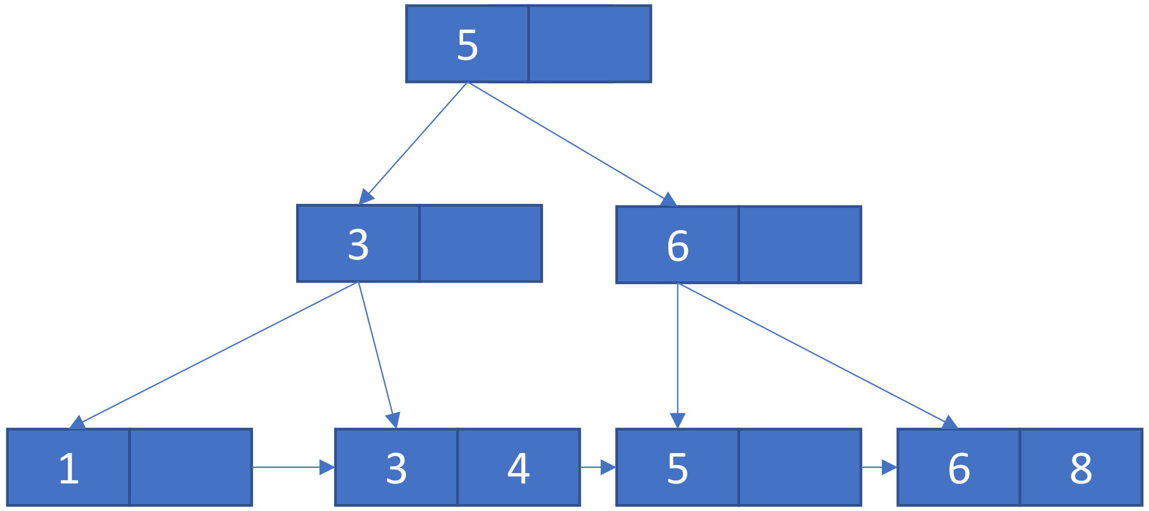
- Allow multiple threads to do reads and writes simultaneously
- Potential issues:
  - Concurrent writes on the same object
  - Read on split/merged data

- Thread 1: delete 6
- Thread 2: find 4



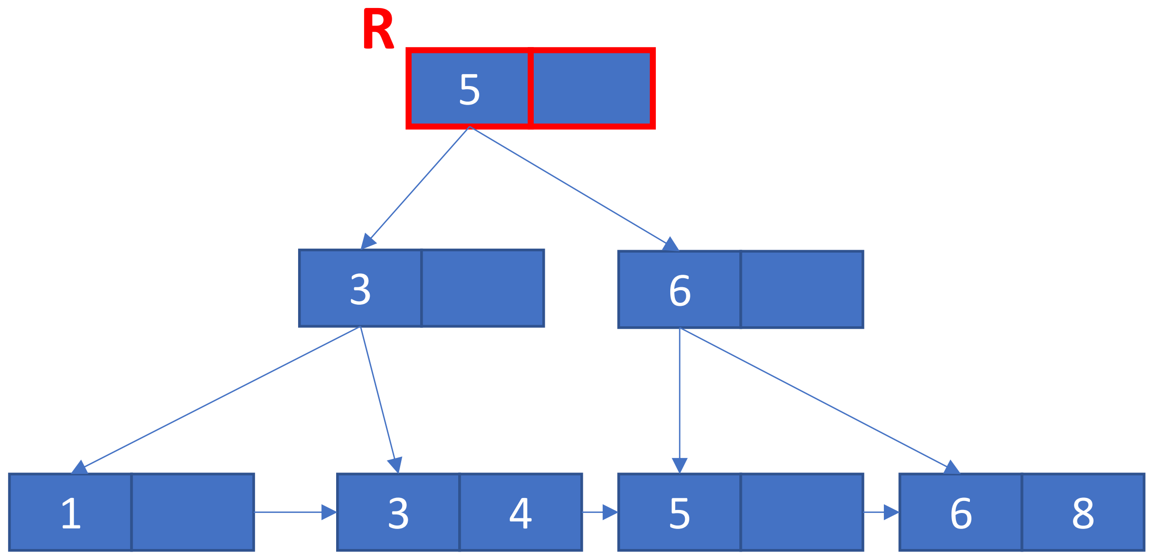
B+ tree: multithreading

- Latch coupling
  - Acquire lock for parent
  - Acquire lock for child
  - Release parent when able to



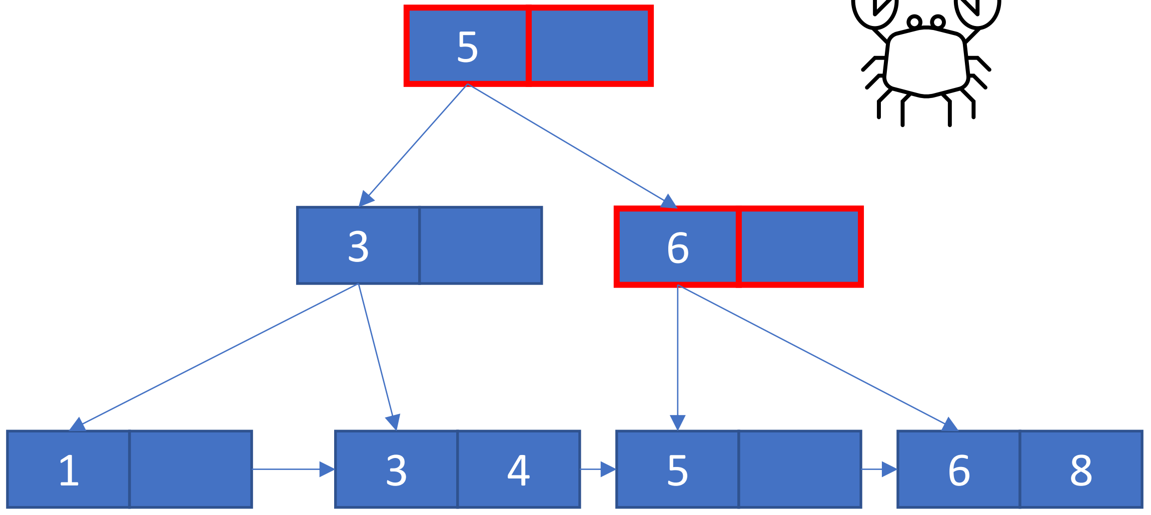
B+ tree: multithreading

- Latch coupling
  - Acquire lock for parent
  - Acquire lock for child
  - Release parent when able to
- Find 6



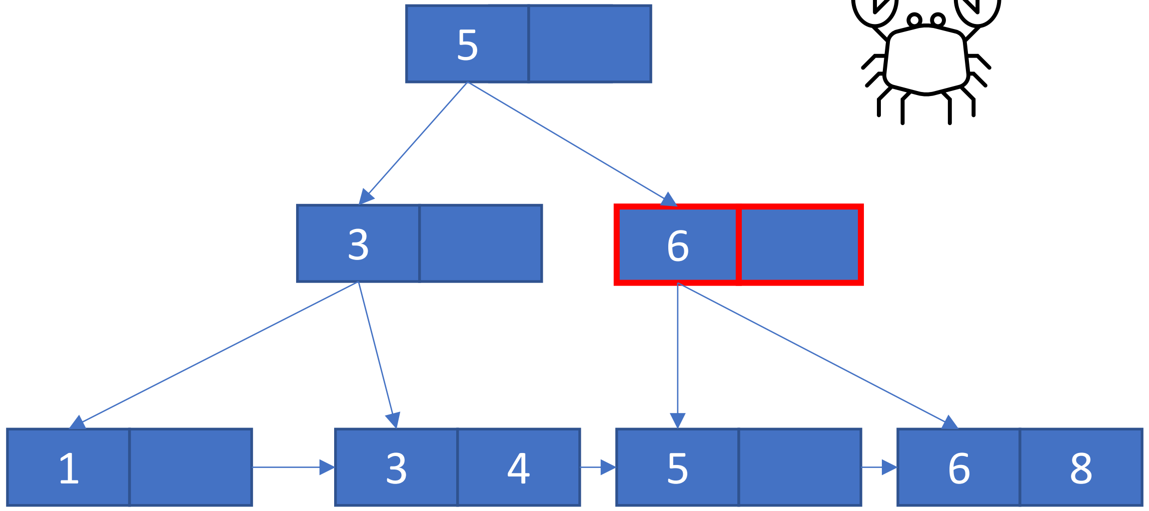
B+ tree: multithreading

- Latch coupling
    - Acquire lock for parent
    - Acquire lock for child
    - Release parent when able to
- Find 6



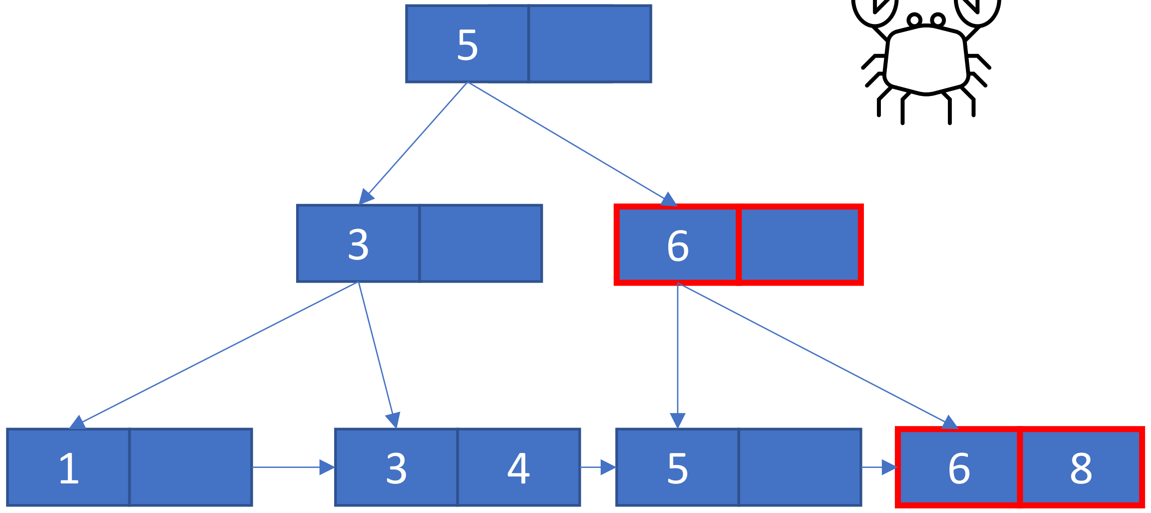
B+ tree: multithreading

- Latch coupling
  - Acquire lock for parent
  - Acquire lock for child
  - Release parent when able to
- Find 6



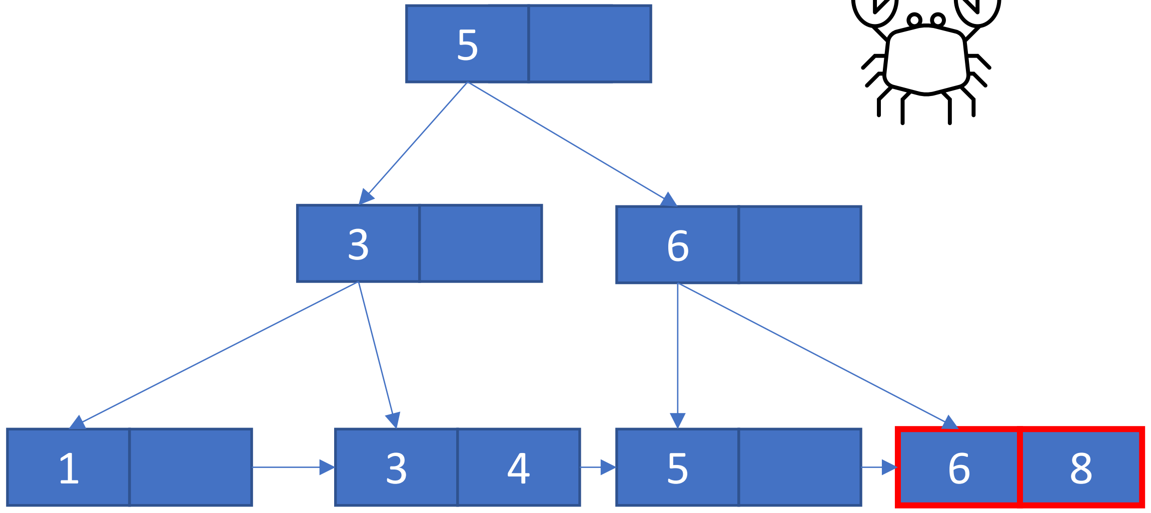
B+ tree: multithreading

- Latch coupling
  - Acquire lock for parent
  - Acquire lock for child
  - Release parent when able to
- Find 6



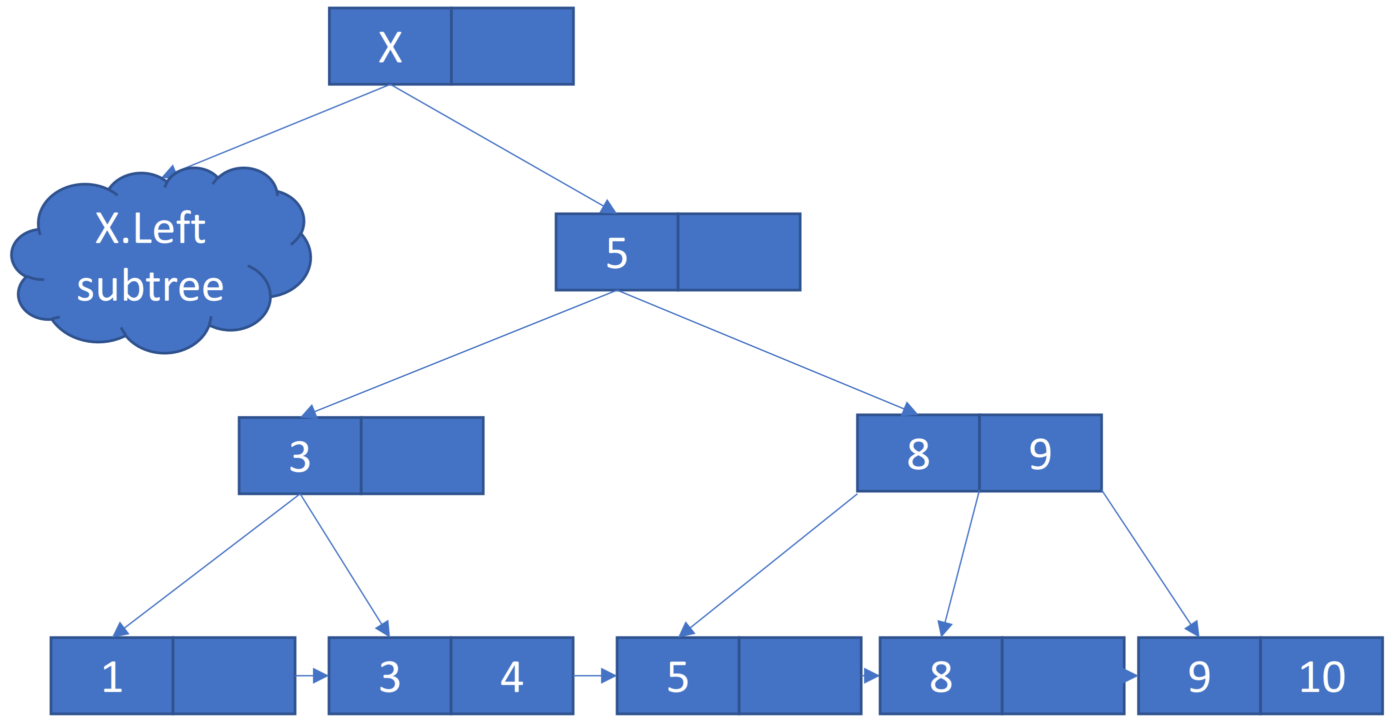
B+ tree: multithreading

- Latch coupling
  - Acquire lock for parent
  - Acquire lock for child
  - Release parent when able to
- Find 6: done



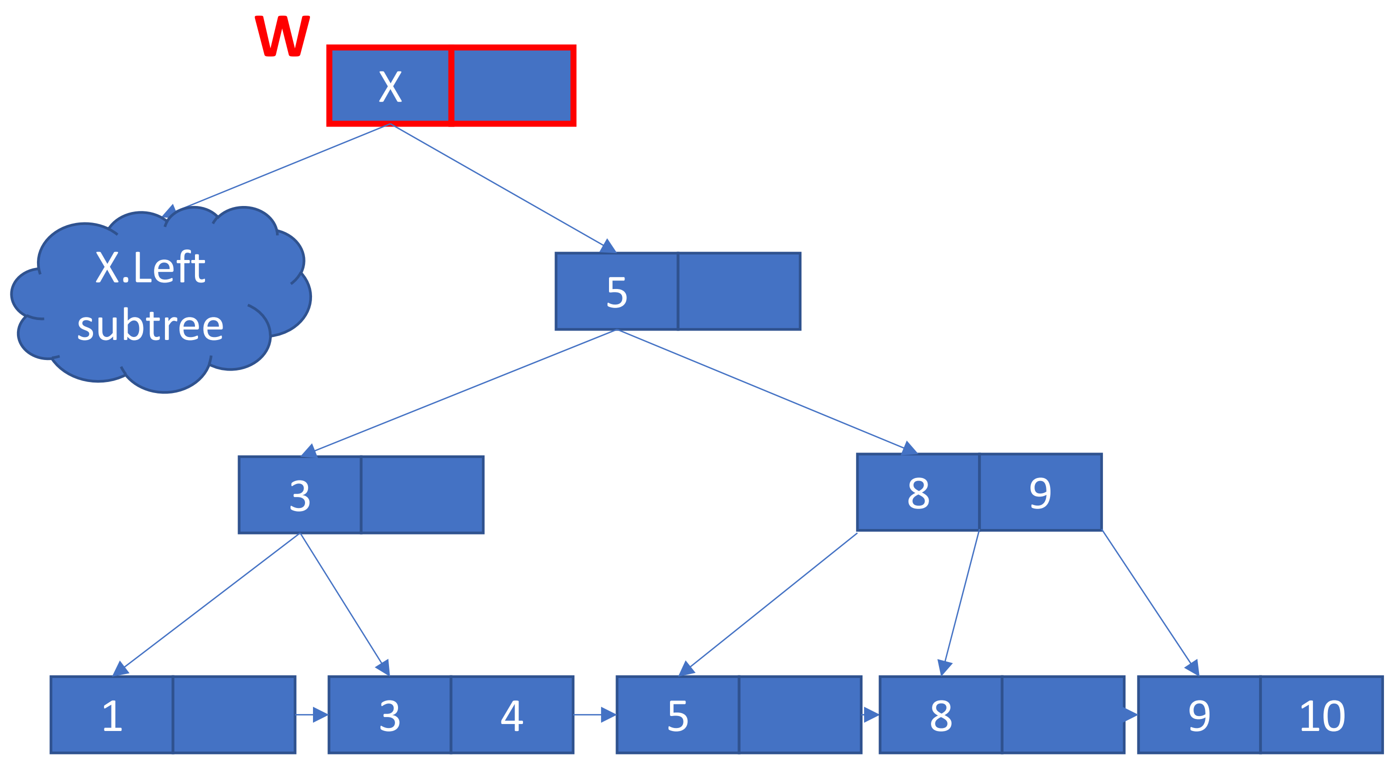
B+ tree: multithreading

- Latch coupling
  - Acquire lock for parent
  - Acquire lock for child
  - Release parent when able to
- Delete 9



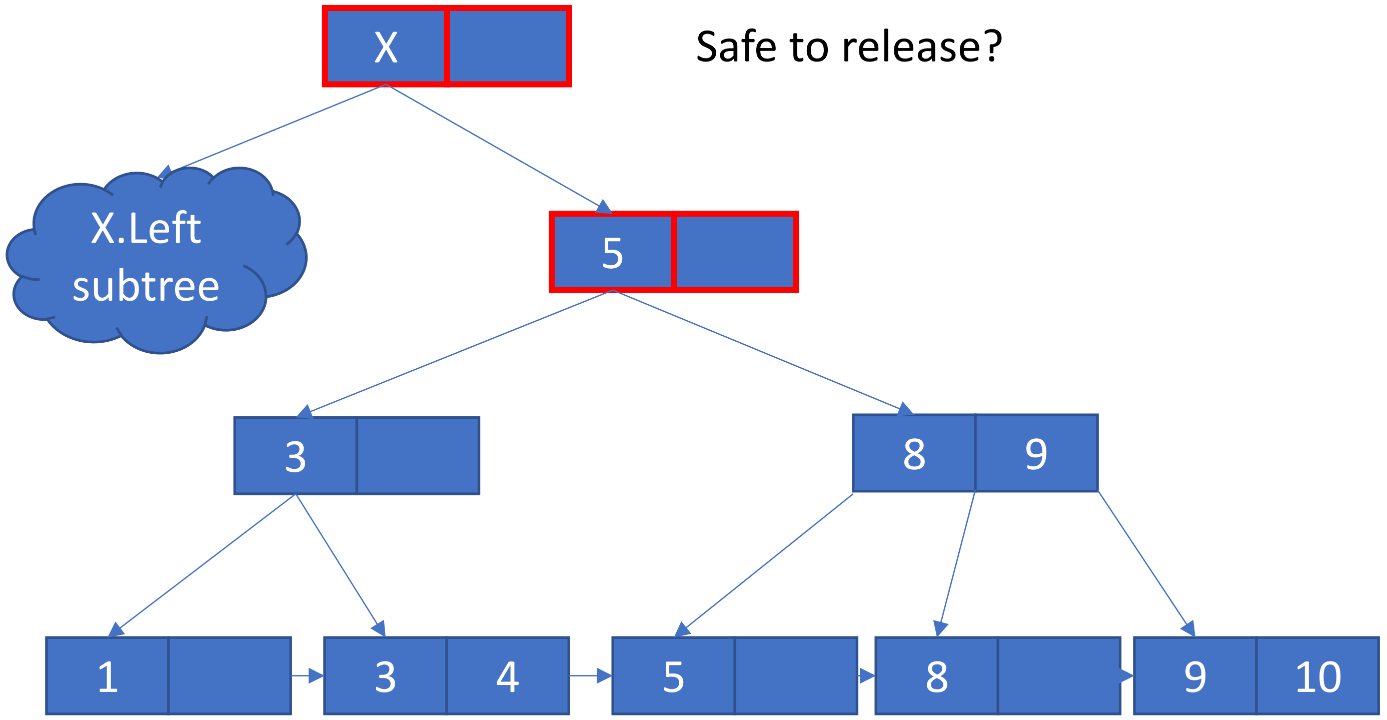
B+ tree: multithreading

- Latch coupling
  - Acquire lock for parent
  - Acquire lock for child
  - Release parent when able to
- Delete 9



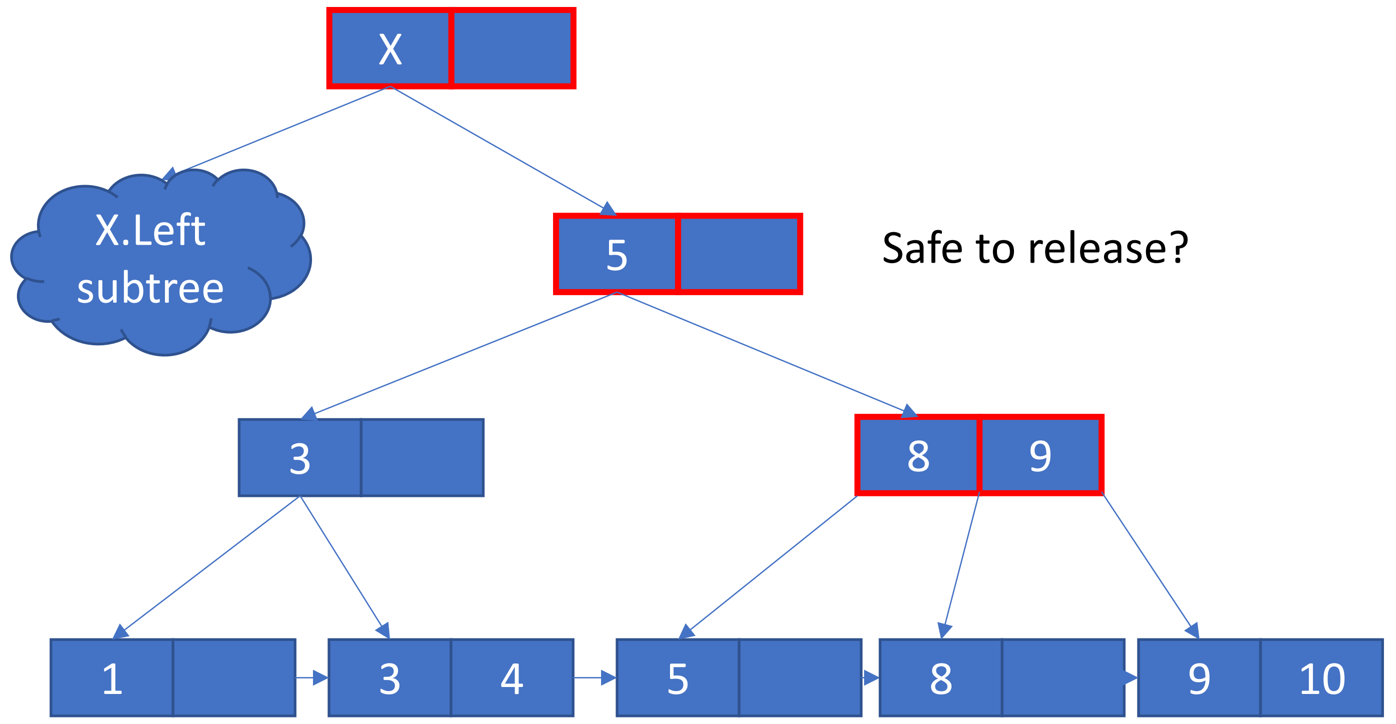
B+ tree: multithreading

- Latch coupling
  - Acquire lock for parent
  - Acquire lock for child
  - Release parent when able to
- Delete 9



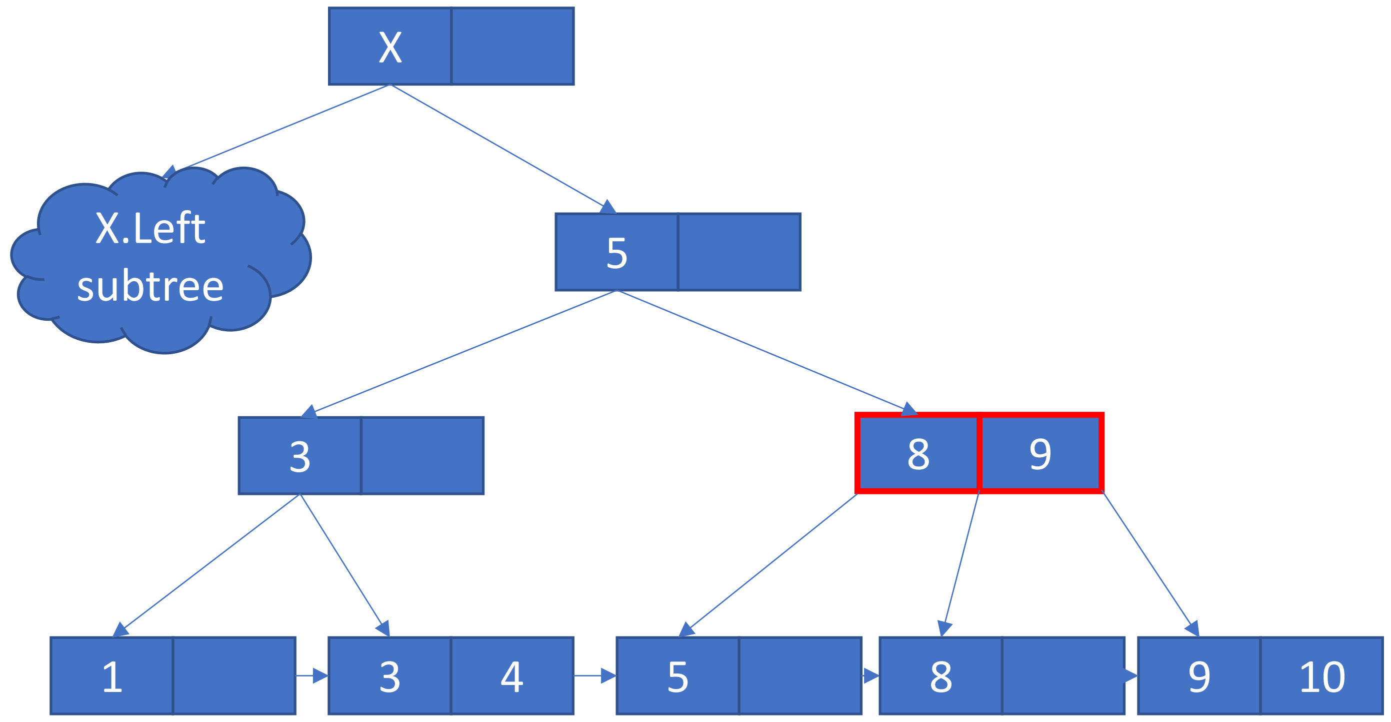
B+ tree: multithreading

- Latch coupling
  - Acquire lock for parent
  - Acquire lock for child
  - Release parent when able to
- Delete 9



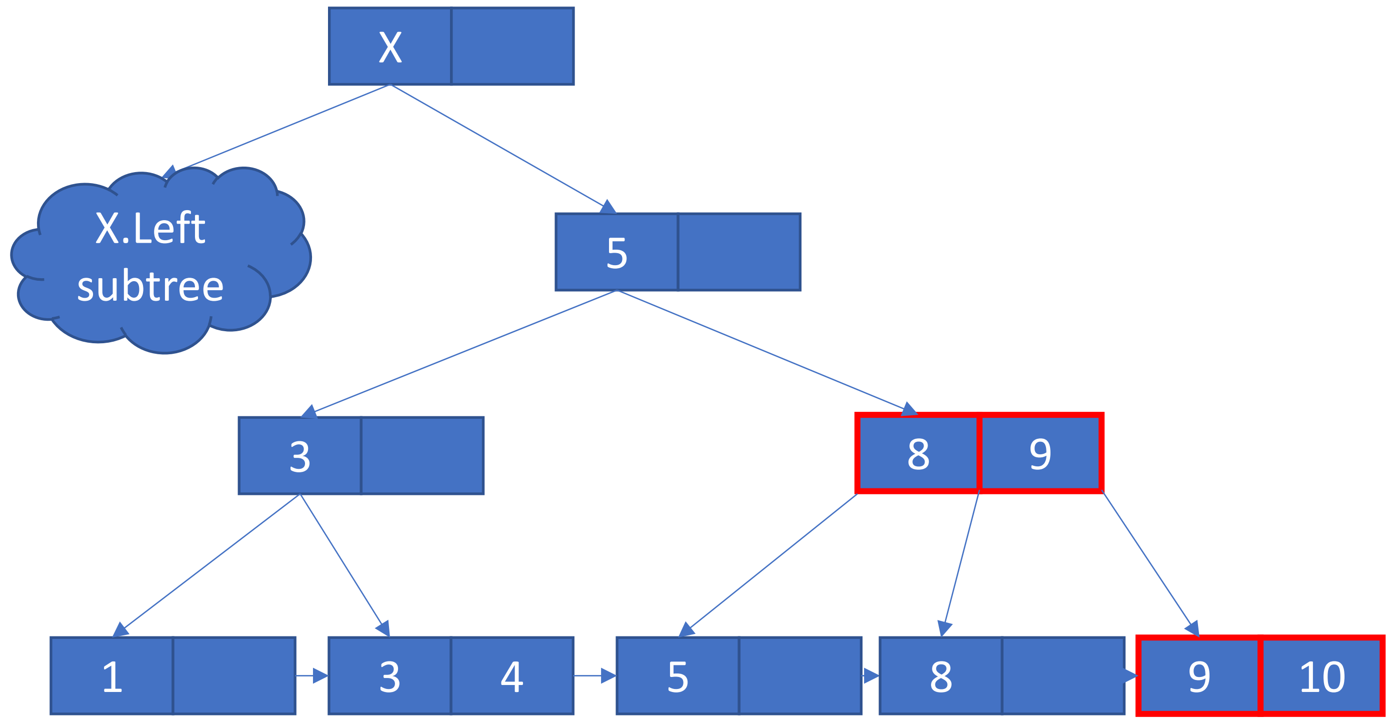
B+ tree: multithreading

- Latch coupling
  - Acquire lock for parent
  - Acquire lock for child
  - Release parent when able to
- Delete 9



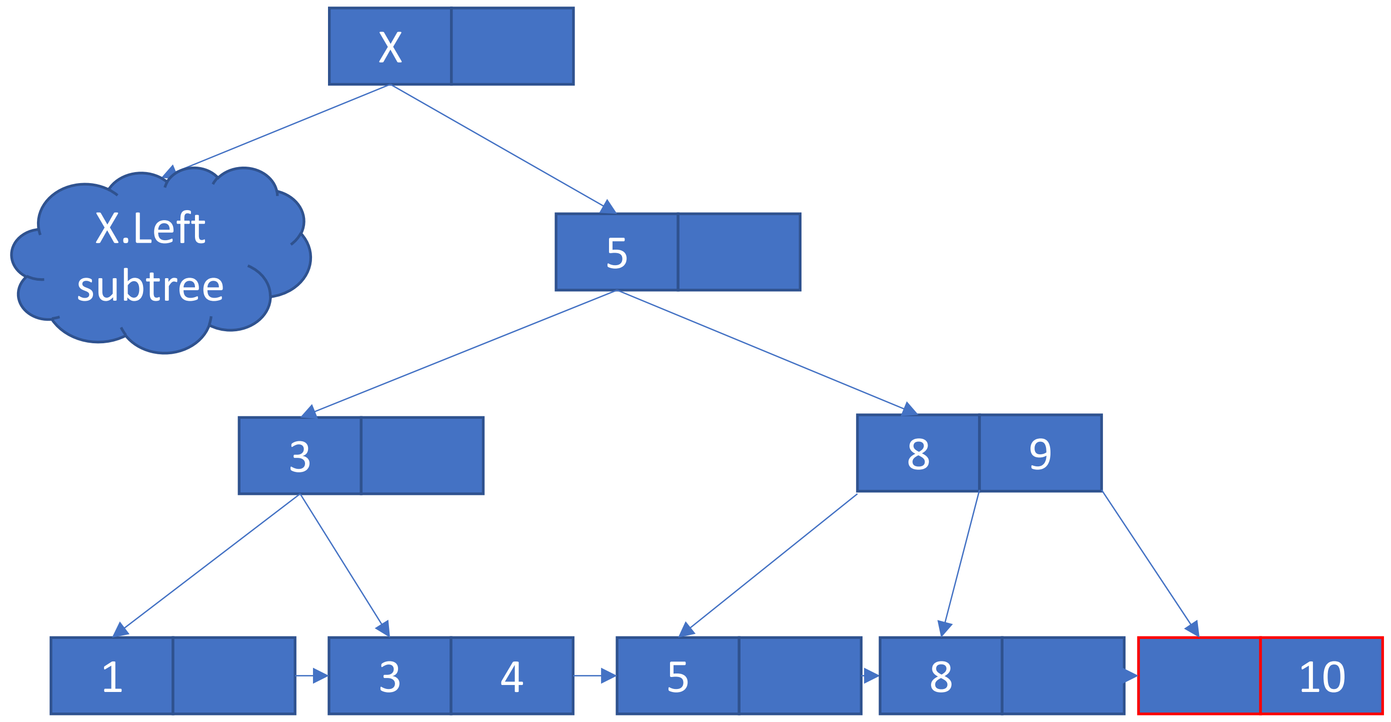
B+ tree: multithreading

- Latch coupling
  - Acquire lock for parent
  - Acquire lock for child
  - Release parent when able to
- Delete 9



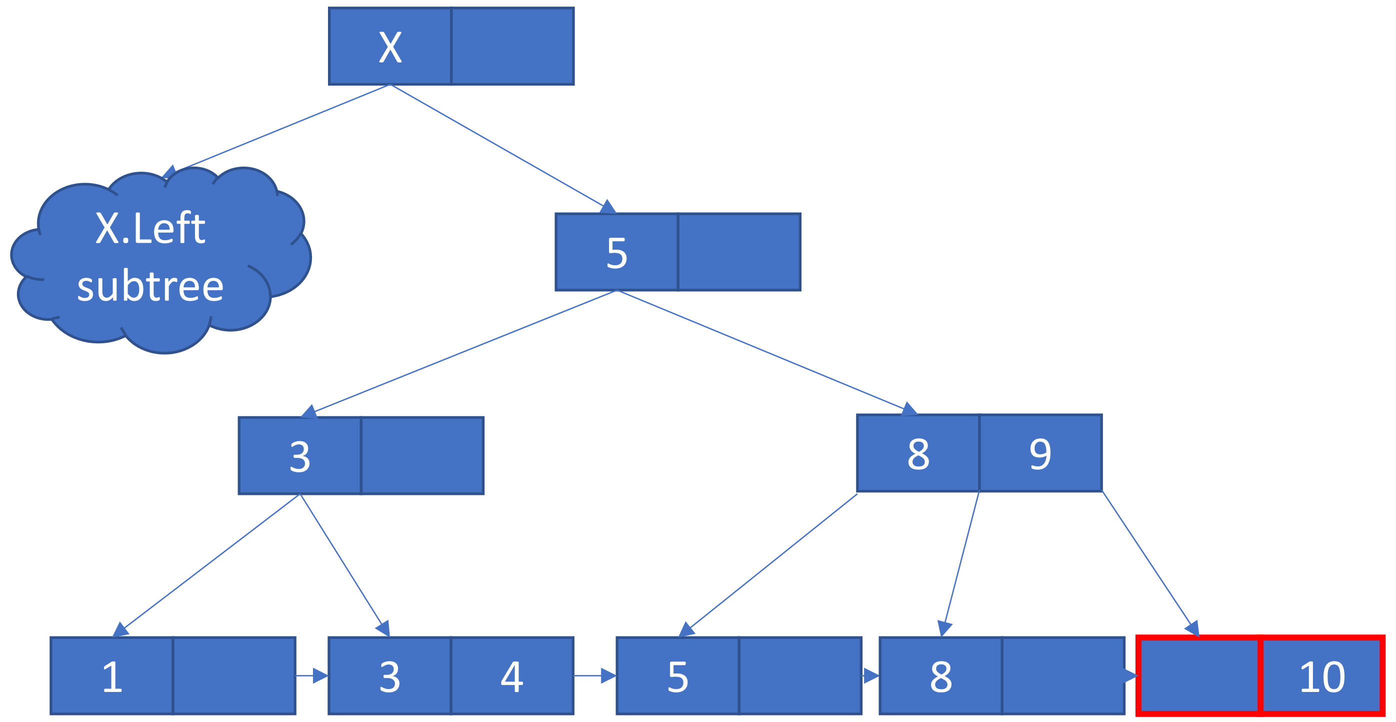
B+ tree: multithreading

- Latch coupling
  - Acquire lock for parent
  - Acquire lock for child
  - Release parent when able to
- Delete 9



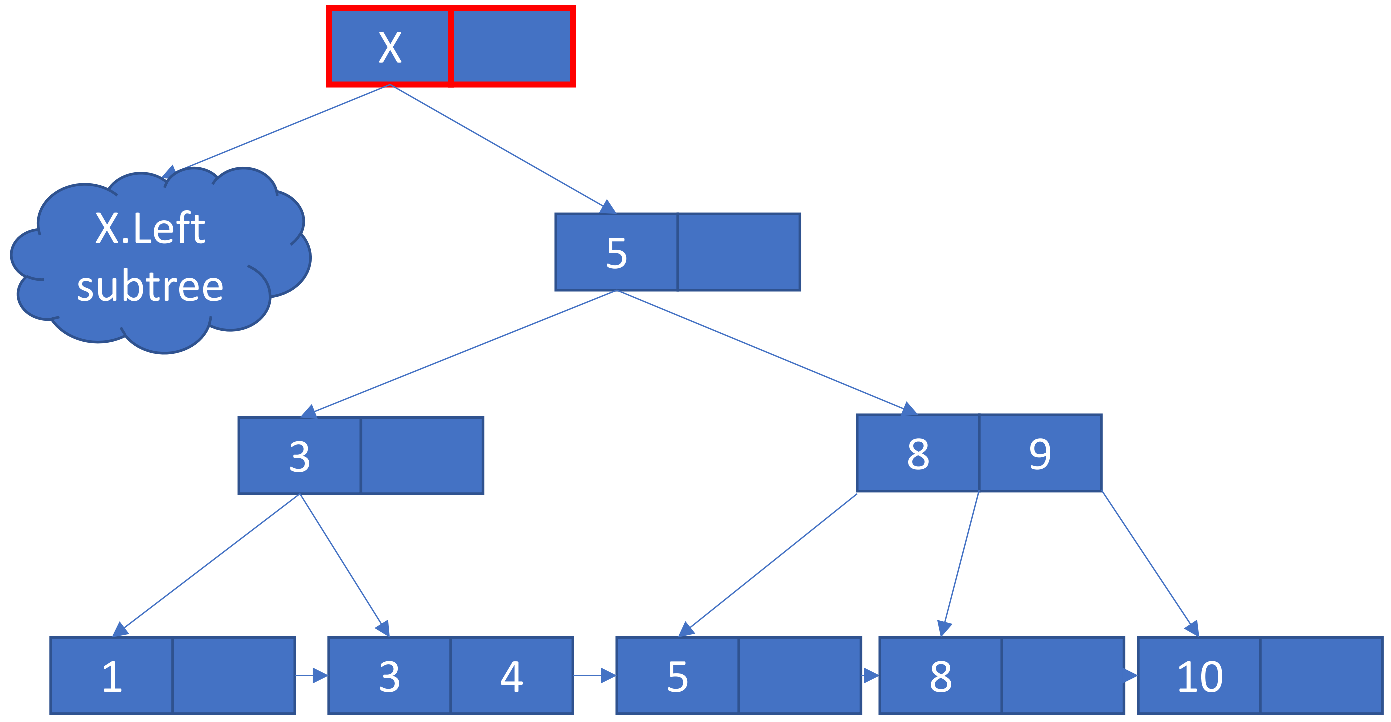
B+ tree: multithreading

- Latch coupling
  - Acquire lock for parent
  - Acquire lock for child
  - Release parent when able to
- Delete 9



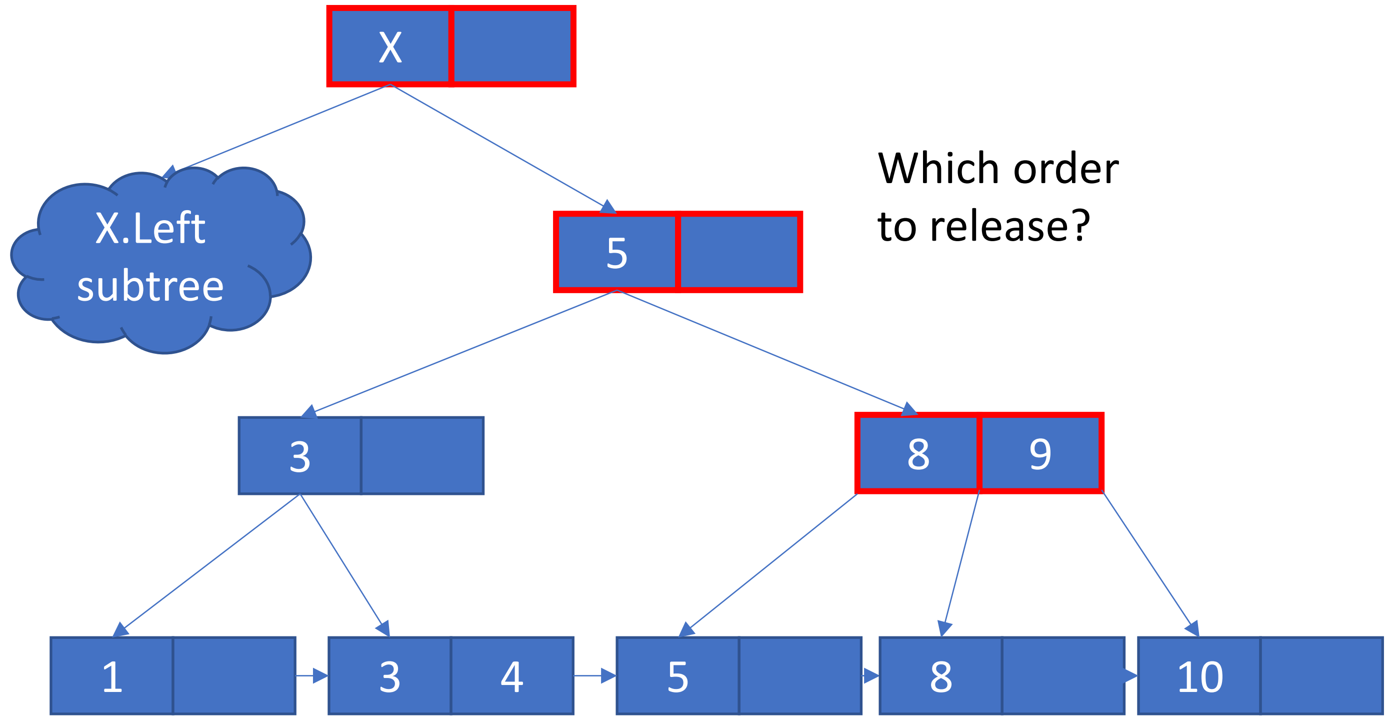
B+ tree: multithreading

- Latch coupling
  - Acquire lock for parent
  - Acquire lock for child
  - Release parent when able to
- Delete 10



B+ tree: multithreading

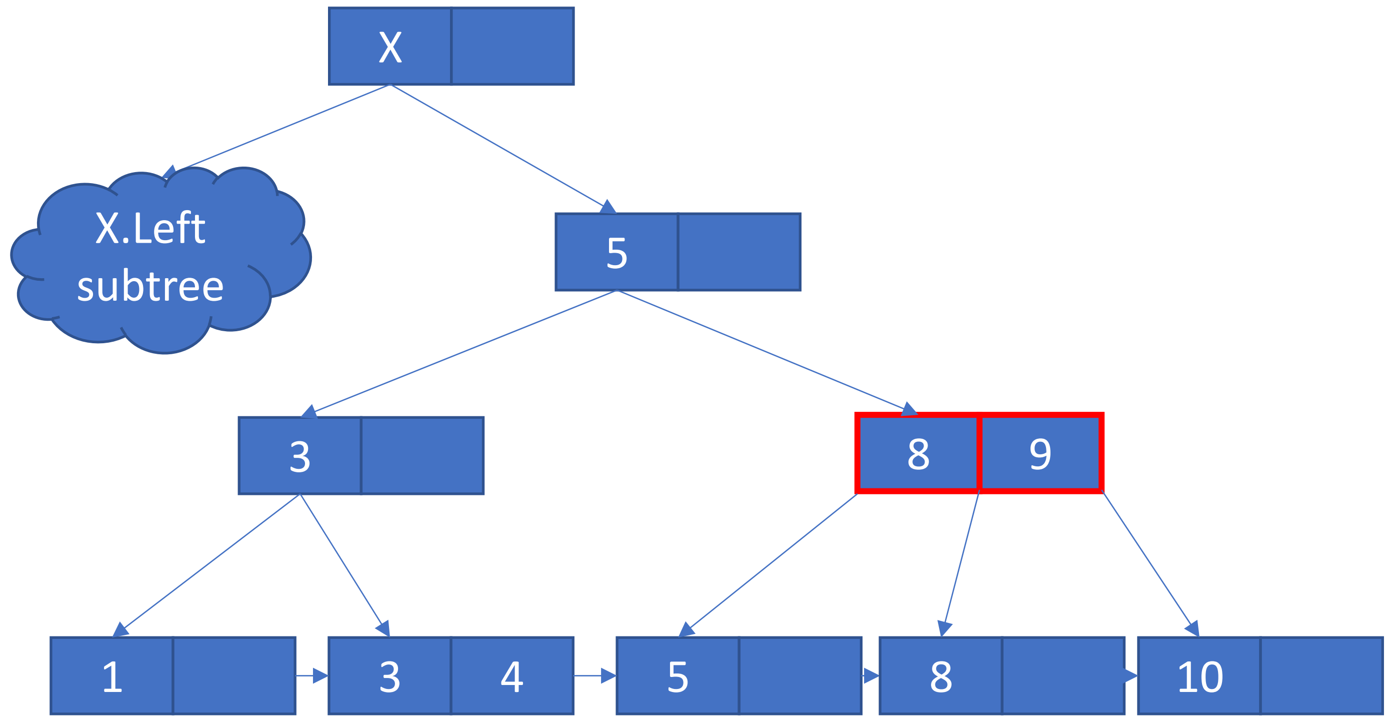
- Latch coupling
  - Acquire lock for parent
  - Acquire lock for child
  - Release parent when able to
- Delete 10



Which order to release?

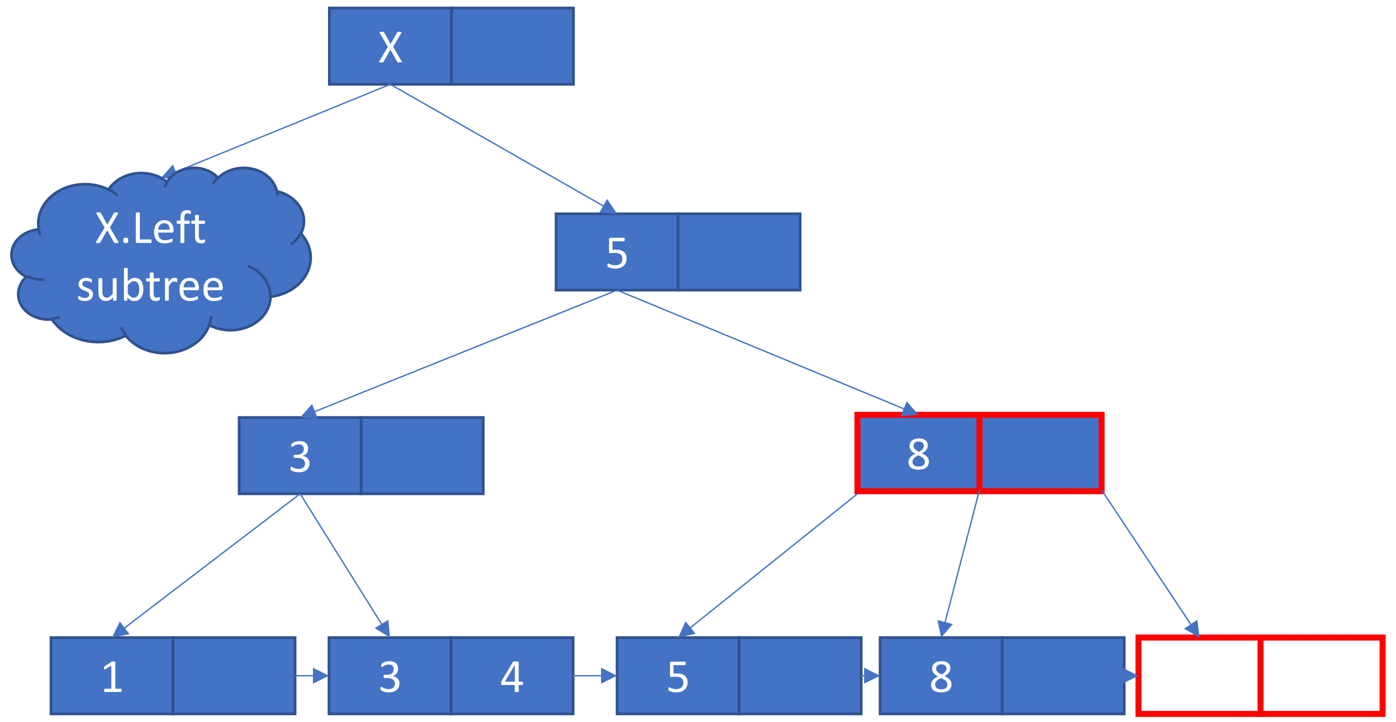
B+ tree: multithreading

- Latch coupling
    - Acquire lock for parent
    - Acquire lock for child
    - Release parent when able to
- Delete 10



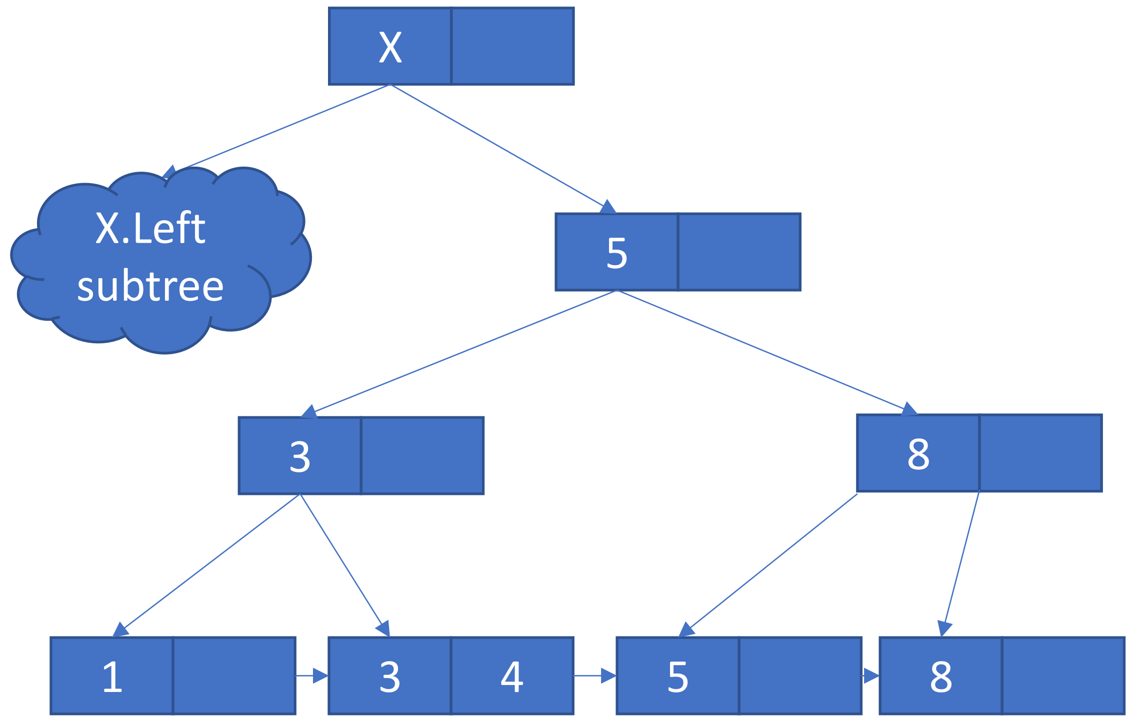
B+ tree: multithreading

- Latch coupling
  - Acquire lock for parent
  - Acquire lock for child
  - Release parent when able to
- Delete 10



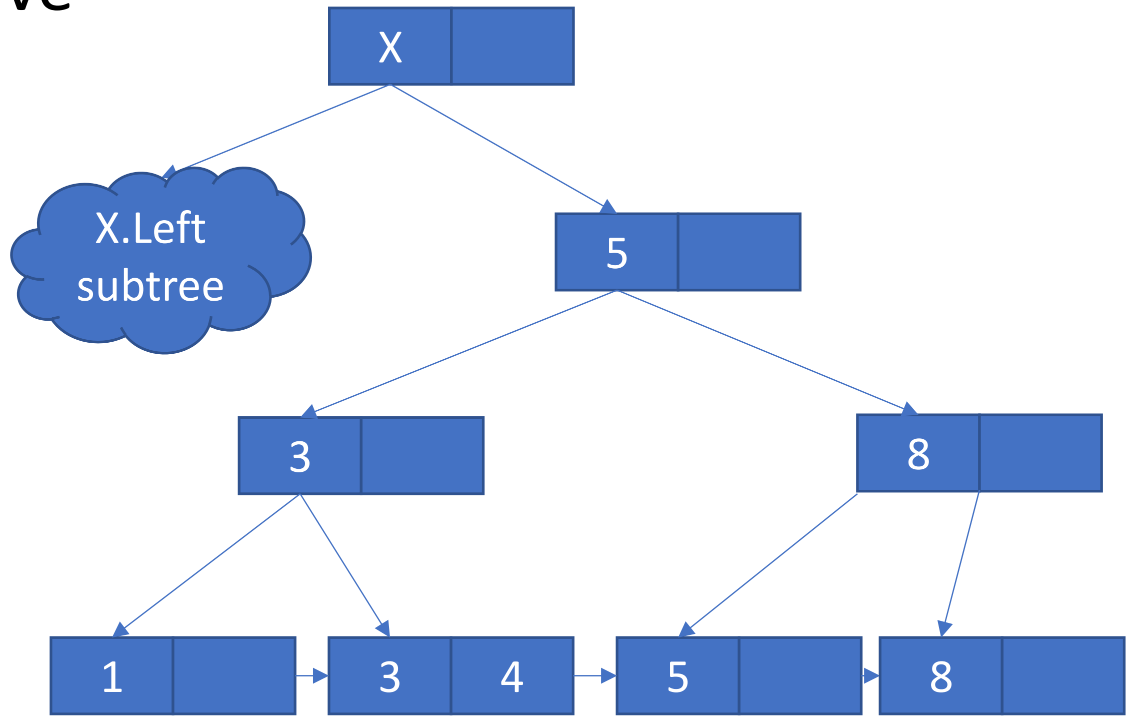
B+ tree: multithreading

- Latch coupling
  - Acquire lock for parent
  - Acquire lock for child
  - Release parent when able to
- Delete 10: done



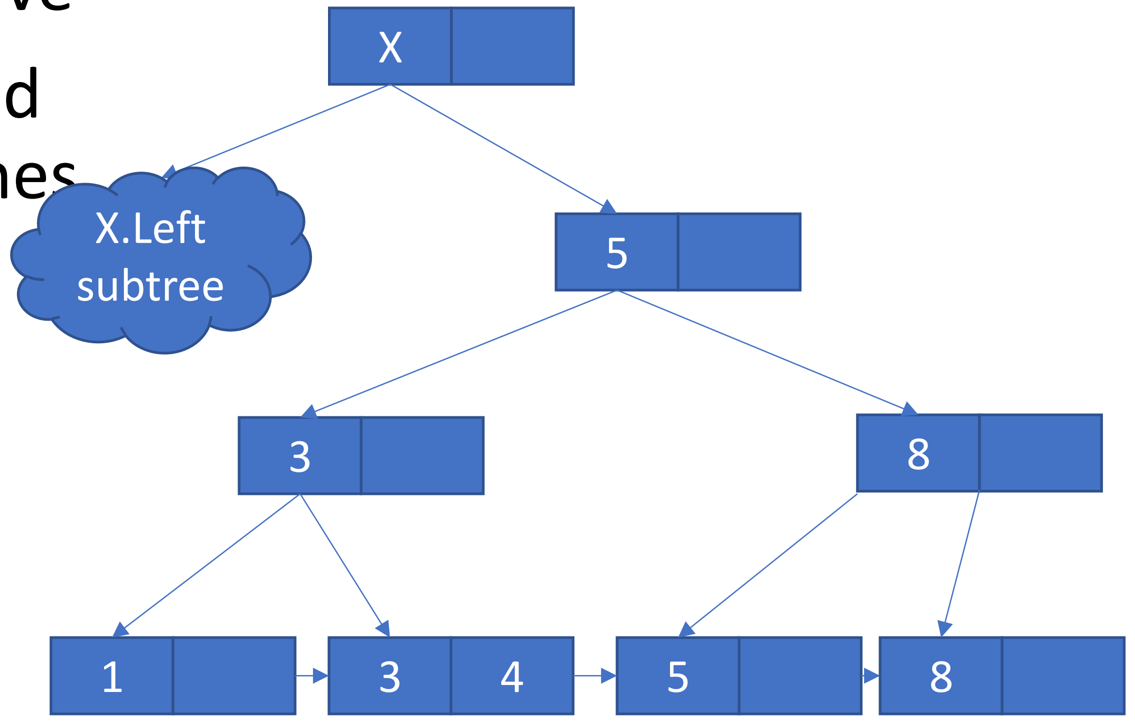
Anything better?

- Write-Latching root is expensive



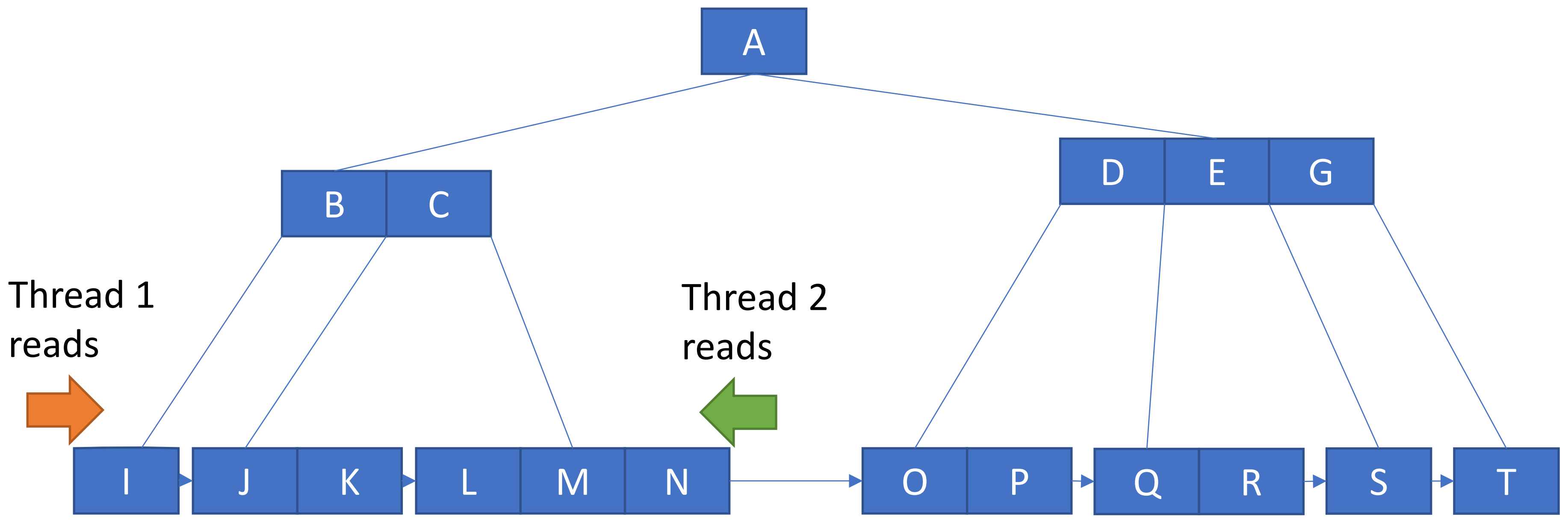
Anything better?

- Write-Latching root is expensive
- Optimistic assumption: instead of write latches use read latches
- On conflict – abort & rerun

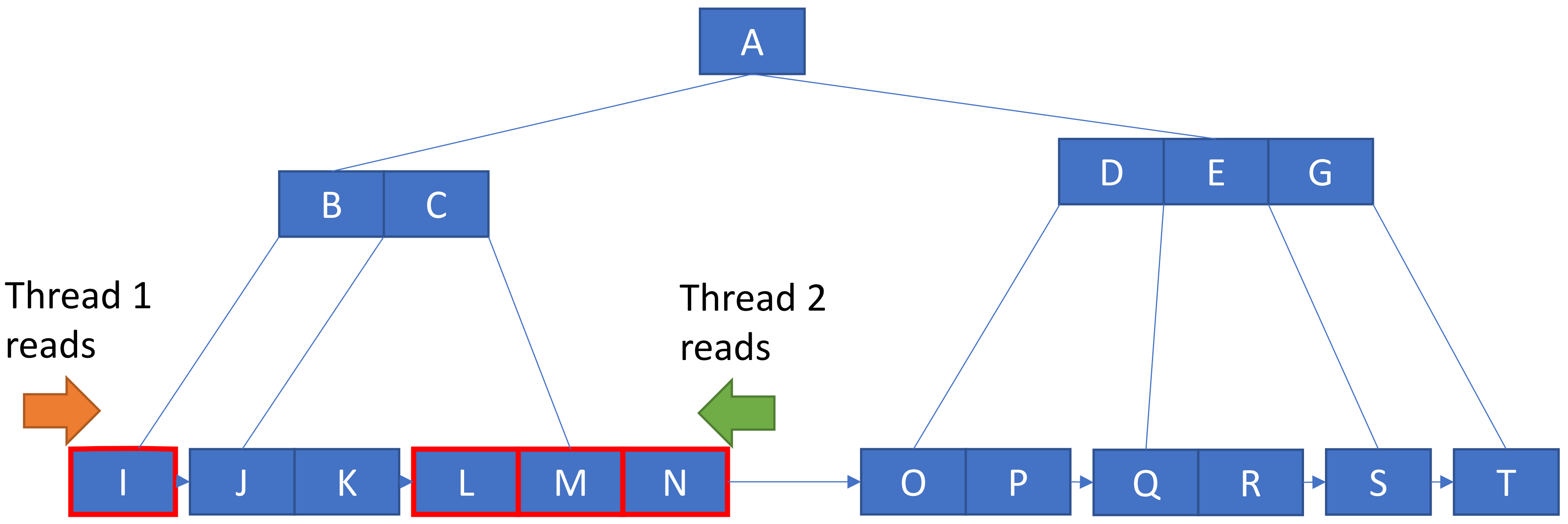


Leaves traversal

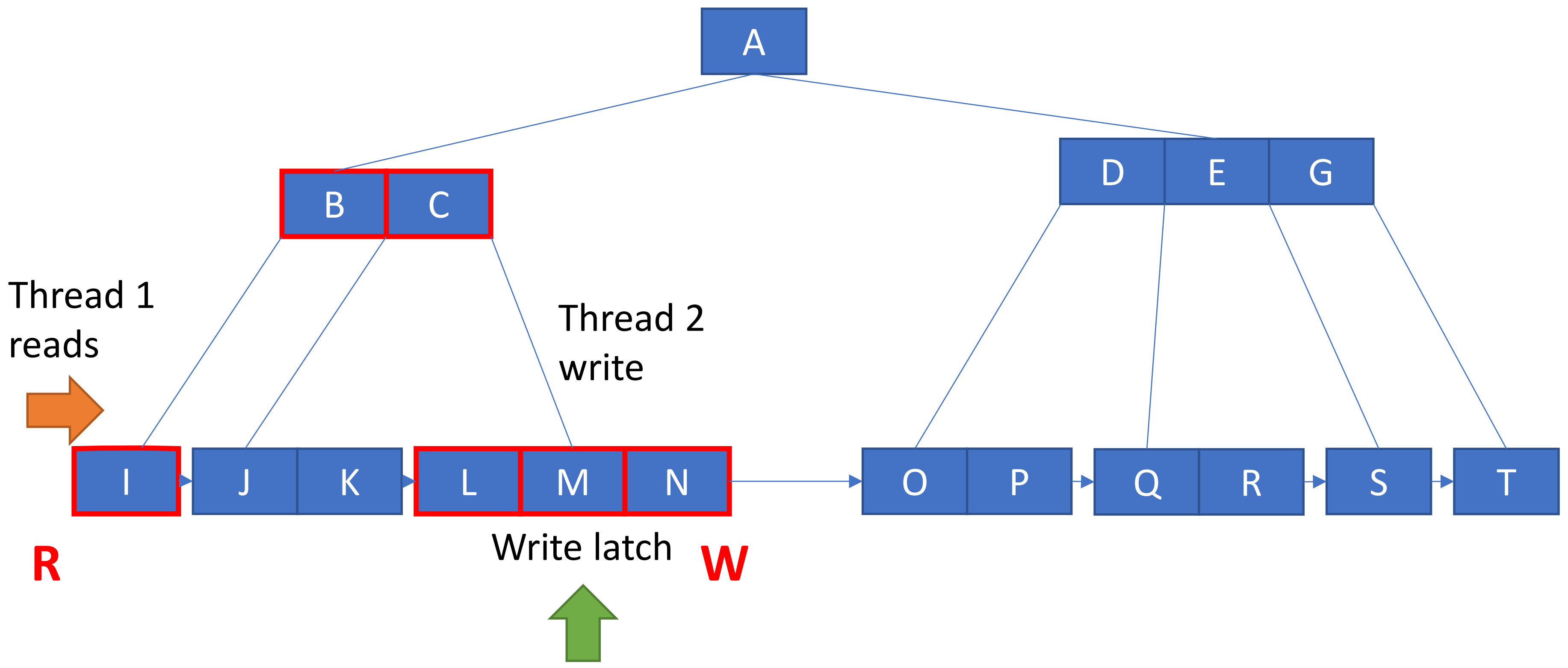
Thread 2: find all less than N



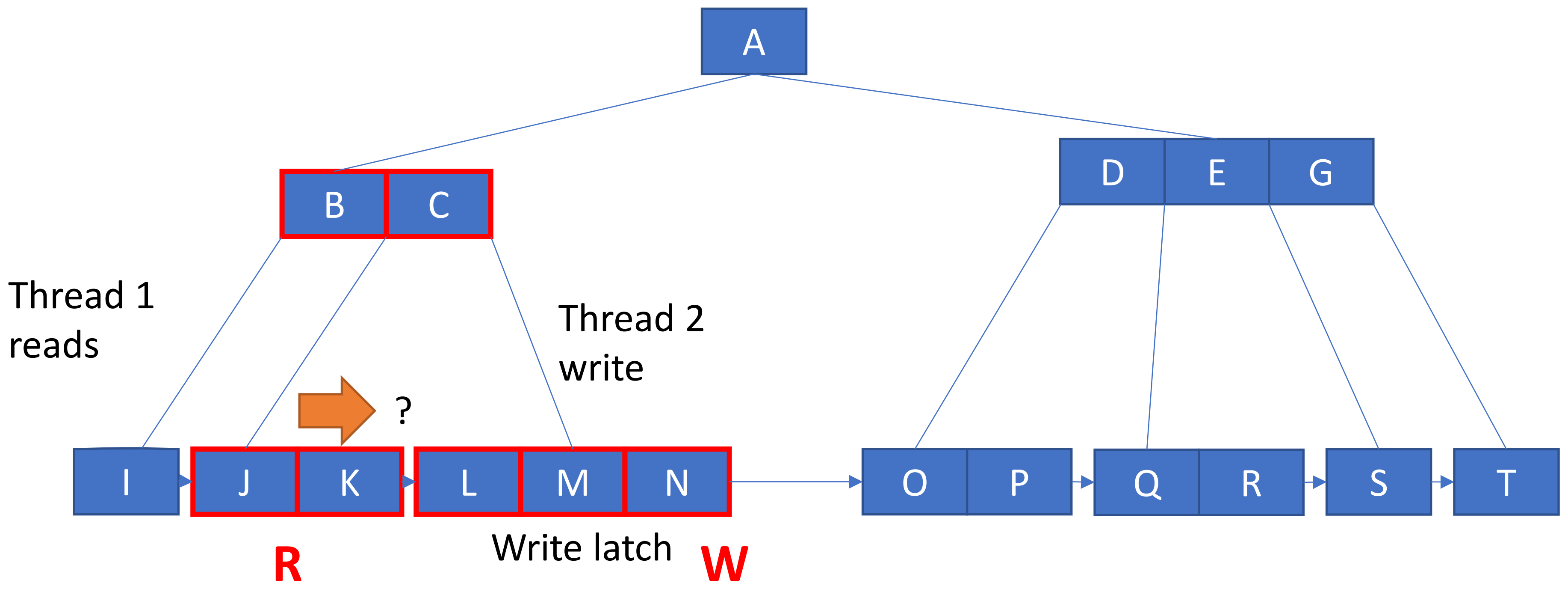
Leaves traversal



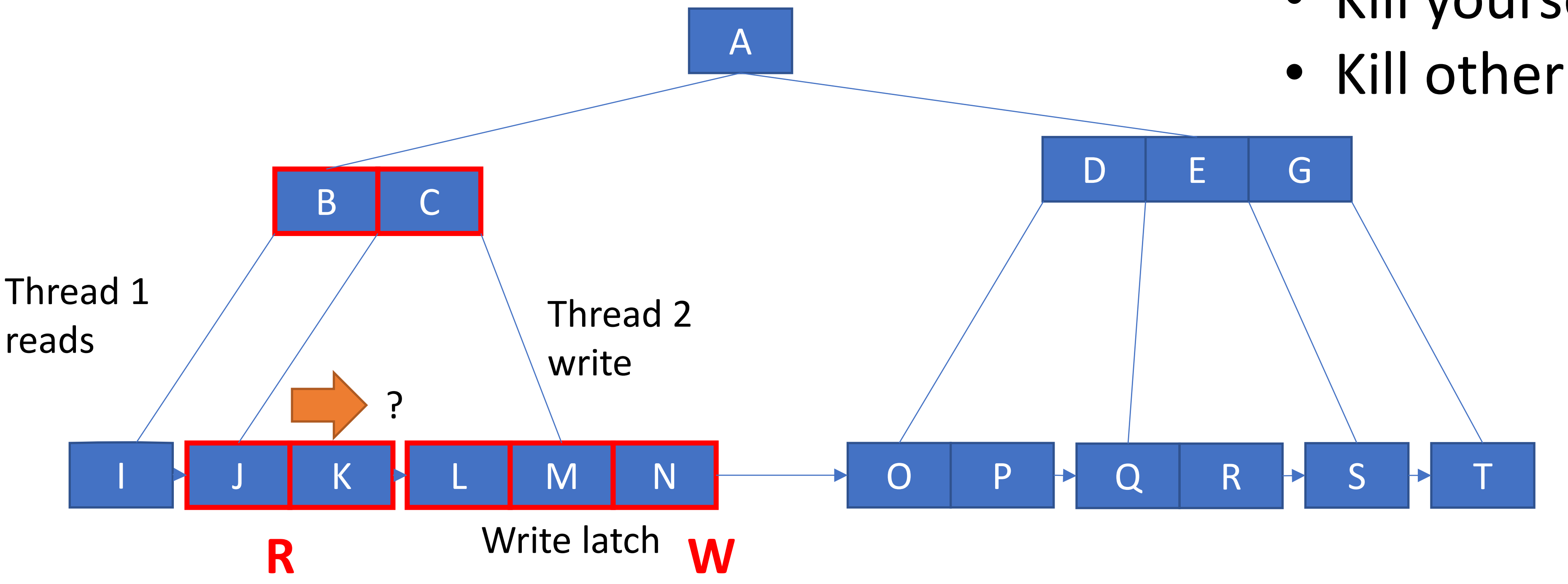
Leaves traversal



Leaves traversal



Leaves traversal



- Wait
- Kill yourself
- Kill other

Resources

- [1] Introduction to Algorithms, Thomas H. Cormen, chapters 18.
- [2] B tree visualization
- [3] B+ tree visualization
- [4] further reading: Bw tree

BACKUP