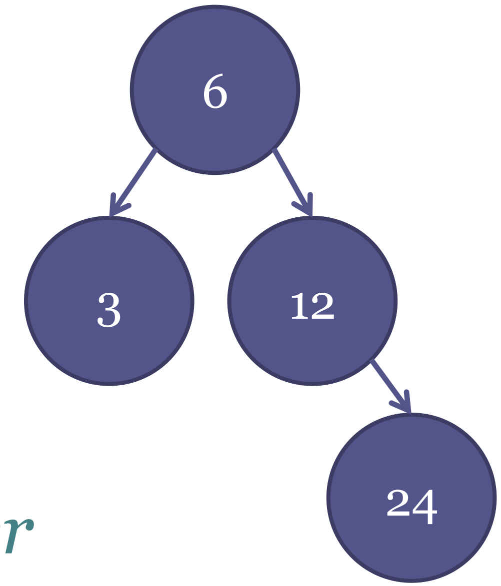
Practical Concurrent Binary Search Trees via Logical Ordering

Dana Drachsler (Technion, Israel)
Martin Vechev (ETH, Switzerland)
Eran Yahav (Technion , Israel)

Binary Search Tree

- A data-structure that stores elements
  - Each element has a unique key
  - Duplications are not allowed
- The BST consists of nodes
  - Each node stores an element
- For each node
  - Keys in the left sub-tree are *smaller*
  - Keys in the right sub-tree are *bigger*

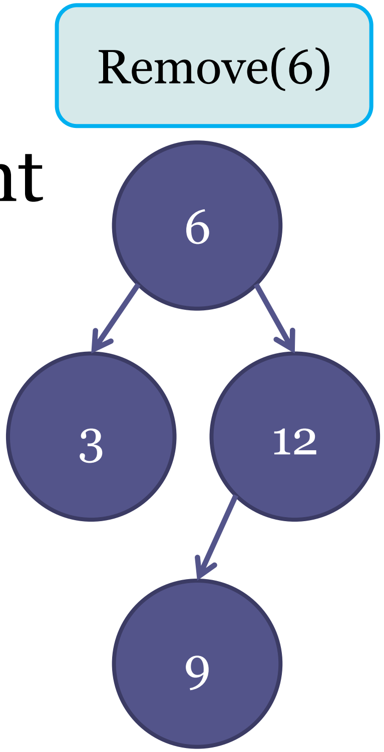


Binary Search Tree Operations

- Contains(k): check if k is present
- Insert(k): insert k if it is not yet present
- Remove(k): remove k if present

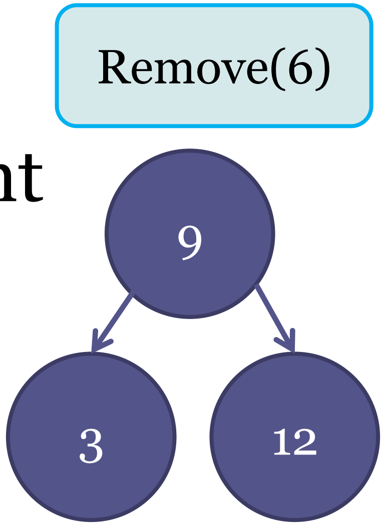
Binary Search Tree Operations

- Contains(k): check if k is present
- Insert(k): insert k if it is not yet present
- Remove(k): remove k if present
  - Removal of a node with 2 children
    - Find the successor: the left-most node of the right sub-tree



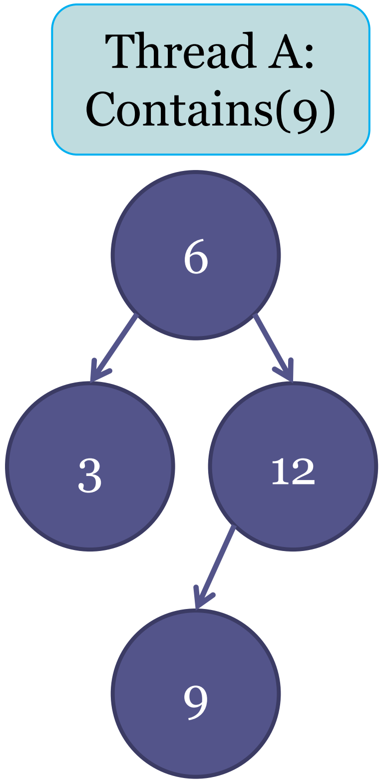
Binary Search Tree Operations

- Contains(k): check if k is present
- Insert(k): insert k if it is not yet present
- Remove(k): remove k if present
  - Removal of a node with 2 children
    - Find the successor: the left-most node of the right sub-tree
    - Relocate the successor



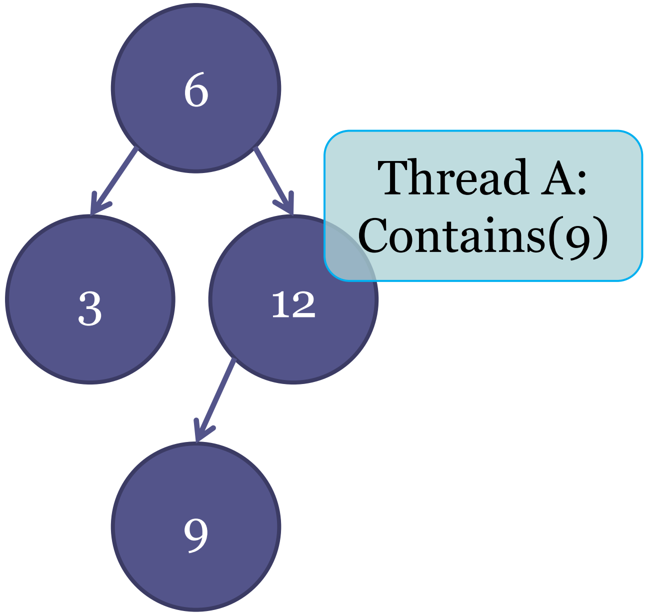
BST: Challenge with Concurrency

1. Thread A searches for 9



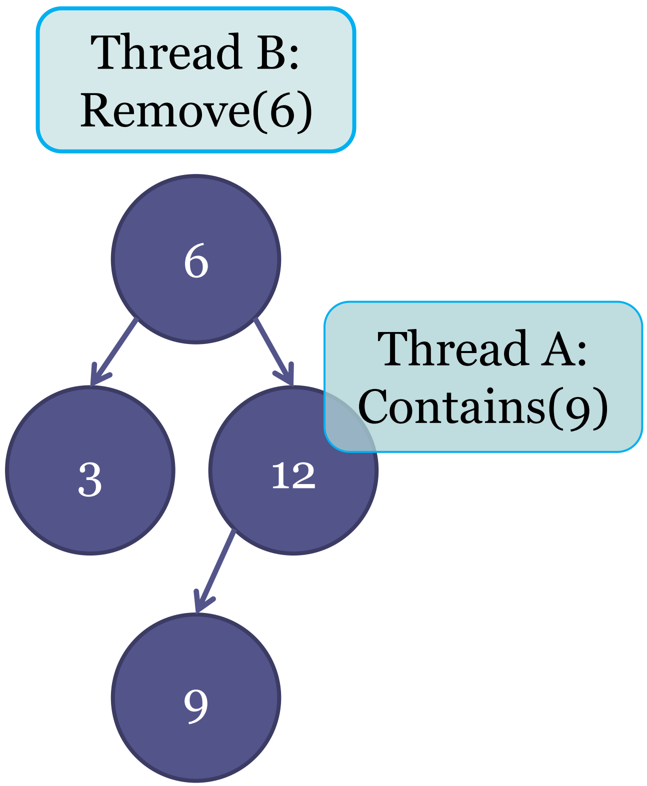
BST: Challenge with Concurrency

1. Thread A searches for 9 and pauses



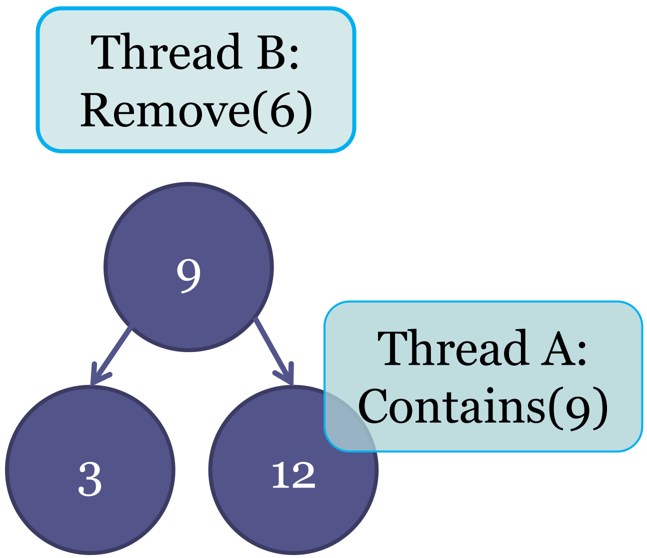
BST: Challenge with Concurrency

1. Thread A searches for 9 and pauses
2. Thread B removes 6



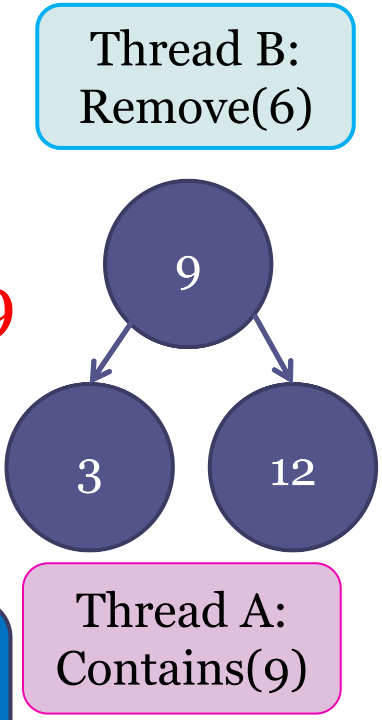
BST: Challenge with Concurrency

1. Thread A searches for 9 and pauses
2. Thread B removes 6



BST: Challenge with Concurrency

1. Thread A searches for 9 and pauses
2. Thread B removes 6
3. Thread A resumes and misses 9



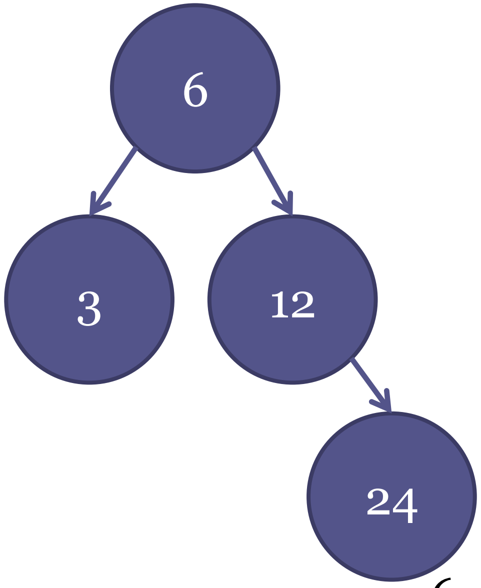
Search operation unaware of concurrent changes to BST layout

Our Contribution

- We present a new perspective on BST
  - Locking is based on a *logical ordering layout*, and not only on the BST layout
- The additional layout requires
  - Extra space for the new links
  - Extra time for maintaining the new links
  - Extra lock acquires of the new links
- Yet, it performs as state-of-the-art algorithms
  - Sometimes even better

Key Idea

- There is a total order between the keys



Key Idea

- There is a total order between the keys

$$-\infty < 3 < 6 < 12 < 24 < \infty$$

Key Idea

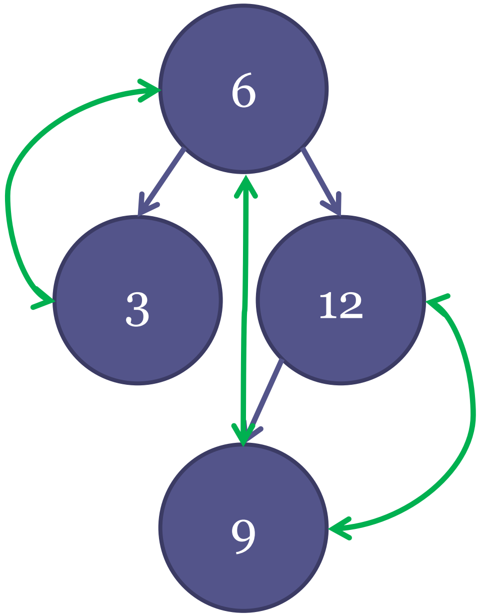
- There is a total order between the keys
- The order induces *intervals*
  - A key is present in the tree if it is an end point of some interval
- We explicitly maintain the intervals
  - *The logical ordering layout*

$-\infty < 3 < 6 < 12 < 24 < \infty$

$(-\infty, 3)$ $(3,6)$ $(6,12)$ $(12,24)$ $(24, \infty)$

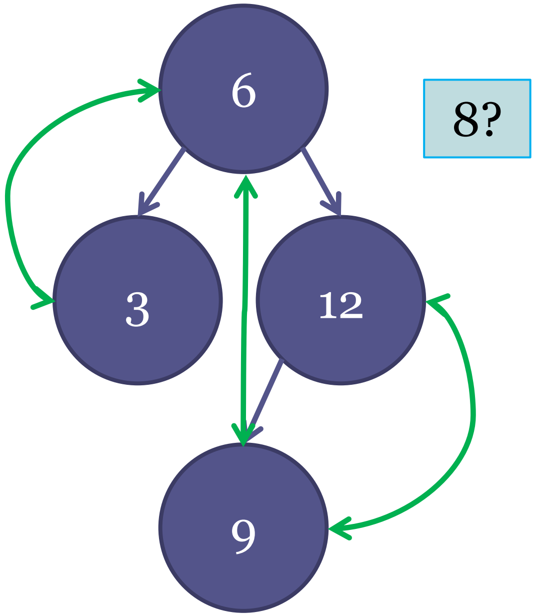
Logical Ordering Layout

- Connect n to its predecessor, p, and successor, s
  - n can access efficiently to (p, n),(n, s)



Logical Ordering Layout

- Connect n to its predecessor, p, and successor, s
  - n can access efficiently to (p, n),(n, s)
- To query whether k is in the tree
  - Find (p, s) such that $k \in [p, s]$

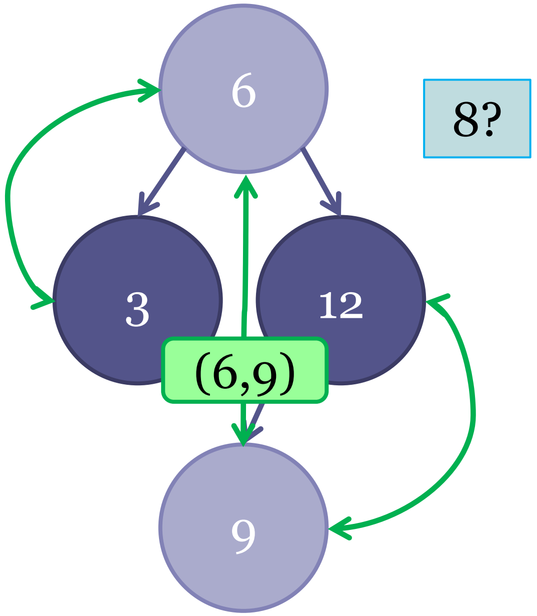


Logical Ordering Layout

- Connect n to its predecessor, p, and successor, s
  - n can access efficiently to (p, n),(n, s)

- To query whether k is in the tree
  - Find (p, s) such that $k \in [p, s]$

- For (p, s): p, s might be non adjacent in the tree

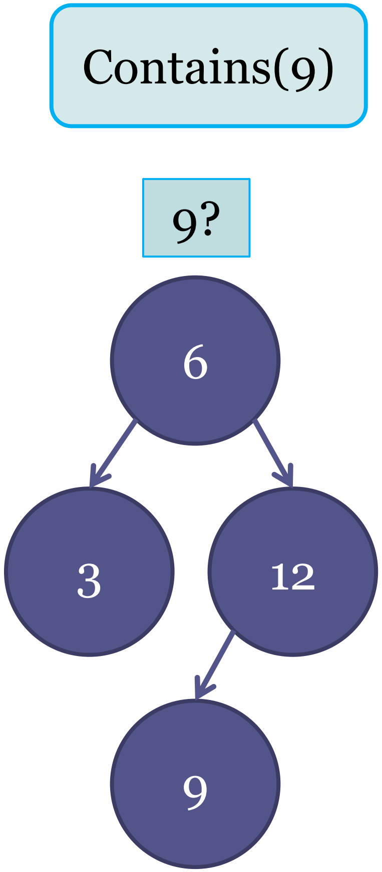


Main Advantages

- Efficiently answer membership queries even under concurrent updates to the BST layout
  - Includes relocating the successor in a removal
  - Includes applying sequential balancing operations
- Efficiently find the successor of a node
  - Important for the removal of a two-nodes parent
- Efficiently find the minimal/maximal keys
  - Can be used to implement a priority queue

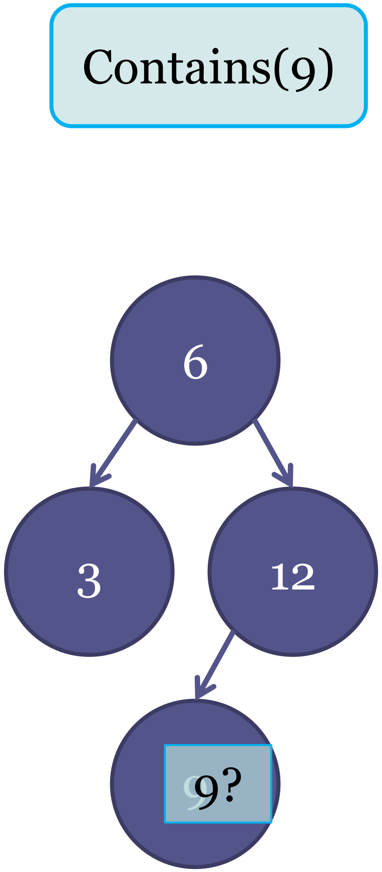
The Sequential Contains(k)

- Traverse downwards in the tree



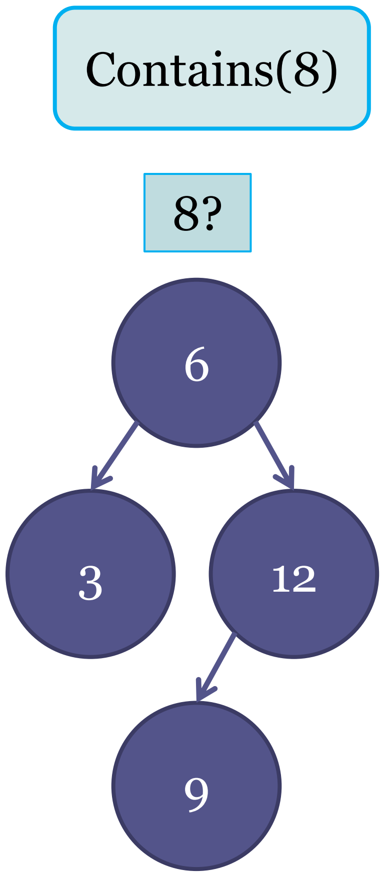
The Sequential Contains(k)

- Traverse downwards in the tree
- If k was found, return true



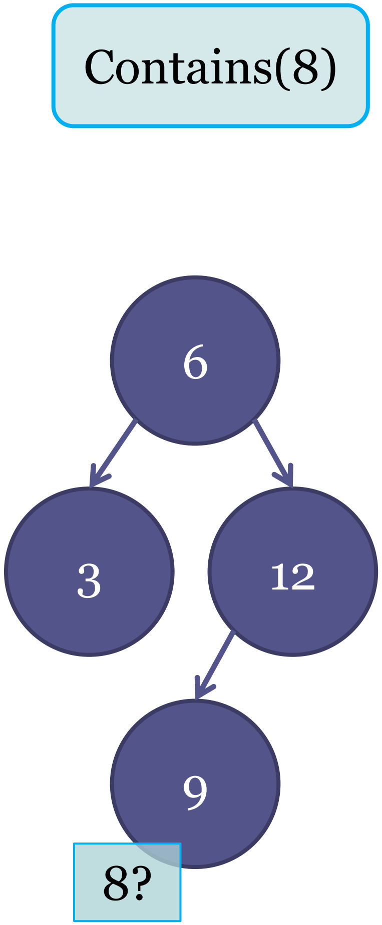
The Sequential Contains(k)

- Traverse downwards in the tree



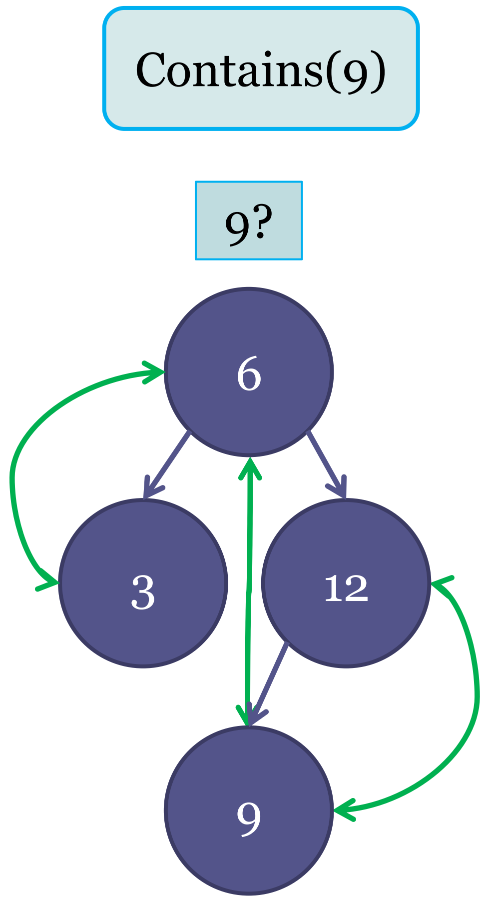
The Sequential Contains(k)

- Traverse downwards in the tree
- If reached to an end of a path,
return false



The Concurrent Contains(k)

- Traverse downwards in the tree

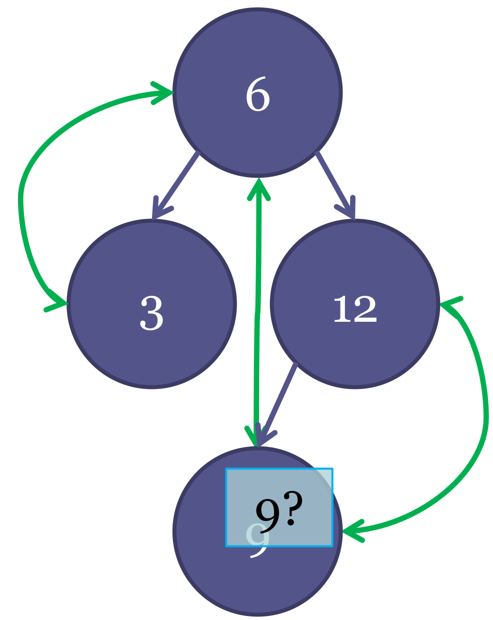


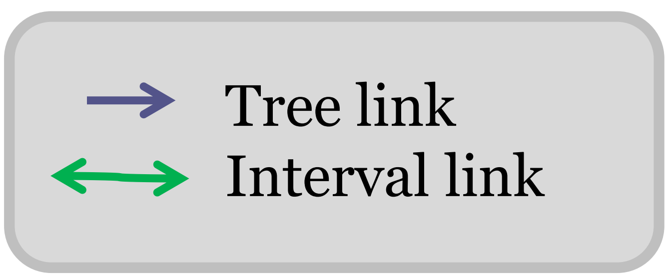
Tree link
Interval link

The Concurrent Contains(k)

Contains(9)

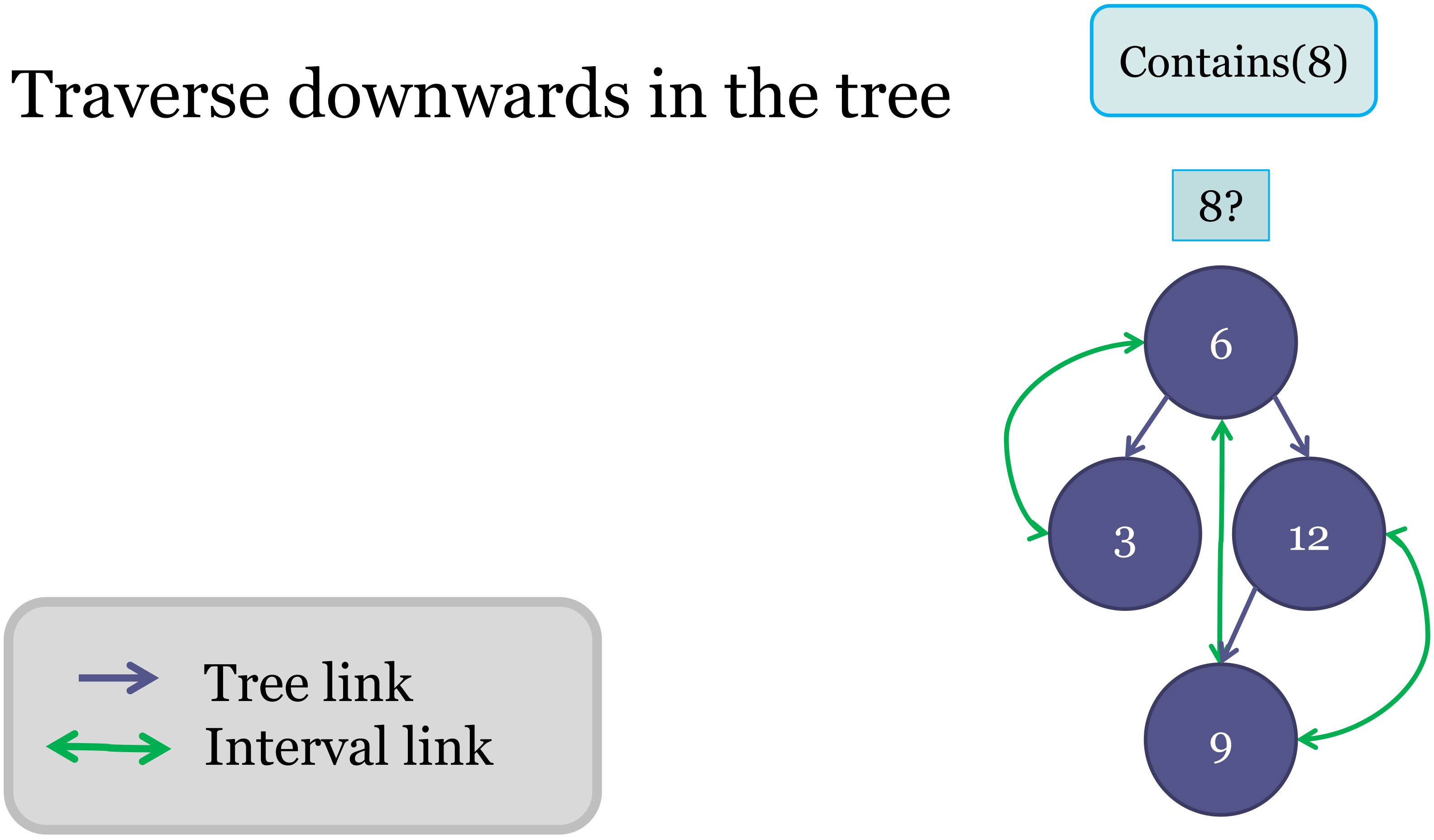
- Traverse downwards in the tree
- If k was found, return true





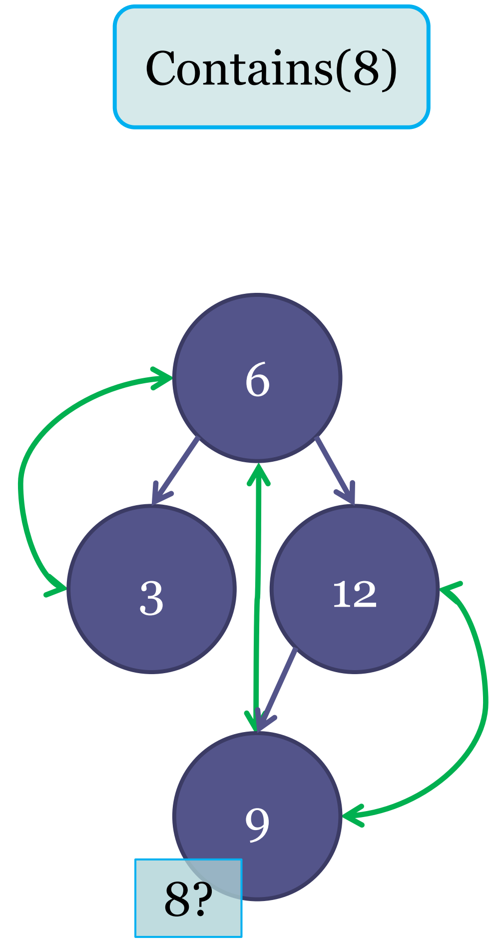
The Concurrent Contains(k)

- Traverse downwards in the tree



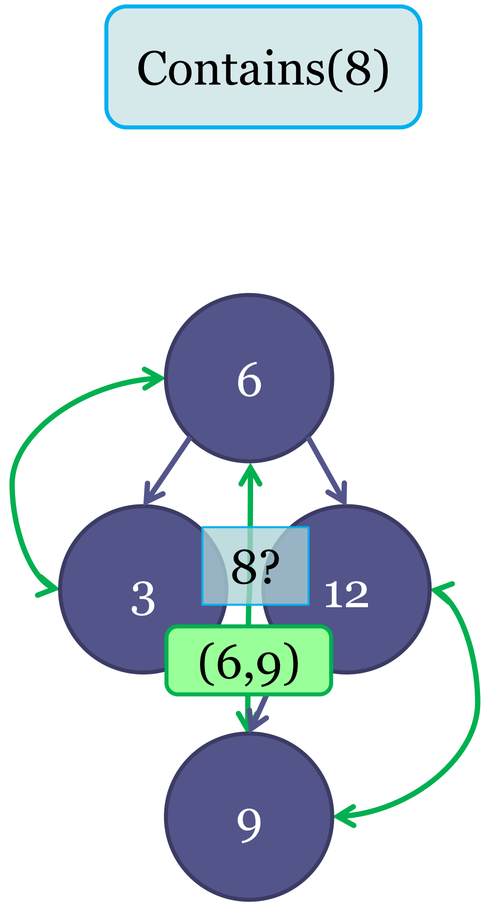
The Concurrent Contains(k)

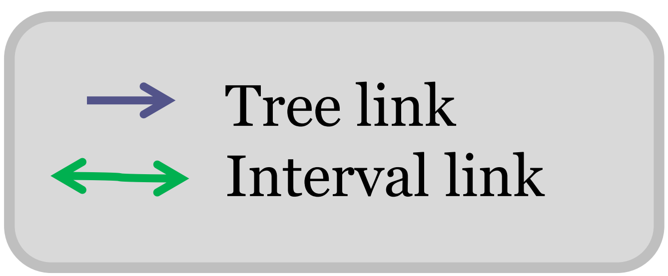
- Traverse downwards in the tree
- If reached to an end of a path, traverse via the ordering layout to find (p, s) such that $k \in [p, s]$
  - Return false iff $k \neq p, s$



The Concurrent Contains(k)

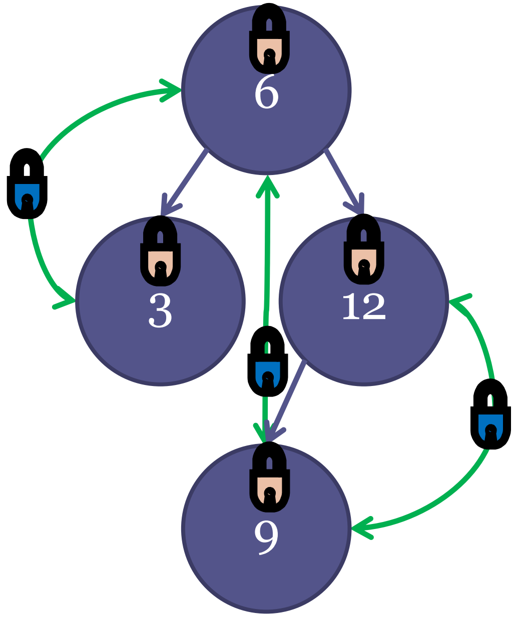
- Traverse downwards in the tree
- If reached to an end of a path, traverse via the ordering layout to find (p, s) such that $k \in [p, s]$
  - Return false iff $k \neq p, s$
- This operation is non-blocking

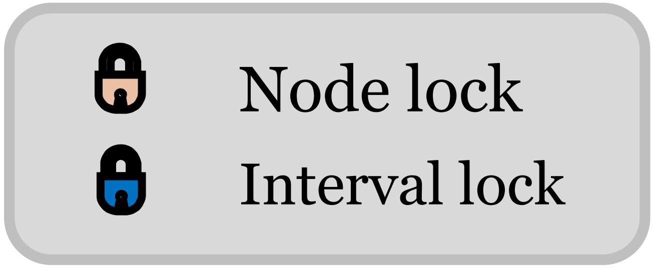




Insert and Remove Operations

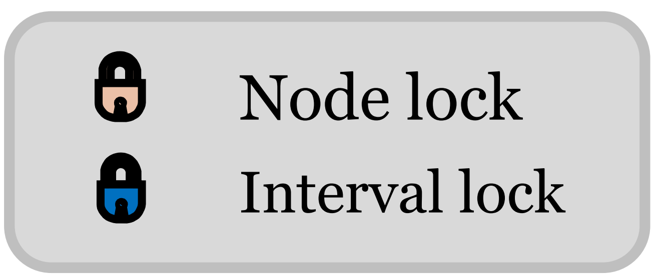
- The synchronization is based on locks
- The operations lock
  - The relevant nodes in the tree
  - The relevant intervals

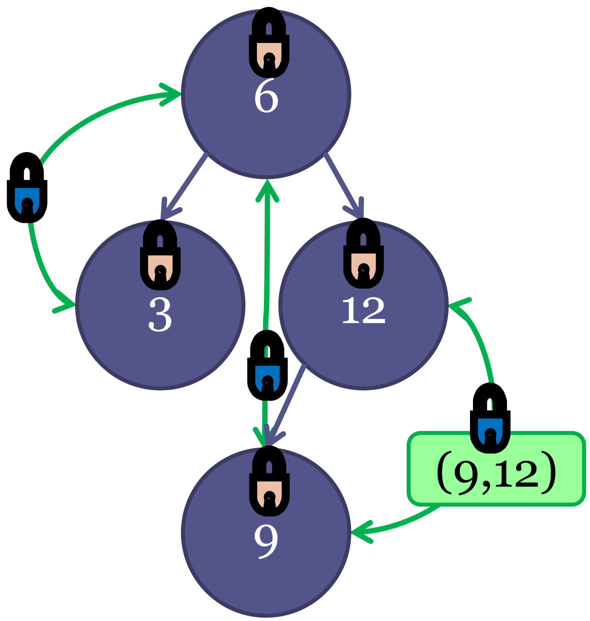




Insert and Remove Operations

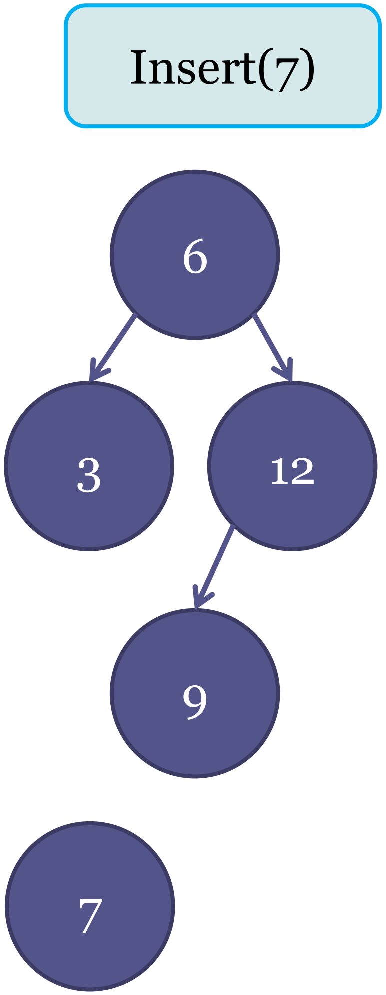
- The synchronization is based on locks
- The operations lock
  - The relevant nodes in the tree
  - The relevant intervals





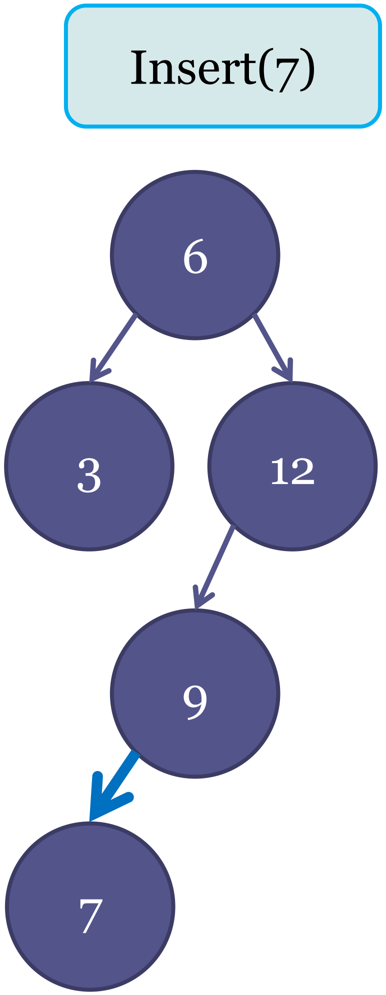
The Sequential Insert(k)

- Traverse downwards in the tree
- If k was found: cannot insert
- Otherwise, let l be the node at the end of a path



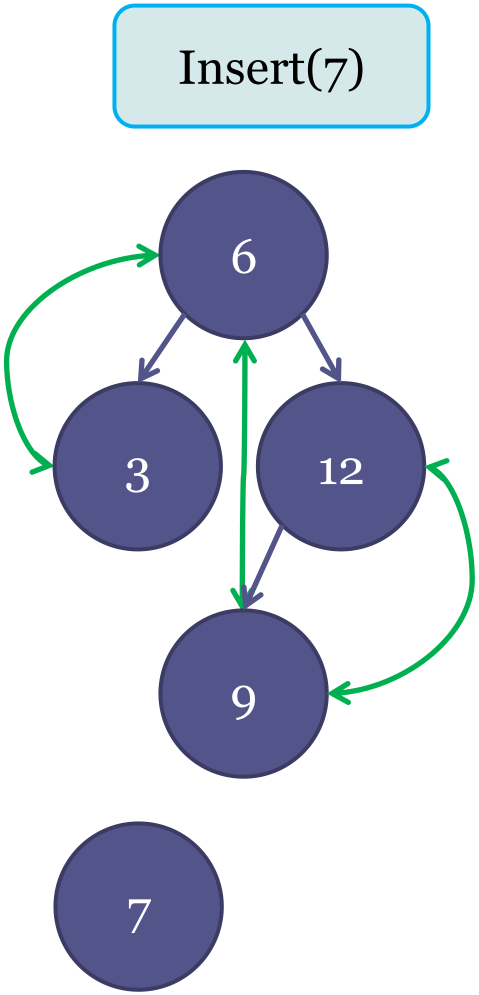
The Sequential Insert(k)

- Traverse downwards in the tree
- If k was found: cannot insert
- Otherwise, let l be the node at the end of a path
  - Connect l to the new node



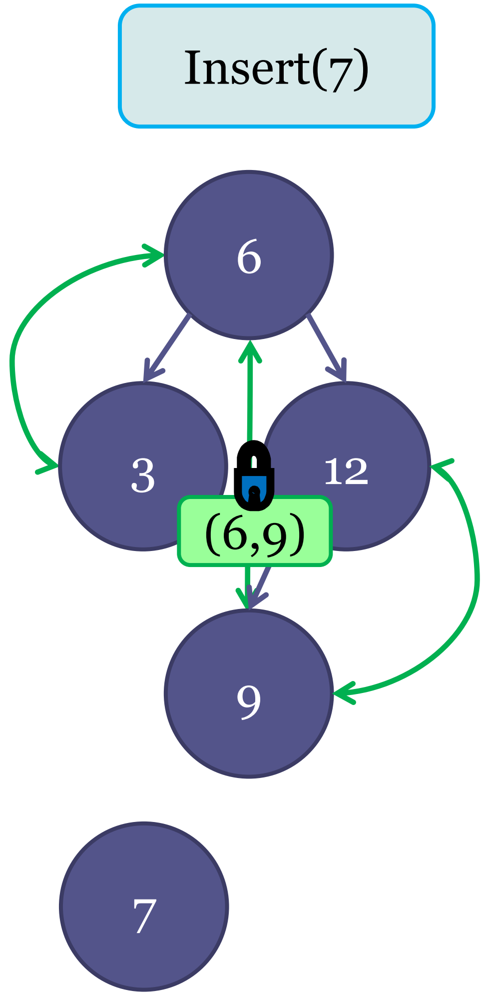
The Concurrent Insert(k)

- Traverse downwards in the tree
- If k was found: cannot insert
- Otherwise, let l be the node at the end of a path



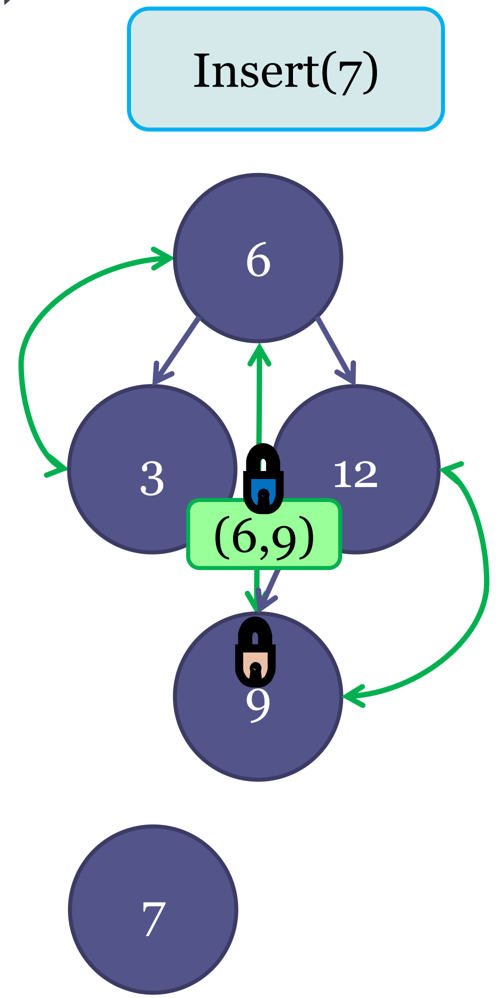
The Concurrent Insert(k)

- Traverse downwards in the tree
- If k was found: cannot insert
- Otherwise, let l be the node at the end of a path
- Lock relevant interval
  - If $k \leq l$: lock (l's pred, l)



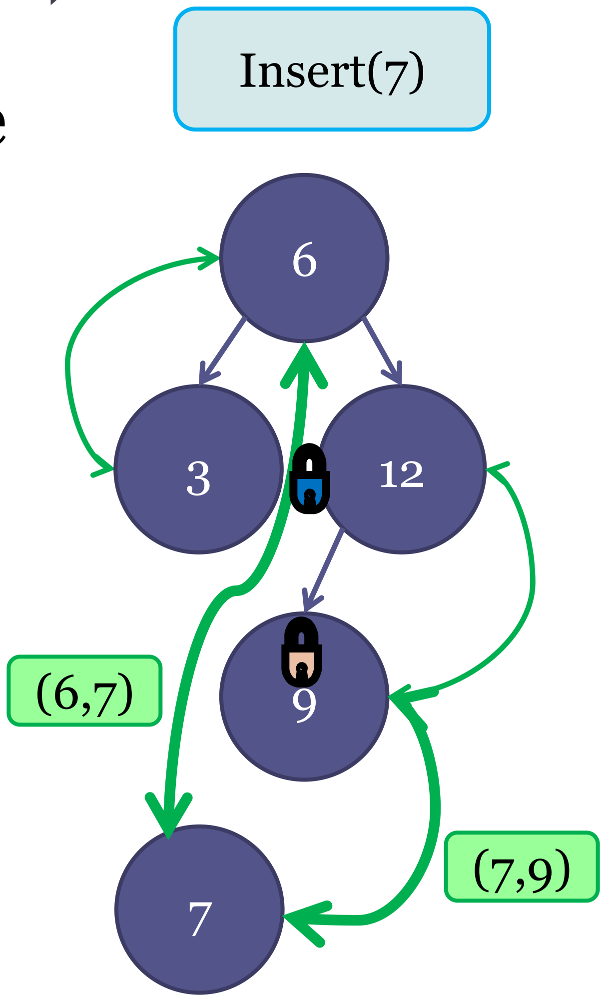
The Concurrent Insert(k)

- Traverse downwards in the tree
- If k was found: cannot insert
- Otherwise, let l be the node at the end of a path
- Lock relevant interval
  - If $k \leq l$: lock (l's pred, l)
- Lock l



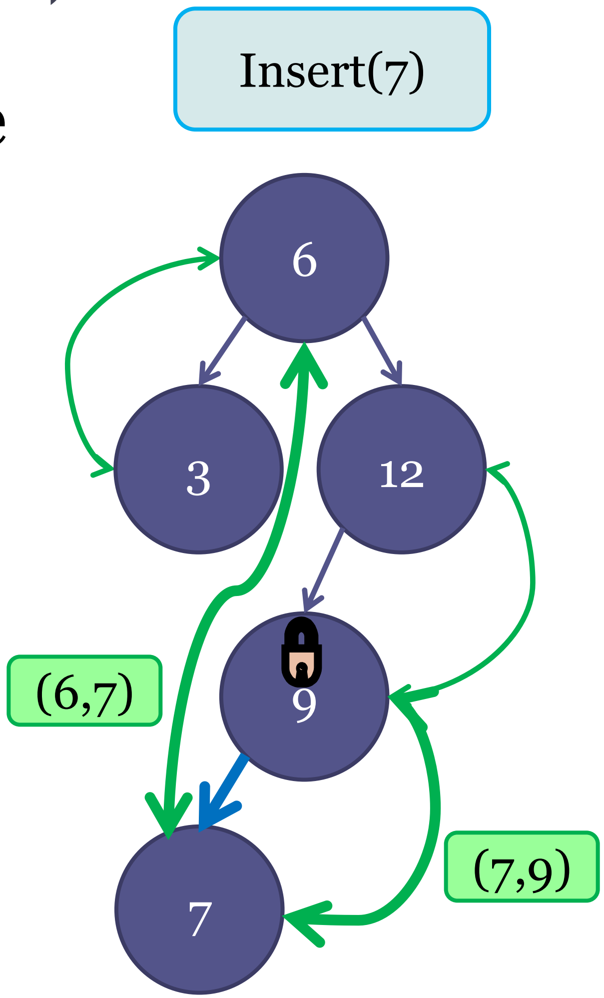
The Concurrent Insert(k)

- Traverse downwards in the tree
- If k was found: cannot insert
- Otherwise, let l be the node at the end of a path
- Lock relevant interval
  - If $k \leq l$: lock (l's pred, l)
- Lock l
- Update predecessor-successor



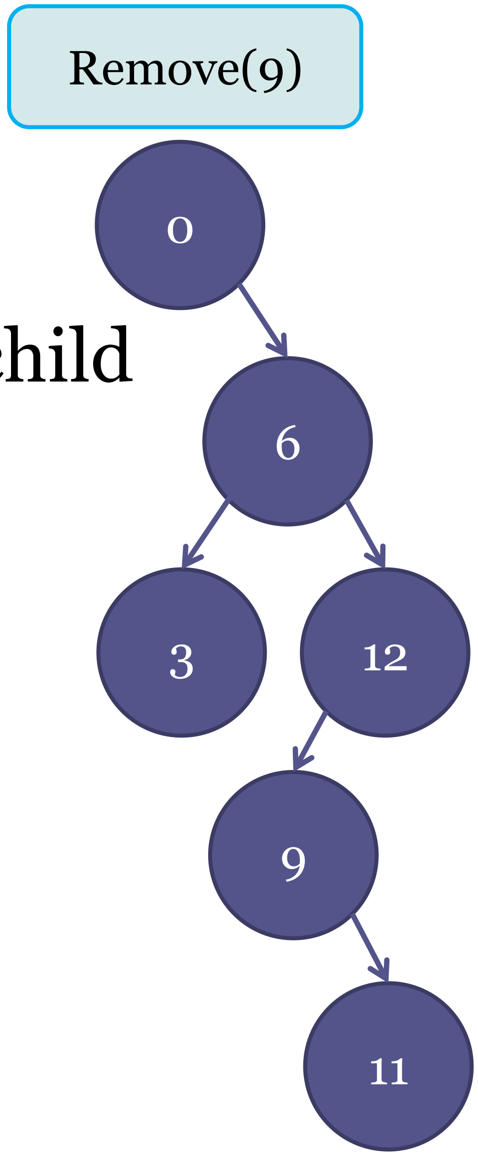
The Concurrent Insert(k)

- Traverse downwards in the tree
- If k was found: cannot insert
- Otherwise, let l be the node at the end of a path
- Lock relevant interval
  - If $k \leq l$: lock (l's pred, l)
- Lock l
- Update predecessor-successor
- Connect l to the new node



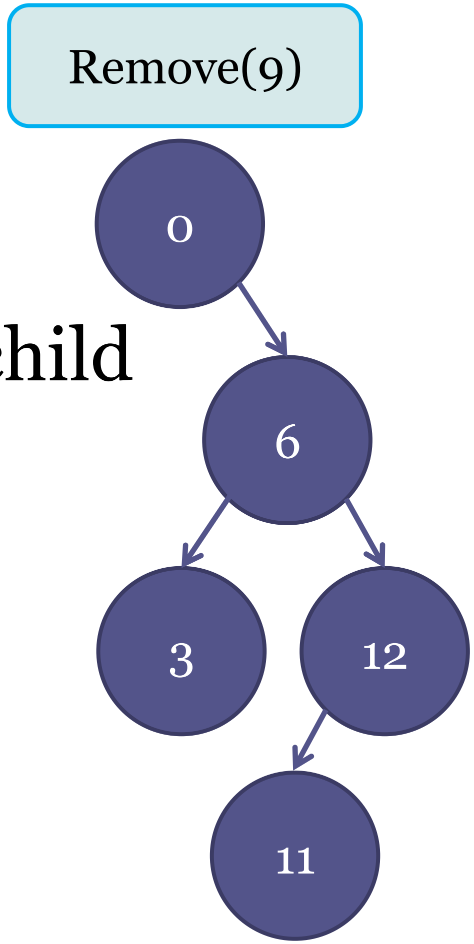
The Sequential Remove(k)

- Traverse downwards in the tree
- If the node to remove has at most 1 child
  - Set its parent to point to its child (may be null)



The Sequential Remove(k)

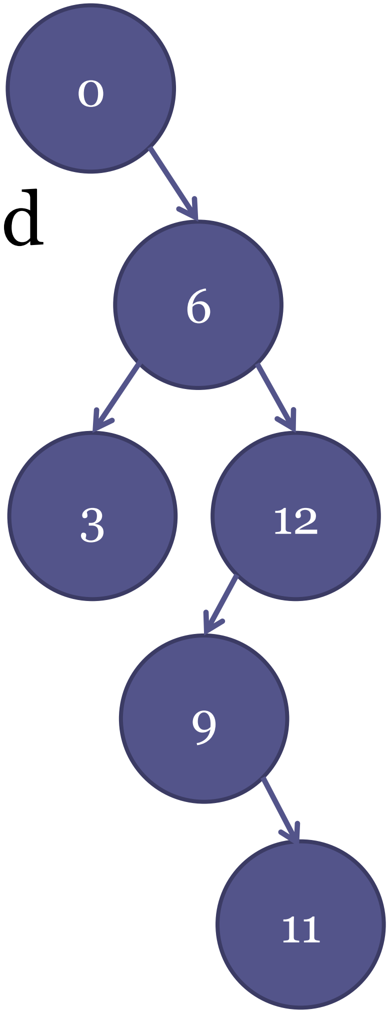
- Traverse downwards in the tree
- If the node to remove has at most 1 child
  - Set its parent to point to its child (may be null)



The Sequential Remove(k)

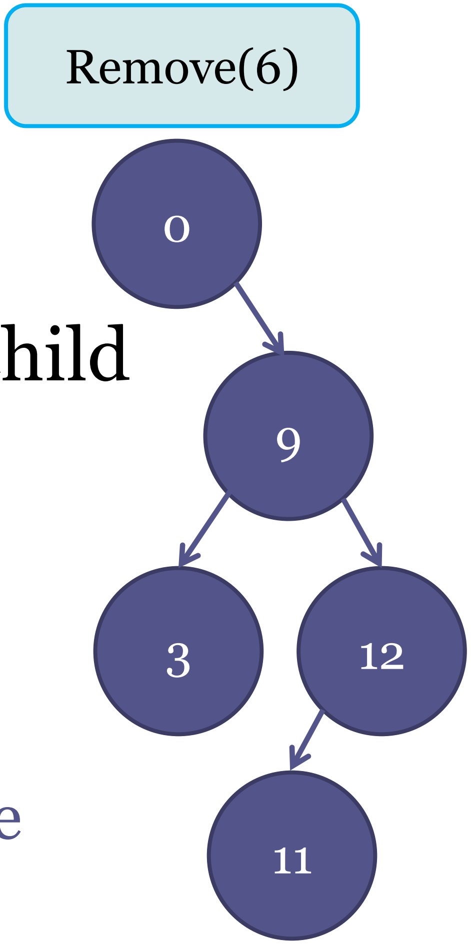
Remove(6)

- Traverse downwards in the tree
- If the node to remove has at most 1 child
  - Set its parent to point to its child (may be null)
- If the node to remove has 2 children
  - Search for its successor, s
    - The left most node in the right sub-tree
  - Relocate s to its location



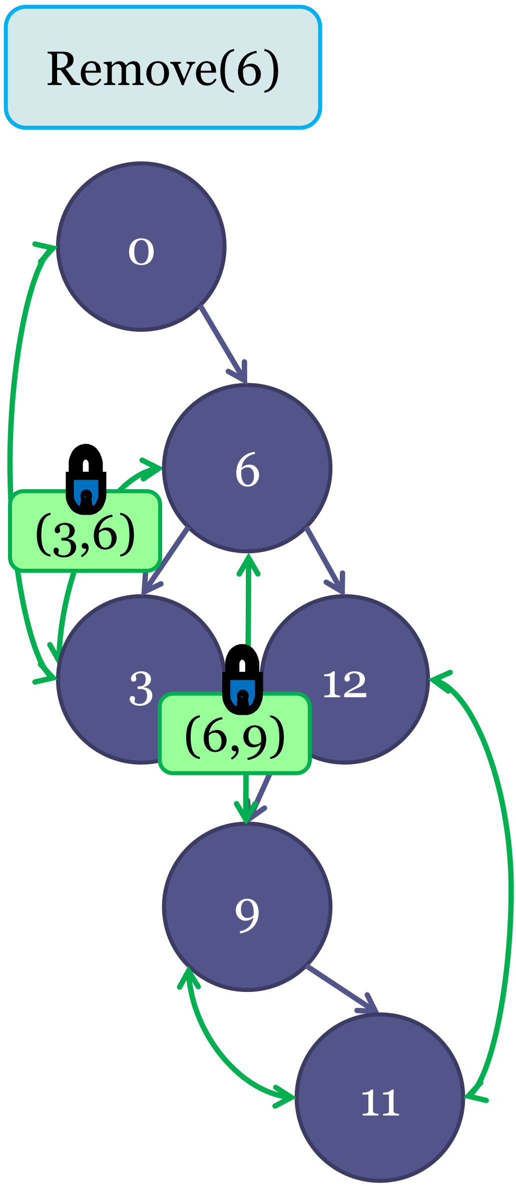
The Sequential Remove(k)

- Traverse downwards in the tree
- If the node to remove has at most 1 child
  - Set its parent to point to its child (may be null)
- If the node to remove has 2 children
  - Search for its successor, s
    - The left most node in the right sub-tree
  - Relocate s to its location



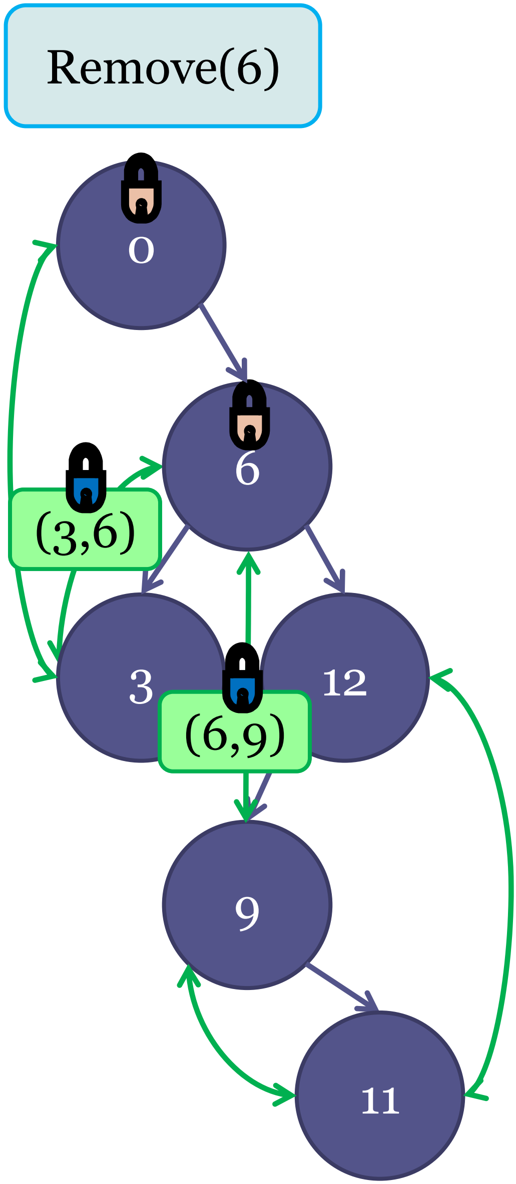
The Concurrent Remove(k)

- Let n be the node to remove
- Lock (n's pred, n)
- Lock (n, n's succ)



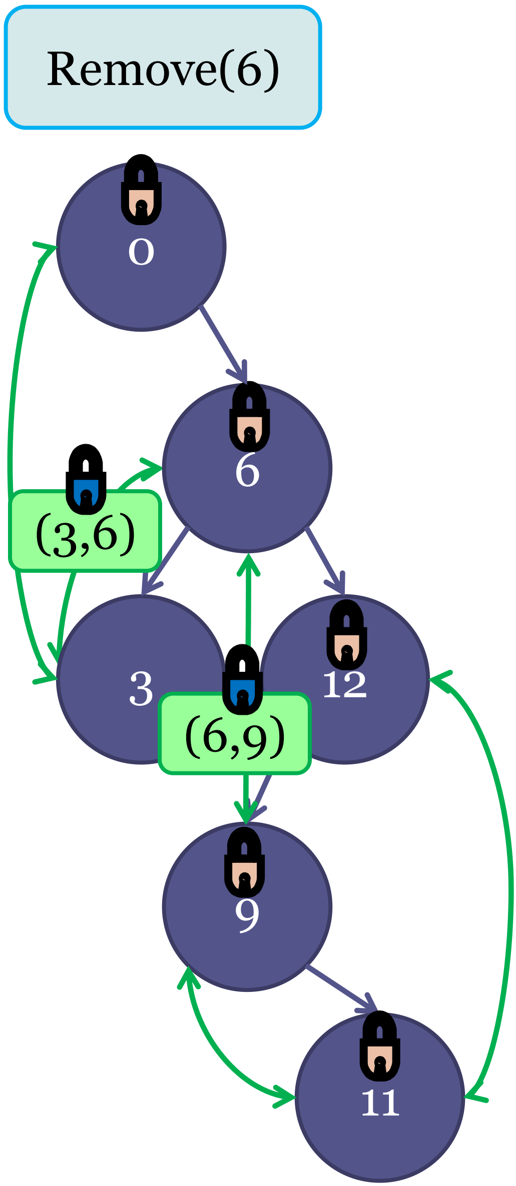
The Concurrent Remove(k)

- Let n be the node to remove
- Lock (n’s pred, n)
- Lock (n, n’s succ)
- Lock n and its parent



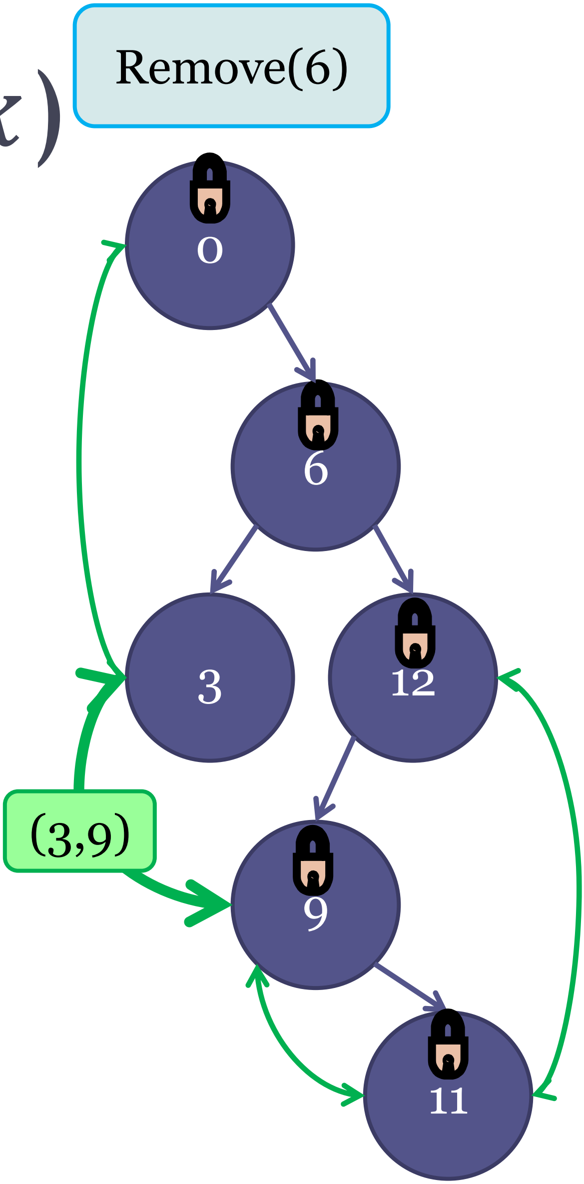
The Concurrent Remove(k)

- Let n be the node to remove
- Lock (n's pred, n)
- Lock (n, n's succ)
- Lock n and its parent
- If n has 2 children
  - Lock n's successor, its parent and child



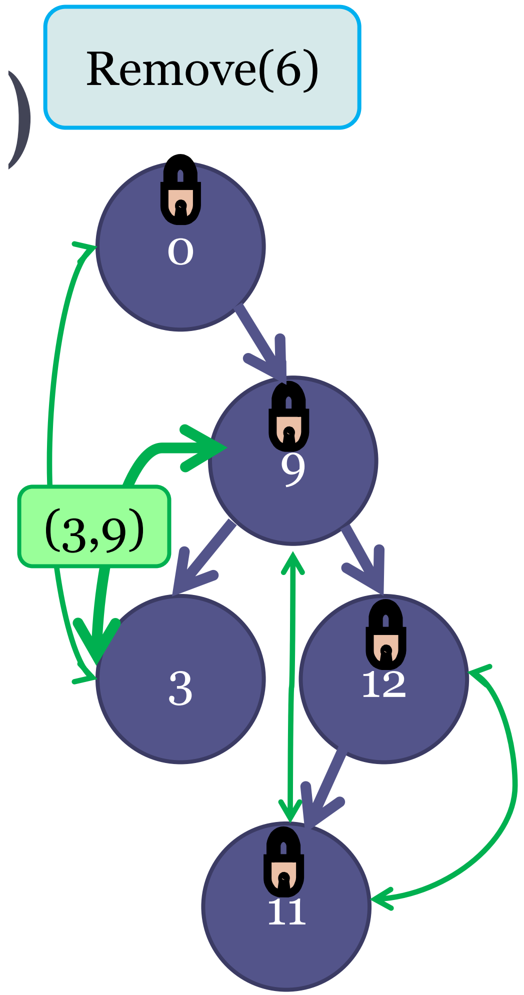
The Concurrent Remove(k)

- Let n be the node to remove
- Lock (n's pred, n)
- Lock (n, n's succ)
- Lock n and its parent
- If n has 2 children
  - Lock n's successor, its parent and child
  - Update predecessor-successor



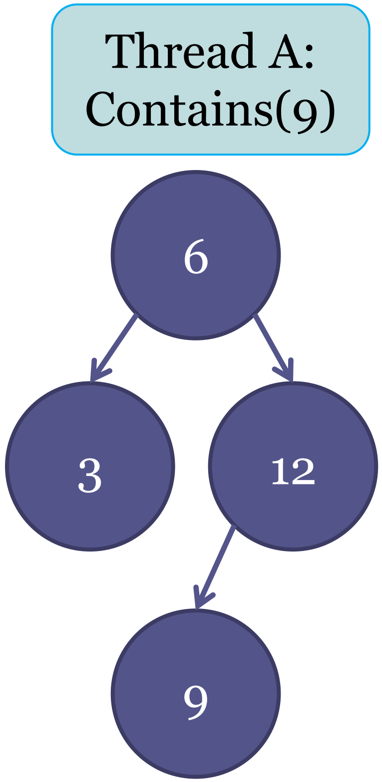
The Concurrent Remove(k)

- Let n be the node to remove
- Lock (n's pred, n)
- Lock (n, n's succ)
- Lock n and its parent
- If n has 2 children
  - Lock n's successor, its parent and child
  - Update predecessor-successor
  - Relocate the successor to n's location



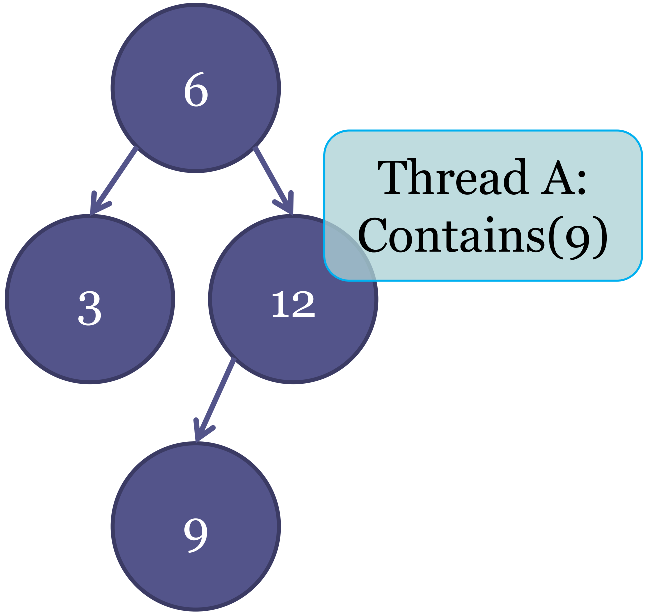
BST: Challenge with Concurrency

1. Thread A searches for 9



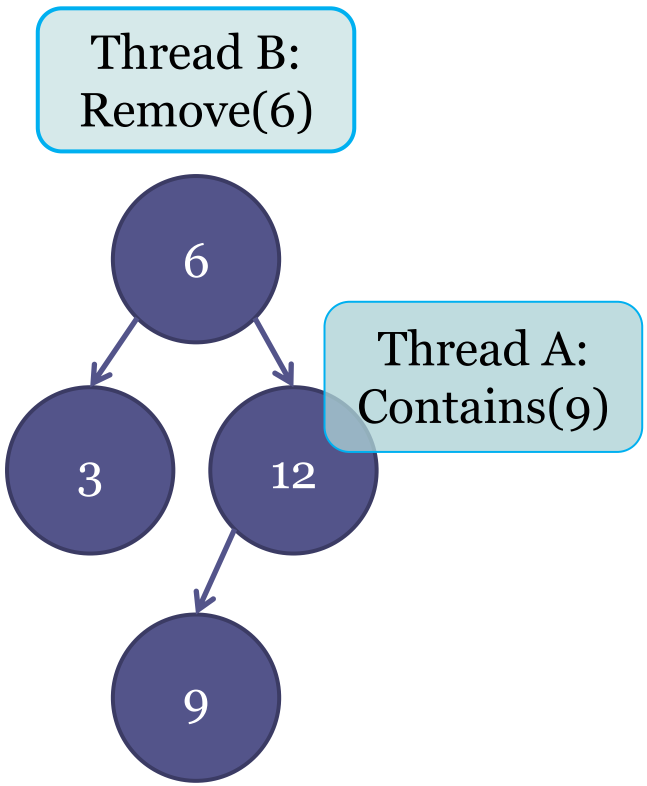
BST: Challenge with Concurrency

1. Thread A searches for 9 and pauses



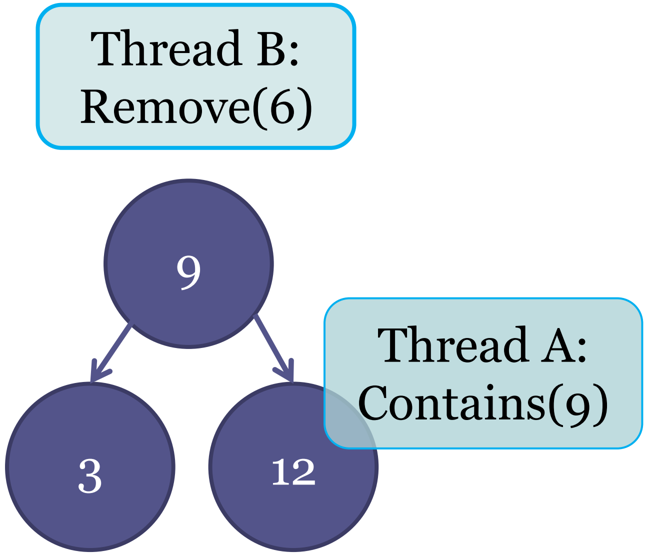
BST: Challenge with Concurrency

1. Thread A searches for 9 and pauses
2. Thread B removes 6



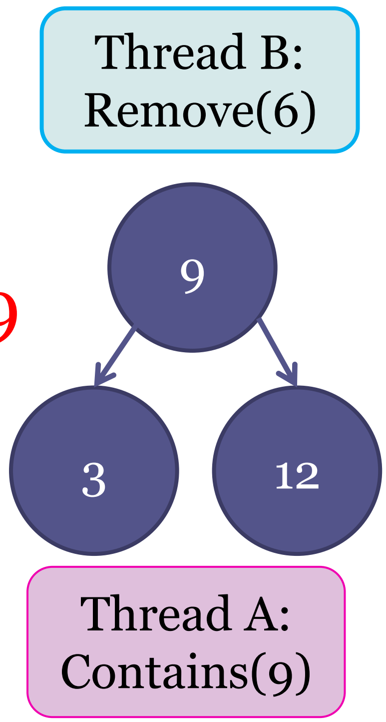
BST: Challenge with Concurrency

1. Thread A searches for 9 and pauses
2. Thread B removes 6



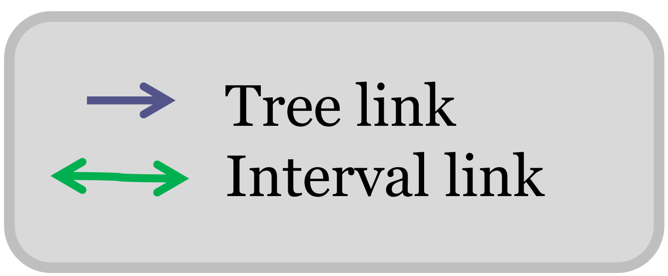
BST: Challenge with Concurrency

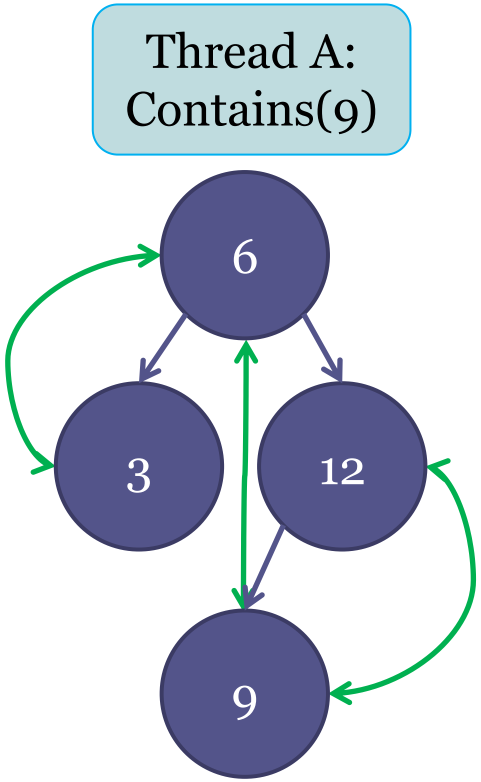
1. Thread A searches for 9 and pauses
2. Thread B removes 6
3. Thread A resumes and misses 9



Solution

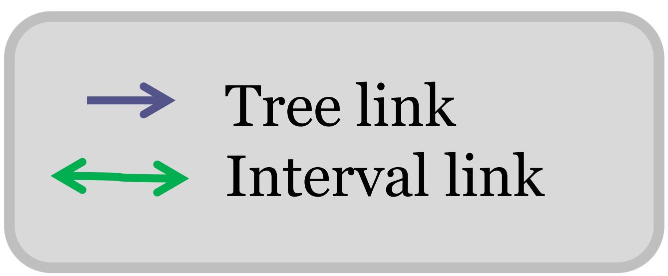
- Consult the logical ordering layout before making final decisions

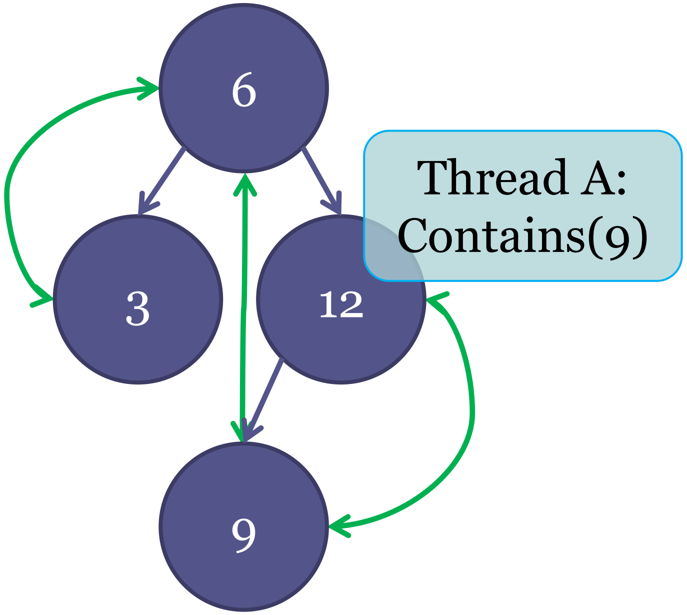




Solution

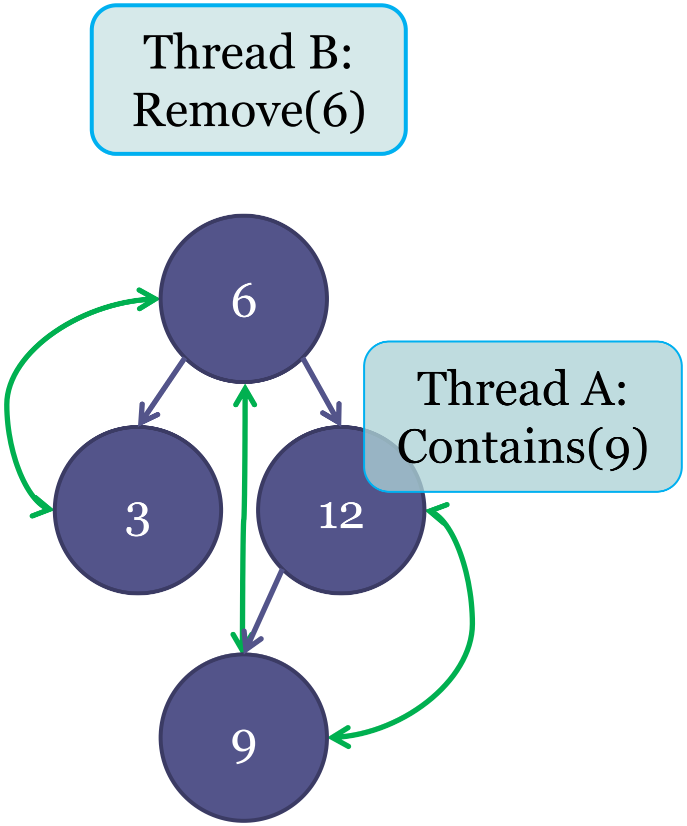
- Consult the logical ordering layout before making final decisions





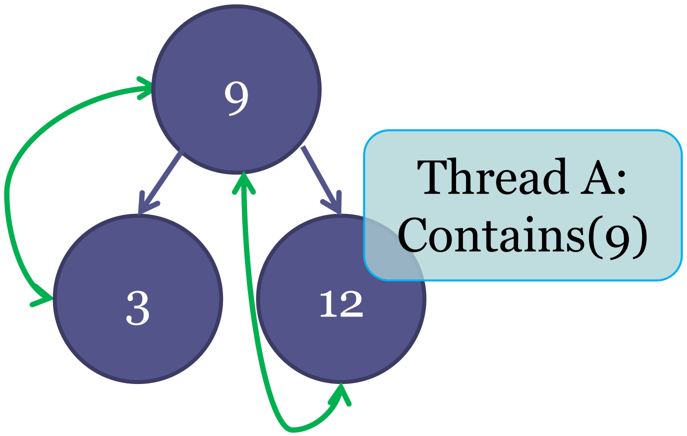
Solution

- Consult the logical ordering layout before making final decisions



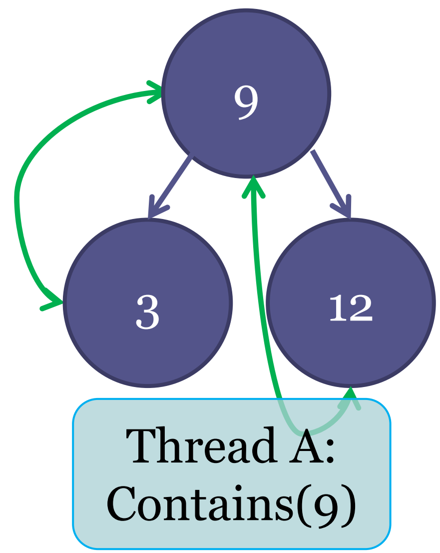
Solution

- Consult the logical ordering layout before making final decisions



Solution

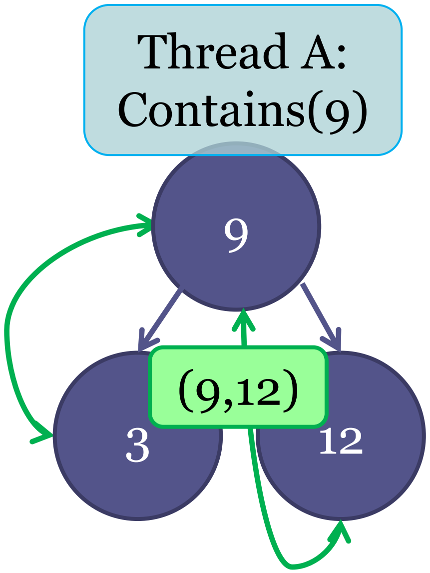
- Consult the logical ordering layout before making final decisions



Tree link
Interval link

Solution

- Consult the logical ordering layout before making final decisions



From BST to AVL Tree

- After each update, apply balancing operations
- Balancing operations relocate nodes in the tree
  - Requires only node locks
- Concurrent threads cannot miss keys, since they consult the logical ordering layout

Implementation

- We implemented our BST and AVL tree in Java
- We compared to state-of-the-art algorithms

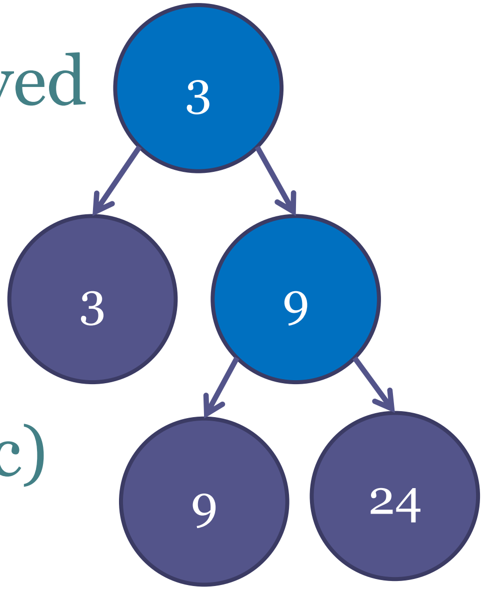
Comparison to Existing Algorithms

- Partially external trees
  - Internal nodes are only marked as removed
    - A follow-up insert can revive them

- Locked-based, partially external trees
  - Bronson et al., PPoPP 2010 (BCCO)
  - A variation of our work (Our LR-AVL)

Comparison to Existing Algorithms

- External tree
  - Elements are kept only in the leaves
  - Inner nodes serve as routing nodes
  - Only leaves can be asked to be removed
  - Traversal paths are typically longer



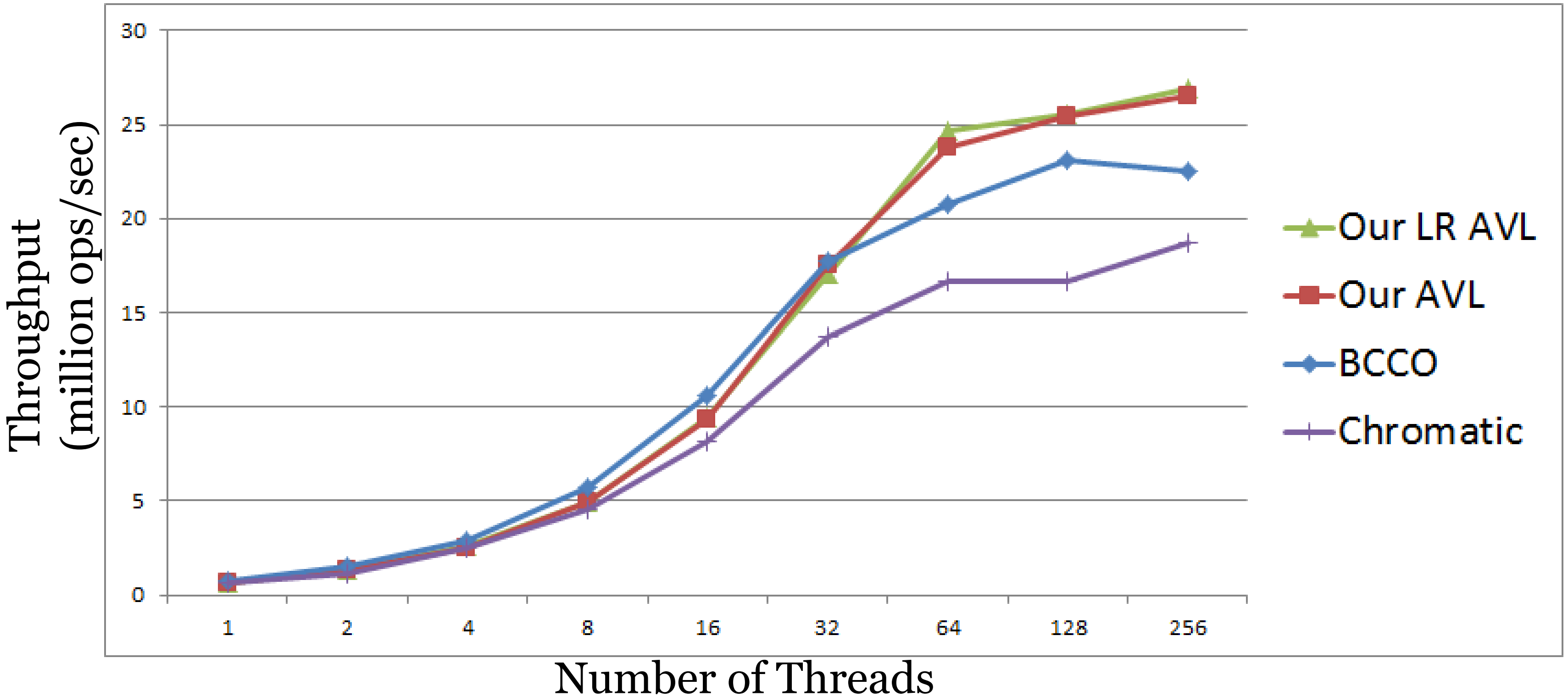
- Non-Blocking external tree
  - Brown et al., PPoPP 2014 (Chromatic)

Evaluation

- A 4-socket AMD Opteron, with 64 h/w threads
- Threads randomly chose operation type and key
  - Different workloads for the operation type
    - 100% contains, 0% insert, 0% remove
    - 70% contains, 20% insert, 10% remove
  - Different key ranges
    - $2 \cdot 10^6, 2 \cdot 10^5$

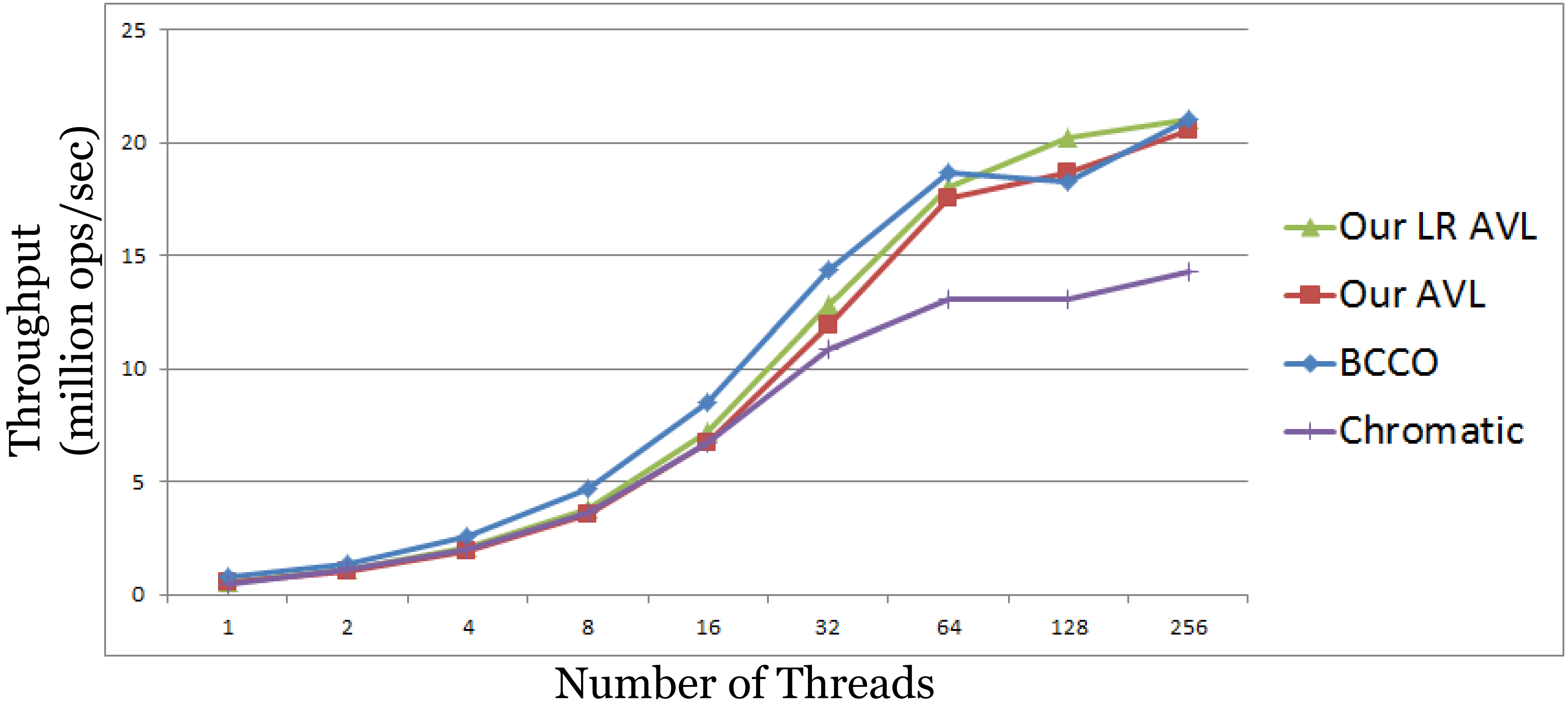
Evaluation

- 100% contains, 0% insert, 0% remove
- Key range: $2 \cdot 10^6$



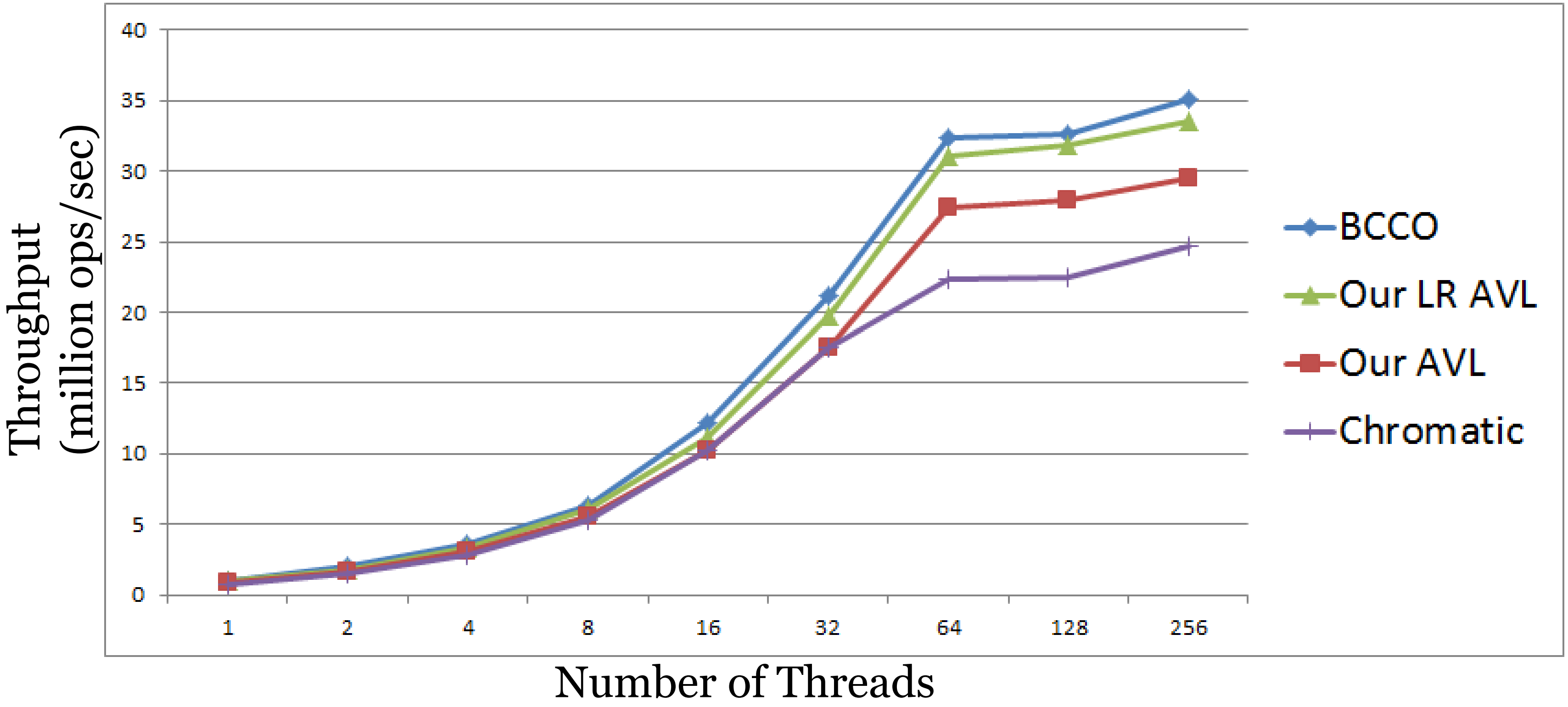
Evaluation

- 70% contains, 20% insert, 10% remove
- Key range: $2 \cdot 10^6$



Evaluation

- 70% contains, 20% insert, 10% remove
- Key range: $2 \cdot 10^5$



Summary

- We presented a new practical concurrent BST
  - Non-blocking search
  - Balanced
  - Efficient
  - Simple
- Our main insight
  - Maintain explicitly the intervals

Thank you!