Practical Concurrent Traversals in Search Trees

Tyler Townsend & Robert Bland

Authors:
Dana Draschler-Cohen,
Martin Vechev,
Eran Yahav

Overview

- PaVT AVL & BST Data Structures
- Design of Data Structures
- GCC STM Data Structure
- Experimental Results
- Conclusion

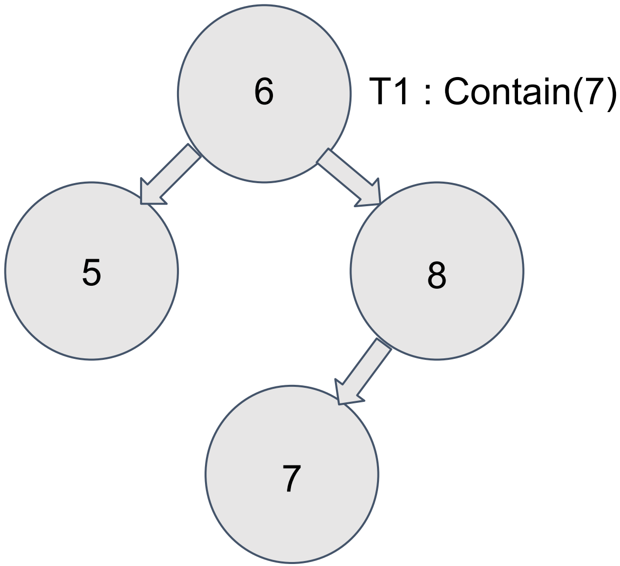
PaVT AVL & BST Data Structure

- Purpose of the paper is to propose a necessary and sufficient condition to validate traversals in Search Trees, PaVT.
- The authors then showed how to create a lock-free membership test that can be applied to any search tree using PaVT.
- The authors demonstrated PaVT in Binary Search Trees and AVL Trees.

UCF

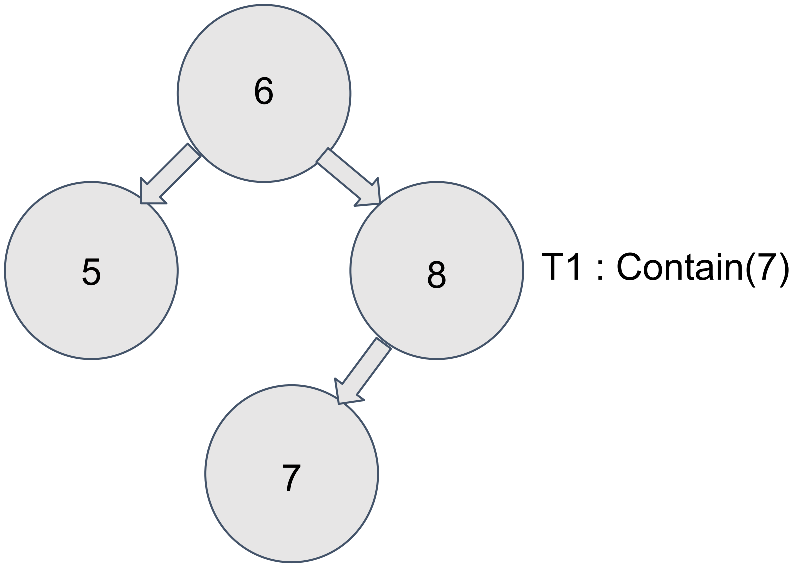
PaVT Motivation

- T1: Calls Contain(7)



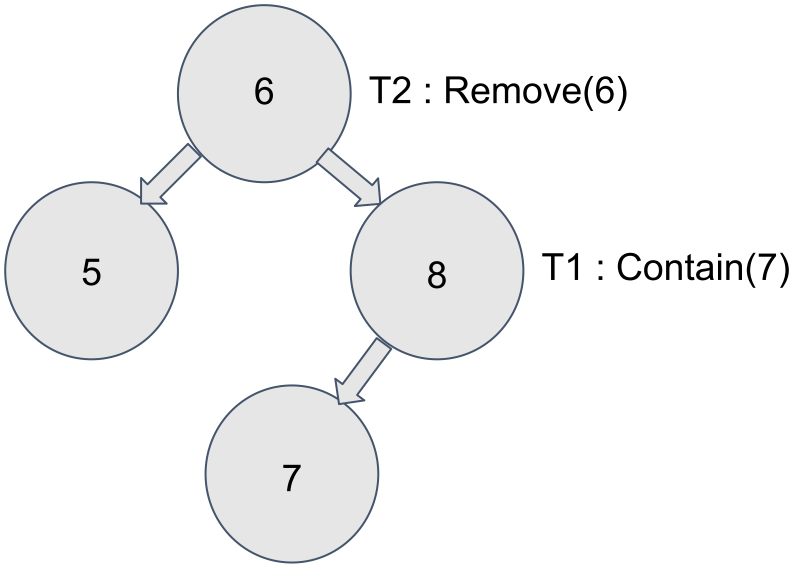
PaVT Motivation

- T1: Calls Contain(7)
- T1: Gets to node 8



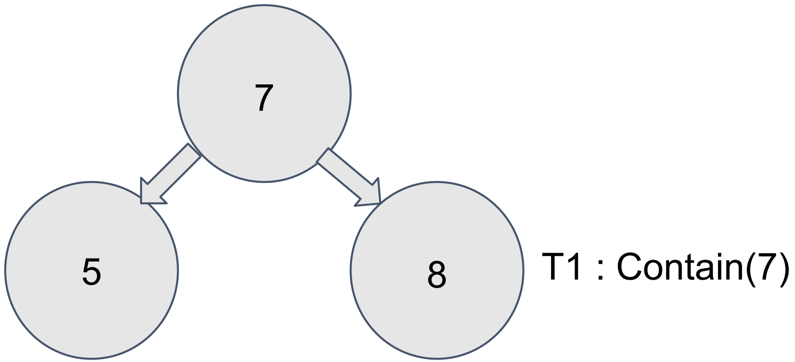
PaVT Motivation

- T1: Calls Contain(7)
- T1: Gets to node 8
- T2:Calls Remove(6)



PaVT Motivation

- T1: Calls Contain(7)
- T1: Gets to node 8
- T2:Calls Remove(6)
- T2: Removes 6 and replaces with 7
- T1: Returns false



Contributions over State-of-the-art

- PaVT (Path Validation in Search Trees)
  - Provides a condition to which a target node is or not in the tree.
  - Applies to any Search Tree
    - BST, Ternary Trees, 2-D Trees, Tries, ect..
- Observe a limited number of nodes that are the succinct path snapshot (SPS) of the path leading to the key.
- Concurrent Updates may modify the path and the traversal may still complete successfully if it has found the correct snapshot.

UCF

PaVT with SPS in BST's

- Predecessor, Successor Pairs
- Similar to authors previous work of BST's with Logical Ordering
- When validating the path, check that the key may be reach from this path.
- Depends on whether the the last node traversed was left or right of parent.

```
if (parentIsLarger && key <=node.Predecessor or
   !parentIsLarger && key >=node.Successor) {

        restart;
}
```

UCF

Design of Data Structure

- Authors provide 4 Algorithms in Pseudocode
  - TraverseNLock(Traverse)
  - Insert
  - Remove
  - Contains (Implied from Traverse)
  - UpdateSnaps
- Up to Us
  - Tree Structure
  - AVL Operations
  - Snap Shots (mentioned in previous work)

Node and Tree

```
class Node {
        int data;
        Lock;
        Node left;
        Node right;
        Node parent;
        atomic<Node> leftSnap;
        atomic<Node> rightSnap;
        int height;
        bool mark;
}
```

```
class Tree {

        Node root;
        atomic<Node> minSentinel;
        atomic<Node> maxSentinel;

        traverse(Node, data);
        insert(data)
        remove(data)
        contains(data)
        rebalance(data)
}
```

Locking Protocol

Each node has a lock and n1's lock is acquired before n2's lock if

1. n2 is reachable from n1 or
2. n1 and n2 are reachable via fi and fj, respectively, from their lowest common ancestor and $i < j$

Lock from top to bottom, left to right ordering.
Locks (parents) acquired out of order are done optimistically and call for restart if failed

- Protocol is Livelock and Deadlock free since locks are acquired in same order by all threads
- Lowest node in tree is guaranteed to succeed in acquiring locks

Snapshots

- Snapshots are updated by Traversing from mutated Node up the path
- Can be modified directly in the case of BST's
- Modified and updated within insert/remove directly
- Correctness
  - Must acquire locks before mutating path
- Linearizabilty
  - When snapshots are read.
  - Linearized just before linearization point of insertion/removal
  - This represents a consistent path and since the snapshot has not been updated yet this implies the other thread possibly modifying it has not completed its operation.

Traverse

- Used by Insert and Remove
- Contains is exactly the same as traverse except we do not use locks.
- Correctness follows from PaVT and Linearizes when node is locked and returns

```
Traverse(node, data) :
    n = node
    f = nextField(node, data)
    while(n.f != null) {
        n = n.f
        if node is found
            if marked then restart
            lock n and return n
    }
    check if data is in  S(n,f)
    if true restart
    else lock n and return n
```

Rebalance(node)

- Continually traverse up to root from node
- Acquire locks of parent, node, child, grandchild
- Check balance factors of node, child
- Perform single/double rotations when necessary
- If current height of node is unchanged and balance factor is satisfied then we are done
  - Based off relaxed AVL approach
- Linearizes after last rotation per traversal
- Must be very careful with new locking order!

Insert(7)

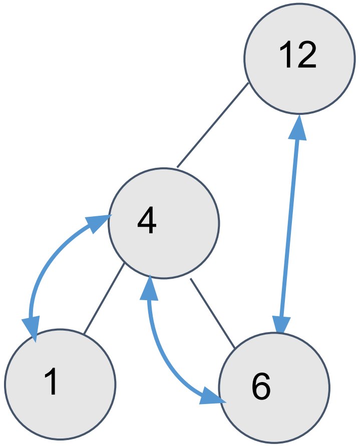
Logical Ordering: 1, 4, 6, 12

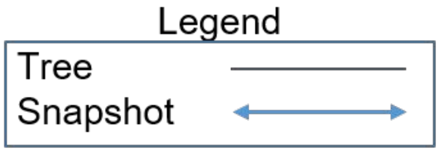
Insert(data) :
  node = traverse(data)
  check if insert to null pointer
  newNode = new Node(data)

  updateParent(node, newNode)
  setField(node, newNode)

  Copy snaps of node into newNode
  UpdateSnaps of node and oldSnap

  Unlock(node)





Insert(7)

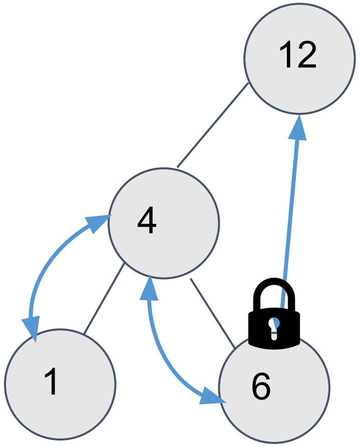
Logical Ordering: 1, 4, 6, 12

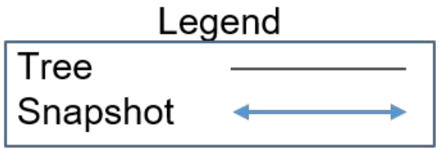
Insert(data) :
  node = traverse(data)
  check if insert to null pointer
  newNode = new Node(data)

  updateParent(node, newNode)
  setField(node, newNode)

  Copy snaps of node into newNode
  UpdateSnaps of node and oldSnap

  Unlock(node)





Insert(7)

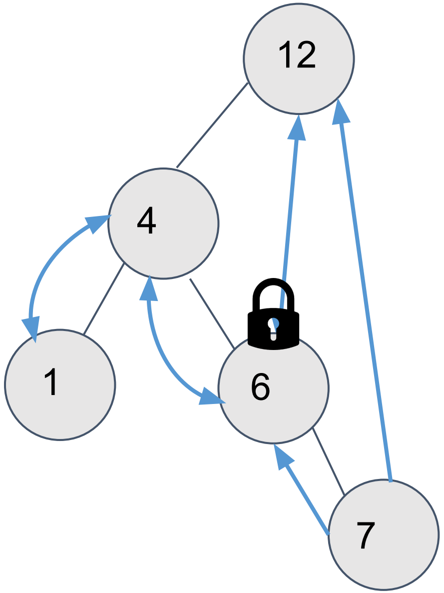
Logical Ordering: 1, 4, 6, 12

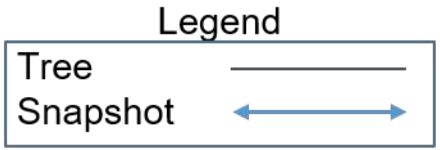
Insert(data) :
  node = traverse(data)
  check if insert to null pointer
  newNode = new Node(data)

  updateParent(node, newNode)
  setField(node, newNode)

  Copy snaps of node into newNode
  UpdateSnaps of node and oldSnap

  Unlock(node)





Insert(7)

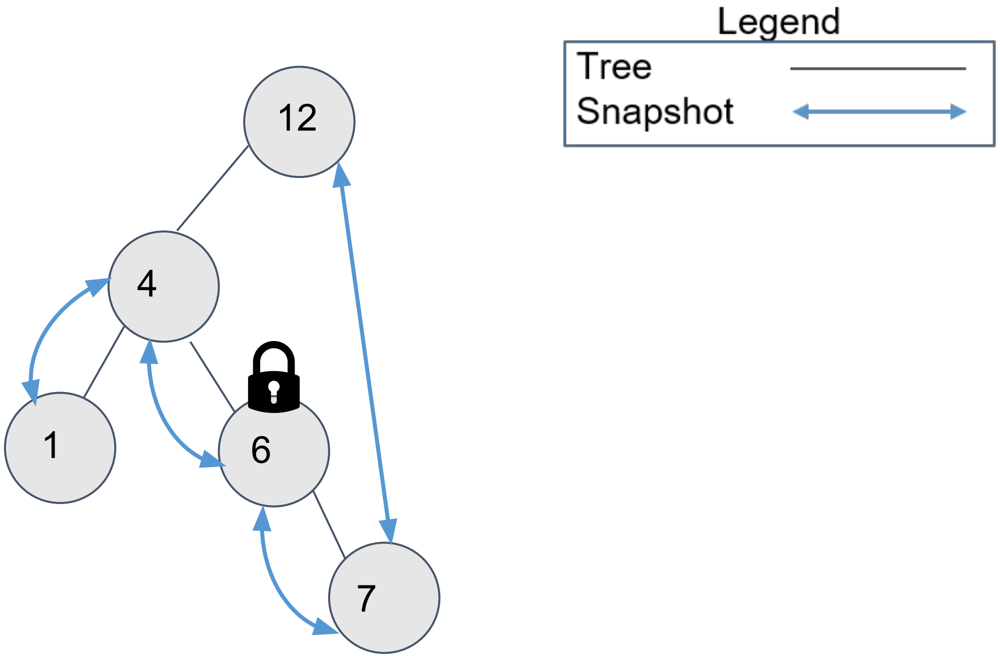
Logical Ordering: 1, 4, 6, 7, 12

Insert(data) :
  node = traverse(data)
  check if insert to null pointer
  newNode = new Node(data)

  updateParent(node, newNode)
  setField(node, newNode)

  Copy snaps of node into newNode
  UpdateSnaps of node and oldSnap

  Unlock(node)



Insert(Data)

- AVL implementation attempts to rebalance after unlocking if it is needed
- Insert linearizes when the node is added to the parent

Remove(Data)

- "Five" possible Scenarios for a node being removed
  a. Node has no children
  b. Node has 1 leaf
  c. Node has 2 children (let r = node right child)
     - r has no left child
     - successor's parent is r
     - successor's parent is not r
- Biggest Challenge is preserving locking/unlocking order!

UCF

Remove(7)

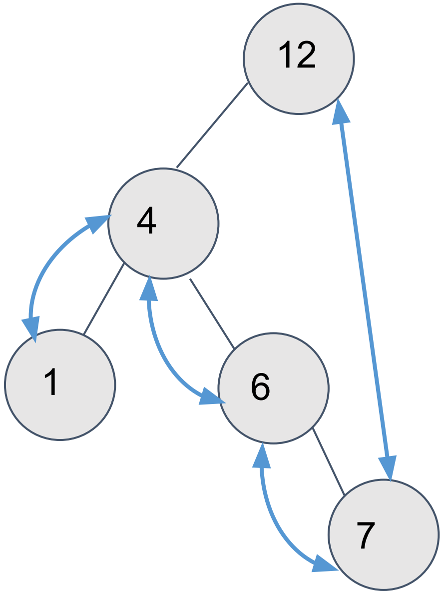
Logical Ordering: 1, 4, 6, 7, 12

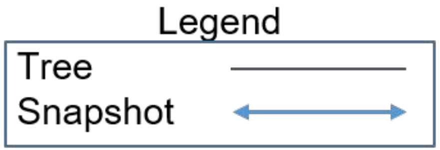
Remove(data) :
  node = traverse(data)
  parent.try_lock()
  mark(node)
  setField(Parent, null)

  updateParent(parent, rightSnapshot)

  Change right snap of parent to childs successor

  UnlockAll()





UCF

Remove(7)

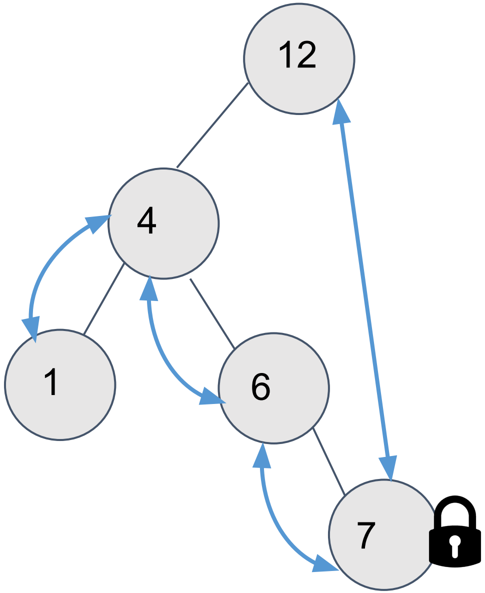
Logical Ordering: 1, 4, 6, 7, 12

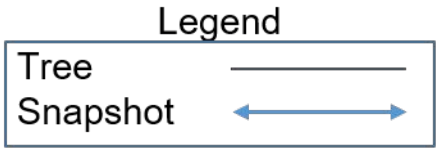
Remove(data) :
  node = traverse(data)
  parent.try_lock()
  mark(node)
  setField(Parent, null)

  updateParent(parent, rightSnapshot)

  Change right snap of parent to childs successor

  UnlockAll()





Remove(7)

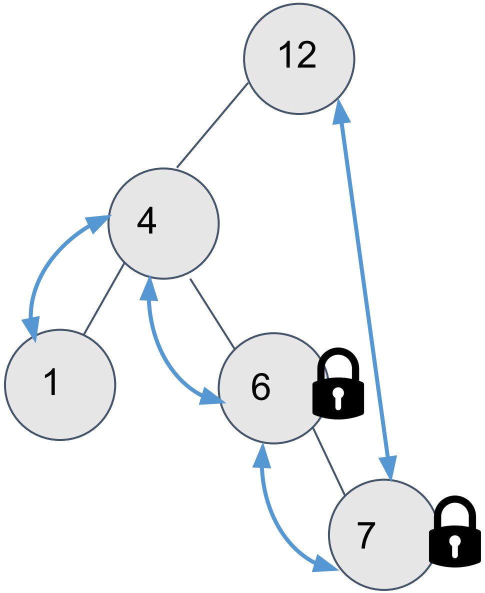
Logical Ordering: 1, 4, 6, 7, 12

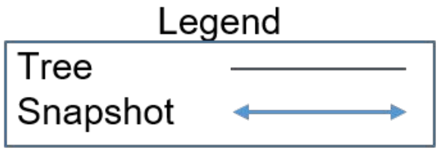
Remove(data) :
  node = traverse(data)
  parent.try_lock()
  mark(node)
  setField(Parent, null)

  updateParent(parent, rightSnapshot)

  Change right snap of parent to childs successor

  UnlockAll()





Remove(7)

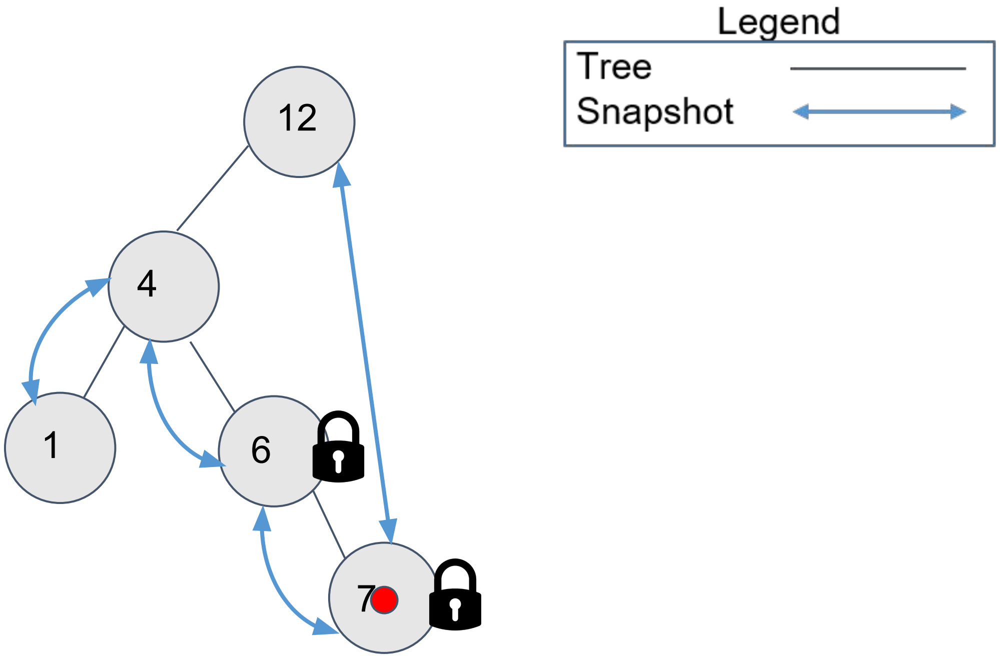
Logical Ordering: 1, 4, 6, 7, 12

Remove(data) :
  node = traverse(data)
  parent.try_lock()
  mark(node)
  setField(Parent, null)

  updateParent(parent, rightSnapshot)

  Change right snap of parent to childs successor

  UnlockAll()



Remove(7)

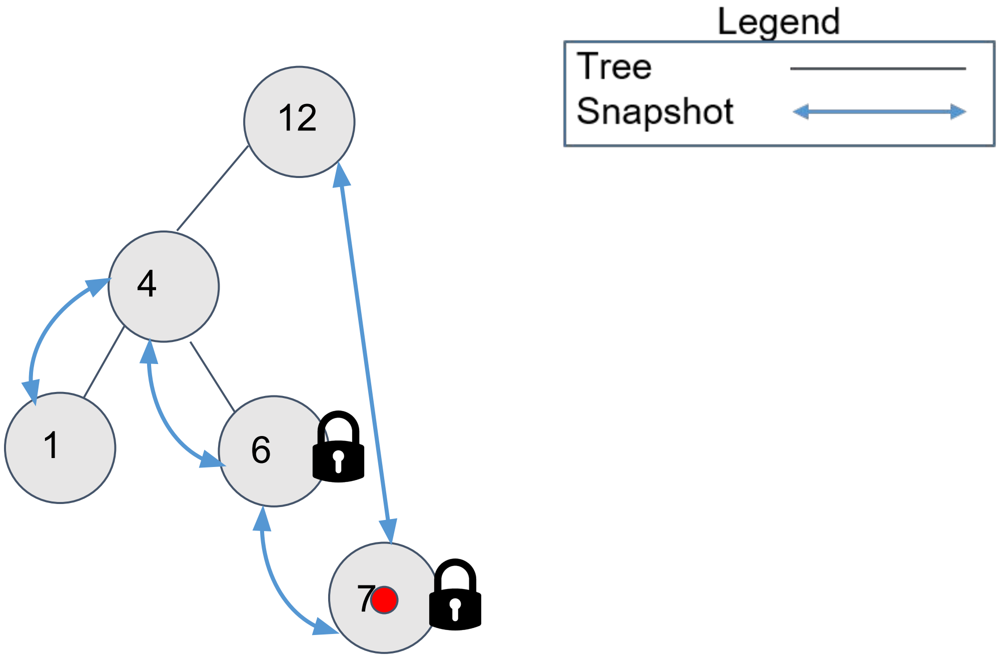
Logical Ordering: 1, 4, 6, 7, 12

Remove(data) :
  node = traverse(data)
  parent.try_lock()
  mark(node)
  setField(Parent, null)

  updateParent(parent, rightSnapshot)

  Change right snap of parent to childs successor

  UnlockAll()



UCF

Remove(7)

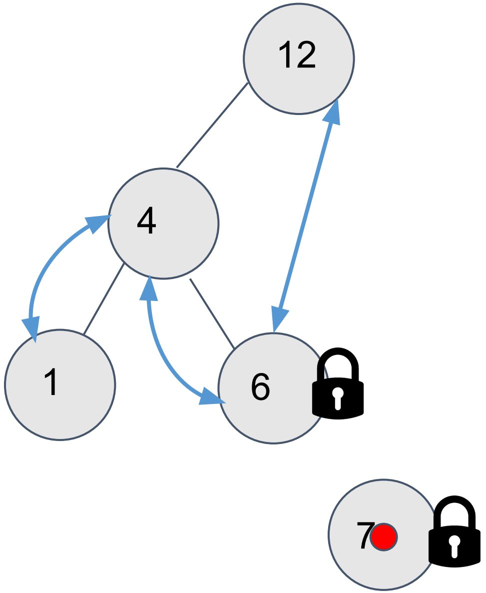
Logical Ordering: 1, 4, 6, 12

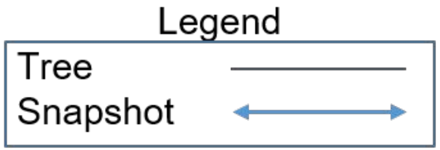
Remove(data) :
  node = traverse(data)
  parent.try_lock()
  mark(node)
  setField(Parent, null)

  updateParent(parent, rightSnapshot)

  Change right snap of parent to childs successor

  UnlockAll()



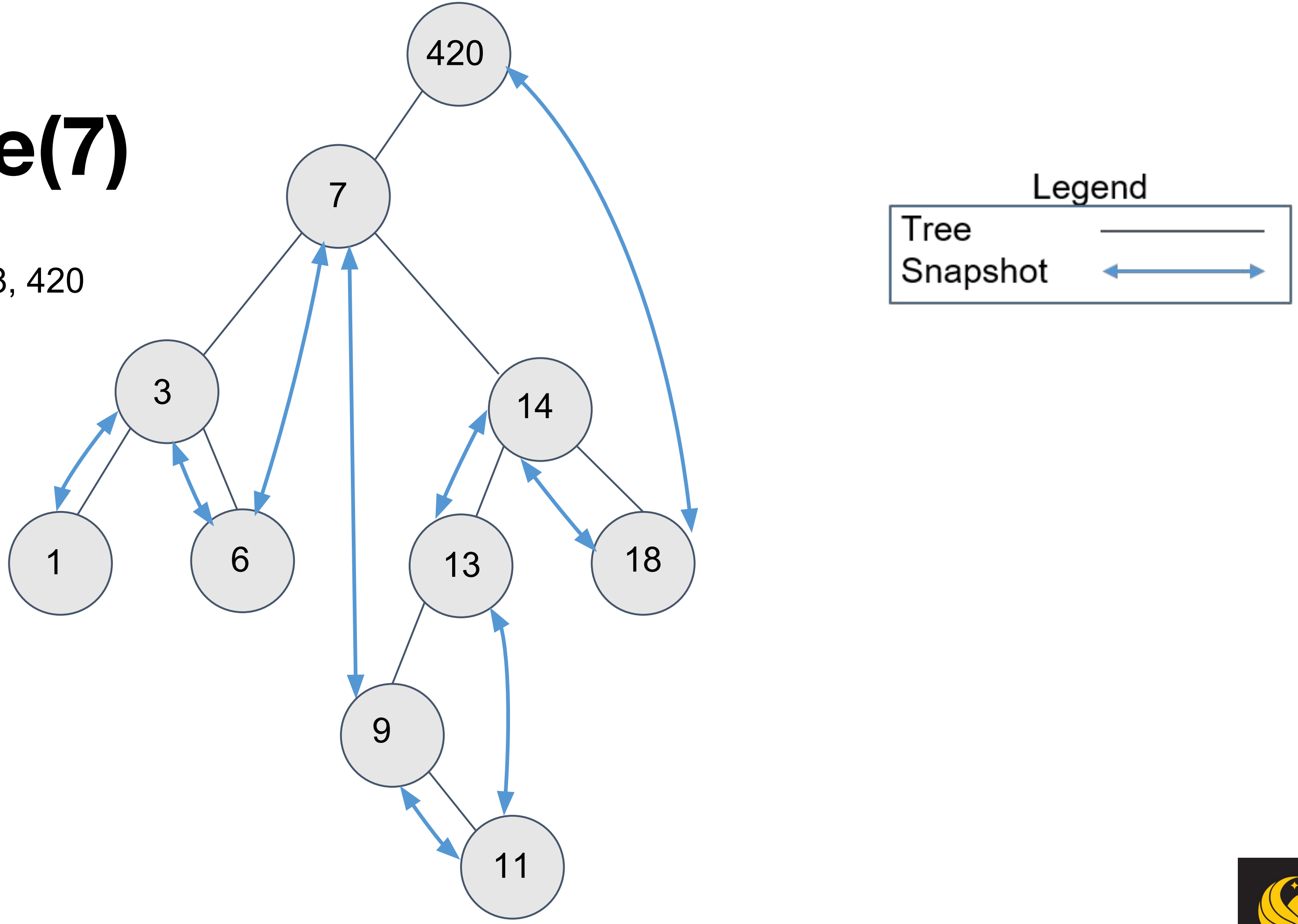


UCF

Remove(7)

Logical Ordering :
1, 3, 6, 7, 9, 11, 13, 14, 18, 420

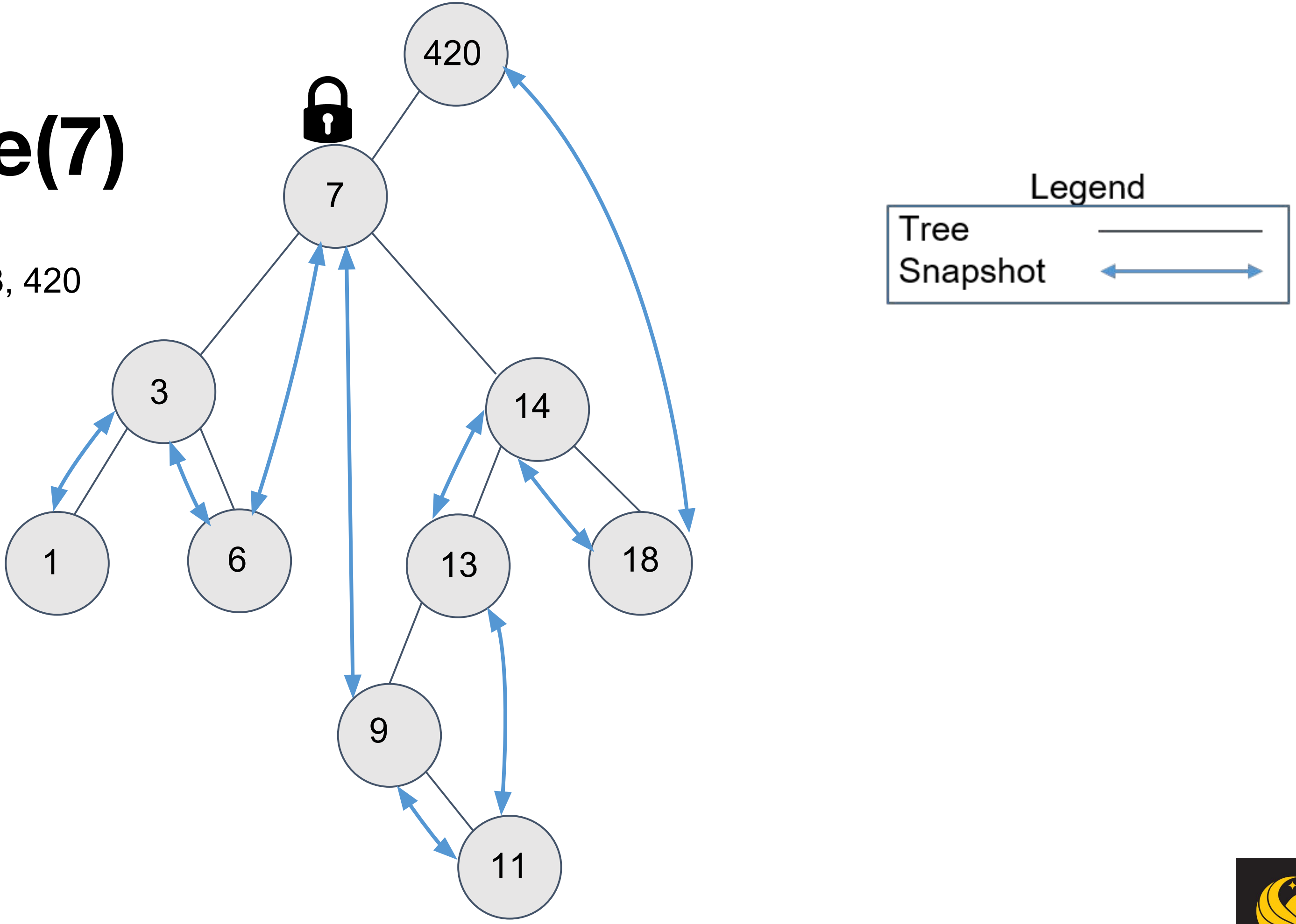
Remove(data):
node = traverse(data)
parent.try_lock()
leftChild.lock()
rightChild.lock()
pred.lock()
succParent.lock()
succ.lock()
succChild.lock()
mark(node)
bstRemove()
updateSnaps(pred, succ)
unlockAll()



Remove(7)

Logical Ordering :
1, 3, 6, 7, 9, 11, 13, 14, 18, 420

Remove(data):
node = traverse(data)
parent.try_lock()
leftChild.lock()
rightChild.lock()
pred.lock()
succParent.lock()
succ.lock()
succChild.lock()
mark(node)
bstRemove()
updateSnaps(pred, succ)
unlockAll()

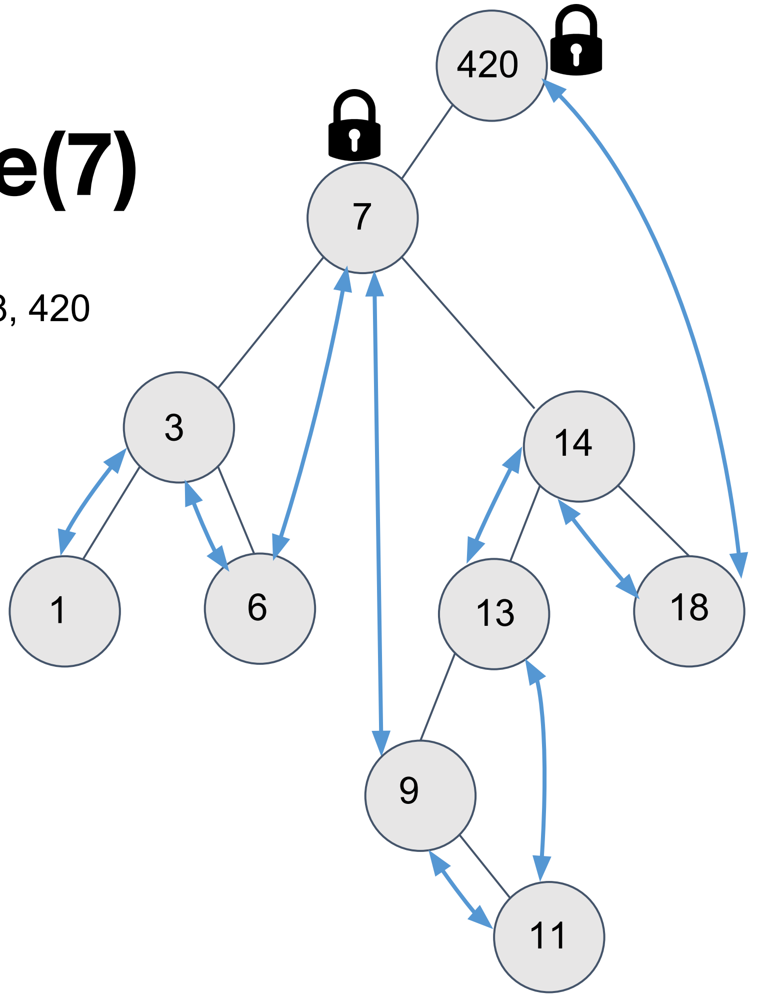


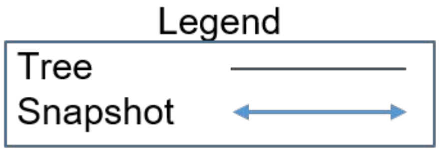
UCF

Remove(7)

Logical Ordering :
1, 3, 6, 7, 9, 11, 13, 14, 18, 420

Remove(data):
node = traverse(data)
parent.try_lock()
leftChild.lock()
rightChild.lock()
pred.lock()
succParent.lock()
succ.lock()
succChild.lock()
mark(node)
bstRemove()
updateSnaps(pred, succ)
unlockAll()

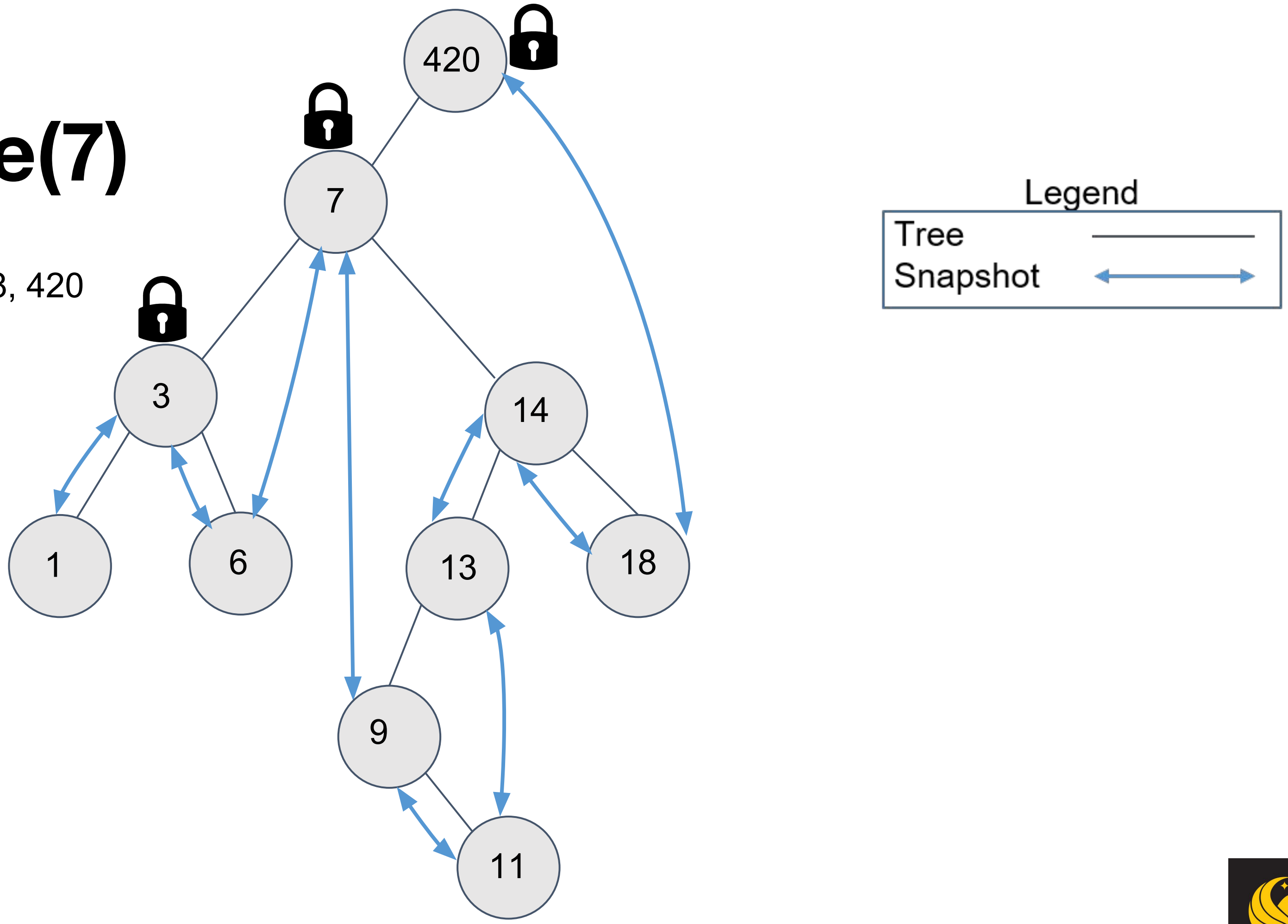




Remove(7)

Logical Ordering :
1, 3, 6, 7, 9, 11, 13, 14, 18, 420

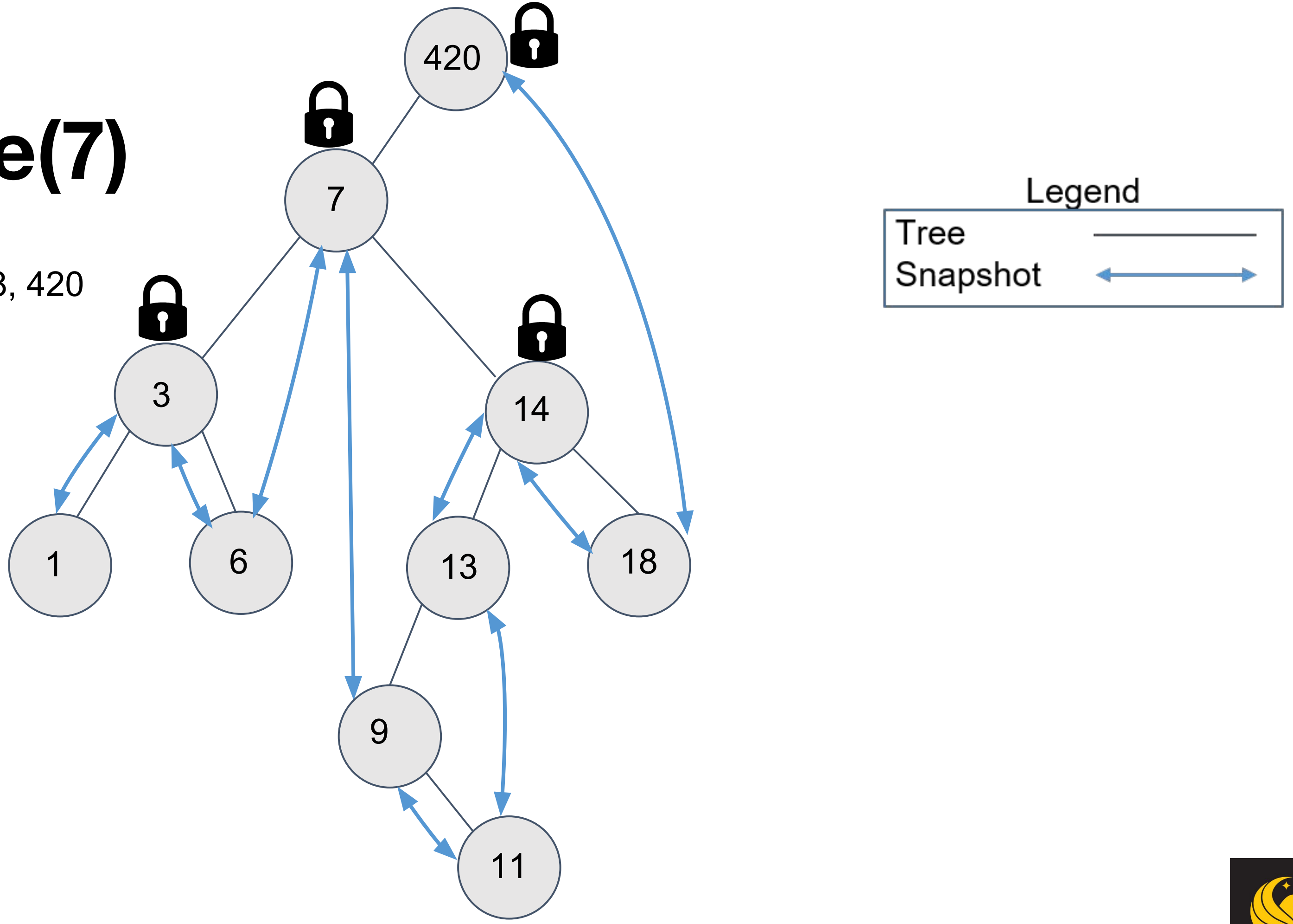
Remove(data):
node = traverse(data)
parent.try_lock()
leftChild.lock()
rightChild.lock()
pred.lock()
succParent.lock()
succ.lock()
succChild.lock()
mark(node)
bstRemove()
updateSnaps(pred, succ)
unlockAll()



Remove(7)

Logical Ordering :
1, 3, 6, 7, 9, 11, 13, 14, 18, 420

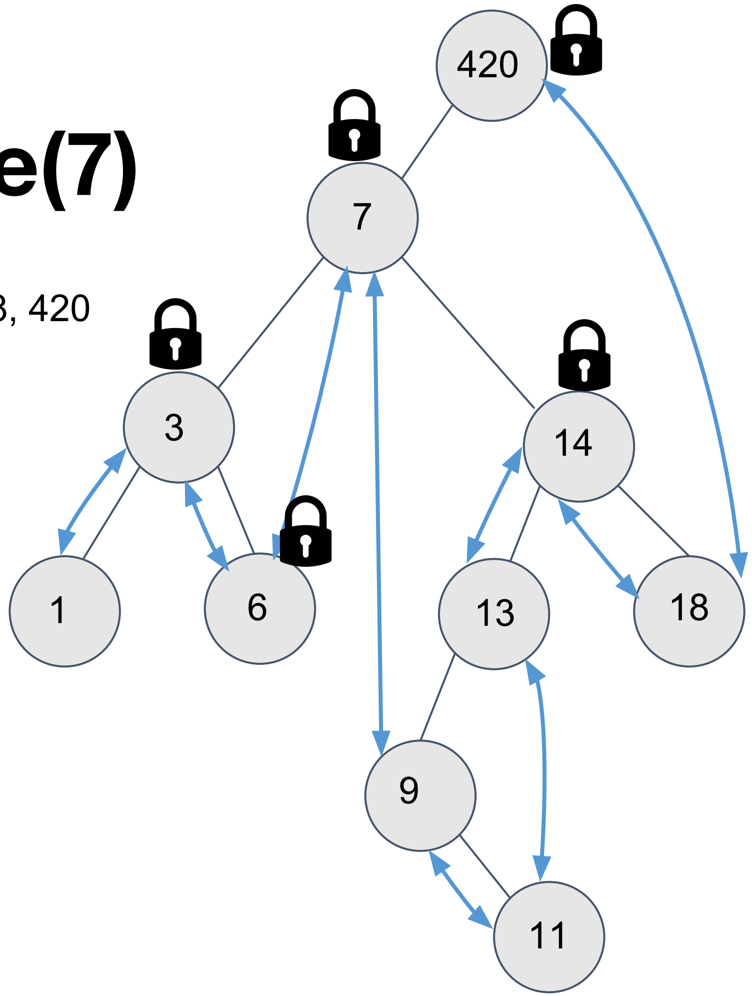
Remove(data):
node = traverse(data)
parent.try_lock()
leftChild.lock()
rightChild.lock()
pred.lock()
succParent.lock()
succ.lock()
succChild.lock()
mark(node)
bstRemove()
updateSnaps(pred, succ)
unlockAll()

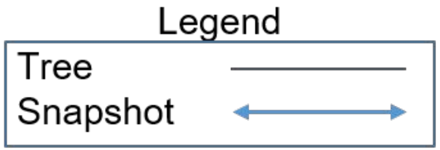


Remove(7)

Logical Ordering :
1, 3, 6, 7, 9, 11, 13, 14, 18, 420

Remove(data):
node = traverse(data)
parent.try_lock()
leftChild.lock()
rightChild.lock()
pred.lock()
succParent.lock()
succ.lock()
succChild.lock()
mark(node)
bstRemove()
updateSnaps(pred, succ)
unlockAll()



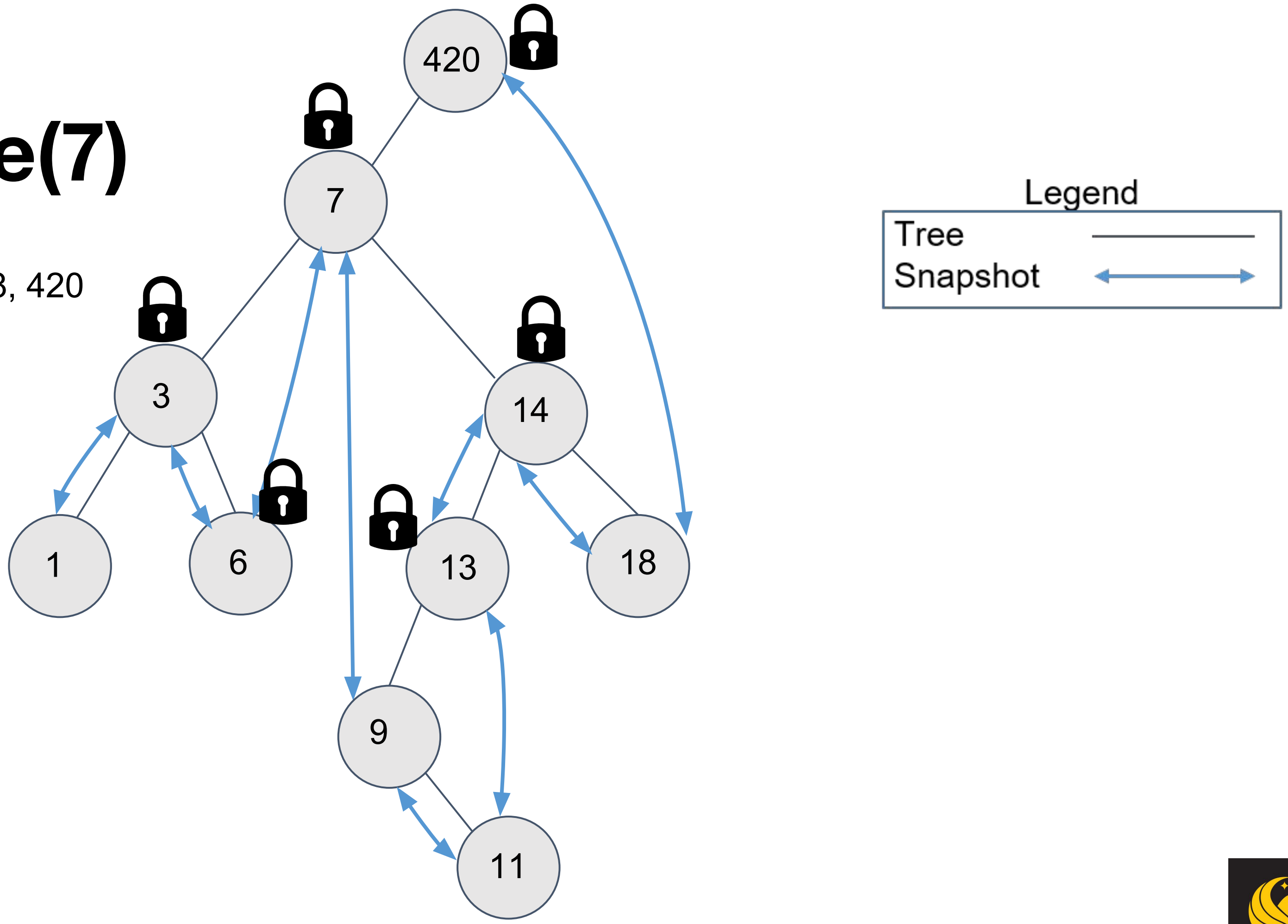


UCF

Remove(7)

Logical Ordering :
1, 3, 6, 7, 9, 11, 13, 14, 18, 420

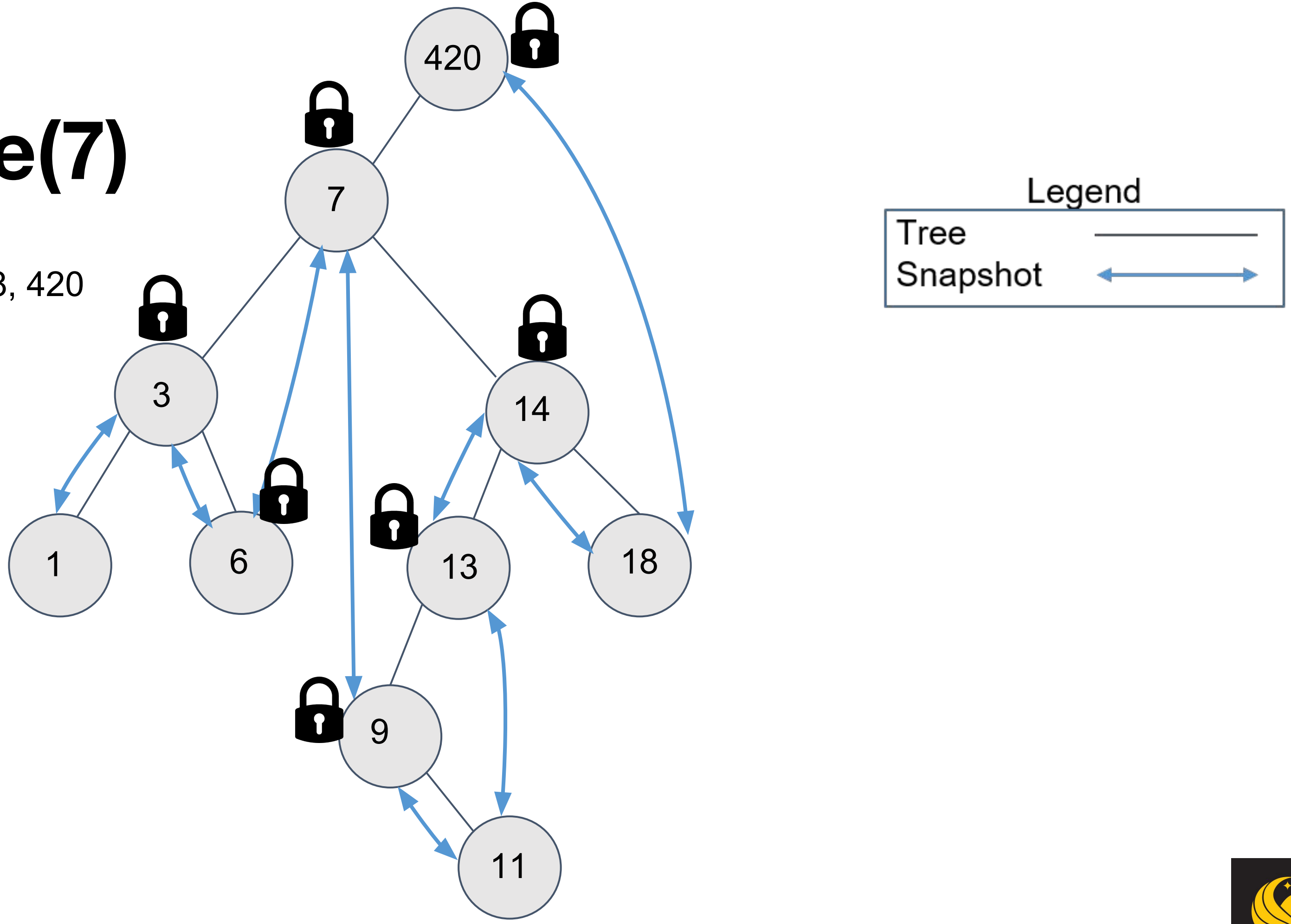
Remove(data):
node = traverse(data)
parent.try_lock()
leftChild.lock()
rightChild.lock()
pred.lock()
succParent.lock()
succ.lock()
succChild.lock()
mark(node)
bstRemove()
updateSnaps(pred, succ)
unlockAll()



Remove(7)

Logical Ordering :
1, 3, 6, 7, 9, 11, 13, 14, 18, 420

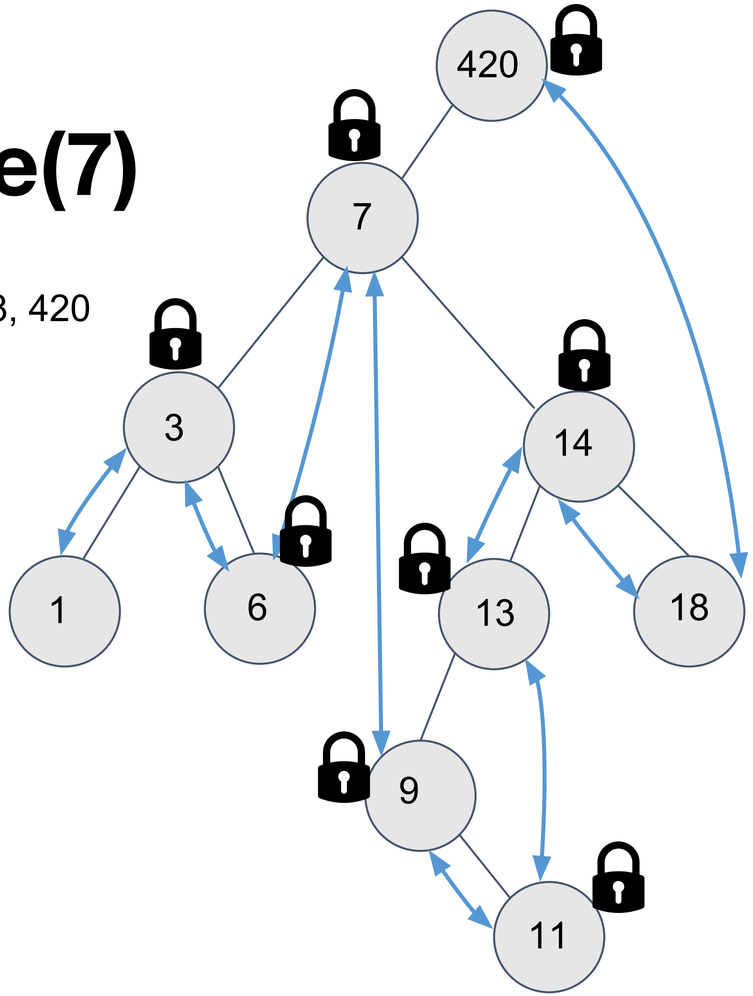
Remove(data):
node = traverse(data)
parent.try_lock()
leftChild.lock()
rightChild.lock()
pred.lock()
succParent.lock()
succ.lock()
succChild.lock()
mark(node)
bstRemove()
updateSnaps(pred, succ)
unlockAll()

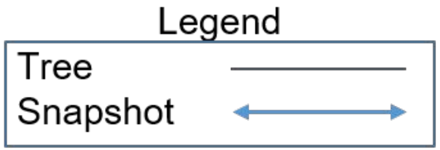


Remove(7)

Logical Ordering :
1, 3, 6, 7, 9, 11, 13, 14, 18, 420

Remove(data):
node = traverse(data)
parent.try_lock()
leftChild.lock()
rightChild.lock()
pred.lock()
succParent.lock()
succ.lock()
succChild.lock()
mark(node)
bstRemove()
updateSnaps(pred, succ)
unlockAll()

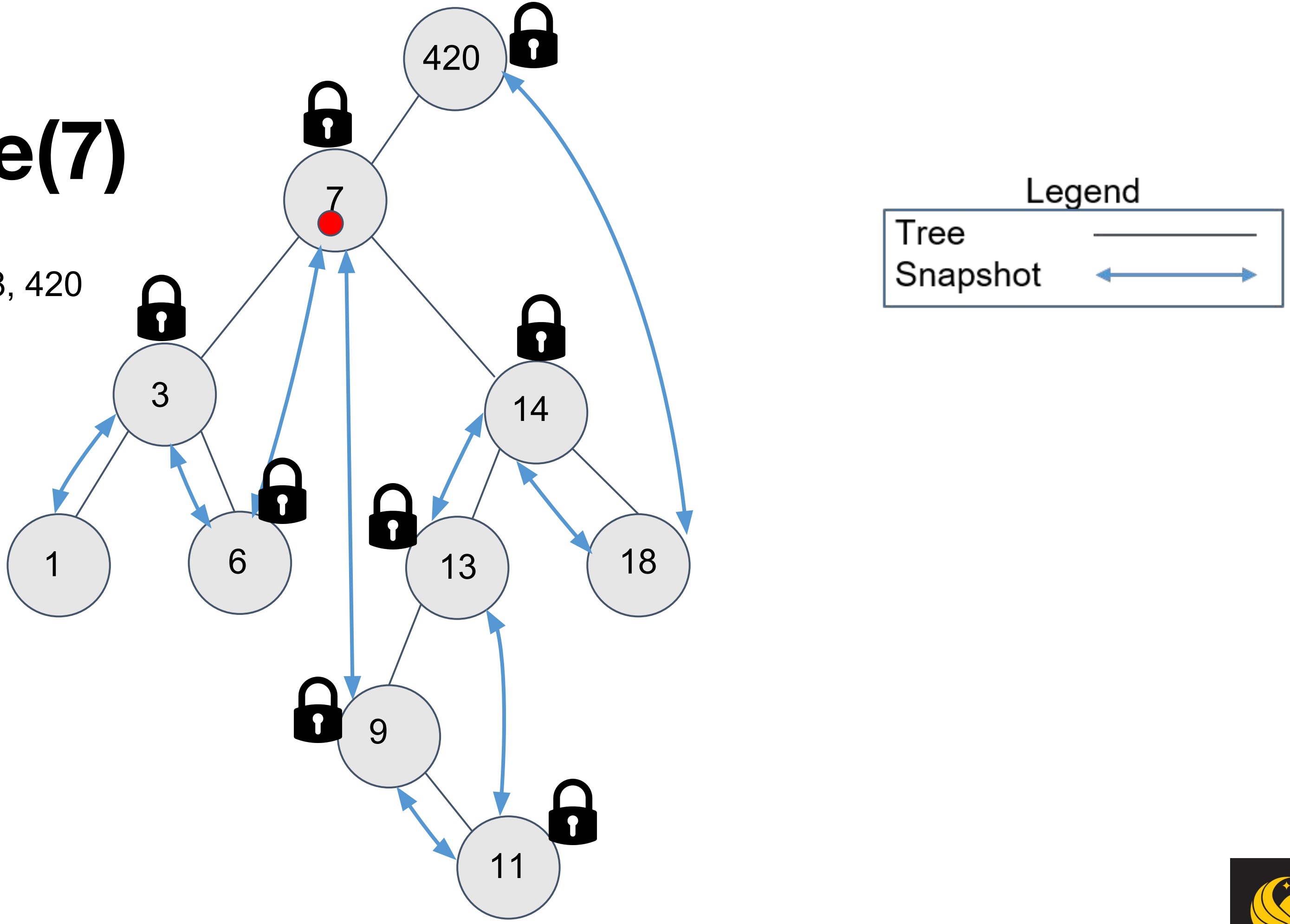




Remove(7)

Logical Ordering :
1, 3, 6, 7, 9, 11, 13, 14, 18, 420

Remove(data):
node = traverse(data)
parent.try_lock()
leftChild.lock()
rightChild.lock()
pred.lock()
succParent.lock()
succ.lock()
succChild.lock()
mark(node)
bstRemove()
updateSnaps(pred, succ)
unlockAll()

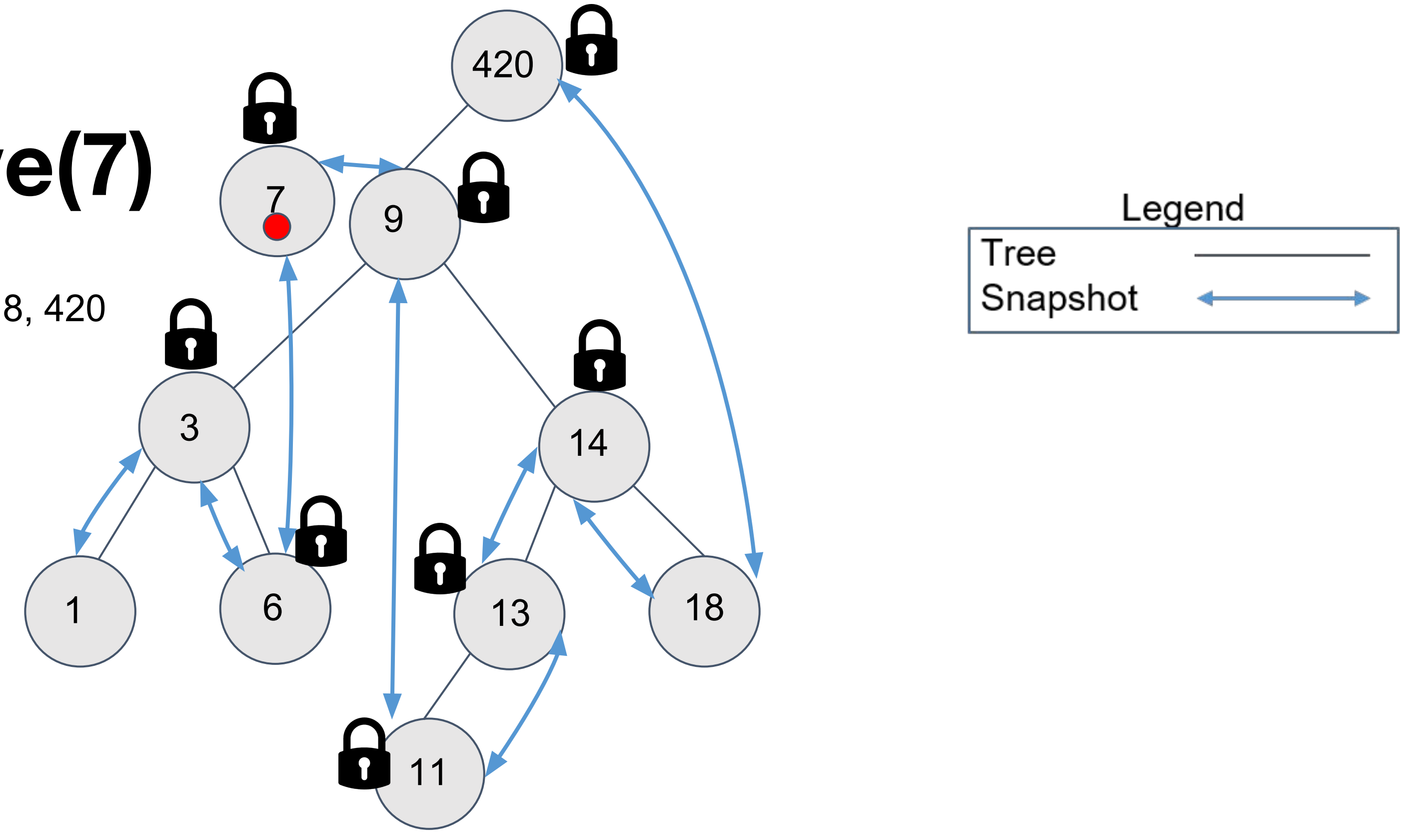


UCF

Remove(7)

Logical Ordering :
1, 3, 6, 7, 9, 11, 13, 14, 18, 420

Remove(data):
node = traverse(data)
parent.try_lock()
leftChild.lock()
rightChild.lock()
pred.lock()
succParent.lock()
succ.lock()
succChild.lock()
mark(node)
bstRemove()
updateSnaps(pred, succ)
unlockAll()

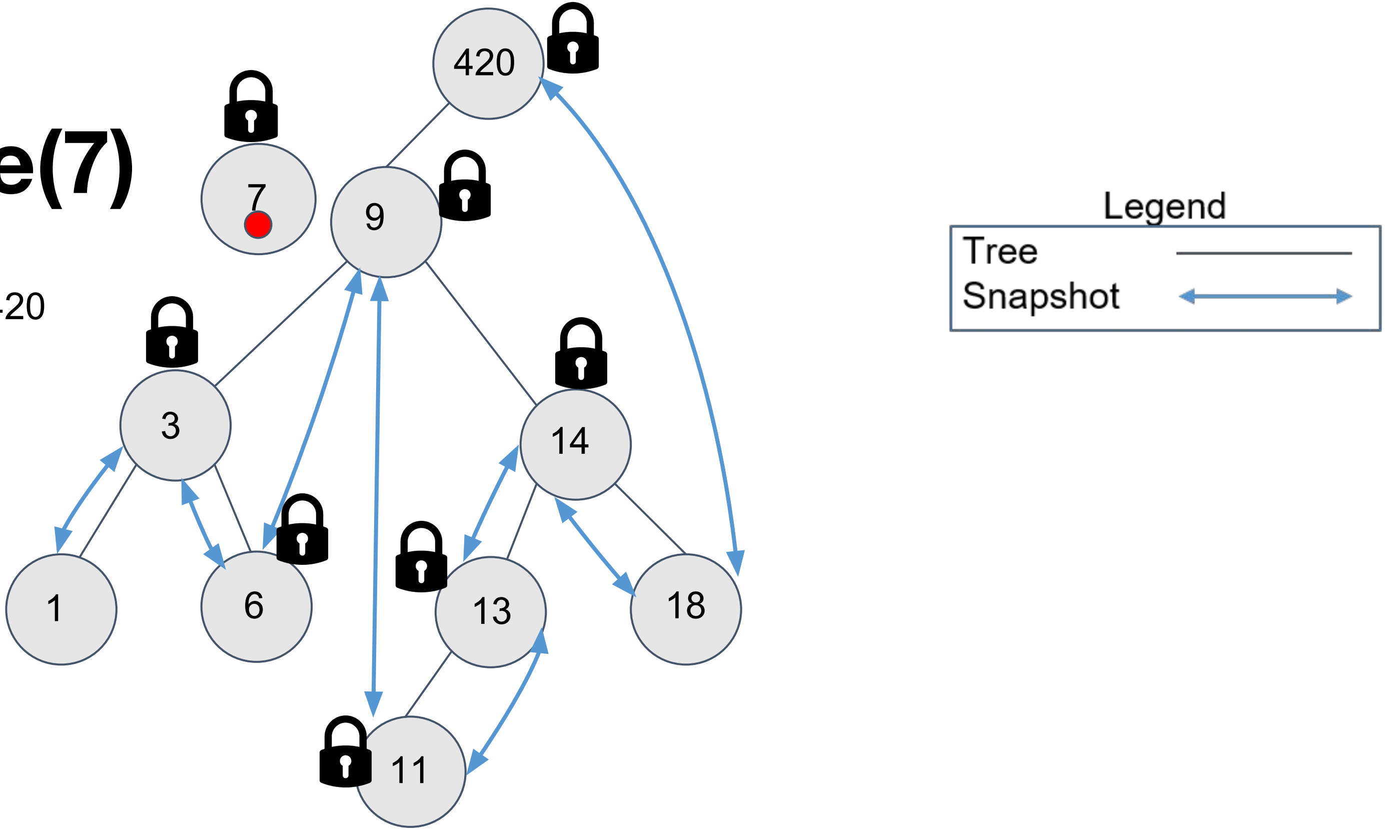


UCF

Remove(7)

Logical Ordering :
1, 3, 6, 9, 11, 13, 14, 18, 420

Remove(data):
node = traverse(data)
parent.try_lock()
leftChild.lock()
rightChild.lock()
pred.lock()
succParent.lock()
succ.lock()
succChild.lock()
mark(node)
bstRemove()
updateSnaps(pred, succ)
unlockAll()



Remove(data)

- Deletes the node from the tree and replaces it if the node being deleted is not a root
- Uses locks by calling traverse
- Uses additional locks based on whether or not the node being removed has children
- Attempts to lock against locking order when locking the parent so only a trylock is used
  - If this fails then we restart
- Attempts to rebalance once removal is finished if it is an AVL tree
- Remove linearizes when it successfully marks the removed node

Reimplementation

- We faced two main challenges with our implementation
  - Creating and maintaining snapshots
    - Snapshots were a bit obfuscated in the paper
    - The authors were not precise on what they represented in their example
  - Handling the worst case scenario of Remove
    - The worst case scenario involves 8 locks
    - This made handling and debugging them a terror when other scenarios only had to lock a few nodes
    - No mention of locking the predecessor of the node being removed

GCC STM Data Structure

- GCC Transactional memory is used to create blocks of code that will be executed atomically
- Important Method/Properties of GCC
  - __transaction_atomic{}
    - All code in this statement block will be executed atomically
    - By default it executes in an all or nothing fashion and will repeat until it successfully commits
    - Will not execute transaction unsafe code.
  - transaction_safe & transaction_unsafe
    - A property that can be applied directly to functions
    - By default the GCC can determine transaction_safe functions
    - However, this is only true if the code is defined in the source file

STM Implementation

The steps we took to create the GCC implementation are:

1) Remove atomic wrappers from objects as they are not transactionally safe
2) Mark methods outside the source file as transactionally safe
3) Atomic Transactions:

```
insert(data) {
  __transaction_atomic {
      node = traverse(root, data)
      ...
      if (avl) rebalance(node)
  }
}
```

```
remove(data) {
  __transaction_atomic {
      node = traverse(root, data)
      ...
      if (avl) rebalance(node)
  }
}
```

STM Implementation

Contains only requires atomic reads for the snaps shots so it still provides concurrency!

Improved GCC STM Implementation

- Decouple two functions from the atomic block
- Traverse
  - Snapshots are now checked inside of traverse
  - Atomic blocks contain O(1) write operations, no longer traverseing
- Rebalancing
  - Decoupled rebalancing from all functions where they are called
  - Rebalancing has is its own atomic transaction
  - Thread will allow others to operate while it attempts to rebalance the tree.
  - Traversal is done outside of atomic updates

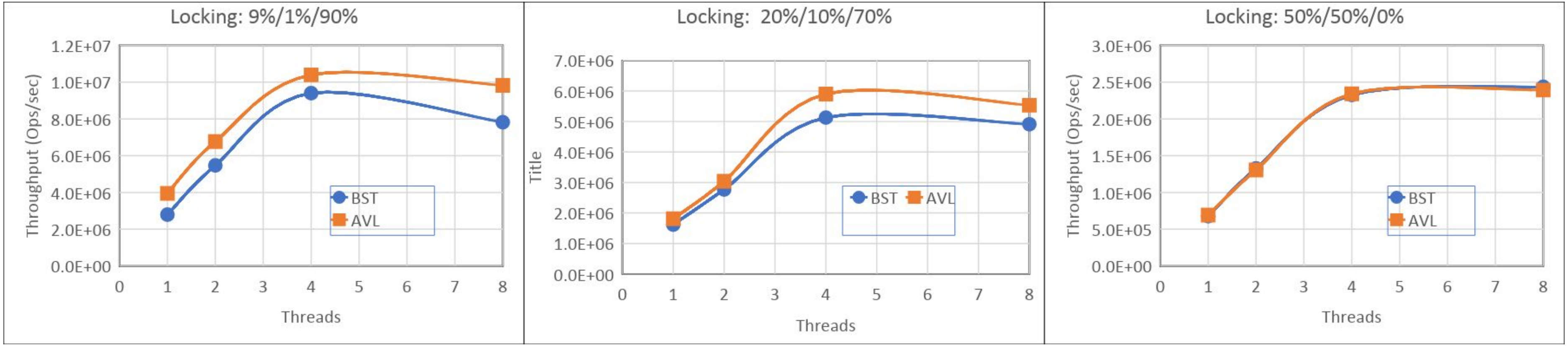
UCF

Experimental Results

- Experimental Details
  - Intel(R) Core(TM) i5-4460 CPU @ 3.2GHz
  - Number of Cores/Threads 4/4
  - 24.0 GB RAM
  - Ubuntu 18.04
- BenchMarks (Insert /Remove /Contains):
  - 9% /1% /90%
  - 20% /10% /70%
  - 50% /50% /0%
- Tree initially filled to expected capacity (90%, 2/3, 50%)

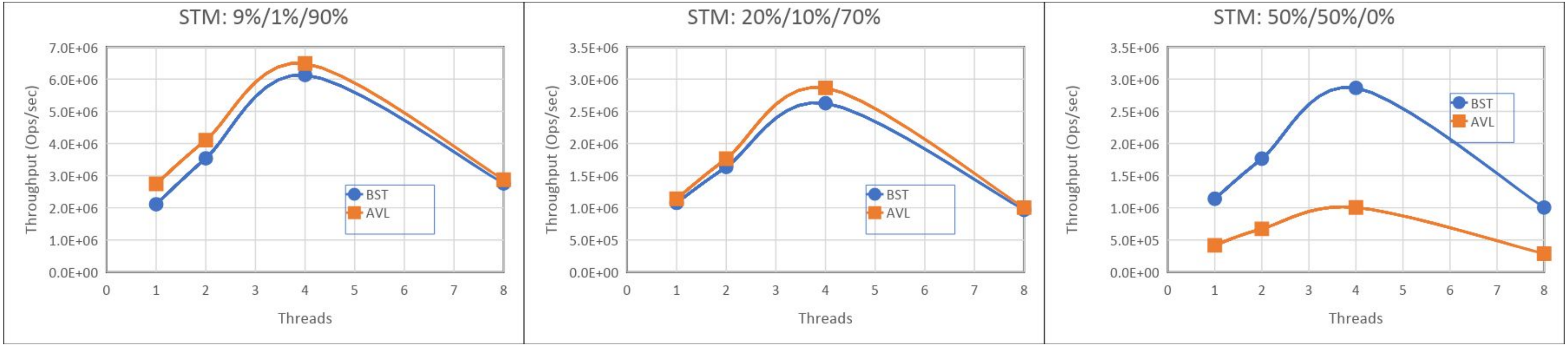
UCF

Data Structure Results



- AVL generally outperforms BST except where there are high number of inserts/removes
  - This scenario would force more rebalancing and AVL would have less leafs as a result
- Scaled linearly until we hit a wall at four threads which we suspect is due to hardware limitations

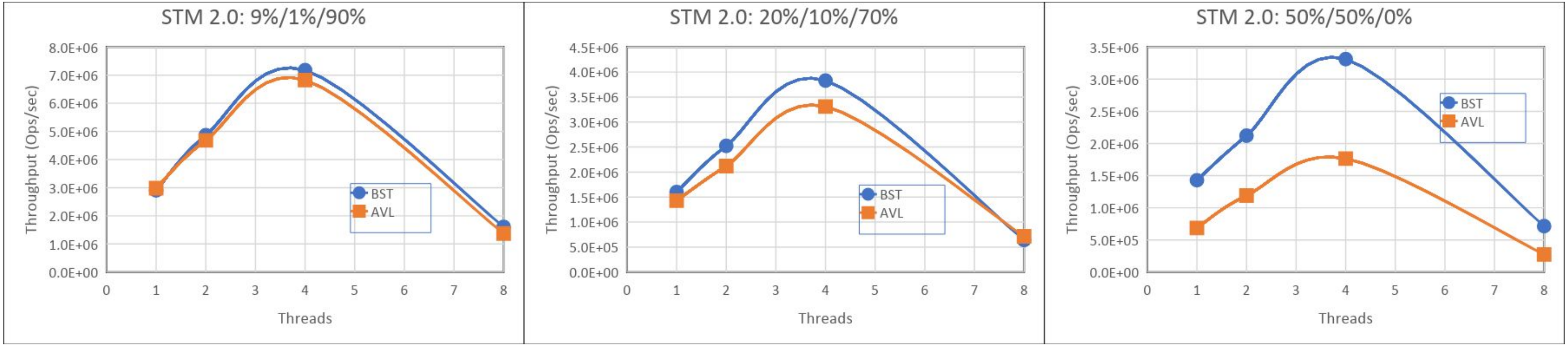
STM Results



- AVL outperformed BST on most occasions except when it comes to high inserts/removes
  - This is due to the amount of rebalancing that gets called that would interfere with all transactions
- Performance sharply drops in all cases due to hardware

UCF

STM 2.0: Results



- In the first graph the output for BSTs increased by 50 percent with one and two thread and by about 10 percent with four threads
- In the second graph the output for BSTs increased by 50 percent up to 4 threads
- AVL has a slight increase overall with the notable being the last graph having a 50% increase with four threads

UCF

Conclusion

- Our implementation scales well until we hit our hardware limitation
- Transactional implementation performs worse than our original data structure in most scenarios except when there are a high number of inserts & removes
- PaVT is show to provide lock free membership tests.
- Snapshots are easy to maintain and implement for the search trees and take up very little space