A Practical Concurrent Binary Search Tree

Nathan Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun

Stanford University

PPoPP 2010

SnapTree

- Optimistically concurrent
  - Linearizable reads and writes, invisible readers
- Good performance and scalability
  - 31% single-thread overhead vs. Java's TreeMap
  - Faster than ConcurrentSkipListMap for many operation mixes and thread counts
- Fast atomic clone
  - Lazy copy-on-write with structural sharing
  - Provides snapshot isolation for iteration

Concurrent binary tree challenges

- Every operation accesses the root, so concurrent reads must be highly scalable

  → Optimistic concurrency allows invisible readers
- It's hard to predict on first access whether a node will be modified later

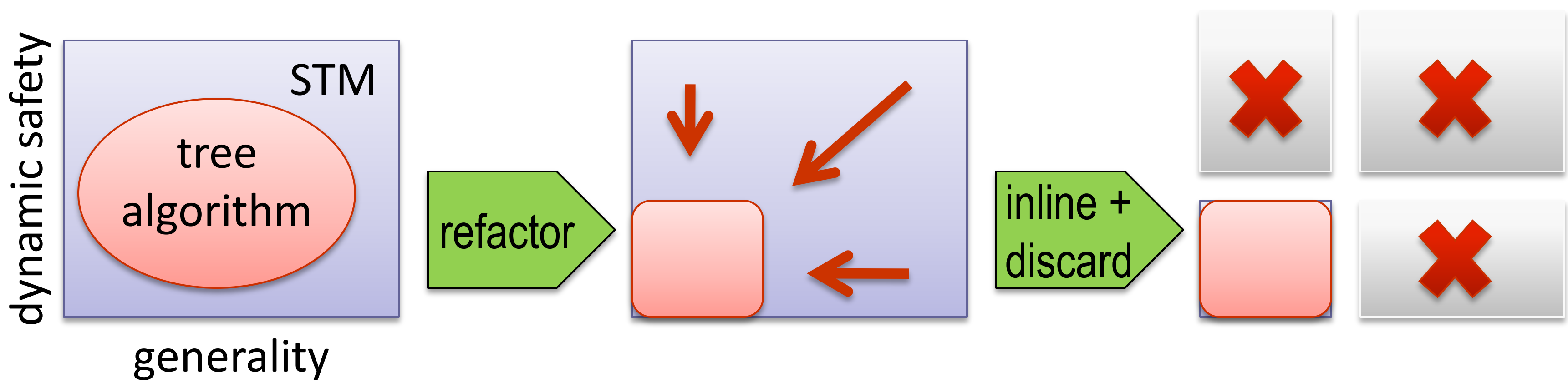
  → STMs avoid the deadlock problem of lock upgrades
- Multiple links must be updated atomically

  → STMs provide atomicity and isolation across writes

Software Transactional Memory (STM) addresses all these problems, but has high single-thread overheads

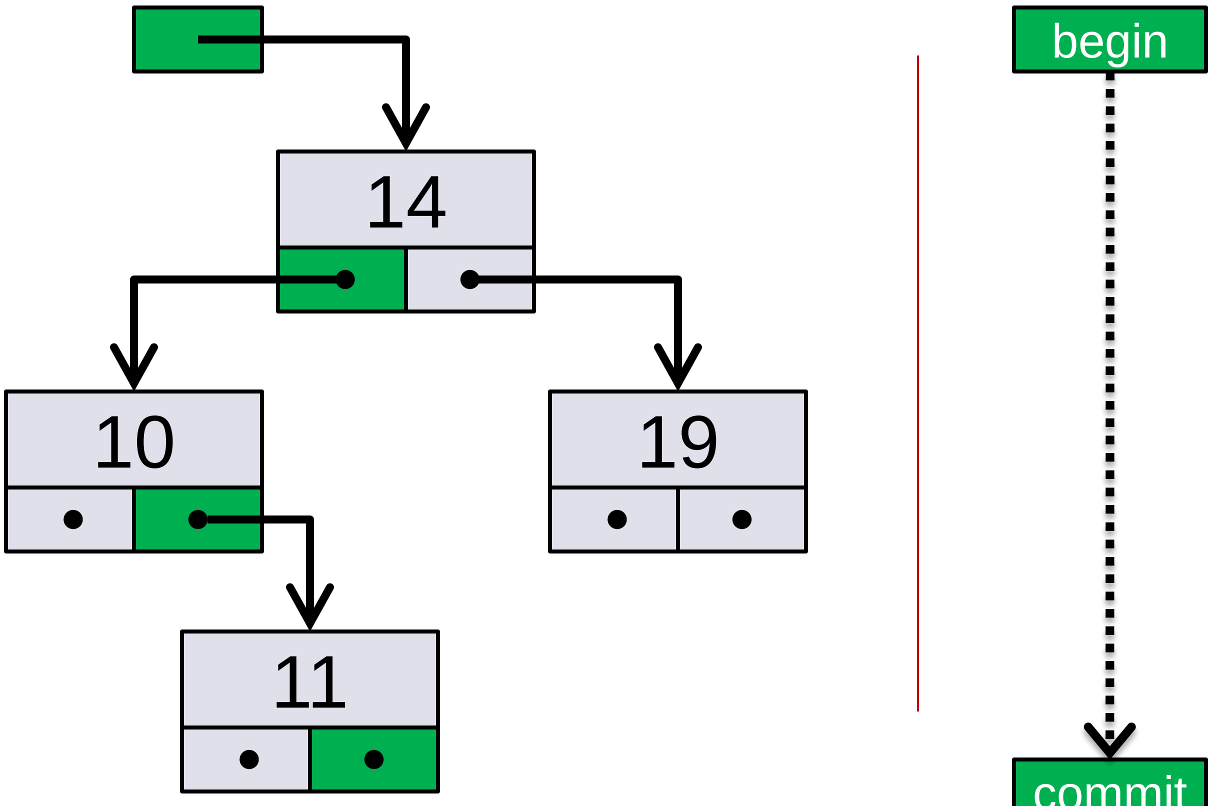
Tailoring STM ideas for trees

1. Provide no transactional interface to the outside world
2. Reason directly about *semantic* conflicts
3. Change the algorithm to avoid dynamically-sized txns
4. Inline control flow and metadata
   - No explicit read set or write buffer, no indirection
5. Move safety into the algorithm
   - No deadlock detection, privatization safety, or opacity in the STM



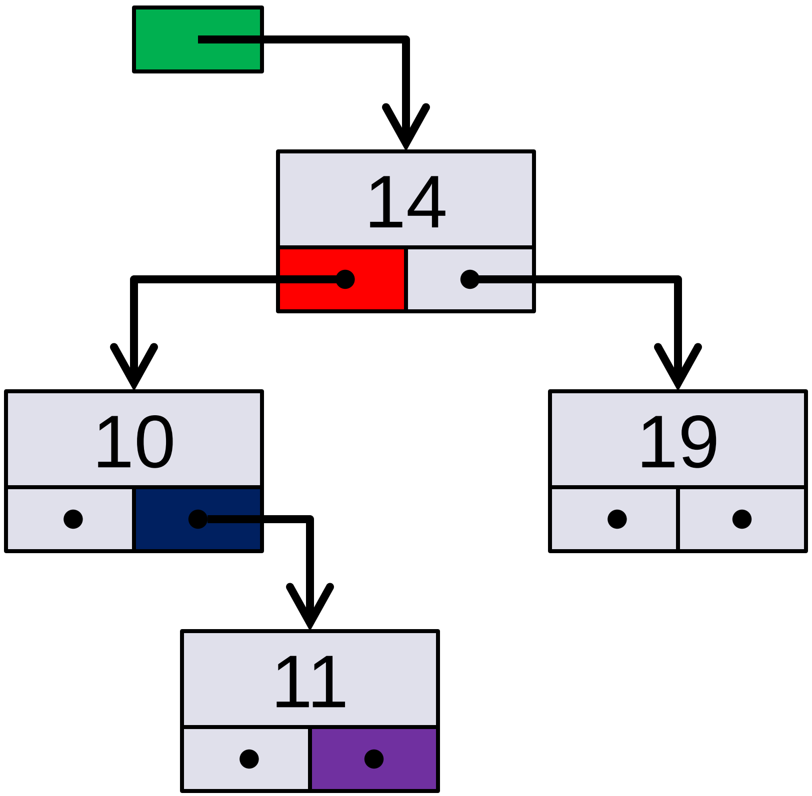
Bad: Searching in a single big txn

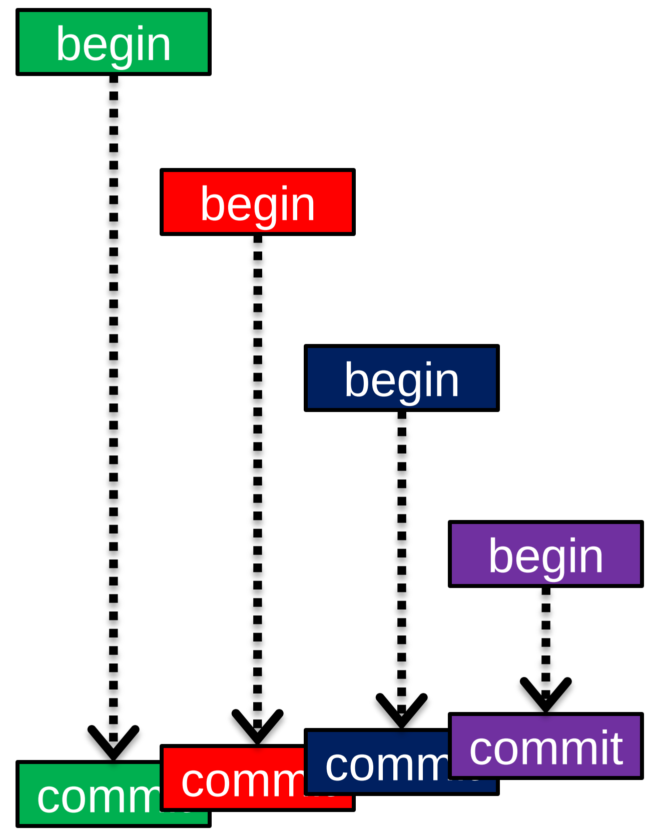
- Optimistic failure → start over
- Concurrent write anywhere on the path → start over



Better: Nest for partial rollback

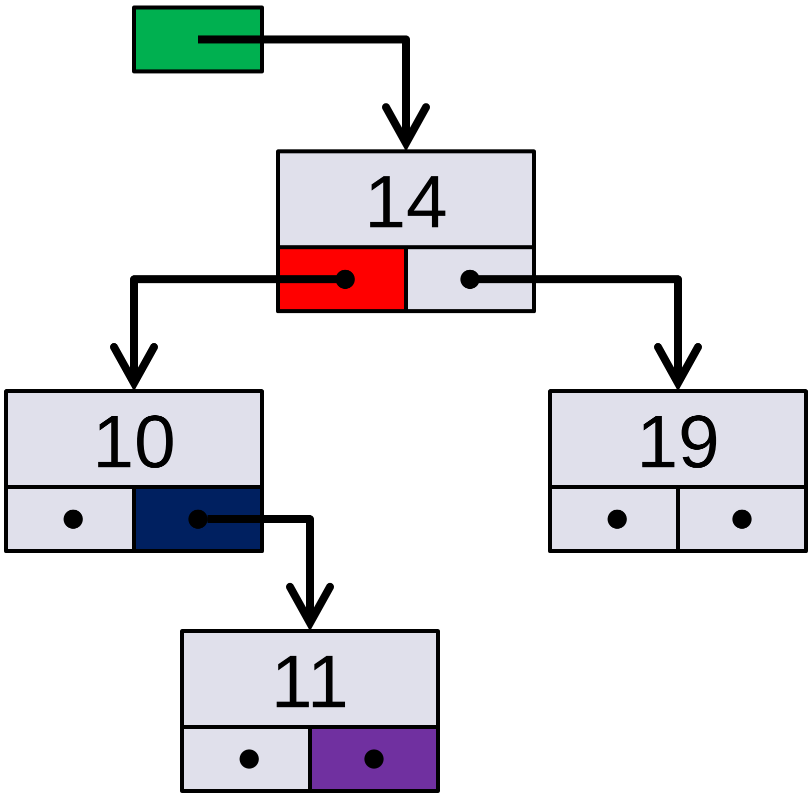
- Optimistic failure → partial rollback
- Concurrent write anywhere on the path → partial rollback

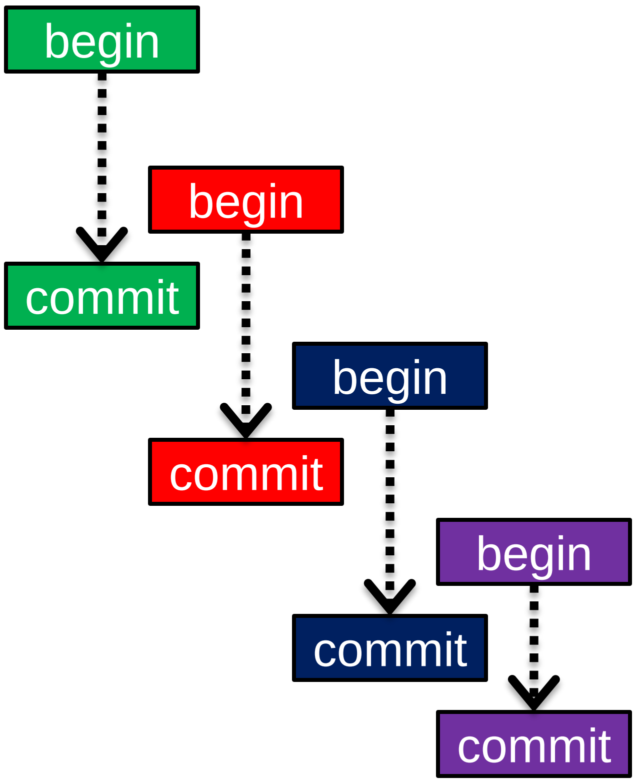




Even better: Hand-over-hand txns

- Hand-over-hand optimistic validation
- Commit early to mimic hand-over-hand locking





Overlapping non-nested txns?

```
a = Atomic.begin();
  r1 = read_in_a;
  b = Atomic.begin();
    r2 = read_in_b;
a.commit();
  ...
  b.commit();
```

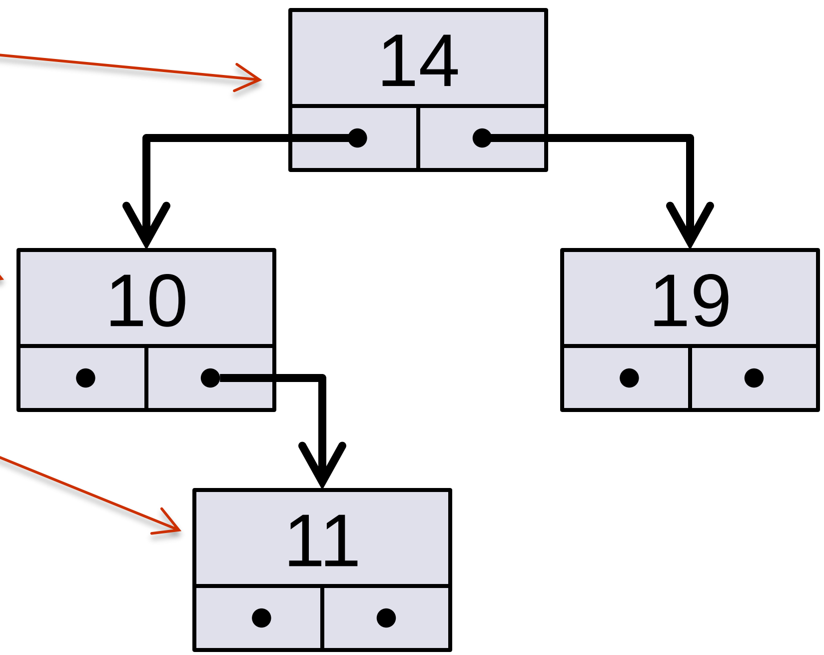
What does this mean?

- “read-only commit” == “roll back if reads are not valid”*
  - Just a conditional non-local control transfer
- This gives a meaning, but what about correctness?

* - *A bit sloppy, but generally accurate for STMs that linearize during commit*

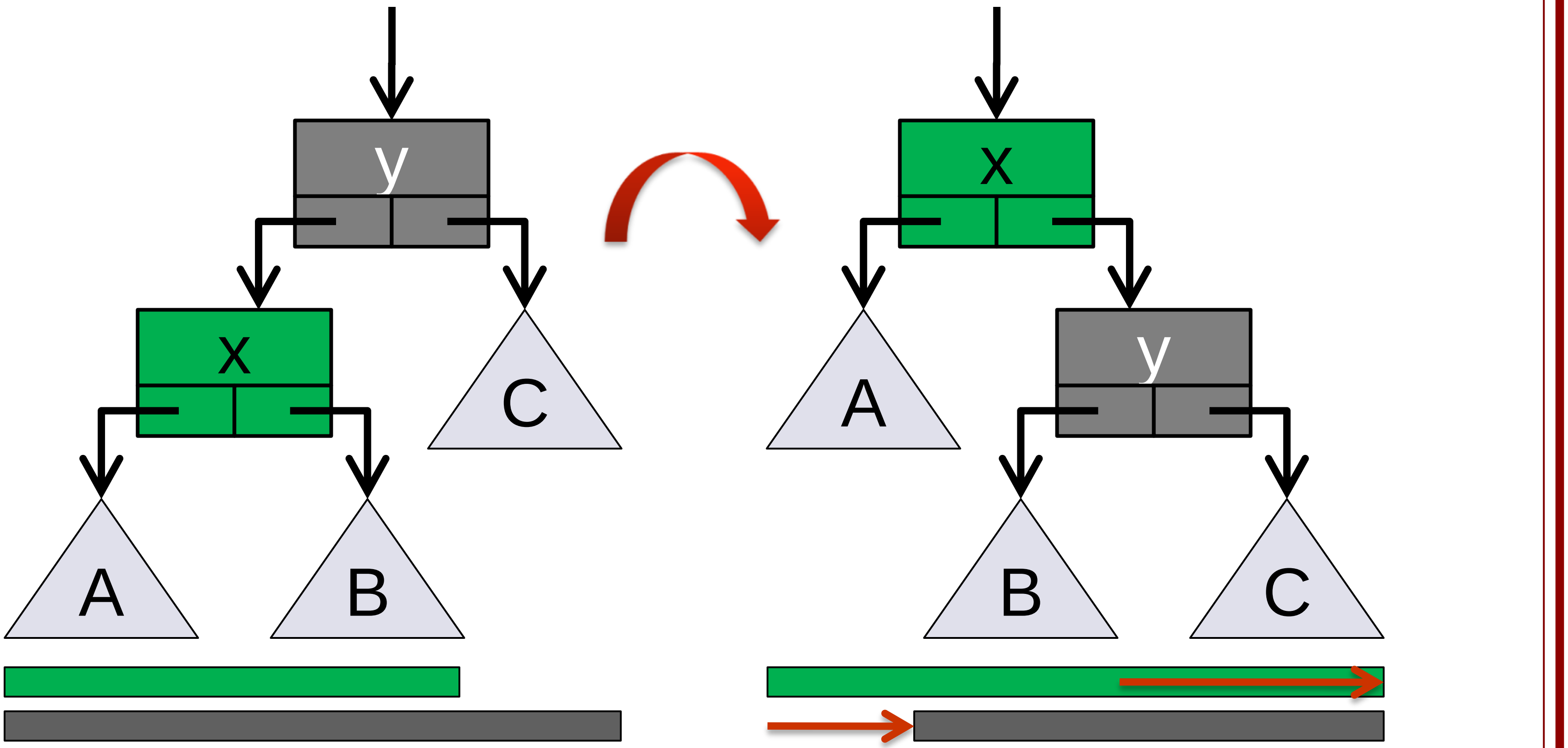
Correctness of hand-over-hand

- Explicit state = current node *n*
- Implicit state = range of keys rooted at *n*
  - *Guarantees that if a node exists, we will find it*



What concurrent mutations are possible?

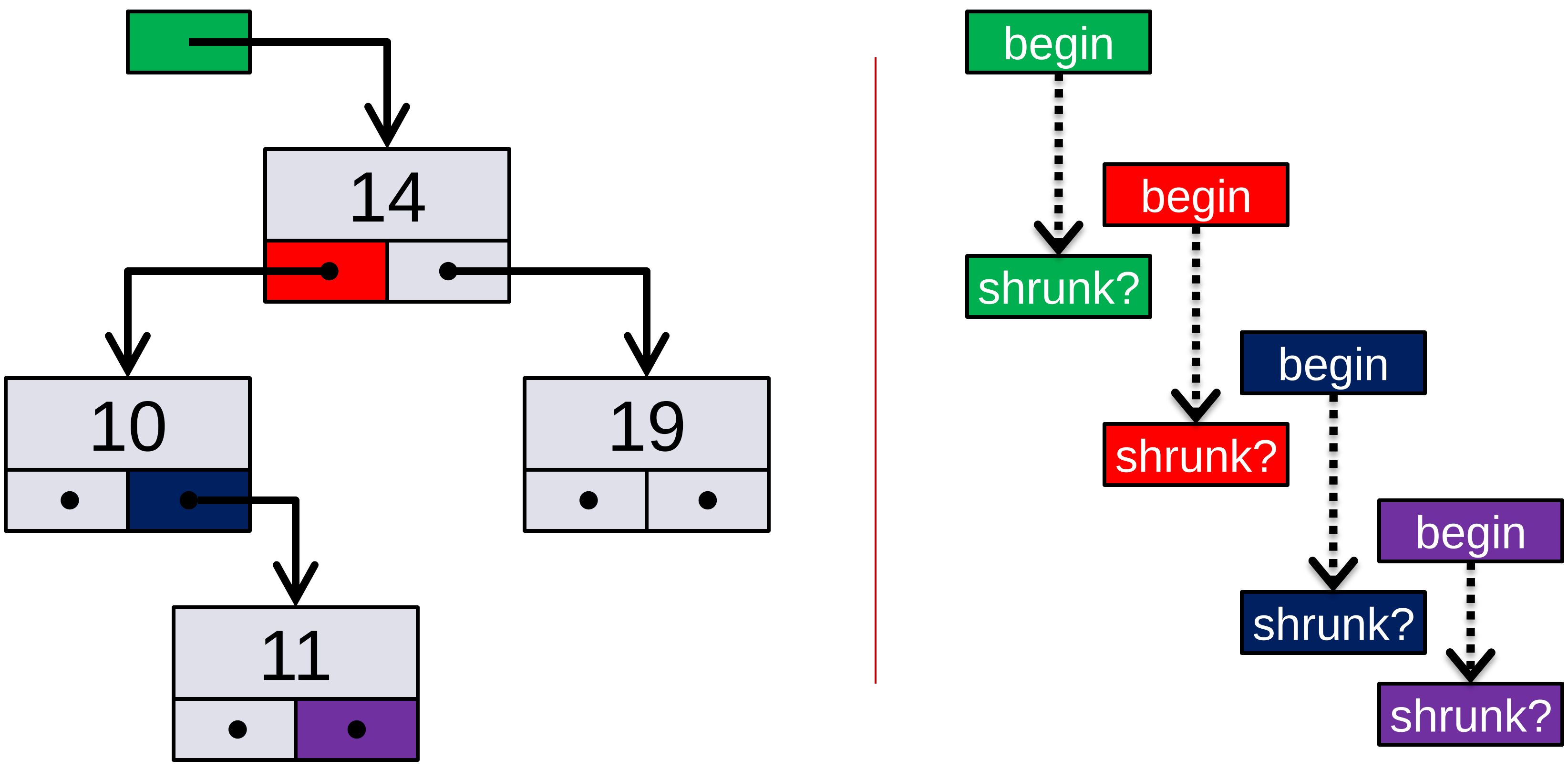
Conflict between search and rotation



Branch rooted at x **grows** → search at x is okay
Branch rooted at y **shrinks** → search at y is invalid

Best: Tree-specific validation

- Hand-over-hand optimistic validation
- Version number only incremented during ‘shrink’



Updating with fixed-size txns

- Insert can be the end of a hand-over-hand chain
- Restoring balance in one fixed-size txn is not possible
  - Red-black trees may recolor O(log *n*) nodes
  - AVL trees may perform O(log *n*) rotations
- Solution → relaxed balance
  - Extend rebalancing rules to trees with multiple defects
    - *Possible for red-black trees and AVL trees, AVL is simpler*
  - Defer rebalancing rotations
    - *Originally this was done on a background thread*
    - *We will rebalance immediately, just in separate txns*
  - Tree will be properly balanced when quiescent

Inlining example: recursive search

```
Node search(K key) {
  Txn txn = Atomic.begin();
  return search(txn, root, key);
}
Node search(Txn parentTxn, Node node, K key) {
  int c = node == null ? 0 : key.compareTo(node.key);
  if (c == 0) {
    parentTxn.commit();
    return node;
  } else {
    Txn txn = Atomic.begin();
    Node child = c < 0 ? node.left : node.right;
    parentTxn.commit();
    return search(txn, child, key);
  }
}
```

hand-over-hand transactions

transactional read barriers

Inlining STM control flow

```
Node RETRY = new Node(null); // special value

Node search(K key) {
  while (true) {
    Txn txn = Atomic.begin();
    Node result = search(txn, root, key);
    if (result == RETRY) continue;
    return result;
  }
}
Node search(Txn parentTxn, Node node, K key) {
  int c = node == null ? 0 : key.compareTo(node.key);
  if (c == 0) {
    if (!parentTxn.isValid()) return RETRY;
    return node;
  } else {
  ...
```

Inlining txn state + barriers

```
class Node { volatile long version; ... }
final Node rootHolder = new Node(null);

Node search(K key) {
  while (true) {
    long v = rootHolder.version;
    if (isChanging(v)) { awaitUnchanging(rootHolder); continue; }
    Node result = search(rootHolder, v, rootHolder.right, key);
    if (result == RETRY) continue;
    return result;
  }
}
Node search(Node parent, long parentV, Node node, K key) {
  int c = node == null ? 0 : key.compareTo(node.key);
  if (c == 0) {
    if (parent.version != parentV) return RETRY;
    return node;
  } else {
    ...
```

Inlined read barrier

Inlined read set

Inlined validation

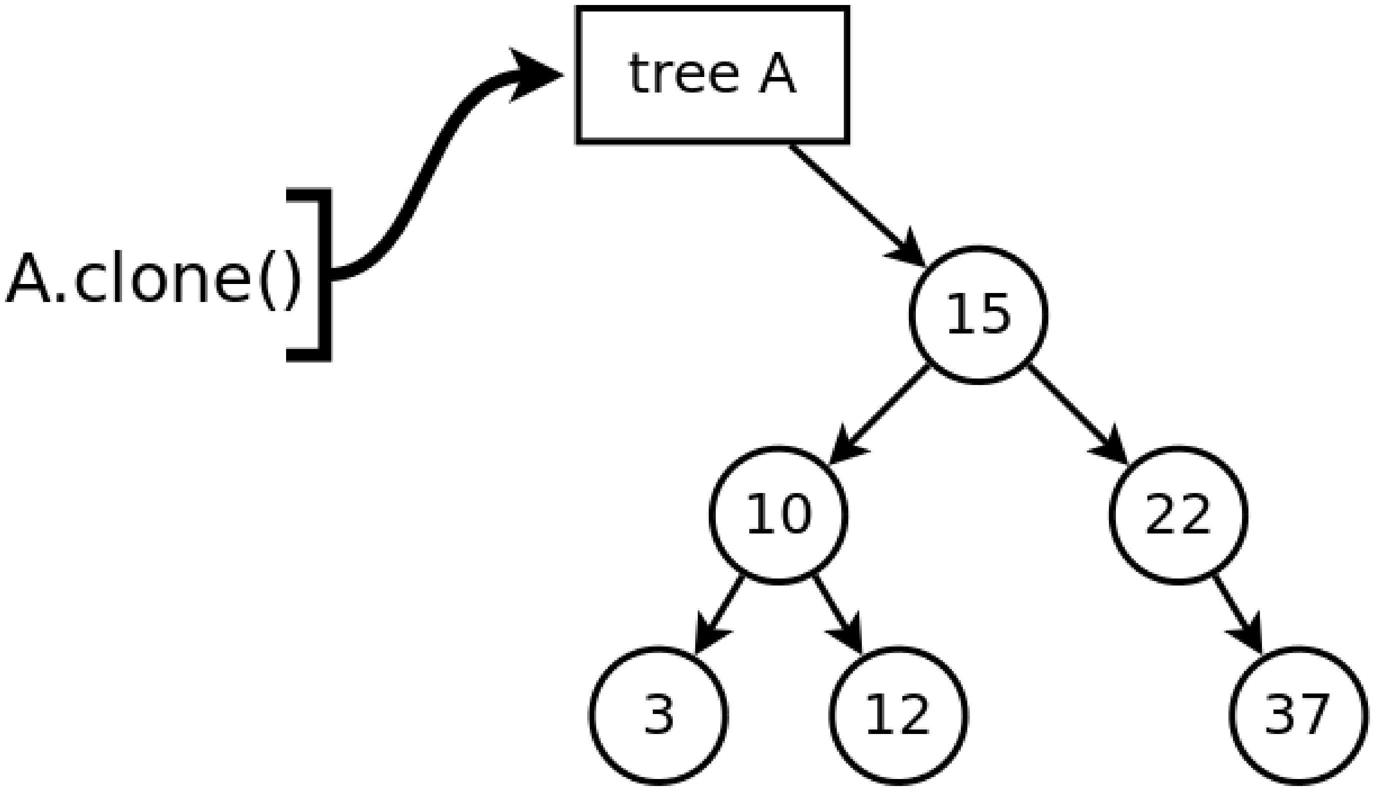
Atomic clone()

Goal: snapshot isolation for consistent iteration

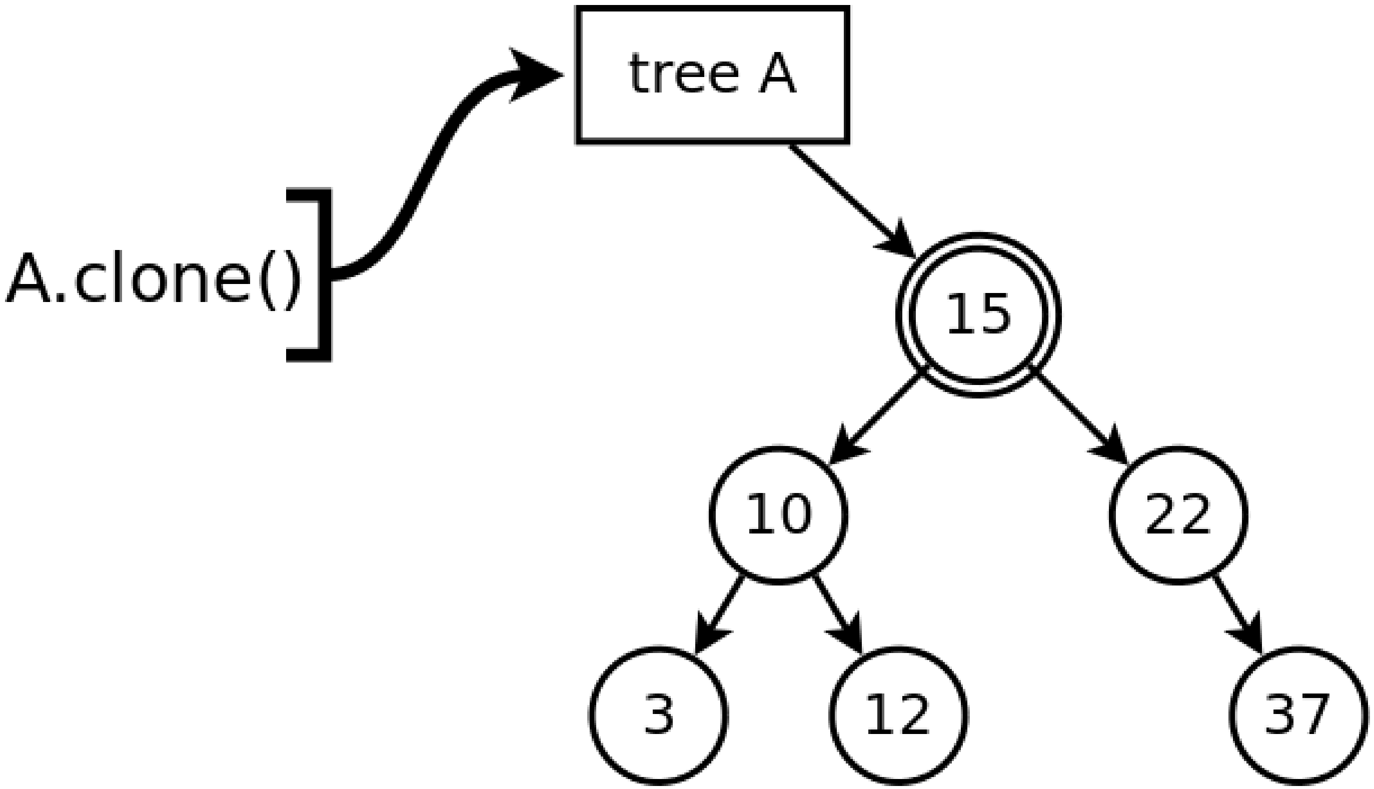
Strategy: use copy-on-write to share nodes

1. Separate mutating operations into *epochs*
   - Nodes from an old epoch may not be modified
   - Epoch tracking resembles a striped read/write lock
     - *Tree reads ignore epochs*
     - *Tree writes acquire shared access*
2. Mark lazily
   - Initially, only mark the root
   - Mark the children before making a copy
3. Copy lazily
   - Make private copies during the downward traversal

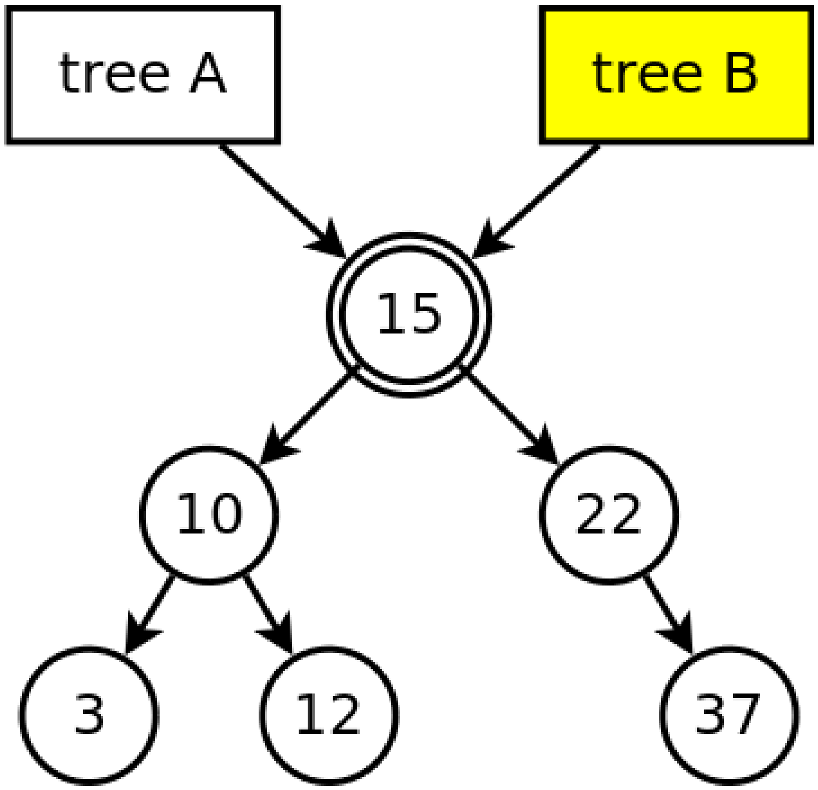
Cloning with structural sharing



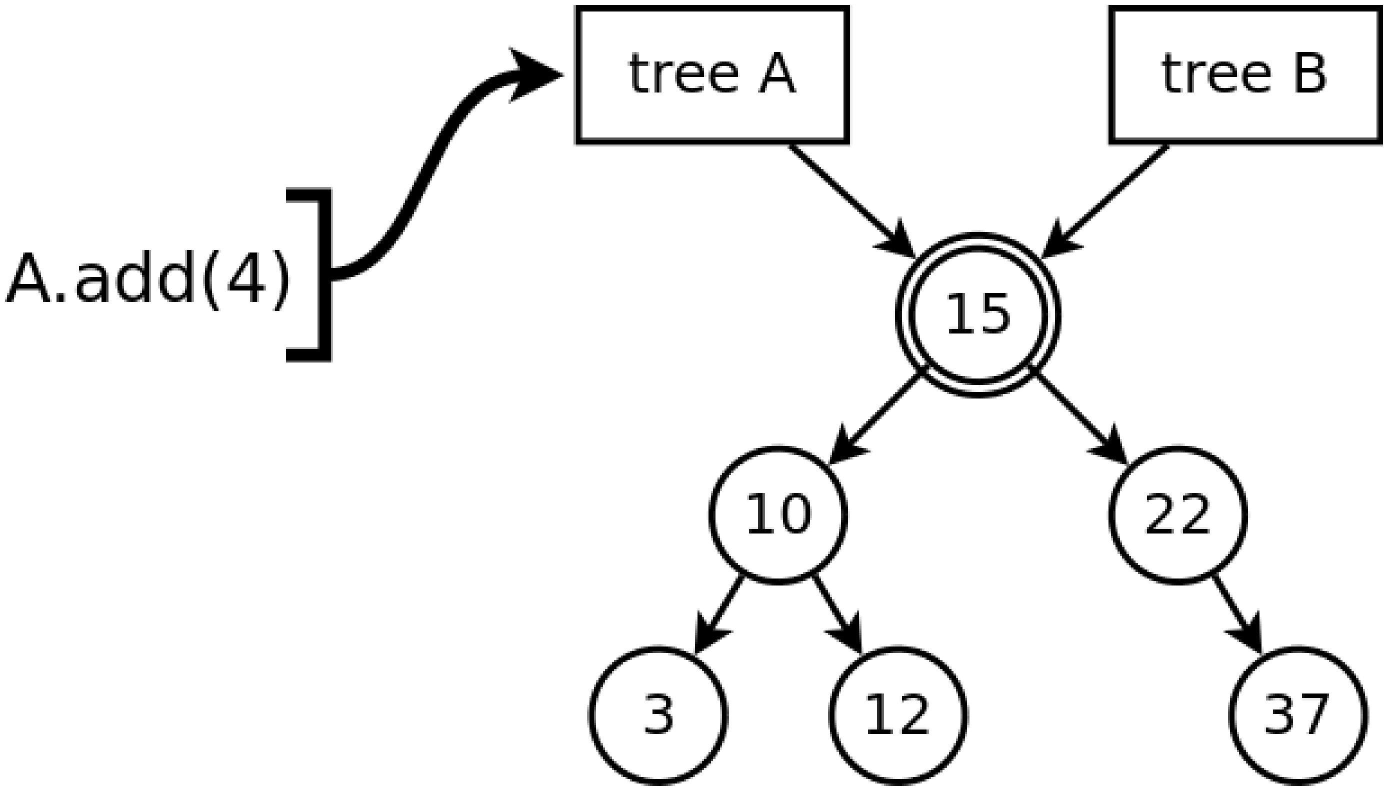
Cloning with structural sharing



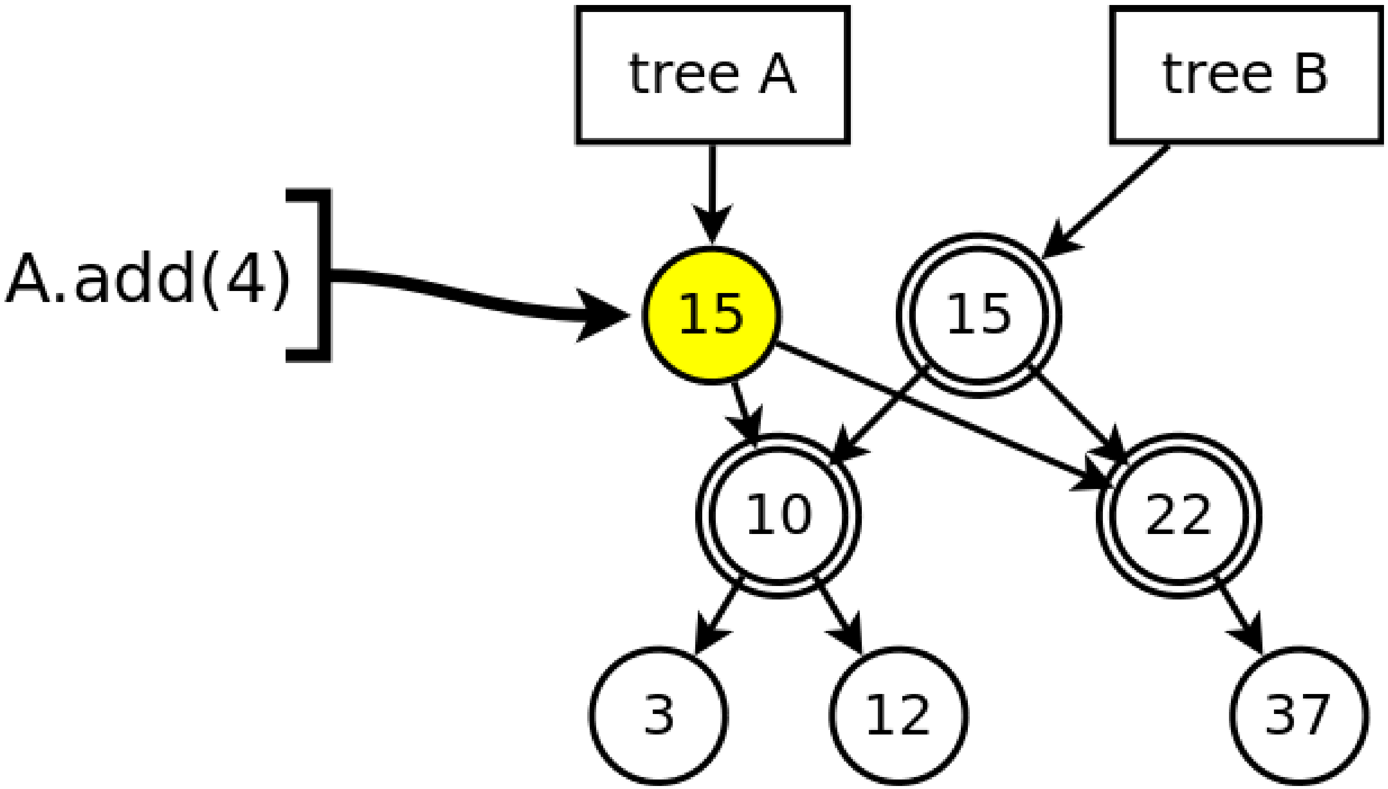
Cloning with structural sharing



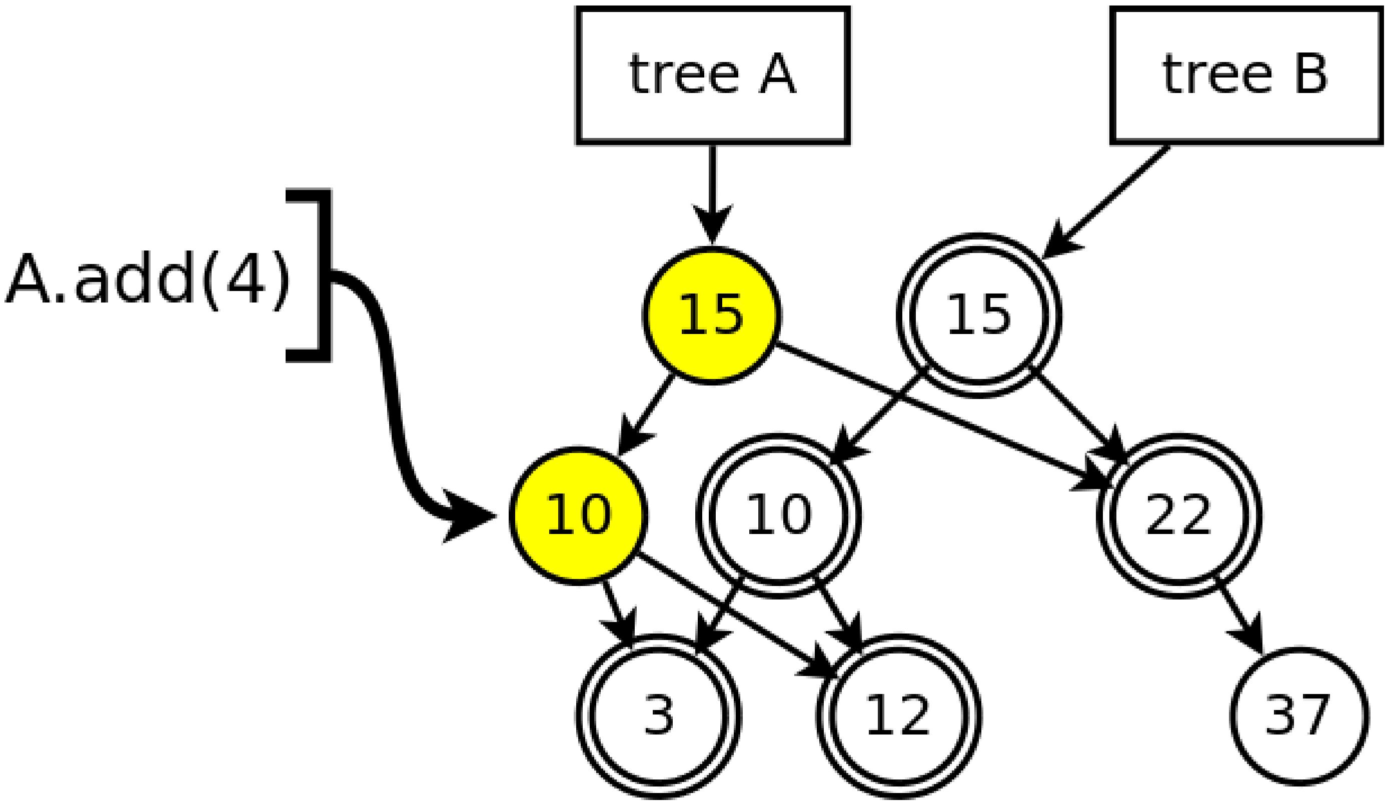
Lazy marking and copy-on-write



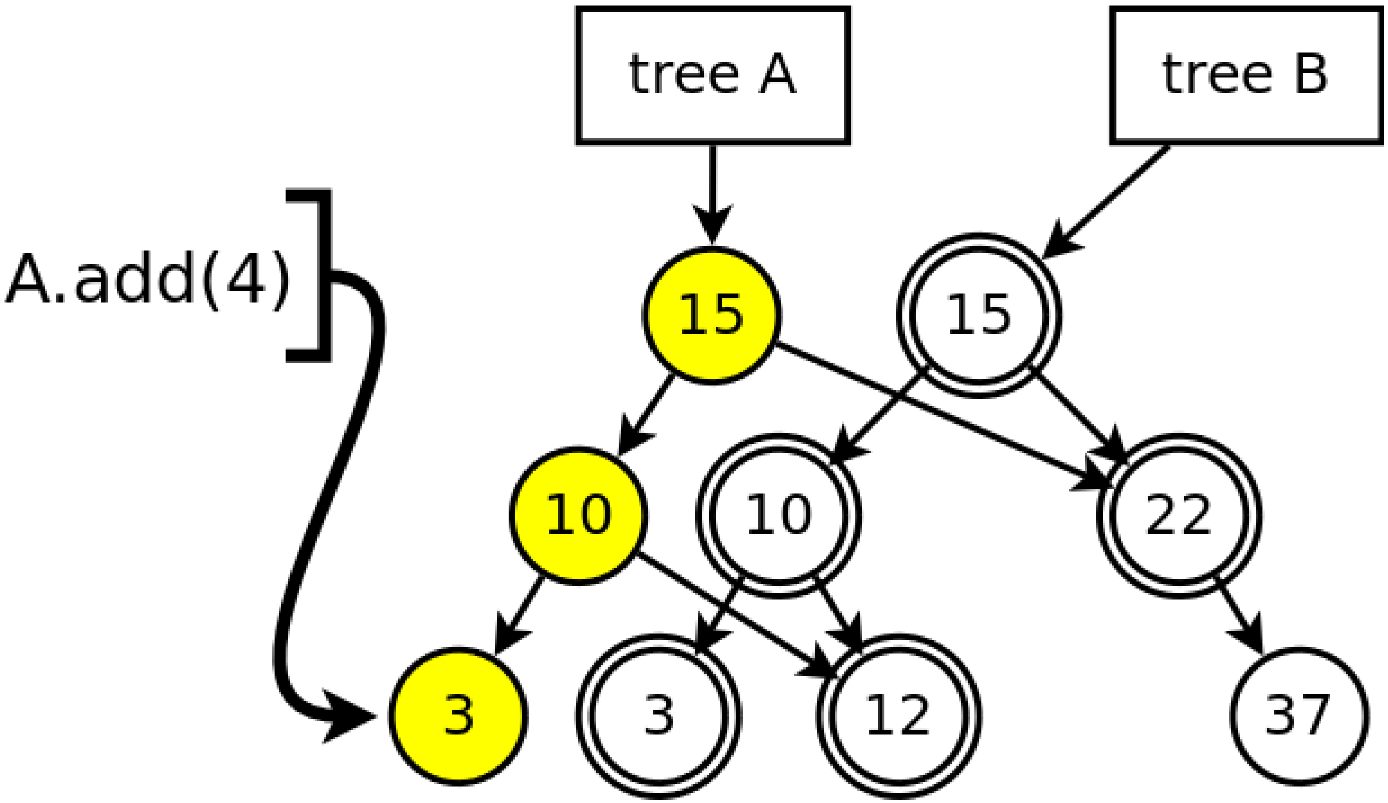
Lazy marking and copy-on-write



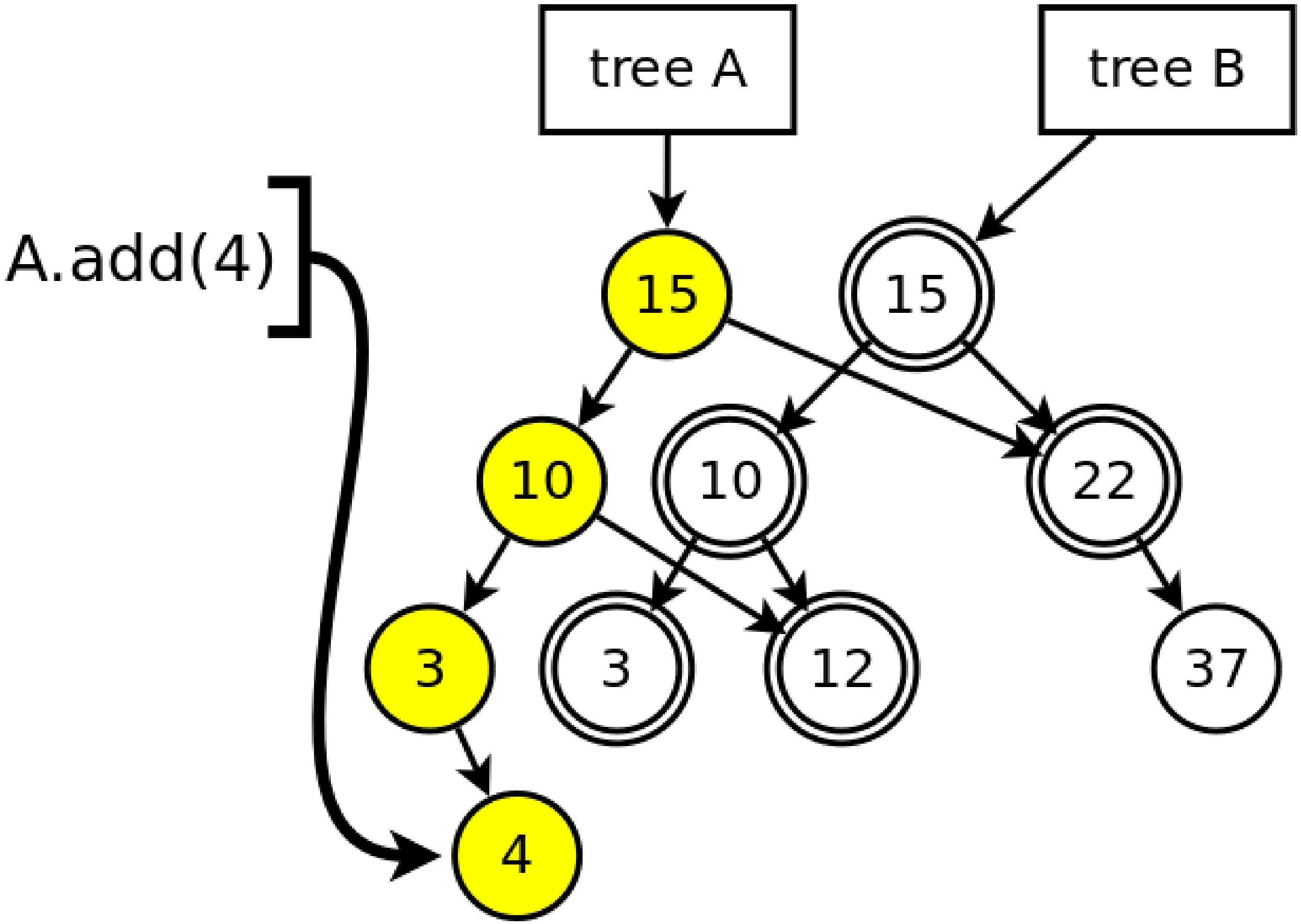
Lazy marking and copy-on-write



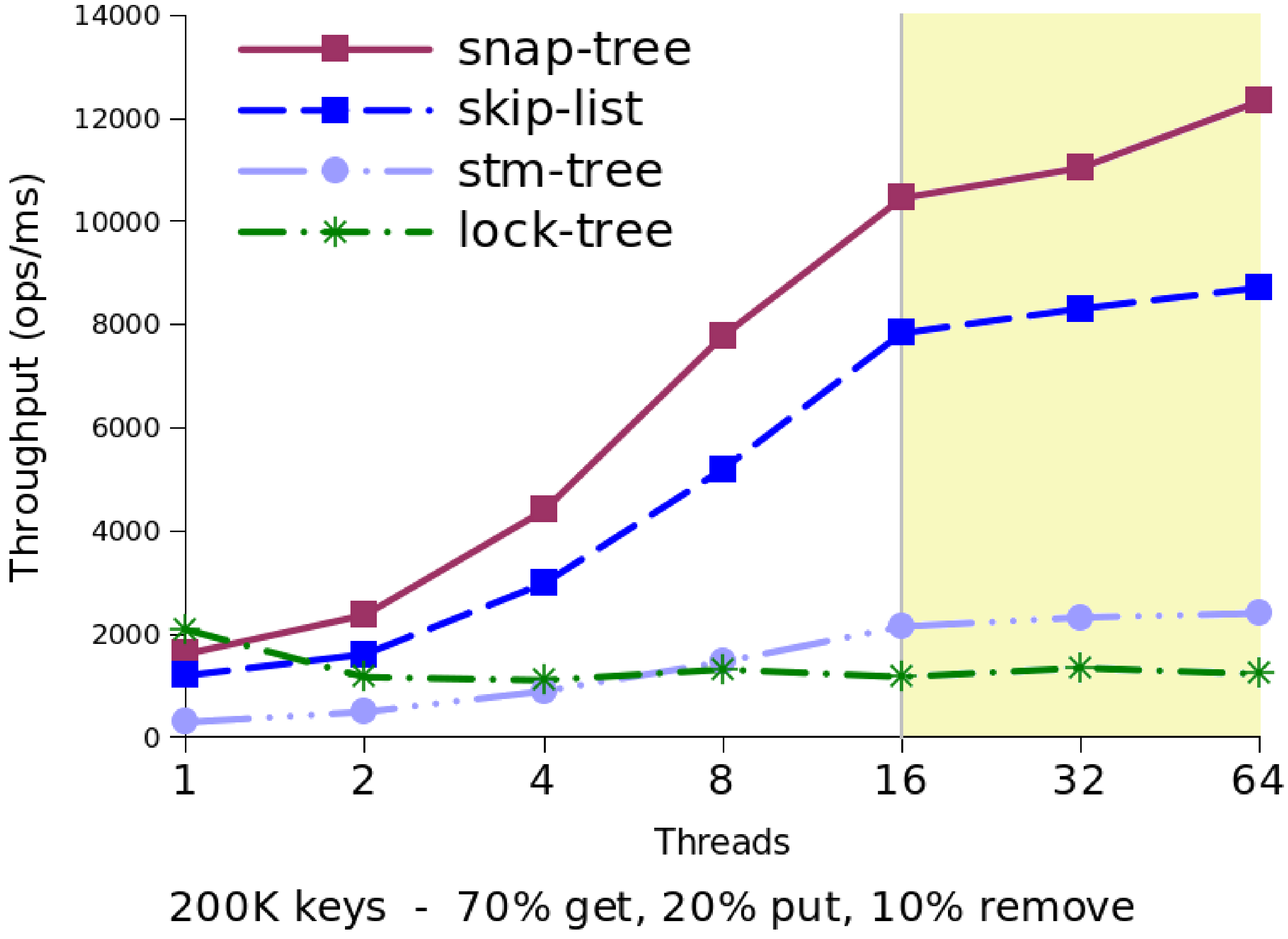
Lazy marking and copy-on-write



Lazy marking and copy-on-write



SnapTree performance



8 cores, 16 hardware threads. Skip-list and lock-tree are from JDK 1.6

Conclusion – Questions?

- Optimistic concurrency tailored for trees
  - Specialization of generic STM techniques
  - Specialization of the tree algorithm
- Good performance and scalability
  - Small penalty for supporting concurrent access
- Fast atomic clone
  - Provides snapshot isolation for iteration

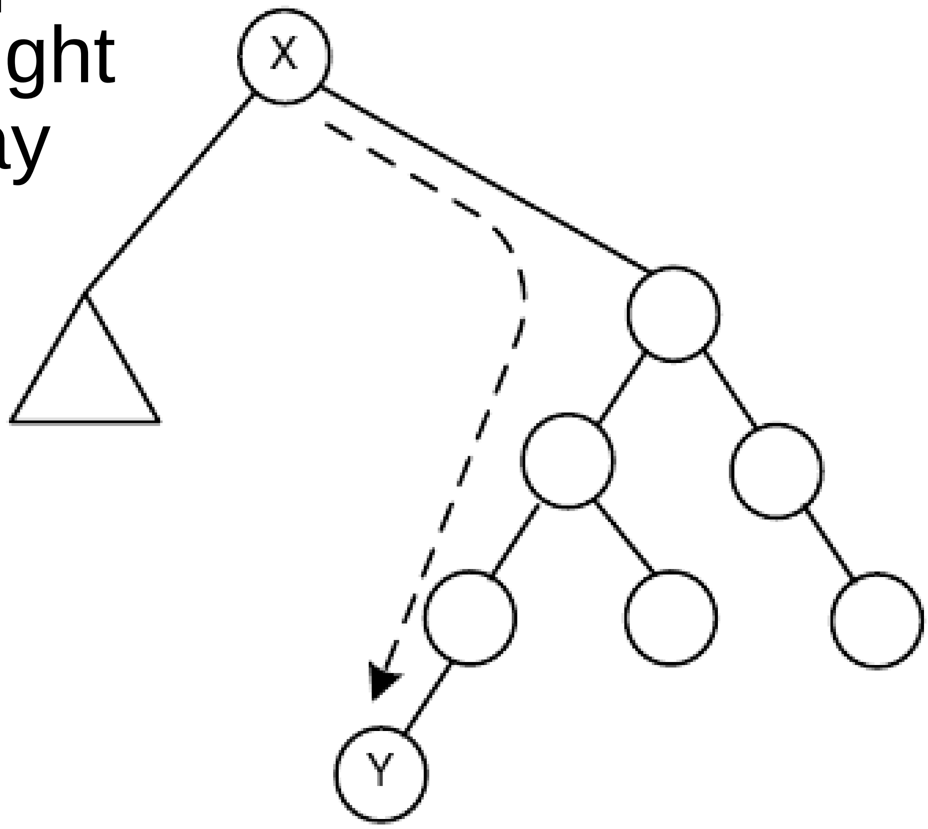
Code available at

http://github.com/nbronson/snaptree

Deleting with fixed-size txns

Nodes with two children cause problems

- Successor must be spliced in atomically, but it might be O(log *n*) hops away
- Many nodes must be shrunk

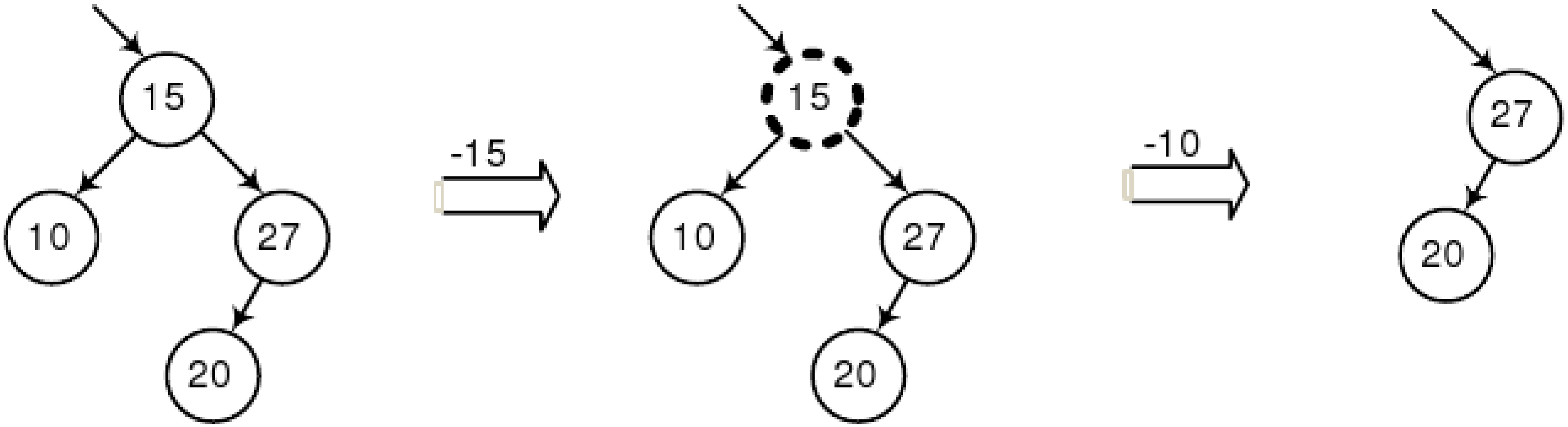


External tree?

- Wastes *n*-1 nodes

"Partially external" trees

- Unlink when convenient
  - During deletion, during rebalancing
- Retain as routing node when inconvenient
  - If fixed-size transaction is not sufficient for unlink



Node counts for randomly built trees