RCU-HTM: Combining RCU with HTM to Implement Highly Efficient Concurrent Binary Search Trees

Dimitrios Siakavaras, Konstantinos Nikas, Georgios Goumas and Nectarios Koziris

National Technical University of Athens (NTUA)

School of Electrical and Computer Engineering (ECE)

Computing Systems Laboratory (CSLab)

{jimsiak,knikas,goumas,nkoziris}@cslab.ece.ntua.gr

http://research.cslab.ece.ntua.gr

PACT 2017

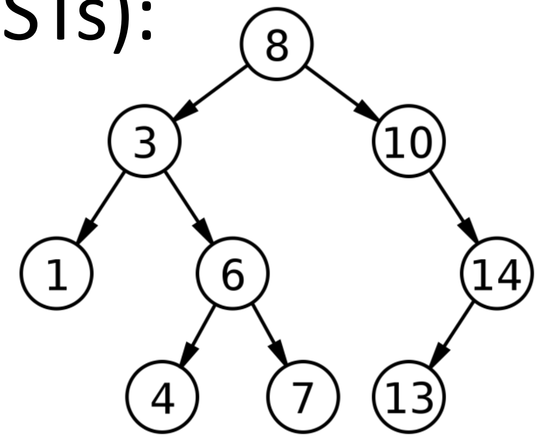
Motivation

- Multi-cores are ubiquitous

- Multi-threaded applications

  -> Concurrent data structures

- Concurrent Binary Search Trees (BSTs):
  - Widely used
  - Linux kernel
  - Database Index



Our Contributions

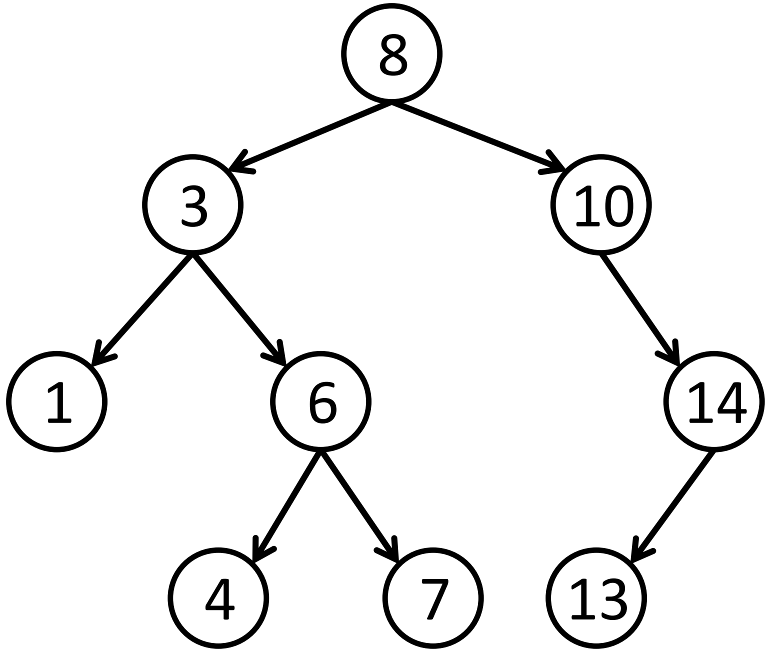
- We introduce ***RCU-HTM***
  - Combines
    1. Read-Copy-Update (RCU)
    2. Hardware Transactional Memory (HTM)
  - Provides
    - Highly efficient concurrent binary search trees

Our Contributions

- We introduce ***RCU-HTM***
  - Combines
    1. Read-Copy-Update (RCU)
    2. Hardware Transactional Memory (HTM)
  - Provides
    - Highly efficient concurrent binary search trees

- We apply **RCU-HTM** in AVL and Red-Black trees
  - 18% better performance, on average
  - Excellent performance on <u>read-only</u> workloads
  - Very good performance on <u>write-intensive</u> workloads

Binary Search Trees (BSTs)



- A classic binary tree with an additional property:
  - Keys in left subtree < root key
  - Keys in right subtree > root key

- Most commonly used to implement *dictionaries:*
  - <key,value> pairs
  - 3 operations: *lookup(key), insert(key, value)* and *delete(key)*

Serial BSTs

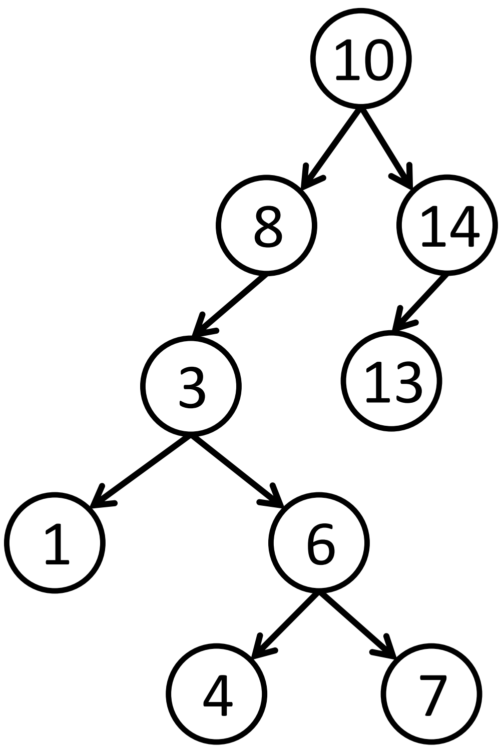
Typical serial BSTs have the following characteristics that boost their performance:

Serial BSTs

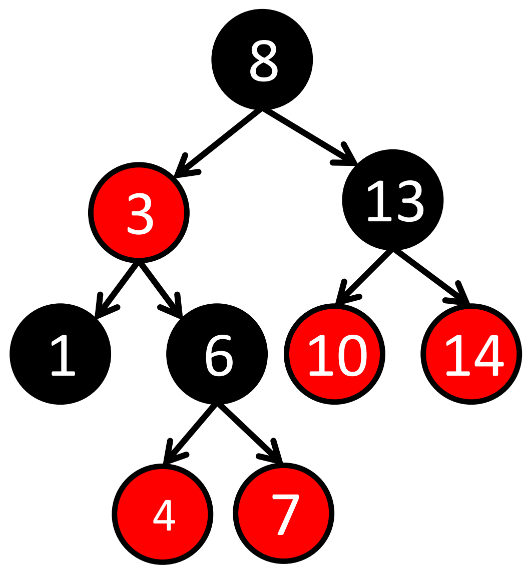
Typical serial BSTs have the following characteristics that boost their performance:

1. Balance

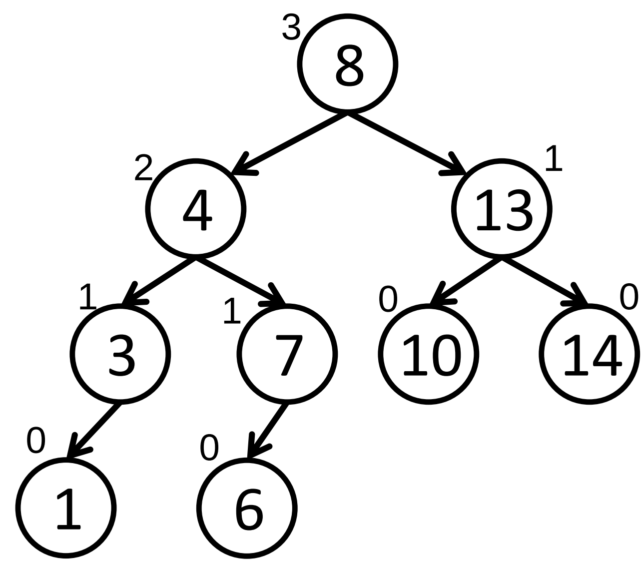
Unbalanced Tree



Red-Black Tree



AVL Tree



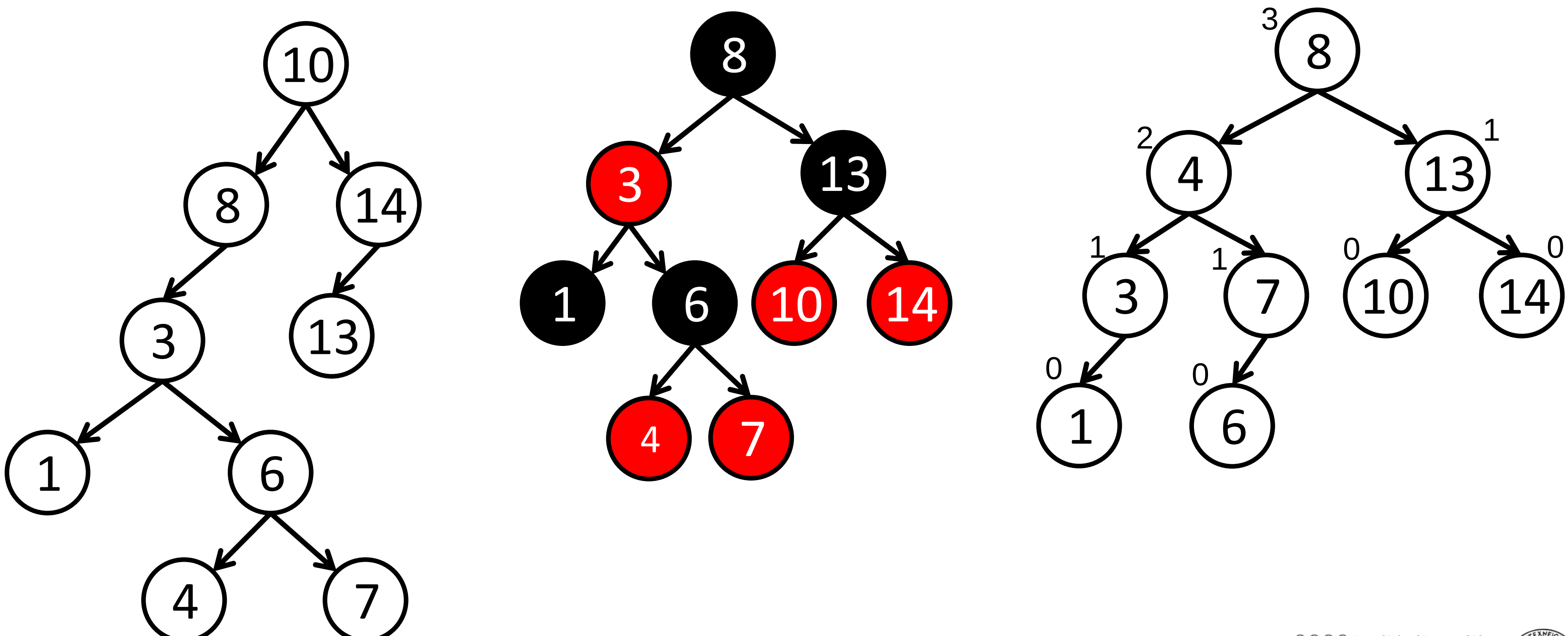
Serial BSTs

Typical serial BSTs have the following characteristics that boost their performance:

1. Balance

\+ Shorter path lengths
\- Rebalancing requires additional effort after insertions/deletions

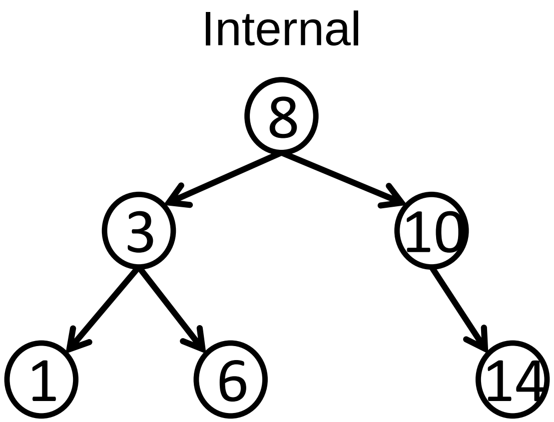
Unbalanced Tree | Red-Black Tree | AVL Tree

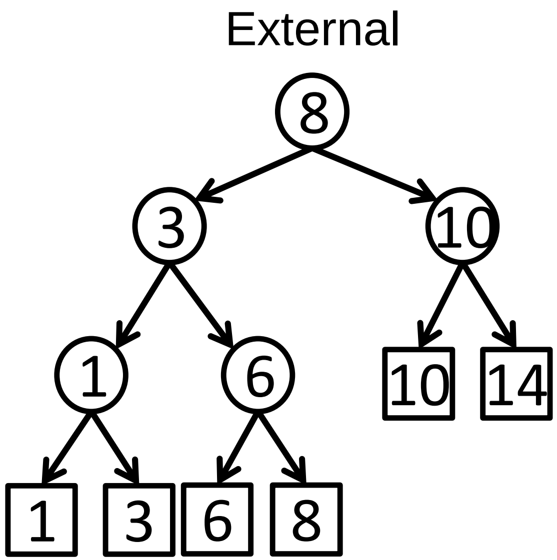


Serial BSTs

Typical serial BSTs have the following characteristics that boost their performance:

1. Balance
2. Internal



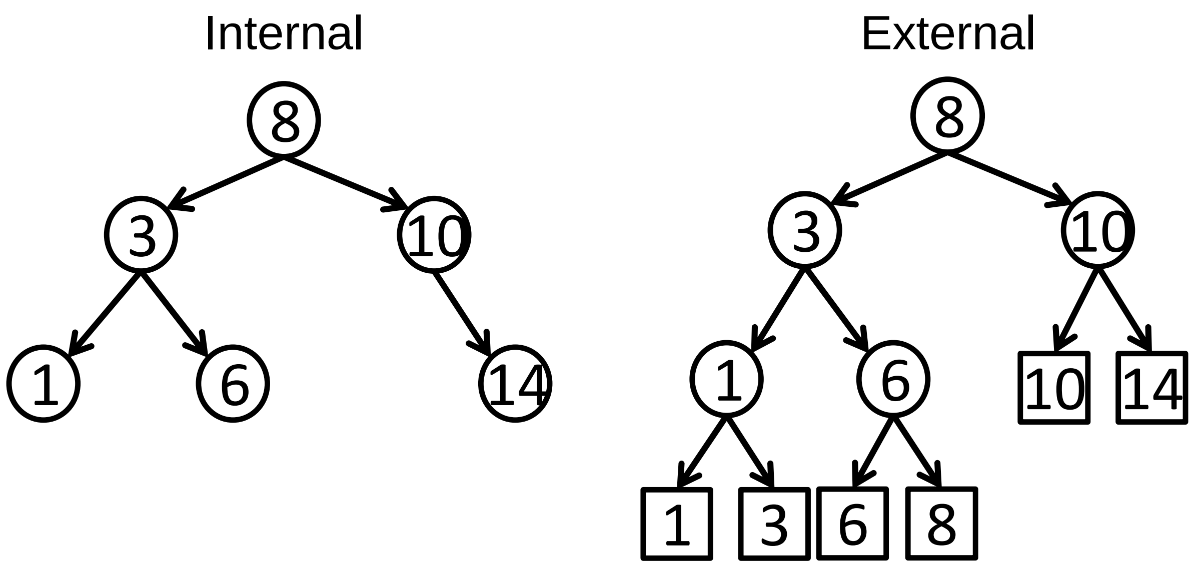


Serial BSTs

Typical serial BSTs have the following characteristics that boost their performance:

1. Balance

2. Internal

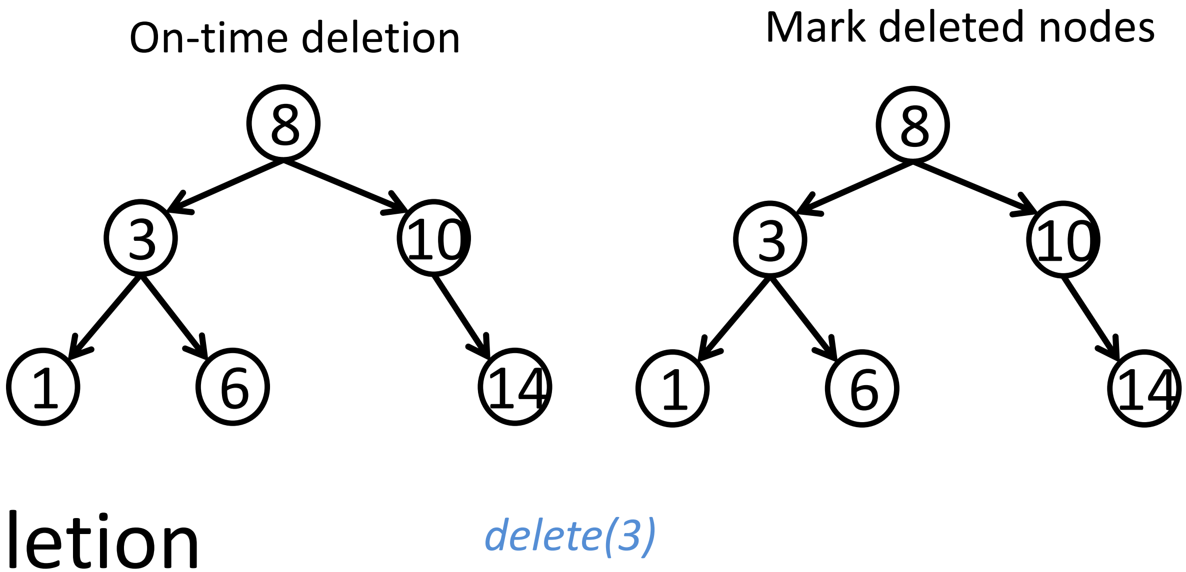


+ Shorter path lengths
+ Less memory overhead
- Complexity of *delete()* operation

Serial BSTs

Typical serial BSTs have the following characteristics that boost their performance:

1. Balance
2. Internal
3. On-time deletion

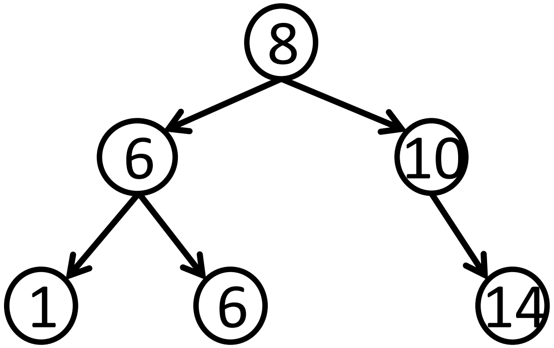


Serial BSTs

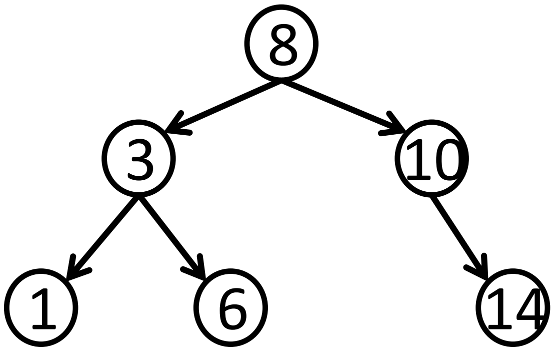
Typical serial BSTs have the following characteristics that boost their performance:

1. Balance
2. Internal
3. On-time deletion

On-time deletion



Mark deleted nodes



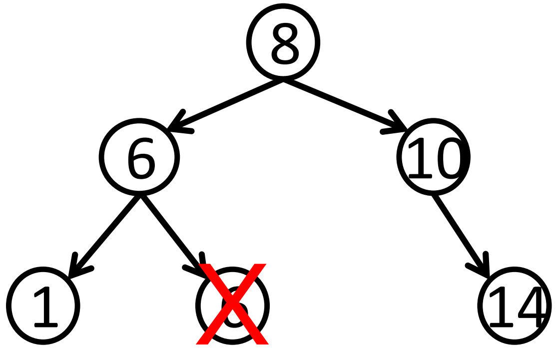
delete(3)

Serial BSTs

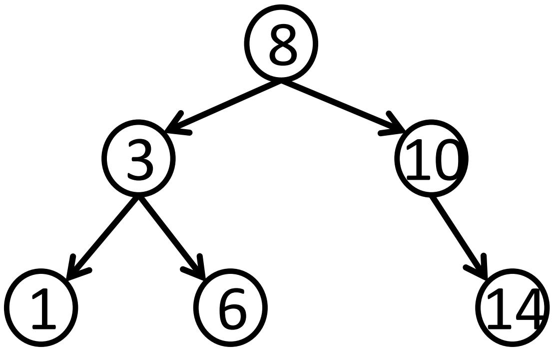
Typical serial BSTs have the following characteristics that boost their performance:

1. Balance
2. Internal
3. On-time deletion

On-time deletion



Mark deleted nodes



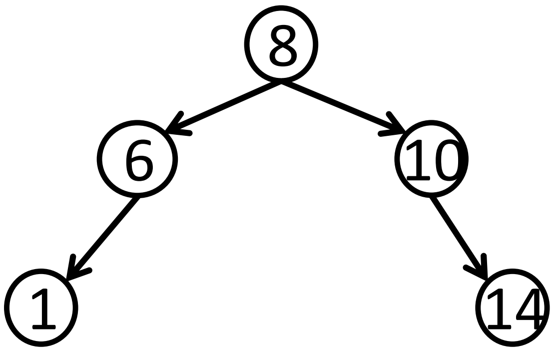
delete(3)

Serial BSTs

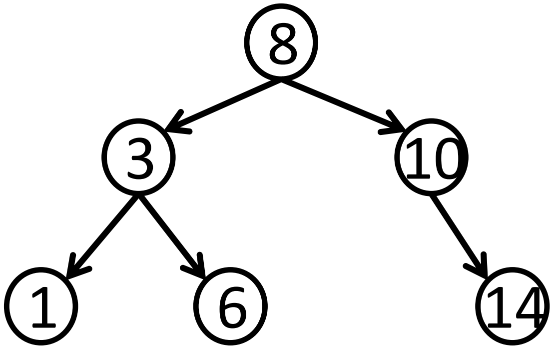
Typical serial BSTs have the following characteristics that boost their performance:

1. Balance
2. Internal
3. On-time deletion

On-time deletion



Mark deleted nodes

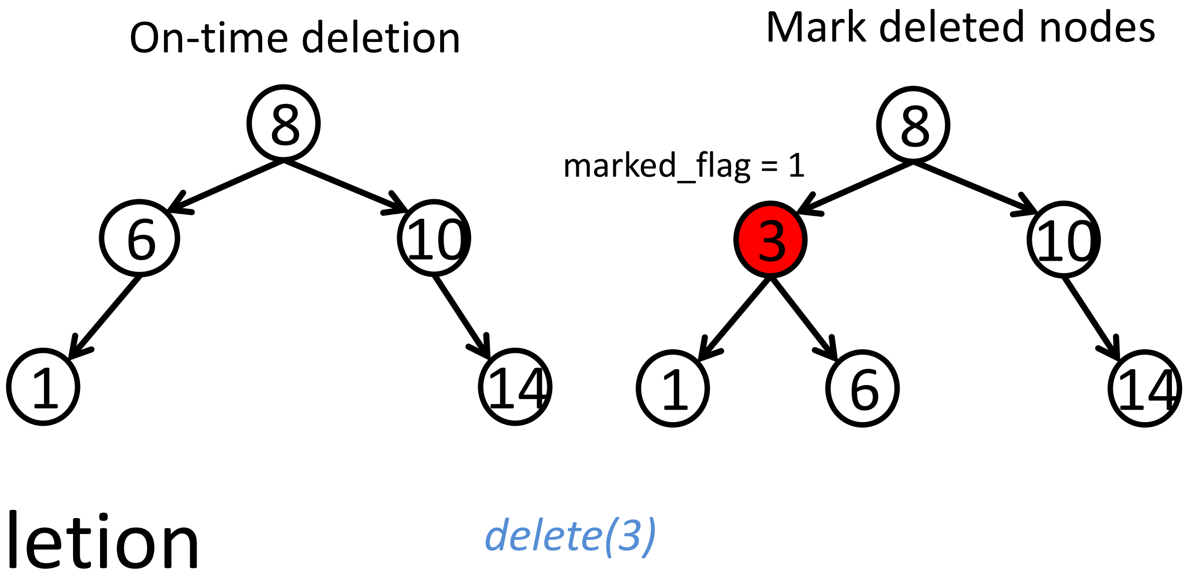


delete(3)

Serial BSTs

Typical serial BSTs have the following characteristics that boost their performance:

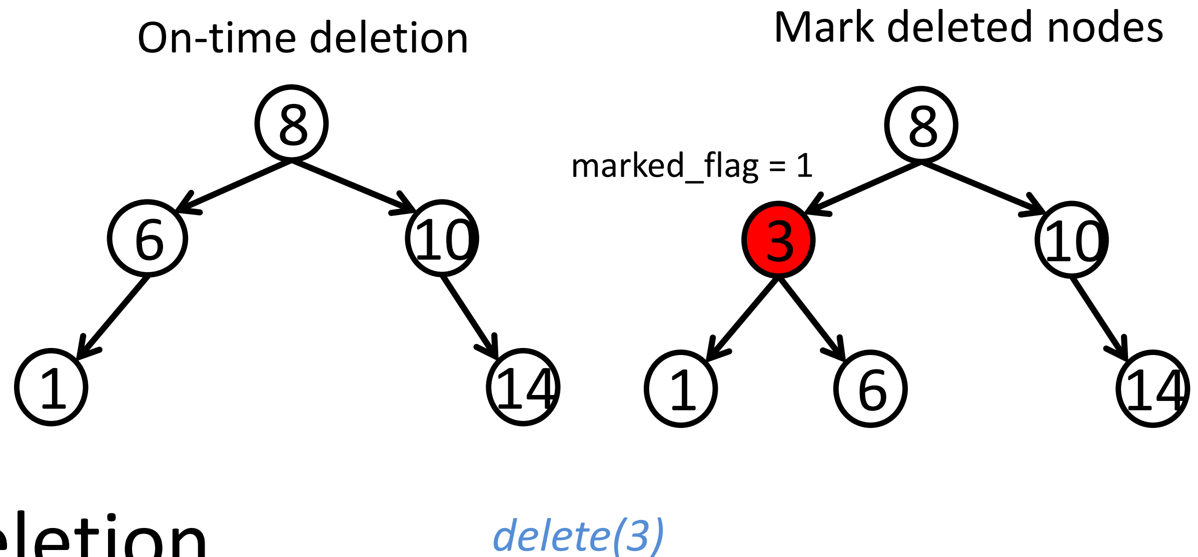
1. Balance
2. Internal
3. On-time deletion



Serial BSTs

Typical serial BSTs have the following characteristics that boost their performance:

1. Balance
2. Internal
3. On-time deletion



+ Shorter path lengths
+ Less memory overhead
- Complexity of *delete()* operation

Concurrent BSTs

These 3 characteristics:

\+ Boost the performance of serial BSTs

\- Are difficult to implement in concurrent BSTs

Why?

Rebalancing and **internal deletion** require multiple node modifications to be performed in a “single atomic” step

Concurrent BSTs

In concurrent BSTs 2 more characteristics are of high importance:

1. Asynchronized traversals
   - The most common operation -> need to be fast
   - Avoid synchronization overhead
2. Multiple updaters
   - Updates on disjoint parts of the tree should be allowed to execute concurrently

RCU-HTM vs previous works

	Name	Balanced	Internal	On-time deletion	Asynchronized traversals	Multiple updaters
Lock-free	Ellen et al. [PODC'10]					
	Howley et al. [SPAA'12]					
	Natarajan et al. [PPoPP'14]					
	Chatterjee et al. [PODC'14]					
	Brown et al. [PPoPP'14]					
Locks	Bronson et al. [PPoPP'10]					
	Crain et al. [EuroPar'13]					
	Drachsler et al. [PPoPP'14]					
RCU	Howard et al. [CCPE'14]					
	Arbel et al. [PODC'14]					
TM	Crain et al. [PPoPP'14]					
	Avni et al. [TRANSACT'14]					
	RCU-HTM					

National Technical University of Athens CSLab

RCU-HTM vs previous works

	Name	Balanced	Internal	On-time deletion	Asynchronized traversals	Multiple updaters
Lock-free	Ellen et al. [PODC’10]	✗	✗	✓	✓	✓
	Howley et al. [SPAA’12]	✗	✓	✗	✗	✓
	Natarajan et al. [PPoPP’14]	✗	✗	✓	✓	✓
	Chatterjee et al. [PODC’14]	✗	✓	✗	✗	✓
	Brown et al. [PPoPP’14]	✗	✗	✓	✓	✓
Locks	Bronson et al. [PPoPP’10]					
	Crain et al. [EuroPar’13]					
	Drachsler et al. [PPoPP’14]					
RCU	Howard et al. [CCPE’14]					
	Arbel et al. [PODC’14]					
TM	Crain et al. [PPoPP’14]					
	Avni et al. [TRANSACT’14]					
	RCU-HTM					

RCU-HTM vs previous works

	Name	Balanced	Internal	On-time deletion	Asynchronized traversals	Multiple updaters
Lock-free	Ellen et al. [PODC’10]	✗	✗	✓	✓	✓
	Howley et al. [SPAA’12]	✗	✓	✗	✗	✓
	Natarajan et al. [PPoPP’14]	✗	✗	✓	✓	✓
	Chatterjee et al. [PODC’14]	✗	✓	✗	✗	✓
	Brown et al. [PPoPP’14]	✗	✗	✓	✓	✓
Locks	Bronson et al. [PPoPP’10]	✗	✗	✗	✗	✓
	Crain et al. [EuroPar’13]	✗	✗	✗	✓	✓
	Drachsler et al. [PPoPP’14]	✗	✓	✓	✓	✓
RCU	Howard et al. [CCPE’14]					
	Arbel et al. [PODC’14]					
TM	Crain et al. [PPoPP’14]					
	Avni et al. [TRANSACT’14]					
	RCU-HTM					

National Technical University of Athens CSLab

RCU-HTM vs previous works

	Name	Balanced	Internal	On-time deletion	Asynchronized traversals	Multiple updaters
Lock-free	Ellen et al. [PODC'10]	✗	✗	✓	✓	✓
Lock-free	Howley et al. [SPAA'12]	✗	✓	✗	✗	✓
Lock-free	Natarajan et al. [PPoPP'14]	✗	✗	✓	✓	✓
Lock-free	Chatterjee et al. [PODC'14]	✗	✓	✗	✗	✓
Lock-free	Brown et al. [PPoPP'14]	✗	✗	✓	✓	✓
Locks	Bronson et al. [PPoPP'10]	✗	✗	✗	✗	✓
Locks	Crain et al. [EuroPar'13]	✗	✗	✗	✓	✓
Locks	Drachsler et al. [PPoPP'14]	✗	✓	✓	✓	✓
RCU	Howard et al. [CCPE'14]	✓	✓	✗	✓	✗
RCU	Arbel et al. [PODC'14]	✗	✓	✗	✓	✓
TM	Crain et al. [PPoPP'14]					
TM	Avni et al. [TRANSACT'14]					
	RCU-HTM					

RCU-HTM vs previous works

	Name	Balanced	Internal	On-time deletion	Asynchronized traversals	Multiple updaters
Lock-free	Ellen et al. [PODC'10]	✗	✗	✓	✓	✓
Lock-free	Howley et al. [SPAA'12]	✗	✓	✗	✗	✓
Lock-free	Natarajan et al. [PPoPP'14]	✗	✗	✓	✓	✓
Lock-free	Chatterjee et al. [PODC'14]	✗	✓	✗	✗	✓
Lock-free	Brown et al. [PPoPP'14]	✗	✗	✓	✓	✓
Locks	Bronson et al. [PPoPP'10]	✗	✗	✗	✗	✓
Locks	Crain et al. [EuroPar'13]	✗	✗	✗	✓	✓
Locks	Drachsler et al. [PPoPP'14]	✗	✓	✓	✓	✓
RCU	Howard et al. [CCPE'14]	✓	✓	✗	✓	✗
RCU	Arbel et al. [PODC'14]	✗	✓	✗	✓	✓
TM	Crain et al. [PPoPP'14]	✗	✓	✗	✗	✓
TM	Avni et al. [TRANSACT'14]	✓	✓	✓	✗	✓
	RCU-HTM					

National Technical University of Athens CSLab

RCU-HTM vs previous works

	Name	Balanced	Internal	On-time deletion	Asynchronized traversals	Multiple updaters
Lock-free	Ellen et al. [PODC’10]	✗	✗	✓	✓	✓
	Howley et al. [SPAA’12]	✗	✓	✗	✗	✓
	Natarajan et al. [PPoPP’14]	✗	✗	✓	✓	✓
	Chatterjee et al. [PODC’14]	✗	✓	✗	✗	✓
	Brown et al. [PPoPP’14]	✗	✗	✓	✓	✓
Locks	Bronson et al. [PPoPP’10]	✗	✗	✗	✗	✓
	Crain et al. [EuroPar’13]	✗	✗	✗	✓	✓
	Drachsler et al. [PPoPP’14]	✗	✓	✓	✓	✓
RCU	Howard et al. [CCPE’14]	✓	✓	✗	✓	✗
	Arbel et al. [PODC’14]	✗	✓	✗	✓	✓
TM	Crain et al. [PPoPP’14]	✗	✓	✗	✗	✓
	Avni et al. [TRANSACT’14]	✓	✓	✓	✗	✓
	RCU-HTM	✓	✓	✓	✓	✓

RCU-HTM

By combining RCU and HTM, **RCU-HTM** enables the implementation of:

1. Balanced
2. Internal BSTs with
3. On-time deletion

That also provide:

4. Asynchronized traversals
5. Multiple concurrent updaters

RCU-HTM

RCU-HTM: Asynchronized traversals

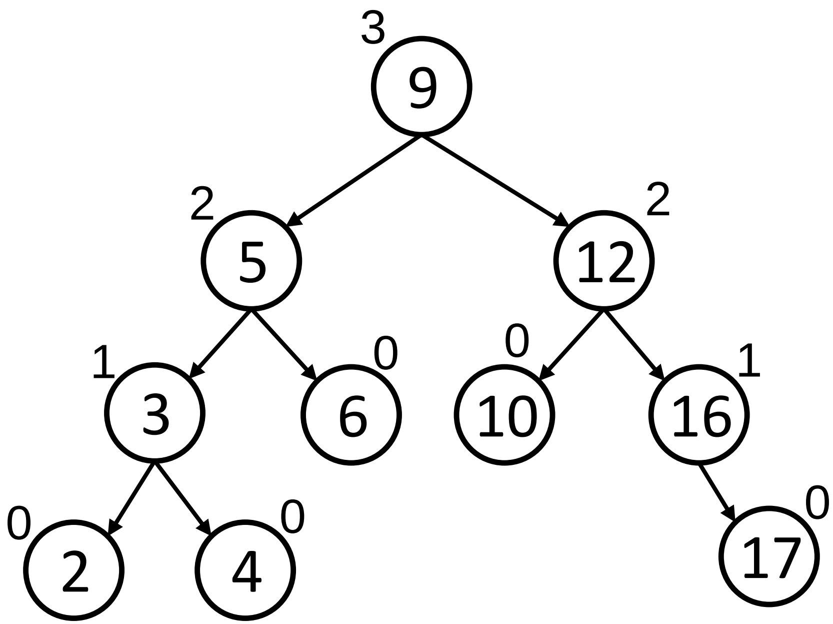
RCU-HTM

RCU-HTM: Asynchronized traversals

RCU-HTM

Why do we need synchronization for the traversals at the first place?

- Concurrent rotations may lead traversals to a wrong path

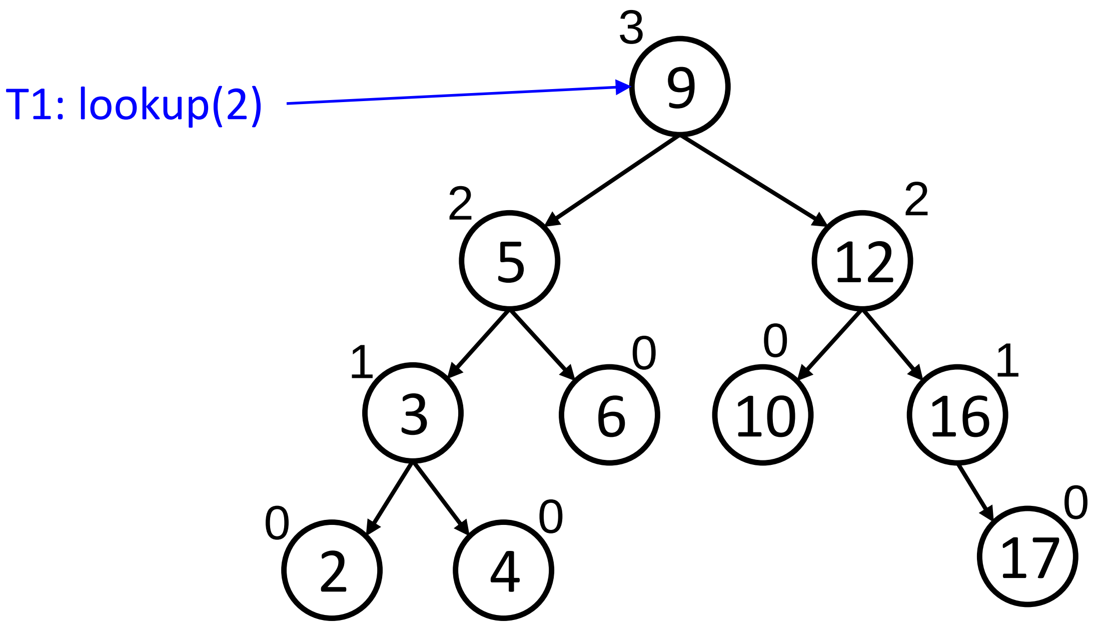


RCU-HTM: Asynchronized traversals

RCU-HTM

Why do we need synchronization for the traversals at the first place?

- Concurrent rotations may lead traversals to a wrong path

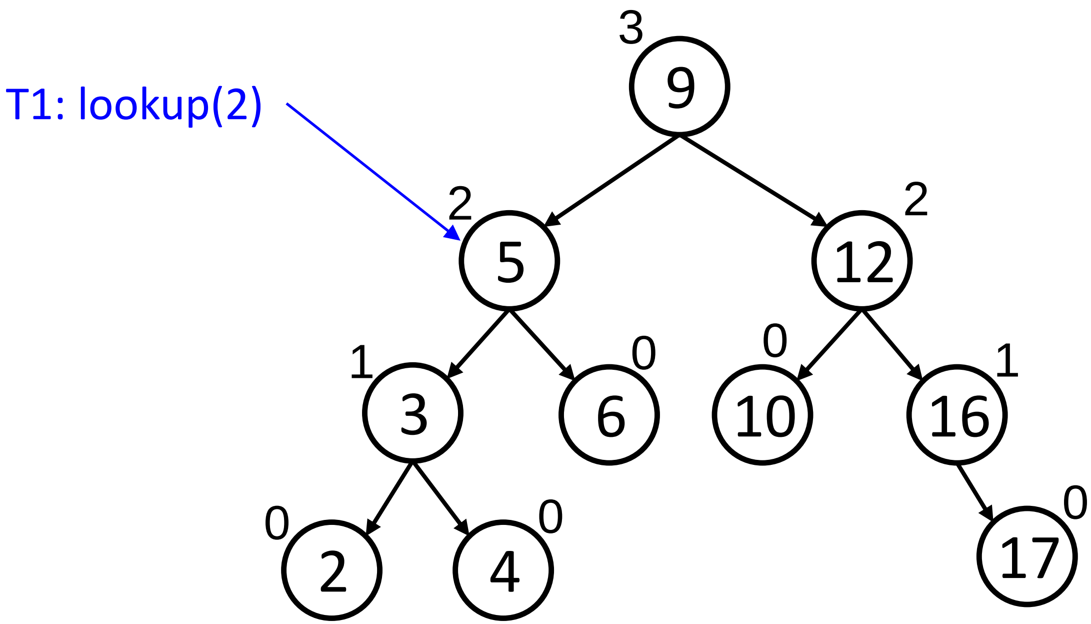


RCU-HTM: Asynchronized traversals

RCU-HTM

Why do we need synchronization for the traversals at the first place?

- Concurrent rotations may lead traversals to a wrong path



RCU-HTM: Asynchronized traversals

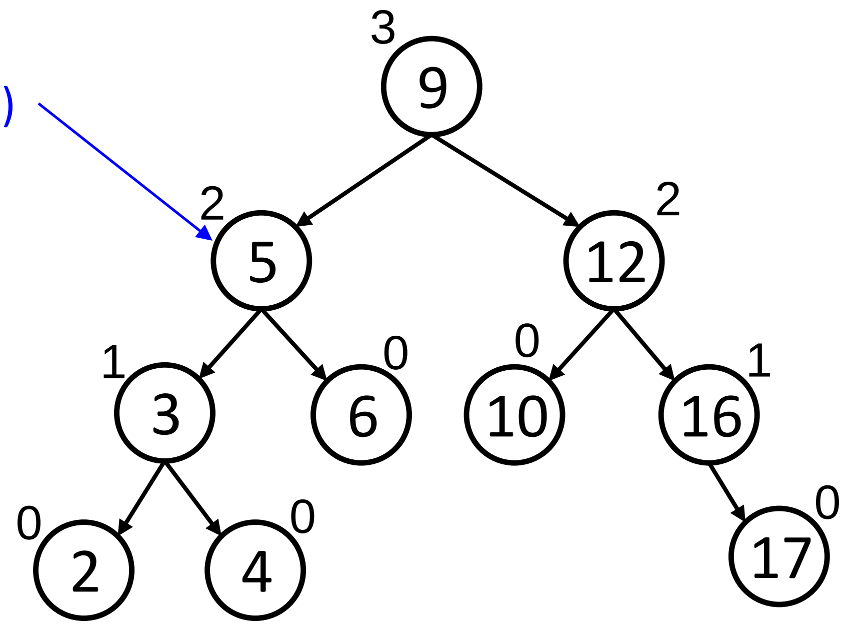
RCU-HTM

Why do we need synchronization for the traversals at the first place?

- Concurrent rotations may lead traversals to a wrong path

T1: lookup(2)

T2: insert(1)



RCU-HTM: Asynchronized traversals

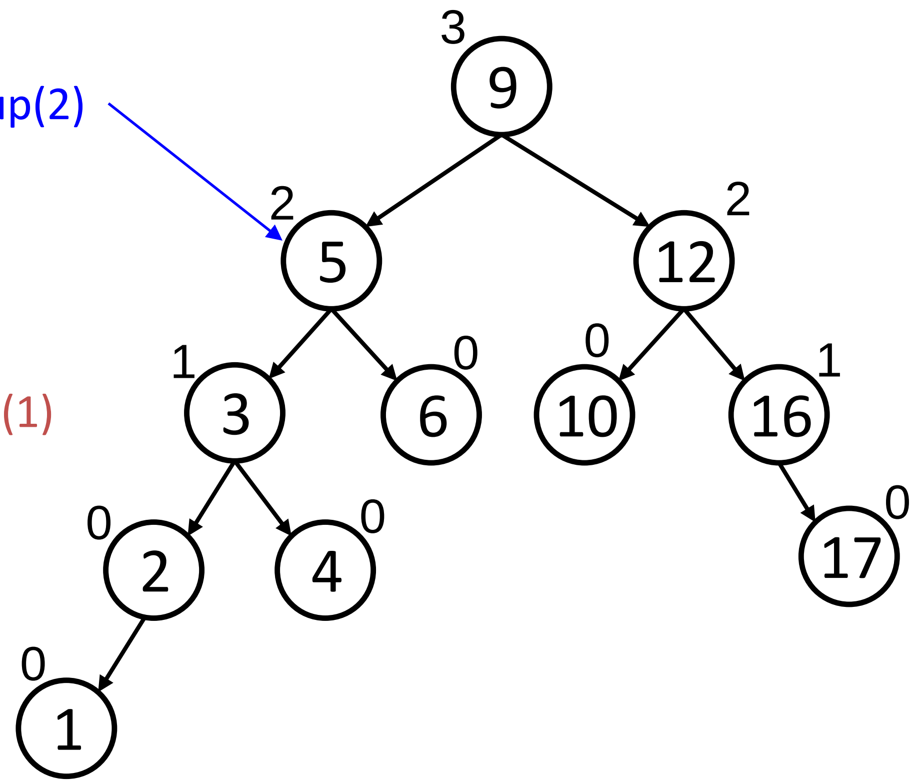
RCU-HTM

Why do we need synchronization for the traversals at the first place?

- Concurrent rotations may lead traversals to a wrong path

T1: lookup(2)

T2: insert(1)



RCU-HTM: Asynchronized traversals

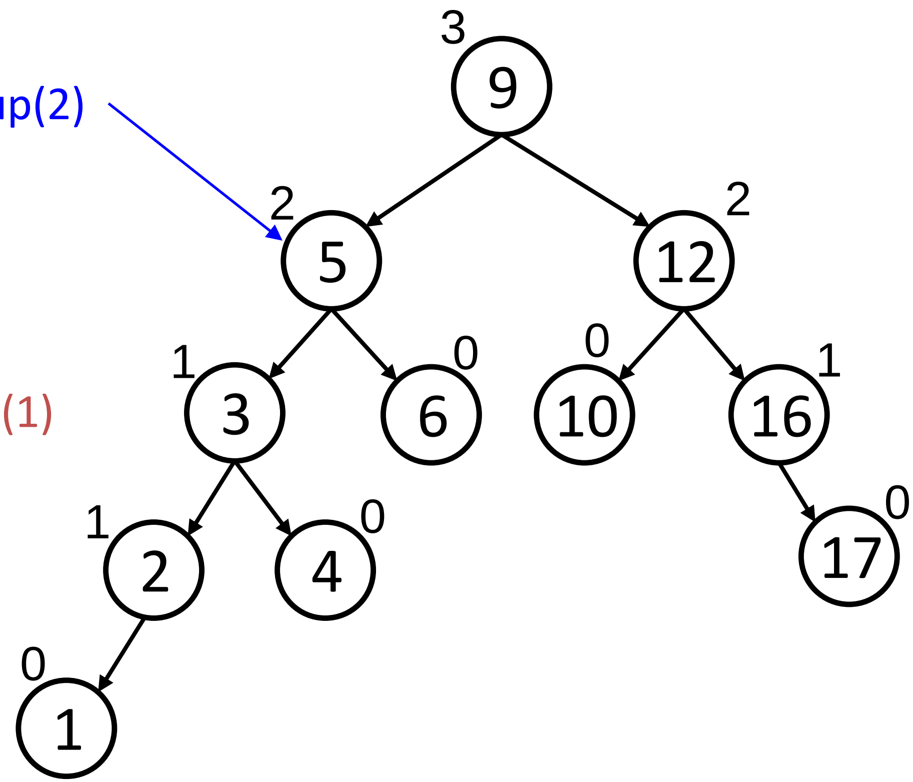
RCU-HTM

Why do we need synchronization for the traversals at the first place?

- Concurrent rotations may lead traversals to a wrong path

T1: lookup(2)

T2: insert(1)



National Technical University of Athens CSLab

RCU-HTM: Asynchronized traversals

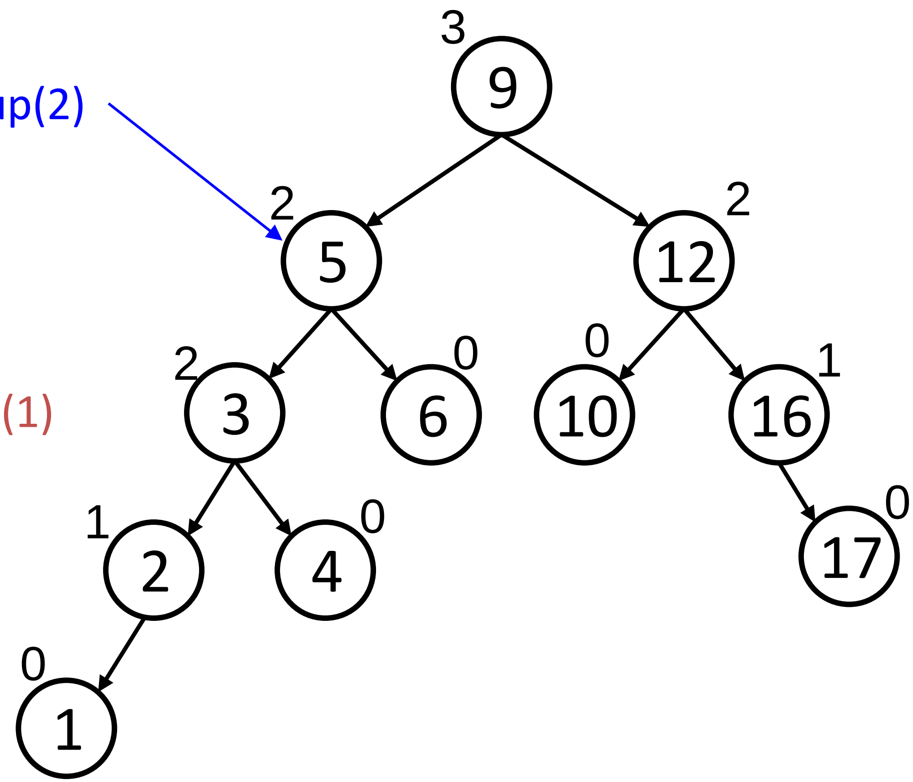
RCU-HTM

Why do we need synchronization for the traversals at the first place?

- Concurrent rotations may lead traversals to a wrong path

T1: lookup(2)

T2: insert(1)



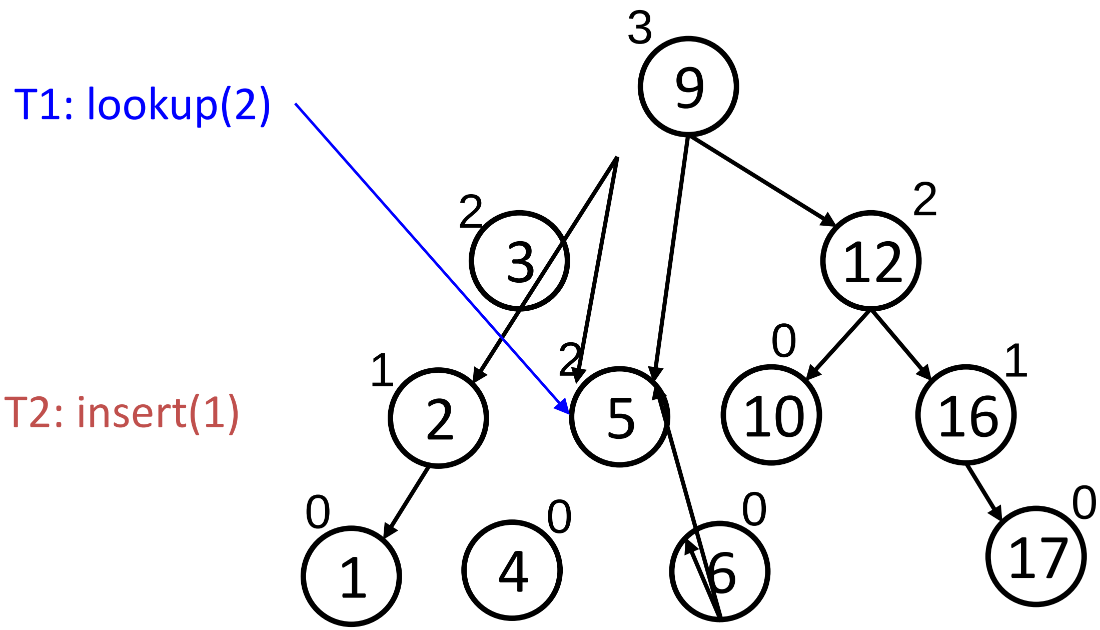
National Technical University of Athens CSLab

RCU-HTM: Asynchronized traversals

RCU-HTM

Why do we need synchronization for the traversals at the first place?

- Concurrent rotations may lead traversals to a wrong path

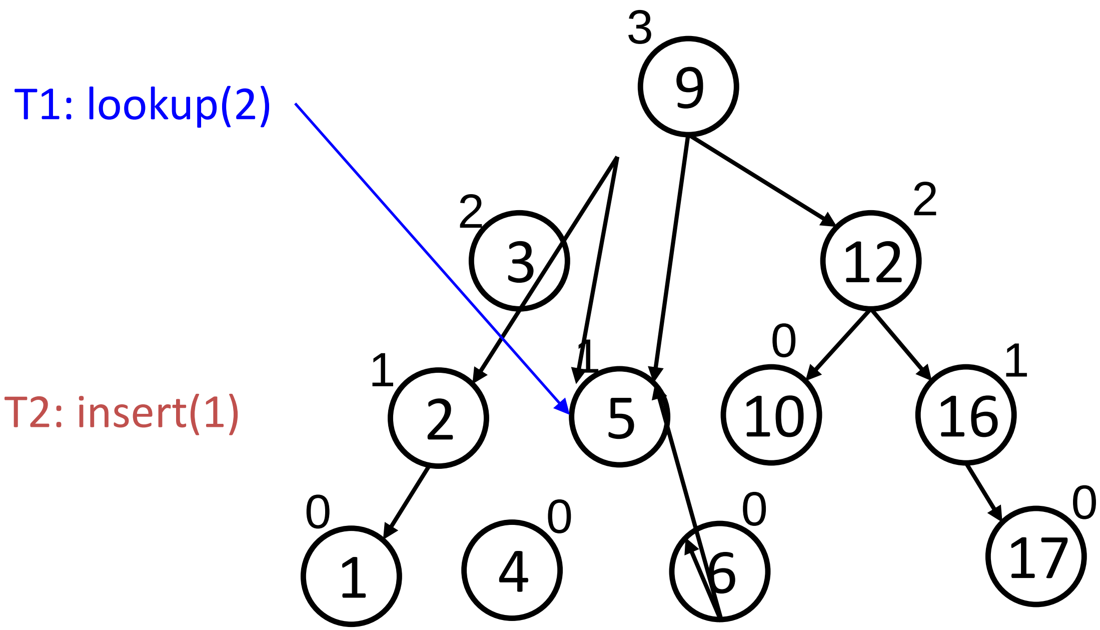


RCU-HTM: Asynchronized traversals

RCU-HTM

Why do we need synchronization for the traversals at the first place?

- Concurrent rotations may lead traversals to a wrong path

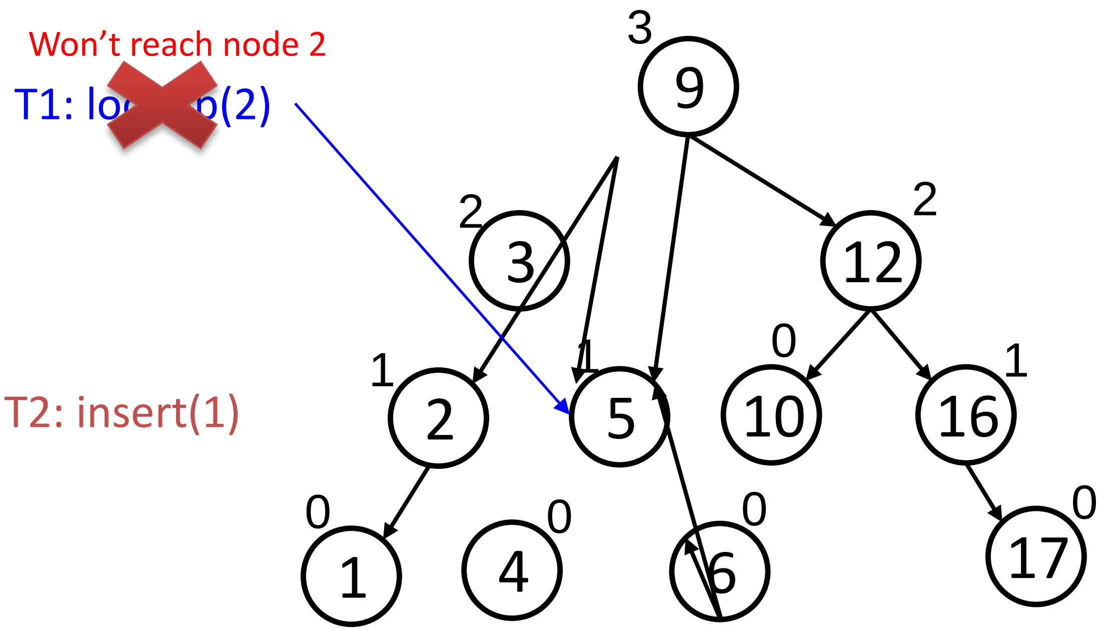


RCU-HTM: Asynchronized traversals

RCU-HTM

Why do we need synchronization for the traversals at the first place?

- Concurrent rotations may lead traversals to a wrong path

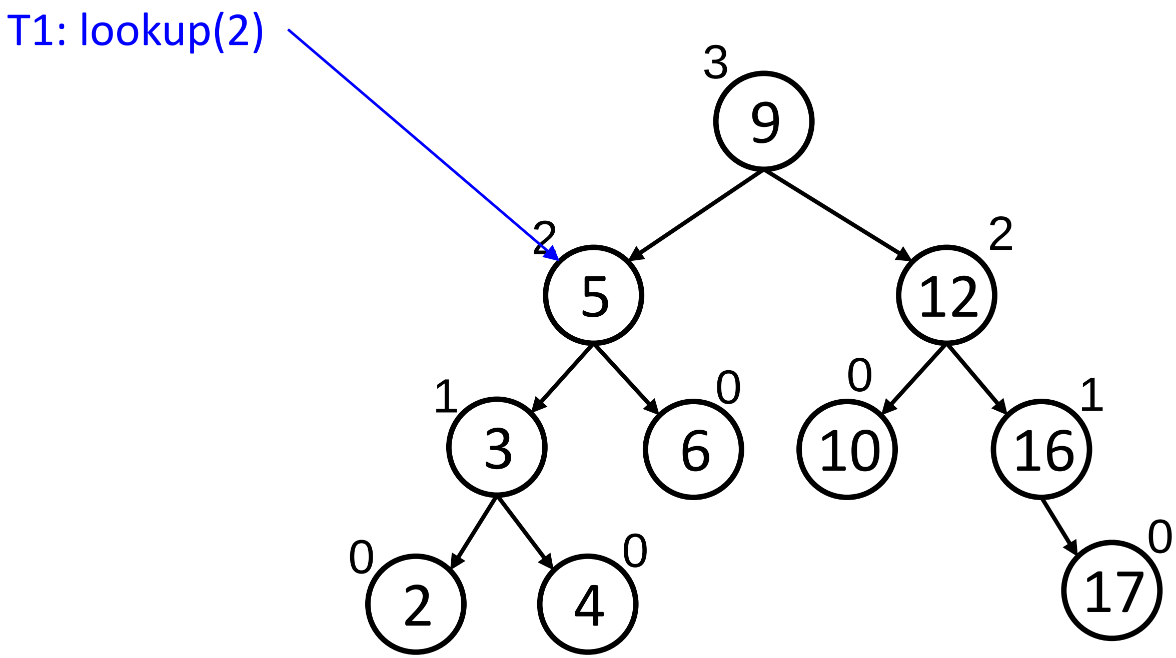


RCU-HTM: Asynchronized traversals

RCU-HTM

How does RCU avoids erroneous executions while allowing asynchronized traversals?

- Assume a single updater for now



RCU-HTM: Asynchronized traversals

RCU-HTM

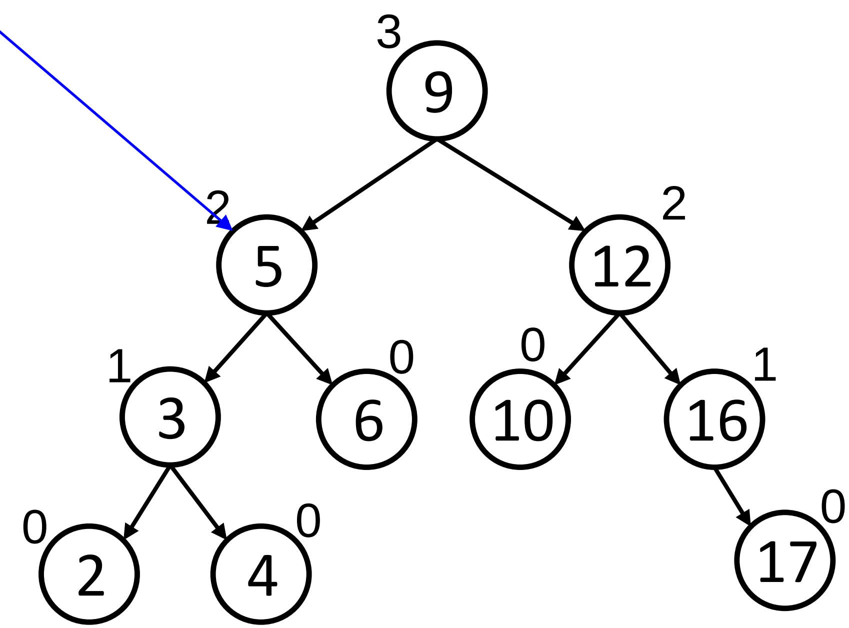
How does RCU avoids erroneous executions while allowing asynchronized traversals?

- Assume a single updater for now

T1: lookup(2)

1. Updaters create copies of the modified parts

T2: insert(1)



RCU-HTM: Asynchronized traversals

RCU-HTM

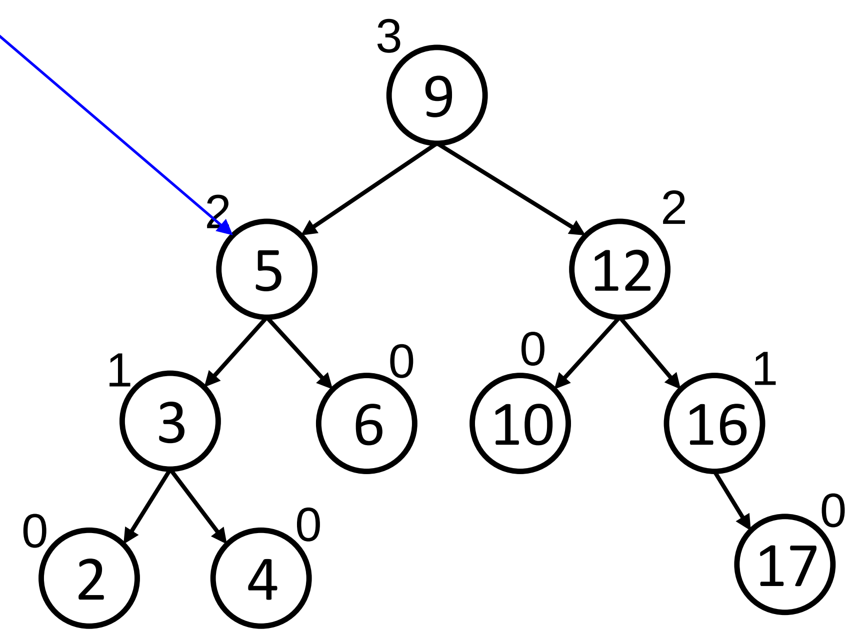
How does RCU avoids erroneous executions while allowing asynchronized traversals?

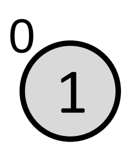
- Assume a single updater for now

T1: lookup(2)

1. Updaters create copies of the modified parts

T2: insert(1)





RCU-HTM: Asynchronized traversals

RCU-HTM

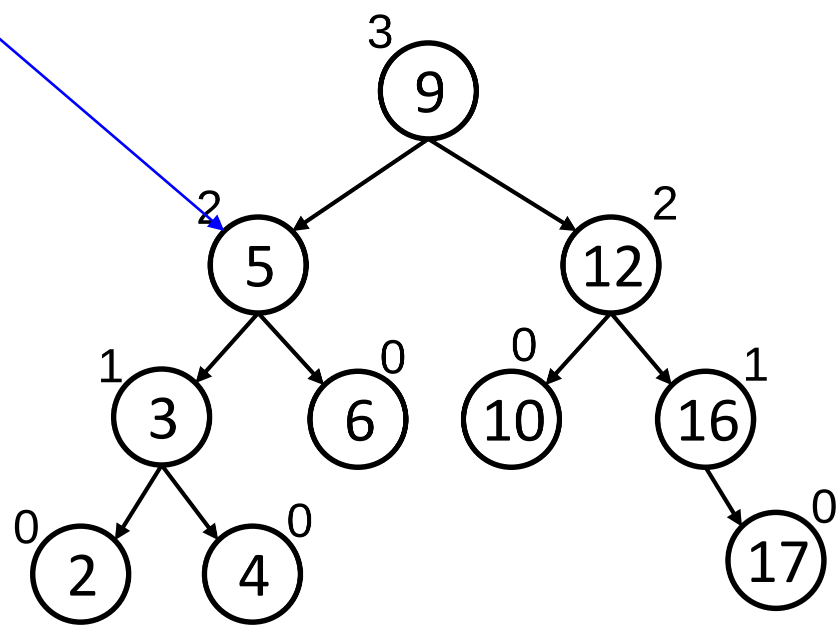
How does RCU avoids erroneous executions while allowing asynchronized traversals?

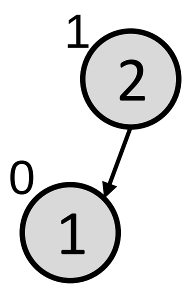
- Assume a single updater for now

T1: lookup(2)

1. Updaters create copies of the modified parts

T2: insert(1)



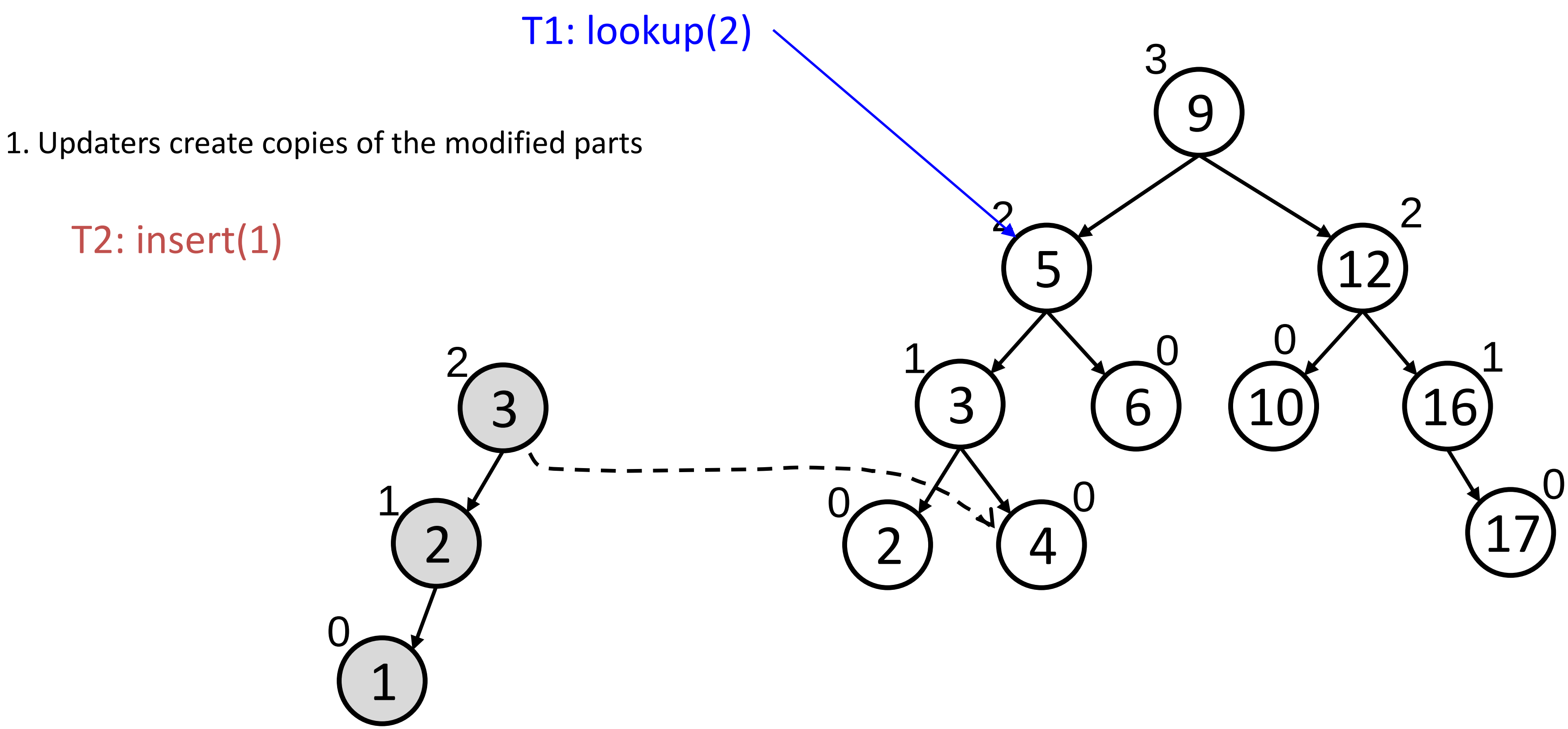


RCU-HTM: Asynchronized traversals

RCU-HTM

How does RCU avoids erroneous executions while allowing asynchronized traversals?

- Assume a single updater for now



RCU-HTM: Asynchronized traversals

RCU-HTM

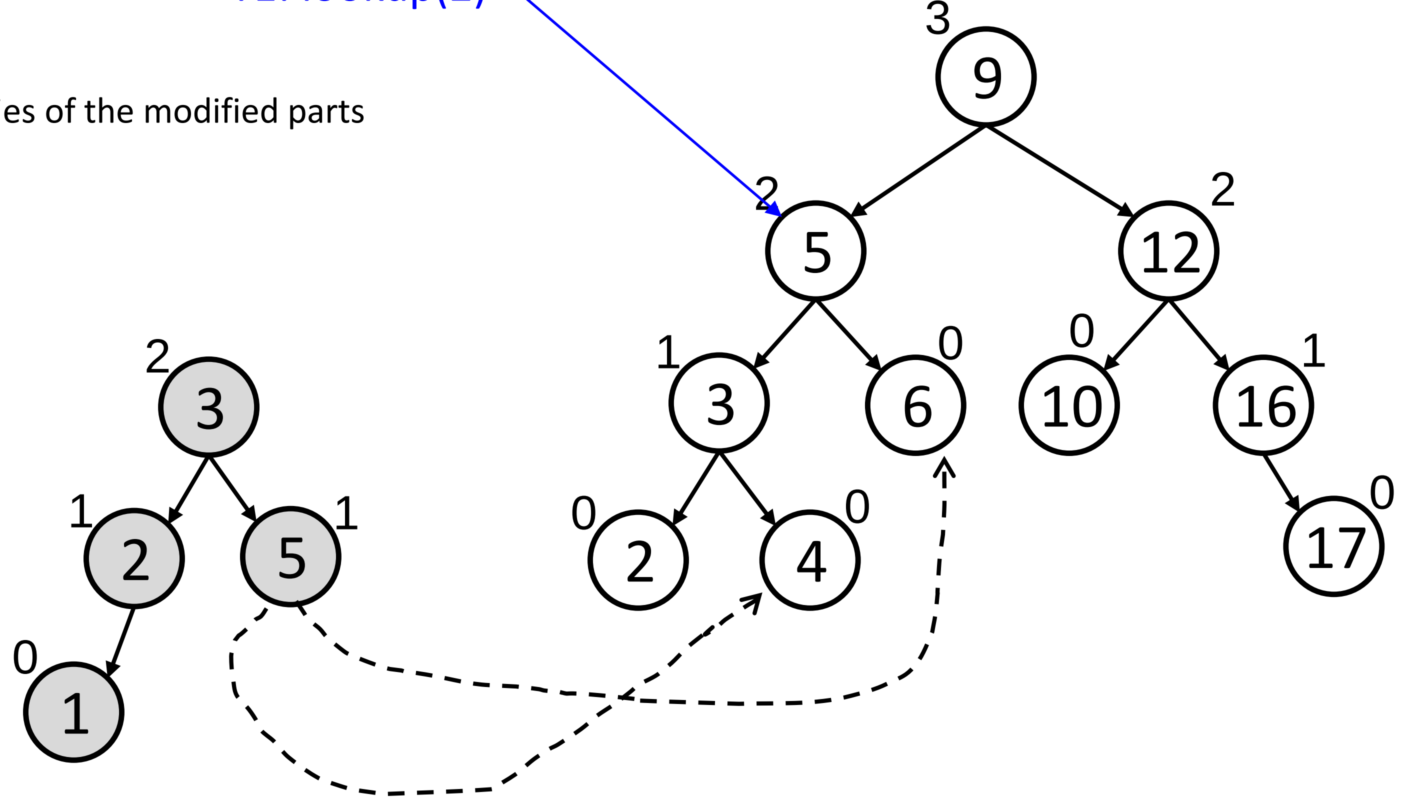
How does RCU avoids erroneous executions while allowing asynchronized traversals?

- Assume a single updater for now

T1: lookup(2)

1. Updaters create copies of the modified parts

T2: insert(1)



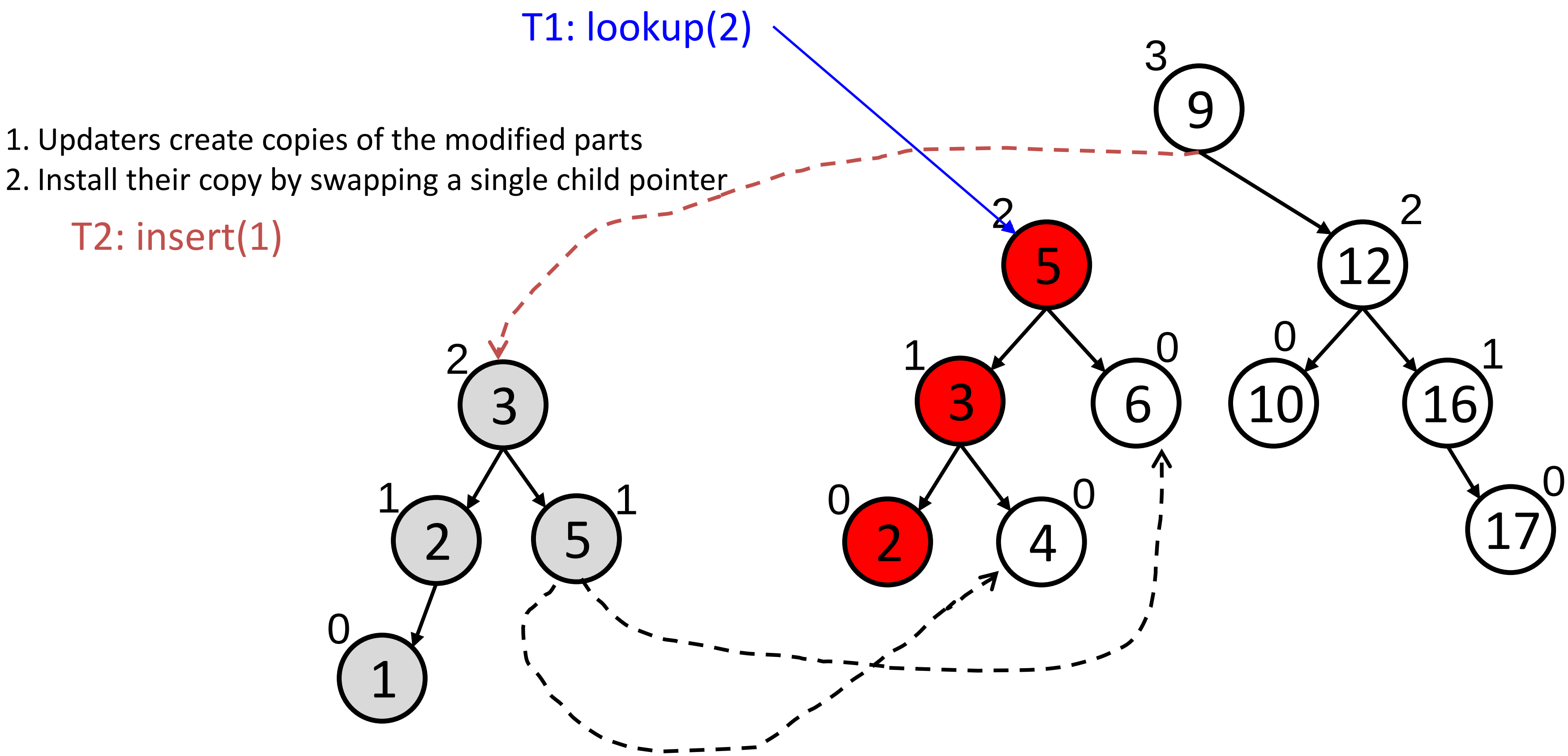
National Technical University of Athens
CSLab

RCU-HTM: Asynchronized traversals

RCU-HTM

How does RCU avoids erroneous executions while allowing asynchronized traversals?

- Assume a single updater for now

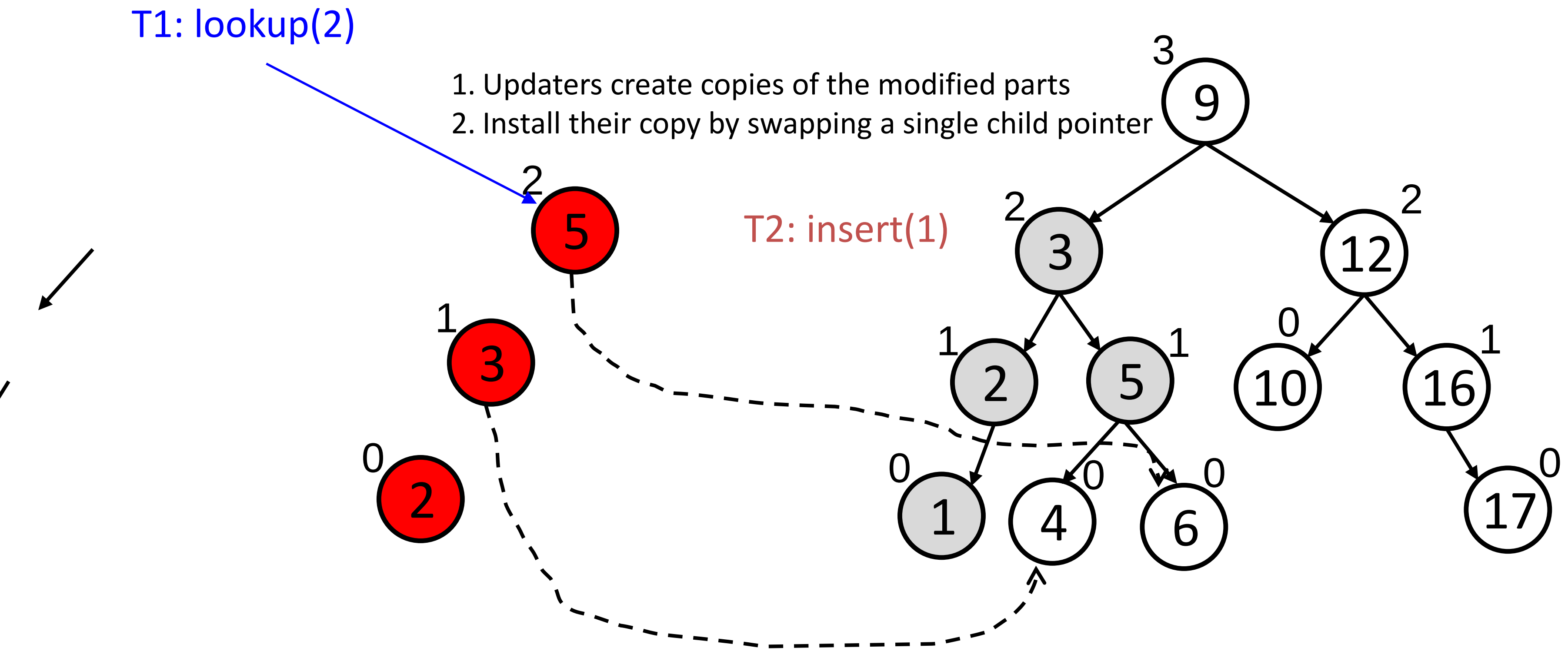


RCU-HTM: Asynchronized traversals

RCU-HTM

How does RCU avoids erroneous executions while allowing asynchronized traversals?

- Assume a single updater for now

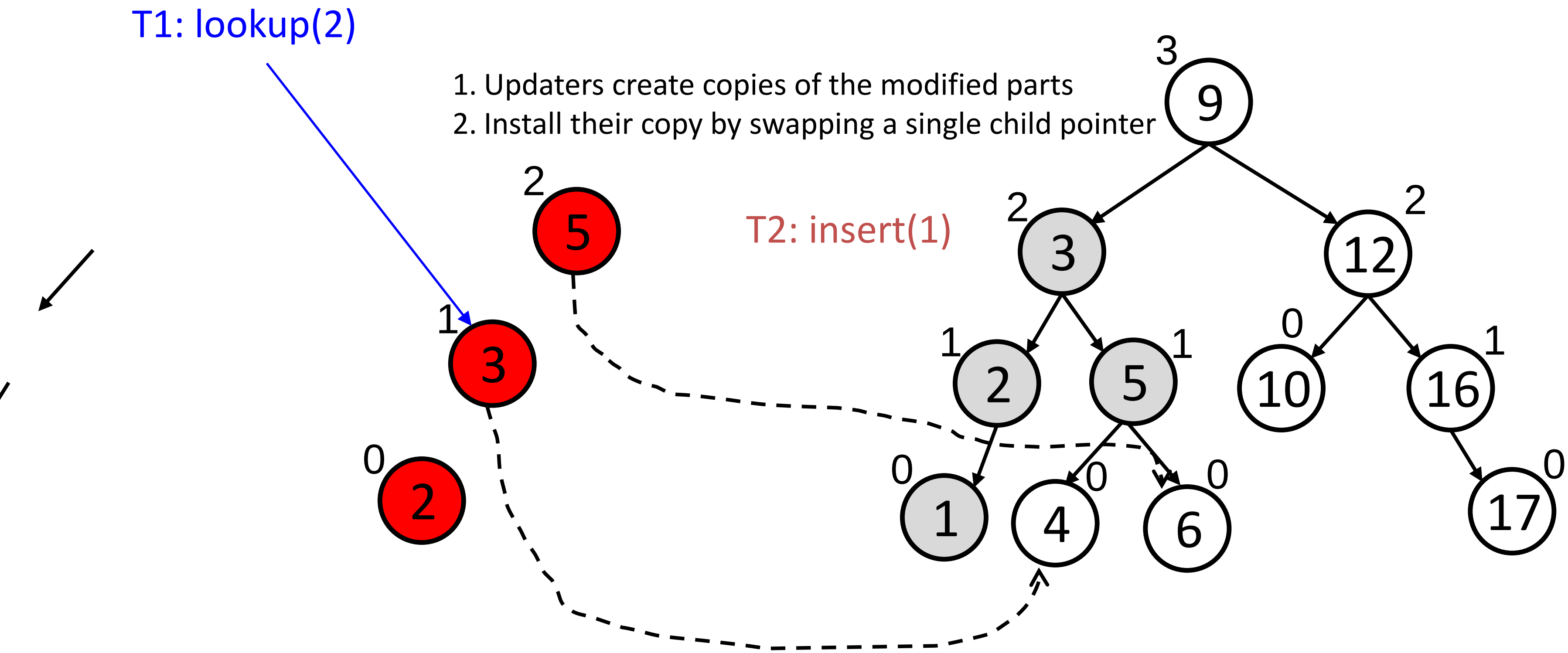


RCU-HTM: Asynchronized traversals

RCU-HTM

How does RCU avoids erroneous executions while allowing asynchronized traversals?

- Assume a single updater for now

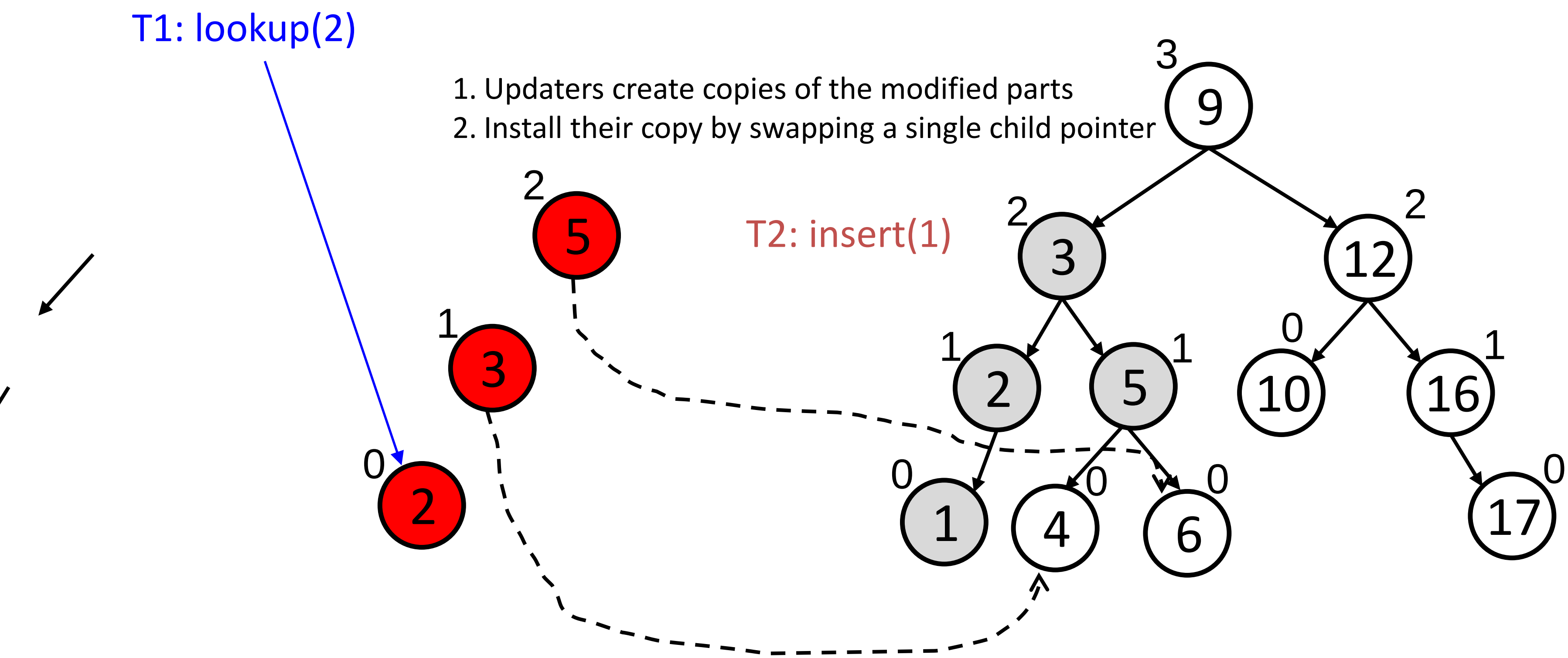


RCU-HTM: Asynchronized traversals

RCU-HTM

How does RCU avoids erroneous executions while allowing asynchronized traversals?

- Assume a single updater for now



National Technical University of Athens CSLab

RCU-HTM: Asynchronized traversals

RCU-HTM

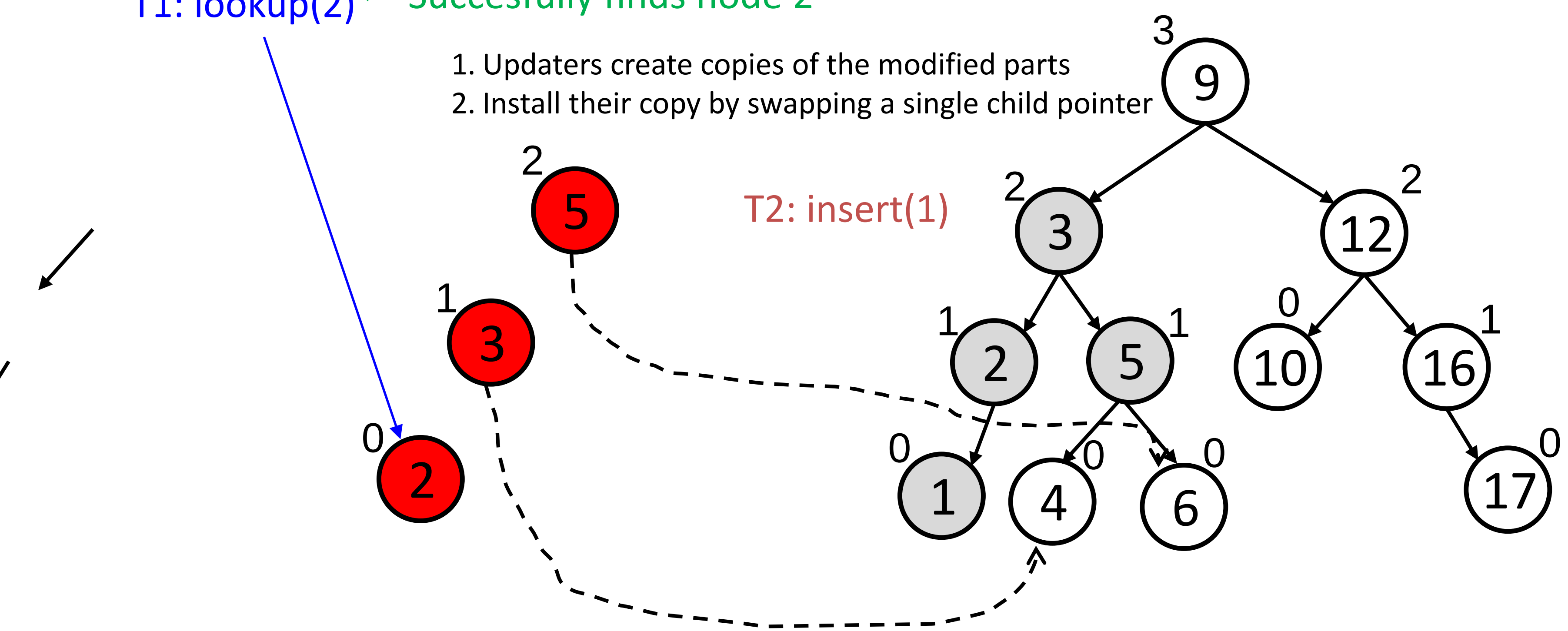
How does RCU avoids erroneous executions while allowing asynchronized traversals?

- Assume a single updater for now

T1: lookup(2) ✓ Succesfully finds node 2

1. Updaters create copies of the modified parts
2. Install their copy by swapping a single child pointer

T2: insert(1)

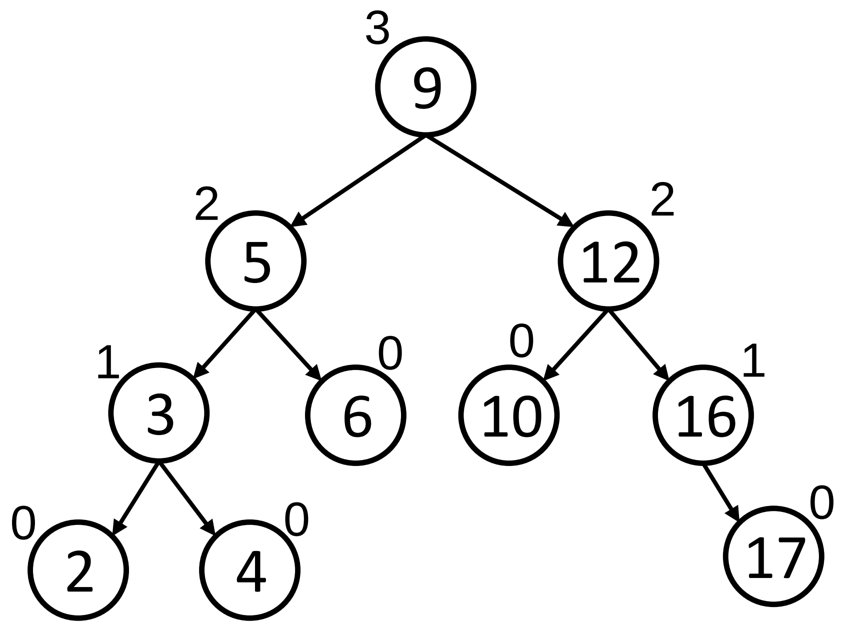


RCU-HTM: Multiple Updaters

RCU-HTM

The previous example assumed a single updater

- If multiple updaters were allowed, modifications could be “lost”



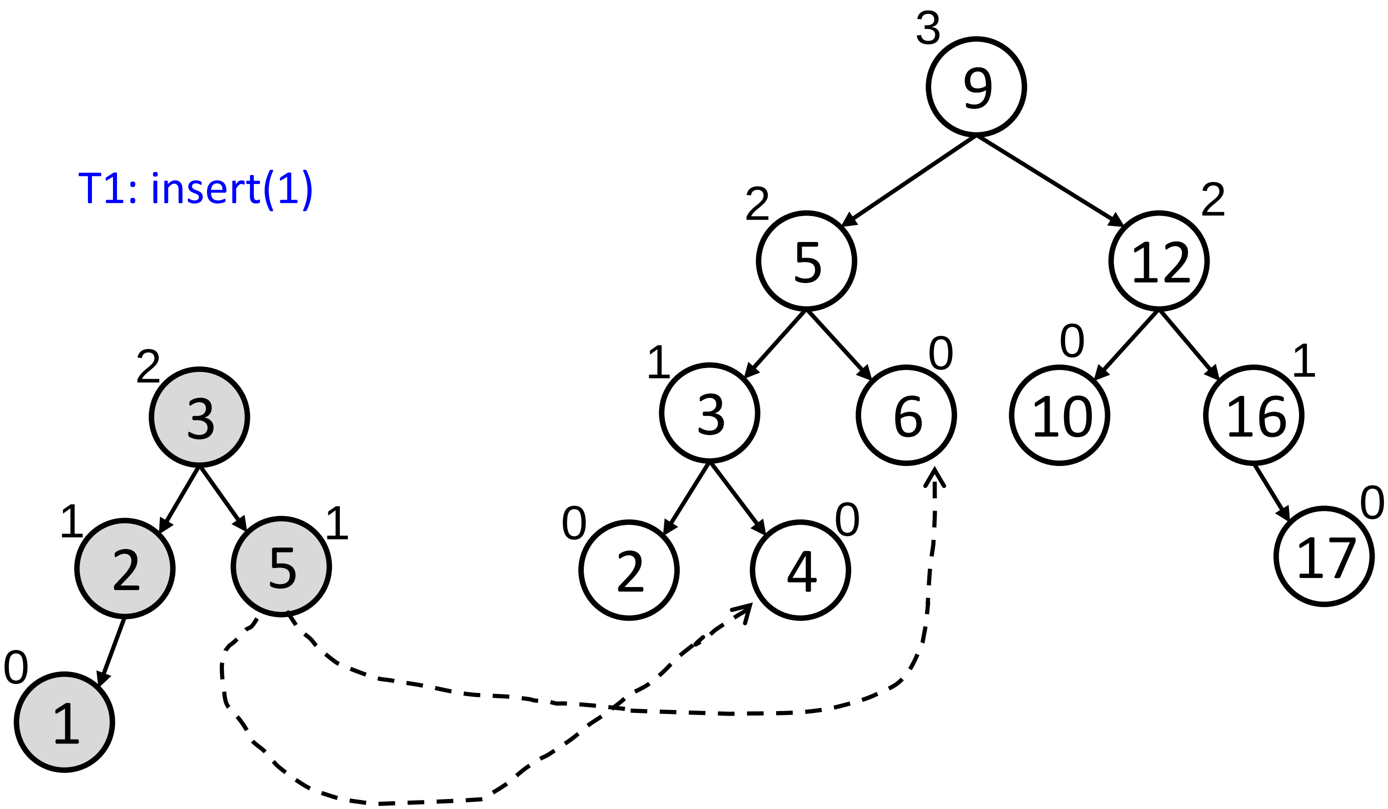
RCU-HTM: Multiple Updaters

RCU-HTM

The previous example assumed a single updater

- If multiple updaters were allowed, modifications could be “lost”

T1: insert(1)



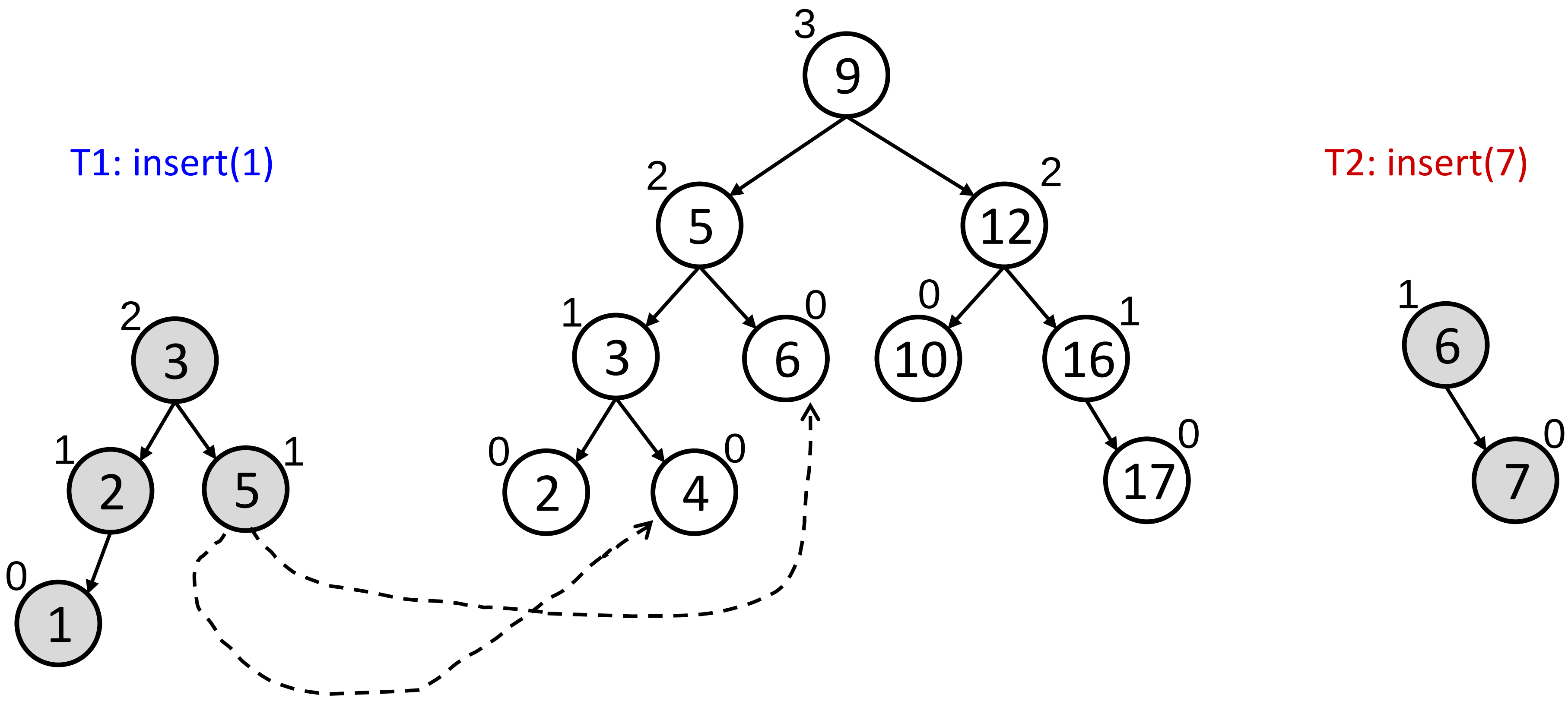
National Technical University of Athens CSLab

RCU-HTM: Multiple Updaters

RCU-HTM

The previous example assumed a single updater

- If multiple updaters were allowed, modifications could be “lost”

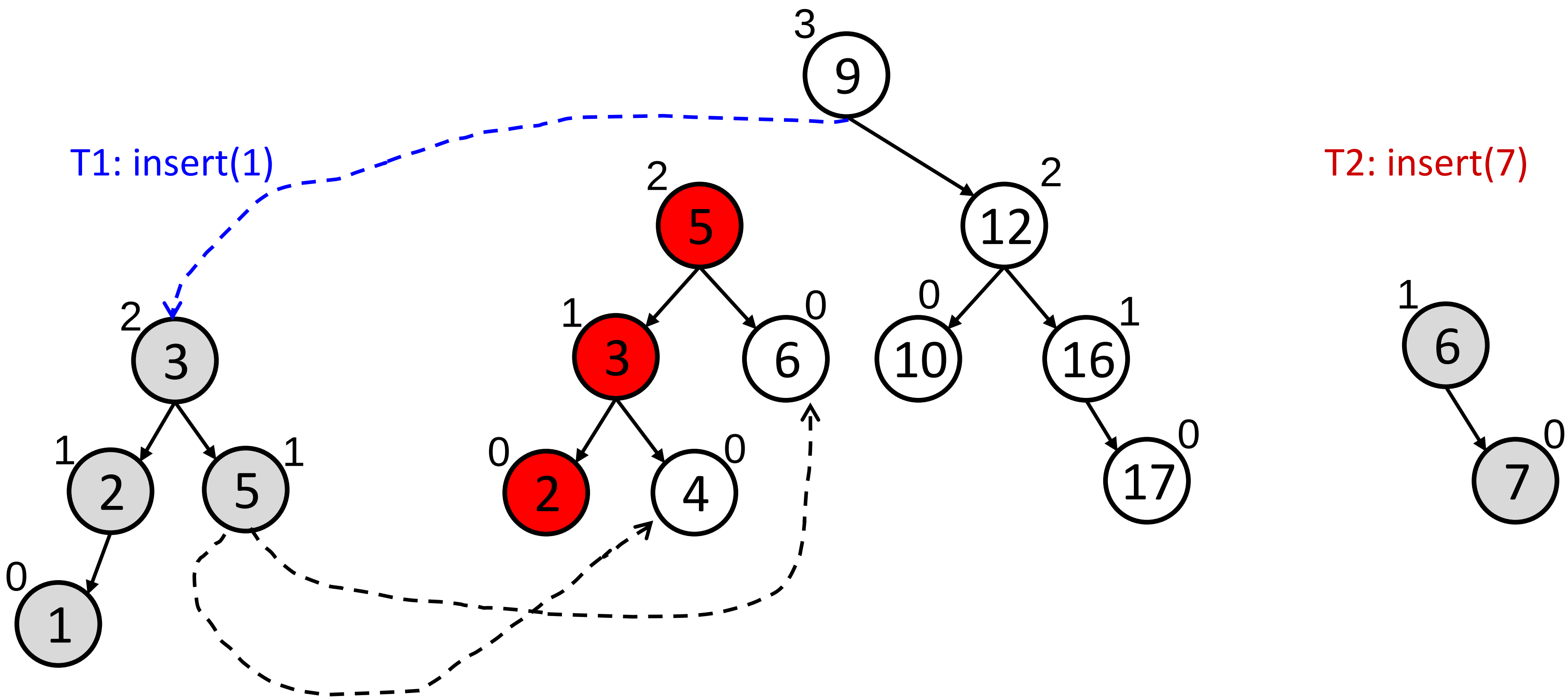


RCU-HTM: Multiple Updaters

RCU-HTM

The previous example assumed a single updater

- If multiple updaters were allowed, modifications could be "lost"

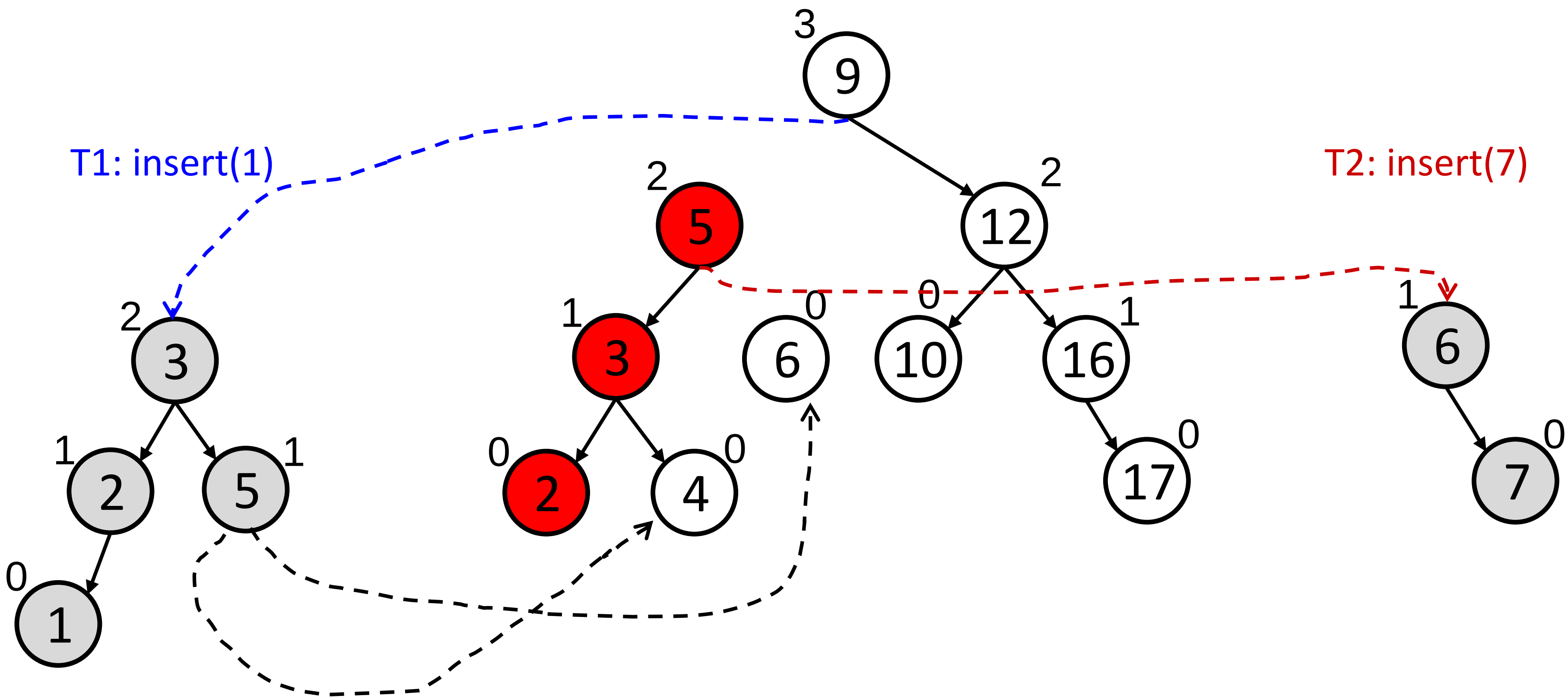


RCU-HTM: Multiple Updaters

RCU-HTM

The previous example assumed a single updater

- If multiple updaters were allowed, modifications could be "lost"

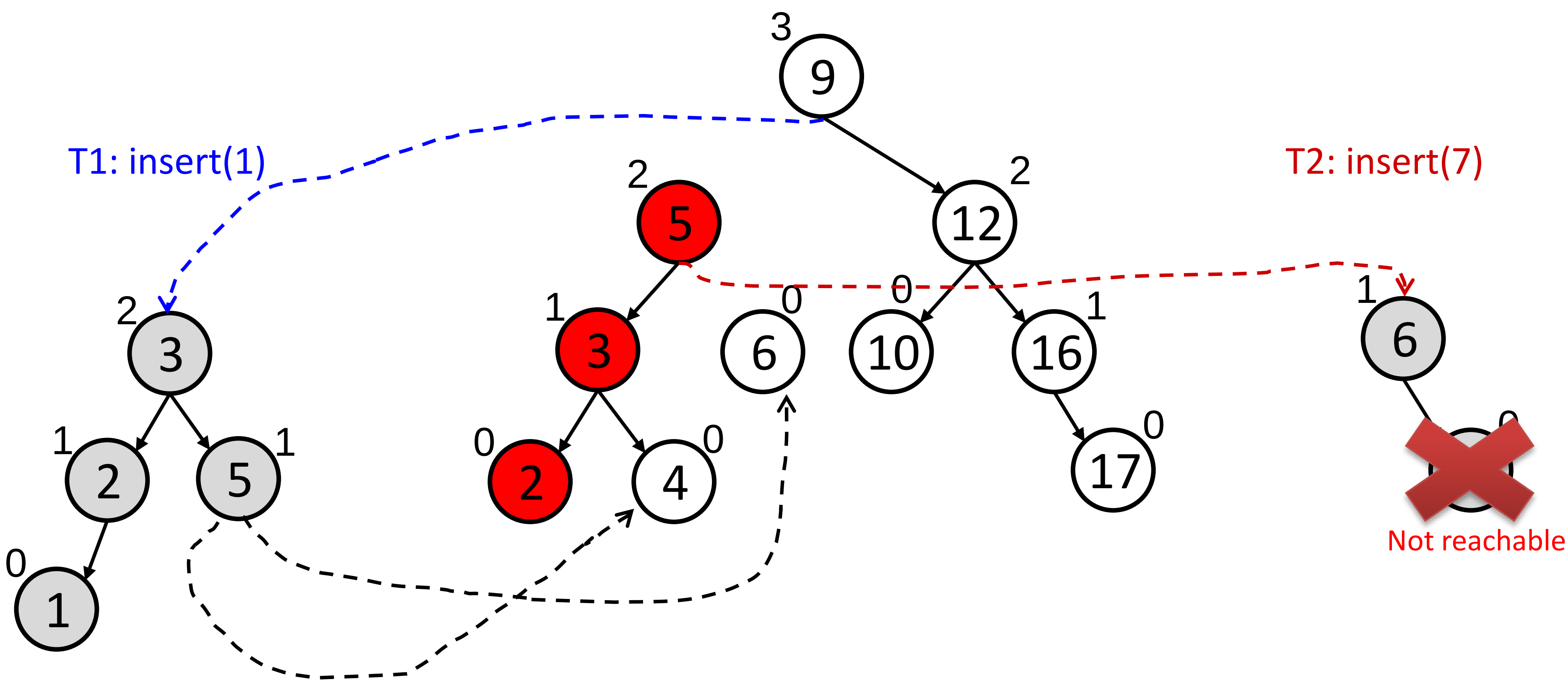


RCU-HTM: Multiple Updaters

RCU-HTM

The previous example assumed a single updater

- If multiple updaters were allowed, modifications could be “lost”



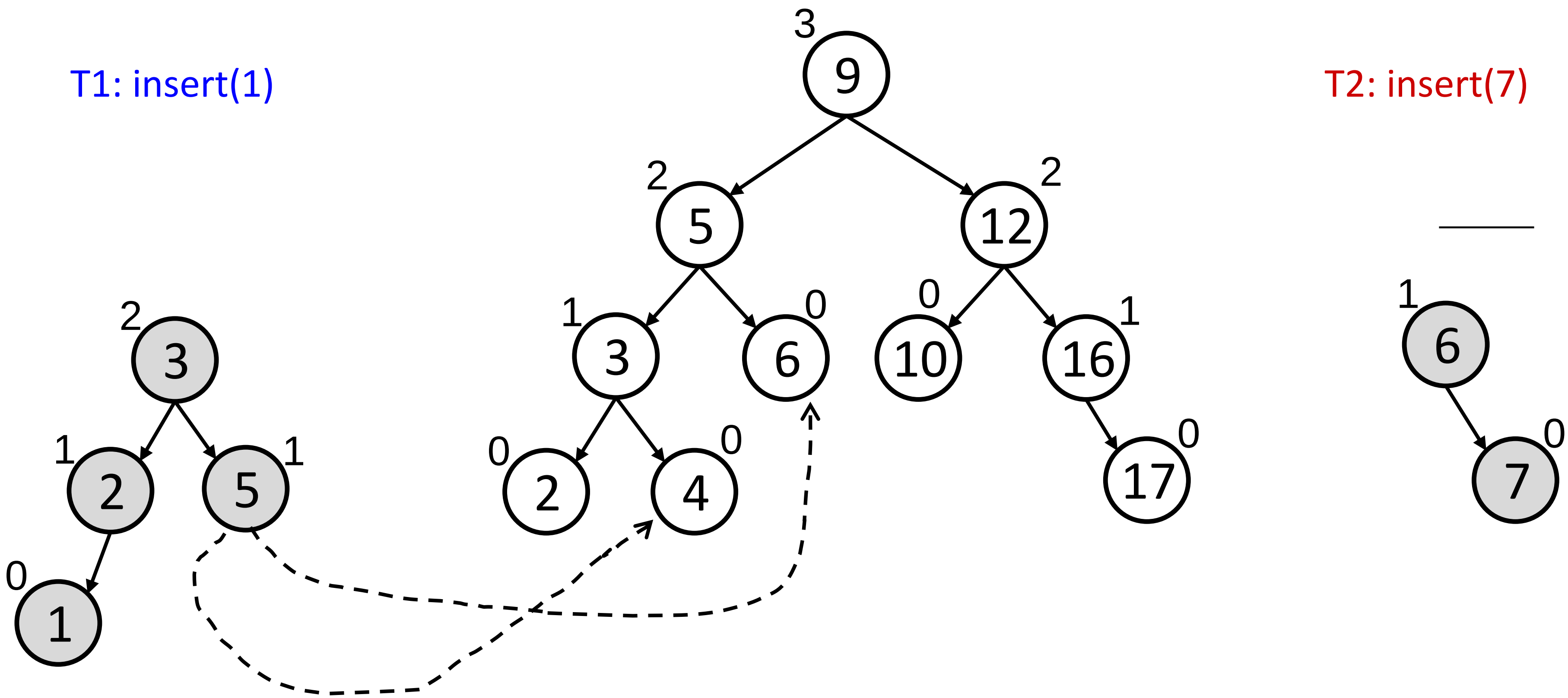
RCU-HTM: Multiple Updaters

RCU-HTM overcomes the problem of "lost" updates by exploiting HTM in the following way:

- Updaters keep track of the state of the traversed and the copied nodes, i.e., the addresses of the children pointers
- Before installing their copy they validate that all these nodes have remained intact
  - validation and installation are performed atomically using an HTM transaction

T1: insert(1)

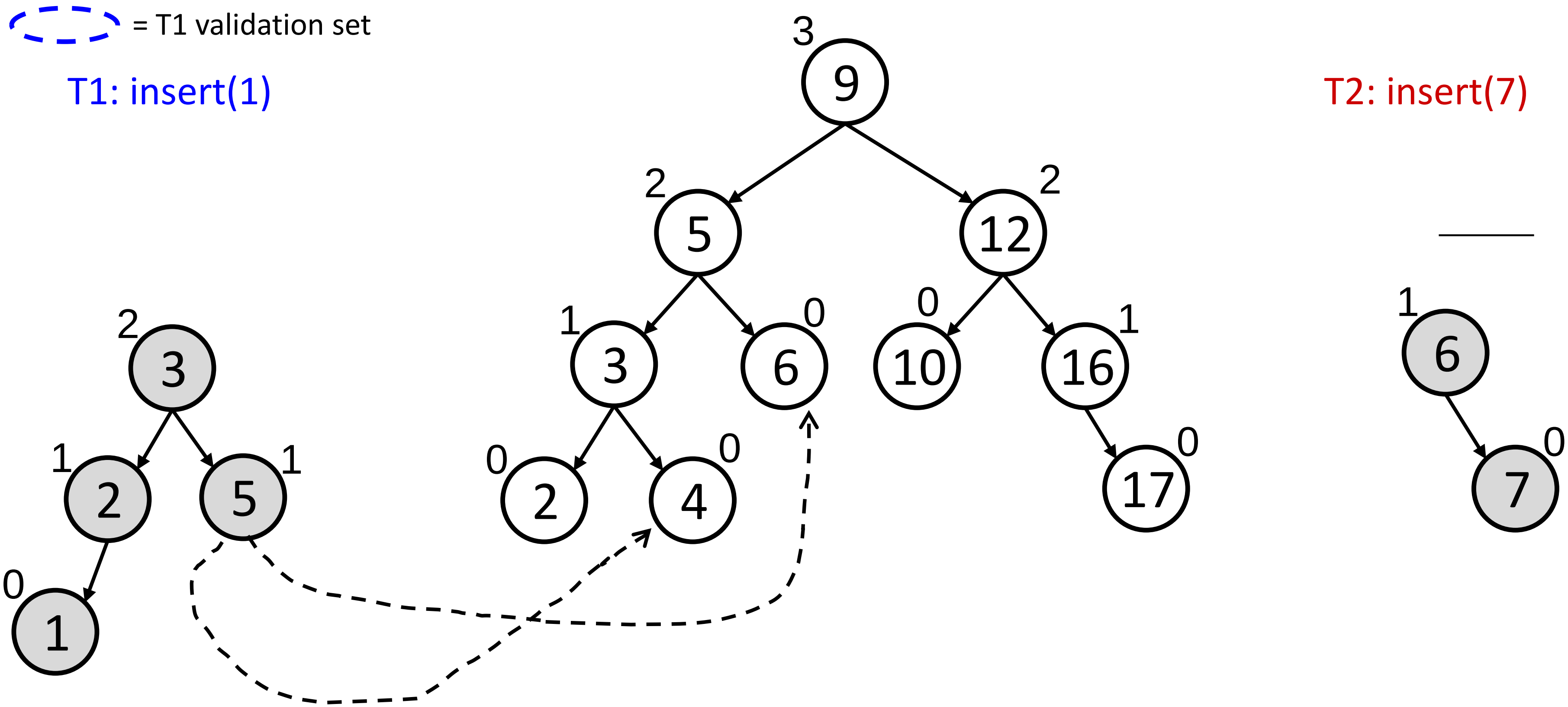
T2: insert(7)



RCU-HTM: Multiple Updaters

RCU-HTM overcomes the problem of "lost" updates by exploiting HTM in the following way:

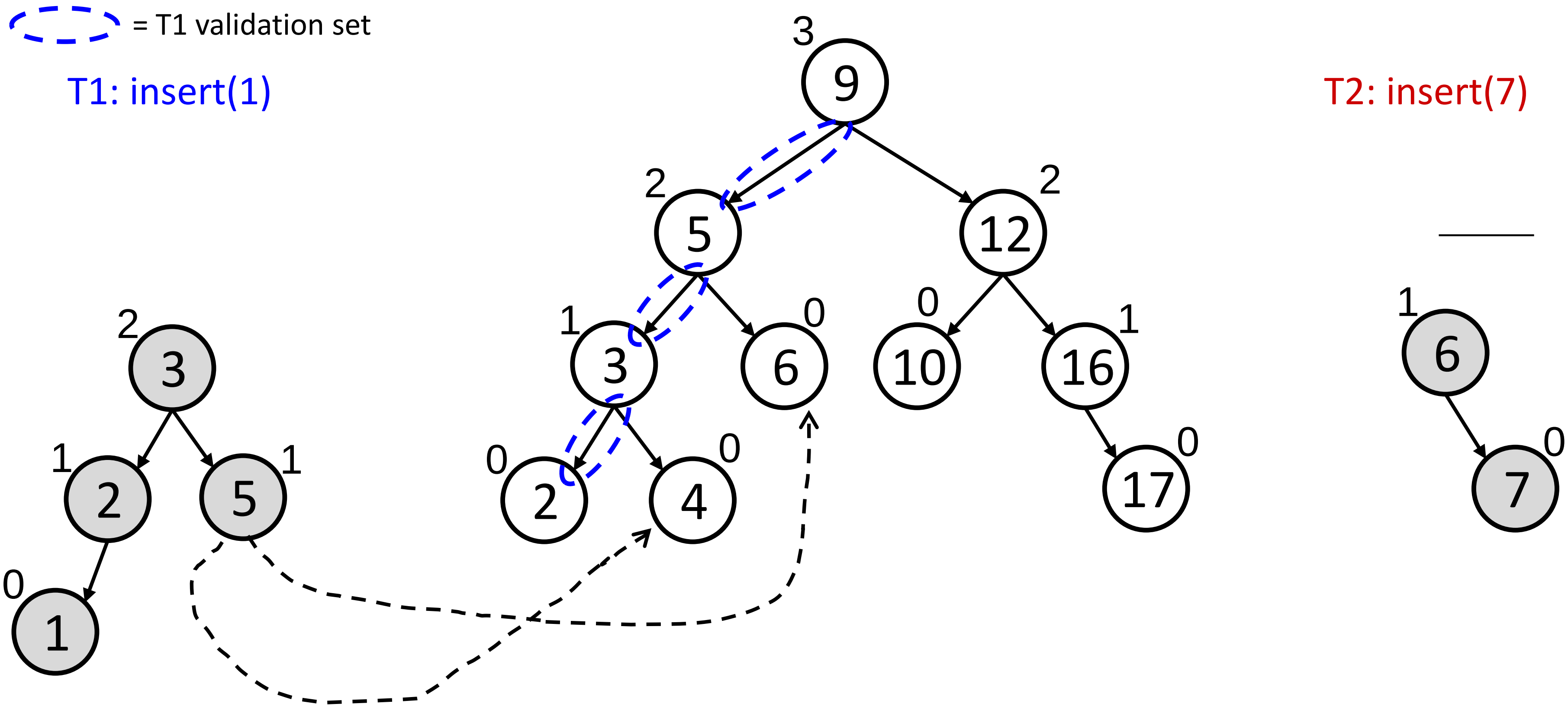
- Updaters keep track of the state of the traversed and the copied nodes, i.e., the addresses of the children pointers
- Before installing their copy they validate that all these nodes have remained intact
  - validation and installation are performed atomically using an HTM transaction



RCU-HTM: Multiple Updaters

RCU-HTM overcomes the problem of “lost” updates by exploiting HTM in the following way:

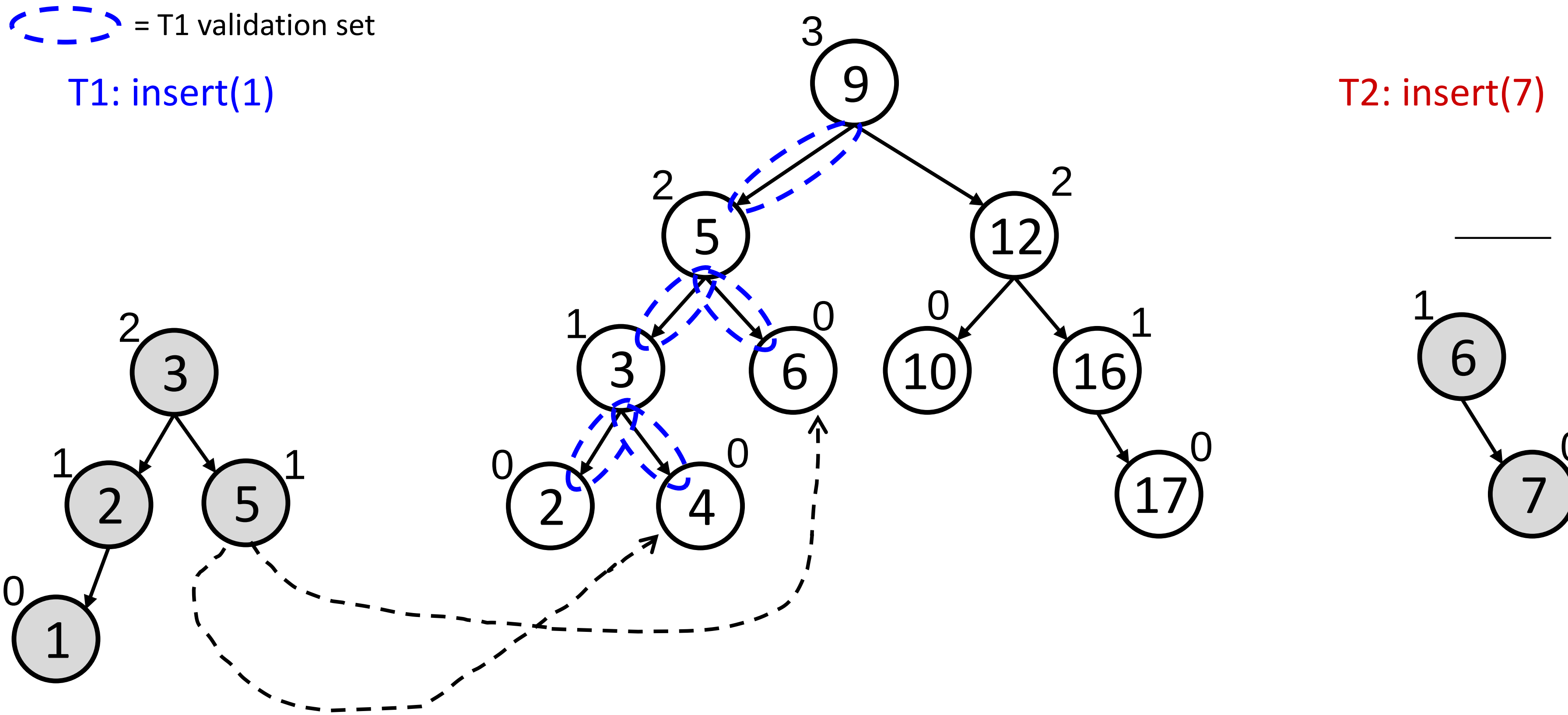
- Updaters keep track of the state of the traversed and the copied nodes, i.e., the addresses of the children pointers
- Before installing their copy they validate that all these nodes have remained intact
  - validation and installation are performed atomically using an HTM transaction



RCU-HTM: Multiple Updaters

RCU-HTM overcomes the problem of "lost" updates by exploiting HTM in the following way:

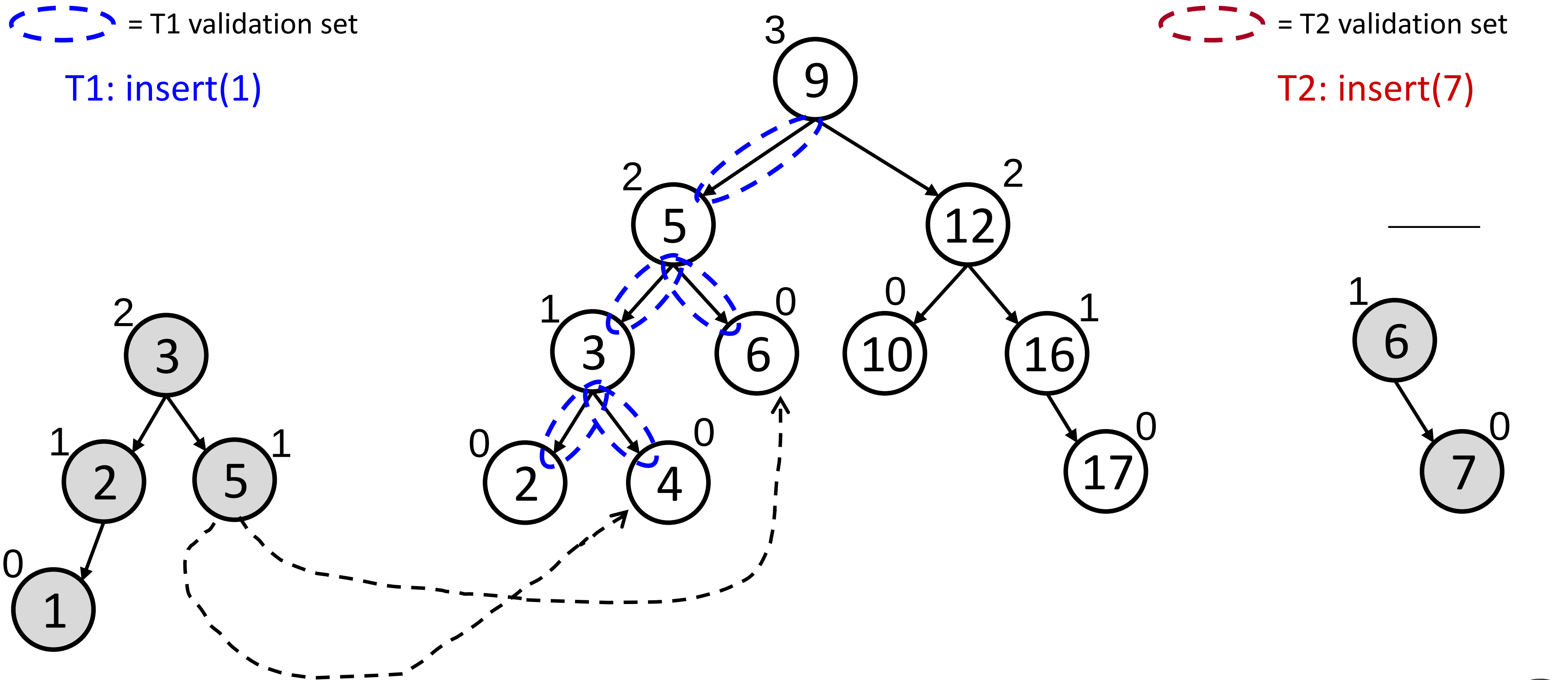
- Updaters keep track of the state of the traversed and the copied nodes, i.e., the addresses of the children pointers
- Before installing their copy they validate that all these nodes have remained intact
  - validation and installation are performed atomically using an HTM transaction



RCU-HTM: Multiple Updaters

RCU-HTM overcomes the problem of “lost” updates by exploiting HTM in the following way:

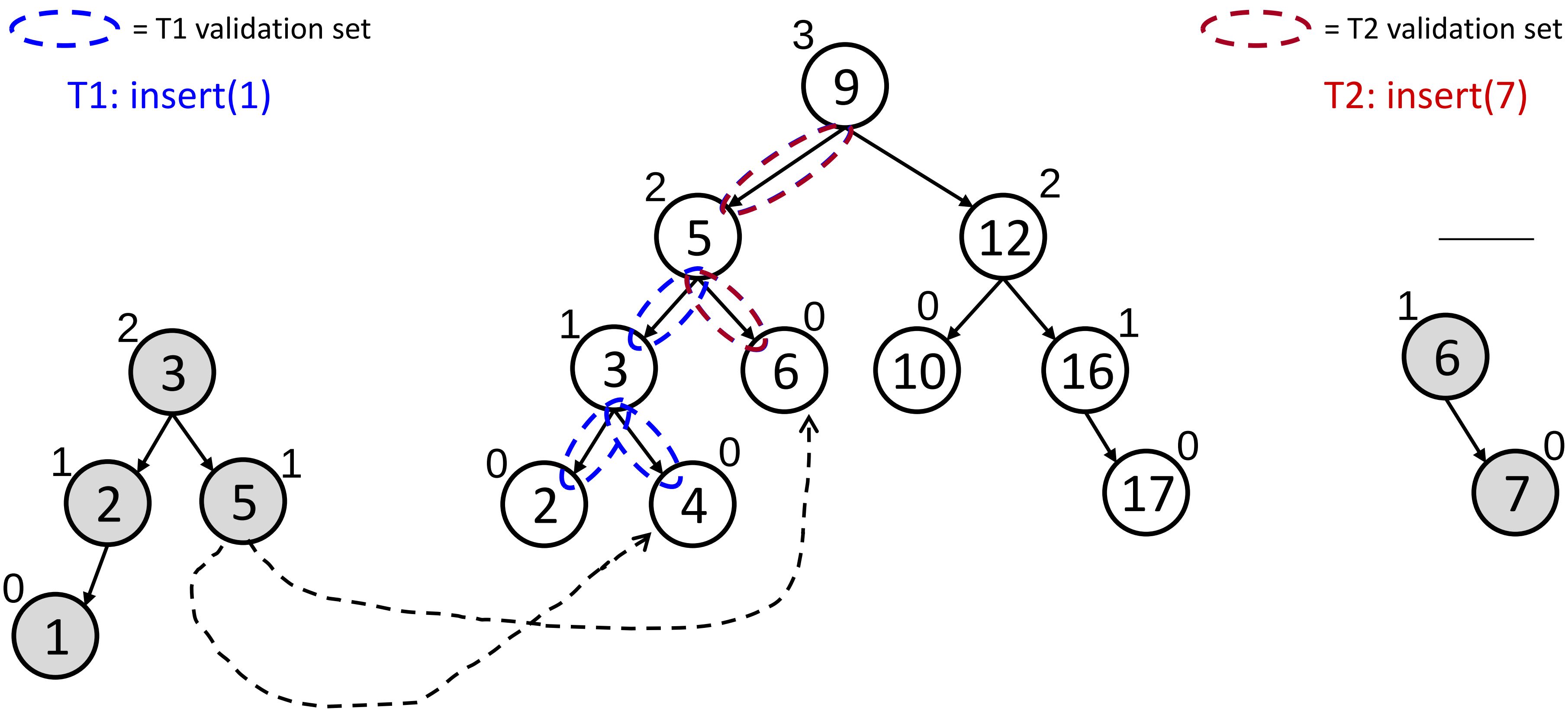
- Updaters keep track of the state of the traversed and the copied nodes, i.e., the addresses of the children pointers
- Before installing their copy they validate that all these nodes have remained intact
  - validation and installation are performed atomically using an HTM transaction



RCU-HTM: Multiple Updaters

RCU-HTM overcomes the problem of “lost” updates by exploiting HTM in the following way:

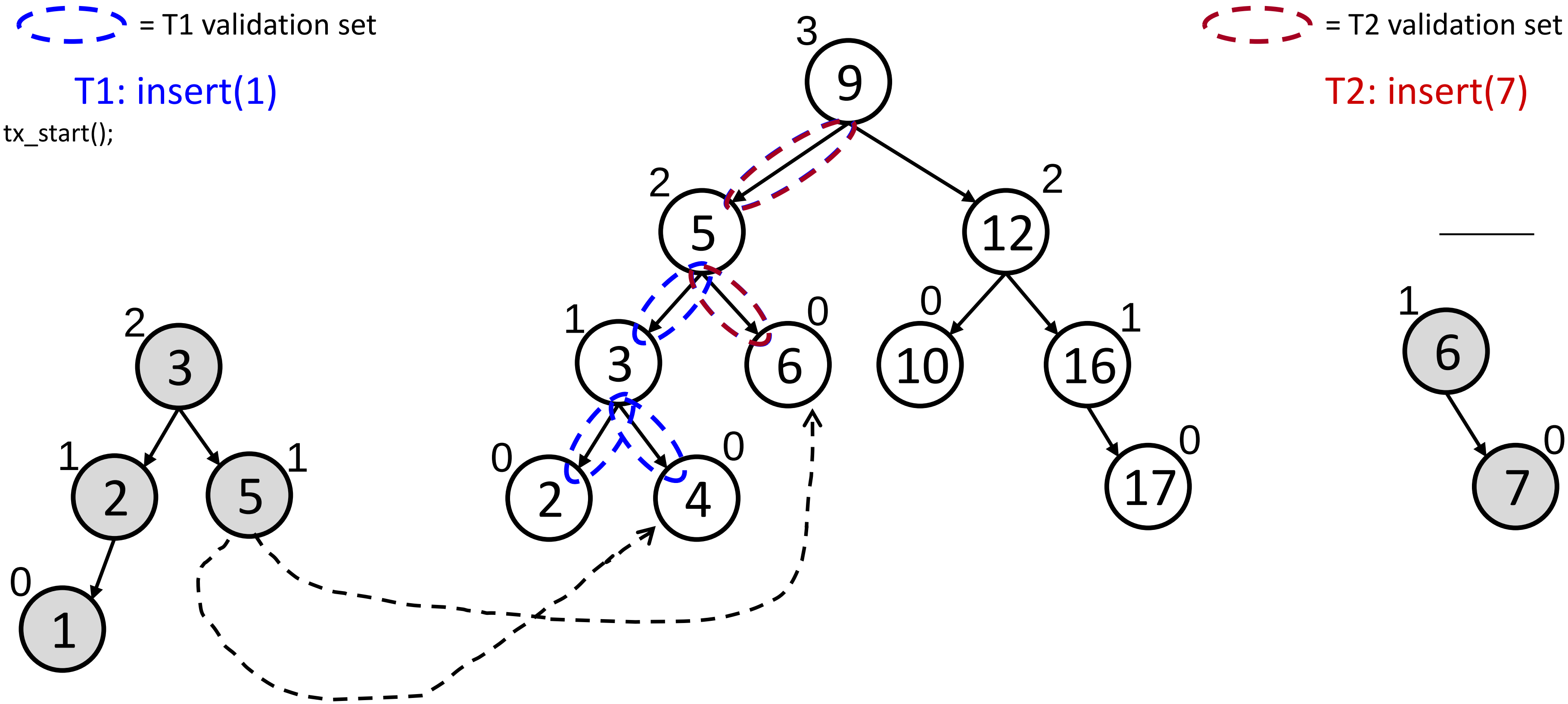
- Updaters keep track of the state of the traversed and the copied nodes, i.e., the addresses of the children pointers
- Before installing their copy they validate that all these nodes have remained intact
  - validation and installation are performed atomically using an HTM transaction



RCU-HTM: Multiple Updaters

RCU-HTM overcomes the problem of “lost” updates by exploiting HTM in the following way:

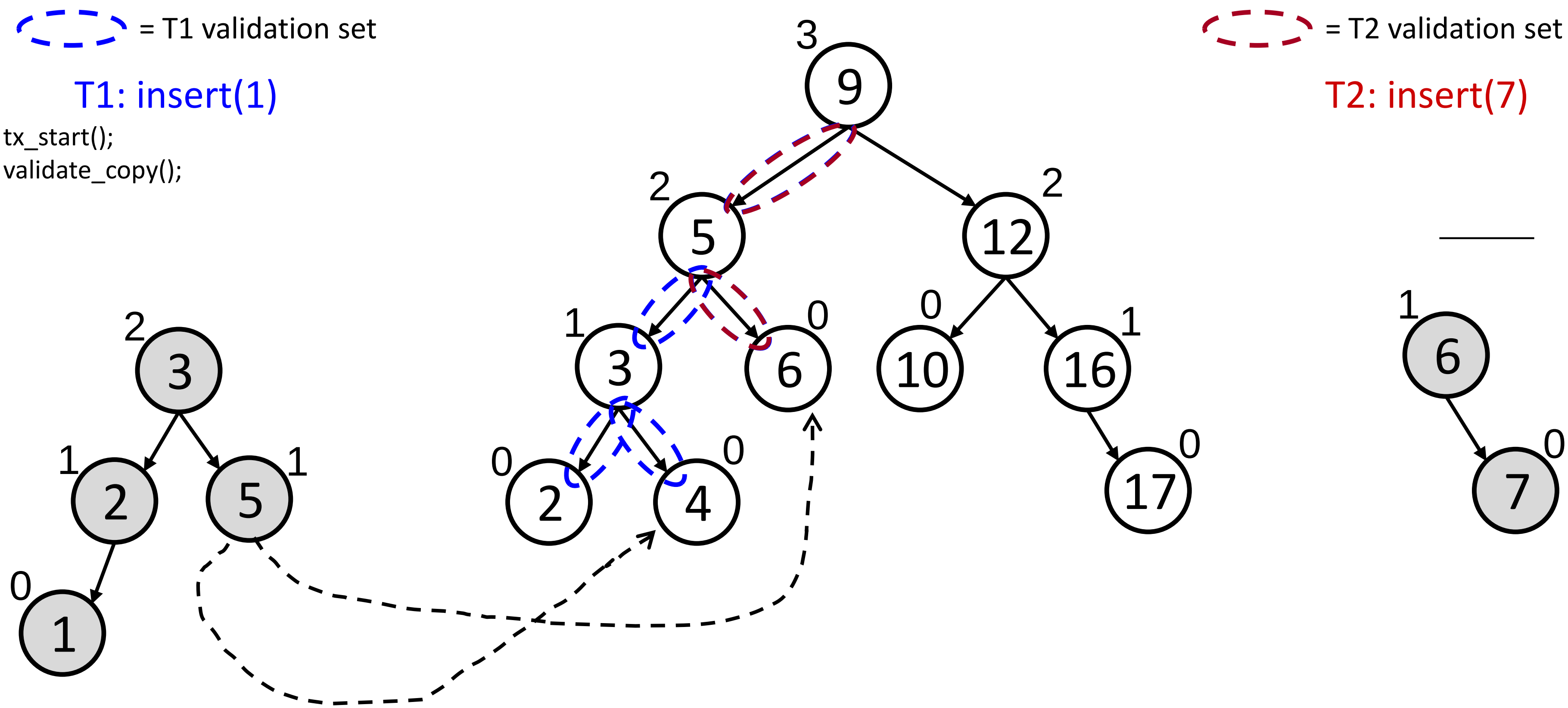
- Updaters keep track of the state of the traversed and the copied nodes, i.e., the addresses of the children pointers
- Before installing their copy they validate that all these nodes have remained intact
  - validation and installation are performed atomically using an HTM transaction



RCU-HTM: Multiple Updaters

RCU-HTM overcomes the problem of "lost" updates by exploiting HTM in the following way:

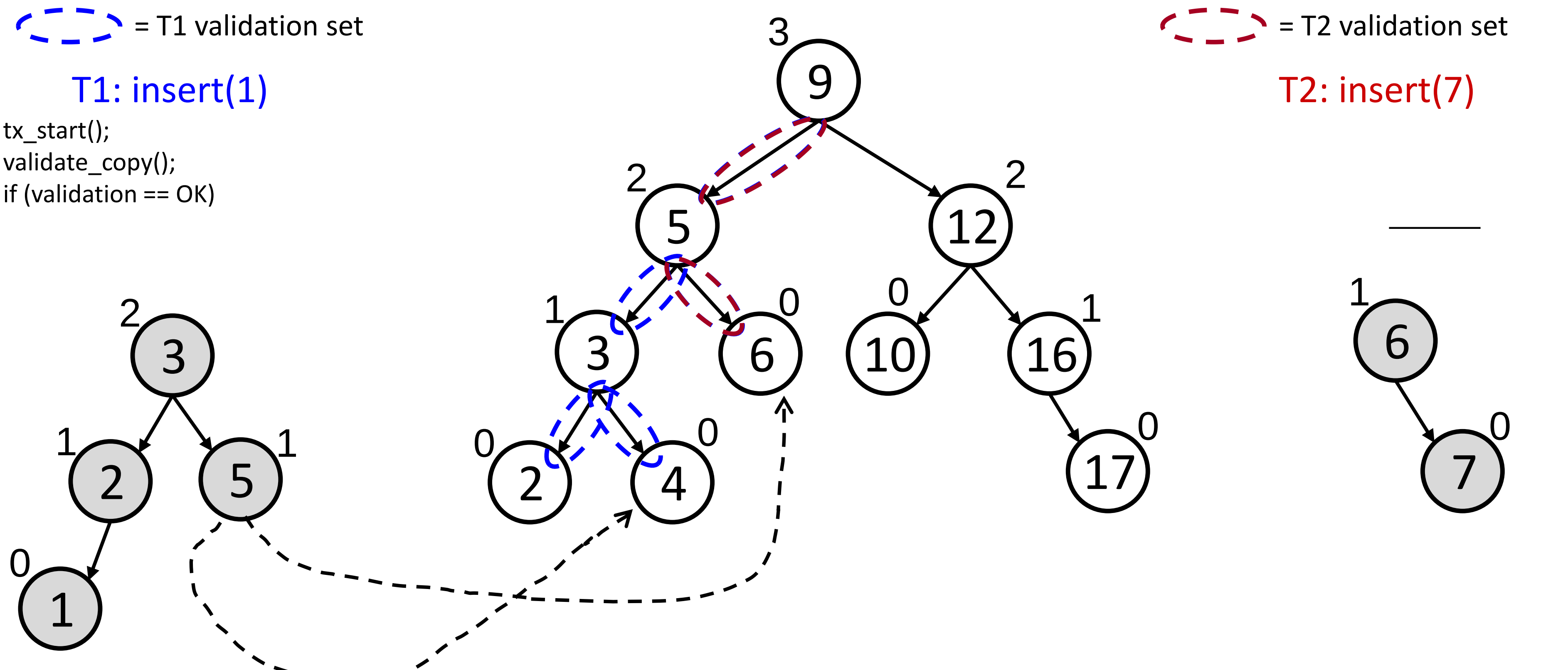
- Updaters keep track of the state of the traversed and the copied nodes, i.e., the addresses of the children pointers
- Before installing their copy they validate that all these nodes have remained intact
  - validation and installation are performed atomically using an HTM transaction



RCU-HTM: Multiple Updaters

RCU-HTM overcomes the problem of “lost” updates by exploiting HTM in the following way:

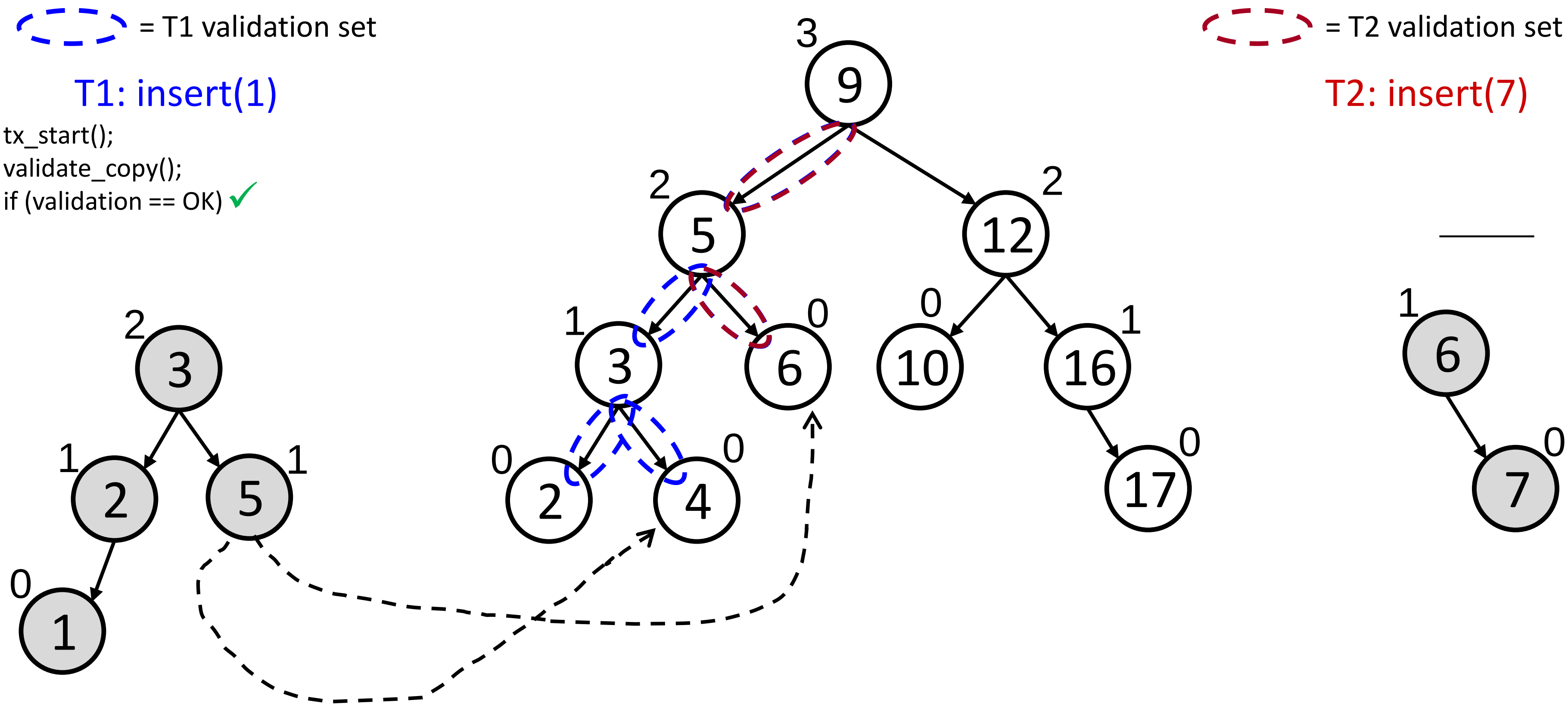
- Updaters keep track of the state of the traversed and the copied nodes, i.e., the addresses of the children pointers
- Before installing their copy they validate that all these nodes have remained intact
  - validation and installation are performed atomically using an HTM transaction



RCU-HTM: Multiple Updaters

RCU-HTM overcomes the problem of “lost” updates by exploiting HTM in the following way:

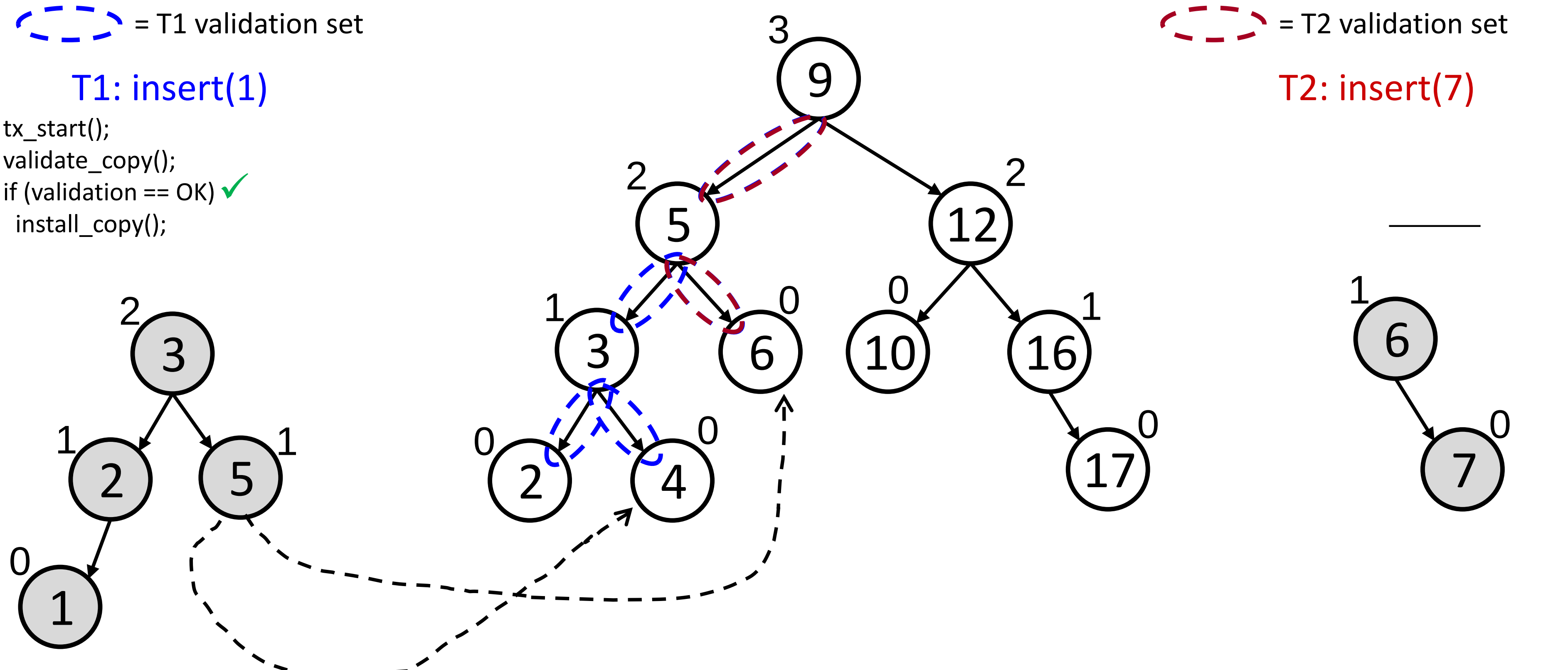
- Updaters keep track of the state of the traversed and the copied nodes, i.e., the addresses of the children pointers
- Before installing their copy they validate that all these nodes have remained intact
  - validation and installation are performed atomically using an HTM transaction



RCU-HTM: Multiple Updaters

RCU-HTM overcomes the problem of "lost" updates by exploiting HTM in the following way:

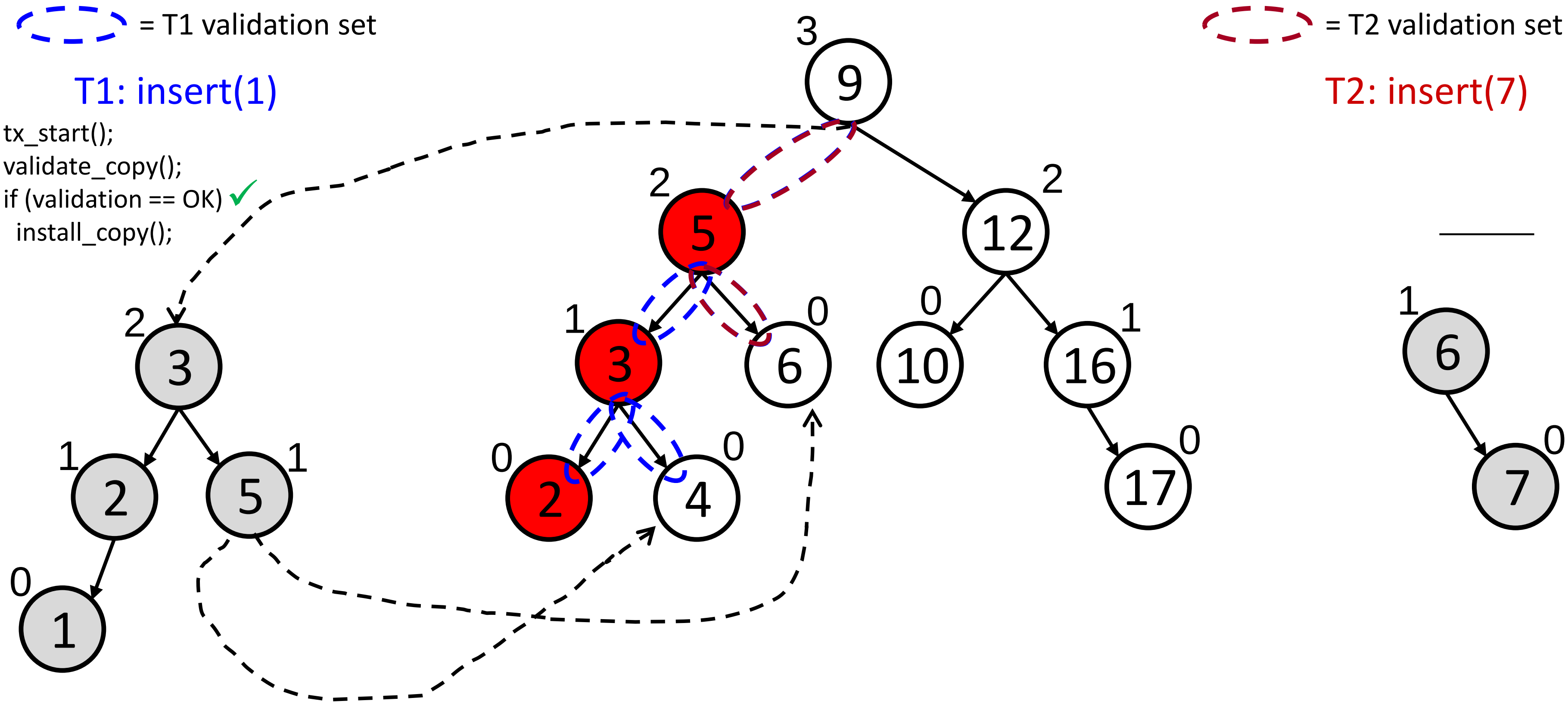
- Updaters keep track of the state of the traversed and the copied nodes, i.e., the addresses of the children pointers
- Before installing their copy they validate that all these nodes have remained intact
  - validation and installation are performed atomically using an HTM transaction



RCU-HTM: Multiple Updaters

RCU-HTM overcomes the problem of “lost” updates by exploiting HTM in the following way:

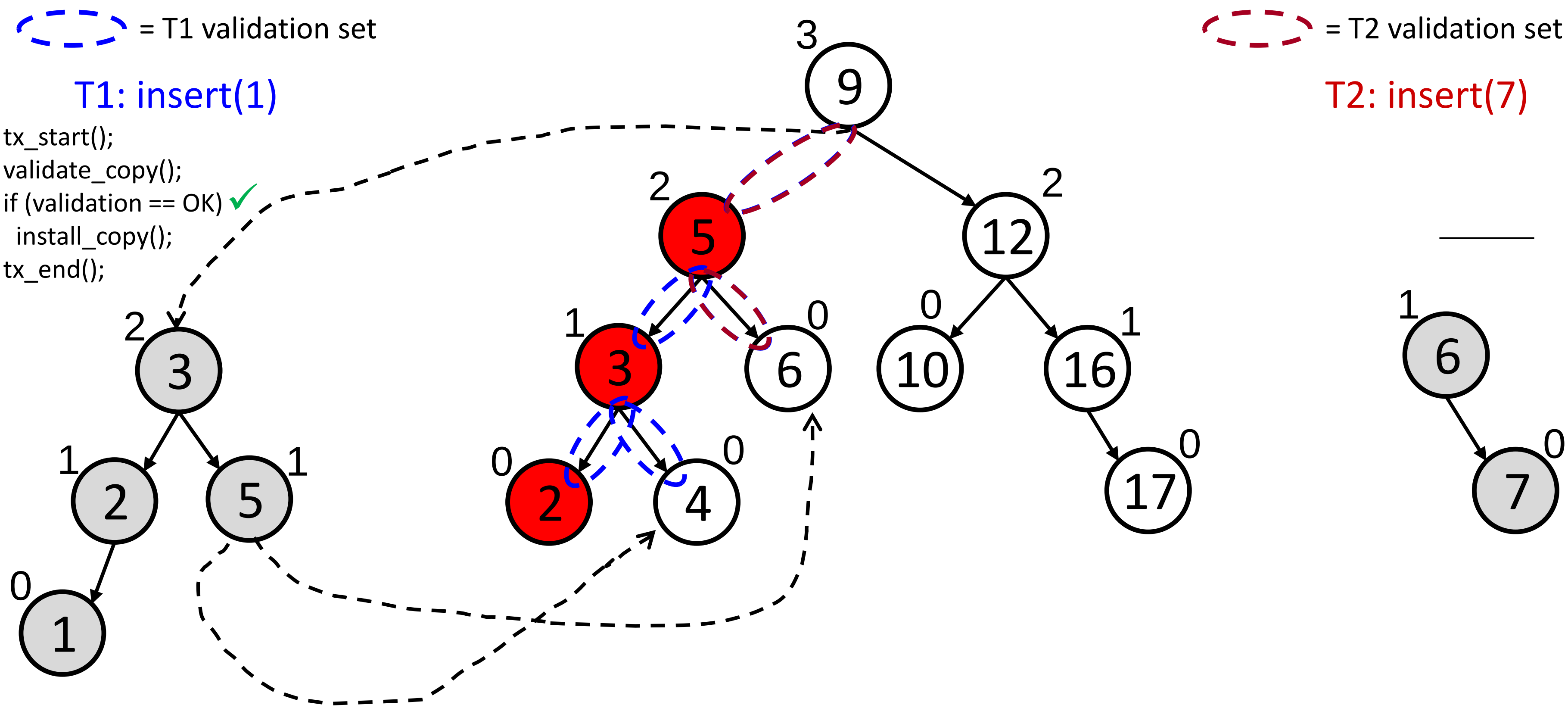
- Updaters keep track of the state of the traversed and the copied nodes, i.e., the addresses of the children pointers
- Before installing their copy they validate that all these nodes have remained intact
  - validation and installation are performed atomically using an HTM transaction



RCU-HTM: Multiple Updaters

RCU-HTM overcomes the problem of “lost” updates by exploiting HTM in the following way:

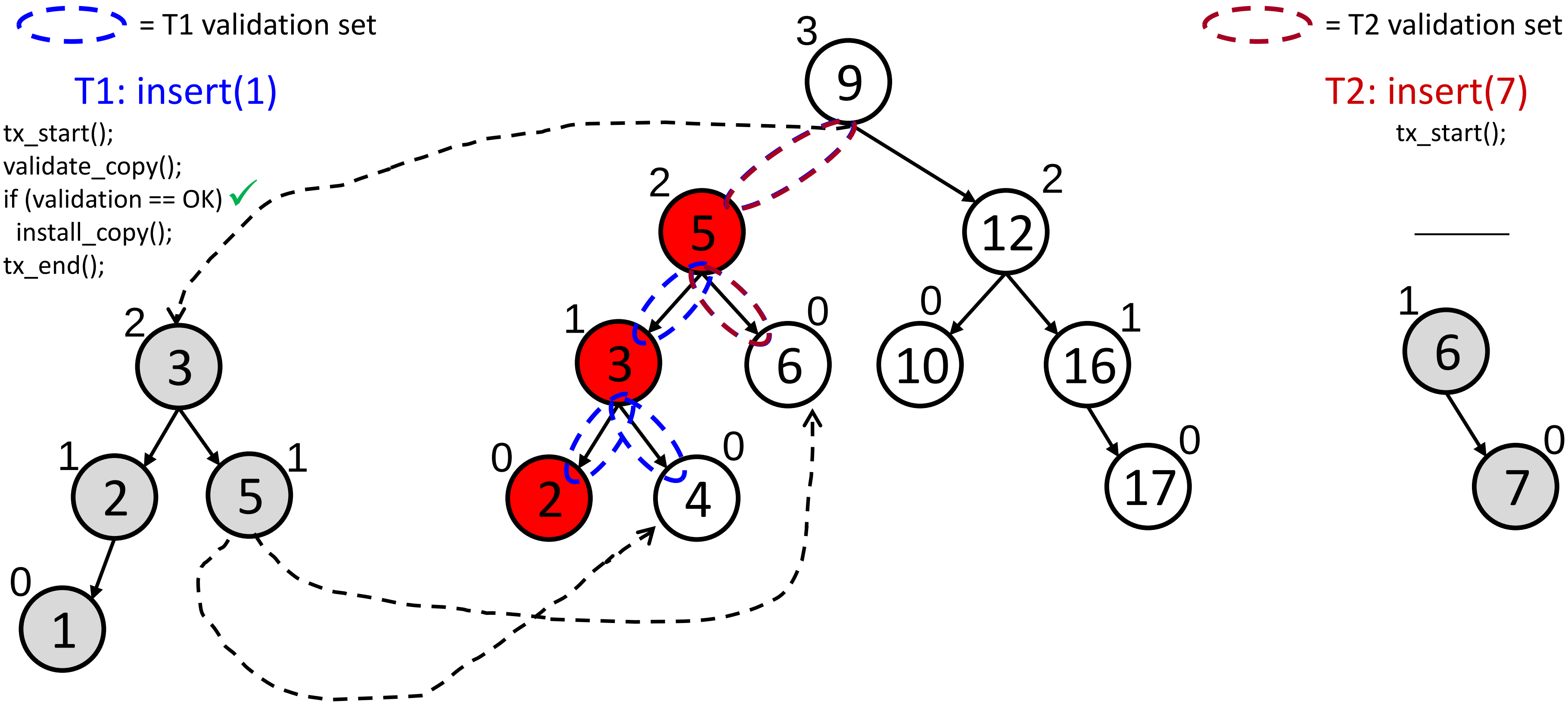
- Updaters keep track of the state of the traversed and the copied nodes, i.e., the addresses of the children pointers
- Before installing their copy they validate that all these nodes have remained intact
  - validation and installation are performed atomically using an HTM transaction



RCU-HTM: Multiple Updaters

RCU-HTM overcomes the problem of "lost" updates by exploiting HTM in the following way:

- Updaters keep track of the state of the traversed and the copied nodes, i.e., the addresses of the children pointers
- Before installing their copy they validate that all these nodes have remained intact
  - validation and installation are performed atomically using an HTM transaction

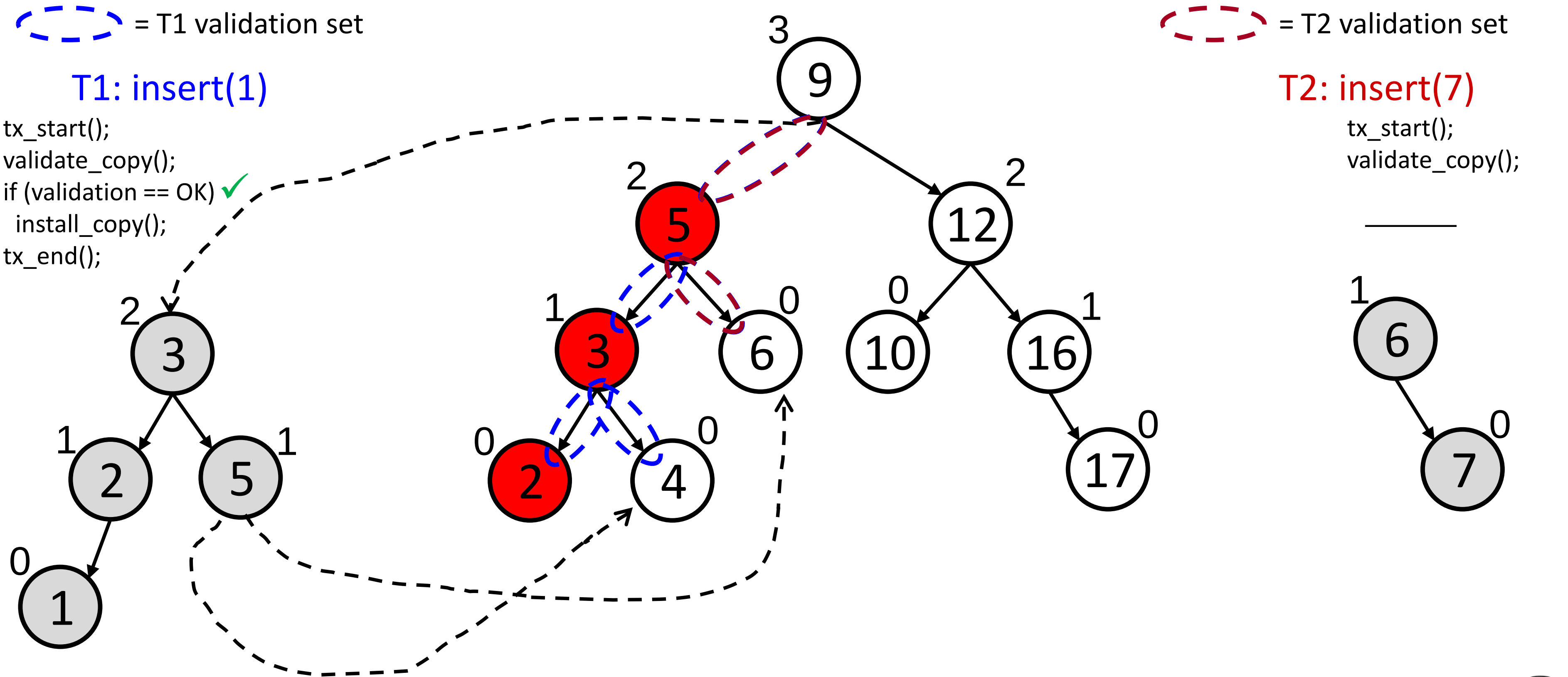


CSLab National Technical University of Athens

RCU-HTM: Multiple Updaters

RCU-HTM overcomes the problem of “lost” updates by exploiting HTM in the following way:

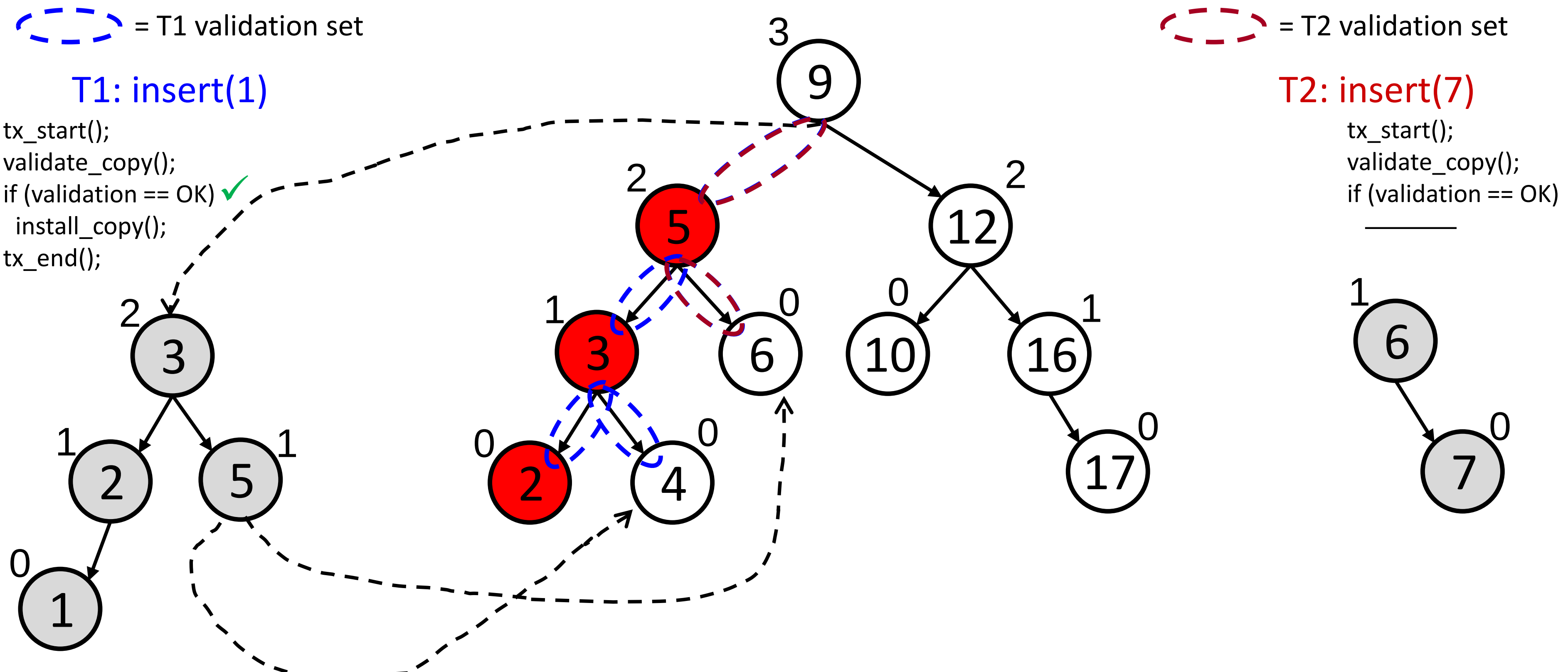
- Updaters keep track of the state of the traversed and the copied nodes, i.e., the addresses of the children pointers
- Before installing their copy they validate that all these nodes have remained intact
  - validation and installation are performed atomically using an HTM transaction



RCU-HTM: Multiple Updaters

RCU-HTM overcomes the problem of “lost” updates by exploiting HTM in the following way:

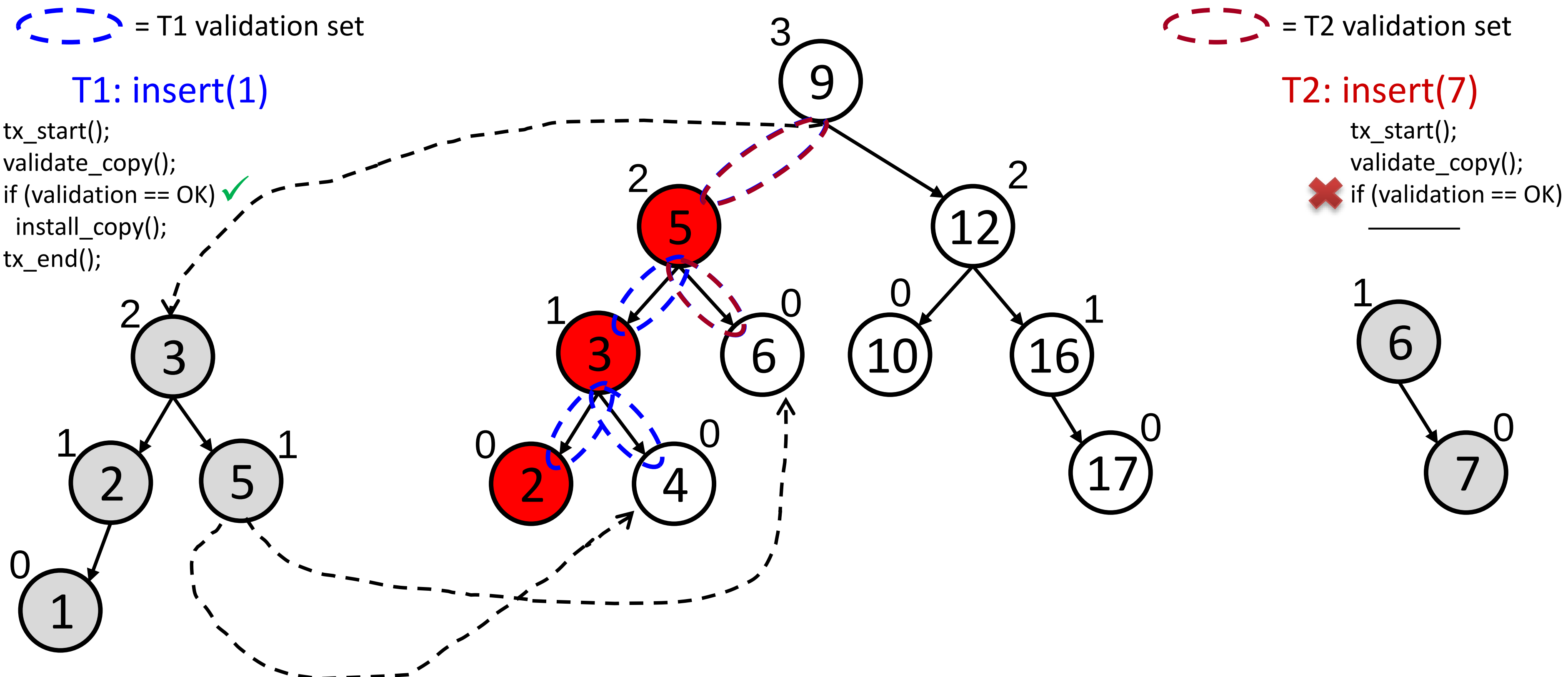
- Updaters keep track of the state of the traversed and the copied nodes, i.e., the addresses of the children pointers
- Before installing their copy they validate that all these nodes have remained intact
  - validation and installation are performed atomically using an HTM transaction



RCU-HTM: Multiple Updaters

RCU-HTM overcomes the problem of “lost” updates by exploiting HTM in the following way:

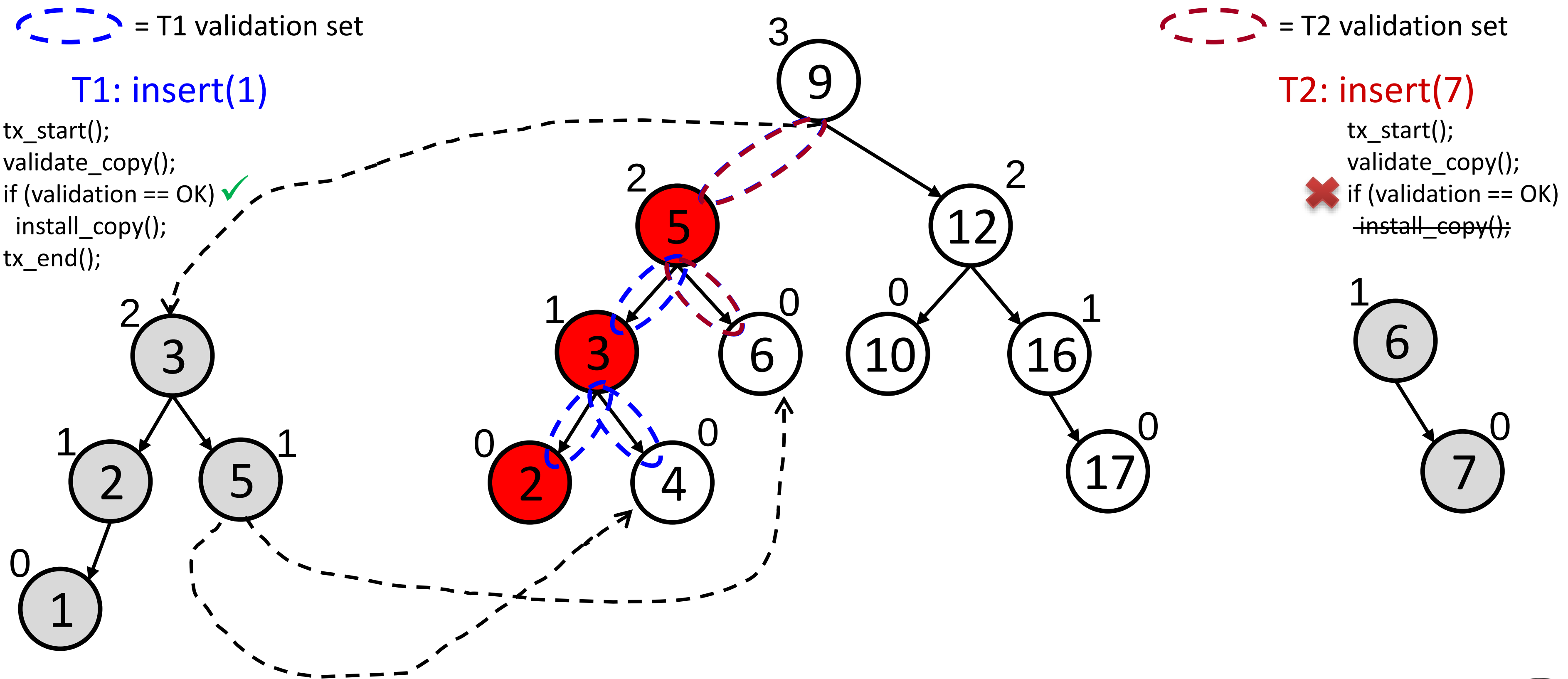
- Updaters keep track of the state of the traversed and the copied nodes, i.e., the addresses of the children pointers
- Before installing their copy they validate that all these nodes have remained intact
  - validation and installation are performed atomically using an HTM transaction



RCU-HTM: Multiple Updaters

RCU-HTM overcomes the problem of “lost” updates by exploiting HTM in the following way:

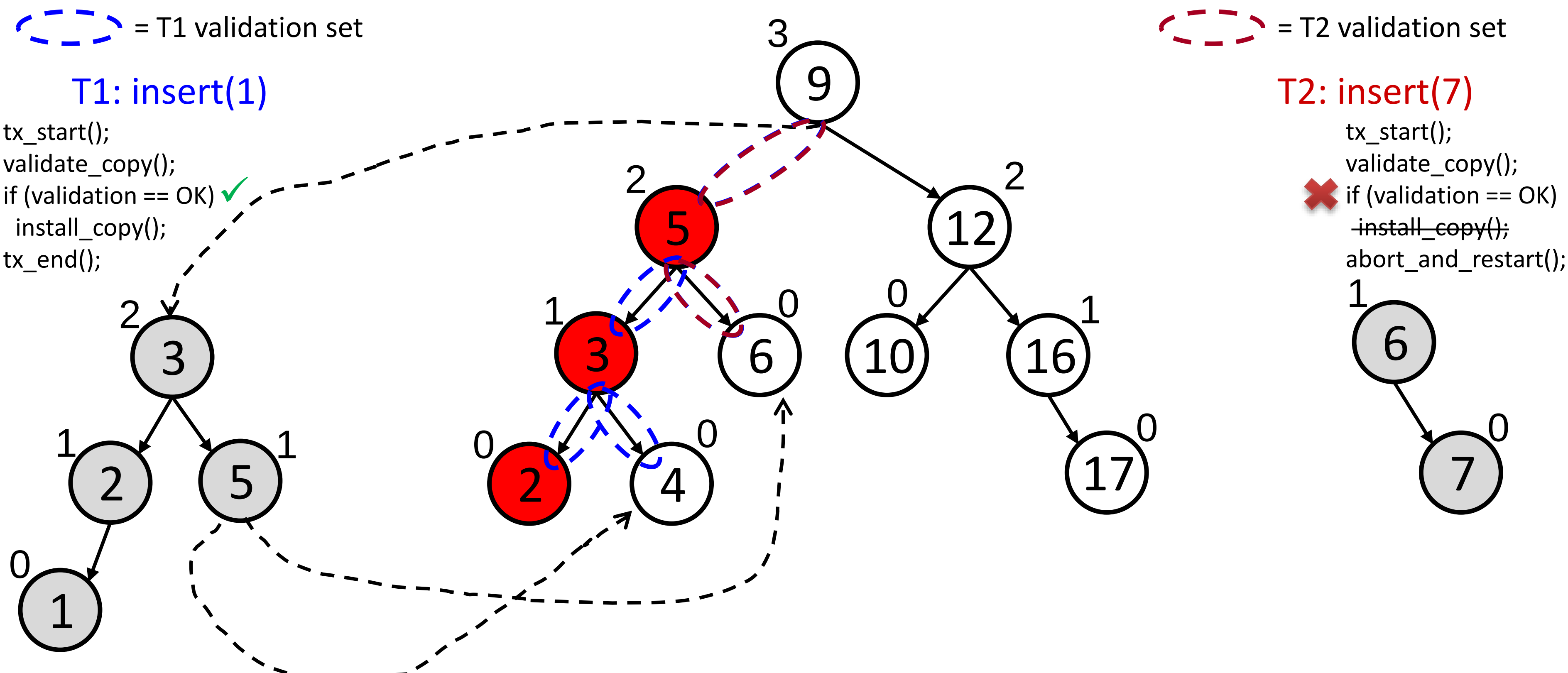
- Updaters keep track of the state of the traversed and the copied nodes, i.e., the addresses of the children pointers
- Before installing their copy they validate that all these nodes have remained intact
  - validation and installation are performed atomically using an HTM transaction



RCU-HTM: Multiple Updaters

RCU-HTM overcomes the problem of "lost" updates by exploiting HTM in the following way:

- Updaters keep track of the state of the traversed and the copied nodes, i.e., the addresses of the children pointers
- Before installing their copy they validate that all these nodes have remained intact
  - validation and installation are performed atomically using an HTM transaction



Experimental Setup

- Intel Broadwell-EP Xeon E5-2699 v4
  - 22 cores / 44 hyperthreads @ 2.2GHz
  - 64GB RAM
- Experimental methodology:
  - Threads run for 2 seconds, executing randomly chosen operations (lookups/inserts/deletes)
  - 3 Workloads:
    - Read-only: 100% lookups
    - Read-dominated: 80% lookups, 10% inserts, 10% deletes
    - Write-only: 0% lookups, 50% inserts, 50% deletes
  - 5 tree sizes
    - Small (200 keys) to large (20M keys)

Concurrent BST implementations

Category	Name	Balanced	Internal	On-time deletion	Asynchronized traversals	Multiple updaters
Lock-free	Ellen et al. [PODC’10]	✗	✗	✓	✓	✓
Lock-free	Howley et al. [SPAA’12]	✗	✓	✗	✗	✓
Lock-free	Natarajan et al. [PPoPP’14]	✗	✗	✓	✓	✓
Lock-free	Chatterjee et al. [PODC’14]	✗	✓	✗	✗	✓
Lock-free	Brown et al. [PPoPP’14]	✗	✗	✓	✓	✓
Locks	Bronson et al. [PPoPP’10]	✗	✗	✗	✗	✓
Locks	Crain et al. [EuroPar’13]	✗	✗	✗	✓	✓
Locks	Drachsler et al. [PPoPP’14]	✗	✓	✓	✓	✓
RCU	Howard et al. [CCPE’14]	✓	✓	✗	✓	✗
RCU	Arbel et al. [PODC’14]	✗	✓	✗	✓	✓
TM	Crain et al. [PPoPP’14]	✗	✓	✗	✗	✓
TM	Avni et al. [TRANSACT’14]	✓	✓	✓	✗	✓
	RCU-HTM	✓	✓	✓	✓	✓

Concurrent BST implementations

	Name	Balanced	Internal	On-time deletion	Asynchronized traversals	Multiple updaters
Lock-free	Ellen et al. [PODC'10]	✗	✗	✓	✓	✓
	Howley et al. [SPAA'12]	✗	✓	✗	✗	✓
	Natarajan et al. [PPoPP'14]	✗	✗	✓	✓	✓
	Chatterjee et al. [PODC'14]	✗	✓	✗	✗	✓
	Brown et al. [PPoPP'14]	✗	✗	✓	✓	✓
Locks	Bronson et al. [PPoPP'10]	✗	✗	✗	✗	✓
	Crain et al. [EuroPar'13]	✗	✗	✗	✓	✓
	Drachsler et al. [PPoPP'14]	✗	✓	✓	✓	✓
RCU	Howard et al. [CCPE'14]	✓	✓	✗	✓	✗
	Arbel et al. [PODC'14]	✗	✓	✗	✓	✓
TM	Crain et al. [PPoPP'14]	✗	✓	✗	✗	✓
	Avni et al. [TRANSACT'14]	✓	✓	✓	✗	✓
	RCU-HTM	✓	✓	✓	✓	✓

Concurrent BST implementations

	Name	Balanced	Internal	On-time deletion	Asynchronized traversals	Multiple updaters
Lock-free	Ellen et al. [PODC'10]	✗	✗	✓	✓	✓
Lock-free	Howley et al. [SPAA'12]	✗	✓	✗	✗	✓
Lock-free	Natarajan et al. [PPoPP'14]	✗	✗	✓	✓	✓
Lock-free	Chatterjee et al. [PODC'14]	✗	✓	✗	✗	✓
Lock-free	Brown et al. [PPoPP'14]	✗	✗	✓	✓	✓
Locks	Bronson et al. [PPoPP'10]	✗	✗	✗	✗	✓
Locks	Crain et al. [EuroPar'13]	✗	✗	✗	✓	✓
Locks	Drachsler et al. [PPoPP'14]	✗	✓	✓	✓	✓
RCU	Howard et al. [CCPE'14]	✓	✓	✗	✓	✗
RCU	Arbel et al. [PODC'14]	✗	✓	✗	✓	✓
TM	Crain et al. [PPoPP'14]	✗	✓	✗	✗	✓
TM	Avni et al. [TRANSACT'14]	✓	✓	✓	✗	✓
	RCU-HTM	✓	✓	✓	✓	✓

Concurrent BST implementations

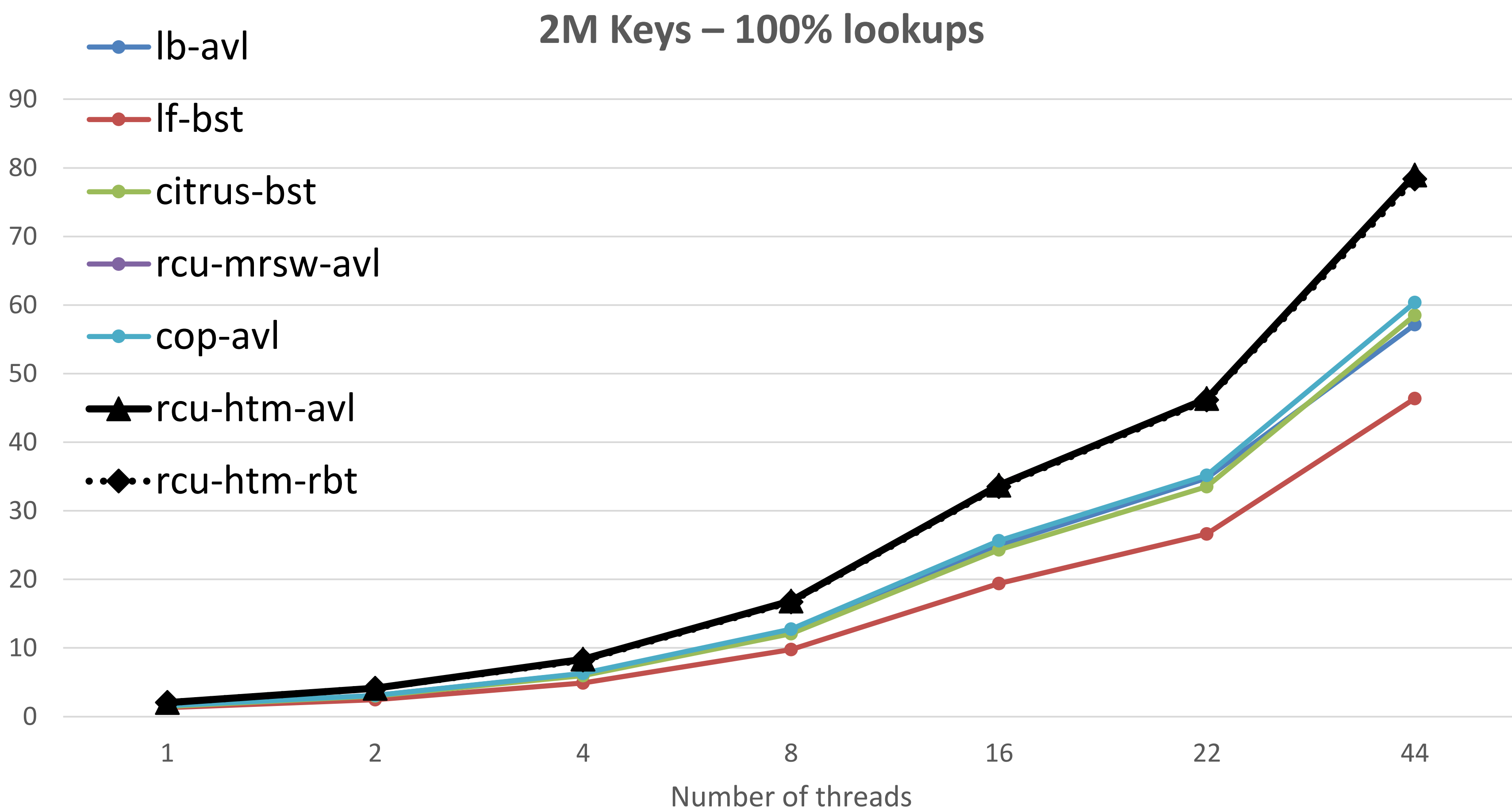
	Name	Balanced	Internal	On-time deletion	Asynchronized traversals	Multiple updaters
Lock-free	Ellen et al. [PODC’10]	✗	✗	✓	✓	✓
	Howley et al. [SPAA’12]	✗	✓	✗	✗	✓
	Natarajan et al. [PPoPP’14]	✗	✗	✓	✓	✓
	Chatterjee et al. [PODC’14]	✗	✓	✗	✗	✓
	Brown et al. [PPoPP’14]	✗	✗	✓	✓	✓
Locks	Bronson et al. [PPoPP’10]	✗	✗	✗	✗	✓
	Crain et al. [EuroPar’13]	✗	✗	✗	✓	✓
	Drachsler et al. [PPoPP’14]	✗	✓	✓	✓	✓
RCU	Howard et al. [CCPE’14]	✓	✓	✗	✓	✗
	Arbel et al. [PODC’14]	✗	✓	✗	✓	✓
TM	Crain et al. [PPoPP’14]	✗	✓	✗	✗	✓
	Avni et al. [TRANSACT’14]	✓	✓	✓	✗	✓
	RCU-HTM	✓	✓	✓	✓	✓

Concurrent BST implementations

	Name	Balanced	Internal	On-time deletion	Asynchronized traversals	Multiple updaters
Lock-free	Ellen et al. [PODC'10]	✗	✗	✓	✓	✓
	Howley et al. [SPAA'12]	✗	✓	✗	✗	✓
	Natarajan et al. [PPoPP'14]	✗	✗	✓	✓	✓
	Chatterjee et al. [PODC'14]	✗	✓	✗	✗	✓
	Brown et al. [PPoPP'14]	✗	✗	✓	✓	✓
Locks	Bronson et al. [PPoPP'10]	✗	✗	✗	✗	✓
	Crain et al. [EuroPar'13]	✗	✗	✗	✓	✓
	Drachsler et al. [PPoPP'14]	✗	✓	✓	✓	✓
RCU	Howard et al. [CCPE'14]	✓	✓	✗	✓	✗
	Arbel et al. [PODC'14]	✗	✓	✗	✓	✓
TM	Crain et al. [PPoPP'14]	✗	✓	✗	✗	✓
	Avni et al. [TRANSACT'14]	✓	✓	✓	✗	✓
	RCU-HTM	✓	✓	✓	✓	✓

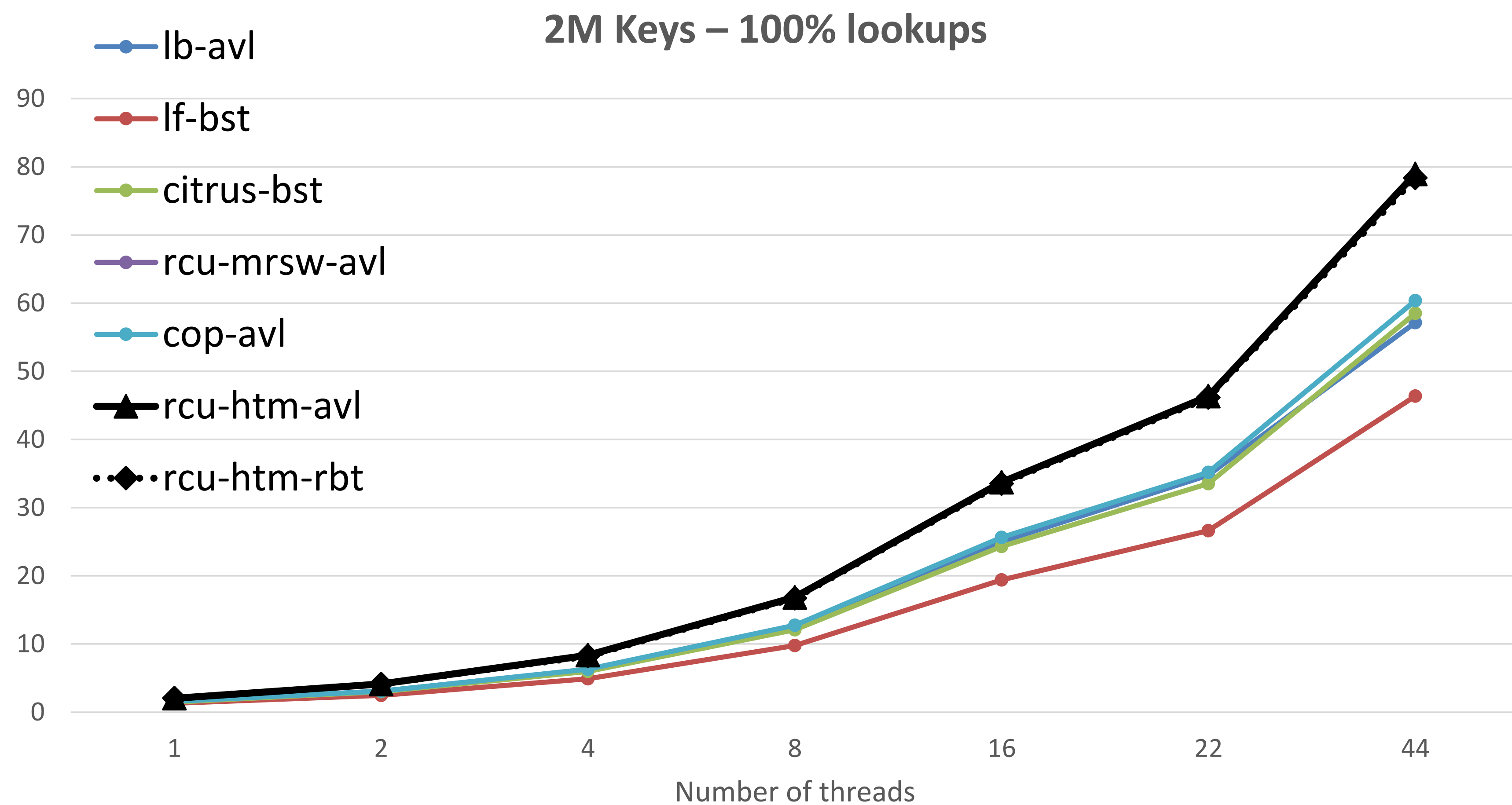
Performance: read-only workloads

Performance: read-only workloads

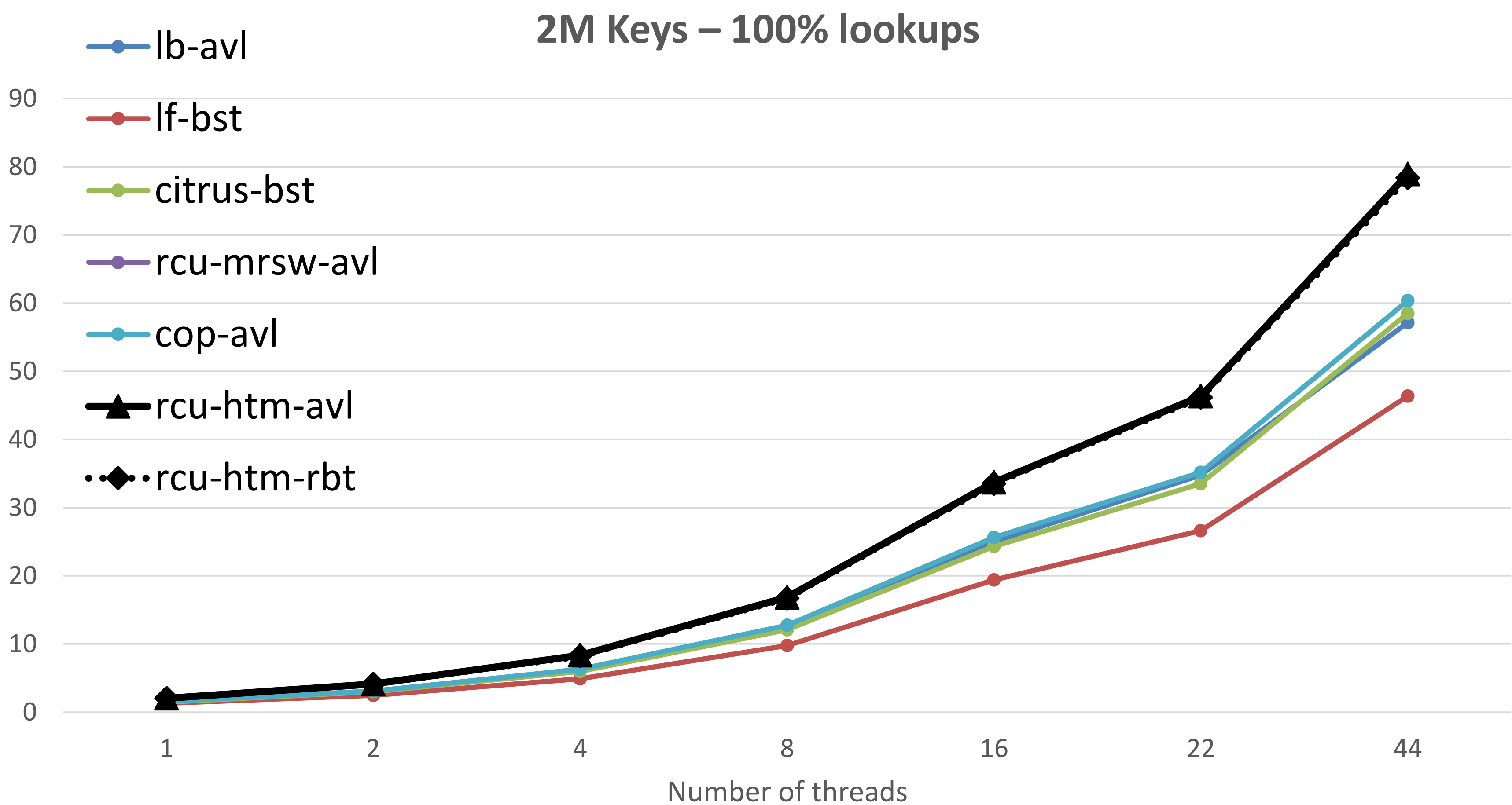


National Technical University of Athens
CSLab

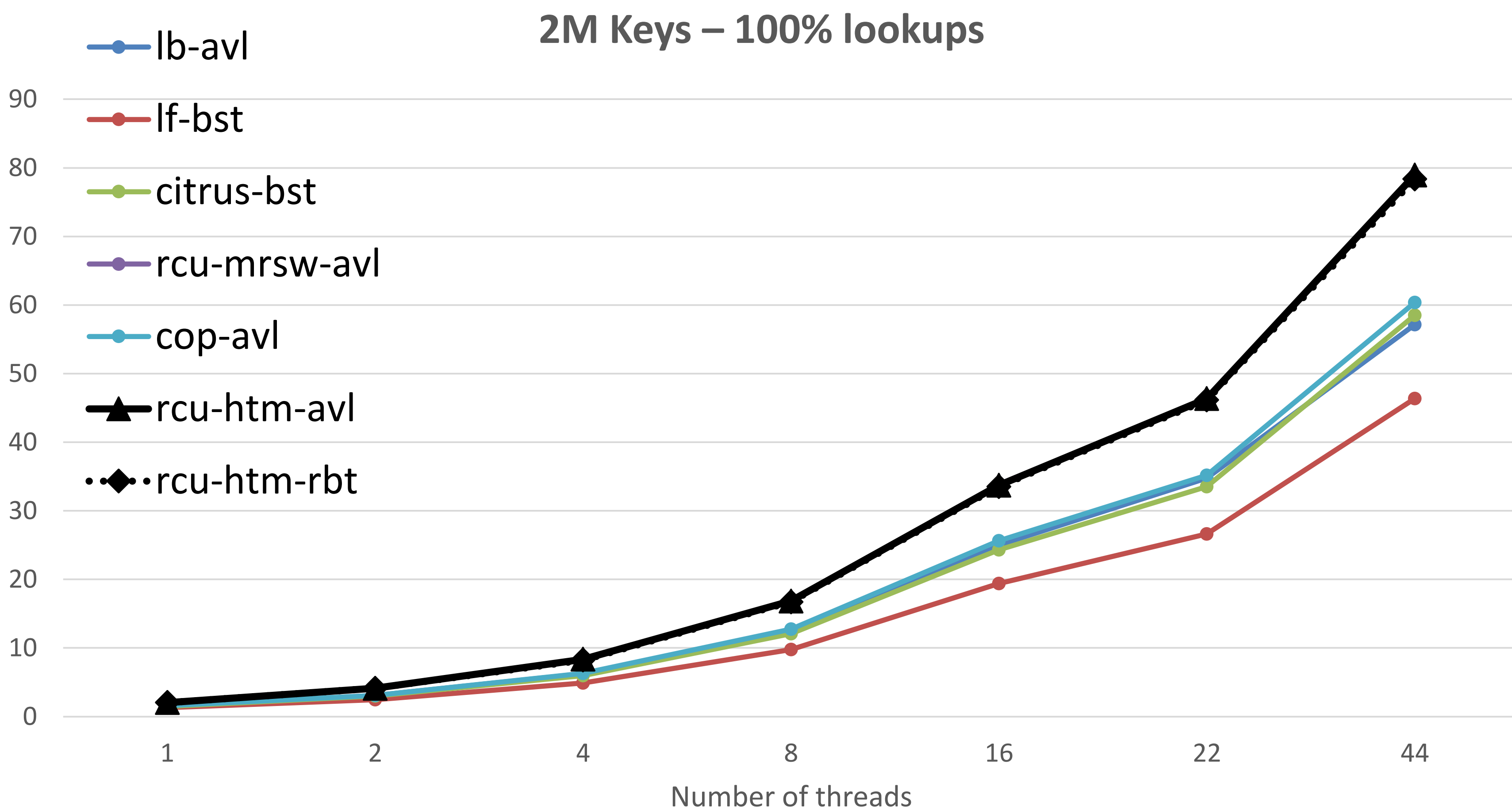
Performance: read-only workloads



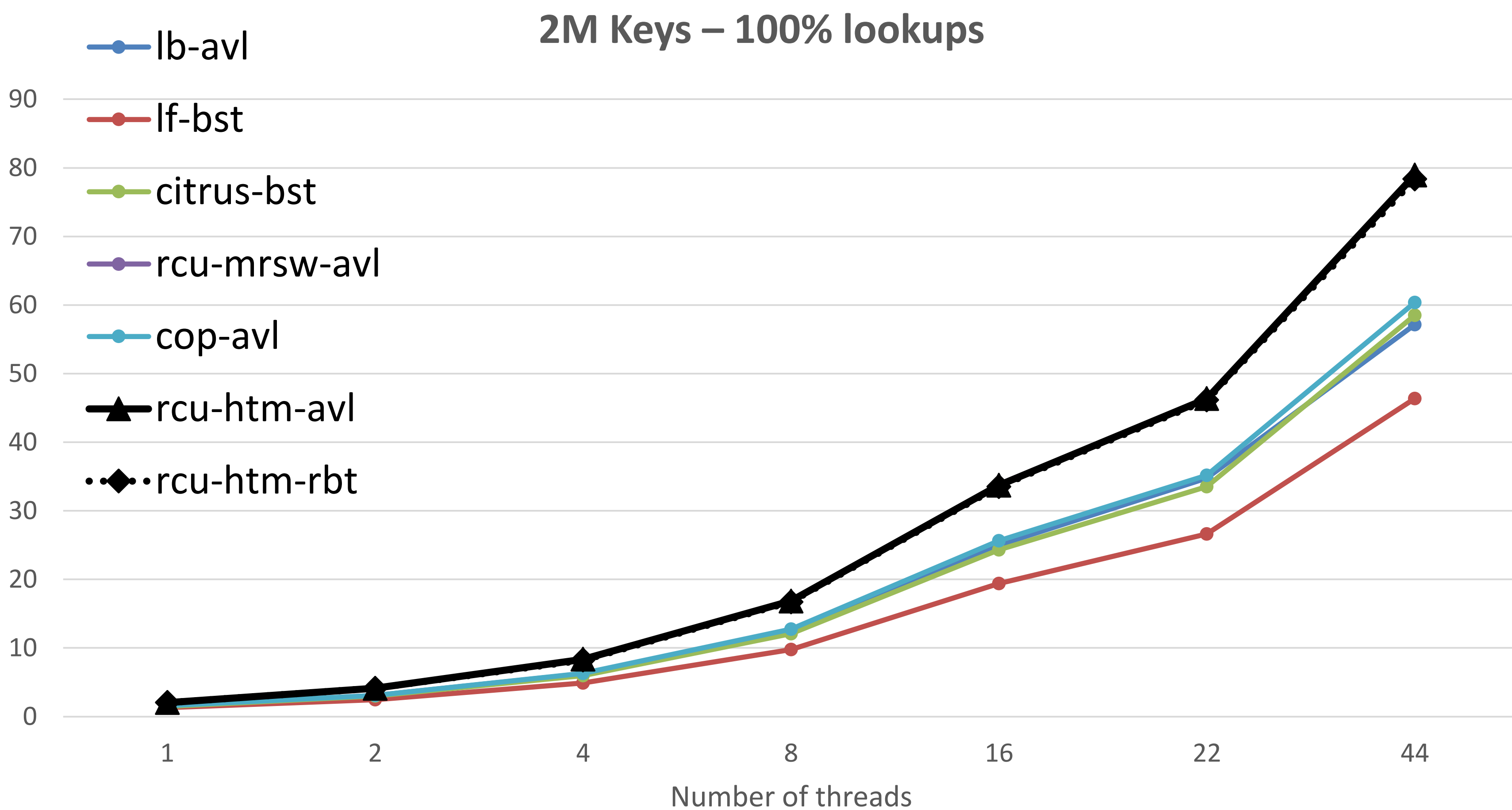
Performance: read-only workloads



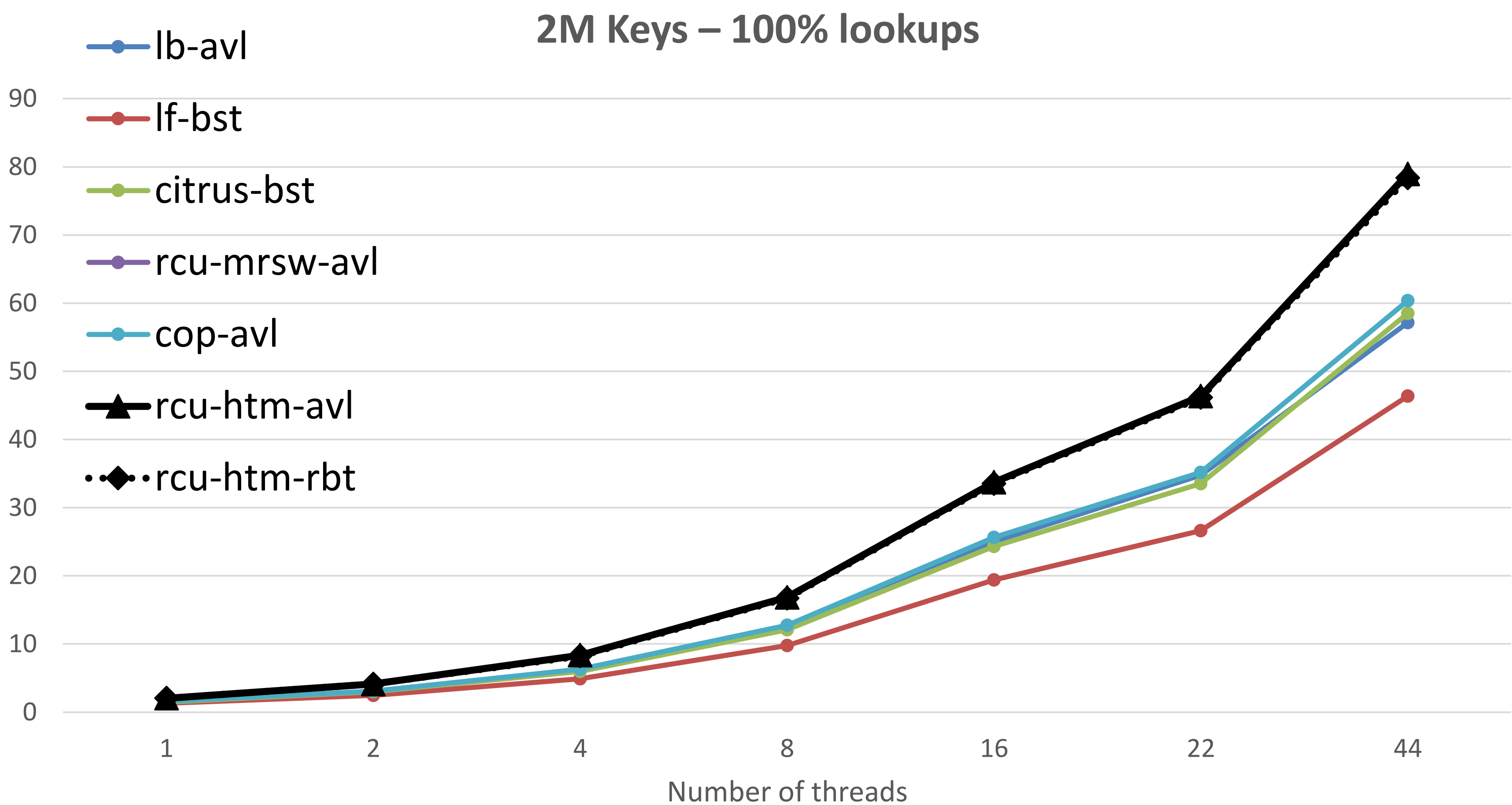
Performance: read-only workloads



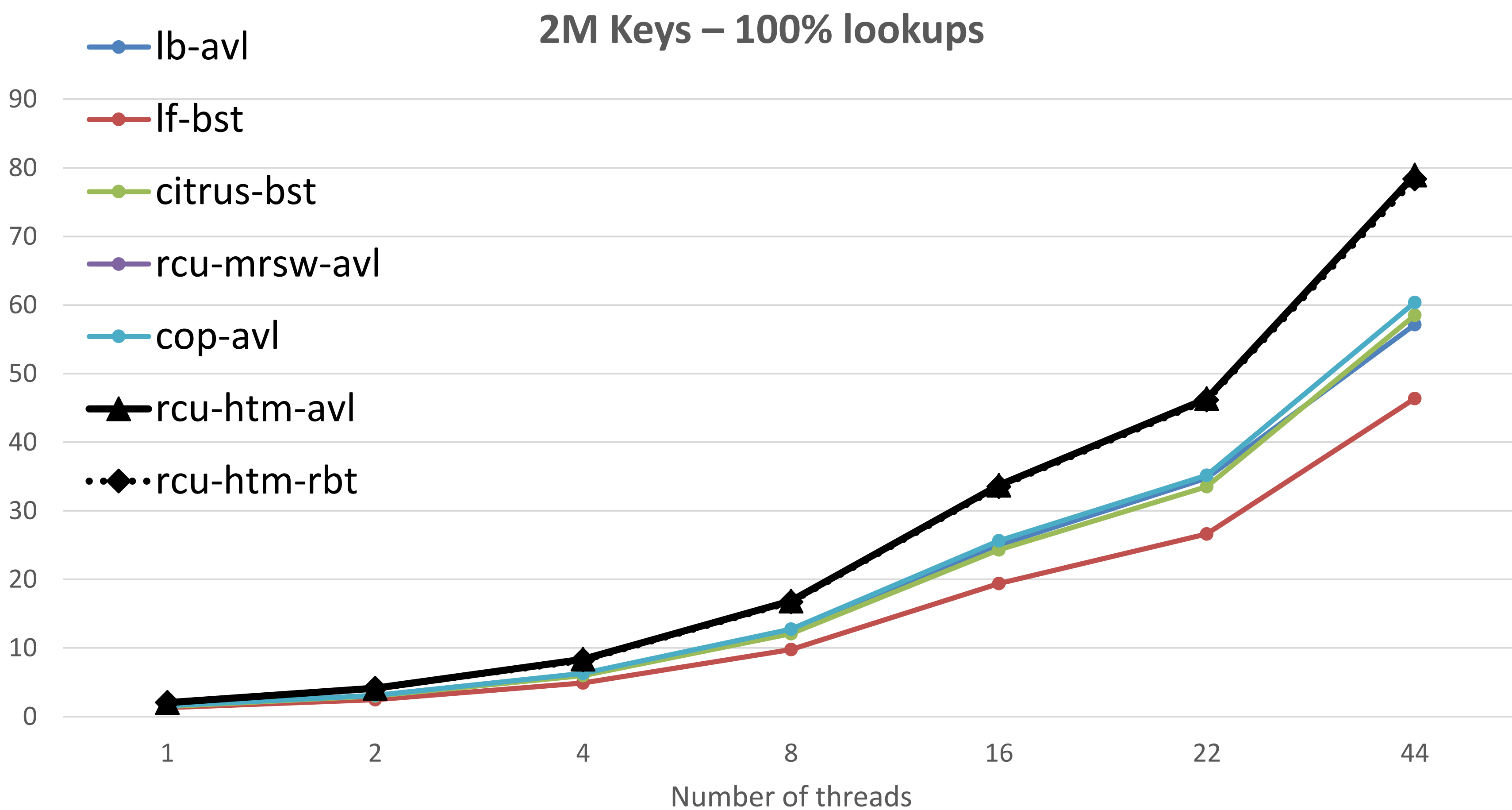
Performance: read-only workloads



Performance: read-only workloads

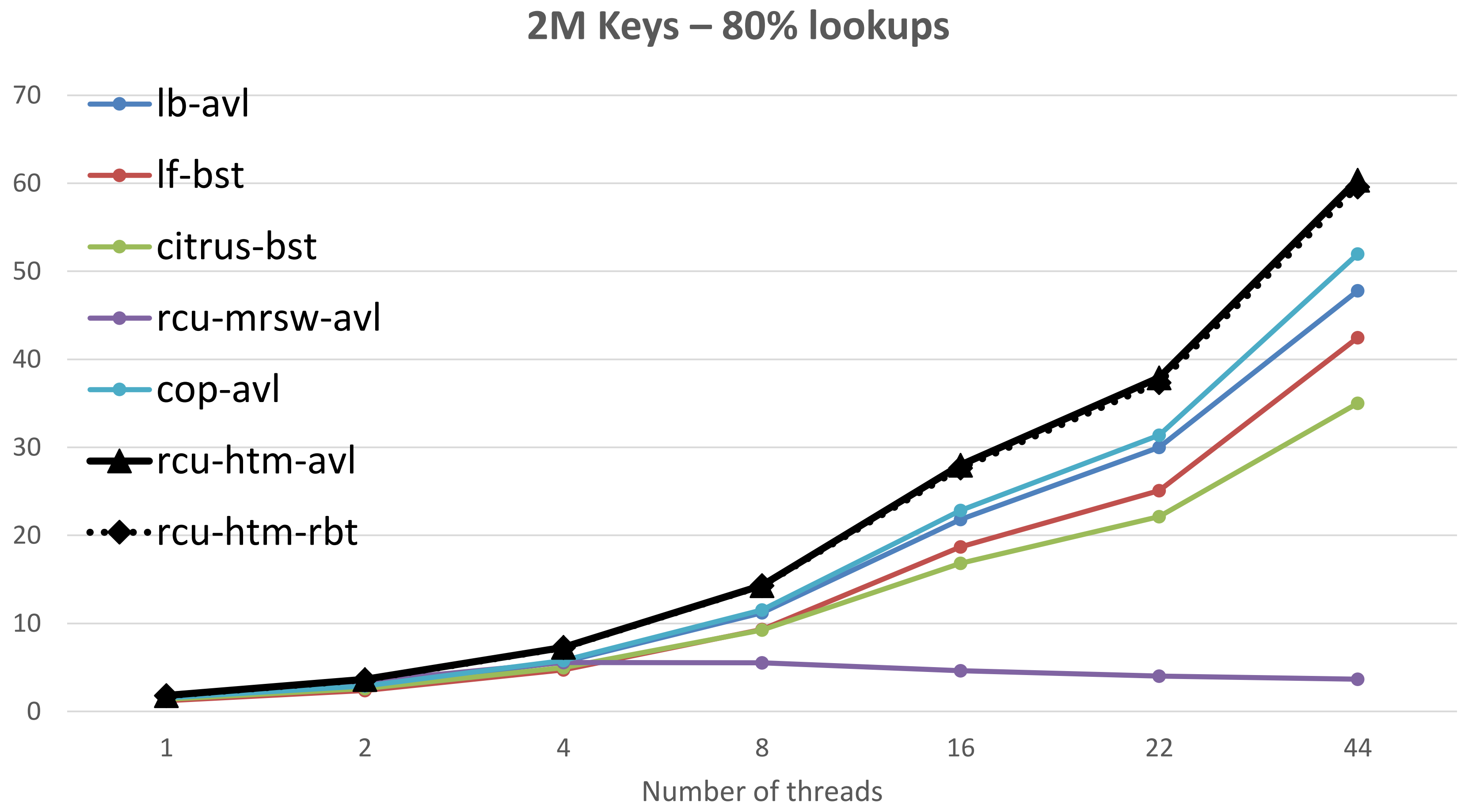


Performance: read-only workloads

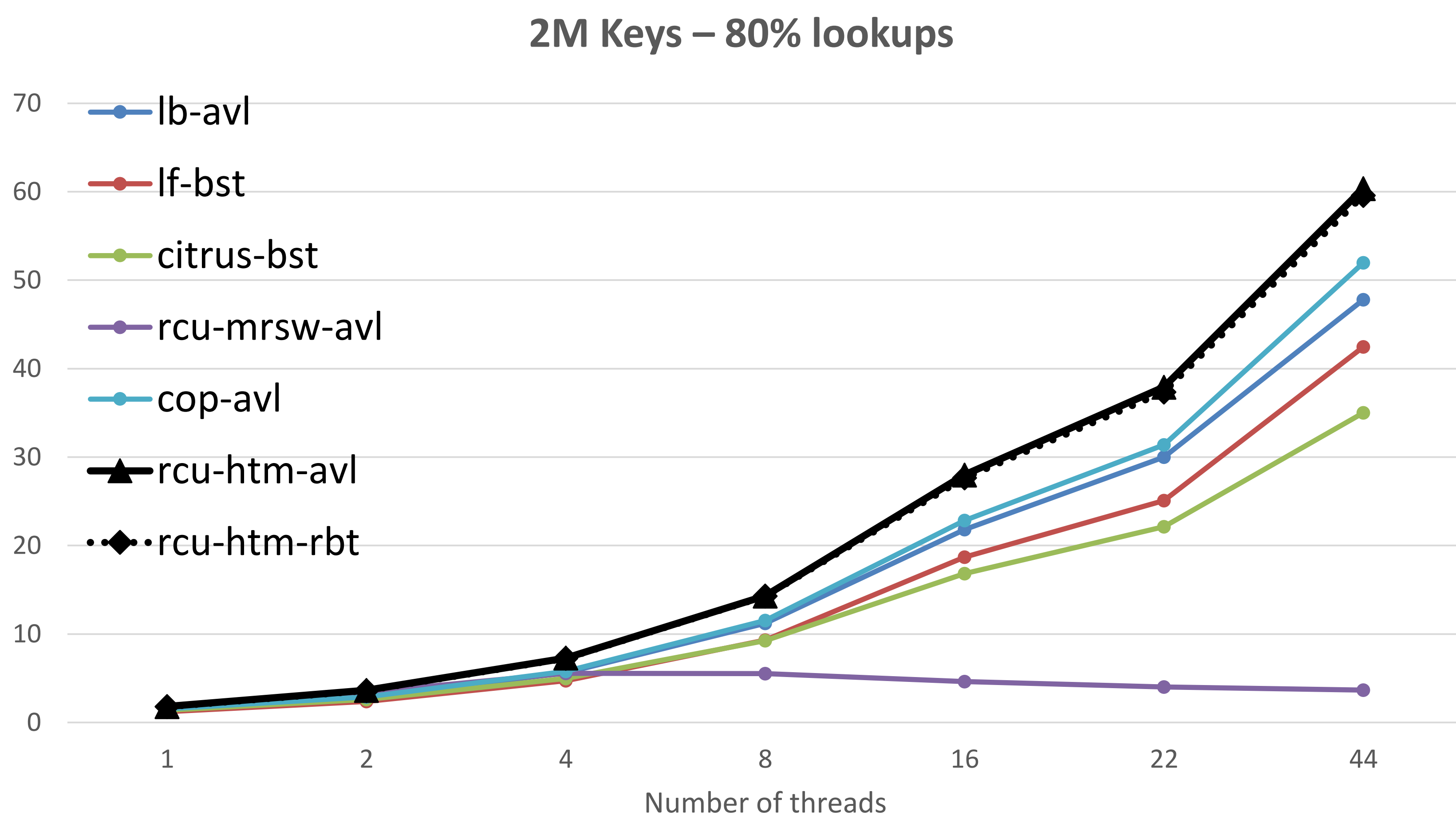


Performance: read-dominated workloads

Performance: read-dominated workloads

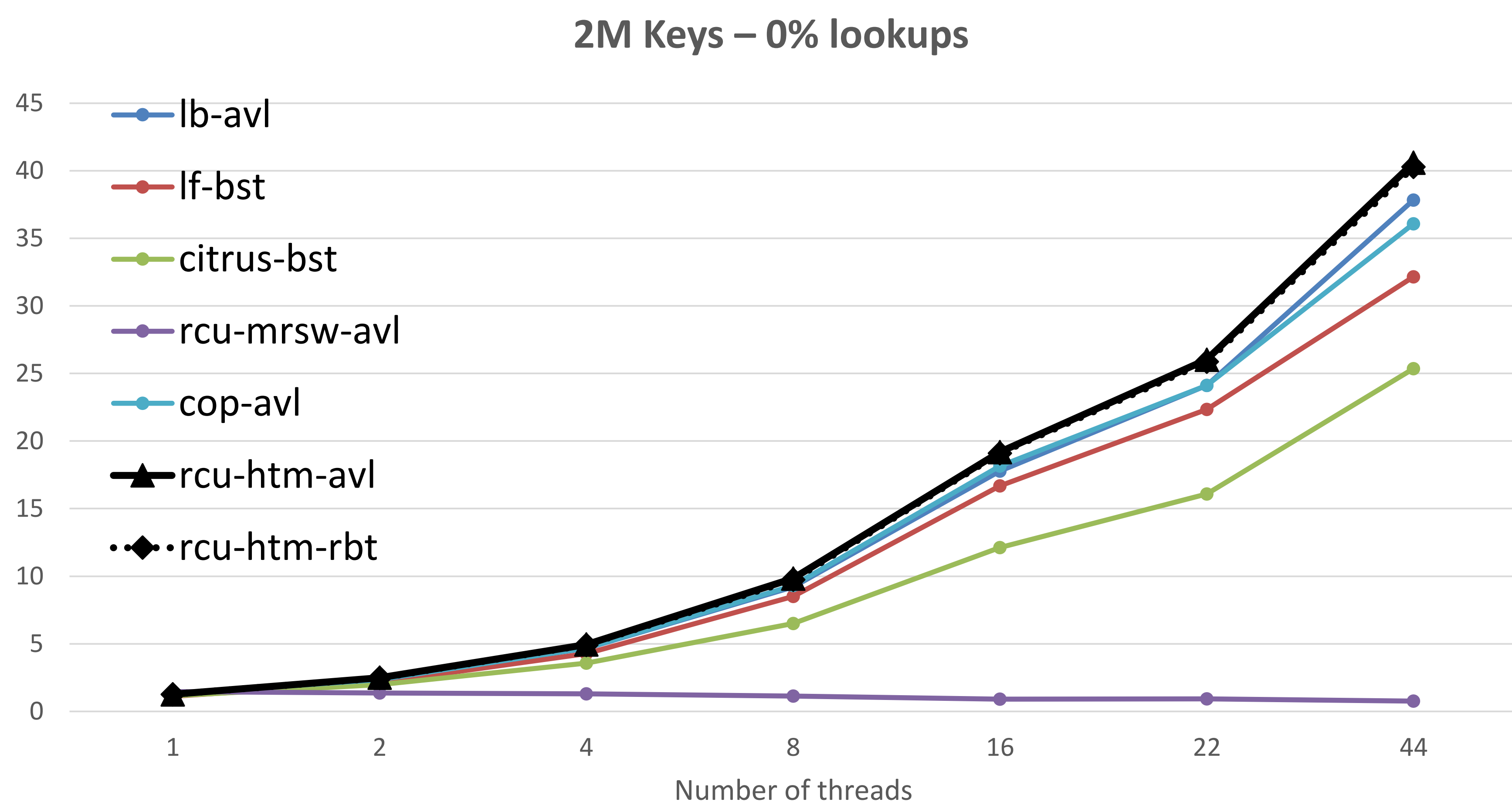


Performance: read-dominated workloads

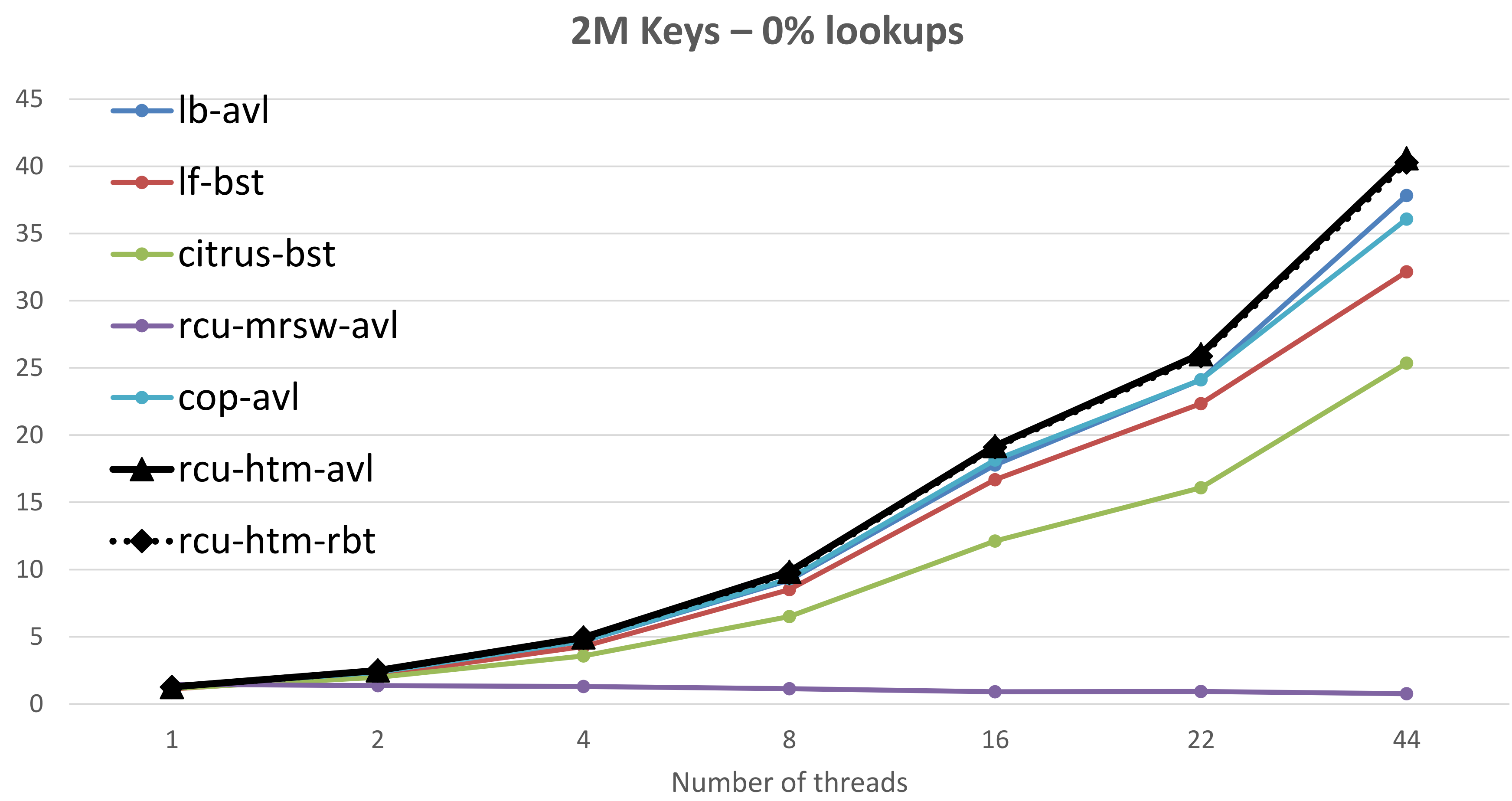


Performance: write-only workloads

Performance: write-only workloads



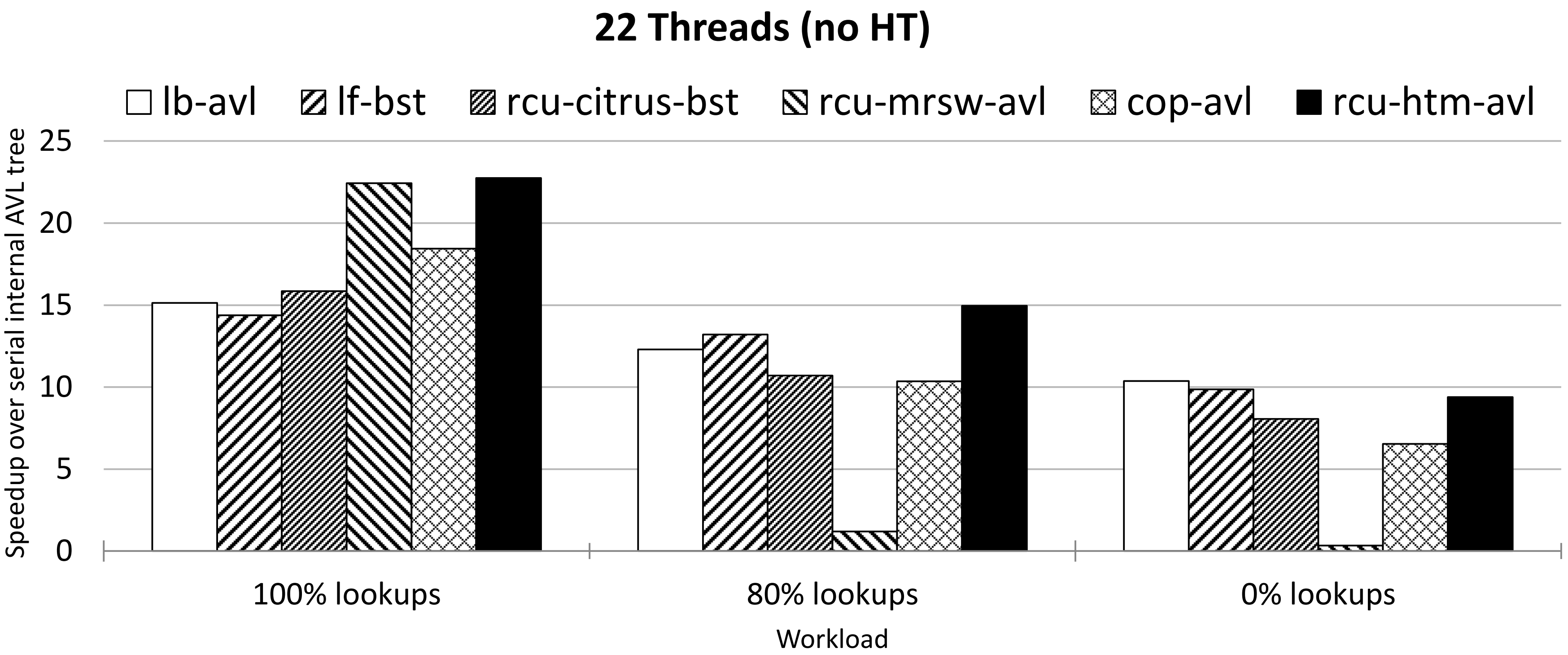
Performance: write-only workloads



National Technical University of Athens CSLab

Performance: per workload average

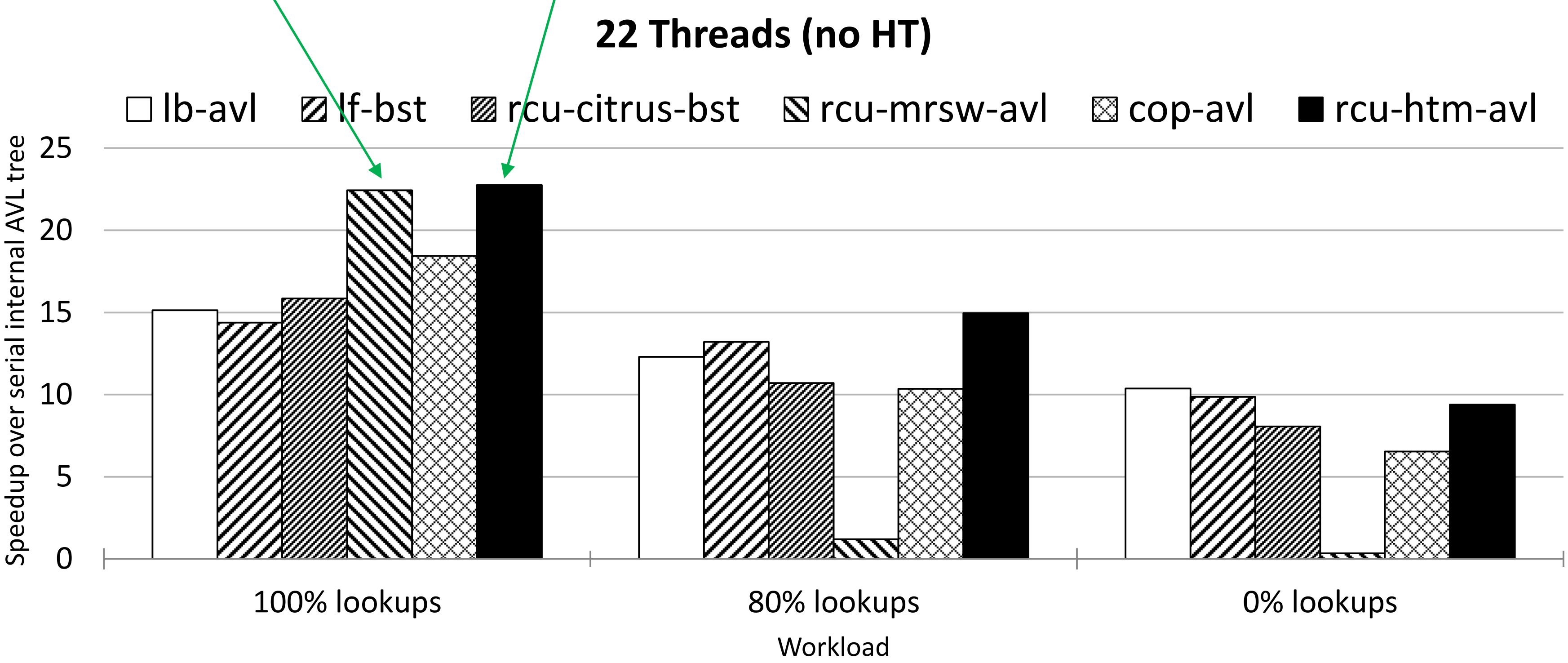
Performance: per workload average



Performance: per workload average

RCU provides the fastest lookups

RCU-HTM is as fast as RCU

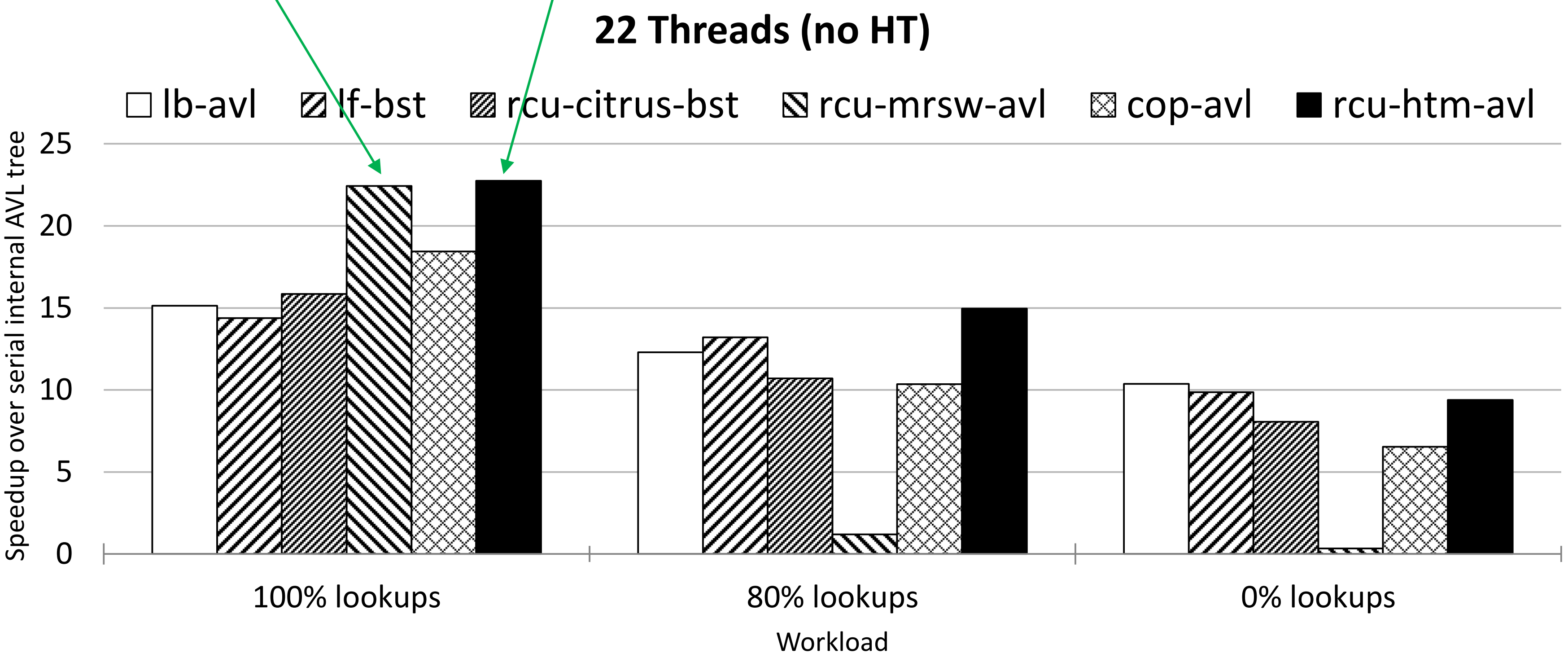


CSLab National Technical University of Athens

Performance: per workload average

RCU provides the fastest lookups

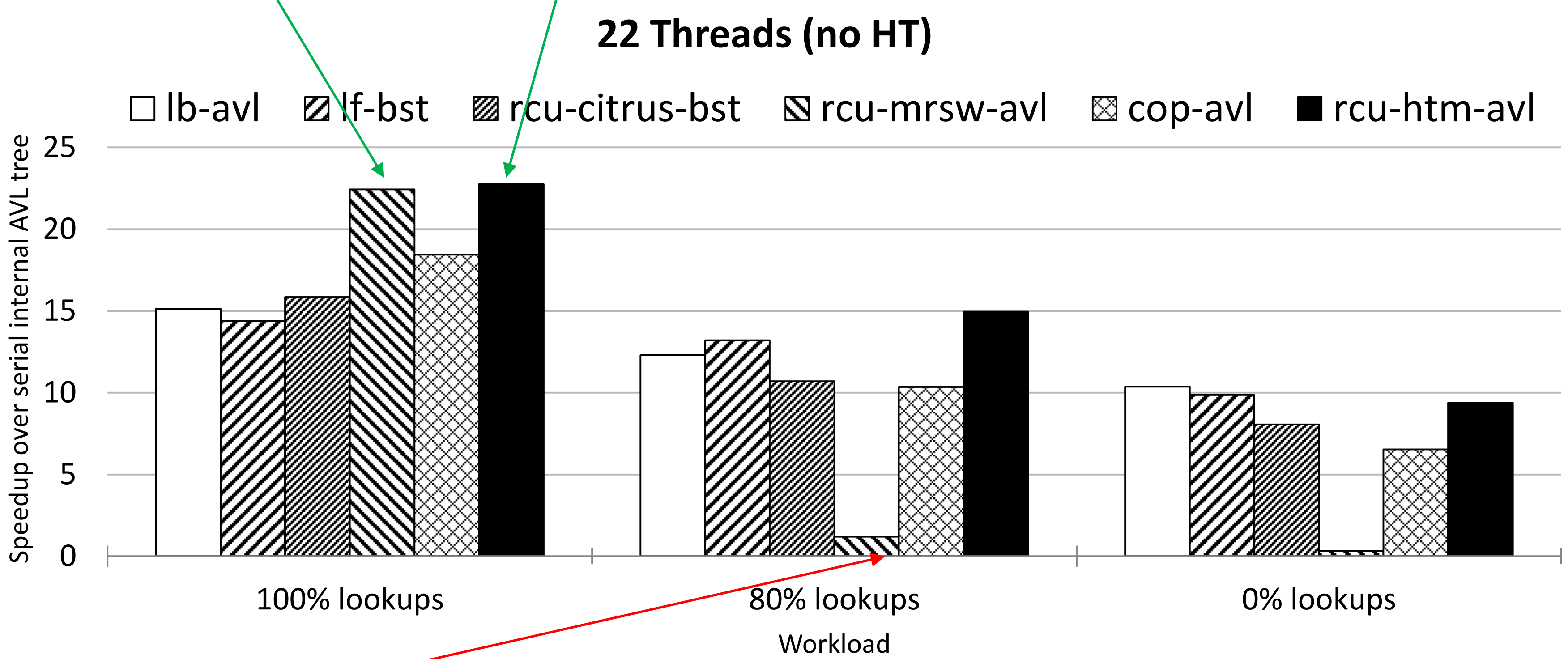
RCU-HTM is as fast as RCU



Performance: per workload average

RCU provides the fastest lookups

RCU-HTM is as fast as RCU

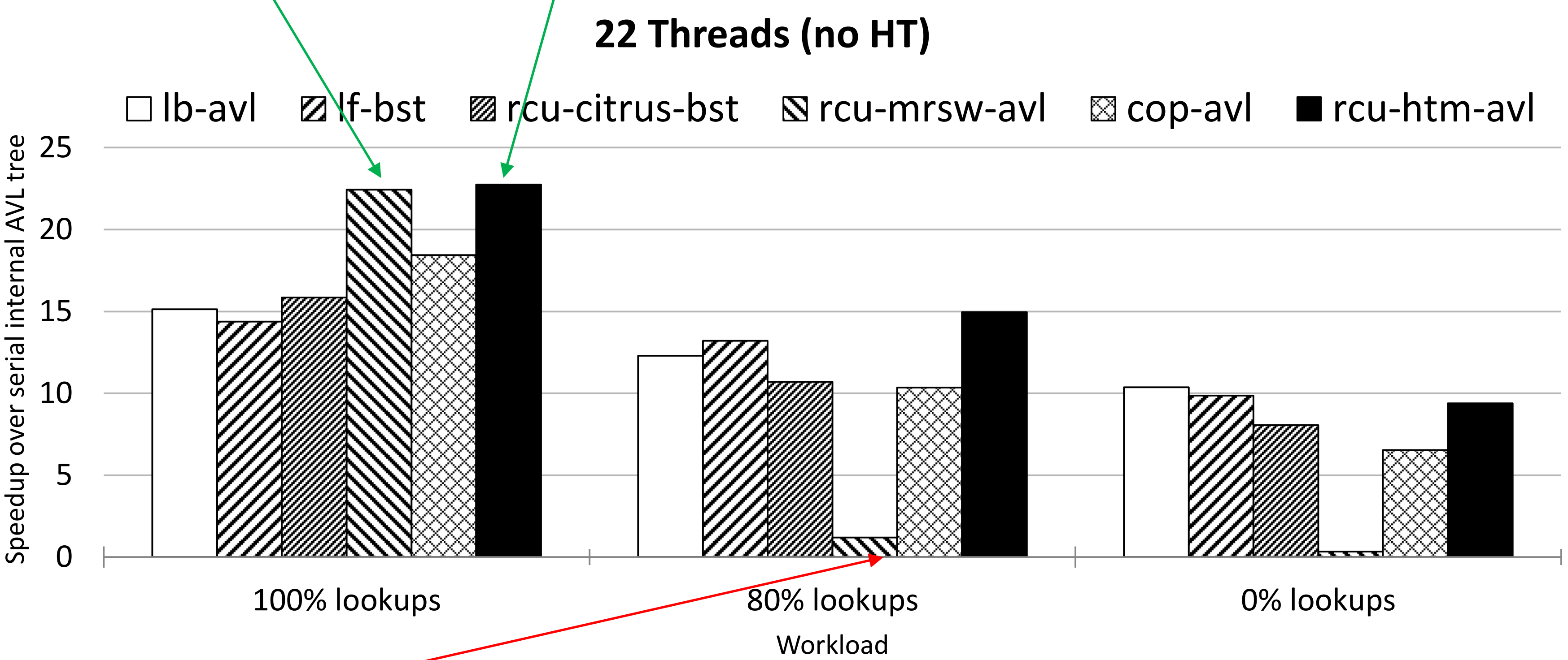


Even with 20% update operations RCU performance collapses due to the writers' single global lock

Performance: per workload average

RCU provides the fastest lookups

RCU-HTM is as fast as RCU

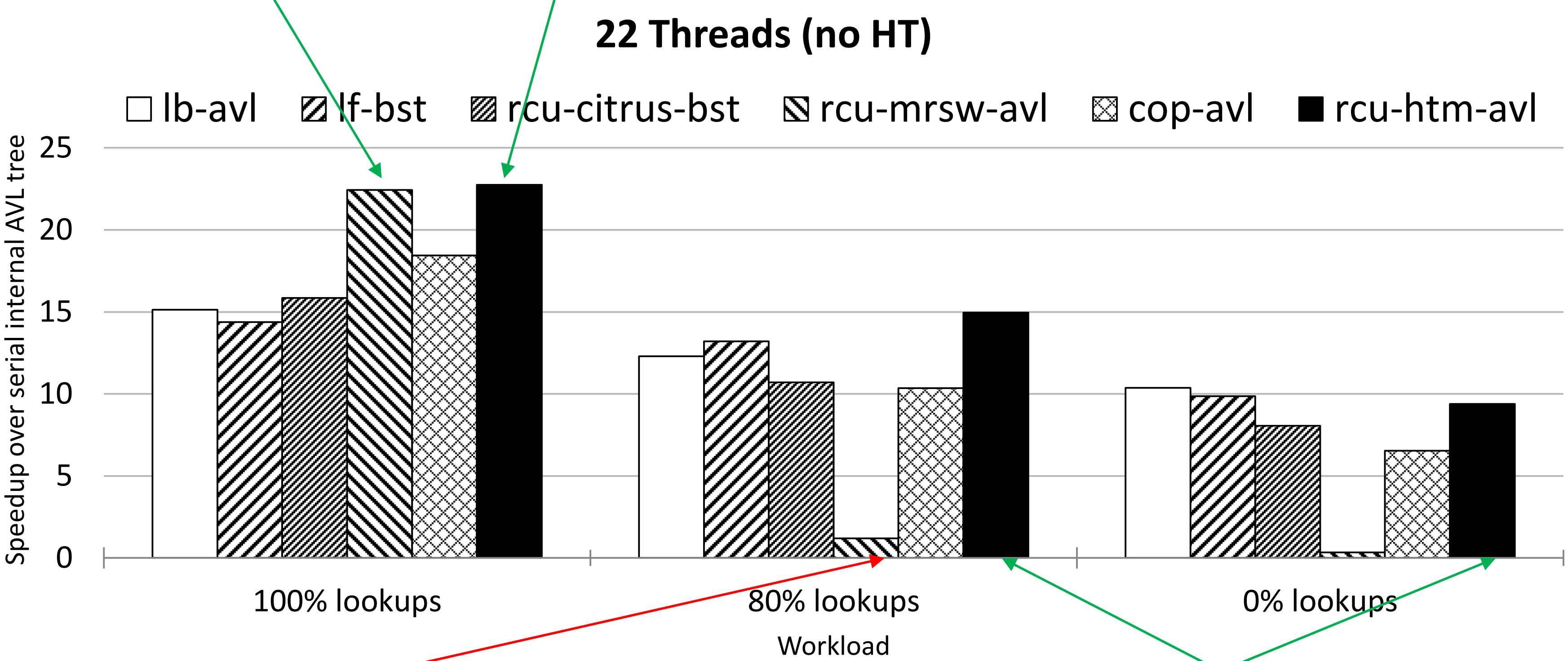


Even with 20% update operations RCU performance collapses due to the writers' single global lock

Performance: per workload average

RCU provides the fastest lookups

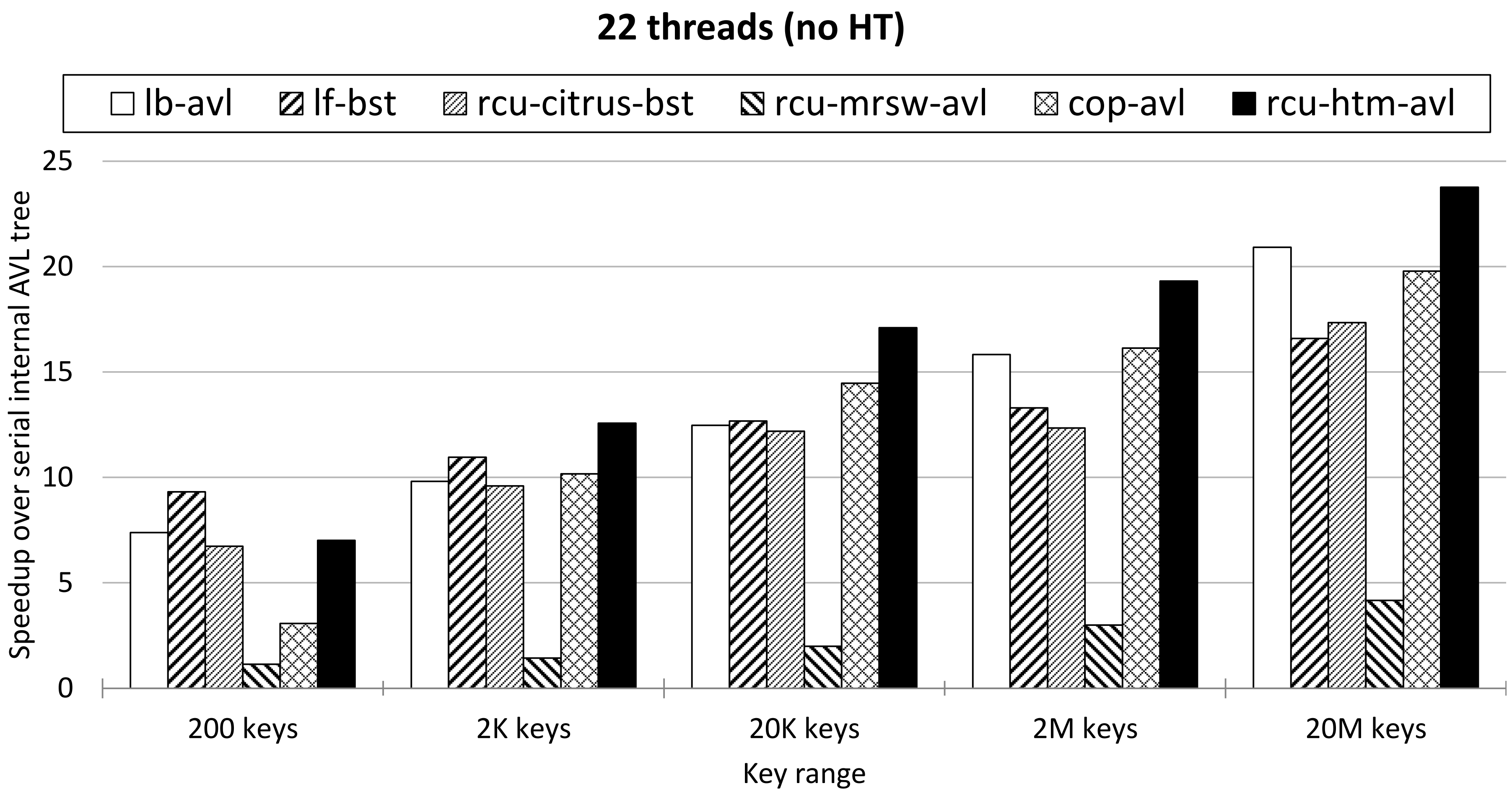
RCU-HTM is as fast as RCU



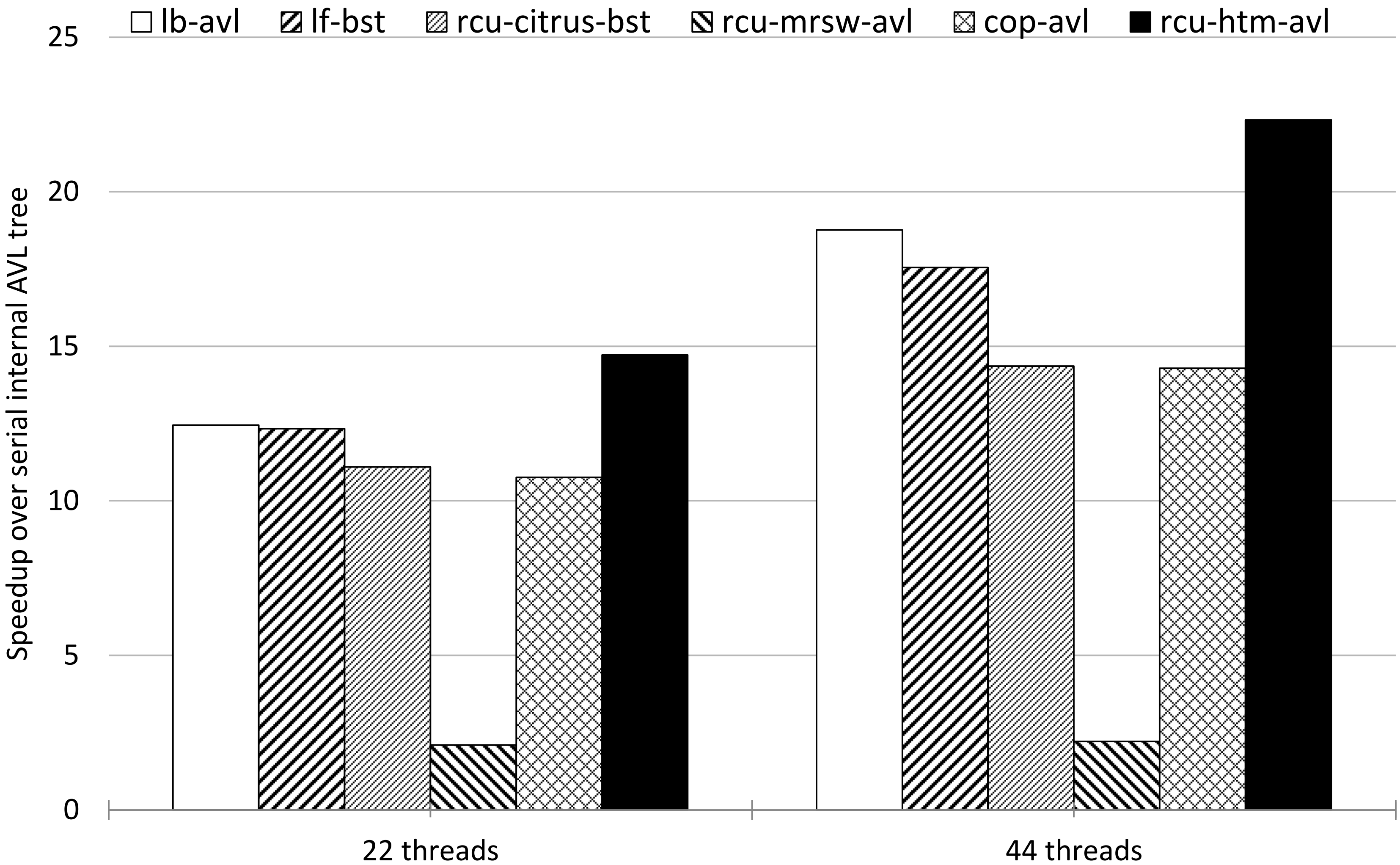
Even with 20% update operations RCU performance collapses due to the writers' single global lock

RCU-HTM efficiently allows multiple updaters and manages to retain its performance on workloads with high update ratio

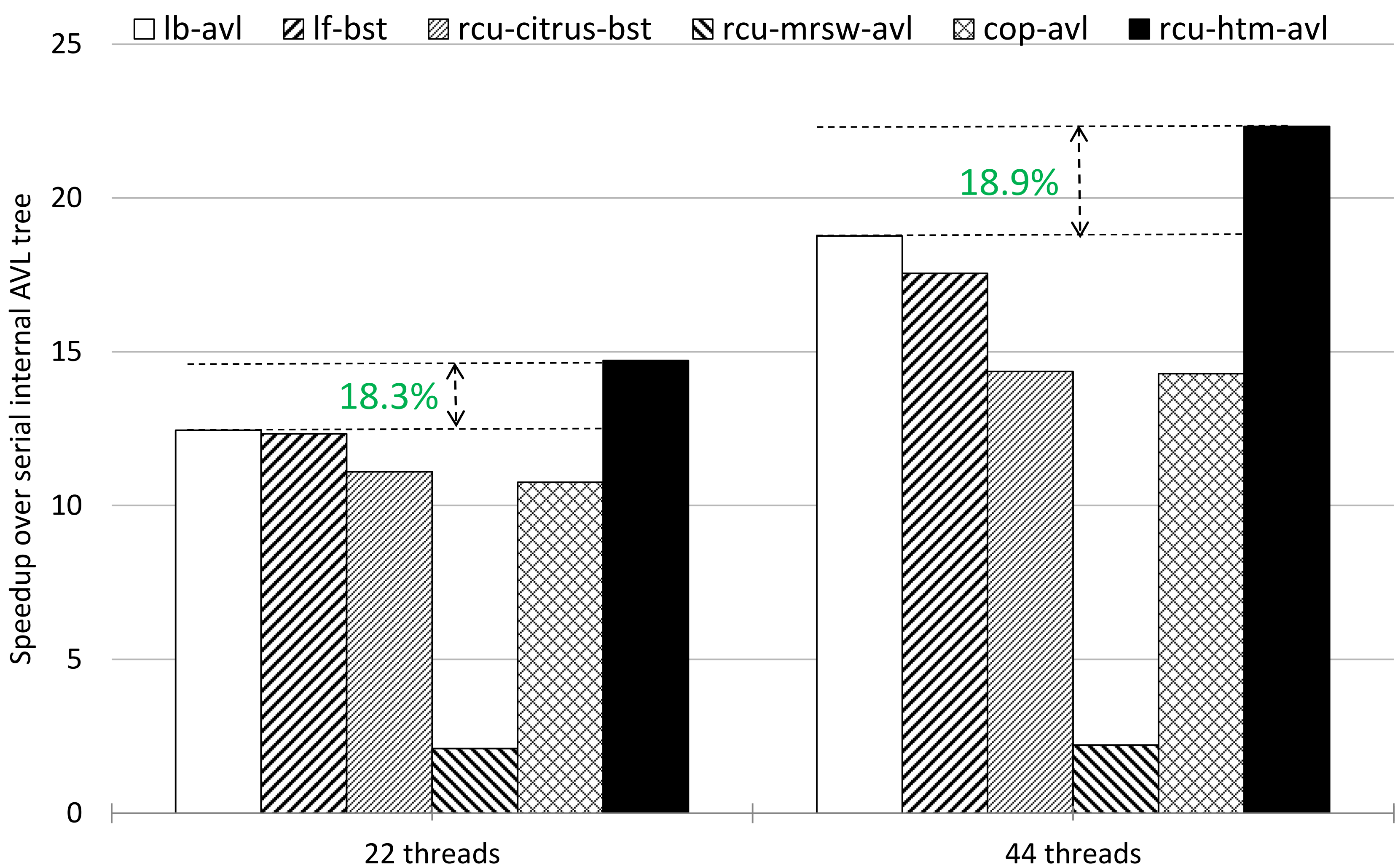
Performance: per tree size average



Performance: overall



Performance: overall



Conclusions

RCU-HTM

- Efficiently combines RCU with HTM
- Provides concurrent binary search trees:
    1. Balanced
    2. Internal
    3. On-time deletion
    4. Asynchronized traversals
    5. Multiple updaters
- **18%** better performance than state-of-the-art BSTs

RCU-HTM AVL and Red-Black trees publicly available:

- `https://github.com/rcu-htm/rcu-htm`

RCU-HTM: Combining RCU with HTM to Implement Highly Efficient Concurrent Binary Search Trees

Dimitrios Siakavaras, Konstantinos Nikas, Georgios Goumas and Nectarios Koziris

National Technical University of Athens (NTUA)
School of Electrical and Computer Engineering (ECE)
Computing Systems Laboratory (CSLab)
{jimsiak,knikas,goumas,nkoziris}@cslab.ece.ntua.gr
http://research.cslab.ece.ntua.gr

THANK YOU!
QUESTIONS?

Backup Slides

Concurrent BSTs

Name	Balanced	Internal	On-time deletion
Ellen et al. [PODC'10]	✘	✘	✔
Howley et al. [SPAA'12]	✘	✔	✘
Natarajan et al. [PPoPP'14]	✘	✘	✔
Chatterjee et al. [PODC'14]	✘	✔	✘
Brown et al. [PPoPP'14]	✘	✘	✔

Lock-free

Atomic operations (e.g., CAS) can only modify a single memory word

Concurrent BSTs

Name	Balanced	Internal	On-time deletion
Ellen et al. [PODC'10]	✘	✘	✔
Howley et al. [SPAA'12]	✘	✔	✘
Natarajan et al. [PPoPP'14]	✘	✘	✔
Chatterjee et al. [PODC'14]	✘	✔	✘
Brown et al. [PPoPP'14]	✘	✘	✔

Lock-free
Atomic operations (e.g., CAS) can only modify a single memory word

Bronson et al. [PPoPP'10]	✘	✘	✘
Crain et al. [EuroPar'13]	✘	✘	✘
Drachsler et al. [PPoPP'14]	✘	✔	✔

Lock-based
Rebalancing would require multiple lock acquisitions and extra effort to avoid deadlocks

Concurrent BSTs

Name	Balanced	Internal	On-time deletion
Ellen et al. [PODC'10]	✗	✗	✓
Howley et al. [SPAA'12]	✗	✓	✗
Natarajan et al. [PPoPP'14]	✗	✗	✓
Chatterjee et al. [PODC'14]	✗	✓	✗
Brown et al. [PPoPP'14]	✗	✗	✓

Lock-free

Atomic operations (e.g., CAS) can only modify a single memory word

Bronson et al. [PPoPP'10]	✗	✗	✗
Crain et al. [EuroPar'13]	✗	✗	✗
Drachsler et al. [PPoPP'14]	✗	✓	✓

Lock-based

Rebalancing would require multiple lock acquisitions and extra effort to avoid deadlocks

Howard et al. [CCPE'14]	✓	✓	✗
Arbel et al. [PODC'14]	✗	✓	✗

RCU-based

No on-time deletion

Concurrent BSTs

Name	Balanced	Internal	On-time deletion
Ellen et al. [PODC'10]	✗	✗	✓
Howley et al. [SPAA'12]	✗	✓	✗
Natarajan et al. [PPoPP'14]	✗	✗	✓
Chatterjee et al. [PODC'14]	✗	✓	✗
Brown et al. [PPoPP'14]	✗	✗	✓
Bronson et al. [PPoPP'10]	✗	✗	✗
Crain et al. [EuroPar'13]	✗	✗	✗
Drachsler et al. [PPoPP'14]	✗	✓	✓
Howard et al. [CCPE'14]	✓	✓	✗
Arbel et al. [PODC'14]	✗	✓	✗
Crain et al. [PPoPP'14]	✗	✓	✗
Avni et al. [TRANSACT'14]	✓	✓	✓

Lock-free
Atomic operations (e.g., CAS) can only modify a single memory word

Lock-based
Rebalancing would require multiple lock acquisitions and extra effort to avoid deadlocks

RCU-based
No on-time deletion

TM-based
Avni provides all three characteristics but has other drawbacks

RCU-HTM vs previous works

	Name	Balanced	Internal	On-time deletion	Asynchronized traversals	Multiple updaters
Lock-free	Ellen et al. [PODC’10]	✗	✗	✓	✓	✓
Lock-free	Howley et al. [SPAA’12]	✗	✓	✗	✗	✓
Lock-free	Natarajan et al. [PPoPP’14]	✗	✗	✓	✓	✓
Lock-free	Chatterjee et al. [PODC’14]	✗	✓	✗	✗	✓
Lock-free	Brown et al. [PPoPP’14]	✗	✗	✓	✓	✓
Locks	Bronson et al. [PPoPP’10]	✗	✗	✗	✗	✓
Locks	Crain et al. [EuroPar’13]	✗	✗	✗	✓	✓
Locks	Drachsler et al. [PPoPP’14]	✗	✓	✓	✓	✓
RCU	Howard et al. [CCPE’14]	✓	✓	✗	✓	✗
RCU	Arbel et al. [PODC’14]	✗	✓	✗	✓	✓
TM	Crain et al. [PPoPP’14]	✗	✓	✗	✗	✓
TM	Avni et al. [TRANSACT’14]	✓	✓	✓	✗	✓

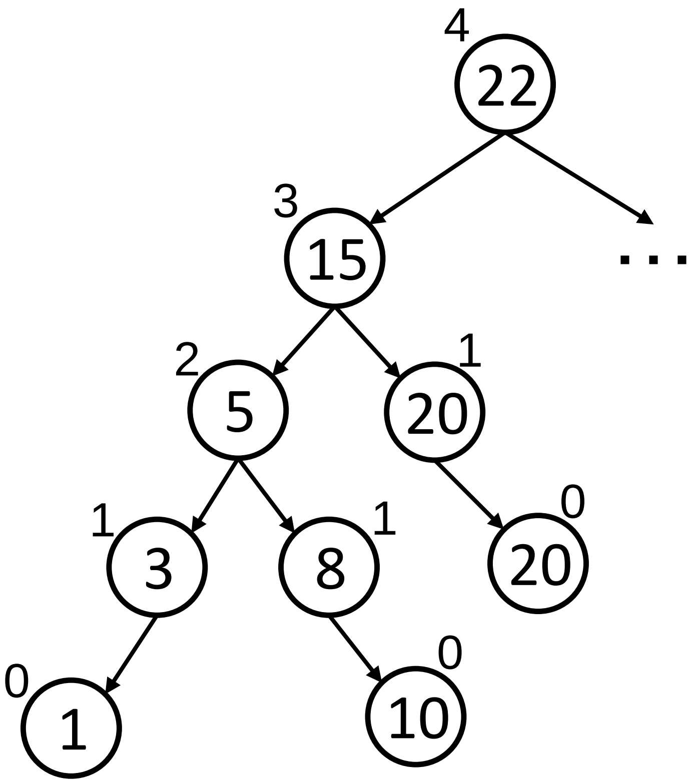
RCU-HTM vs previous works

	Name	Balanced	Internal	On-time deletion	Asynchronized traversals	Multiple updaters
Lock-free	Ellen et al. [PODC’10]	✗	✗	✓	✓	✓
	Howley et al. [SPAA’12]	✗	✓	✗	✗	✓
	Natarajan et al. [PPoPP’14]	✗	✗	✓	✓	✓
	Chatterjee et al. [PODC’14]	✗	✓	✗	✗	✓
	Brown et al. [PPoPP’14]	✗	✗	✓	✓	✓
Locks	Bronson et al. [PPoPP’10]	✗	✗	✗	✗	✓
	Crain et al. [EuroPar’13]	✗	✗	✗	✓	✓
	Drachsler et al. [PPoPP’14]	✗	✓	✓	✓	✓
RCU	Howard et al. [CCPE’14]	✓	✓	✗	✓	✗
	Arbel et al. [PODC’14]	✗	✓	✗	✓	✓
TM	Crain et al. [PPoPP’14]	✗	✓	✗	✗	✓
	Avni et al. [TRANSACT’14]	✓	✓	✓	✗	✓
	RCU-HTM	✓	✓	✓	✓	✓

RCU-HTM: Internal Deletion

RCU-HTM replaces the whole path from the internal node to the successor

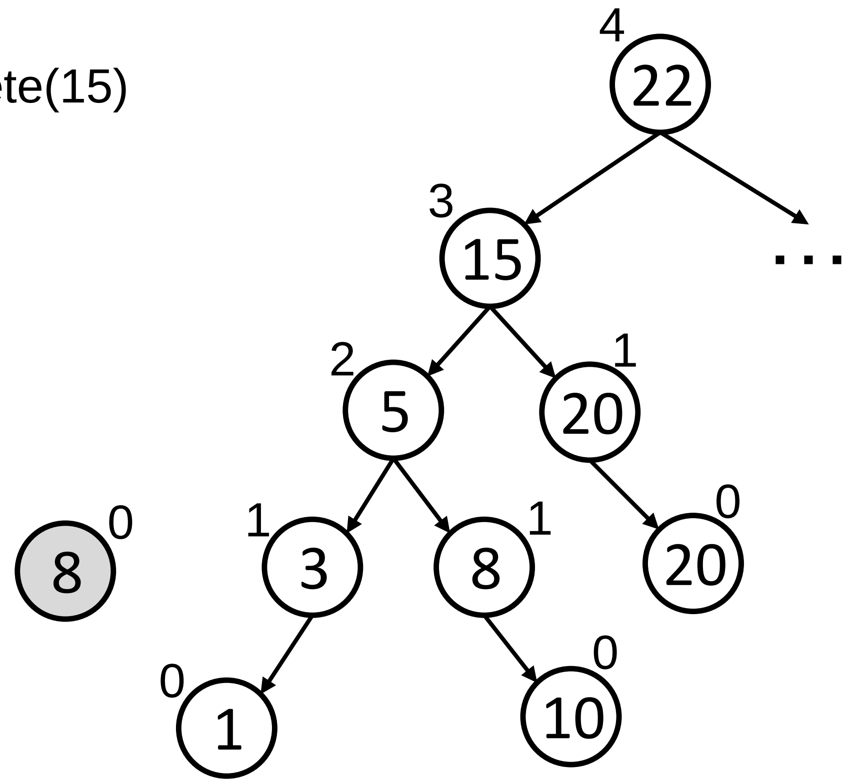
delete(15)



RCU-HTM: Internal Deletion

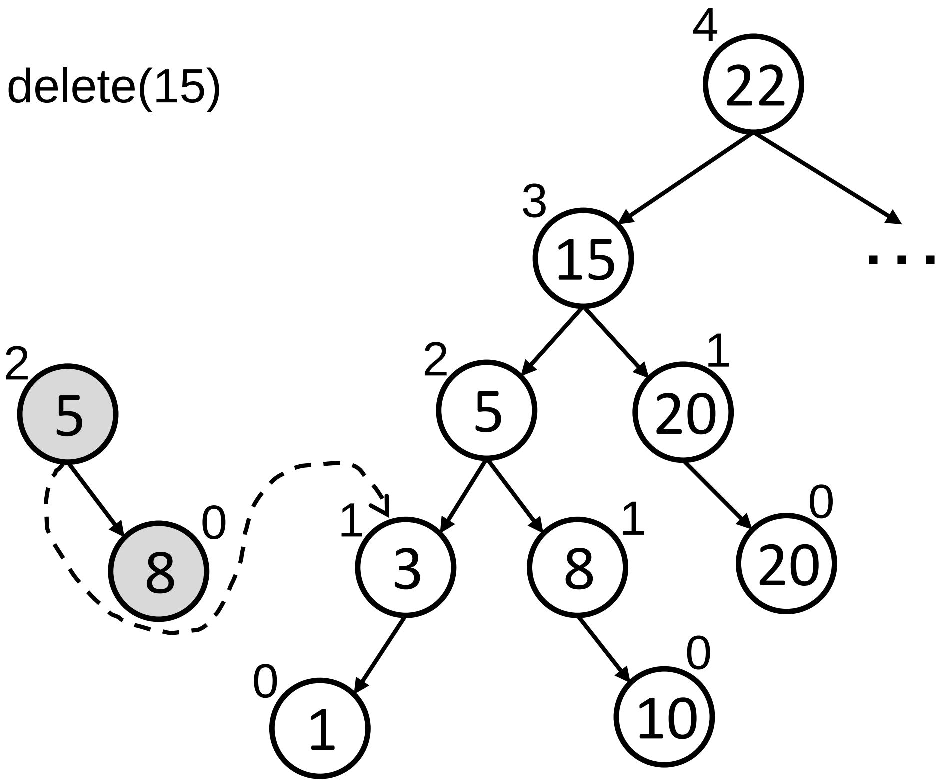
RCU-HTM replaces the whole path from the internal node to the successor

delete(15)



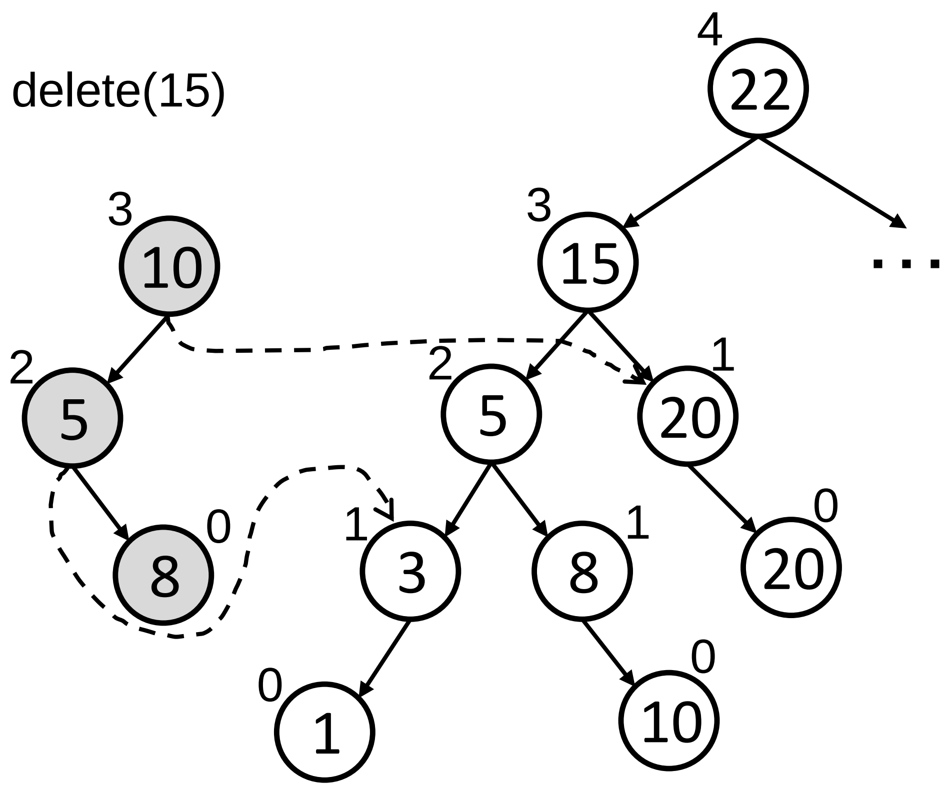
RCU-HTM: Internal Deletion

RCU-HTM replaces the whole path from the internal node to the successor



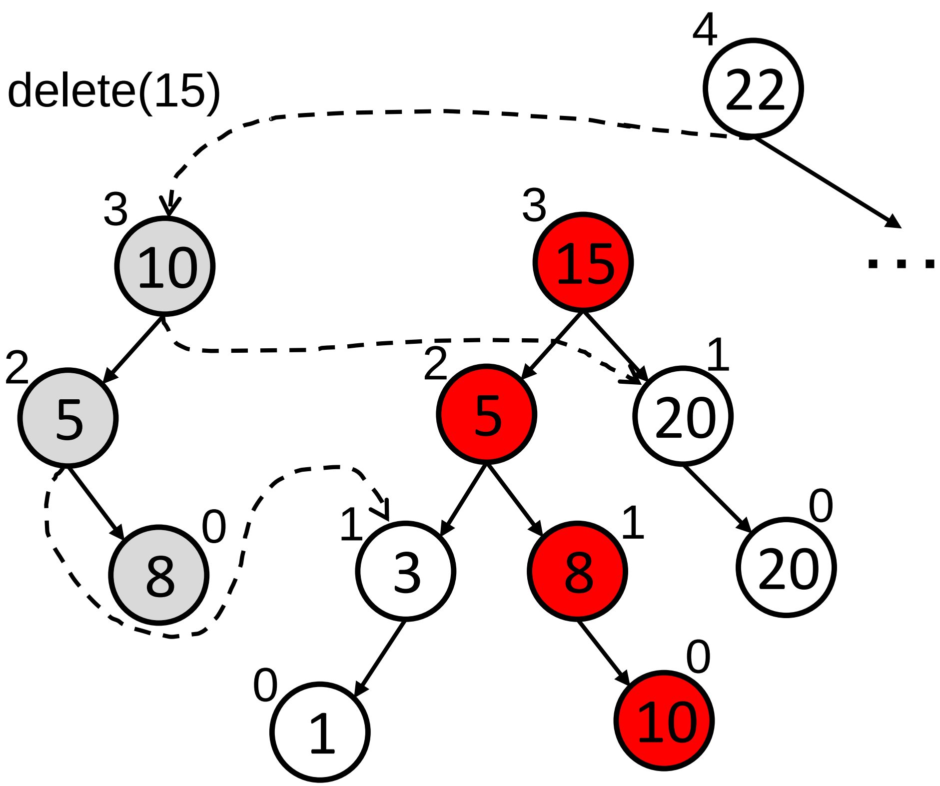
RCU-HTM: Internal Deletion

RCU-HTM replaces the whole path from the internal node to the successor



RCU-HTM: Internal Deletion

RCU-HTM replaces the whole path from the internal node to the successor



CSLab National Technical University of Athens

RCU-HTM: Internal Deletion

RCU-HTM replaces the whole path from the internal node to the successor