Efficient Concurrent Aggregate Queries

Concurrent Computing

Gal Sela, EPFL

Multi-Element Queries

- Snapshots

Multi-Element Queries

- Snapshots, range queries
- Aggregate queries

size 5 i-th item *4-th* ⇨

Summarize a range of items with consecutive keys into a succinct value

Aggregate Queries Applications

size 5

collections and maps
in Java

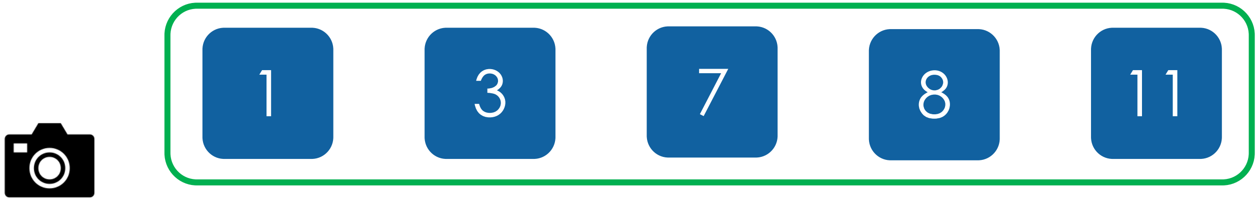
i-th item *4-th* ⇨ 8

access sortedcontainer[i]
in Python
implemented using order-statistic tree

similar augmented trees
in some text editors
to access the cursor's location

Augmented trees – used in
interval trees, link/cut trees, tango trees etc.

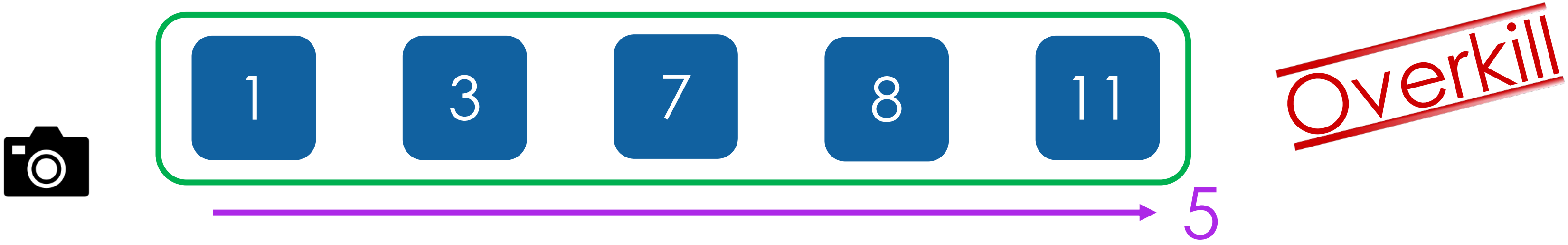
Concurrent Snapshots



Extensive research
in the last 35 years

Concurrent Aggregate Queries

Could be implemented using snapshot



✓ Correct: linearizable

✗ Inefficient: θ(num of elements)

Concurrent Aggregate Queries

1	3	7	8	11

- ✓ Correct
- ✓ Efficient

difficult! Emerging research

Concurrent Aggregate Queries

Correct and efficient – difficult!

Emerging research

Concurrent size [OOPSLA'22]

Wait-free trees with asymptotically-efficient range queries [IPDPS'24]

Concurrent aggregate queries [DISC'24]

Lock-free augmented trees [DISC'24]

Concurrent Aggregate Queries

Correct and efficient – difficult!

Emerging research

For sets and dictionaries

Concurrent Size

Gal Sela and Erez Petrank

OOPSLA'22

Concurrent Size

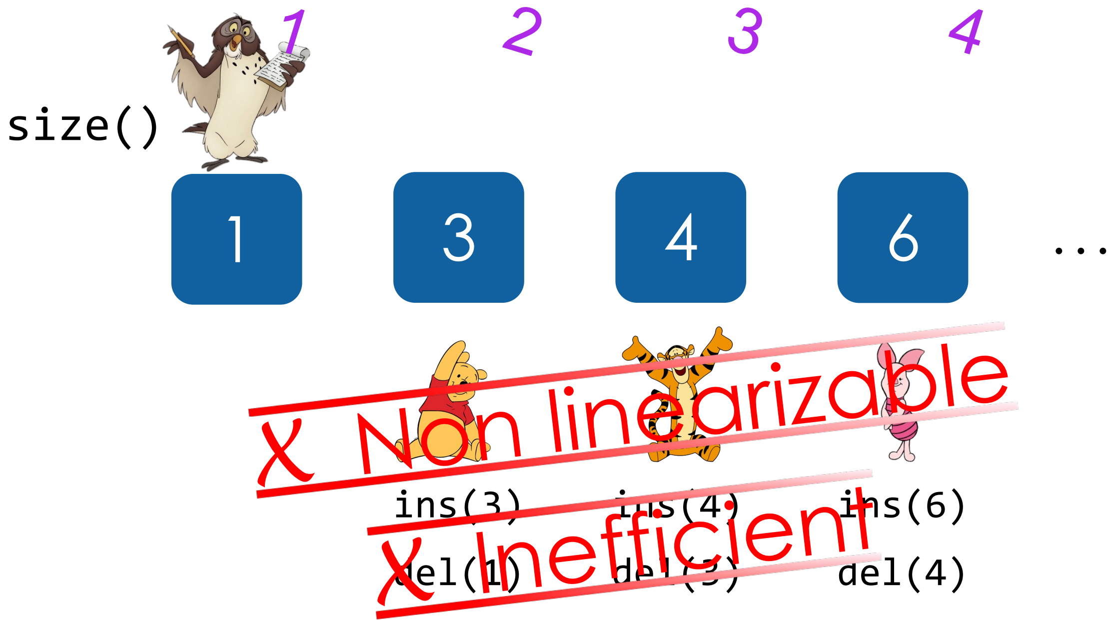
Difficult due to concurrent updates

1 3 4 6 …

`ins(3)`	`ins(4)`	`ins(6)`
`del(1)`	`del(3)`	`del(4)`

Concurrent Size



ConcurrentLinkedQueue

ConcurrentLinkedDeque

Concurrent Size

For efficiency: *Metadata*

for computing size

`ConcurrentSkipListMap` `ConcurrentHashMap`

Concurrent Size

For efficiency: Metadata

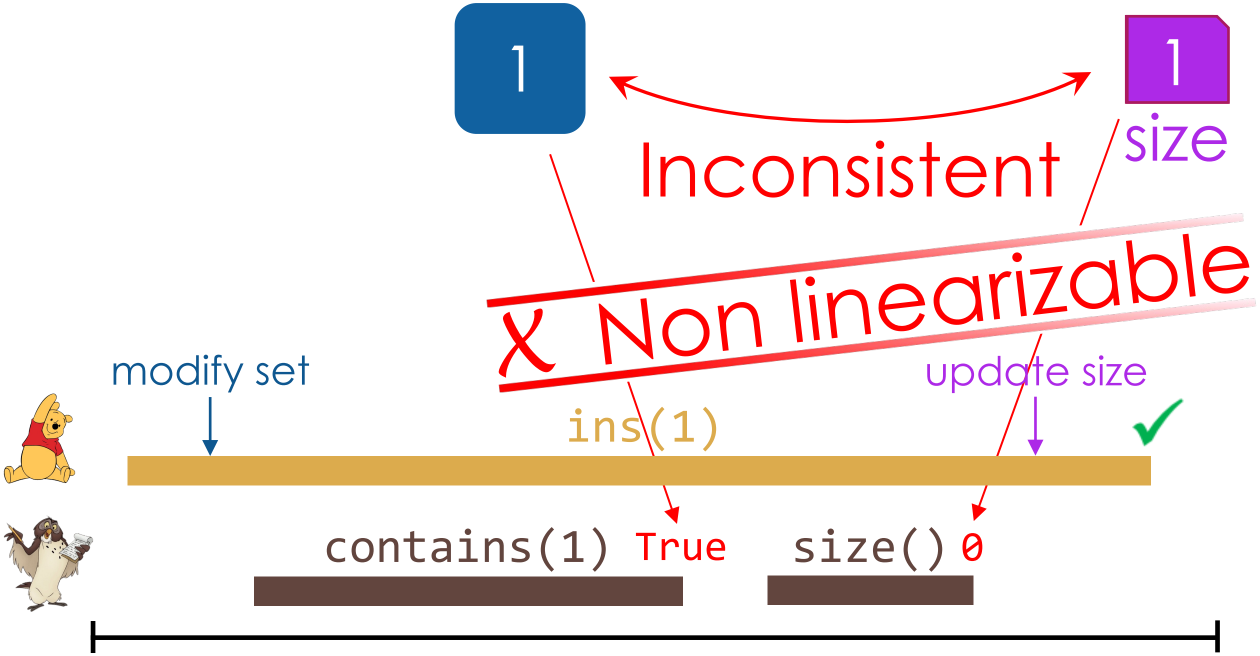
for computing size

✓ Efficient

What about Correctness?

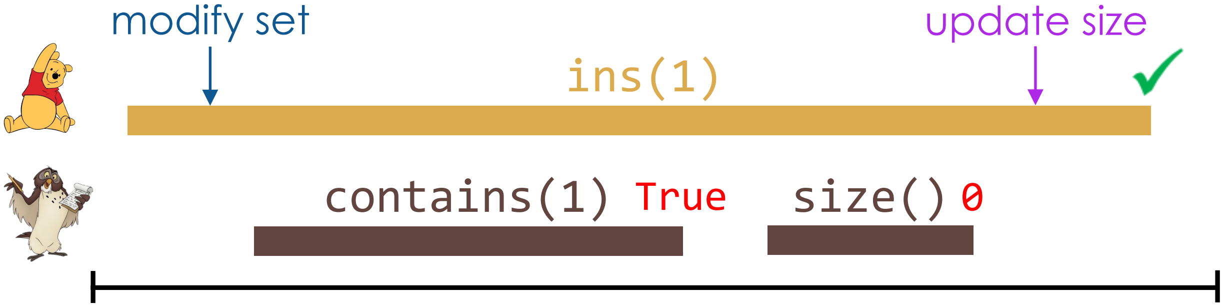
Size as Metadata

Maintained by `ins` & `del`: modify set then update size



Linearizability

Each operation appears to happen at once during its interval.



Negative Size

Maintained by `ins` & `del`: modify set then update size

1 3 4 6 …

-2
size

`size()`

del(1)	del(3)	del(4)
ins(3)	ins(4)	ins(6)

✗ Incorrect

Negative Size

ConcurrentHashMap

```java
public int size() {
    long n = sumCount();
    return ((n < 0L) ? 0 :
            (n > (long)Integer.MAX_VALUE) ? Integer.MAX_VALUE :
            (int)n);
}
```

```java
public boolean isEmpty() {
    return sumCount() <= 0L; // ignore transient negative values
}
```

ConcurrentSkipListMap

```java
final long getAdderCount() {
    LongAdder a; long c;
    do {} while ((a = adder) == null &&
                 !ADDER.compareAndSet(this, null, a = new LongAdder()));
    return ((c = a.sum()) <= 0L) ? 0L : c; // ignore transient negatives
}
```

```java
public int size() {
    long c;
    return ((baseHead() == null) ? 0 :
            ((c = getAdderCount()) >= Integer.MAX_VALUE) ?
            Integer.MAX_VALUE : (int) c);
}
```

Size as Metadata

Maintained
by `ins` & `del`: modify set then update size

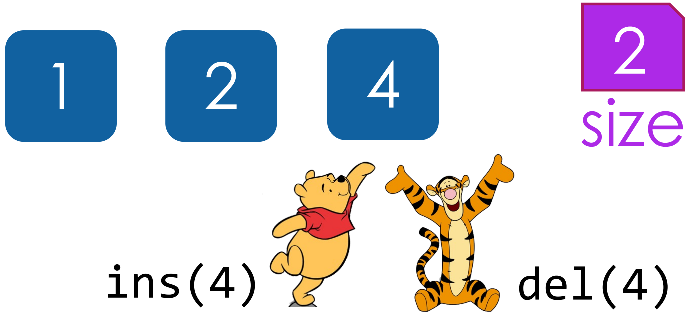
The problem: not atomic

incorrect size

Concurrent Size

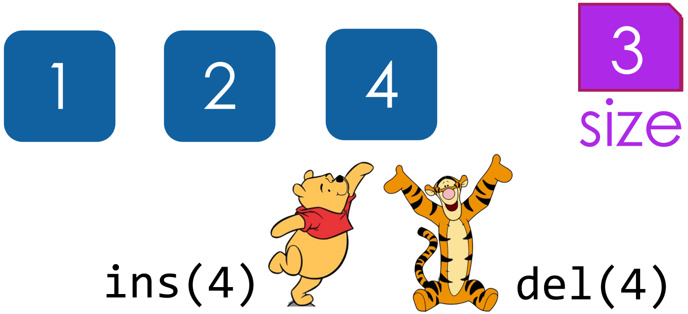
- ✓ For efficiency: metadata
- ✓ For correctness: atomically update data structure & metadata:

`ins` / `del` linearized at metadata update



Concurrent Size

- ✓ For efficiency: metadata
- ✓ For correctness: atomically update data structure & metadata
- ✓ For progress: HELP!



Concurrent Size

- ✓ For efficiency: metadata
- ✓ For correctness: atomically update data structure & metadata
- ✓ For progress: 

How update metadata exactly once?

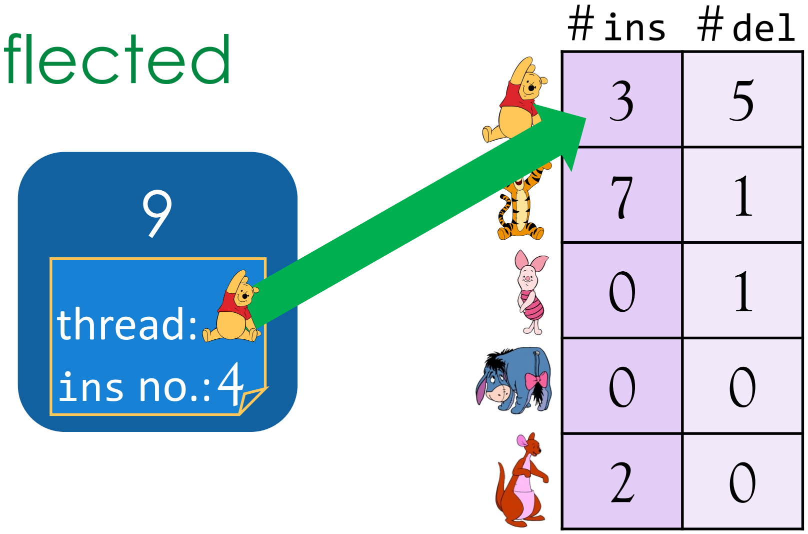
Using suitable metadata ➡ Threads can determine whether reflects certain op, otherwise update

Metadata: Monotonically Increasing Counters

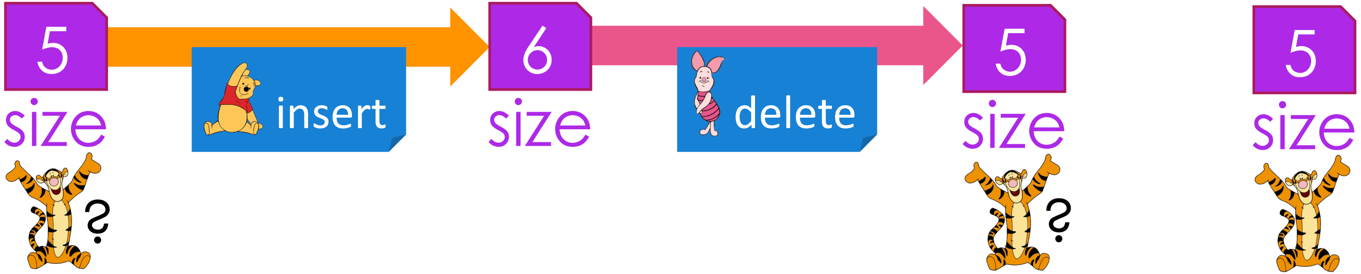
	#ins	#del
	3	5
	7	1
	0	1
	0	0
	2	0

Metadata: Monotonically Increasing Counters

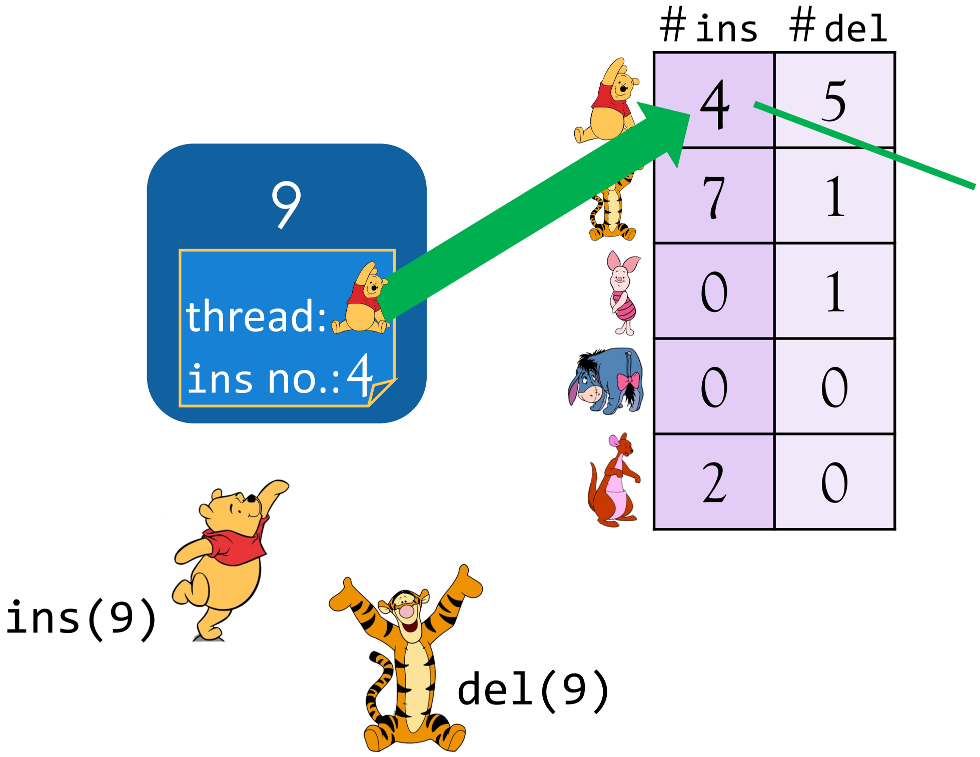
Easy to detect if certain op is reflected



Unlike size field, or even `#ins-#del` per thread



Metadata: Monotonically Increasing Counters



CAS(3,4)

(CompareAndSwap)

No need to repeat

Compute Size from Metadata

	#ins	#del
	3	5
	7	1
	0	1
	0	0
	2	0

$$size = \Sigma(\#ins - \#del)$$

Compute Size from Metadata
requires snapshot

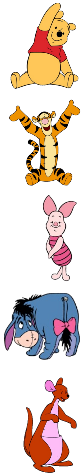
ins(9)

del(9)

	#ins	#del
	1	0
	0	0
	0	0
	0	0
	0	1

$size = \Sigma(\#ins - \#del) = -1$

Compute Size from Metadata
requires snapshot



#ins	#del
1	0
0	0
0	0
0	0
0	1

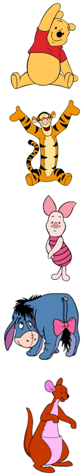
Traversal ➡ inconsistent size

Need small linearizable snapshot

not of all items ☺

size = Σ(#ins-#del)

Compute Size from Metadata
requires snapshot



	#ins	#del
	1	0
	0	0
	0	0
	0	0
	0	1

Traversal ➡ inconsistent size

Need small linearizable snapshot

Jayanti [2005], Petrank & Timnat [2013]

size = Σ(#ins-#del)

Efficient Size

Our size mechanism

SizeHashTable   SizeBST   SizeSkipList

Snapshot-based size

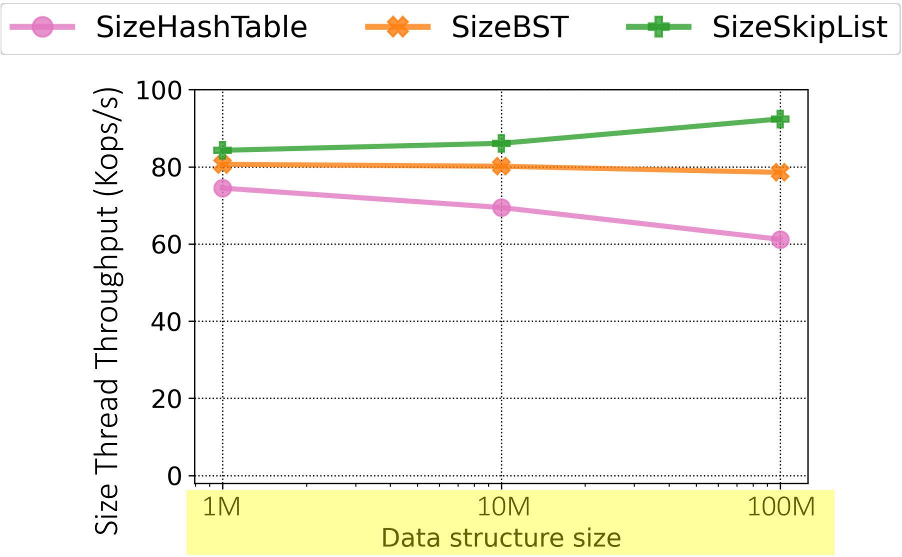
VcasBST-64   SnapshotSkipList

Adaptation of BST by Wei et al. [2021]

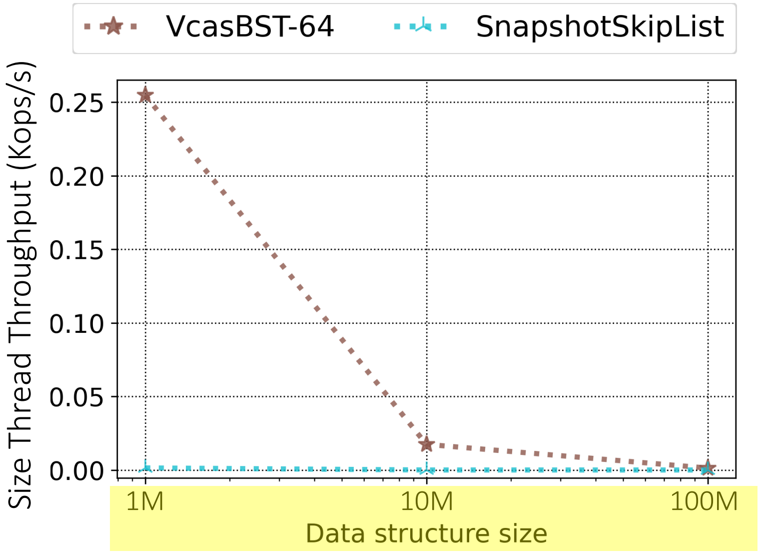
Skiplist by Petrank & Timnat [2013]

Efficient Size

Our size mechanism



Snapshot-based size

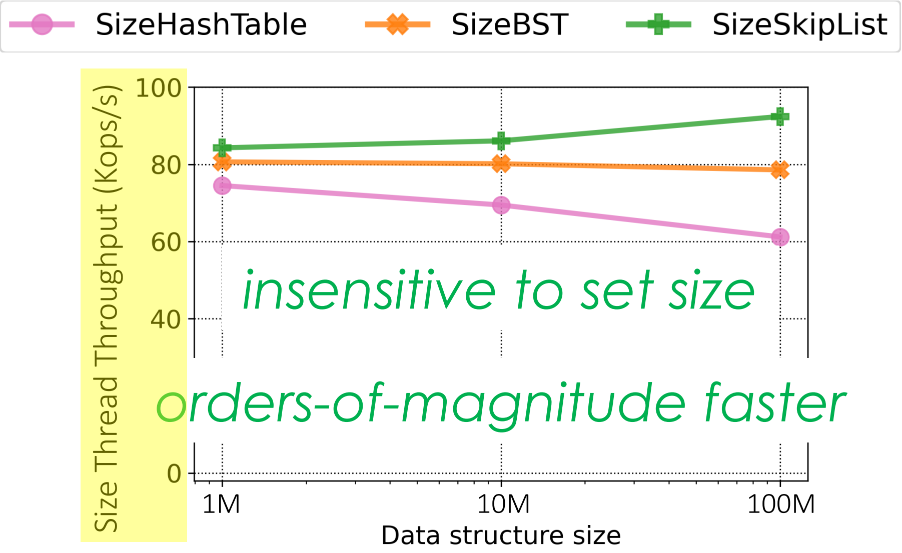


31 threads: 3% `ins`, 2% `del`, 95% `contains`

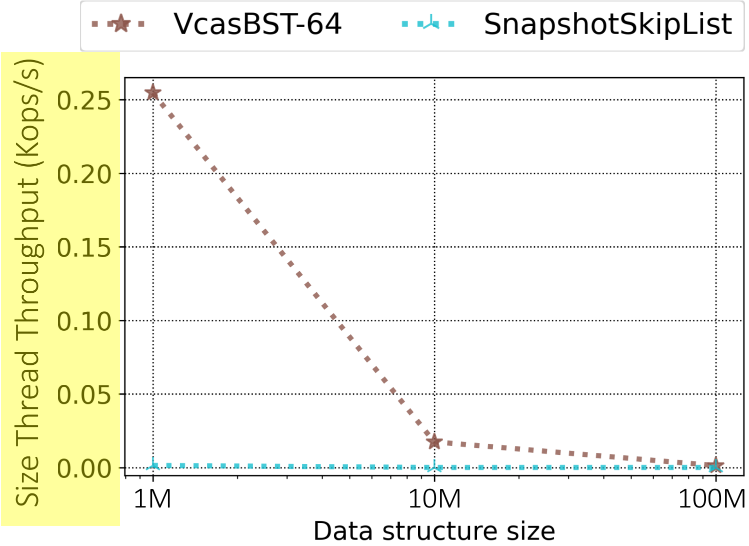
1 thread: `size` ➡ measure its throughput

Efficient Size

Our size mechanism



Snapshot-based size



31 threads: 3% `ins`, 2% `del`, 95% `contains`

1 thread: `size` ➡ measure its throughput

Concurrent Size / Main Ideas

- ✓ Efficient (using metadata)
- ✓ Correct (using atomic update of set & metadata + metadata snapshot)
- ✓ Preserves progress (using HELP!)
- ✗ Incurs overhead on `ins`, `del`, `contains`

Similar ideas in tree aggregate queries

Trees

Set or dictionary

Keys

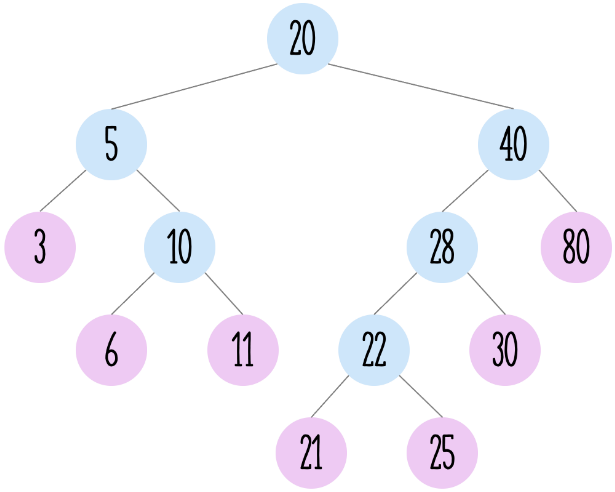
3 6 11 21 25 30 80

Trees

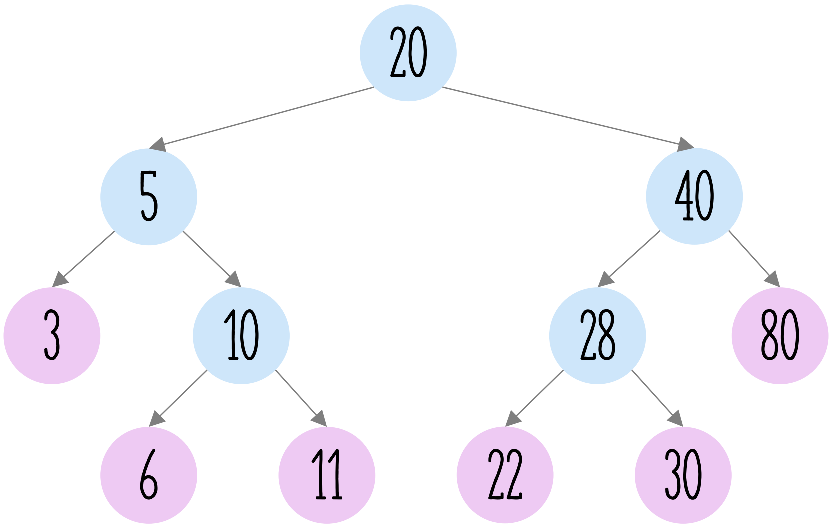
Set or dictionary represented by binary search tree

Keys

3 6 11 21 25 30 80



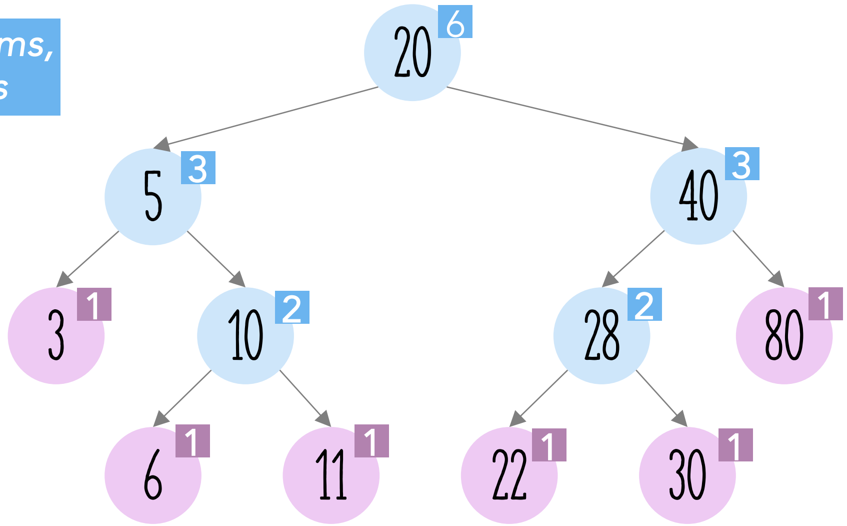
Tree Aggregate Queries



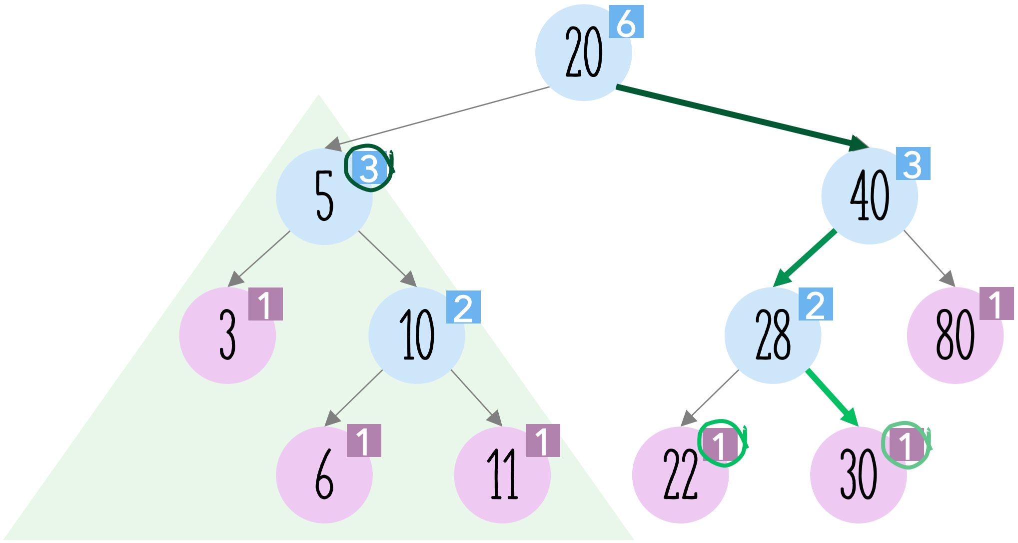
$\texttt{index}(30) = 5$

Efficient Tree Aggregate Queries

function of subtree's items, e.g., number of items



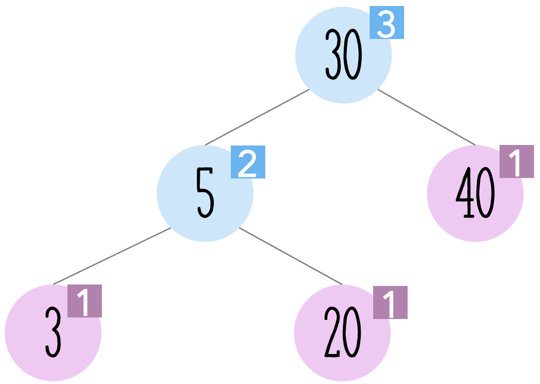
Efficient Tree Aggregate Queries



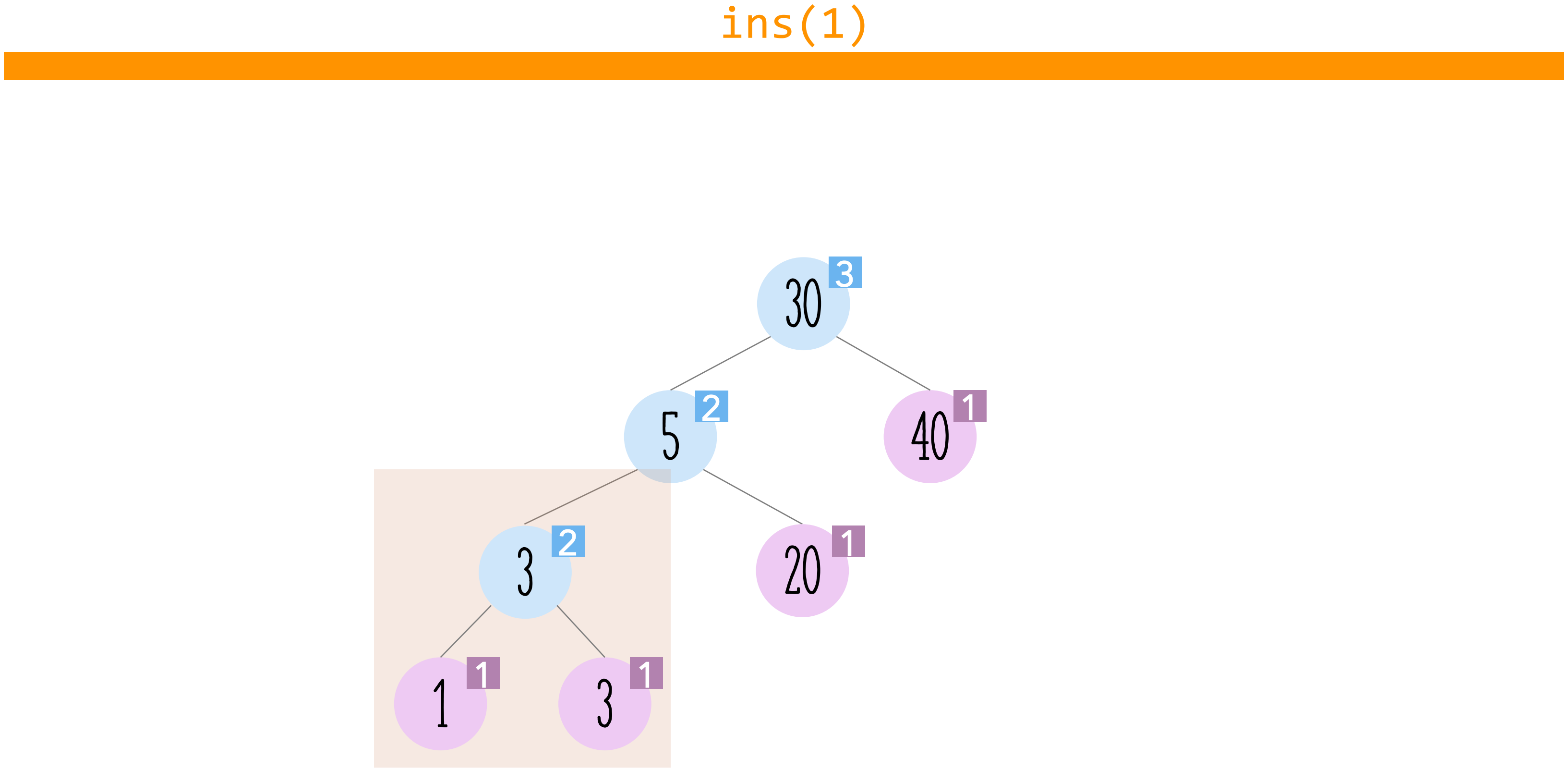
index(30) = 3 + 1 + 1 = 5

Concurrent Tree Aggregate Queries

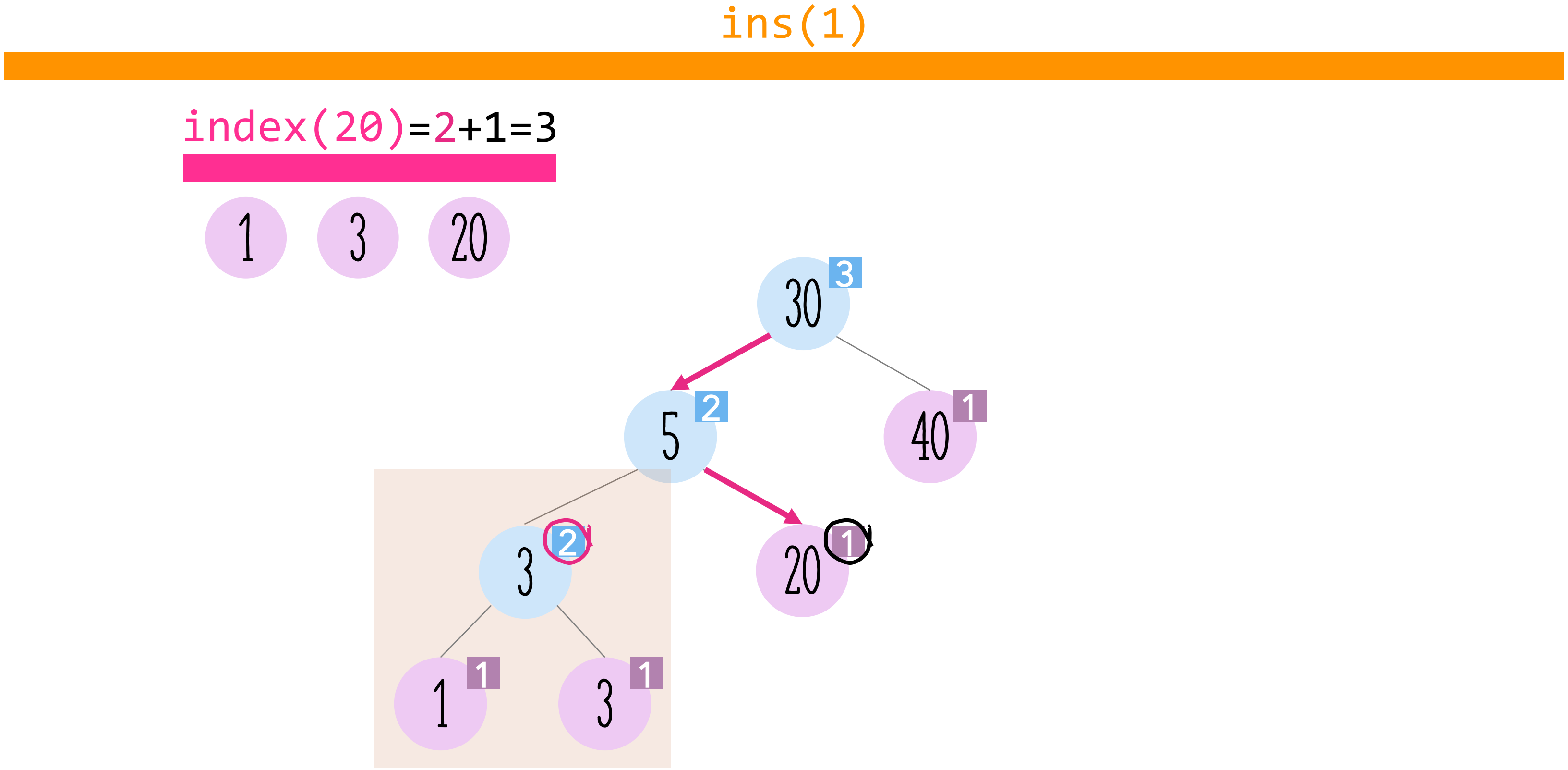
`ins(1)`



Concurrent Tree Aggregate Queries



Concurrent Tree Aggregate Queries



Concurrent Tree Aggregate Queries

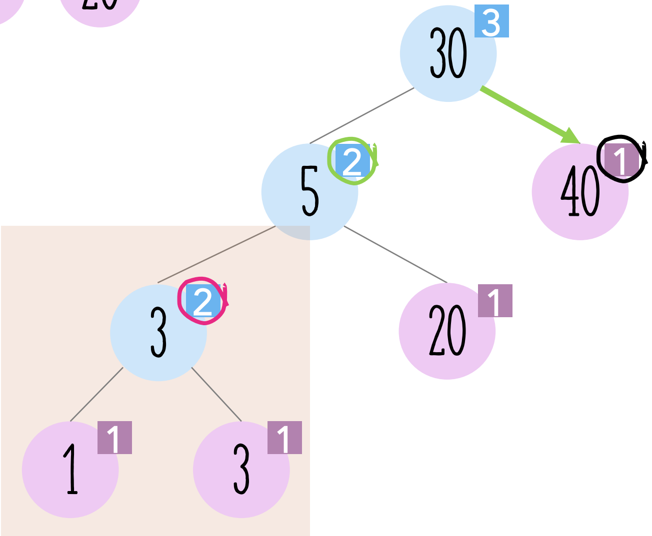
ins(1)

index(20)=2+1=3

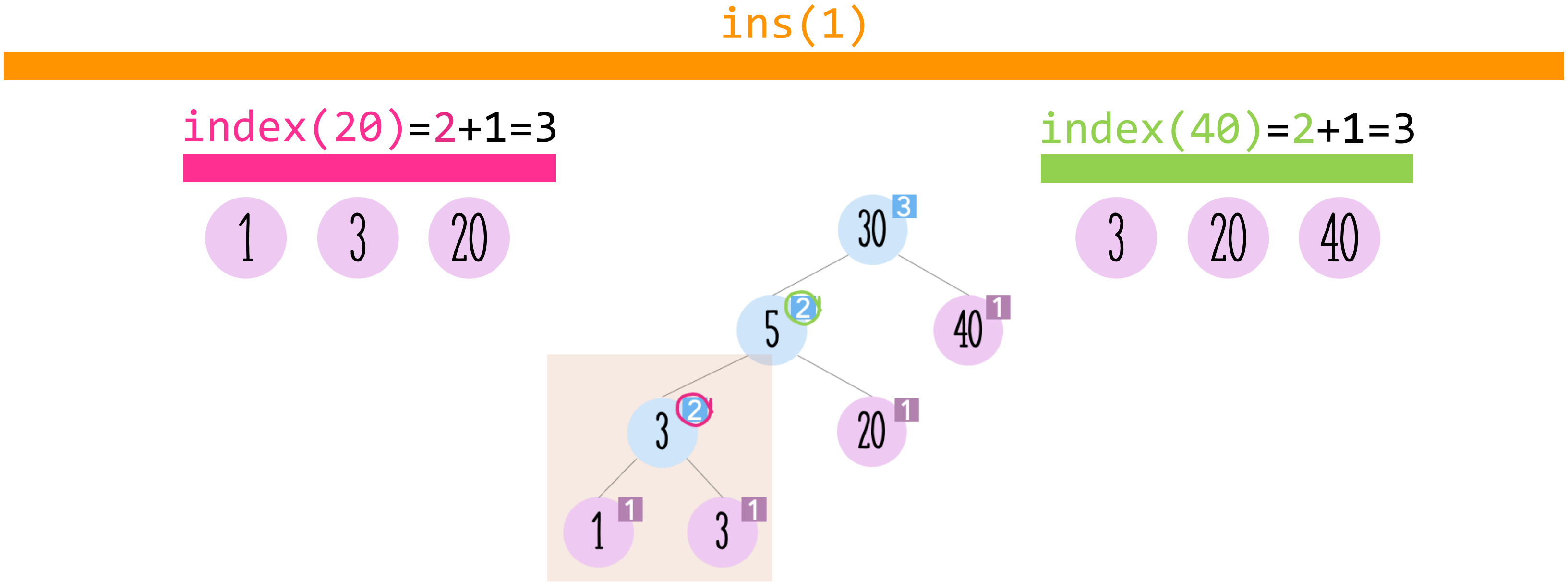
1 3 20

index(40)=2+1=3

3 20 40



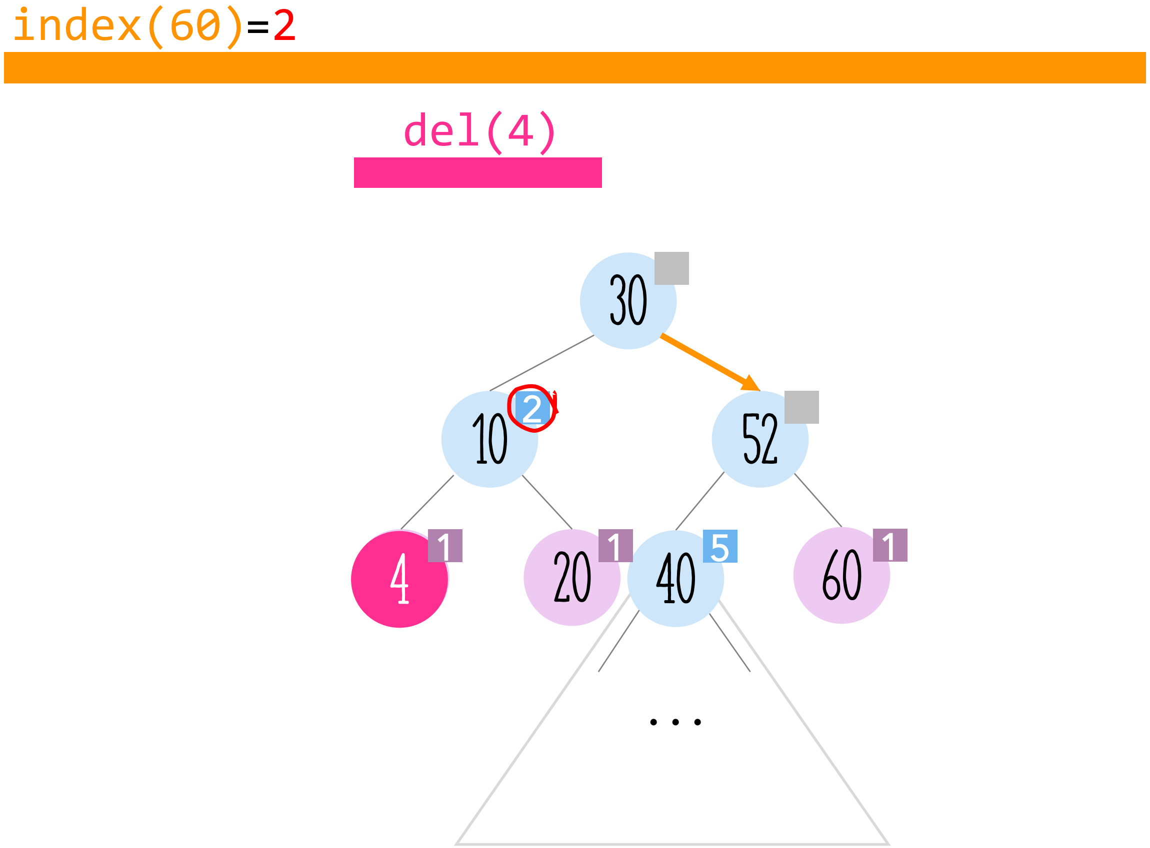
Concurrent Tree Aggregate Queries



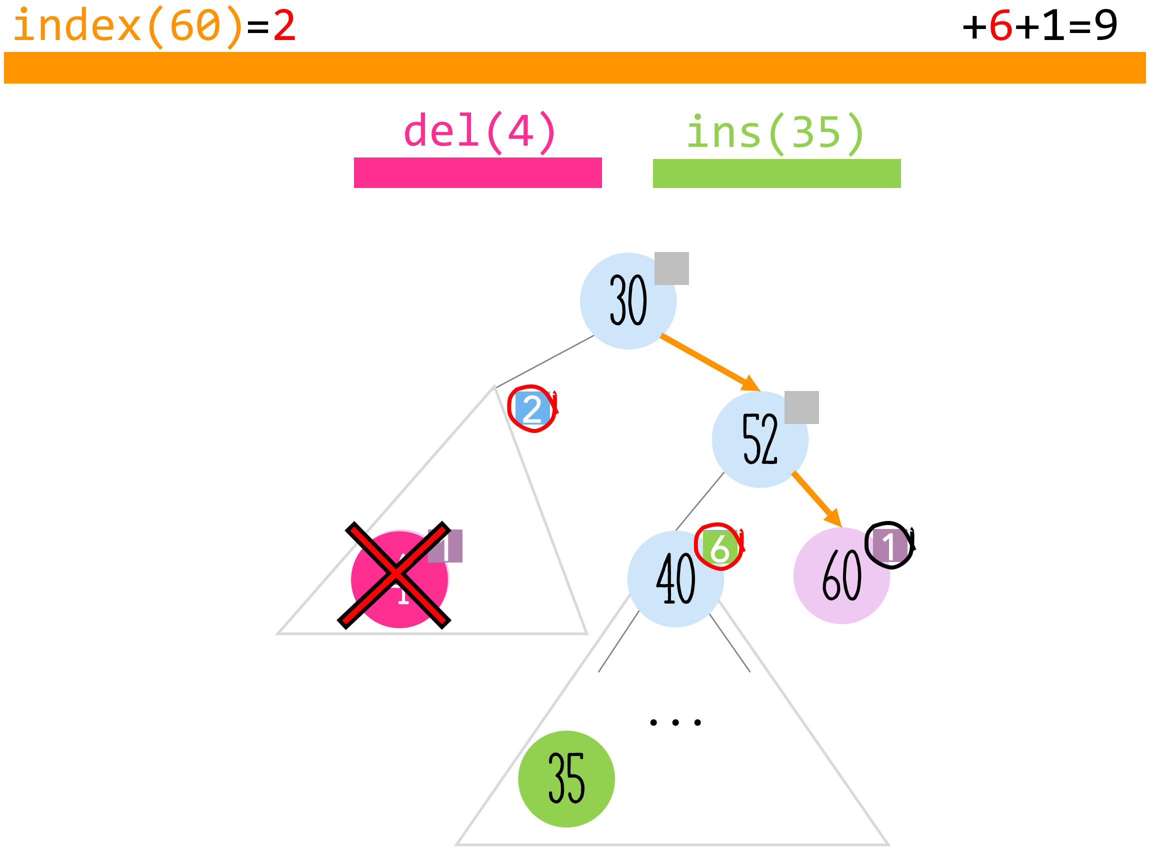
Problem: Some queries observe `ins(1)`, others don't

Need atomic update of tree and all metadata

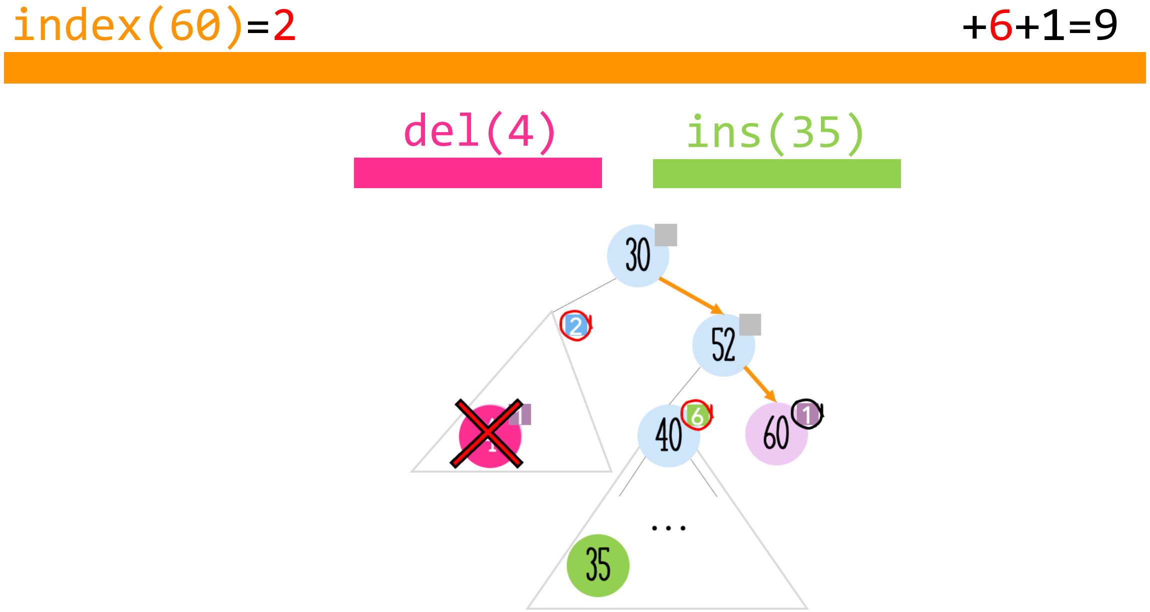
Concurrent Tree Aggregate Queries



Concurrent Tree Aggregate Queries



Concurrent Tree Aggregate Queries



Problem: The query sees new updates after missing old updates

Queries should obtain a snapshot view of the traversed path

Concurrent Tree Aggregate Queries

Problem: Updates modify multiple locations

➡ Some queries see an update, others don't

Need atomic update of tree and all metadata

Problem: Queries read multiple values

➡ Queries might see new updates after missing old updates

Queries should obtain a snapshot view of the traversed path

Wait-Free Trees with Asymptotically-Efficient Range Queries

Ilya Kokorin, Victor Yudov, Vitaly Aksenov, and Dan Alistarh,

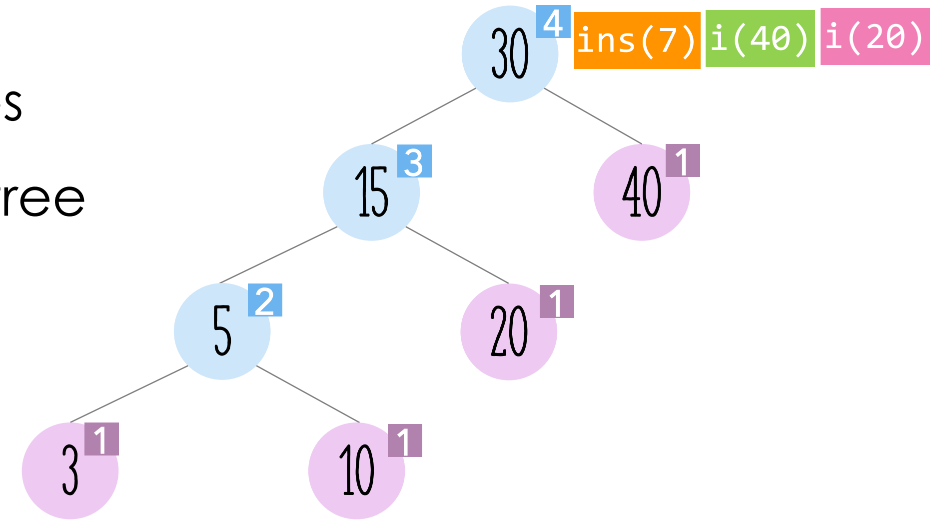
IPDPS'24

Concurrent Tree Aggregate Queries

Kokorin, Yudov, Aksenov, and Alistarh

Op queues in all nodes

with ops to apply to subtree

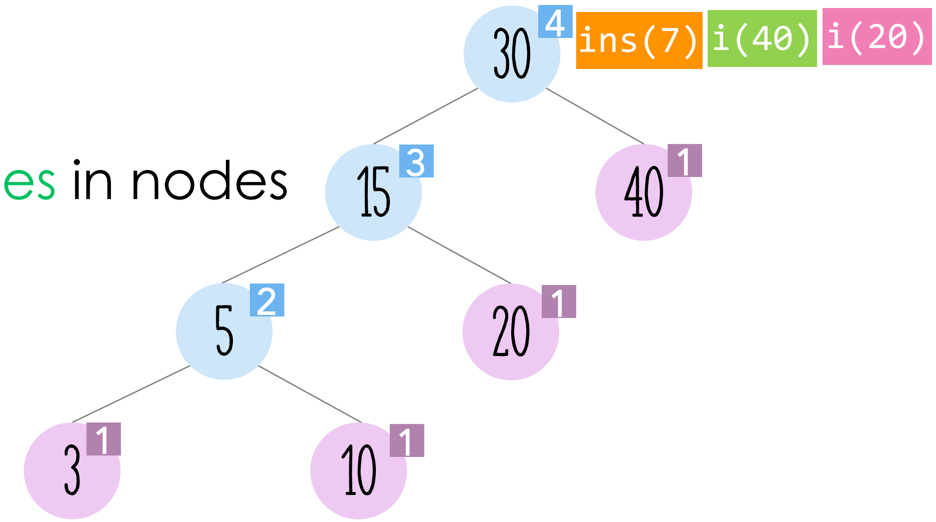


Concurrent Tree Aggregate Queries

Kokorin, Yudov, Aksenov, and Alistarh

Serialize on root

Advance through op queues in nodes while helping others

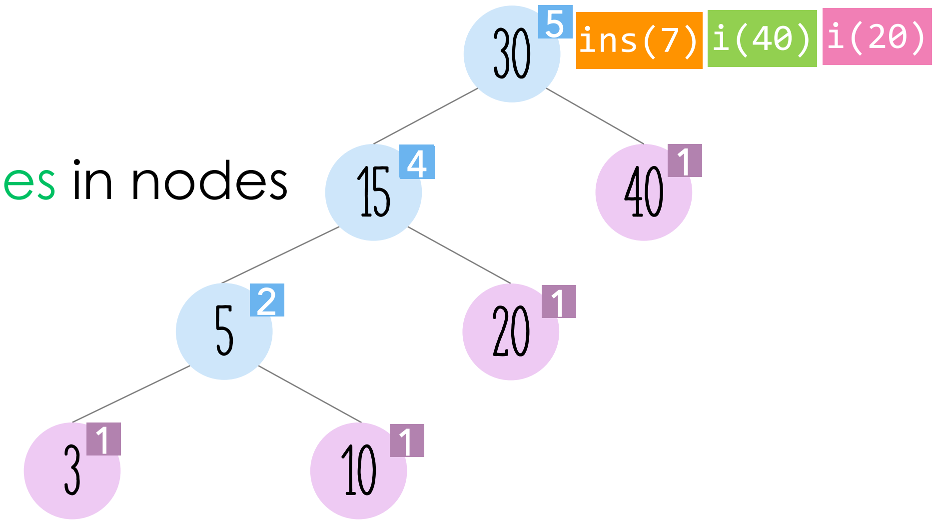


Concurrent Tree Aggregate Queries

Kokorin, Yudov, Aksenov, and Alistarh

Serialize on root

Advance through op queues in nodes while helping others



Concurrent Tree Aggregate Queries

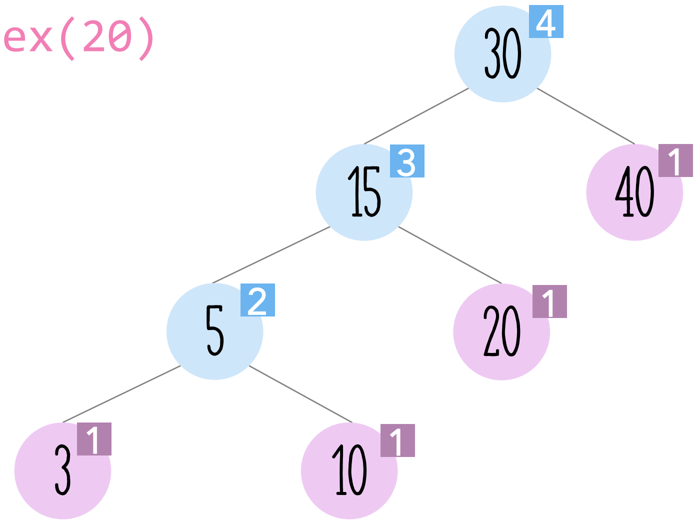
Kokorin, Yudov, Aksenov, and Alistarh

`ins(7)` `index(40)` `index(20)`

Goal:

`index(40)`, `index(20)`

consider `ins(7)`

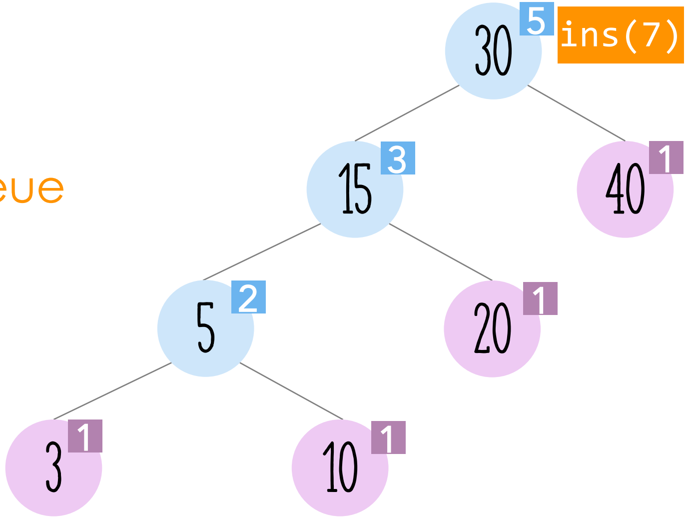


Concurrent Tree Aggregate Queries

Kokorin, Yudov, Aksenov, and Alistarh

`ins(7)`
ts=11

Serialize op: Add to root queue and obtain timestamp



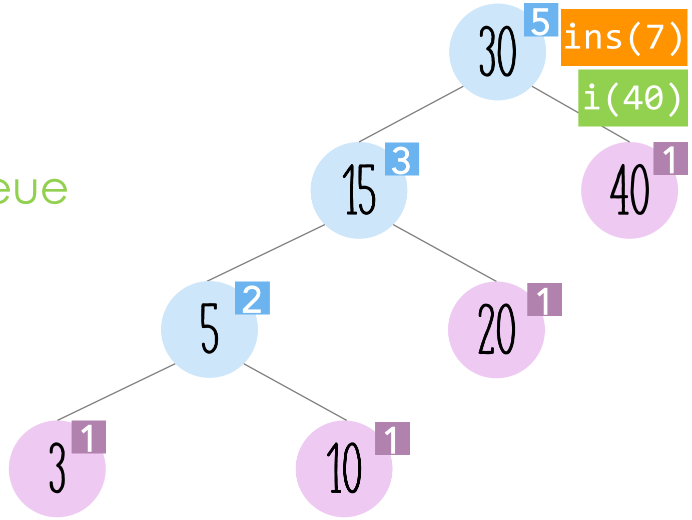
Concurrent Tree Aggregate Queries

Kokorin, Yudov, Aksenov, and Alistarh

`ins(7)` ts=11

`index(40)` ts=12

Serialize op: Add to root queue and obtain timestamp



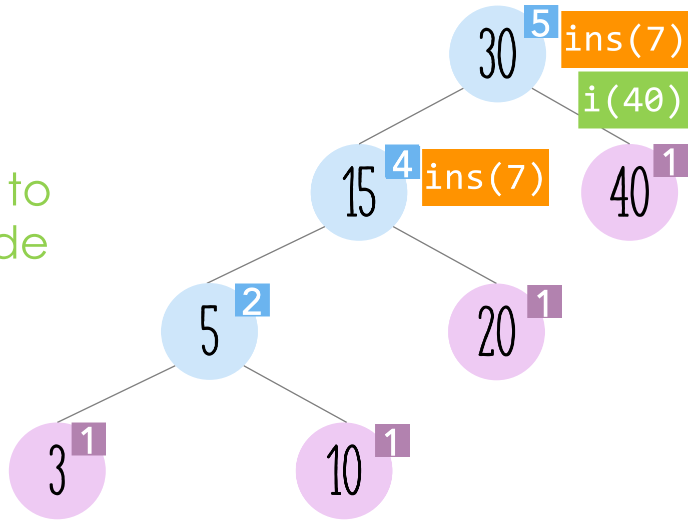
Concurrent Tree Aggregate Queries

Kokorin, Yudov, Aksenov, and Alistarh

`ins(7)` ts=11

`index(40)` ts=12

Advance all operations up to ts=12 in each traversed node



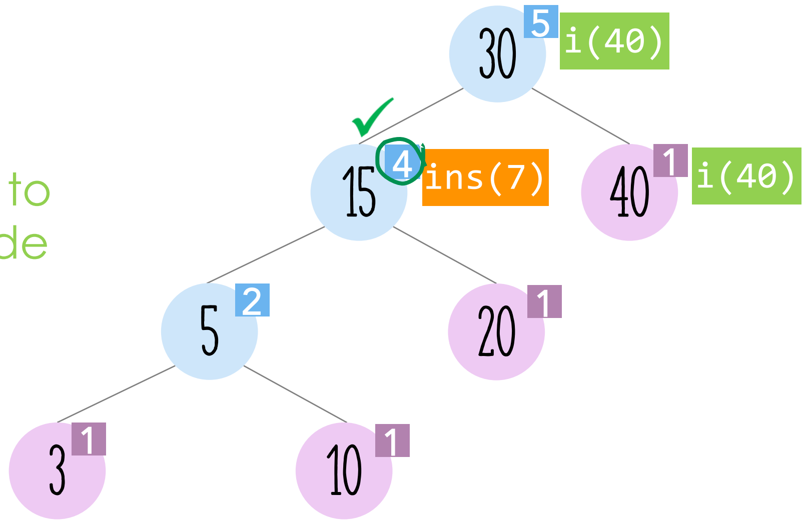
Concurrent Tree Aggregate Queries

Kokorin, Yudov, Aksenov, and Alistarh

`ins(7)` ts=11

`index(40)` ts=12

Advance all operations up to ts=12 in each traversed node



Concurrent Tree Aggregate Queries

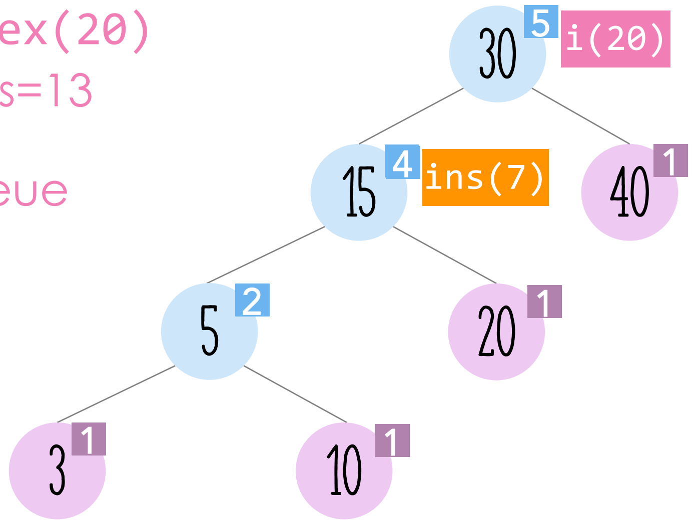
Kokorin, Yudov, Aksenov, and Alistarh

ins(7) ts=11

index(40) ts=12

index(20) ts=13

Serialize op: Add to root queue
and obtain timestamp



Concurrent Tree Aggregate Queries

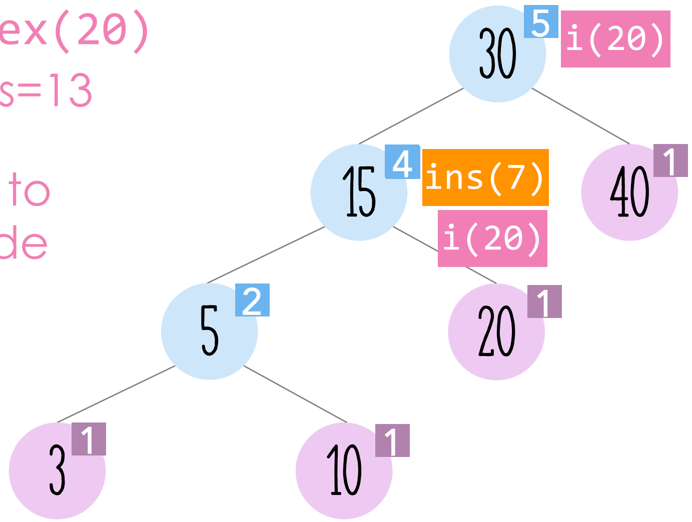
Kokorin, Yudov, Aksenov, and Alistarh

ins(7) ts=11

index(40) ts=12

index(20) ts=13

Advance all operations up to ts=13 in each traversed node

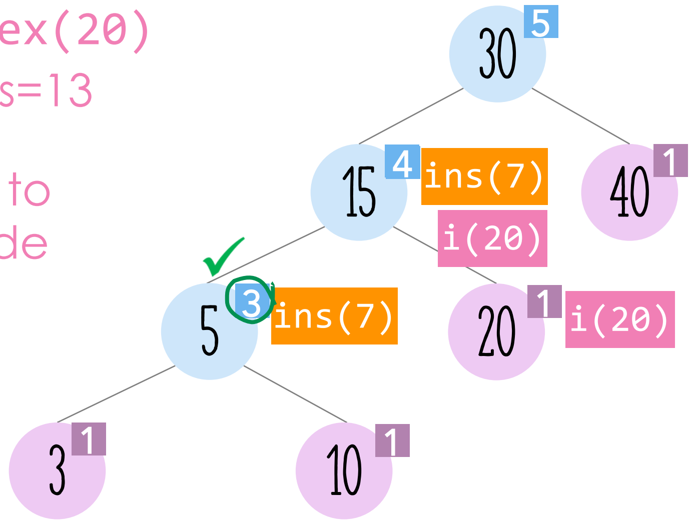


Concurrent Tree Aggregate Queries

Kokorin, Yudov, Aksenov, and Alistarh

ins(7) ts=11 | index(40) ts=12 | index(20) ts=13

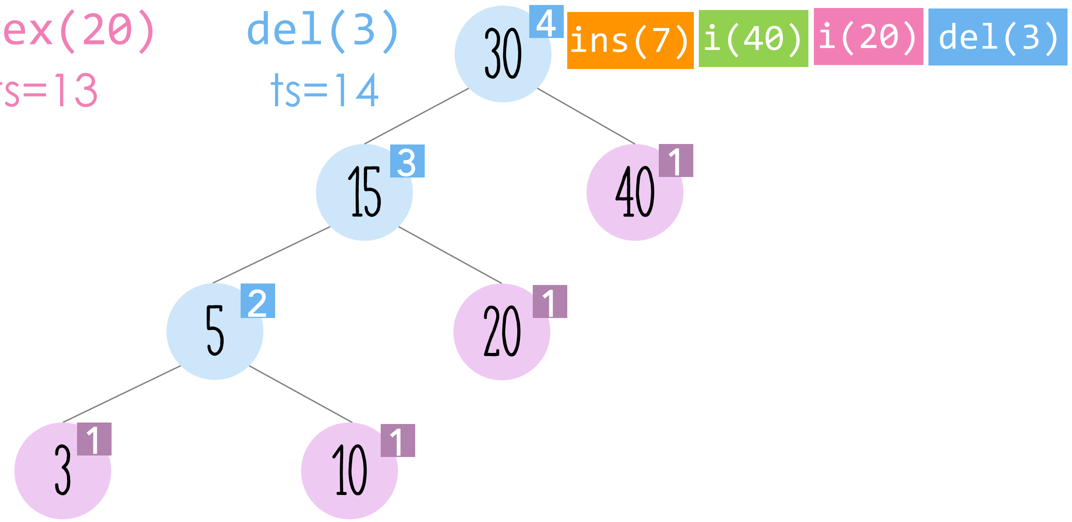
Advance all operations up to ts=13 in each traversed node



Concurrent Tree Aggregate Queries

Kokorin, Yudov, Aksenov, and Alistarh

`ins(7)` ts=11 `index(40)` ts=12 `index(20)` ts=13 `del(3)` ts=14



Goal:

`index(40)`, `index(20)`
consider `ins(7)`

Goal:

`index(40)`, `index(20)`
don't consider `del(3)`

Concurrent Tree Aggregate Queries

Kokorin, Yudov, Aksenov, and Alistarh

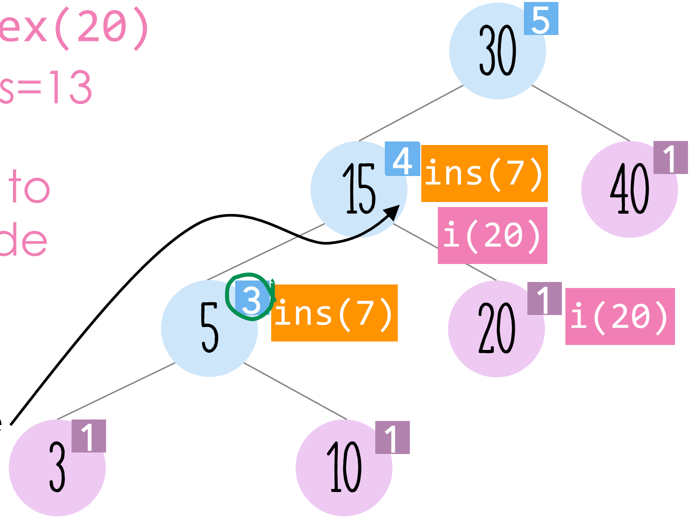
ins(7)
ts=11

index(20)
ts=13

Advance all operations up to ts=13 in each traversed node

Thanks to timestamps:

- index(20) keeps executing
- ins(7) doesn't enqueue here



Concurrent Tree Aggregate Queries

Kokorin, Yudov, Aksenov, and Alistarh

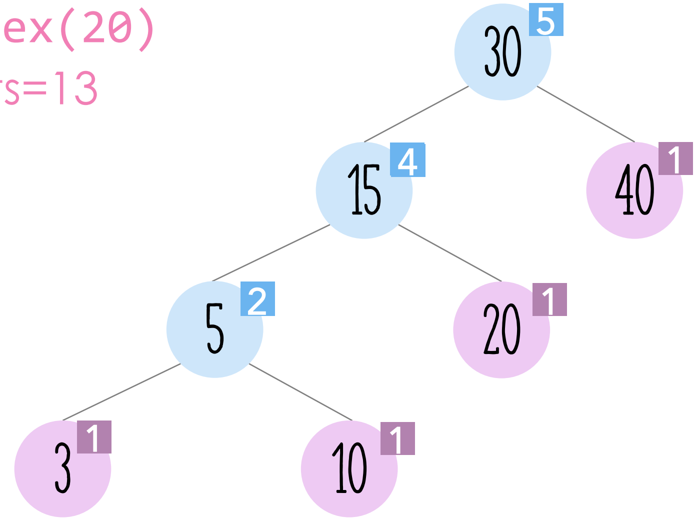
Main Ideas

- Op queues in all nodes
- Serialize on root
- Advance through op queues in nodes
  while helping others
  to form virtual snapshot view

Concurrent Tree Aggregate Queries

Kokorin, Yudov, Aksenov, and Alistarh

`ins(7)` ts=11 `index(40)` ts=12 `index(20)` ts=13



No support for failing operations

Concurrent Aggregate Queries

Gal Sela and Erez Petrank

DISC'24

Timestamps

Ongoing update operations

`ins(7)` ts=11

`del(10)` ts=12

`del(4)` ts=13

consider

by announcements

ignore

by multi-versioning

Aggregate queries

`index(20)` ts=**12**

Multi-Versioning

to ignore concurrent updates
with bigger timestamps

`del(4)`
ts=13

`index(20)`
ts=12

Similar to multi-versioning for snapshots

Wei et al. [PPoPP'21]

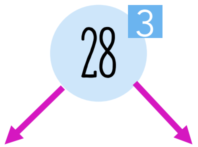
Multi-Versioning

to ignore concurrent updates with bigger timestamps

del(4)
ts=13

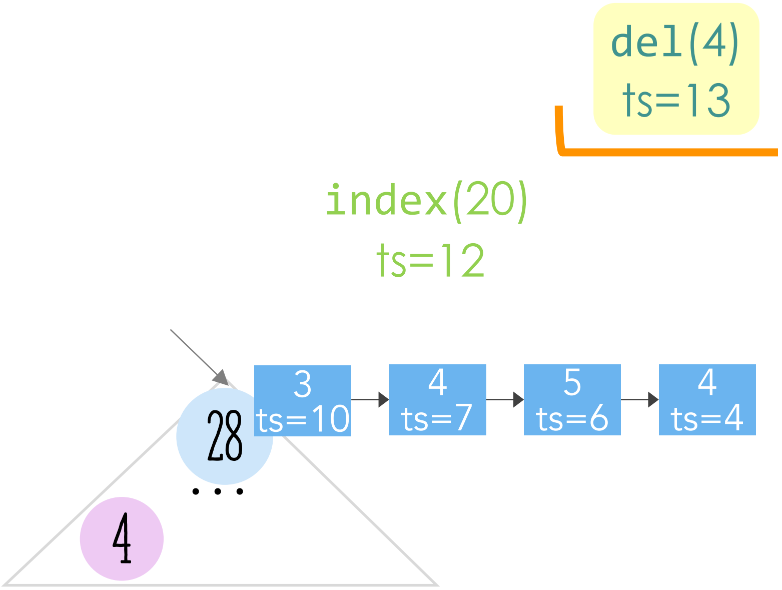
index(20)
ts=12

aggregate value, child pointers ➡ version lists



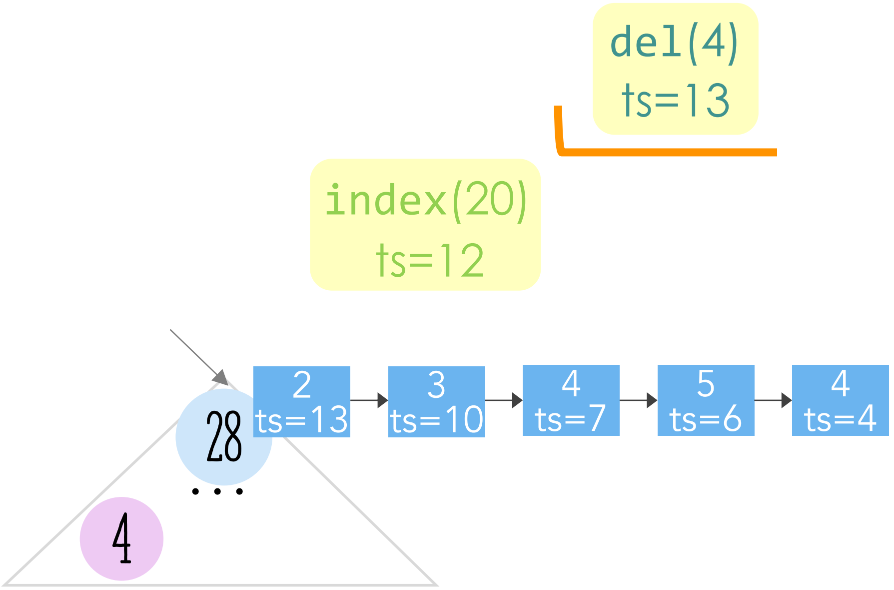
Multi-Versioning

to ignore concurrent updates
with bigger timestamps



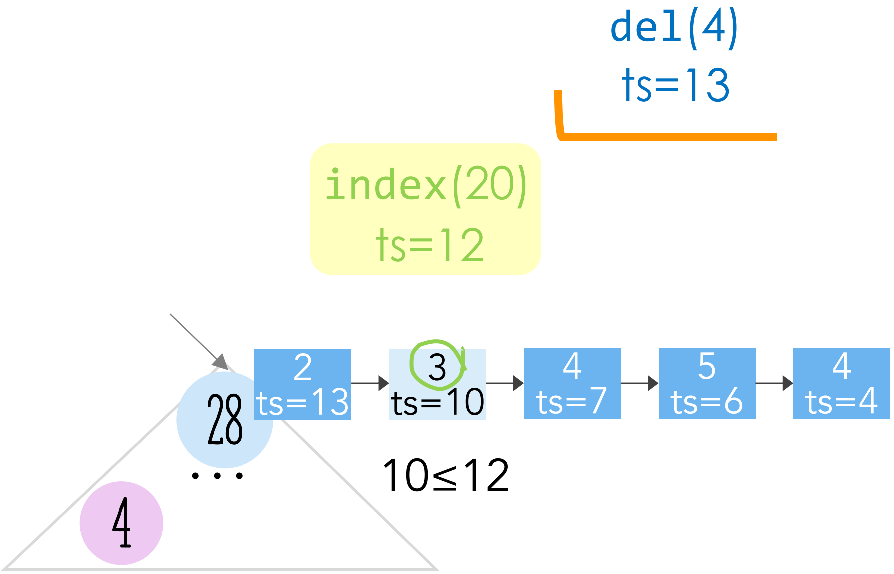
Multi-Versioning

to ignore concurrent updates with bigger timestamps



Multi-Versioning

to ignore concurrent updates with bigger timestamps



Timestamps

Ongoing update operations

`ins(7)` ts=11 `del(10)` ts=12 `del(4)` ts=13

consider *by announcements*

ignore *by multi-versioning* ✓

Aggregate queries

`index(20)` ts=**12**

Announcements

to consider concurrent updates with timestamps ≤ query

Ongoing update operations

ins(7) ts=11

del(10) ts=12

Aggregate queries

index(20) ts=12

Concurrent Aggregate Queries

Design Main Ideas

- Timestamps
- Multi-versioning
- Announcements

2 implementing algorithms – optimizing either *update time* or *aggregate query time*

Timestamps in FastQueryTree

By enqueueing to announcement queue

`ins(7)`	`del(7)`	`ins(20)`	`del(3)`
ts=11	ts=12	ts=13	ts=14

Timestamps in FastUpdateTree

global timestamp

3

Incremented by aggregate queries

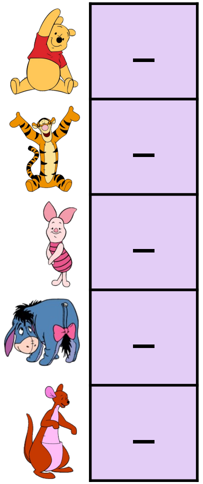
Timestamps in FastUpdateTree

global timestamp

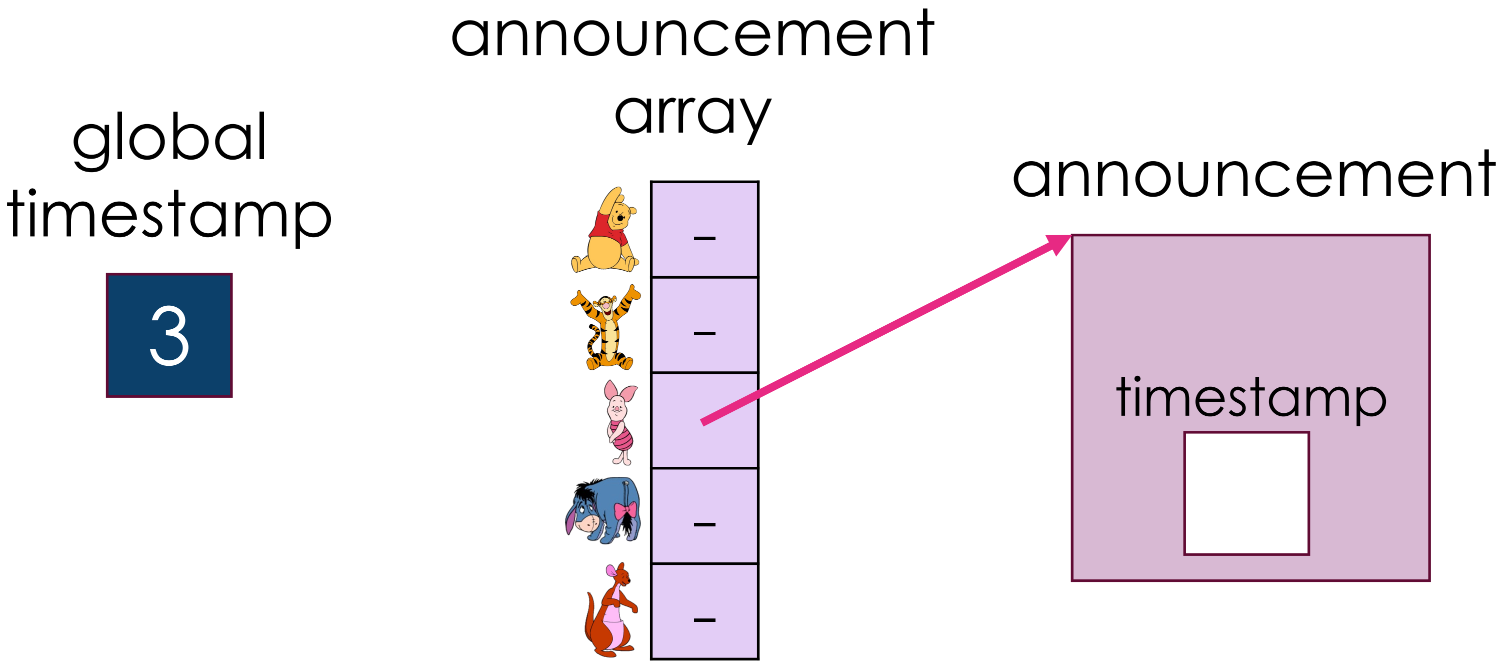
3

Incremented by aggregate queries

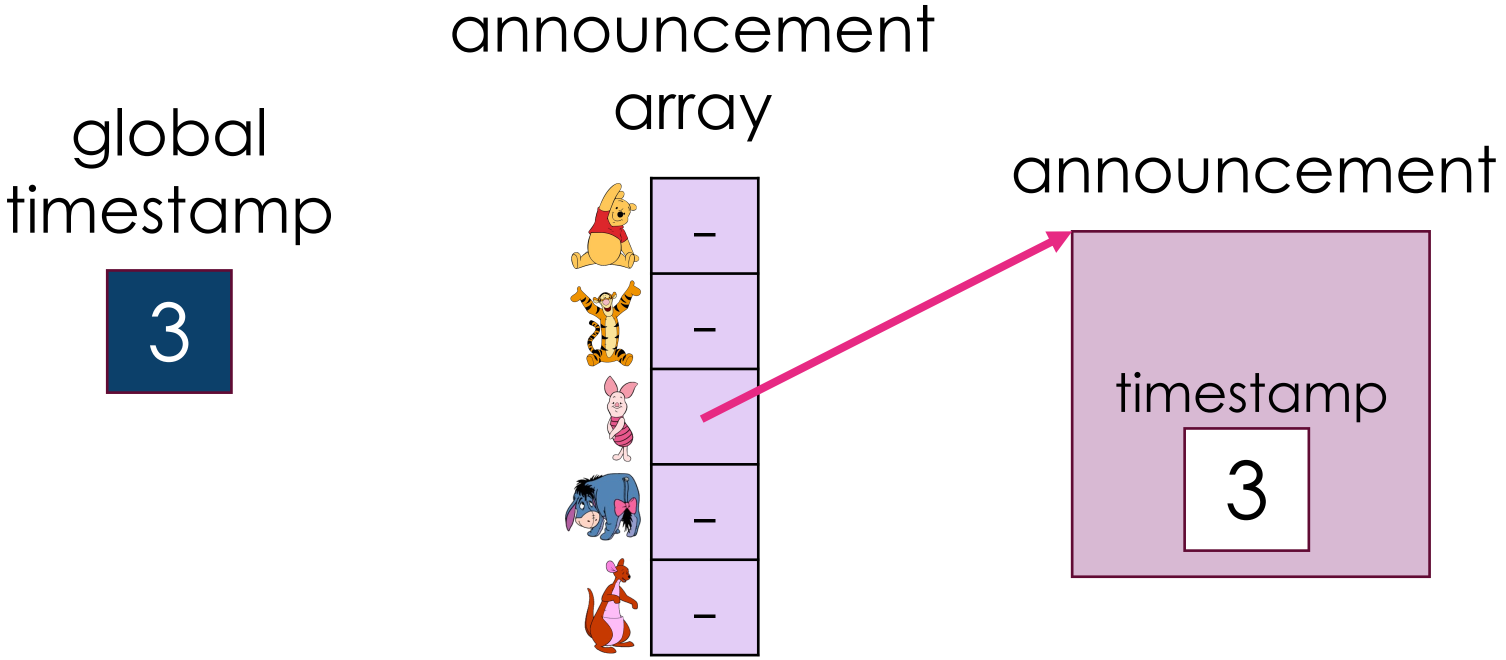
announcement array



Timestamps in FastUpdateTree



Timestamps in FastUpdateTree

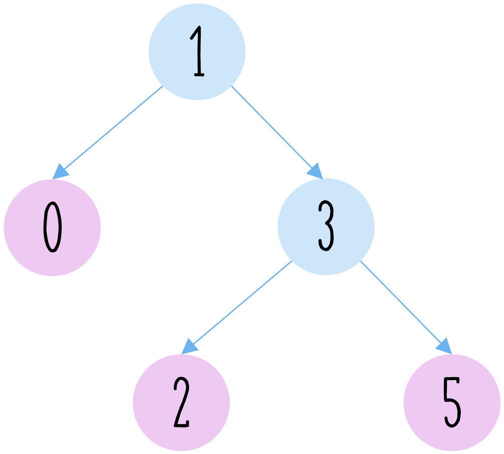


Lock-Free Augmented Trees

Panagiota Fatourou and Eric Ruppert

DISC'24

Lock-Free Augmented Trees

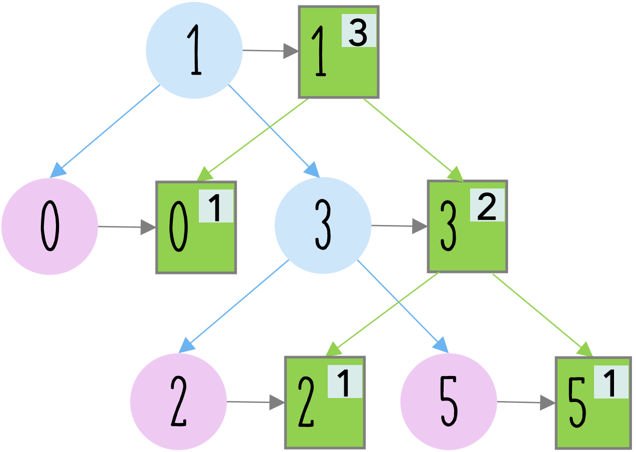


Multi-versioning

- No version lists
- No timestamps
  - Order determined by arrival at root

Lock-Free Augmented Trees

`ins(4)`



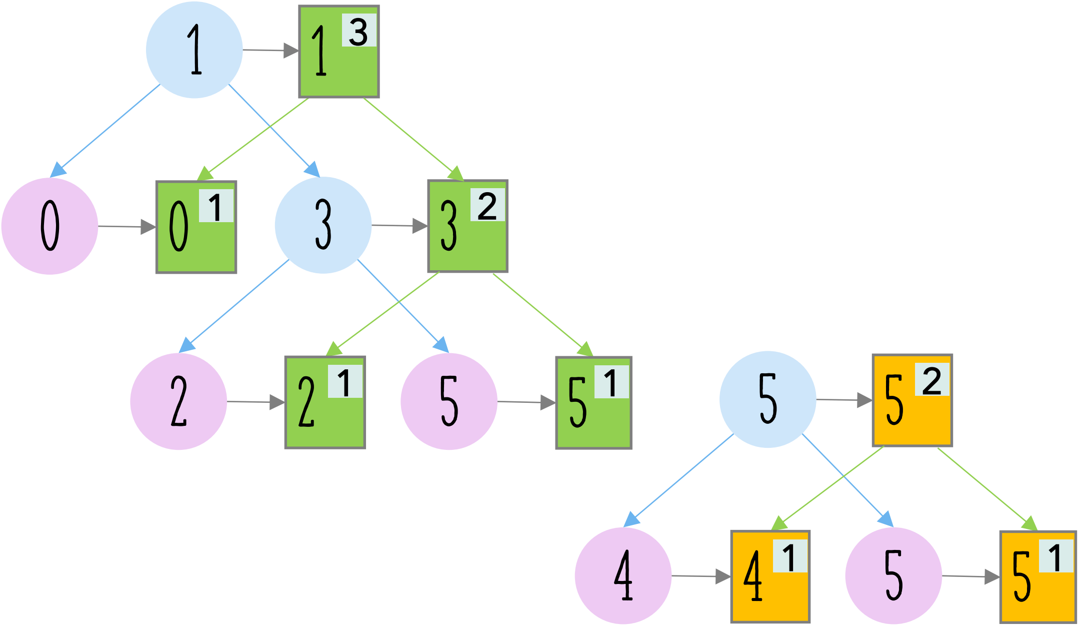
Multi-versioning

Tree replication

with aggregate metadata

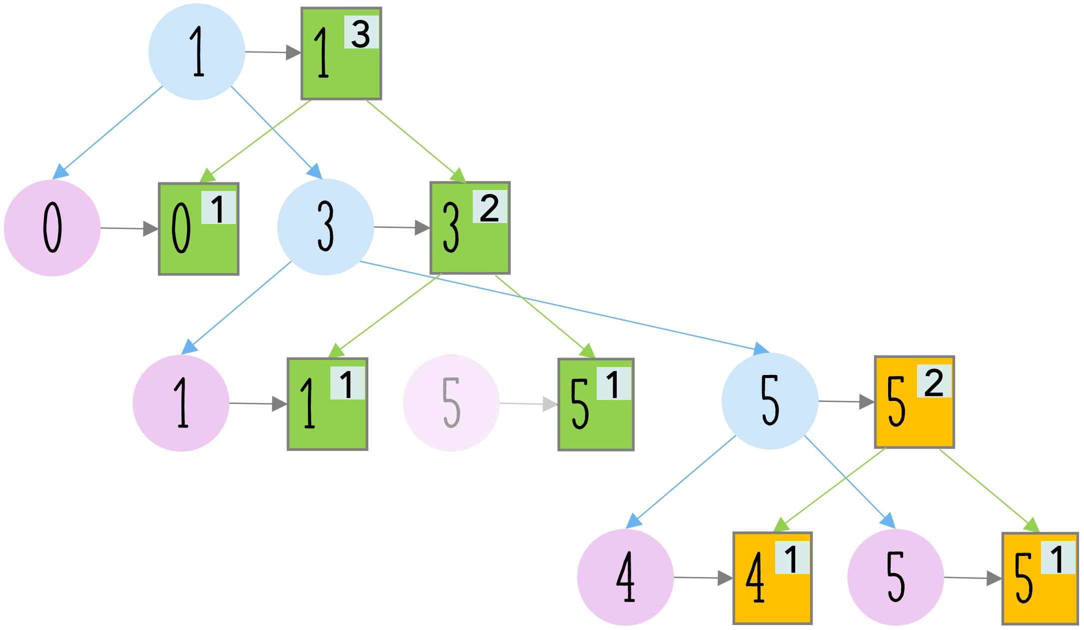
Lock-Free Augmented Trees

`ins(4)`



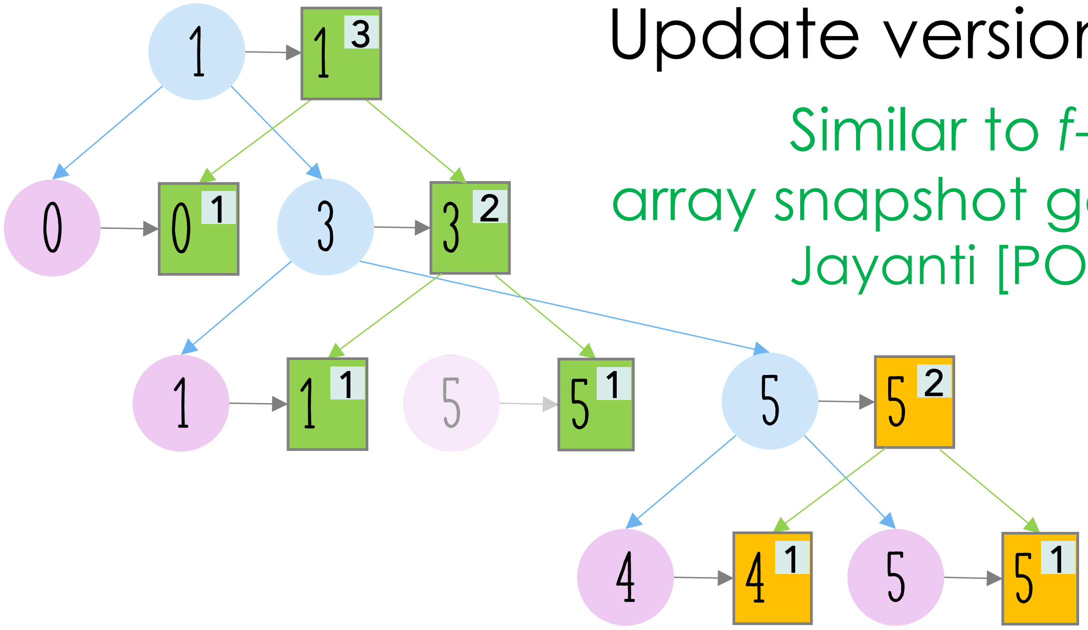
Lock-Free Augmented Trees

ins(4)



Lock-Free Augmented Trees

`ins(4)`



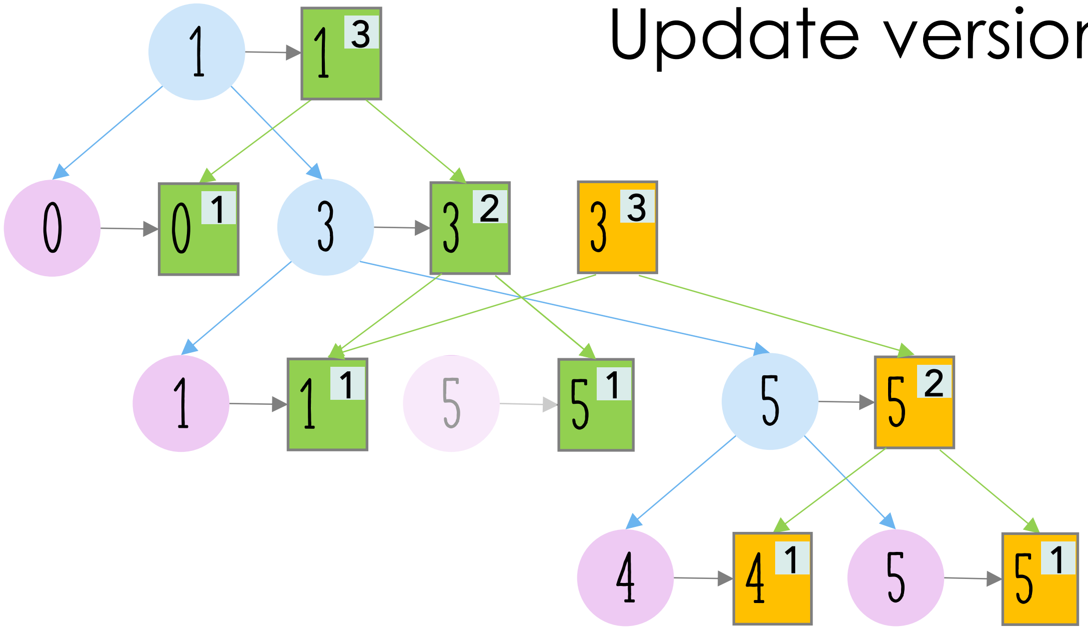
Update versions upwards

Similar to *f-Arrays*
array snapshot generalization
Jayanti [PODC'02]

Lock-Free Augmented Trees

`ins(4)`

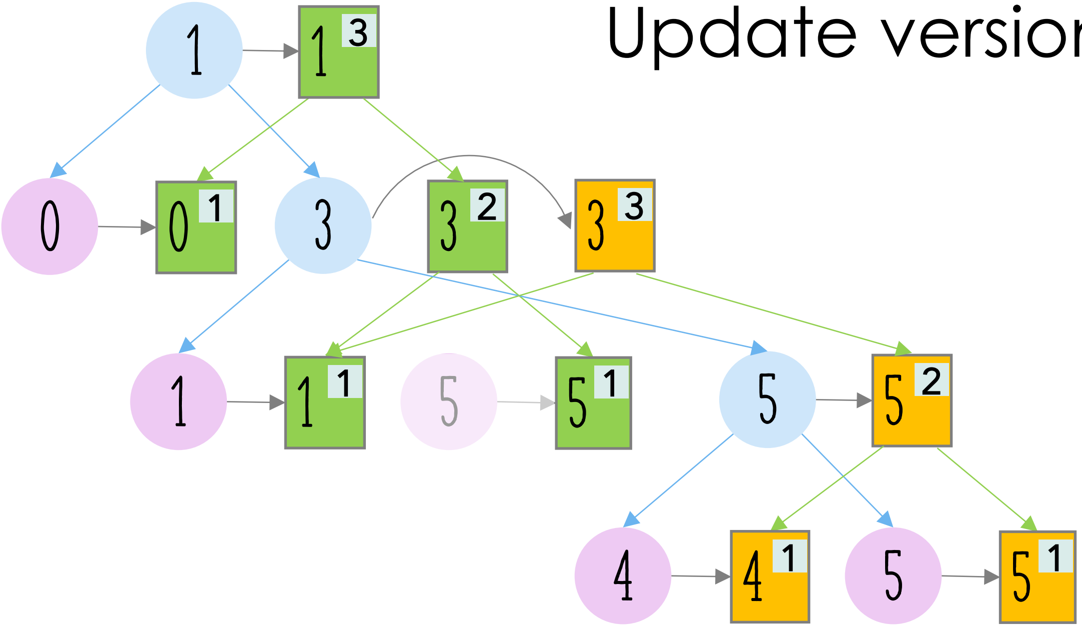
Update versions upwards



Lock-Free Augmented Trees

`ins(4)`

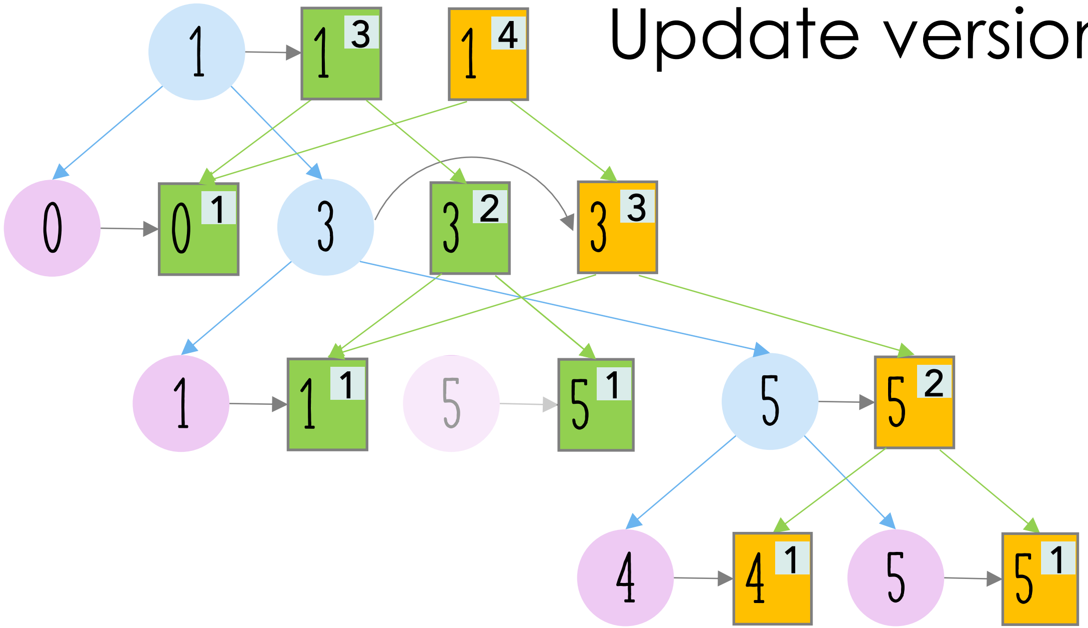
Update versions upwards



Lock-Free Augmented Trees

`ins(4)`

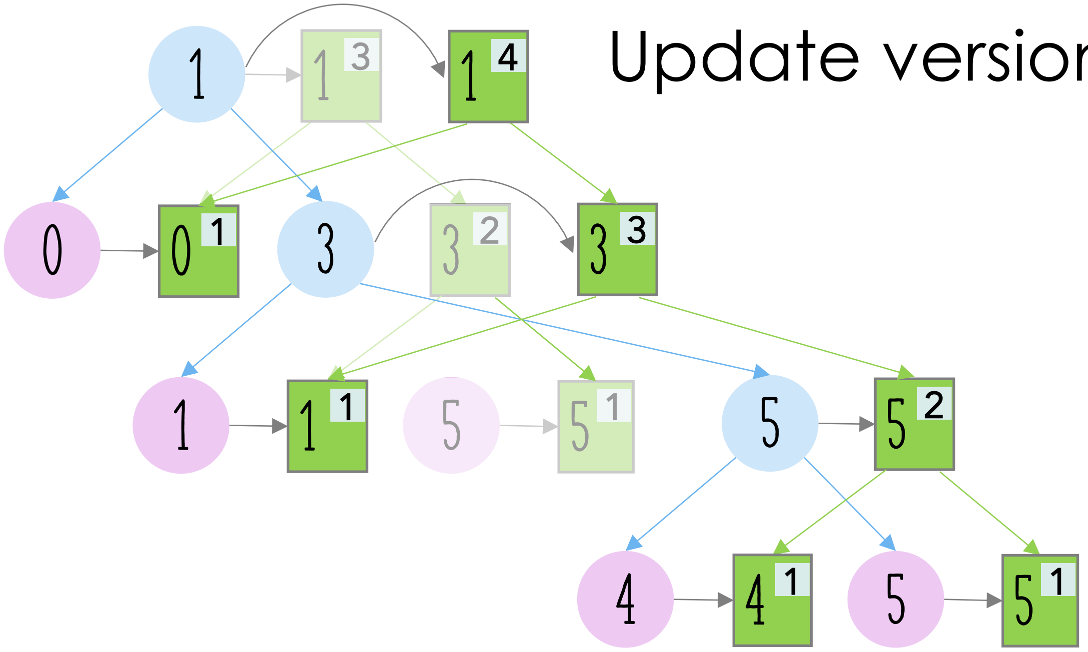
Update versions upwards



Lock-Free Augmented Trees

ins(4)

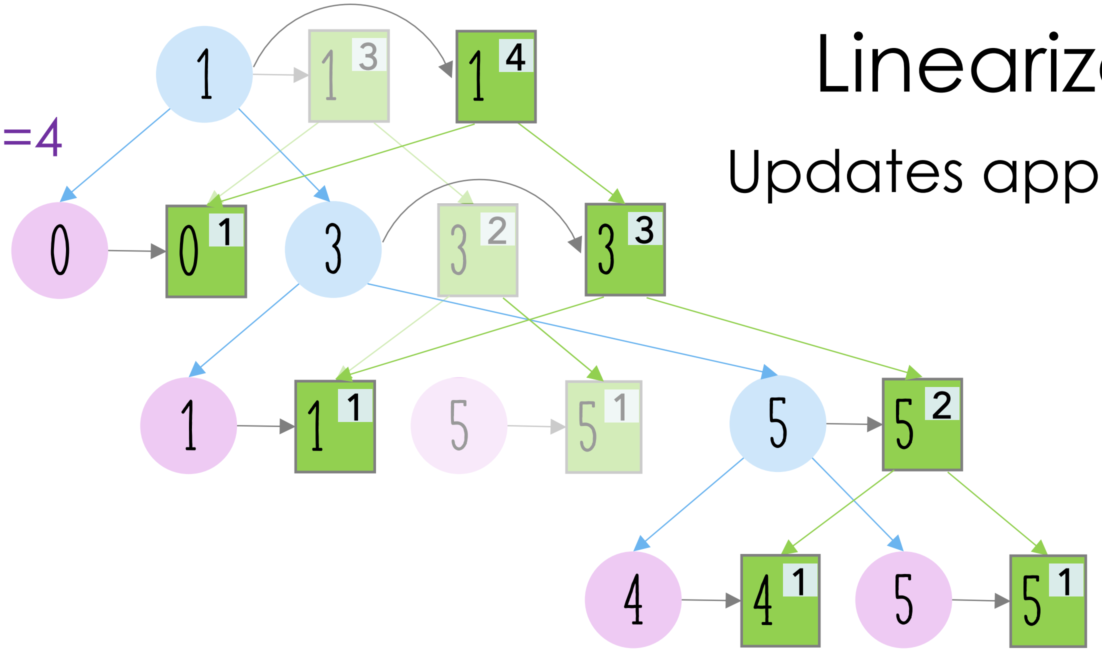
Update versions upwards



Lock-Free Augmented Trees

`ins(4)`

`index(5)=4`

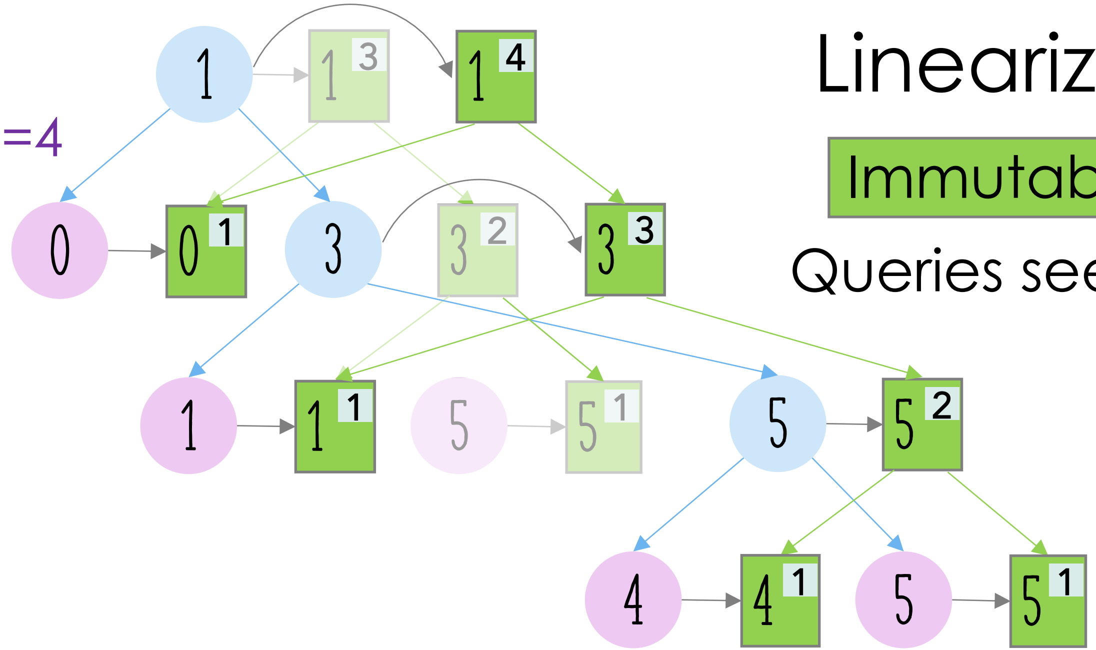


Linearize at root

Updates appear atomically

Lock-Free Augmented Trees

`ins(4)`
`index(5)=4`



Linearize at root

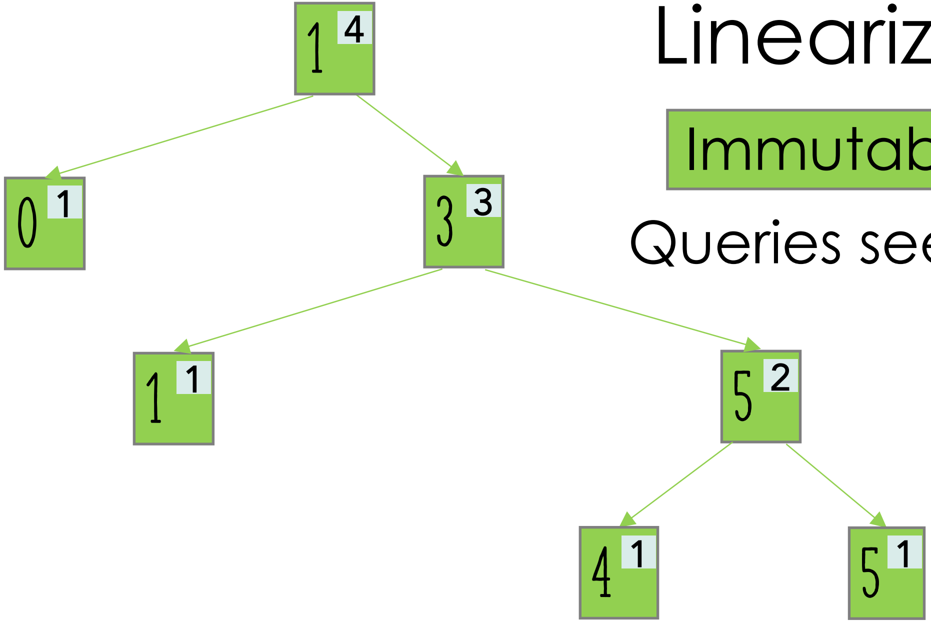
Immutable versions

Queries see a snapshot

Lock-Free Augmented Trees

ins(4)

index(5)=4



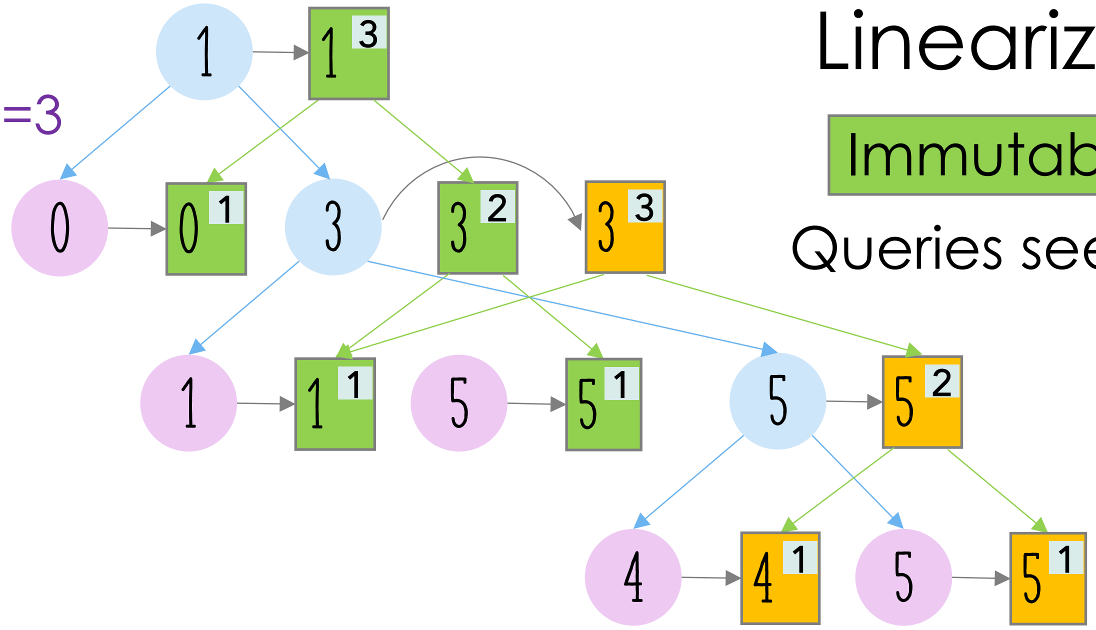
Linearize at root

Immutable versions

Queries see a snapshot

Lock-Free Augmented Trees

`ins(4)`
`index(5)=3`



Linearize at root

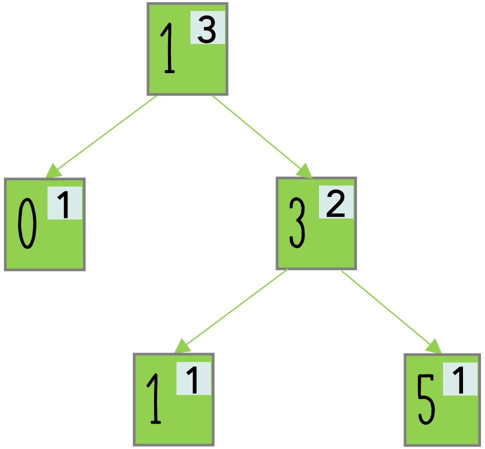
Immutable versions

Queries see a snapshot

Lock-Free Augmented Trees

`ins(4)`

`index(5)=3`



Linearize at root

Immutable versions

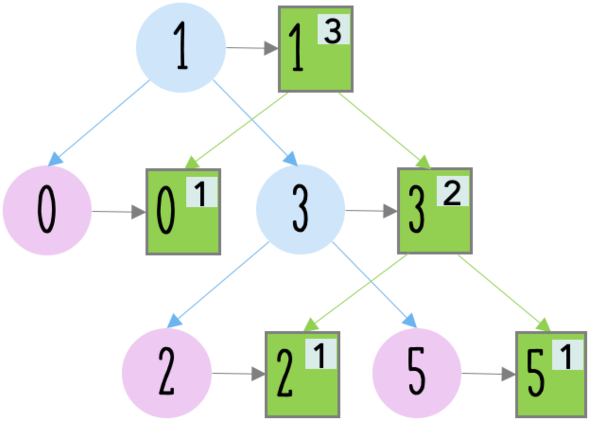
Queries see a snapshot

Lock-Free Augmented Trees

Main Ideas

Multi-versioning

- ❖ No version lists
- ❖ No timestamps
  - ➢ Order determined by arrival at root



Outlook

- Fast solutions (to be used in libraries)
  - Reduce overhead, possibly relax progress or correctness
  - Investigate trade-offs
    - between time of existing ops and aggregate queries
    - time–space

Outlook

- Fast solutions (to be used in libraries)
- Memory management for multi-versioning
- Apply to more data structures
- Apply to more operations (not only queries)