Lock-free Augmented Trees

Panagiota Fatourou, U. of Crete and FORTH, Greece
Eric Ruppert, York University, Canada

DISC 2024
Madrid, Spain

YORK UNIVERSITÉ UNIVERSITY

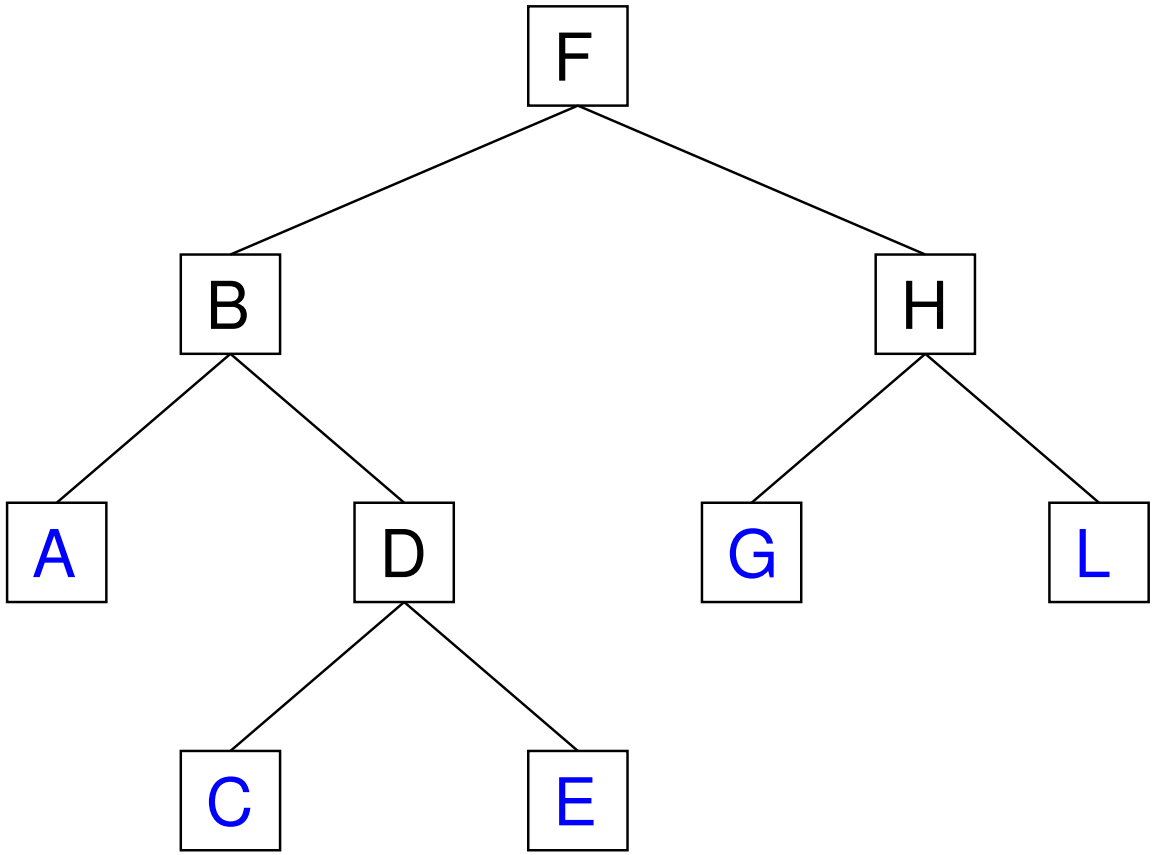
Main Result

Technique to augment lock-free search trees to support more operations.

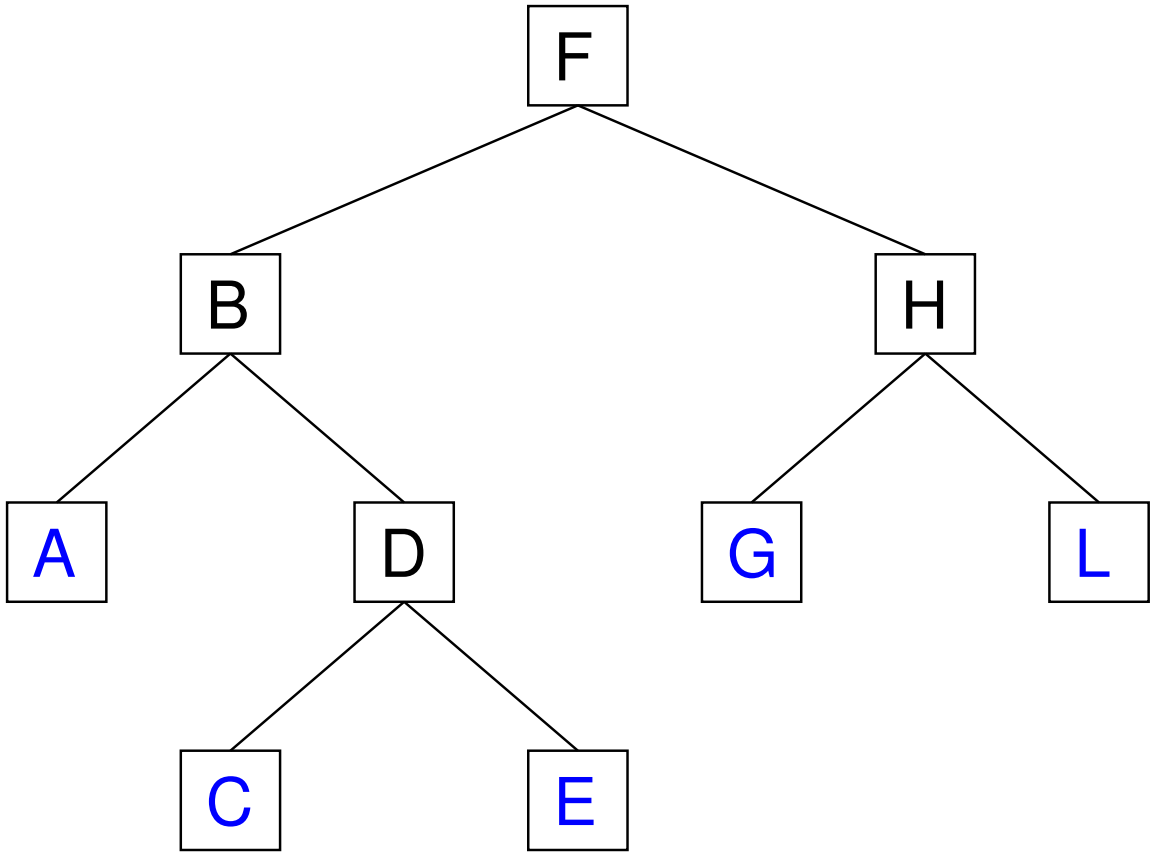
- Simple to implement using single-word CAS
- General: can handle any augmentation
- Efficient: queries as fast as in sequential system, minimal overhead for updates
- Wait-free: additional work for augmentation is wait-free
- Snapshots of tree easily support complex queries

YORK UNIVERSITÉ UNIVERSITY

Augmented Search Tree



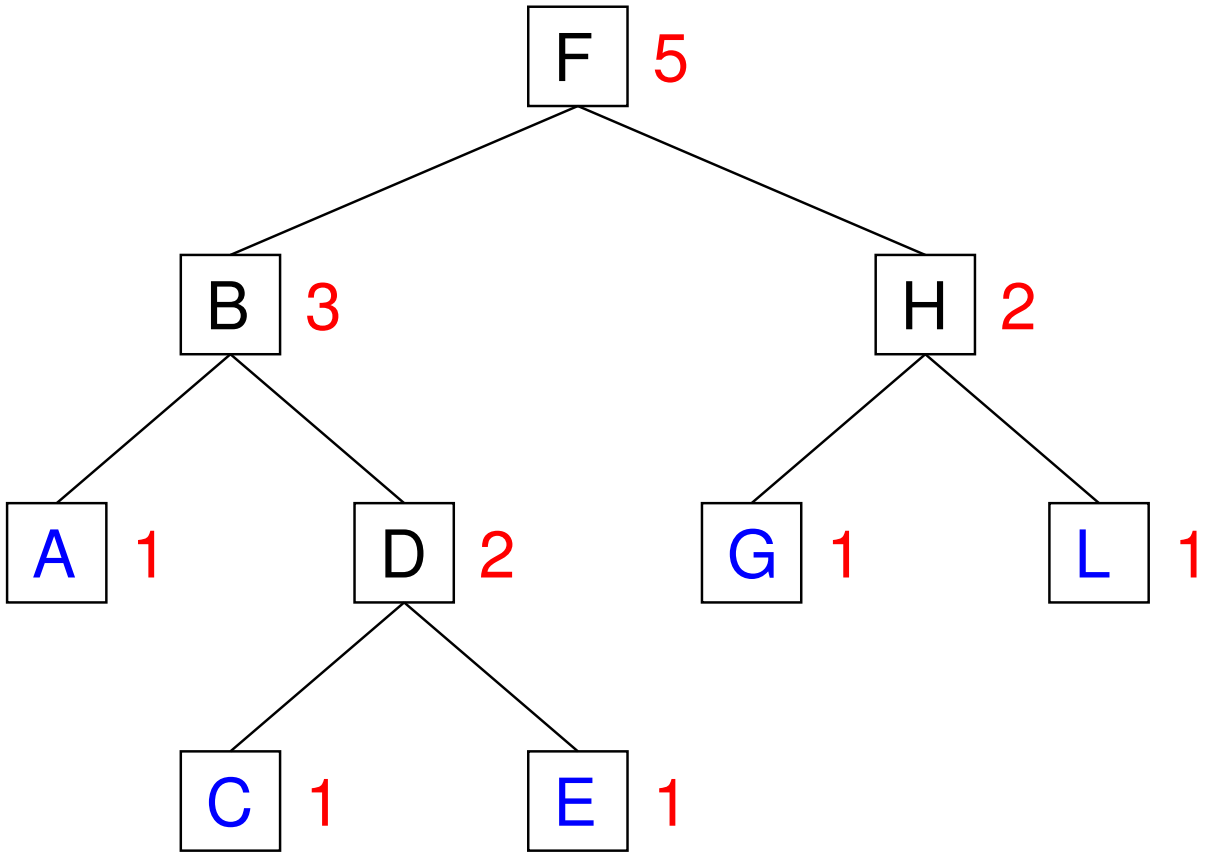
Augmented Search Tree



Leaf-oriented BST. Set = {A, C, E, G, L}.

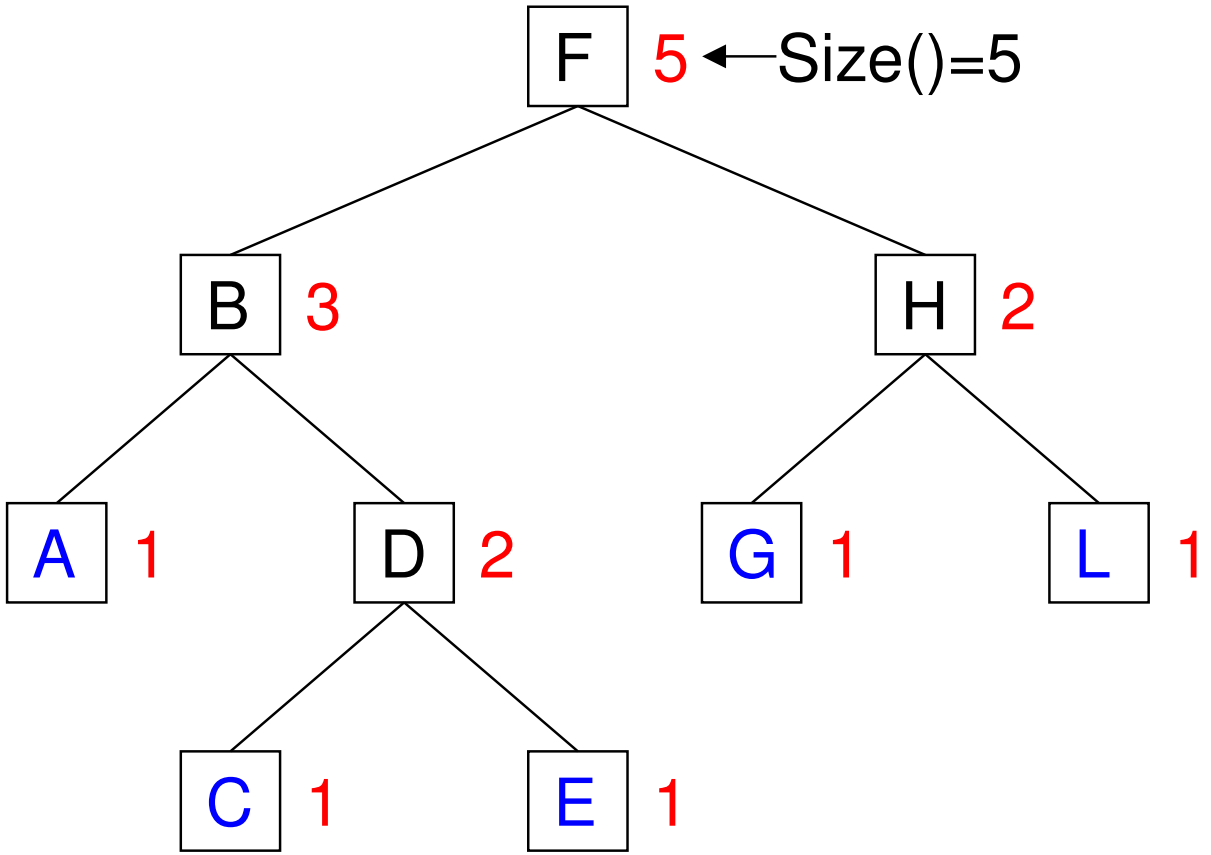
YORK UNIVERSITÉ UNIVERSITY

Augmented Search Tree



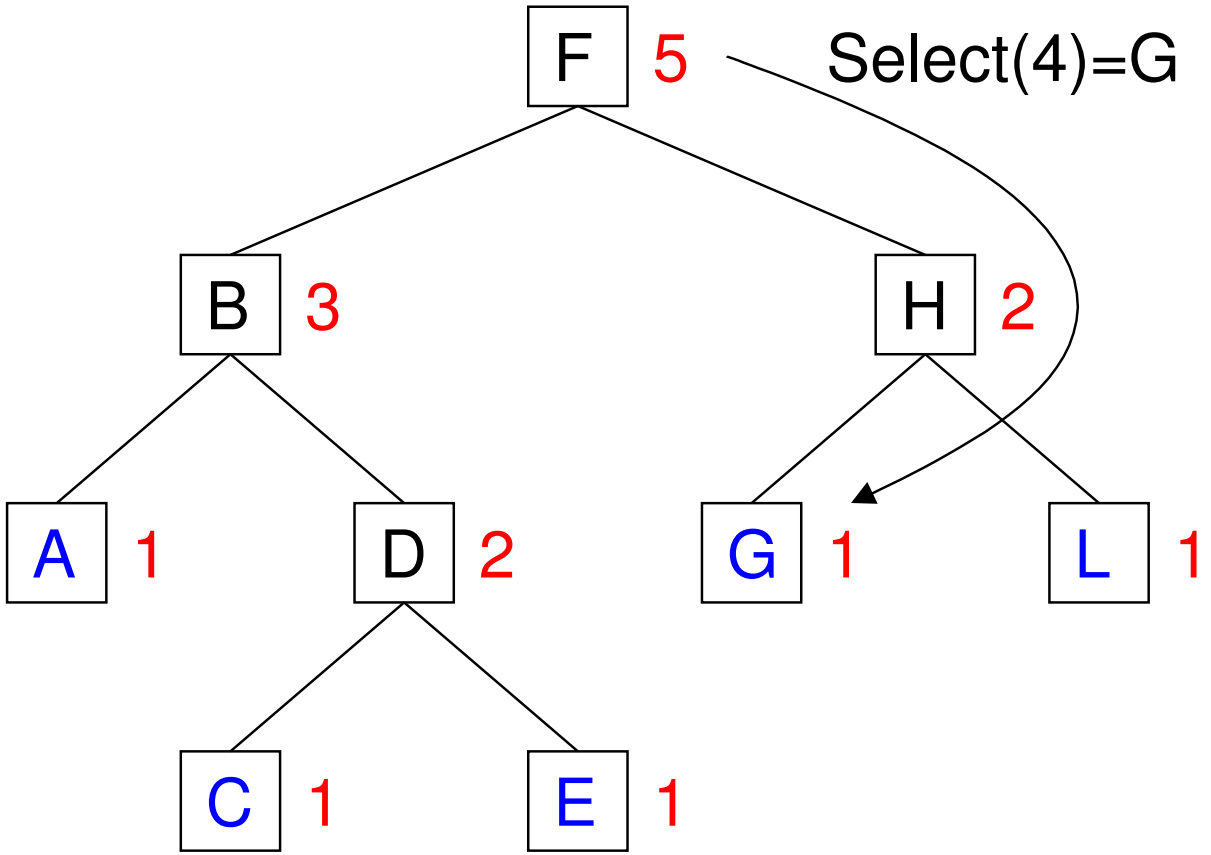
Augmentation: Each node stores number of leaves in subtree.

Augmented Search Tree



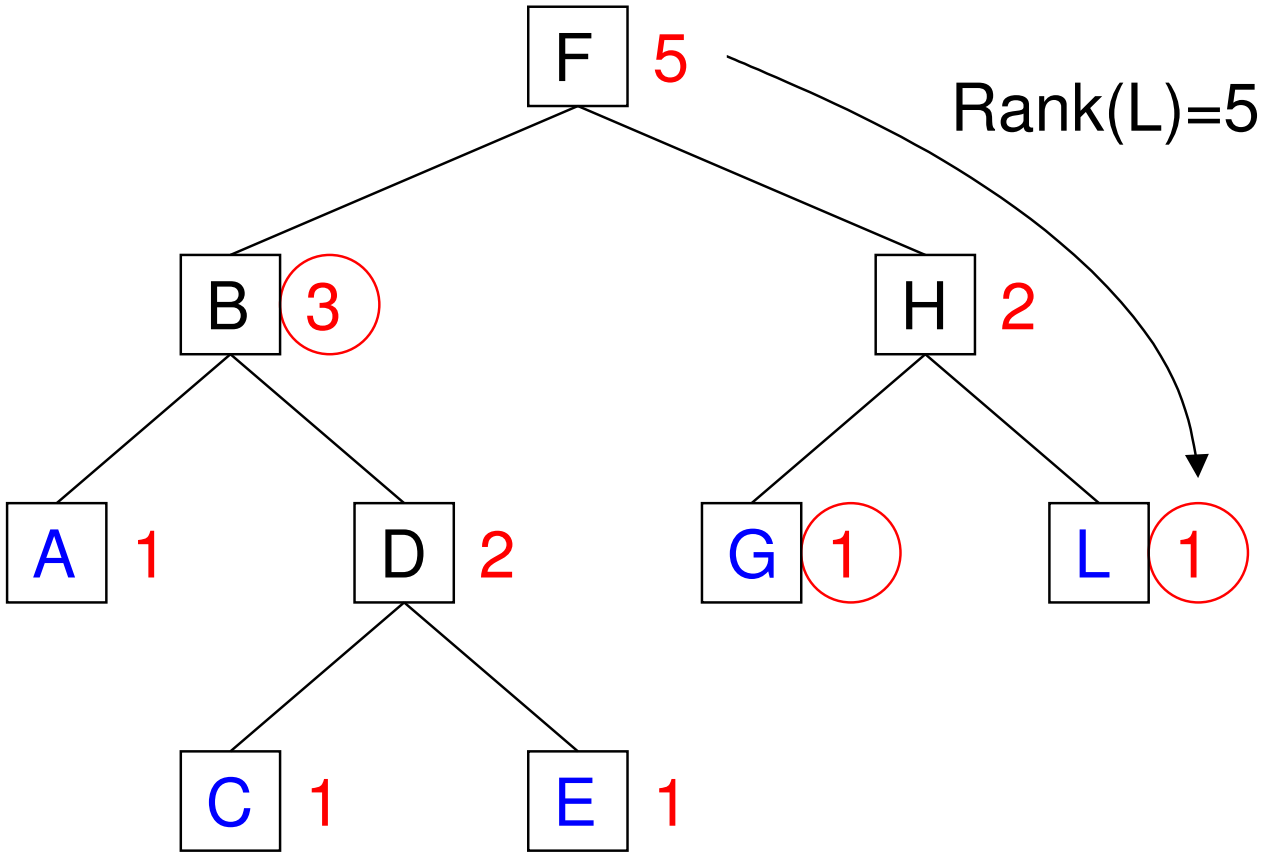
Augmentation: Each node stores number of leaves in subtree.

Augmented Search Tree



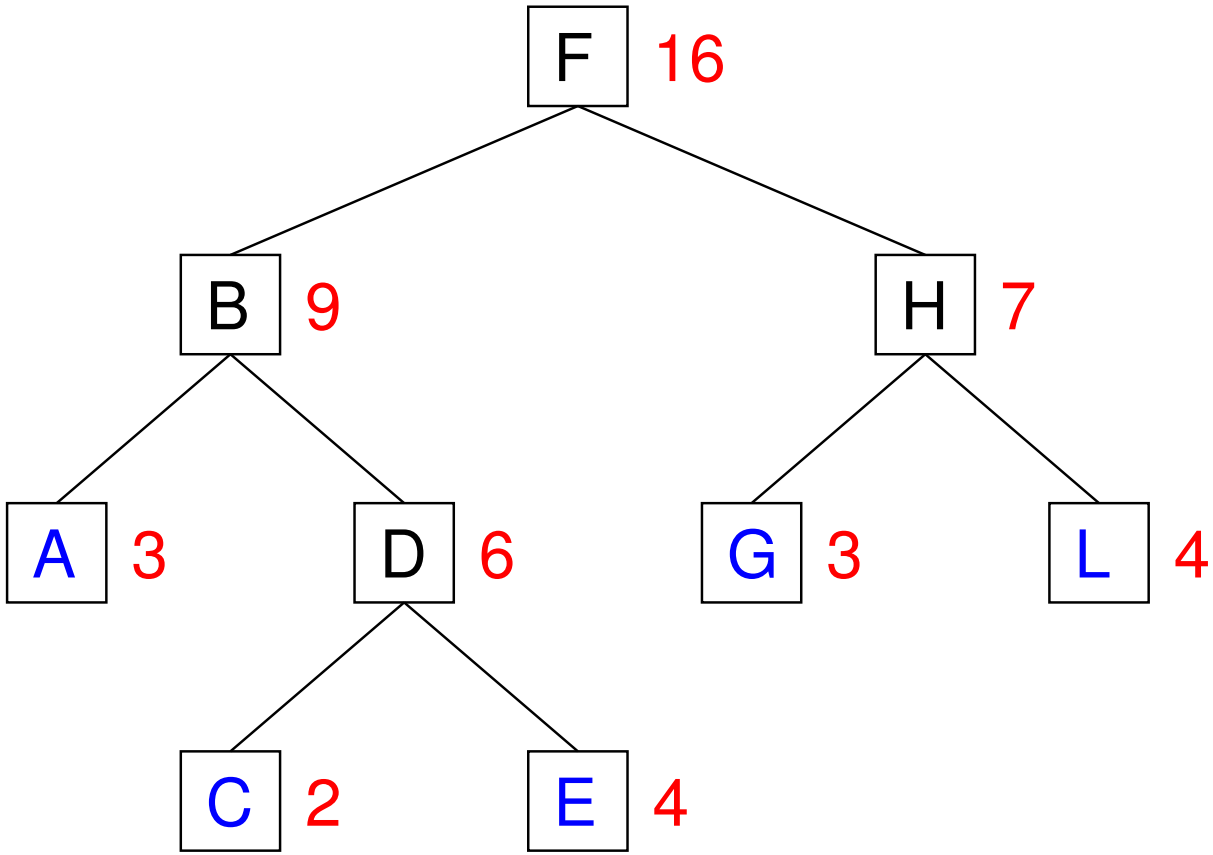
Augmentation: Each node stores number of leaves in subtree.

Augmented Search Tree



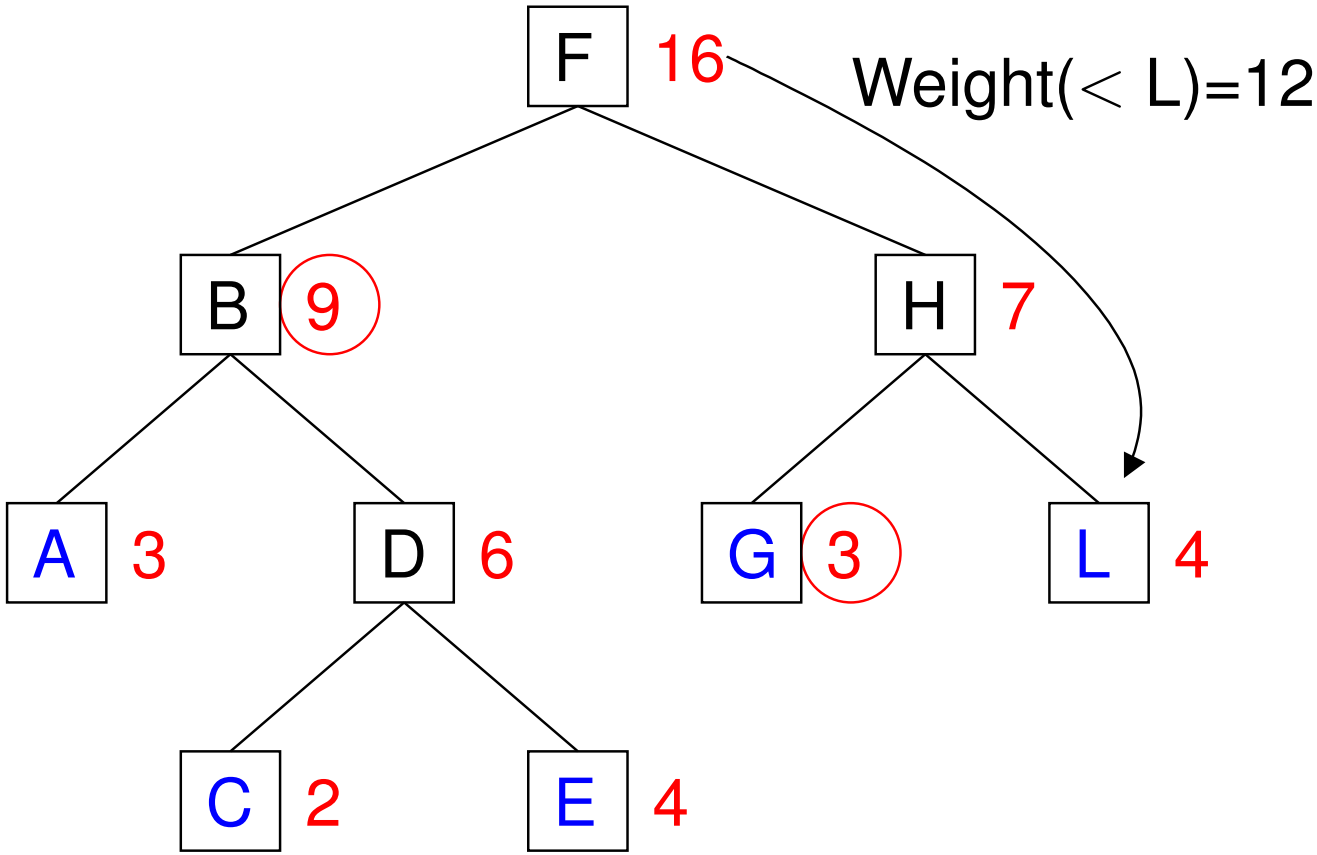
Augmentation: Each node stores number of leaves in subtree.

Augmented Search Tree



Keys have weights; each node stores sum of subtree's weights.

Augmented Search Tree



Keys have weights; each node stores sum of subtree's weights.

Other Augmentations

Many other ways to augment a BST.

- For database of employees, number of women in subtree.
  How many women's salaries are more than €100,000?
- Store min key, max key, and smallest gap in subtree.
  Find two closest keys in the set.

Key Property of Augmentations

Values of a node's new field(s) can be computed from information in node and its children.

YORK UNIVERSITÉ UNIVERSITY

Other Augmentations

Many other ways to augment a BST.

- For database of employees, number of women in subtree.
  How many women's salaries are more than €100,000?
- Store min key, max key, and smallest gap in subtree.
  Find two closest keys in the set.

Key Property of Augmentations

Values of a node's new field(s) can be computed from information in node and its children.

YORK UNIVERSITÉ UNIVERSITY

Other Augmentations

Many other ways to augment a BST.

- For database of employees, number of women in subtree.
  How many women's salaries are more than €100,000?
- Store min key, max key, and smallest gap in subtree.
  Find two closest keys in the set.

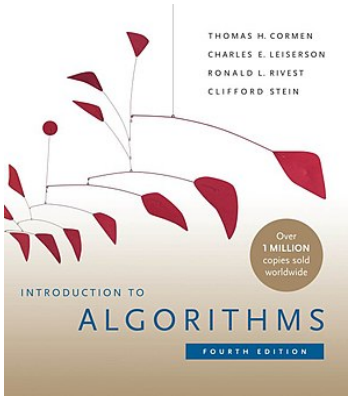
Key Property of Augmentations

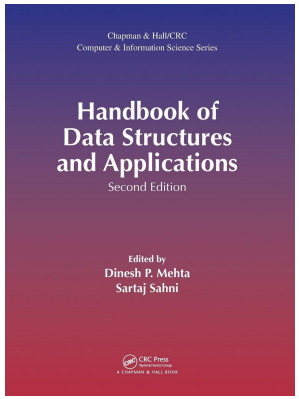
Values of a node's new field(s) can be computed from information in node and its children.

Applications of Augmented Trees

Augmented BSTs are basis of many other data structures.

- Interval tree
- Tango tree
- Measure tree
- Priority search tree
- Segment tree
- Link/cut tree
- . . .





Lots of applications in computational geometry, databases, graph algorithms, ...

Result

Main Result

Technique to augment lock-free search trees.

Example: Lock-free Binary Search Tree

- Can handle any augmentation.
- Adds only $O(height)$ steps to insert, delete.
- Supports simple snapshots.
- Wait-free queries run sequential code.
- Based on BST from PODC 2014.

YORK UNIVERSITÉ UNIVERSITY

Result

Main Result

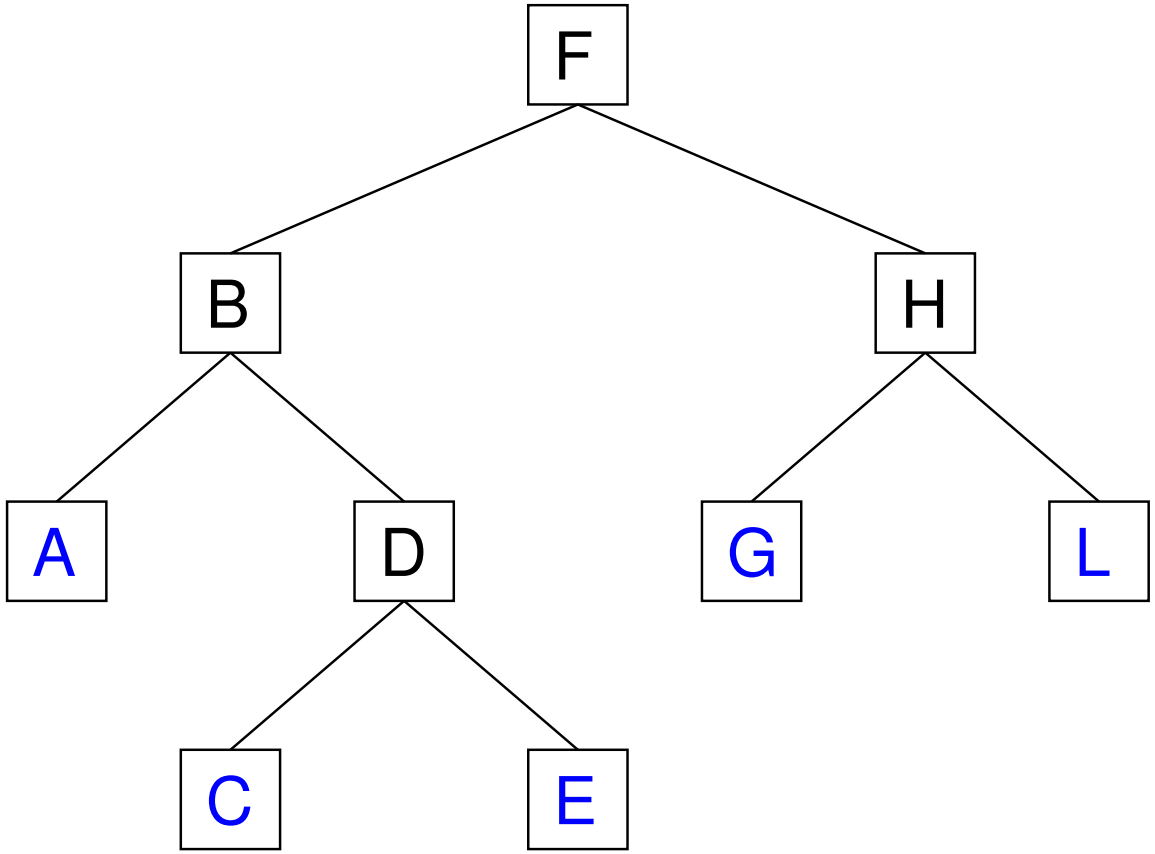
Technique to augment lock-free search trees.

Example: Lock-free Binary Search Tree

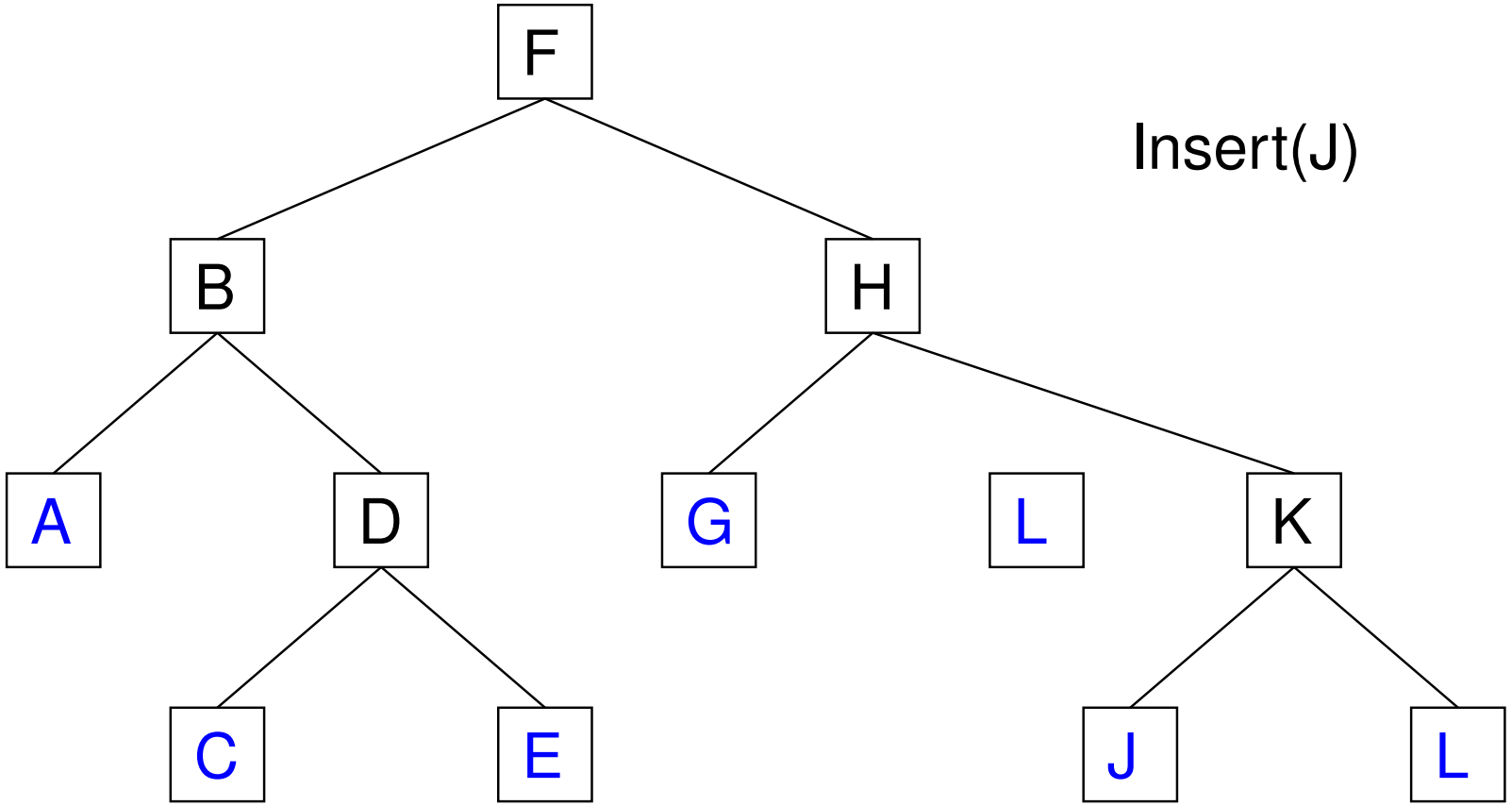
- Can handle any augmentation.
- Adds only $O(height)$ steps to insert, delete.
- Supports simple snapshots.
- Wait-free queries run sequential code.
- Based on BST from PODC 2014.

YORK UNIVERSITÉ UNIVERSITY

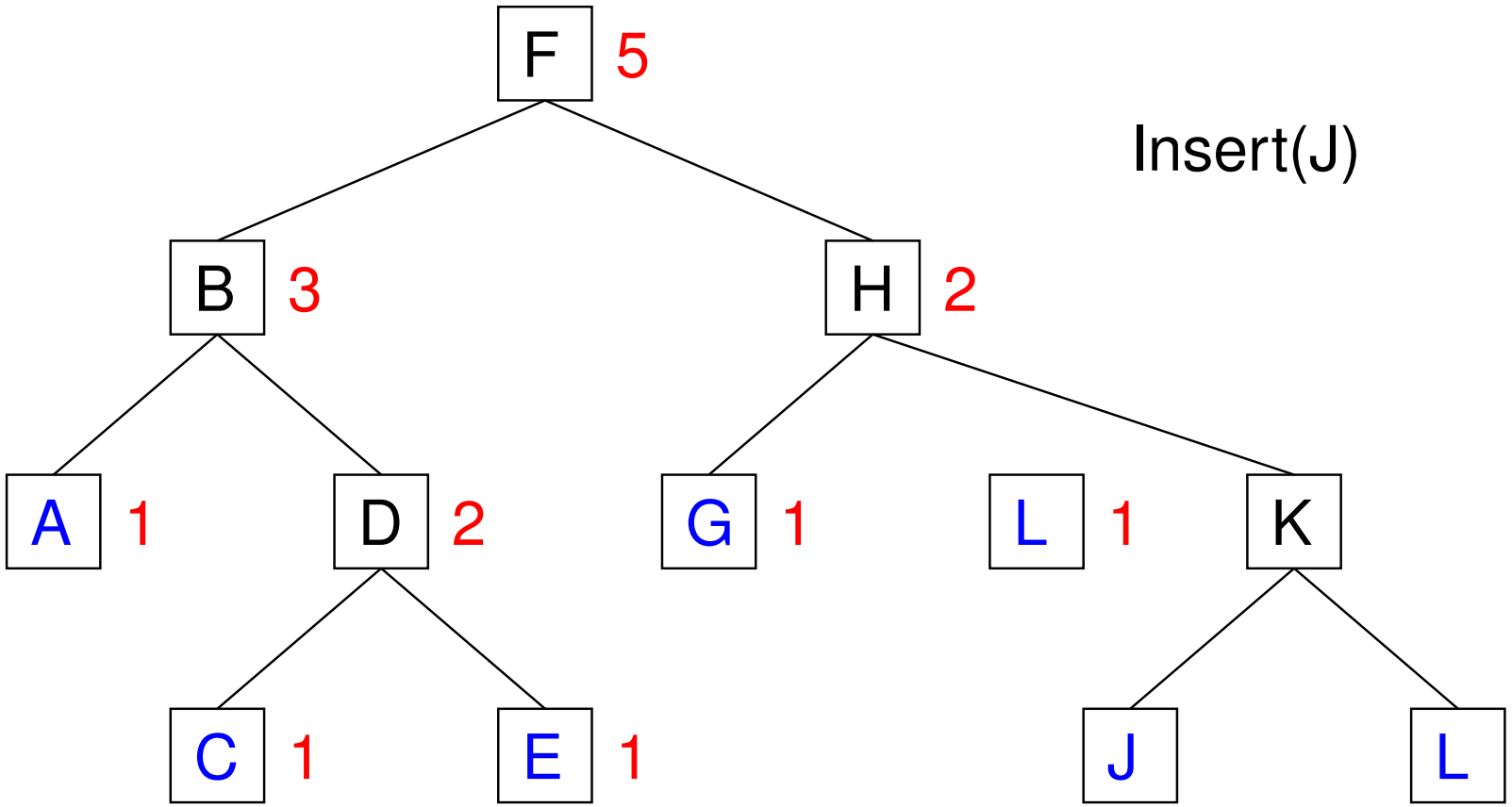
Updating an Augmented Tree



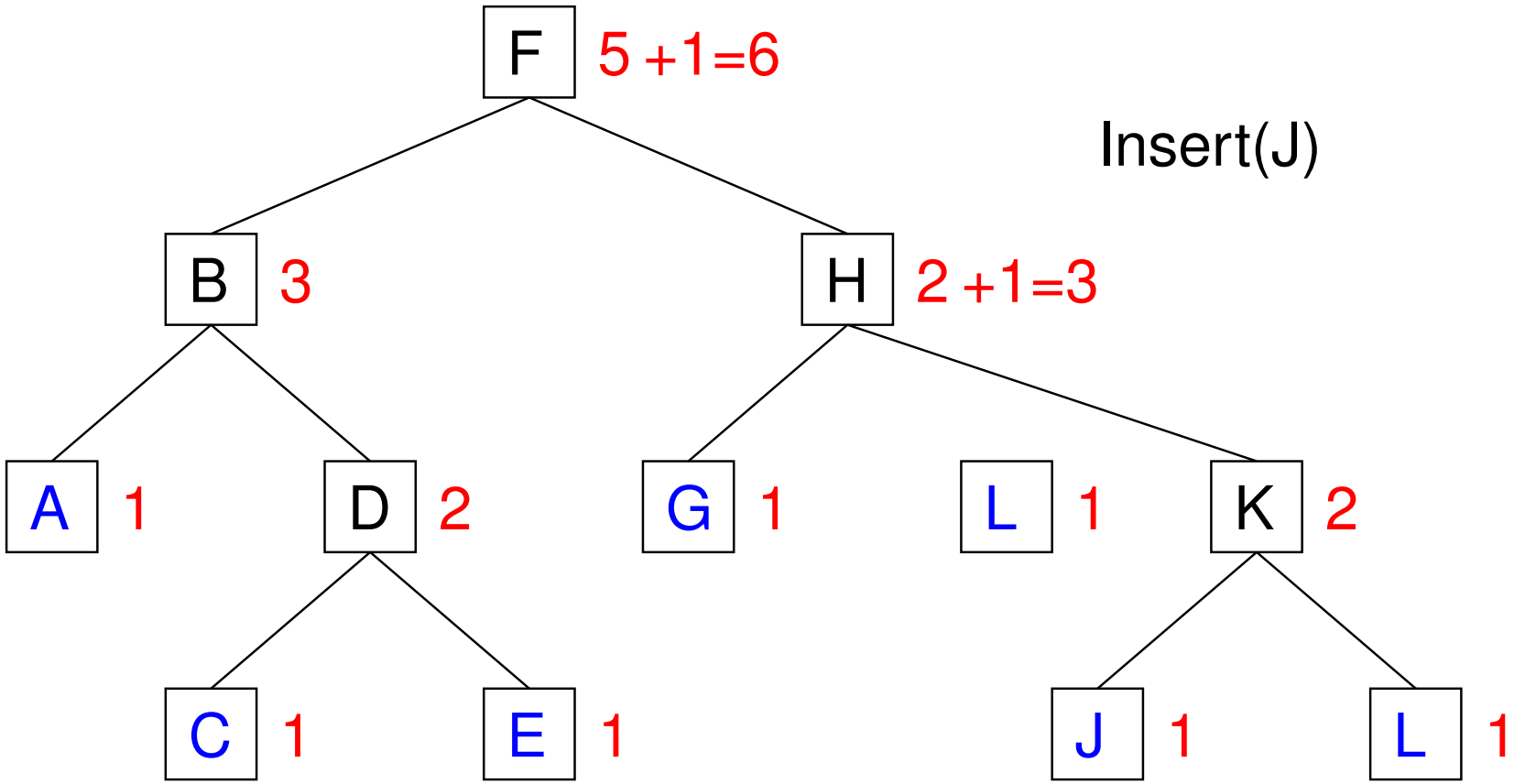
Updating an Augmented Tree



Updating an Augmented Tree



Updating an Augmented Tree



Challenges of Concurrency

Goal

Create lock-free augmented tree.

Challenges

- An update changes many nodes along a path
  All changes must appear atomic
- Queries traverse a path while concurrent updates change it
- Contention: all updates need to modify root's size

YORK UNIVERSITÉ UNIVERSITY

Challenges of Concurrency

Goal

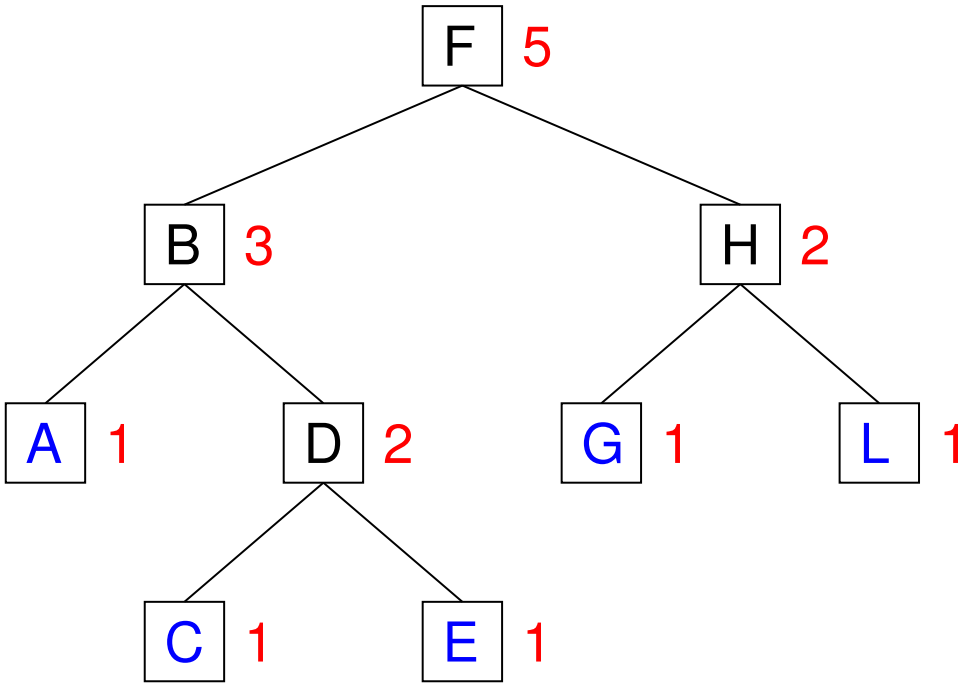
Create lock-free augmented tree.

Challenges

- An update changes many nodes along a path
  All changes must appear atomic
- Queries traverse a path while concurrent updates change it
- Contention: all updates need to modify root's size

YORK UNIVERSITÉ UNIVERSITY

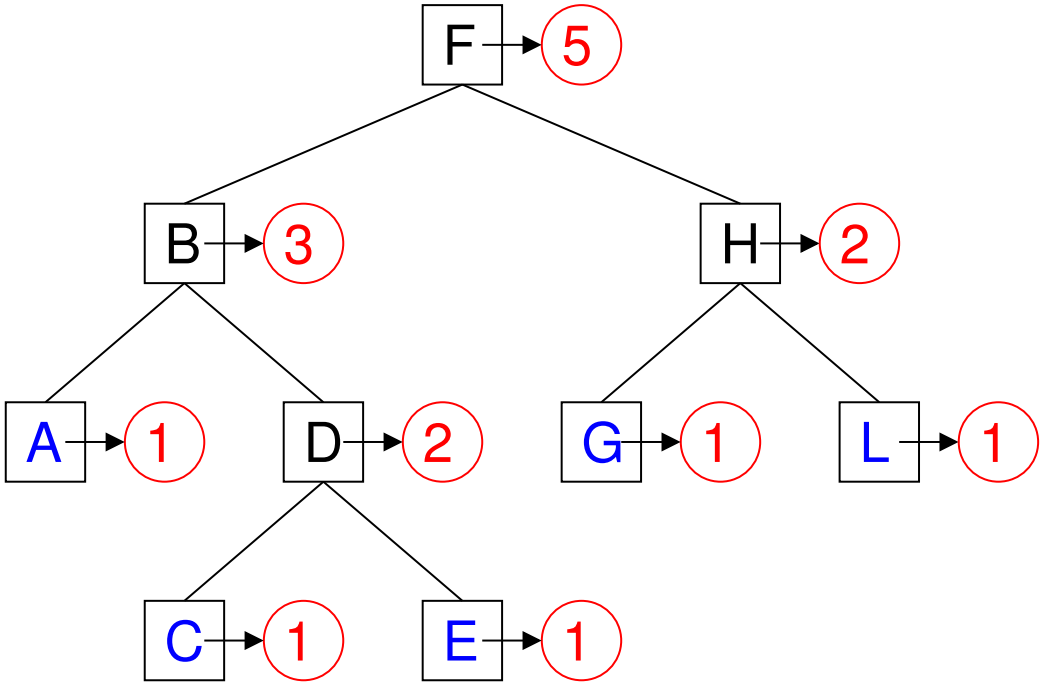
Key Idea



Node stores pointer to current version of augmented field.

- Old versions can still be used by queries in progress.
- Pointers between versions provide consistent view.

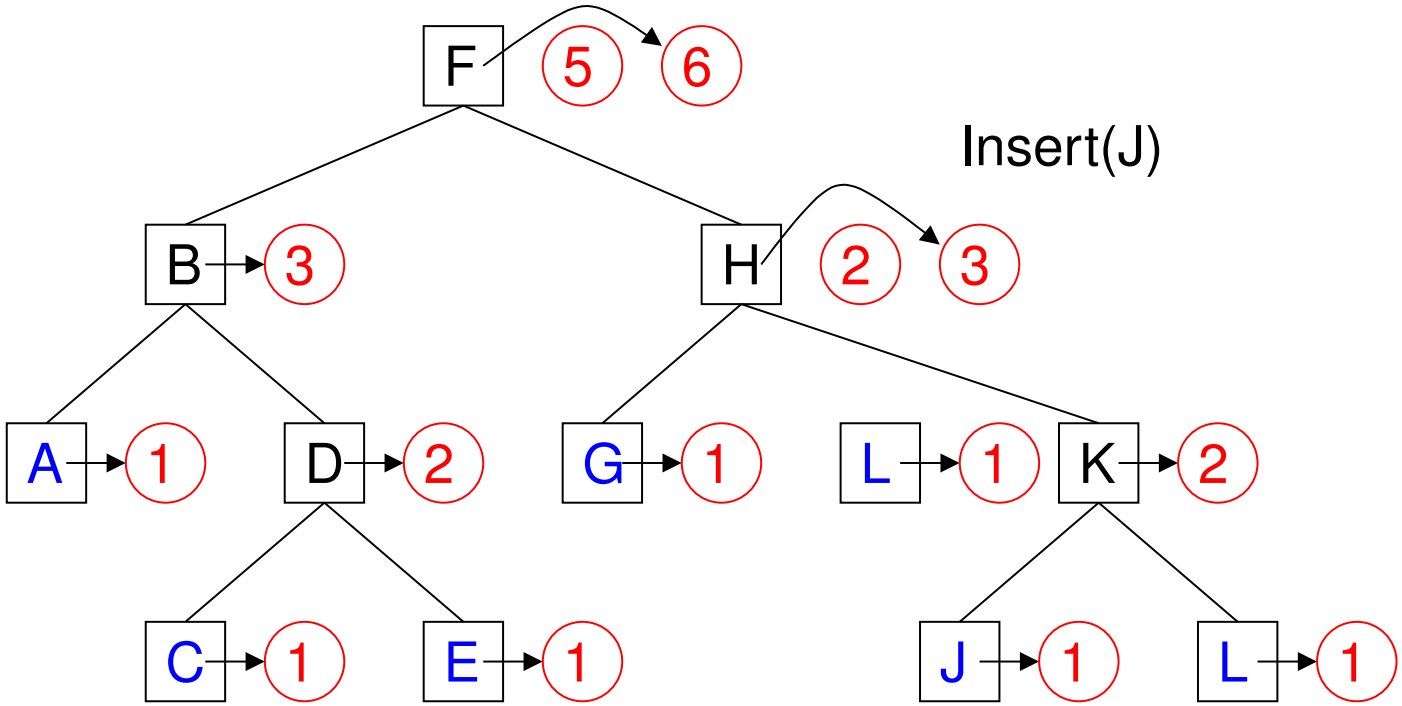
Key Idea



Node stores pointer to current version of augmented field.

- Old versions can still be used by queries in progress.
- Pointers between versions provide consistent view.

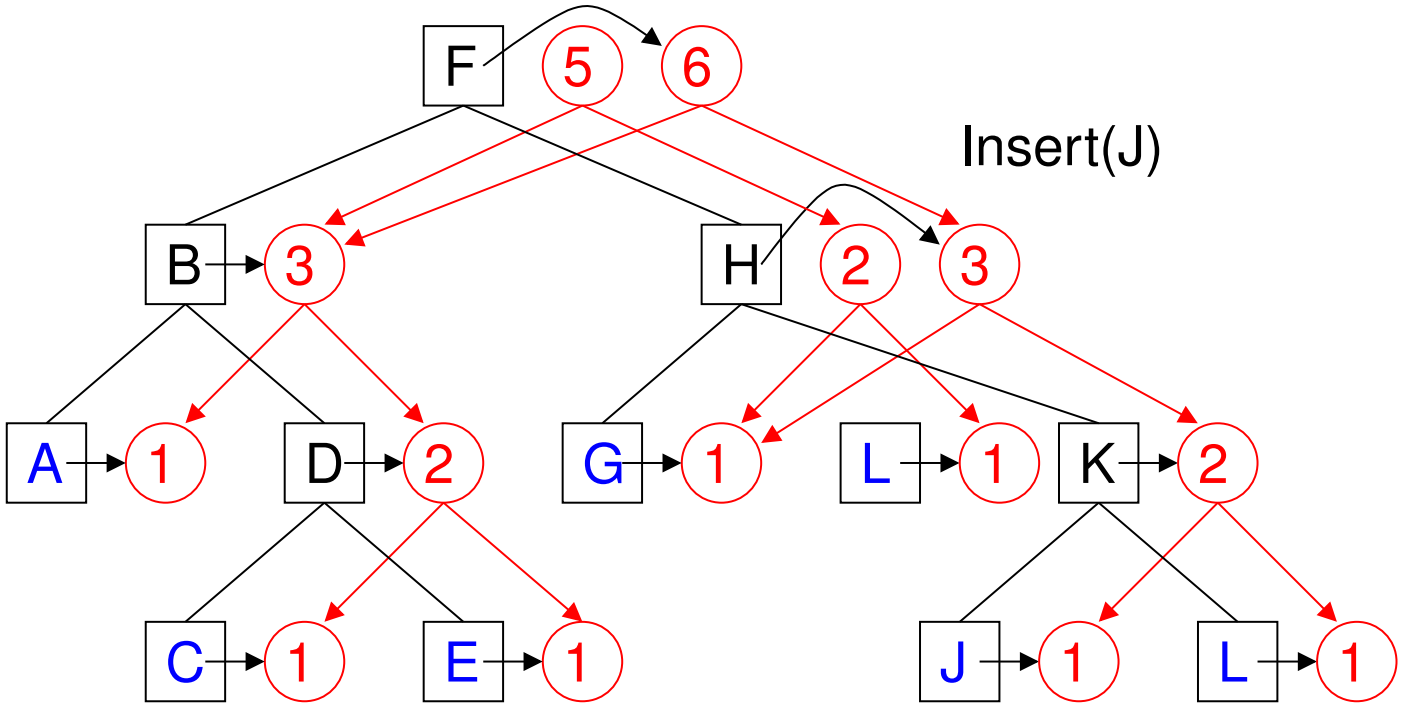
Key Idea



Node stores pointer to current version of augmented field.

- Old versions can still be used by queries in progress.
- Pointers between versions provide consistent view.

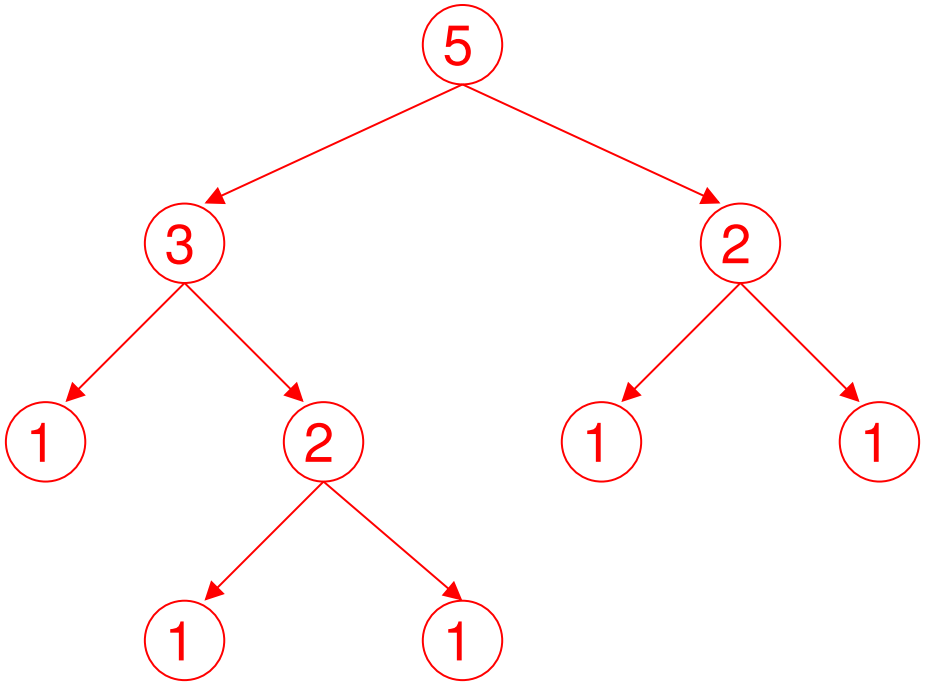
Key Idea



Node stores pointer to current version of augmented field.

- Old versions can still be used by queries in progress.
- Pointers between versions provide consistent view.

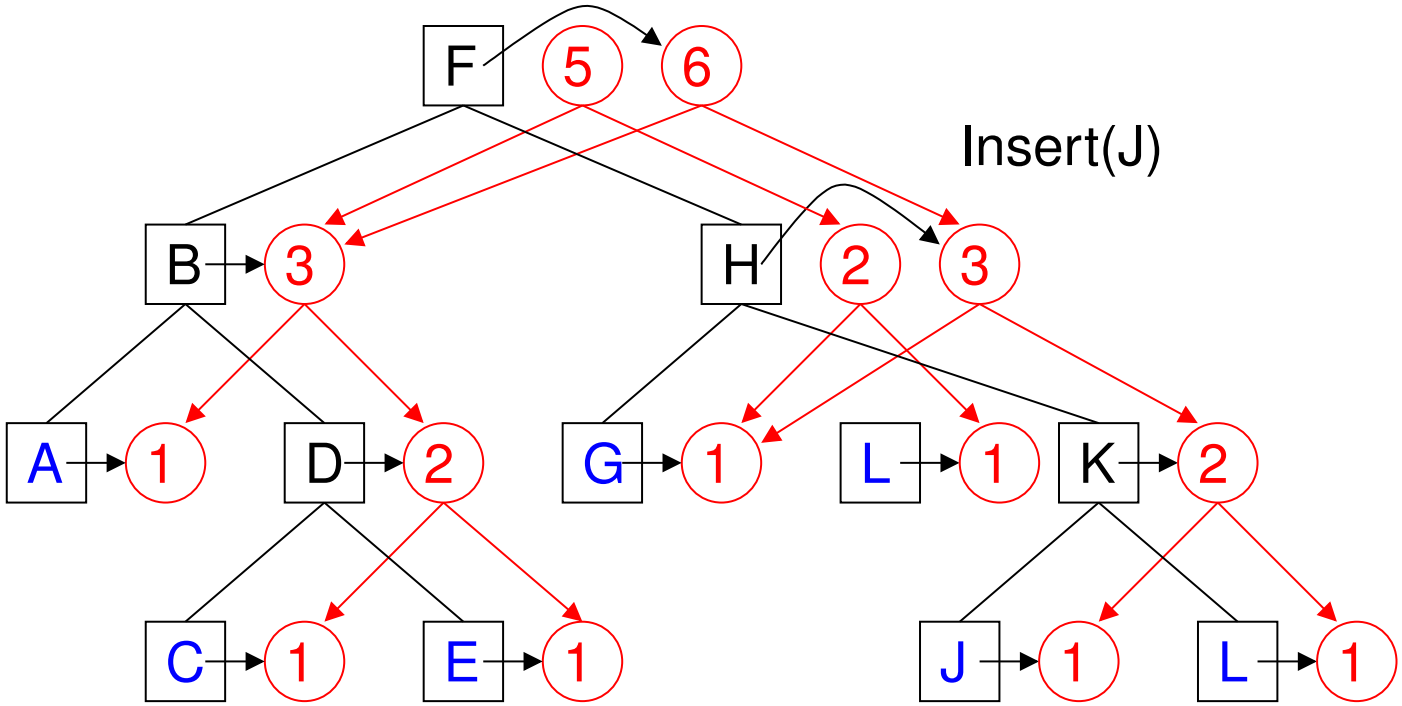
Key Idea



Accessing root's version provides snapshot of version tree.

- Versions also store keys to direct searches.
- Supports any sequential query operation.
- Old versions are unreachable when no longer needed.

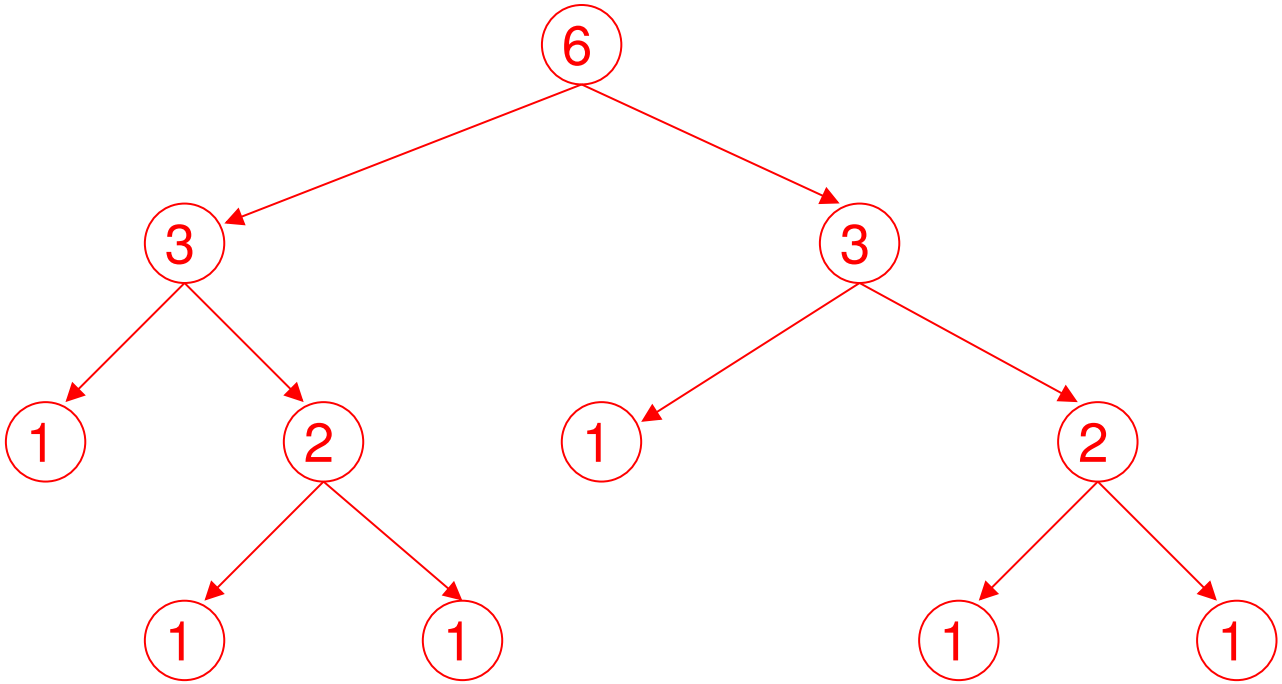
Key Idea



Accessing root's version provides snapshot of version tree.

- Versions also store keys to direct searches.
- Supports any sequential query operation.
- Old versions are unreachable when no longer needed.

Key Idea



Accessing root's version provides snapshot of version tree.

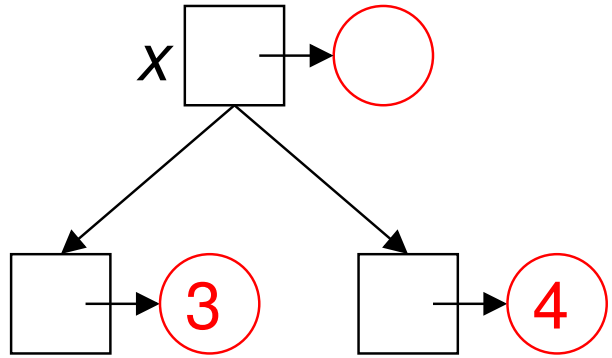
- Versions also store keys to direct searches.
- Supports any sequential query operation.
- Old versions are unreachable when no longer needed.

Updating Versions

After Insert or Delete, propagate changes up to root.

Propagate

for each node x on path to root
    do (at most) twice
        // Refresh x's version
        create new version v
        $v.left \leftarrow x.left.version$
        $v.right \leftarrow x.right.version$
        compute contents of v
        CAS $x.version$ to v

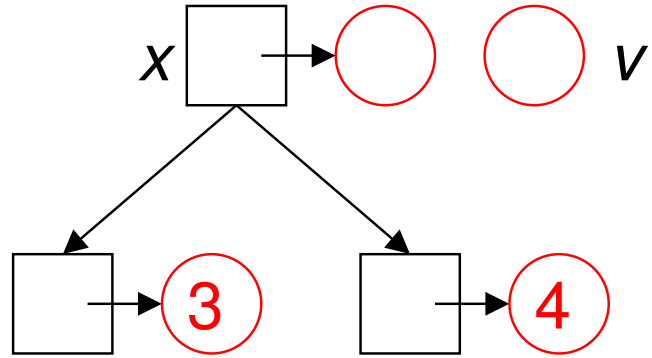


Updating Versions

After Insert or Delete, propagate changes up to root.

Propagate

for each node x on path to root
do (at most) twice
// Refresh x's version
• create new version v
$v.left \leftarrow x.left.version$
$v.right \leftarrow x.right.version$
compute contents of v
CAS $x.version$ to v

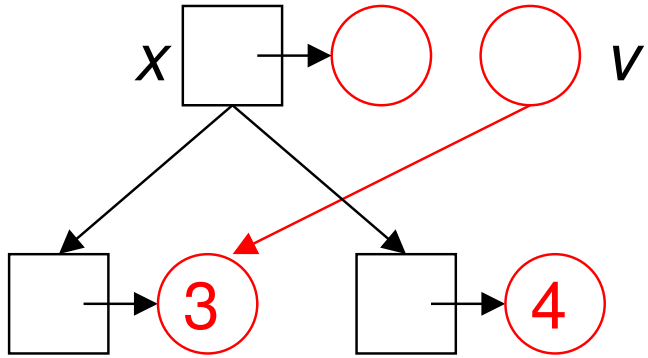


Updating Versions

After Insert or Delete, propagate changes up to root.

Propagate

for each node x on path to root
    do (at most) twice
        // Refresh x's version
        create new version v
        $v.left \leftarrow x.left.version$
        $v.right \leftarrow x.right.version$
        compute contents of v
        CAS $x.version$ to v

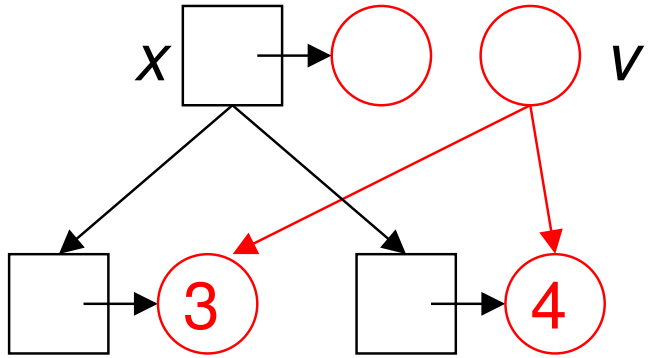


Updating Versions

After Insert or Delete, propagate changes up to root.

Propagate

for each node x on path to root
    do (at most) twice
        // Refresh x's version
        create new version v
        $v.left \leftarrow x.left.version$
        $v.right \leftarrow x.right.version$
        compute contents of v
        CAS $x.version$ to v

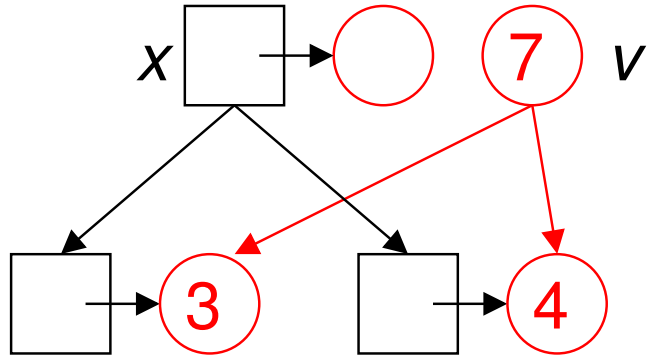


Updating Versions

After Insert or Delete, propagate changes up to root.

Propagate

for each node x on path to root
    do (at most) twice
        // Refresh x's version
        create new version v
        $v.left \leftarrow x.left.version$
        $v.right \leftarrow x.right.version$
        compute contents of v
        CAS $x.version$ to v

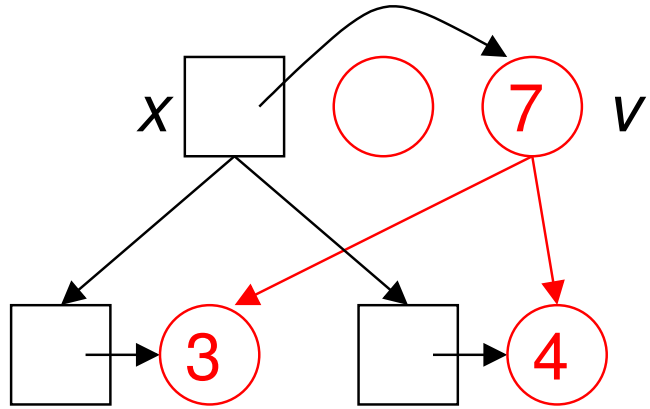


Updating Versions

After Insert or Delete, propagate changes up to root.

Propagate

for each node x on path to root
do (at most) twice
// Refresh x's version
create new version v
$v.left \leftarrow x.left.version$
$v.right \leftarrow x.right.version$
compute contents of v
• CAS $x.version$ to v

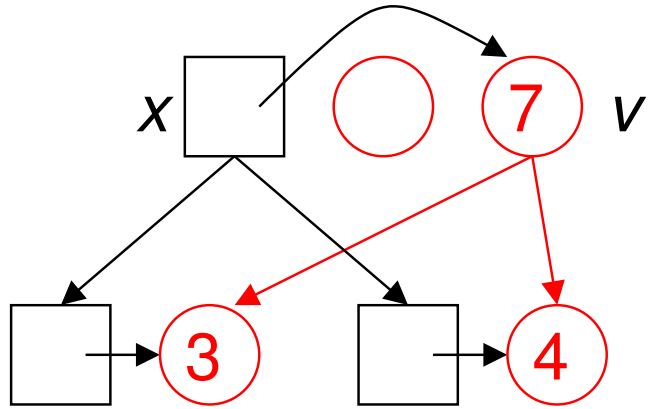


Updating Versions

After Insert or Delete, propagate changes up to root.

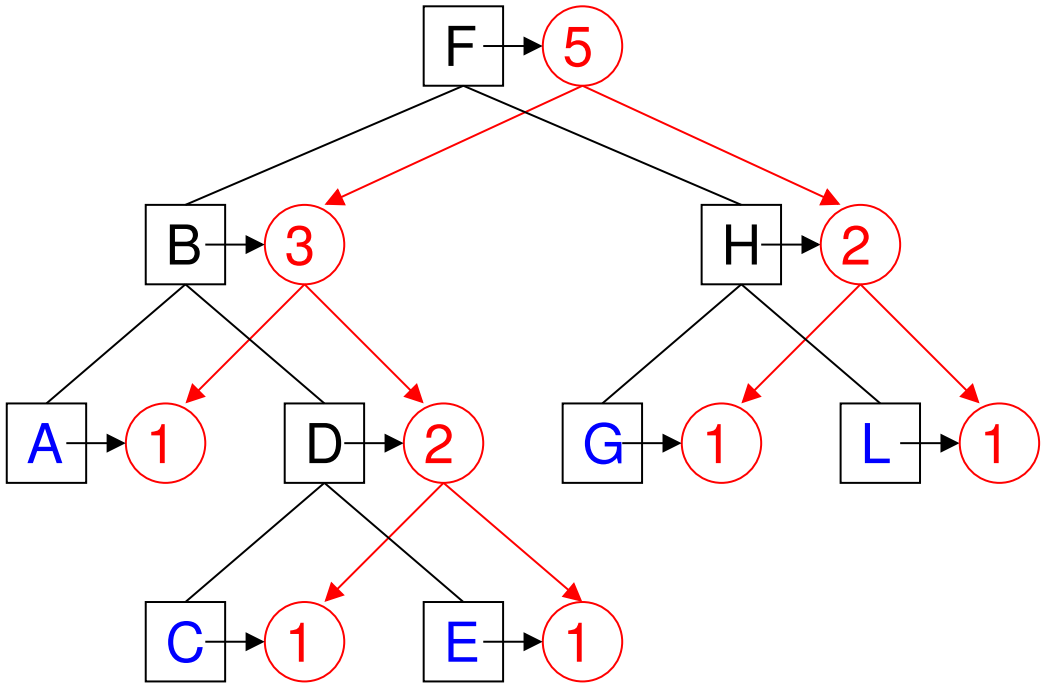
Propagate

for each node x on path to root
    do (at most) twice
        // Refresh x's version
        create new version v
        $v.left \leftarrow x.left.version$
        $v.right \leftarrow x.right.version$
        compute contents of v
        CAS $x.version$ to v

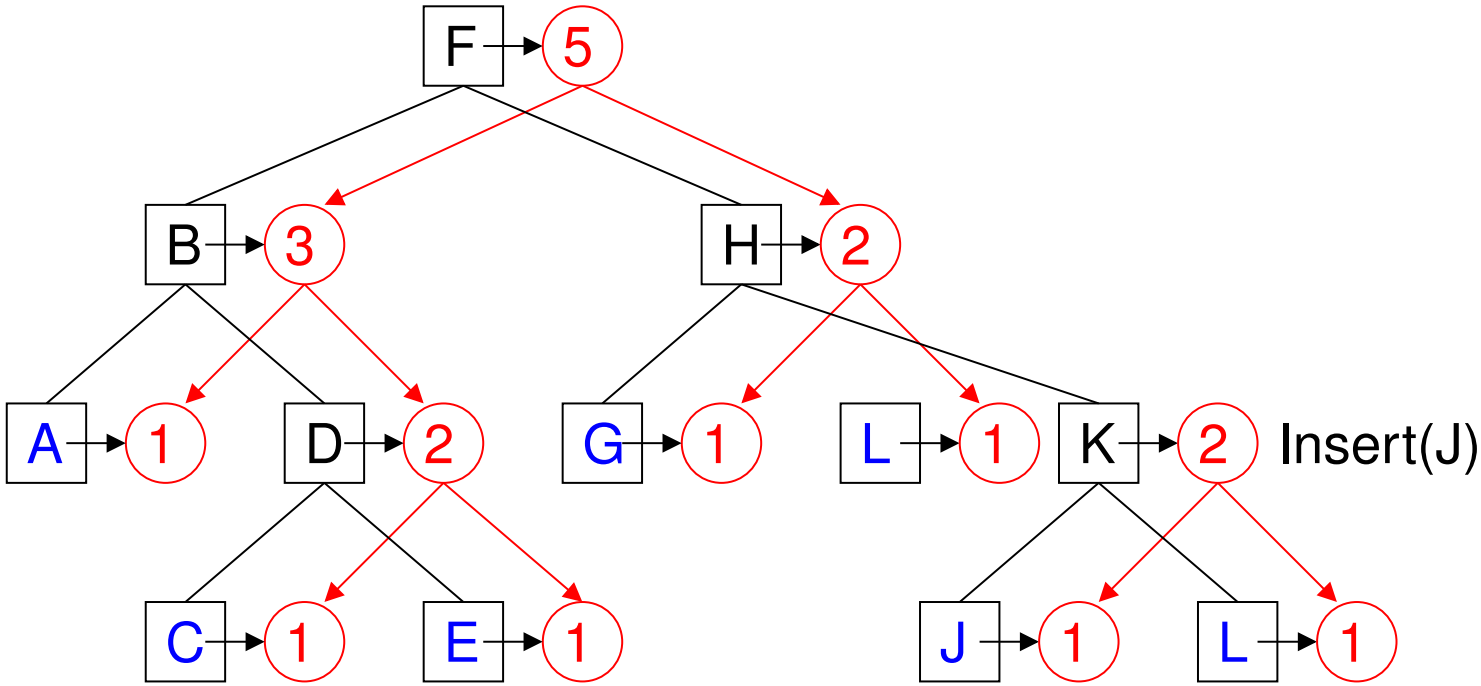


Contents of version never change once it is attached to tree.
Propagation is wait-free.

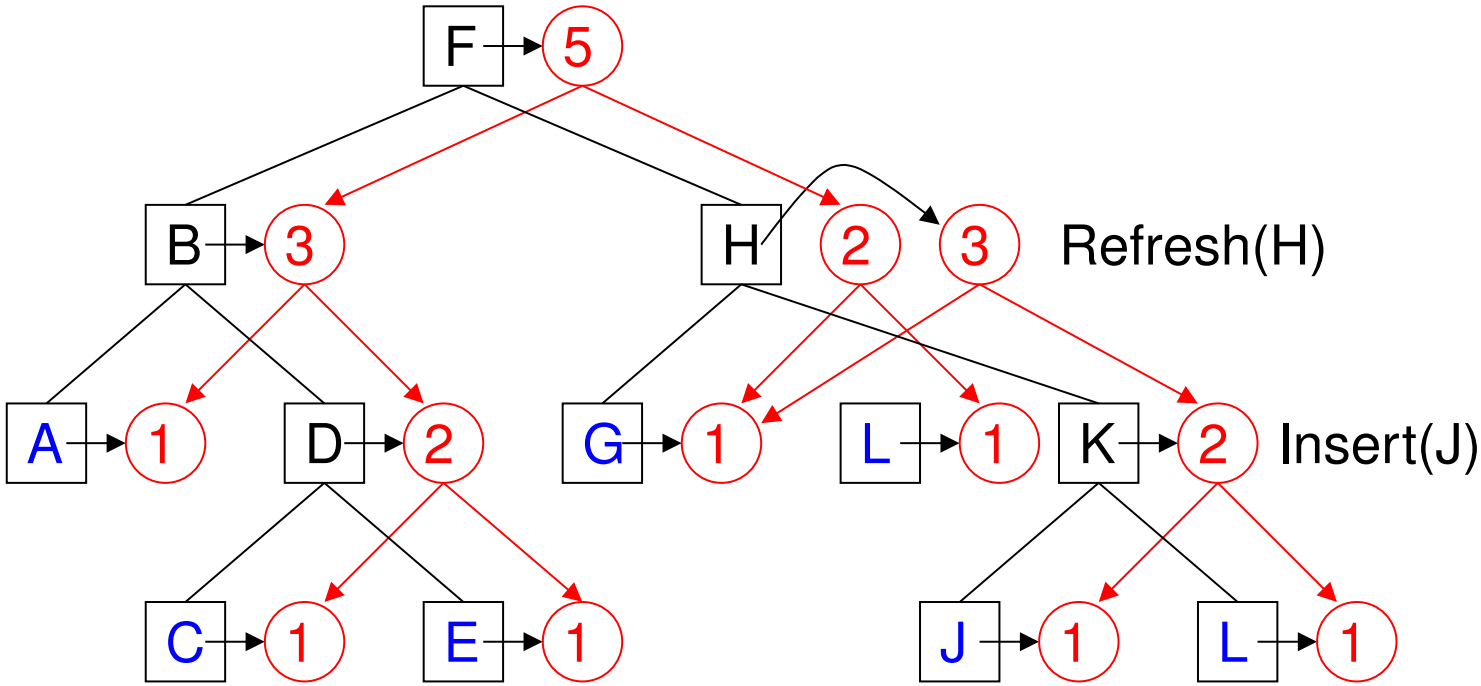
Propagating an Insert



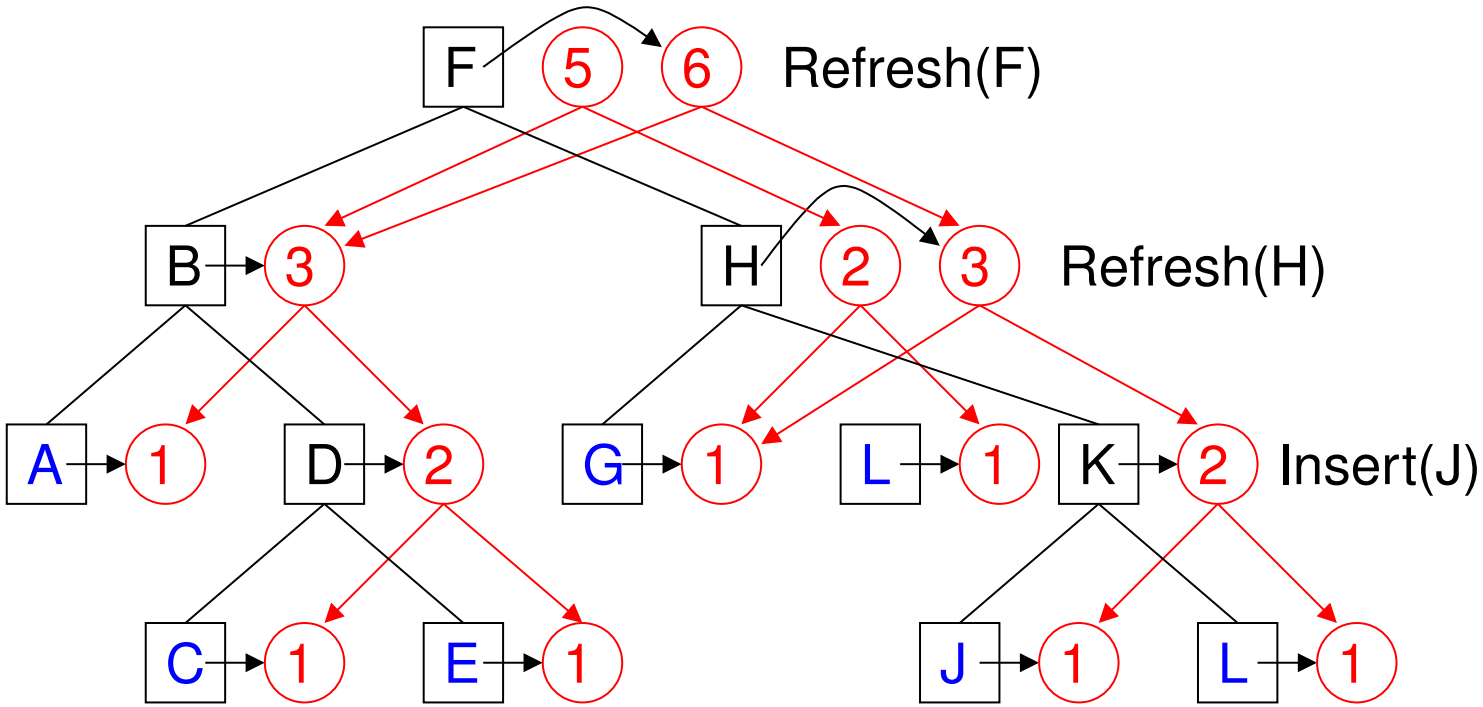
Propagating an Insert



Propagating an Insert



Propagating an Insert



Double Refresh

Refresh on each node x uses CAS to update x's version.

What if the CAS fails?

Try again.

What if the CAS fails again?

Stop; someone else's refresh has propagated your change to x.

Cooperation and Contention

- Updates are propagated cooperatively
- One change can propagate many operations together
- All update operations perform CAS on root
  BUT not all have to succeed

Double Refresh

Refresh on each node x uses CAS to update x's version.

What if the CAS fails?

Try again.

What if the CAS fails again?

Stop; someone else's refresh has propagated your change to x.

Cooperation and Contention

- Updates are propagated cooperatively
- One change can propagate many operations together
- All update operations perform CAS on root
  BUT not all have to succeed

Double Refresh

Refresh on each node x uses CAS to update x's version.

What if the CAS fails?

Try again.

What if the CAS fails again?

Stop; someone else's refresh has propagated your change to x.

Cooperation and Contention

- Updates are propagated cooperatively
- One change can propagate many operations together
- All update operations perform CAS on root
  BUT not all have to succeed

Double Refresh

Refresh on each node x uses CAS to update x's version.

What if the CAS fails?

Try again.

What if the CAS fails again?

Stop; someone else's refresh has propagated your change to x.

Cooperation and Contention

- Updates are propagated cooperatively
- One change can propagate many operations together
- All update operations perform CAS on root
  BUT not all have to succeed

Double Refresh

Refresh on each node x uses CAS to update x's version.

What if the CAS fails?

Try again.

What if the CAS fails again?

Stop; someone else's refresh has propagated your change to x.

Cooperation and Contention

- Updates are propagated cooperatively
- One change can propagate many operations together
- All update operations perform CAS on root
  BUT not all have to succeed

Algorithm

Insert or Delete operation

Run original algorithm to perform update
Refresh each ancestor (at most) twice

Adds $O(height)$ to step complexity of updates.

Query operation

Read $Root.version$ to get snapshot of version tree
Run standard *sequential* algorithm on that snapshot

Step complexity same as *sequential* query time.

YORK UNIVERSITÉ UNIVERSITY

Algorithm

Insert or Delete operation

Run original algorithm to perform update
Refresh each ancestor (at most) twice

Adds $O(height)$ to step complexity of updates.

Query operation

Read *Root.version* to get snapshot of version tree
Run standard *sequential* algorithm on that snapshot

Step complexity same as *sequential* query time.

YORK UNIVERSITÉ UNIVERSITY

Algorithm

Insert or Delete operation

Run original algorithm to perform update
Refresh each ancestor (at most) twice

Adds $O(height)$ to step complexity of updates.

Query operation

Read $Root.version$ to get snapshot of version tree
Run standard *sequential* algorithm on that snapshot

Step complexity same as *sequential* query time.

YORK UNIVERSITÉ UNIVERSITY

Algorithm

Insert or Delete operation

Run original algorithm to perform update
Refresh each ancestor (at most) twice

Adds $O(height)$ to step complexity of updates.

Query operation

Read $Root.version$ to get snapshot of version tree
Run standard *sequential* algorithm on that snapshot

Step complexity same as *sequential* query time.

YORK UNIVERSITÉ UNIVERSITY

Proving Correctness

Challenges

- Updates on original tree, queries on version tree
- The two trees might look very different

(Strong) Linearizability

- Define *arrival point* of each update at each node on its leaf-to-root path.
- Invariant: tree rooted at $x.version$ reflects all operations that have arrived at x (done in order of their arrival points).
- Linearization point = arrival point at the root.
- Show that propagate ensures all operations arrive at root.

YORK UNIVERSITÉ UNIVERSITY

Proving Correctness

Challenges

- Updates on original tree, queries on version tree
- The two trees might look very different

(Strong) Linearizability

- Define *arrival point* of each update at each node on its leaf-to-root path.
- Invariant: tree rooted at $x.version$ reflects all operations that have arrived at x (done in order of their arrival points).
- Linearization point = arrival point at the root.
- Show that propagate ensures all operations arrive at root.

YORK UNIVERSITÉ UNIVERSITY

Proving Correctness

Challenges

- Updates on original tree, queries on version tree
- The two trees might look very different

(Strong) Linearizability

- Define *arrival point* of each update at each node on its leaf-to-root path.
- Invariant: tree rooted at $x.version$ reflects all operations that have arrived at x (done in order of their arrival points).
- Linearization point = arrival point at the root.
- Show that propagate ensures all operations arrive at root.

YORK UNIVERSITÉ UNIVERSITY

Proving Correctness

Challenges

- Updates on original tree, queries on version tree
- The two trees might look very different

(Strong) Linearizability

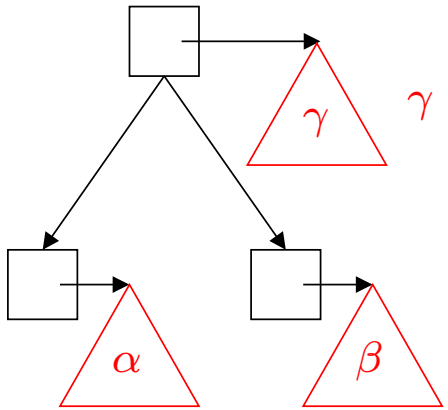
- Define *arrival point* of each update at each node on its leaf-to-root path.
- Invariant: tree rooted at $x.version$ reflects all operations that have arrived at x (done in order of their arrival points).
- Linearization point = arrival point at the root.
- Show that propagate ensures all operations arrive at root.

YORK UNIVERSITÉ UNIVERSITY

Improving Query Time

We can improve query time for unbalanced trees.

Simply plug in a different augmentation:
x.*version* is a red-black tree of all keys in *x*'s subtree



$\gamma = \text{Join}(\alpha, \beta)$

Use path copying to avoid changing α, β

Join can be done in $O(\log n)$ time.
Update takes additional $O(\textit{height} \cdot \log n)$ steps.
Queries take $O(\log n)$ steps, even if tree height is $\Theta(n)$.

YORK UNIVERSITÉ UNIVERSITY

Related Work

- Lock-free BST augmented with size
  [Kokorin, Alistarh, Aksenov IPDPS 2024]
  - Each operation must join a queue at each node and help all those ahead.
  - Not generalizable to other augmentations.
  - $\Omega((\#processes) \cdot height)$ steps per operation.
- Lock-based tree augmentation [Sela, Petrank DISC 2024]
- Much work on taking snapshots of shared data structures
  - They are more complicated, and have slower queries
  - Those based on multiversioning have complex GC
- Double refresh has been used in other ways
  [Afek, Dauber, Touitou 1995]

YORK UNIVERSITÉ UNIVERSITY

Related Work

- Lock-free BST augmented with size
  [Kokorin, Alistarh, Aksenov IPDPS 2024]
  - Each operation must join a queue at each node and help all those ahead.
  - Not generalizable to other augmentations.
  - $\Omega((\#processes) \cdot height)$ steps per operation.
- Lock-based tree augmentation [Sela, Petrank DISC 2024]
- Much work on taking snapshots of shared data structures
  - They are more complicated, and have slower queries
  - Those based on multiversioning have complex GC
- Double refresh has been used in other ways
  [Afek, Dauber, Touitou 1995]

Related Work

- Lock-free BST augmented with size
  [Kokorin, Alistarh, Aksenov IPDPS 2024]
  - Each operation must join a queue at each node and help all those ahead.
  - Not generalizable to other augmentations.
  - $\Omega((\#processes) \cdot height)$ steps per operation.
- Lock-based tree augmentation [Sela, Petrank DISC 2024]
- Much work on taking snapshots of shared data structures
  - They are more complicated, and have slower queries
  - Those based on multiversioning have complex GC
- Double refresh has been used in other ways
  [Afek, Dauber, Touitou 1995]

YORK UNIVERSITÉ UNIVERSITY

Related Work

- Lock-free BST augmented with size
  [Kokorin, Alistarh, Aksenov IPDPS 2024]
  - Each operation must join a queue at each node and help all those ahead.
  - Not generalizable to other augmentations.
  - $\Omega((\#processes) \cdot height)$ steps per operation.
- Lock-based tree augmentation [Sela, Petrank DISC 2024]
- Much work on taking snapshots of shared data structures
  - They are more complicated, and have slower queries
  - Those based on multiversioning have complex GC
- Double refresh has been used in other ways
  [Afek, Dauber, Touitou 1995]

YORK UNIVERSITÉ UNIVERSITY

Conclusion

Our scheme for augmenting concurrent trees

- is simple to implement
- works for any augmentation
- adds $O(height)$ to step complexity of updates
- preserves lock- or wait-freedom of updates
- has wait-free, fast queries
- supports simple snapshots

YORK UNIVERSITÉ UNIVERSITY