Non-blocking Binary Search Trees

Faith Ellen, University of Toronto

Panagiota Fatourou, ICS FORTH & University of Crete

Eric Ruppert, York University

Franck van Breugel, York University

July 26, 2010

State of the Art

The Java standard library has several non-blocking data structures, but no search trees.

"You might wonder why this doesn't use some kind of search tree instead . . . . The reason is that there are no known efficient lock-free insertion and deletion algorithms for search trees."

Doug Lea in java.util.concurrent.ConcurrentSkipListMap

Non-Blocking Data Structures

Non-blocking: some operation makes progress.

- Studied for 20+ years
- Universal constructions [1988–present]
  Disadvantage: inefficient
- Array-based structures [1990–2005]
  snapshots, stacks, queues
- List-based structures [1995–2005]
  singly-linked lists, stacks, queues, skip lists
- A few others [1995–present]
  union-find, ...

Non-Blocking Data Structures

Non-blocking: some operation makes progress.

- Studied for 20+ years
- Universal constructions [1988–present]
  Disadvantage: inefficient
- Array-based structures [1990–2005]
  snapshots, stacks, queues
- List-based structures [1995–2005]
  singly-linked lists, stacks, queues, skip lists
- A few others [1995–present]
  union-find, ...

Non-Blocking Data Structures

Non-blocking: some operation makes progress.

- Studied for 20+ years
- Universal constructions [1988–present]
  Disadvantage: inefficient
- Array-based structures [1990–2005]
  snapshots, stacks, queues
- List-based structures [1995–2005]
  singly-linked lists, stacks, queues, skip lists
- A few others [1995–present]
  union-find, ...

Non-Blocking Data Structures

Non-blocking: some operation makes progress.

- Studied for 20+ years
- Universal constructions [1988–present]
  Disadvantage: inefficient
- Array-based structures [1990–2005]
  snapshots, stacks, queues
- List-based structures [1995–2005]
  singly-linked lists, stacks, queues, skip lists
- A few others [1995–present]
  union-find, ...

Non-Blocking Data Structures

Non-blocking: some operation makes progress.

- Studied for 20+ years
- Universal constructions [1988–present]
  Disadvantage: inefficient
- Array-based structures [1990–2005]
  snapshots, stacks, queues
- List-based structures [1995–2005]
  singly-linked lists, stacks, queues, skip lists
- A few others [1995–present]
  union-find, ...

Prior Work on Concurrent Search Trees

- Many lock-based implementations [1978–present]
- Valois outlined how his linked lists might generalize to BSTs [1995]
  - complicated and lacks detail
- Non-blocking BST [Fraser 2003]
  - uses 8-word CAS
- Bender et al. outlined how their lock-based cache-oblivious B-trees might be made non-blocking [2005]
  - lacks details and proof of correctness

New Result

A non-blocking implementation of BSTs from single-word CAS.

Some properties:

- Conceptually simple
- Fast searches
- Concurrent updates to different parts of tree do not conflict
- Technique seems generalizable
- Experiments show good performance

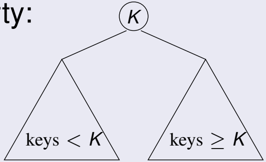
Model

- Asynchronous
- Crash failures allowed
- Shared memory with single-word compare-and-swap
- Linearizable

Leaf-oriented BST

Definition

- One leaf for each key in set
- Internal nodes used only for routing
- Each internal node has exactly 2 children
- BST property:

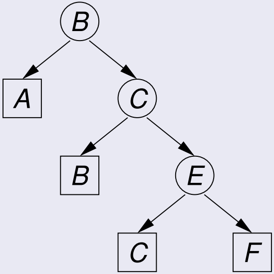


Advantages of Leaf-Oriented Trees

- Deletions much simpler
- Average depth only slightly higher

Example

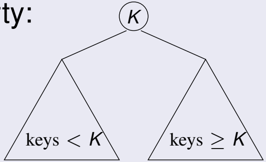
Leaf-oriented BST storing key set $\{A, B, C, F\}$



Leaf-oriented BST

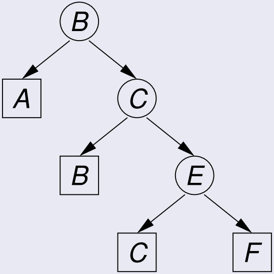
Definition

- One leaf for each key in set
- Internal nodes used only for routing
- Each internal node has exactly 2 children
- BST property:



Example

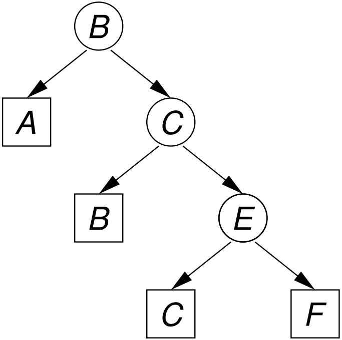
Leaf-oriented BST storing key set $\{A, B, C, F\}$



Advantages of Leaf-Oriented Trees

- Deletions much simpler
- Average depth only slightly higher

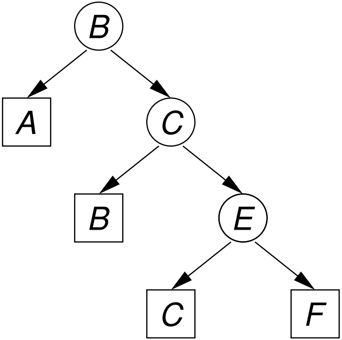
Insertion (non-concurrent version)



Insert(D)

1. Search for D
2. Remember leaf and its parent
3. Create new leaf, replacement leaf, and one internal node
4. Swing pointer

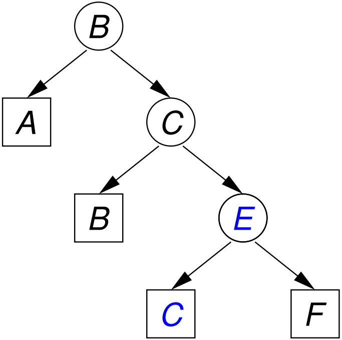
Insertion (non-concurrent version)



Insert(D)

1. Search for D
2. Remember leaf and its parent
3. Create new leaf, replacement leaf, and one internal node
4. Swing pointer

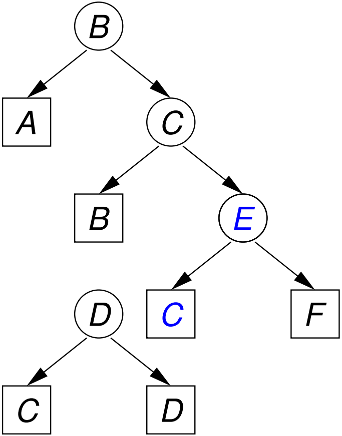
Insertion (non-concurrent version)



Insert(D)

1. Search for D
2. Remember leaf and its parent
3. Create new leaf, replacement leaf, and one internal node
4. Swing pointer

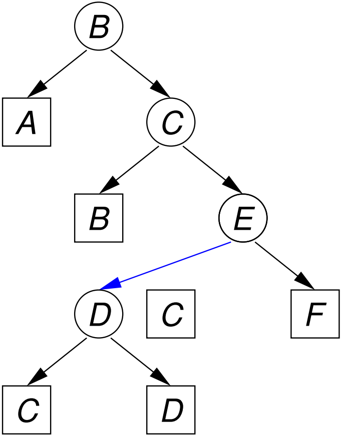
Insertion (non-concurrent version)



Insert(D)

1. Search for D
2. Remember leaf and its parent
3. Create new leaf, replacement leaf, and one internal node
4. Swing pointer

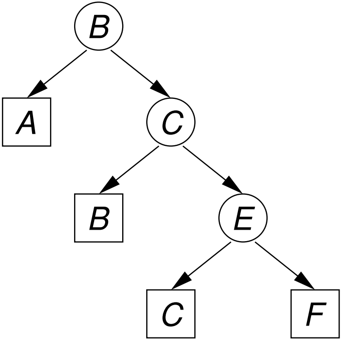
Insertion (non-concurrent version)



Insert(D)

1. Search for D
2. Remember leaf and its parent
3. Create new leaf, replacement leaf, and one internal node
4. Swing pointer

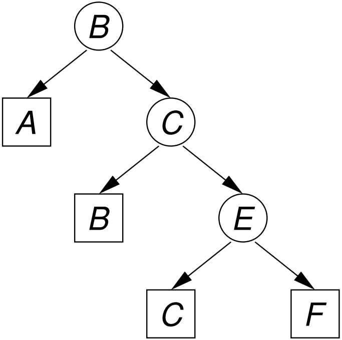
Deletion (non-concurrent version)



Delete(C)

1. Search for C
2. Remember leaf, its parent and grandparent
3. Swing pointer

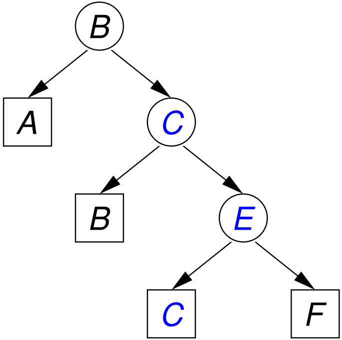
Deletion (non-concurrent version)



Delete(C)

1. Search for C
2. Remember leaf, its parent and grandparent
3. Swing pointer

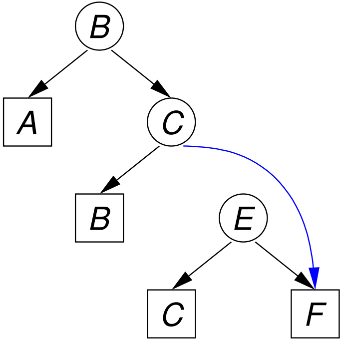
Deletion (non-concurrent version)



Delete(C)

1. Search for C
2. Remember leaf, its parent and grandparent
3. Swing pointer

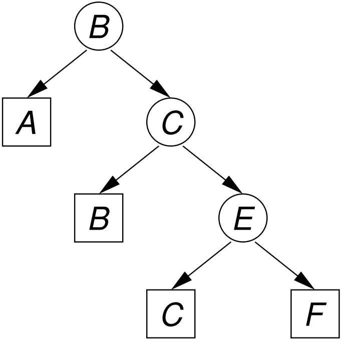
Deletion (non-concurrent version)



Delete(C)

1. Search for C
2. Remember leaf, its parent and grandparent
3. Swing pointer

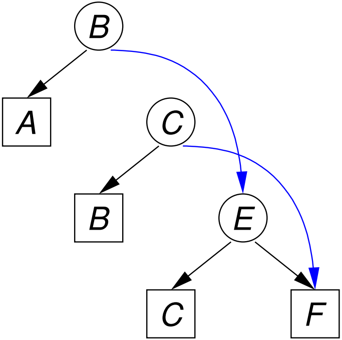
Challenges of Concurrency (1)



Concurrent Delete(B) and Delete(C).

$\Rightarrow$ C is still reachable from the root!

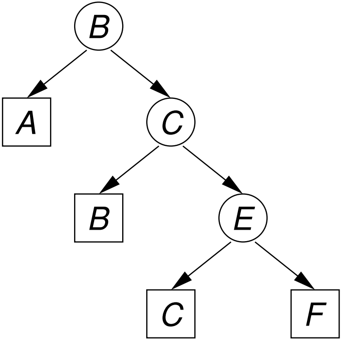
Challenges of Concurrency (1)



Concurrent Delete(B) and Delete(C).

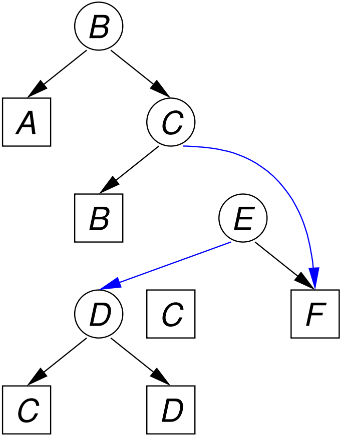
$\Rightarrow$ C is still reachable from the root!

Challenges of Concurrency (2)



Concurrent Delete(C) and Insert(D).

$\Rightarrow$ D is not reachable from the root!



Concurrent Delete(C) and Insert(D).

$\Rightarrow$ D is not reachable from the root!

Coordination Required

Crucial problem: A node's child pointer is changed while the node is being removed from the tree.

Solution: Updates to the same part of the tree must coordinate.

Desirable Properties of Coordination Scheme

- Avoid exclusive-access locks
- Maintain invariant that tree is always a BST
- Allow searches to pass unhindered
- Make updates as local as possible
- Algorithmic simplicity

Coordination Required

Crucial problem: A node's child pointer is changed while the node is being removed from the tree.

Solution: Updates to the same part of the tree must coordinate.

Desirable Properties of Coordination Scheme

- Avoid exclusive-access locks
- Maintain invariant that tree is always a BST
- Allow searches to pass unhindered
- Make updates as local as possible
- Algorithmic simplicity

Flags and Marks

An internal node can be either flagged or marked (but not both). Status is changed using CAS.

Flag

Indicates that an update is changing a child pointer.

- Before changing an internal node x's child pointer, flag x.
- Unflag x after its child pointer has been changed.

Mark

Indicates an internal node has been (or soon will be) removed from the tree.

- Before removing an internal node, mark it.
- Node remains marked forever.

Flags and Marks

An internal node can be either flagged or marked (but not both).
Status is changed using CAS.

Flag

Indicates that an update is changing a child pointer.

- Before changing an internal node x's child pointer, flag x.
- Unflag x after its child pointer has been changed.

Mark

Indicates an internal node has been (or soon will be) removed from the tree.

- Before removing an internal node, mark it.
- Node remains marked forever.

Flags and Marks

An internal node can be either flagged or marked (but not both). Status is changed using CAS.

Flag

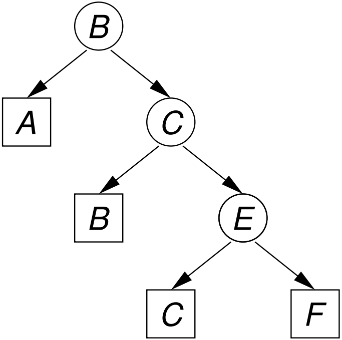
Indicates that an update is changing a child pointer.

- Before changing an internal node x's child pointer, flag x.
- Unflag x after its child pointer has been changed.

Mark

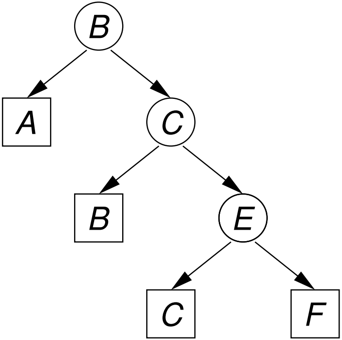
Indicates an internal node has been (or soon will be) removed from the tree.

- Before removing an internal node, mark it.
- Node remains marked forever.



Insert(D)

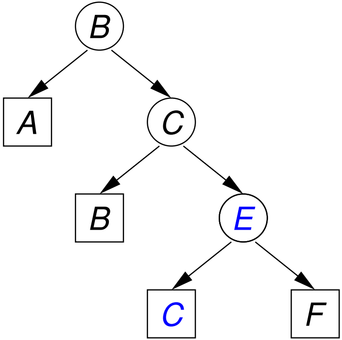
1. Search for D
2. Remember leaf and its parent
3. Create three new nodes
4. Flag parent (if this fails, retry from scratch)
5. Swing pointer (using CAS)
6. Unflag parent



Insert(D)

1. Search for D
2. Remember leaf and its parent
3. Create three new nodes
4. Flag parent (if this fails, retry from scratch)
5. Swing pointer (using CAS)
6. Unflag parent

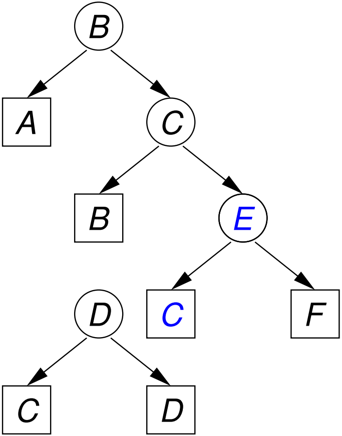
Insertion Algorithm



Insert(D)

1. Search for D
2. Remember leaf and its parent
3. Create three new nodes
4. Flag parent (if this fails, retry from scratch)
5. Swing pointer (using CAS)
6. Unflag parent

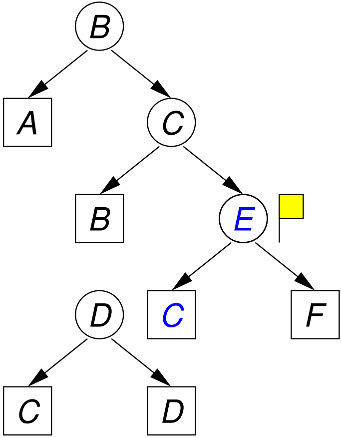
Insertion Algorithm



Insert(D)

1. Search for D
2. Remember leaf and its parent
3. Create three new nodes
4. Flag parent (if this fails, retry from scratch)
5. Swing pointer (using CAS)
6. Unflag parent

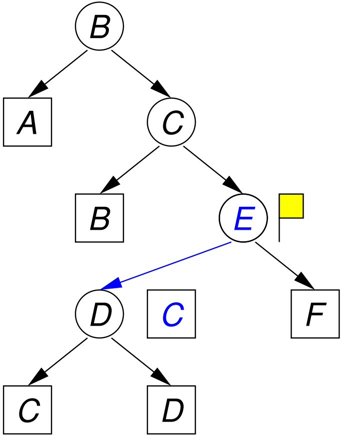
Insertion Algorithm



Insert(D)

1. Search for D
2. Remember leaf and its parent
3. Create three new nodes
4. Flag parent (if this fails, retry from scratch)
5. Swing pointer (using CAS)
6. Unflag parent

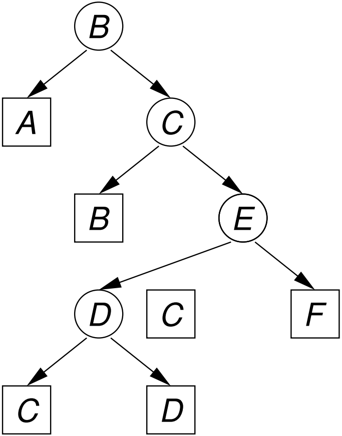
Insertion Algorithm



Insert(D)

1. Search for D
2. Remember leaf and its parent
3. Create three new nodes
4. Flag parent (if this fails, retry from scratch)
5. Swing pointer (using CAS)
6. Unflag parent

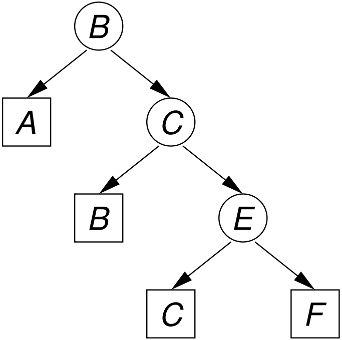
Insertion Algorithm



Insert(D)

1. Search for D
2. Remember leaf and its parent
3. Create three new nodes
4. Flag parent (if this fails, retry from scratch)
5. Swing pointer (using CAS)
6. Unflag parent

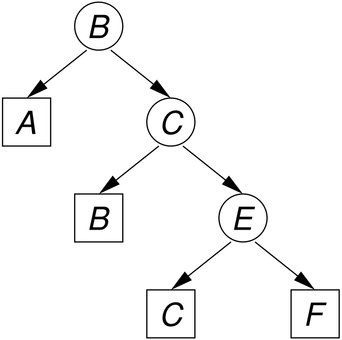
Deletion Algorithm



Delete(C)

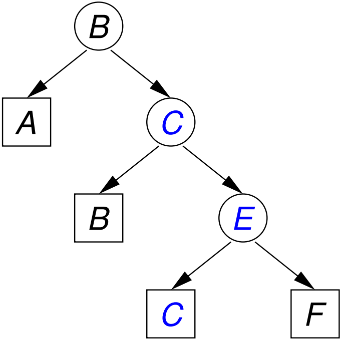
1. Search for C
2. Remember leaf, its parent and grandparent
3. Flag grandparent (if this fails, retry from scratch)
4. Mark parent (if this fails, unflag grandparent and retry from scratch)
5. Swing pointer (using CAS)
6. Unflag grandparent

Deletion Algorithm



Delete(C)

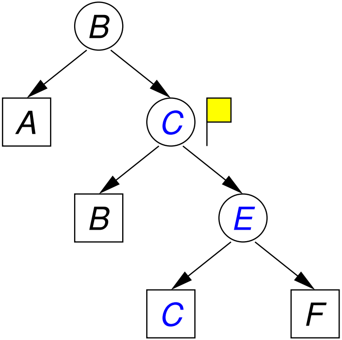
1. Search for C
2. Remember leaf, its parent and grandparent
3. Flag grandparent (if this fails, retry from scratch)
4. Mark parent (if this fails, unflag grandparent and retry from scratch)
5. Swing pointer (using CAS)
6. Unflag grandparent



Delete(C)

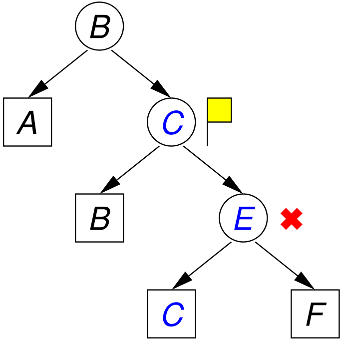
1. Search for C
2. Remember leaf, its parent and grandparent
3. Flag grandparent (if this fails, retry from scratch)
4. Mark parent (if this fails, unflag grandparent and retry from scratch)
5. Swing pointer (using CAS)
6. Unflag grandparent

Deletion Algorithm



Delete(C)

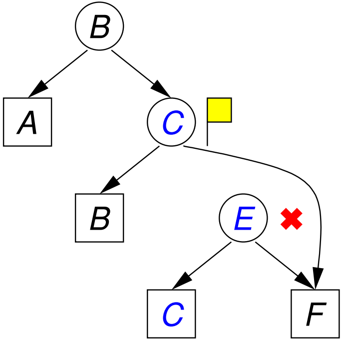
1. Search for C
2. Remember leaf, its parent and grandparent
3. Flag grandparent (if this fails, retry from scratch)
4. Mark parent (if this fails, unflag grandparent and retry from scratch)
5. Swing pointer (using CAS)
6. Unflag grandparent



Delete(C)

1. Search for C
2. Remember leaf, its parent and grandparent
3. Flag grandparent (if this fails, retry from scratch)
4. Mark parent (if this fails, unflag grandparent and retry from scratch)
5. Swing pointer (using CAS)
6. Unflag grandparent

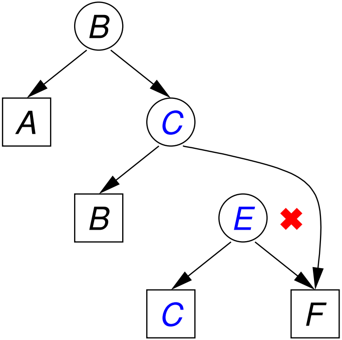
Deletion Algorithm



Delete(C)

1. Search for C
2. Remember leaf, its parent and grandparent
3. Flag grandparent (if this fails, retry from scratch)
4. Mark parent (if this fails, unflag grandparent and retry from scratch)
5. Swing pointer (using CAS)
6. Unflag grandparent

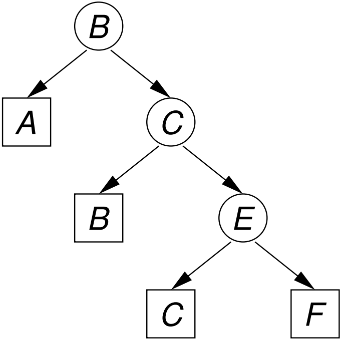
Deletion Algorithm



Delete(C)

1. Search for C
2. Remember leaf, its parent and grandparent
3. Flag grandparent (if this fails, retry from scratch)
4. Mark parent (if this fails, unflag grandparent and retry from scratch)
5. Swing pointer (using CAS)
6. Unflag grandparent

Conflicting Deletions Now Work



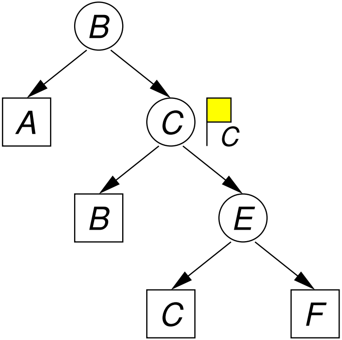
Concurrent Delete(B) and Delete(C)

Case I: Delete(C)'s flag succeeds.

$\Rightarrow$ Even if Delete(B)'s flag succeeds, its mark will fail.

$\Rightarrow$ Delete(C) will complete
Delete(B) will retry

Conflicting Deletions Now Work



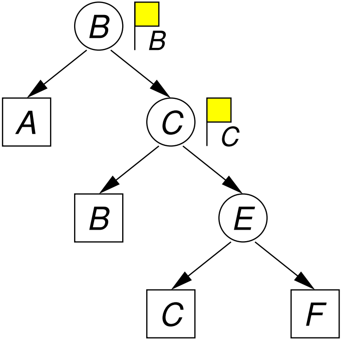
Concurrent Delete(B) and Delete(C)

Case I: Delete(C)'s flag succeeds.

⇒ Even if Delete(B)'s flag succeeds, its mark will fail.

⇒ Delete(C) will complete
Delete(B) will retry

Conflicting Deletions Now Work

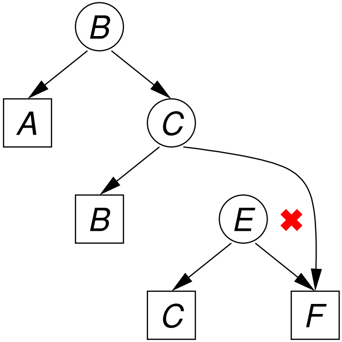


Concurrent Delete(B) and Delete(C)

Case I: Delete(C)'s flag succeeds.

$\Rightarrow$ Even if Delete(B)'s flag succeeds, its mark will fail.

$\Rightarrow$ Delete(C) will complete
Delete(B) will retry



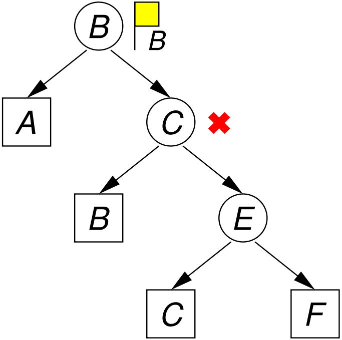
Concurrent Delete(B) and Delete(C)

Case I: Delete(C)'s flag succeeds.

$\Rightarrow$ Even if Delete(B)'s flag succeeds, its mark will fail.

$\Rightarrow$ Delete(C) will complete
Delete(B) will retry

Conflicting Deletions Now Work



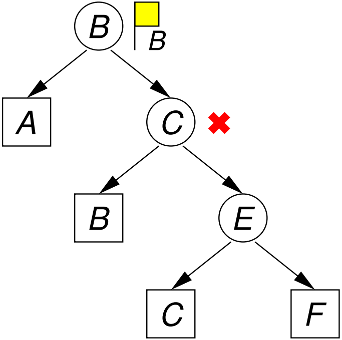
Concurrent Delete(B) and Delete(C)

Case II: Delete(B)'s flag and mark succeed.

$\Rightarrow$ Delete(C)'s flag fails.

$\Rightarrow$ Delete (B) will complete
Delete(C) will retry

Conflicting Deletions Now Work



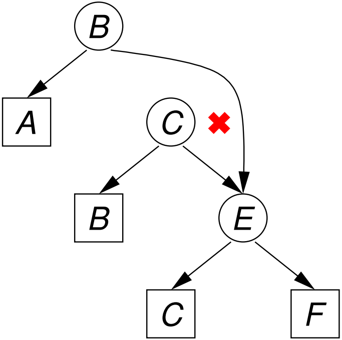
Concurrent Delete(B) and Delete(C)

Case II: Delete(B)'s flag and mark succeed.

$\Rightarrow$ Delete(C)'s flag fails.

$\Rightarrow$ Delete (B) will complete
Delete(C) will retry

Conflicting Deletions Now Work



Concurrent Delete(B) and Delete(C)

Case II: Delete(B)'s flag and mark succeed.

$\Rightarrow$ Delete(C)'s flag fails.

$\Rightarrow$ Delete (B) will complete
Delete(C) will retry

Locks

Can think of flag or mark as a lock on the child pointers of a node.

- Flag corresponds to temporary ownership of lock.
- Mark corresponds to permanent ownership of lock.

Remark

Easier version of these ideas were used for singly-linked lists. Locking two child pointers with one flag or mark is harder.

Remark

Each update needs only one or two locks, searches need none. (Previous lock-based BSTs use more locks.)

Locks

Can think of flag or mark as a lock on the child pointers of a node.

- Flag corresponds to temporary ownership of lock.
- Mark corresponds to permanent ownership of lock.

Remark

Easier version of these ideas were used for singly-linked lists. Locking two child pointers with one flag or mark is harder.

Remark

Each update needs only one or two locks, searches need none. (Previous lock-based BSTs use more locks.)

Locks

Can think of flag or mark as a lock on the child pointers of a node.

- Flag corresponds to temporary ownership of lock.
- Mark corresponds to permanent ownership of lock.

Remark

Easier version of these ideas were used for singly-linked lists. Locking two child pointers with one flag or mark is harder.

Remark

Each update needs only one or two locks, searches need none. (Previous lock-based BSTs use more locks.)

Wait a second . . .

We want the data structure to be non-blocking!

Whenever "locking" a node, leave a key under the doormat.

A flag or mark is actually a pointer to a small record that tells a process how to help the original operation.

If an operation fails to acquire a lock, it helps complete the update that holds the lock before retrying.

Thus, locks are owned by operations, not processes.

Some similarities to Barnes's cooperative technique.

Wait a second . . .

We want the data structure to be non-blocking!

Whenever "locking" a node, leave a key under the doormat.

A flag or mark is actually a pointer to a small record that tells a process how to help the original operation.

If an operation fails to acquire a lock, it helps complete the update that holds the lock before retrying.

Thus, locks are owned by operations, not processes.

Some similarities to Barnes's cooperative technique.

Wait a second . . .

We want the data structure to be non-blocking!

Whenever “locking” a node, leave a key under the doormat.

A flag or mark is actually a pointer to a small record that tells a process how to help the original operation.

If an operation fails to acquire a lock, it helps complete the update that holds the lock before retrying.

Thus, locks are owned by operations, not processes.

Some similarities to Barnes’s cooperative technique.

Searching

Searches just traverse edges of the BST until reaching a leaf.

They can ignore flags and marks.

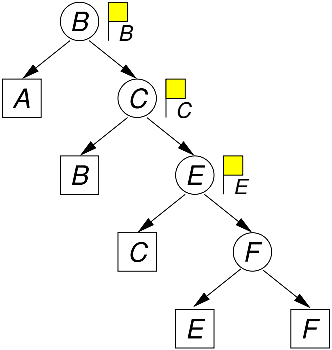
Can prove by induction that each node visited by a Search(K) was on the search path for K at some time during the Search.

Progress

Goal: Show data structure is non-blocking (some operation completes).

- If an Insert successfully flags, it finishes.
- If a Delete successfully flags and marks, it finishes.
- If updates stop happening, searches must finish.

One CAS fails only if another succeeds.
$\Rightarrow$ A successful CAS guarantees progress, except for a Delete's flag.



A Delete may flag, then fail to mark, then unflag to retry.

$\Rightarrow$ The Delete's changes may cause other CAS's to fail.

However, lowest Delete will make progress.

Some Details Omitted

The formal proof of correctness is surprisingly difficult (20 pages long).

See the Technical Report.

Further Work

- Balancing the tree
- Proving worst-case complexity bounds
- Can same approach yield (efficient) wait-free BSTs?
  (Or at least wait-free Finds?)
- Other data structures