Non-blocking Trees

TRANSFORM Summer School

Trevor Brown, University of Toronto
Faith Ellen, University of Toronto
Eric Ruppert, York University

June 10, 2013

YORK UNIVERSITÉ UNIVERSITY

Motivation

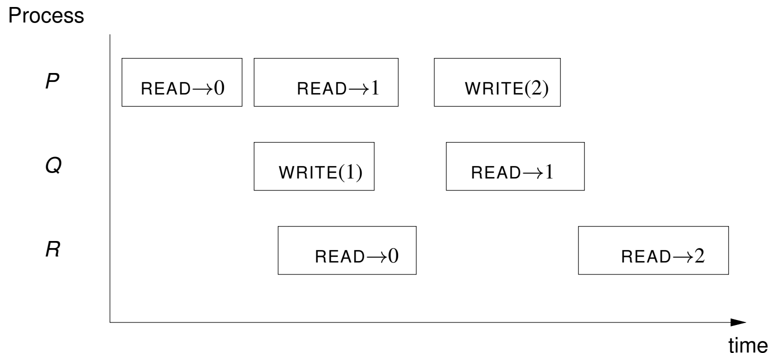
Goal: Concurrent Data Structures

Data structures that can be accessed concurrently by many processes are

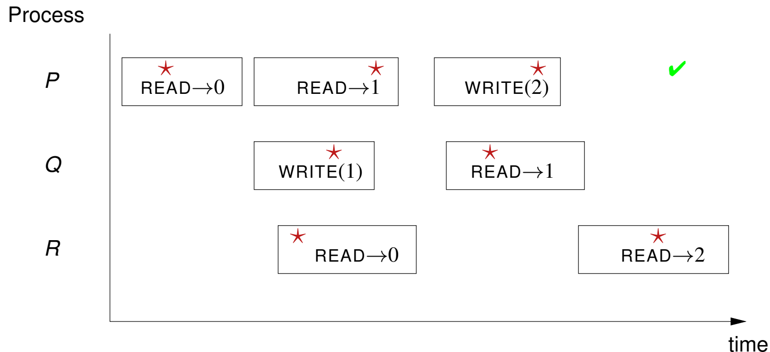
- important
- hard to design
- hard to prove correct

We focus on linearizable, non-blocking data structures.

Linearizability



Linearizability



Linearizability

An object is linearizable if its operations seem to occur instantaneously.

More formally:

For every execution that uses the objects,
we can choose a $\star$ inside each operation's time interval so that all operations would return the same results if they were performed sequentially in the order of their $\star$s.

The $\star$ is called the linearization point of the operation.

Linearizability

An object is linearizable if its operations seem to occur instantaneously.

More formally:

For every execution that uses the objects,
we can choose a $\star$ inside each operation's time interval so that all operations would return the same results if they were performed sequentially in the order of their $\star$s.

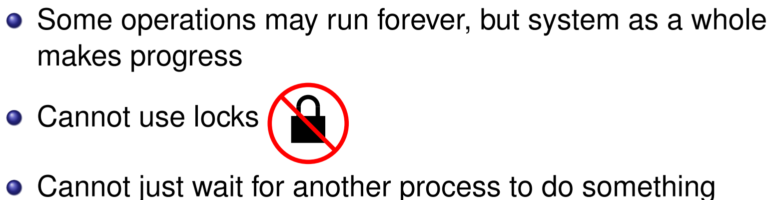
The $\star$ is called the linearization point of the operation.

Non-blocking

Non-blocking Progress Property

Some operation eventually completes, even if some processes crash.

- Some operations may run forever, but system as a whole makes progress
- Cannot use locks



- Cannot just wait for another process to do something

Compare&Swap

CAS is a low-level universal primitive supported by hardware.
Operates on a single word.

CAS(X, *old*, *new*)
    if $X = old$ then
        $X \leftarrow new$
        return *old*
    else
        return X
end CAS

CAS performs this code atomically.

YORK UNIVERSITÉ UNIVERSITY

Compare&Swap

CAS is a low-level universal primitive supported by hardware.
Operates on a single word.

CAS(X, *old*, *new*)
  if $X = old$ then
    $X \leftarrow new$
    return ~~*old*~~ true
  else
    return ~~X~~ false
end CAS

CAS performs this code atomically.

The Problem

How can we build complex data structures that are linearizable and non-blocking on a machine that provides CAS?

Software Transactional Memory

Transactional memory

Enclose each data structure operation in an atomic transaction.

Pros:

- simple to design
- simple to prove

Cons:

- can be less efficient than hand-crafted data structures
- coarse-grained transactions limit concurrency

Right solution for “casual” data structure designers.

Implementations Directly from CAS

Handcrafted non-blocking implementations directly from CAS.

Pros:

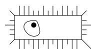
- allows good efficiency
- allows high degree of concurrency

Cons:

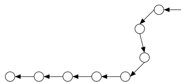
- hard to get implementation (provably) right

Right solution for designing libraries of data structures.

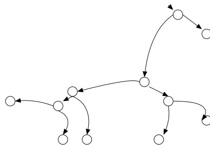
The Evolution of Non-blocking Data Structures



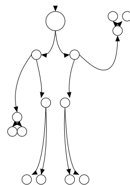
Proterozoic
ca. 1980



Paleozoic
ca. 1995



Mesozoic
2010



Cenozoic
2013

YORK UNIVERSITÉ UNIVERSITY

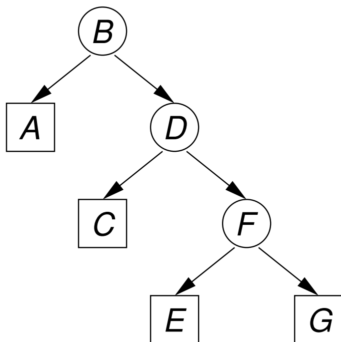
Why Is Evolution So Slow?

Key difficulty of implementing data structures from CAS:

- Data structure operations access several words atomically
- CAS operates only on a single word

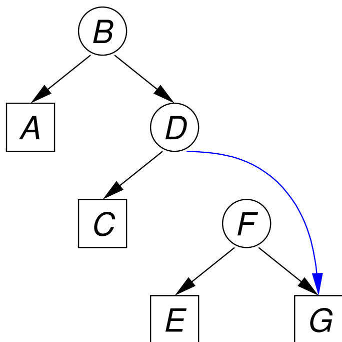
Example: Binary Search Tree

How to DELETE(E) from a leaf-oriented binary search tree



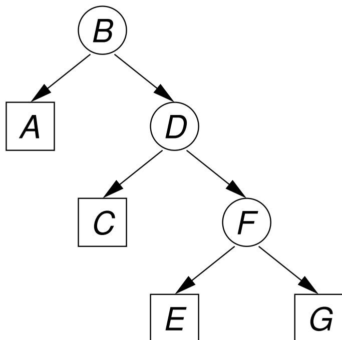
Example: Binary Search Tree

How to DELETE(E) from a leaf-oriented binary search tree



Example: Binary Search Tree

Problems arise with concurrent deletions of C and E.



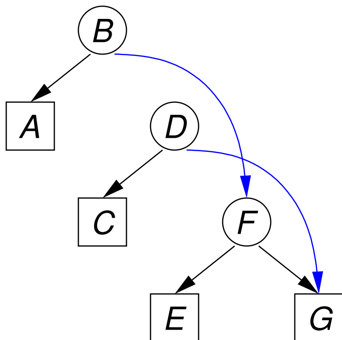
Leaf E is still reachable

DELETE(C) removes node D.
Should succeed only if D's child pointers are unchanged.

$\Rightarrow$ Need multi-word primitives.

Example: Binary Search Tree

Problems arise with concurrent deletions of C and E.



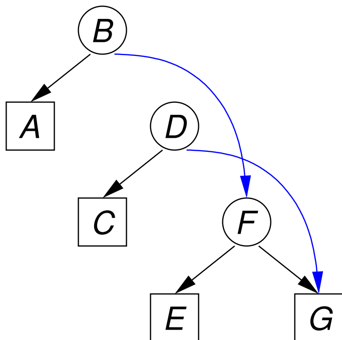
Leaf E is still reachable

DELETE(C) removes node D.
Should succeed only if D's child pointers are unchanged.

$\Rightarrow$ Need multi-word primitives.

Example: Binary Search Tree

Problems arise with concurrent deletions of C and E.



Leaf E is still reachable

DELETE(C) removes node D.
Should succeed only if D's child pointers are unchanged.

$\Rightarrow$ Need multi-word primitives.

Our Approach

Build “medium-level” primitives that can access multiple words.

- higher-level than CAS or LL/SC
- lower-level than full transactional memory

Advantages:

- General enough to be used in many data structures
- Specialized enough to create quite efficient implementations
- Modular proof of correctness: large parts can be reused

YORK UNIVERSITÉ UNIVERSITY

Our Primitives: LLX and SCX

Our Primitives

Our primitives extend load-link (LL) and store-conditional (SC).

LL/SC object

- stores a single word
- LL reads value stored
- SC(v) (store-conditional) writes v only if value has not changed since last LL by process performing SC.

YORK UNIVERSITÉ UNIVERSITY

Data Records

Our operations work on data records.

Each data record has

- some mutable fields (one word each)
- some immutable fields

Use a data record for some natural "unit" of a data structure

- node in a tree
- entry in a table

YORK UNIVERSITÉ UNIVERSITY

LLX and SCX

LLX(r) returns a snapshot of the mutable fields of *r*

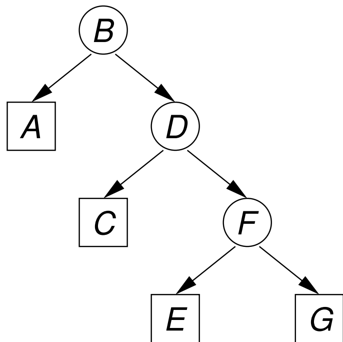
SCX(V, R, field, new) by process *p*

- writes value *new* into *field*,
  which is a mutable field of a data record in *V*
- finalizes all data records in *R*
- only if no record in *V* has changed since *p*'s LLX on it

After a data record is finalized, no further changes allowed.

Example: Deletion in BST

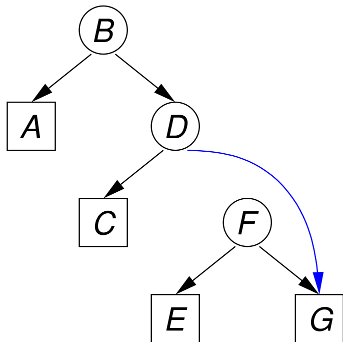
DELETE(E) using LLX and SCX.
Use one data record for each node.



1. LLX(D)
   $\rightarrow \langle D.left = C, D.right = F \rangle$
2. LLX(F)
   $\rightarrow \langle F.left = E, F.right = G \rangle$
3. SCX($\langle D, F \rangle, \langle F \rangle, D.right, G$)
   - changes $D.right$ to G
   - finalizes F
   - succeeds only if no changes since LLXs on D and F

Example: Deletion in BST

DELETE(E) using LLX and SCX.
Use one data record for each node.



1. LLX(D)
   $\rightarrow \langle D.left = C, D.right = F \rangle$
2. LLX(F)
   $\rightarrow \langle F.left = E, F.right = G \rangle$
3. SCX($\langle D, F \rangle, \langle F \rangle, D.right, G$)
   - changes $D.right$ to G
   - finalizes F
   - succeeds only if no changes since LLXs on D and F

Other Multi-word Primitives

Others have built medium-level multi-word primitives.

- Large LL/SC objects (Anderson Moir 1995, ...)
  $\Rightarrow$ unable to access multiple objects atomically
- Multi-word CAS (Israeli Rappoport 1994, ...)
  $\Rightarrow$ more general, less efficient
- Multi-word RMW (Afek Merritt Taubenfeld Touitou 1997, ...)
  $\Rightarrow$ even more general, less efficient
- *k*-compare-single-swap (Luchangco Moir Shavit 2009)
  $\Rightarrow$ lacks ability to finalize
  $\Rightarrow$ less efficient if data records are large
  $\Rightarrow$ implementation is only obstruction-free

YORK UNIVERSITÉ UNIVERSITY

Other Multi-word Primitives

Others have built medium-level multi-word primitives.

- Large LL/SC objects (Anderson Moir 1995, ...)
  $\Rightarrow$ unable to access multiple objects atomically
- Multi-word CAS (Israeli Rappoport 1994, ...)
  $\Rightarrow$ more general, less efficient
- Multi-word RMW (Afek Merritt Taubenfeld Touitou 1997, ...)
  $\Rightarrow$ even more general, less efficient
- *k*-compare-single-swap (Luchangco Moir Shavit 2009)
  $\Rightarrow$ lacks ability to finalize
  $\Rightarrow$ less efficient if data records are large
  $\Rightarrow$ implementation is only obstruction-free

YORK
UNIVERSITÉ
UNIVERSITY

Other Multi-word Primitives

Others have built medium-level multi-word primitives.

- Large LL/SC objects (Anderson Moir 1995, ...)
  $\Rightarrow$ unable to access multiple objects atomically
- Multi-word CAS (Israeli Rappoport 1994, ...)
  $\Rightarrow$ more general, less efficient
- Multi-word RMW (Afek Merritt Taubenfeld Touitou 1997, ...)
  $\Rightarrow$ even more general, less efficient
- *k*-compare-single-swap (Luchangco Moir Shavit 2009)
  $\Rightarrow$ lacks ability to finalize
  $\Rightarrow$ less efficient if data records are large
  $\Rightarrow$ implementation is only obstruction-free

YORK UNIVERSITÉ UNIVERSITY

Other Multi-word Primitives

Others have built medium-level multi-word primitives.

- Large LL/SC objects (Anderson Moir 1995, ...)
  $\Rightarrow$ unable to access multiple objects atomically
- Multi-word CAS (Israeli Rappoport 1994, ...)
  $\Rightarrow$ more general, less efficient
- Multi-word RMW (Afek Merritt Taubenfeld Touitou 1997, ...)
  $\Rightarrow$ even more general, less efficient
- k-compare-single-swap (Luchangco Moir Shavit 2009)
  $\Rightarrow$ lacks ability to finalize
  $\Rightarrow$ less efficient if data records are large
  $\Rightarrow$ implementation is only obstruction-free

More Detailed Specification: LLX

LLX(r) can return one of the following results.

- a snapshot of mutable fields of r
- FINALIZED (iff r has been finalized by an SCX)
- FAIL (in our implementation this happens only if a concurrent SCX accesses r)

We also allow reads of individual mutable fields.

More Detailed Specification: SCX

Before calling SCX(V, R, *field*, *new*)
must get a snapshot from an LLX(r) on each record r in V.

The last such LLX(r) is linked to the SCX.

If any r in V was changed since the linked LLX(r)
$\Rightarrow$ SCX returns FAIL.

Non-failed SCX sets *field* $\leftarrow$ *new* and finalizes records in R.

Spurious failures of SCX are allowed.

Progress Properties of LLX and SCX

Individual LLXs and SCXs are wait-free, but may fail.

- If LLXs and SCXs are performed infinitely often, they succeed infinitely often.
- If SCXs are performed infinitely often, they succeed infinitely often.

Also, if no overlap between V-sets of SCX's, all will succeed.

Progress Properties of LLX and SCX

Individual LLXs and SCXs are wait-free, but may fail.

- If LLXs and SCXs are performed infinitely often, they succeed infinitely often.
- If SCXs are performed infinitely often, they succeed infinitely often.

Also, if no overlap between V-sets of SCX's, all will succeed.

Progress Properties of LLX and SCX

Individual LLXs and SCXs are wait-free, but may fail.

- If LLXs and SCXs are performed infinitely often, they succeed infinitely often.
- If SCXs are performed infinitely often, they succeed infinitely often.

Also, if no overlap between V-sets of SCX's, all will succeed.

Implementing LLX and SCX from CAS

YORK UNIVERSITÉ UNIVERSITY

Key Idea

Lock-free Locks

- "Locks" on data records acquired by SCX operations
- If a record you need is locked by an SCX,
  you can help that SCX and release lock
- Finalized records remain permanently locked

Based on cooperative technique of
Turek et al. [1992] and Barnes [1993]

YORK U
UNIVERSITÉ
UNIVERSITY

SCX Records

Each SCX creates an SCX record.

An SCX record contains all information needed to help SCX:

- parameters of the SCX
- old values to be used for CAS steps
- information that monitors progress of SCX:
  - Is the SCX in progress, aborted or committed?
  - Have all locks succeeded?

YORK UNIVERSITÉ UNIVERSITY

Augmenting Data Records

Add two fields to each data record *r*:

- *info*: pointer to SCX record of last SCX that locked *r*
- *marked*: boolean used to finalize *r*

An SCX locks a data record *r* by CASing pointer to its SCX record into *r.info*

Expected value for CAS comes from LLX.
⇒ CAS succeeds only if info field unchanged since LLX

Augmenting Data Records

Add two fields to each data record r:

- *info*: pointer to SCX record of last SCX that locked r
- *marked*: boolean used to finalize r

An SCX locks a data record r by CASing pointer to its SCX record into $r.info$

Expected value for CAS comes from LLX.
$\Rightarrow$ CAS succeeds only if info field unchanged since LLX

YORK UNIVERSITÉ UNIVERSITY

LLX algorithm

```
LLX(r)
    read r.info and state fields of SCX record r.info points to
    if r is not locked then
        read mutable fields of r
        re-read r.info
        if r.info unchanged then return values read as a snapshot
    if r is finalized then
        return FINALIZED
    else
        HELP SCX operation in progress at r (if any)
        return FAIL
end LLX
```

SCX algorithm

```
SCX(V, R, field, new)
    create SCX record s
    for each r in V
        try to CAS s into r.info
        if r.info ≠ s then
            if s.locksSucceeded then
                return TRUE % Someone else finished the operation
            else
                s.state ← aborted
                return FALSE
    s.locksSucceeded ← TRUE
    r.marked ← TRUE for each data record in R
    CAS new into field
    s.state ← committed
    return TRUE
end SCX
```

SCX algorithm

~~SCX(V, R, *field*, *new*)~~ HELP(s)
    ~~create SCX record s~~
    for each r in V
        try to CAS s into $r.info$
        if $r.info \neq s$ then
            if $s.locksSucceeded$ then
                return TRUE % Someone else finished the operation
            else
                $s.state \leftarrow aborted$
                return FALSE
    $s.locksSucceeded \leftarrow$ TRUE
    $r.marked \leftarrow$ TRUE for each data record in R
    CAS *new* into *field*
    $s.state \leftarrow committed$
    return TRUE
end ~~SCX~~ HELP

Key Things to Prove

- Locking correctly protects all mutable fields of a record
- Inter-process coordination via state fields of SCX-records works
- All helpers of an SCX agree on outcome (aborted/committed)
- No ABA problems on fields accessed by CAS
- Progress properties

Caveat Auditor

A few details of process coordination have been omitted.
See paper.

All details of proof have been omitted.
The rigorous proof is quite involved (15 pages).

YORK UNIVERSITÉ UNIVERSITY

Caveat Auditor

A few details of process coordination have been omitted.
See paper.

All details of proof have been omitted.
The rigorous proof is quite involved (15 pages).

ABA Problem

We use CAS to implement LLX/SCX.

CAS is subject to ABA problem.

How to avoid ABA problem in implementation of LLX/SCX?

1. Counters
2. Wrappers
3. Punt problem to user

YORK UNIVERSITÉ UNIVERSITY

ABA Problem

We use CAS to implement LLX/SCX.

CAS is subject to ABA problem.

How to avoid ABA problem in implementation of LLX/SCX?

1. Counters → sometimes possible, ignoring wraparound
2. Wrappers → general, but slows down accesses
3. Punt problem to user→ most efficient, least convenient
   But it works out naturally for some data structures

Avoiding Livelock

Again, punt problem to user (but make it very easy to handle).

Constraint

After SCX's stop succeeding, eventually all new SCX's must have consistent order on V-sets.

Ensures that SCXs "lock" data records in consistent order
$\Rightarrow$ one will eventually succeed

Easy to satisfy, because you can ignore concurrency.

Complexity

With no contention:
SCX performs

- $k + 1$ CAS steps if it depends on k LLXs
- $f + 2$ writes if it finalizes f data records

LLX only performs reads.

With contention, LLXs and SCXs may have to help and/or retry.

Future work: Amortized complexity bounds with contention.

YORK U UNIVERSITÉ UNIVERSITY

Using LLX and SCX to Build Trees

YORK UNIVERSITÉ UNIVERSITY

Trees Everywhere

binary search tree

splay tree

Patricia trie

X-fast trie

binary heap

prefix tree

quadtree

B-tree

treap

YORK UNIVERSITÉ UNIVERSITY

Trees Everywhere

binary search tree

(2,3)-tree

van Emde Boas tree

splay tree

Patricia trie

Fibonacci tree

binary heap

X-fast trie

binomial tree

prefix tree

R-tree

quadtree

red-black tree

interval tree

B-tree

link/cut tree

treap

Trees Everywhere

binary search tree

AVL tree

(2,3)-tree

van Emde Boas tree

Cartesian tree

Fenwick tree

splay tree

B+tree

Patricia trie

Fibonacci tree

binary heap

X-fast trie

fusion tree

binomial tree

PQ tree

prefix tree

R-tree

quadtree

red-black tree

interval tree

range tree

B-tree

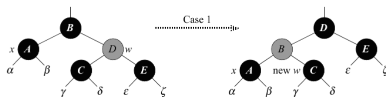
VP-tree

link/cut tree

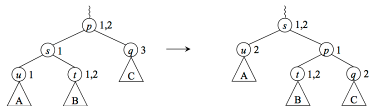
treap

Modifying a Tree Data Structure

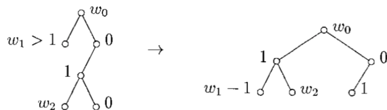
Tree data structures are often modified by local changes.



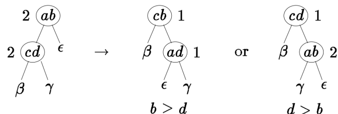
[Cormen et al. 2009]



[Haeupler, Sen, Tarjan 2009]



[Boyar, Fagerberg, Larsen 1997]

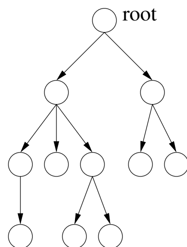


[Høyer 1995]

YORK UNIVERSITÉ UNIVERSITY

Goal

We give a template to implement down-trees using LLX/SCX.



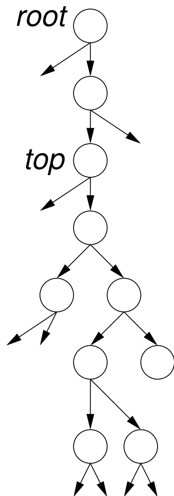
Yields tree implementations from single-word CAS that are

- linearizable,
- non-blocking,
- relatively simple to prove correct,
- and allow disjoint updates to succeed concurrently.

Representing the Tree Using Data Records

- Represent each node as a data record.
- Child pointers are mutable fields.
- Other data stored in node is immutable.
  (To change contents, must make a new copy of node.)

The Template

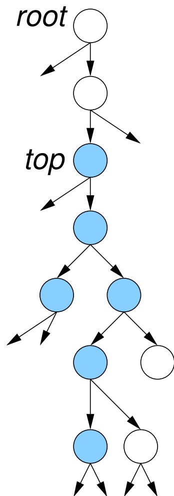


1. read child pointers, from *root* to some node *top*
2. LLX *top* and contiguous set of its descendants
3. Select subgraph *R* to replace and create replacement subgraph *N*
4. SCX to swing child pointer of *par*:
   - replaces *R* by *N*
   - and finalizes *R*
   - only if LLXed nodes unchanged

Requirements

- Children of *R* = Children of *N*
- Must LLX *par* and all nodes in *R*

The Template

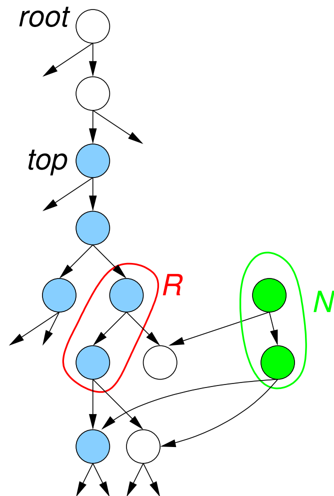


1. read child pointers, from *root* to some node *top*
2. LLX *top* and contiguous set of its descendants
3. Select subgraph *R* to replace and create replacement subgraph *N*
4. SCX to swing child pointer of *par*:
   - replaces *R* by *N*
   - and finalizes *R*
   - only if LLXed nodes unchanged

Requirements

- Children of *R* = Children of *N*
- Must LLX *par* and all nodes in *R*

The Template

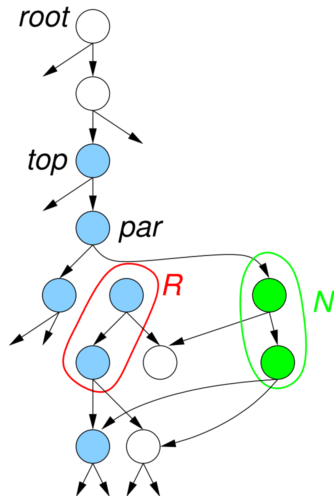


1. read child pointers, from *root* to some node *top*
2. LLX *top* and contiguous set of its descendants
3. Select subgraph *R* to replace and create replacement subgraph *N*
4. SCX to swing child pointer of *par*:
   - replaces *R* by *N*
   - and finalizes *R*
   - only if LLXed nodes unchanged

Requirements

- Children of *R* = Children of *N*
- Must LLX *par* and all nodes in *R*

The Template

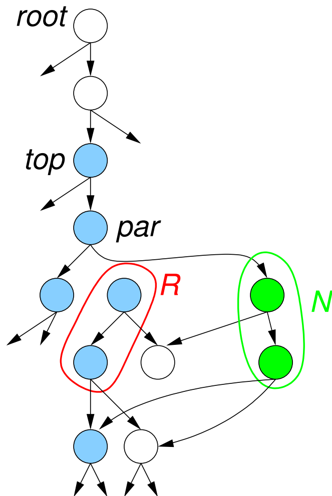


1. read child pointers, from *root* to some node *top*
2. LLX *top* and contiguous set of its descendants
3. Select subgraph *R* to replace and create replacement subgraph *N*
4. SCX to swing child pointer of *par*:
   - replaces *R* by *N*
   - and finalizes *R*
   - only if LLXed nodes unchanged

Requirements

- Children of *R* = Children of *N*
- Must LLX *par* and all nodes in *R*

The Template

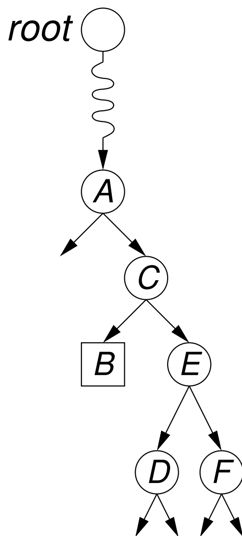


1. read child pointers, from *root* to some node *top*
2. LLX *top* and contiguous set of its descendants
3. Select subgraph *R* to replace and create replacement subgraph *N*
4. SCX to swing child pointer of *par*:
   - replaces *R* by *N*
   - and finalizes *R*
   - only if LLXed nodes unchanged

Requirements

- Children of *R* = Children of *N*
- Must LLX *par* and all nodes in *R*

Example: Deletion in Leaf-oriented BST

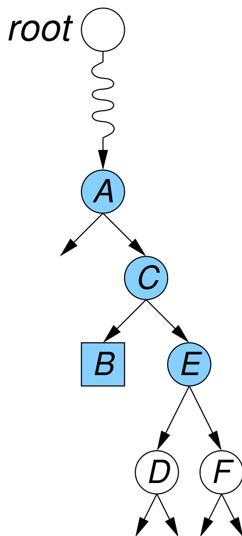


DELETE(B):

1. Read path from root to B
2. LLX A, C, B, E
3. Create new node E' to replace $R = \{B, C, E\}$
4. SCX($\langle A, B, C, E \rangle, \langle B, C, E \rangle, A.right, E'$)
   - changes A.right to E' and
   - finalizes B, C, E
   - only if A, B, C, E unchanged

YORK UNIVERSITÉ UNIVERSITY

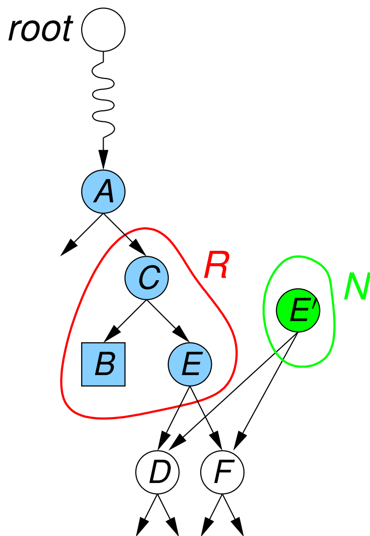
Example: Deletion in Leaf-oriented BST



DELETE(B):

1. Read path from root to B
2. LLX A, C, B, E
3. Create new node E' to replace $R = \{B, C, E\}$
4. SCX($\langle A, B, C, E \rangle, \langle B, C, E \rangle, A.right, E'$)
   - changes A.right to E' and
   - finalizes B, C, E
   - only if A, B, C, E unchanged

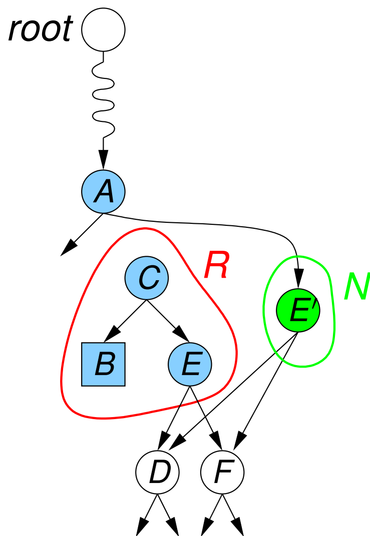
Example: Deletion in Leaf-oriented BST



DELETE(B):

1. Read path from root to B
2. LLX A, C, B, E
3. Create new node E' to replace $R = \{B, C, E\}$
4. SCX($\langle A, B, C, E\rangle$, $\langle B, C, E\rangle$, $A.right$, E')
   - changes A.right to E' and
   - finalizes B, C, E
   - only if A, B, C, E unchanged

Example: Deletion in Leaf-oriented BST



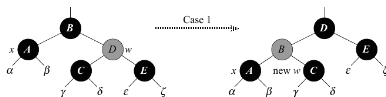
DELETE(B):

1. Read path from root to B
2. LLX A, C, B, E
3. Create new node E' to replace $R = \{B, C, E\}$
4. SCX($\langle A, B, C, E\rangle, \langle B, C, E\rangle, A.right, E'$)
   - changes A.right to E' and
   - finalizes B, C, E
   - only if A, B, C, E unchanged

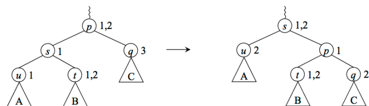
YORK UNIVERSITÉ UNIVERSITY

Tree Updates Made Easy

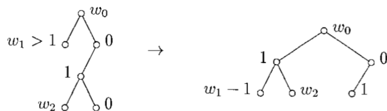
Can do any update to a down-tree atomically using template.



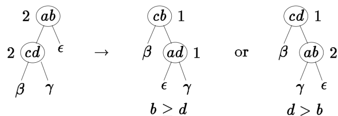
[Cormen et al. 2009]



[Haeupler, Sen, Tarjan 2009]



[Boyar, Fagerberg, Larsen 1997]



[Høyer 1995]

YORK UNIVERSITÉ UNIVERSITY

Linearizing the Tree Updates

Linearization point of each update: the successful SCX.

Invariant

Tree is same as it would if tree updates were done in order by linearization points.

Fast Queries

Some queries can be done quickly by reads only.

Example: SEARCH(k) **in a BST**

- Just read sequence of pointers from root to leaf.
- Ignore concurrent updates along the path

Leaf reached was in the tree at some time during the SEARCH
$\Rightarrow$ sufficient to linearize the SEARCH

No ABA Problem

When a child pointer changes, it points to a brand new node
$\Rightarrow$ No ABA problem
$\Rightarrow$ LLX/SCX are more efficient

What if I want to write a nil pointer into a child field?
Write a sentinel node that represents nil instead.

YORK U UNIVERSITÉ UNIVERSITY

No ABA Problem

When a child pointer changes, it points to a brand new node
$\Rightarrow$ No ABA problem
$\Rightarrow$ LLX/SCX are more efficient

What if I want to write a nil pointer into a child field?
Write a sentinel node that represents nil instead.

YORK UNIVERSITÉ UNIVERSITY

Avoiding Livelock

Order the nodes an SCX depends on in a consistent way.

For example, breadth-first search order, starting from *top*.

Key required property

Different SCX's have consistent orders
IF the tree has not changed.
(Don't have to worry about what happens if tree is changing.)

$\Rightarrow$ Sufficient to ensure SCX's succeed infinitely often.

YORK UNIVERSITÉ UNIVERSITY

Teaser

Using the template, we can build complex trees
(and prove they are correct!)

See Trevor Brown's talk later for non-blocking balanced BSTs:

Summary

- LLX and SCX make efficient non-blocking data structures easier to design
- Also: VLX (generalizes validate instruction)
- Modular proof of correctness
- Template for building non-blocking trees

Further information:

- Pragmatic Primitives for Non-blocking Data Structures (PODC 2013)
- A General Technique for Non-blocking Trees (manuscript; includes the balanced BST)

YORK UNIVERSITÉ UNIVERSITY