Concurrent Binary Search Trees
Design and Optimizations

Arunmoezhi Ramachandran
Supervisor - Neeraj Mittal

The University of Texas at Dallas

Overview

Overview

Overview

Overview

Overview

Overview

Overview

Overview

Introduction

- CPUs aren't getting faster (memory wall, ILP wall and power wall)
- Shift towards multicore and manycore

Problem

How to keep all the cores **busy**?

Introduction

- CPUs aren't getting faster (memory wall, ILP wall and power wall)
- Shift towards multicore and manycore

Problem

How to keep all the cores **busy**?

Solution

Concurrent computing

Concurrent computing

- Example - A web crawler, mouse/keyboard
- **deal** lot of things simultaneously
- can be done on a single CPU
- **non-deterministic** control flow
- is about **hiding latency**
- **very hard** to debug

Designing Concurrent Data Structures

- Shared-memory multiprocessors concurrently execute multiple threads
- Threads communicate and synchronize through data structures in shared memory
- Threads can interleave in exponential number of ways
- Concurrent data structure must preserve its properties for all possible interleavings

Example - Shared Counter

Let x be a shared counter which can be incremented using a function fetchAndIncrement()

Example - Shared Counter

Let x be a shared counter which can be incremented using a function fetchAndIncrement()
Here are some possible implementations of this function

```
r1 = x;
inc(r1);
x = r1;
```

fetchAndIncrement: sequential

Example - Shared Counter

Let x be a shared counter which can be incremented using a function fetchAndIncrement()
Here are some possible implementations of this function

```
r1 = x;
inc(r1);
x = r1;
```

fetchAndIncrement: sequential

```
acquire(lock);
r1 = x;
inc(r1);
x = r1;
release(lock);
```

fetchAndIncrement: Using locks

Example - Shared Counter

Let x be a shared counter which can be incremented using a function fetchAndIncrement()
Here are some possible implementations of this function

```
r1 = x;
inc(r1);
x = r1;
```

fetchAndIncrement: sequential

```
acquire(lock);
r1 = x;
inc(r1);
x = r1;
release(lock);
```

fetchAndIncrement: Using locks

```
repeat
    rOld = x;
    rNew = rOld+1;
until (x.compareAndSwap(rOld,rNew));
```

fetchAndIncrement: using atomic instructions

compareAndSwap updates(atomically) the value of x to *rNew* only if the read value of x is equal to *rOld*. Returns *true* if it succeeds in updating the value of x

Design Approaches

How to handle contention among threads?

Design Approaches

How to handle contention among threads?

- Blocking Algorithms

- Non-Blocking Algorithms

Design Approaches

How to handle contention among threads?

- Blocking Algorithms
  - use locks to resolve contention
  - coarse grained or fine grained locking
  - easier to design
  - weaker progress guarantees (locking)
  - are prone to deadlock, priority inversion
- Non-Blocking Algorithms

Design Approaches

How to handle contention among threads?

- Blocking Algorithms
  - use locks to resolve contention
  - coarse grained or fine grained locking
  - easier to design
  - weaker progress guarantees (locking)
  - are prone to deadlock, priority inversion
- Non-Blocking Algorithms
  - use atomic (Read-Modify-Write) instructions to resolve contention. E.g. Compare-And-Swap(CAS) instruction
  - lock-free or wait-free
  - stronger progress guarantees (helping)
  - deadlock or priority inversion not possible
  - harder to design

Linearizability

a correctness condition for concurrent objects

Linearizability

a correctness condition for concurrent objects

- methods - take time
- methods - intervals with a start point (invocation) and an end point (response)
- history - sequence of method invocations and responses

Linearizability

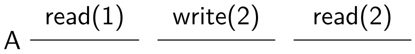
Linearizability gives a total ordering of a history

Linearizability

Linearizability gives a total ordering of a history

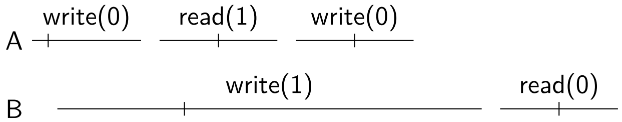
- ordering of method calls w.r.t calls on the same thread should remain unchanged
- overlapping method calls can be ordered based on the history
- non-overlapping method calls across threads should preserve real-time ordering

Linearizability - Examples



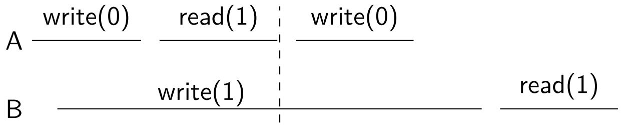
A history of a sequential object

Linearizability - Examples



A history of a concurrent object - linearizable

Linearizability - Examples



A history of a concurrent object - not linearizable

Binary Search Tree - Defintion

A *binary search tree* (BST) is a data structure which meets the following requirements:

- it is a binary tree (a node can contain atmost two children)
- each node contains a key k
- left subtree of a node contains keys lesser than k
- right subtree of a node contains keys greater than k

Binary Search Tree - Defintion

A *binary search tree* (BST) is a data structure which meets the following requirements:

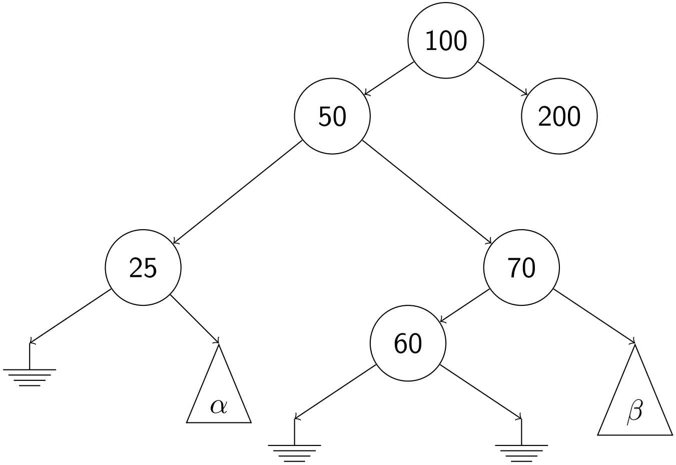
- it is a binary tree (a node can contain atmost two children)
- each node contains a key k
- left subtree of a node contains keys lesser than k
- right subtree of a node contains keys greater than k

Operations on a BST

- **search(k)** - returns *true* only if key k is present in the tree
- **insert(k)** - inserts k into the tree if it does not already exist
- **delete(k)** - deletes k from the tree if it already exist

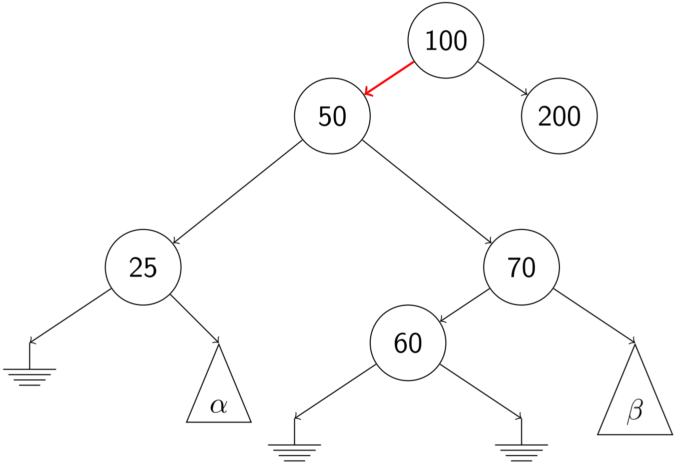
BST - Search

search(70**)**



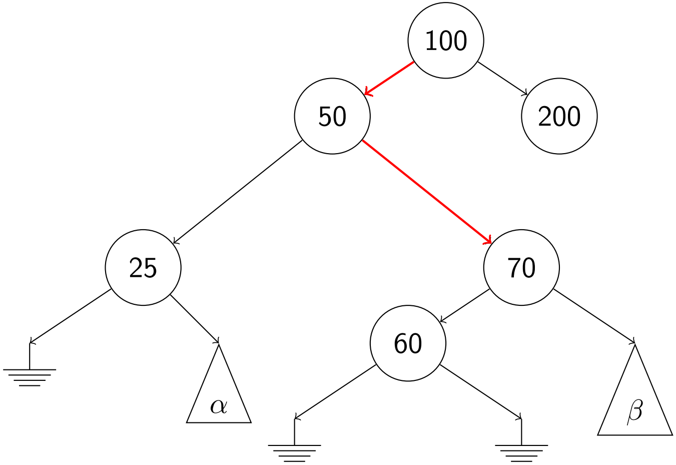
BST - Search

search(70**)**



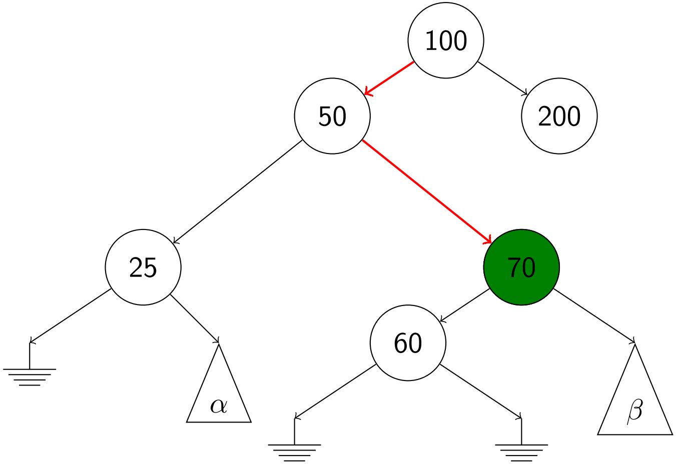
BST - Search

search(70**)**



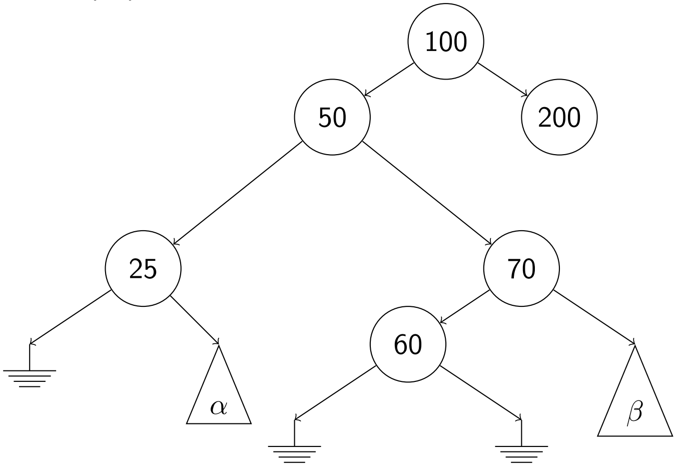
BST - Search

search(70**)**



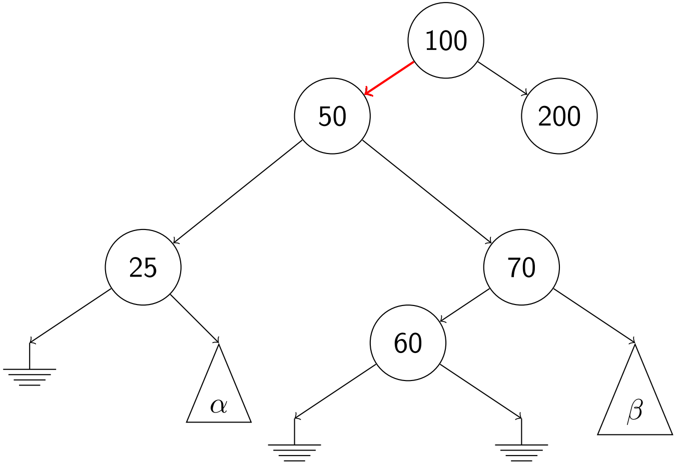
BST - Search

search(55**)**



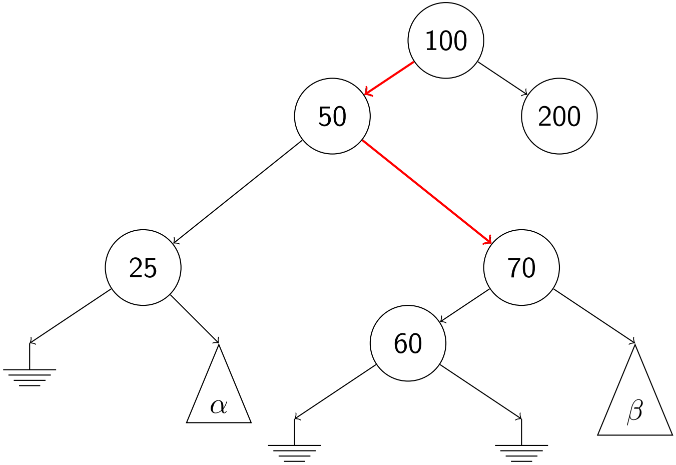
BST - Search

search(55**)**



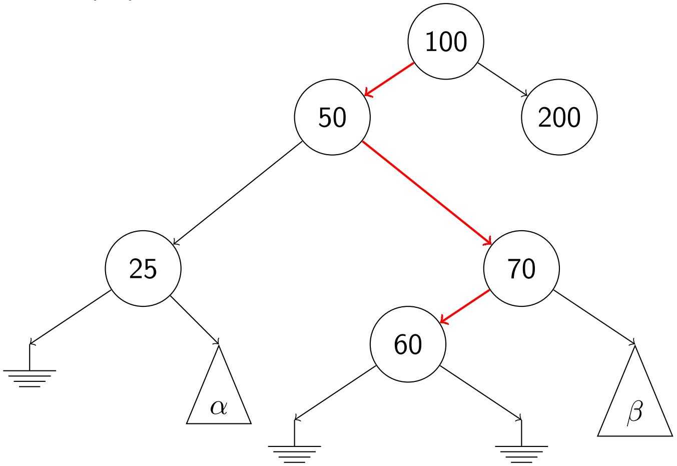
BST - Search

search(55**)**



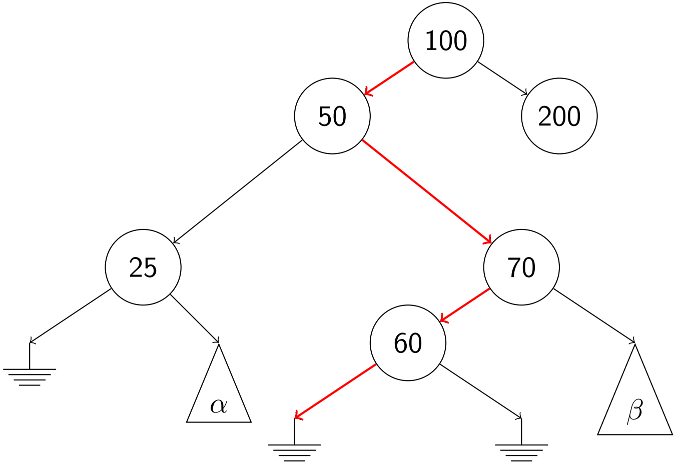
BST - Search

search(55**)**



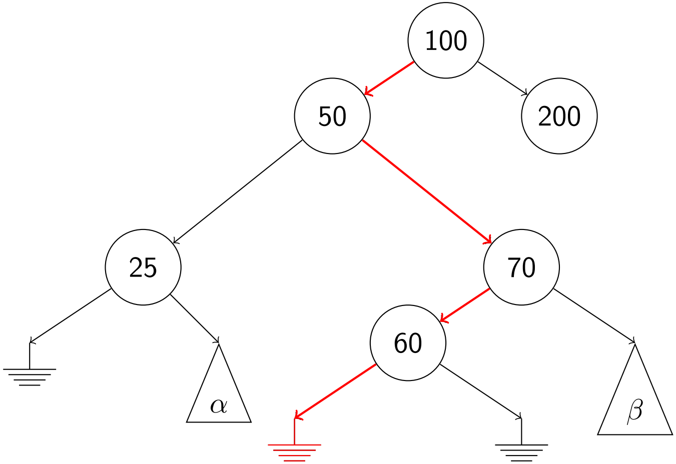
BST - Search

search(55**)**



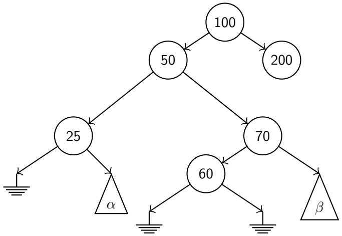
BST - Search

search(55**)**



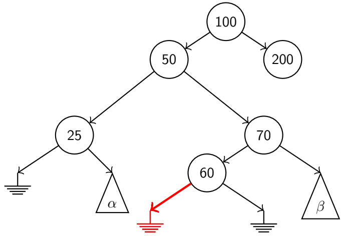
BST - Insert

insert(55)



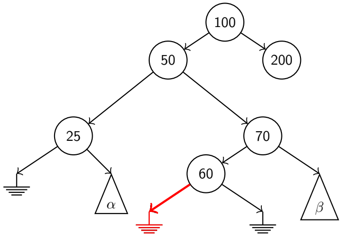
BST - Insert

insert(55**)**

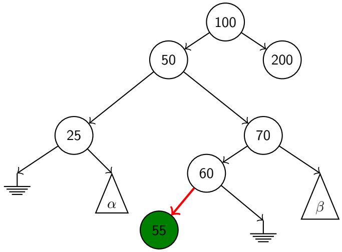


BST - Insert

insert(55)



$\Longrightarrow$

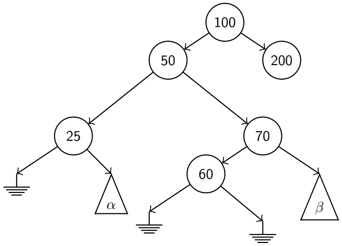


Types of delete

- simple - removing a node which has atmost one child
- complex - removing a node which has exactly two children

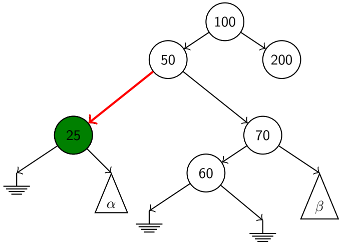
BST - Simple Delete

delete(25**)**



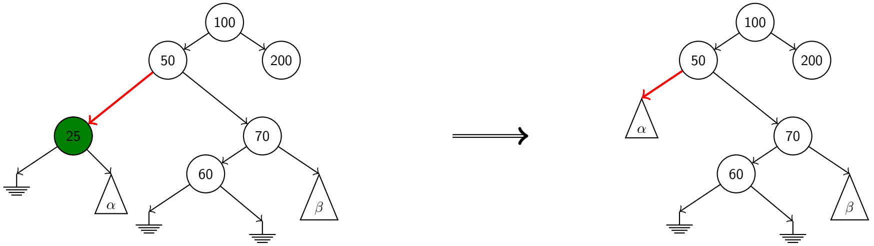
BST - Simple Delete

delete(25**)**



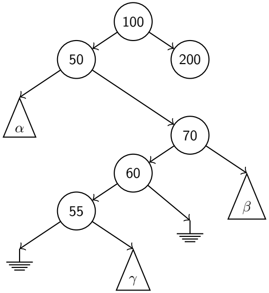
BST - Simple Delete

delete(25**)**



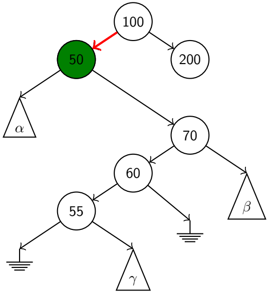
BST - Complex Delete

delete(50**)**



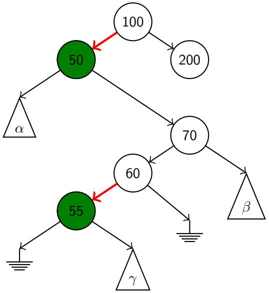
BST - Complex Delete

delete(50**)**



BST - Complex Delete

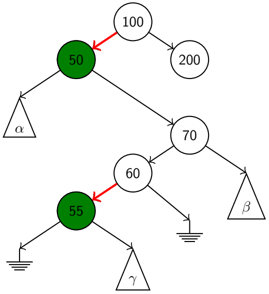
delete(50**)**



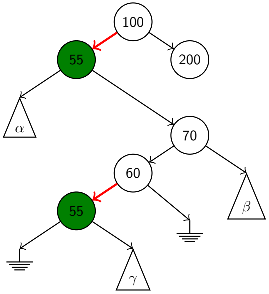
find successor

BST - Complex Delete

delete(50)



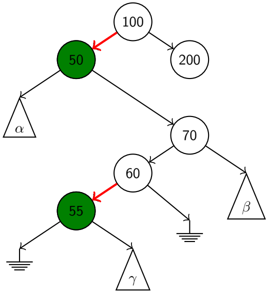
find successor



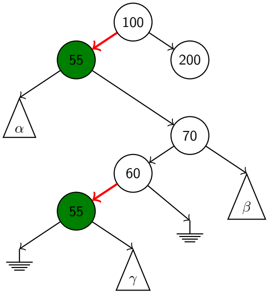
promote key

BST - Complex Delete

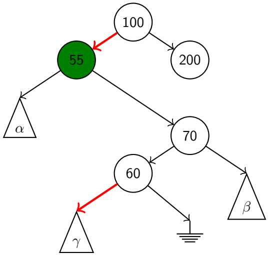
delete(50**)**



find successor



promote key



remove successor

Related Works

#	**Algorithm Type**	**Works At**	**BST Type**	**Authors**
1	lock free	node level	external	Ellen et.al[PODC'10]
2	lock free	node level	internal	Howley & Jones[SPAA'12]
3	lock free	edge level	external	Natarajan &Mittal[PPoPP'14]
4	lock based	node level	internal	Arbel & Attiya[PODC'14]
5	lock based	node level	internal	Drachsler et.al[PPoPP'14]

Lock Based BST[PPoPP'15 Poster]

Contributions

- combine edge-based locking with internal representation of BST
- optimistic tree traversal

Lock Based BST[PPoPP'15 Poster]

- common workloads have more searches than updates
  - design is optimized for searches
  - search operations are oblivious to locks

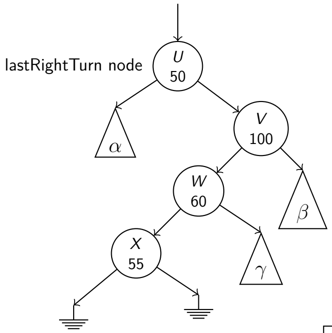
Lock Based BST[PPoPP'15 Poster]

- common workloads have more searches than updates
  - design is optimized for searches
  - search operations are oblivious to locks
- Any real life workload will have more inserts than deletes
  - insert operations do not obtain any locks
  - performs only one atomic operation

Lock Based BST[PPoPP'15 Poster]

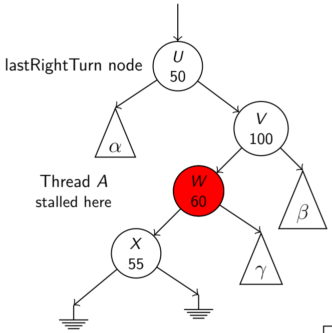
- common workloads have more searches than updates
  - design is optimized for searches
  - search operations are oblivious to locks
- Any real life workload will have more inserts than deletes
  - insert operations do not obtain any locks
  - performs only one atomic operation
- removal of a node in a concurrent BST is challenging
  - delete operations uses locks
  - locks can be obtained on nodes or edges
  - locking edges instead of nodes increases concurrency

Lock Based BST - Challenges in search



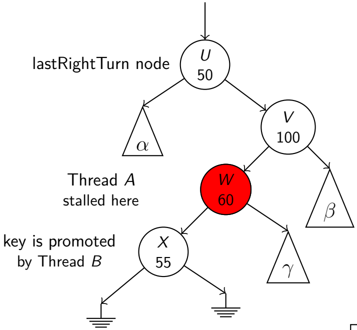
Thread A - search(55)
Thread B - delete(50)

Lock Based BST - Challenges in search



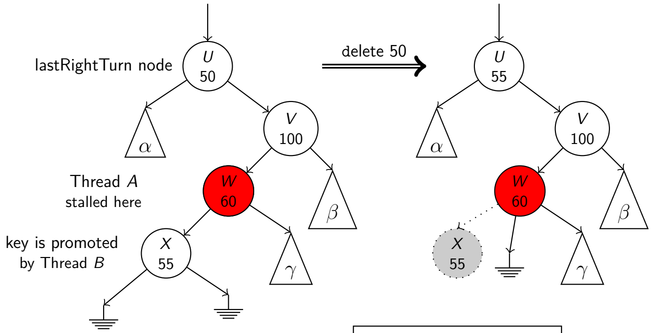
Thread A - search(55)
Thread B - delete(50)

Lock Based BST - Challenges in search



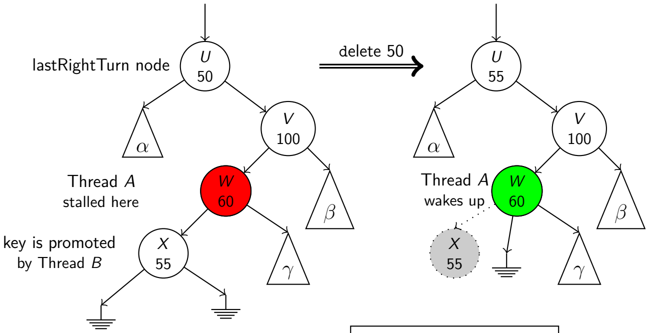
Thread A - search(55)
Thread B - delete(50)

Lock Based BST - Challenges in search



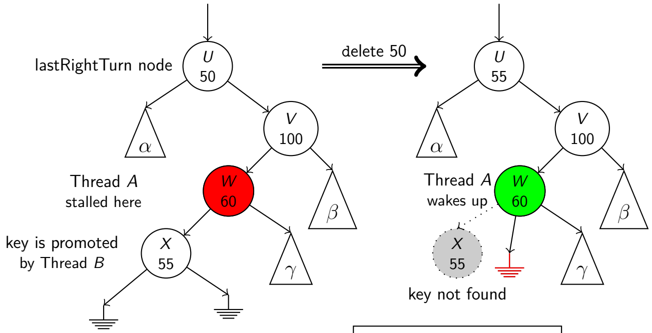
Thread A - search(55)
Thread B - delete(50)

Lock Based BST - Challenges in search



Thread A - search(55)
Thread B - delete(50)

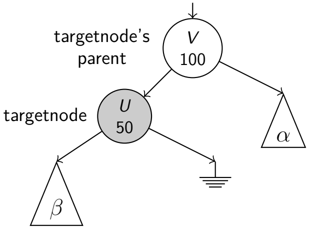
Lock Based BST - Challenges in search



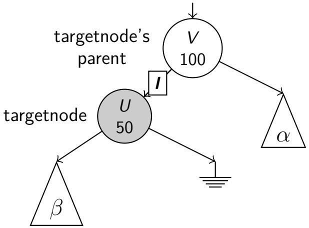
Thread A - search(55)
Thread B - delete(50)

Keep track of last right turn node and its key. If search terminates at a NULL node, check if the current key in the last right turn node has changed. If yes restart the operation from root.

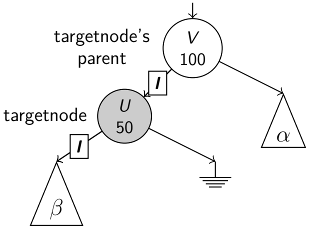
Lock Based BST - Simple Delete



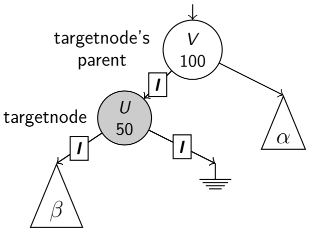
Lock Based BST - Simple Delete



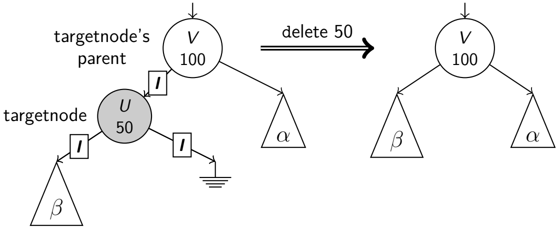
Lock Based BST - Simple Delete



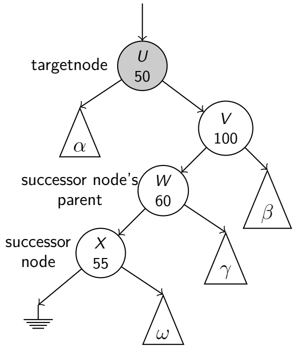
Lock Based BST - Simple Delete



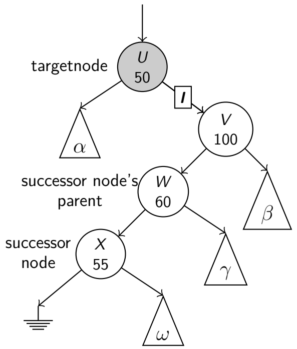
Lock Based BST - Simple Delete



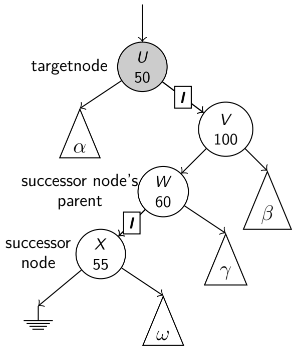
Lock Based BST - Complex Delete



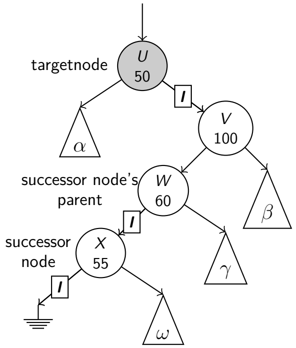
Lock Based BST - Complex Delete



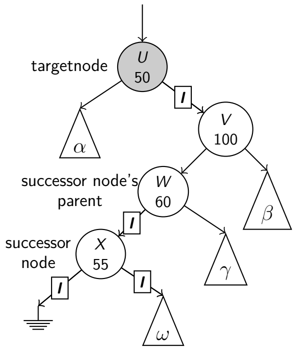
Lock Based BST - Complex Delete



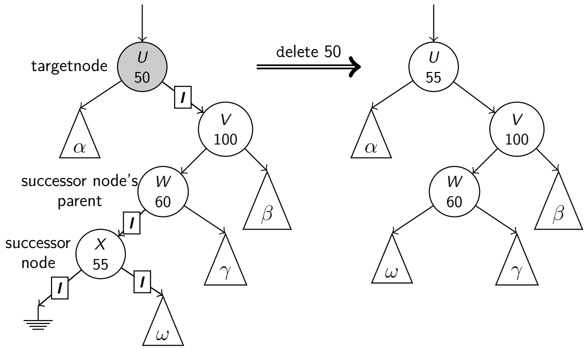
Lock Based BST - Complex Delete



Lock Based BST - Complex Delete

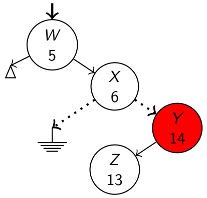


Lock Based BST - Complex Delete



Lock Based BST - More challenges in search

A scenario in which the last right turn node is removed

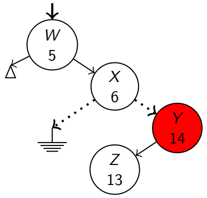


(a)

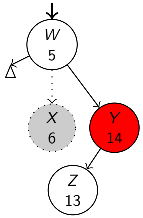
- Search(13) gets stalled at Y in (a). Its last right turn node is X

Lock Based BST - More challenges in search

A scenario in which the last right turn node is removed



(a)

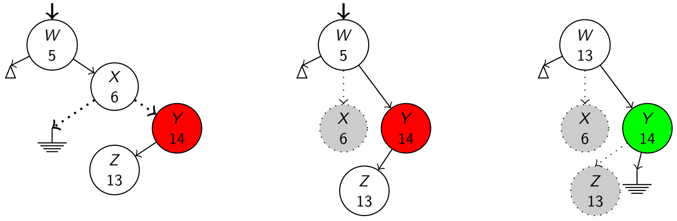


(b)

- Delete(6) removes X from the tree in (b). The key stored in X is still 6

Lock Based BST - More challenges in search

A scenario in which the last right turn node is removed

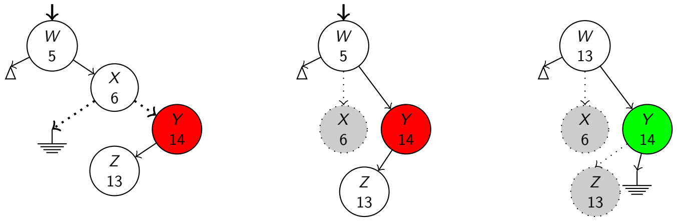


(a) (b) (c)

- Delete(5) results in 13 moving up the tree from Z to W in (c). When search(13) wakes up, it will miss 13 as the key in the last right turn node has not changed

Lock Based BST - More challenges in search

A scenario in which the last right turn node is removed

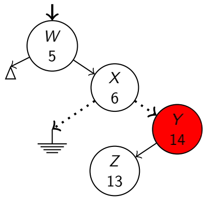


(a) (b) (c)

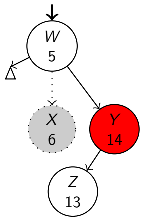
- In the first traversal search(13) saw the node X

Lock Based BST - More challenges in search

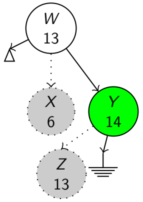
A scenario in which the last right turn node is removed



(a)



(b)



(c)

- In the second traversal there are two cases
  - case1, search(13) did not find X - save the traversal and restart
  - case2, search(13) did find X - use the results of previous traversal

Lock Free BST[ICDCN'15]

Contributions

- combine edge-based locking with internal representation of BST
- optimistic tree traversal

Lock Free BST[ICDCN'15]

Contributions

- combine edge-based locking with internal representation of BST
- optimistic tree traversal
- lock-free algorithm

Lock Free BST[ICDCN'15]

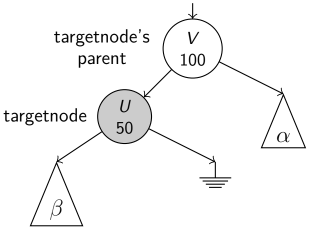
- search and inserts are same as in lock Based BST
- to maintain lock-free property, if an insert or delete operation fails, it helps a pending delete operation(if needed)

Lock Free BST - Simple Delete

- flag is owned by an operation
- if a thread which installed the flag is stalled, other threads can help complete the operation

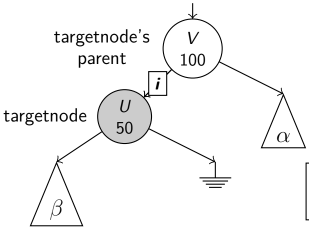
Lock Free BST - Simple Delete

- flag is owned by an operation
- if a thread which installed the flag is stalled, other threads can help complete the operation



Lock Free BST - Simple Delete

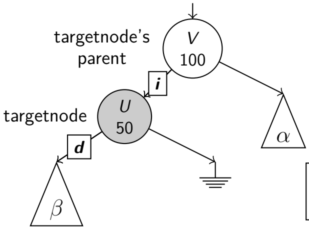
- flag is owned by an operation
- if a thread which installed the flag is stalled, other threads can help complete the operation



i - intent flag set
d - delete flag set

Lock Free BST - Simple Delete

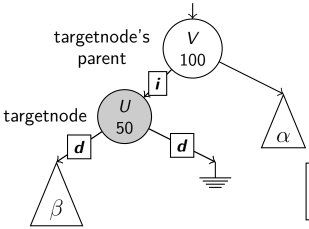
- flag is owned by an operation
- if a thread which installed the flag is stalled, other threads can help complete the operation



i - intent flag set
d - delete flag set

Lock Free BST - Simple Delete

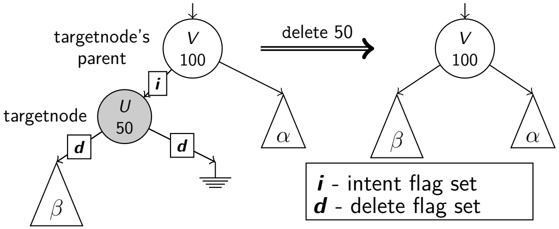
- flag is owned by an operation
- if a thread which installed the flag is stalled, other threads can help complete the operation



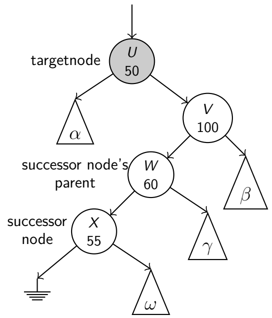
i - intent flag set
d - delete flag set

Lock Free BST - Simple Delete

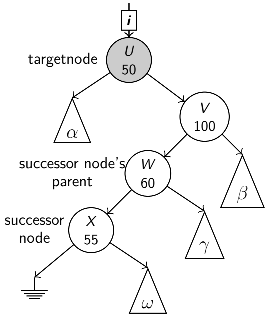
- flag is owned by an operation
- if a thread which installed the flag is stalled, other threads can help complete the operation



Lock Free BST - Complex Delete

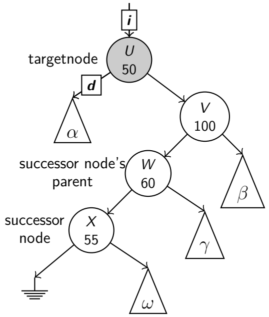


Lock Free BST - Complex Delete



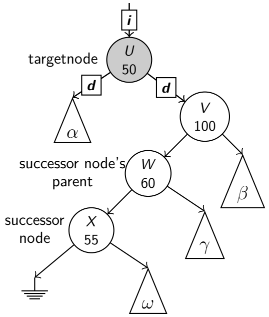
i - intent flag set ***d*** - delete flag set
p - promote flag set

Lock Free BST - Complex Delete



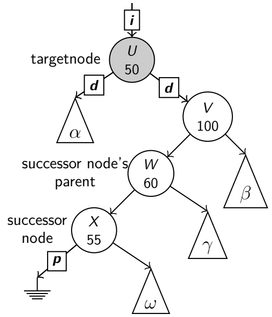
i - intent flag set ***d*** - delete flag set
p - promote flag set

Lock Free BST - Complex Delete



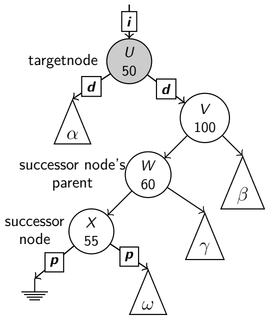
i - intent flag set ***d*** - delete flag set
p - promote flag set

Lock Free BST - Complex Delete



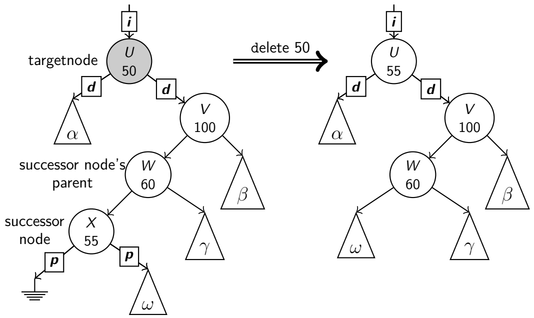
i - intent flag set ***d*** - delete flag set
p - promote flag set

Lock Free BST - Complex Delete



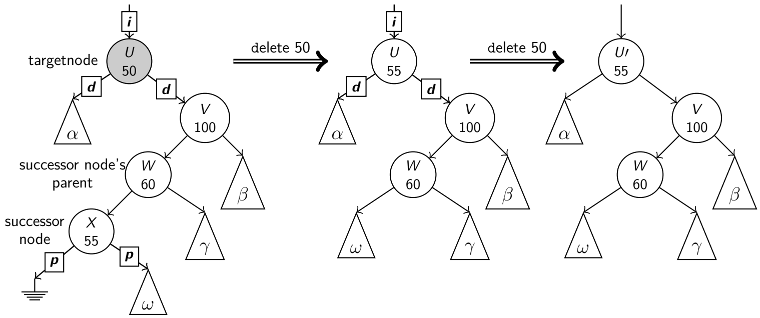
i - intent flag set ***d*** - delete flag set
p - promote flag set

Lock Free BST - Complex Delete



i - intent flag set ***d*** - delete flag set
p - promote flag set

Lock Free BST - Complex Delete



i - intent flag set ***d*** - delete flag set
p - promote flag set

Local recovery[PPoPP'16 Poster]

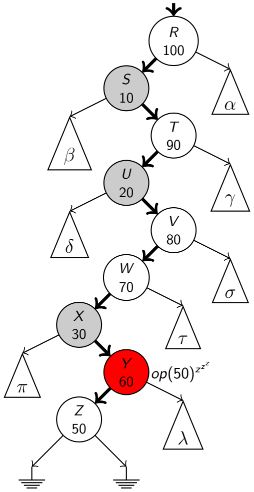
Overview

- a general technique for local recovery for concurrent BSTs
- reduces tree traversal cost during failures by restarting closer to an operation's window

Motivation

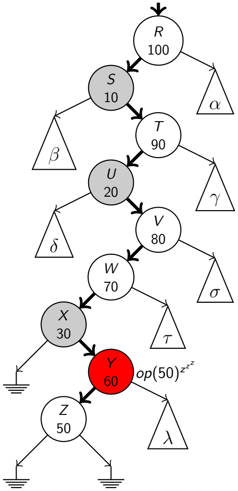
- in most concurrent BSTs, execution phase of an operation have constant time complexity
- seek phase is where an operation may end up spending most of its time (esp for large trees)
- this technique reduces the seek time

Example



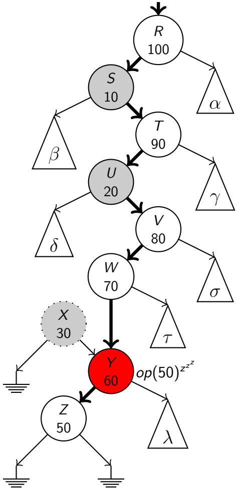
Operation $op(50)$ is suspended at node Y during its traversal

Example



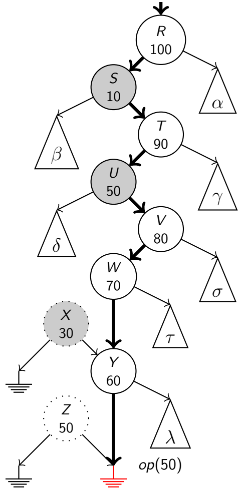
All keys in subtree π are deleted one-by-one

Example



Key 30 is deleted (simple delete); node X is removed

Example

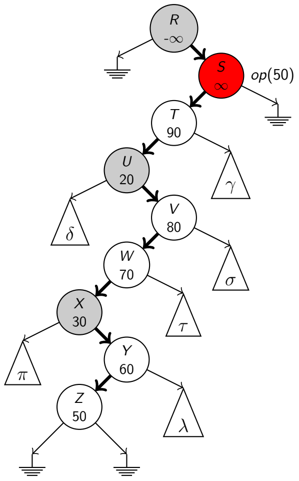


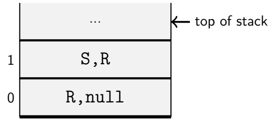
Key 20 is deleted (complex delete); key 20 is replaced with key 50 in node *U* and node *Z* is removed

Traversal Stack

- a stack to keep track of anchor nodes of all nodes in the traversal path
- reduces tree traversal cost during failures by restarting closer to an operation's window

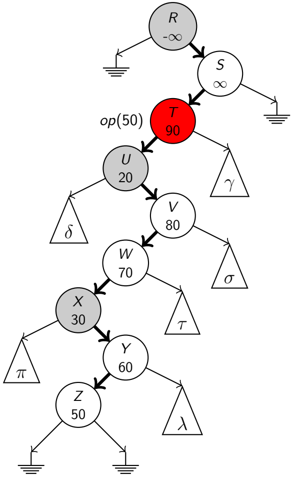
Traversal Stack

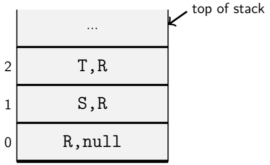




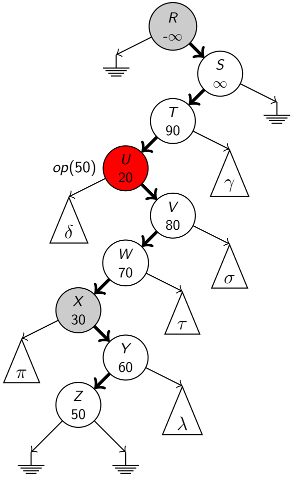
Operation *op*(50) starting at R and suspended at S along with the stack

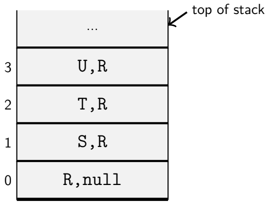
Traversal Stack



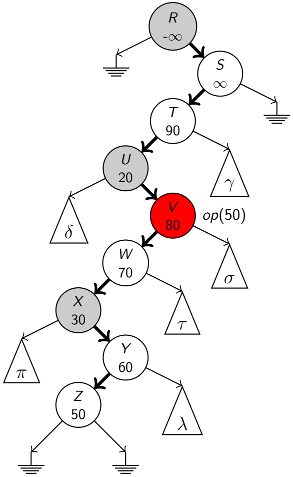


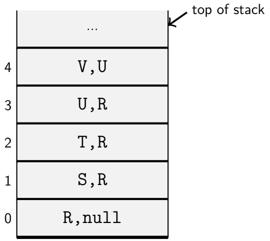
Traversal Stack





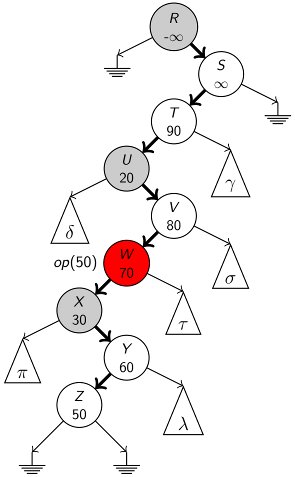
Traversal Stack

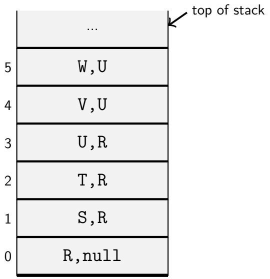




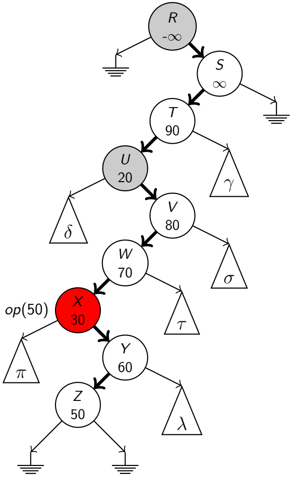
	...
4	V,U
3	U,R
2	T,R
1	S,R
0	R,null

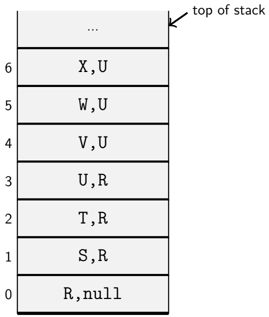
Traversal Stack





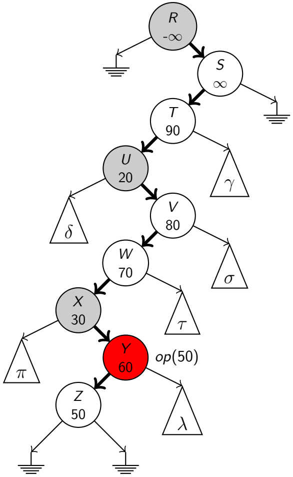
Traversal Stack





	...
6	X,U
5	W,U
4	V,U
3	U,R
2	T,R
1	S,R
0	R,null

Traversal Stack

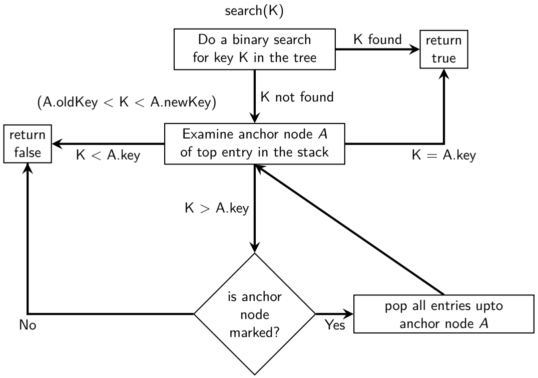


top of stack

	...
7	Y,X
6	X,U
5	W,U
4	V,U
3	U,R
2	T,R
1	S,R
0	R,null

Search

search operations do not restart

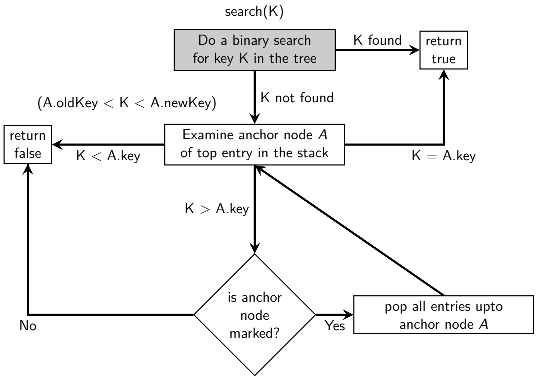


search(K)
Do a binary search for key K in the tree
K found
return true
K not found
(A.oldKey < K < A.newKey)
return false
K < A.key
Examine anchor node *A* of top entry in the stack
K = A.key
K > A.key
is anchor node marked?
No
Yes
pop all entries upto anchor node *A*

Sequence of steps in a search operation

Search

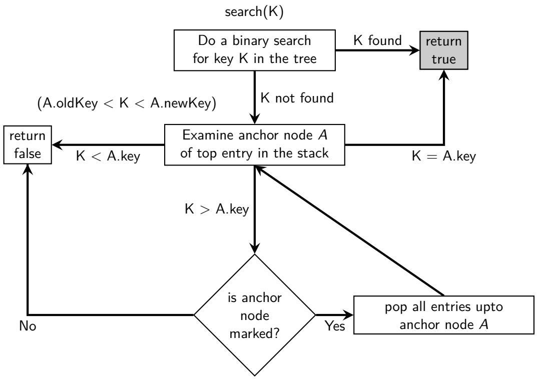
search operations do not restart


search(K)
Do a binary search for key K in the tree
K found
return true
K not found
(A.oldKey < K < A.newKey)
return false
K < A.key
Examine anchor node *A* of top entry in the stack
K = A.key
K > A.key
is anchor node marked?
No
Yes
pop all entries upto anchor node *A*

Sequence of steps in a search operation

Search

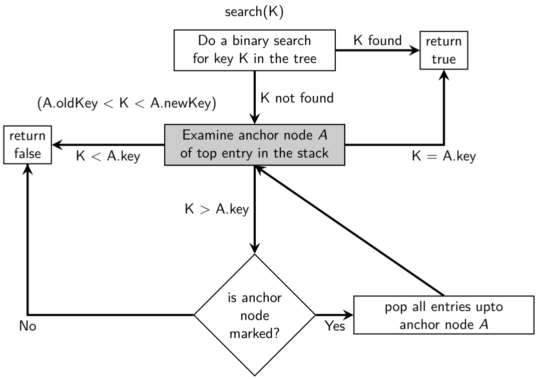
search operations do not restart


search(K)
Do a binary search for key K in the tree
K found
return true
K not found
(A.oldKey < K < A.newKey)
return false
K < A.key
Examine anchor node A of top entry in the stack
K = A.key
K > A.key
is anchor node marked?
No
Yes
pop all entries upto anchor node A

Sequence of steps in a search operation

Search

search operations do not restart

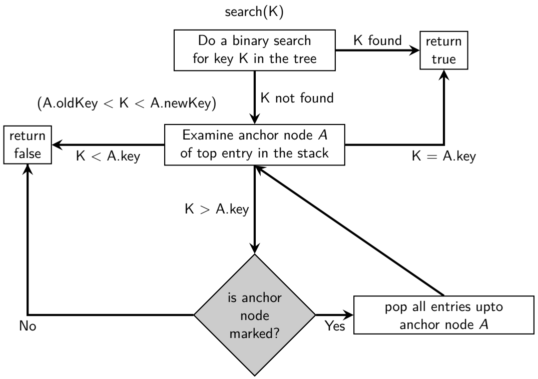


search(K)
Do a binary search for key K in the tree
K found
return true
K not found
(A.oldKey < K < A.newKey)
return false
Examine anchor node *A* of top entry in the stack
K < A.key
K = A.key
K > A.key
is anchor node marked?
No
Yes
pop all entries upto anchor node *A*

Sequence of steps in a search operation

Search

search operations do not restart

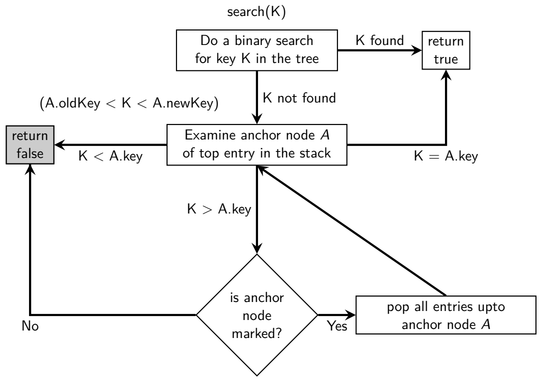


search(K)
Do a binary search for key K in the tree
K found
return true
K not found
(A.oldKey < K < A.newKey)
return false
K < A.key
Examine anchor node *A* of top entry in the stack
K = A.key
K > A.key
is anchor node marked?
No
Yes
pop all entries upto anchor node *A*

Sequence of steps in a search operation

Search

search operations do not restart

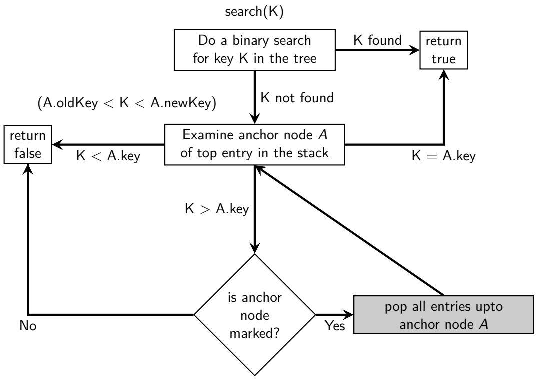


search(K)
Do a binary search for key K in the tree
K found
return true
K not found
(A.oldKey < K < A.newKey)
return false
K < A.key
Examine anchor node *A* of top entry in the stack
K = A.key
K > A.key
is anchor node marked?
No
Yes
pop all entries upto anchor node *A*

Sequence of steps in a search operation

Search

search operations do not restart

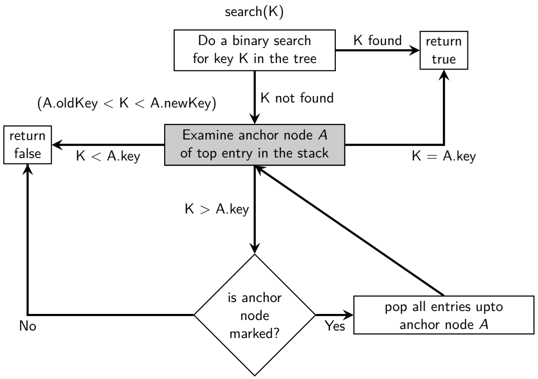


search(K)
Do a binary search for key K in the tree
K found
return true
K not found
(A.oldKey < K < A.newKey)
return false
K < A.key
Examine anchor node A of top entry in the stack
K = A.key
K > A.key
is anchor node marked?
No
Yes
pop all entries upto anchor node A

Sequence of steps in a search operation

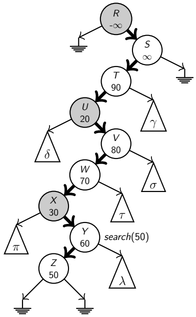
Search

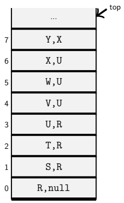
search operations do not restart

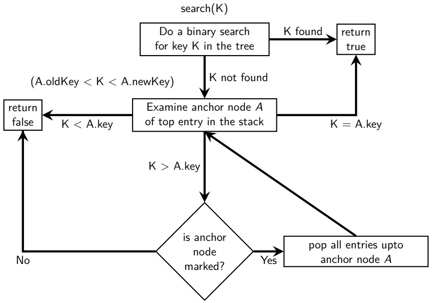

search(K)
Do a binary search for key K in the tree
K found
return true
K not found
(A.oldKey < K < A.newKey)
return false
K < A.key
Examine anchor node A of top entry in the stack
K = A.key
K > A.key
is anchor node marked?
No
Yes
pop all entries upto anchor node A

Sequence of steps in a search operation

Search - consistent anchors



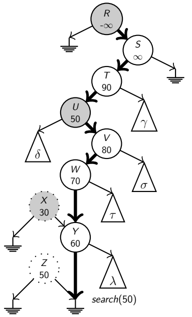


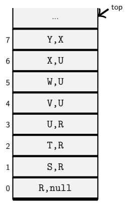


search(K)
Do a binary search for key K in the tree
K found
return true
K not found
(A.oldKey < K < A.newKey)
return false
K < A.key
Examine anchor node A of top entry in the stack
K = A.key
K > A.key
is anchor node marked?
No
Yes
pop all entries upto anchor node A

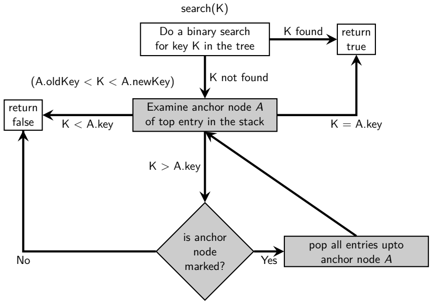
Operation *search*(50) starting at R and suspended at Y along with the stack

Search - consistent anchors





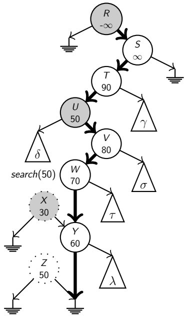
	...
7	Y,X
6	X,U
5	W,U
4	V,U
3	U,R
2	T,R
1	S,R
0	R,null

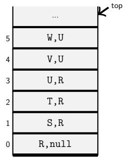


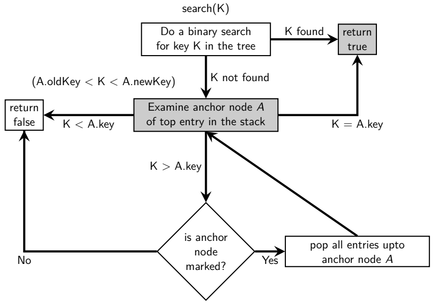
search(K)
Do a binary search for key K in the tree
K found
return true
K not found
(A.oldKey < K < A.newKey)
return false
K < A.key
Examine anchor node A of top entry in the stack
K = A.key
K > A.key
is anchor node marked?
No
Yes
pop all entries upto anchor node A

Key 30 is deleted;key 20 is deleted & replaced with key 50 in node U and node Z is removed

Search - consistent anchors



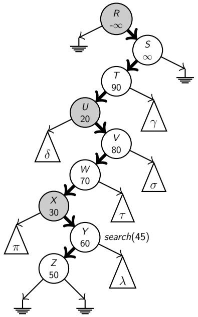


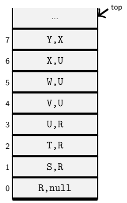


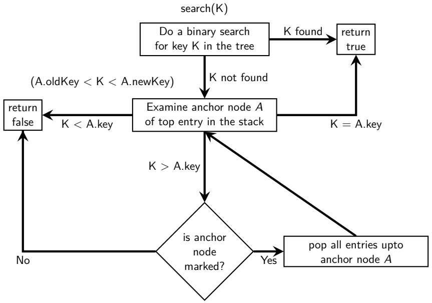
search(K)
Do a binary search for key K in the tree
K found
return true
K not found
(A.oldKey < K < A.newKey)
return false
K < A.key
Examine anchor node A of top entry in the stack
K = A.key
K > A.key
is anchor node marked?
No
Yes
pop all entries upto anchor node A

Pop upto marked anchor node X. Top of stack is now W. Examine anchor node U

Search - inconsistent anchor



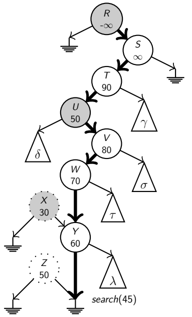


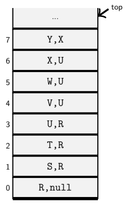


search(K)
Do a binary search for key K in the tree
K found
return true
K not found
(A.oldKey < K < A.newKey)
return false
Examine anchor node A of top entry in the stack
K < A.key
K = A.key
K > A.key
is anchor node marked?
No
Yes
pop all entries upto anchor node A

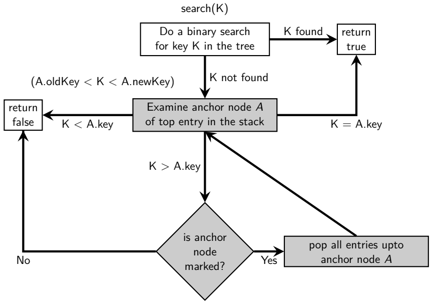
Operation *search*(45) starting at R and suspended at Y along with the stack

Search - inconsistent anchor





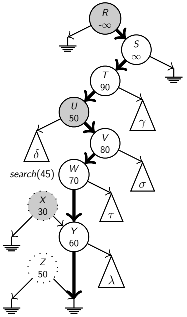
	...
7	Y,X
6	X,U
5	W,U
4	V,U
3	U,R
2	T,R
1	S,R
0	R,null

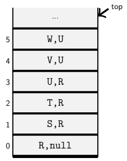


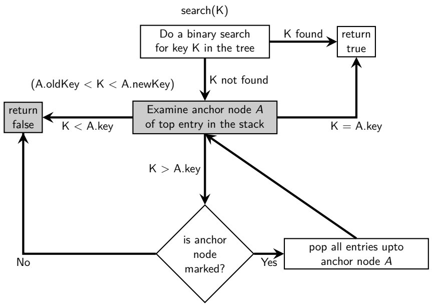
search(K)
Do a binary search for key K in the tree
K found
return true
K not found
(A.oldKey < K < A.newKey)
return false
K < A.key
Examine anchor node A of top entry in the stack
K = A.key
K > A.key
is anchor node marked?
No
Yes
pop all entries upto anchor node A

Key 30 is deleted;key 20 is deleted & replaced with key 50 in node U and node Z is removed

Search - inconsistent anchor



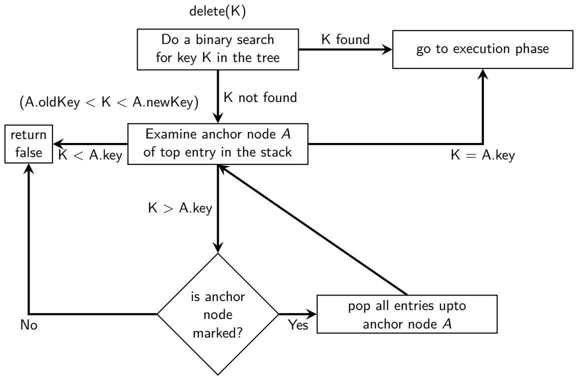



search(K)
Do a binary search for key K in the tree
K found
return true
K not found
(A.oldKey < K < A.newKey)
return false
K < A.key
Examine anchor node A of top entry in the stack
K = A.key
K > A.key
is anchor node marked?
No
Yes
pop all entries upto anchor node A

Pop upto marked anchor node X. Top of stack is now W. Examine anchor node U. A.oldKey(20) $<$ K(45) $<$ A.newKey(50). Inconsistent anchor

Delete

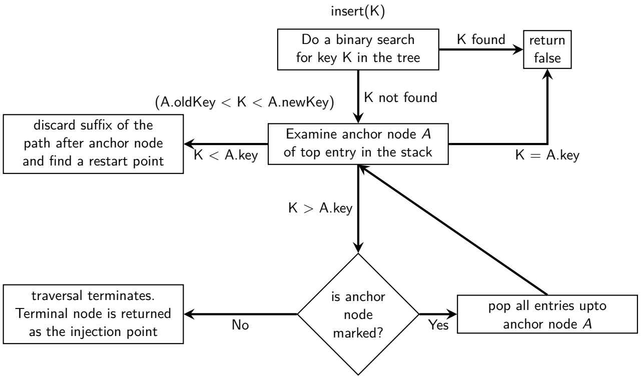
A delete operation do not restart except when there is a failure in the execution phase


delete(K)
Do a binary search for key K in the tree
K found
go to execution phase
K not found
(A.oldKey < K < A.newKey)
return false
K < A.key
Examine anchor node A of top entry in the stack
K = A.key
K > A.key
is anchor node marked?
No
Yes
pop all entries upto anchor node A

Sequence of steps in a delete operation

Insert

An insert operation needs to restart only if one of the anchor nodes in the path has become inconsistent



insert(K)
Do a binary search for key K in the tree
K found
return false
K not found
(A.oldKey < K < A.newKey)
discard suffix of the path after anchor node and find a restart point
K < A.key
Examine anchor node *A* of top entry in the stack
K = A.key
K > A.key
traversal terminates. Terminal node is returned as the injection point
No
is anchor node marked?
Yes
pop all entries upto anchor node *A*

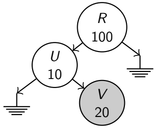
Sequence of steps in an insert operation

Wait Free Search

wait-free - every thread is able to complete its operations in a finite number of steps over an infinite period of time

- two light-weight techniques to make search operations for concurrent internal BSTs, *wait-free*
- low additional overhead
- no write instructions on share memory
- minimizes cache traffic

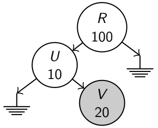
Wait Free Search



(a) Thread A gets stalled at node V

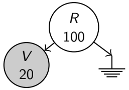
A scenario in which contains operation is not wait-free

Wait Free Search



(a) Thread *A* gets stalled at node *V*

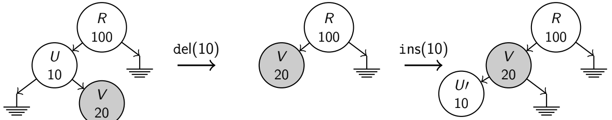
$\xrightarrow{\text{del}(10)}$



(b) Thread *B* executes delete(10) and node *U* is removed

A scenario in which contains operation is not wait-free

Wait Free Search



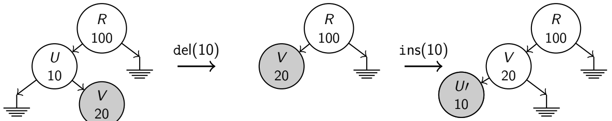
(a) Thread *A* gets stalled at node *V*

(b) Thread *B* executes delete(10) and node *U* is removed

(c) Thread *B* executes insert(10) and node *U′* is added. Thread *A* wakes up and reaches node *U′*

A scenario in which contains operation is not wait-free

Wait Free Search



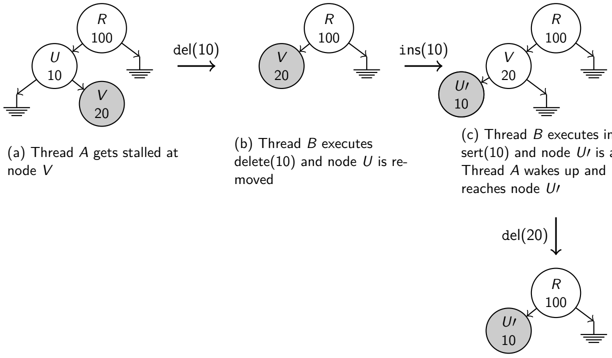
(a) Thread *A* gets stalled at node *V*

(b) Thread *B* executes delete(10) and node *U* is removed

(c) Thread *B* executes insert(10) and node *U′* is added. Thread *A* wakes up and reaches node *U′*

A scenario in which contains operation is not wait-free

Wait Free Search



(a) Thread A gets stalled at node V

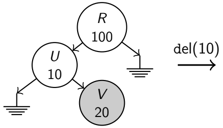
(b) Thread B executes delete(10) and node U is removed

(c) Thread B executes insert(10) and node $U\prime$ is added. Thread A wakes up and reaches node $U\prime$

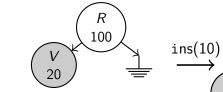
(d) Thread B executes delete(20) and node V is removed

A scenario in which contains operation is not wait-free

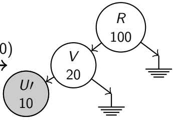
Wait Free Search



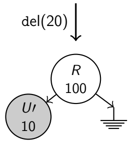
(a) Thread A gets stalled at node V



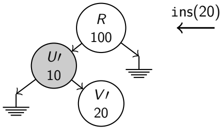
(b) Thread B executes delete(10) and node U is removed



(c) Thread B executes insert(10) and node $U\prime$ is added. Thread A wakes up and reaches node $U\prime$



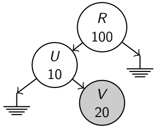
(d) Thread B executes delete(20) and node V is removed



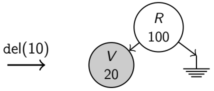
(e) Thread B executes insert(20) and a node $V\prime$ is added. Thread A wakes up and reaches node $V\prime$

A scenario in which contains operation is not wait-free

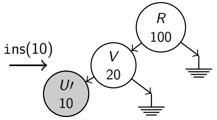
Wait Free Search



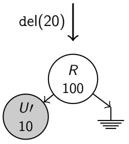
(a) Thread *A* gets stalled at node *V*



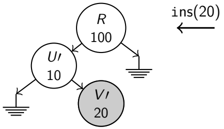
(b) Thread *B* executes delete(10) and node *U* is removed



(c) Thread *B* executes insert(10) and node *U′* is added. Thread *A* wakes up and reaches node *U′*



(d) Thread *B* executes delete(20) and node *V* is removed



(e) Thread *B* executes insert(20) and a node *V′* is added. Thread *A* wakes up and reaches node *V′*

A scenario in which contains operation is not wait-free

No Modification to Tree Node

- as long as a key is *continuously* present in the tree, its distance from root is *monotonically non-increasing*
- if a key is not found after visiting a "certain" number of nodes in the tree, then traversal stops
- sufficient to examine the path traversed to check whether or not the key has moved up
- In case the key is not continuously present in the tree, it is acceptable to return either:
  - present - linearized after the insert operation that added the key to the tree
  - not present - linearized after the delete operation that removed the key from the tree

No Modification to Tree Node

when to stop?

No Modification to Tree Node

when to stop?

Each process maintains two counters:

- insert counter - number of true inserts
- delete counter - number of true deletes

$IC[i]$ and $DC[i]$ denote the number of insert and delete operations, respectively, process P_i has performed so far

No Modification to Tree Node

- insert counter incremented before adding a key
- delete counter incremented before removing a key
- insert (delete) counter at a process is an upper (lower) bound on the number of keys that the process has added to (removed from) the tree

No Modification to Tree Node

```
read and aggregate delete counter values of all processes $DC = \sum_{i=1}^{p} DC[i]$;
read and aggregate insert counter values of all processes $IC = \sum_{i=1}^{p} IC[i]$;
$IC - DC \geq actualtreesize$ as IC $\geq$ actual inserts and DC $\geq$ actual deletes;
```

pseudocode: waitFreeSearch

$IC - DC$ gives an upper bound on number of keys to traverse before stopping the search operation

With Modification to Tree Node

- previous approach - time complexity depends on tree size
- this approach - time complexity depends on the tree height
- but needs modifications to tree node structure
- each node has a timestamp on when it was created
- timestamp - ⟨process id,process sequence number⟩
- process sequence number is incremented before a node is added to the tree

With Modification to Tree Node

```
read current sequence number of all processes;
let $label[i]$ denote the sequence number of procecess $p_i$;
stop the downward traversal of the tree once a node with timestamp $\langle$i,v$\rangle$
such that $v > labels[i]$ is encountered;
```

pseudocode: waitFreeSearch

Experimental Setup

To compare the performance of various concurrent BSTs we considered the following parameters:

- Maximum Tree Size
  - key space size varied from 2^{13} (8Ki) to 2^{24} (16Mi).
- Relative Distribution of Operations
  - Read-Dominated (90% search, 9% insert and 1% delete)
  - Mixed (70% search, 20% insert and 10% delete)
  - Write-Dominated (0% search, 50% insert and 50% delete)
- Maximum degree of Contention
  - number of threads that can concurrently operate on the tree
  - we collected data for 32 threads

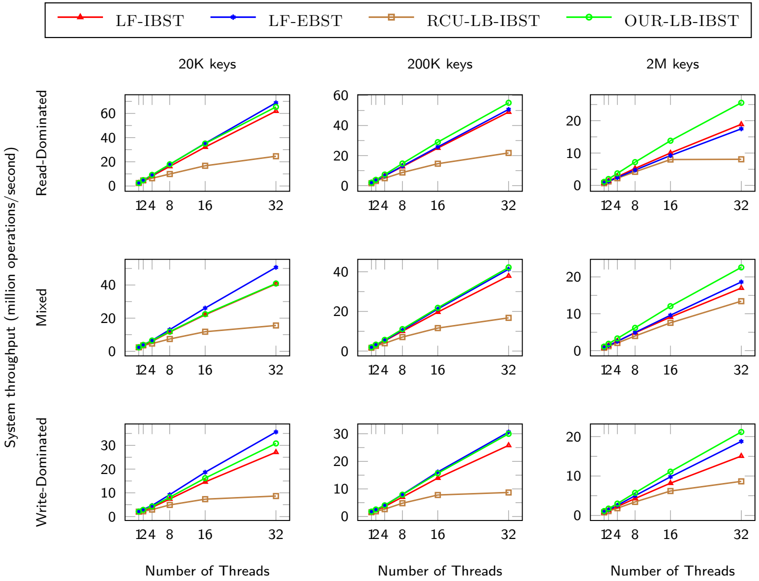
Experimental Setup

- Throughput computed as millions of operations per second (MOPS)
- each trial was run for 2 minutes
- Average over 5 trials
- *pre-populated* the tree to 50% of its maximum size to capture steady state behaviour
- beginning of each run consisted of a 1 second "warm-up" phase whose numbers were excluded in the computed statistics to avoid initial caching effects
- The machine we used is a Dell PowerEdge R820 server with 4 Intel E5-4650 @ 2.70GHz 8-core processors (32 cores in total) and 1TB of DDR3 memory with HT disabled. 256KB L2 and 20MB shared L3

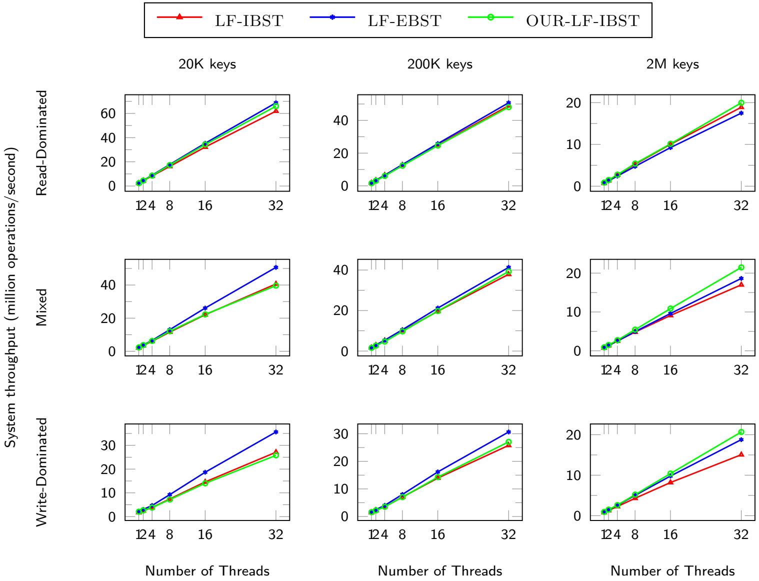
Other Concurrent BSTs

- a lock-free internal BST by Howley and Jones[SPAA'12], denoted by $\mathrm{LF\text{-}IBST}$
- a lock-free external BST by Natarajan and Mittal[PPoPP'14], denoted by $\mathrm{LF\text{-}EBST}$
- RCU-based internal BST by Arbel and Attiya[PODC'14], denoted by $\mathrm{RCU\text{-}LB\text{-}IBST}$

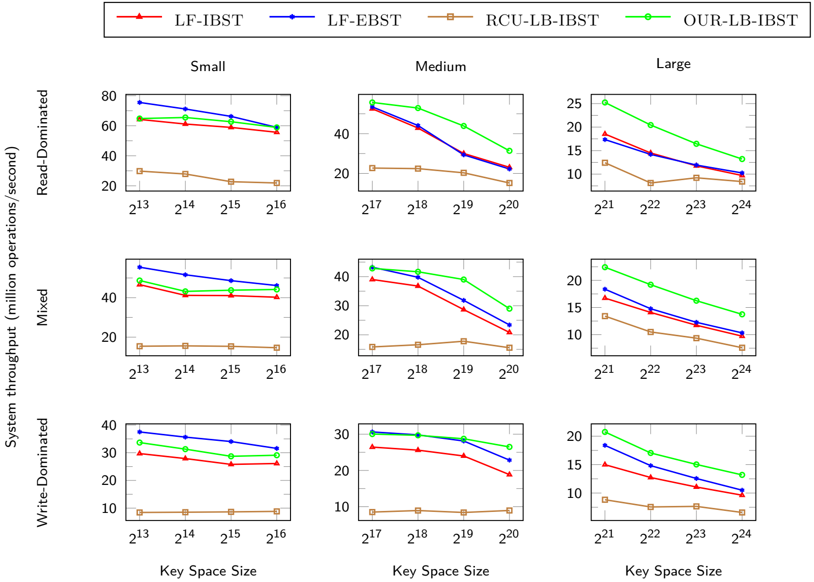
Lock Based BST - thread sweep



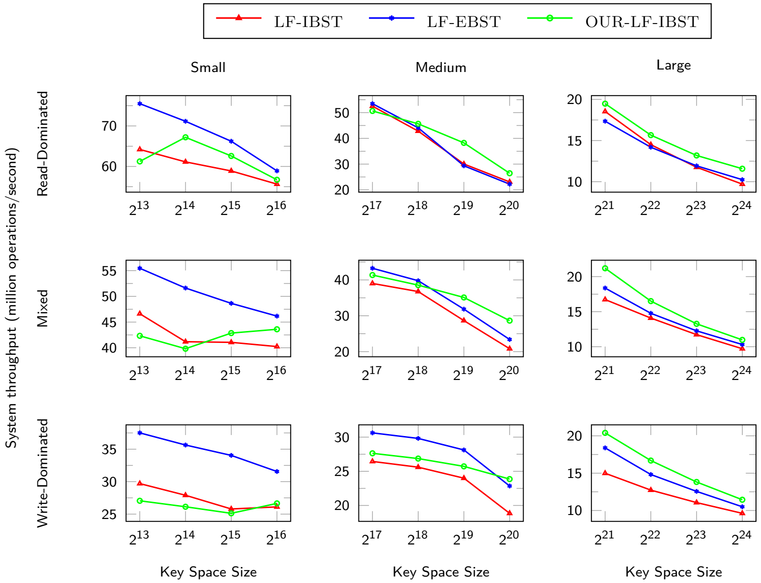
Lock Free BST - thread sweep



Lock Based BST - key sweep



Lock Free BST - key sweep



Results Summary

Comparison of different concurrent BSTs in the absence of contention

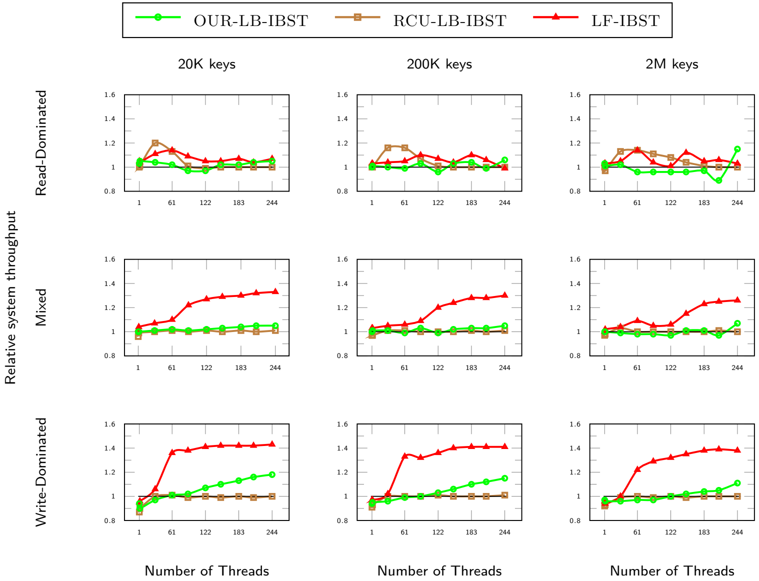
- speedup is calculated over the second best algorithm

	Speedup	
Workload	**Lock Based BST**	**Lock Free BST**
Read-Dominated	46%	27%
Mixed	33%	22%
Write-Dominated	26%	13%

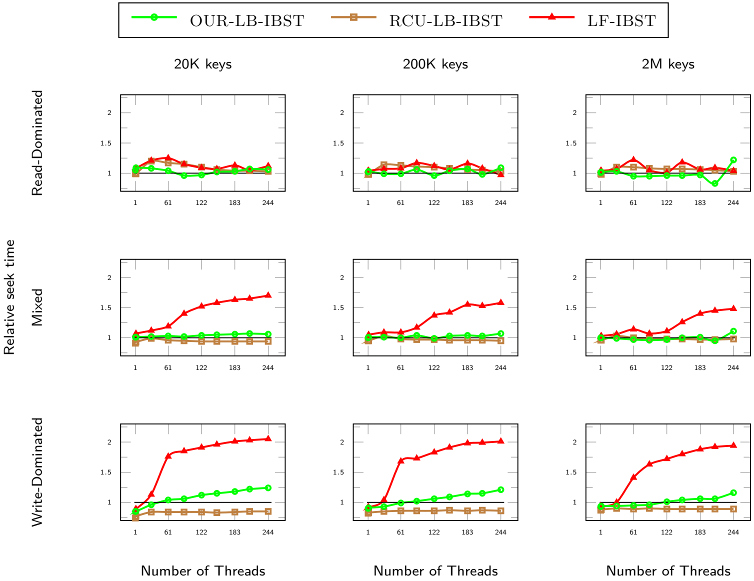
Local recovery

- helpful only for high contention cases
- uniform distribution usually causes less contention
- zipf distribution (a power-law distribution) causes high contention
- experiments run on a 61 core coprocessor
- 4 hardware threads per core - 244 total threads

Local recovery - Throughput - relative



Local recovery - Seek Time - relative



Future Work

- analyze our local recovery algorithm (amortized time complexity)
- develop concurrent K-ary BST which can improve spatial locality
- work on other data structures like tries, bloom filters, etc.
- evaluate using real workloads.

Thank you