

Concurrent Internal Binary Search Trees



Arunmoezhi Ramachandran
Supervisor - Neeraj Mittal

The University of Texas at Dallas

Overview

Introduction

Design Approaches

Linearizability

Binary Search Tree

Related Works

Lock Based Binary Search Tree

Lock Free Binary Search Tree

Experimental Evaluation

Future Work

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**?

Solutions

Parallel computing (obvious choice)

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**?

Solutions

Parallel computing (obvious choice)

Concurrent computing (a better choice)

Concurrency vs Parallelism

Concurrency is not parallelism (it's better!!)

Concurrency vs Parallelism

Concurrency is not parallelism (it's better!!)

Parallel Computing

- ▶ decades of research done
- ▶ Example - Matrix-Matrix Multiplication
- ▶ **do** lot of things simultaneously
- ▶ cannot be done on a single CPU
- ▶ **deterministic** control flow
- ▶ is about **speedup**
- ▶ **hard** to debug

Concurrent Computing

- ▶ Relatively new
- ▶ 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 (thread owns a lock)
- ▶ 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 (thread owns a lock)
- ▶ 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 (operation owns a lock - helping)
- ▶ deadlock or priority inversion not possible
- ▶ harder to design

Linearizability

An object has:

- ▶ state
- ▶ a set of methods which operate on the object making the object move from one valid state to another
- ▶ A sequence of method invocations and responses is called a *history*
- ▶ Every method has a set of pre-conditions and post-conditions
- ▶ Pre-conditions captures the state of an object before a method is invoked
- ▶ Post-conditions captures the state after a method returns

Linearizability

- ▶ In a sequential specification of an object, each method is described in isolation
- ▶ The interactions among methods are captured by the side-effects on the object state
- ▶ So a sequential object needs a meaningful state only between method calls
- ▶ New methods can be added without modifying the existing methods
- ▶ proving correctness requires that the any history respects the sequential specification of the object

Linearizability

- ▶ For a concurrent object, multiple methods might be executing concurrently
- ▶ Since method calls overlap, an object might never be between method calls
- ▶ exponential possible interactions among method calls
- ▶ Proving the correctness requires that some total ordering of any history respects the sequential specification of the object

Linearizability

Linearizability requires two properties:

- ▶ the object (or data structure) be sequentially consistent¹
- ▶ the total ordering which makes it sequentially consistent respect the *real-time ordering* among the operations in the execution

respecting real-time ordering - if an operation op_1 completed before another operation op_2 , then op_1 must be ordered before op_2

¹ the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program

Linearizability

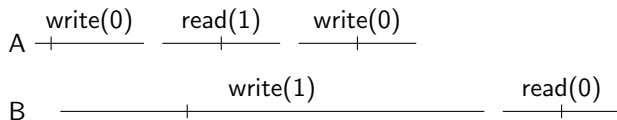
- ▶ *linearization point* - a distinct point between a method invocation and response where the method appears to have taken effect instantaneously
- ▶ Order the method calls based on their linearization points
- ▶ Resulting order should be in the sequential specification of the object.

Linearizability - Examples

A read(1) write(2) read(2)

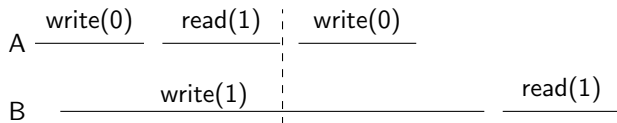
A history of a sequential object

Linearizability - Examples



A history of a concurrent object - linearizable

Linearizability - Examples



A history of a concurrent object - not linearizable

Binary Search Tree - Definition

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

- ▶ it is a binary tree (a node can contain at most 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 - Definition

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

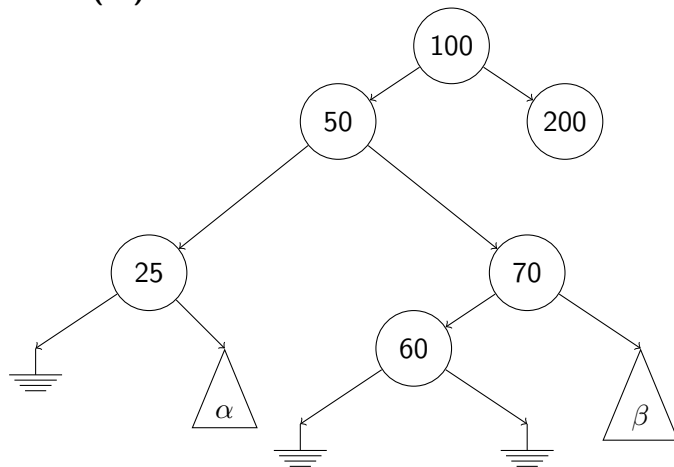
- ▶ it is a binary tree (a node can contain at most two children)
- ▶ each node contains a key k
- ▶ left subtree of a node contains keys lesser than k
- ▶ right subtree of a node contains keys greater than k

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

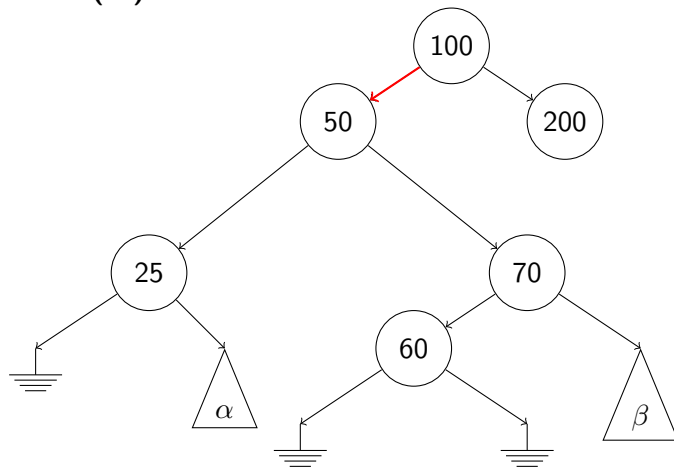
BST - Search

search(70)



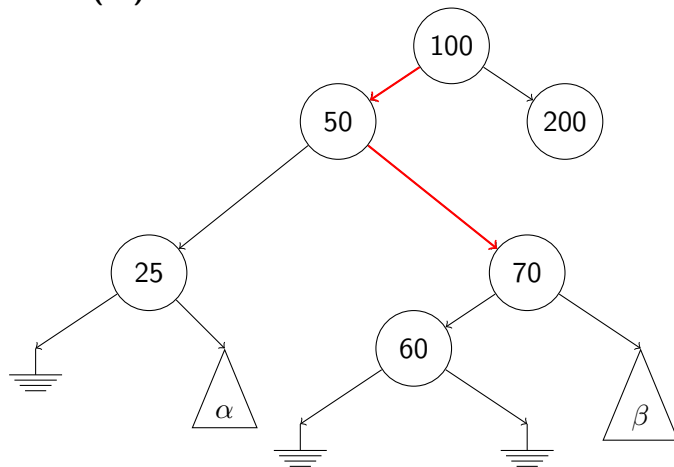
BST - Search

search(70)



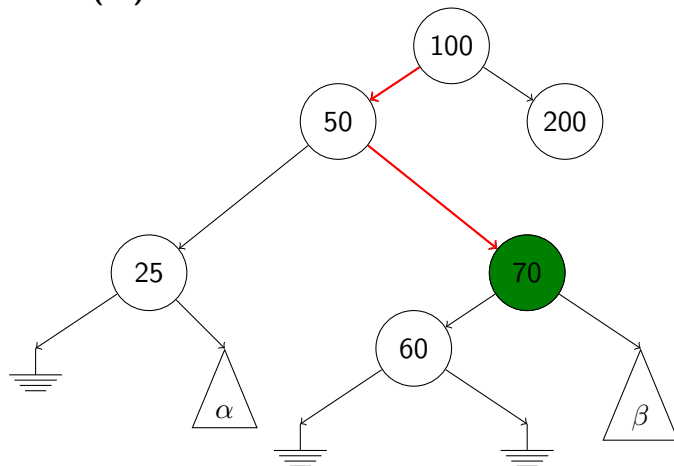
BST - Search

search(70)



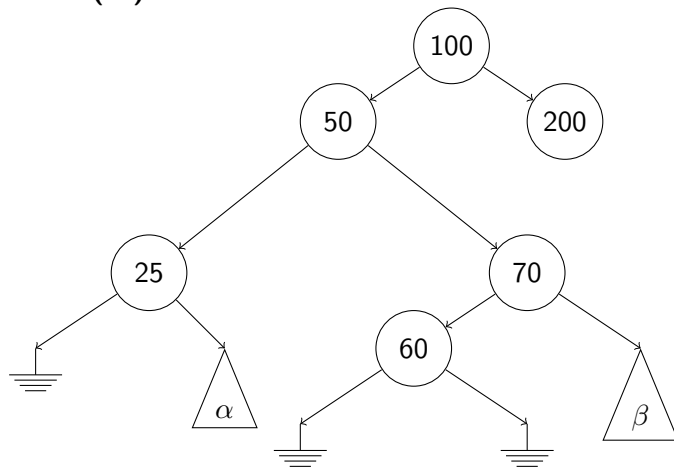
BST - Search

search(70)



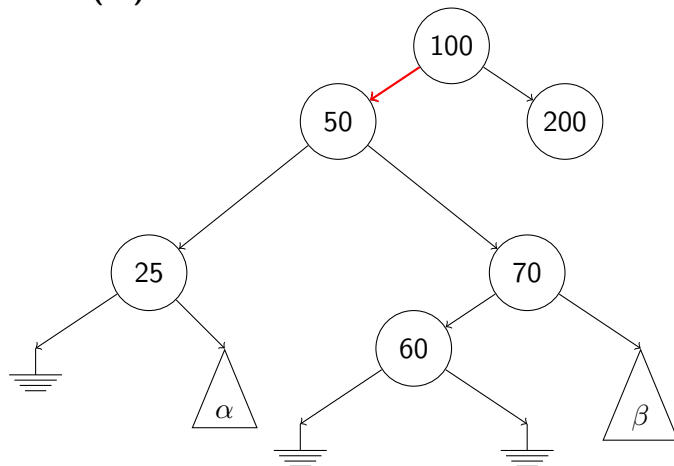
BST - Search

search(55)



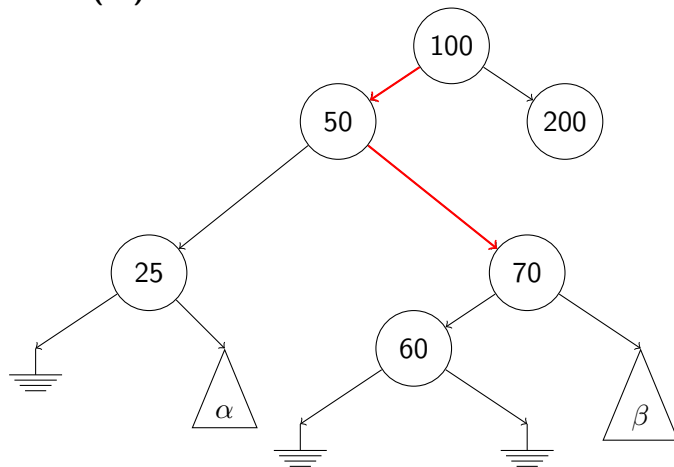
BST - Search

search(55)



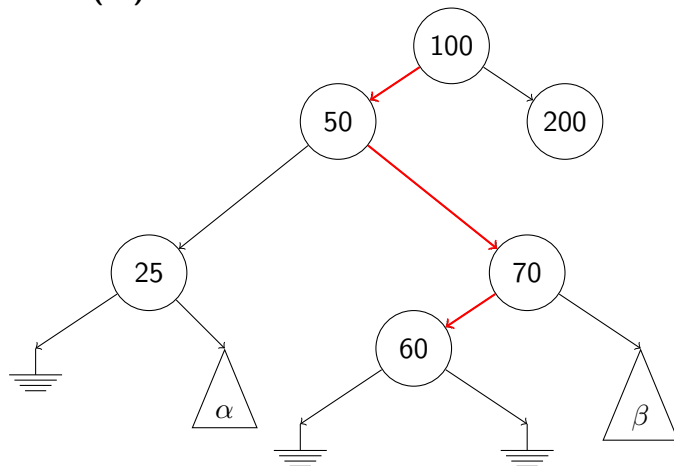
BST - Search

search(55)



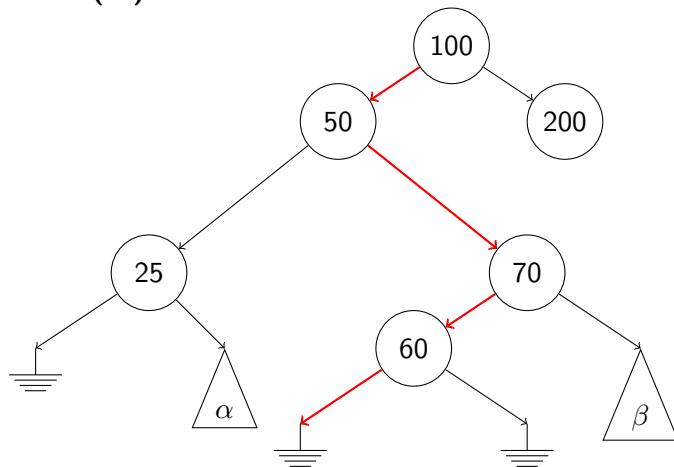
BST - Search

search(55)



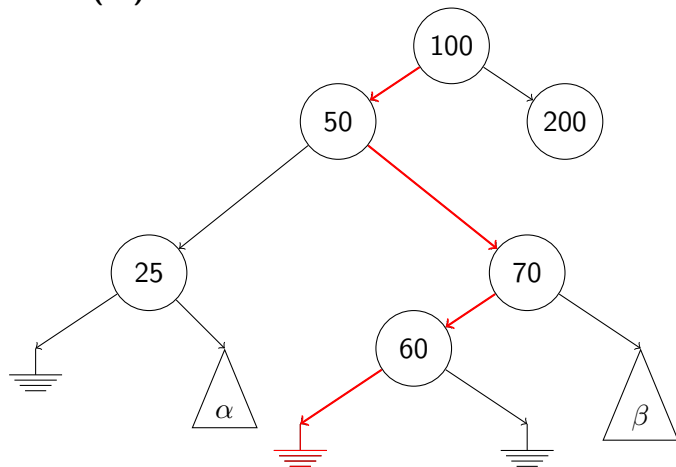
BST - Search

search(55)



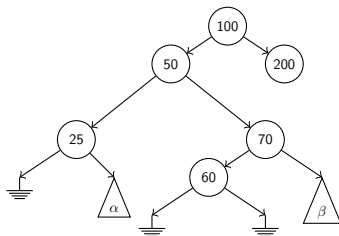
BST - Search

search(55)



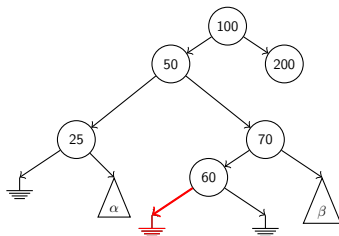
BST - Insert

insert(55)



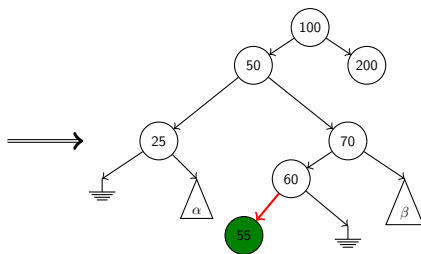
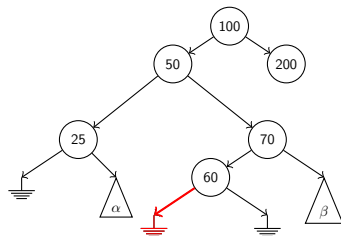
BST - Insert

insert(55)



BST - Insert

insert(55)

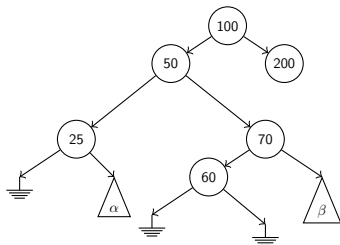


Types of delete

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

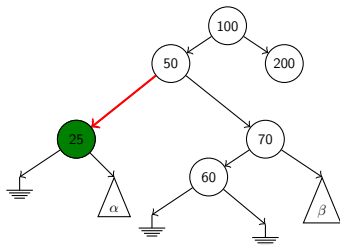
BST - Simple Delete

delete(25)



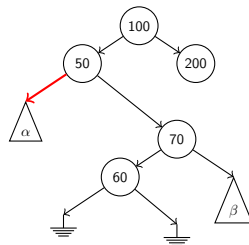
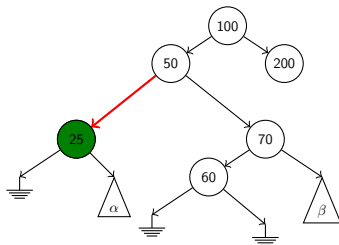
BST - Simple Delete

delete(25)



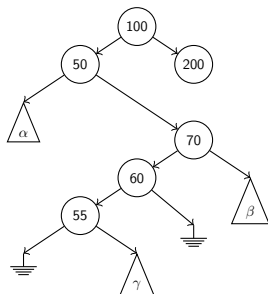
BST - Simple Delete

delete(25)



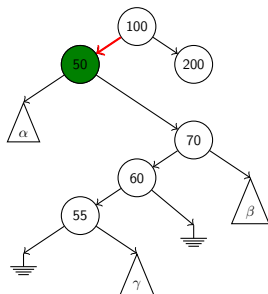
BST - Complex Delete

delete(50)



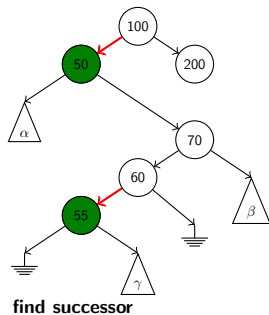
BST - Complex Delete

delete(50)



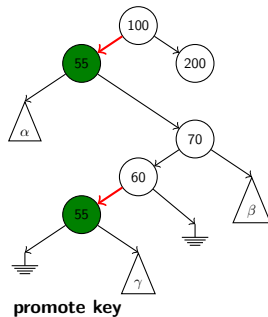
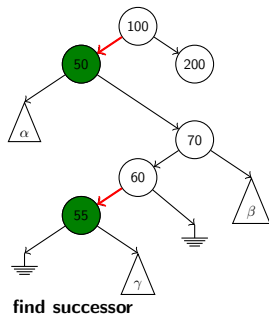
BST - Complex Delete

delete(50)



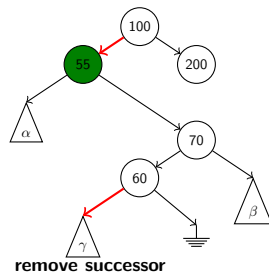
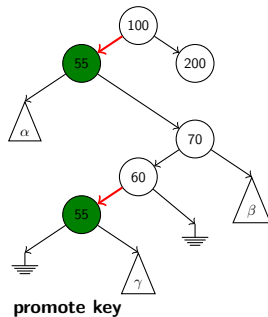
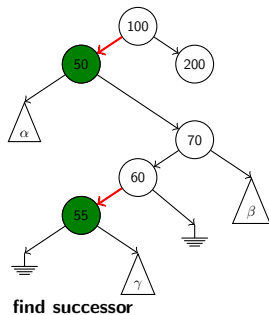
BST - Complex Delete

delete(50)



BST - Complex Delete

delete(50)



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

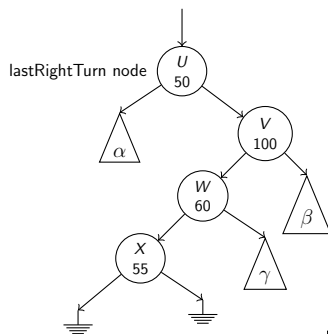
Lock Based BST[PPoPP'15 Poster]

- ▶ 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]

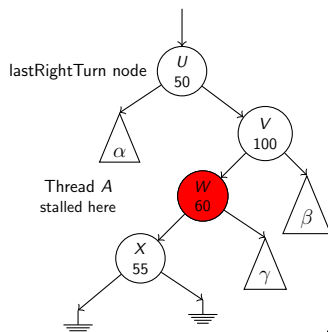
- ▶ common workloads have more searches than updates
 - ▶ design is optimized for searches
 - ▶ search operations are oblivious to locks
- ▶ Any real life workload will have more inserts than deletes
 - ▶ insert operations do not obtain any locks
 - ▶ performs only one atomic operation
- ▶ removal of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



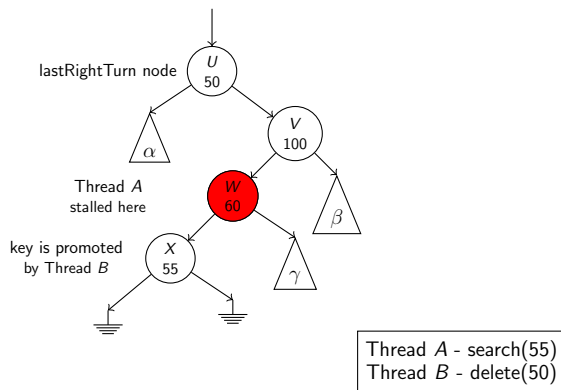
Thread A - search(55)
Thread B - delete(50)

Lock Based BST - Challenges in search

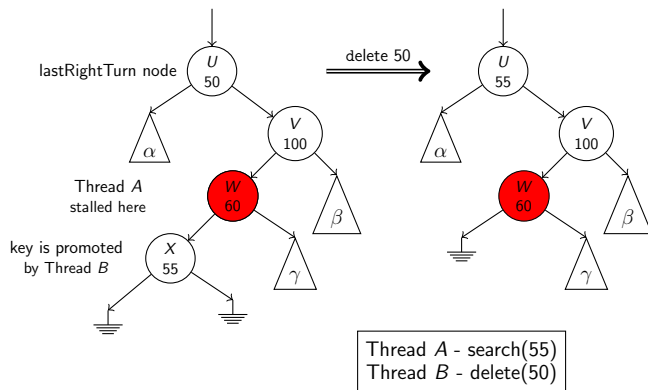


Thread A - search(55)
Thread B - delete(50)

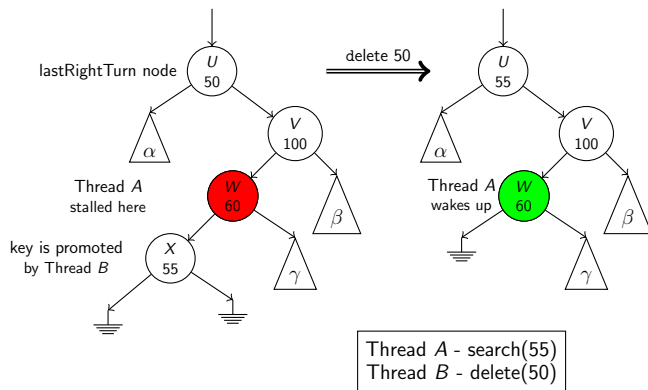
Lock Based BST - Challenges in search



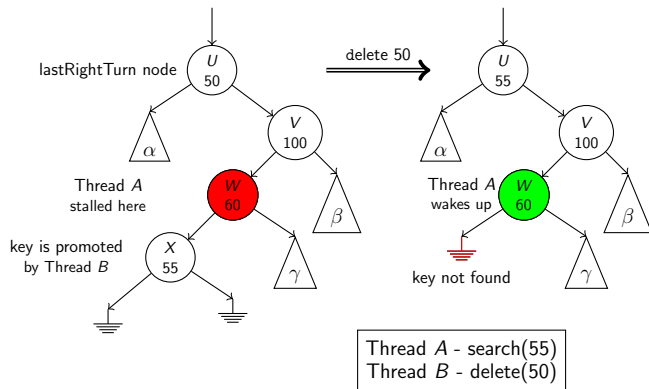
Lock Based BST - Challenges in search



Lock Based BST - Challenges in search



Lock Based BST - Challenges in search



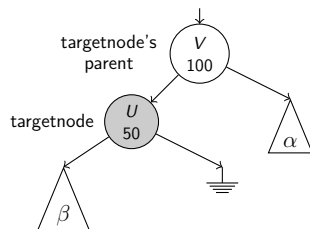
Keep track of last right turn node and its key. If search terminates at a NULL node, check if the current key in the last right turn node has changed. If yes restart the operation from root.

Lock Based BST - Delete

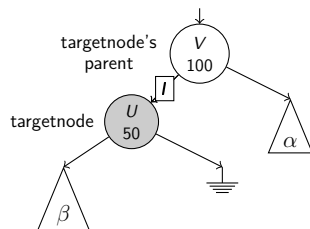
pseudocode for delete

```
locate the node to delete;
if simple delete then
    lock the edge ⟨parent,node⟩;
    lock the children edges;
    make the parent point to the non-null child using a simple write
    instruction;
    release all locks;
else // complex delete
    lock the edge ⟨node,rightChild⟩;
    find the successor;
    lock the edge ⟨successorParent,successor⟩;
    lock the children edges of successor;
    promote key;
    remove successor by a making successorParent point to non-null child of
    successor;
    release all locks;
end
```

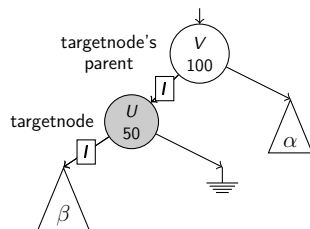
Lock Based BST - Simple Delete



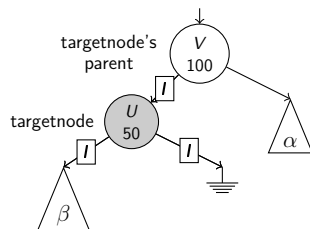
Lock Based BST - Simple Delete



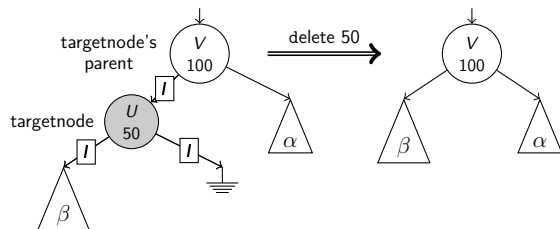
Lock Based BST - Simple Delete



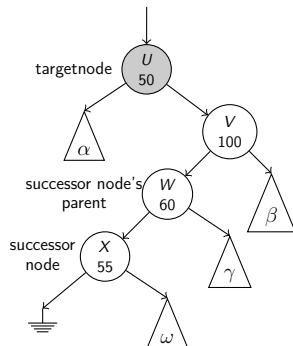
Lock Based BST - Simple Delete



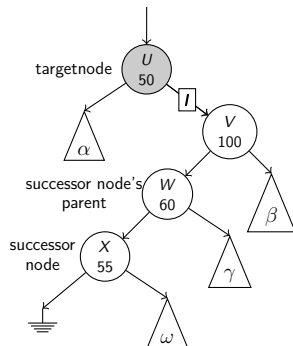
Lock Based BST - Simple Delete



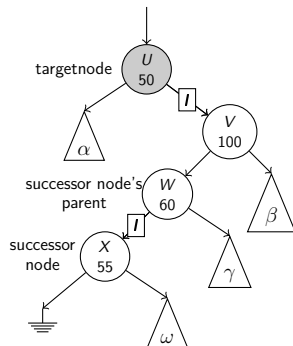
Lock Based BST - Complex Delete



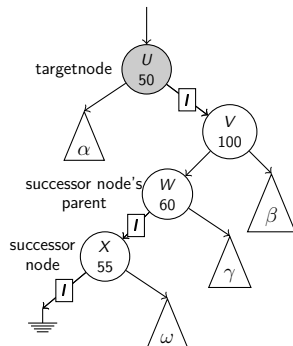
Lock Based BST - Complex Delete



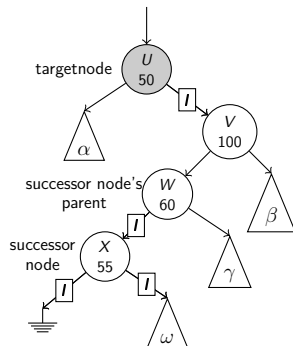
Lock Based BST - Complex Delete



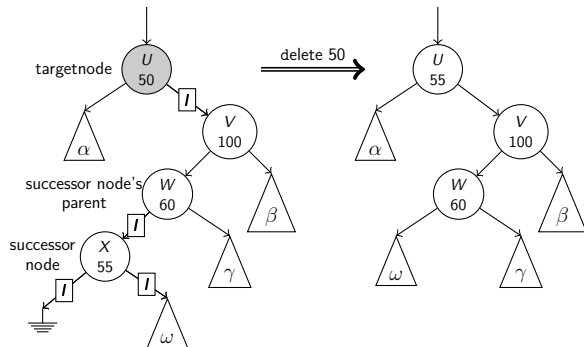
Lock Based BST - Complex Delete



Lock Based BST - Complex Delete

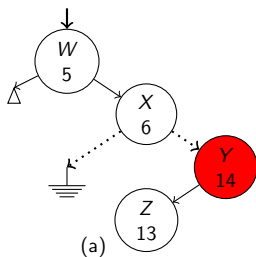


Lock Based BST - Complex Delete



Lock Based BST - More challenges in search

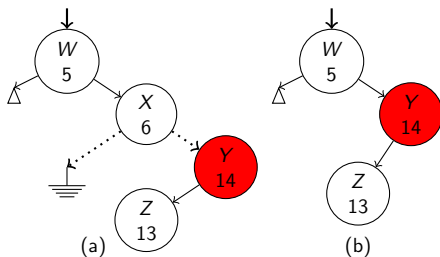
A scenario in which the last right turn node is removed



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

Lock Based BST - More challenges in search

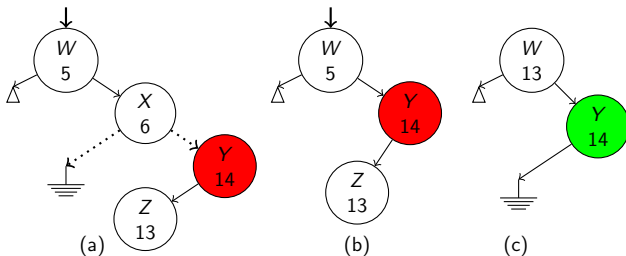
A scenario in which the last right turn node is removed



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

Lock Based BST - More challenges in search

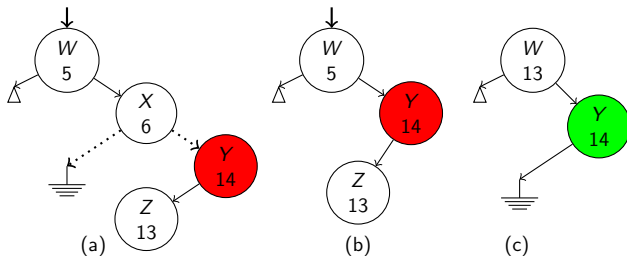
A scenario in which the last right turn node is removed



- 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



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

pseudocode for delete

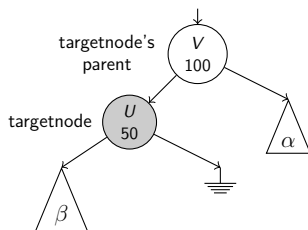
```
locate the node to delete;  
flag the children edges for deletion;  
if simple delete then  
| make the parent point to the non-null child atomically;  
else // complex delete  
| find the successor;  
| flag the children edges of successor for promotion;  
| promote key;  
| remove successor by a simple delete;  
| replace node with a fresh copy;  
end
```

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

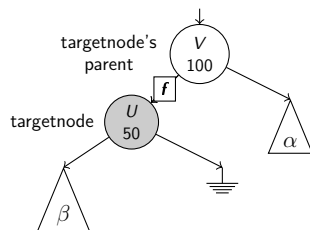
Lock Free BST - Simple Delete

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



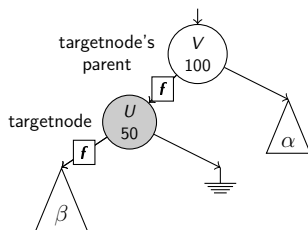
Lock Free BST - Simple Delete

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



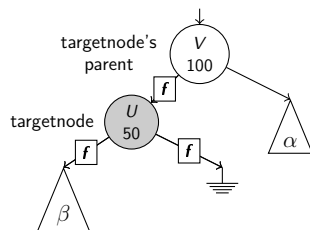
Lock Free BST - Simple Delete

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



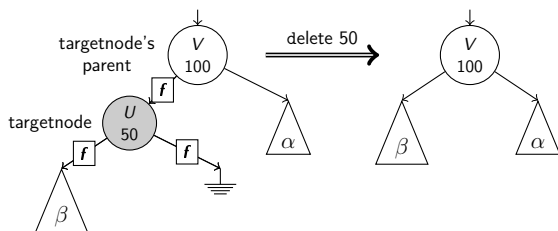
Lock Free BST - Simple Delete

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

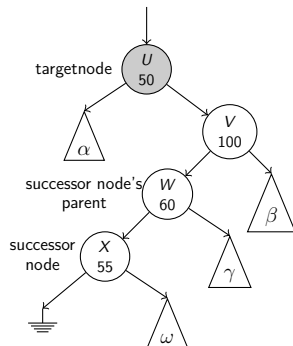


Lock Free BST - Simple Delete

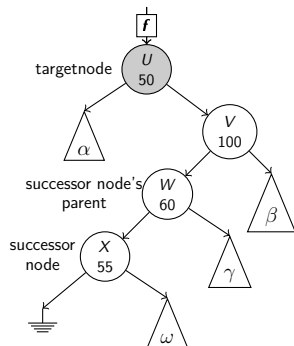
- ▶ flag is owned by an operation
- ▶ if a thread which installed the flag is stalled, other threads can help complete the operation



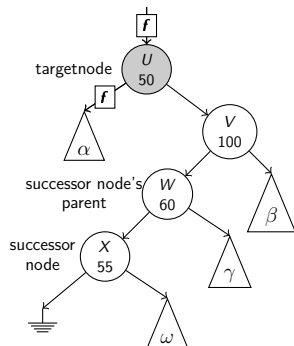
Lock Free BST - Complex Delete



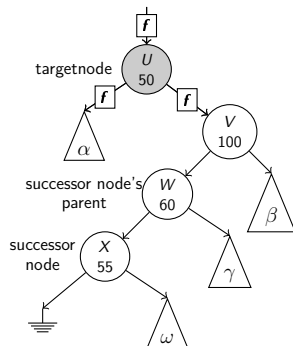
Lock Free BST - Complex Delete



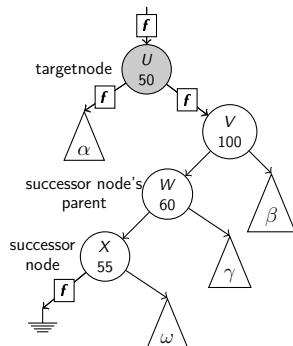
Lock Free BST - Complex Delete



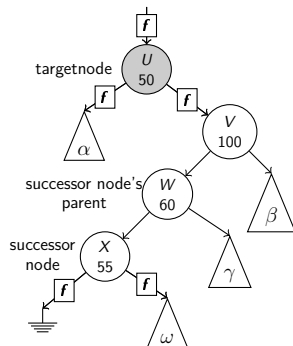
Lock Free BST - Complex Delete



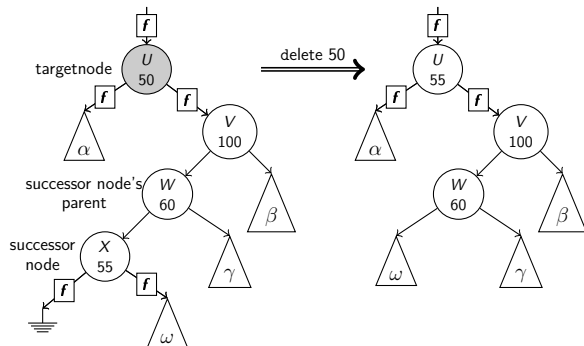
Lock Free BST - Complex Delete



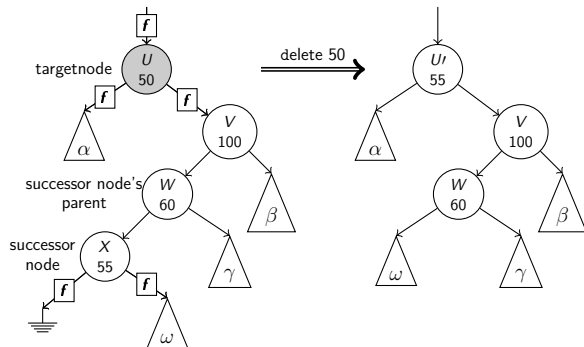
Lock Free BST - Complex Delete



Lock Free BST - Complex Delete



Lock Free BST - Complex Delete



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

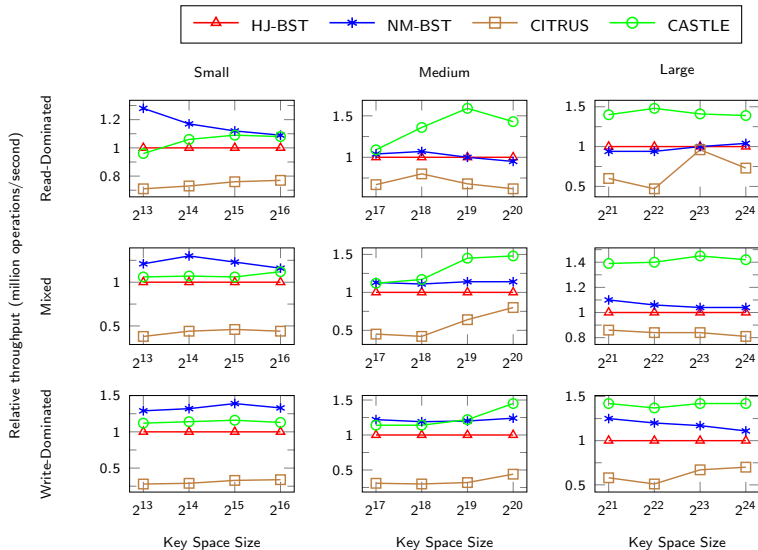
Experimental Setup

- ▶ Throughput computed as millions of operations per second (MOPS)
- ▶ each trial was run for 10 seconds
- ▶ Average over 3 trials
- ▶ *pre-populated* the tree to 50% of its maximum size to capture steady state behaviour
- ▶ beginning of each run consisted of a “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

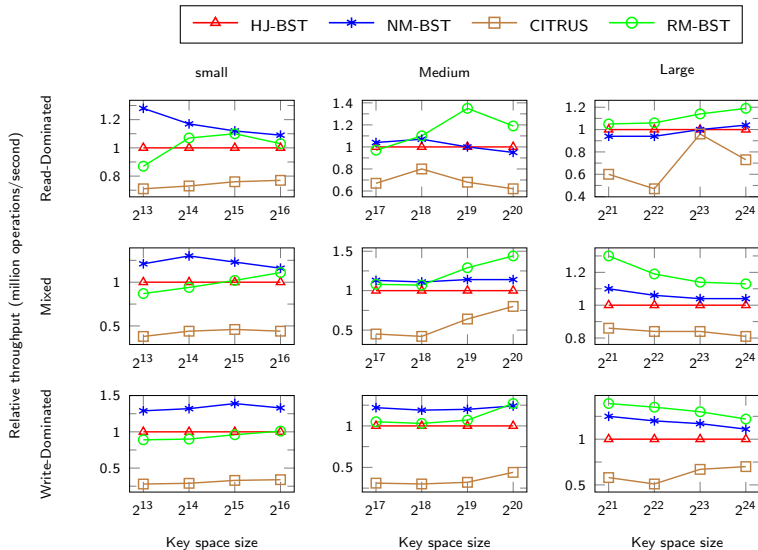
Other Concurrent BSTs

- ▶ a lock-free internal BST by Howley and Jones[SPAA'12], denoted by HJ-BST
- ▶ a lock-free external BST by Natarajan and Mittal[PPoPP'14], denoted by NM-BST
- ▶ RCU-based internal BST by Arbel and Attiya[PODC'14], denoted by CITRUS

Lock Based BST



Lock Free BST



Results Summary

Comparison of different lock-free algorithms in the absence of contention

Algorithm	Number of Objects Allocated		Number of Atomic Instructions Executed	
	Insert	Delete	Insert	Delete
HJ-BST	2	simple : 1 complex: 1	3	simple : 4 complex: 9
NM-BST	2	0	1	3
CASTLE (Lock Based BST)	1	simple : 0 complex: 0	1	simple : 3 complex: 4
RM-BST (Lock Free BST)	1	simple : 0 complex: 1	1	simple : 4 complex: 7

Results Summary

Comparison of different lock-free algorithms in the absence of contention

Algorithm	Number of Objects Allocated		Number of Atomic Instructions Executed	
	Insert	Delete	Insert	Delete
HJ-BST	2	simple : 1 complex: 1	3	simple : 4 complex: 9
NM-BST	2	0	1	3
CASTLE (Lock Based BST)	1	simple : 0 complex: 0	1	simple : 3 complex: 4
RM-BST (Lock Free BST)	1	simple : 0 complex: 1	1	simple : 4 complex: 7

- speedup is calculated over the second best algorithm

Workload	Speedup	
	Lock Based BST	Lock Free BST
Read-Dominated	59%	35%
Mixed	39%	26%
Write-Dominated	28%	13%

Future Work

- ▶ do local recovery upon failures
- ▶ develop concurrent K-ary BST which can improve spatial locality
- ▶ work on other data structures like tries

Future Work - Local Recovery

- ▶ currently upon failure, an operation restarts from the root
- ▶ Ellen et.al[PODC'14] have shown that local recovery can be done for external BST
- ▶ Local recovery on an internal BST is hard due to key movements
- ▶ We are currently working on extending our algorithms to enable local recovery

Future Work - K-ary BST

- ▶ ideas from Lock Based BST can be extended to external K-ary BST
- ▶ updates are relatively easier to handle as they obtain locks
- ▶ inserts might result in node splits
- ▶ searches are hard if we need to maintain their lock-free property

Future Work - Tries

- ▶ Tries are extensively used in text processing
- ▶ Tree like structure. So our ideas *can possibly* be applied