SkipLists and Balanced Search

The Art Of Multiprocessor Programming
Chapter 14

Rana Shahout
Sets

- Collection of distinct objects

**Methods**: Add, Remove and Contains

- And the most popular method is? Contains
 Concurrent 
Sets

Concurrent Search in 
logarithmic time
Implementation of Sets

- Linked List
- Hash Tables
- Search complexity? $O(n)$
Suggestions?

• First Try: AVL Trees

• Problem?

  Rebalancing to maintain the structure hurts in concurrent structures, causing bottlenecks and contention
Suggestions?

- Second Try: SkipList
- No need to rebalance

Provide logarithmic time search
What’s On The Menu

• Sequential Skiplist

• Concurrent Set implementation with Skiplist
  • Lock Based Concurrent Skiplist
  • Lock-Free Concurrent Skiplist
Sequential SkipList
Sequential SkipList

- Collection of levels, each one is a sorted linked lists
- Each link in level i skips over $2^i$ nodes
- Probabilistic data structure, means each node is created with a random top level

https://en.wikipedia.org/wiki/Skip_list#/media/File:Skip_list.svg
SkipList Property

- Level 0 contains all nodes
- Each layer is a sublist of lower levels.
Find(8)

10 > 8, FAR!
Lets Go Down ↓
Find(8)

7 < 8, OK!
Let's Go Right
Find(8)

10 > 8, FAR!
Let's Go Down

0 → 2 → 4 → 7 → 8 → 10
Find(8)

10 > 8, FAR!
Lets Go Down
Find(8)

8 == 8, OK!
Find(8)
Find(key) output

• Two arrays - predecessors and successors
  • pred[i] - reference to node in level i that has biggest key < searched key
  • succ[i] - reference to node in level i that has smallest key > searched key
  • if found returns searched key top level, otherwise, returns -1.
Find(7)
Add(key)

- Call find(key) to get preds[] and succ[]
- If find returns $v \neq -1$, then key is already in
- Otherwise, generate random top level $l$, insert key
  between pred[i] and succ[i] in every level between
  0 and $l$. 
Remove(key)

• Call find(key) to get preds[] and succ[]

• If find(key) returns v == -1, then key isn’t in the list

• Otherwise, redirect each predecessor’s next reference to victim’s successor
Lazy SkipList

Lock Based Concurrent Skiplist
Method Strategies

Add() and Remove() use Optimistic fine grained locking:

Optimistic Synchronization [Herlihy-Shavit 2008]

- Find nodes without locking
- Lock nodes
- Check that everything is OK

This side is taken from Erez Petrank: http://www.cs.technion.ac.il/~erez/courses/seminar18/talks/05.pdf
Method Strategies

Contains() is wait-free

Wait-Freedom:

If you schedule enough steps of any thread, it will make progress.

This side is taken from Erez Petrank: http://www.cs.technion.ac.il/~erez/courses/seminar18/talks/05.pdf
Locked Base Concurrent SkipList

- Node

  - lock
  - marked
  - fullyLinked

if marked is set - the node is logically removed

if fullyLinked is set - the node is linked in all its levels

A node is in the set if, and only if, unmarked + fullyLinked
Locked Base Concurrent SkipList

- Each level is a LazyList

- Maintains the skipList property: Higher-level lists are always contained in lower-level lists
Add(key)

1. Choose random top level

2. Call Find() to check if the node is already in the skiplist, gets preds[] and succs[]

3. Lock preds[] in ascending order and validate

4. Allocate new node with random top level

5. Link preds[] next to node, link node to succ[]

6. Set FullyLinked field, Linearization point of successful Add()

7. Release locks
Add(key)

1. Choose random top level

2. Call Find() to check if the node is already in the skiplist, gets preds[] and succs[]

3. if the node unmarked and fullyLinked then return false (The node is not fullyLinked, wait until it is linked and return false

4. Allocate new node with random top level

5. Link preds[] next to node, link node to succ[]

6. Set FullyLinked field, Linearization point of successful Add() call

7. Release locks
Add(key)

1. Choose random top level

2. Call Find() to check if the node is already in the skiplist, gets preds[] and succs[]

3. Lock preds[] in ascending order and validate

4. Allocate new node with random top level

5. Link preds[] next to node, link node to succ[]

6. Set FullyLinked field, Linearization point of successful Add()

7. Release locks

In Ascending Order! Why?

To avoid deadlock with remove
Add(key)

1. Choose random top level

2. Call Find() to check if the node is already in the skiplist, gets preds[] and succs[]

3. Lock preds[] in ascending order and validate

4. Allocate new node with random top level

5. Link preds[] next to node, link node to succ[]

6. Set FullyLinked field, Linearization point of successful Add()

7. Release locks
Add(key)

1. Choose random top level

2. Call Find() to check if the node is already in the skiplist, gets preds[] and succs[]

3. Lock preds[] in ascending order and validate

4. Allocate new node with random top level

5. Link preds[] next to node, link node to succ[]

6. Set FullyLinked field, Linearization point of successful Add()

7. Release locks
Add(key)

1. Choose random top level
2. Call Find() to check if the node is already in the skiplist, gets preds[] and succs[]
3. Lock preds[] in ascending order and validate
4. Allocate new node with random top level
5. Link preds[] next to node, link node to succ[]
6. Set FullyLinked field
7. Release locks

Why we don't lock succs??
Add(key)

1. Choose random top level
2. Call Find() to check if the node is already in the skiplist, gets preds[] and succs[]
3. Lock preds[] in ascending order and validate
4. Allocate new node with random top level
5. Link preds[] next to node, link node to succ[]
6. Set FullyLinked field, Linearization point of successful Add()
7. Release locks
Add(key)

1. Choose random top level

2. Call Find() to check if the node is already in the skiplist, gets preds[] and succs[]

3. Lock preds[] in ascending order and validate

4. Allocate new node with random top level

5. Link preds[] next to node, link node to succ[]

6. Set FullyLinked field, Linearization point of successful Add()

7. Release locks
Add(key)

• The only time a thread modifies an unlocked node's next field is when it initializes the new node's next references.

• Why it is safe?

   It occurs before the new node is accessible
Add(5)

1. Choose random top level = 4

Assume all the nodes are unmarked + fullyLinked
Add(5)

Call Find() to check if 5 is already in the set
Add(5)

Get preds[] and succs[]
Add(5)

Get `preds[]` and `succe[]`
Add(5)

Lock preds[]
Add(5)

Validate preds[]
Add(5)

Allocate new node, top level=4
Add(5)
Add(5)

Link preds[] next to node
Link node next to succ[]
Add(5)

Release locks
Add(5)
Remove(key)

1. Call Find() to check if the node is in the skiplist, get preds[] and succs[]

2. Lock node

3. Check if the node is unmarked and FullyLinked

4. Mark the node - logically deleting

5. Lock preds[] up to node to level and validate

6. Remove node at a time from top to bottom, why?

7. Release all the locks

If node is marked, OR not fullyLinked return false.

If the node unmarked, fullyLinked then the node is ready to be deleted.

If the node unmarked and not fullyLinked

Then another thread is adding it but not reached linearization point

returns false
Remove(key)

1. Call Find() to check if the node is in the skiplist, get preds[] and succs[]
2. Lock node
3. Check if the node is unmarked and FullyLinked
4. Mark the node - logically deleting
5. Lock preds[] up to node to level and validate
6. Remove node at a time from top to bottom, why?
7. Release all the locks
Remove(key)

1. Call Find() to check if the node is in the skiplist, get preds[] and succs[]

2. Lock node

3. Check if the node is unmarked and FullyLinked

4. Mark the node - logically deleting

5. Lock preds[] up to node to level and validate

6. Remove node at a time from top to bottom, why?

7. Release all the locks
Remove(key)

1. Call Find() to check if the node is in the skiplist, get preds[] and succs[]

2. Lock node

3. Check if the node is unmarked and FullyLinked

4. Mark the node - logically deleting

5. Lock preds[] up to node to check node is deleted

6. Remove node at a time from top to bottom, why?

7. Release all the locks

Linearization point of successful remove
Remove(key)

1. Call Find() to check if the node is in the skiplist, get preds[] and succs[]

2. Lock node

3. Check if the node is unmarked and FullyLinked

4. If not marked, marks the node - logically deleting

5. Lock preds[] up to node top level and validate

6. Remove node at a time from top to bottom, why?

7. Release all the locks

In ascending order to avoid deadlock
Remove(key)

1. Call Find() to check if the node is in the skiplist, get preds[] and succs[]
2. Lock node
3. Check if the node is unmarked and FullyLinked
4. If not marked, marks the node - logically deleting
5. Lock preds[] up to node to level and validate
6. Remove node at a time from top to bottom, why?
7. Release all the locks

Maintain Skiplist property
Remove(key)

1. Call Find() to check if the node is in the skiplist, get preds[] and succs[]
2. Lock node
3. Check if the node is unmarked and FullyLinked
4. If not marked, marks the node - logically deleting
5. Lock preds[] up to node to level and validate
6. Remove node at a time from top to bottom, why?
7. Release all the locks
Remove(key)

1. Call Find() to check if the node is in the skiplist, get preds[] and succs[]

2. Lock node

3. Check if the node is unmarked and FullyLinked

4. If not marked, marks the node - logically deleting

5. Lock preds[] up to node to level and validate

6. Remove node at a time from top to bottom, why?

7. Release all the locks
Remove(5)

Call Find() to check if 5 is already in the set
Remove(5)

Get preds[] and succs[]
Remove(5)

Get `preds[]` and `succs[]`
Remove(5)

Lock node

unmarked
marked
fullyLinked - 1
Remove(5)

Check if the node is unmarked and FullyLinked
Remove(5)

Mark the node
Remove(5)

Lock preds[]

[Diagram of linked list with nodes marked and unmarked, showing the process of removal]
Remove(5)

Validate preds are still unamrked and refers to node
Remove(5)

Remove node at a time from top to bottom
Remove node at a time from top to bottom

Remove(5)
Remove(5)

Remove node at a time from top to bottom
Remove(5)

Remove node at a time from top to bottom

fullyLinked - 1

unmarked

marked
Remove(5)

Release all the locks.
Contains(node)

• Call Find()

• Check whether it is unmarked and fullyLinked
Remove(8)

Remove(18) succeeds because its fullyLinked now

Remove(18)

Add(18) Fail!

Unmarked, fullyLinked
Lock node, set mark bit

Not fullyLinked, Add(18) Spin
If the validation fails at any level, release the locks for the predecessors (not for the node!) and call find to acquire the new set of predecessors!
The fullyLinked flags acts like?
Lazy SkipList - Summary

- Holds lock on all locations to be modified
- Validate nothing important has changed
- Complete Modifications
- Release the locks
LockFree SkipList

Lock-Free Concurrent SkipList
Reminder: Lock-Free

If you schedule enough steps across all threads, one of them will make progress.

This definition is taken from Erez Petrank: http://www.cs.technion.ac.il/~erez/courses/seminar18/talks/05.pdf
Lock-Free SkipList

• Each level is a LockFreeList

• No locks thus manipulating references at all the levels at the same time is impossible
No SkipList Property

• The abstract set is defined by the bottom-level list

• Nodes in higher-level than bottom-level serve only as shortcuts to bottom level.

• Does we really need now fullyLinked flags from the previous approach?

   NO
Reminder: LockFreeList

**Remove()**: Mark & Delete

**Add()**: Use CAS to insert

**Find()**: When you find a “logically” deleted node in your path:

- Finish the job: CAS the predecessor’s next field.
- Proceed (repeat as needed).

This side is taken from Erez Petrark: [http://www.cs.technion.ac.il/~erez/courses/seminar18/talks/05.pdf](http://www.cs.technion.ac.il/~erez/courses/seminar18/talks/05.pdf)
SkipList Operation Overview

• Add(): Use CAS to insert at each level.

• Remove(): Mark node’s next reference.

• Find(): Traverse the skipList, fill preds[] and succs[] also cleans up marked nodes!
Add(key)

1. Call find() Check if a node with key is already in the skipList (at the bottom-level!)

2. Prepare the new node with random top-level and key, node next references are unmarked.

3. Link next references to succs[].

4. Link the new node into the bottom-level (same as LockFreeList) using CAS while validating that the nodes still refer one to other. Logical add.

5. Link the node to higher levels.

If unmarked found at **bottom level** return false

unsuccessful add()'s linearization point
Add(key)

1. Call find() Check if a node with key is already in the skipList (at the bottom-level!)

2. Prepare the new node with random top-level and key, node next references are unmarked

3. Link next references to succs[]

4. Link the new node into the bottom-level (same as LockFreeList) using CAS while validating that the nodes still refer one to other.Logical add

5. Link the node to higher levels.
Add(key)

1. Call find() Check if a node with key is already in the skipList (at the bottom-level!)

2. Prepare the new node with random top-level and key, node next references are unmarked

3. Link next references to succ[]

4. Link the new node into the bottom-level (same as LockFreeList) using CAS while validating that the nodes still refer one to other. Logical add

5. Link the node to higher levels.
Add(key)

1. Call find() Check if a node with key is already in the skipList (at the bottom-level!)

2. Prepare the new node with random top-level and key, node next references are unmarked

3. Link next references to succs[]

4. Link the new node into the bottom-level (same as LockFreeList) using CAS while validating that the nodes still refer one to other. Logical add

5. Link... If CAS fails, restart, else the key is added
Add(key)

1. Call find() Check if a node with key is already in the skipList (at the bottom-level!)

2. Prepare the new node with random top-level and key, node next references are unmarked

3. Link next references to succs[]

4. Link the new node into the bottom-level (same as LockFreeList) using CAS while validating that the nodes still refer one to other. Logical add

5. Link the node to higher levels.
Add(key)

1. Call find() Check if a node with key is already in the skipList (at the bottom-level!)

2. Prepare the new node with random top-level and key, node next references are unmarked

3. Link next references to succs[]

4. Link the new node into the bottom-level (same as LockFreeList) using CAS while validating that the nodes still refer one to other. Logical add

5. Link the node to higher levels.
Remove(key)

1. Call Find() to check if there is unmarked node with key in bottom-level

2. Mark next pointers from its top level, not including the bottom-level
   - By marking next pointer, as in LockFreeList

3. Mark the bottom level, logical remove

4. Physical remove is done by remove() itself or find() of other threads.
   - Optimization?

   If not founded, or the node is marked, return false

   unsuccessful remove()'s linearization point
Remove(key)

1. Call Find() to check if there is unmarked node with key in bottom-level

2. Mark next references from its top level, not including the bottom-level
   • By marking next pointer, as in LockFreeList

3. Mark the bottom level, logical remove

4. Physical remove is done by remove() itself or find() of other threads.
   • Optimization?
Remove(key)

1. Call Find() to check if there is unmarked node with key in bottom-level

2. Mark next references from its top level, not including the bottom-level
   • By marking next pointer, as in LockFreeList

3. Mark the bottom level, logical remove

4. Physical remove is done by remove() itself or find() of other threads.
   • Optimization?
Remove(key)

1. Call Find() to check if there is unmarked node with key in bottom-level

2. Mark next references from its top level, not including the bottom-level
   - By marking next pointer, as in LockFreeList

3. Mark the bottom level, logical remove

4. Physical remove is done by remove() itself or find() of other threads.
   - Optimization?

Call find() before return true
Remove(key)

1. Call Find() to check if there is unmarked node with key in bottom-level

2. Mark next references from its top level, not including the bottom-level
   - By marking next pointer, as in LockFreeList

3. Mark the bottom level, logical remove

4. Physical remove is done by remove() itself or find() of other threads.
   - Optimization?
CAS Failures

- Remove():
  - **CAS failure + next reference is marked:**
    - Return false, why?
  - **CAS failure + next reference is unmarked:**
    - Retry marking
      Why there is no need to call find() again?
CAS Failures

• Add():
  • It means the predecessors and successors might have changed, call find() again!
Find(key)

• Traverse the list, from top-level to bottom.
• Ignore keys of marked nodes.
• Fill preds[] and succs[]
• Eliminate marked nodes.
• If key in bottom level, return true.
Contains(key)

- Traverse the list, from top-level to bottom
- Ignore keys of marked nodes.
Remove(2)  
Remove(9)  
Remove(15)  

Add(12) calls find(12)  

0 : unmarked  
1: marked
Summary

• SkipList provides logarithmic search complexity
• Two different concurrent implementation
• One is Lock Based
• The other is Lock Free
• Both have wait free contains()
Questions?!