Ask Question

148

I recently came across the data structure known as a Skip list. They seem to have very similar behavior to a binary search tree... my question is - why would you ever want to use a skip list over a binary search tree?

algorithm language-agnostic data-structures binary-tree skip-lists

118

share improve this question

edited Jun 13 '10 at 2:33
Lazer
28.8k 80 226 318

asked Nov 2 '08 at 4:39
Claudiu
108k 103 350 538

You might want to look at splay trees too. They are also quite easy to implement and tend toward balance. I would try to avoid randomized approximation algorithms (e.g., skip lists) if you're going to write unit tests for the data structure. – Cybis Nov 2 '08 at 5:30

1 If you're abandoning a perfectly good solution because it might be hard to write unit tests for it, you're doing it wrong. Unit tests are supposed to serve your application and not the other way round. – Seun Osewa May 26 '10 at 5:57

Besides, the randomization in the algorithm serves a purpose that is internal to the skip list. Its interface remains the same, which is the interface of a sorted dictionary kind of structure. The randomization does not affect the expected behavior. Unit tests are supposed to test the public behavior of a data structure as described by its interface to the client code, and not the internals of the implementation. – Ernesto Apr 5 '12 at 18:12

add a comment

7 Answers

active oldest votes

187

Skip lists are more amenable to concurrent access/modification. Herb Sutter wrote an article about data structure in concurrent environments. It has more indepth information.

The most frequently used implementation of a binary search tree is a red-black tree. The concurrent problems come in when the tree is modified it often needs to rebalance. The rebalance operation can affect large portions of the tree, which would require a mutex lock on many of the tree nodes. Inserting a node into a skip list is far more localized, only nodes directly linked to the affected node need to be locked.

Update from Jon Harrops comments

I read Fraser and Harris's latest paper Concurrent programming without locks. Really good stuff if you're interested in lock-free data structures. The paper focuses on Transactional Memory and a theoretical operation multiword-compare-and-swap MCAS. Both of these are simulated in software as no hardware supports them yet. I'm fairly impressed that they were able to build MCAS in software at all.

I didn't find the transactional memory stuff particularly compelling as it requires a garbage collector. Also software transactional memory is plagued with performance issues. However, I'd be very excited if hardware transactional memory ever becomes common. In the end it's still research and won't be of use for production code for another decade or so.

In section 8.2 they compare the performance of several concurrent tree implementations. I'll summarize their findings. It's worth it to download the pdf as it has some very informative graphs on pages 50, 53, and 54.

- **Locking skip lists** are insanely fast. They scale incredibly well with the number of concurrent accesses. This is what makes skip lists special, other lock based data structures tend to croak under pressure.
- **Lock-free skip lists** are consistently faster than locking skip lists but only barely.
- **transactional skip lists** are consistently 2-3 times slower than the locking and non-locking versions.
- **locking red-black trees** croak under concurrent access. Their performance degrades linearly with each new concurrent user. Of the two known locking red-black tree implementations, one essentially has a global lock during tree rebalancing. The other uses fancy (and complicated) lock escalation but still doesn't significantly out perform the global lock version.
- **lock-free red-black trees** don't exist (no longer true, see Update).
- **transactional red-black trees** are comparable with transactional skip-lists. That was very surprising and very promising. Transactional memory, though slower if far easier to write. It can be as easy as quick search and replace on the non-concurrent version.

Update
Here is paper about lock-free trees: Lock-Free Red-Black Trees Using CAS.
I haven't looked into it deeply, but on the surface it seems solid.

share improve this answer edited Jun 13 '12 at 7:02 answered Nov 3 '08 at 23:03

deft_code
29.6k 19 109 185

3 Not to mention that in a non-degenerate skiplist, about 50% of the nodes should only have a single link which makes insert and delete remarkably efficient. – Adisak Oct 30 '09 at 3:44

2 Rebalancing does not require a mutex lock. See cl.cam.ac.uk/research/srg/netos/lock-free – Jon Harrop May 20 '10 at 21:00

3 @Jon, yes and no. There are no known lock-free red-black tree implementations. Fraser and Harris show how a transactional memory based red-black tree is implemented and its performance. Transactional memory is still very much in the research arena, so in production code, a red-black tree will still need to lock large portions of the tree. – deft_code May 21 '10 at 16:20

1 I wanted to update this answer. There are currently two lock based efficient binary search trees. One is based on AVL trees (dl.acm.org/citation.cfm?id=1693488) and the other (Warning! shameless plug) is based on red black trees. See actapress.com/Abstract.aspx?paperId=453069 – Juan Besa Mar 2 '12 at 20:01

4 @deft_code: Intel recently announced an implementation of Transactional Memory via TSX on Haswell. This may prove interesting w.r.t those lock free data structures you mentioned. – Mike Bailey Oct 3 '12 at 5:07

show **7** more comments

35

First, you cannot fairly compare a randomized data structure with one that gives you worst-case guarantees.

A skip list is equivalent to a randomly balanced binary search tree (RBST) in the way that is explained in more detail in Dean and Jones' "Exploring the Duality Between Skip Lists and Binary Search Trees".

The other way around, you can also have deterministic skip lists which guarantee worst case performance, cf. Munro et al.

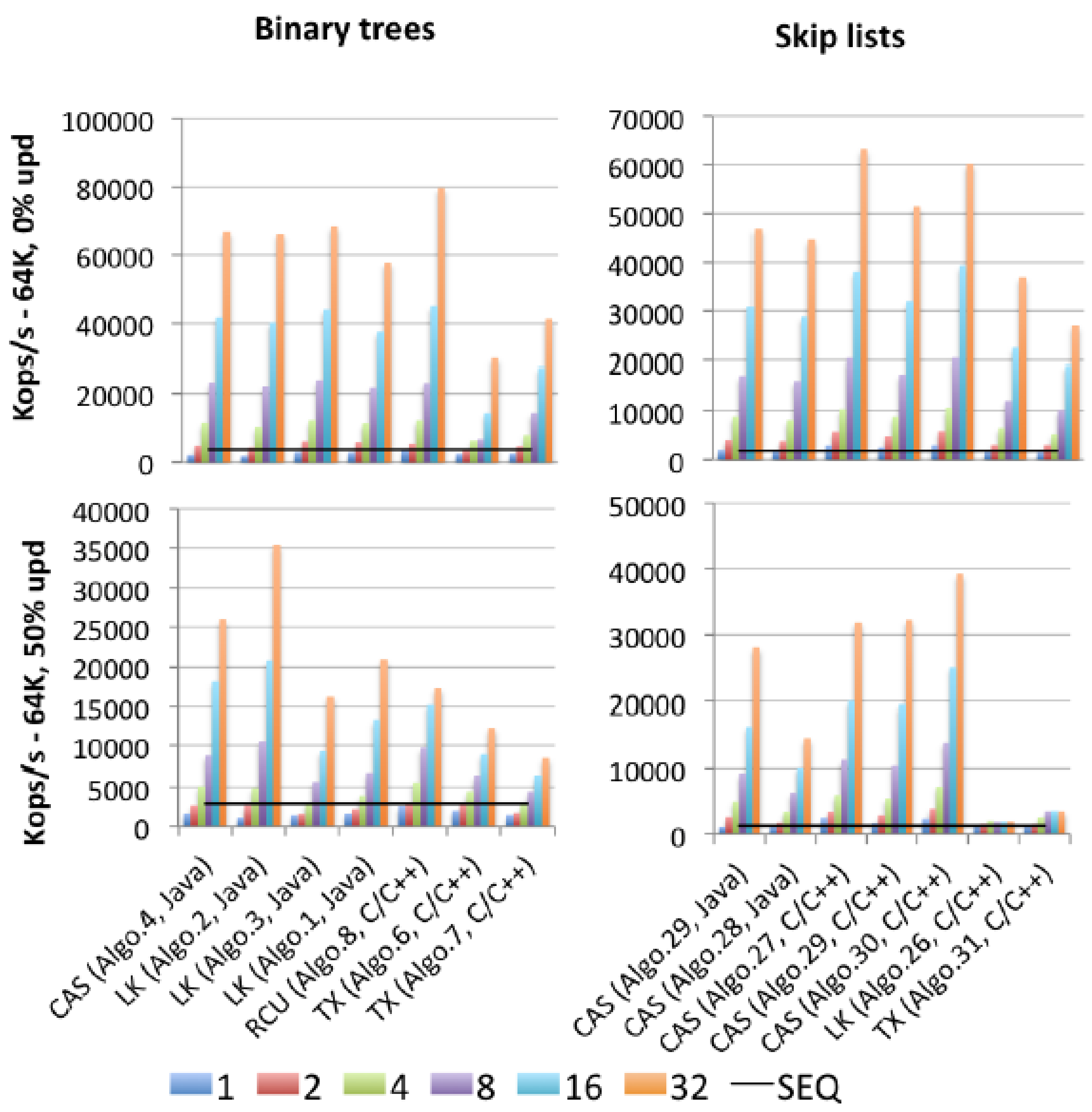
Contra to what some claim above, you can have implementations of binary search trees (BST) that work well in concurrent programming. A potential problem with the concurrency-focused BSTs is that you can't easily get the same had guarantees about balancing as you would from a red-black (RB) tree. (But "standard", i.e. randomzided, skip lists don't give you these guarantees either.) There's a trade-off between maintaining balancing at all times and good (and easy to program) concurrent access, so *relaxed* RB trees are usually used when good concurrency is desired. The relaxation consists in not re-balancing the tree right away. For a somewhat dated (1998) survey see Hanke's "The Performance of Concurrent Red-Black Tree Algorithms" [ps.gz].

One of the more recent improvements on these is the so-called *chromatic tree* (basically you have some weight such that black would be 1 and red would be zero, but you also allow values in between). And how does a chromatic tree fare against skip list? Let's see what Brown et al. "A General Technique for Non-blocking Trees" (2014) have to say:

> with 128 threads, our algorithm outperforms Java's non-blocking skiplist by 13% to 156%, the lock-based AVL tree of Bronson et al. by 63% to 224%, and a RBT that uses software transactional memory (STM) by 13 to 134 times

EDIT to add: Pugh's lock-based skip list, which was benchmarked in Fraser and Harris (2007) "Concurrent Programming Without Lock" as coming close to their own lock-free version (a point amply insisted upon in the top answer here), is also tweaked for good concurrent operation, cf. Pugh's "Concurrent Maintenance of Skip Lists", although in a rather mild way. Nevertheless one newer/2009 paper "A Simple Optimistic skip-list Algorithm" by Herlihy et al., which proposes a supposedly simpler (than Pugh's) lock-based implementation of concurrent skip lists, criticized Pugh for not providing a proof of correctness convincing enough for them. Leaving aside this (maybe too pedantic) qualm, Herlihy et al. show that their simpler lock-based implementation of a skip list actually fails to scale as well as the JDK's lock-free implementation thereof, but only for high contention (50% inserts, 50% deletes and 0% lookups)... which Fraser and Harris didn't test at all; Fraser and Harris only tested 75% lookups, 12.5% inserts and 12.5% deletes (on skip list with ~500K elements). The simpler implementation of Herlihy et al. also comes close to the lock-free solution from the JDK in the case of low contention that they tested (70% lookups, 20% inserts, 10% deletes); they actually beat the lock-free solution for this scenario when they made their skip list big enough, i.e. going from 200K to 2M elements, so that the probability of contention on any lock became negligible. It would have been nice if Herlihy et al. had gotten over their hangup over Pugh's proof and tested his implementation too, but alas they didn't do that.

EDIT2: I found a (2015 published) motherlode of all benchmarks: Gramoli's "More Than You Ever Wanted to Know about Synchronization. Synchrobench, Measuring the Impact of the Synchronization on Concurrent Algorithms": Here's a an excerpted image relevant to this question.



"Algo.4" is a precursor (older, 2011 version) of Brown et al.'s mentioned above. (I don't know how much better or worse the 2014 version is). "Algo.26" is Herlihy's mentioned above; as you can see it gets trashed on updates, and much worse on the Intel CPUs used here than on the Sun CPUs from the original paper. "Algo.28" is ConcurrentSkipListMap from the JDK; it doesn't do as well as one might have hoped compared to other CAS-based skip list implementations. The winners under high-contention are "Algo.2" a lock-based algorithm (!!) described by Crain et al. in "A Contention-Friendly Binary Search Tree" and "Algo.30" is the "rotating skiplist" from "Logarithmic data structures for multicores". "Algo.29" is the "No hot spot non-blocking skip list". Be advised that Gramoli is a co-author to all three of these winner-algorithm papers. "Algo.27" is the C++ implementation of Fraser's skip list.

Gramoli's conclusion is that's much easier to screw-up a CAS-based concurrent tree implementation than it is to screw up a similar skip list. And based on the figures, it's hard to disagree. His explanation for this fact is:

> The difficulty in designing a tree that is lock-free stems from the difficulty of modifying multiple references atomically. Skip lists consist of towers linked to each other through successor pointers and in which each node points to the node immediately below it. They are often considered similar to trees because each node has a successor in the successor tower and below it, however, a major distinction is that the downward pointer is generally immutable hence simplifying the atomic modification of a node. This distinction is probably the reason why skip lists outperform trees under heavy contention as observed in Figure [above].

Overriding this difficulty was a key concern in Brown et al.'s recent work. They have a whole separate (2013) paper "Pragmatic Primitives for Non-blocking Data Structures" on building multi-record LL/SC compound "primitives", which they call LLX/SCX, themselves implemented using (machine-level) CAS. Brown et al. used this LLX/SCX building block in their 2014 (but not in their 2011) concurrent tree implementation.

I think it's perhaps also worth summarizing here the fundamental ideas of the "no hot spot"/contention-friendly (CF) skip list. It addapts an essential idea from the relaxed RB trees (and similar concrrency friedly data structures): the towers are no longer built up immediately upon insertion, but delayed until there's less contention. Conversely, the deletion of a tall tower can create many contentions; this was observed as far back as Pugh's 1990 concurrent skip-list paper, which is why Pugh introduced pointer reversal on deletion (a tidbit that Wikipedia's page on skip lists still doesn't mention to this day, alas). The CF skip list takes this a step further and delays deleting the upper levels of a tall tower. Both kinds of delayed operations in CF skip lists are carried out by a (CAS based) separate garbage-collector-like thread, which its authors call the "adapting thread".

The Synchrobench code (including all algorithms tested) is available at: https://github.com/gramoli/synchrobench. The latest Brown et al. implementation (not included in the above) is available at http://www.cs.toronto.edu/~tabrown/chromatic/ConcurrentChromaticTreeMap.java Does anyone have a 32+ core machine available? J/K My point is that you can run these yourselves.

share improve this answer   edited Feb 2 '15 at 18:32   answered Feb 2 '15 at 2:59
Fizz
2,627 14 33

2 This answer should have more upvotes than it does – Travis Dec 13 '16 at 18:37

add a comment

11

Also, in addition to the answers given (ease of implementation combined with comparable performance to a balanced tree). I find that implementing in-order traversal (forwards and backwards) is far simpler because a skip-list effectively has a linked list inside its implementation.

share improve this answer   answered Nov 2 '08 at 6:32
Evan Teran
60.1k 18 147 210

1 isn't in-order traversal for a bin tree as simple as: "def func(node): func(left(node)); op(node); func(right(node))"? – Claudiu Nov 2 '08 at 18:35

6 Sure, that true if you want to traverse all in one function call. but it gets much more annoying if you want to have iterator style traversal like in std::map. – Evan Teran Nov 3 '08 at 4:20

@Evan :Not in a functional language where you can just write in CPS. – Jon Harrop May 20 '10 at 19:02

@Evan: `def iterate(node): for child in iterate(left(node)): yield child; yield node; for child in iterate(right(node)): yield child;` ? =). non-local control iz awesom.. @Jon: writing in CPS is a pain, but maybe you mean with continuations? generators are basically a special case of continuations for python. – Claudiu Sep 29 '10 at 20:50

1 @Evan: yes it works as long as the node parameter is cut out of the tree during a modification. The C++ traversal has the same constraint. – deft_code Nov 18 '10 at 0:27

show **3** more comments

8

In practice I've found that B-tree performance on my projects has worked out to be better than skip-lists. Skip lists do seem easier to understand but implementing a B-tree is not *that* hard.

The one advantage that I know of is that some clever people have worked out how to implement a lock-free concurrent skip list that only uses atomic operations. For example, Java 6 contains the ConcurrentSkipListMap class, and you can read the source code to it if you are crazy.

But it's not too hard to write a concurrent B-tree variant either - I've seen it done by someone else - if you preemptively split and merge nodes "just in case" as you walk down the tree then you won't have to worry about deadlocks and only ever need to hold a lock on two levels of the tree at a time. The synchronization overhead will be a bit higher but the B-tree is probably faster.

share improve this answer

answered Nov 16 '08 at 6:45
Jonathan
1,590 1 8 7

add a comment

8

From the Wikipedia article you quoted:

> Θ(n) operations, which force us to visit every node in ascending order (such as printing the entire list) provide the opportunity to perform a behind-the-scenes derandomization of the level structure of the skip-list in an optimal way, bringing the skip list to O(log n) search time. [...] A skip list, upon which we have not recently performed [any such] Θ(n) operations, **does not provide the same absolute worst-case performance guarantees as more traditional balanced tree data structures**, because it is always possible (though with very low probability) that the coin-flips used to build the skip list will produce a badly balanced structure

EDIT: so it's a trade-off: Skip Lists use less memory at the risk that they might degenerate into an unbalanced tree.

share improve this answer

edited Jun 11 '14 at 8:41
nawfal
34.7k 30 213 260

answered Nov 2 '08 at 4:47
Mitch Wheat
232k 30 369 471

this would be a reason against using the skip list. – Claudiu Nov 2 '08 at 4:50

7 quoting MSDN, "The chances [for 100 level 1 elements] are precisely 1 in 1,267,650,600,228,229,401,496,703,205,376". – peterchen Nov 2 '08 at 10:03

8 Why would you say that they use less memory? – Jonathan Nov 16 '08 at 6:46

1 @peterchen: I see, thanks. So this does not occur with deterministic skip lists? @Mitch: "Skip Lists use less memory". How do skip lists use less memory than balanced binary trees? Looks like they've got 4 pointers in every node and duplicate nodes whereas trees have only 2 pointers and no duplicates. – Jon Harrop May 21 '10 at 15:45

1 @Jon Harrop: The nodes at level one only need one pointer per node. Any nodes at higher levels need only two pointers per node (One to the next node and one to the level below it), though of course a level 3 node means you are using 5 pointers total for that one value. Of course, this will still suck up a lot of memory (moreso than a binary search if you want a non-useless skip list and have a large dataset)...but I think I'm missing something... – Brian Sep 7 '10 at 7:03

show **3** more comments

2

Skip lists are implemented using lists.

Lock free solutions exist for singly and doubly linked lists - but there are no lock free solutions which directly using only CAS for any O(logn) data structure.

You can however use CAS based lists to create skip lists.

(Note that MCAS, which is created using CAS, permits arbitrary data structures and a proof of concept red-black tree had been created using MCAS).

So, odd as they are, they turn out to be very useful :-)

share improve this answer

answered Mar 24 '09 at 20:21

Blank Xavier

4 "there are no lock free solutions which directly using only CAS for any O(logn) data structure". Not true. For counter examples see cl.cam.ac.uk/research/srg/netos/lock-free – Jon Harrop May 20 '10 at 21:02

add a comment

-1

Skip Lists do have the advantage of lock stripping. But, the runt time depends on how the level of a new node is decided. Usually this is done using Random(). On a dictionary of 56000 words, skip list took more time than a splay tree and the tree took more time than a hash table. The first two could not match hash table's runtime. Also, the array of the hash table can be lock stripped in a concurrent way too.

Skip List and similar ordered lists are used when locality of reference is needed. For ex: finding flights next and before a date in an application.

An inmemory binary search splay tree is great and more frequently used.

Skip List Vs Splay Tree Vs Hash Table Runtime on dictionary find op

share improve this answer

edited Apr 6 '12 at 8:17

answered Apr 5 '12 at 20:45

Harisankar Krishna Swamy
440 2 13

Harisankar Krishna Swamy
9 1

I took a quick look and your results seem to show SkipList as faster than SplayTree. – Chinasaur Aug 31 '13 at 21:35

It is misleading to assume randomisation as part of skip-list. How elements are skipped is crucial. Randomisation is added for probabilistic structures. – user568109 Jul 7 '14 at 8:59

add a comment