SOFTWARE ENGINEERING

Which self balancing binary tree would you recommend?

Asked 9 years, 2 months ago Active 3 years, 8 months ago Viewed 13k times

18

6

I'm learning Haskell and as an exercise I'm making binary trees. Having made a regular binary tree, I want to adapt it to be self balancing. So:

- Which is most efficient?
- Which is easiest to implement?
- Which is most often used?

But crucially, which do you recommend?

I assume this belongs here because it's open to debate.

haskell functional-programming data-structures binary-tree

asked Jan 1 '11 at 10:06

In terms of efficiency and ease of implementation the general efficiencies are well defined but for your implementation, I reckon the best thing would be to implement as many as you can find and then let us know which works best... – glenatron Jan 2 '11 at 8:49

6 Answers

Active Oldest Votes

15

I would recommend you start with either a Red-Black tree, or an AVL tree.

The red-black tree is faster for inserting, but the AVL tree has a slight edge for lookups. The AVL tree is probably a little easier to implement, but not by all that much based on my own experience.

The AVL tree ensures that the tree is balanced after each insert or delete (no sub-tree has a balance factor greater than 1/-1, while the Red-black tree ensures that the tree is reasonably balanced at any time.

edited Jan 1 '11 at 11:10

answered Jan 1 '11 at 11:01
Quick Joe Smith
293 2 6

1 Personally, I find red-black insert easier than AVL insert. The reason is through the (imperfect) analogy to B-trees. Inserts are fiddly, but deletes are *evil* (so many cases to consider). In fact I no longer have a C++ red-black delete implementation of my own - I deleted it when I realised (1) I was never using it - every time I wanted to delete I was deleting multiple items, so I converted from tree to list, delete from the list, then convert back to a tree, and (2) it was broken anyway. – Steve314 Jan 1 '11 at 11:43

2 @Steve314, red-black trees are easier, but you haven't been able to make an implementation that works? What are AVL trees like then? – dan_waterworth Jan 1 '11 at 14:01

@dan_waterworth - I haven't made an implementation with even an insert method that works yet - have notes, understand the basic principle, but never got the right combination of motivation, time and confidence. If I just wanted versions that work, that's just copy-pseudocode-from-textbook-and-translate (and don't forget C++ has standard library containers), but where's the fun in that? – Steve314 Jan 2 '11 at 10:09

BTW - I believe (but can't provide the reference) that a fairly popular textbook includes a buggy implementation of one of the balanced binary tree algorithms - not sure, but it might be red-black delete. So it's not just me ;-) – Steve314 Jan 2 '11 at 11:52

1 @Steve314, I know, trees can be fiendishly complicated in imperative language, but surprisingly, implementing them in Haskell has been a breeze. I've written a regular AVL tree and also a 1D spatial variant over the weekend and they're both only about 60 lines. – dan_waterworth Jan 3 '11 at 9:57

10

From a high-level point of view, it's a tree structure, except that it's not implemented as a tree but as a list with multiple layers of links.

You'll get O(log N) insertions / searches / deletes, and you won't have to deal with all those tricky rebalancing cases.

I've never considered implementing them in a Functional Language though, and the wikipedia page does not show any, so it may not be easy (wrt to immutability)

edited Jul 17 '16 at 8:55
user541686
7,466 5 33 48

answered Jan 1 '11 at 13:47
Matthieu M.
12.6k 4 39 57

I really enjoy skip lists and I've implemented them before, though not in a functional language. I think I'll attempt them after this, but right now I'm on self-balancing trees. – dan_waterworth Jan 1 '11 at 13:57

Also, people often use skiplists for concurrent data structures. It may be better, instead of forcing immutability, to use haskell's concurrency primitives (like MVar or TVar). Although, this won't teach me a lot about writing functional code. – dan_waterworth Jan 1 '11 at 14:08

2 @Fanatic23, a Skip List isn't an ADT. The ADT is either a set or an associative array. – dan_waterworth Jan 1 '11 at 14:12

@dan_waterworth my bad, you are correct. – Fanatic23 Jan 1 '11 at 14:53

5

If you want a relatively easy structure to start with (both AVL trees and red-black trees are fiddly), one option is a treap - named as a combination of "tree" and "heap".

Each node gets a "priority" value, often randomly assigned as the node is created. Nodes are positioned in the tree so that key ordering is respected, and so that heap-like ordering of priority values is respected. Heap-like ordering means that both children of a parent have lower priorities than the parent.

EDIT deleted "within key values" above - the priority and key ordering apply together, so priority is significant even for unique keys.

It's an interesting combination. If keys are unique and priorities are unique, there is a unique tree structure for any set of nodes. Even so, inserts and deletes are efficient. Strictly speaking, the tree can be unbalanced to the point where it is effectively a linked list, but this is extremely unlikely (as with standard binary trees), including for normal cases such as keys inserted in order (unlike standard binary trees).

edited Jan 1 '11 at 11:55

answered Jan 1 '11 at 11:33
Steve314

1 +1. Treaps is my personal choice, I even wrote a blog post about how they're implemented. – P Shved Jan 1 '11 at 20:40

5

> Which is most efficient?

Vague and difficult to answer. The computational complexities are all well-defined. If that's what you mean by efficiency, there's no real debate. Indeed, all good algorithms come with proofs and complexity factors.

If you mean "run time" or "memory use" then you'll need to compare actual implementations. Then language, run-time, OS and other factors come into play, making the question difficult to answer.

> Which is easiest to implement?

Vague and difficult to answer. Some algorithms may appear complex to you, but trivial to me.

> Which is most often used?

Vague and difficult to answer. First there's the "by whom?" part of this? Haskell only? What about C or C++? Second, there's the proprietary software problem where we don't have access to the source to do a survey.

> But crucially, which do you recommend?
>
> I assume this belongs here because it's open to debate.

Correct. Since your other criteria aren't very helpful, this is all you're going to get.

You can get source for a large number of tree algorithms. If you want to learn something, you might simply implement every one you can find. Rather than ask for a "recommendation", just collect every algorithm you can find.

Here's the list:

http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree

S.Lott
44.4k 6 85 151

3

If you're interested in Splay trees, there is a simpler version of those which I believe was first described in a paper by Allen and Munroe. It doesn't have the same performance guarantees, but avoids complications in dealing with "zig-zig" vs. "zig-zag" rebalancing.

Basically, when searching (including searches for an insert point or node to delete), the node you find gets rotated directly towards the root, bottom up (e.g. as a recursive search function exits). At each step, you select a single left or right rotation depending on whether the child you want to pull up another step toward the root was the right child or left child (if I remember my rotation directions correctly, that's respectively).

Like Splay trees, the idea is that recently accessed items are always near the root of the tree, so quick to access again. Being simpler, these Allen-Munroe rotate-to-root trees (what I call them - don't know the official name) can be faster, but they don't have the same amortized performance guarantee.

One thing - since this data structure by definition mutates even for find operations, it would probably need to be implemented monadically. IOW it's maybe not a good fit for functional programming.

edited Jan 1 '11 at 11:46

answered Jan 1 '11 at 11:39
Steve314
8,608 1 26 45

Splays are a bit annoying given they modify the tree even when finding. This would be pretty painful in multi-threaded environments, which is one of the big motivations for using a functional language like Haskell in the first place. Then again, I've never used functional languages before, so perhaps this isn't a factor. – Quick Joe Smith Jan 1 '11 at 12:00

@Quick - depends on how you intend to use the tree. If you were using it in true functional-style code, you'd either drop the mutation on every find (making a Splay tree a bit silly), or you'd end up duplicating a substantial part of the binary tree on each lookup, and keep track of which tree state you're working with as your work progresses (the reason for probably using a monadic style). That copying might be optimised away by the compiler if you no longer reference the old tree state after the new one is created (similar assumptions are common in functional programming), but it might not. – Steve314 Jan 2 '11 at 10:19

Neither approach sounds worth the effort. Then again, neither do purely functional languages for the most part. – Quick Joe Smith Jan 2 '11 at 12:09

1 @Quick - Duplicating the tree is what you'll do for any tree data structure in a pure functional language for mutating algorithms such as inserts. In source terms, the code won't be that different from imperative code that does in-place updates. The differences have already handled, presumably, for unbalanced binary trees. So long as you don't try to add parent links to nodes, the duplicates will share common subtrees at a minimum, and the

2

A very simple balanced tree is an AA tree. It's invariant is simpler and thus easier implement. Because of its simplicity, its performance is still good.

As an advanced exercise, you can try to use GADTs to implement one of the variants of balanced trees whose invariant is enforced by the type system type.

answered Feb 4 '13 at 7:56

Petr Pudlák

5,087 2 21 43