Why are Red-Black trees so popular?

It seems that everywhere I look, data structures are being implemented using red-black trees (`std::set` in C++, `SortedDictionary` in C#, etc.)

Having just covered (a,b), red-black & AVL trees in my algorithms class, here's what I got out (also from asking around professors, looking through a few books and googling a bit):

- AVL trees have smaller average depth than red-black trees, and thus searching for a value in AVL tree is consistently faster.
- Red-black trees make less structural changes to balance themselves than AVL trees, which could make them potentially faster for insert/delete. I'm saying potentially, because this would depend on the cost of the structural change to the tree, as this will depend a lot on the runtime and implemntation (might also be completely different in a functional language when the tree is immutable?)

There are many benchmarks online that compare AVL and Red-black trees, but what struck me is that my professor basically said, that usually you'd do one of two things:

- Either you don't really care that much about performance, in which case the 10-20% difference of AVL vs Red-black in most cases won't matter at all.
- Or you really care about performance, in which you case you'd ditch both AVL and Red-black trees, and go with B-trees, which can be tweaked to work much better (or (a,b)-trees, I'm gonna put all of those in one basket.)

The reason for that is because a B-tree stores data more compactly in memory (one node contains many values) there will be much fewer cache misses. You could also tweak the implementation based on the use case, and make the order of the B-tree depend on the CPU cache size, etc.

The problem is that I can't find almost any source that would analyze real life usage of different implementations of search trees on real modern hardware. I've looked through many books on algorithms and haven't found anything that would compare different tree variants together, other than showing that one has smaller average depth than the other one (which doesn't really say much of how the tree will behave in real programs.)

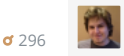
That being said, is there a particular reason why Red-black trees are being used everywhere, when based on what is said above, B-trees should be outperforming them? (as the only benchmark I could find also shows http://lh3lh3.users.sourceforge.net/udb.shtml, but it might just be a matter of the specific implementation). Or is the reason why everyone uses Red-black trees because they're rather easy to implement, or to put it in different words, hard to implement poorly?

Also, how does this change when one moves to the realm of functional languages? It seems that both Clojure and Scala use Hash array mapped tries, where Clojure uses a branching factor of 32.

`data-structures` `search-trees` `applied-theory` `dictionaries` `balanced-search-trees`

edited Apr 29 '15 at 18:03

Raphael ♦
55.4k 21 135 296

asked Apr 29 '15 at 17:50

Jakub Arnold
367 3 9

8 To add to your pain, most articles that compare different kinds of search trees perform ... less than ideal experiments. – Raphael ♦ Apr 29 '15 at 18:05

1 I've never understood this myself, in my opinion AVL trees are easier to implement than red-black trees (fewer cases when rebalancing), and I've never noticed a significant difference in performance. – Jordi Vermeulen Apr 29 '15 at 18:11

3 A relevant discussion by our friends at stackoverflow Why is std::map implemented as a red-black tree?. – Hendrik Jan Apr 30 '15 at 0:55

4 Answers

To quote from the answer to "Traversals from the root in AVL trees and Red Black Trees" question

> For some kinds of binary search trees, including red-black trees but not AVL trees, the "fixes" to the tree can fairly easily be predicted on the way down and performed during a single top-down pass, making the second pass unnecessary. Such insertion algorithms are typically implemented with a loop rather than recursion, and often run slightly faster in practice than their two-pass counterparts.

So a RedBlack tree insert can be implemented without recursion, on **some** CPUs recursion is **very** expensive if you overrun the function call cache (e.g SPARC due to is use of Register window)

(I have seen software run over 10 times as fast on the Sparc by removing one function call, that resulted in a often called code path being too deep for the register window. As you don't know how deep the register window will be on your customer's system, and you don't know how far down the call stack you are in the "hot code path", not using recursion make like more predictable.)

Also not risking running out of stack is a benefit.

edited Apr 13 '17 at 12:48

Community ♦
1

answered Jul 11 '15 at 13:22

Ian Ringrose
518 5 11

But a balanced tree with 2^32 nodes would require no more than about 32 levels of recursion. Even if your stack frame is 64 bytes, that's no more than 2 kb of stack space. Can that really make a difference? I would doubt it. – Björn Lindqvist Sep 17 '17 at 21:35

@BjörnLindqvist, On the SPARC processor in the 1990s I often got more then a 10 fold speed up by changing a common code path from a stack depth of 7 to 6! Read up on how it did register files.... – Ian Ringrose Sep 21 '17 at 22:34

I've been researching this topic recently as well, so here are my findings, but keep in mind that I am not an expert in data structures!

There are some cases where you can't use B-trees at all.

One prominent case is `std::map` from C++ STL. The standard requires that `insert` does not invalidate existing iterators

> No iterators or references are invalidated.

http://en.cppreference.com/w/cpp/container/map/insert

This rules out B-tree as an implementation because insertion would move around existing elements.

Another similar use case are intrusive datastructures. That is, instead of storing your data inside the node of the tree, you store pointers to children/parents inside your structure:

```
// non intrusive
struct Node<T> {
    T value;
    Node<T> *left;
    Node<T> *right;
};
using WalrusList = Node<Walrus>;

// intrusive
struct Walrus {
    // Tree part
    Walrus *left;
    Walrus *right;

    // Object part
    int age;
    Food[4] stomach;
};
```

You just can't make a B-tree intrusive, because it is not a pointer-only data structure.

Intrusive red-black trees are used, for example, in jemalloc to manage free blocks of memory. This is also a popular data structure in the Linux kernel.

I also believe that "single pass tail recursive" implementation is **not** the reason for red black tree popularity as a *mutable* data structure.

First of all, stack depth is irrelevant here, because (given $\log n$ height) you would run out of the main memory before you run out of stack space. Jemalloc is happy with preallocating worst case depth on the stack.

There are a number of flavors of red-black tree implementation. A famous one are left leaning red black trees by Robert Sedgewick (**CAUTION!** there are other variants which also are named "left leaning", but use a different algorithm). This variant indeed allows to perform rotations on the way down the tree, **but** it lack the important property of $O(1)$ amortized number of fixups, and this makes it slower (as measured by the author of jemalloc). Or, as opendatastrutures puts it

> Andersson's variant of red-black trees, Sedgewick's variant of red-black trees, and AVL trees are all simpler to implement than the RedBlackTree structure defined here. Unfortunately, none of them can guarantee that the amortized time spent rebalancing is $O(1)$ per update.

The variant described in opendatastructures uses parent pointers, a recursive down pass for insertion and an iterative loop up pass for fixups. The recursive calls are in a tail positions and compilers optimize this to a loop (I've checked this in Rust).

That is, you can get a constant memory loop implementation of a mutable search tree without any red-black magic if you use parent pointers. This works for B-trees as well. You need magic for single pass tail recursive immutable variant, and it will break $O(1)$ fixup anyway.

edited Nov 11 '16 at 11:53

answered Nov 10 '16 at 0:13

matklad
81 ● 1 ● 2

Well, this is not an authoritative answer, but whenever I have to code a balanced binary search tree, it's a red-black tree. There are a few reasons for this:

1) Average insertion cost is constant for red-black trees (if you don't have to search), while it's logarithmic for AVL trees. Furthermore, it involves at most one complicated restructuring. It's still O(log N) in the worst case, but that's just simple recolorings.

2) They require only 1 bit of extra information per node, and you can often find a way to get that for free.

3) I don't have to do this very often, so every time I do it I have to figure it out how to do it all over again. The simple rules and the correspondence with 2-4 trees makes it seem easy *every time*, even though the code turns out to be complicated *every time*. I still hope that someday the code will turn out simple.

4) The way the red-black tree splits the corresponding 2-4 tree node and inserts the middle key into the parent 2-4 node *just by recoloring* is super elegant. I just love to do it.

edited Mar 23 '16 at 2:08

answered Mar 23 '16 at 2:03

Matt Timmermans

269 ● 1 ● 4

Red-black or AVL trees have an advantage over B-trees and the like when the key is long or for some other reason moving a key is expensive.

I created my own alternative to `std::set` within a major project, for a number of performance reasons. I chose AVL over red-black for performance reasons (but that small performance enhancement was not the justification for rolling my own instead of std::set). The "key" being complicated and hard to move was a significant factor. Do (a,b) trees still make sense if you need another level of indirection in front of the keys? AVL and red-black trees can be restructured without moving keys, so they have that advantage when keys are expensive to move.

edited Jun 11 '15 at 7:42

Gilles ♦

31k ● 7 ● 85 ● 155

answered May 11 '15 at 12:48

JSF

1

Ironically, red-black trees are "only" a special case of (a,b)-trees, so the matter seems to come down to a tweaking of parameters? (cc @Gilles) – Raphael ♦ Jun 11 '15 at 8:12