Red-Black Trees

Asked 16 years, 4 months ago Modified 8 years, 1 month ago

Viewed 17k times

58

I've seen binary trees and binary searching mentioned in several books I've read lately, but as I'm still at the beginning of my studies in Computer Science, I've yet to take a class that's really dealt with algorithms and data structures in a serious way.

I've checked around the typical sources (Wikipedia, Google) and most descriptions of the usefulness and implementation of (in particular) Red-Black trees have come off as dense and difficult to understand. I'm sure for someone with the necessary background, it makes perfect sense, but at the moment it reads like a foreign language almost.

So what makes binary trees useful in some of the common tasks you find yourself doing while programming? Beyond that, which trees do you prefer to use (please include a sample implementation) and why?

algorithm binary-tree red-black-tree

edited Nov 3, 2016 at 16:34

Gaslight Deceive Subvert

Share

Improve this question

Follow

20.3k ● 20 ● 91 ● 129

asked Aug 21, 2008 at 18:37

wfarr

1,573 ● 1 ● 13 ● 13

12 Answers

Sorted by: Highest score (default)

58

Red Black trees are good for creating well-balanced trees. The major problem with binary search trees is that you can make them unbalanced very easily. Imagine your first number is a 15. Then all the numbers after that are increasingly smaller than 15. You'll have a tree that is very heavy on the left side and has nothing on the right side.

Red Black trees solve that by forcing your tree to be balanced whenever you insert or delete. It accomplishes this through a series of rotations between ancestor nodes and child nodes. The algorithm is actually pretty straightforward, although it is a bit long. I'd suggest picking up the CLRS (Cormen, Lieserson, Rivest and Stein) textbook, "Introduction to Algorithms" and reading up on RB Trees.

The implementation is also not really so short so it's probably not really best to include it here. Nevertheless, trees are used *extensively* for high performance apps that need access to lots of data. They provide a very efficient way of finding nodes, with a relatively small overhead of insertion/deletion. Again, I'd suggest looking at CLRS to read up on how they're used.

While BSTs may not be used explicitly - one example of the use of trees in general are in almost every single modern RDBMS. Similarly, your file system is almost certainly represented as some sort of tree structure, and files are likewise indexed that way. Trees power Google. Trees power just about every website on the internet.

Share Improve this answer Follow

edited Sep 11, 2013 at 17:43
Jacob
34.6k ● 15 ● 112 ● 167

answered Aug 21, 2008 at 18:56
FreeMemory
8,604 ● 7 ● 38 ● 50

It's worth noting that sometimes a RB Tree, or any self-balancing tree, can produce inferior performance to an unbalanced tree. This seeming contradiction can happen if an unbalanced tree is arranged so that the most frequently accessed nodes are closer to the root than a RB tree would have them. With a randomly distributed data source this happens more than you'd imagine. The important point being RB does not distribute according to access count. It distributes to provide the fastest access to any given key - regardless of how often any given key is retrieved.
– user1899861 Feb 20, 2015 at 0:58

---

1 Unbalanced trees being fed randomly distributed data are more likely to encounter frequently occurring data before rarely occurring data, thus the more frequently occurring data ends up closer to the root node than less frequently occurring

data. This is a subtle point to be sure, but it can be a very important point where resources are limited, or performance is critical. – user1899861 Feb 20, 2015 at 1:00

@RocketRoy Can you take a look at my red black tree remove method? stackoverflow.com/questions/28705454/… – committedandroider Feb 25, 2015 at 5:28

I just use the Map from C/C++ STL, but I believe this is the Java equivalent of that. docs.oracle.com/javase/7/docs/api/java/util/TreeMap.html. If you Google around you might be able to find the source code for this Java class - or the online source for Mark Allen Weiss' book. Sorry to punt your request, but it's been over 20 years since I wrote a self-balancing tree and I'm afraid you've caught me flat-footed and busy. Hope this helps a little. – user1899861 Feb 25, 2015 at 8:37

"one example of the use of trees in general are in almost every single modern RDBMS" - this one is inaccurate. DB use B-trees. – Viet Dec 6, 2016 at 4:23

20

I'd like to address only the question "So what makes binary trees useful in some of the common tasks you find yourself doing while programming?"

This is a big topic that many people disagree on. Some say that the algorithms taught in a CS degree such as binary search trees and directed graphs are not used in day-to-day programming and are therefore irrelevant. Others disagree, saying that these algorithms and data structures are the foundation for all of our programming and it is essential to understand them, even if you never have to write one for yourself. This filters into conversations about good interviewing and hiring practices. For example, Steve Yegge has an article on interviewing at Google that addresses this question. Remember this debate; experienced people disagree.

In typical business programming you may not need to create binary trees or even trees very often at all. However, you will use many classes which internally operate using trees. Many of the core organization classes in every language use trees and hashes to store and access data.

If you are involved in more high-performance endeavors or situations that are somewhat outside the norm of business programming, you will find trees to be an immediate friend. As another poster said, trees are core data structures for databases and indexes of all kinds. They are useful in data mining and visualization, advanced graphics (2d and 3d), and a host of other computational problems.

I have used binary trees in the form of BSP (binary space partitioning) trees in 3d graphics. I am currently looking at trees again to sort large amounts of geocoded data and other data for information visualization in Flash/Flex applications. Whenever you are pushing the boundary of the hardware or you want to run on lower hardware specifications, understanding and selecting the best algorithm can make the difference between failure and success.

Share Improve this answer Follow

answered Aug 21, 2008 at 19:27

Jonathan Branam
744 ●5 ●10

@user347594 Can you check out my remove method in RedBlackTree? stackoverflow.com/questions/28705454/…
– committedandroider Feb 25, 2015 at 19:02

12

None of the answers mention what it is exactly BSTs are good for.

If what you want to do is just lookup by values then a hashtable is much faster, O(1) insert and lookup (amortized best case).

A BST will be O(log N) lookup where N is the number of nodes in the tree, inserts are also O(log N).

RB and AVL trees are important like another answer mentioned because of this property, if a plain BST is created with in-order values then the tree will be as high as the number of values inserted, this is bad for lookup performance.

The difference between RB and AVL trees are in the the rotations required to rebalance after an insert or delete, AVL trees are O(log N) for rebalances while RB trees are O(1). An example of benefit of this constant complexity is in a case where you might be keeping a persistent data source, if you need to track changes to roll-back you would have to track O(log N) possible changes with an AVL tree.

Why would you be willing to pay for the cost of a tree over a hash table? ORDER! Hash tables have no order, BSTs on the other hand are always naturally ordered by virtue of their structure. So if you find yourself throwing a bunch of data in an array or other container and then sorting it later, a BST may be a better solution.

The tree's order property gives you a number of ordered iteration capabilities, in-order, depth-first, breadth-first, pre-order, post-order. These iteration algorithms are useful in different circumstances if you want to look them up.

Red black trees are used internally in almost every ordered container of language libraries, C++ Set and Map, .NET SortedDictionary, Java TreeSet, etc...

So trees are very useful, and you may use them quite often without even knowing it. You most likely will never *need* to write one yourself, though I would highly recommend it as an interesting programming exercise.

Share Improve this answer Follow

edited Dec 21, 2015 at 11:26
Hemant Parihar
752 ● 1 ● 6 ● 19

answered Dec 2, 2009 at 6:46
joshperry
42.2k ● 16 ● 92 ● 104

@joshperry can you take a look my red black tree remove question? stackoverflow.com/questions/28705454/…
– committedandroider Feb 25, 2015 at 5:23

4

Red Black Trees and B-trees are used in all sorts of persistent storage; because the trees are balanced the performance of breadth and depth traversals are mitigated.

Nearly all modern database systems use trees for data storage.

Share Improve this answer Follow

answered Aug 21, 2008 at 19:09

mmattax
27.6k ● 42 ● 117 ● 151

@mattax can you take a look at my RedBlackTree remove method? stackoverflow.com/questions/28705454/…
– committedandroider Feb 25, 2015 at 8:03

2

BSTs make the world go round, as said by Micheal. If you're looking for a good tree to implement, take a look at AVL trees (Wikipedia). They have a balancing condition, so they are guaranteed to be O(logn). This kind of searching efficiency makes it logical to put into any kind of indexing process. The only thing that would be more efficient would be a hashing function, but those get ugly quick, fast, and in a hurry. Also, you run into the Birthday Paradox (also known as the pigeon-hole problem).

What textbook are you using? We used Data Structures and Analysis in Java by Mark Allen Weiss. I actually have it open in my lap as i'm typing this. It has a great section about Red-Black trees, and even includes the code necessary to implement all the trees it talks about.

Share Improve this answer Follow

answered Aug 21, 2008 at 19:12

helloandre
10.7k ● 9 ● 49 ● 64

can you take a look at my RedBlackTree remove method? It's based off the implementation by Mark Allen Weiss stackoverflow.com/questions/28705454/… – committedandroider Feb 25, 2015 at 8:03

2

Red-black trees stay balanced, so you don't have to traverse deep to get items out. The time saved makes RB trees O(log()n)) in the WORST case, whereas unlucky binary trees can get into a lop sided configuration and cause retrievals in O(n) a bad case. This does happen in practice or on random data. So if you need time critical code (database retrievals, network server etc.) you use RB trees to support ordered or unordered lists/sets .

But RBTrees are for noobs! If you are doing AI and you need to perform a search, you find you fork the state information alot. You can use a persistent red-black to fork new states in O(log(n)). A persistent red black tree keeps a copy of the tree before and after a morphological operation (insert/delete), but without copying the entire tree (normally and O(log(n)) operation). I have open sourced a persistent red-black tree for java

http://edinburghhacklab.com/2011/07/a-java-implementation-of-persistent-red-black-trees-open-sourced/

Share Improve this answer Follow

answered Jul 25, 2011 at 20:17
Tom Larkworthy
2,314 ● 1 ● 22 ● 33

do you use path-copying to implement persistence? is it fully or partially persistent? – mrk Jan 25, 2015 at 22:56

that implementation is path copying i.e. fully persistent – Tom Larkworthy Jan 27, 2015 at 22:02

@TomLarkworthy Can you take a look at my implementation of remove in Red Black Tree? stackoverflow.com/questions/28705454/… – committedandroider Feb 25, 2015 at 8:08

2

The best description of red-black trees I have seen is the one in Cormen, Leisersen and Rivest's 'Introduction to Algorithms'. I could even understand it enough to partially implement one (insertion only). There are also quite a few applets such as This One on various web pages that animate the process and allow you to watch and step through a graphical representation of the algorithm building a tree structure.

Share Improve this answer edited Apr 5, 2013 at 11:43

Follow

community wiki
2 revs, 2 users 86%
ConcernedOfTunbridgeWells

1 CLR does have a pretty good description. I have an early version of the CLR though (took comp sci classes about 20 years ago) and unfortunately, they leave coding the "Delete" function as an exercise to the reader. – Adisak Oct 30, 2009 at 3:36

1

Since you ask which tree people use, you need to know that a Red Black tree is fundamentally a 2-3-4 B-tree (i.e a B-tree of order 4). A B-tree is *not* equivalent to a binary tree(as asked in your question).

Here's an excellent resource describing the initial abstraction known as the symmetric binary B-tree that later evolved into the RBTree. You would need a good grasp on B-trees before it makes sense. To summarize: a 'red' link on a Red Black tree is a way to represent nodes that are part of a B-tree node (values within a key range), whereas 'black' links are nodes that are connected vertically in a B-tree.

So, here's what you get when you translate the rules of a Red Black tree in terms of a B-tree (I'm using the format *Red Black tree rule => B Tree equivalent*):

1) A node is either red or black. => A node in a b-tree can either be part of a node, or as a node in a new level.

2) The root is black. (This rule is sometimes omitted, since it doesn't affect analysis) => The root node can be thought of either as a part of an internal root node as a child of an imaginary parent node.

3) All leaves (NIL) are black. (All leaves are same color as the root.) => Since one way of representing a RB tree is by omitting the leaves, we can rule this out.

4)Both children of every red node are black. => The children of an internal node in a B-tree always lie on another level.

5)Every simple path from a given node to any of its descendant leaves contains the same number of black nodes. => A B-tree is kept balanced as it requires that all leaf nodes are at the same depth (Hence the height of a B-tree node is represented by the number of black links from the root to the leaf of a Red Black tree)

Also, there's a simpler 'non-standard' implementation by Robert Sedgewick here: (He's the author of the book *Algorithms* along with Wayne)

Share Improve this answer Follow

edited Jan 17, 2013 at 14:03

answered Jan 17, 2013 at 13:19

arviman

5,255 ● 45 ● 48

can you take a look at my red black tree remove question? stackoverflow.com/questions/28705454/…
– committedandroider Feb 25, 2015 at 5:24

1

Lots and lots of heat here, but not much light, so lets see if we can provide some.

First, a RB tree is an associative data structure, unlike, say an array, which cannot take a key and return an associated value, well, unless that's an integer "key" in a 0% sparse index of contiguous integers. An array cannot grow in size either (yes, I know about realloc() too, but under the covers that requires a new array and then a memcpy()), so if you have either of these requirements, an array won't do. An array's memory efficiency is perfect. Zero waste, but not very smart, or flexible - realloc() not withstanding.

Second, in contrast to a bsearch() on an array of elements, which IS an associative data structure, a RB tree can grow (AND shrink) itself in size dynamically. The bsearch() works fine for indexing a data structure of a known size, which will remain that size. So if you don't know the size of your data in advance, or new elements need to be added, or deleted, a bsearch() is out. Bsearch() and qsort() are both well supported in classic C, and have good memory efficiency, but are not dynamic enough for many applications. They are my personal favorite though because they're quick, easy, and if you're not dealing with real-time apps, quite often are flexible enough. In addition, in C/C++ you can sort an array of pointers to data records, pointing to the struc{} member, for example, you wish to compare, and then rearranging the pointer in the pointer array such that reading the pointers in order at the end of the pointer sort yields your data in sorted order. Using this with memory-mapped data files is extremely memory efficient, fast, and fairly easy. All you need to do is add a few "*"s to your compare function/s.

Third, in contrast to a hashtable, which also must be a fixed size, and cannot be grown once filled, a RB tree will automagically grow itself and balance itself to maintain its O(log(n)) performance guarantee. Especially if the RB tree's key is an int, it can be faster than a hash, because even though a hashtable's complexity is O(1), that 1 can be a very expensive hash calculation. A tree's multiple 1-clock integer compares often outperform 100-clock+ hash calculations, to say nothing of rehashing, and malloc()ing space for hash collisions and rehashes. Finally, if you want ISAM access, as well as key access to your data, a hash is ruled out, as there is no ordering of the data inherent in the hashtable, in contrast to the natural ordering of data in any tree implementation. The classic use for a hash table is to provide keyed access to a table of reserved words for a compiler. It's memory efficiency is excellent.

Fourth, and very low on any list, is the linked, or doubly-linked list, which, in contrast to an array, naturally supports element insertions and deletions, and as that implies, resizing. It's the slowest of all the data structures, as each element only knows how to get to the next element, so you have to search, on average, (element_knt/2) links to find your datum. It is mostly used where insertions and deletions somewhere in the middle of the list are common, and especially, where the list is circular and feeds an expensive process which makes the time to read the links relatively small. My general RX is to use an arbitrarily large array instead of a linked list if your only requirement is that it be able to increase in size. If you run out of size with an array, you can realloc() a larger array. The STL does this for you "under the covers" when you use a vector. Crude, but potentially 1,000s of times faster if you don't need insertions, deletions or keyed lookups. It's memory efficiency is poor, especially for doubly-linked lists. In fact, a doubly-linked list, requiring two pointers, is exactly as memory inefficient as a red-black tree while having NONE of its appealing fast, ordered retrieval characteristics.

Fifth, trees support many additional operations on their sorted data than any other data structure. For example, many database queries make use of the fact that a range of leaf values can be easily specified by specifying their common parent, and then focusing subsequent processing on the part of the tree that parent "owns". The potential for multi-threading offered by this approach should be obvious, as only a small region of the tree needs to be locked - namely, only the nodes the parent owns, and the parent itself.

In short, trees are the Cadillac of data structures. You pay a high price in terms of memory used, but you get a completely self-maintaining data structure. This is why, as pointed out in other replies here, transaction databases use trees almost exclusively.

Share Improve this answer Follow

edited Feb 19, 2015 at 23:54
user1899861

answered Sep 3, 2013 at 21:19
user2548100
4,661 ● 1 ● 19 ● 18

can you take a look at my RedBlackTree remove method? stackoverflow.com/questions/28705454/…
– committedandroider Feb 25, 2015 at 8:05

0

If you would like to see how a Red-Black tree is supposed to look graphically, I have coded an implementation of a Red-Black tree that you can download here

Share Improve this answer Follow

answered Dec 8, 2008 at 16:06
Brock Woolf
47.3k ● 51 ● 124 ● 145

0

IME, almost no one understands the RB tree algorithm. People can repeat the rules back to you, but they don't understand *why* those rules and where they come from. I am no exception :-)

For this reason, I prefer the AVL algorithm, because it's easy to *comprehend*. Once you understand it, you can then code it up from scratch, because it make sense to you.

Share Improve this answer Follow

answered Mar 24, 2009 at 21:04

user82238

You should probably read up Introduction to Algorithms for the explanation. They *why* is pretty well explained and will make sense when you analyze the various cases that arise on inserting a new node. – arviman Jan 17, 2013 at 12:44

0

Trees can be fast. If you have a million nodes in a balanced binary tree, it takes twenty comparisons on average to find any one item. If you have a million nodes in a linked list, it takes five hundred thousands comparisons on average to find the same item.

If the tree is unbalanced, though, it can be just as slow as a list, *and* also take more memory to store. Imagine a tree where most nodes have a right child, but no left child; it *is* a list, but you still have to hold memory space to put in the left node if one shows up.

Anyways, the AVL tree was the first balanced binary tree algorithm, and the Wikipedia article on it is pretty clear. The Wikipedia article on red-black trees is clear as mud, honestly.

Beyond binary trees, B-Trees are trees where each node can have many values. B-Tree is *not* a binary tree, just happens to be the name of it. They're really useful for utilizing memory efficiently; each node of the tree can be sized to fit in one block of memory, so that you're not (slowly) going and finding tons of different things in memory that was paged to disk. Here's a phenomenal example of the B-Tree.

Share Improve this answer Follow

answered Jul 9, 2010 at 19:57

Dean J
40.2k ● 17 ● 70 ● 94