Red-Black Trees

CSE 30331/34331

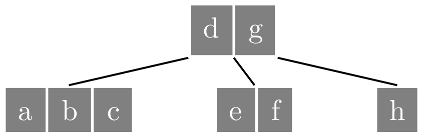
Fall 2015 (version 1.1)

We've studied BSTs and B-trees, which each have their own advantages. The advantage of B-trees is that they are perfectly balanced, guaranteeing $O(\log n)$ time operations. The advantage of BSTs is that the nodes have a fixed, small size, which makes them easy to work with. Can we get the best of both worlds? Yes – we can do this with *red-black trees*. Because of their nice properties, they are usually the data structure of choice for implementing `std::map`.

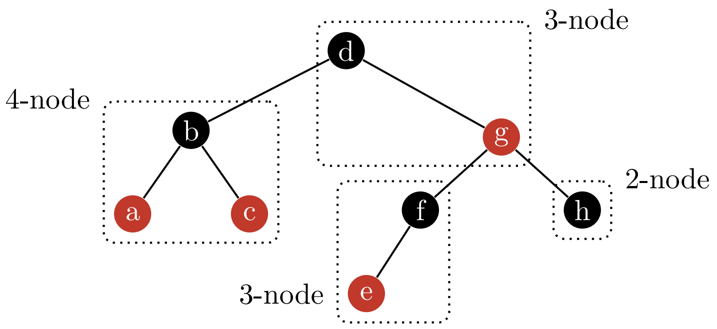
1 Definition

The trick is to represent each variable-sized node of a B-tree using a little BST. In effect, the result is a tree of trees, but we store it as one big binary tree with a flag to mark the root of each little tree. Traditionally, we say that the root of each little tree is colored black, and other nodes are colored red, so the tree is called a red-black tree. To avoid confusion, we will write "node" to refer to the nodes of the red-black tree, and "B-node" to refer to the nodes of the B-tree. (Similarly, we write "B-sister," "B-parent," "B-root.") A 2-node is a B-node with 2 children, a 3-node is a B-node with 3 children, and a 4-node is a B-node with 4 children.

In all of the following diagrams, we draw the B-nodes with dotted lines, but they are *not* part of the data structure; they are just there for illustration.



is represented as



A red-black tree simulates a B-tree of order 4 (also known as a 2-3-4 tree). As you can see above, a 3-node can be either left or right branching. And a 4-node must be balanced, as shown above. So we have the following constraints:

- Every B-node is a 2-node, a 3-node, or a balanced 4-node.
- Every path has the same number of B-nodes.

These constraints are often formulated directly in terms of node colors instead of in terms of B-nodes:

- The root node is black.
- No path has two red nodes in a row.
- Every path has the same number of black nodes.

Question 1. Explain why a red-black tree, when you ignore the colors, is a BST.

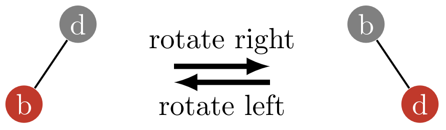
Question 2. How do you search for an element in a red-black tree?

Question 3. What is the worst-case running time of searching in a red-black tree?

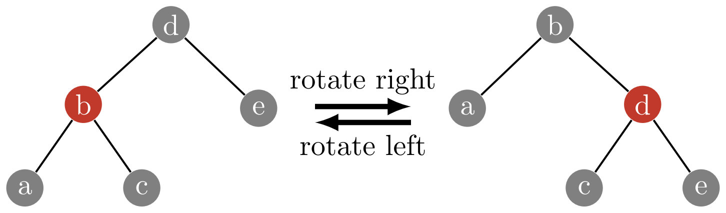
2 Rotation

Below we will repeatedly make use of an operation on BSTs called *rotation*. Rotation is also used in AVL trees and other kinds of balanced binary trees.

Basically, rotation inverts a parent-child relationship. If node b is the left child of node d, then a *right rotation* changes node d into the right child of node b. A *left rotation* is exactly the inverse operation. (In these figures, a gray node stands for either a red or black node.)



What happens if b and d have (other) children? The left child of b and the right child of d are unaffected, but the middle child (c) has to be reconnected:



Note that the two nodes that swap positions also swap colors. The lower node must be red in order to preserve the correctness of the red-black tree.

3 Insertion

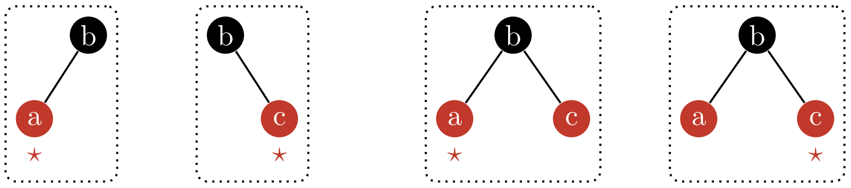
In a B-tree, the first step of insertion is to find the B-node where the new element belongs and then insert the element into that B-node (possibly making it overfull). In a red-black tree, this is identical to insertion into a BST, for the same reason that searching in a red-black tree is identical to searching in a BST.

If the tree was empty to begin with, the new node must be black, and we're done. Otherwise, it must be red.

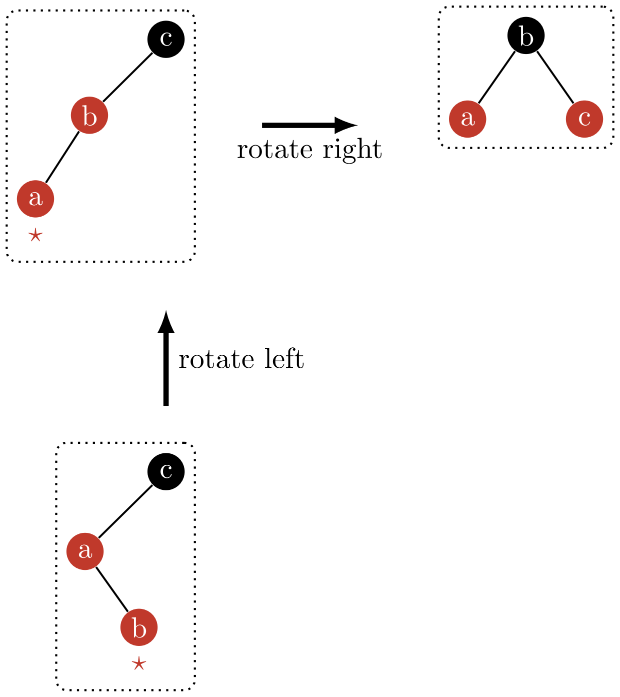
Question 4. Why does the newly inserted node have to be red?

We mark the newly inserted node with a star. In an actual implementation, this would be some kind of pointer named `current` or something like that.

(1) If the resulting B-node is a 3-node or a balanced 4-node (in other words, if the new node's parent is black), remove the red star and stop.



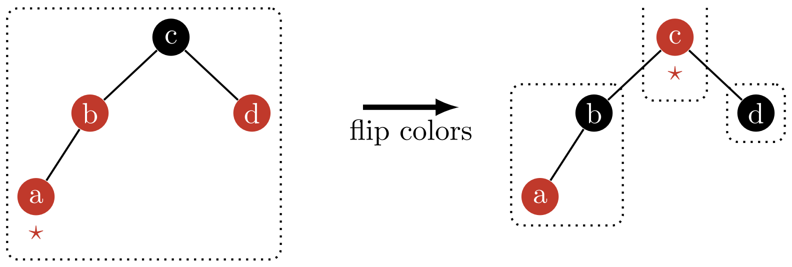
If the resulting B-node is an unbalanced 4-node, we need to balance it using one or two rotations:



Then remove the red star and stop.

Question 5. Work out the mirror-image cases.

But if the resulting B-node is a 5-node, it is overfull. Recall that in a B-tree, an overfull B-node has to be split, and one of the elements is promoted to the B-parent. In a red-black tree, remarkably, the same effect can be achieved just by flipping the colors of the top three nodes:



The 5-node is split into a 2-node and a 3-node, and the old root is promoted to the B-parent. There are three other configurations that you can find a 5-node in, but they are all handled exactly the same way.

As a result of the color flip, the B-parent got a new node. So move the star to that node. Now we have to worry about this newly-starred node. If it's the root node of the tree, that means the height of the B-tree grew by one and the B-parent is the new root B-node. So we just color the root node black and stop. Otherwise, the B-parent may now be unbalanced or overfull, so go back to (1). Thus, the red star climbs up the tree until it can be removed.

Question 6. What is the worst-case number of rotations and color flips that have to be performed for a single insertion?

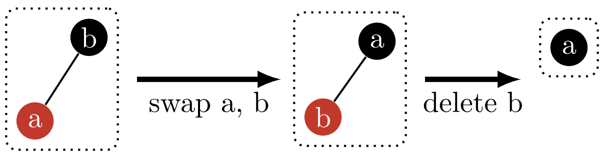
4 Deletion

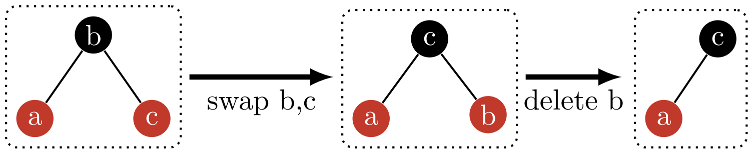
B-tree deletion is more complicated than insertion (and is often unfairly left as an exercise to the reader). The first step is easy, though: it's exactly like BST deletion.

Recall how B-tree deletion works: Find the B-node that contains the element to be deleted. If it's in a leaf B-node, delete the element from it. If it's not in a leaf B-node, swap the element with the next biggest or next smallest element (which must be in a leaf B-node) and delete the element from it.

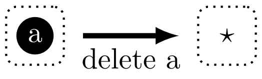
Similarly, in a red-black tree, find the node to be deleted. If it's not in a leaf B-node, swap it with the next biggest or next smallest element. Then, since a B-node is just a little BST, find the node within the B-node, and if it is not a leaf or unary node, swap it with the next biggest or next smallest element (which must be a leaf or unary node). Finally, delete it.

When we swap elements, the colors stay in the same positions. Thus, deleting a black node in a 3-node or 4-node looks like this:



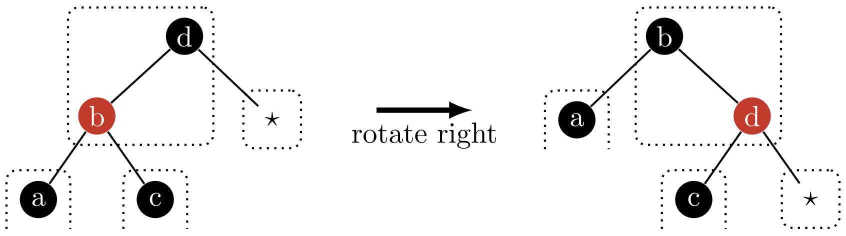


However, if the node to be deleted roots a 2-node, we have a problem. In terms of the B-tree, deleting it would result in a 1-node, which has zero elements and one child, which is underfull. In term of the red-black tree, deleting a black node results in a path that has one too few black nodes. To indicate this deficit, we draw a black star which means "need another black node here":



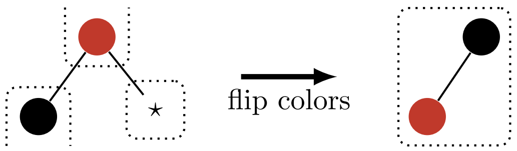
There are only two ways to get rid of the black star. First, if the black star is at the very top of the tree. In that case, there isn't really a deficit of black nodes (because all paths go through the root), and we can safely remove the black star. Second, if we do a color flip that would turn a node in this position red, then there is no more deficit of black nodes and we can remove the black star.

(2) Recall that, in a B-tree, if we have an underfull B-node, we merge it with the B-sister and then resplit if necessary. So we have to identify the B-sister. If the sister node is black, then it's the root of the B-sister as well. But if the sister node is red, we have a situation like the one shown below on the left. In order to reduce the number of cases we have to worry about, use a rotation to make the sister node black:



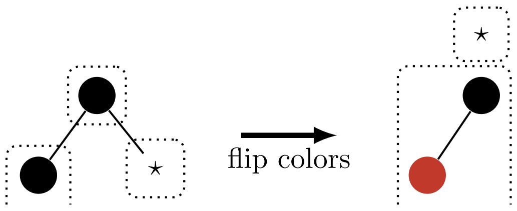
Question 7. How do we know that if the sister is red, the tree must look like the above left (or its mirror image)?

Although it's not necessarily the most efficient way, we can decompose what's next into three steps: merge, balance, split. First, merge the underfull B-node and its B-sister by flipping colors. This is the reverse of the color flip we used to split a B-node:



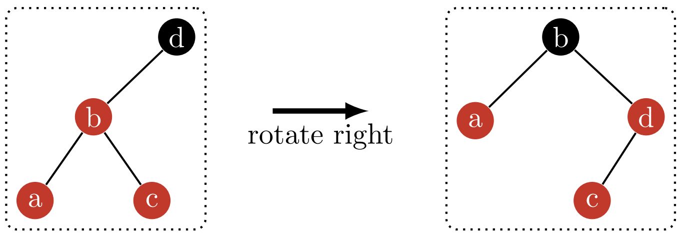
Note that the underfull B-node has gone away in the merge. In terms of the red-black tree, the color flip would change a node at the black star (if there were one) to red, so the black star goes away.

But if the parent node was already black, we have another problem. In terms of the B-tree, the merge demoted an element from the B-parent, which in turn became an underfull 1-node. In terms of the red-black tree, we're again short one black node. So now we mark the parent node with a black star:



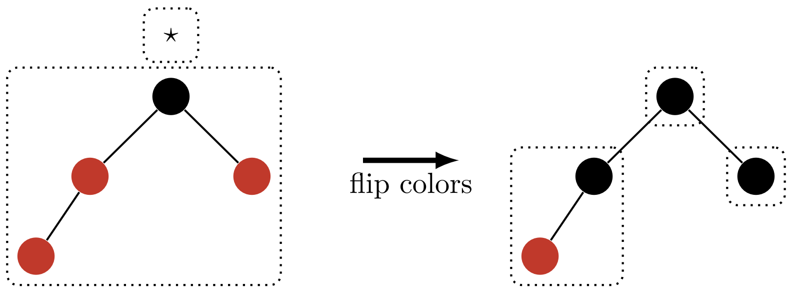
Let's ignore this new black star for now.

Second, rebalance the merged B-node if necessary. If it's a 3-node, it doesn't need to be rebalanced. If it's a 4-node, rebalancing happens just as in insertion. If it's a 5-node, we rebalance it with a single rotation:



Question 8. Why is the above left (and its mirror image) the only possible shape of a 5-node?

Third, split the B-node again if possible. If it's a 5-node, it splits into a 2-node and a 3-node. If it's a 4-node, it's not overfull and doesn't have to be split, but it's a good idea to split it anyway. Just as in insertion, we split the B-node with a color flip. If the parent node was marked with a black star, the node stays black, but the black star goes away:



However, if the B-node is a 3-node, it cannot be split. So the parent node still has a black star. If the parent node is the root, we can just remove the black star and stop. In terms of the B-tree, this means that the B-tree has shrunk by one level. Otherwise, go back to (2). Thus, the black star climbs up the tree until it can be safely removed.

Question 9. Why is it better to split a 4-node instead of leaving it alone?

Question 10. What is the worst-case number of rotations and color flips that have to be performed for a single deletion?