Evan Delanty
12/9/23

A red-black tree is a non-linear data structure created by Leonidas J. Guibas and Robert Sedgewick that stems from the original idea of a binary-search tree (BST). The red-black tree promotes an effective evolution of the original BST called self-balancing, which was popularized by Georgy Adelson-Velsky and Evgenii Landis in 1962 with their invention of the AVL tree. The process of self balancing for an AVL tree involves values called balance factors, which are calculated by taking the longest path of each branch of a node and subtracting. If these balance factors reach -2 or 2, this calls for a balancing technique called rotations, which ensures that the balance factor of all the nodes falls below the threshold.

Red-black trees optimize this technique by ensuring four main rules for upholding balance. The first rule states that every node has a new property called color, and that the only two colors allowed are red and black. The second rule ensures that every leaf node's NULL, or sentinel, is black. The third rule is that if a node is red, then both its children must be black. The fourth and last rule states that every simple path from a node to a descendant leaf contains the same number of black nodes. If all four of these rules are enforced in the implementation of this data structure, balance will remain and will reward $O(log\ n)$ time complexity for all three main functions of search, insertion, and deletion.

Before you can insert and delete nodes of any tree data structure, you need to know how to rotate nodes in order to maintain balance. In a red-black tree, there are two types of rotations, right and left shown in figure 1a shown below.

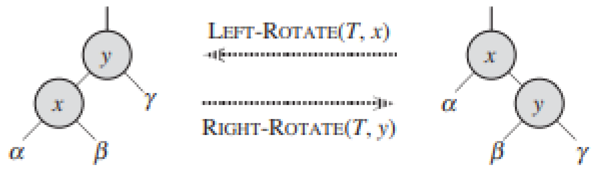

Figure 1a (Cormen 313)

The left rotate operation alters the arrangement of the two nodes on the right to match the configuration on the left, achieved through a fixed number of pointer adjustments. Conversely, the right rotate operation reverses this process, transforming the configuration on the left into that on the right. In this context, the symbols alpha, beta, and gamma imply arbitrary subtrees.

Logarithmic time complexity is much faster than the original BST, which can easily degenerate into the worst case time complexity of $O(n)$ in all three main functions. When inserting a node into a red-black tree, we are always inserting the new node as red. Similar to heaps, this can destroy the four properties of ensuring balance, so you fix this in the implementation of the insertion function. There are two ways that we can fix the balance, recoloring and rotations. We will always try to recolor, but if we don't maintain proper balance or abstain from the properties, we will have to use rotations. There are three cases for how a violation can be fixed after insertion. Case one involves the violation of property two shown below in figure 2a below.

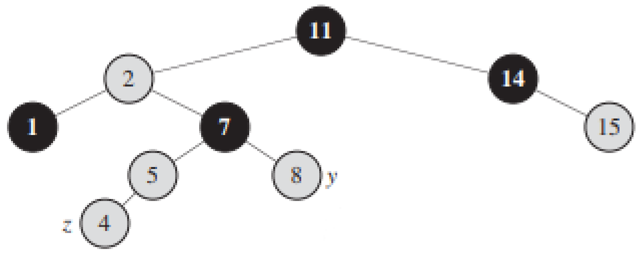

Figure 2a (Cormen p317)

We can try and fix this by recoloring 5, 7, and 8 shown in figure 2b below, but there remains a violation of property two due to 2 and 7. Since both the node z (7) and its parent are red, but z's uncle y is black, and since z is the right child of its parent, case 2 will apply in order to maintain the red-black tree properties.

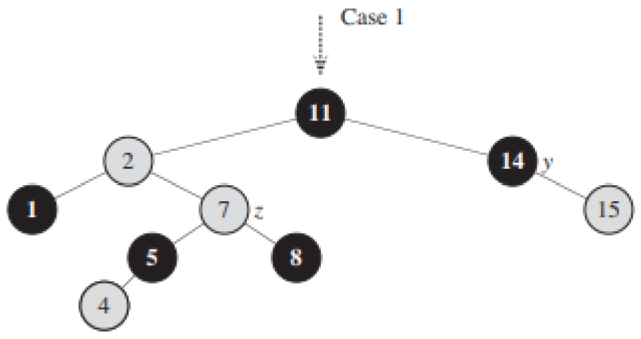

Figure 2b (Cormen p317)

We can perform a left rotation which results can be seen in figure 2c below.

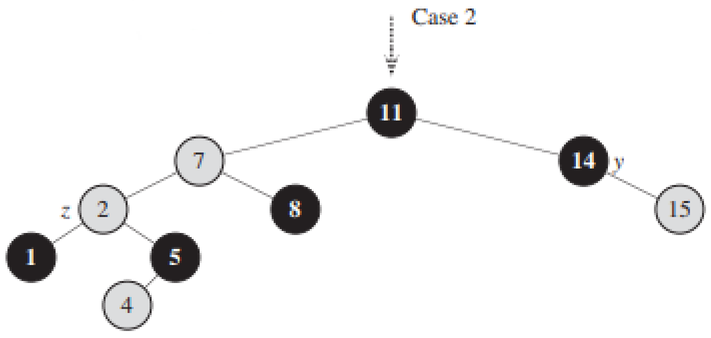

Figure 2c (Cormen p317)

Now, z is positioned as the left child of its parent, triggering the application of case 3. Recoloring and performing a right rotation results in the tree depicted in figure 2d shown below, establishing a valid red-black tree configuration.

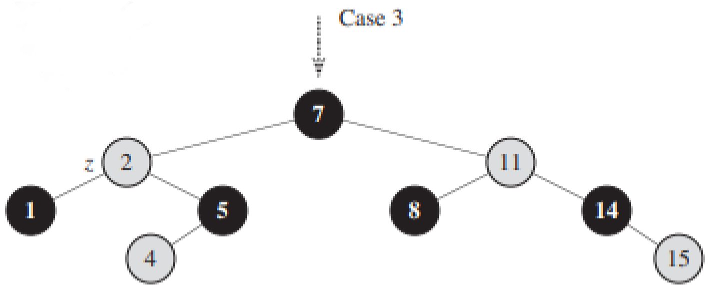

Figure 2d (Cormen p317)

These were all the cases for a proper insertion and insertion fix, the next important main function is deletion, which is very similar to a traditional BST, but with four cases during the fixup function to make sure the red-black properties are maintained. It’s important to note that in all of figure 3 is the tree structure after a deletion has occurred and that the dark nodes are black, the gray shaded nodes are red, and the lightly shaded nodes are either color, so color doesn’t matter outside the black and gray nodes. Case one of the fixup function is when the node x’s sibling w is red shown clearly below in figure 3a. Case two is when x’s sibling w is black, and both of w’s children are black shown in 3b. Case three is when x’s sibling w is black, w’s left child is red, and w’s right child is black shown in figure 3c. Finally, case four is when x’s sibling w is black, and w’s right child is red shown in figure 3d.

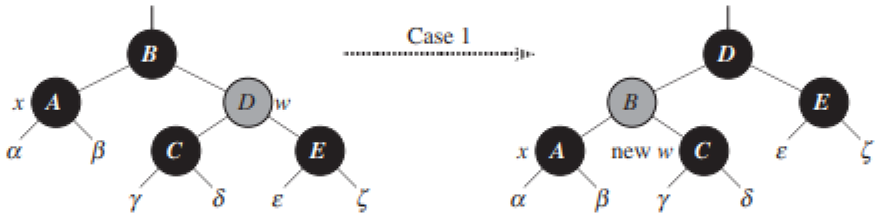

Figure 3a (Cormen p329)

Because in case one w must have black children, we can interchange the colors of w and x’s parent. Following this color switch, a left-rotation is performed on x’s parent without violating any of the red-black properties. Consequently, the new sibling of x, previously one of w’s children is now black. This transformation effectively converts case one into either case 2, 3, or 4.

In case two shown below in figure 3b, both of w’s children are black. Since w is also black, we take one black off both x and w, leaving x with only one black and leaving w red. To compensate for removing one black from x and w, we would like to add an extra black to x's parent, which was originally either red or black. We do so by repeating the fixup function with x's parent as the new node x. Observe that if we enter case two through case one, the new node x is red and black, since the original x's parent was red. So, the value c of the color attribute of the new node x is red, and the main loop terminates when it tests the loop condition. We then color the new node x black.

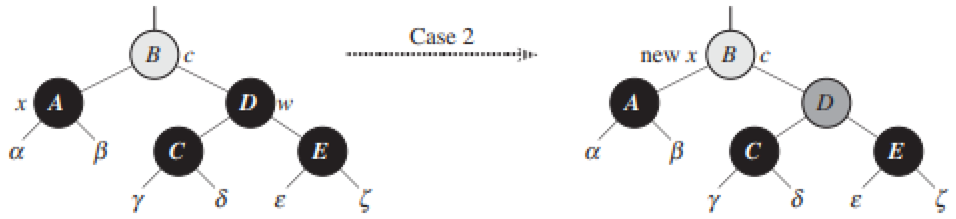

Figure 3b (Cormen p329)

In case three shown below in figure 3c, w is a black node with a red left child and a black right child. By exchanging the colors of w and its left child and subsequently executing a right rotation on w, we maintain the integrity of the red-black properties. The resulting configuration turns w into a black node with a red right child. Case three has now been converted into case four shown in figure 3d.

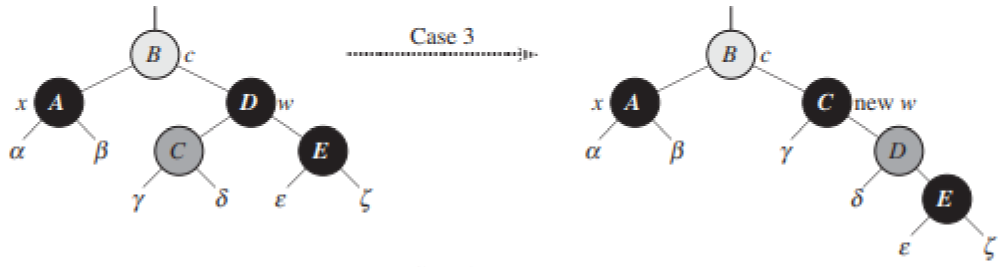

Figure 3c (Cormen p329)

In case four, x’s sibling w is black, and w’s right child is red. Through adjustments in color and executing a left rotation on x's parent, we can eliminate the extra “blackness” on x (blackness in this context implies that we need to make sure we don’t violate property four), reverting it to a singly black state, all while preserving the red-black properties. Setting x as the new root ensures the termination of the fixup function upon satisfying its main loop condition.

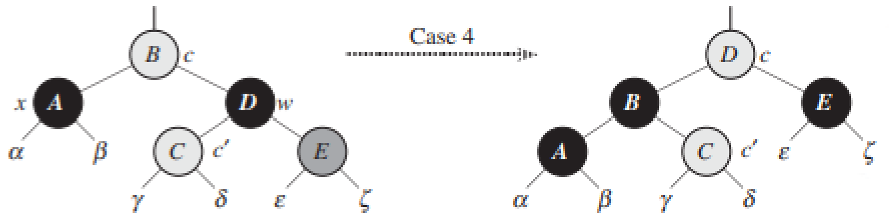

Figure 3d (Cormen p329)

These were all the cases for maintaining the red-black properties after the deletion of any node, which takes logarithmic time to complete.

The final main function of a red-black tree is search. Because the red-black tree has very similar properties to a traditional BST, the search function is nearly identical. Beginning the search from the root of the tree, compare the element you're searching for with the value at the root. If the element is less than the root, recursively navigate to the left subtree, else recurse to the right subtree. If the search element is located within the tree, return true, otherwise, return false.

The red-black tree is great for insertion, deletion, and search compared to traditional linear and even other non-linear data structures. The red-black tree is simple to understand visually, but when implemented is quite challenging without external sources which is quite different compared to other BST alternatives. Specifically compared to the AVL tree, self-balancing usually involves less steps especially when using larger sets of data. So when inserting and deleting nodes constantly, it will always be better to use the red-black tree, but if you are simply searching and not adding or deleting nodes, the AVL tree is more balanced and therefore giving you closer real time results towards the sought after $O(log\ n)$ time complexity.

Sources:

Carrano, F. M., & Henry, T. (2017). Chapter 19. In Data Abstraction & problem solving with C++: Walls and mirrors. essay, Pearson.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Red-black trees. In Introduction to algorithms, Third edition (pp. 308–338). essay, MIT Press.

GeeksforGeeks. (2023, March 15). *Introduction to red-black tree*. GeeksforGeeks. https://www.geeksforgeeks.org/introduction-to-red-black-tree/?ref=lbp

Morris, J. (n.d.). Red-black Trees. Data Structures and algorithms: Red-black trees. https://www.eecs.umich.edu/courses/eecs380/ALG/red_black.html