COS 226. **Bob's Notes 2: Deletion in Binary Search Trees** October 27, 2019
Fall 2019

Deletion in most kinds of binary search trees is more complicated than insertion. Consider deleting a key from an ordinary (unbalanced) binary search tree. Let *x* be the tree node holding key *k*. If *x* is a *leaf* (it has two null child links), we merely delete *x*. If *x* is *unary* (it has exactly one null child link), we replace *x* by its only child. (Exercise: draw pictures illustrating these two cases.). But if *x* has two children (*x* is *binary*), we are in a bit of a pickle: we cannot replace *x* by both of its children. The classical way to handle this case is *Hibbard deletion* [T. N. Hibbard, "Some combinatorial properties of certain trees with applications to searching and sorting," *J. ACM*, 9(1):13–28, 1962]: find the node *y* containing the smallest key, say *k'*, larger than *k,* and *swap k* and *k'* and their associated values (so *k'* is now in *x* and *k* is in *y*). This messes up the symmetric order of keys, since *k'* now precedes *k* rather than following it, but this is only temporary: we now delete *y* if it is a leaf or replace *y* by its only child if it is unary, thereby deleting *k* and re-establishing symmetric order. Node *y* is the first node in symmetric order in the right subtree of *x*, found by starting at the right child of *x* and proceeding through left children until reaching a node with a null left child link. Since *y* has no left child, it is unary or a leaf. (Exercise: draw a picture illustrating Hibbard deletion.)

To delete a key from a *balanced* search tree, such as a left-leaning red-black tree, we first delete the key by the method in the previous paragraph. This may leave the tree unbalanced, so we may need to do some rotations and recolorings to restore balance. In describing the needed steps I shall treat null links as black. Recall the constraints that define left-leaning red-black trees: (i) each red link is a left link; (ii) each node has at most one incident red link; and (iii) every path from the root to a null link, including the null link, contains the same number of black links.

Recall how insertion in a left-leaning red-black tree works: We insert the new node *x* at the bottom of the tree and color the new link leading to *x* red. Since both child links of *x* are null and hence black, this preserves (iii) but may violate (i) and/or (ii). To rebalance the tree, we repeatedly apply one of the following transformations until none applies:

T1: Some node has two red child links: color these links black and the link from the parent red.

T2: Some node has a red right child link but its left child link is black: do a left rotation on the red child link, making it a left link.

T3: Some node has a red left child link and the link from its parent is left and red: do a right rotation on the link from the parent, making it right. The next transformation is T1.

Insertion rebalancing maintains the invariant that at most one node is incident to two red links and none is incident to three, there is at most one red right link, and if one of the child links of *x* is red and the link from its parent is red, then the link from its parent is left. Exercise: verify this invariant, prove the correctness of insertion rebalancing, and prove that it takes O(log*n*) time in a tree of $n > 1$ nodes.

Now we consider deletion. We shall describe a method that first does a Hibbard deletion to delete the specified key. If the removed node *x* were the root, the new tree is empty and hence balanced. If *x* were connected to its parent by a red link, *x* was a leaf. Removing *x* replaces the red link connecting *x* to its parent by a null black link, leaving the tree balanced. Similarly, if the were unary unary, it had a left child, to which it was connected by a red link, and it was connected to its parent by a black link. (Why?) Replacing it by its child and making the link from its parent black leaves the tree balanced. (Why?)

What remains is the hard case: deletion of a leaf whose link from its parent is black. We remove the node but make the new null link replacing the link from its parent *double black*. A double black link counts two rather than one when counting black links along a tree path. Making the new null link double black preserves balance, but leaves us with a temporary illegal state, a double black link. If *x* is the node having the double black link as one of its child links, its other link is non-null and black. (Why?) We use transformations T1, T2, and T3, and the following two additional transformations, to move the balance violation up toward the root, until it is finally eliminated.

T4: Some node has a red left child link and a double black right child link: do a right rotation on the red link, making it a right child link. The double black link remains a double black right link.

T5: Some node *x* has a black child link and a double black child link. Make the black link red, make the double black link black, and if *x* has a link from its parent, make this link black if it is red, double black if it is black.

Unlike insertion, in which there is only one transformation that applies at any given time, in deletion there can be more than one applicable transformation, and to make sure the rebalancing process stops we need to restrict the order of application. Specifically, we need to prevent an alternation between T4 and T2, which could go on forever. It suffices to require that T5 follow T4. This converts the new red right link created by T4 into a black link, so T2 no longer applies to it (although T5 can also create a new red right link, to which T4 does apply).

As long as T4 is always immediately followed by T5, the tree will become balanced after a number of transformations at most a constant times the depth of the tree. This needs a proof, which we leave as an exercise.

There is still some flexibility in the application of the transformations. One extreme is to apply T5 and T4 to push the double black link up the tree until there is no double black. This process can leave behind some number of nodes incident to two red links, which we then go back and eliminate. It is better to eliminate the red violations on the way up the tree. Let us work out how to do this.

Each transform preserves (iii) and either leaves the number of double black links unchanged or reduces it by one. We conclude that there is always at most one double black link, and once there are no double black links the only violations are of (i) and (ii), which we handle using T1, T2, and T3 as in insertion.

Initially there is only one violation, a double black link. Let *z* be the node having a double black link to a child *x*, and let *y* be the other child of *z*. There are three possibilities: *y* is a left child and its link from *z* is red; *y* is a left child and its link from *z* is black; *y* is a right child and its link from *z* is black. By (i) it cannot be the case that *y* is a right child and its link from *z* is red. We treat the second and third possibilities as one case.

Case 1: Node *y* is a left child whose link from *z* is red. We do T4 followed by T5. This makes *z* the right child of *y*; *x* remains the right child of *z*. The right child links of both *y* and *z* are black; the left child link of *z* is red. There is no longer a double black link, but there may be two red left links in a row. (If *w* is the left child of *z*, its left child link may be red.) If there are two red left links in a row, we do T3 followed by T1. This eliminates all violations: the new red link is the right child link of *y*; the link from the parent of *y* must be black, since it is the same color as the link from the parent of *z* before the application of T4, and this link must be black, since before the application of T4 the left child link of *z* is red, and by (ii) the link from its parent must be black.

Case 2: The link from *z* to *y* is black. (Node *z* can be either a left or a right child. We subdivide this case below.). We do T5. This makes the link from *z* to *y* red, the link from *z* to *x* black, and the link to *z* from its parent either black or double black. If *y* has a red left child link, this violates (ii), and if *y* is a right child it also violates (i). We consider three subcases:

Case 2a: Both child links of *y* are black. If *y* is a right child, we do T2. Now (i) and (ii) hold, and the only possible violation is that *z*, or *y* if the rotation was done, is double black. If there is a double black node, apply Case 1 or 2 as appropriate.

Case 2b: Node *y* is a left child and its link to its left child is red. We do T3 and then T1, except that when applying T1, if the link from the parent is double black, make it black. This eliminates the double black if it exists. It also creates a new red link if and only if the link to *z* from its parent was red before the application of T5. This means that the application of T1 eliminates all violations.

Case 2c: Node *y* is a right child and its link to its left child is red. In this case we have to fix the violations of (i) and (ii), but this case does not occur in insertion, because the right red link is on top of the left red link, not the other way around. Using the specified transformations, we can fix the red violations by doing T2 twice, followed by T3 and then T1. The first T1 produces a node with a red right child link whose link from its parent is red and left. The second T1 produces a node with a red left child whose link from its parent is red and left, so T3 applies. This sequence of transforms does three rotations. A more efficient way to obtain the same result is to first do a right rotation on the red left child link of *y*, then do a left rotation on the red right child link of *z*, and finally do T1. This takes only two rotations. After T1, there are no violations, by the same argument as in Case 2b.

The only case that does not eliminate all violations is 2a, which may create a new double black link farther up the tree. The deletion rebalancing algorithm repeats Case 2a until there is no double black link or until some other case applies and eliminates the double black. Violations of (i) and (ii) (the red rules) only occur temporarily in the middle of a case.

Exercise: Prove that deletion rebalancing is correct and that it takes O(log*n*) time.

Deletion rebalancing is more complicated than insertion rebalancing. This is true in all kinds of balanced trees, not just left-leaning red-black trees. The deletion algorithm presented above does the rebalancing purely bottom-up, after deletion of the appropriate key. The book outlines a different deletion algorithm that does some rebalancing top-down during the search for the key to be deleted, followed by additional rebalancing bottom-up after deletion of the key.

There are other versions of red-black trees that have been proposed: one can allow both child links of a node to be red and/or allow a node with one black and one red child link to have the right child link be red. Changing the balance conditions requires corresponding changes in the insertion and deletion rebalancing algorithms. There are even more general ways to define balance. My favorite kinds of balanced trees are not red-black trees but variants of AVL trees, specifically weak AVL trees and relaxed AVL trees. Take COS 423 to find out about these.