13.4 Deletion in red-black trees

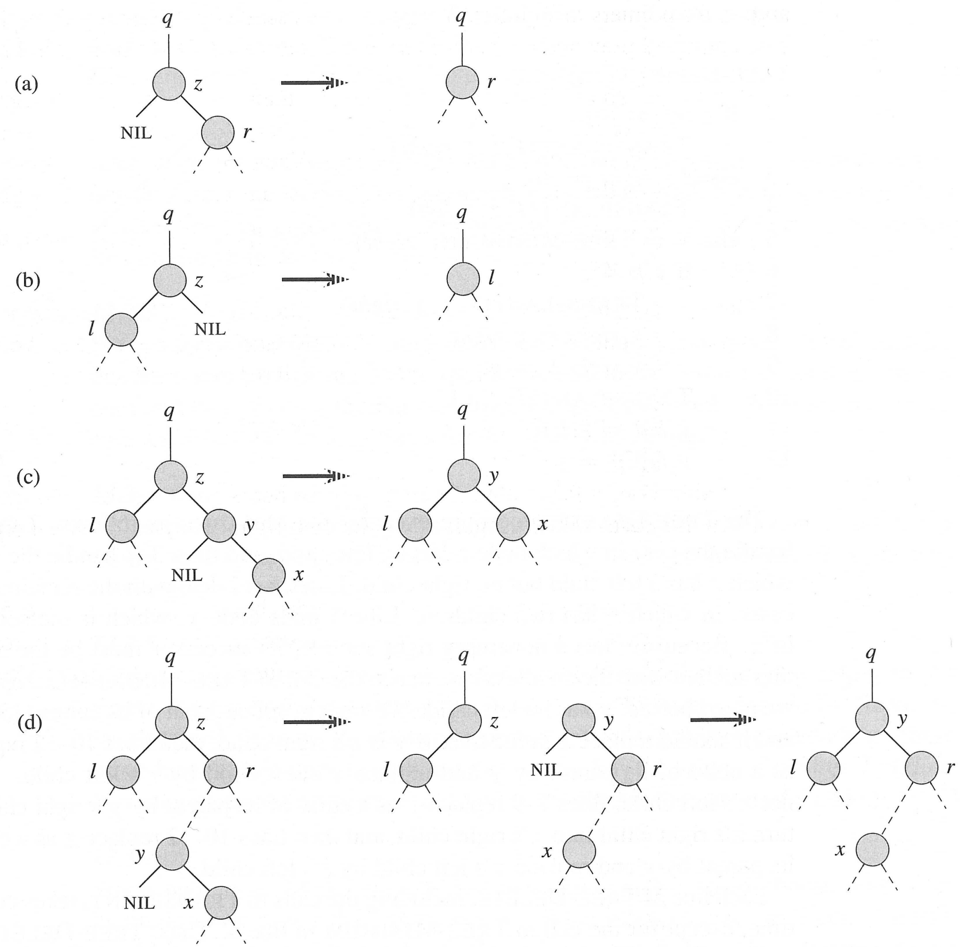
Deletion in a red-black tree is similar to insertion.

- Apply the deletion algorithm for binary search trees.
- Apply node color changes and left/right rotations to fix the violations of RBT tree properties.

Suppose we delete a node z, and there are three cases of z in a binary search tree: (i) z is a leaf; (ii) z has only one child; and (iii) z has two children.

When z has two children, let y be the successor of z in the RBT, the actual deletion proceeds as follows: Copy the content of y (not its color) to node z, node y then is "spliced out" (check slides 17-18 in Lecture 12 about the deletion operation in a binary search tree), where y only has only the right child in the binary search tree.

13.1 Deletion in Binary Search Trees



Deletion in binary search trees (Cormen et al., p297)

13.4 Deletion in red-black trees

We now adjust the colors of some nodes in the tree.

- If y was RED, the new tree is already a red-black tree, done.
- If y was BLACK, the new tree due to the removal of y has several types of red-black property violations:
  - Case one: If y was the tree root, the new root might be RED
  - Case two: If both the parent and the child of y are RED, they now have the relationship between the parent and the child
  - Case three: Any path passing through y now has fewer BLACK nodes.

13.4 Deletion in red-black trees

Suppose the color of y was BLACK and x is the sole child of y in the tree (Why?.

- If y was the root and a red child x of y now becomes the new root, then Property 2 of the RBT is violated.
- If both x and $p[y]$ are RED before the deletion of y, then Property 4 of the RBT is violated.
- The removal of y causes any path containing y has a black node less. This causes Property 5 of the RBT to be violated.

We correct the above errors by saying that node x has an "extra" BLACK color.

13.4 Deletion in red-black trees

To restore Property 5 of the RBT, we consider four cases:

- Case 1: x's sibling w is red
- Case 2: x's sibling w is black, and both children of w are black
- Case 3: x's sibling w is black, w's left child is red and w's right child is black
- Case 4: x's sibling w is black, and the right child of w is red

The key idea is that in each case the number of black nodes from the subtree root to each of the subtrees $\alpha, \beta, \ldots$ is preserved by the transformation.

The general strategy: (a) Transform Case 1 to Case 2 and Case 3 to Case 4. (b) Solve Case 2 and Case 4, respectively.

13.4 Deletion in red-black trees

Case 1: x's sibling w is red.

Since w must have black children (Why?),

- swap the colors of w and $p(x)$,
- perform a left-rotation on $p(x)$ (without violating any of red-black tree properties)

The new sibling of x, which is one of the w's children prior to the rotation, now is black. Thus, Case 1 has been converted to one of Cases 2, 3, and 4.

Cases 2, 3, and 4 occur when w is black, they are distinguished by the colors of w's children.

13.4 Deletion in red-black trees

Case 2: x's sibling w is black, and both children of node w are black.

Since w is black, take one black off from both x and w (as both children of w are black), leaving x with only one black and leaving (or coloring) w red.

To compensate for removing one black from x and w, we add an extra black to $p(x)$, which originally either red or black. If $p(x)$ was red, now it is black, done; otherwise, node $p(x)$ now becomes new node x, i.e., $p(x)$ now has double black on it.

Notice that node x (**with double black**) has been pushed one-layer higher if $p(x)$ was black already, where a layer near to the tree root is a higher layer. Otherwise, done.

The above procedure continues until (i) it reaches the tree root; (ii) or is converted into one of the other three cases. If the tree root becomes double black, remove the extra black, and Property 5 of the RBT is still held.

This case takes $O(h)$ time where h is the height of the RBT.

13.4 Deletion in red-black trees

Case 3: x's sibling w is black, w's left child is red while w's right child is black.

- swap the colors of w and its left child $left(w)$,
- perform a right rotation on w without violating any of the red-black tree properties.

The verification of the red-black tree properties can be done by checking whether any path passing through node w still contains the same number of black nodes from its source node to any of its descendant leaves before and after this transformation.

The new sibling w of x now is black with a red right child, this becomes Case 4.

13.4 Deletion in red-black trees

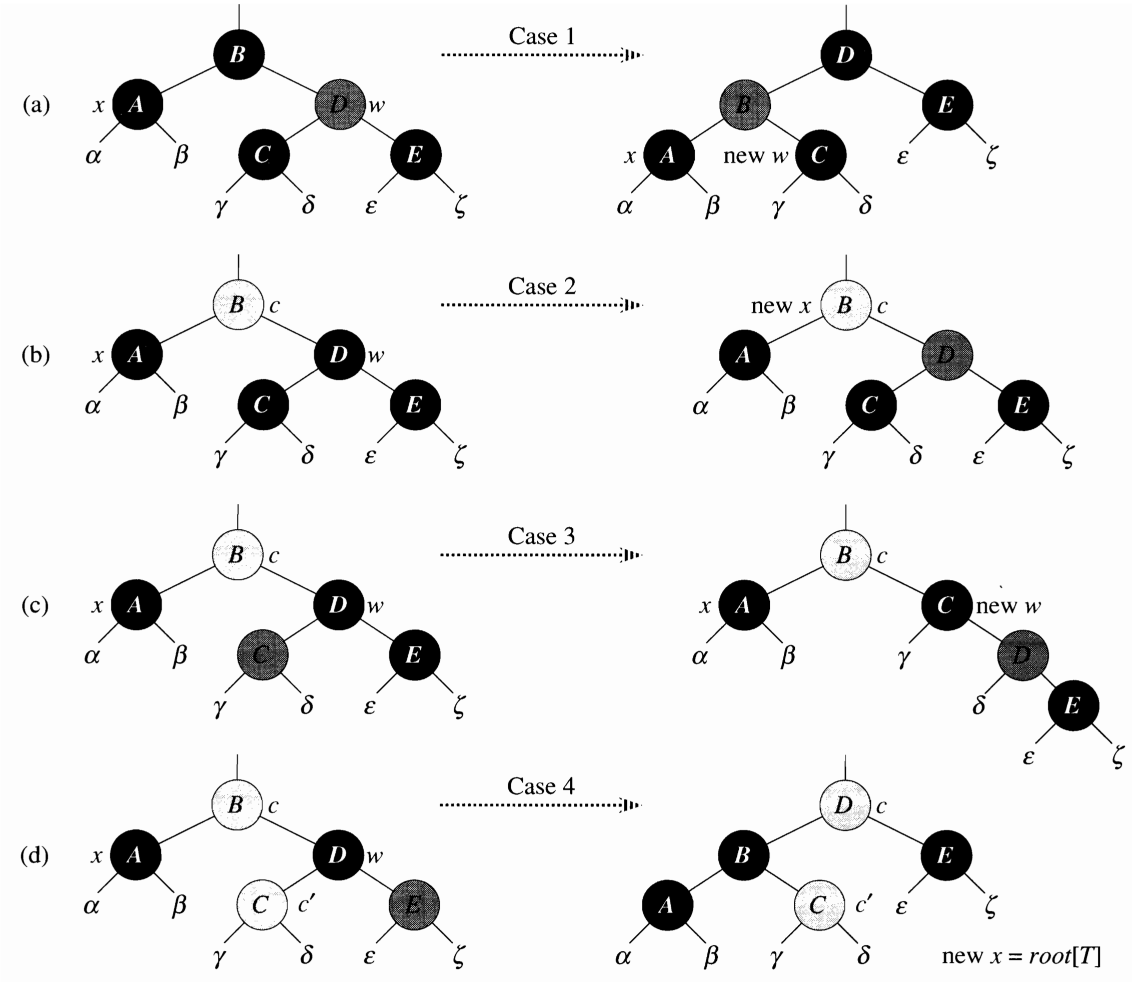
Case 4: x's sibling w is black, and the right child of w is red.

Remove the extra black represented by x, by changing some colors and performing a left rotation on node $p(x)$, the extra black of x can be removed, making it a single black without violating any red-black tree properties.

- swap the colors of w and its right child $right(w)$,
- swap colors of $p(x)$ and w
- perform the left rotation on $p(x)$,
- color x's parent $p(x)$ with black (i.e., remove the extra black from x.

The above operations do not violating any of the red-black tree properties.

13.4 Deletion in red-black trees



Four cases in fixing red-black errors during deletion (Cormen et al., p329)

13.4 Analysis on Deletion in red-black trees

Theorem: Given a RBT containing N nodes, the removal of one node from it takes $O \log N)$ time if the resulting tree is still a RBT.

The sketch of the proof: Since the height of any red-black tree is $O(\log N)$, each of Cases 1, 3 and 4 takes $O(1)$ time by performing a left or right rotation and re-coloring, while Case 2 takes $O(h)$ time, where h is the height of the red-black tree and $h = O(\log N)$ by the lemma in slides 5-6 of Lecture 13. Thus, the deletion operation takes $O(\log N)$ time.