Selected Solutions for Chapter 14: Augmenting Data Structures

Solution to Exercise 14.1-7

Let $A[1\,..\,n]$ be the array of n distinct numbers.

One way to count the inversions is to add up, for each element, the number of larger elements that precede it in the array:

$$\#\text{ of inversions} = \sum_{j=1}^{n} |\mathit{Inv}(j)| \ ,$$

where $\mathit{Inv}(j) = \{i : i < j \text{ and } A[i] > A[j]\}$.

Note that $|\mathit{Inv}(j)|$ is related to $A[j]$'s rank in the subarray $A[1\,..\,j]$ because the elements in $\mathit{Inv}(j)$ are the reason that $A[j]$ is not positioned according to its rank. Let $r(j)$ be the rank of $A[j]$ in $A[1\,..\,j]$. Then $j = r(j) + |\mathit{Inv}(j)|$, so we can compute

$$|\mathit{Inv}(j)| = j - r(j)$$

by inserting $A[1], \ldots, A[n]$ into an order-statistic tree and using OS-RANK to find the rank of each $A[j]$ in the tree immediately after it is inserted into the tree. (This OS-RANK value is $r(j)$.)

Insertion and OS-RANK each take $O(\lg n)$ time, and so the total time for n elements is $O(n \lg n)$.

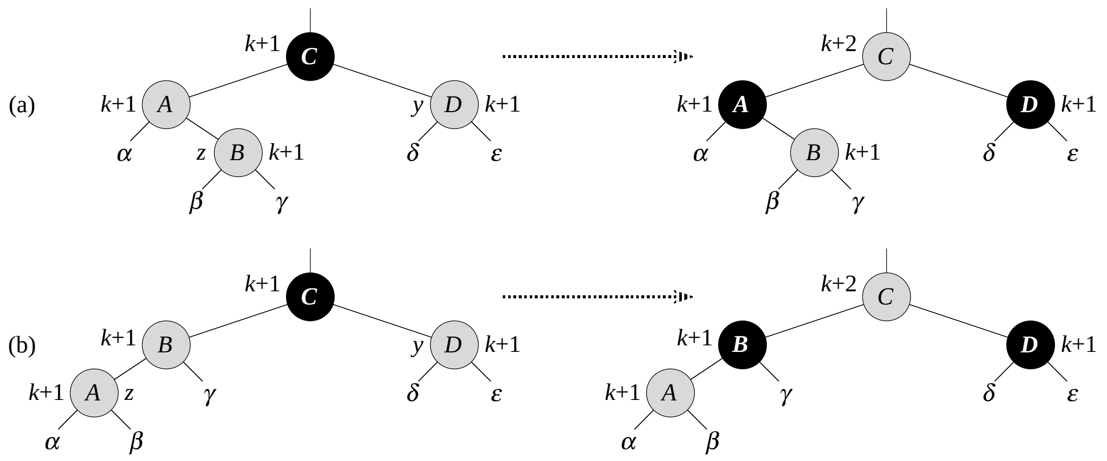
Solution to Exercise 14.2-2

Yes, we can maintain black-heights as attributes in the nodes of a red-black tree without affecting the asymptotic performance of the red-black tree operations. We appeal to Theorem 14.1, because the black-height of a node can be computed from the information at the node and its two children. Actually, the black-height can be computed from just one child's information: the black-height of a node is the black-height of a red child, or the black height of a black child plus one. The second child does not need to be checked because of property 5 of red-black trees.

Within the RB-INSERT-FIXUP and RB-DELETE-FIXUP procedures are color changes, each of which potentially cause $O(\lg n)$ black-height changes. Let us show that the color changes of the fixup procedures cause only local black-height changes and thus are constant-time operations. Assume that the black-height of each node x is kept in the attribute $x.bh$.

For RB-INSERT-FIXUP, there are 3 cases to examine.

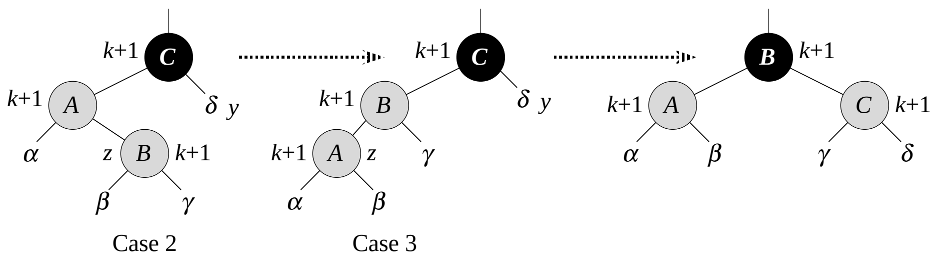
Case 1: z's uncle is red.



- Before color changes, suppose that all subtrees $\alpha, \beta, \gamma, \delta, \epsilon$ have the same black-height k with a black root, so that nodes A, B, C, and D have black-heights of $k + 1$.
- After color changes, the only node whose black-height changed is node C. To fix that, add $z.p.p.bh = z.p.p.bh + 1$ after line 7 in RB-INSERT-FIXUP.
- Since the number of black nodes between $z.p.p$ and z remains the same, nodes above $z.p.p$ are not affected by the color change.

Case 2: z's uncle y is black, and z is a right child.

Case 3: z''s uncle y is black, and z is a left child.

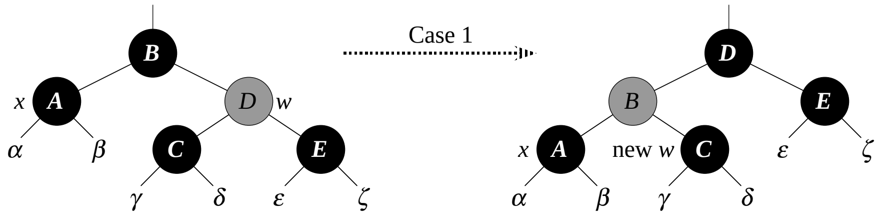


- With subtrees $\alpha, \beta, \gamma, \delta, \epsilon$ of black-height k, we see that even with color changes and rotations, the black-heights of nodes A, B, and C remain the same $(k + 1)$.

Thus, RB-INSERT-FIXUP maintains its original $O(\lg n)$ time.

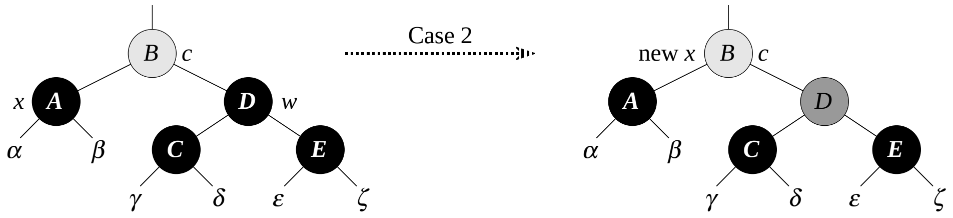
For RB-DELETE-FIXUP, there are 4 cases to examine.

Case 1: x's sibling w is red.



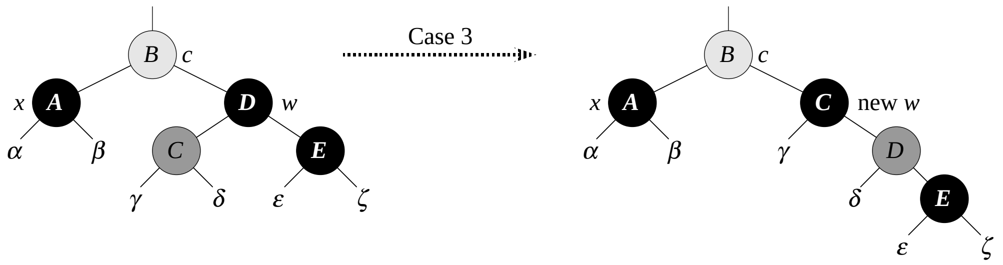
- Even though case 1 changes colors of nodes and does a rotation, black-heights are not changed.
- Case 1 changes the structure of the tree, but waits for cases 2, 3, and 4 to deal with the "extra black" on x.

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



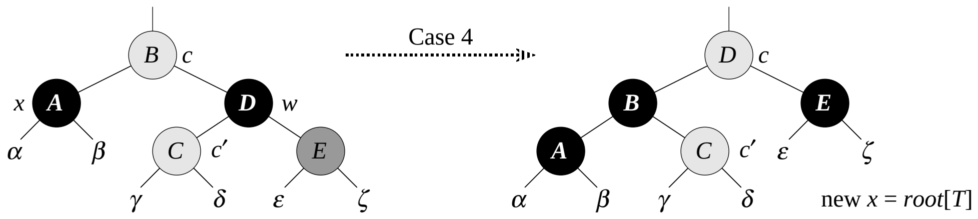
- w is colored red, and x's "extra" black is moved up to $x.p$.
- Now we can add $x.p.bh = x.bh$ after line 10 in RB-DELETE-FIXUP.
- This is a constant-time update. Then, keep looping to deal with the extra black on $x.p$.

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



- Regardless of the color changes and rotation of this case, the black-heights don't change.
- Case 3 just sets up the structure of the tree, so it can fall correctly into case 4.

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



- Nodes A, C, and E keep the same subtrees, so their black-heights don't change.
- Add these two constant-time assignments in RB-DELETE-FIXUP after line 20:

  $$x.p.bh = x.bh + 1$$
  $$x.p.p.bh = x.p.bh + 1$$

- The extra black is taken care of. Loop terminates.

Thus, RB-DELETE-FIXUP maintains its original $O(\lg n)$ time.

Therefore, we conclude that black-heights of nodes can be maintained as attributes in red-black trees without affecting the asymptotic performance of red-black tree operations.

For the second part of the question, no, we cannot maintain node depths without affecting the asymptotic performance of red-black tree operations. The depth of a node depends on the depth of its parent. When the depth of a node changes, the depths of all nodes below it in the tree must be updated. Updating the root node causes $n - 1$ other nodes to be updated, which would mean that operations on the tree that change node depths might not run in $O(n \lg n)$ time.

Solution to Exercise 14.3-7

General idea: Move a sweep line from left to right, while maintaining the set of rectangles currently intersected by the line in an interval tree. The interval tree will organize all rectangles whose x interval includes the current position of the sweep line, and it will be based on the y intervals of the rectangles, so that any overlapping y intervals in the interval tree correspond to overlapping rectangles.

Details:

1. Sort the rectangles by their x-coordinates. (Actually, each rectangle must appear twice in the sorted list—once for its left x-coordinate and once for its right x-coordinate.)
2. Scan the sorted list (from lowest to highest x-coordinate).

   - When an x-coordinate of a left edge is found, check whether the rectangle's y-coordinate interval overlaps an interval in the tree, and insert the rectangle (keyed on its y-coordinate interval) into the tree.
   - When an x-coordinate of a right edge is found, delete the rectangle from the interval tree.

   The interval tree always contains the set of "open" rectangles intersected by the sweep line. If an overlap is ever found in the interval tree, there are overlapping rectangles.

Time: $O(n \lg n)$

- $O(n \lg n)$ to sort the rectangles (we can use merge sort or heap sort).
- $O(n \lg n)$ for interval-tree operations (insert, delete, and check for overlap).