Balanced search trees

Ordinary binary search trees have expected height $\Theta(\log n)$ if items are inserted and deleted in random order, but for other orders the height can be $\Theta(n)$. This is undesirable, since many operations require time $\Theta(\text{tree height})$. Therefore, alternative tree structures have been devised. Common ones are:

- Red-black trees
- Weight-balanced trees
- Height-balanced trees (including AVL trees)
- B-trees (including 2-3 trees)
- Splay trees

Chapter 13 Red-Black Trees

A red-black tree (RBT) is a binary search tree with one extra bit of storage per node: its *colour* can be either RED or BLACK.

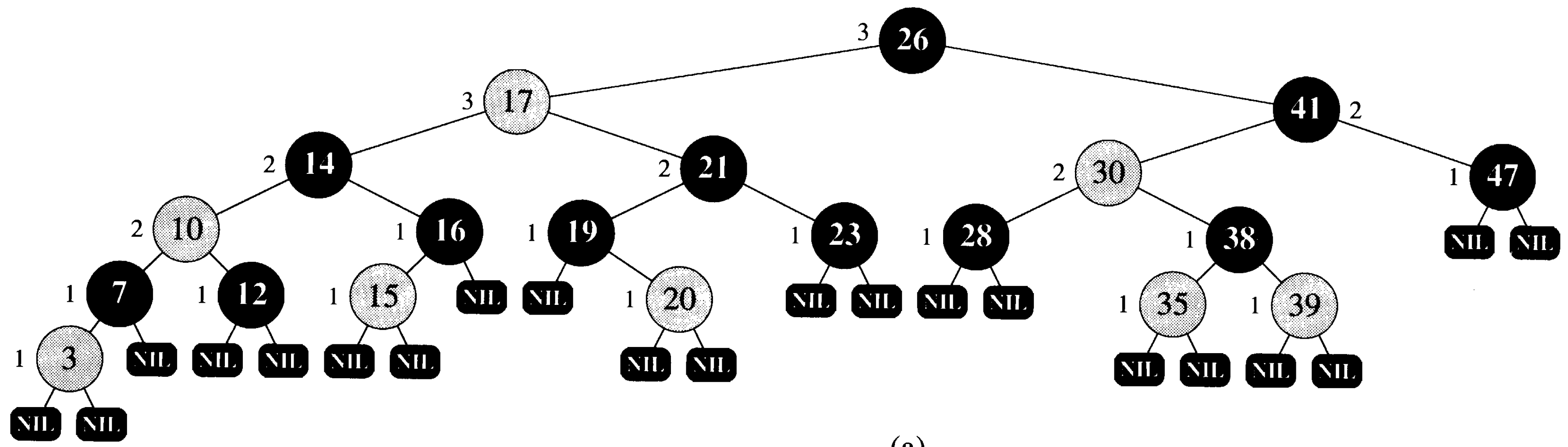
Each node in a RBT contains at least the following fields:

- color
- key
- left child pointer *left*()
- right child pointer *right*()
- parent pointer *p*()

13.1 Properties of red-black trees

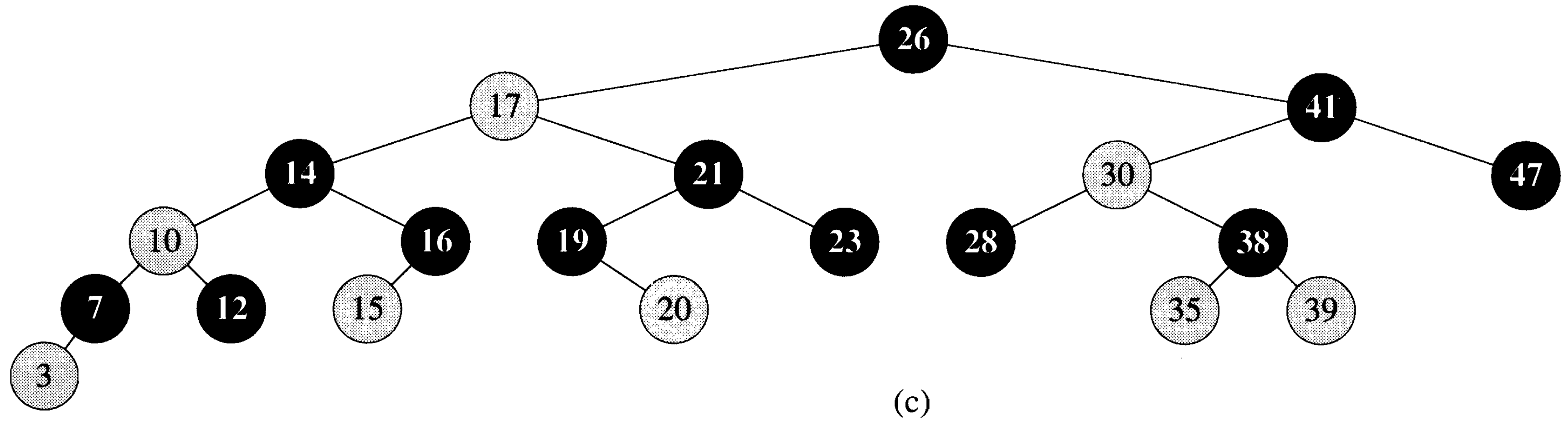
A binary search tree is a red-black tree if it satisfies the following **five red-black tree properties**.

- Every node is colored only with either RED or BLACK.
- The root node must be BLACK.
- Every leaf (NIL) must be BLACK.
- If a node is RED, then both its children are BLACK. This implies this node must be an internal node!
- Each path from a node x to any one of its descendant leaf nodes contains the same number of BLACK nodes. Note that the color of node x will be not counted.



(a)

What red-black trees really look like



(c)

How we draw them (Cormen, p310)

13.1 Properties of red-black trees (continued)

The black height $bh(x)$ of a node x in a RBT is the number of BLACK nodes on the path from node x down to a leaf, **not counting x itself but counting the leaf**. The definition of red-black tree guarantees that it doesn't matter which leaf below x is reached.

Lemma: A red-black tree with n internal nodes has height h at most $2\log(n+1)$.

Proof outline:

- Show by induction that the subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.
- Show that the height of the red-black tree h is at most twice the black height of the root $2 \cdot bh(root)$, i.e., $h \leq 2 \cdot bh(root)$ Why?.

13.1 Properties of red-black trees (cont.)

We show that **the subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes**, by mathematical induction on the height of node x. Let $h(x)$ represent the height of x.

- If the height of x is zero (i.e., $h(x) = 0$), then x must be a leaf, the subtree rooted at x contains at least $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes.
- Consider a node x, assume that its children (if they do exist), By the inductive assumption is held, the internal node in the subtree rooted at a child y of x is no less than $2^{bh(y)} - 1$.
- The rest is to show that the claim holds for node x, i.e., the number of internal node in the subtree rooted at x is at least $2^{bh(x)} - 1$.

13.1 Properties of red-black trees (cont.)

Consider a node x is an internal node with two children. Then, each child of node x has a black-height of either $bh(x)$ or $bh(x) - 1$, depending on whether node x is red or black. By the inductive hypothesis, each of the two children of x contains at least $2^{bh(x)-1} - 1$ internal nodes, the subtree rooted at x thus contains at least $2 \times (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ internal nodes.

13.1 Properties of red-black trees (continued)

- Let h be the height of the RBT, then $h \leq 2 \cdot bh(root)$ Why?, i.e., the black-height of the tree root is at least $h/2$, $\Rightarrow$, $bh(root) \geq h/2$. Then, the number of internal nodes contained in the RBT is at least $2^{bh(root)} - 1 \geq 2^{h/2} - 1$ by the lemma in slide 3.

  If there are n internal nodes in the RBT, then

$$n \geq 2^{h/2} - 1, \quad \Rightarrow \qquad h \leq 2\log(n+1).$$

Exercise: Assuming that a RBT contains n internal nodes, then, what is the maximum number N of nodes the RBT contained?

Answer: As the RBT is a binary tree, the maximum number of leaves of the tree is no more than $2n$ if there are n internal nodes. Thus, the number of node contained in the tree $N \leq n + 2n = 3n$. In other words, each of the mentioned 7 operations in the binary search tree can be implemented in a RBT within $O(h) = O(\log(n+1)) = O(\log N)$ time, where N is the number of elements in the RBT.

13.1 Operations on a red-black tree

Since the height of a red-black tree is at most $2\log(n+1)$, a red-black tree can support the operations SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR in time $O(\log n)$, where n is the number of internal nodes in the RBT.

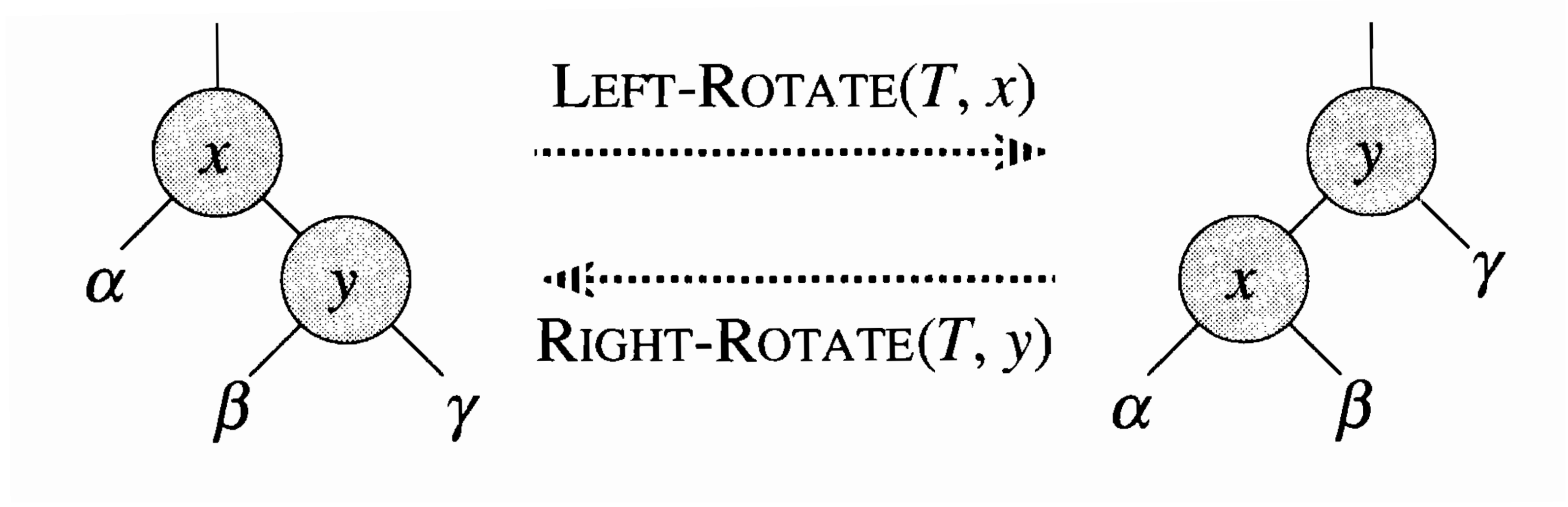
INSERT and DELETE operations on a red-black tree might destroy its defining properties. Insertion and deletion thus have to be accompanied by some operations that restore the red-black property.

For this, we may have to change

- the color of some nodes in the RBT.
- or the pointer structure of nodes in the RBT.

13.2 Rotations

To maintain a balanced tree T, we change the pointer structure by using Rotations: (1) The left Rotation; and (2) The right rotation.



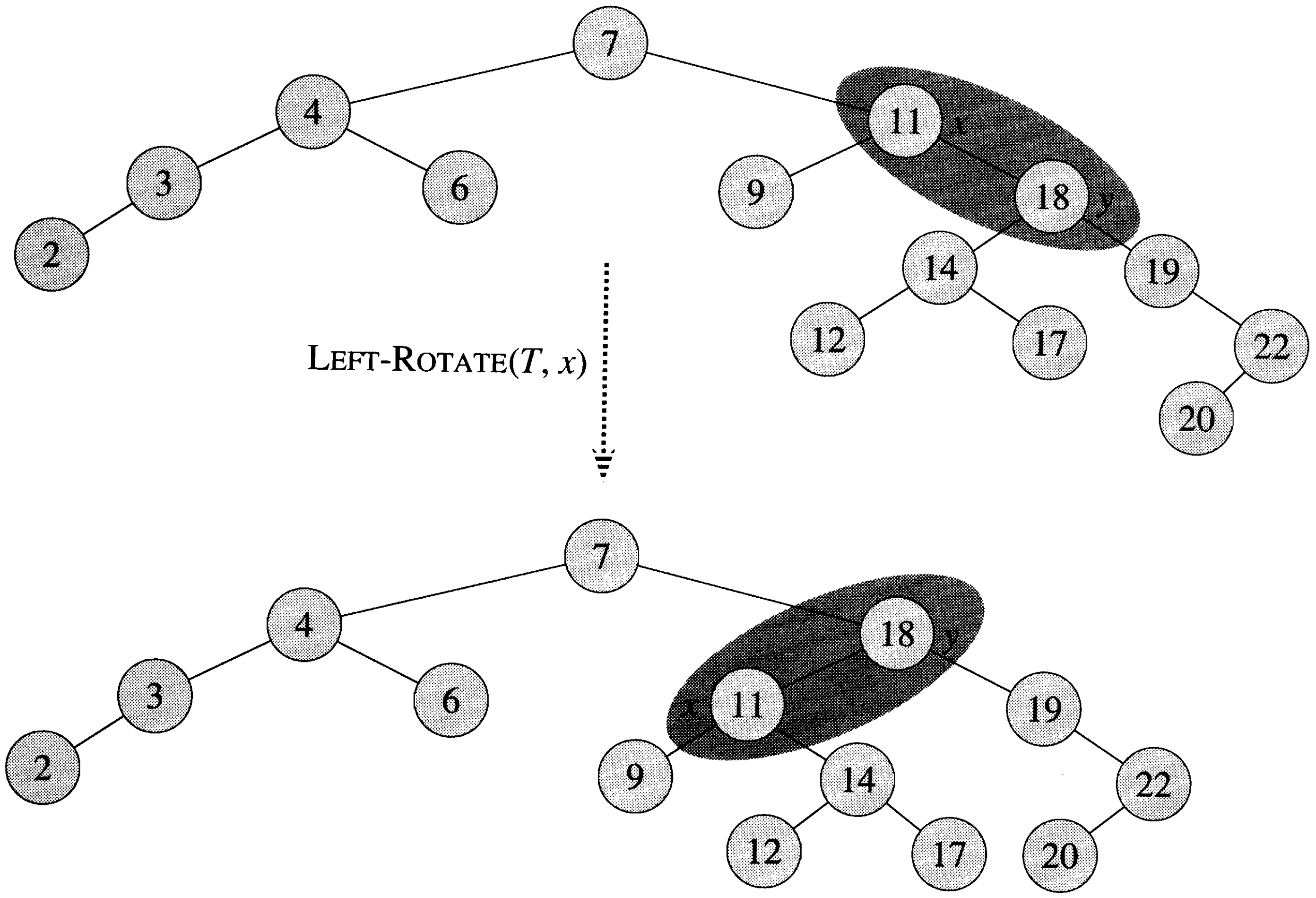
13.2 Rotations (cont.)

Notice that the operation of either the left rotation or the right rotation only takes $O(1)$ time by changing the pointers of several nodes, the binary search tree property of the resulting tree is still held. We show the claim as follows.

We show the claim after the left rotation.

Before the left rotation	The resulting tree after the left rotation
(i) $\alpha.key \leq x.key$	(I) $\alpha.key \leq x.key$
(ii) $x.key < y.key$	(II) $x.key \leq y.key$
(iii)$\beta.key \leq y.key$	(III) $\beta.key \leq y.key$
(iv) $y.key < \gamma.key$	IV) $y.key < \gamma.key$

13.2 Rotations (cont)



Example of a left rotation (Cormen, p314)

13.2 Left rotation

```
LEFT_ROTATE($T$, $x$)
1     $y \leftarrow right[x]$
2     $right[x] \leftarrow left[y]$
3     if $left[y] \neq NIL$
4         then $p[left[y]] \leftarrow x$
5     $p[y] \leftarrow p[x]$
6     if $p[x] = NIL$
7         then $root[T] \leftarrow y$
8         else if $x = left[p[x]]$
9             then $left[p[x]] \leftarrow y$
10            else $right[p[x]] \leftarrow y$
11    $left[y] \leftarrow x$; $p[x] \leftarrow y$
```

RIGHT_ROTATE(T,x) is similar, omitted.

13.3 Insertion in a red-black tree

We use the `TREE_INSERT` procedure for binary search trees to insert a node x into a red-black tree T as if it were a binary search tree, and then colour the new inserted node RED.

At this stage, two of the five red-black tree properties might be violated:

- The root might be RED (the original tree is empty, or the root is colored with red after a sequence of coloring changes on the tree).
- A RED node might have a RED parent.

However, only one violation of the TWO mentioned cases can exist. So, we need procedures for fixing these violations.

13.3 Insertion in a red-black tree (cont.)

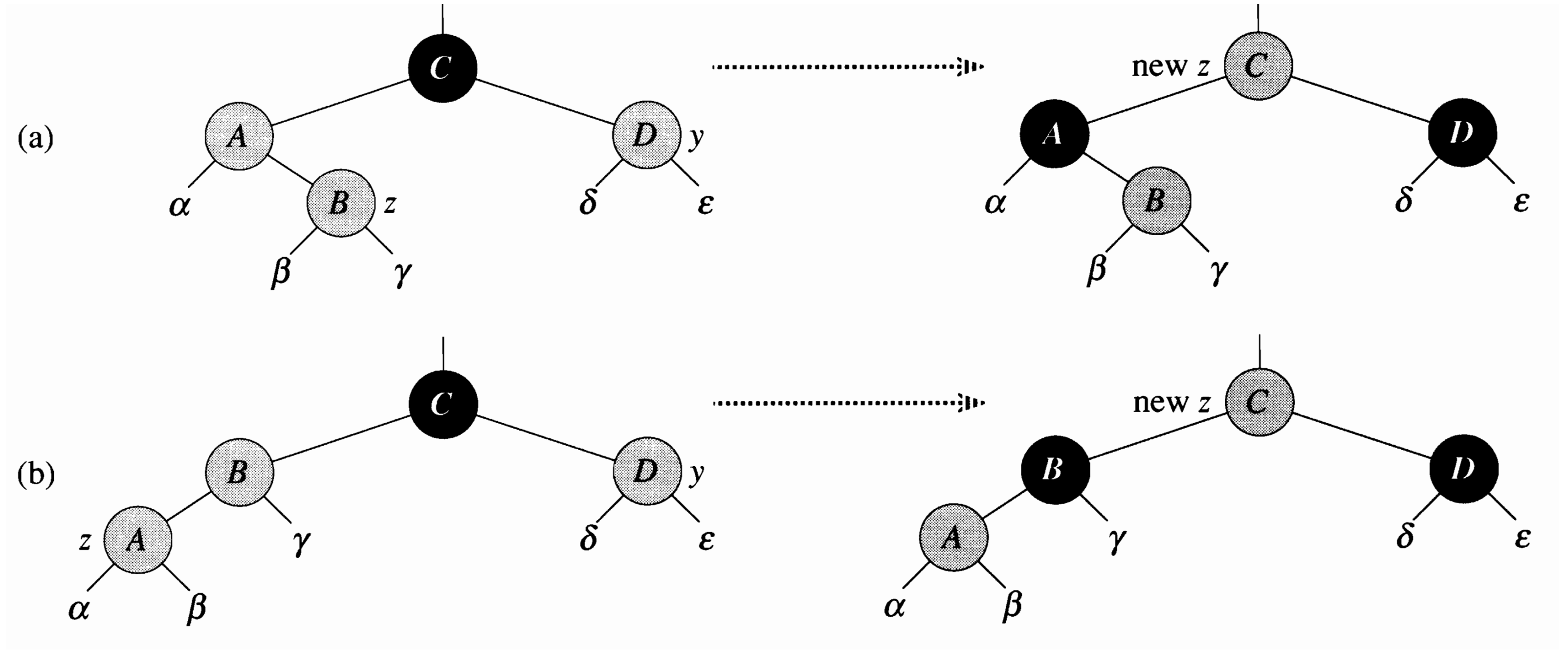
The general strategy to fix the color violations is as follows.

- If the second violation occurs, we do some operations that either fix it or move it to a higher level, where a level near to the tree root is higher.
- If it has not been fixed before we get to the tree root, the first violation occurs. We can fix the first violation by just setting the colour of the tree root to BLACK.

13.3 Insertion in a red-black tree (continued)

The detailed implementation of the general strategy is as follows. Suppose we have a RED node z which has a RED parent. There are three cases.

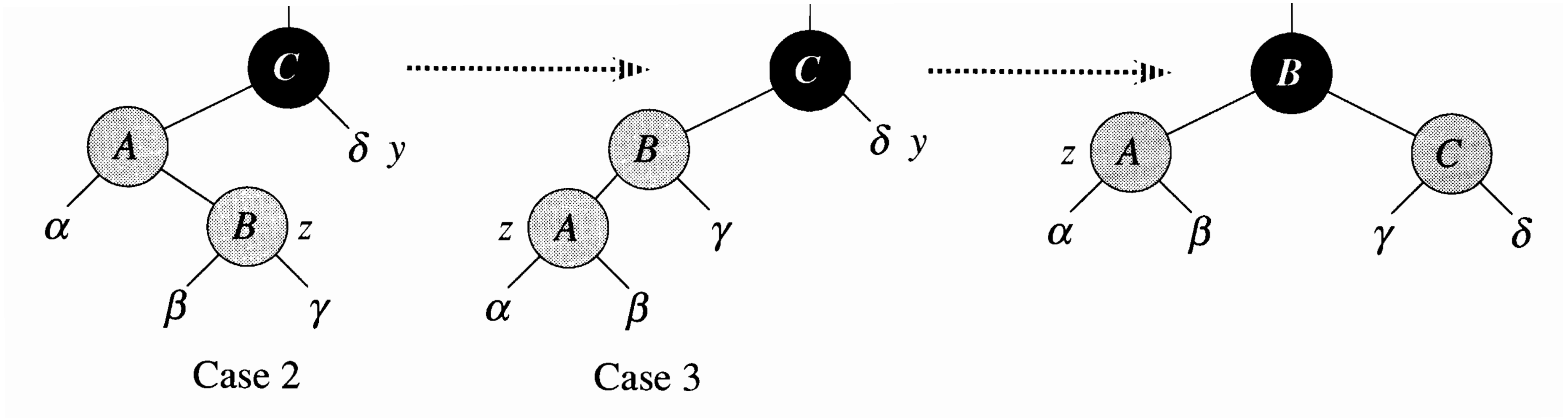
Case 1: z has a RED uncle y. Colour the parent and uncle of z BLACK and the grandparent of z RED.



First case of insertion fixup (Cormen, p320)

13.3 Insertion in a red-black tree (continued)

Cases 2 and 3: z has a BLACK uncle y. **If z is a right child** (case 2), do a left rotation at the parent of z then continue to case 3.
If z is a left child (case 3), colour the parent of z BLACK and the grandparent of z RED, then do a right rotation at the grandparent of z.



Second and third cases of insertion fixup (Cormen, p321)

13.3 Insertion in a red-black tree (continued)

Why the insertion process works.

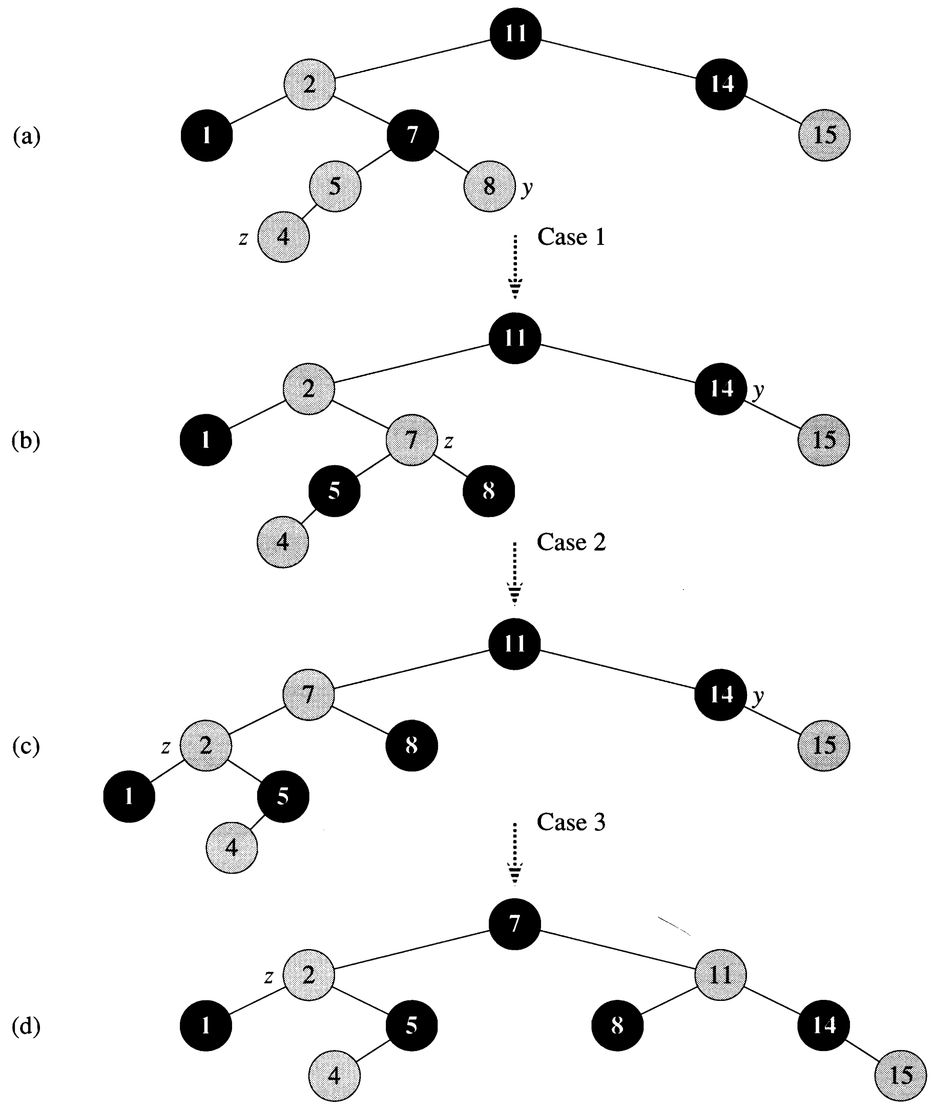
Induction hypothesis: At every stage, either

(a) there is exactly one RED with a RED parent,

(b) or the root is RED,

but not both of them.

- ➤ If case 1 occurs, the prescribed operations either fix the error or move it towards the root.
- ➤ If cases 2 or 3 occur, the prescribed operations fix the tree always.
- ➤ If the root is RED, making it BLACK fixes the tree.


(a)
11
2
14
1
7
15
5
8
y
z
4
Case 1
(b)
11
2
14
y
1
7
z
15
5
8
4
Case 2
(c)
11
7
14
y
z
2
8
15
1
5
4
Case 3
(d)
7
z
2
11
1
5
8
14
4
15