Red-Black Trees

- A red-black tree is a BST with one extra bit of storage per node: its color which can be either RED or BLACK.
- No path from the root to a leaf in a Red-Black tree is more than twice as long as any other.
- So the Red-Black tree is approximately *balanced*.

Red-Black Tree

1. Every node is either red or black
2. The root is black
3. Every leaf (NIL) is black
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Red-Black Trees…

- The number of black nodes on any path from, but not including, a node x down to a leaf is called the black-height of a node, denoted by bh(x).
- By property 5, the notion of a black-height is well defined, since all descending paths from the node have the same number of black nodes.
- The black-height of a red-black tree is the black height of its root.

Red-Black trees…

- A red-black tree with n internal nodes has height at most $2 \lg(n + 1)$
- So the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR can be implemented in $O(\lg n)$ time on red-black trees.

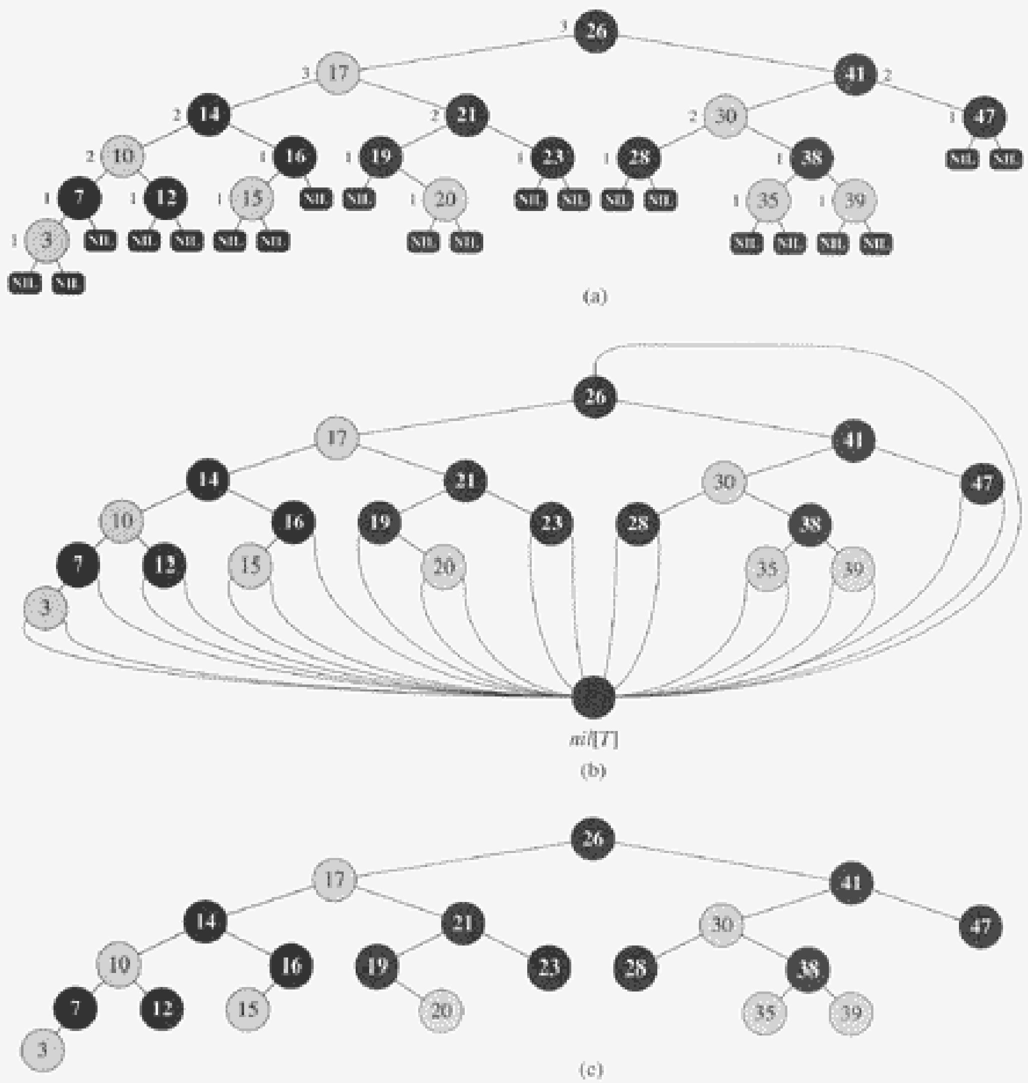


Figure 13.1 A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. **(a)** Every leaf, shown as a NIL, is black. Each non-NIL node is marked with its black-height; NIL's have black-height 0. **(b)** The same red-black tree but with each NIL replaced by the single sentinel $nil[T]$, which is always black, and with black-heights omitted. The root's parent is also the sentinel. **(c)** The same red-black tree but with leaves and the root's parent omitted entirely. We shall use this drawing style in the remainder of this chapter.

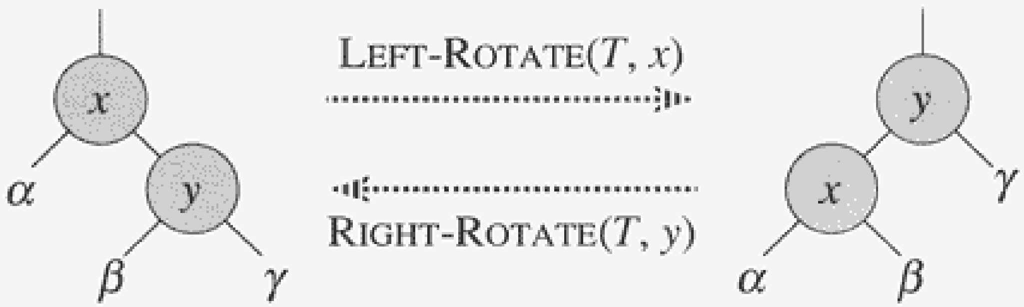


Figure 13.2 The rotation operations on a binary search tree. The operation LEFT-ROTATE(T, x) transforms the configuration of the two nodes on the left into the configuration on the right by changing a constant number of pointers. The configuration on the right can be transformed into the configuration on the left by the inverse operation RIGHT-ROTATE(T, y). The letters α, β, and γ represent arbitrary subtrees. A rotation operation preserves the binary-search-tree property: the keys in α precede $key[x]$, which precedes the keys in β, which precede $key[y]$, which precedes the keys in γ.

- To deal with boundary conditions in red-black tree code, we use a single sentinel to represent NIL.

- The sentinel $nil[T]$ is an object with the same fields as an ordinary node in the tree. Its color field is BLACK, and its other fields – *parent, left, right* and *key* – can be set to arbitrary values.

- $nil[T]$ is used to represent all the NILs – all leaves and the root's parent. The values parent, left, right and key fields of $nil[T]$ are immaterial.

- An *internal* node of a red-black tree is a node that holds a key value.

- *Black-height* of a node is the number of black nodes on any path from, but not including, a node x down to a leaf and is denoted as $bh(x)$

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

First we claim that the subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.

The proof of our claim is by induction on the height of x (recall that the height of a node is the length of the longest path from the node to ***a*** leaf).

Base case: Height of x is zero.
If the height of x is zero, then x must be a leaf, which in our notation is $nil[T]$. So the black height of x is 0, and the subtree rooted at x contains at least $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes. So the result is true for the base case.

Induction hypotheses: Assume that the result is true for any node with x with height $< k$.

Induction step: Consider a node x that has positive height k and is an internal node with two children.

- Each child of x has a black-height of either $bh(x)$ or $bh(x) - 1$ depending on whether the child's color is red or black respectively.
- Since the height of a child of x is $(k-1)$ or less, by induction hypotheses, we conclude that each child of x has at least $2^{bh(x)-1} - 1$ internal nodes.
- Thus the subtree rooted at x contains at least $2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 1 = 2^{bh(x)} - 1$ internal nodes thus proving our claim.

To complete the proof of the lemma, let h be the height of the red-black tree with n internal nodes. According to property 4 of the red-black trees, at least half the nodes on any simple path from the root to a leaf, not including the root, must be black. **So, the black-height of the root must be at least $h/2$.**

So, we have $n \geq 2^{h/2} - 1$ which is the same as $h/2 \leq \lg(n+1)$ or $h \leq 2\lg(n+1)$

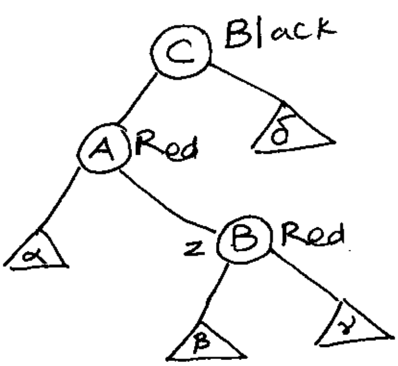
An immediate consequence of the above lemma is that the dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, can be implemented in $O(\lg n)$ time on red-black trees. In fact, INSERT AND DELETE can also be implemented in $O(\lg n)$ time on red-black trees.

Insertion:

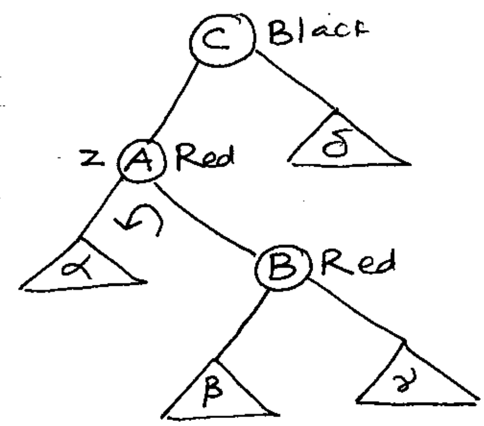
Node z is the node that got inserted. Node z is colored Red.

1. If z is the root, color z Black and we are done.
2. If $p[z]$ is the root, then color $p[z]$ Black (if it is not already) and we are done.
3. If z has a parent and its color is black, no more work needs to be done.
4. If z has a parent and its color is Red:
    - If z has a grandparent:
        - If z's uncle is Red
        - <u>Case 1</u>: (recall that $nil[T]$ has color Black) then color z's parent **and** z's uncle Black, color z's grandparent Red; make z its grandparent and go to step 1.

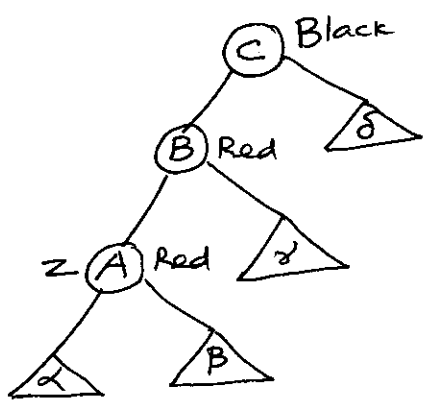
        - If z's uncle is Black:
        - <u>Case 2:</u> z, z's parent and z's grandparent are in a zig-zag pattern. Make z its parent, and rotate with respect to z. This makes the situation similar to case 3 and execute case 3 and we are done.



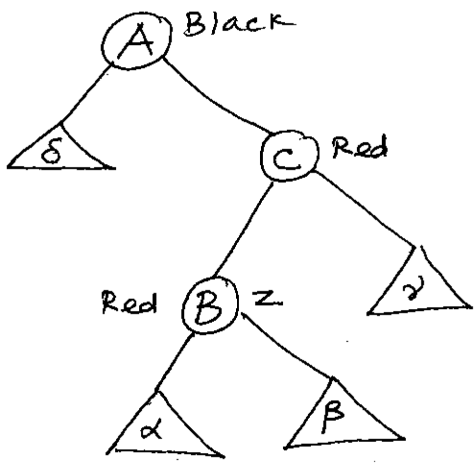
Make z its parent



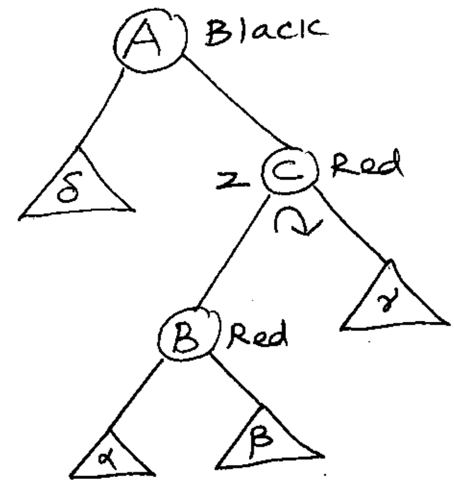
Left rotate with respect to z



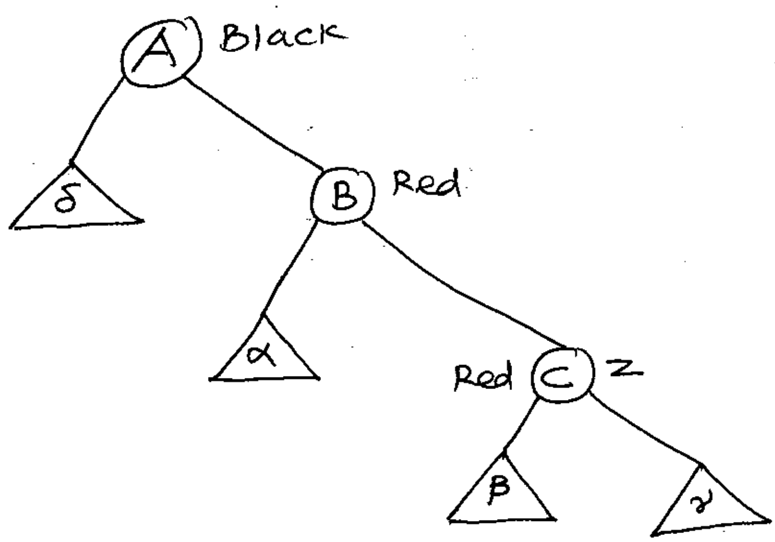
Now we have case 3's situation.



Make *z* its parent

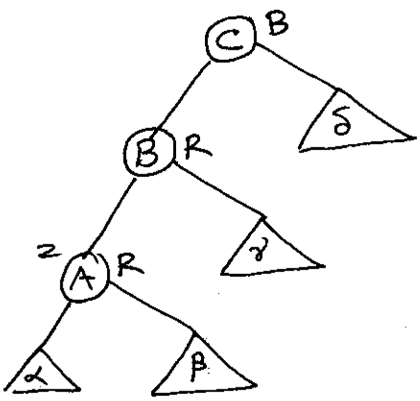


Right rotate with respect to *z*

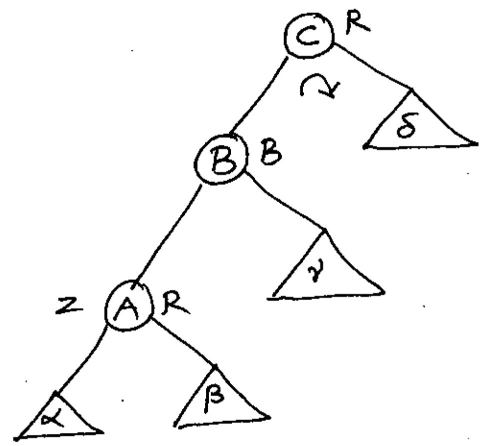


Now we have case 3's situation.

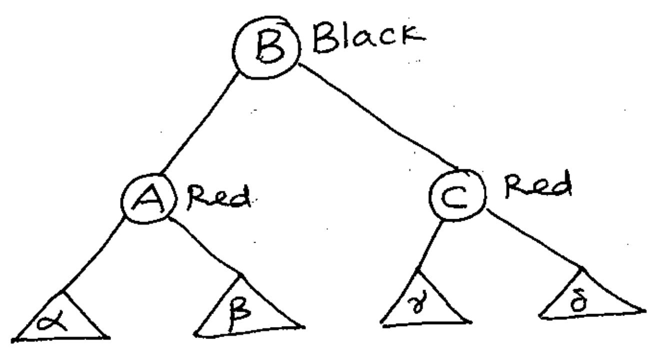
- Case 3: z, z's parent, z's grandparent are in a left-linear or right-linear pattern. Color z's grandparent Red, z's parent Black, rotate with respect to z's grandparent and we are done.



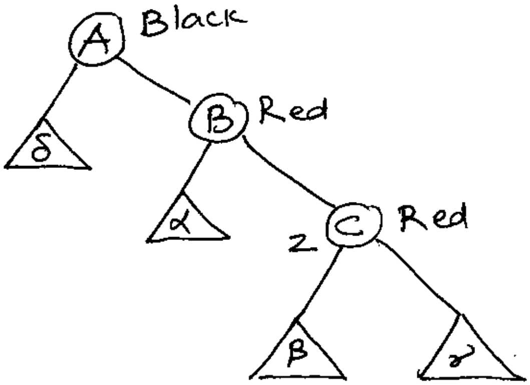
Color z's grandparent Red and z's parent Black



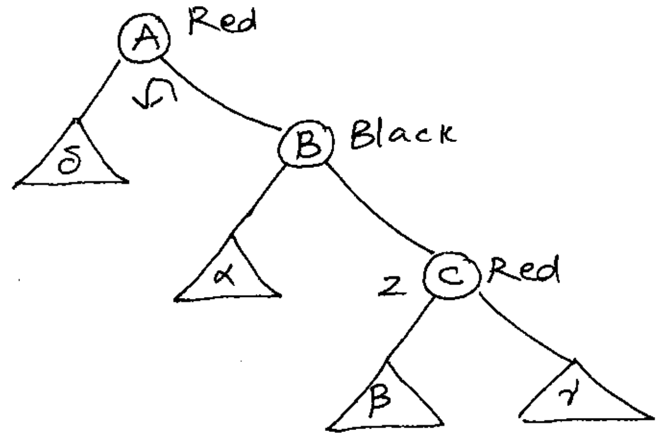
Right-rotate with respect to z's grandparent



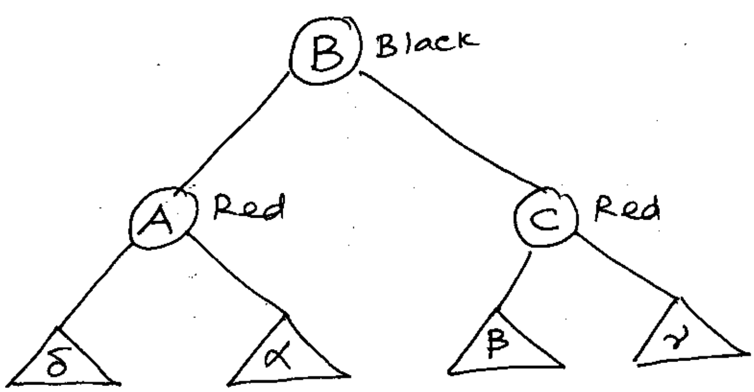
This is the final tree and we are done.



Color z's grandparent Red and z's parent Black



Left-rotate with respect to z's grandparent



This is the final tree and we are done.