# Red-Black Trees

# Introduction

We have seen that a binary search tree is a useful tool. I.e., if its height is *h*, then we can implement any basic operation on it in $O(h)$ units of time.

The problem: given an input of size *n*, how can we arrange it in a binary search tree of height $O(\log n)$? [We cannot expect better then that].

Red-Black trees are one of many search-trees that provide a good balanced solution to this problem.

# Properties of red-black trees

***red-black tree*** is a binary search tree with one extra bit of storage per node: its ***color***, which can be either RED or BLACK. By constraining the way nodes can be colored, red-black trees ensure that the tree is approximately ***balanced.*** I.e., the length of the longest path from the root to a leaf is not more then twice the length of the shortest one.

# Properties of red-black trees

Each node of the tree contains the fields *color, key, left, right,* and *p.* If a child or the parent of a node does not exist, the corresponding pointer field of the node contains the value NIL. We shall regard these NIL's as being pointers to external nodes (leaves) of the binary search tree and the normal, key-bearing nodes as being internal nodes of the tree.

# Properties of red-black trees

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

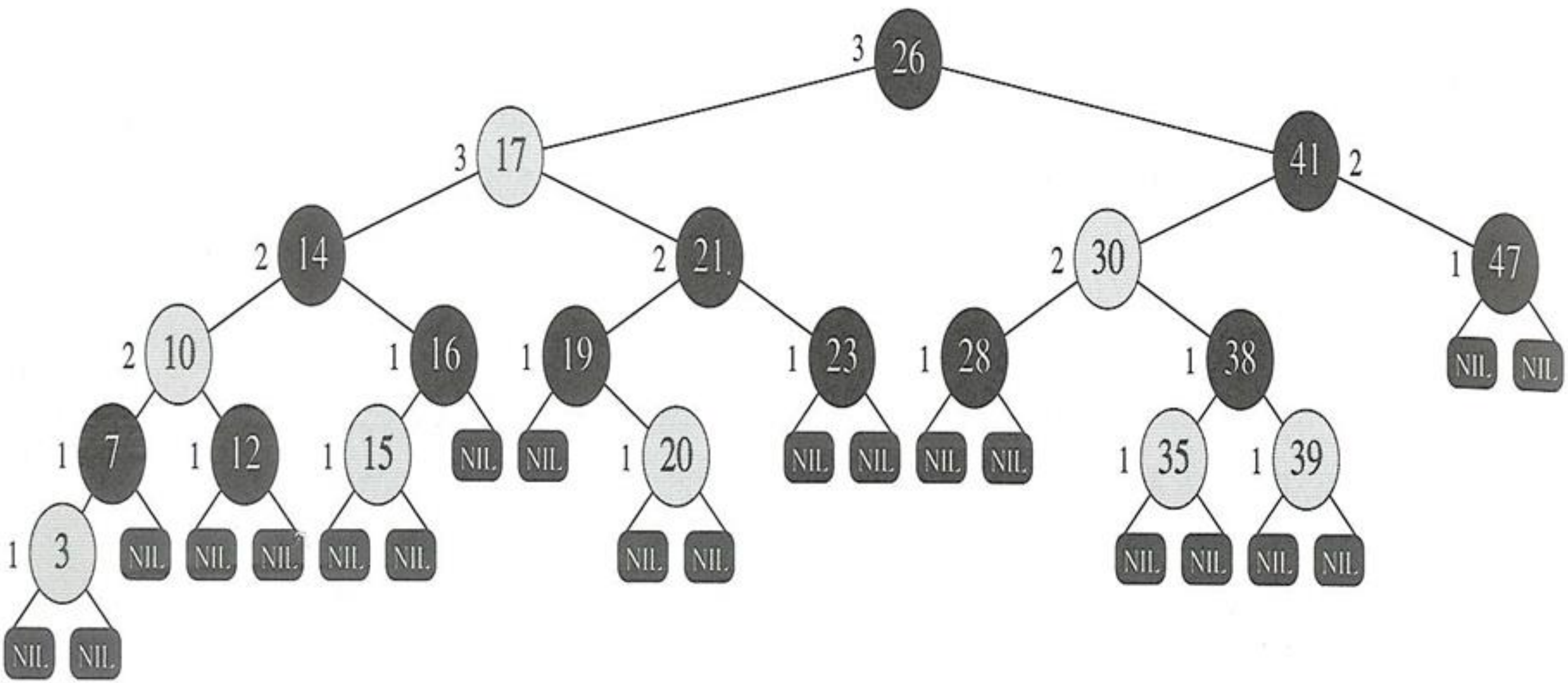
1. Every node is either red or black.

2. Every leaf (NIL) is black.

3. If a node is red, then both its children are black. ( no two red nodes in a row)

4. Every simple path from a node to a descendant leaf contains the same number of  black nodes.

5. (The root is black).



3 26
3 17
41 2
2 14
2 21
2 30
1 47
2 10
1 16
1 19
1 23
1 28
1 38
1 7
1 12
1 15
1 20
1 35
1 39
1 3
NIL

# Properties of red-black trees

We call the number of black nodes on any path from [but not including] a node $x$ to a leaf the ***black-height*** of the node, denoted by *bh(x)*. By property 4, the notion of black height is well defined, since all descending paths from a given node have the same number of black nodes. The black-height of the tree is defined as the black-height of the root.

# Properties of red-black trees

Red-black trees are good search trees

Lemma:

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

# Properties of red-black trees

Proof:

We first show that the sub-tree rooted at any node *x* contains at least $2^{bh(x)}-1$ internal nodes.

We prove this claim by **induction** on the height of *x*.

**If the height of *x* is *0***, then *x* must be a leaf (NIL), and the sub-tree rooted at *x* indeed contains at least $2^{bh(x)}-1 = 2^0-1 = 0$ internal nodes.

# Properties of red-black trees

For the **inductive step**, consider a node *x* that has positive height and is an internal node with two children. Each child has a black-height of either $bh(x)$ or $bh(x)-1$, depending on whether its color is red or black, respectively. Since the height of a child is less than the height of *x* itself, we can apply the **inductive hypothesis** to conclude that each child has at least $2^{bh(x)-1}-1$ internal nodes. Thus, the subtree rooted at *x* contains at least

$$(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1 \text{ internal nodes,}$$

which proves the claim.

# Properties of red-black trees

To complete the proof of the lemma, let $h$ be the height of the tree. According to property 3, at least half the nodes on any simple path from the root to a leaf, not including the root, must be black. Consequently, the black-height of the root must be at least $h/2$; thus:

$$n \geq 2^{h/2} - 1,$$

which yields $2\log(n+1) \geq h$. $\blacksquare$

# Properties of red-black trees

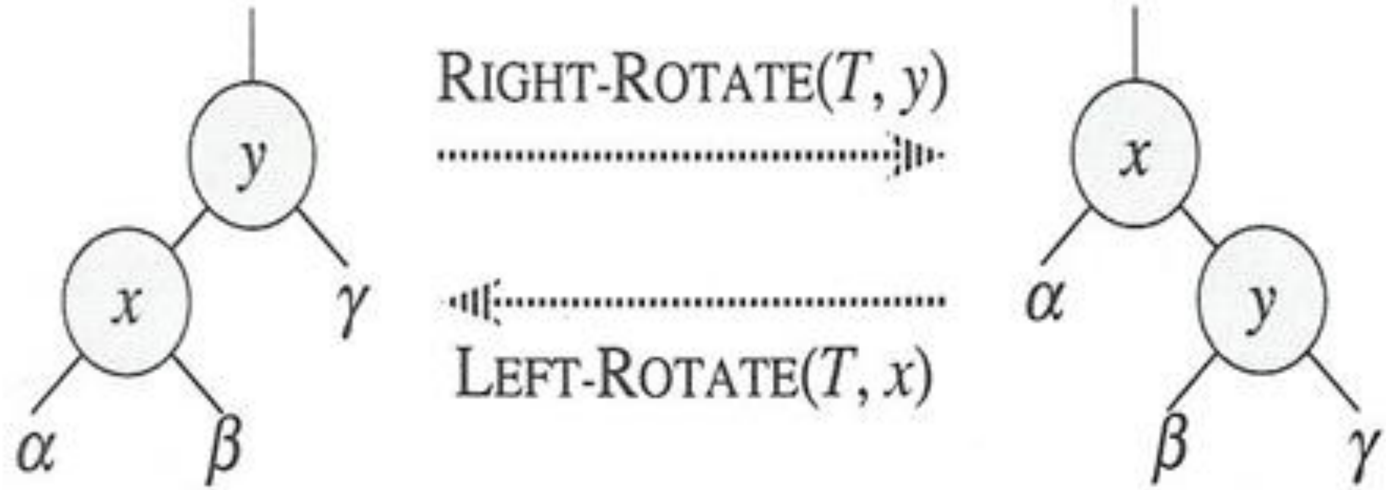
The dynamic-set operations SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR can all be implemented in time $O(\log n)$.

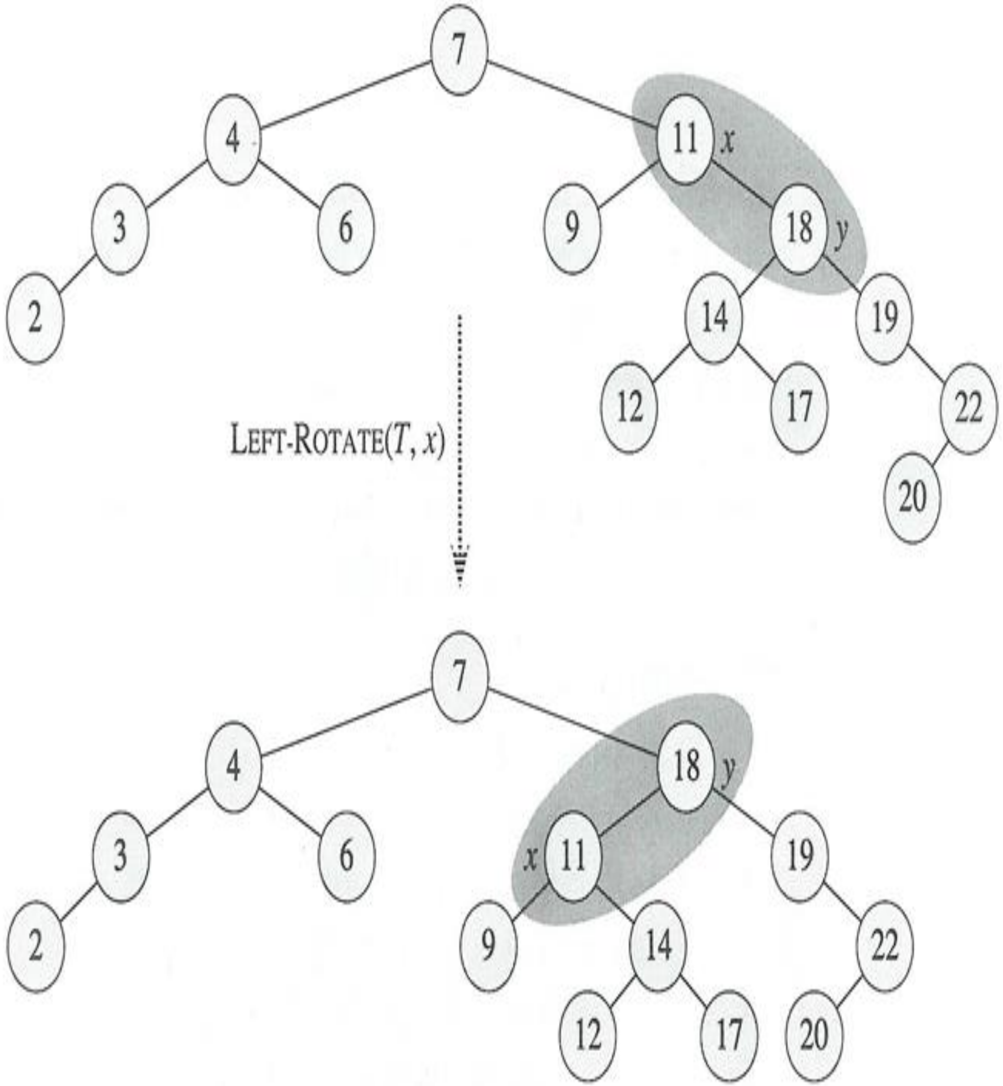
# Rotations

The search-tree operations TREE-INSERT and TREE-DELETE run in time $O(\log n)$. However, the result may violate the red-black properties. To restore these properties, we must recolor some of the nodes in the tree, and make some pointer changes.

We use ***rotations*** to change the pointer structure. They are presented by the following figures .Note that both LEFT-ROTATION and RIGHT-ROTATION run in *O(1)* of time. Only the pointers are changed – the other fields remain the same.

# Rotations






LEFT-ROTATE$(T, x)$

```
y = x.right
x.right = y.left
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x
x.p = y
```

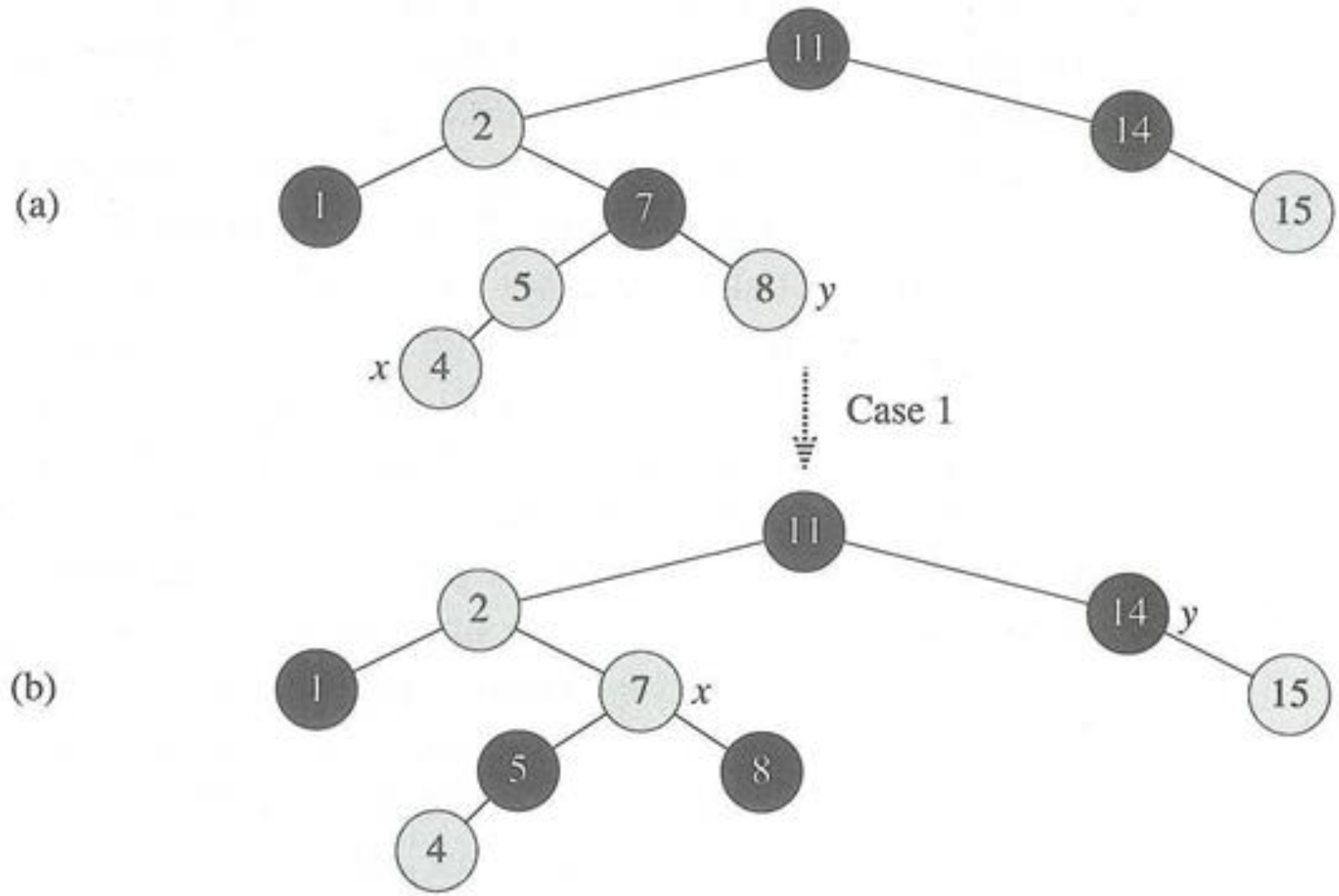
# Insertion

- We begin by inserting a node *x* into a tree *T*, as if *T* is an ordinary Binary search tree.
- We color *x* red.
- We fix up the modified tree by re-coloring nodes and performing rotations, to guarantee that the red-black properties are preserved.

Insertion is accomplished in $O(\log n)$ time.

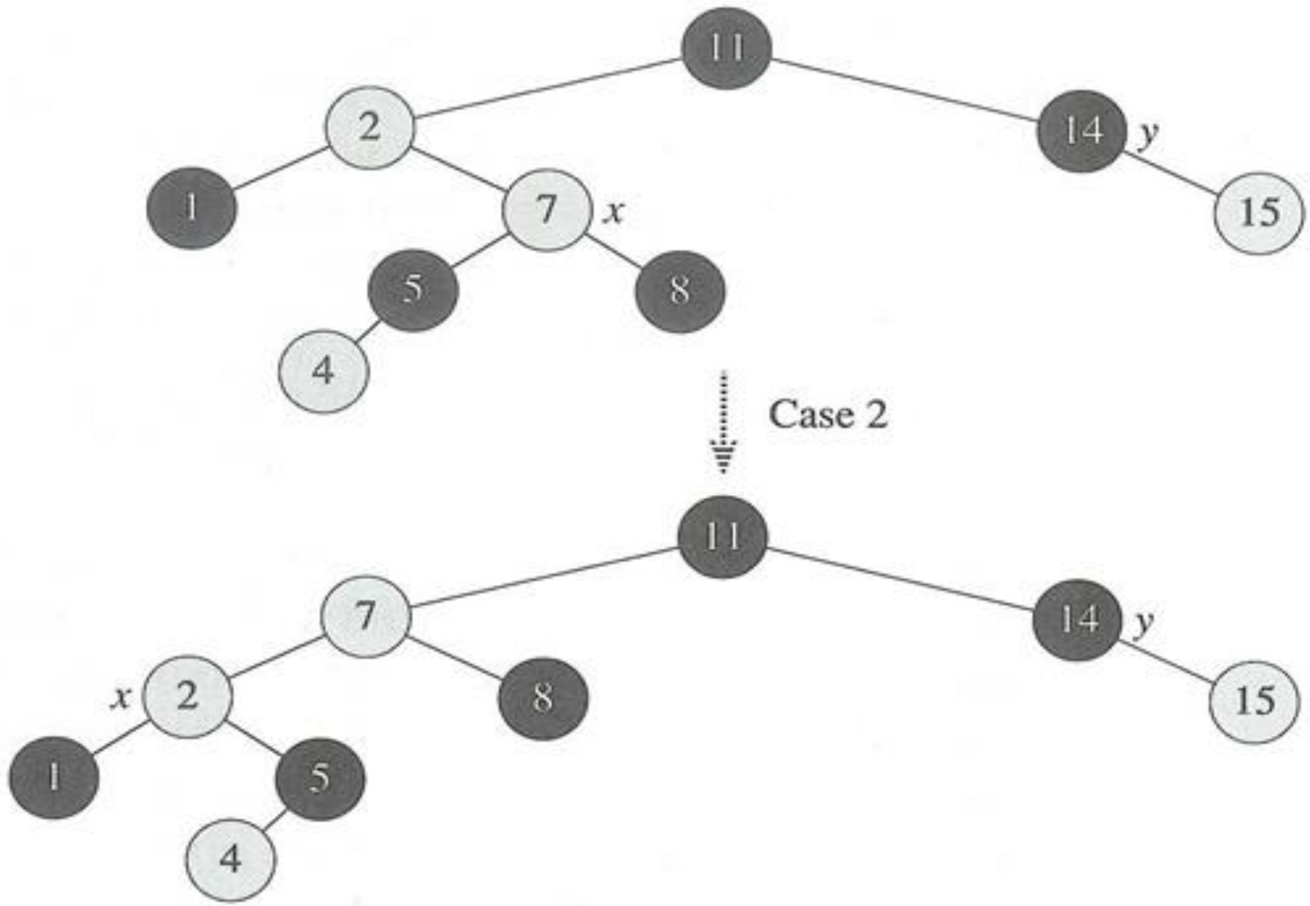
```
RB-INSERT(T, x)
TREE-INSERT(T, x)
color[x] ← RED
while x ≠ root[T] and color[p[x]] = RED
      do if p[x] = left[p[p[x]]]
            then y ← right[p[p[x]]]
                 if color[y] = RED
                    then color[p[x]] ← BLACK                ▷ Case 1
                         color[y] ← BLACK                   ▷ Case 1
                         color[p[p[x]]] ← RED               ▷ Case 1
                         x ← p[p[x]]                        ▷ Case 1
                    else if x = right[p[x]]
                            then x ← p[x]                   ▷ Case 2
                                 LEFT-ROTATE(T, x)          ▷ Case 2
                         color[p[x]] ← BLACK                ▷ Case 3
                         color[p[p[x]]] ← RED               ▷ Case 3
                         RIGHT-ROTATE(T, p[p[x]])           ▷ Case 3
            else (same as then clause
                    with "right" and "left" exchanged)
color[root[T]] ← BLACK
```

# Insertion – case1



# Insertion- case2

(b)




(c)

# Insertion –case3