Theory of algorithms (6th lecture)

Pál Pusztai
pusztai@sze.hu

Outline

- Red-black trees
  - Properties, rotations, insertion
- Augmenting data structure
  - Steps of the process
  - Examples
    - Dynamic order statistics
    - Interval trees
- Exercises

Red-black trees

A **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 node colors on any simple path from the root to a leaf, red-black trees ensure that no such path is more than twice as long as any other.

Red-black trees are approximately **balanced** and guarantee that basic dynamic-set operations take $O(\lg n)$ time in the worst case.

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

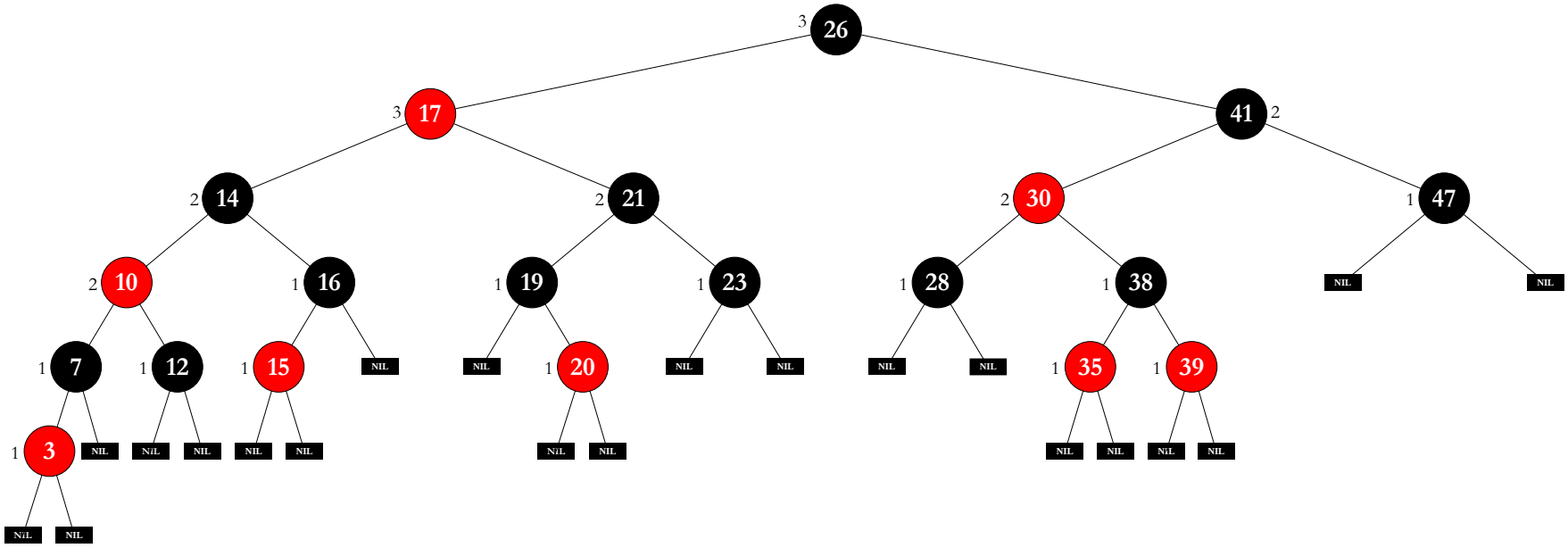
1. Every node is either red or black.
2. The root is black.
3. Every leaf (NIL) is black.
4. If a node is red, then both its children are black.
5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

We call the number of black nodes on any simple path from, but not including, a node x down to a leaf the **black-height** of the node.

The black-height of a red-black tree is the black-height of its root.

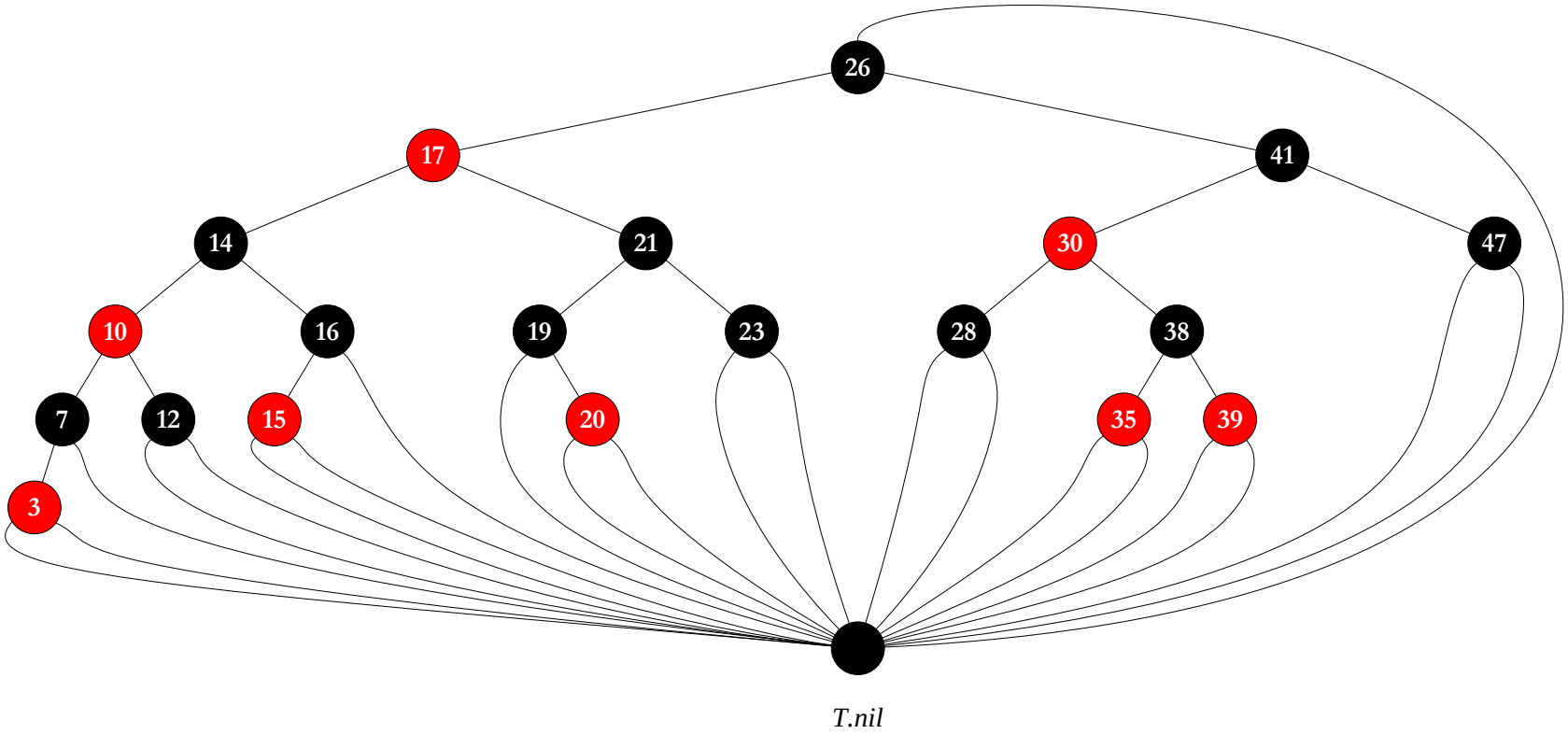
Theory: A red-black tree with n internal nodes has height at most $2\lg(n+1)$.

Red-black trees



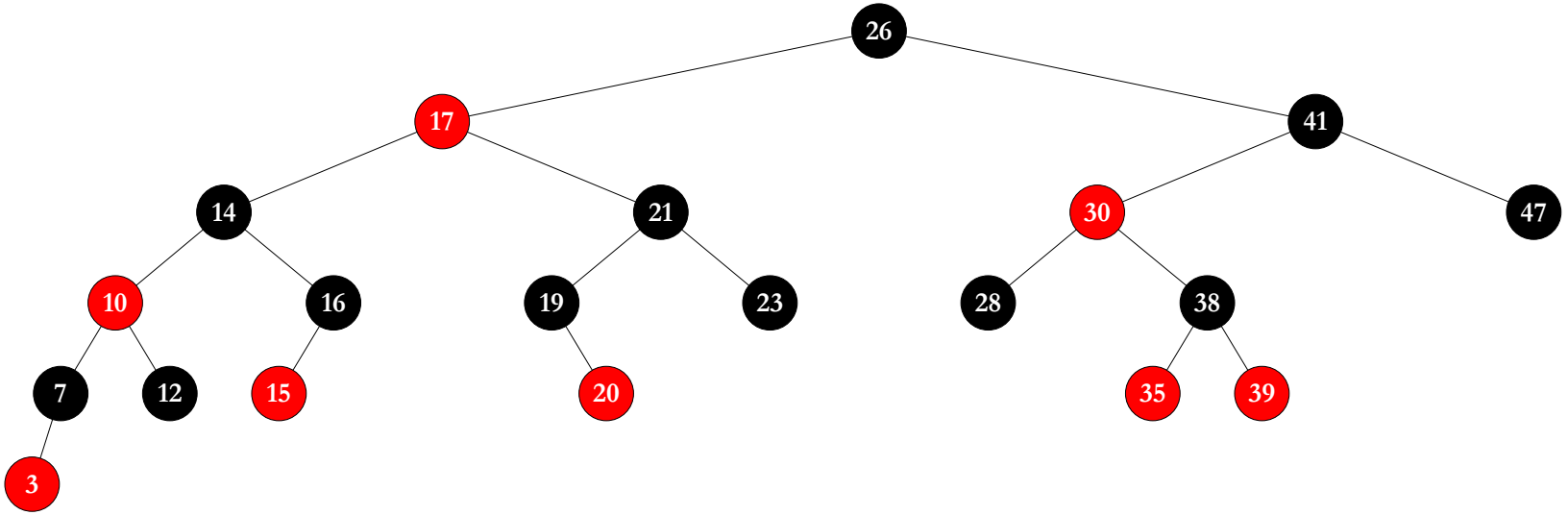
A red-black tree with the black-heights

Red-black trees



The same red-black tree with sentinel *T.nil*

Red-black trees

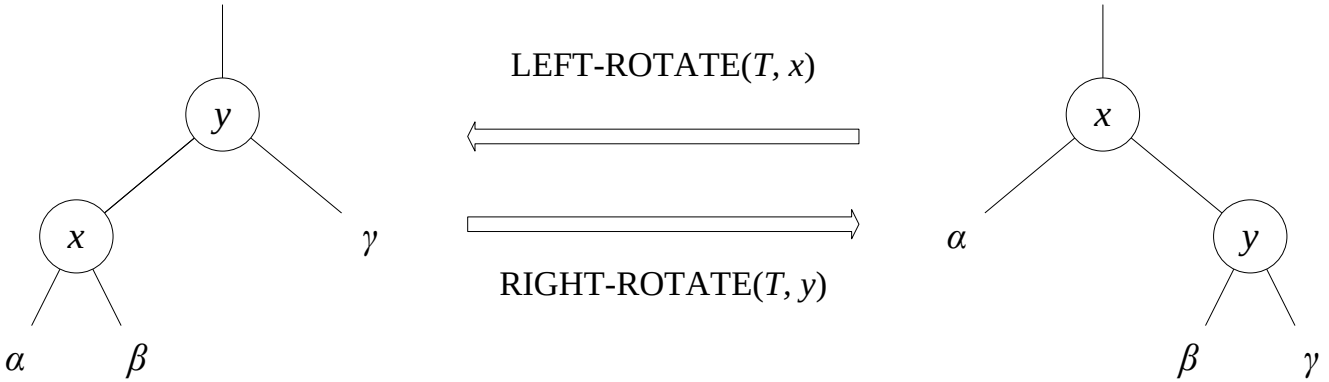


The same red-black tree without leaves and the root's parent

Exercises

- Draw the complete binary search tree of height 3 on the keys {1, …, 15}. Add the NIL leaves and color the nodes in three different ways such that the black-heights of the resulting red-black trees are 2, 3, and 4.
- Draw the red-black tree that results after TREE-INSERT is called on the tree in the previous slide with key 36. If the inserted node is colored red, is the resulting tree a red-black tree? What if it is colored black?
- Describe a red-black tree on n keys that realizes the largest possible ratio of red internal nodes to black internal nodes. What is this ratio? What tree has the smallest possible ratio, and what is the ratio?

Red-black trees



The rotation operations on a binary search tree

Red-black trees

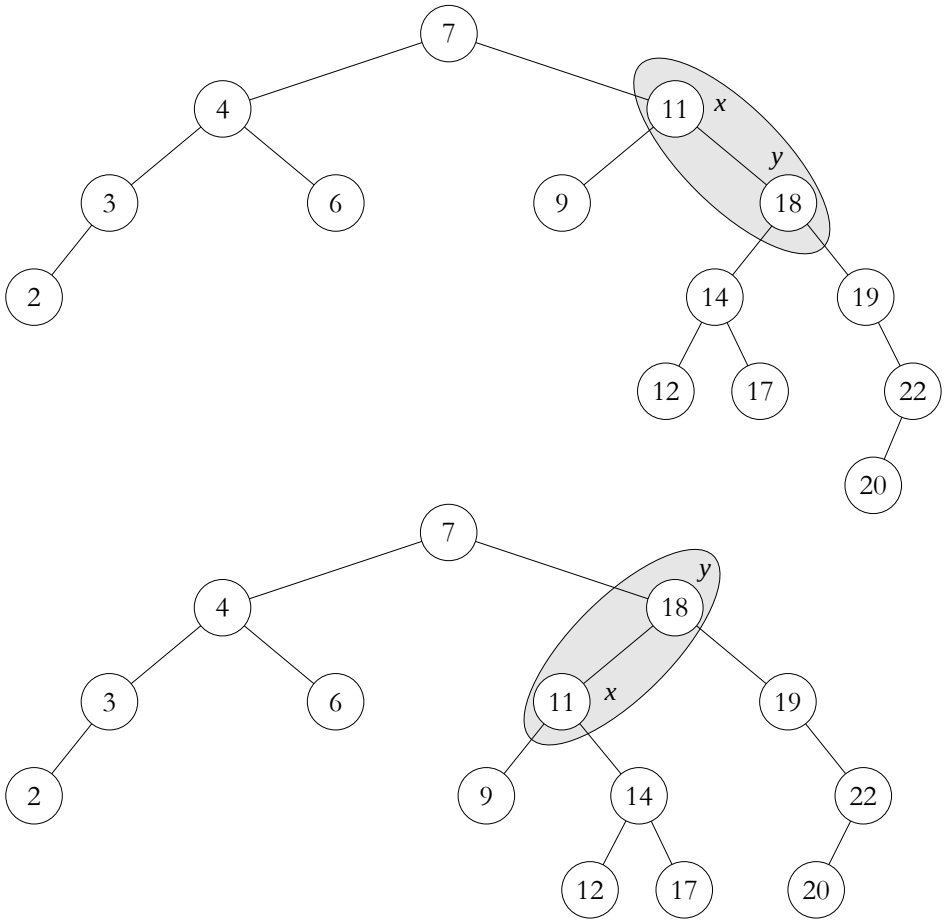
```
LEFT-ROTATE(T, x)
1   y = x.right                 // set y
2   x.right = y.left            // turn y's left subtree into x's right subtree
3   if y.left ≠ T.nil
4           y.left.p = x
5   y.p = x.p                   // link x's parent to y
6   if x.p == T.nil
7           T.root = y
8   else if x = x.p.left
9           x.p.left = y
10  else
11          x.p.right = y
12  y.left = x                  // put x on y's left
13  x.p = y
```

Remarks:

- The pseudocode for LEFT-ROTATE assumes that $x.right \neq T.nil$ and that the root's parent is $T.nil$.
- The code for RIGHT-ROTATE is symmetric.

Efficiency: Both LEFT-ROTATE and RIGHT-ROTATE run in $O(1)$ time.

Red-black trees



The operation of LEFT-ROTATE(*T*, *x*)

Exercises

- Give the binary search tree after LEFT-ROTATE is called on the tree in the bottom of the previous slide with the root element.
- What is the result if RIGHT-ROTATE is called?

Red-black trees

```
RB-INSERT(T, z)
1   y = T.nil
2   x = T.root
3   while x ≠ T.nil
4           y = x
5           if z.key < x.key
6                   x = x.left
7           else
8                   x = x.right
9   z.p = y
10  if y == T.nil
11          T.root = z
12  else
13          if z.key < y.key
14                  y.left = z
15          else
16                  y.right = z
17  z.left = T.nil
18  z.right = T.nil
19  z.color = RED
20  RB-INSERT-FIXUP(T, z]
```

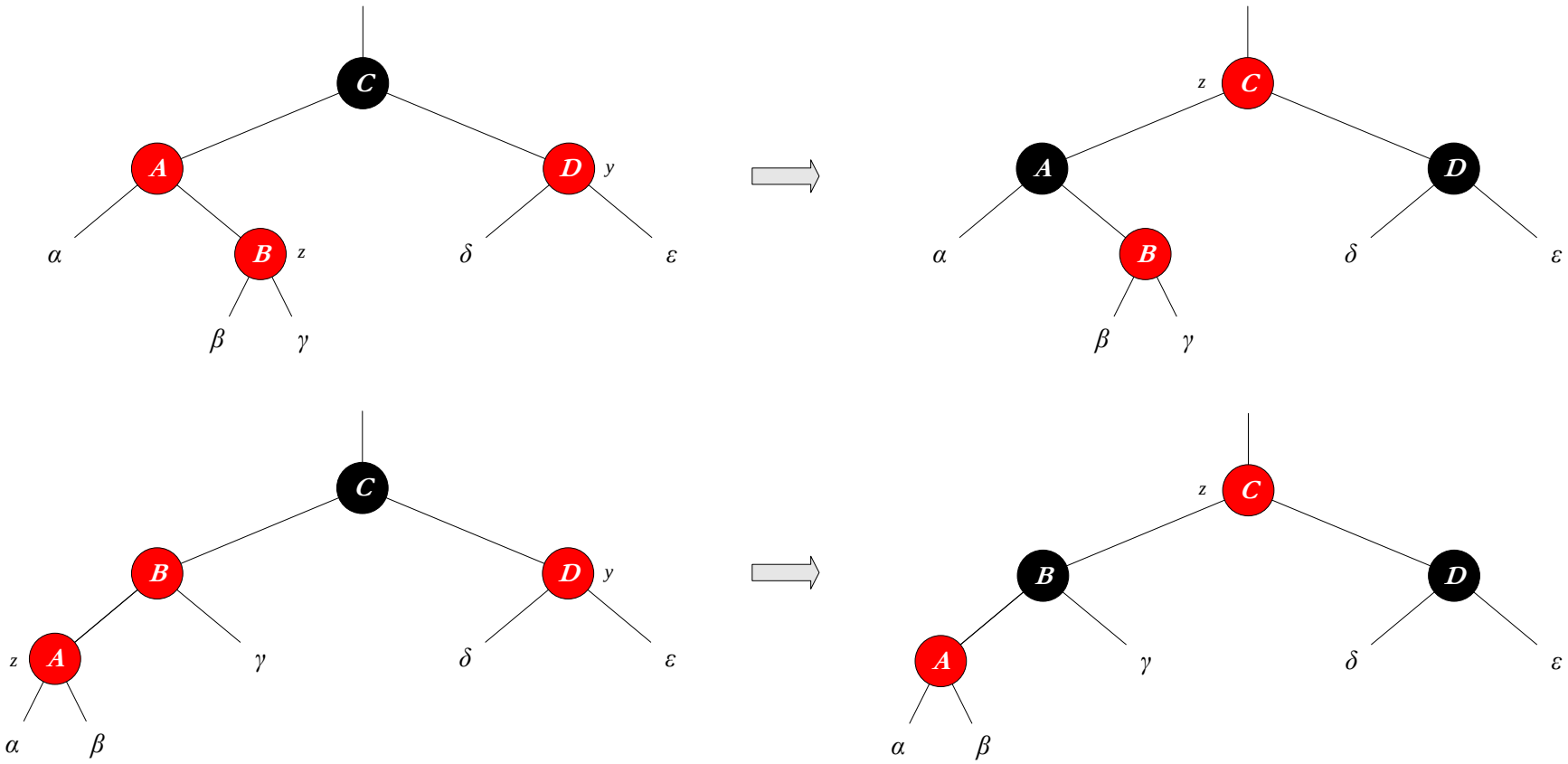
Red-black trees

```
RB-INSERT-FIXUP(T, z)
1    while z.p.color == RED
2          if z.p == z.p.p.left
3                y = z.p.p.right
4                if y.color == RED
5                      z.p.color = BLACK                    // case 1
6                      y.color = BLACK                      // case 1
7                      z.p.p.color = RED                    // case 1
8                      z = z.p.p                            // case 1
9                else
10                     if z == z.p.right
11                           z = z.p                        // case 2
12                           LEFT-ROTATE(T, z)              // case 2
13                     z.p.color = BLACK                    // case 3
14                     z.p.p.color = RED                    // case 3
15                     RIGHT-ROTATE(T, z.p.p)               // case 3
16         else        // z's parent is the right child of its parent
17               same as the true case with „left" and „right" exchanged
18   T.root.color = BLACK
```

Remarks: Note that if *z.p.color* == RED then node *z.p.p* exists.

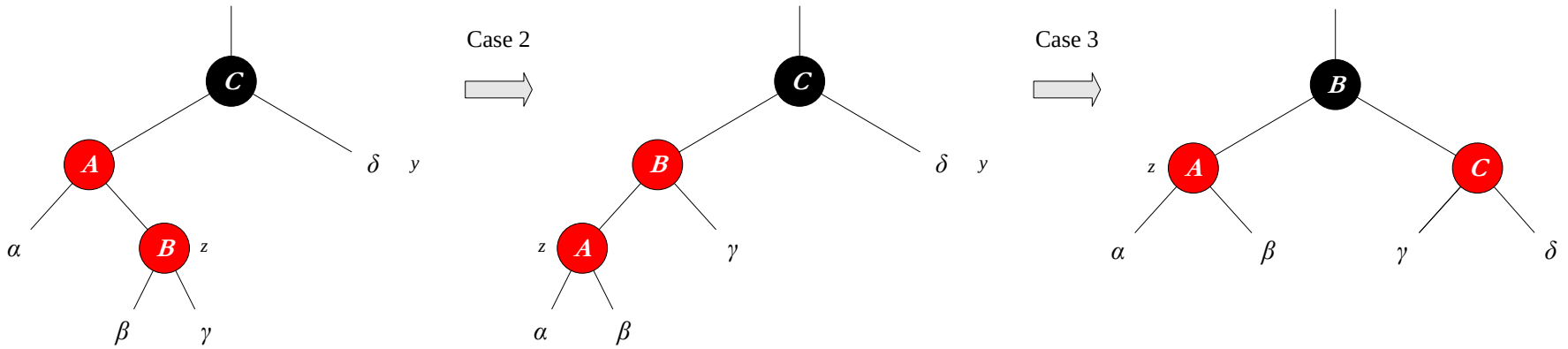
Efficiency: The running time of both procedures in an n-node red-black tree is $O(\lg n)$.

Red-black trees



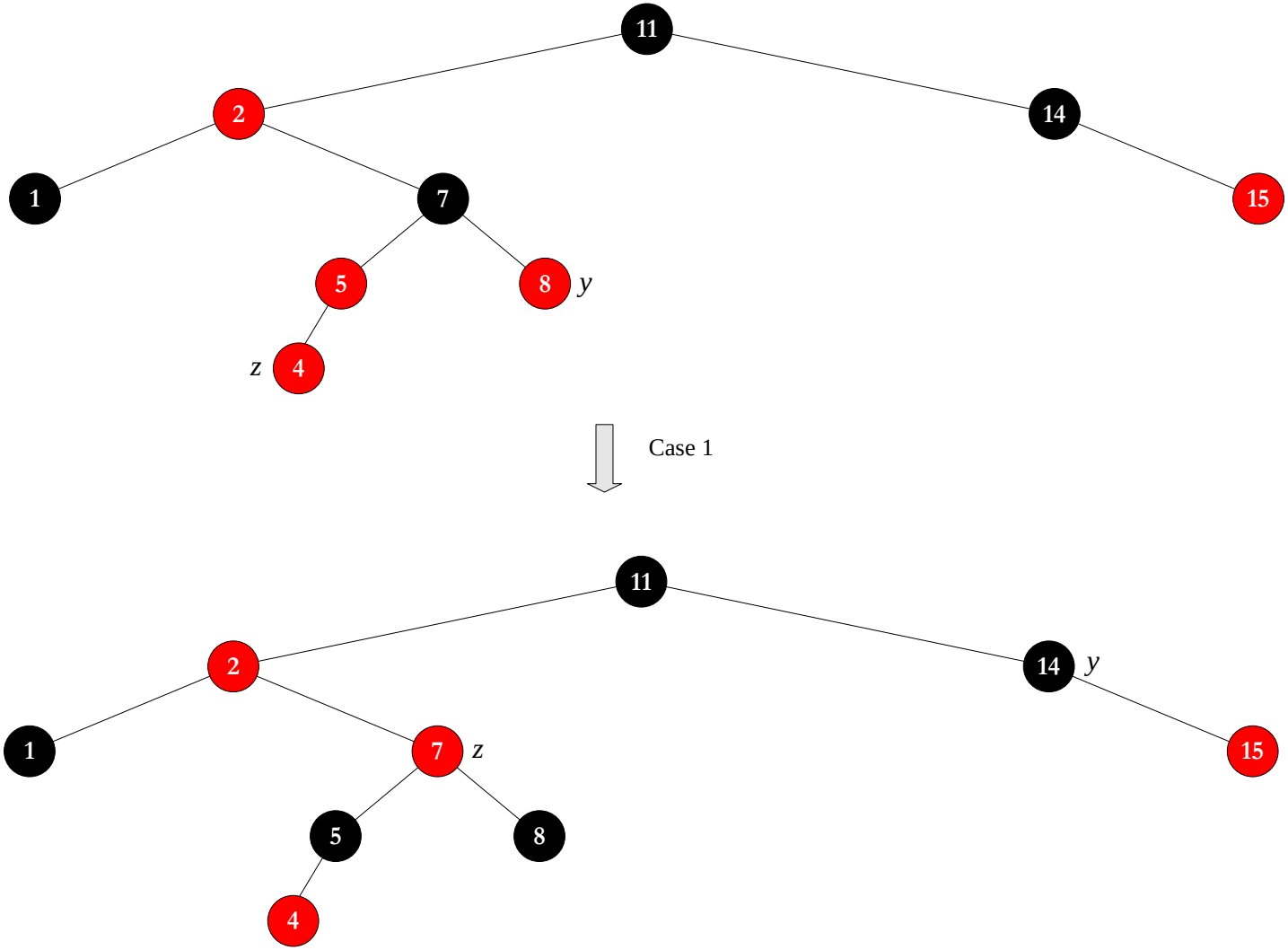
The operation of RB-INSERT-FIXUP case 1

Red-black trees



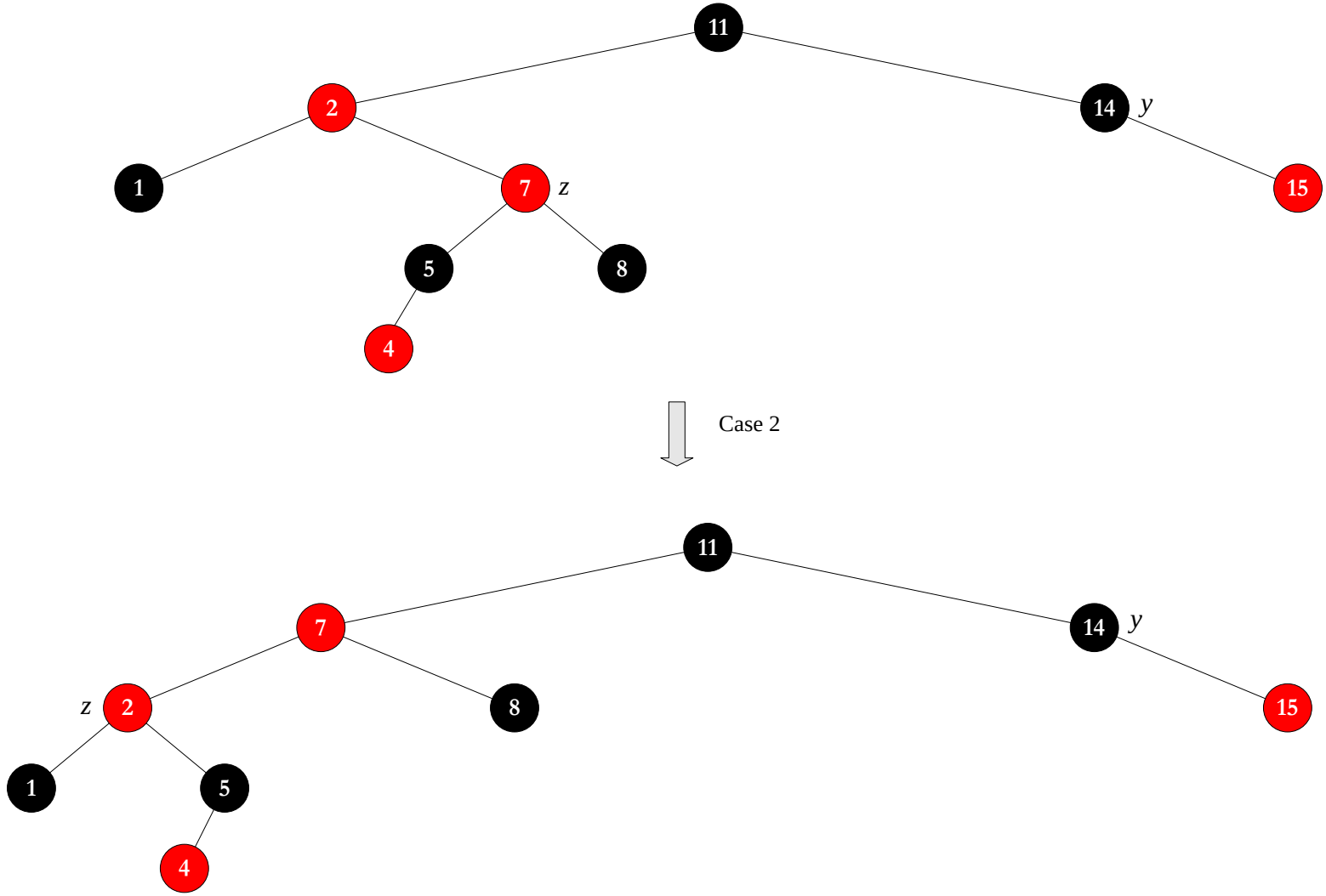
The operation of RB-INSERT-FIXUP case 2, case 3

Red-black trees



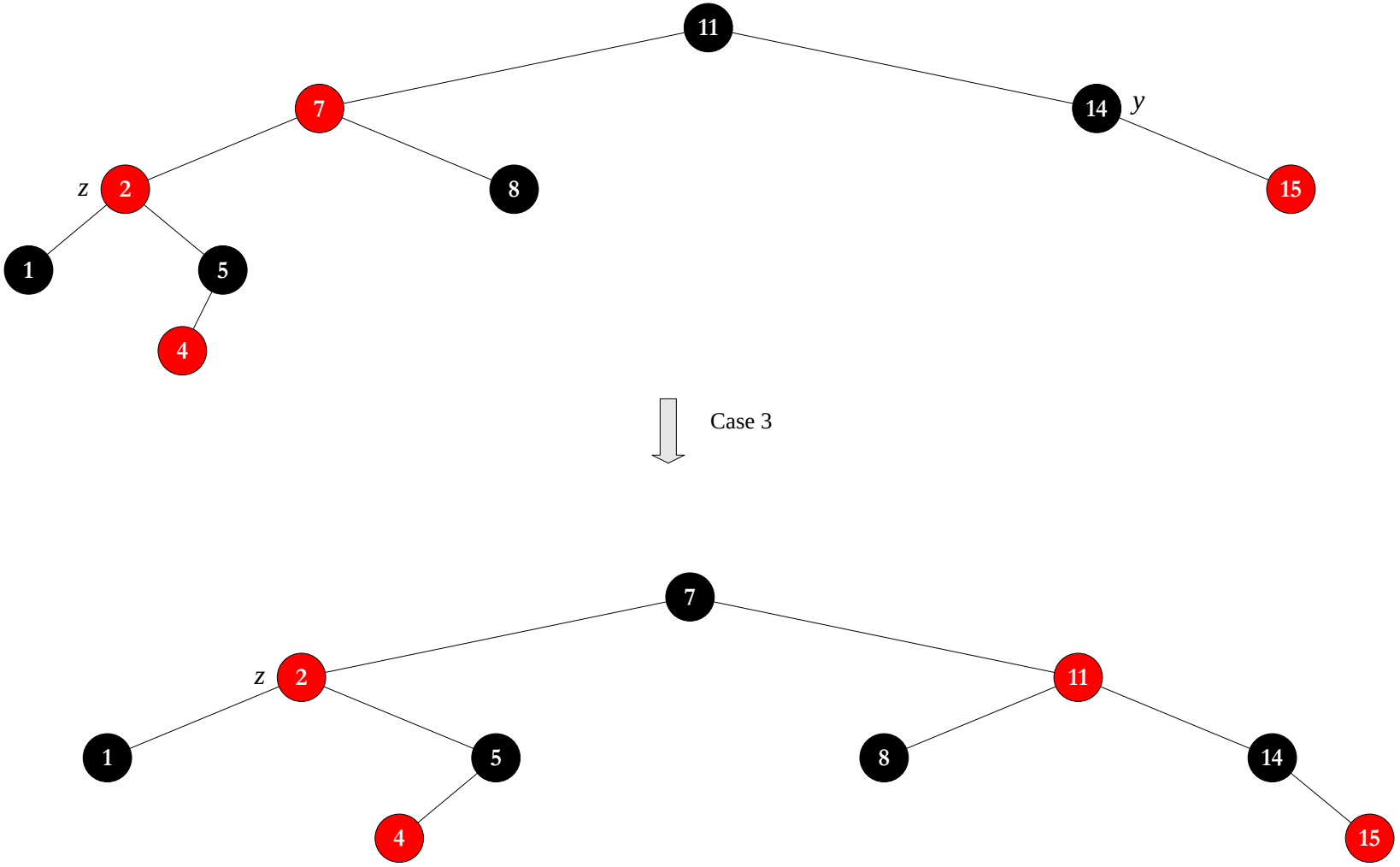
The operation of RB-INSERT-FIXUP

Red-black trees



The operation of RB-INSERT-FIXUP

Red-black trees



The operation of RB-INSERT-FIXUP

Exercises

- In procedure RB-INSERT the inserted node is colored red. If it is colored black then red-black property 4 is satisfied. Why is it not colored black?
- Draw the red-black tree that results after RB-INSERT is called on the bottom tree of the previous slide with key 3. Give the colors of the nodes with letter R or B. Give the black-heights of the nodes too.
- What red-black tree is built with inserting keys 15, 20, 25, 18, 12, 6, 8, 3, 4 into an initially empty tree? What is the black-height of the result tree?

Red-black trees

Remarks:

- The procedure for deleting a node from a red-black tree (RB-DELETE, see textbook) is based on the TREE-DELETE procedure.
- If the deleted node is black then the violated red-black properties have to be restored. This procedure (RB-DELETE-FIXUP, see textbook) runs in $O(\lg n)$ time.

Efficiency: The running time of deleting a node from an n-node red-black tree is $O(\lg n)$.

Augmenting data structures

- The process of augmenting a **basic data structure** to support additional functionality occurs quite frequently in algorithm design.

The steps of the process of augmenting a data structure:

1. Choose an underlying data structure.
2. Determine additional information to maintain in the underlying data structure.
3. Verify that we can maintain the additional information for the basic modifying operations on the underlying data structure.
4. Develop new operations.

Examples:

- Augmenting red-black trees to support general order statistic operations on a dynamic set.
- Augmenting red-black trees to maintain a dynamic set of time intervals.

Theorem: Let f be an attribute that augments a red-black tree T of n nodes, and suppose that the value of f for each node x depends on only the information in nodes x, $x.left$, and $x.right$, possibly including $x.left.f$ and $x.right.f$. Then, we can maintain the values of f in all nodes of T during insertion and deletion without asymptotically affecting the $O(\lg n)$ performance of these operations.

Dynamic order statistics

- The *i*th **order statistic** of a set of *n* elements is the *i*th ($1 \leq i \leq n$) smallest element.
- This element can be found in expected $O(n)$ time from an unordered set (see RANDOMIZED-SELECT procedure).
- With augmenting red-black trees it can be found in $O(\lg n)$ time. The **rank** of a given element in the total ordering of the set also can be calculated in $O(\lg n)$ time.

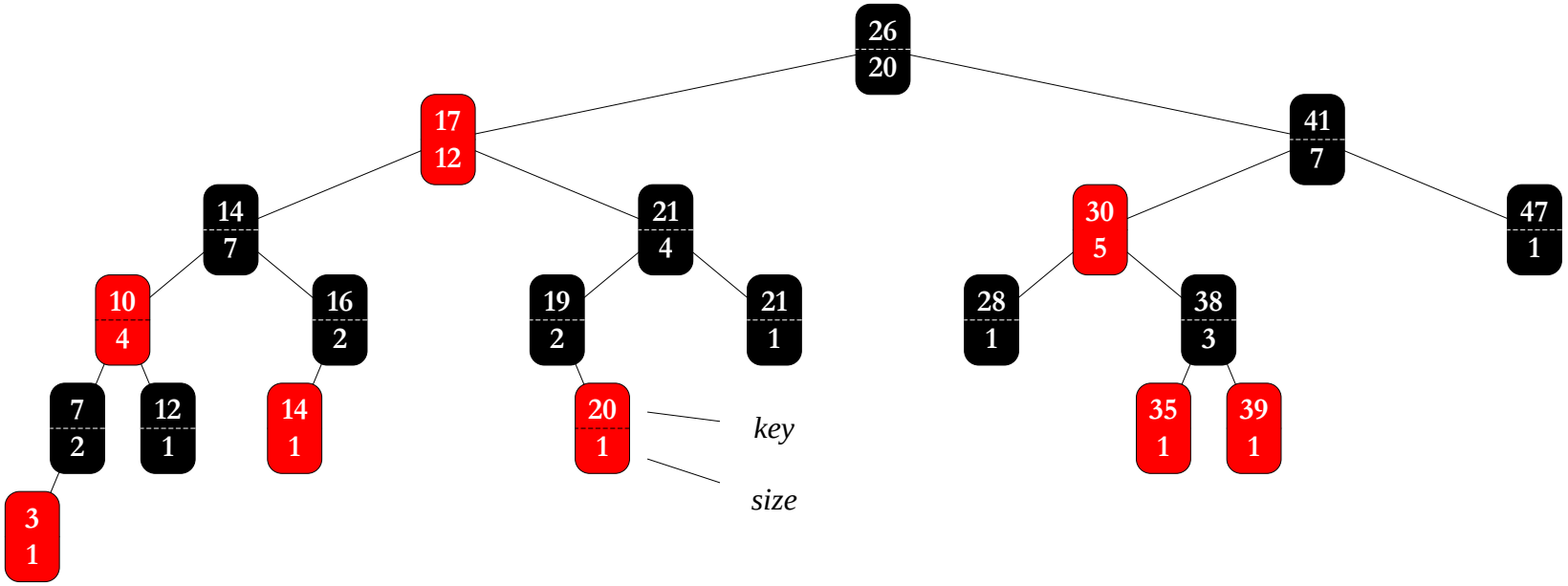
An **order-statistic tree** *T* is simply a red-black tree with additional information stored in each node. Besides the usual red-black tree attributes in a node *x* (*x.key*, *x.color*, *x.p*, *x.left*, and *x.right*), we have another attribute, *x.size*.

This attribute contains the number of (internal) nodes in the subtree rooted at *x* (including *x* itself), that is, the size of the subtree.

If we define the sentinel's size to be 0 (that is, we set *T.nil.size* to be 0) then we have the identity *x.size* = *x.left.size* + *x.right.size* +1.

Remarks: Same keys are possible, so the rank of a node give the position of the inorder tree walk. For example (on the next slide's tree) the rank of the black node with 14 key is 5, and it is 6 for the red color one.

Dynamic order statistics



An order-statistic tree

Dynamic order statistics

```
OS-SELECT(x, i)
1   r = x.left.size+1
2   if i == r
3           return x
4   else
5           if i < r
6                   return OS-SELECT(x.left, i)
7   else
8                   return OS-SELECT(x.right, i−r)
```

Remarks: To find the node with the *i*th smallest key in an order-statistic tree *T*, we call OS-SELECT(*T.root*, *i*).

Efficiency: The running time is $O(\lg n)$ for an n-node order-statistic tree.

Exercises

- What is the result of OS-SELECT(*T.root*, 7) call for the order-statistic tree *T* located in the previous example?
- What nodes are examined by OS-SELECT(*T.root*, 16) call for the same tree?

Dynamic order statistics

```
OS-RANK(T, x)
1   r = x.left.size+1
2   y = x
3   while y ≠ T.root
4         if y == y.p.right
5                   r = r+ y.p.left.size+1
6         y = y.p
7   return r
```

Given a pointer to a node x in an order-statistic tree T, the procedure OS-RANK returns the position of x in the linear order determined by an inorder tree walk of T.

Efficiency: The running time is $O(\lg n)$ for an n-node order-statistic tree.

Exercises

- What is the result of OS-RANK function with the order-statistic tree *T* located in the previous example and the node contains key 19?
- What nodes are examined by OS-RANK function for the same tree and the node contains key 35?

Dynamic order statistics

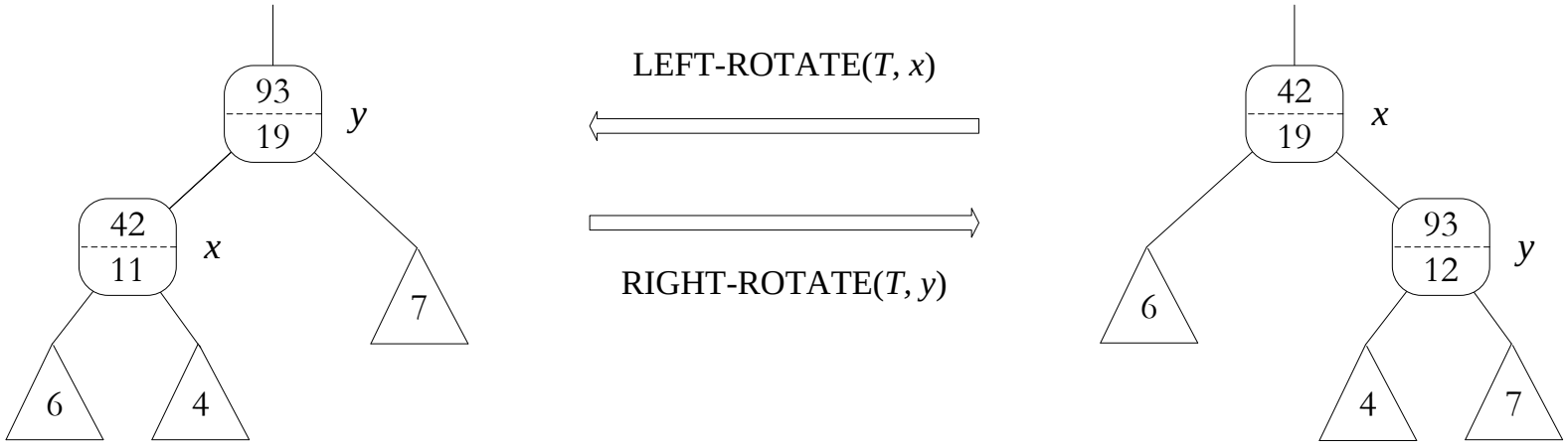
To maintain subtree sizes in procedure LEFT-ROTATE the following lines have to be add :

14 *y.size* = *x.size*

15 *x.size* = *x.left.size*+ *x.right.size*+1

The change to RIGHT-ROTATE is symmetric.

Efficiency: The updates are local, requiring only the *size* information stored in *x*, *y*, and the roots of the subtrees shown as triangles, thus both insertion and deletion take $O(\lg n)$ time for an n-node order-statistic tree.



Updating subtree sizes during rotations

Interval trees

- This example augments red-black trees to support operations on dynamic sets of intervals.

A **closed interval** is an ordered pair of real numbers $[t_1, t_2]$, with $t_1 \leq t_2$. The interval $[t_1, t_2]$ represents the set $\{t \in R: t_1 \leq t \leq t_2\}$.

Open and **half-open** intervals omit both or one of the endpoints from the set, respectively.

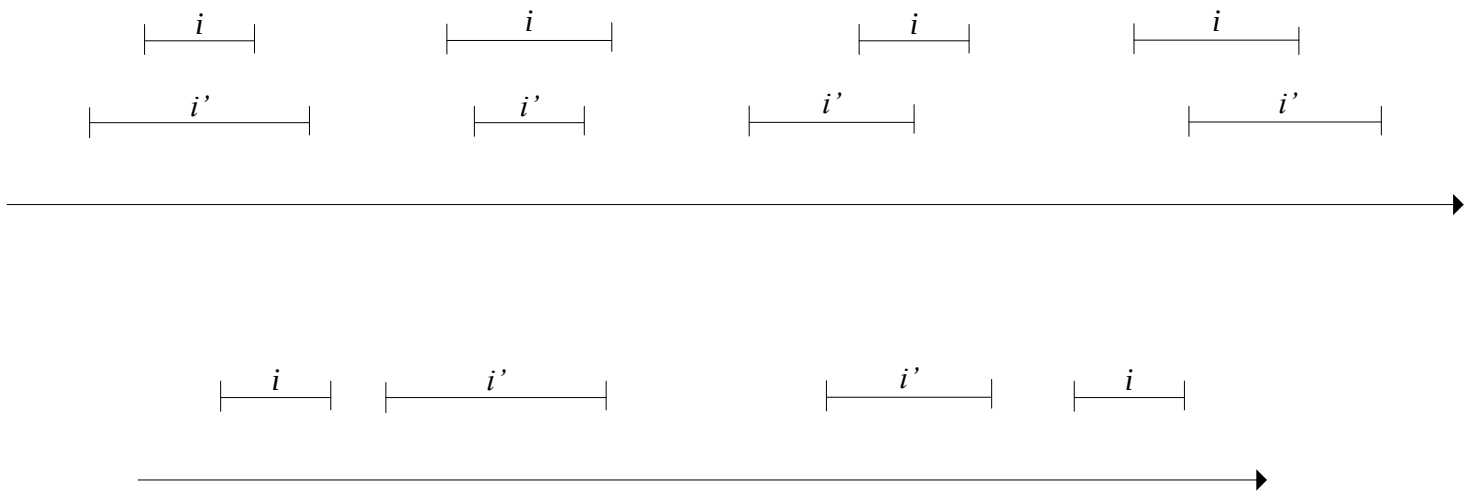
An interval $[t_1, t_2]$ can be represented as an object i, with attributes $i.low = t_1$ (the **low endpoint**) and $i.high = t_2$ (the **high endpoint**).

The intervals i and i' **overlap** if $i \cap i' \neq \emptyset$, that is, $i.low \leq i'.high$ and $i'.low \leq i.high$.

Interval trees

Any two intervals *i* and *i'* satisfy the **interval trichotomy**, that is, exactly one of the following three properties holds:

- *i* and *i'* overlap,
- *i* is to the left of *i'* (i.e. *i.high* < *i'.low*),
- *i* is to the right of *i'* (i.e. *i'.high* < *i.low*).



Interval trichotomy

Interval trees

An **interval tree** is a red-black tree that maintains a dynamic set of elements, with each element x containing an interval $x.int$.

Interval trees support the following operations:

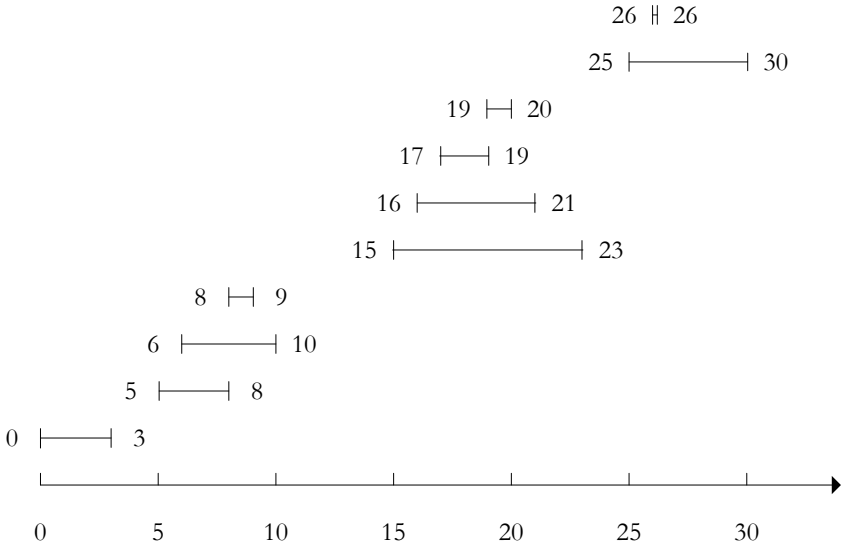
- INTERVAL-INSERT(T, x) adds the element x, whose *int* attribute is assumed to contain an interval, to the interval tree T.
- INTERVAL-DELETE(T, x) removes the element x from the interval tree T.
- INTERVAL-SEARCH(T, i) returns a pointer to an element x in the interval tree T such that $x.int$ overlaps interval i, or a pointer to the sentinel $T.nil$ if no such element is in the set.

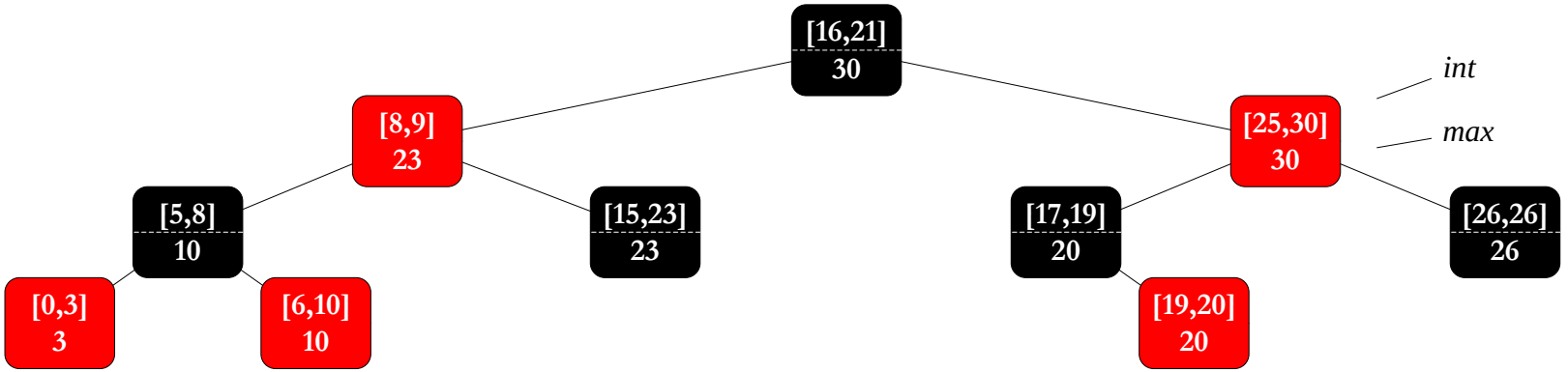
Interval trees

The steps of the process of augmenting a data structure:

1. **Underlying data structure**: it is a red-black tree in which each node x contains an interval $x.int$ and the key of x is the low endpoint ($x.int.low$) of the interval. Thus, an inorder tree walk of the data structure lists the intervals in sorted order by low endpoint.
2. **Additional information**: each node x contains a value $x.max$, which is the maximum value of any interval endpoint stored in the subtree rooted at x.
3. **Maintaining the information**: $x.max$ can be determined given interval $x.int$ and the max values of node x's children as $x.max = \max(x.int.high, x.left.max, x.right.max)$. The max attributes can be update after a rotation in $O(1)$ time, thus insertion and deletion run in $O(\lg n)$ time.
4. **Developing new operations**: the only new operation is INTERVAL-SEARCH(T, i).

Interval trees





An interval tree

Interval trees

```
INTERVAL-SEARCH(T, i)
1   x = T.root
2   while x ≠ T.nil and i does not overlap x.int
3         if x.left ≠ T.nil and x.left.max ≥ i.low
4               x = x.left
5         else
6               x = x.right
7   return x
```

Efficiency: The running time is $O(\lg n)$ for an n element dynamic set.
Remarks: It is enough to examine only one path from the root.

Exercises

- What is the result of INTERVAL-SEARCH function with the interval tree T located in the previous example and the interval i=[4, 7]?
- What nodes are examined by INTERVAL-SEARCH function for the same tree and the interval i=[11, 14]?