Red-Black Tree

Manoj Kumar
DTU, Delhi

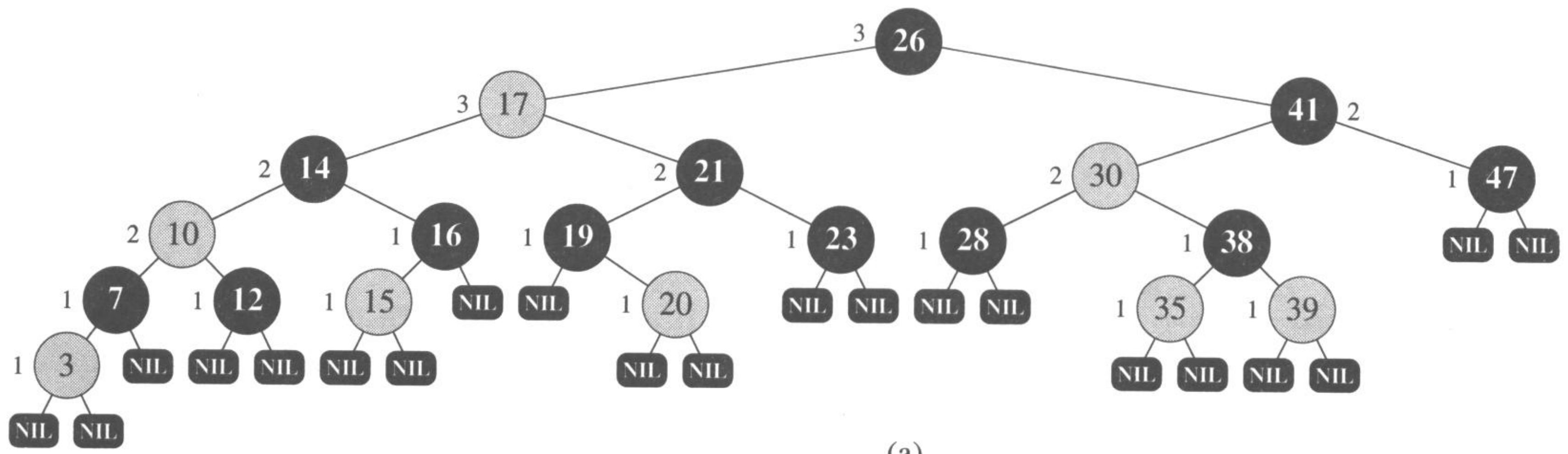
Red Black Tree

- Recall binary search tree
  - Key values in the left subtree <= the node value
  - Key values in the right subtree >= the node value
- Operations:
  - insertion, deletion
  - Search, maximum, minimum, successor, predecessor.
  - O(h), h is the height of the tree.

Red Black Tree

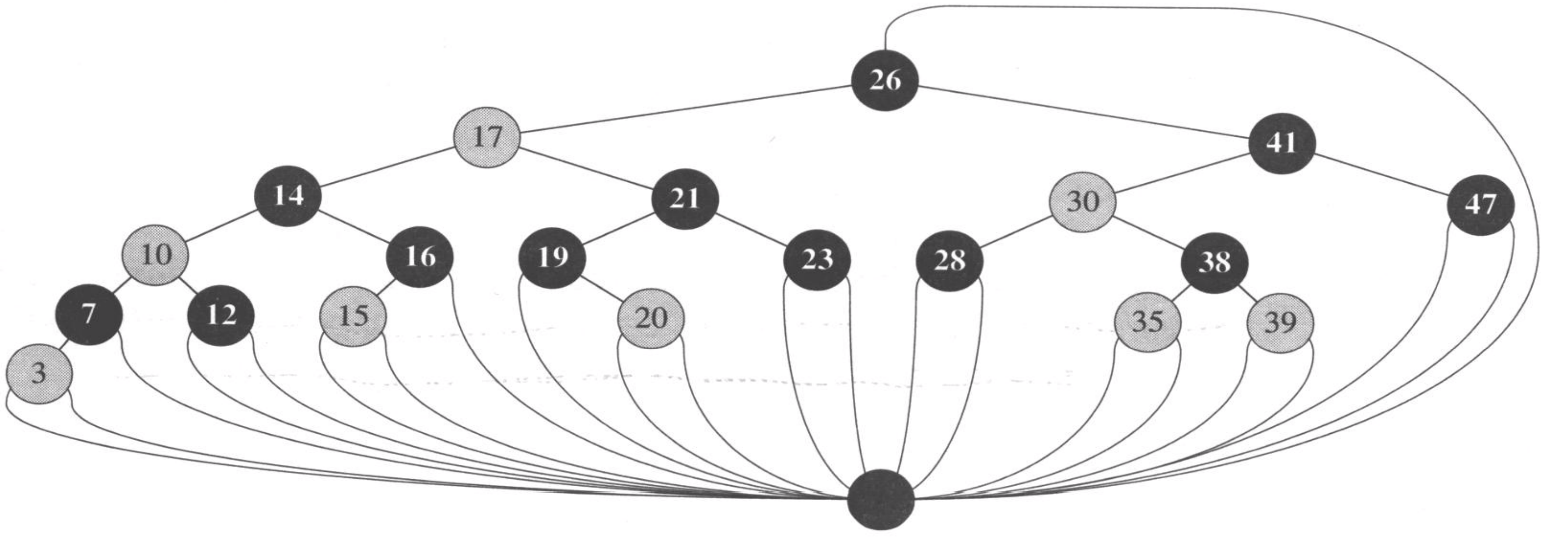
- Definition: a binary tree, satisfying:
  1. Every node is red or black
  2. The root is black
  3. Every leaf is NIL and is black
  4. If a node is red, then both its children are black
  5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.
- Purpose: keep the tree balanced.
- Other balanced search tree:
  - AVL tree, 2-3-4 tree, Splay tree.

Red Black Tree



(a)

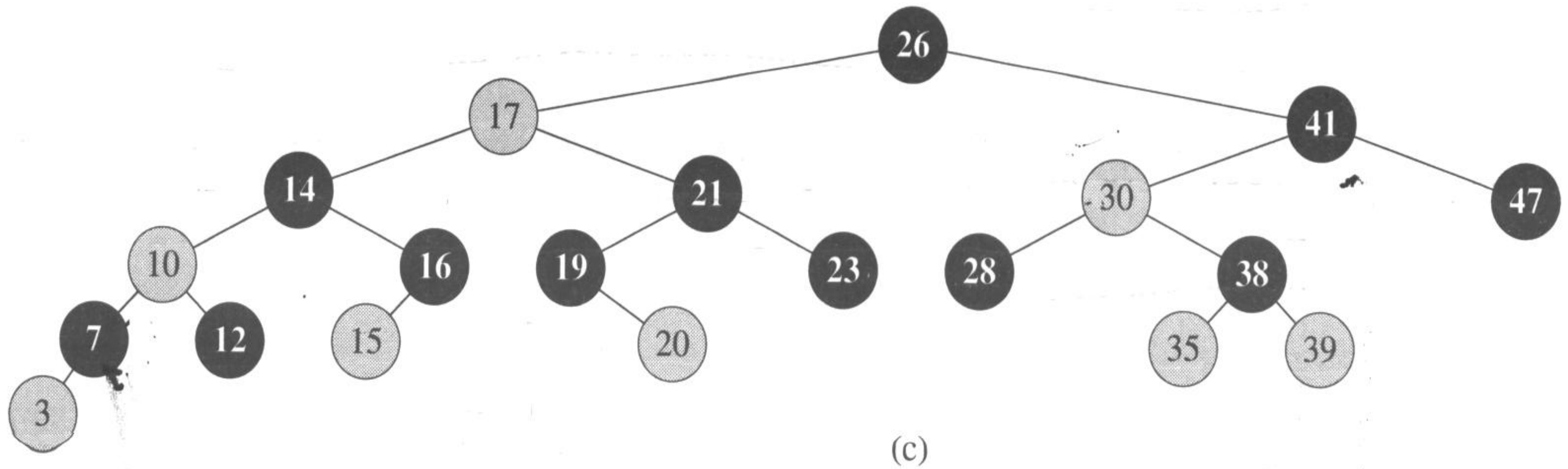
All NIL nodes replaced by single sentinel node *nil[T]*



nil[*T*]

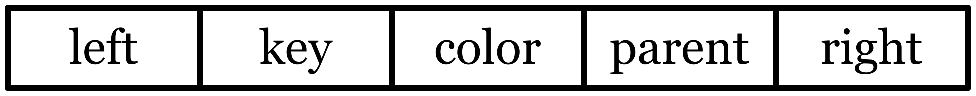
(b)

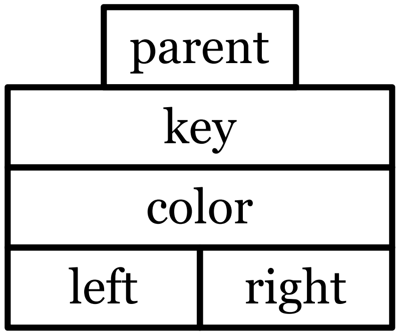
Same red Black tree without NIL nodes



(c)

Node structure





Properties

- **Black height:**-*bh(x)*, black-height of *x*, the number of black nodes on any path from *x* (excluding *x*) to a leaf.
- A red-black tree with n internal nodes has height at most 2log(n+1).
  - Note: **Internal nodes**: all normal key-bearing nodes.
    External nodes: Nil nodes or the Nil Sentinel.
  - A subtree rooted at x contains at least $2^{bh(x)}-1$ internal nodes.
  - By property 4, bh(root)≥h/2.
  - $n \geq 2^{h/2}-1$

Some operations in log(n)

- Search, minimum, maximum, successor, predecessor.
- Let us discuss insert or delete.

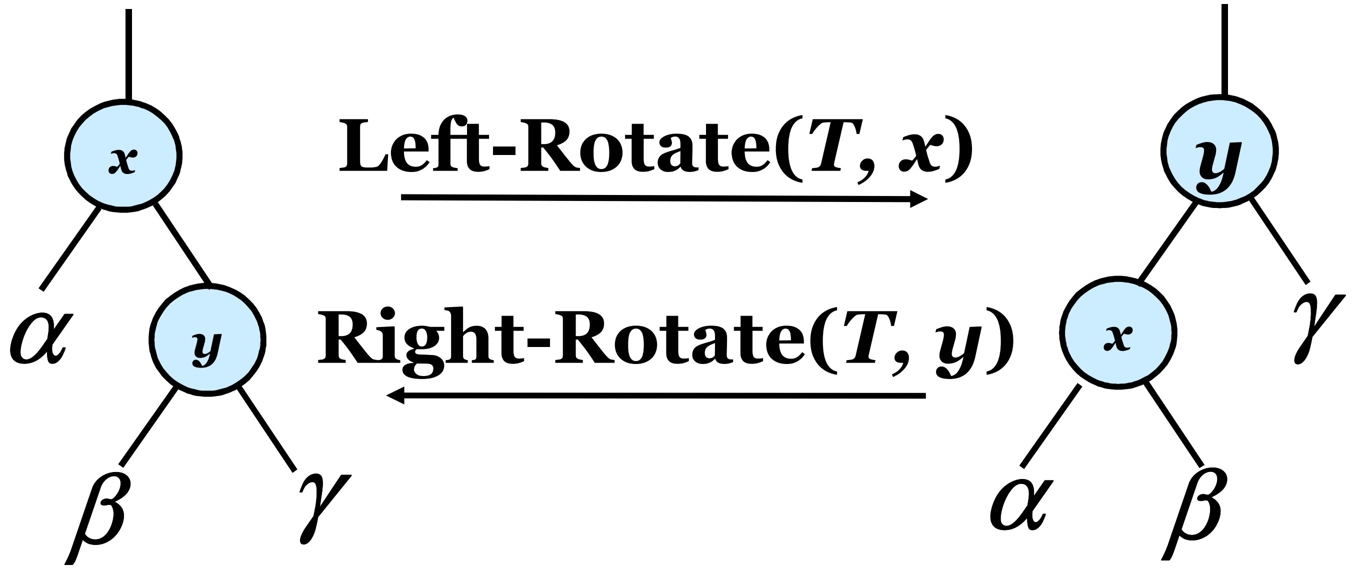
Rotations

.

- Algorithms to restore RBT property to tree after Tree-Insert and Tree-Delete include right and left rotations and re-coloring nodes.
- The number of rotations for insert and delete are constant, but they may take place at every level of the tree, so therefore the running time of insert and delete is O(lg(n))

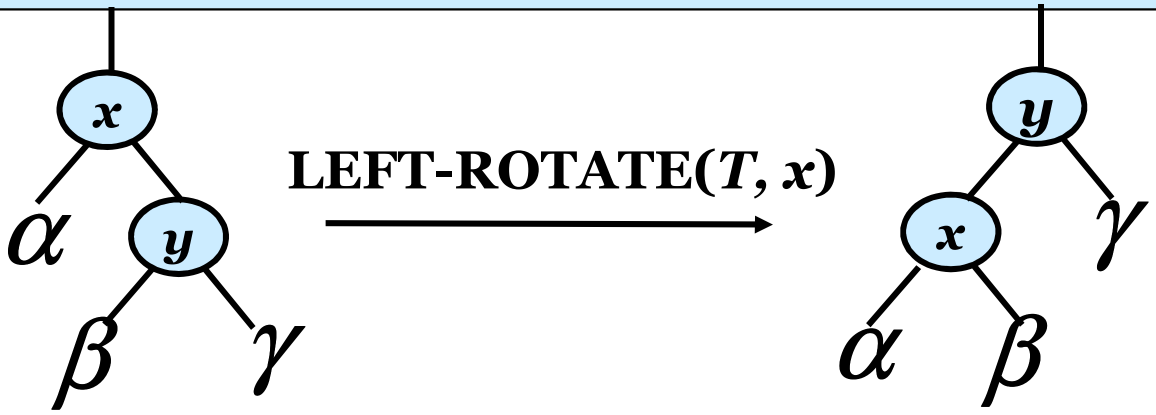
Rotations

- Rotations are the basic tree-restructuring operation for almost all *balanced* search trees.
- Rotation takes a red-black-tree and a node,
- Changes pointers to change the local structure, and
- Won't violate the binary-search-tree property.
- Left rotation and right rotation are inverses.



Left Rotation

```
LEFT-ROTATE(T, x)
1.  y ← right[x]                //set y
2.  right[x] ← left[y]          //turn y's left subtree into x's right subtree
3.  if (left[y]!=nil)
4.     then p[left[y]] ← x      // change parent of β if β is not NIL
5.  p[y] ← p[x]                 //link x's parent to y.
6.  if (p[x]==nil)              // if x was root, make y root node
7.     then root[T] ← y
8.     else if (x ==left[p[x]]) // link y to parent of x
9.             then left[p[x]] ← y
10.            else right[p[x]] ← y
11. left[y] ← x                 // put x on y's left
12. p[x] ← y
```

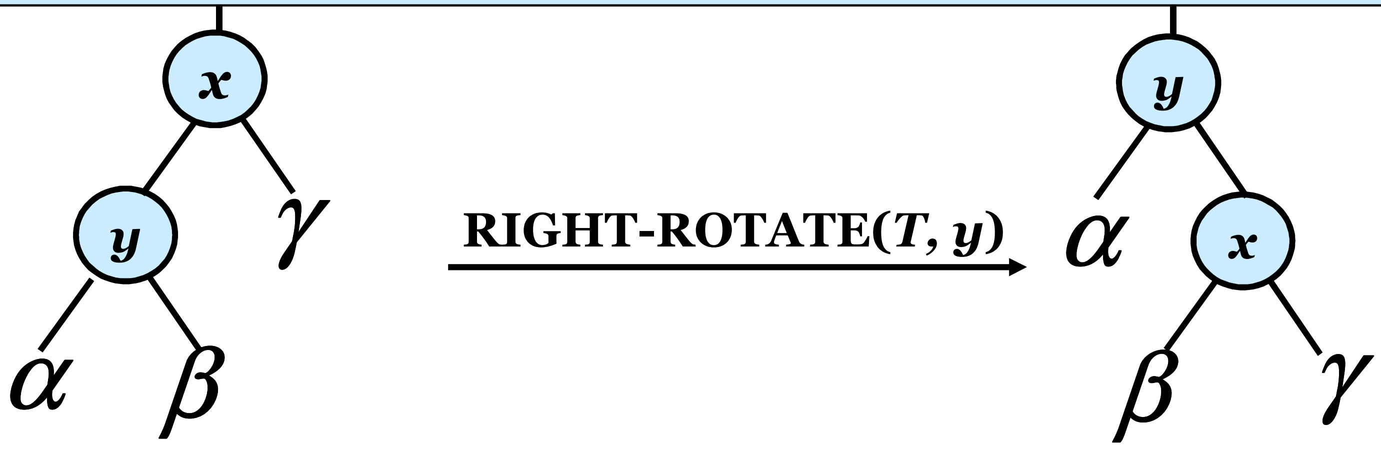


Rotations

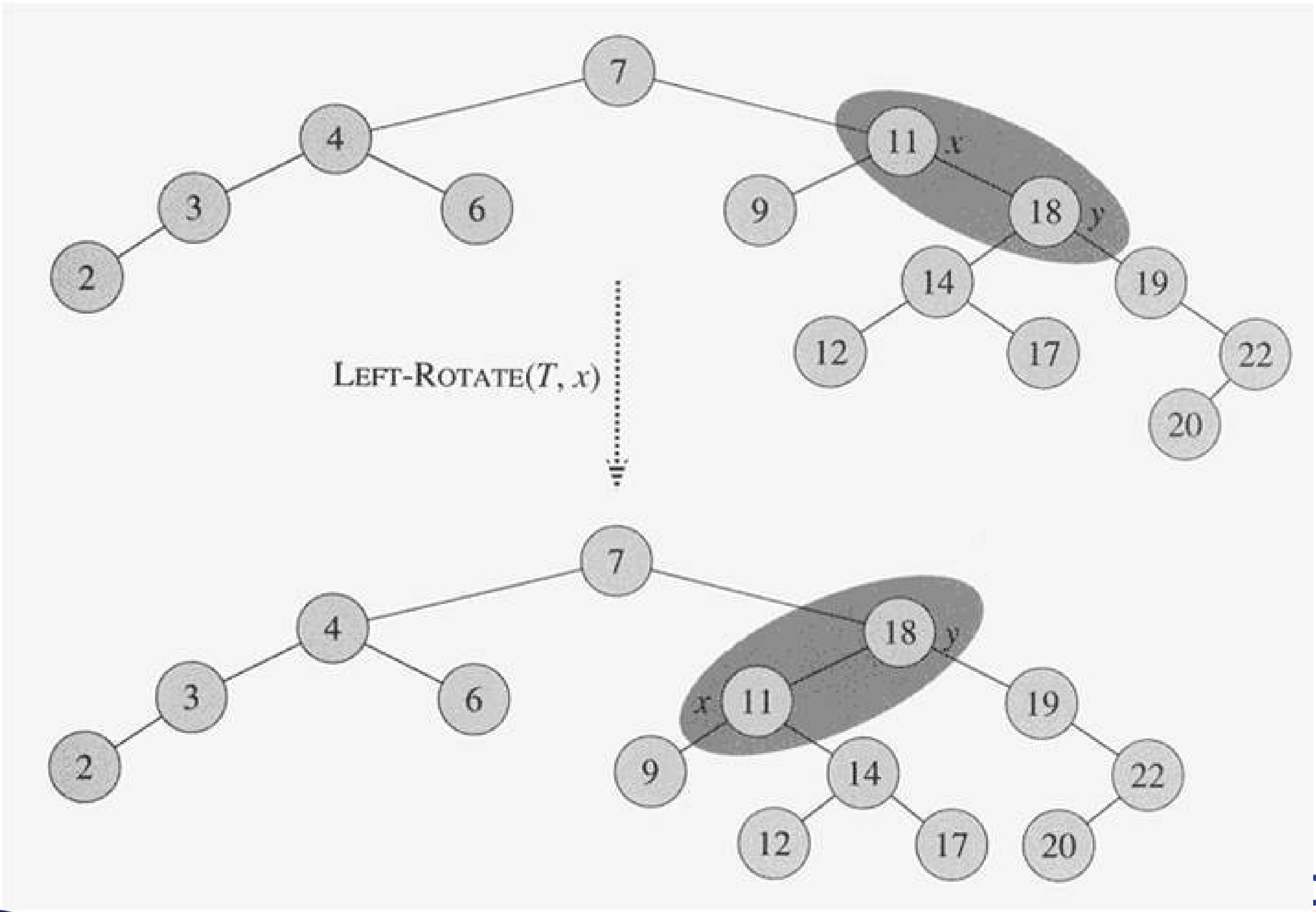
- The pseudo-code for Left-Rotate assumes that
  - *right*[*x*] ≠ *nil*[*T*], and
  - root's parent is *nil*[*T*].
- Left Rotation on *x*, makes *x* the left child of *y*, and the left subtree of *y* into the right subtree of *x*.
- Pseudocode for Right-Rotate is symmetric: exchange *left* and *right* everywhere.
- ***Time:*** $O(1)$ for both Left-Rotate and Right-Rotate, since a constant number of pointers are modified.

Right Rotation

```
RIGHT-ROTATE($T$, $x$)
1.  $y \leftarrow left[x]$                    //set $y$
2.  $left[x] \leftarrow right[y]$             //turn $y$'s right subtree into $x$'s left subtree
3.  if ($right[y] != nil$)
4.      then $p[right[y]] \leftarrow x$       // change parent of $\beta$ if $\beta$ is not NIL
5.  $p[y] \leftarrow p[x]$                    //link $y$ to parent of $x$.
6.  if ($p[y] == nil$)                        // if $x$ was root, make $y$ root node
7.      then $root[T] \leftarrow x$
8.      else if ($x == left[p[x]]$)           // link $x$ to parent of $y$
9.              then $left[p[x]] \leftarrow y$
10.             else $right[p[x]] \leftarrow y$
11. $right[y] \leftarrow x$                   // put $x$ on $y$'s right
12. $p[x] \leftarrow y$
```



Left Rotation: Example



Reminder: 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 paths from the node to descendant leaves contain the same number of black nodes.

Insertion

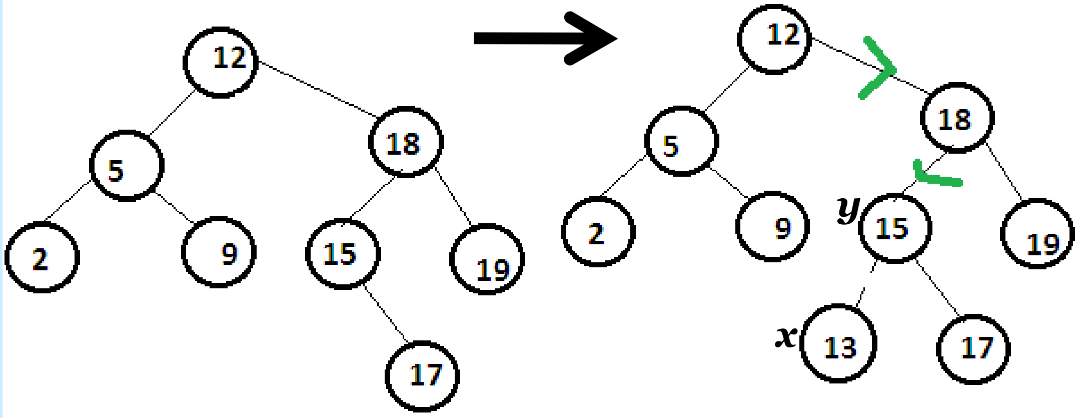
- Insertion must preserve all red-black properties.
- Should an inserted node be colored Red? Black?
- Basic steps:
    - Use TREE-INSERT(T, x) from BST (slightly modified) to insert a node *x* into *T*.
    - Color the node *x* red.
    - Fix the modified tree by re-coloring nodes and performing rotation to preserve RB tree property.
        - Procedure **RB-INSERT-FIXUP(T, x)**.

RB-INSERT()

```
RB-INSERT($T$, $x$)
1.  TREE-INSERT($T$, $x$)
2.  RB-INSERT-FIXUP($T$, $x$)
```

TREE-INSERT()

```
TREE-INSERT(T, x)
1.  y ← NIL
2.  z ← root[T]
3.  while (z != nil[T])
4.       do  y ← z                          //set y to a node where x may be inserted
5.            if (key[x] < key[z])
6.                then  z ← left[z]
7.                else  z ← right[z]
8.  p[x] ← y                                // insert x as child of y
9.  if (y==NIL)
10.     then  root[T] ← x
11.      else  if (key[x] < key[y])
12.               then  left[y] ← x
13.               else  right[y] ← x
14. left[x] ← nil[T]
15. right[x] ← nil[T]
16. color[x] ← RED
```

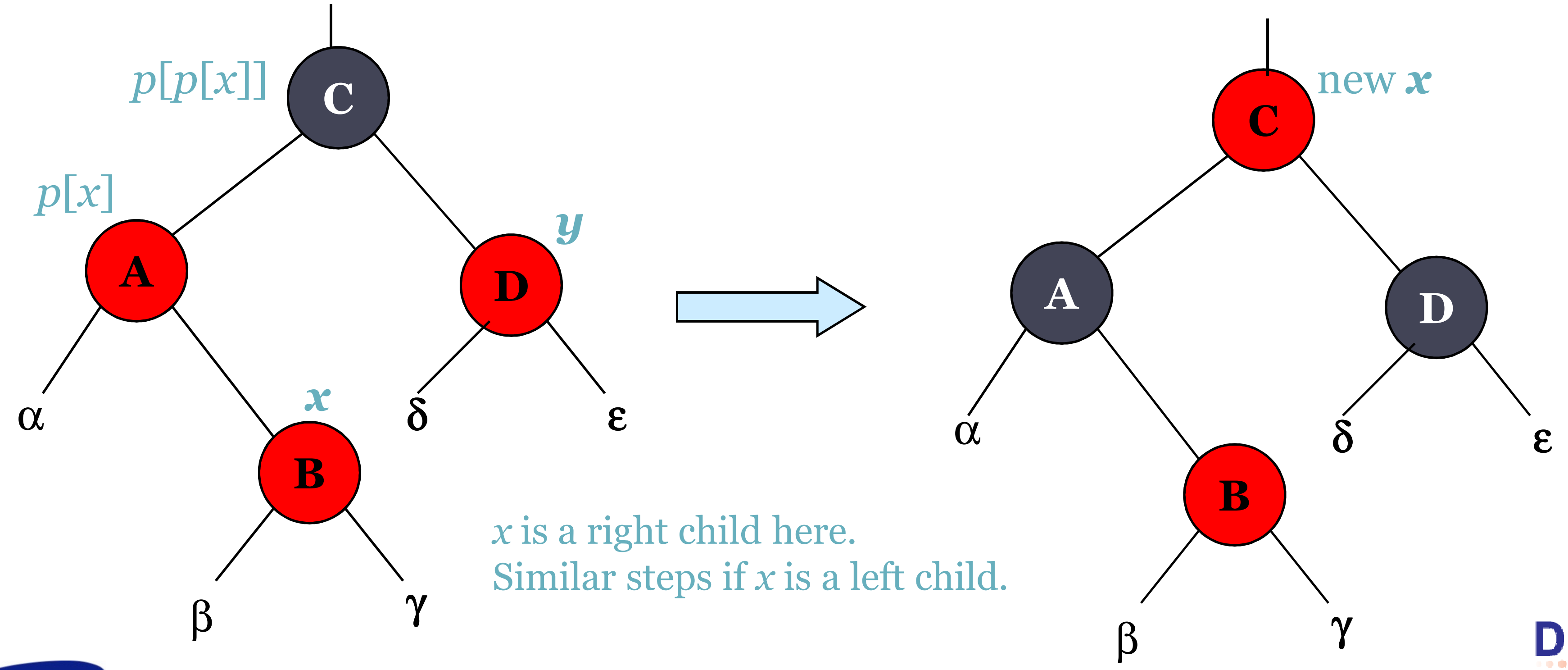


RB-INSERT-FIXUP()

- Problem: we may have one pair of consecutive reds where we did the insertion.
- Solution: rotate it up the tree and away…
  Three cases have to be handled…

Case 1: uncle *y* is red

- *p*[*p*[*x*]] (*x*'s grandparent) must be black, since *x* and *p*[*x*] are both red and there are no other violations of property 4.
- Make *p*[*x*] and *y* black ⇒ now *x* and *p*[*x*] are not both red. But property 5 might now be violated.
- Make *p*[*p*[*x*]] red ⇒ restores property 5.
- The next iteration has *p*[*p*[*x*]] as the new *x* (i.e., *x* moves up 2 levels).



x is a right child here.
Similar steps if *x* is a left child.

RB-INSERT-FIXUP(): Case 1

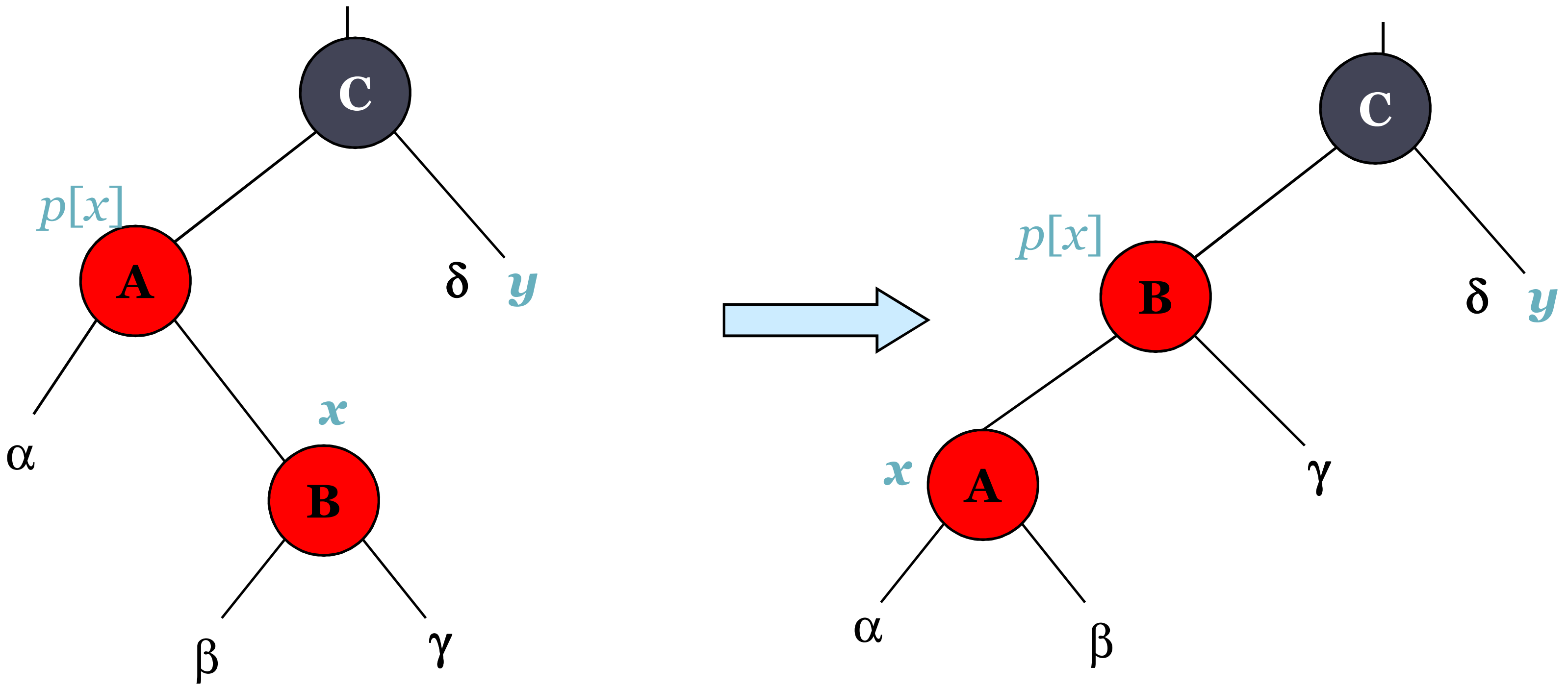
```
RB-Insert-Fixup (T, x)
1.  while (x != root[T] and color[p[x]] == RED )
2.      do if  (p[x] == left[p[p[x]]])
3.          then y ← right[p[p[x]]]
4.              if (color[y] == RED)
5.                  then color[p[x]] ← BLACK  // Case 1
6.                       color[y] ← BLACK       // Case 1
7.                       color[p[p[x]]] ← RED   // Case 1
8.                       x ← p[p[x]]            // Case 1
```

RB-INSERT-FIXUP(): Case 2 & 3

```
RB-INSERT-FIXUP(T, x) (Contd.)
9.              else if x == right[p[x]]              // color[y] ≠ RED
10.                 then x ← p[x]                     // Case 2
11.                     LEFT-ROTATE(T, x)             // Case 2
12.                 color[p[x]] ← BLACK               // Case 3
13.                 color[p[p[x]]] ← RED              // Case 3
14.                 RIGHT-ROTATE(T, p[p[x]])          // Case 3
15.       else if p[x] == right[p[p[x]]])
               then (same as 3-14 with "right" and "left" exchanged)
16. color[root[T ]] ← BLACK
```

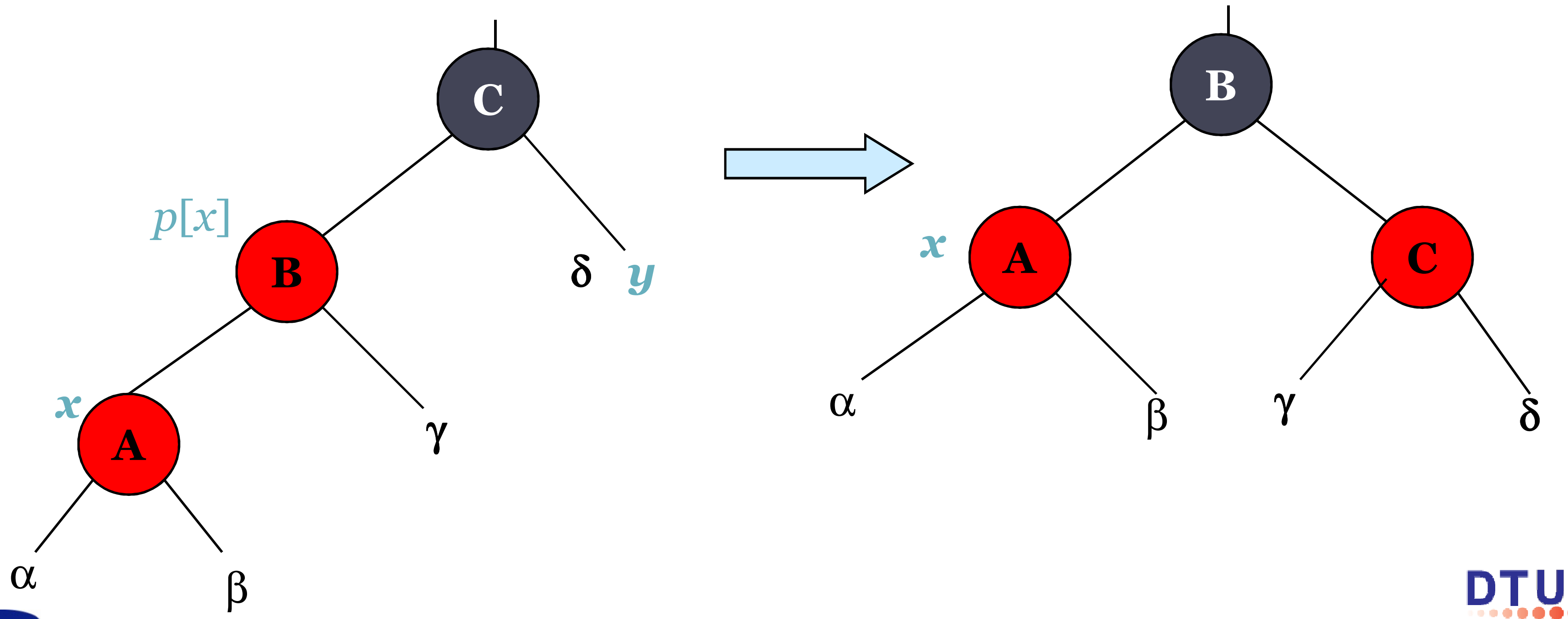
Case 2 – *y* is black, *x* is a right child

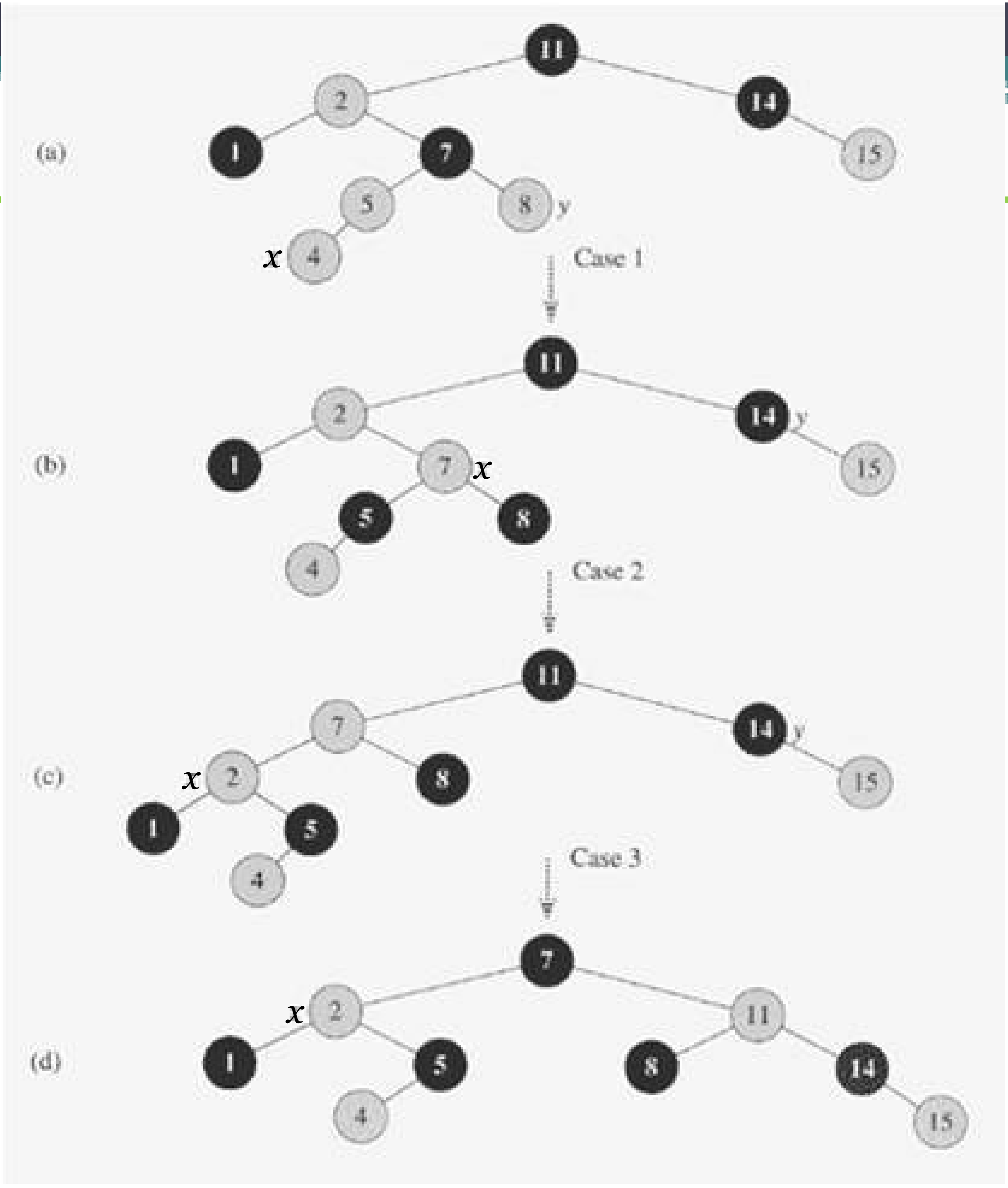
- Left rotate around *p*[*x*], *p*[*x*] and *x* switch roles ⇒ now x is a left child, and both *x* and *p*[*x*] are red.
- Takes us immediately to case 3.



Case 3 – *y* is black, *x* is a left child

- Make *p*[*x*] black and *p*[*p*[*x*]] red.
- Then right rotate on *p*[*p*[*x*]]. Ensures property 4 is maintained.
- No longer have 2 reds in a row.
- *p*[*x*] is now black ⇒ no more iterations.



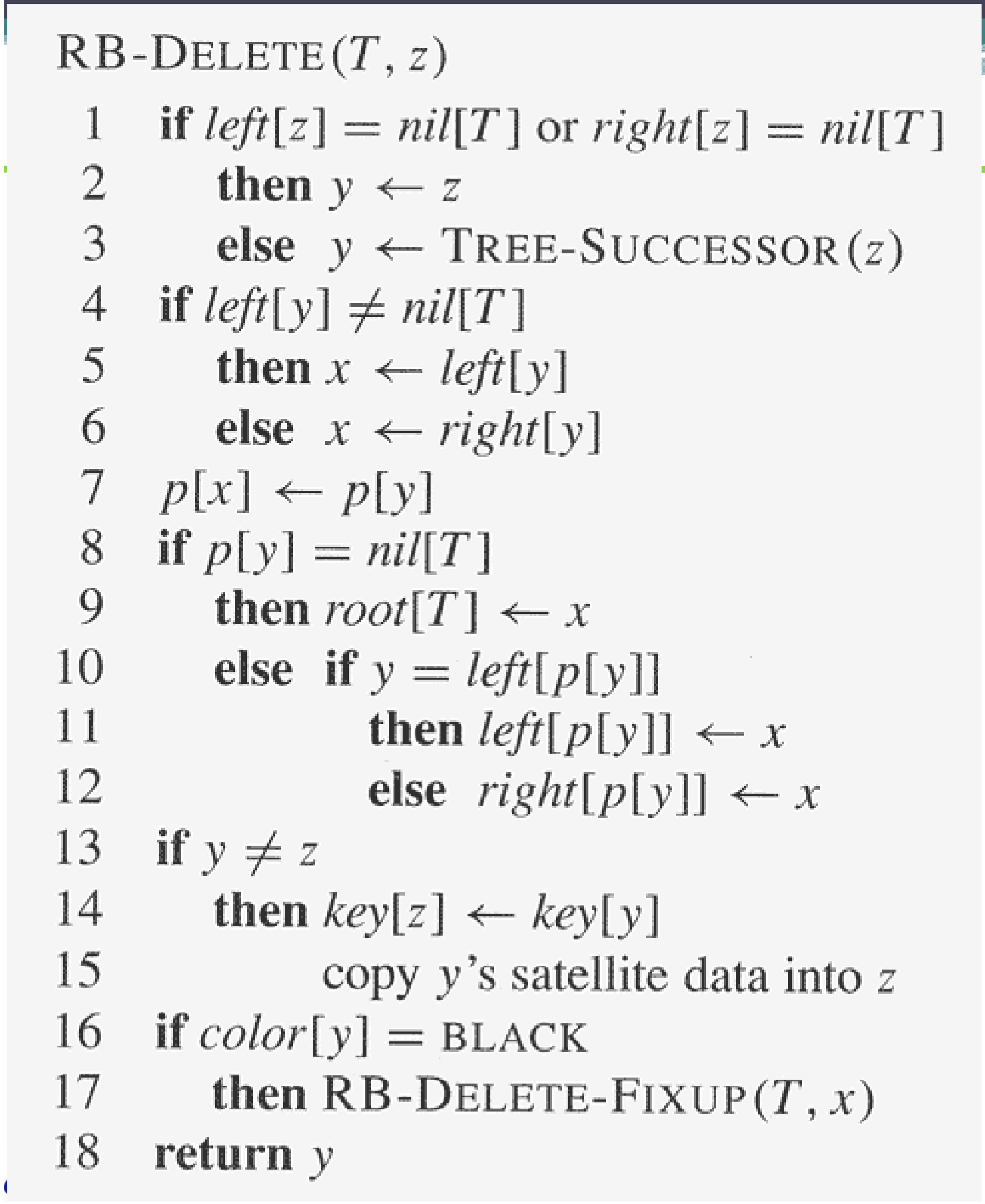

(a)
11
2
14
1
7
15
5
8
y
x
4
Case 1
(b)
11
2
14
y
1
7
x
15
5
8
4
Case 2
(c)
11
7
14
y
x
2
8
15
1
5
4
Case 3
(d)
7
x
2
11
1
5
8
14
4
15

Algorithm Analysis

- $O(\lg n)$ time to get through RB-INSERT up to the call of RB-INSERT-FIXUP.
- Within RB-INSERT-FIXUP:
  - Each iteration takes $O(1)$ time.
  - Each iteration but the last moves x up 2 levels.
  - $O(\lg n)$ levels $\Rightarrow$ $O(\lg n)$ time.
  - Thus, insertion in a red-black tree takes $O(\lg n)$ time.
  - Note: there are at most 2 rotations overall.

Deletion

- Deletion, like insertion, should preserve all the RB properties.
- The properties that may be violated depends on the color of the deleted node.
  - Red – OK. Why?
  - Black?
- Steps:
  - Do regular BST deletion.
  - Fix any violations of RB properties that may result.



```
RB-DELETE$(T, z)$
if $left[z] = nil[T]$ or $right[z] = nil[T]$
    then $y \leftarrow z$
    else $y \leftarrow$ TREE-SUCCESSOR$(z)$
if $left[y] \neq nil[T]$
    then $x \leftarrow left[y]$
    else $x \leftarrow right[y]$
$p[x] \leftarrow p[y]$
if $p[y] = nil[T]$
    then $root[T] \leftarrow x$
    else if $y = left[p[y]]$
            then $left[p[y]] \leftarrow x$
            else $right[p[y]] \leftarrow x$
if $y \neq z$
    then $key[z] \leftarrow key[y]$
         copy $y$'s satellite data into $z$
if $color[y] =$ BLACK
    then RB-DELETE-FIXUP$(T, x)$
return $y$
```

If z contains single child

If z contains both children

x points to child of y

Connect x to parent of y

make x left or right child of parent of y

Replace z with y

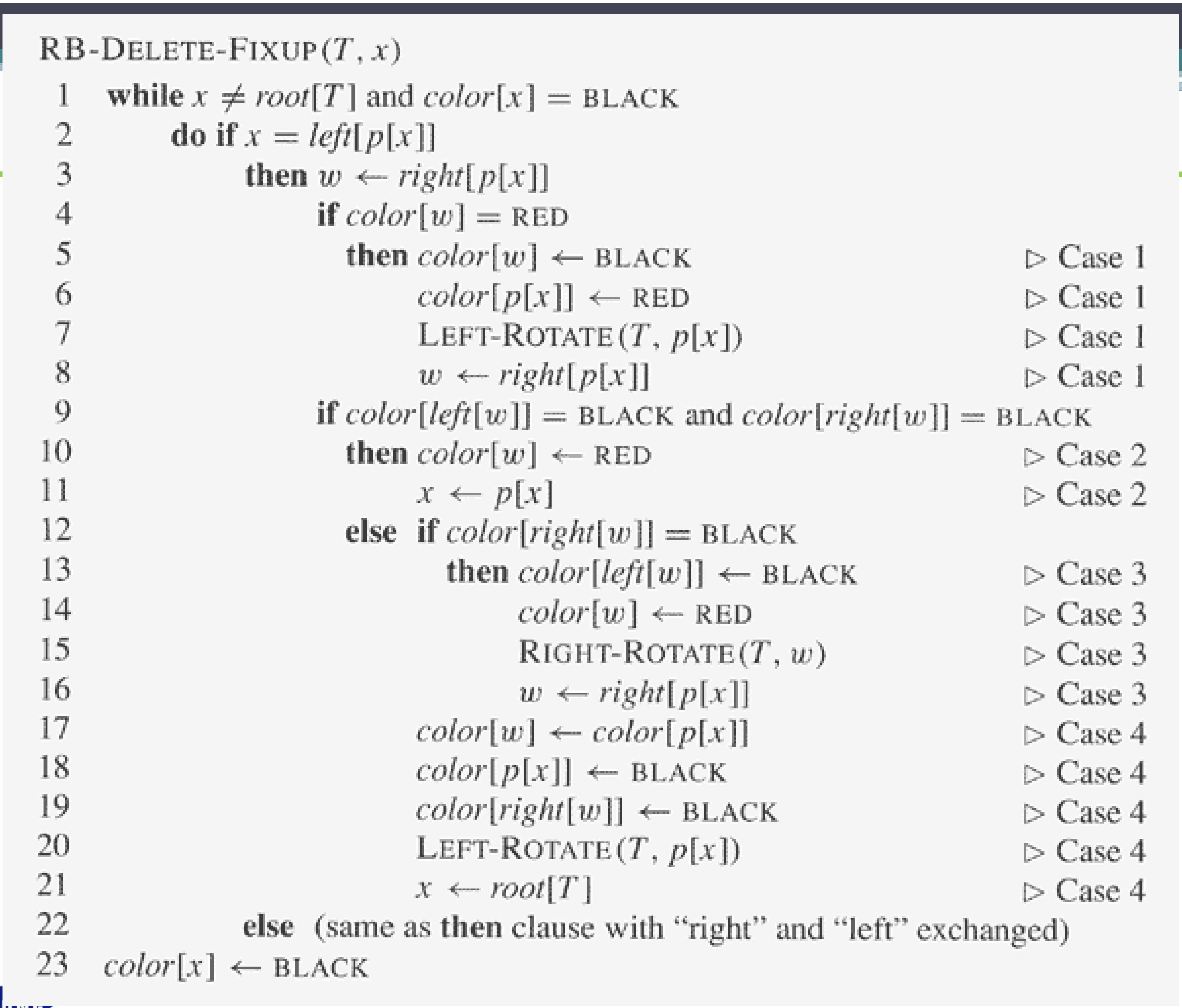
If a BLACK node is deleted, call Fixup on x

RB Properties Violation

- If *y* is black, we could have violations of red-black properties:
  - Prop. 1. OK.
  - Prop. 2. If *y* is the root and *x* is red, then the root has become red.
  - Prop. 3. OK.
  - Prop. 4. Violation if *p*[*y*] and *x* are both red.
  - Prop. 5. Any path containing *y* now has 1 fewer black node.

RB Properties Violation

- Prop. 5. Any path containing *y* now has 1 fewer black node.
  - Correct by giving *x* an "extra black."
  - Add 1 to count of black nodes on paths containing *x*.
  - Now property 5 is OK, but property 1 is not.
  - *x* is either ***doubly black*** (if *color*[*x*] = BLACK) or ***red & black*** (if *color*[*x*] = RED).
  - The attribute *color*[*x*] is still either RED or BLACK. No new values for *color* attribute.
  - In other words, the extra blackness on a node is by virtue of *x* pointing to the node.
- Remove the violations by calling RB-Delete-Fixup.

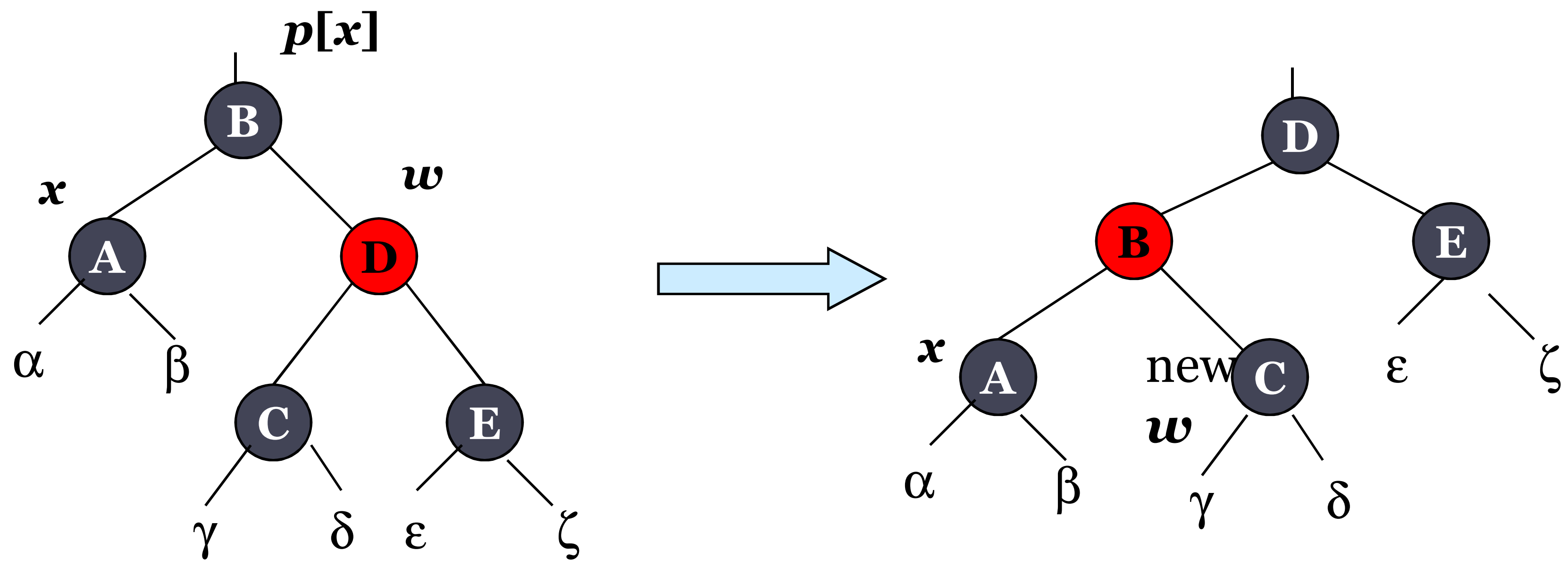


```
RB-DELETE-FIXUP$(T, x)$
while $x \neq root[T]$ and $color[x] = $ BLACK
    do if $x = left[p[x]]$
        then $w \leftarrow right[p[x]]$
            if $color[w] = $ RED
              then $color[w] \leftarrow$ BLACK                          ▷ Case 1
                   $color[p[x]] \leftarrow$ RED                         ▷ Case 1
                   LEFT-ROTATE$(T, p[x])$                               ▷ Case 1
                   $w \leftarrow right[p[x]]$                           ▷ Case 1
            if $color[left[w]] = $ BLACK and $color[right[w]] = $ BLACK
              then $color[w] \leftarrow$ RED                            ▷ Case 2
                   $x \leftarrow p[x]$                                  ▷ Case 2
              else if $color[right[w]] = $ BLACK
                       then $color[left[w]] \leftarrow$ BLACK           ▷ Case 3
                            $color[w] \leftarrow$ RED                   ▷ Case 3
                            RIGHT-ROTATE$(T, w)$                        ▷ Case 3
                            $w \leftarrow right[p[x]]$                  ▷ Case 3
                   $color[w] \leftarrow color[p[x]]$                    ▷ Case 4
                   $color[p[x]] \leftarrow$ BLACK                       ▷ Case 4
                   $color[right[w]] \leftarrow$ BLACK                   ▷ Case 4
                   LEFT-ROTATE$(T, p[x])$                               ▷ Case 4
                   $x \leftarrow root[T]$                               ▷ Case 4
        else (same as then clause with "right" and "left" exchanged)
$color[x] \leftarrow$ BLACK
```

Deletion – Fixup

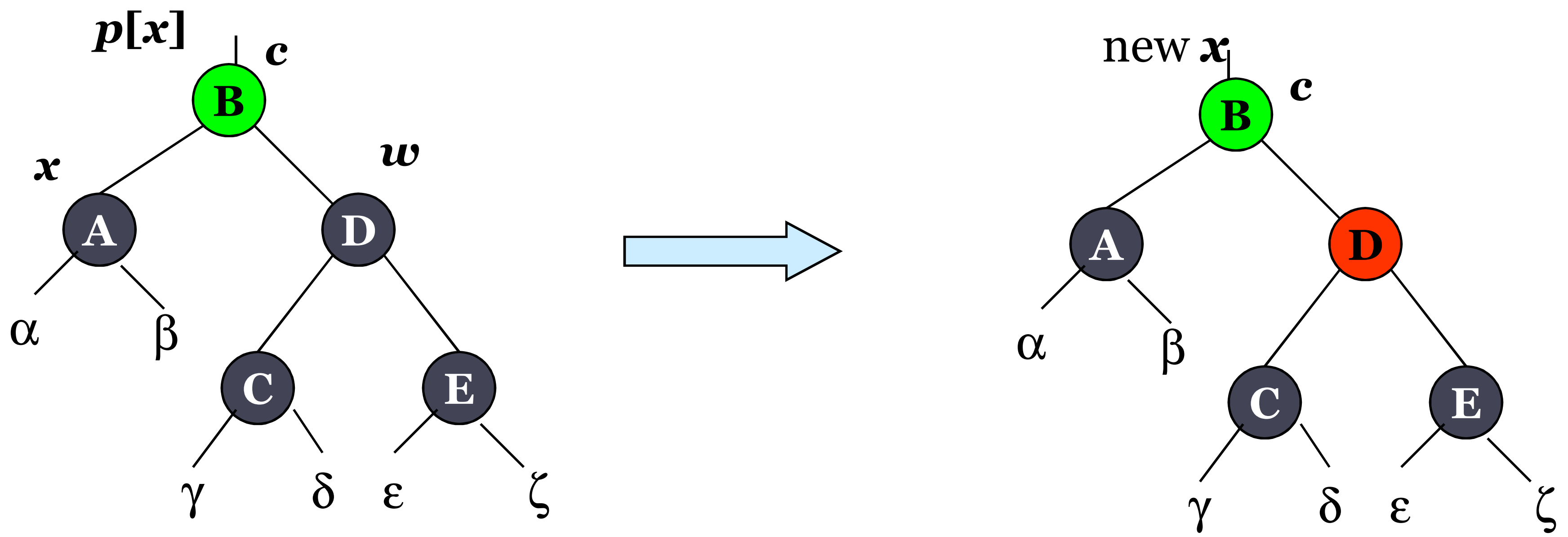
- ***Idea:*** Move the extra black up the tree until x points to a red & black node $\Rightarrow$ turn it into a black node,
- x points to the root $\Rightarrow$ just remove the extra black, or
- We can do certain rotations and recoloring and finish.
- Within the **while** loop:
  - x always points to a non root doubly black node.
  - w is x's sibling.
  - w cannot be $nil[T]$, since that would violate property 5 at $p[x]$.
- 8 cases in all, 4 of which are symmetric to the other.

Case 1 – *w* is red



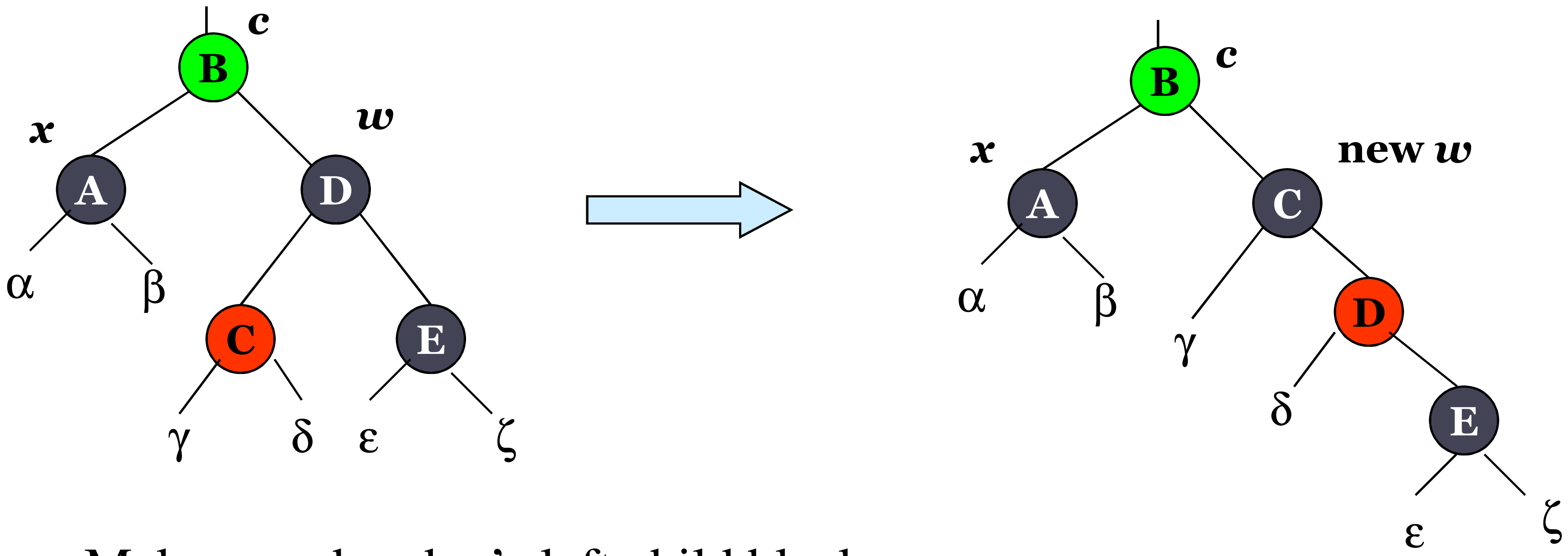
- *w* must have black children.
- Make *w* black and *p*[*x*] red (because *w* is red *p*[*x*] couldn't have been red).
- Then left rotate on *p*[*x*].
- New sibling of *x* was a child of *w* before rotation ⇒ must be black.
- Go immediately to case 2, 3, or 4.

Case 2 – *w* is black, both *w*'s children are black



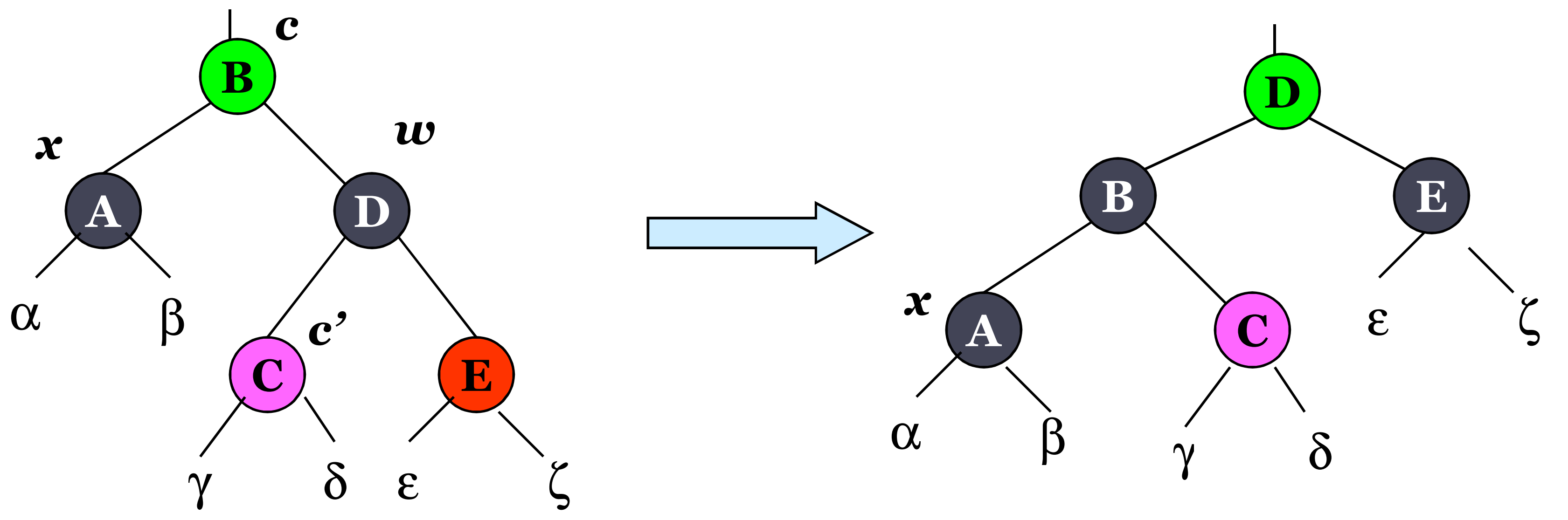
- Take 1 black off *x* (⇒ singly black) and off *w* (⇒ red).
- Move that black to *p*[*x*].
- Do the next iteration with *p*[*x*] as the new *x*.
- If entered this case from case 1, then *p*[*x*] was red ⇒ new *x* is red & black ⇒ color attribute of new *x* is RED ⇒ loop terminates. Then new *x* is made black in the last line.

Case 3 – *w* is black, *w*'s left child is red, *w*'s right child is black



- Make *w* red and *w*'s left child black.
- Then right rotate on *w*.
- New sibling *w* of *x* is black with a red right child $\Rightarrow$ case 4.

Case 4 – *w* is black, *w*'s right child is red



- Make *w* be *p*[*x*]'s color (*c*).
- Make *p*[*x*] black and *w*'s right child black.
- Then left rotate on *p*[*x*].
- Remove the extra black on *x* (⇒ *x* is now singly black) without violating any red-black properties.
- All done. Setting *x* to root causes the loop to terminate.

Analysis

- $O(\lg n)$ time to get through RB-DELETE up to the call of RB-DELETE-FIXUP.
- Within RB-DELETE-FIXUP:
  - Case 2 is the only case in which more iterations occur.
    - x moves up 1 level.
    - Hence, $O(\lg n)$ iterations.
  - Each of cases 1, 3, and 4 has 1 rotation $\Rightarrow \leq 3$ rotations in all.
  - Hence, $O(\lg n)$ time.

Augmenting Data Structures

Augmenting Data Structures

- The idea of augmenting data structures is fairly simple. We want to add data to the elements of our data structure that help us to quickly get to some type of information.
- Technically you can augment a data structure with whatever information you want, but certain things don't allow for the data structure to be efficiently implemented. In order to be practical, the augmented data must be updatable with the same order as the normal operations of the data structure.

Augmenting BSTs

- For BSTs, we can come up with a simple rule for what types of data we can augment the structure with.
- Any augmentation where the value of a node can be calculated from the values of the children will preserve the speed requirements.
- This holds because only O(log n) nodes are going to have one of their descendants changed.
- Most of the values that we typically talk about in relation to trees can be calculated this way which means we can augment the tree with them.

Dynamic order statistic (*i*th element)

- Red-black tree gives a total order via inorder traversal, i.e., reflecting the rank of an element.
  - Two additional operations:
    - Find *i*th smallest element.
    - Find the rank of an element.
- How to modify it?
- Add a field, *size* in every node, i.e., *size*[x] is the size of the subtree rooted at x, including x.
- So assume sentinel's size *size*[NIL]=0, then, *size*[x]=*size*[left[x]]+*size*[right[x]]+1.
- If so, easy to find the *i*th element, or the rank of an element in log(n) time.

Dynamic order statistic tree

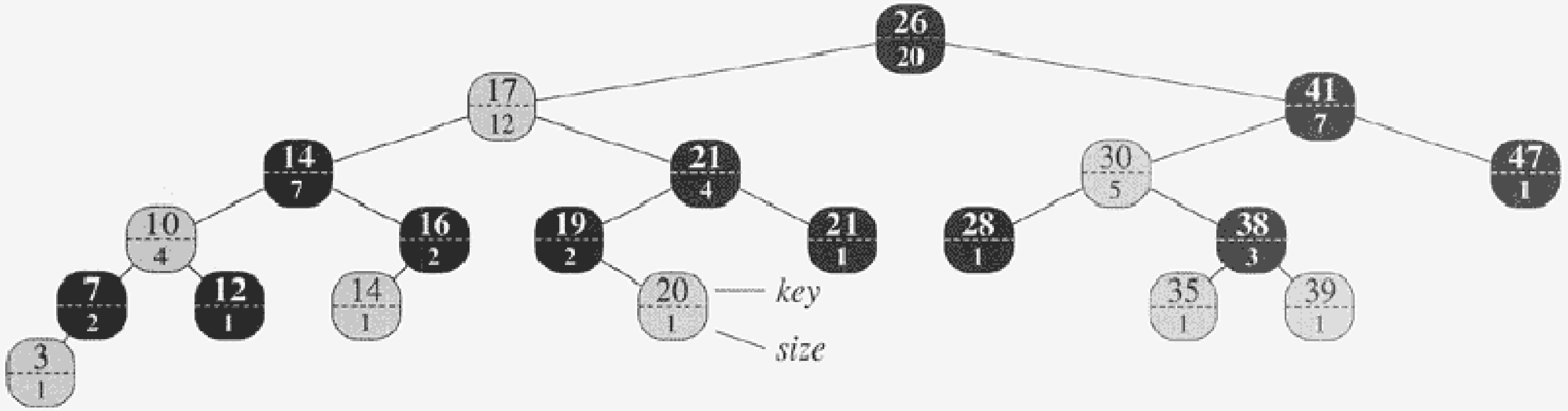


Figure 14.1 An order-statistic tree, which is an augmented red-black tree. Shaded nodes are red, and darkened nodes are black. In addition to its usual fields, each node x has a field $size[x]$, which is the number of nodes in the subtree rooted at x.

Retrieving element with rank i

Find ith smallest element in the tree rooted at x in O(lg n) time

```
OS-SELECT(x,i)
1.  r ← size[left[x]] +1
2.  if  i == r
3.      then  return x
4.  else if  (i <r)
5.          then return OS-SELECT(left[x],i)
6.          else  return OS-SELECT(right[x] , i-r)
```

• Here r is number of keys less than key of x, so rank of x is r+1.

Find rank of an element

Give a pointer to node x in an order-statistic tree T, algorithm returns position of x in linear order determined by an inorder traversal of T.

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

Maintaining subtree sizes

- Insertion: two passes:
  - Insert x into tree, by going down, increase size by 1 for each node visited.
  - Modify the color and rotation by going up.
    - Only the rotation will affect the size of some nodes,
    - Fortunately, local modification.
- Same for deletion operation.

Maintaining subtree sizes

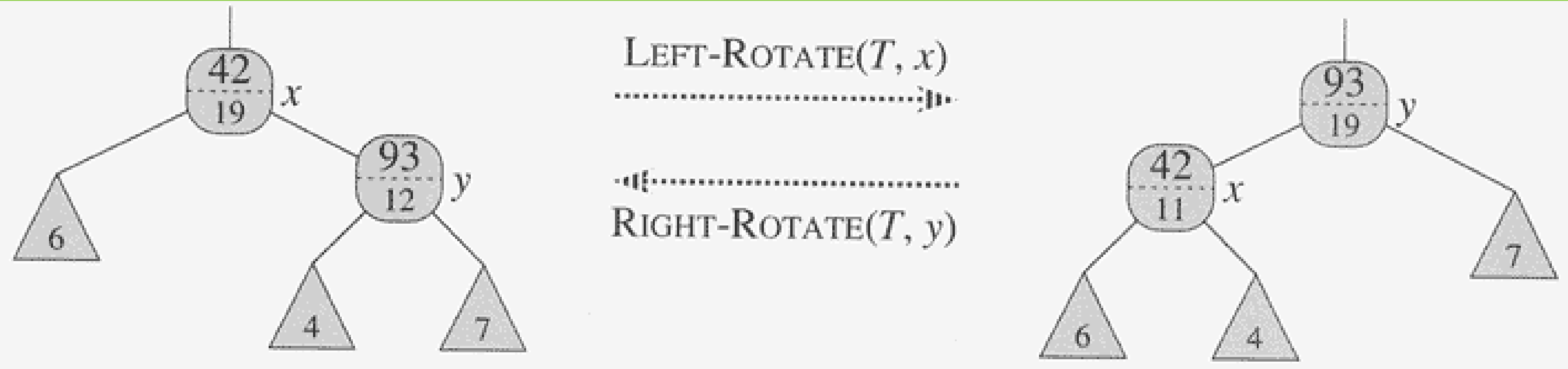


Figure 14.2 Updating subtree sizes during rotations. The link around which the rotation is performed is incident on the two nodes whose *size* fields need to be updated. The updates are local, requiring only the *size* information stored in x, y, and the roots of the subtrees shown as triangles.

Two lines of addition code required in LEFT-ROTATE()

```
        size[y] ← size[x]
        size[x] ← size[left[x]] + size[right[x]] + 1
```

Two lines of addition code required in RIGHT-ROTATE()

```
        size[x] ← size[y]
        size[y] ← size[left[y]] + size[right[y]] +1
```

Interval tree: dynamic set of intervals

Intervals

- Closed intervals $[t_1,t_2]$, with $t_1 \leq t_2$,
- open intervals,
- half-intervals $\leq t_{1,}$ $\geq t_1$
- New operations:
  - INTERVAL-INSERT(T, x), $x=[t_1,t_2]$.
  - INTERVAL-DELETE(T, x), $x=[t_1,t_2]$.
  - INTERVAL-SEARCH(T, i), return a pointer x such that the interval of x overlaps with i.

How to implement?

- Select a underlying DS, red-back tree
  - The node x contains interval int[x], and the low[int[x]] is the node's key.
- Additional information: max
- Maintain the information:
  - max[x]=max(high[int[x]],max[left[x]],max[right[x]]).
- Implementation of INTERVAL-SEARCH.

Interval trichotomy

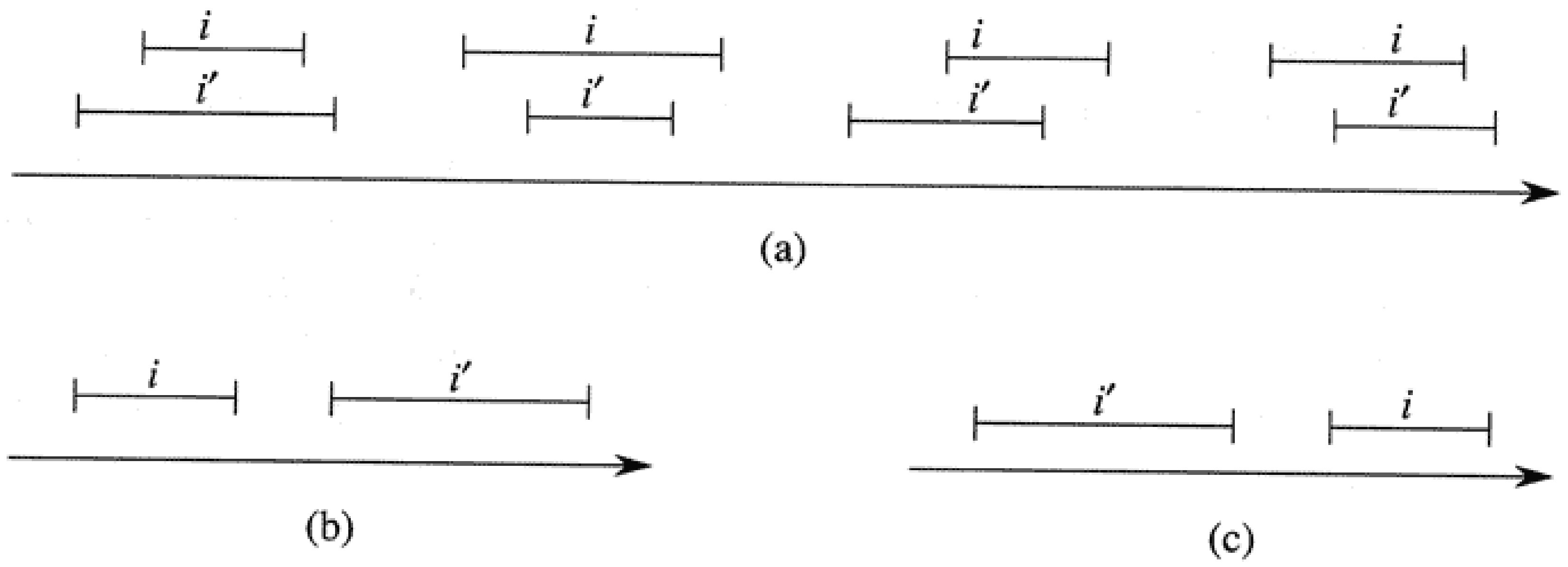
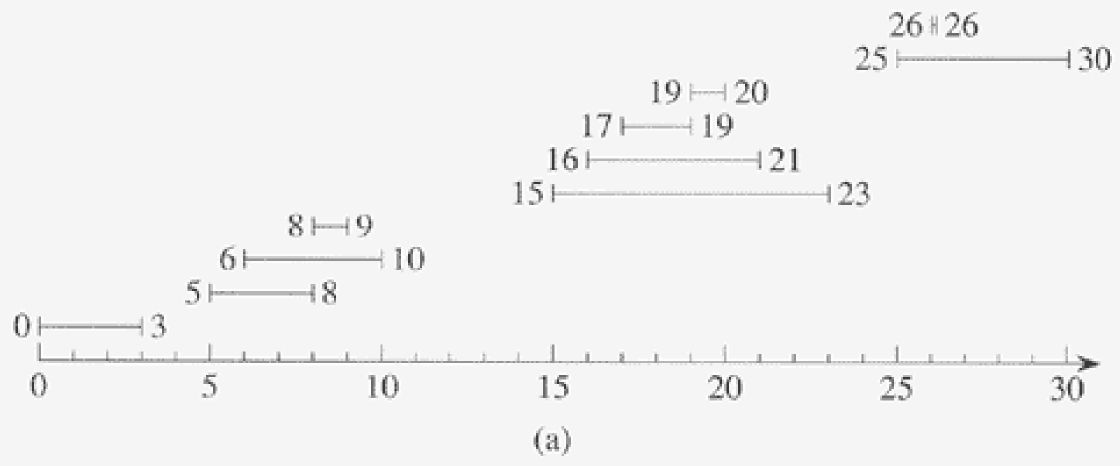


Figure 14.3 The interval trichotomy for two closed intervals i and i'. **(a)** If i and i' overlap, there are four situations; in each, $low[i] \le high[i']$ and $low[i'] \le high[i]$. **(b)** The intervals do not overlap, and $high[i] < low[i']$. **(c)** The intervals do not overlap, and $high[i'] < low[i]$.



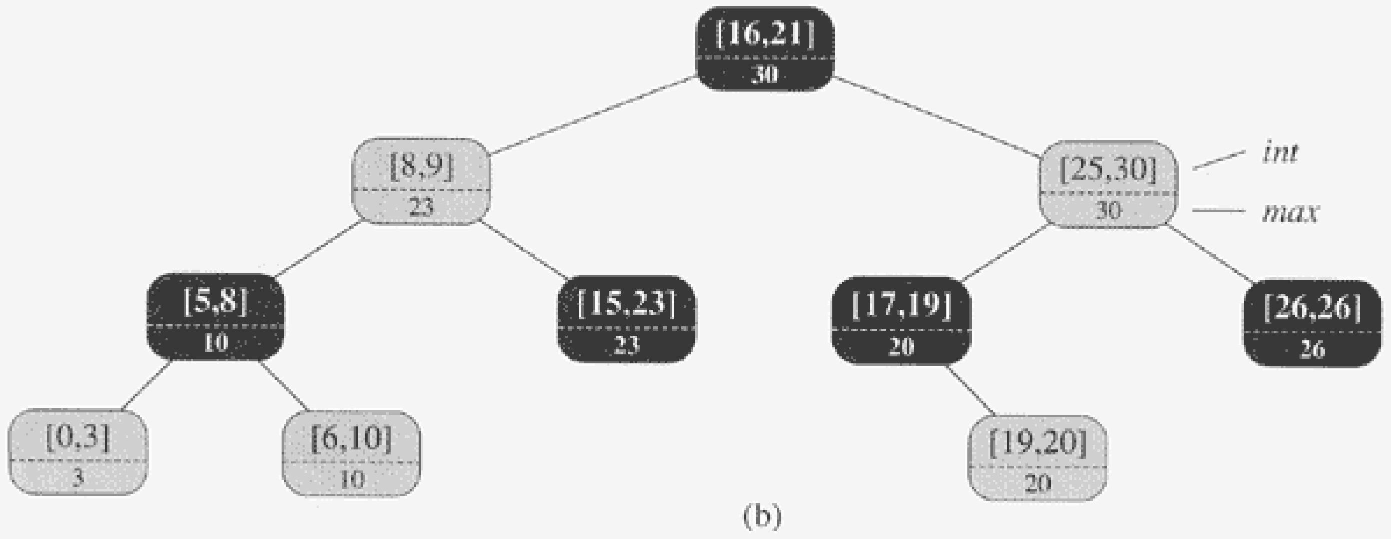
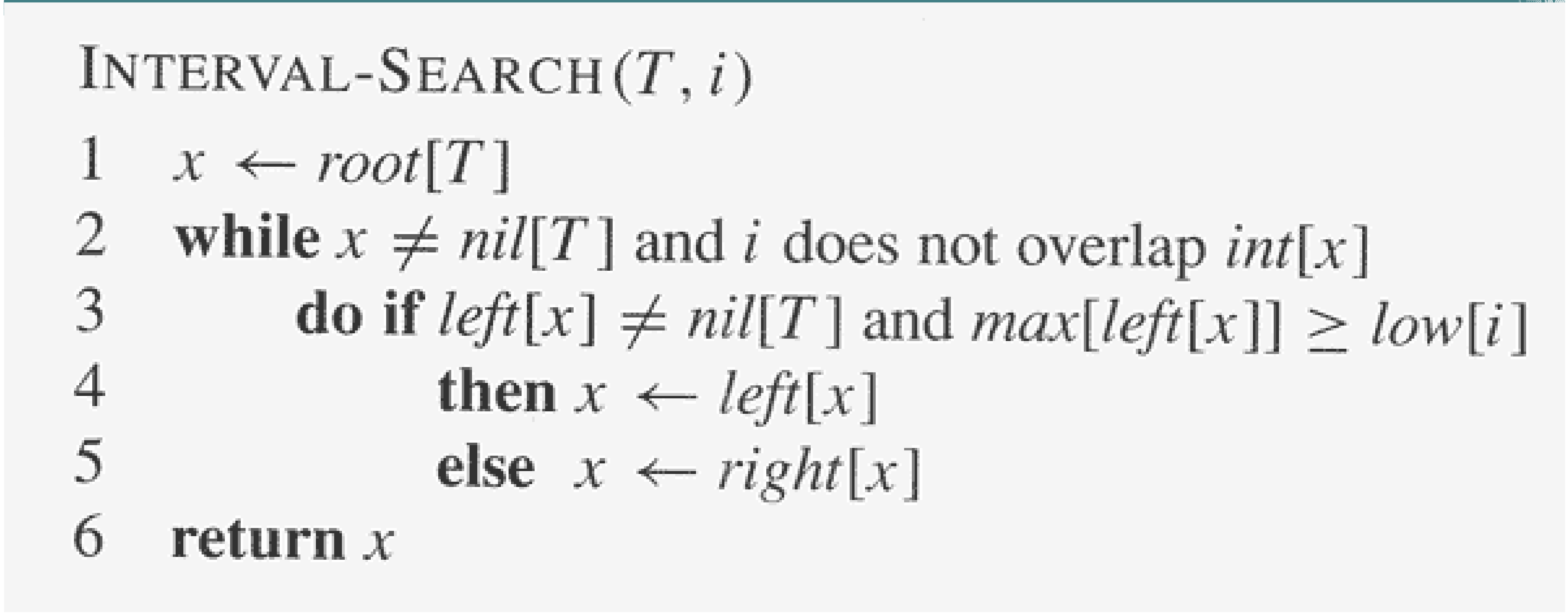


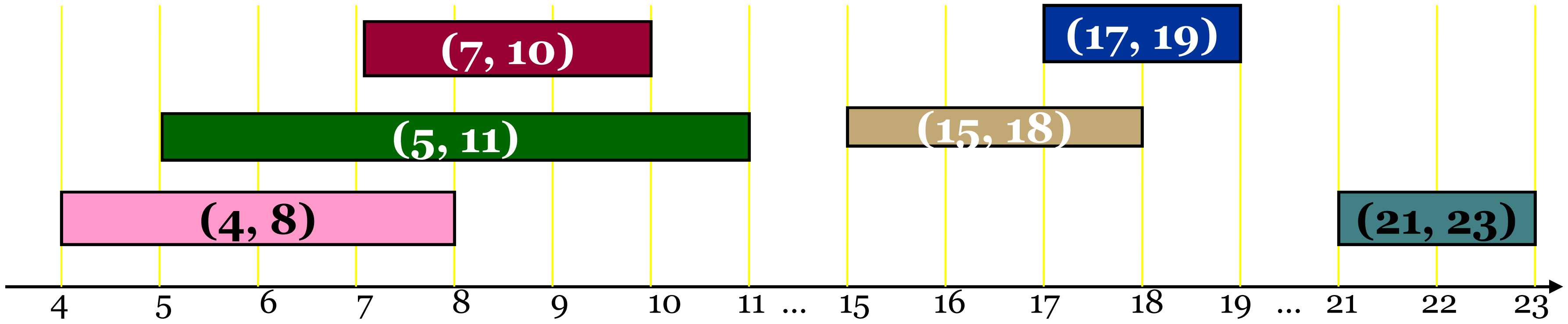
Figure 14.4 An interval tree. **(a)** A set of 10 intervals, shown sorted bottom to top by left endpoint. **(b)** The interval tree that represents them. An inorder tree walk of the tree lists the nodes in sorted order by left endpoint.



```
INTERVAL-SEARCH($T$, $i$)
x ← root[T]
while $x \neq nil[T]$ and $i$ does not overlap $int[x]$
    do if $left[x] \neq nil[T]$ and $max[left[x]] \geq low[i]$
        then $x \leftarrow left[x]$
        else $x \leftarrow right[x]$
return $x$
```

If go to right, then safe since there is no interval in the left overlapping with i.
If go to left, either there is an interval in the left overlapping with i or there is no overlaps. In the latter, we can prove that there will also be no overlaps in the right.

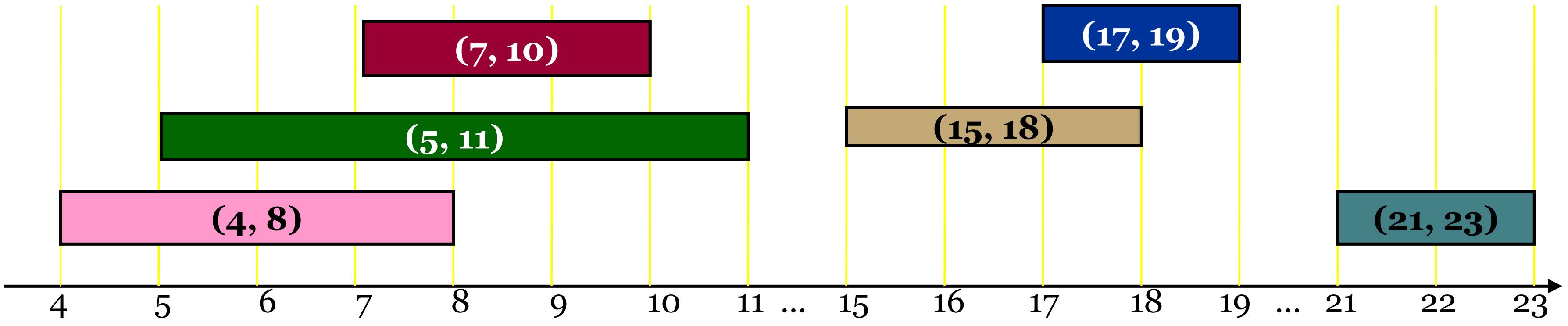
Interval Trees



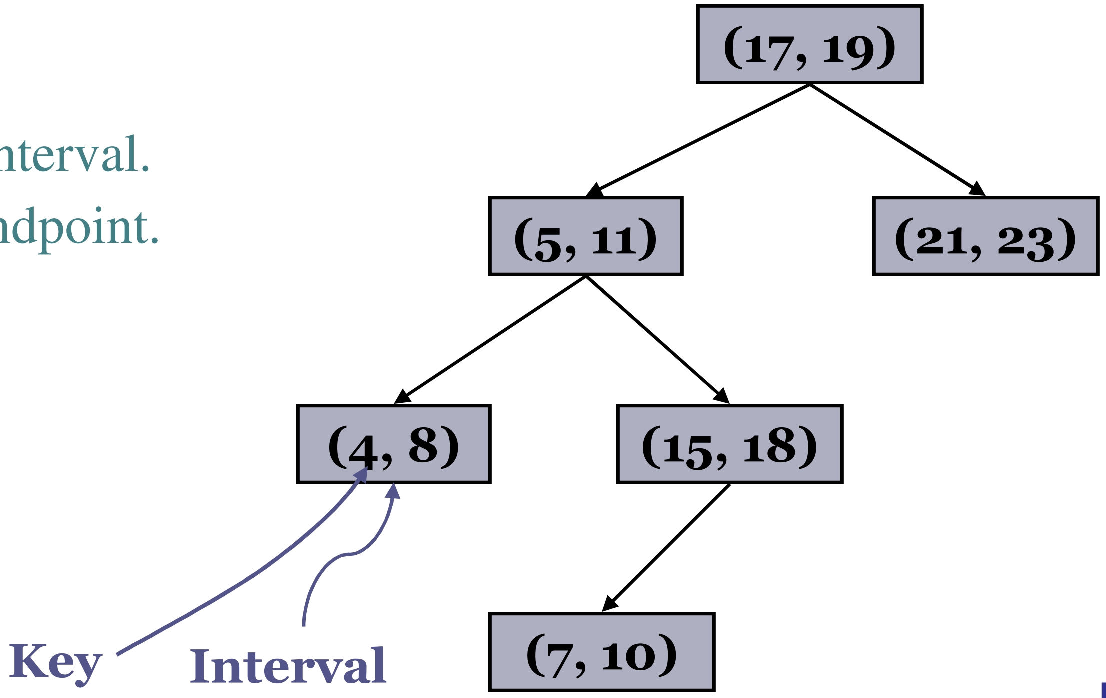
- Support following operations.

- Interval-Insert(i, S): Insert interval i = (ℓ_i, r_i) into tree S.
- Interval-Delete(i, S): Delete interval i = (ℓ_i, r_i) from tree S.
- Interval-Find(i, S): Return an interval x that overlaps i, or report that no such interval exists.

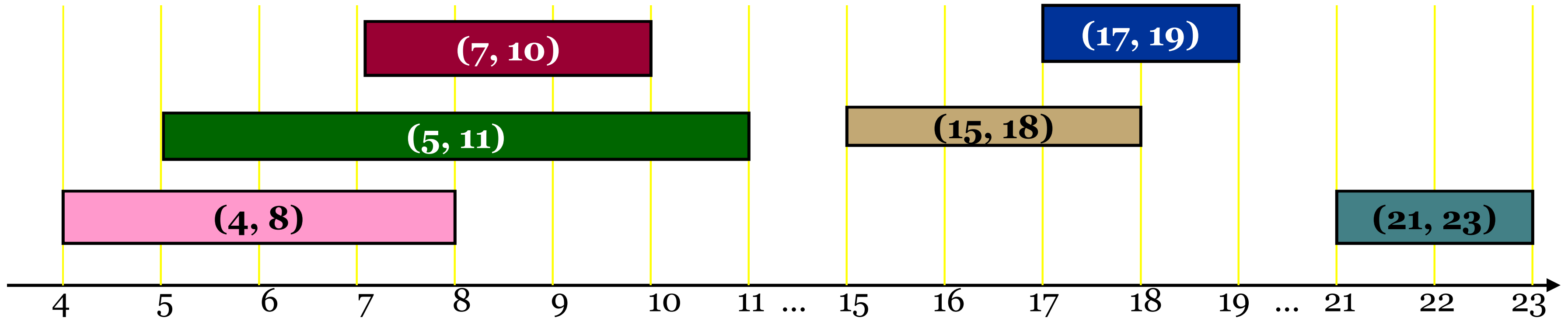
Interval Trees



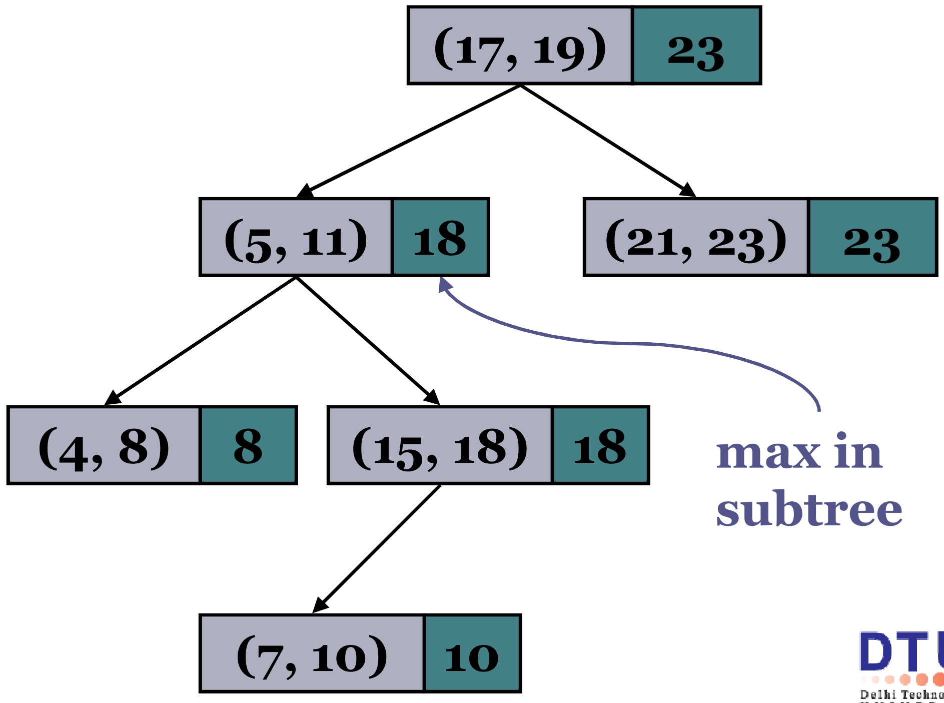
- Key ideas:
  - Tree nodes contain interval.
  - BST keyed on left endpoint.



Interval Trees

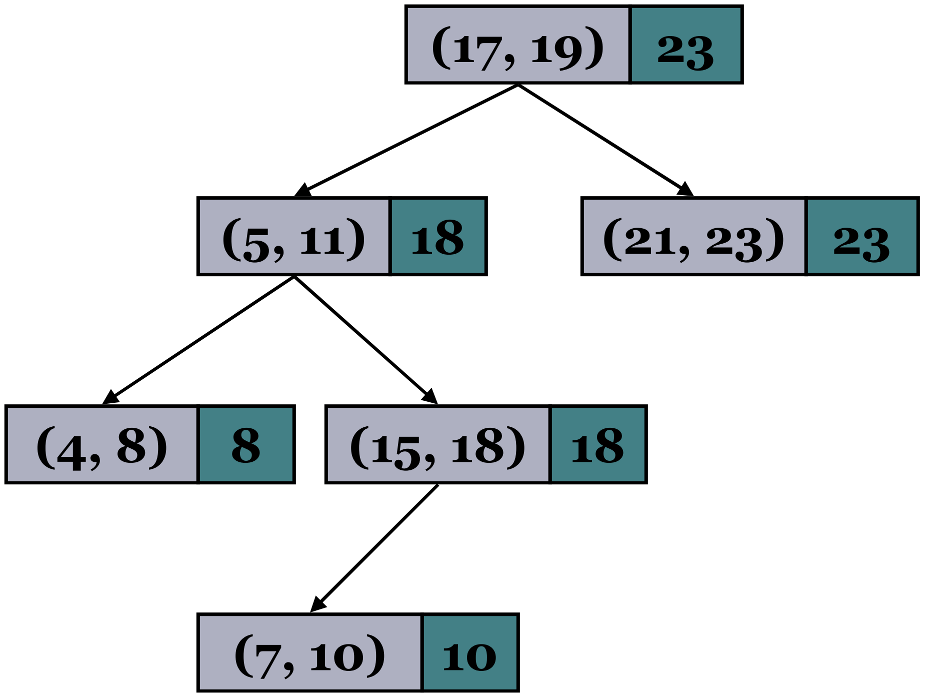


- Key ideas:
  - Tree nodes contain interval.
  - BST keyed on left endpoint.
  - Additional info: store max endpoint in subtree rooted at node.



Finding an Overlapping Interval

- Interval-Find(i, S): return an interval x that overlaps i = (ℓ_i, r_i), or report that no such interval exists.


(17, 19) 23
(5, 11) 18
(21, 23) 23
(4, 8) 8
(15, 18) 18
(7, 10) 10