Red-Black Trees

Data Structures and Algorithms

Andrei Bulatov

Red-Black Trees

All binary search tree operations take O(h) time, where h is the height of the tree

Therefore, it is important to `balance’ the tree so that its height is as small as possible

There are many ways to achieve this

One of them: Red-Black trees

Every node of such a tree contains one extra bit, its color

Another agreement: the Nil pointer is treated as a leaf, an extra node

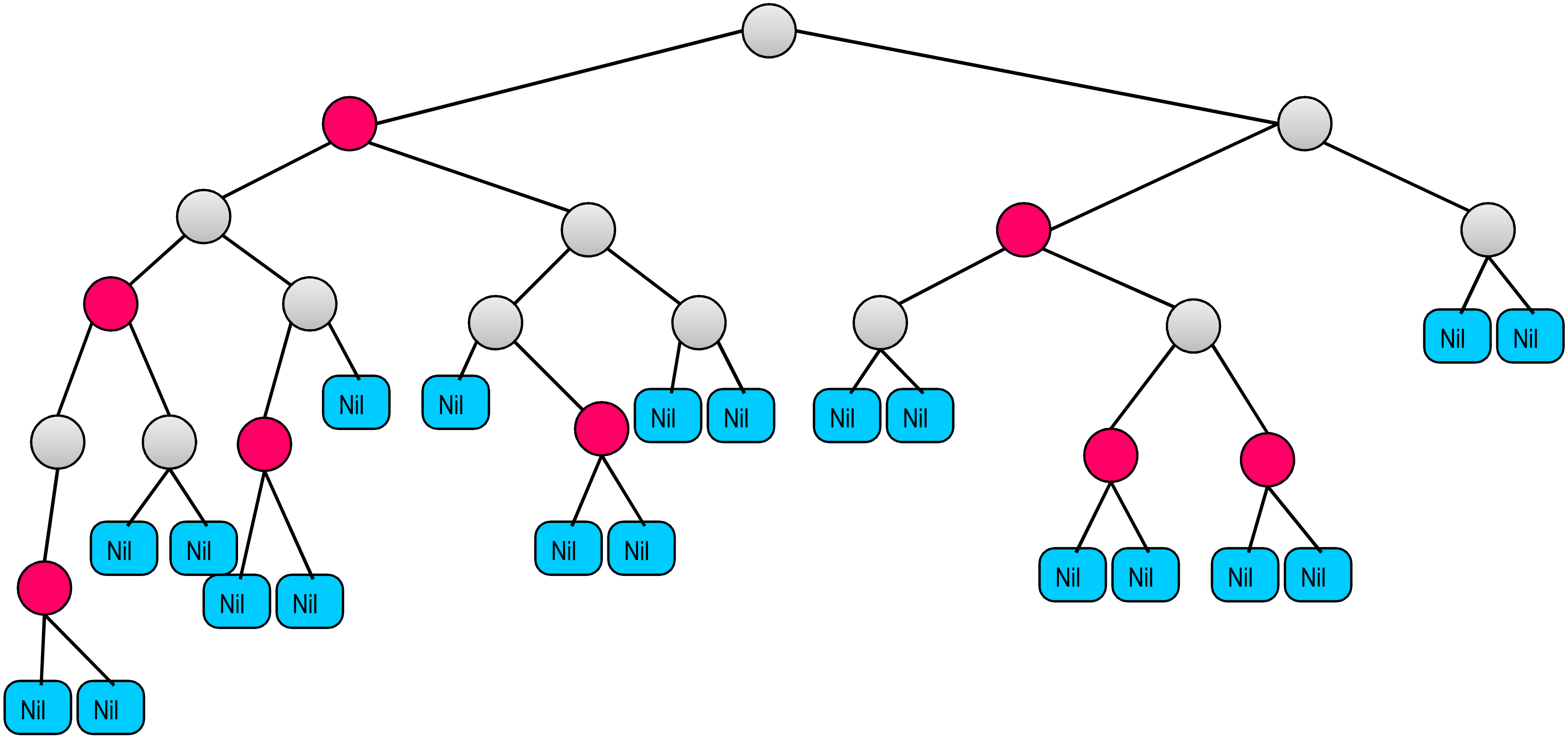
The rest of the nodes are called internal

Red-Black Properties

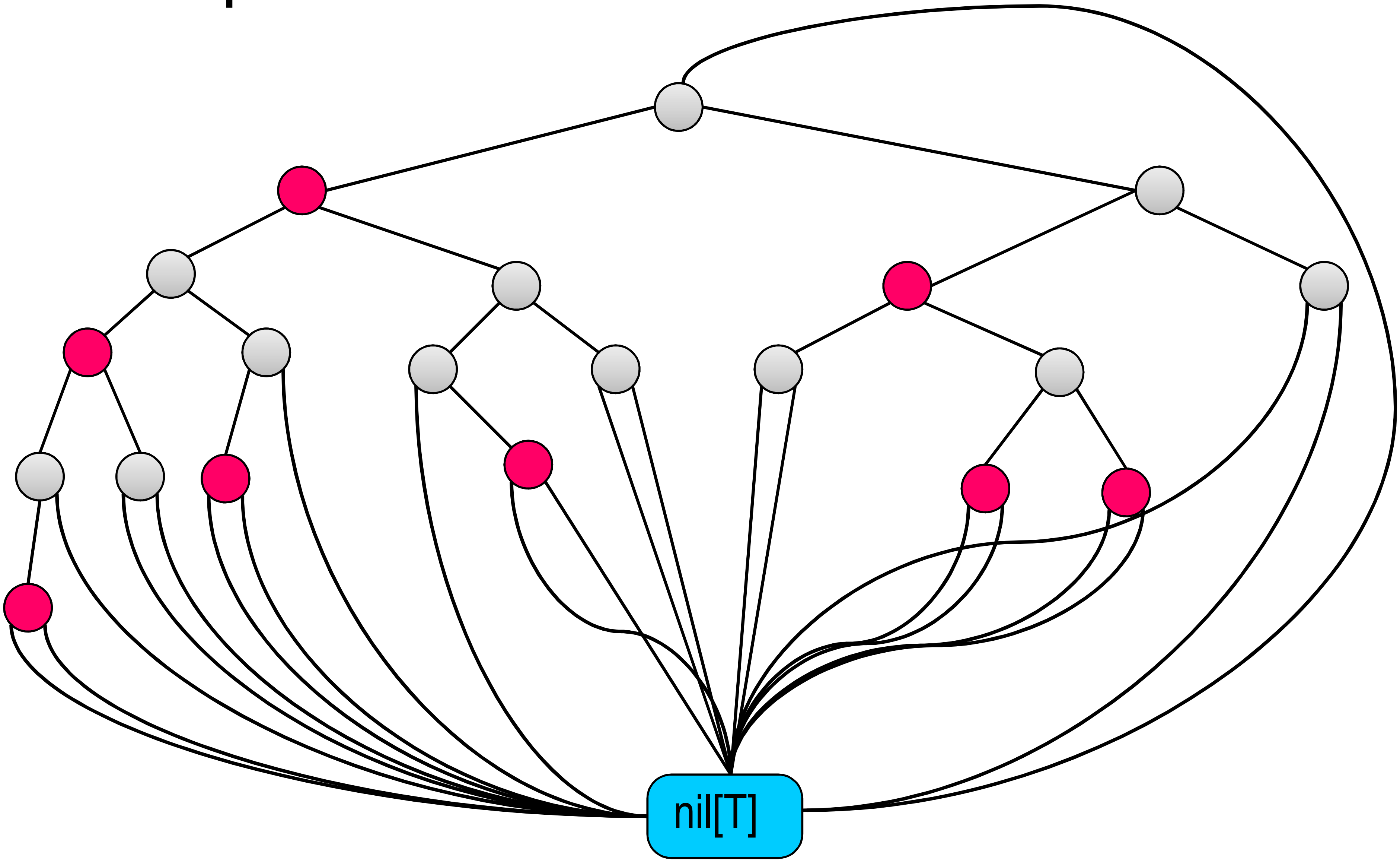
A binary search tree is a red-black tree if it satisfies the following red-black properties:

- Every node is either red or black
- The root is black
- Every leaf (Nil) is black
- If a node is red, then both its children are black
- For each node, all paths from the node to descendant leaves contain the same number of black nodes

Example


Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil
Nil

Example



Black Height

The number of black nodes on paths from node x to its descendant leaves in a red-black tree is called its black height, denoted bh(T)

Lemma

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

Proof

We show first that the subtree rooted at x contains at least $2^{bh(x)} - 1$ nodes

Induction on bh(x)

Base Case: If bh(x) = 0, then x is a leaf, nil[T]

In this case, $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes

Black Height (cntd)

Inductive hypothesis: the claim is true for any y with height less than that of x

Inductive Case: Let bh(x) > 0 and has two children

The black height of the children is either bh(x) or bh(x) – 1, depending on its color

Since the children have smaller height, we can apply the induction hypothesis

Thus the subtree rooted at x contains at least

$$(2^{bh(x)-1}-1)+(2^{bh(x)-1}-1)+1=2^{bh(x)}-1$$

nodes

Black Height (cntd)

Suppose h is the height of the tree

By the red-black property at least half of nodes on every root-to-leaf path are black (not including the root)

Therefore the black height of the root is at least h/2

Thus

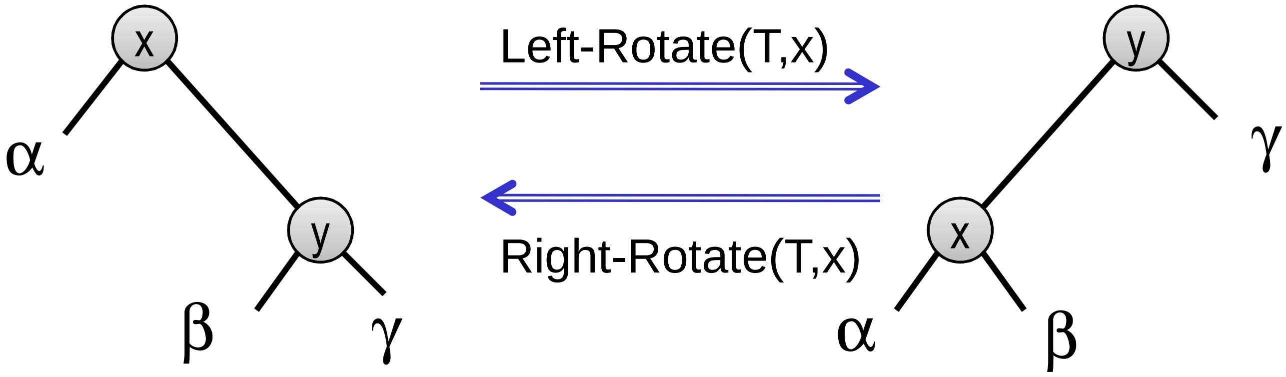
$$n \geq 2^{h/2} - 1$$

$$\log(n+1) \geq h/2$$

QED

Rotations

Sometimes we will need to rearrange pointers inside an RB-tree

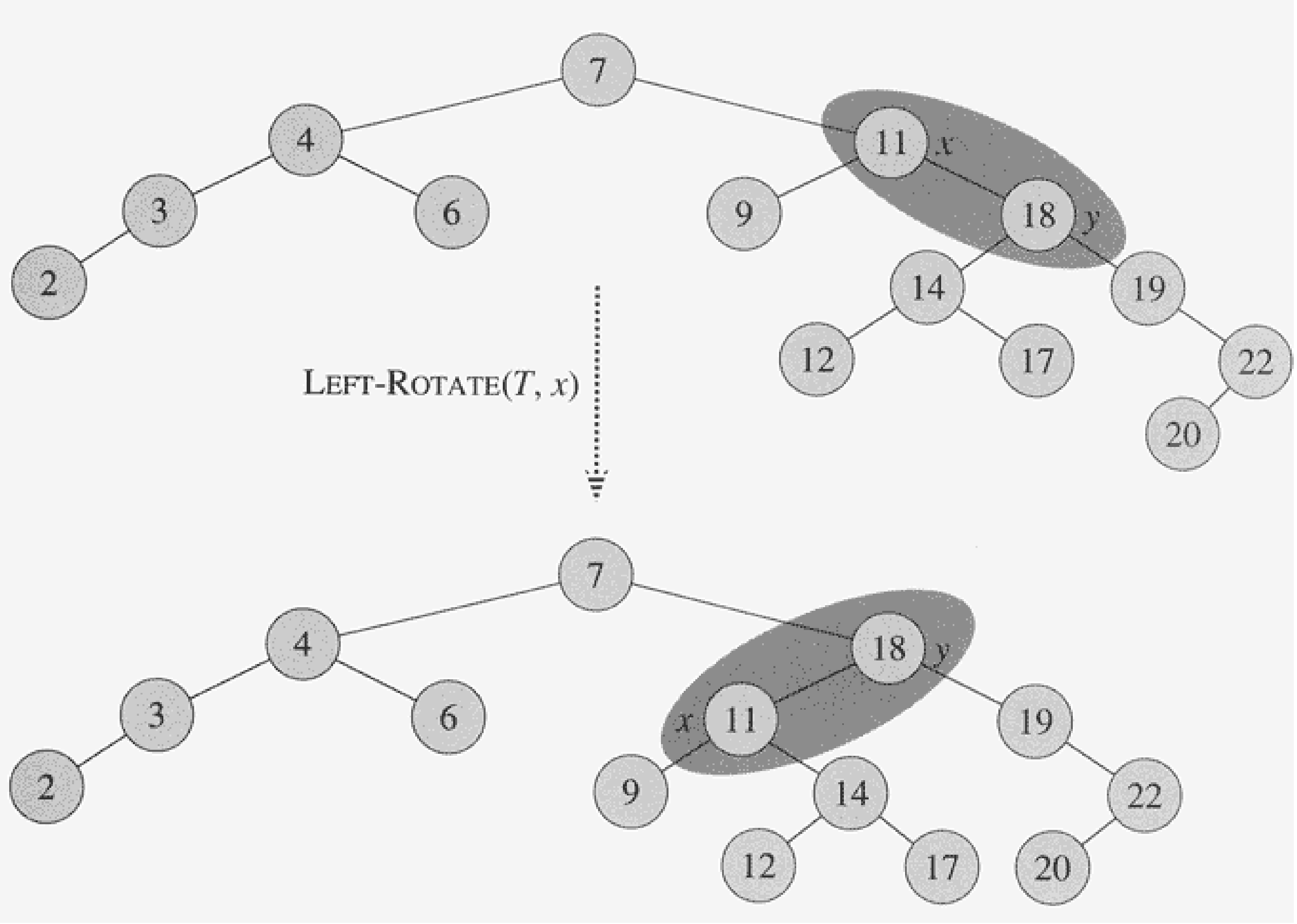


Rotations: Pseudocode

Left-Rotate(T,x)

```
set y:=right[x]
set right[x]:=left[y]
set parent[left[y]]:=x
set parent[y]:=parent[x]
if parent[x]=nil[T] then
    set root[T]:=y
else if x=left[parent[x]] then
         set left[parent[x]]:=y
     else
         right[parent[x]]:=y
set left[y]:=x
set parent[x]:=y
```

Rotations: Example



Insertion

Insertion for RB-trees is done in the same way as for ordinary binary search trees.

Except:

we should be careful about Nil links

the new node is colored red

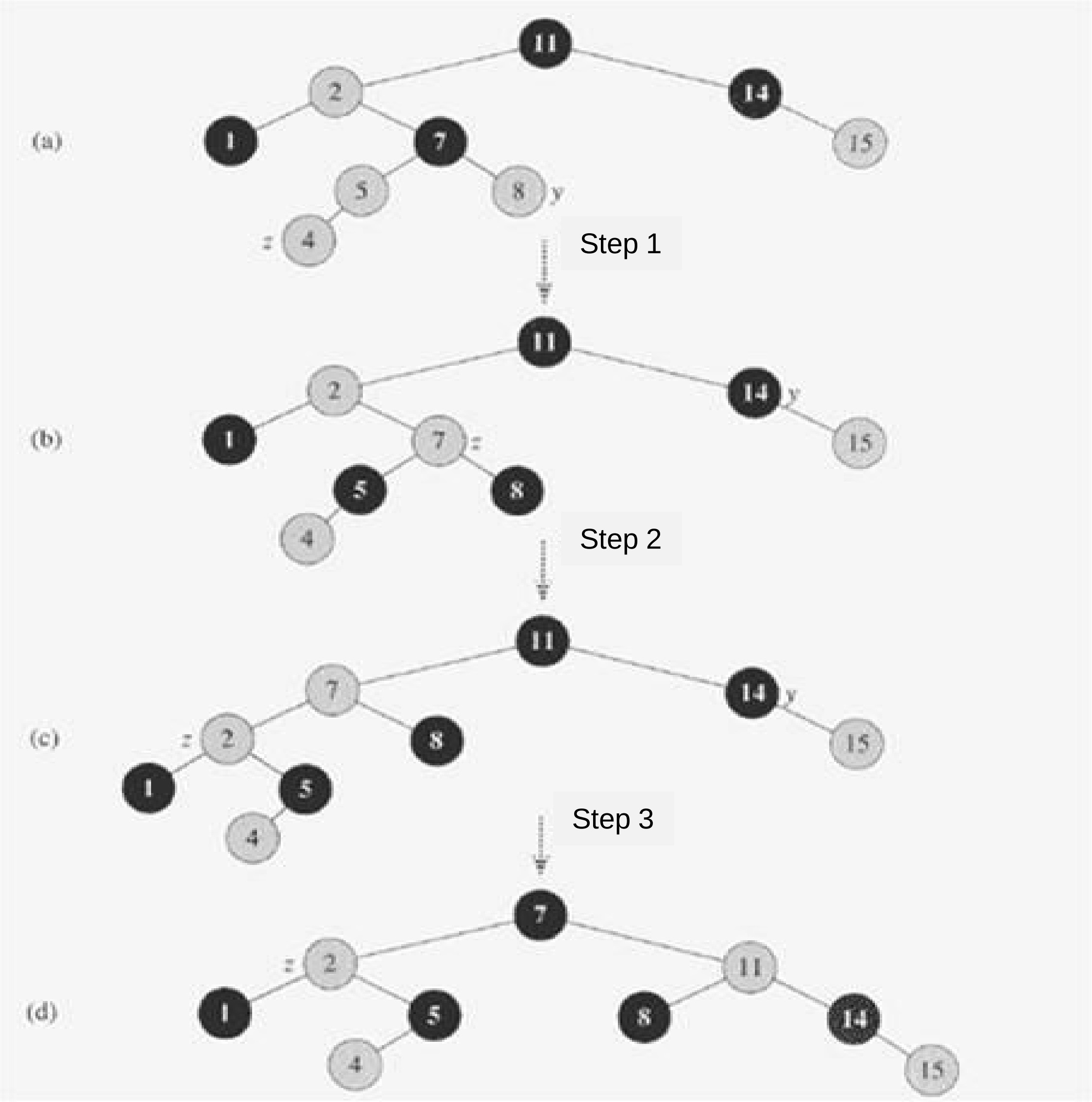
the resulting tree may not be an RB-tree, we need to fix it

Insertion: Pseudocode

RB-Insert(T,z)

```
set y:=Nil[T],   x:=root[T]
while x≠Nil[T] do
   set y:=x
   if key[z]<key[x] then set x:=left[x]
                    else set x:=right[x]
endwhile
set parent[z]:=y
if y=Nil[T] then set root[T]:=z
         else if key[z]<key[y] then set left[y]:=z
                               else set right[y]:=z
set left[z]:=Nil[T]  right[z]:=Nil[T]  color[z]:=RED
RB-Insert-FixUp(T,z)
```

Insertion: FixUp



FixUp: Pseudocode

```
RB-Insert-FixUp(T,z)
while color[parent[z]]=RED do
  if parent[z]=left[parent[parent[z]]] then do
    set y:=right[parent[parent[z]]]
    if color[y]=RED then do
      set color[parent[z]]:=BLACK  color[y]:=BLACK
      set color[parent[parent[z]]]:=RED
      set z:=parent[parent[z]]
    else do
      if z=right[parent[z]] then do
        set z:=parent[z]    Left-Rotate(T,z)
      set color[parent[z]]:=BLACK
      set color[parent[parent[z]]]:=RED
    else   (same as  then  with left and right swopped)
color[root[T]]:=BLACK
```

Soundness

Lemma

Given an RB-tree, algorithm RB-Insert produces an RB-tree

Proof

We show that the main loop preserves the following loop invariant:

(a) Node z is red

(b) If parent[z] is the root then parent[z] is black

(c) If there is a violation of the red-black property, there is at most one violation, and it is of either property 2 or property 4.

If there is a violation of property 2, it occurs because z is the root and red

If there is a violation of property 4, it occurs because both z and parent[z] are red

Soundness: Initialization

(a) When RB-Insert-FixUp is called, z is a red node

(b) If p[z] is the root, it hasn't changed yet, and so is black

(c) Properties 1, 3, and 5 are true

If property 2 is violated, then the red root is just added, that is the root is z. In this case the tree does not have internal vertices. Property 4 is not violated in this case

If property 4 is violated, then, since the children of z are black sentinels and the tree had no prior violations, the violation must be because both z and parent[z] are red.

There are no other violations of red-black properties.

Soundness: Termination

When the big loop of RB-Insert-FixUp terminates parent[z] is black (if z is the root then parent[z] is the sentinel)

Therefore there is no violation of property 4.

If property 2 is violated, then the last line of RB-Insert-FixUp restores it.

Soundness: Maintenance

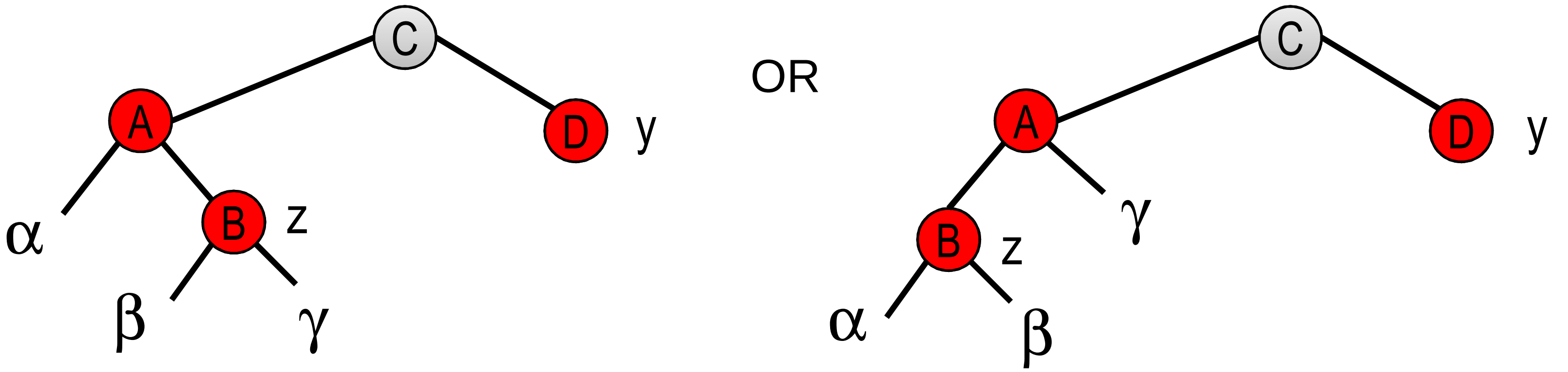
There are several cases to consider. We only consider those in which parent[z] is the left child of parent[parent[z]]

This is the situation given in the pseudocode

If parent[parent[z]] does not exist then parent[z] is the root, and by part (b) is black.

Therefore if the loop does not terminate here, parent[parent[z]] exists

Case 1. z’s uncle y is red

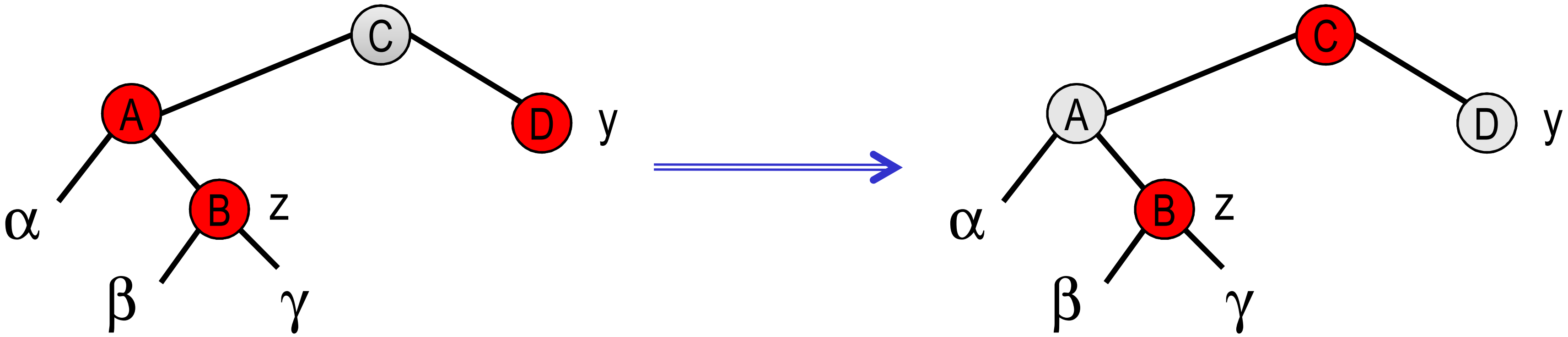


Soundness: Maintenance (cntd)

Both y and parent[z] are red, and parent[parent[z]] is black

We color y and parent[z] black, and parent[parent[z]] – red

We then set parent[parent[z]] to be the new node z



Show that the loop invariant is preserved

(a) Since we color parent[parent[z]] red, the new z (denoted z’) is red

Soundness: Maintenance (cntd)

(b) We have parent[z’] = parent[parent[parent[z]]], and the color of this node does not change.

If it is the root, it is black

(c) It is easy to see that Case 1 maintains property 5, and does not violate properties 1 and 3

If z’ is the root, then Case 1 corrected the only violation of property 4. Since z’ is red and it is the root, property 2 becomes violated, and it is due to z’

If z’ is not the root, then Case 1 has not created a violation of property 2.

Case 1 corrected the violation of property 4 due to z and parent[z].

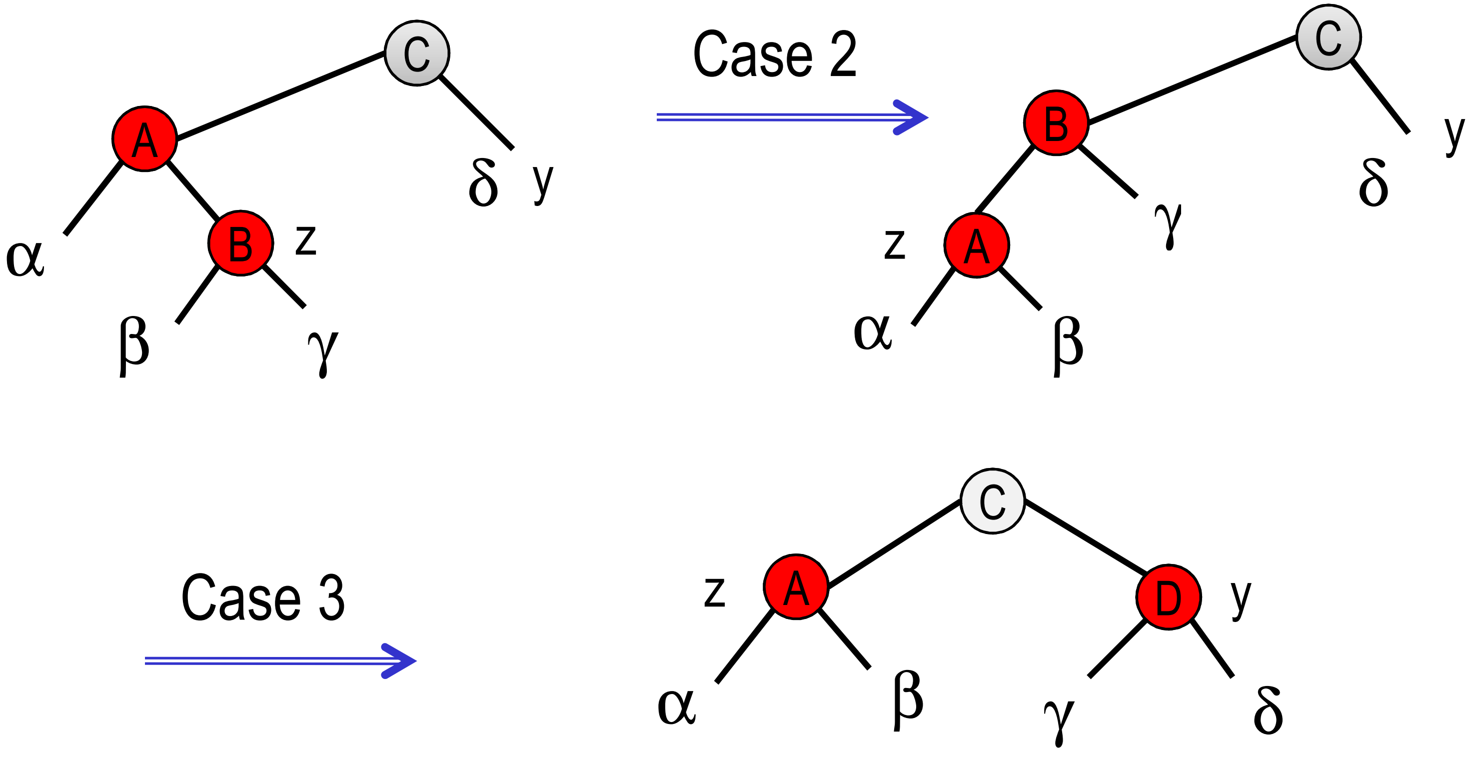
If parent[z’] is black then we are done

Otherwise z’ and parent[z’] create the only violation of property 4

Soundness: Maintenance (cntd)

Case 2. z’s uncle y is black and z is a right child

Case 3. z’s uncle y is black and z is a left child



Soundness: Maintenance (cntd)

Case 2. z’s uncle y is black and z is a right child

Case 3. z’s uncle y is black and z is a left child

In Case 2 the algorithm performs rotation and we get Case 3

Since both nodes being rotated are red no RB properties change

Moreover, parent[parent[z]] exists and it does not change when Case 2 is transformed into Case 3

In Case 3 we perform same rotations and color changes that does not affect property 5

Then since there are no longer 2 red nodes in a row, the while loop is not executed again, for parent[z] is now black

Soundness: Maintenance (cntd)

(a) Case 2 makes z points to parent[z], which is red

Nothing changes in Case 3

(b) Case 2 does not change the color of the root

Case 3 makes parent[z] black, so if it is the root, it is black

(c) As in Case 1, properties 1, 3, and 5 are maintained in Cases 2 and 3

Since node z is not the root, there is no violation of property 2

Cases 2 and 3 do not introduce any violation of property 2, since the only node that becomes red also becomes a child of a black node by the rotation in Case 3

Cases 2 and 3 correct the only violation of property 4 and do not introduce another violation

Running Time

Since the height of an RB-tree with n nodes is O(\log n), RB-Insert takes O(log n) time, except for RB-Insert-FixUp

In RB-Insert-FixUp the while loop repeats only if Case 1 takes place, and in this case z moves up 2 levels.

The total number of iterations is O(log n)

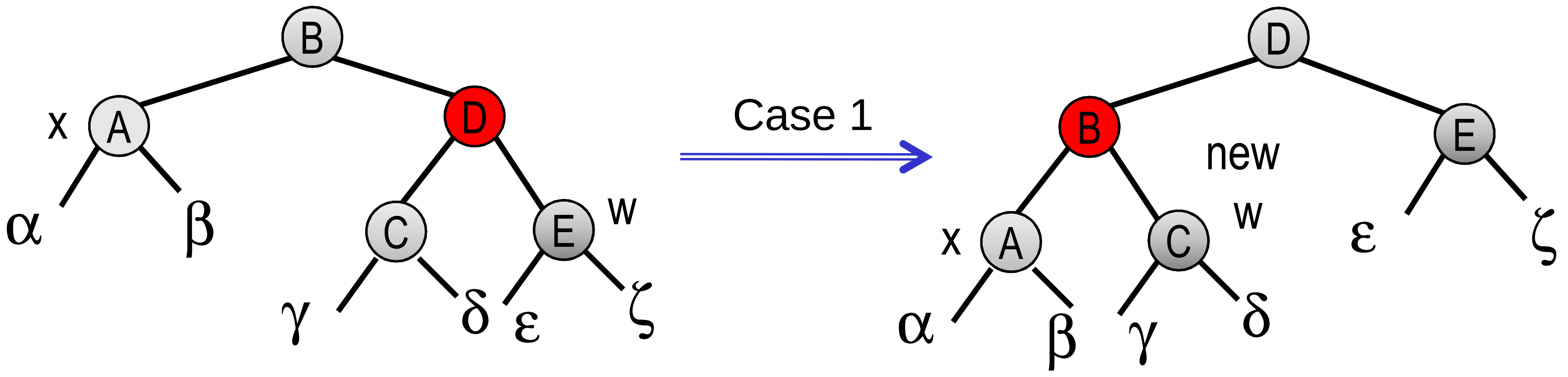
RB-Deletion

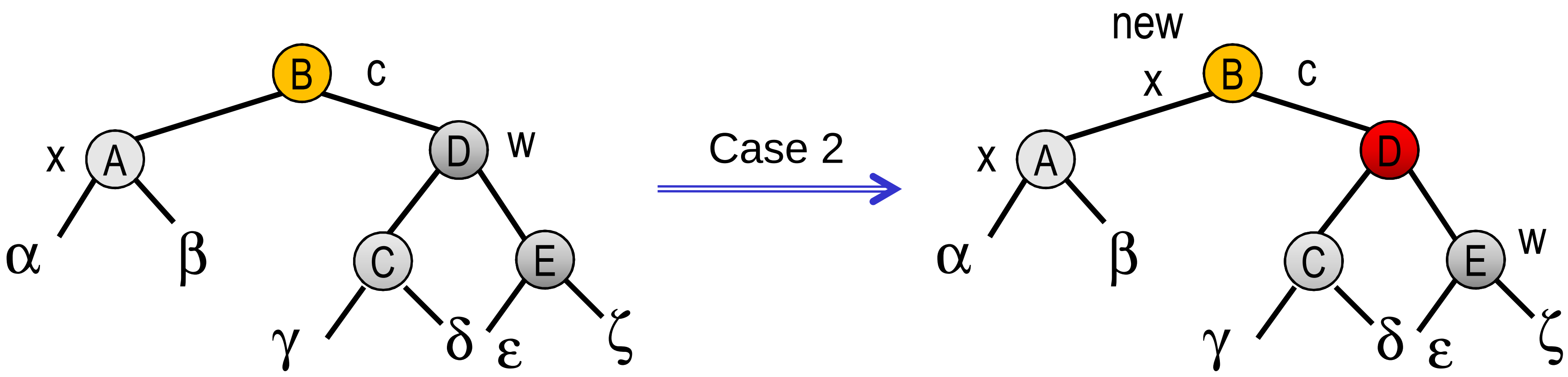
RB-Delete also first runs the regular Deletion algorithm, and then fixes violations of the RB properties

RB-Delete(T,z)

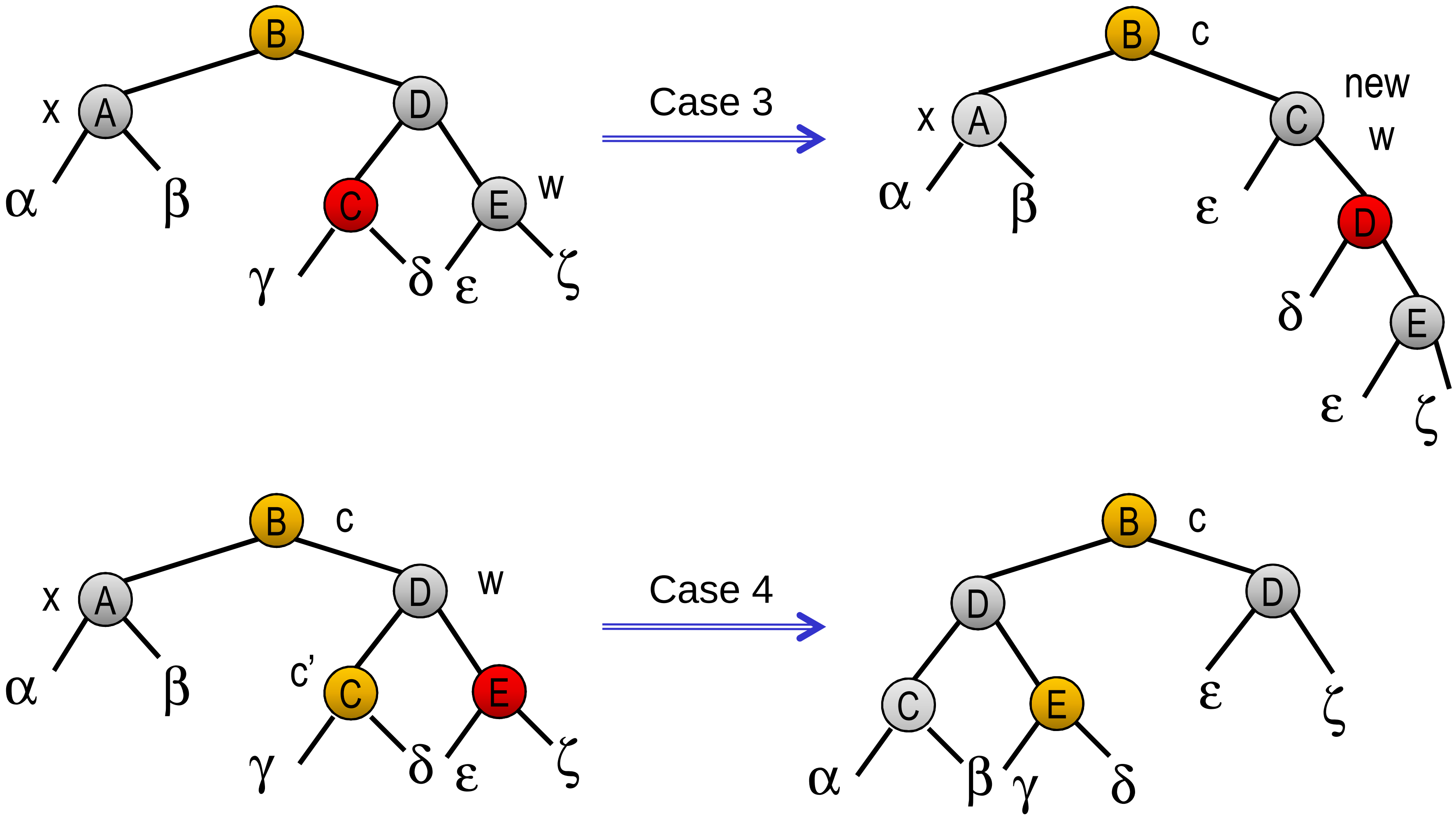
```
if left[z]=Nil[T] or right[z]=Nil[T] then set y:=z
                       else set y:=Tree-Successor(z)
if left[y]≠Nil[T] then set x:=left[y]
                  else set x:=right[y]
if x≠Nil[T] then set parent[x]:=parent[y]
            else set x:=right[x]
if parent[y]=Nil[T] then set root[T]:=x
 else if y=left[parent[y]]:=x then left[parent[y]:=x
                                else right[parent[y]]:=x
if y≠z then do
  set key[z]:=key[y]
  copy y’s data into z
if color[y]=BLACK then  RB-Delete-FixUp(T,x)
return y
```

Deletion: Pictures





Deletion: Pictures



new x = root[T]

RB-Delete-FixUp

RB-Delete-FixUp(T,x)

```
while x≠root[T] and color[x]=BLACK do
  if x=left[parent[x]] then do
    set w:=right[parent[x]]
Case 1
    if color[w]=RED then do
      set color[w]:=BLACK   color[parent[x]]:=RED
      Left-Rotate(T,parent[x])   set w:=right[parent[x]]
Case 2
    if color[left[w]]=BLACK and color[right[w]]=BLACK then
      set color[w]:=RED  x:=parent[x]
    else if color[right[w]]=BLACK then do
Case 3
           set color[left[w]]:=BLACK   color[w]:=RED
           Right-Rotate(T,w)   set w:=right[parent[x]]
Case 4
         set color[w]:=color[parent[x]]
         set color[parent[x]]:=BLACK
         set color[right[w]]:=BLACK
         Left-Rotate(T,parent[x])   set x:=root[T]
  else (same as then  flipping left and right)
set color[x]:=BLACK
```