Red-Black Trees

- Red-black trees are one of many search tree schemes that are "balanced" in order to guarantee that basic set operations, i.e. Search, Insert, Delete, Minimum, Maximum, Predecessor, and Successor take $O(\log_2 n)$ time in worst case.
- Red-black trees are approximately balanced binary search trees.
- Binary search trees:
  - † Let h be the height of the tree.
  - † basic set operations take $O(h)$ time.
  - † if h is large, the performance may be no better than that of linked lists.

- A red-black tree is a binary search tree with one extra bit of storage per node: its color. The color can be either red or black.
- By constraining the way nodes can be colored, red-black trees ensure that no path from root to a leaf is more than twice as long as any other path.
- Therefore red-black trees are approximately balanced.

Definitions

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

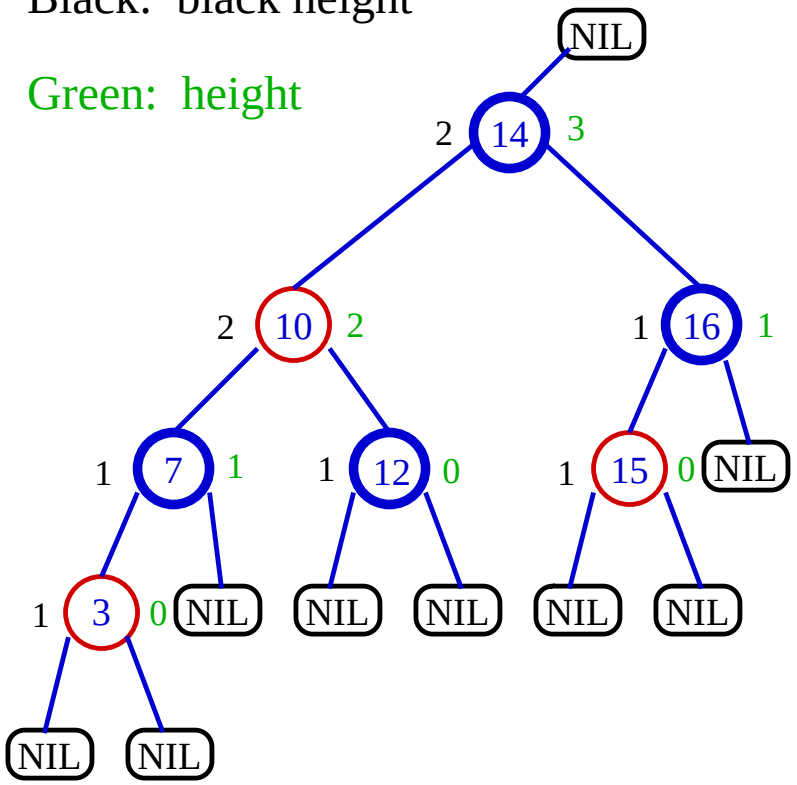
  0) Every node is either red or black,
  1) Root node is black,
  2) Every leaf (NIL) is black,
  3) If a node is red, then both its children are black,
  4) Every simple path from a node to a descendant leaf contains the same number of black nodes.

- Black-height of a node x, $bh(x)$: the number of black nodes on the path from, but not including, node x to a leaf.
- Black-height of a red-black tree: the black-height of its root.

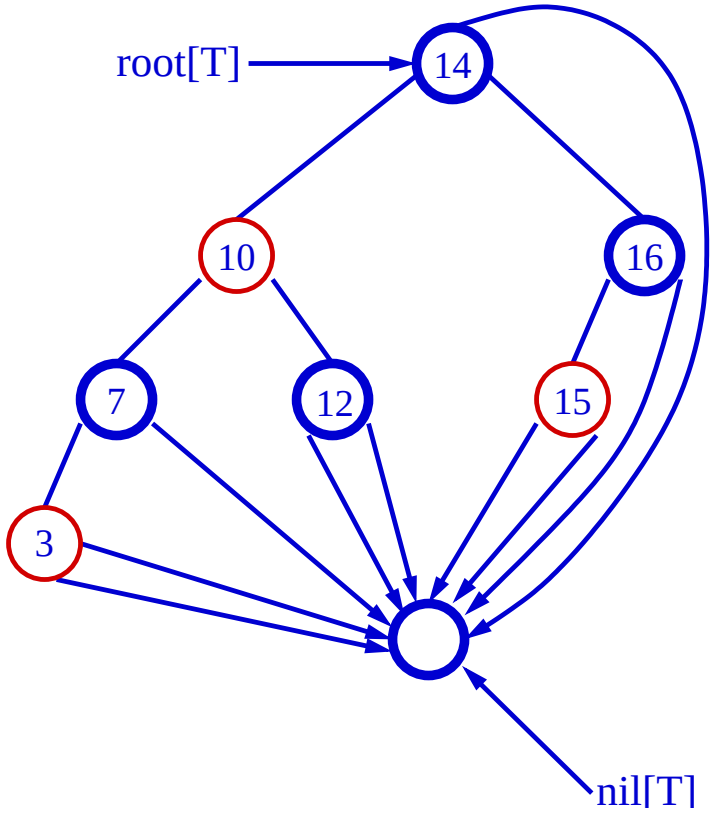
Example:

Black: black height

Green: height



Implementation



- Each node contains several fields: color, key, left, right, parent.
- If a child or a parent of a node does not exist, the corresponding pointer field contains the value NIL.
- We consider these NIL's as being pointers to external nodes (leaves).
- We consider the normal nodes (nodes with keys) as internal nodes.
- Implementation
  - † NIL's can be implemented with one extra node whose color is black and whose left, right, and parent are set to itself initially.
  - † root[T]: pointer to the root node.
  - † nil[T]: pointer to the node of NIL.

The following theorem shows why red-black trees are good search trees.

Theorem. The height of a red-black tree with n internal nodes is at most $2\log_2(n+1)$.

We first prove two facts and then prove the theorem.

1 Let h be the height of the tree, then black-height of the tree is at least $h/2$.

2 Subtree rooted at node x contains at least $2^{bh(x)} - 1$ internal nodes.

Proof: (1)

- Consider the path from, but not including, the root which defines the height of the tree.
- By property 3) of the red-black tree definition, a red node has black child on this path.
- Therefore on this path, the number of red nodes is less or equal to the number of black nodes.
- Hence the black-height of the tree is at least $h/2$.

□

Proof: (2)
(Prove by induction on the height of x.)

- Base case: height of x is 0.

  † x is a leaf (NIL) node.
  † It contains $2^{bh(x)} - 1 = 2^0 - 1 = 1 - 1 = 0$ internal nodes.

- Induction step: height of x is greater than 0.

  † x is an internal nodes with two children.
  † Each child has a black-height of
    – either bh(x), if its color is red,
    – or bh(x)-1, if its color is black.
  † By induction each child has at least $2^{bh(x)-1} - 1$ internal nodes.
  † x has at least $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ internal nodes.

□

Proof: (Theorem)

- Let h and bh be the height and black-height of the tree.
- By (1) $bh \geq h/2$.
- By (2) $n \geq 2^{h/2} - 1$
- Therefore $2^{h/2} \leq n + 1$
  $h/2 \leq \log_2(n+1)$
  $h \leq 2\log_2(n+1)$.

□

An immediate result from the theorem: search operation in red-black trees is in $O(\log_2 n)$.

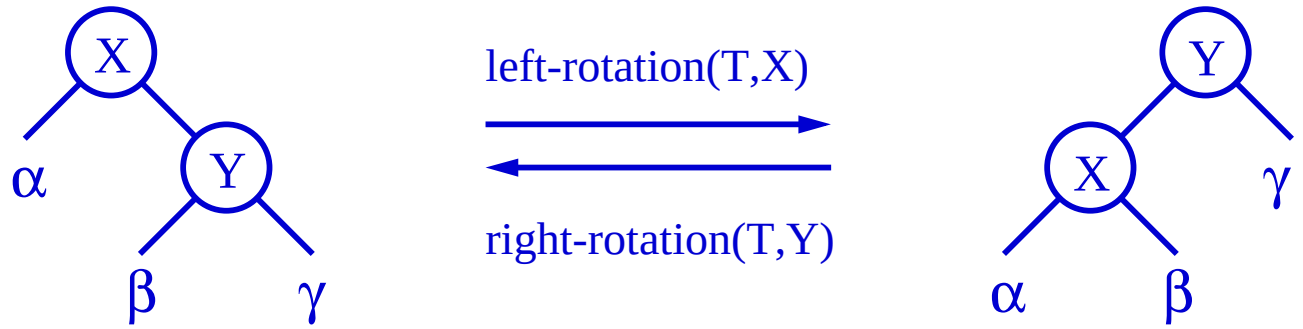
Insertion and deletion can be done in $O(\log_2 n)$ time.

However, since they modify the tree structure, the resulting tree may violate the red-black properties 0), ..., 4).

To restore these properties, we must change the colors of some of the nodes and also change the tree structure (change pointers).

We change the tree structure through **Rotation**.

- rotation is a local operation on the tree.
- rotation preserves the in order key ordering.

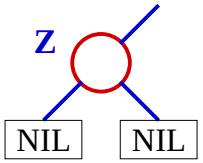


Pseudo code for left_rotation(T, x).
(right[x] $\neq$ nil[T], root's parent is nil[T])

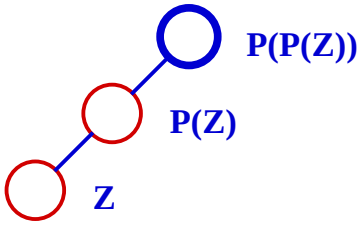
```
1   y ← right[x]                    ▷ Set y
2   right[x] ← left[y]              ▷ Turn y's left subtree into x's right subtree
3   p[left[y]] ← x
4   p[y] ← p[x]                     ▷ Link x's parent to y
5   if p[x] = nil[T]
6        then root[T] ← y
7        else if x = left[p[x]]
8             then left[p[x]] ← y
9             else right[p[x]] ← y
10  left[y] ← x                     ▷ Put x on y's left
11  p[x] ← y
```

Insertion in red-black trees

- Binary search tree insertion.
  - † insert at the bottom of the tree.
  - † change 

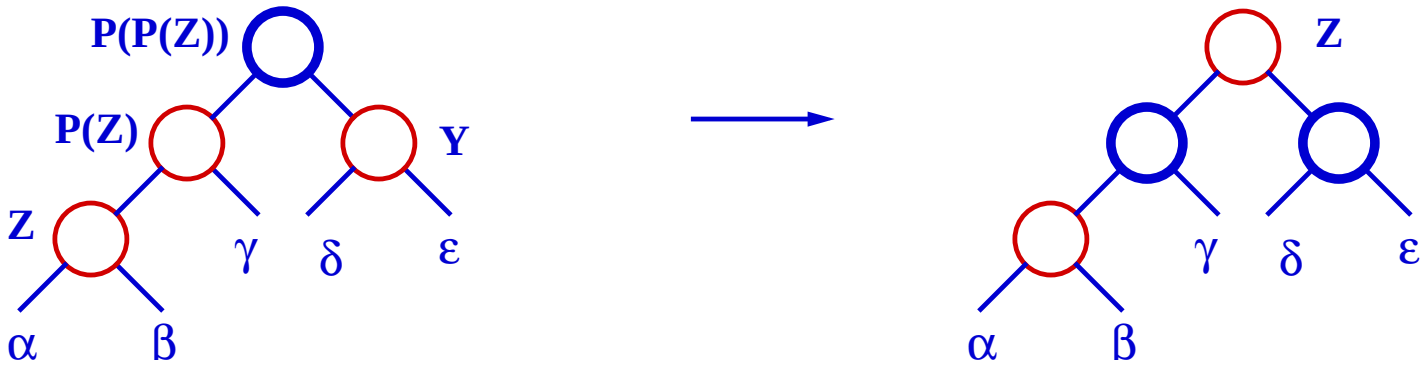
to 

- Color the new node z to red.
  - † no problem with black-height.
  - † no problem if $p(z)$ is black.
- May have problem with the color
  - † if $p(z)$ is red, we have a red node with a red child.
  - † since $p(z)$ is red, we know $p(p(z))$ is black.



How to fix this problem?
Suppose that y is z's uncle (y is the sibling of $p(z)$).

- Case 1: y is red.



Solution: color $p(z)$ and y to black, color $p(p(z))$ to red.

† This solution does not change the black-height.

† If $p(p(p(z)))$ is black, we are done.

† We may still have problem if $p(p(p(z)))$ is red.

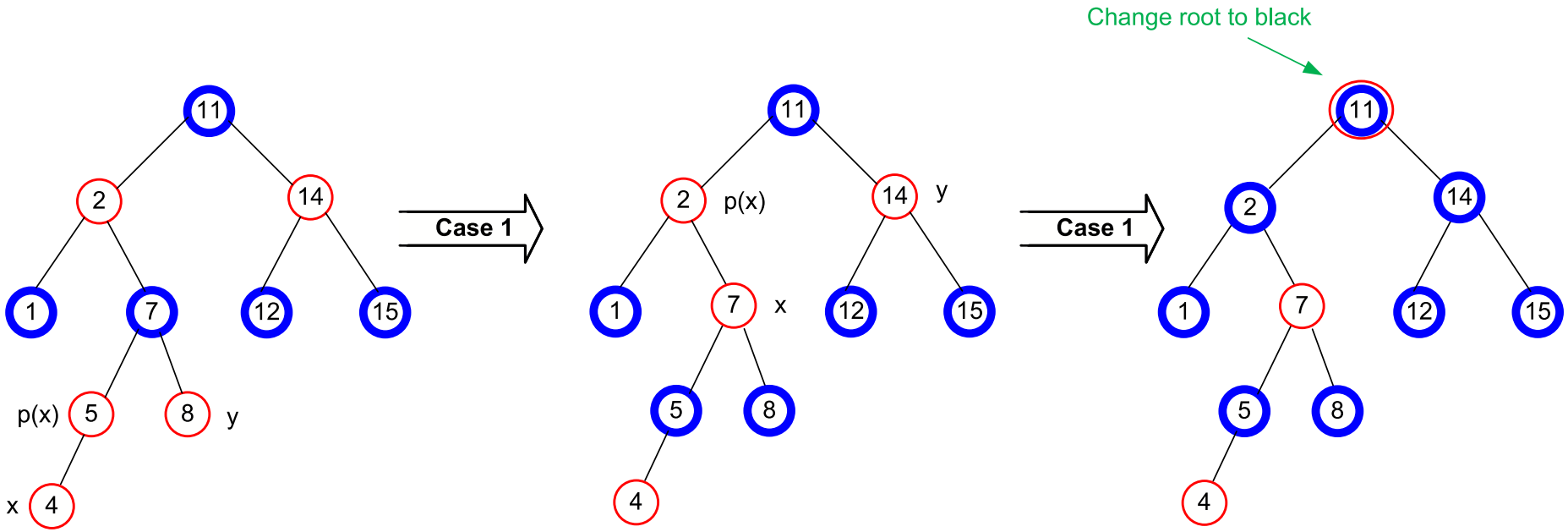
– $p(p(z))$ and $p(p(p(z)))$ are both red.

– but the problem is now two level up.

– when we reach the root, there will be no problem since root has no parent.

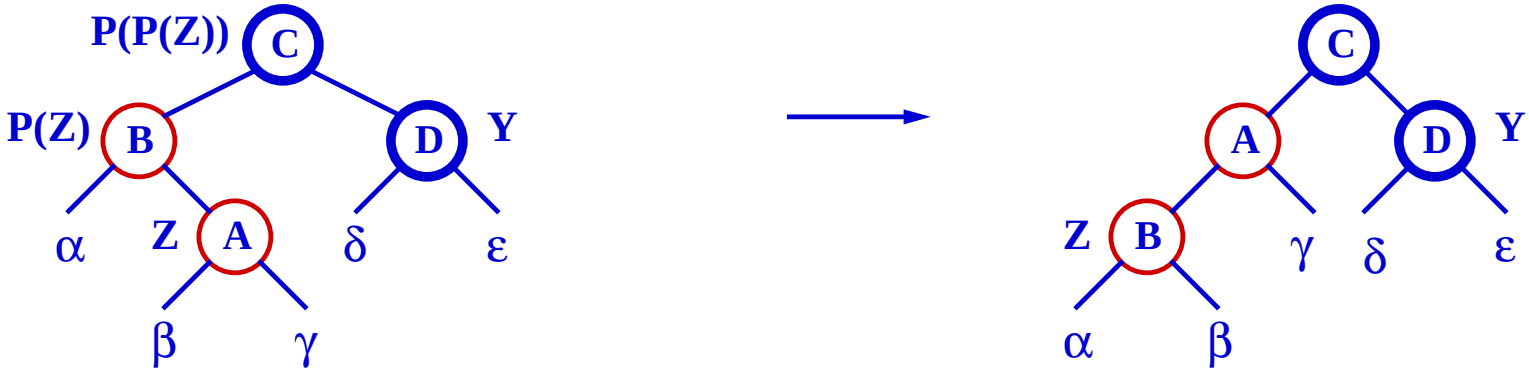
– we always color root to black.

Example: insertion case 1



- Case 2: y is black

2A: y is the right child of $p(p(z))$
z is the right child of $p(z)$.



Solution: left-rotation at $p(z)$.

† no problem with black-height.

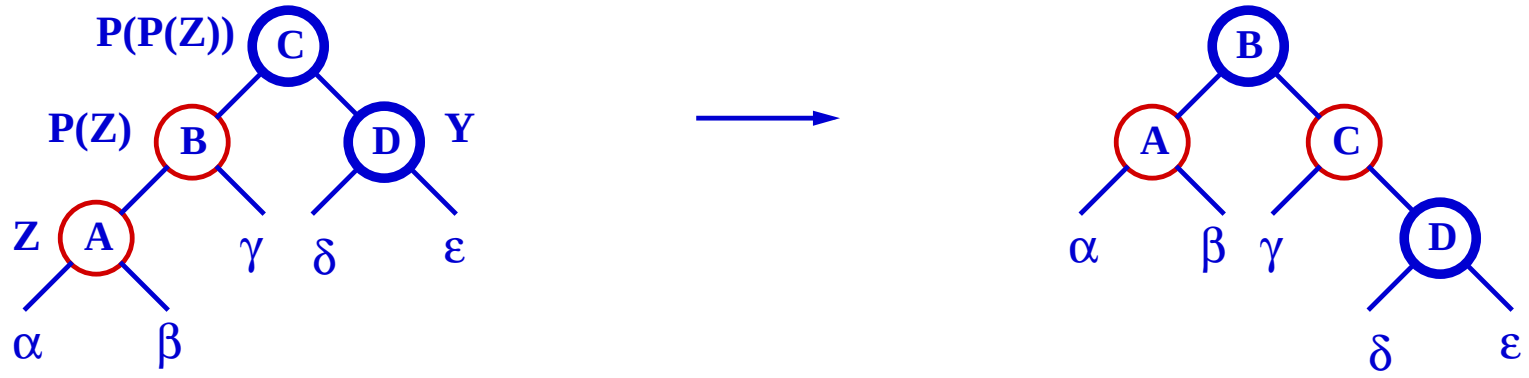
† transform case 2 to case 3.

† problem solved.

2B: y is the left child of $p(p(z))$
z is the left child of $p(z)$.
Solution is similar (right-rotation).

- Case 3: y is black

3A: y is the right child of $p(p(z))$
z is the left child of $p(z)$.



Solution: color $p(z)$ to black, color $p(p(z))$ to red, and right rotation at $p(p(z))$.

† no problem with black-height.

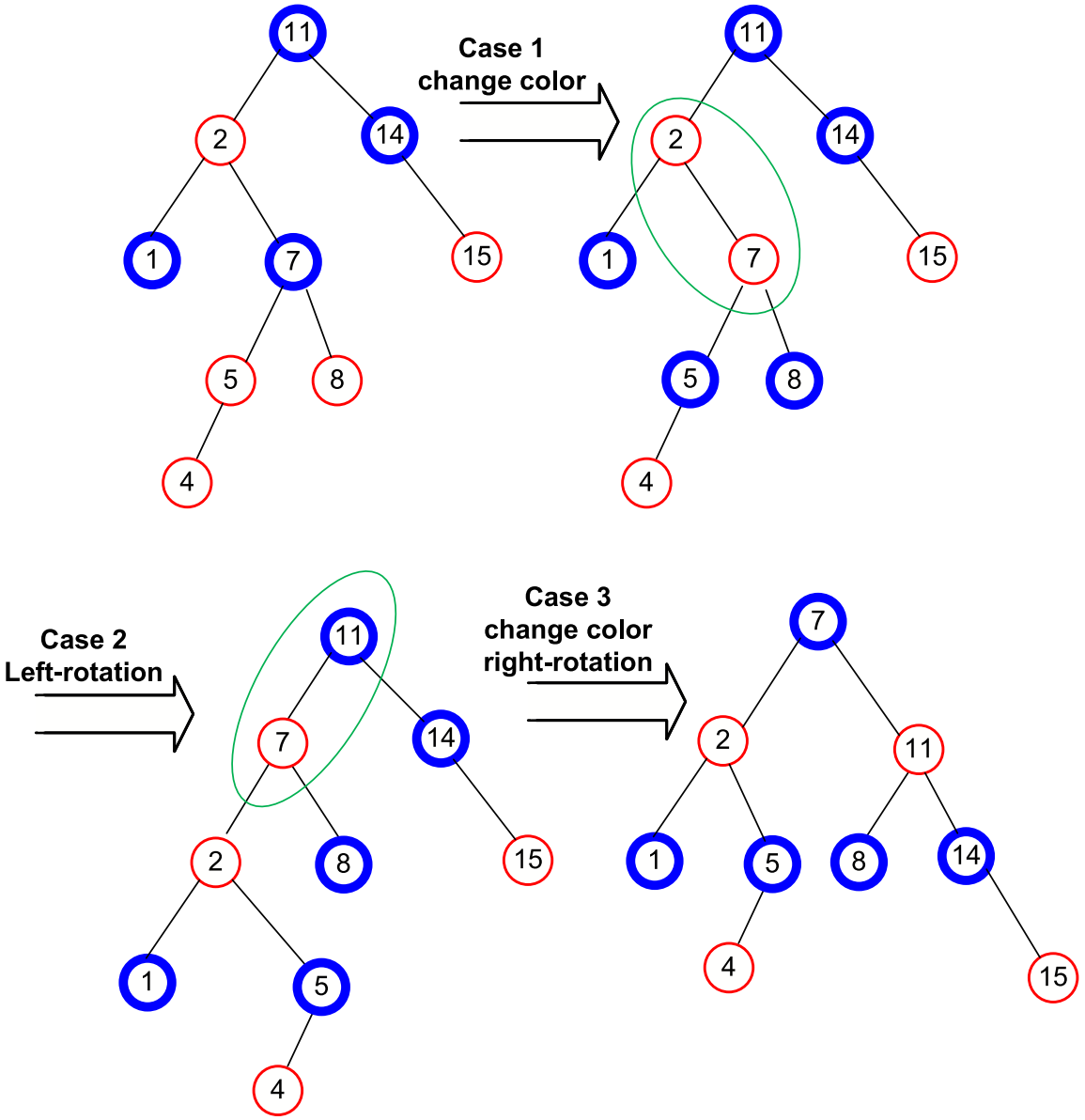
† no problem with color.

† problem solved.

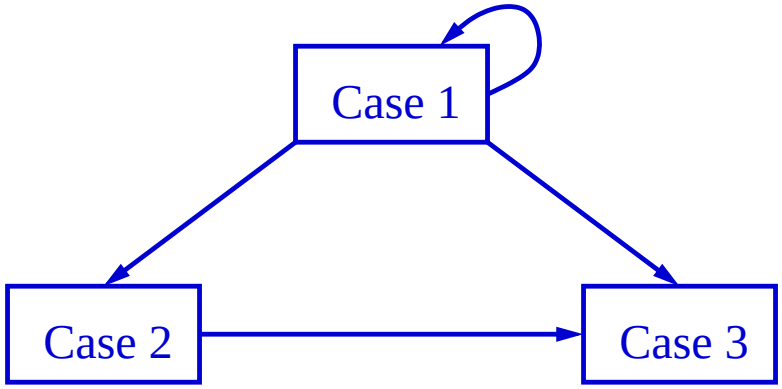
3B: y is the left child of $p(p(z))$
z is the right child of $p(z)$.

Solution is similar (left-rotation).

Example: insertion



- Insertion complexity:
  - † case 1: change colors for three nodes.
  - † case 2: one rotation.
  - † case 3: change colors for two nodes, one rotation.



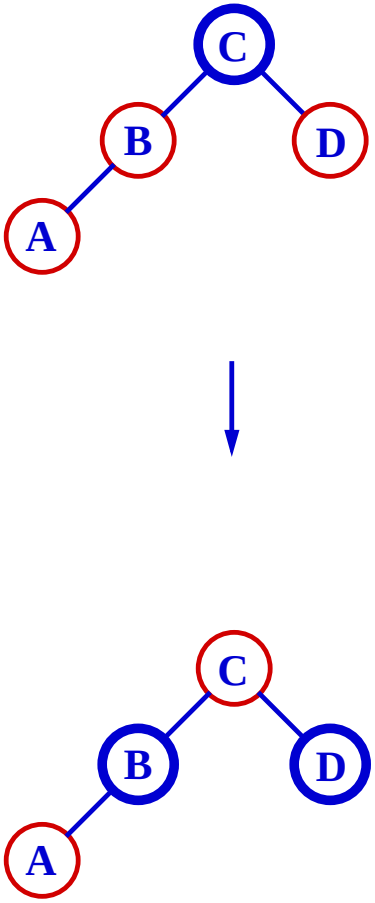
- Binary search tree insertion: $O(\log_2 n)$.
- Red-black tree updates:
  - † $O(\log_2 n)$ color changes.
  - † 2 rotations ($O(1)$ each).
- $O(\log_2 n)$ time with $O(1)$ rotations.

RB_Insert(T, x)

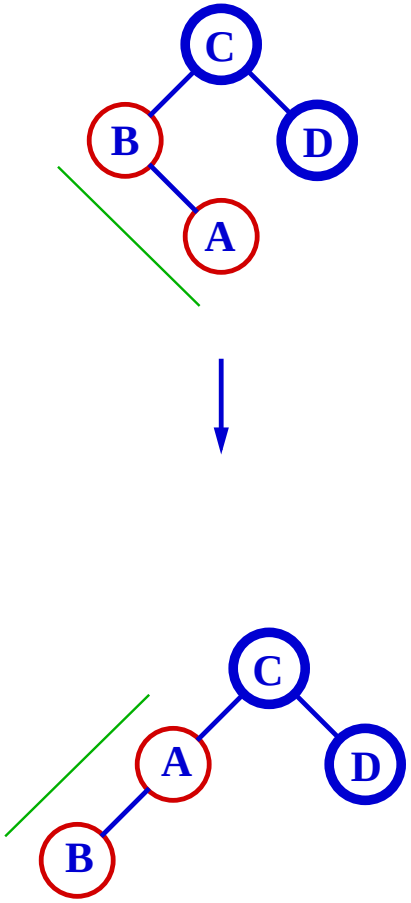
```
1   binary_search_tree_insert(T, x)
2   color[x] ← red
3   while color[p[x]] = red do
4       if p[x] = left[p[p[x]]]
5           then y ← right[p[p[x]]]
6                if color[y] = red
7                    then color[p[x]] ← black                ▷ Case 1
8                         color[y] ← black                   ▷ Case 1
9                         color[p[p[x]]] ← red               ▷ Case 1
10                        x ← p[p[x]]                        ▷ Case 1
11                   else if x = right[p[x]]
12                        then x ← p[x]                      ▷ Case 2
13                             left_rotation(T, x)           ▷ Case 2
14                        color[p[x]] ← black                ▷ Case 3
15                        color[p[p[x]]] ← red               ▷ Case 3
16                        right_rotation(T, p[p[x]])         ▷ Case 3
17          else {same as then clause with "left" and "right" exchanged}
18  end while
19 color[root[T]] ← black
```

Insertion Summary

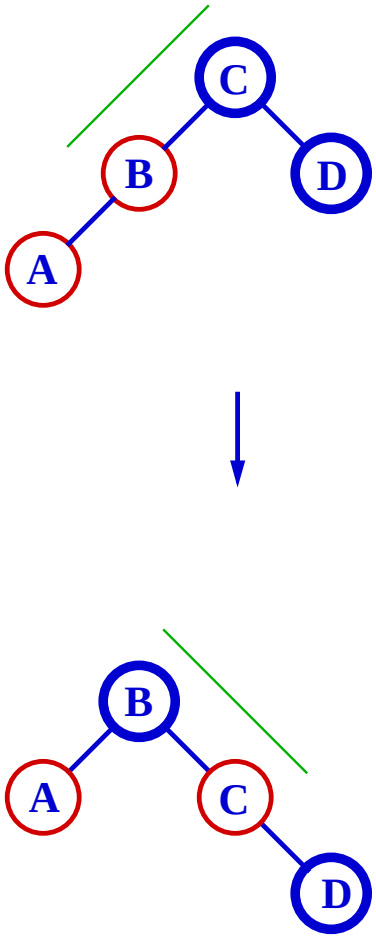
Case 1



Case 2

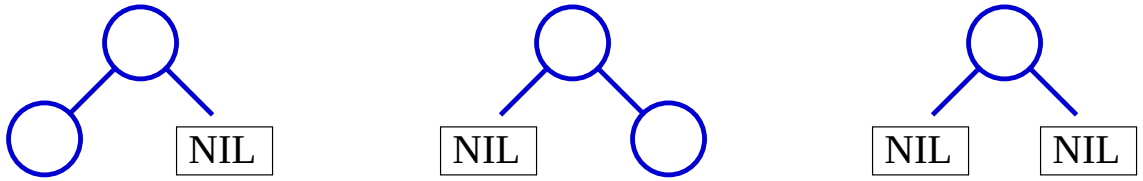


Case 3

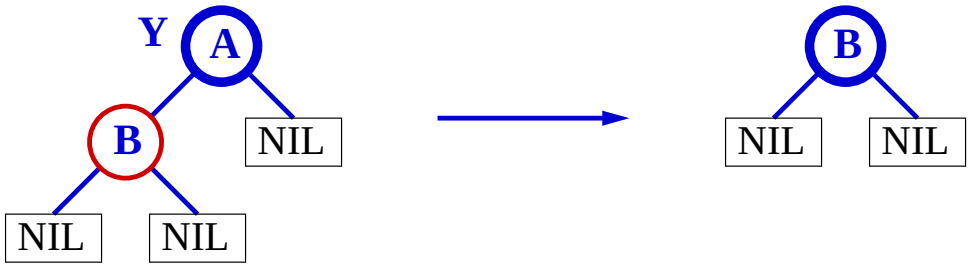


Deletion in red-black trees

- Use binary search tree deletion to determine the node to be deleted.
- Let this node be y, then either $left(y)$ or $right(y)$ must be NIL.

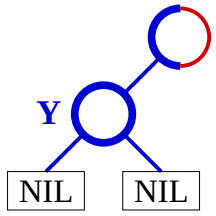


- If y is red, simply delete it. The resulting tree is still a red-black tree.
- If y is black and either $left(y)$ or $right(y)$ is an internal node.
  - † y's internal child must be red. (why?)
  - † color y's internal child to black and then delete y.

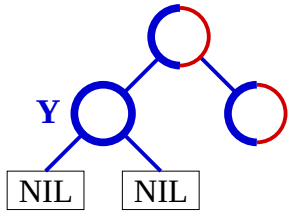


- If y is black and both of its children are external nodes.

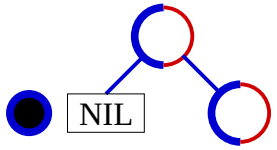
  † Let $y = left(p(y))$. ($y = right(p(y))$ can be handled similarly.)



  † $p(y)$'s right child must be an internal node. (why? since $bh(p(y)) = 2$.)



  † We delete y and add one more black to $p(y)$'s new child.



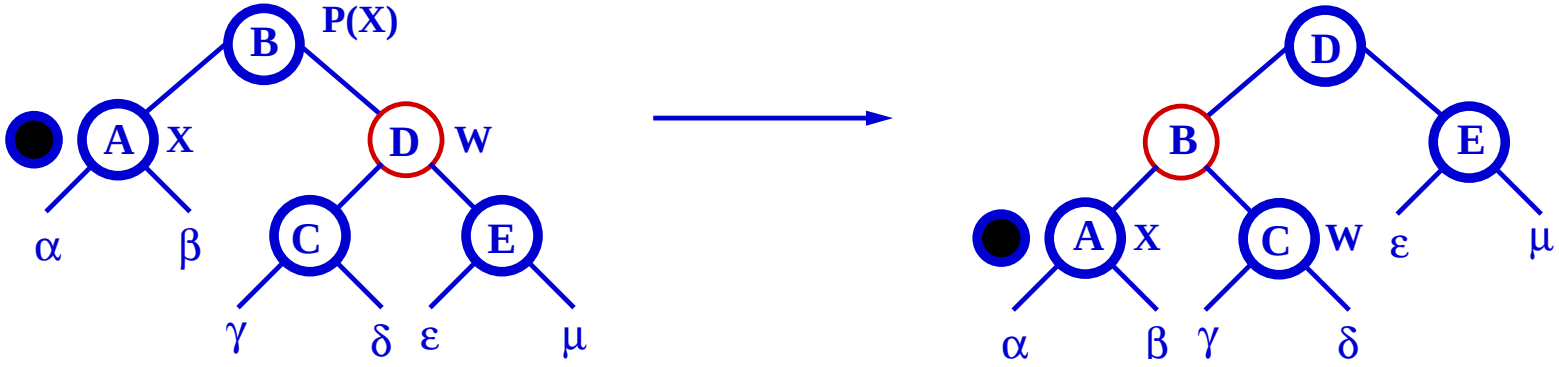
  – if we count this extra black, then black-height is still correct.
  – we have to fix this problem of an extra black.

How to fix this problem?

- Let the node with extra black be x.
- Since we assume that $y = left(p(y))$ before we delete y, $x = left(p(x))$ (after we delete y).
- Let $w = right(p(x)$, w is an internal node. (why?)

- Case 1: w is red.

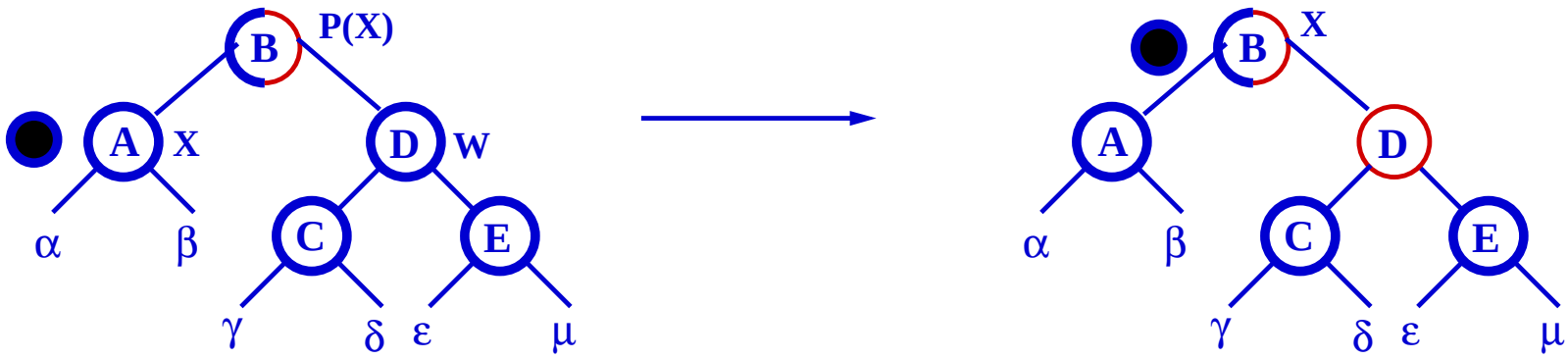
  † $p(x) = p(w)$ is black, and $left(w)$ and $right(w)$ are black.



  † Solution: exchange colors for w and $p(w)$, left rotation at $p(w)$.

  † Case 1 is transformed to case 2, 3, or 4. (w is black).

- Case 2: w is black, both $left(w)$ and $right(w)$ are black.



† Solution: color w to red, move the extra black to $p(x)$.

† We can color w to red since its two children are all black.

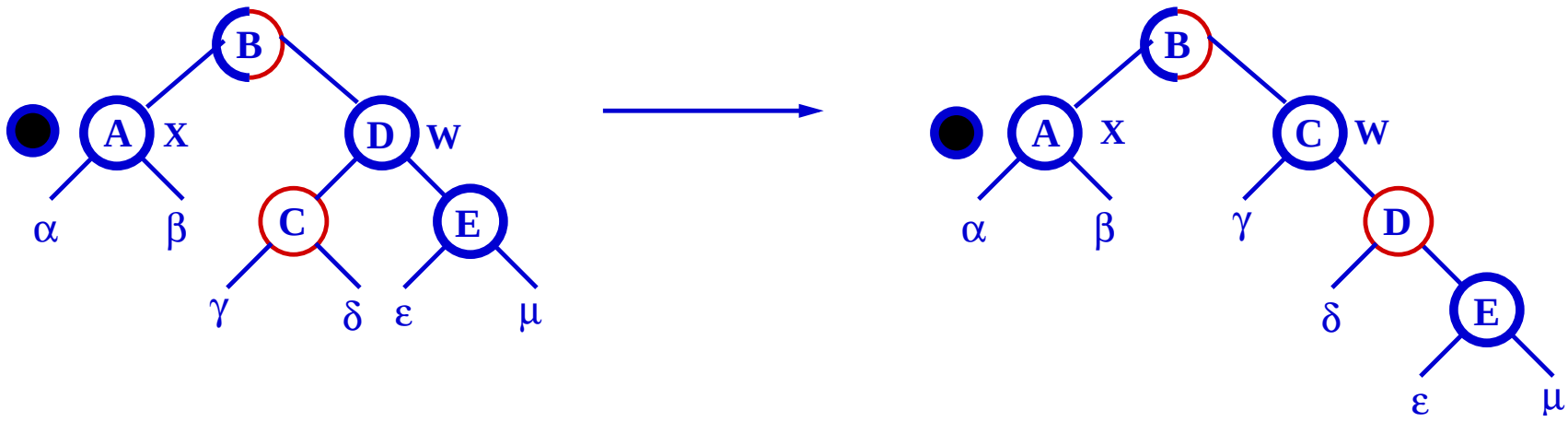
† we can move the extra black up since we take one black from each of $p(x)$'s children.

† If $p(x)$ is red, we just change it to black and solve the problem.

† If $p(x)$ is black, $p(x)$ becomes the new x since it now has an extra black. However the node with extra black is now one level up.

† If we reach the root, we can simply remove the extra black.

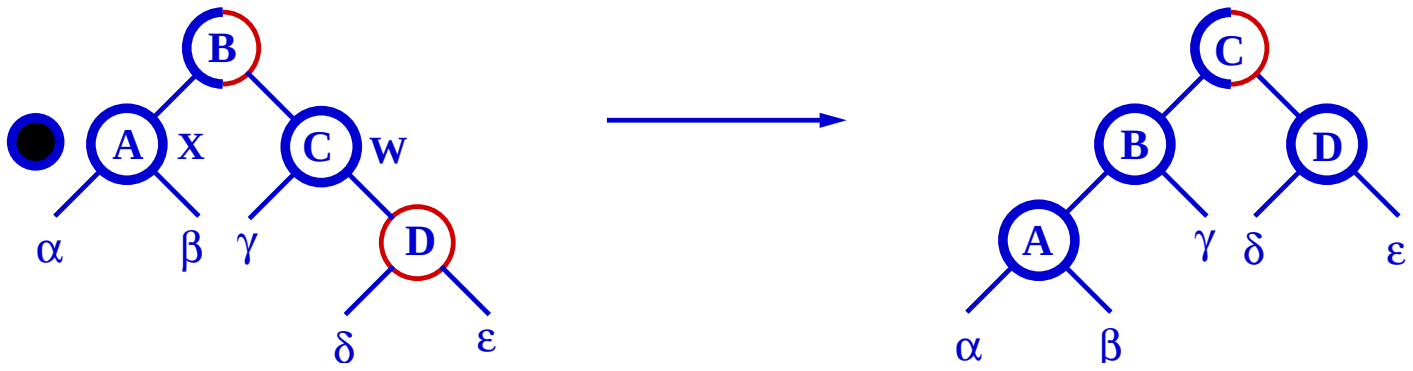
• Case 3: w is black, $right(w)$ is black, $left(w)$ is red.



† Solution: exchange the colors of w and $left(w)$, right rotation at w.

† Transform case 3 to case 4 ($right(w)$ is red).

- Case 4: w is black, $right(w)$ is red.

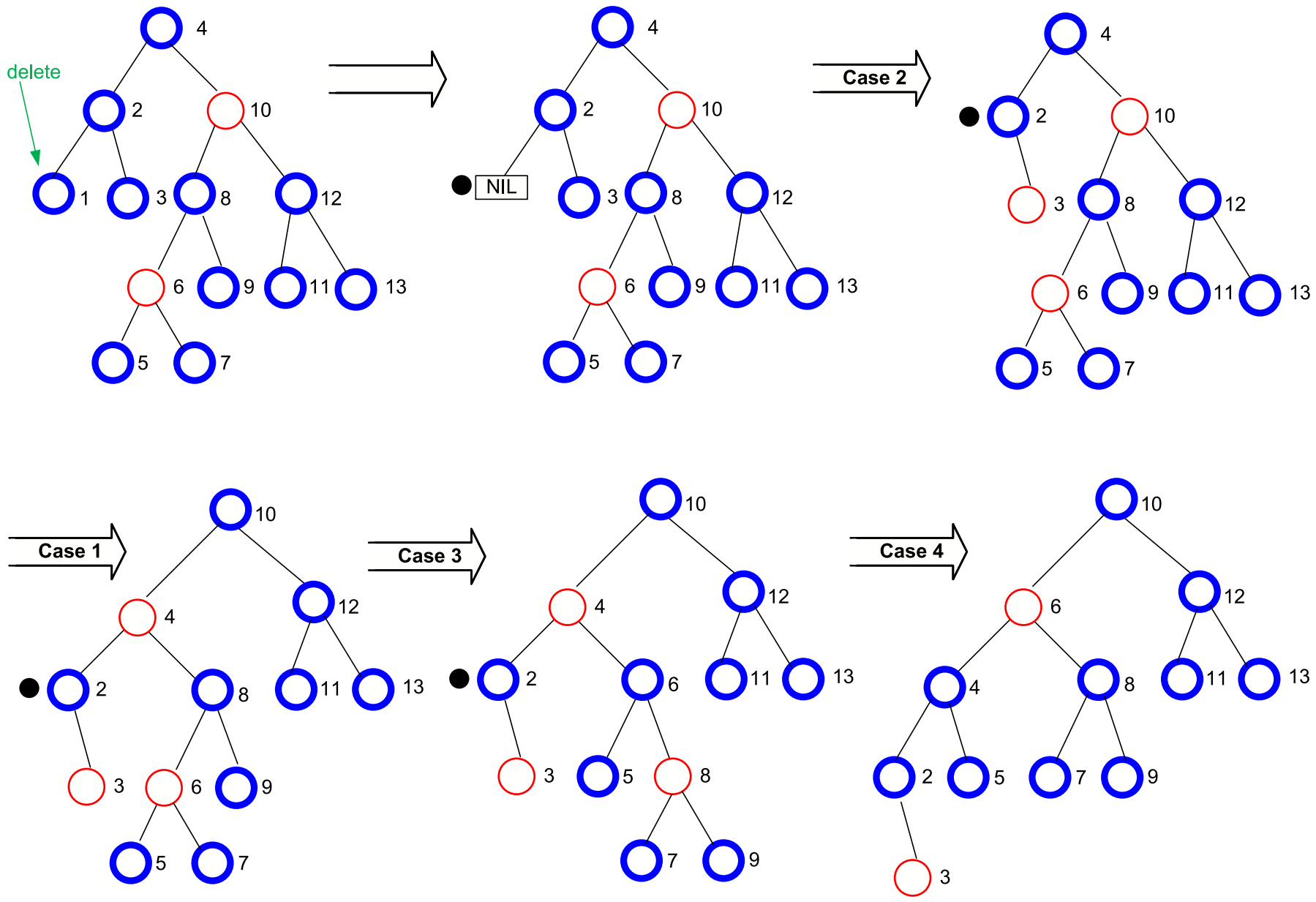


† Solution: exchange colors for w and $p(w)$, color $right(w)$ to black, left rotation at $p(w)$.

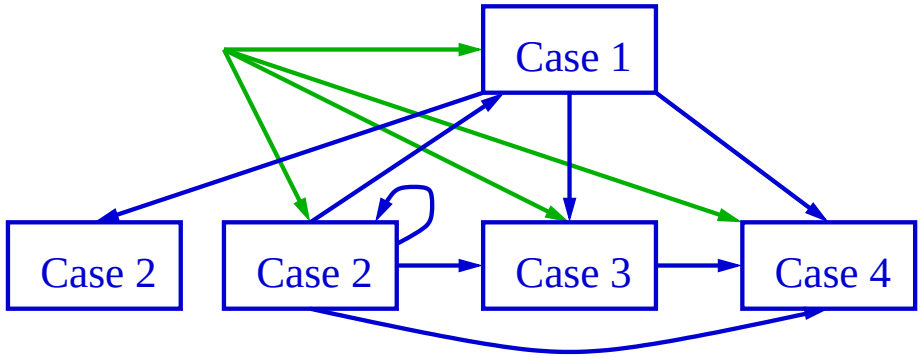
† Black color of w, now in node $p(x)$, replaced the extra black in x!

† Problem solved.

Example: deletion



- Deletion Complexity:
  - † case 1: change colors for two nodes, one rotation.
  - † case 2: change color for one or two nodes.
  - † case 3: change colors for two nodes, one rotation.
  - † case 4: change colors for three nodes, one rotation.



† Only case 2 can repeat

† Case 2 following case 1 cannot repeat.

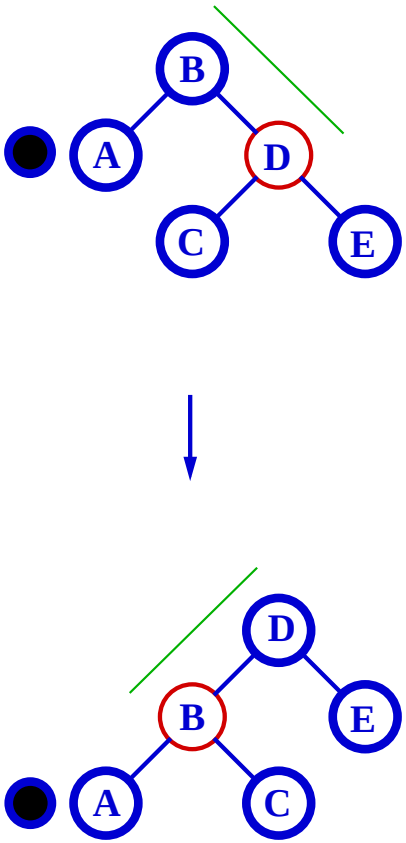
† At most three rotations: case 1, then case 3, then case 4.

† $O(\log_2 n)$ time.

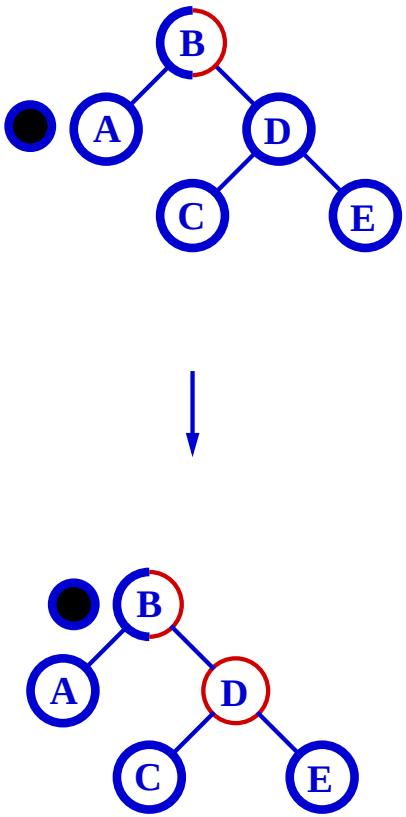
- $O(1)$ rotations, $O(\log_2 n)$ color changes.
- $O(\log_2 n)$ for binary search tree deletion.

Deletion Summary

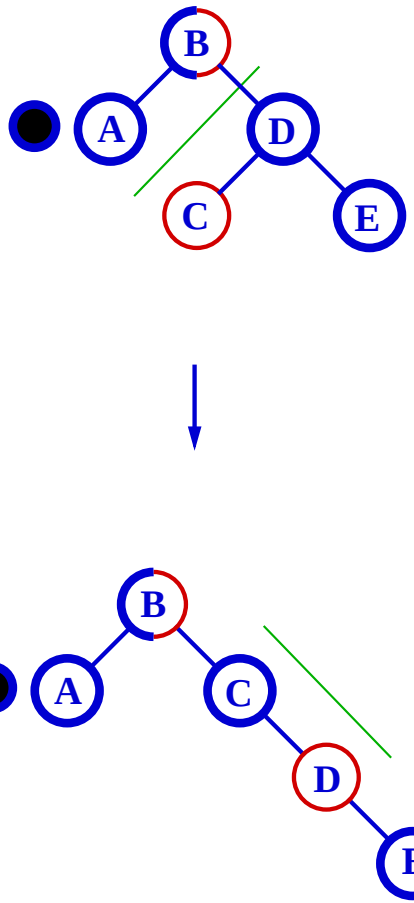
Case 1



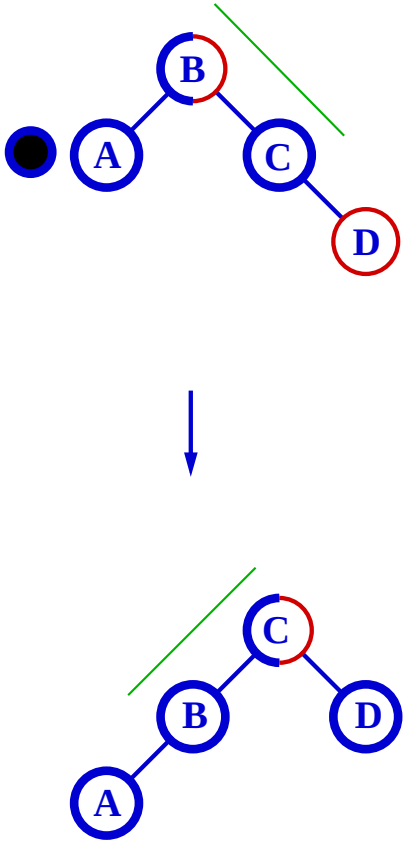
Case 2



case 3



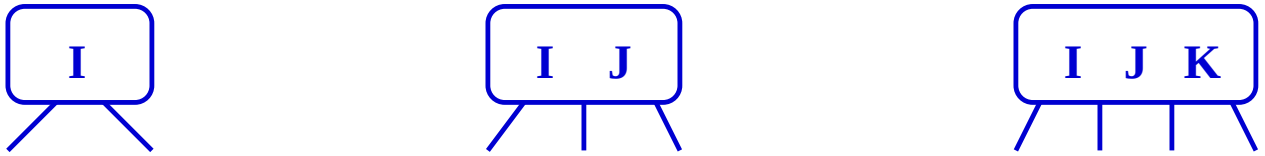
Case 4



B-trees and Red-Black trees

- B-tree: for $t \geq 2$, each node contains at least $t - 1$ keys and at most $2t - 1$ keys.
- Let $t = 2$.

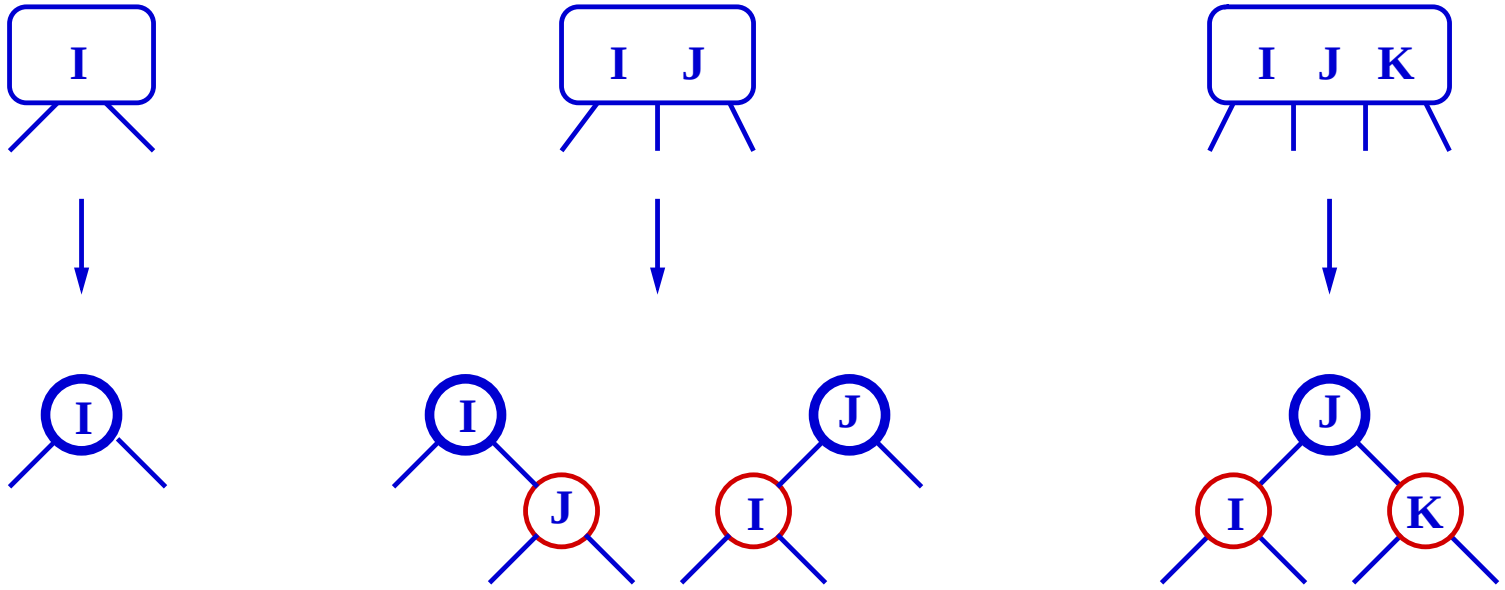
  † at least 1 key and at most 3 keys.



  † at least 2 children and at most four children.

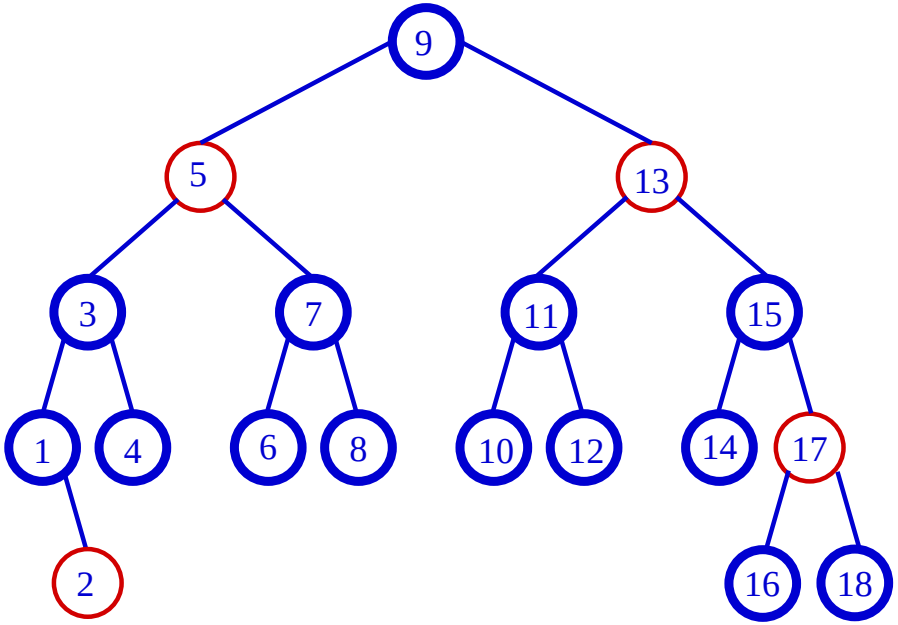
- This is called 2-3-4 trees.

- Red-black trees can be used to implement 2-3-4 trees.



Example.

Red-Black trees



2-3-4 trees