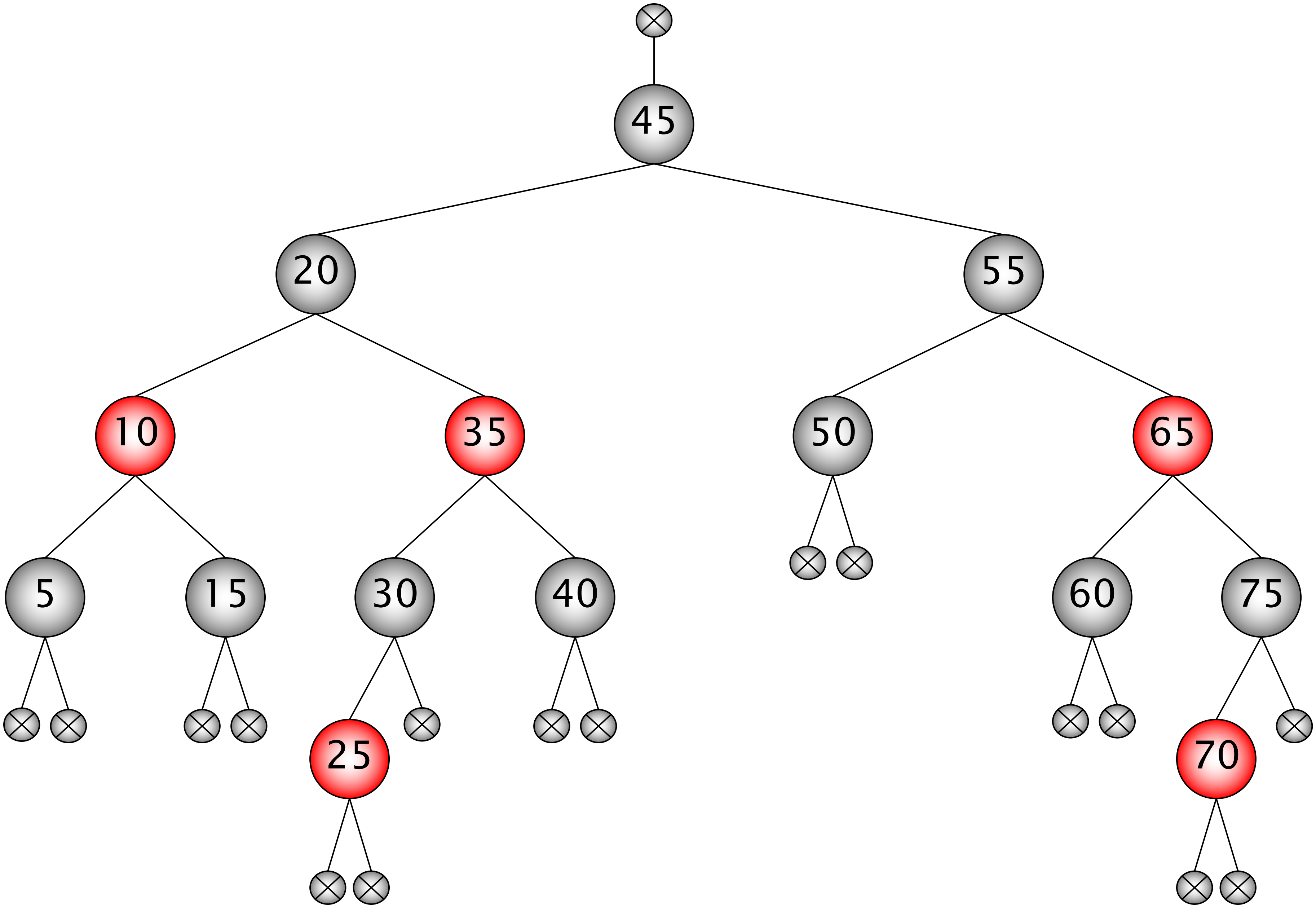
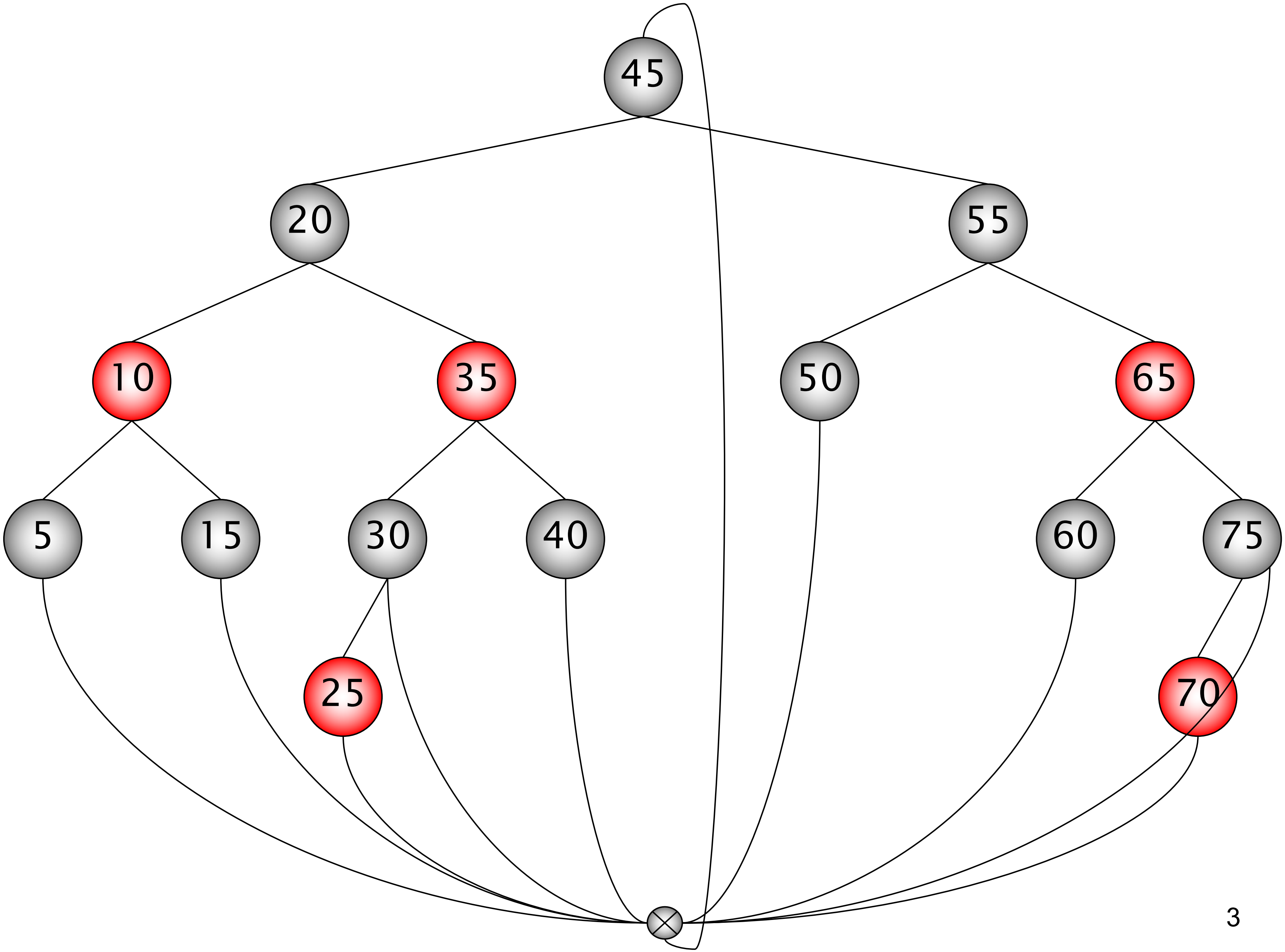
Red-Black Trees

- A *red-black tree* (RBT) is a type of self-balancing BST
  - Each node in an RBT is labeled as ***red*** or ***black***
- Operations on RBTs take $O(\log n)$ time in the worst case since the height of an RBT is at most $2\log_2(n+1)$
  - The height of an AVL tree is at most $1.44\log_2 n$
- However, a careful nonrecursive implementation can be done relatively effortlessly compared with AVL trees


45
20
55
10
35
50
65
5
15
30
40
60
75
25
70


45
20
55
10
35
50
65
5
15
30
40
60
75
25
70

Definition of Red-Black Trees

An RBT is a BST that satisfies the following *red-black criteria*:

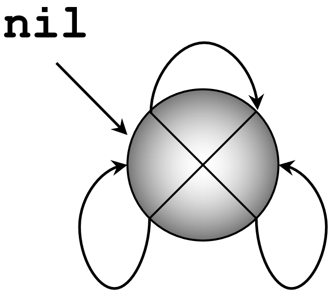
1. Every node is either ***red*** or ***black***
2. The root of the tree is always ***black***
3. All leaves are ***black***
4. If a node is ***red***, then its parent is ***black***
5. Any path from a node to any of its leaves contains the same amount of black nodes, called *black height*

Representation of Red-Black Trees

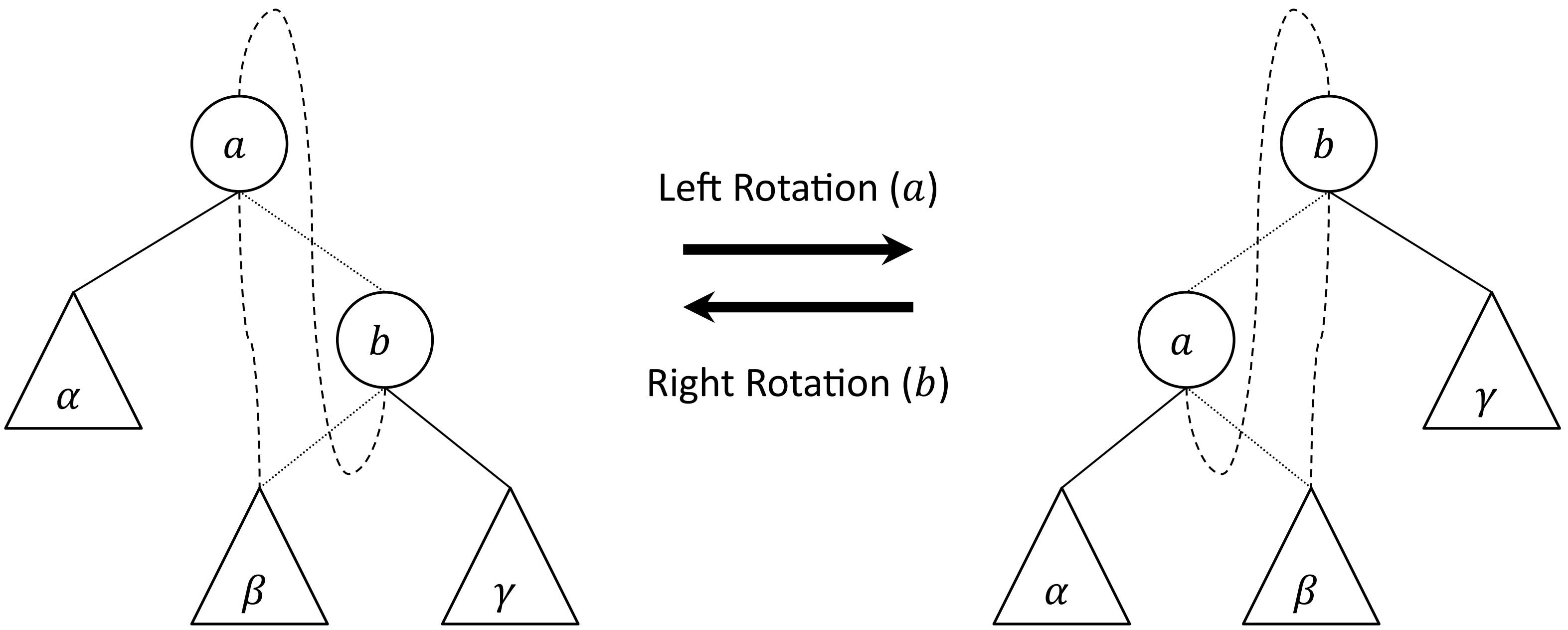
```
typedef struct Node * Ref;
struct  Node {
   int key;
   int color;
   Ref parent, left, right;
};
ref getNode(int key, int color, Ref nil) {
   p = new Node;
   p->key   = key;
   p->color = color;
   p->left  = p->right = p->parent = nil;
   return  p;
}
```

The Initial State of a Red-Black Tree

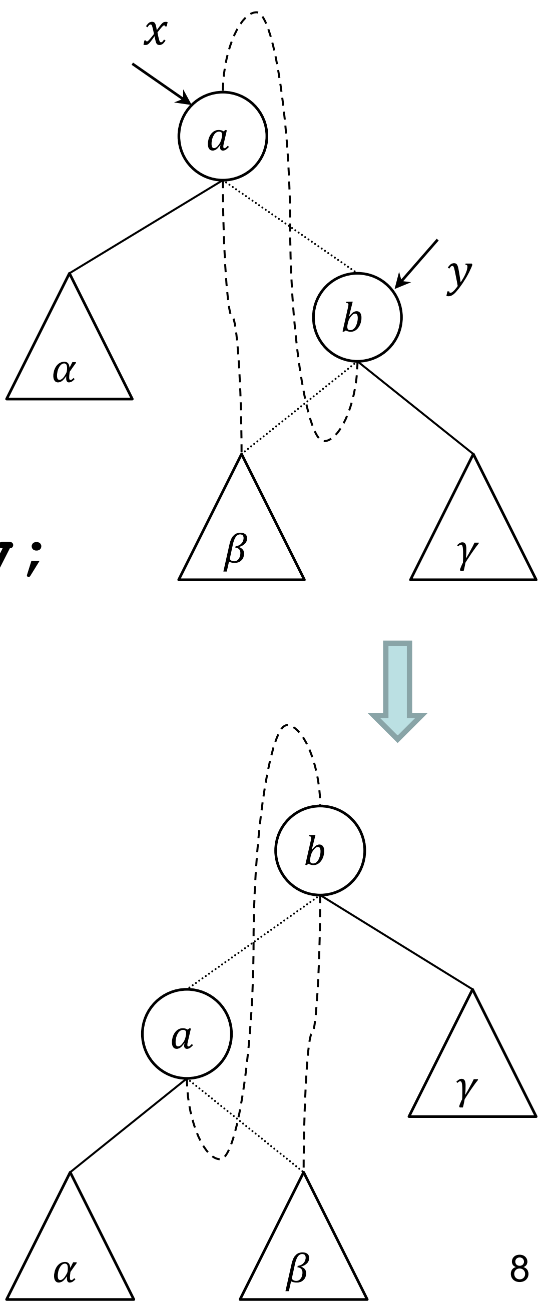
```
Ref nil, root;
…
nil = new Node;
nil->color = BLACK;
nil->key = -1;
nil->left =
nil->right =
nil->parent = nil;
…
root = nil;
```



Tree Rotation



```
leftRotate(Ref & root, Ref x) {
  y = x->right;
  x->right = y->left;
  if (y->left != nil)
    y->left->parent = x;
  y->parent = x->parent;
  if (x->parent == nil) root = y;
  else
    if (x == x->parent->left)
      x->parent->left = y;
    else
      x->parent->right = y;
  y->left = x;
  x->parent = y;
}
```



Search Operation

- An RBT is a BST so the search algorithm for an RBT is the same as the search algorithm for a BST

Insertion Operation

- The new node, as usual, is placed as a leaf in the tree
- The node must be colored ***red***
  - If the parent is ***black***: The red-black criteria are maintained
  - If the parent is ***red***: The operation causes a violation of the criterion “no consecutive red nodes”

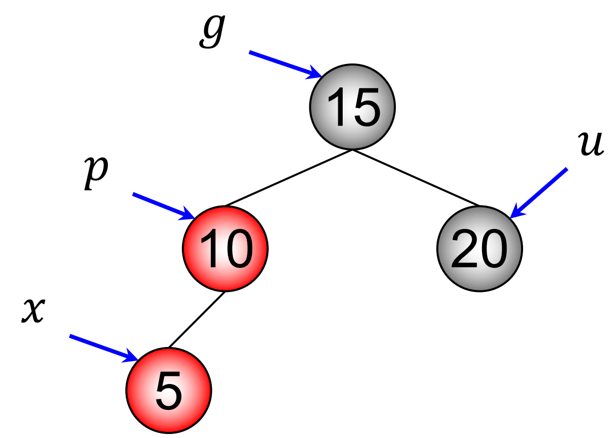
    ➡️ Rebalancing the tree is needed

Insertion Operation: Pseudo-code

```
void    RBT_Insertion(Ref & root, int key) {
  x = getNode(key, RED, nil);
  BST_Insert(root, x);
  Insertion_FixUp(root, x);
}
```

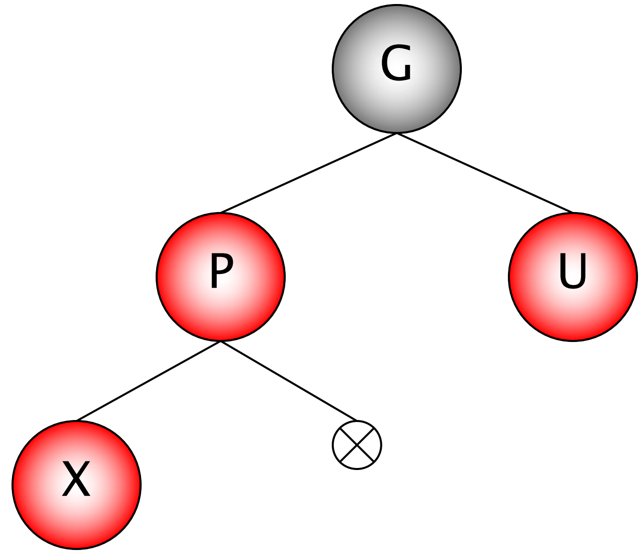
```
BST_Insert(Ref & root, ref x) {
  y = nil;
  z = root;
  while (z != nil) {
    y = z;
    if (x->key < z->key)       z = z->left;
    else if (x->key > z->key)  z = z->right;
    else                       return;
  }
  x->parent = y;
  if (y == nil)   root = x;
  else
    if (x->key < y->key)  y->left = x;
    else                  y->right = x;
}
```

Some Conventions

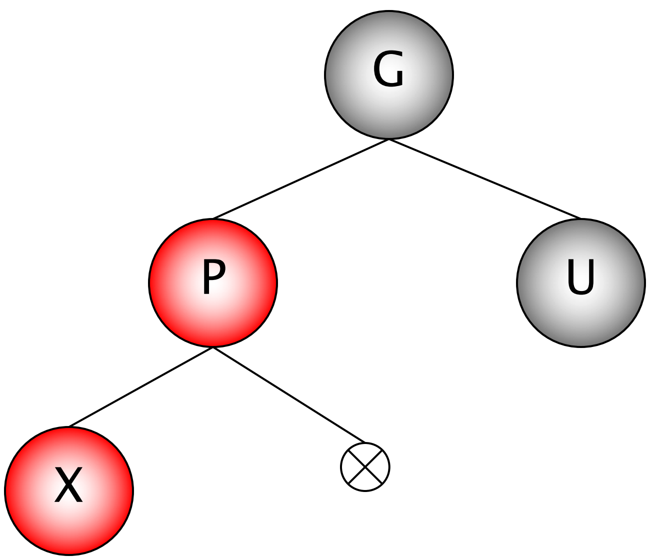


- x: The pointer designated to point to the newly added leaf
- p: The pointer designated to point to the *parent* of the node pointed to by x
- u: The pointer designated to point to the *uncle* of the node pointed to by x
- g: The pointer designated to point to the *grandparent* of the node pointed to by x

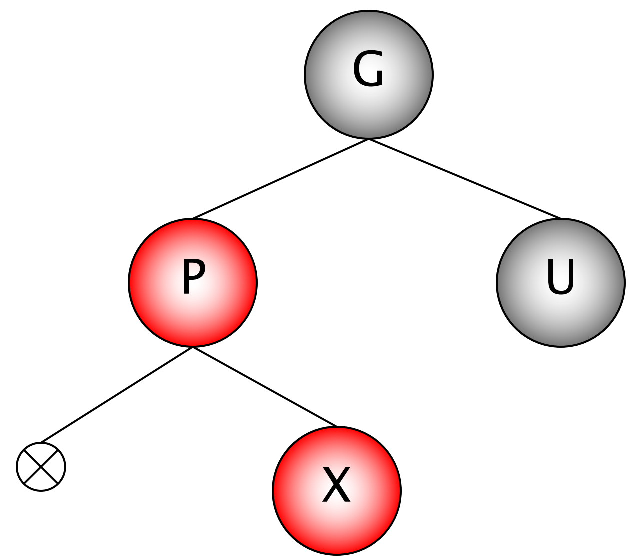
Imbalance Cases



Case 1



Case 2

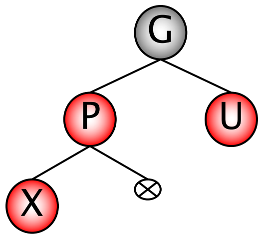


Case 3

The Strategies for Rebalancing an RBT

- Case 1

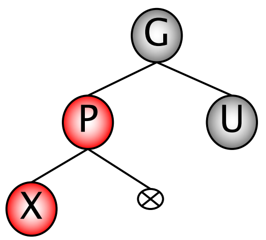
  → Reverse the color of three nodes: u, p, and g



- Case 2

  → Reverse the color of two nodes: p and g

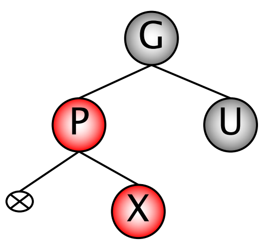
  → Run a rotation at g



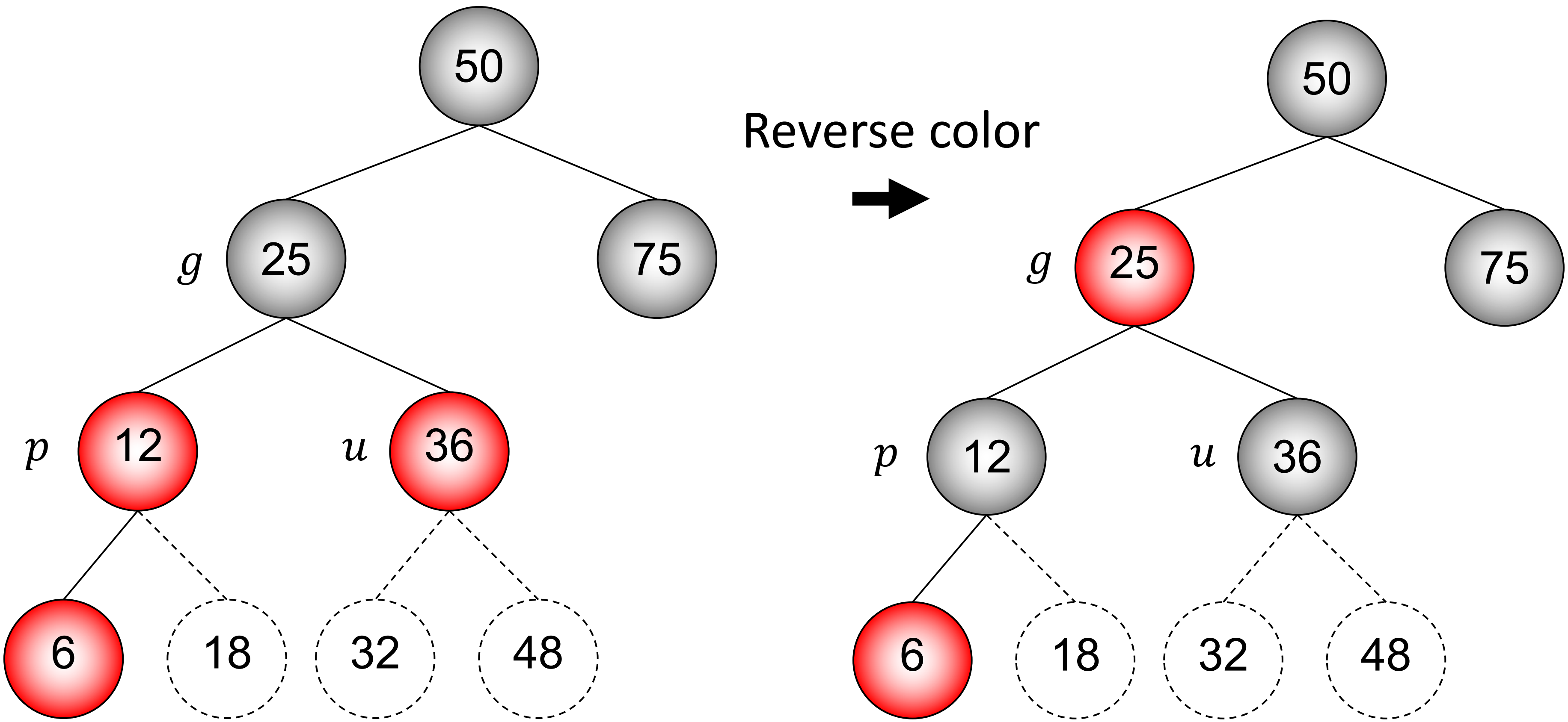
- Case 3

  → Run a rotation at parent p

  → Go to Case 2

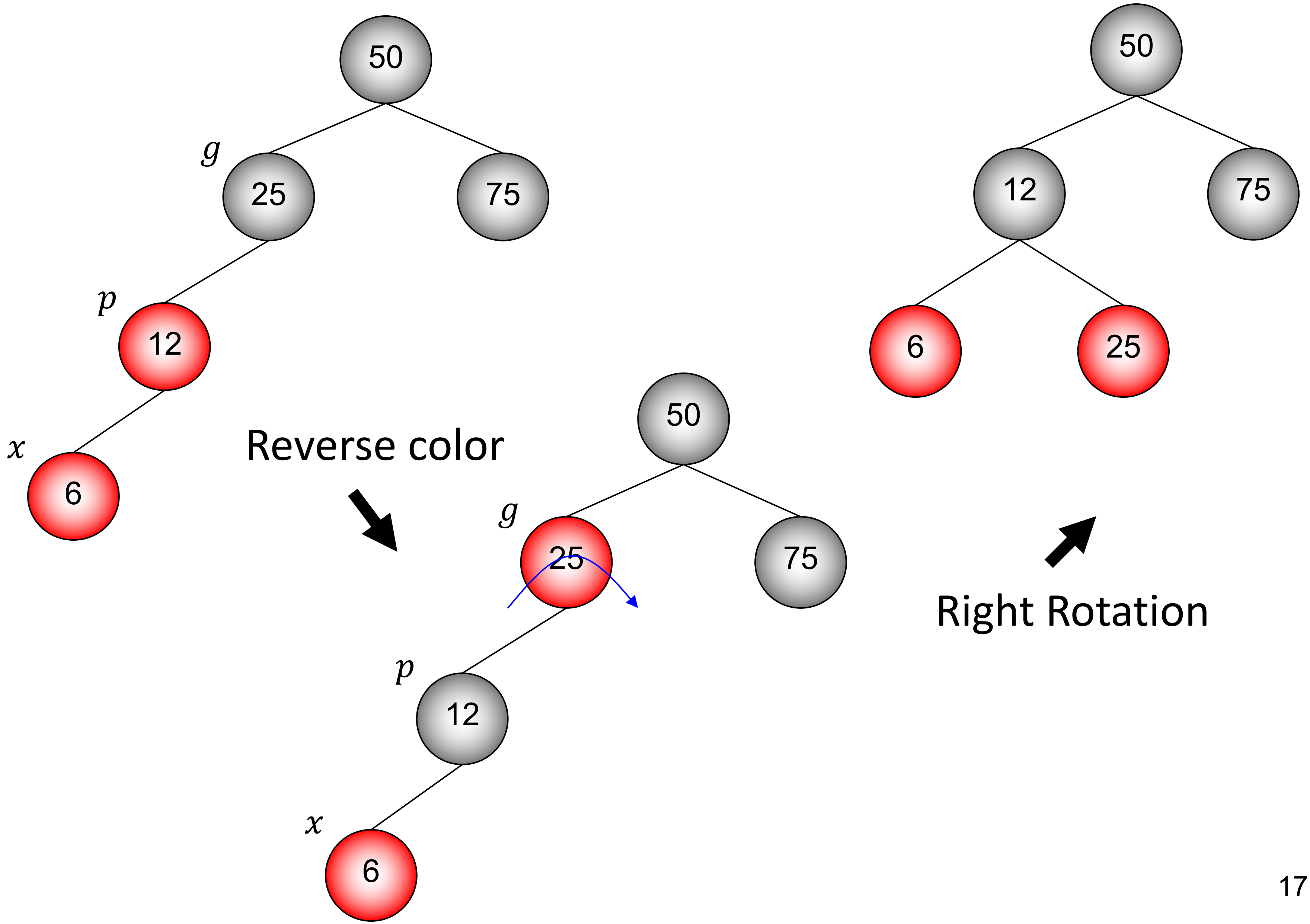


Case 1



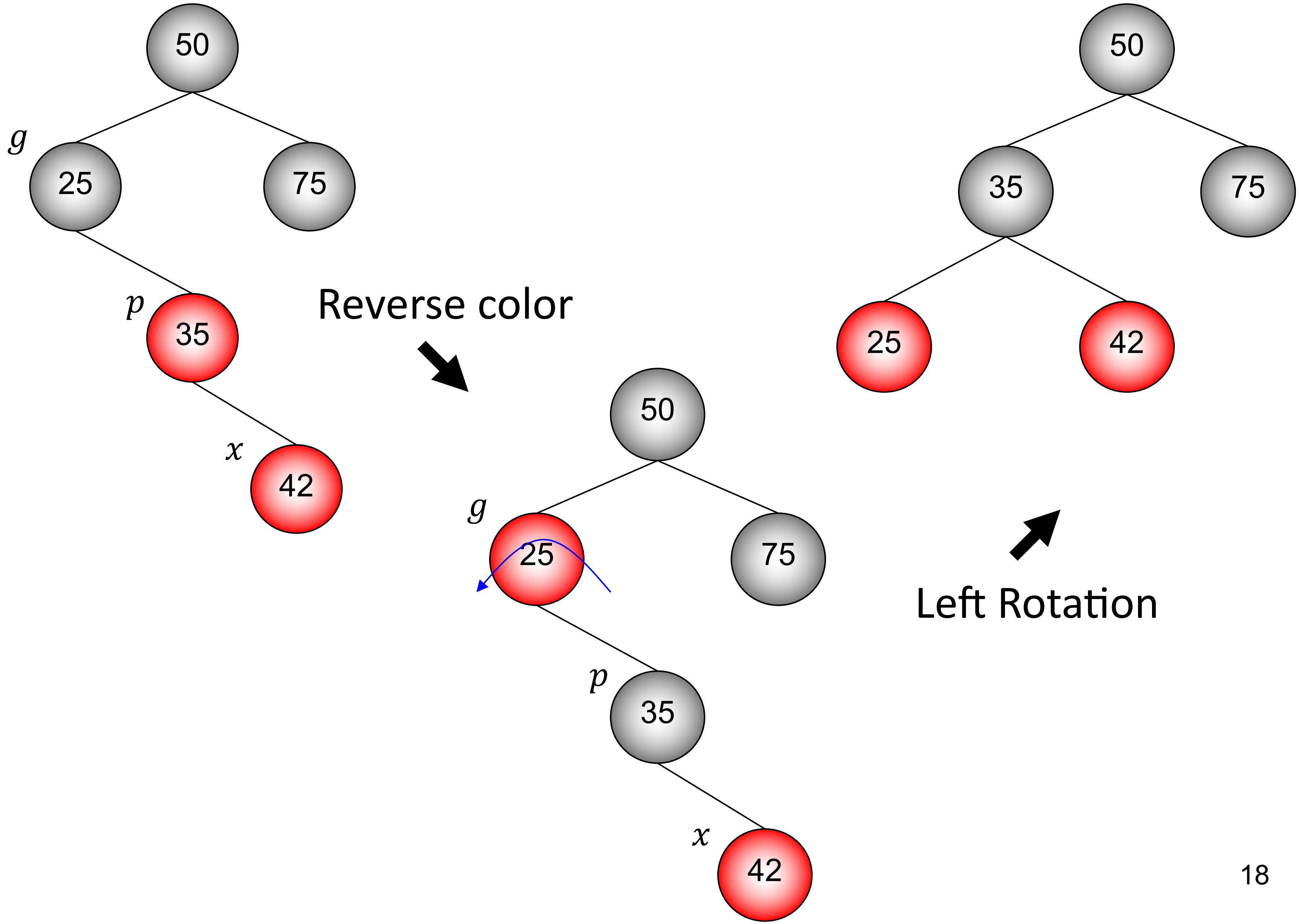
50
g
25
75
p
12
u
36
6
18
32
48
Reverse color
50
g
25
75
p
12
u
36
6
18
32
48

Case 2

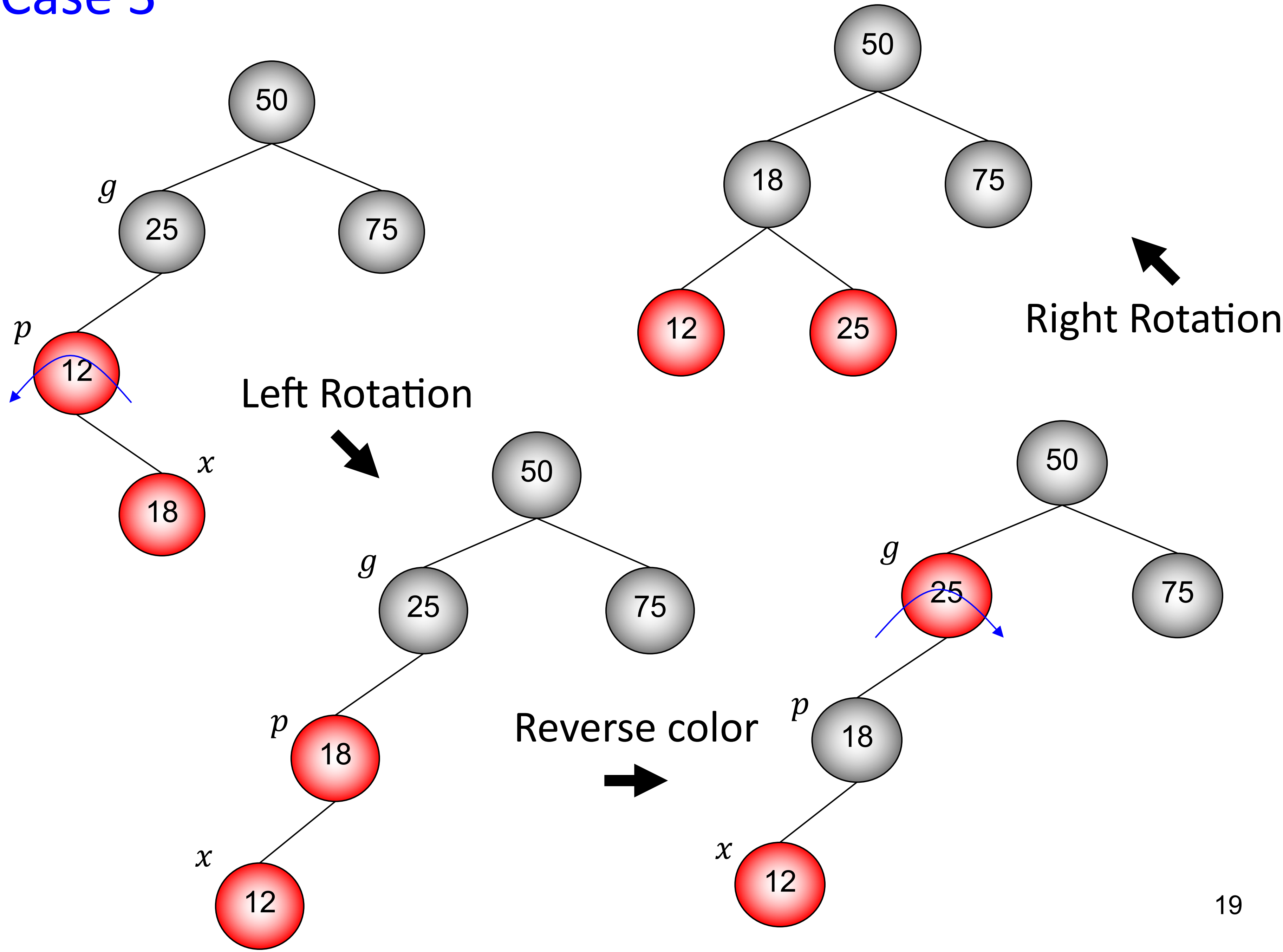


50
g
25
75
p
12
x
6
Reverse color
50
g
25
75
p
12
x
6
Right Rotation
50
12
75
6
25

Case 2

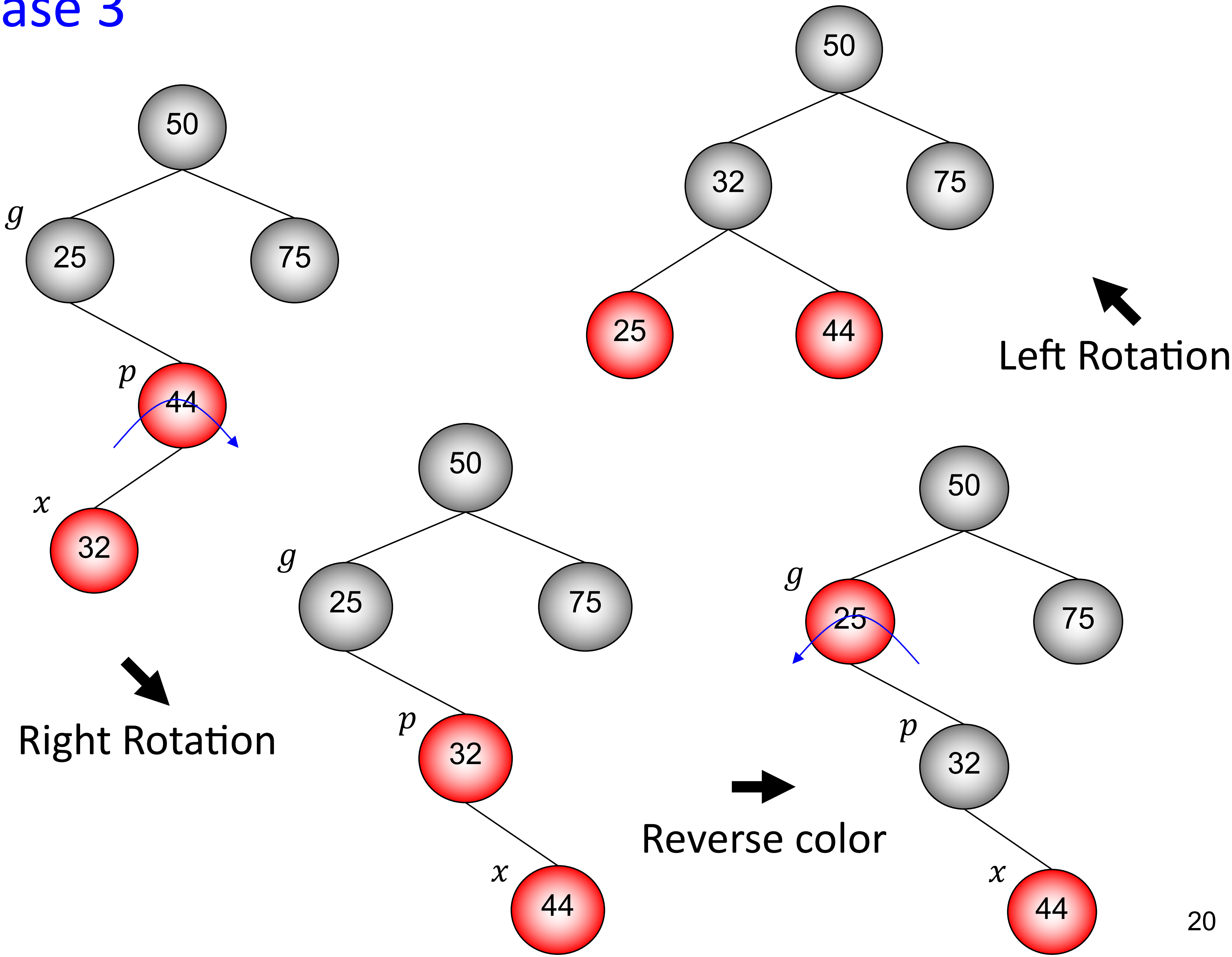

50
g
25
75
p
35
x
42
Reverse color
50
g
25
75
p
35
x
42
50
35
75
25
42
Left Rotation

Case 3



50
25
75
12
18
g
p
x
Left Rotation
Reverse color
Right Rotation

Case 3


50
g
25
75
p
44
x
32
Right Rotation
50
g
25
75
p
32
x
44
Reverse color
50
g
25
75
p
32
x
44
Left Rotation
50
32
75
25
44

Insertion Operation: Pseudo-code

```
Insertion_FixUp(Ref & root, Ref x) {
  while (x->parent->color == RED)
    if (x->parent == x->parent->parent->left)
      ins_leftAdjust(root, x);
    else
      ins_rightAdjust(root, x);
  root->color = BLACK;
}
```

```
ins_leftAdjust(Ref & root, Ref & x) {
  u = x->parent->parent->right;
  if (u->color == RED) {
    x->parent->color = u->color = BLACK;
    x->parent->parent->color = RED;
    x = x->parent->parent;
  }
  else {
    if (x == x->parent->right) {
      x = x->parent; leftRotate(root, x);
    }
    x->parent->color = BLACK;
    x->parent->parent->color = RED;
    rightRotate(root, x->parent->parent);
  }
}
```

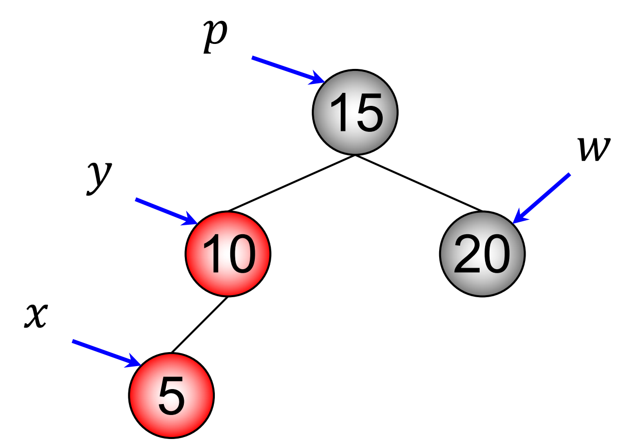
Deletion Operation

- Let r be the node to be deleted. One of the following three cases will arise:
  1. r is a leaf
  2. r has one nonempty subtree
  3. r has both nonempty subtrees
- Let's consider the third case:
  - Find the predecessor or successor, say s, of r
  - Copy the data of s into r *except the color*

  ➡️ Now, the node to be deleted is s

```
RBT_Deletion(Ref & root, int k) {
  z = searchTree(root, k);
  if (z == nil) return;
  y = (z->left == nil) || (z->right == nil) ?
            z : Predecessor(root, z);
  x = (y->left == nil) ? y->right : y->left;
  x->parent = y->parent;
  if (y->parent == nil)   root = x;
  else
    if (y == y->parent->left) y->parent->left = x;
    else                      y->parent->right= x;
  if (y != z)  z->key = y->key;
  // if (y->color == BLACK) Del_FixUp(root, x);
  delete y;
}
```

Some Conventions



- y: The pointer designated to point to the node to be deleted
- x: The pointer designated to point to the child of the node pointed to by y
- p: The pointer designated to point to the parent of the node pointed to by y
- w: The pointer designated to point to the sibling of the node pointed to by y

What'll Happen After Deleting a Node?

- If y was red, the red-black criteria still hold because:
  - No black heights in the tree have changed
  - No red nodes have been made adjacent
- If y was black:
  - The number of black nodes on every path passing through node y is decreased by 1
  - The "no two consecutive red nodes" criterion is also violated if both nodes, p and x, are red

➡️ Rebalancing the tree is needed

```
RBT_Deletion(Ref & root, int k) {
  z = searchTree(root, k);
  if (z == nil) return;
  y = (z->left == nil) || (z->right == nil) ?
            z : Predecessor(root, z);
  x = (y->left == nil) ? y->right : y->left;
  x->parent = y->parent;
  if (y->parent == nil)   root = x;
  else
    if (y == y->parent->left) y->parent->left = x;
    else                      y->parent->right= x;
  if (y != z)  z->key = y->key;
  if (y->color == BLACK) Del_FixUp(root, x);
  delete y;
}
```

Black Token

- *Black token* is an abstract concept
  - It's used to explain the meaning of the rebalancing process
- If y is black then x will receive black token
  - If x is black then x is called *doubly-black node*
  - If x is red then x is called *red-black node*
- What is the role of black token?
  - After deleting y, all black heights passing through x are one unit shorter than the others
  - The occurrence of black token logically implies that every black height passing through x is supplemented by one black

Black Token

- The black token tends *to be pushed toward the root* of the tree
- It could (i) be neutralized on the way to the root or (ii) finally be attached to the root, thus making the root as a doubly-black node

➡️ In both cases, the rebalancing process stops immediately

The Strategies for Rebalancing an RBT

- There are 4 cases that may happen when a black node is deleted
- We focus our attention on node x, the one with black token

Case 1

- If x is a red-black node

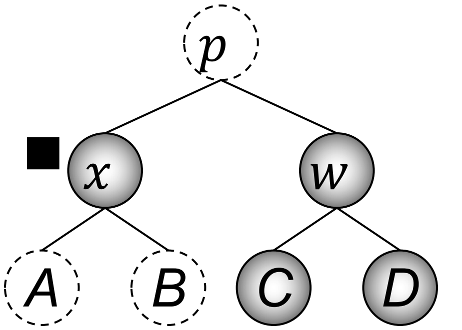
→ The node will be neutralized by coloring it black

- If x is the root

→ The black token will be eliminated

Case 2

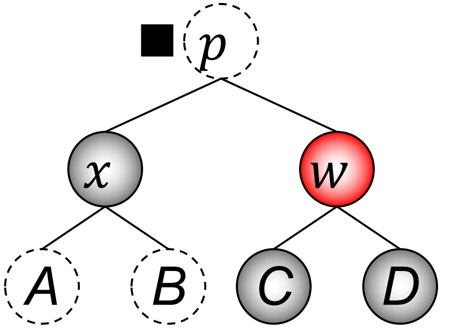
- Node x is doubly-black
- Its sibling w is black
- Both of w's children are black



Solution:

→ Reverse the color of sibling w

→ Attach black token to parent p



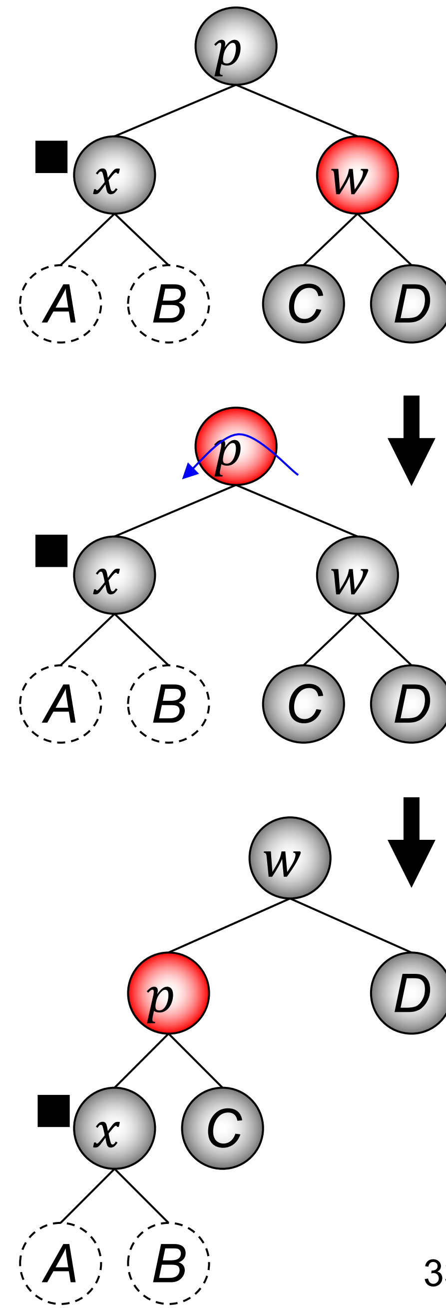
Case 3

- Node x is doubly-black
- Its sibling w is red

Solution:

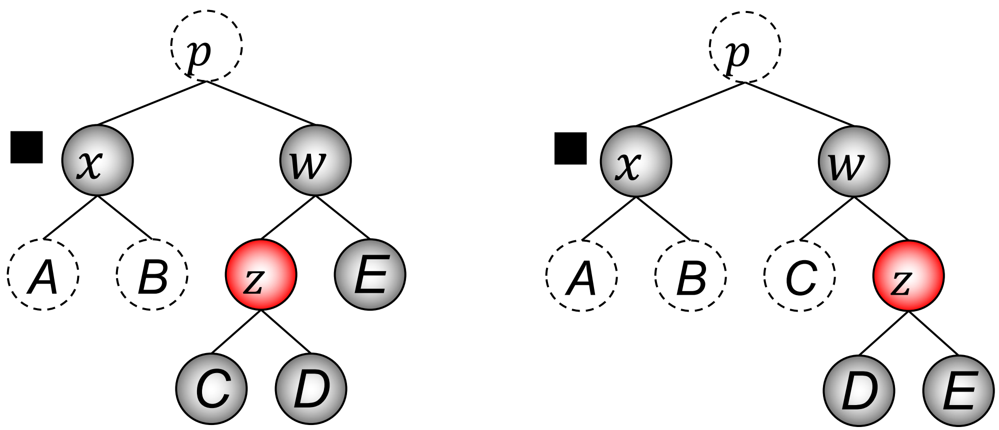
→ Reverse the color of parent p and sibling w

→ Perform a rotation at parent p



Case 4

- Node x is doubly-black
- Its sibling w is black
- At least one of w's children is red



➡️ The solution depends on the w's child node which is the external grandchild of grandparent p

Case 4.a

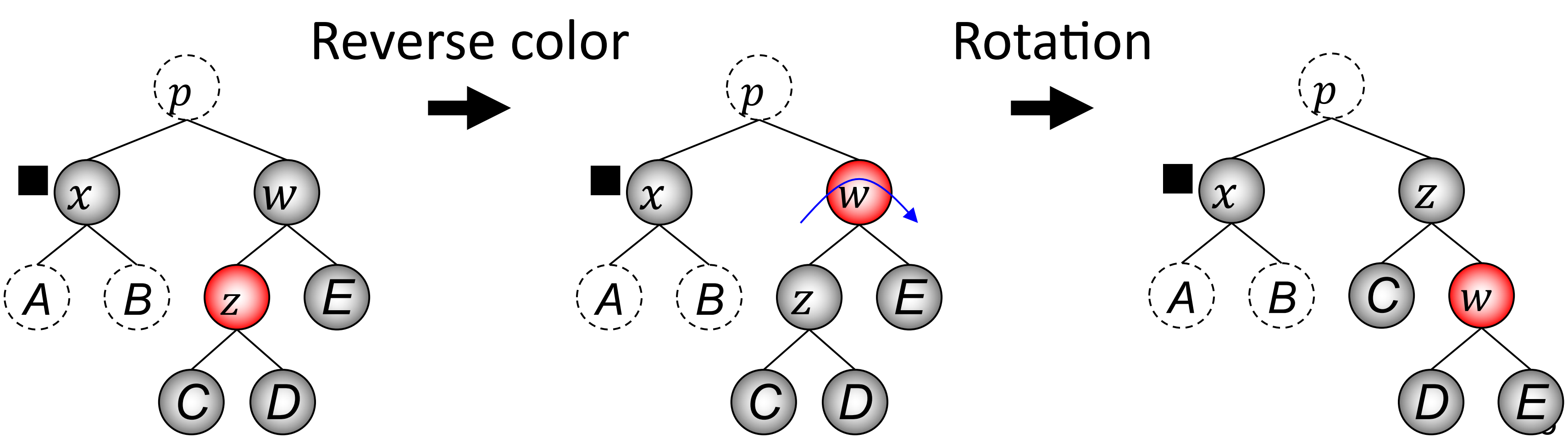
- The external grandchild is black

Solution:

→ Reverse the color of internal grandchild z and sibling w

→ Perform a rotation at the sibling w

→ Go to Case 4.b



Case 4.b

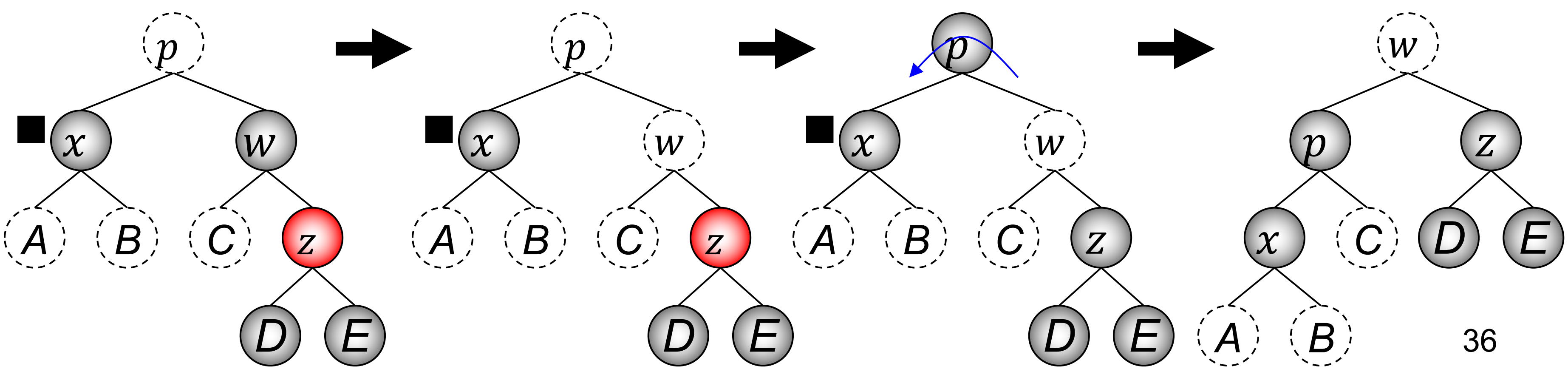
- The external grandchild is red

Solution:

→ The sibling w inherits the color of its parent p

→ The color of grandparent p and external grandchild z is set to black

→ Perform a rotation at grandparent p



```
Del_FixUp(Ref root, Ref x) {
  while (x->color == BLACK && x != root)
    if (x == x->parent->left)
      del_leftAdjust(root, x);
    else
      del_rightAdjust(root, x);
  x->color = BLACK;
}
del_leftAdjust(Ref & root, Ref & x) {
  w = x->parent->right;
  if (w->color == RED) {
    w->color          = BLACK;
    x->parent->color = RED;
    leftRotate(root, x->parent);
    w = x->parent->right;
  }
```

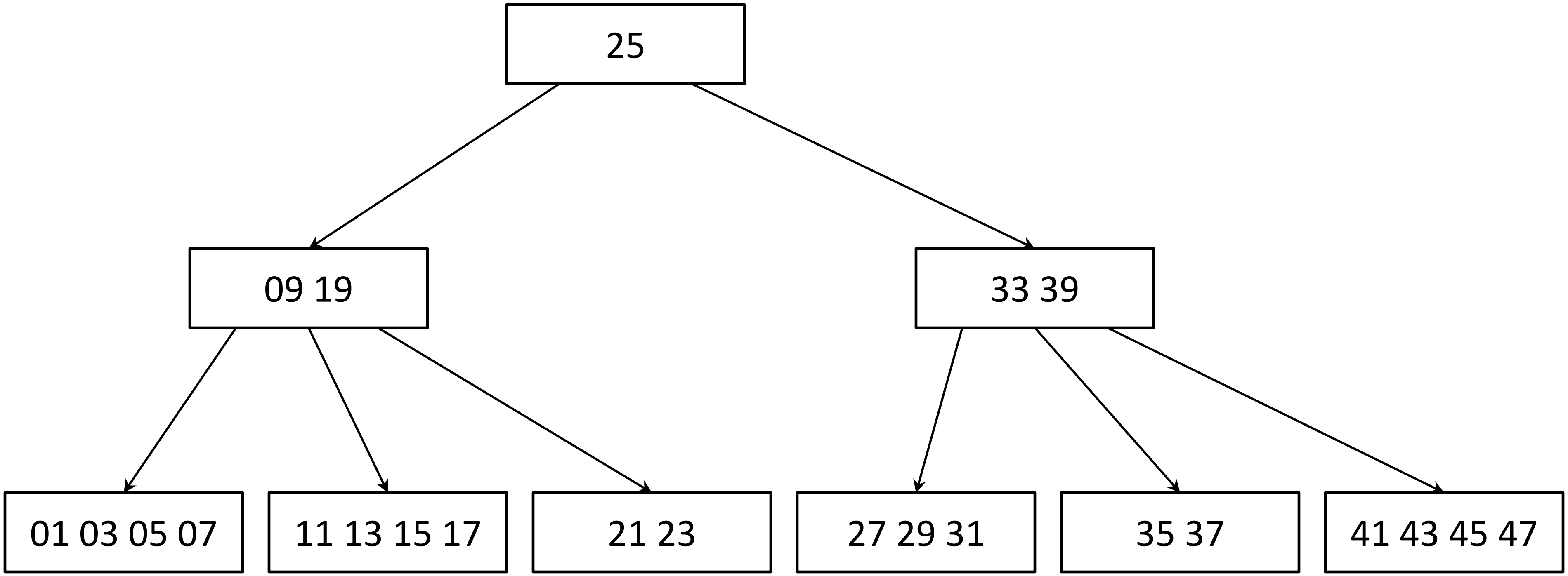
```
  if (w->right->color == w->left->color == BLACK){
    w->color = RED;
    x = x->parent;
  }
  else {
    if (w->right->color == BLACK) {
      w->left->color  = BLACK;
      w->color        = RED;
      rightRotate(root, w);
      w = x->parent->right;
    }
    w->color            = x->parent->color;
    x->parent->color = w->right->color = BLACK;
    leftRotate(root, x->parent);
    x = root;
  }
}
```

B-Trees

A B-tree of order t is either empty, or has the following properties:

- Every node contains at most $2t$ keys
- All nodes, except the root, contain at least t keys
- Every node is either a leaf or it has $m + 1$ descendants, where m is its number of keys
- All leaves are on the same level

Example



A Node (or Page) in B-Trees

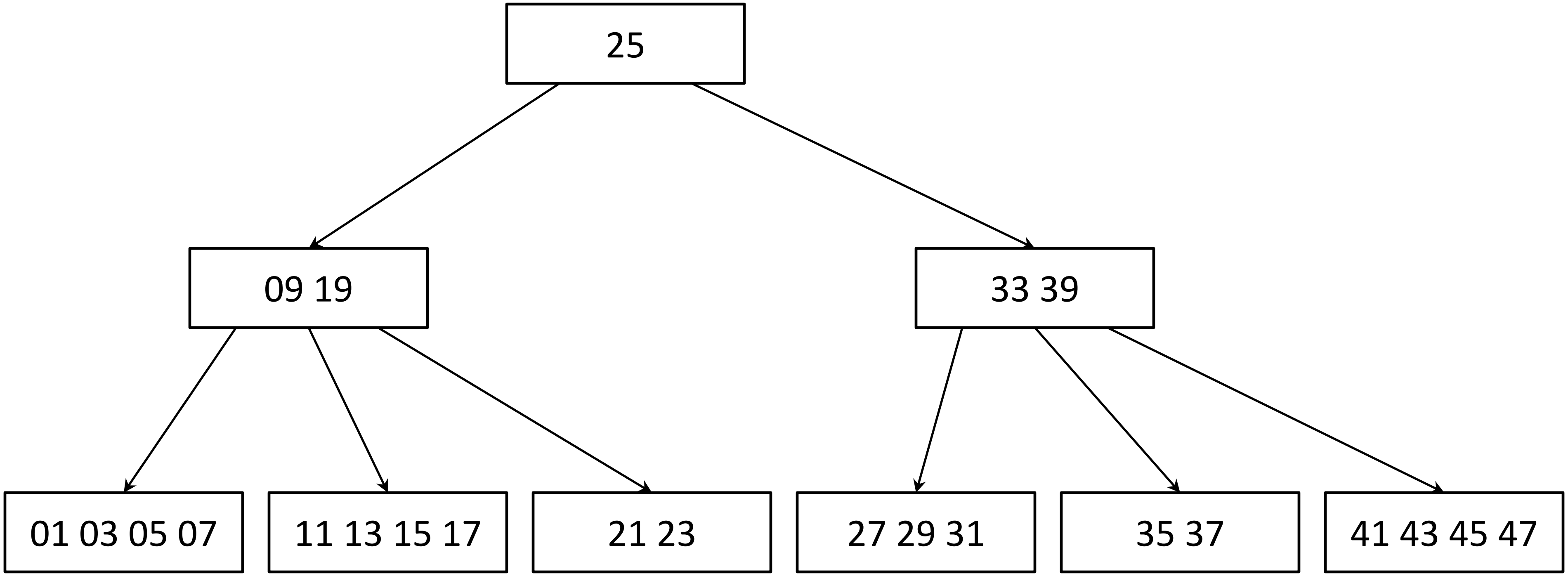
The form of a node is as follows:

p_0	k_1	p_1	k_2	p_2	…	k_m	p_m

where

- $k_1 < k_2 < \cdots < k_m$
- p_i is a pointer to a descendant
  - If it's a leaf: $p_i = \mathrm{NULL}, \forall i \in [0, m]$
- All keys in the node to which p_i points are greater than k_i and less than k_{i+1}

Example



(((01 03 05 07) 09 (11 13 15 17) 19 (21 23)) 25 ((27 29 31) 33 (35 37) 39 (41 43 45 47)))

Search Operation

- Let k be the search key. Assume that the node being considered contains m keys: $k_1, k_2, \dots, k_m$
- The search must start at the root of tree
  - If m is sufficiently large, one may use binary search; otherwise, a sequential search will do
- If the search is unsuccessful:
  - $k < k_1$: Search the node pointed to by p_0
  - $k > k_m$: Search the node pointed to by p_m
  - $k_i < k < k_{i+1}$: Search the node pointed to by p_i
- If the designated pointer is a null pointer: Stop!!!

Insertion Operation

- Assume that the key to be inserted is new

  ➡️ The search process terminates at a leaf

- The new key is inserted into the leaf if there is room
- If the leaf is full
  - *Insert the new key into the leaf*
  - Split the leaf into two nodes
  - Move the median key to the parent node
- The splitting can propogate upward up to the root, causing the tree to increase in height
  - This is the only way that a B-tree may increase its height

Deletion Operation

- Assume that the key to be deleted, say k, is in the tree

  ➡️ The search process terminates at the node containing k

- There are two different circumstances:
  - It's a leaf: The removal algorithm is plain and simple
  - Otherwise: The key must be replaced by its predecessor or successor, which happen to be on leaves and can easily be deleted

  ➡️ In either cases, the key that actually to be deleted is always on a leaf

Deletion Operation

- If the leaf contains more than t keys
  - Delete k and no further action is required
- If the leaf contains only t keys
  - If one of the adjacent siblings has more than t keys: Move one key from that sibling to the parent and one key from the parent to the leaf, and then delete k
  - Otherwise: Combine one of the adjacent siblings with the leaf and the median key from the parent, and then delete k

  ➡️ This process may propogate all the way up to the root which could result in reducing the height of the B-tree

B-Trees: Summary

- In practice, B-trees are designed to store and manage a large data on secondary storage devices
- A node of a B-tree normally corresponds to a *disk page*
  - For a typical disk, a page might be 2^{11} to 2^{14} bytes in length
- The time needed to access a disk page is typically ~10^5 larger than the time needed to compare keys in RAM
  - The number of disk accesses (or the height of B-trees) is the principal indicator of the efficiency of this data structure