Chapter 5

Introduction to Trees

The Main Topics

- Learn about binary trees and the basic terminologies used in binary trees
- Explore various binary tree traversal algorithms
- Learn about
  - Binary search trees
  - Binary search balanced trees
  - Red-Black trees
  - B-Trees

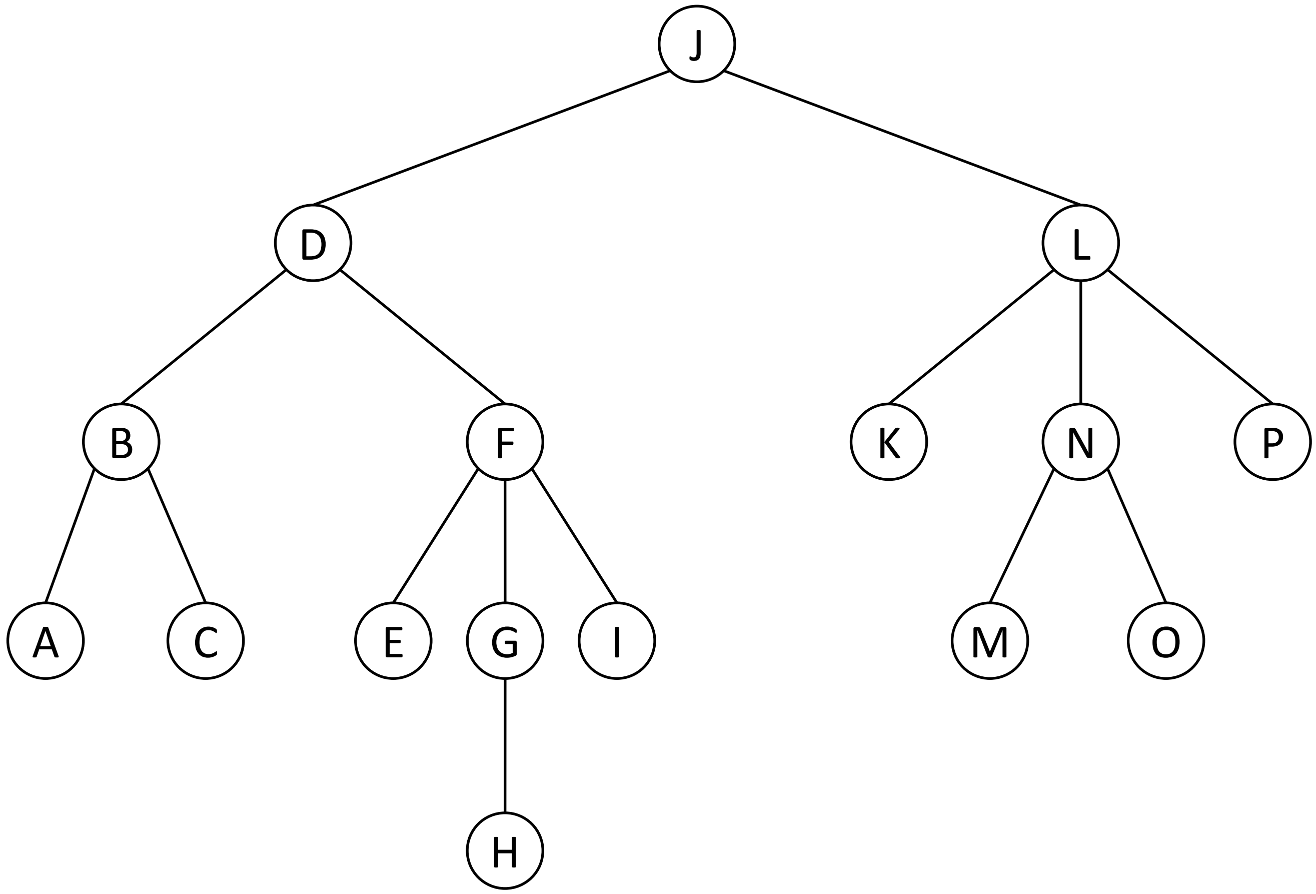
  and explore how to implement the basic operations on them

Definition of a Tree

A *tree structure* (or *tree* for short) with base type T is either:

- The empty structure, or
- A node of type T, called *root*, with a finite number of associated disjoint tree structures with base type T, called *subtrees*

Representation of a Tree



Basic Concepts

- The top node is commonly called the *root*
- If an edge is between node n and node m, and node n is above node m in the tree:
  - n is the *parent* of m
  - m is a *child* of n

  ➡️ The *parent-child* relationship between the nodes is generalized to the *ancestor*-*descendant* relationship
- A node that has no children is called a *leaf* of the tree
- A node which is not leaf is an *interior node*

Basic Concepts

- Children of the same parent are called *siblings*
- A *subtree* in a tree is any node in the tree together with all of its descendants
  - A subtree of a node n is a subtree rooted at a child of n
- The number of children of an interior node is called its *degree*
- The maximum degree over all nodes is the *degree of the tree*

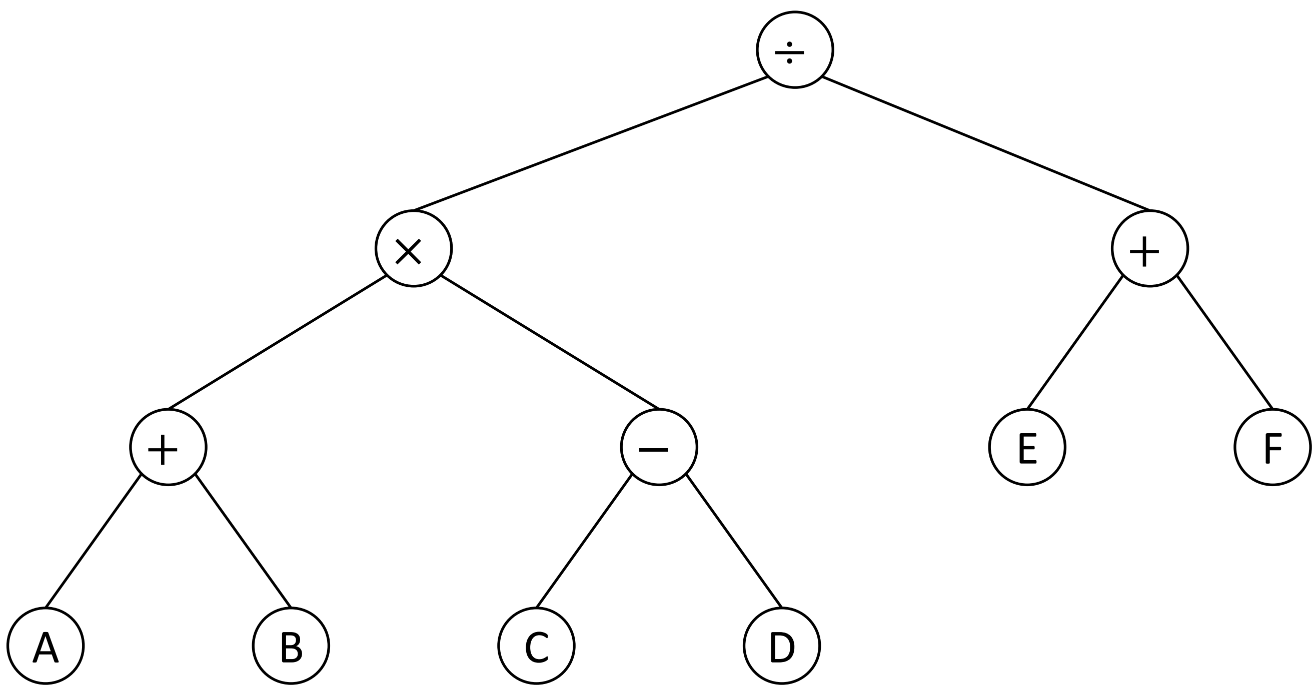
➡️ The tree name is named after the degree of the tree

Basic Concepts

- The sequence of edges that connects an ancestor and a descendant is called a *path*
- The *height of a tree* is the number of nodes on the longest path from the root to a leaf
  - Alternatively, the number of edges of the longest path denotes the height
- If node n is at *level* i, then its child is said to be at *level* $i + 1$. The root of tree is at *level* 1
  - Some authors define the root is at *level* 0

Binary Trees

A *binary tree* is a finite set of nodes which either is empty or consists of a root with two disjoint binary trees called the *left* and the *right subtree* of the root



Representation of Binary Trees

- A binary tree consists of nodes of a type defined as follows:

```
typedef struct Node * Ref;
struct Node {
  int key;
  Ref left, right;
};
```

- An *empty* binary tree is represented as follows:

```
Ref root = NULL;
```

Constructing a Perfectly Balanced Binary Tree

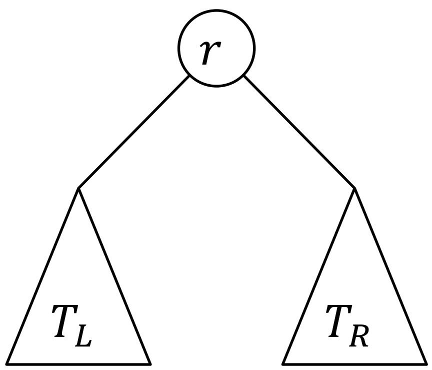
- A binary tree is *perfectly balanced* if for each node the numbers of nodes in its left and right subtrees differ by at most 1
- An algorithm for constructing a PBBT with n nodes is best formulated in recursive terms:
  1. Use one node for the root
  2. Construct the left subtree with $n_l = \left\lfloor \frac{n}{2} \right\rfloor$ nodes *in this way*
  3. Construct the right subtree with $n_r = n - n_l - 1$ nodes *in this way*

Pseudo-code

```
Ref tree(n) {
  if (n == 0)   return NULL;
  nl = n / 2;
  nr = n - nl - 1;
  cin >> k;
  Ref r = new Node;
  r->key = k;
  r->left = tree(nl);
  r->right = tree(nr);
  return r;
}
Ref root = tree(n);
```

Binary Tree Traversal

- A traversal algorithm for a binary tree visits each and every node in the tree
  - Assume that visiting a node simply means displaying the key portion of the node
- According to the recursive definition of a binary tree, the binary tree T is either empty or is of the form:



Binary Tree Traversal

- If T is empty, the traversal algorithm takes no action
  - An empty tree is the base case
- Otherwise, the traversal algorithm must perform three tasks: display the key in the root r, and traverse two subtrees T_L and T_R
- The algorithm has three choices when to visit r:
  - $r \to T_L \to T_R$ (*preorder travesal*)
  - $T_L \to r \to T_R$ (*inorder travesal*)
  - $T_L \to T_R \to r$ (*postorder travesal*)

Binary Tree Traversal: Pseudo-code

```
void preOrder(Ref r) {
  if (r) {
    cout << r->key;
    preOrder(r->left);
    preOrder(r->right);
  }
}
void inOrder(Ref r) {
  …
}
void postOrder(Ref r) {
  …
}
```

Binary Search Trees

- A *binary search tree* (or BST) is one that satisfies the following property:

  "For each node its value is greater than all values in its left subtree and less than all values in its right subtree"

- BSTs are also called *ordered binary trees*

Binary Search Tree Traversal

- The traversal of a BST is the same as that of a binary tree
- An *inorder traversal* of a BST will visit the tree's nodes in sorted order according to their keys

Search Operation

- Firstly, check whether the root (of the whole tree or the current subtree) contains the search key k
  - If it does then the search process finishes succefully
  - If k is less than the value stored in the root, the search process continues with the left subtree
  - If k is greater than the value stored in the root, the search process continues with the right subtree
- If k is not in the tree, the search process always ends at an *empty* subtree

Search Operation: Pseudo-code

```
Ref   search(Ref r, int k) {
  while (r)
    if (r->key == k)
      return r;
    else
      if (r->key > k)
        r = r->left;
      else
        r = r->right;
  return  NULL;
}
```

Insertion Operation

- A new node is always inserted to a tree as a new leaf
- It cannot be inserted to a node whose both subtrees are nonempty
- Since BSTs are ordered ones, so the new key has to be inserted to an appropriate place in the tree
  - The algorithm must search for the right place first

➡️ The operation should be called *tree search with insertion* task

Insertion Operation: Rough Algorithm

- Firstly, the key will be searched for in the tree
- If the key is found, the algorithm does nothing more than returns back
- Otherwise, the search process will lead us to an empty subtree
- The new node must be inserted at the place of the empty tree

Insertion Operation: Pseudo-code

```
void  searchAdd(Ref & r, int k) {
  if (r == NULL) {
    r = new Node;
    r->key = k;
    r->left = r->right = NULL;
  }
  else
    if (r->key > k) searchAdd(r->left, k);
    else
      if (r->key < k) searchAdd(r->right, k);
      else
        return;
}
```

Deletion Operation

- Similar to the insertion operation, this operation consists of search task and deletion task

➡️ This operation should be called *tree search with deletion* task

- Firstly, the key will be searched for in the tree
  - If the key is not found: Stop!!!
  - Otherwise, the search process will lead us to the node that contains the key

➡️ It's time to run deletion task

Deletion Operation

- It is simple if the node to be deleted is a leaf or one with a single child
- The difficulty lies in removing a node with two children
  - A single pointer cannot point in two directions

Solution: The deleted node is to be replaced by either its *predecessor* or *successor*

  - One of them will be actually deleted

Deletion Operation: Pseudo-code

```
void  searchDel(Ref & r, int k) {
  if (r == NULL) return;
  if (r->key > k) searchDel(r->left, k);
  else
    if (r->key < k) searchDel(r->right, k);
    else {      //   ==
      q = r;
      if (q->right == NULL) r = q->left;
      else
        if (q->left == NULL) r = q->right;
        else
          del(r->left, q);
      delete q;
    }
}
```

Deletion Operation: Pseudo-code

```
void  del(Ref & r, Ref & q) {
  if (r->right)
    del(r->right, q);
  else {
    q->key = r->key;
    q = r;
    r = r->left;
  }
}
```

```
void searchDel(Ref & r, int k)
{
   …
   else {    //     ==
      q = r;
      if (q->right == NULL)
         r = q->left;
      else
         if (q->left == NULL)
            r = q->right;
         else
            del(r->left, q);
      delete q;
   }
}
```

Balanced Binary Search Trees

- In 1962, two Soviet scientists, Adelson-Velskii and Landis, proposed a new type of BSTs called *balanced binary search tree*
  - Nowadays, it's well known as AVL tree – named after its inventors
- The noteworthy point of AVL trees is its balance criterion

The Balance Criterion

- The criterion is stated as follows:

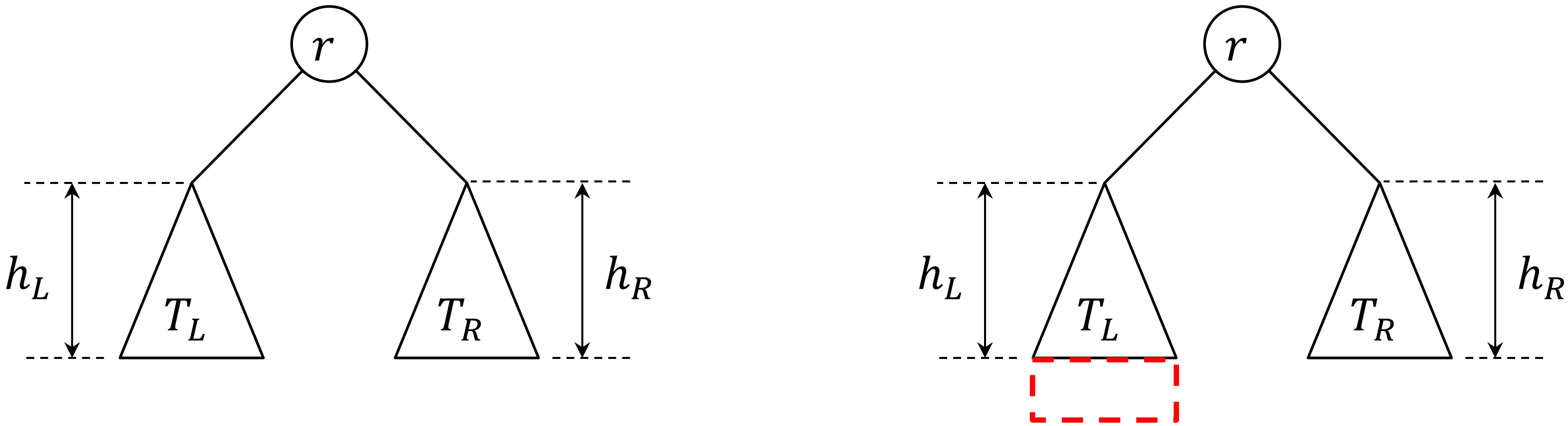
  A binary tree is balanced if and only if for every node the heights of its two subtrees differ by at most 1

- The criterion makes AVL trees have 3 advantages over the others:
  - The height of AVL trees is at most $1.44 \log_2 n$
  - AVL trees could not be deformed as BSTs
  - The cost of rebalancing AVL trees is nearly constant time and in the worst case, it's $\mathrm{O}(\log n)$

The Basic Operations

- The search algorithm for an AVL tree is the same as the search algorithm for a BST
- Insertion and deletion operations on AVL trees are somewhat different from the ones discussed for BSTs
  - After inserting a node to (or deleting a node from) an AVL tree, the resulting tree may still be an AVL tree or a *rebalancing* must be done to restore the balance criterion
- Three operations run in $O(\log n)$

When Does an Imbalance Arise?

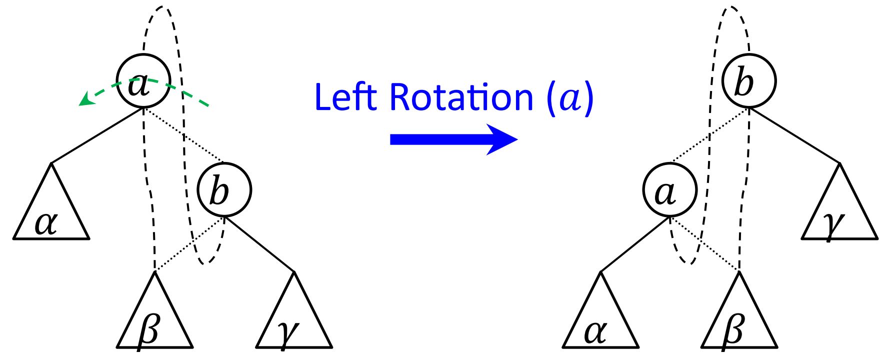


- Given a root r with the left and right subtrees: T_L and T_R
  - In general, r can be any node in an AVL tree
- Let h_L and h_R be the heights of T_L and T_R, respectively
- Assume that the new node is inserted to T_L causing its height to increase by 1

When Does an Imbalance Arise?

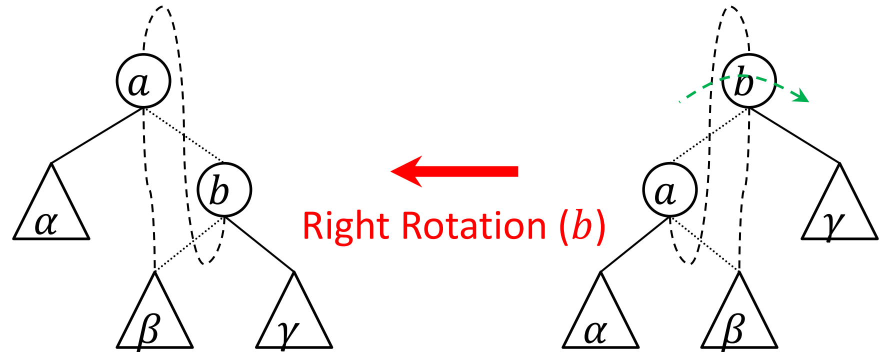
Before	After
$h_L = h_R$	$h_L > h_R$: h_L and h_R become of unequal height, but the balance criterion is not violated
$h_L < h_R$	$h_L = h_R$: h_L and h_R obtain equal height, the balance has even been improved
$h_L > h_R$	The balance criterion is violated, and the tree must be restructured

The Rebalancing Procedure



- It's also called *rotating* the tree
- Suppose the rotation occurs at a node r:
  - *Left rotation*: Certain nodes from the right subtree of r move to its left subtree; the root of the right subtree of r becomes the new root of the reconstructed subtree

The Rebalancing Procedure



- It's also called *rotating* the tree
- Suppose the rotation occurs at a node r:
  - *Left rotation*: Certain nodes from the right subtree of r move to its left subtree; the root of the right subtree of r becomes the new root of the reconstructed subtree
  - *Right rotation*: Certain nodes from the left subtree of r move to its right subtree; the root of the left subtree of r becomes the new root of the reconstructed subtree

The Rebalancing Procedure: Case 1

The Rebalancing Procedure: Case 2

The Rebalancing Procedure: Case 3

The Rebalancing Procedure: Case 4

Insertion Operation

- Assume that the key to be added to the AVL tree is the new one
- Insertion operation includes 2 stages:
    1. Search the tree and add the new node to the appropriate place
    2. After inserting the new node in the tree, the resulting tree might not be an AVL tree
       ➡️ The rebalancing procedure must be activated

Insertion Operation

- How does the rebalancing procedure work?
  - It’s accomplished by retreating along the search path and check if the balance criterion is violated at each node
  - If the balance criterion is violated at a node, the subtree rooted at that node will be rebalanced
- Once balance is established, the subtree no longer grew in height
  ➡️ The algorithm can ignore the remaining nodes on the path back to the root

Deletion Operation

- Assume that the key to be deleted is in the AVL tree
- Deletion operation includes 2 stages:
  1. Find the node containing the key to be deleted and delete it
  2. After deleting the node in the tree, the resulting tree might not be an AVL tree

     ➡️ The rebalancing procedure must be activated

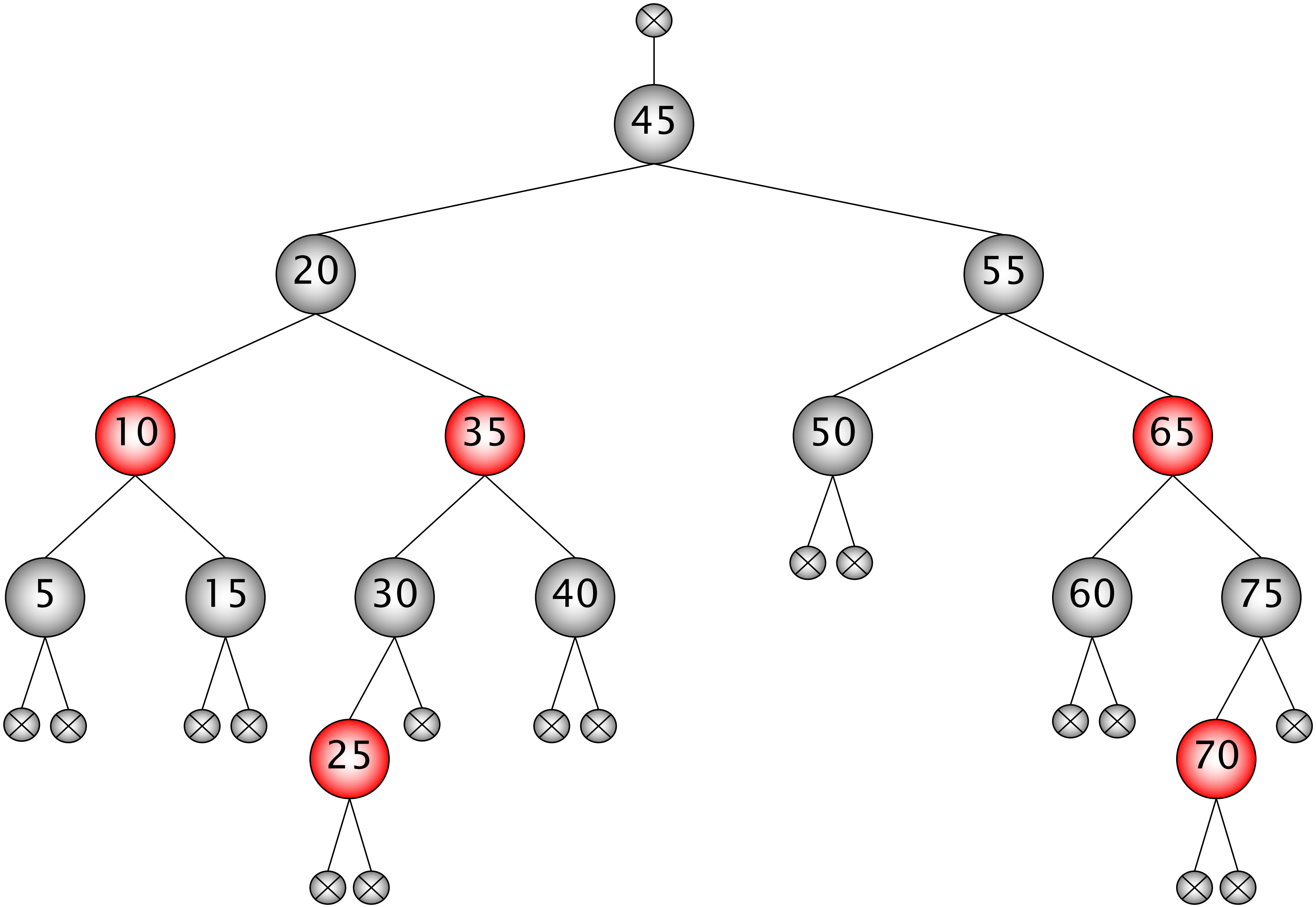
Deletion Operation

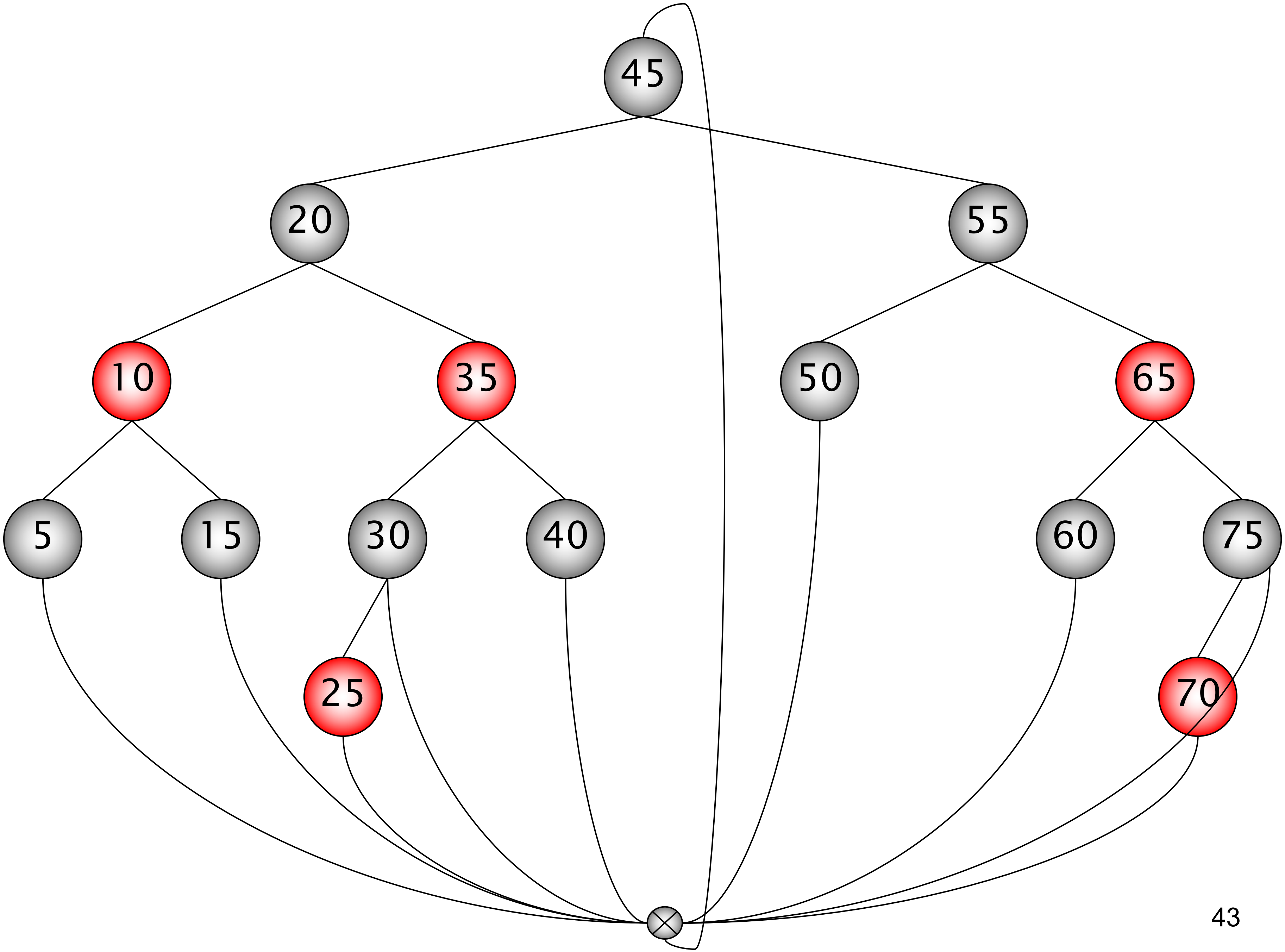
- How does the rebalancing procedure work?
- Its performance is similar to the case of insertion operation except two things:
  - After rebalancing a subtree, the overall tree might still not be an AVL tree

  ➡️ The algorithm has to continuously traverse back to the root node
  - The second thing?

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*:

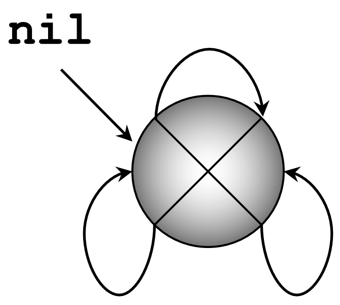
1. Every node is either ***red*** or ***black***
2. The root of the tree is always ***black***
3. All leaves are ***black***
4. If a node is ***red***, then its parent is ***black***
5. Any path from a node to any of its leaves contains the same number of black nodes, called *black height*

Representation of Red-Black Trees

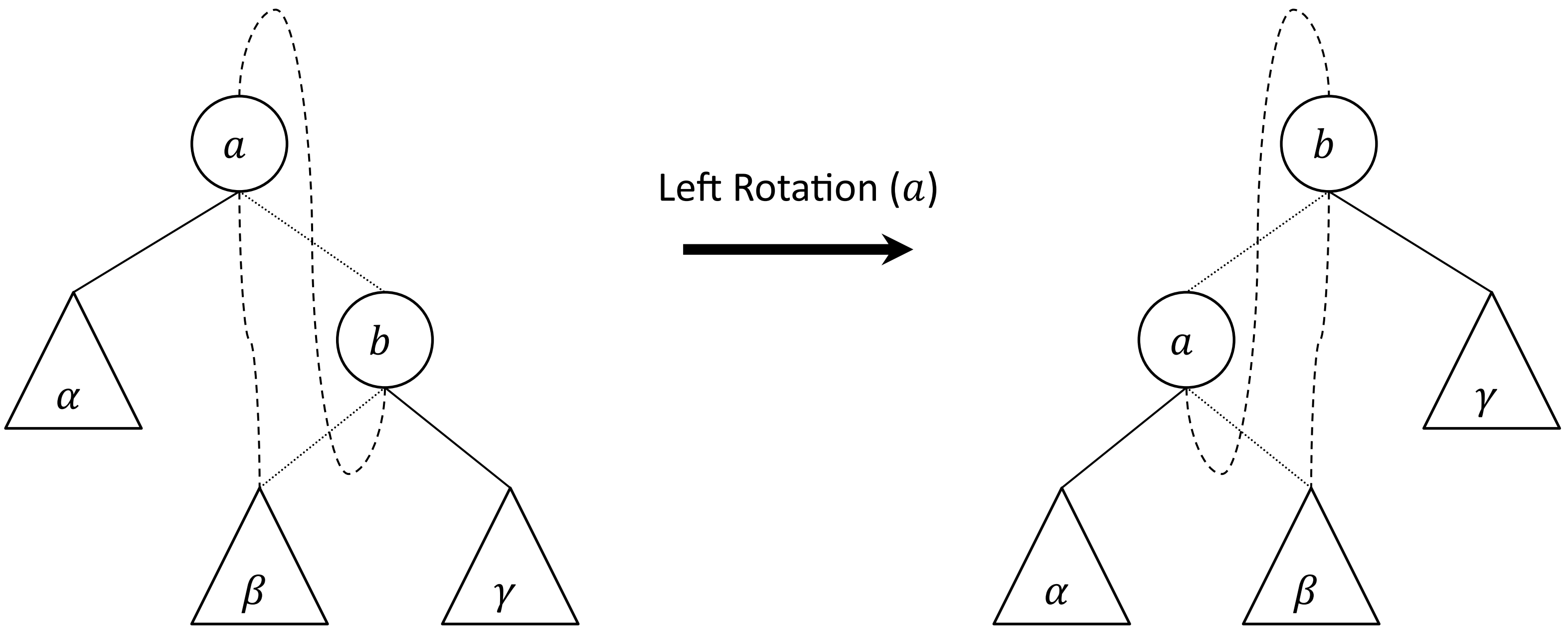
```
typedef struct Node * Ref;
struct  Node {
   int key;
   int color;
   Ref parent, left, right;
};
ref getNode(int key, int color, Ref nil) {
   p = new Node;
   p->key   = key;
   p->color = color;
   p->left  = p->right = p->parent = nil;
   return  p;
}
```

The Initial State of a Red-Black Tree

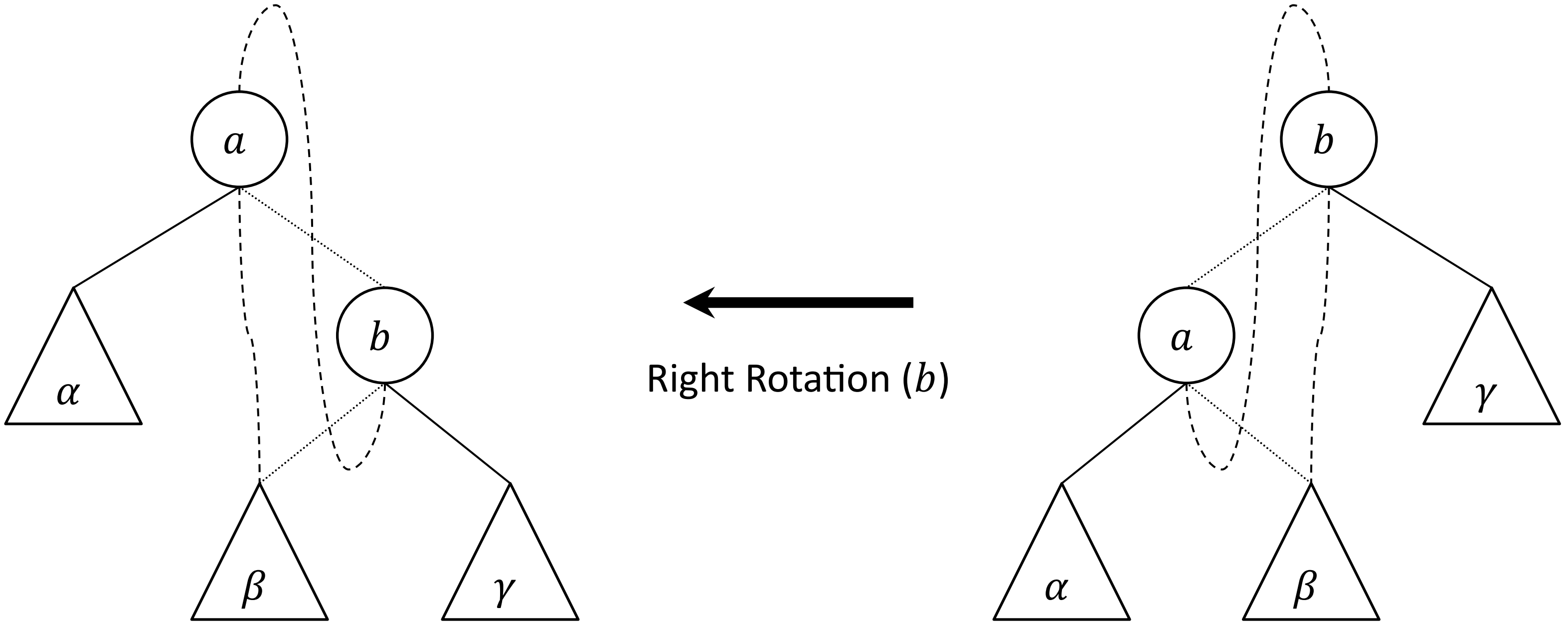
```
Ref nil, root;
…
nil = new Node;
nil->color = BLACK;
nil->key = -1;
nil->left =
nil->right =
nil->parent = nil;
…
root = nil;
```



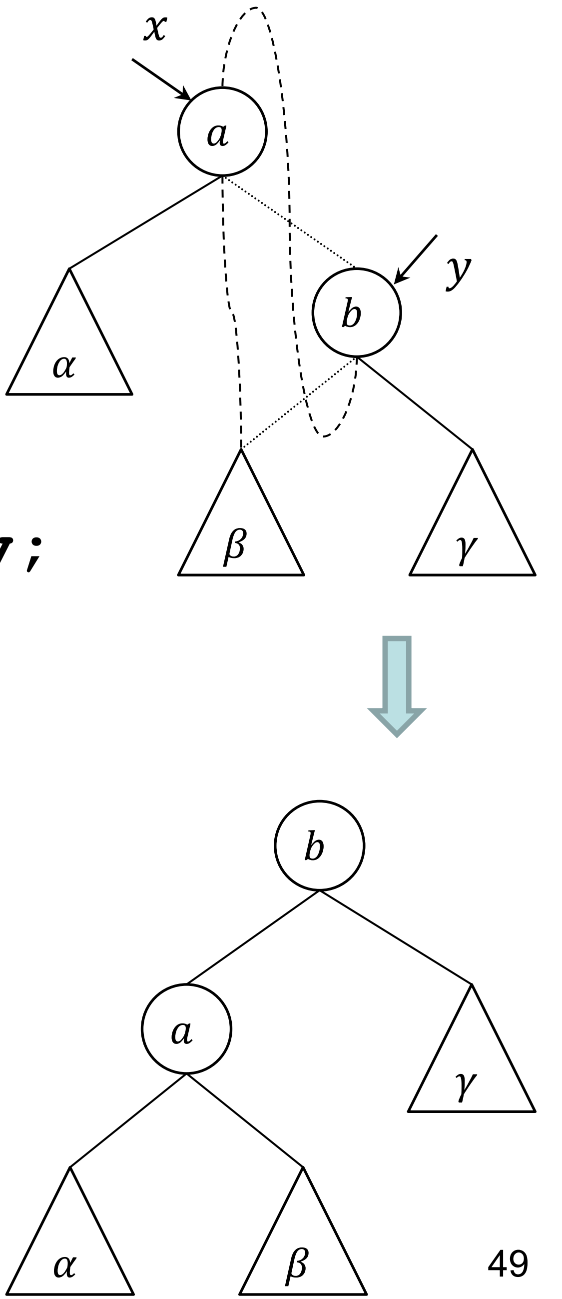
Tree Rotation



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 criterion "no two consecutive red nodes" is violated

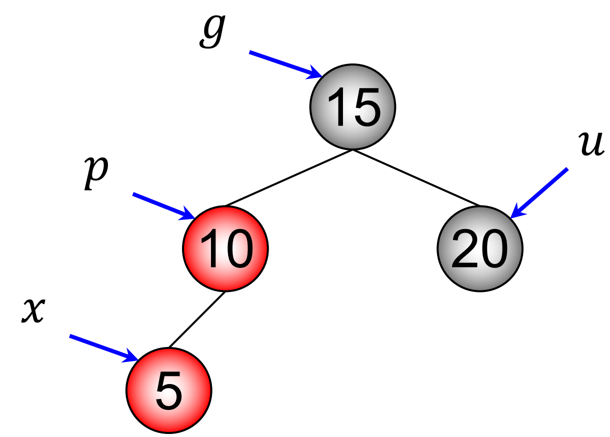
    ➡️ Rebalancing the tree is needed

Insertion Operation: Pseudo-code

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

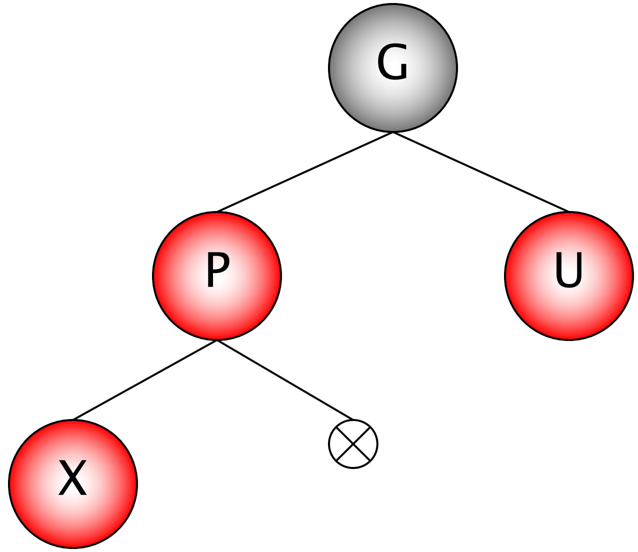
```
BST_Insert(Ref & root, ref x) {
  y = nil;
  z = root;
  while (z != nil) {
    y = z;
    if (x->key < z->key)         z = z->left;
    else if (x->key > z->key)  z = z->right;
    else                           return;
  }
  x->parent = y;
  if (y == nil)   root = x;
  else
    if (x->key < y->key)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

Some Conventions

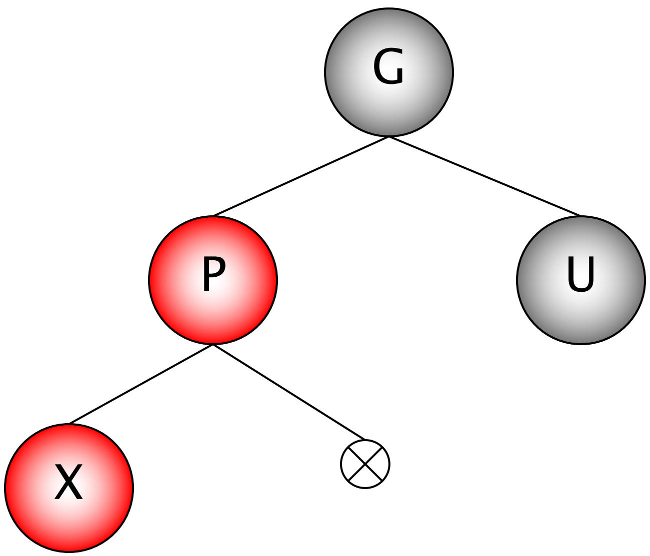


- x: The pointer designated to point to the newly added leaf
- p: The pointer designated to point to the *parent* of the node pointed to by x
- u: The pointer designated to point to the *uncle* of the node pointed to by x
- g: The pointer designated to point to the *grandparent* of the node pointed to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

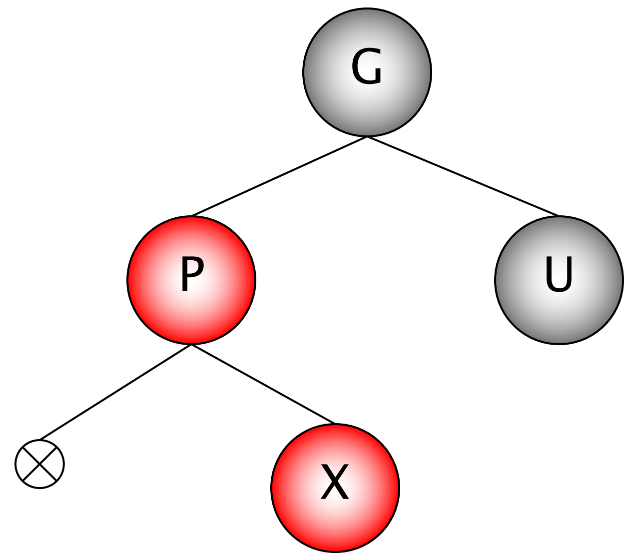
Imbalance Cases



Case 1



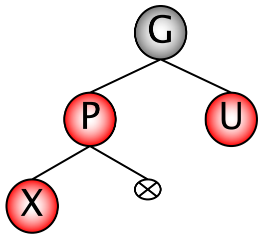
Case 2



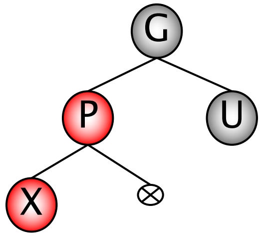
Case 3

The Strategies for Rebalancing an RBT

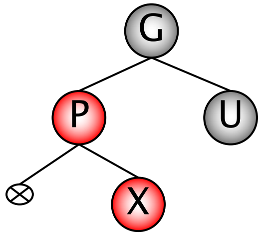
- Case 1
  - ➞ Reverse the color of three nodes: u, p, and g



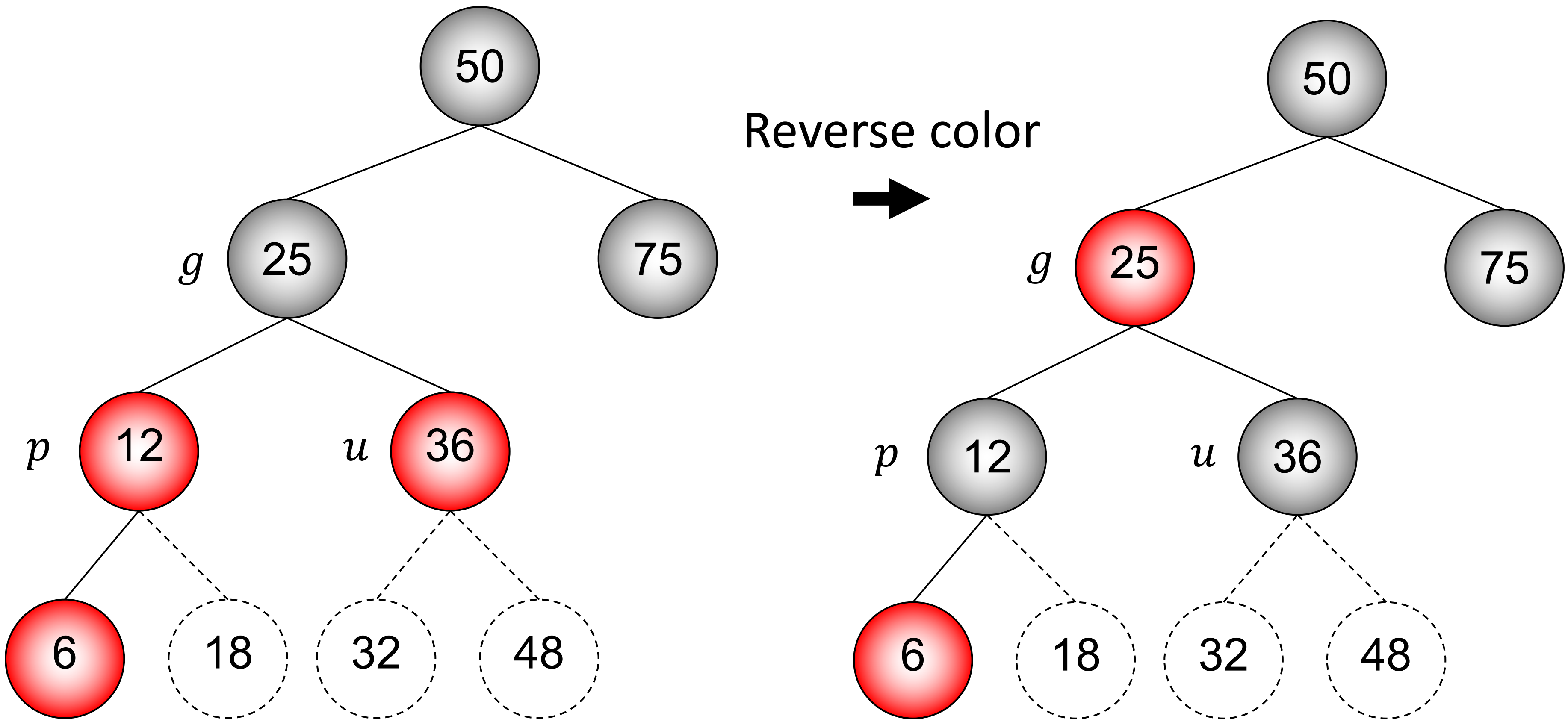
- Case 2
  - ➞ Reverse the color of two nodes: p and g
  - ➞ Run a rotation at g



- Case 3
  - ➞ Run a rotation at parent p
  - ➞ Go to Case 2

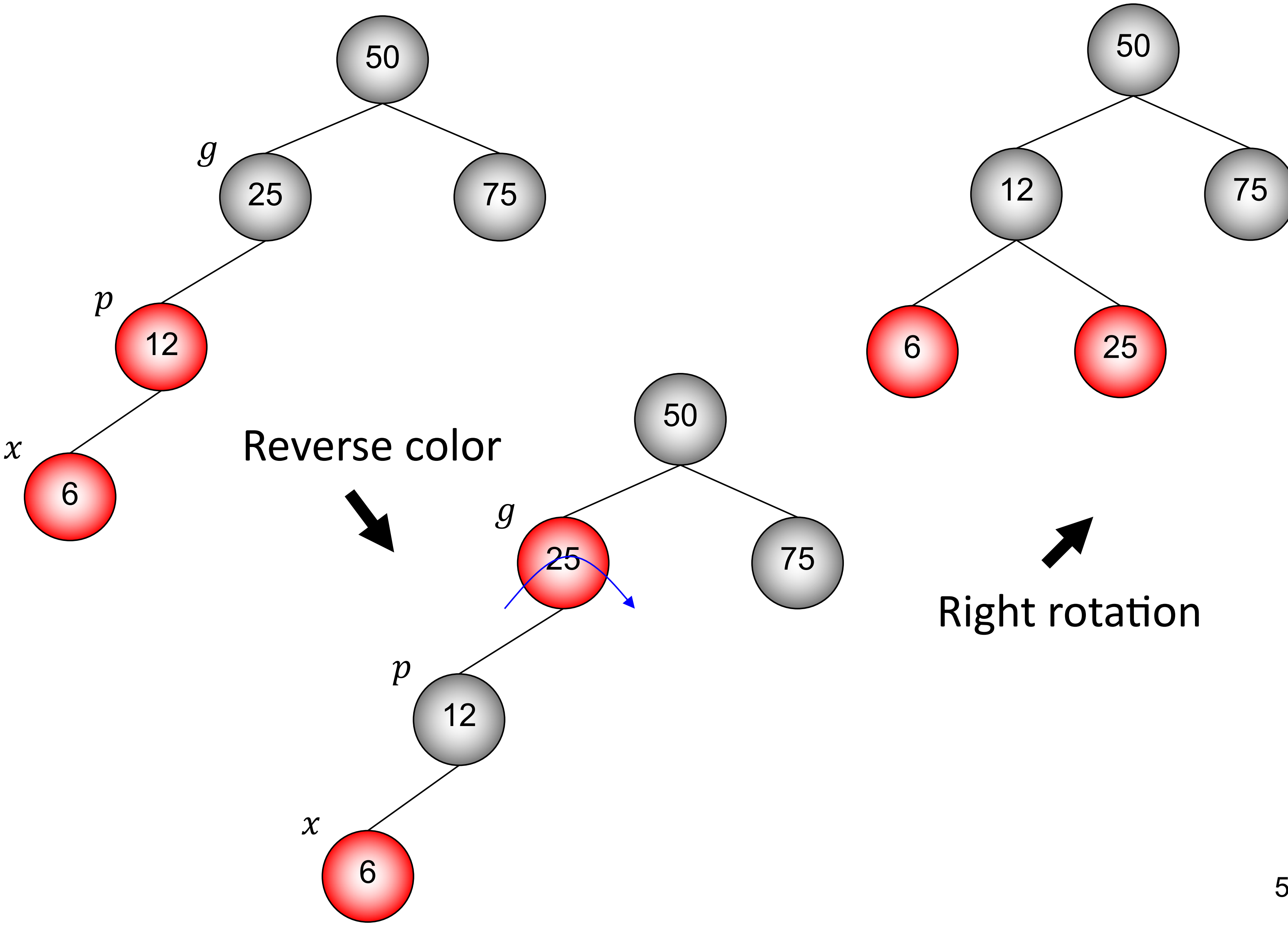


Case 1



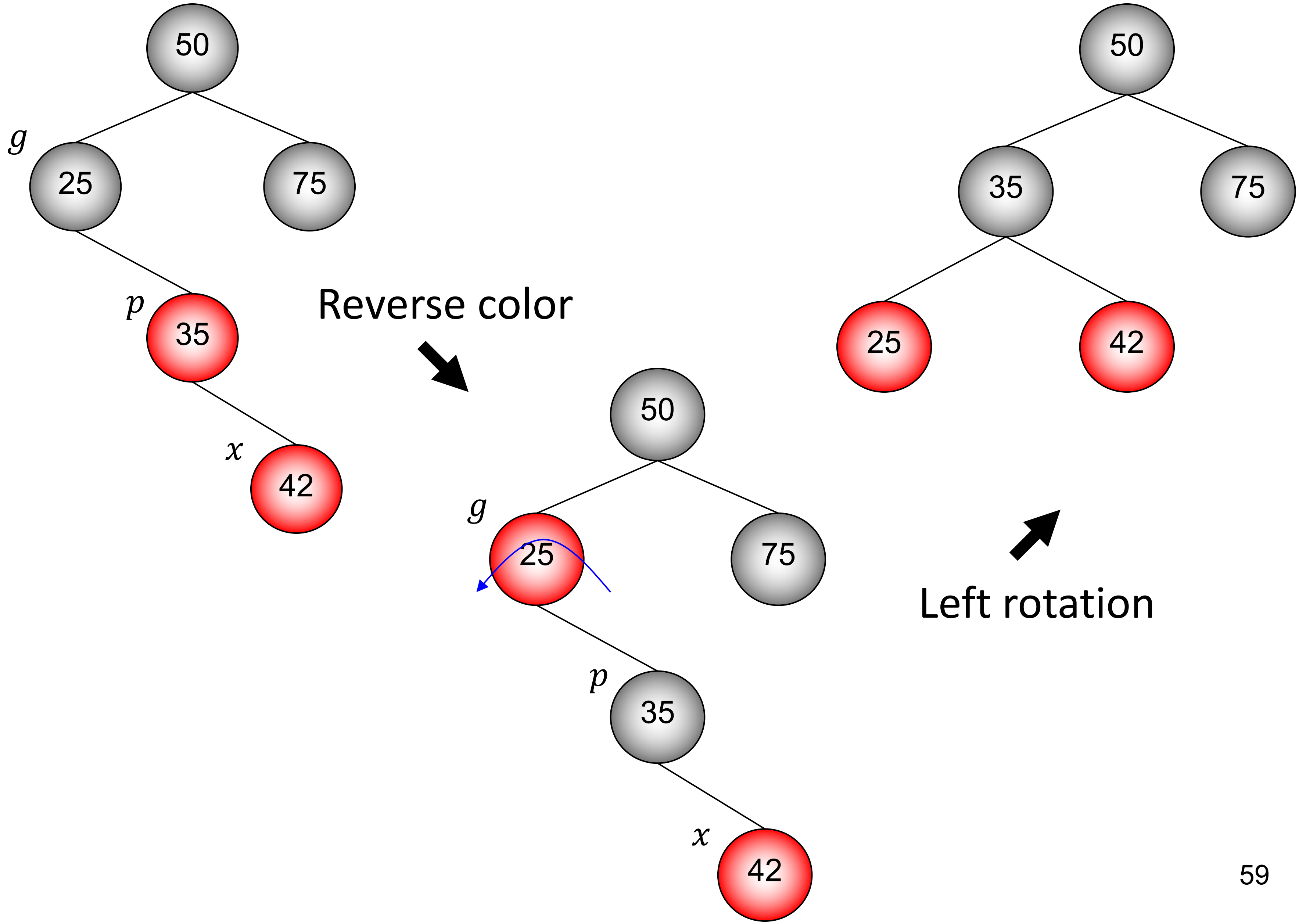
50
25
75
12
36
6
18
32
48
g
p
u
Reverse color
50
25
75
12
36
6
18
32
48
g
p
u

Case 2

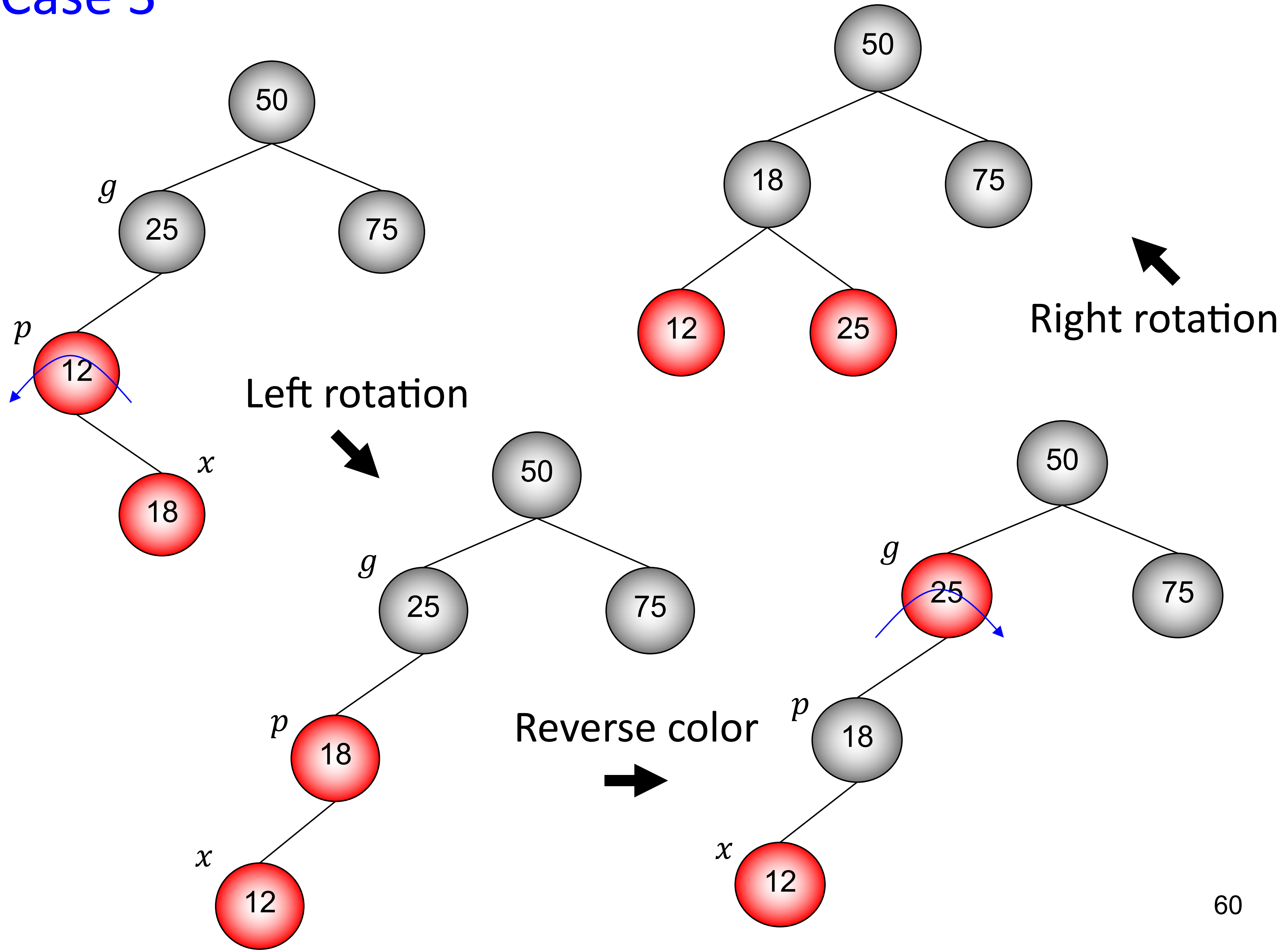


50
g
25
75
p
12
x
6
Reverse color
50
g
25
75
p
12
x
6
Right rotation
50
12
75
6
25

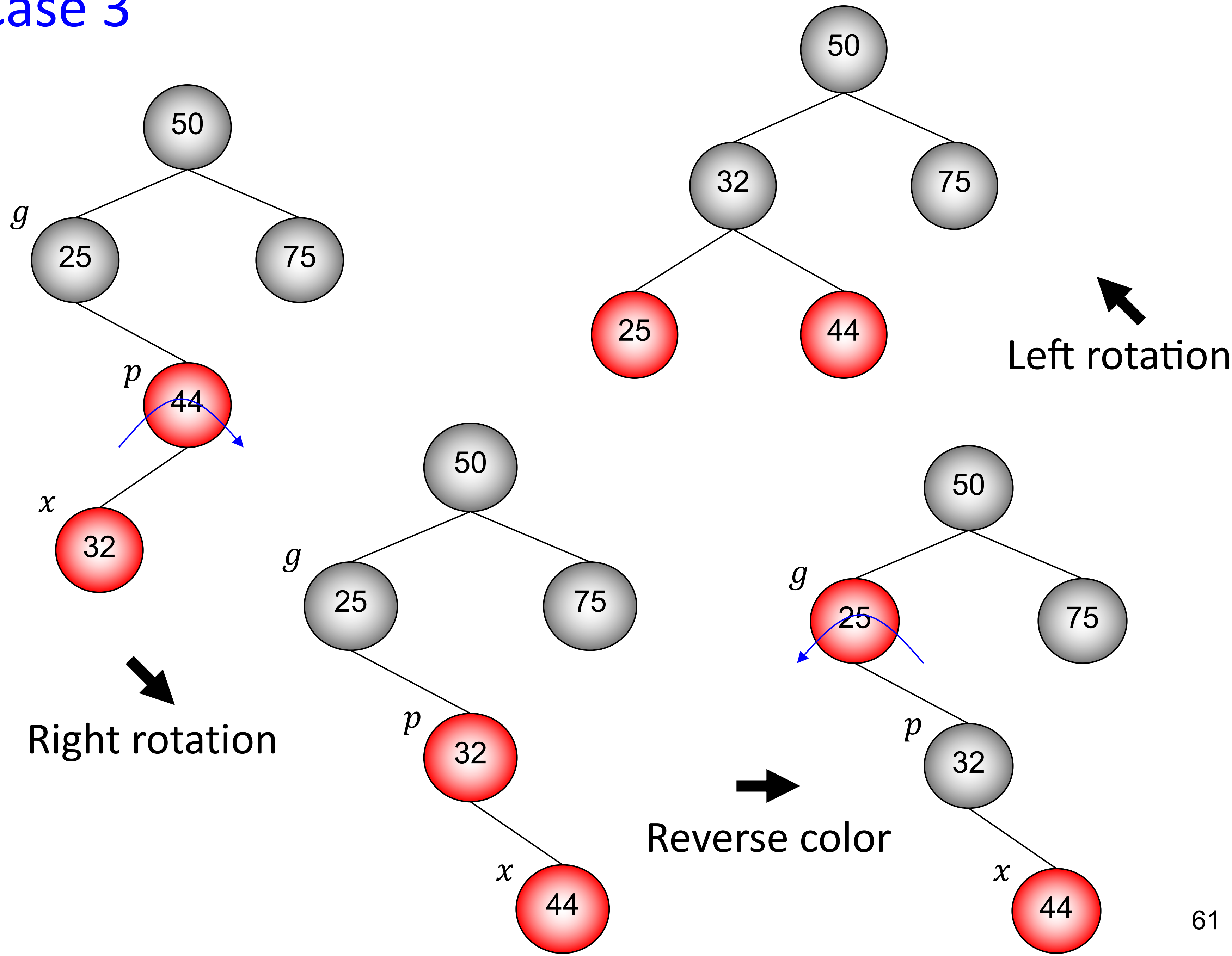
Case 2


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

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

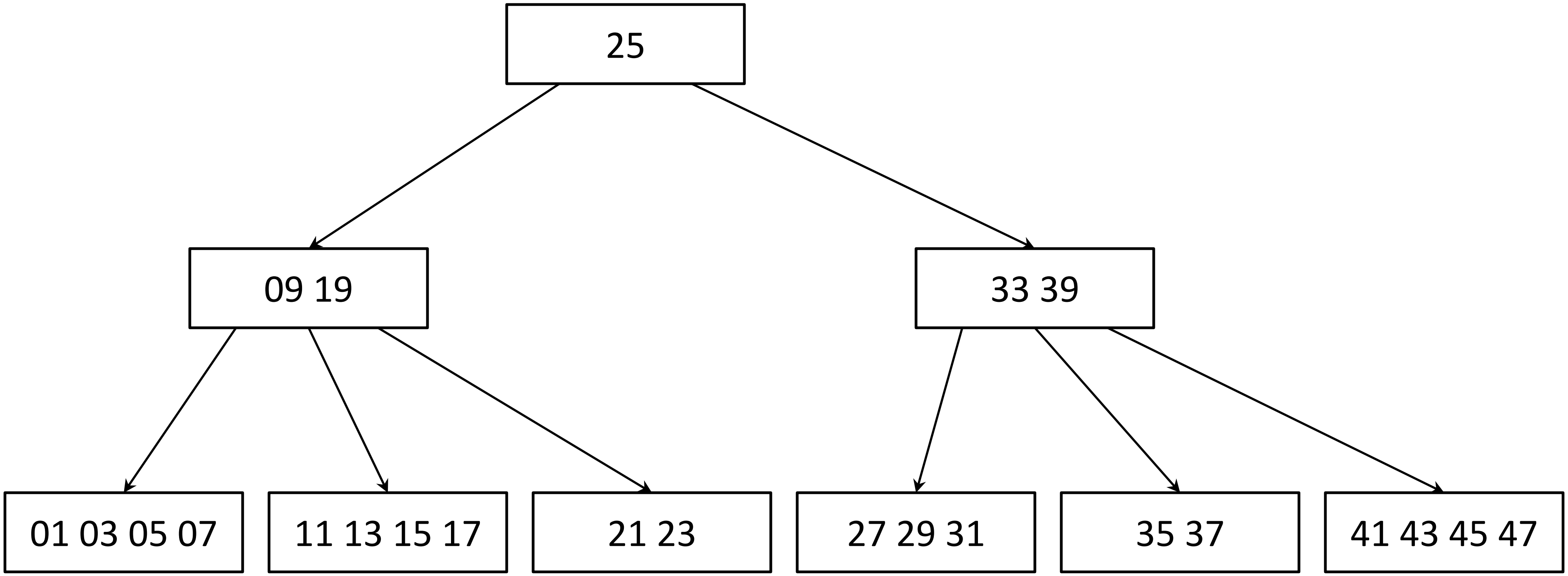
Insertion Operation: Pseudo-code

```
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);
  }
}
```

B-Trees

Example: A B-tree of Order 2

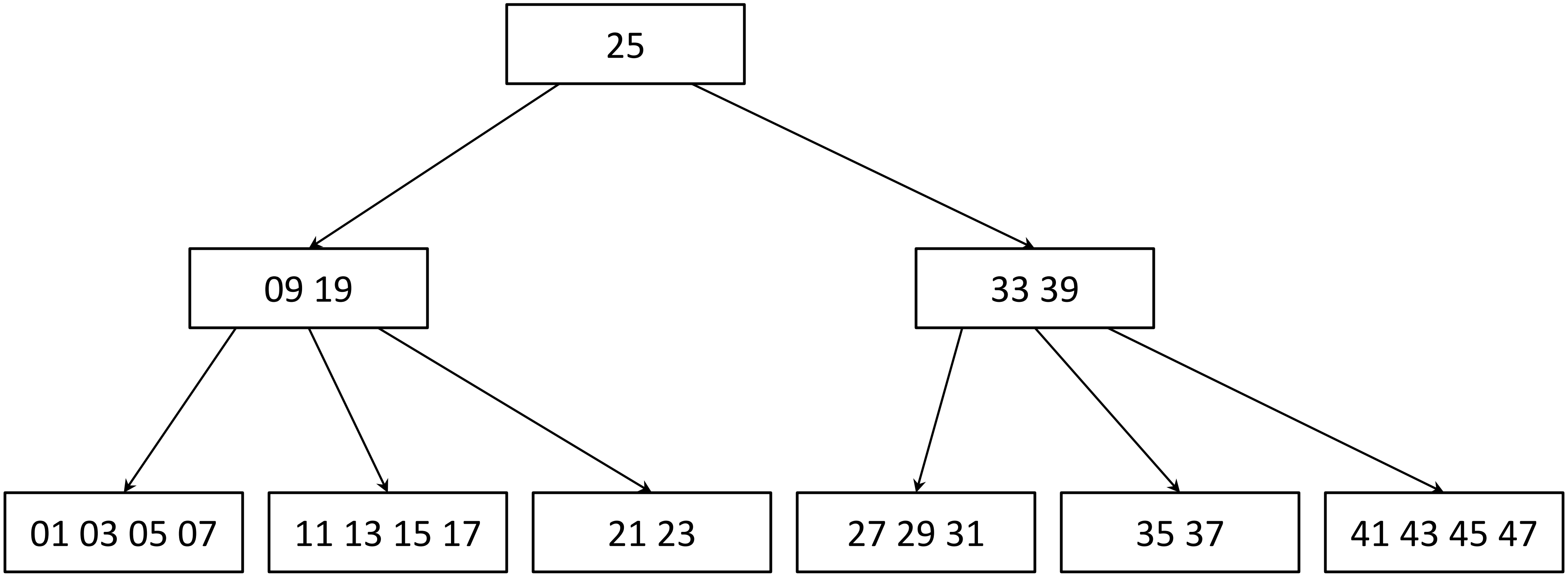


Definition of B-Trees

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 $m + 1$ children, where m is its number of keys
- All leaves are on the same level

Example: A B-tree of Order 2



The Structure of a Node (or Page)

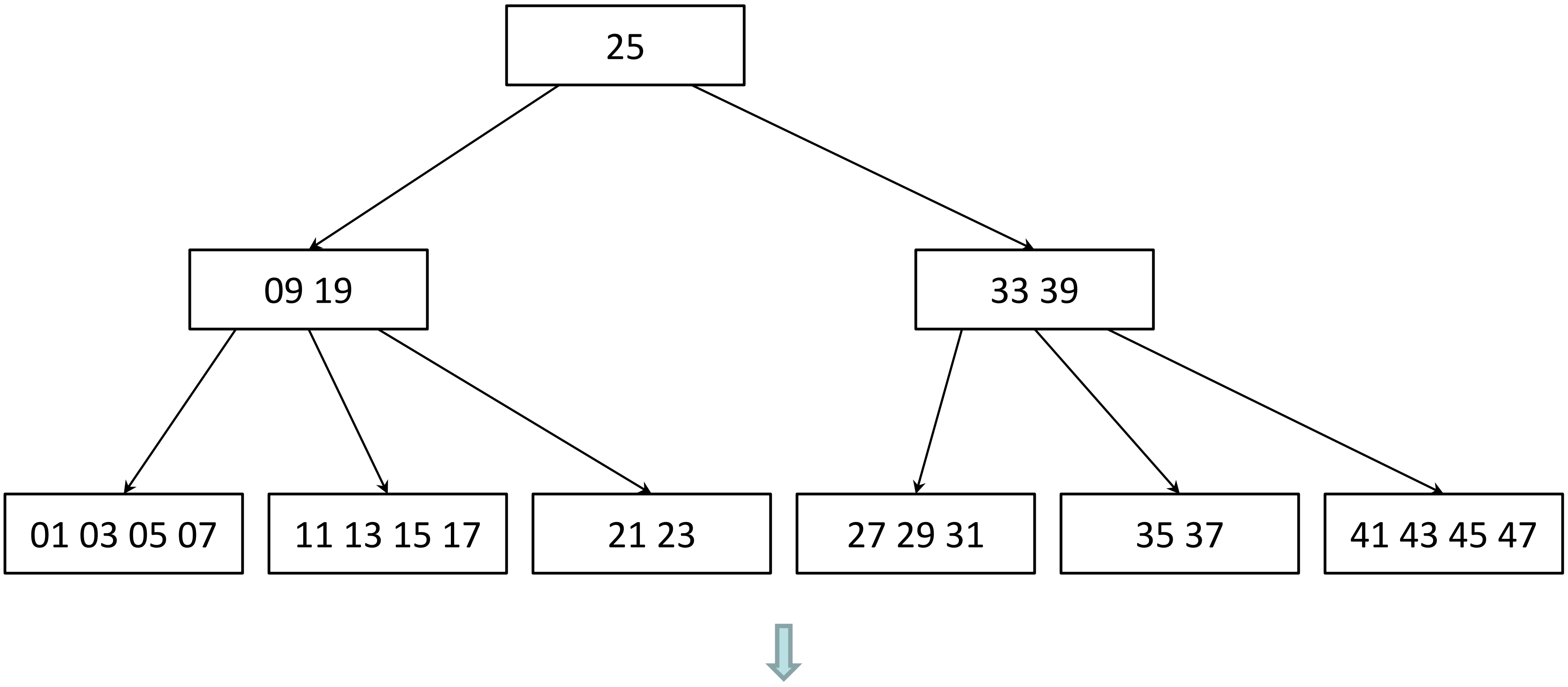
The structure of a node is as follows:

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 child
  - 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

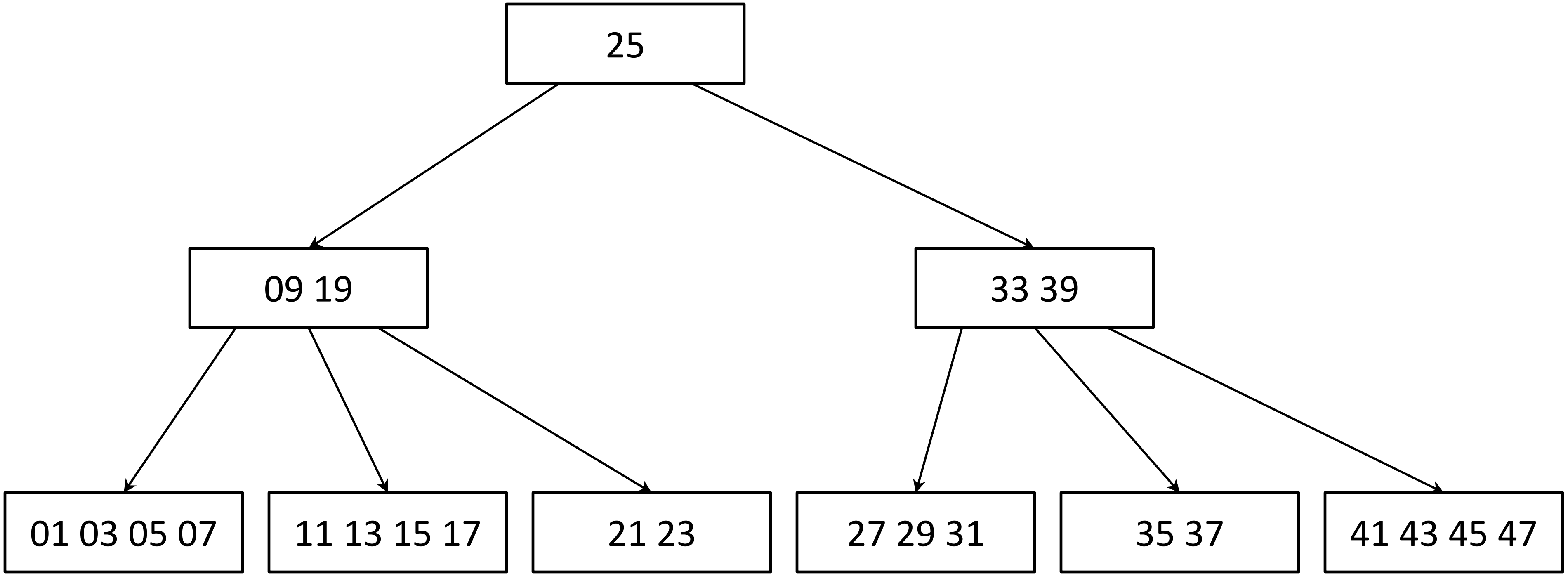


(((01 03 05 07) 09 (11 13 15 17) 19 (21 23)) 25 ((27 29 31) 33 (35 37) 39 (41 43 45 47)))

Search Operation

- The search must start at the root of tree
  - Assume that the node being considered contains m keys: $k_1, k_2, \ldots, k_m$
  - If m is sufficiently large, one may use binary search; otherwise, a sequential search will do
- Let k be the search key. 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!!!

Example



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
  - (for pedagogical reasons) *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

Example

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

  ➡️ 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

Example

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 height of B-trees is the principal indicator of the efficiency of this data structure