Algorithms and Data Structures 2

15 – Red-black trees

Dr Michele Sevegnani

School of Computing Science
University of Glasgow

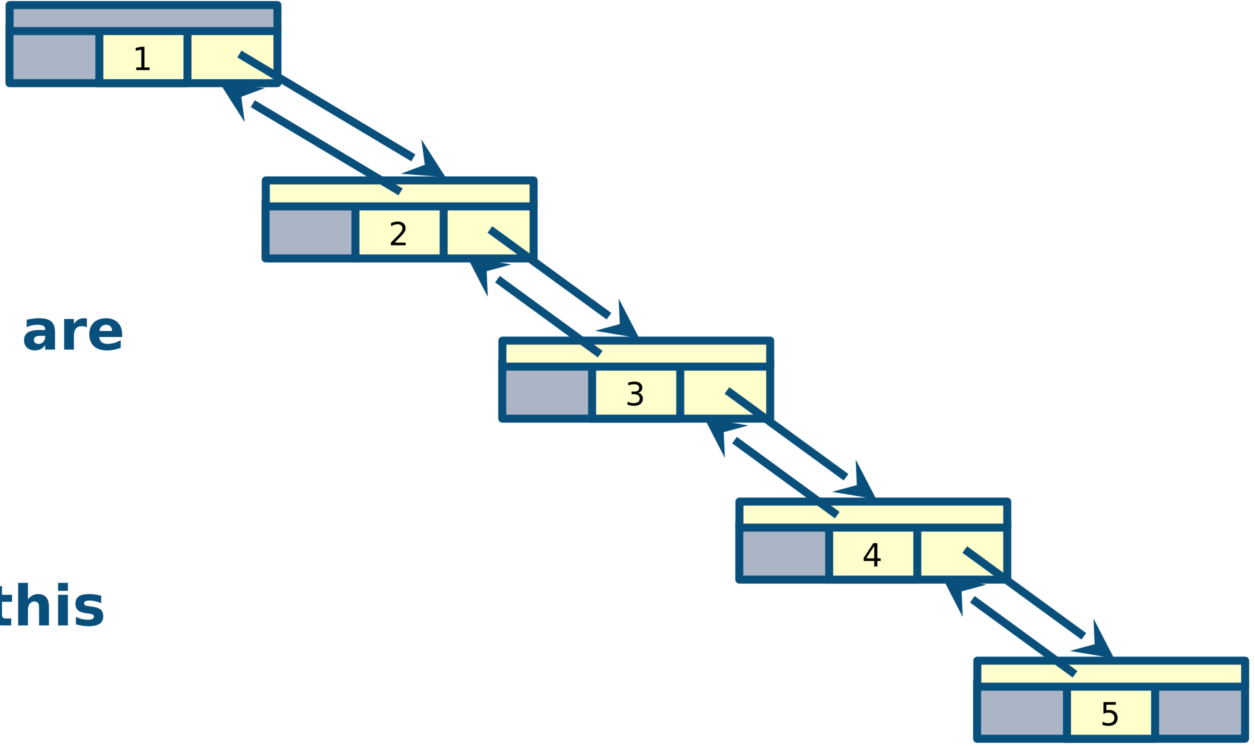
michele.sevegnani@glasgow.ac.uk

Outline

- **Recap**
- **Red-black trees**
  - Definition
  - Representation
  - Properties
  - Insertion
- **Comparison with AVL trees**

Recap

- **Each of the basic operations on a binary search tree runs in O(h) time**
  - h is the height of the tree
- **However, the height varies as items are inserted and deleted**
- **Try to insert elements 1,2,3,4,5 (in this order) into an empty BST**
  - Unbalanced tree with height 4
  - Height is O(n) in unbalanced trees



Self-balancing trees

- **Several extensions to the basic BST definition have been introduced to keep the height small as items are dynamically inserted and deleted**
  - Red-black trees (in this lecture)
  - AVL trees
  - B-trees

- **A common method to keep the tree balanced is to perform rotations after each deletion and insertion**
  - Local operation in a search tree that preserves the binary-search-tree property

Red-Black trees

- **Self-balancing binary search trees introduced by Bayer in 1972**
  - Initially known as symmetric binary B-trees
  - Bayer, Rudolf. "Symmetric binary B-trees: Data structure and maintenance algorithms." Acta informatica 290-306
- **Red/black convention introduced by Sedgewick in 1978**
  - Guibas, Leo J., and Robert Sedgewick. "A dichromatic framework for balanced trees." 19th Annual Symposium on Foundations of Computer Science. IEEE

- **Basic operations on a red-black tree take O(log n) time in the worst case**

Definition

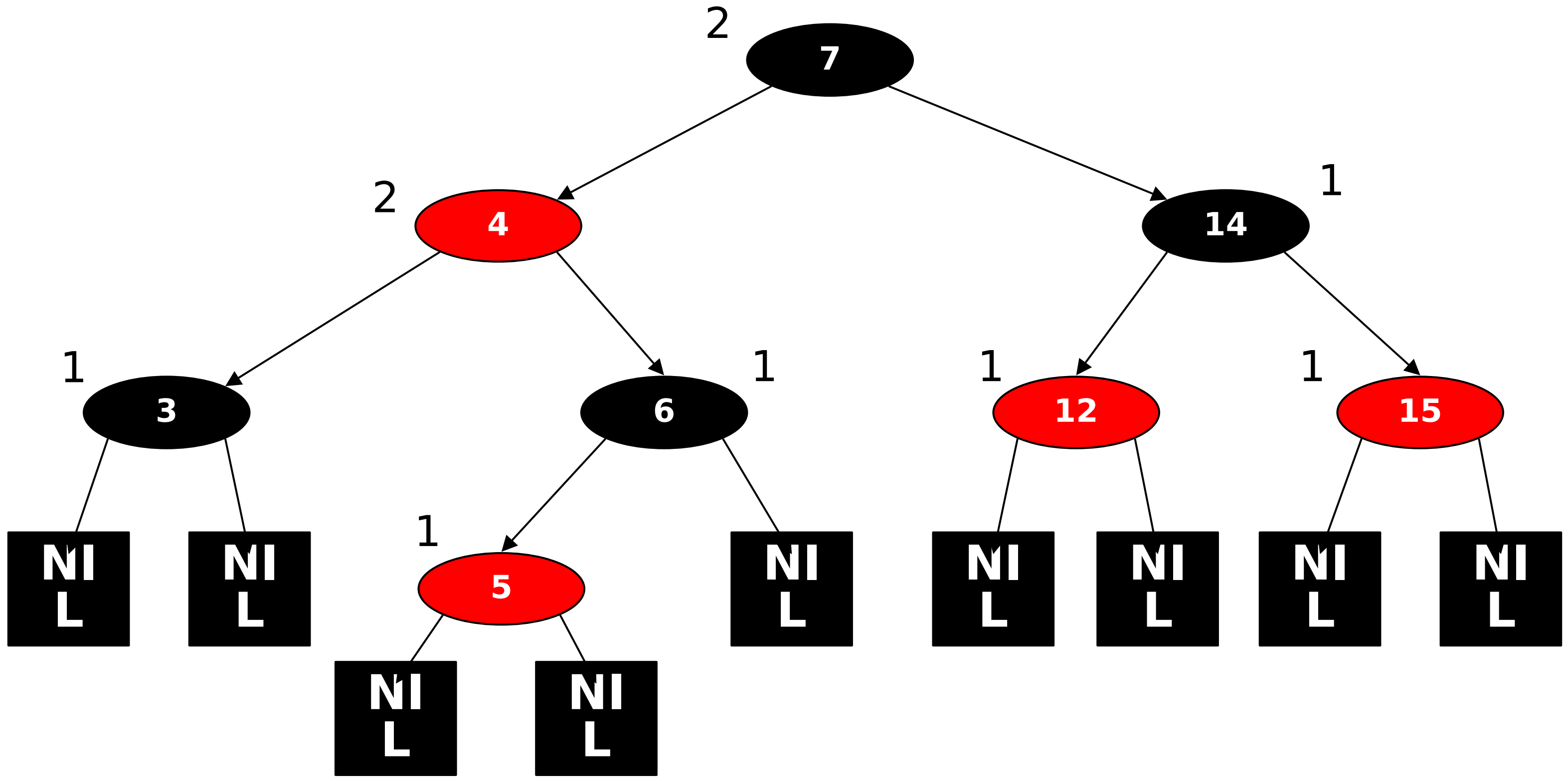
- **A red-black tree is a binary search tree with an extra attribute colour, which can be either RED or BLACK and satisfies the red-black properties**

  - [redacted]
  - [redacted]
  - [redacted]
  - [redacted]
  - [redacted]

- **All leaves (external nodes) of the tree contain the value NIL**
- **Internal nodes are the normal nodes containing regular keys**

Example

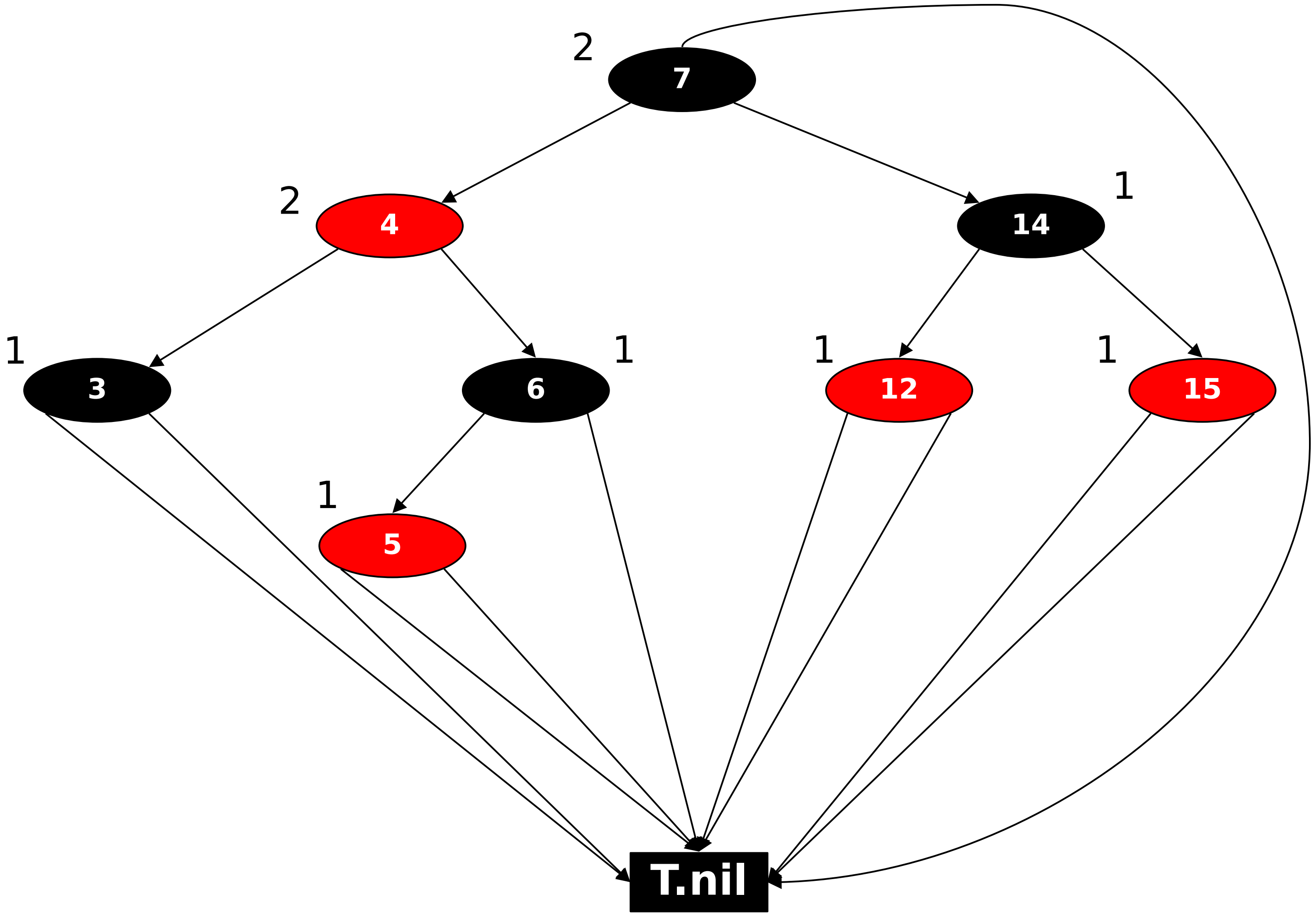
- **In this representation**
  - Pointer attribute p is not shown
- **Each NIL leaf is black**
- **Each internal node x is marked with its black-height (bh(x))**
  - [redacted] black [redacted] x [redacted]
  - [redacted]



Sentinel

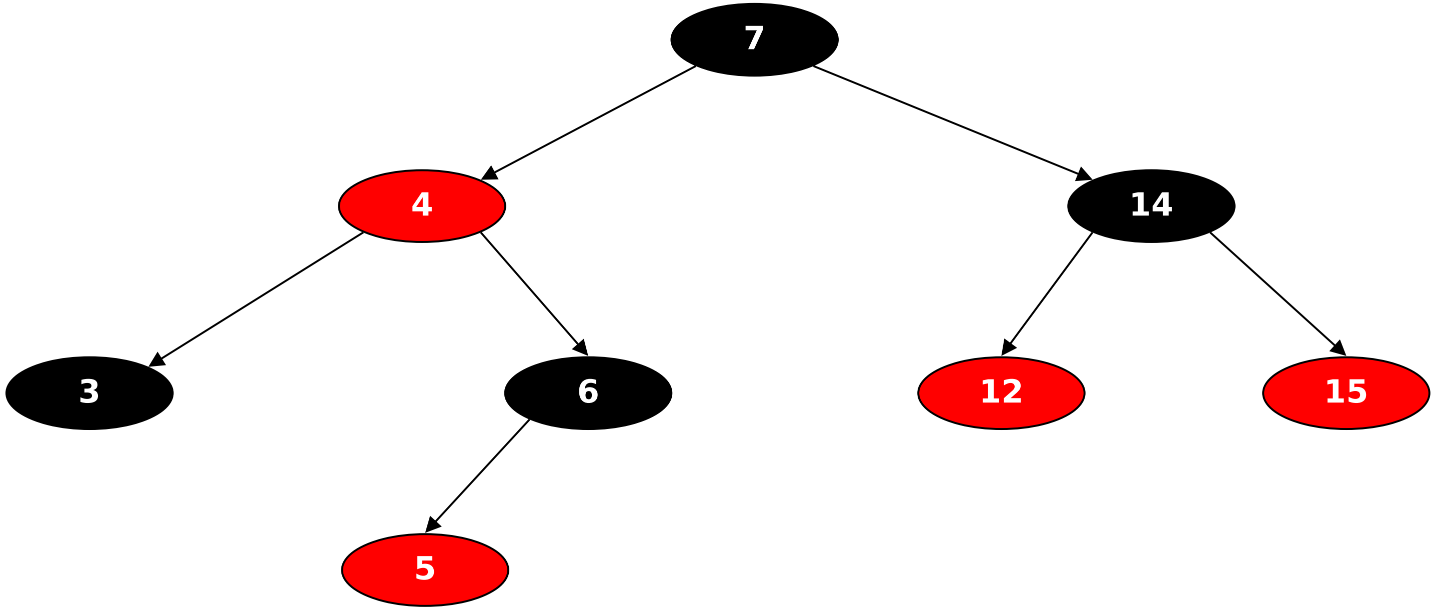
- **For a tree T, NIL nodes are represented by a single sentinel T.nil**
  - Colour attribute is BLACK
  - [redacted]
  - [redacted]
  - [redacted]

- [redacted]



Standard representation

- **Sentinel is omitted and only internal nodes are shown**
- **Black-heights are omitted**



Properties

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

- **Proof is in two steps**
  1. Prove that the subtree rooted at x contains at least $2^{bh(x)} - 1$ internal nodes
  2. Prove the bound of the height: $h \leq 2 \log(n + 1)$

Proof of 1

1. **The subtree rooted at x contains at least $2^{bh(x)} - 1$ internal nodes**
   - By induction on h(x), the height of x

- **Base h(x) = 0**
  - Then x is the leaf T.nil and its subtree contains $2^{bh(x)} - 1 = 2^0 - 1 = 0$
- **Inductive step**
  - Consider an internal node x with positive height and two children
  - Each child has black-height bh(x) (if red) or bh(x) – 1 (if black
  - [redacted] $2^{bh(x)-1} - 1$ [redacted]
  - [redacted] x [redacted] $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ [redacted]

Proof of 2

2. **Prove $h \le 2 \log(n + 1)$**
   - By property 4, at least half of the nodes on a any simple path from the root (excluded) to the leaf, must be black
   - [redacted] bh(x) [redacted] h/2. [redacted] $n \ge 2^{bh(x)} - 1$

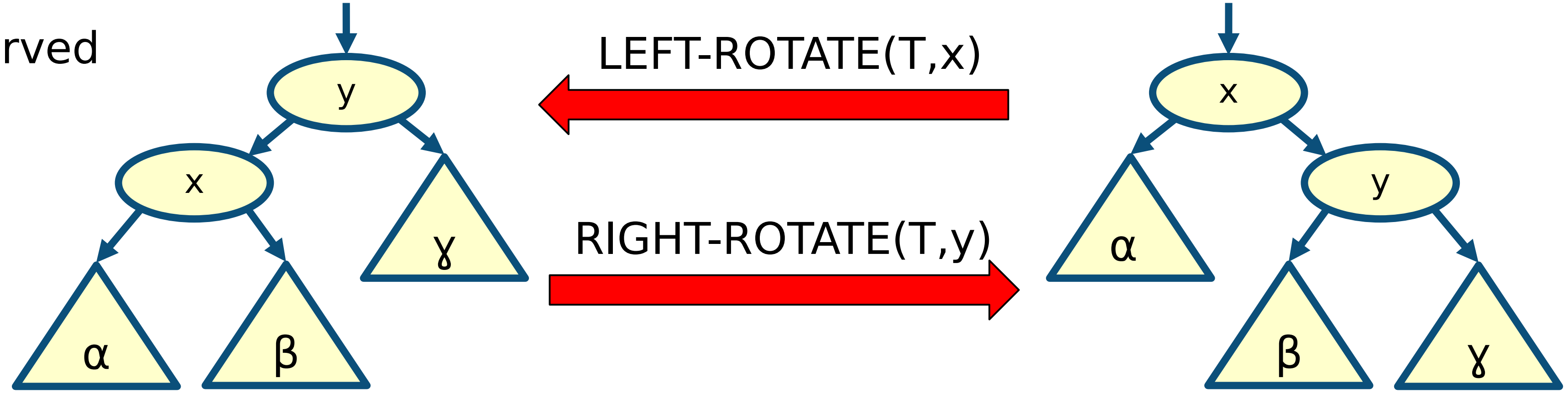
     [redacted]

     $h \le 2 \log (n + 1)$

- **This guarantees that operations on a red-black tree take $O(\log n)$ time in the worst case**

Rotations

- **Insertions and deletions may violate the red-black properties**
- **To restore these properties, we must**
  1. Change the colours of some of the nodes (later in this lecture)
  2. Change the pointer structure through left and right rotations
- **Example rotations**
  - α, β, and ɣ are arbitrary subtrees
  - BST property is preserved



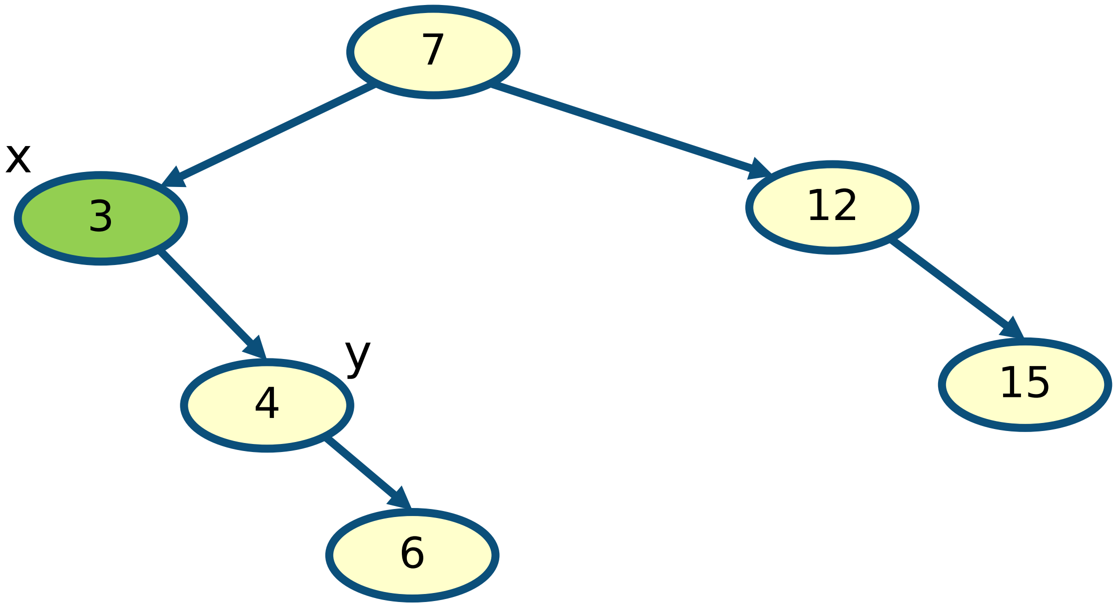
Left rotation

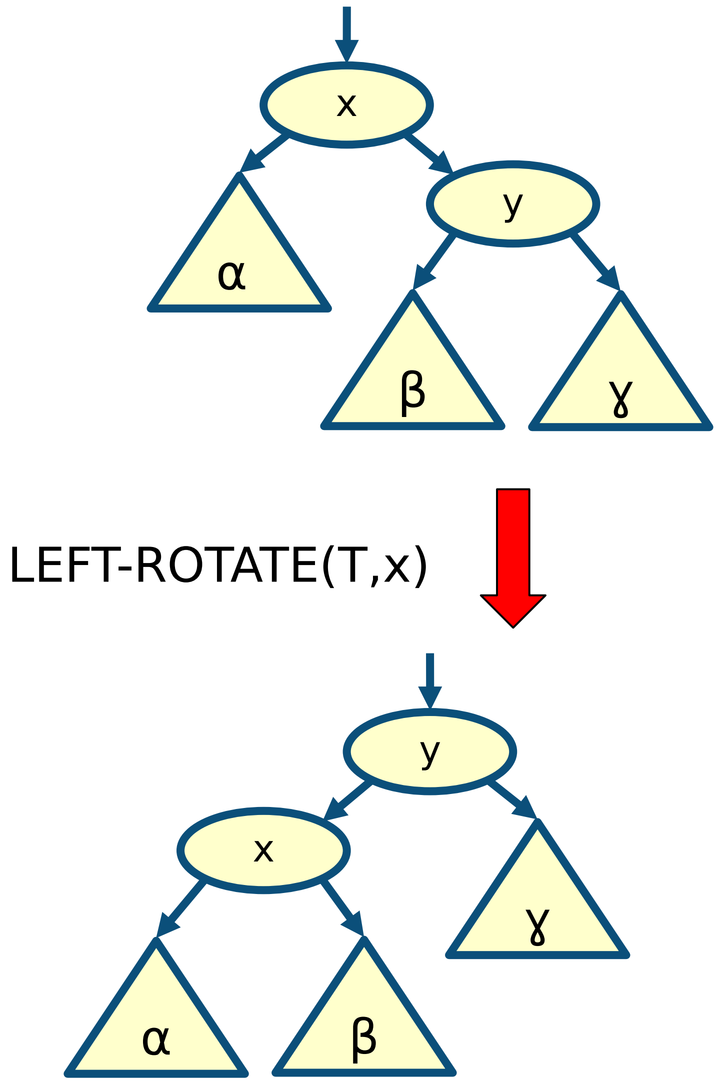
- **The left rotation "pivots" around the link from x to y**
- **Assumptions**
  - Right child of x is not T.nil
  - The root's parent is T.nil
- **Running time is O(1)**
- **The pseudocode for RIGHT-ROTATE is symmetric**

```
LEFT-ROTATE(T,x)
  y := x.right
  x.right := y.left
  if y.left != NIL
    y.left.p := x
  y.p := x.p
  if x.p = T.nil
    T.root := y
  elseif x = x.p.left
    x.p.left := y
  else x.p.right := y
  y.left := x
  x.p := y
```

Example

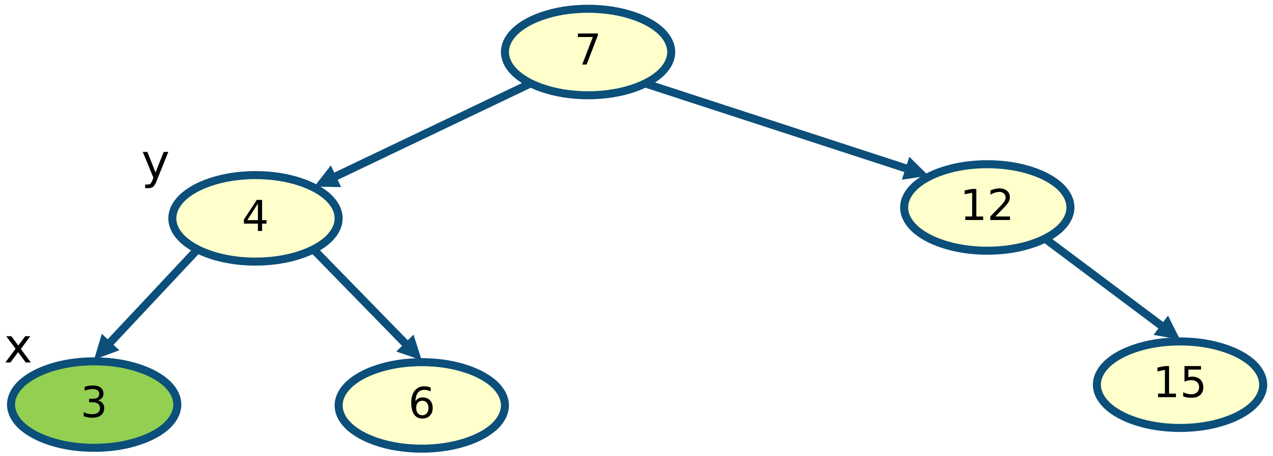
- **Try to left rotate on node with key 3**
  - α=β=T.nil
  - ɣ is the subtree rooted at 6

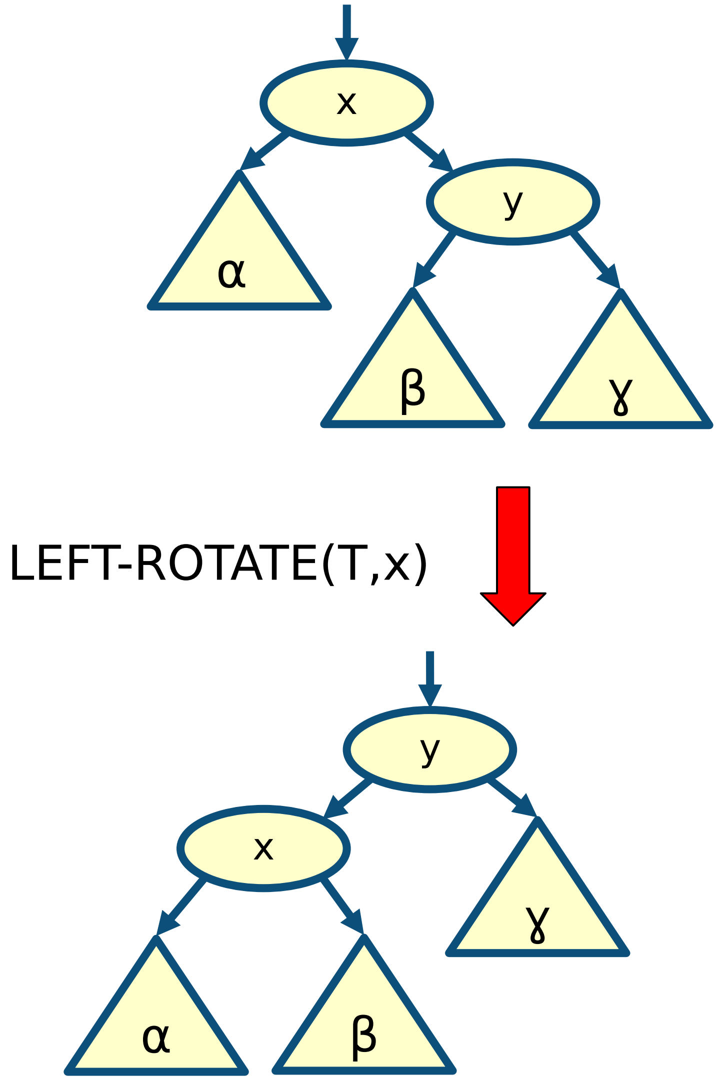




Example

- **Left rotation on node with key 3**
  - α=β=T.nil
  - ɣ is the subtree rooted at 6





LEFT-ROTATE(T,x)

Insertion

- **Like insertion for BST with a few differences**
  1. Sentinel T.nil replaces NIL
  2. New node z is coloured red
  3. FIXUP is called at the end to restore the red-black properties

- **The definition of FIXUP is quite involved**
  - We first analyse the violations to the red-black properties introduced by inserting a new red node

```
INSERT(T,z)
  y := T.nil
  x := T.root
  while x != NIL
    y := x
    if z.key < x.key
      x := x.left
    else x := x.right
  z.p := y
  if y = T.nil
    T.root := z
  elseif z.key < y.key
    y.left := z
  else y.right := z
  z.left := T.nil
  z.right := T.nil
  z.colour := RED
  FIXUP(T,z)
```

Red-black properties violations

- **When a new red node z is inserted**
  - Properties 1,3 and 5 still hold
  - Property 2 is violated if z is the root
  - Property 4 is violated if z’s parent is red

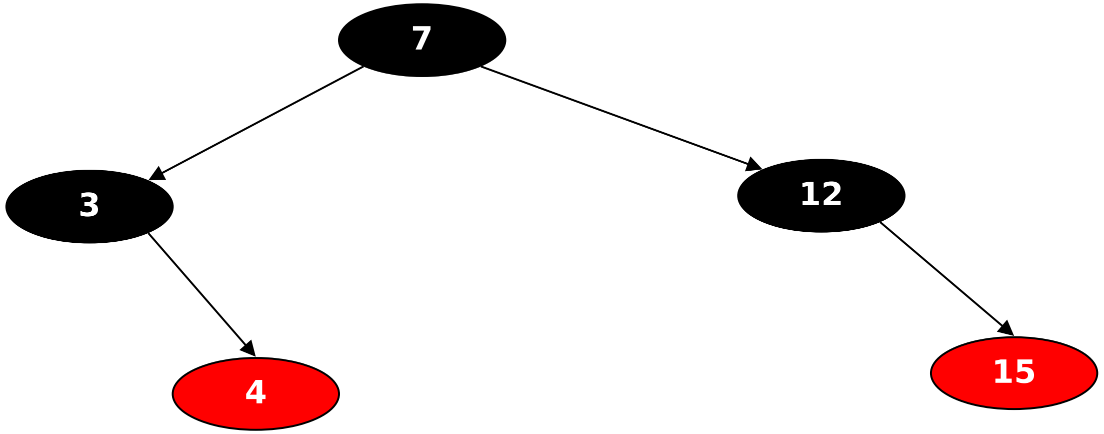
Red-black properties

1. Every node is either red or black

[redacted]

Example violation

- **Try to insert 6 in the red-black tree below**



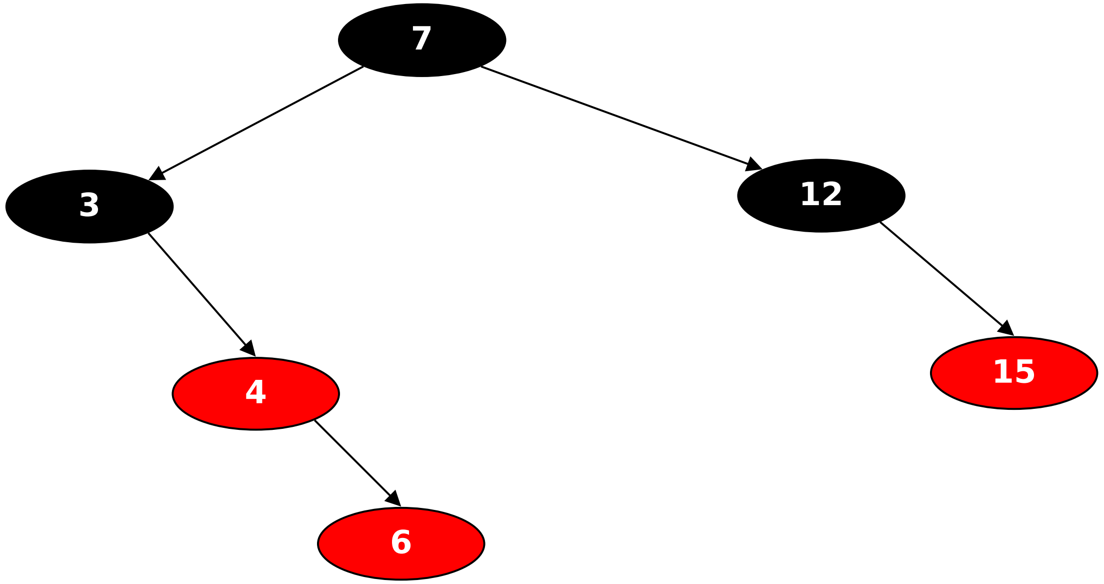
Red-black properties

1. Every node is either red or black

[redacted]

Example violation

- **Violation of Property 4**



Red-black properties

1. Every node is either red or black

[redacted]

FIXUP

- **Operation that restores the red-black tree property after an insertion or a deletion**
  - Recolour nodes and perform rotations

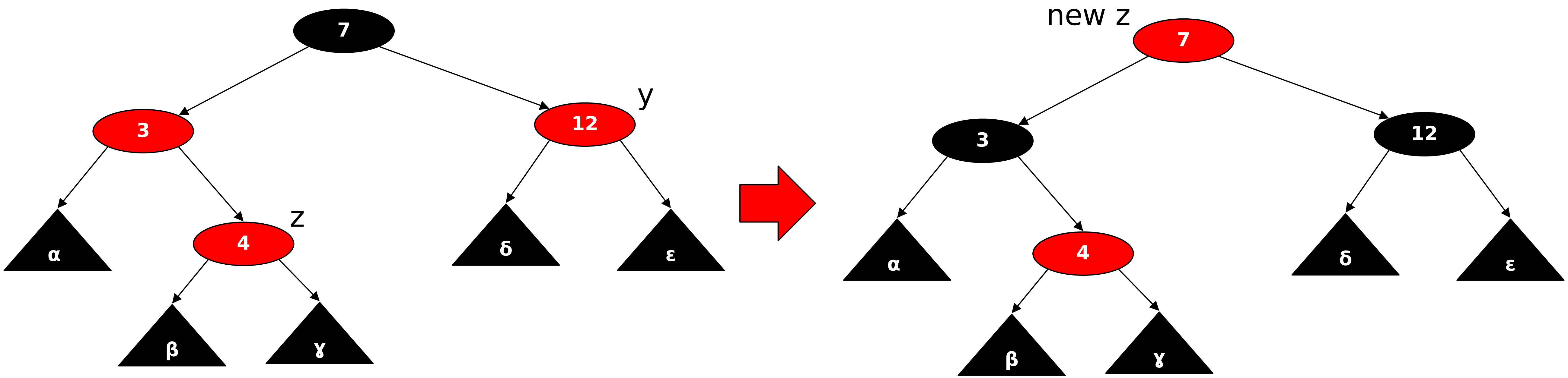
- **It consists of a while loop and three different cases**

- **We cover them one by one**

```
FIXUP(T,z)
  while z.p.colour = RED
    if z.p = z.p.p.left
      y := z.p.p.right
      if y.colour = RED
        z.p.colour := BLACK    // case 1
        y.colour := BLACK      // case 1
        z.p.p.colour := RED    // case 1
        z := z.p.p             // case 1
      else
        if z = z.p.right
          z := z.p             // case 2
          LEFT-ROTATE(T,z)     // case 2
        z.p.colour := BLACK    // case 3
        z.p.p.colour := RED    // case 3
        RIGHT-ROTATE(T,z.p.p)  // case 3
    else
    …                       // symmetric
  T.root.colour := BLACK
```

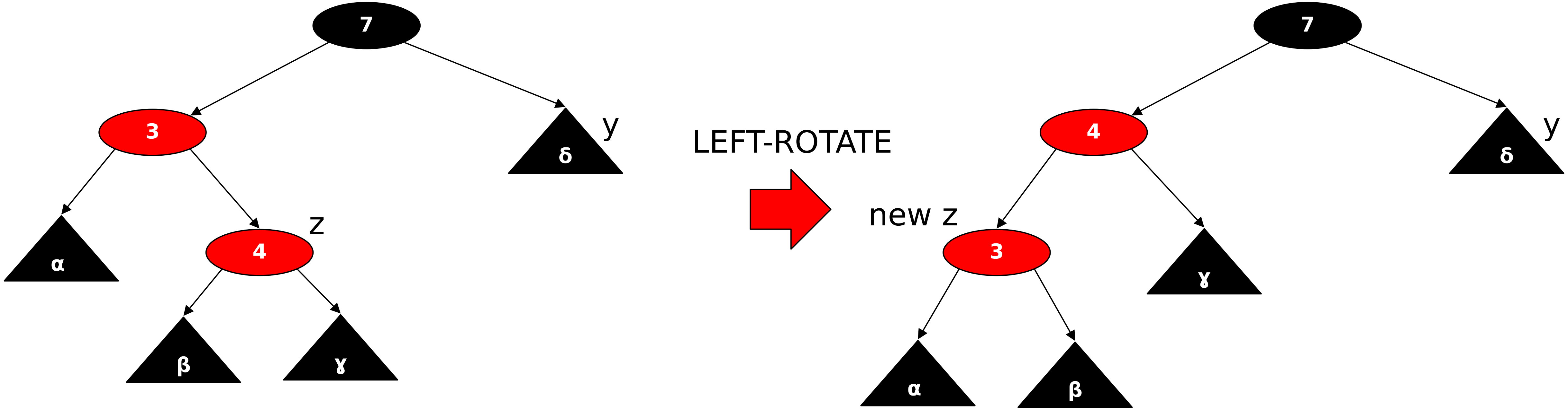
Case 1: z’s uncle y is red

- **z is a right child (same when z is a left child)**
- **Each of the subtrees α, β, γ, δ, and ε has a black root, and each has the same black-height**
- **The while loop continues with node z’s grandparent z.p.p as the new z**



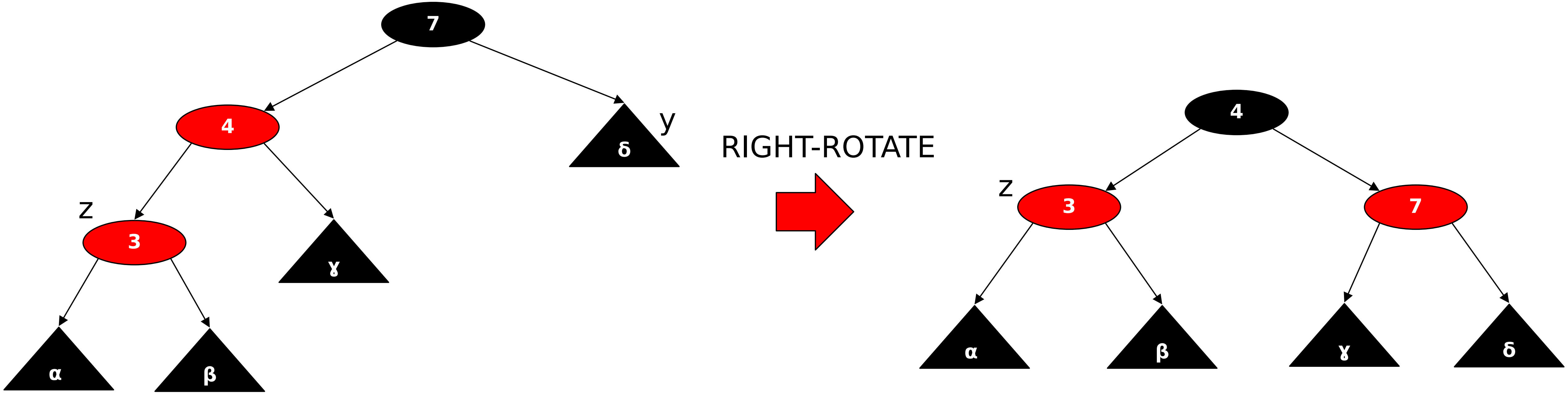
Case 2: z’s uncle y is black and z is a right child

- **Transform into case 3 with a single left rotation on z’s parent**



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

- **Perform two colour changes (z.p and z.p.p) and a right rotation on z.p.p**
- **The while loop then terminates, because there are no longer two red nodes in a row**

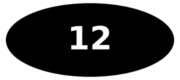


Example

- **Insert keys 12,7,3,15,4,6,14,5 in the empty red-black tree T**

Example

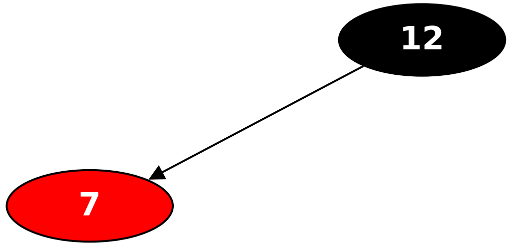
- **Insert keys 12,7,3,15,4,6,14,5 in the empty red-black tree T**
  - Insert 12
  - Loop not executed



```
FIXUP(T,z)
  while z.p.colour = RED
    if z.p = z.p.p.left
      y := z.p.p.right
      if y.colour = RED
        z.p.colour := BLACK
        y.colour := BLACK
        z.p.p.colour := RED
        z := z.p.p
      else
        if z = z.p.right
          z := z.p
          LEFT-ROTATE(T,z)
        z.p.colour := BLACK
        z.p.p.colour := RED
        RIGHT-ROTATE(T,z.p.p)
    else
    …
  T.root.colour := BLACK
```

Example

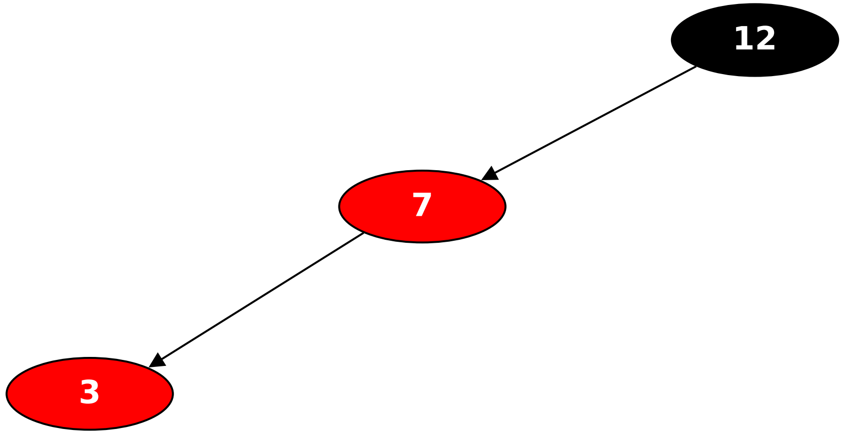
- **Insert keys 12,7,3,15,4,6,14,5 in the empty red-black tree T**
  - Insert 7
  - Loop not executed



```
FIXUP(T,z)
  while z.p.colour = RED
    if z.p = z.p.p.left
      y := z.p.p.right
      if y.colour = RED
        z.p.colour := BLACK
        y.colour := BLACK
        z.p.p.colour := RED
        z := z.p.p
      else
        if z = z.p.right
          z := z.p
          LEFT-ROTATE(T,z)
        z.p.colour := BLACK
        z.p.p.colour := RED
        RIGHT-ROTATE(T,z.p.p)
    else
    …
  T.root.colour := BLACK
```

Example

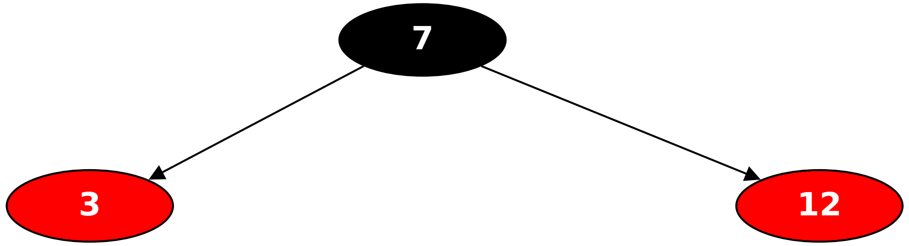
- **Insert keys 12,7,3,15,4,6,14,5 in the empty red-black tree T**
  - Insert 3
  - Case 3: RIGHT-ROTATE



```
FIXUP(T,z)
  while z.p.colour = RED
    if z.p = z.p.p.left
      y := z.p.p.right
      if y.colour = RED
        z.p.colour := BLACK
        y.colour := BLACK
        z.p.p.colour := RED
        z := z.p.p
      else
        if z = z.p.right
          z := z.p
          LEFT-ROTATE(T,z)
        z.p.colour := BLACK
        z.p.p.colour := RED
        RIGHT-ROTATE(T,z.p.p)
    else
    …
  T.root.colour := BLACK
```

Example

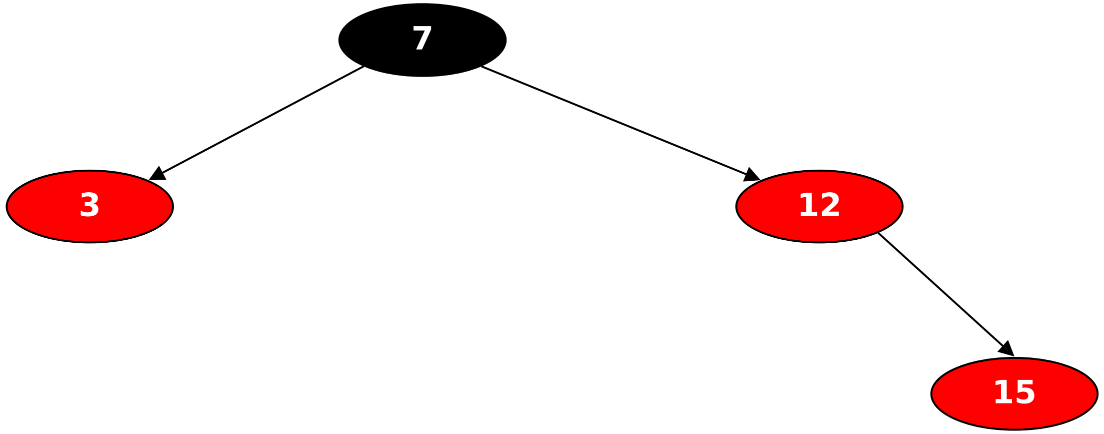
- **Insert keys 12,7,3,15,4,6,14,5 in the empty red-black tree T**
  - After FIXUP



```
FIXUP(T,z)
  while z.p.colour = RED
    if z.p = z.p.p.left
      y := z.p.p.right
      if y.colour = RED
        z.p.colour := BLACK
        y.colour := BLACK
        z.p.p.colour := RED
        z := z.p.p
      else
        if z = z.p.right
          z := z.p
          LEFT-ROTATE(T,z)
        z.p.colour := BLACK
        z.p.p.colour := RED
        RIGHT-ROTATE(T,z.p.p)
    else
    …
  T.root.colour := BLACK
```

Example

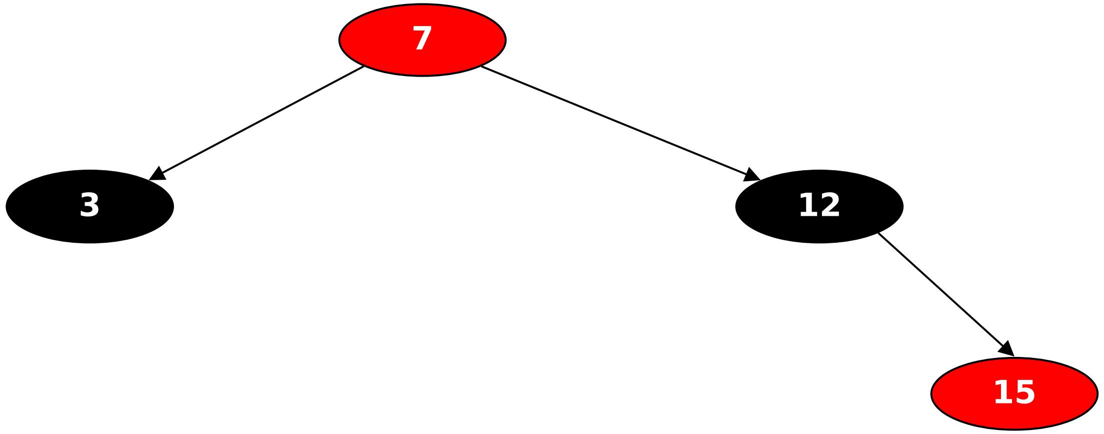
- **Insert keys 12,7,3,15,4,6,14,5 in the empty red-black tree T**
  - Insert 15
  - Case 1: push blackness down from grandparent



```
FIXUP(T,z)
  while z.p.colour = RED
    if z.p = z.p.p.left
      y := z.p.p.right
      if y.colour = RED
        z.p.colour := BLACK
        y.colour := BLACK
        z.p.p.colour := RED
        z := z.p.p
      else
        if z = z.p.right
          z := z.p
          LEFT-ROTATE(T,z)
        z.p.colour := BLACK
        z.p.p.colour := RED
        RIGHT-ROTATE(T,z.p.p)
    else
    …
  T.root.colour := BLACK
```

Example

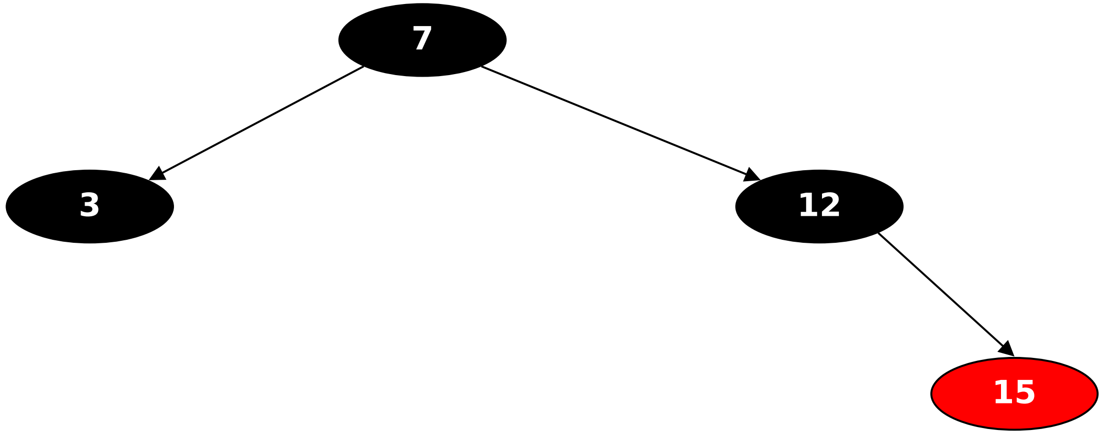
- **Insert keys 12,7,3,15,4,6,14,5 in the empty red-black tree T**
  - Insert 15
  - Case 1: push blackness down from grandparent



```
FIXUP(T,z)
  while z.p.colour = RED
    if z.p = z.p.p.left
      y := z.p.p.right
      if y.colour = RED
        z.p.colour := BLACK
        y.colour := BLACK
        z.p.p.colour := RED
        z := z.p.p
      else
        if z = z.p.right
          z := z.p
          LEFT-ROTATE(T,z)
        z.p.colour := BLACK
        z.p.p.colour := RED
        RIGHT-ROTATE(T,z.p.p)
    else
    …
  T.root.colour := BLACK
```

Example

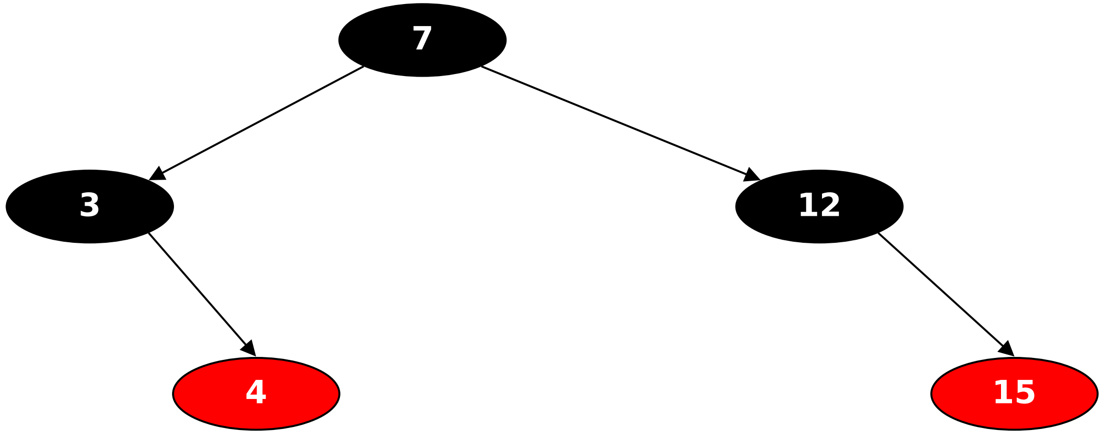
- **Insert keys 12,7,3,15,4,6,14,5 in the empty red-black tree T**
  - Restore root



```
FIXUP(T,z)
  while z.p.colour = RED
    if z.p = z.p.p.left
      y := z.p.p.right
      if y.colour = RED
        z.p.colour := BLACK
        y.colour := BLACK
        z.p.p.colour := RED
        z := z.p.p
      else
        if z = z.p.right
          z := z.p
          LEFT-ROTATE(T,z)
        z.p.colour := BLACK
        z.p.p.colour := RED
        RIGHT-ROTATE(T,z.p.p)
    else
      …
  T.root.colour := BLACK
```

Example

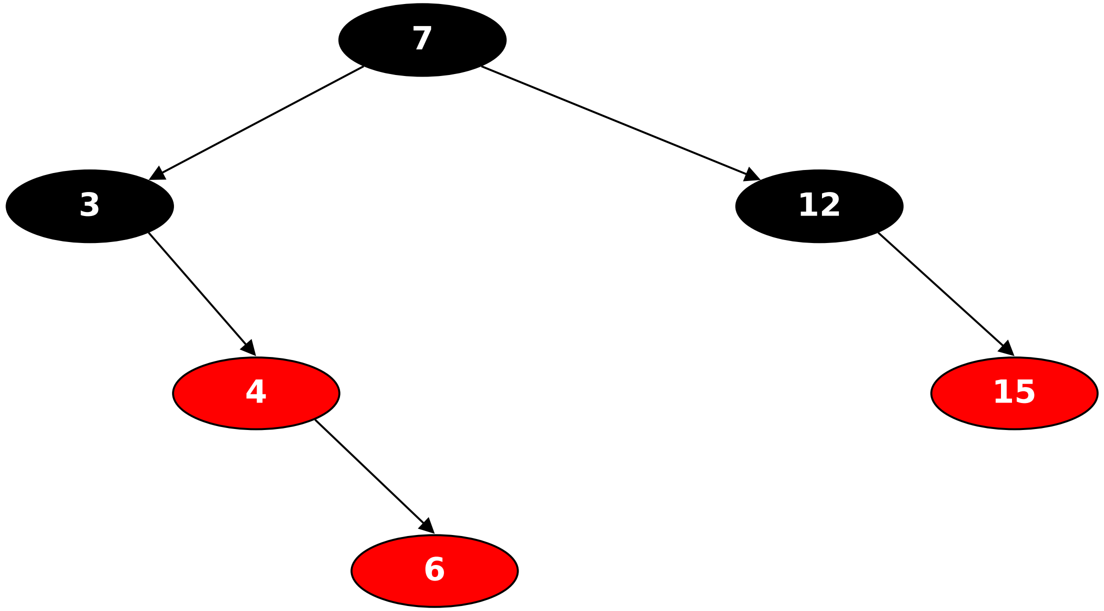
- **Insert keys 12,7,3,15,4,6,14,5 in the empty red-black tree T**
  - Insert 4
  - No violation



```
FIXUP(T,z)
  while z.p.colour = RED
    if z.p = z.p.p.left
      y := z.p.p.right
      if y.colour = RED
        z.p.colour := BLACK
        y.colour := BLACK
        z.p.p.colour := RED
        z := z.p.p
      else
        if z = z.p.right
          z := z.p
          LEFT-ROTATE(T,z)
        z.p.colour := BLACK
        z.p.p.colour := RED
        RIGHT-ROTATE(T,z.p.p)
    else
    …
  T.root.colour := BLACK
```

Example

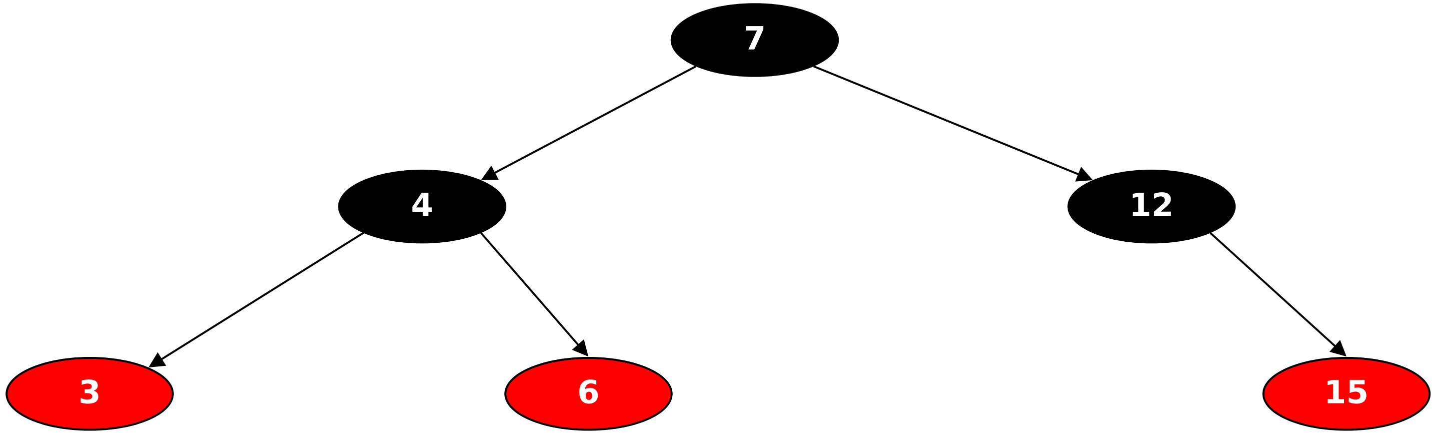
- **Insert keys 12,7,3,15,4,6,14,5 in the empty red-black tree T**
  - Insert 6
  - Case 3 (symmetric): LEFT-ROTATE



```
FIXUP(T,z)
  while z.p.colour = RED
    if z.p = z.p.p.left
      y := z.p.p.right
      if y.colour = RED
        z.p.colour := BLACK
        y.colour := BLACK
        z.p.p.colour := RED
        z := z.p.p
      else
        if z = z.p.right
          z := z.p
          LEFT-ROTATE(T,z)
        z.p.colour := BLACK
        z.p.p.colour := RED
        RIGHT-ROTATE(T,z.p.p)
    else
    …
  T.root.colour := BLACK
```

Example

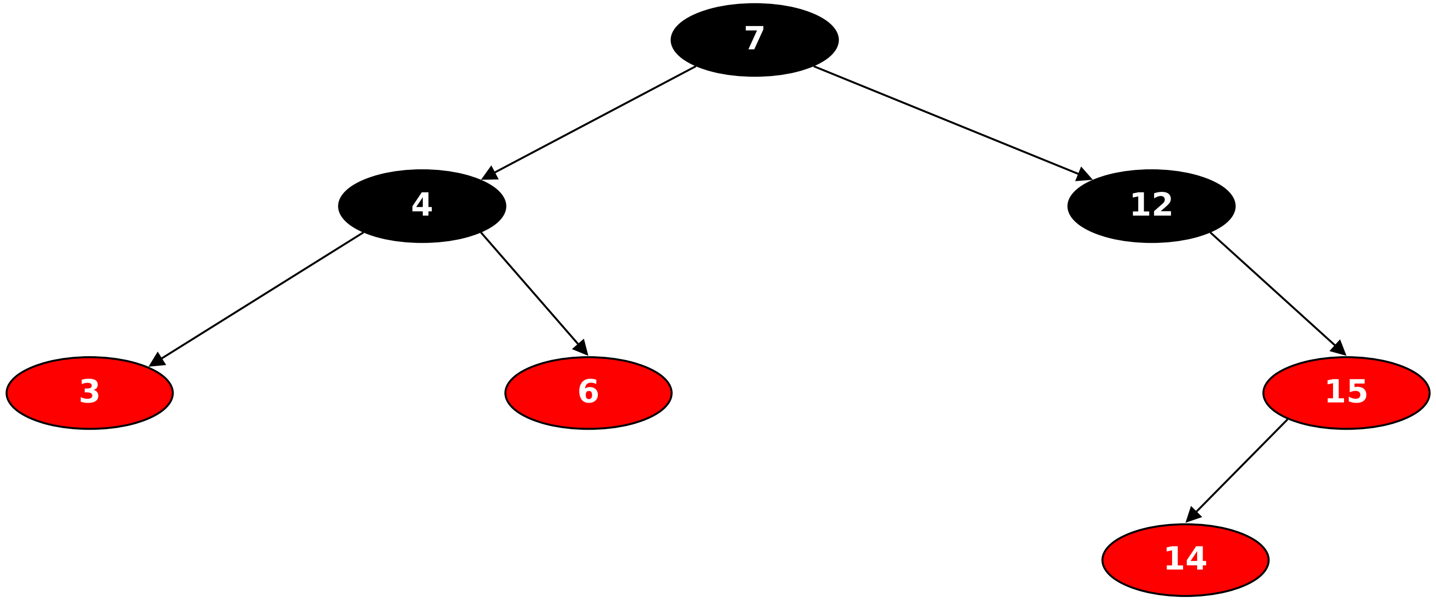
- **Insert keys 12,7,3,15,4,6,14,5 in the empty red-black tree T**
  - After FIXUP



```
FIXUP(T,z)
  while z.p.colour = RED
    if z.p = z.p.p.left
      y := z.p.p.right
      if y.colour = RED
        z.p.colour := BLACK
        y.colour := BLACK
        z.p.p.colour := RED
        z := z.p.p
      else
        if z = z.p.right
          z := z.p
          LEFT-ROTATE(T,z)
        z.p.colour := BLACK
        z.p.p.colour := RED
        RIGHT-ROTATE(T,z.p.p)
    else
    …
  T.root.colour := BLACK
```

Example

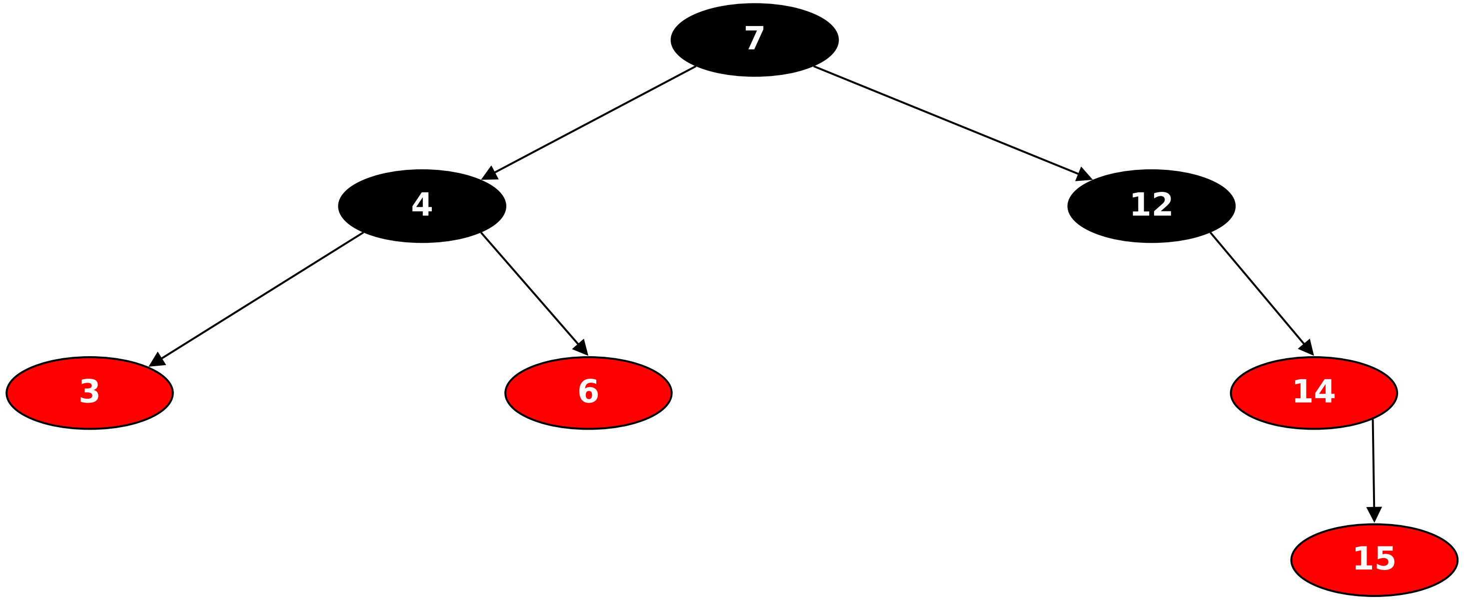
- **Insert keys 12,7,3,15,4,6,14,5 in the empty red-black tree T**
  - Insert 14
  - Case 2 (symmetric): RIGHT-ROTATE



```
FIXUP(T,z)
  while z.p.colour = RED
    if z.p = z.p.p.left
      y := z.p.p.right
      if y.colour = RED
        z.p.colour := BLACK
        y.colour := BLACK
        z.p.p.colour := RED
        z := z.p.p
      else
        if z = z.p.right
          z := z.p
          LEFT-ROTATE(T,z)
        z.p.colour := BLACK
        z.p.p.colour := RED
        RIGHT-ROTATE(T,z.p.p)
    else
    …
  T.root.colour := BLACK
```

Example

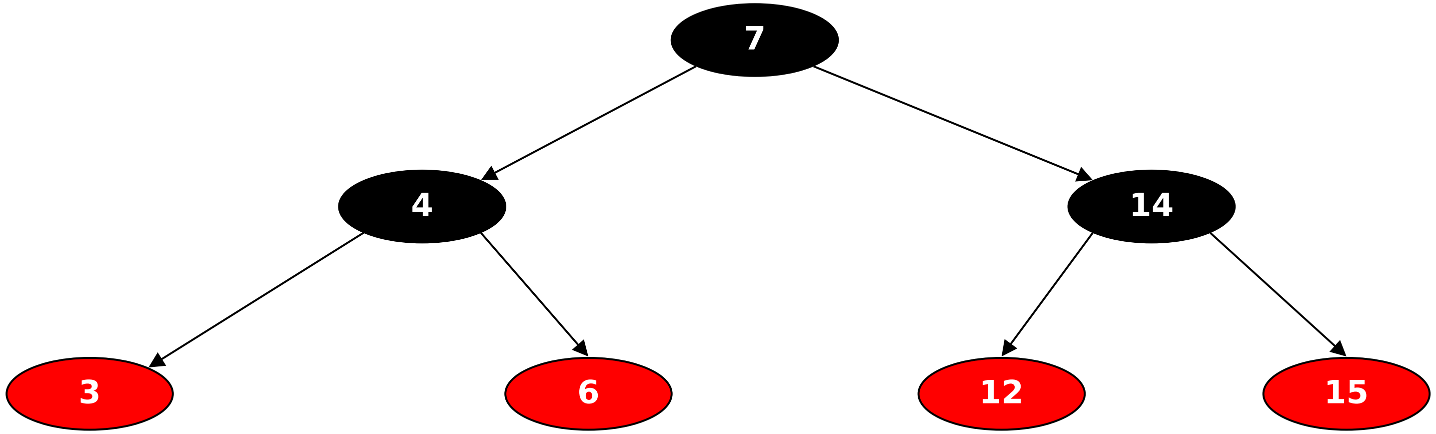
- **Insert keys 12,7,3,15,4,6,14,5 in the empty red-black tree T**
  - Case 3 (symmetric): LEFT-ROTATE



```
FIXUP(T,z)
  while z.p.colour = RED
    if z.p = z.p.p.left
      y := z.p.p.right
      if y.colour = RED
        z.p.colour := BLACK
        y.colour := BLACK
        z.p.p.colour := RED
        z := z.p.p
      else
        if z = z.p.right
          z := z.p
          LEFT-ROTATE(T,z)
        z.p.colour := BLACK
        z.p.p.colour := RED
        RIGHT-ROTATE(T,z.p.p)
    else
      …
  T.root.colour := BLACK
```

Example

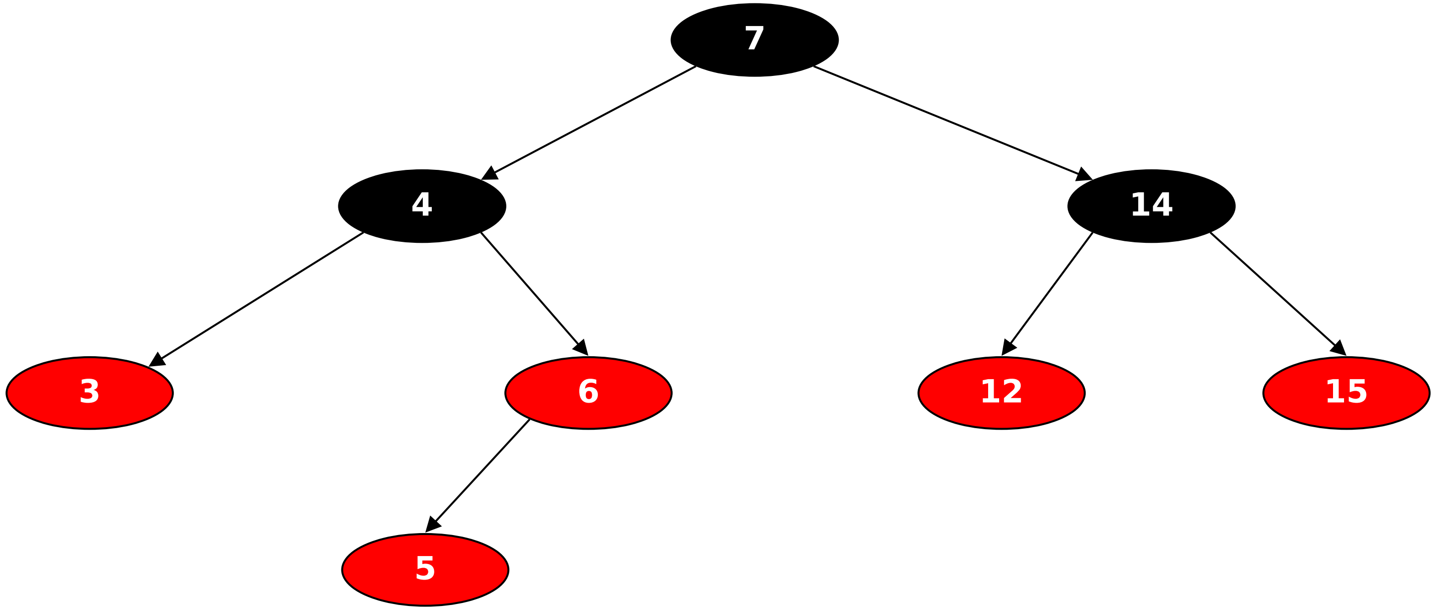
- **Insert keys 12,7,3,15,4,6,14,5 in the empty red-black tree T**
  - After FIXUP



```
FIXUP(T,z)
  while z.p.colour = RED
    if z.p = z.p.p.left
      y := z.p.p.right
      if y.colour = RED
        z.p.colour := BLACK
        y.colour := BLACK
        z.p.p.colour := RED
        z := z.p.p
      else
        if z = z.p.right
          z := z.p
          LEFT-ROTATE(T,z)
        z.p.colour := BLACK
        z.p.p.colour := RED
        RIGHT-ROTATE(T,z.p.p)
    else
    …
  T.root.colour := BLACK
```

Example

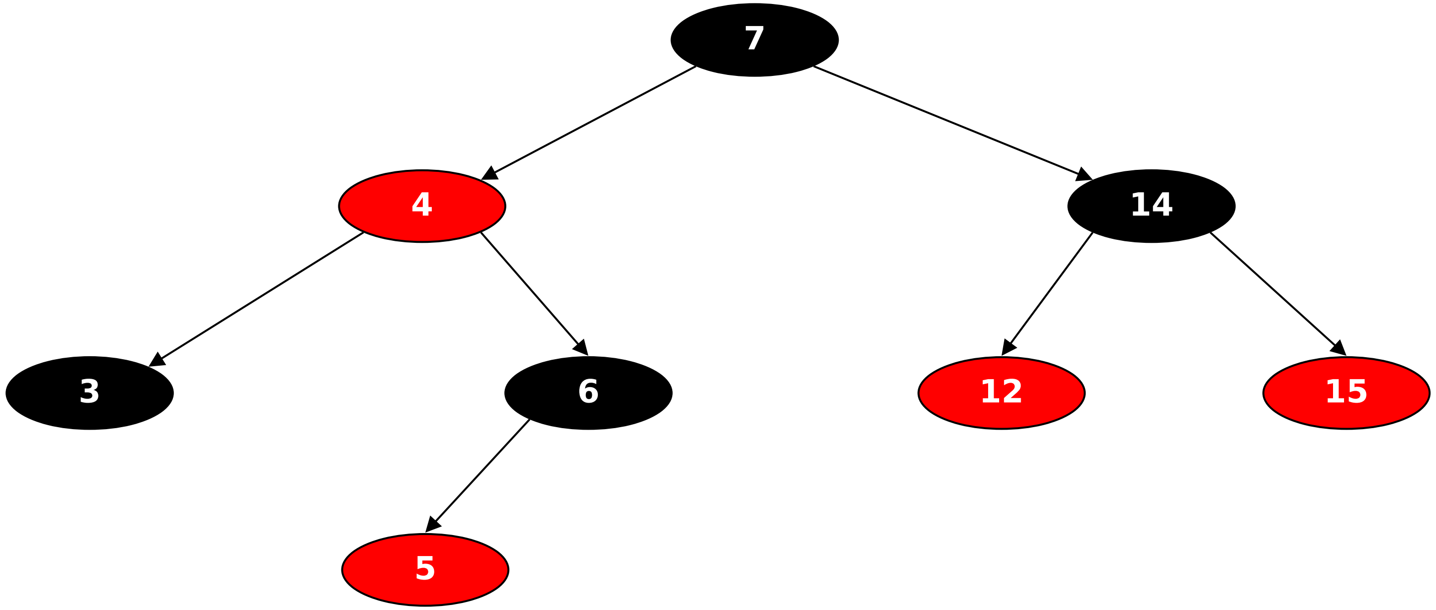
- **Insert keys 12,7,3,15,4,6,14,5 in the empty red-black tree T**
  - Insert 5
  - Case 1: push blackness down from grandparent



```
FIXUP(T,z)
  while z.p.colour = RED
    if z.p = z.p.p.left
      y := z.p.p.right
      if y.colour = RED
        z.p.colour := BLACK
        y.colour := BLACK
        z.p.p.colour := RED
        z := z.p.p
      else
        if z = z.p.right
          z := z.p
          LEFT-ROTATE(T,z)
        z.p.colour := BLACK
        z.p.p.colour := RED
        RIGHT-ROTATE(T,z.p.p)
    else
    …
  T.root.colour := BLACK
```

Example

- **Insert keys 12,7,3,15,4,6,14,5 in the empty red-black tree T**
    - After FIXUP



```
FIXUP(T,z)
  while z.p.colour = RED
    if z.p = z.p.p.left
      y := z.p.p.right
      if y.colour = RED
        z.p.colour := BLACK
        y.colour := BLACK
        z.p.p.colour := RED
        z := z.p.p
      else
        if z = z.p.right
          z := z.p
          LEFT-ROTATE(T,z)
        z.p.colour := BLACK
        z.p.p.colour := RED
        RIGHT-ROTATE(T,z.p.p)
    else
    …
  T.root.colour := BLACK
```

Analysis of INSERT

- **Since the height of a red-black tree on n nodes is O(log n) the first part of INSERT takes O(log n) time**

- **In FIXUP, the while loop repeats only if case 1 occurs, and then the pointer z moves two levels up the tree**
  - The total number of times the while loop can be executed is O(log n)

- **Therefore, INSERT takes a total of O(log n) time**
  - Observe that it never performs more than two rotations, since the while loop terminates if case 2 or case 3 is executed

Deletion of node x

- **Complex operation consisting of four cases**
  1. x’s sibling w is red
  2. x’s sibling w is [redacted]

  [redacted]

  [redacted]

- [redacted]

- [redacted]

- [redacted]

AVL tree

- **G. Adelson-Velskii and E.M. Landis, "An algorithm for the organization of information." Doklady Akademii Nauk SSSR, 146:263–266, 1962**

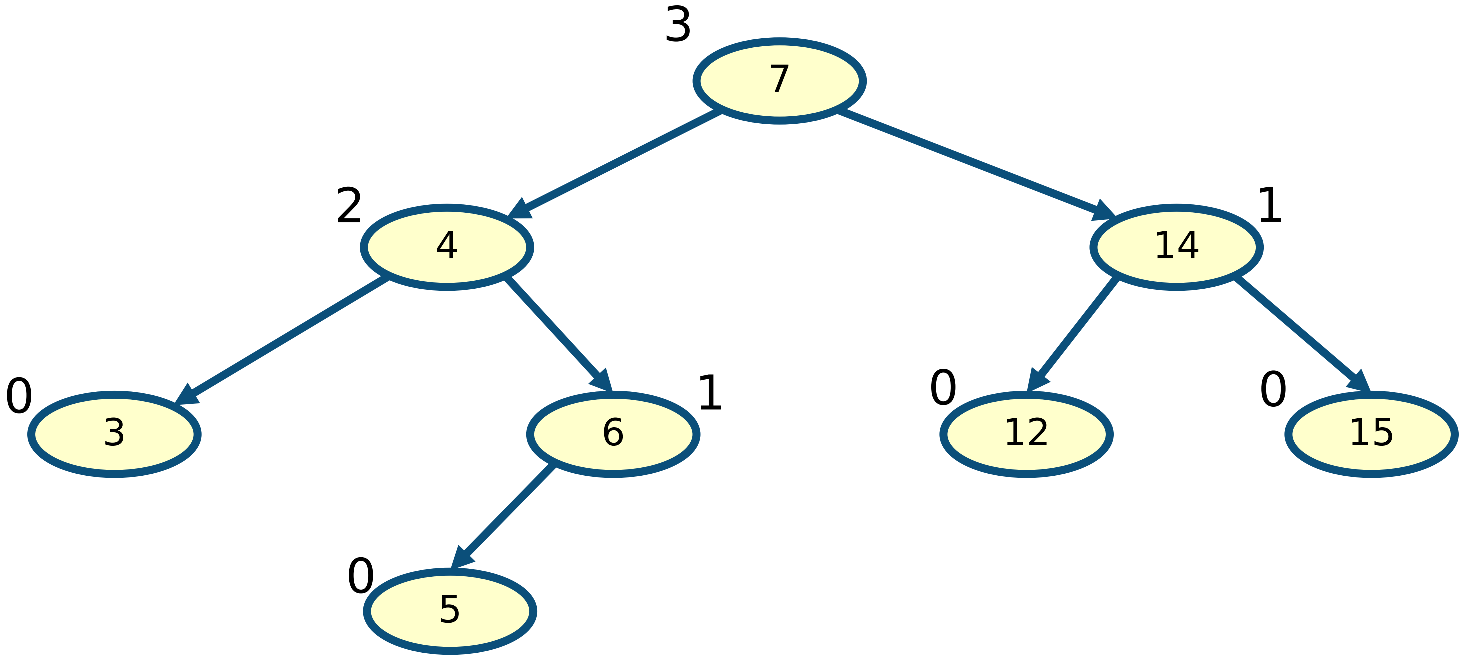
- **An AVL tree is a binary search tree that is height balanced: for each node x, the heights of the left and right subtrees of x differ by at most 1**

- **To implement an AVL tree, we maintain an extra attribute in each node: x.h is the height of node x**

- **T.root points to the root node**

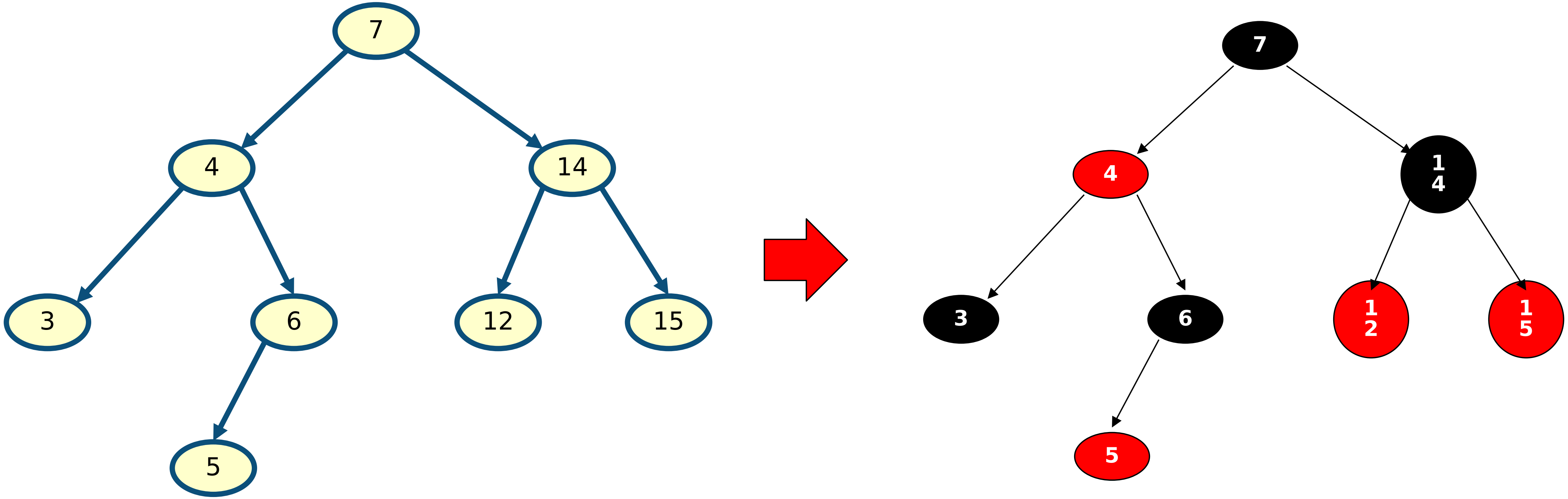
Example

- **Height at each node is marked**



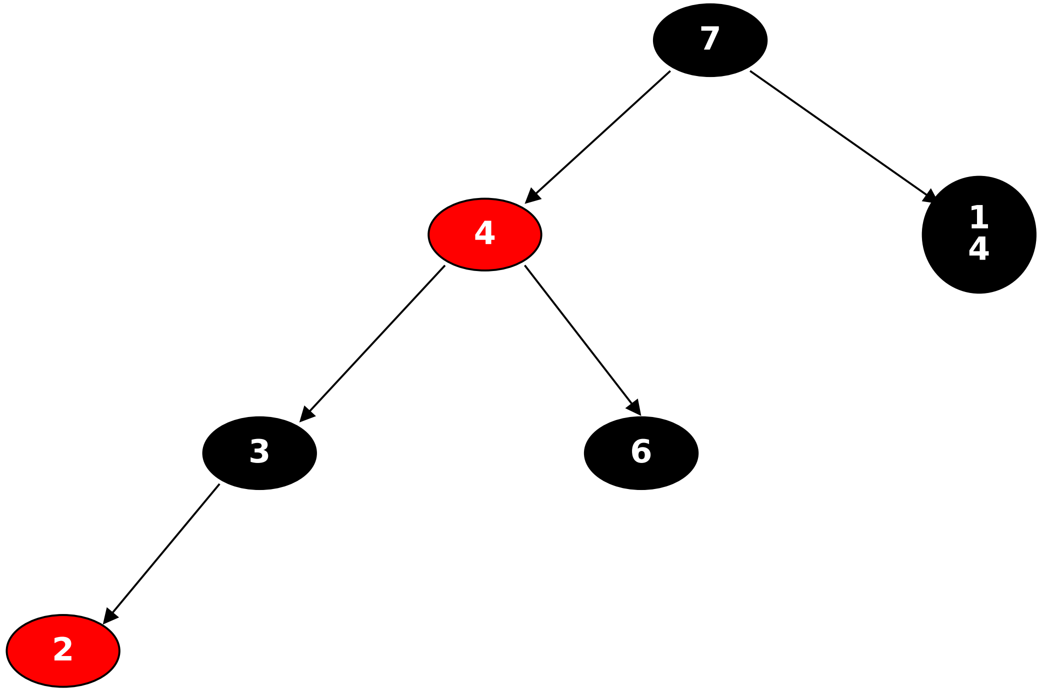
AVL trees are a subset of red-black trees

- **AVL trees can be coloured red-black**
- **There are red-black trees which are not AVL balanced**



Example red-black tree, but not AVL tree

- **Height of left and right subtrees of 7 differ by more than 1**
- **AVL requires a right rotation on 7**



Comparison

- **AVL trees are more rigidly balanced than red-black trees**
  - $h \leq 2 \log (n + 1)$ in red-black trees
  - $h \leq 1.44 \log n$ in AVL trees

- **AVL trees provide faster lookups than red-black tree**
  - Used in databases

- **Red-black trees provide faster insertion and removal operations than AVL trees as fewer rotations are performed**
  - Used in real-time applications and the current Linux kernel scheduler
  - Used in most libraries to implement common ADTs such as Set, Multiset, and Map

Summary

- **Red-black trees**
  - Definition
  - Representation
  - Properties
  - Insertion
- **Comparison with AVL trees**