11.7 Red-black binary search trees

Red-black trees are balanced binary search trees.

The tree is modified when making additions and removals in order to keep the tree balanced and to make sure that searching is never inefficient, not even when the elements are added to the tree in an inconvenient order.

- a red-black tree can never be reduced to a list like with unbalanced binary search trees

The basic idea of red-black trees:

- each node contains one extra bit: its *colour*
  - either *red* or *black*
- the other fields are the good old key, $left$, $right$ and p
  - we're going to leave the satellite data out in order to keep the main points clear and to make sure they are not lost behind details
- the colour fields are used to maintain the *red-black*

invariant, which quarantees that the height of the tree is kept in $\Theta(\lg n)$

The invariant of red-black trees:

1. If the node is red, it either has
   - no children, or
   - it has two children, both of them black.
2. for each node, all paths from a node down to descendant leaves contains the same number of black nodes.
3. The root is black.

The *black-height* of the node x $bh(x)$ is the amount of black nodes on the path from it down to the node with 1 or 0 children.

- by property 2 in the invariant, the black height of each node is well defined and unique
- all alternate paths from the root downwards have the same amount of black nodes
- the black height of the tree is the black height of its root

The maximum height of a red-black tree

- denote the height = h and the amount of nodes = n
- at least half the nodes on any path from the root down to a leaf ($\lfloor \frac{h}{2} \rfloor + 1$) are black (properties 1 and 3 in the invariant)
- there is the same amount of black nodes on each path from the root down to a leaf (property 2 of the invariant)

  $\Rightarrow$ at least $\lfloor \frac{h}{2} \rfloor + 1$ upmost levels are full

  $\Rightarrow n \geq 2^{\frac{h}{2}}$

  $\Rightarrow h \leq 2 \lg n$

Therefore, the invariant does quarantee that the height of the tree is kept logarithmic to the number of the nodes in the tree.
$\Rightarrow$ The operations of a dynamic set SEARCH, MINIMUM, MAXIMUM, SUCCESSOR and PREDECESSOR can be made in $O(\lg n)$ time with red-black trees.

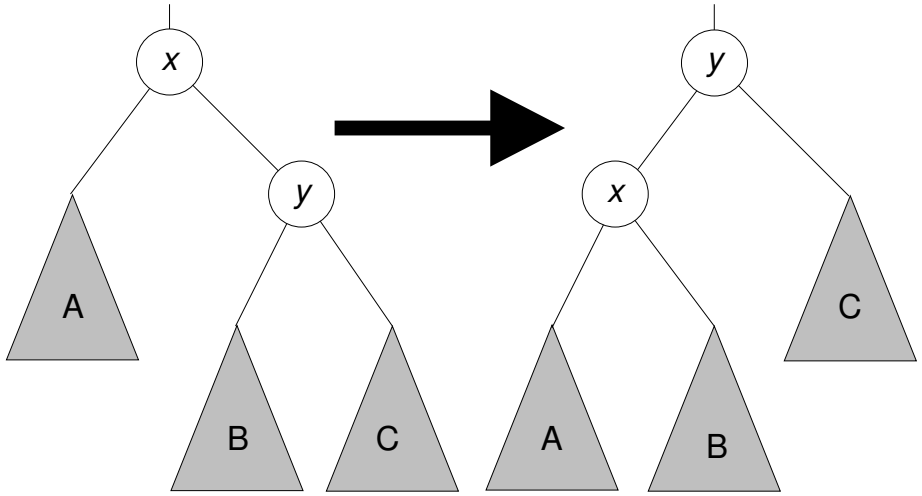
- the operations work with binary search trees in $O(h)$ time, and a red-black tree is a binary search tree with $h = \Theta(\lg n)$

The same addition and deletion algorithms from the binary search trees cannot be used with red-black trees as they might break the invariant.

Let's use the algorithms RB-INSERT and RB-DELETE instead.

- operations RB-INSERT and RB-DELETE are based on *rotations*
- there are two rotations: to the left and to the right
- they modify the structure of the tree so that the basic properties of the binary search trees are maintained for each node

The idea of a left rotation:



- rotation is done to the left
  - assumes that the nodes x and y exist
- right rotate similarly
  - $left$ and $right$ have switched places

```
LEFT-ROTATE(T, x)
1   y := x→right; x→right := y→left
2   if y→left ≠ NIL then
3       y→left→p := x
4   y→p := x→p
5   if x→p = NIL then
6       T.root := y
7   else if x = x→p→left then
8       x→p→left := y
9   else
10      x→p→right := y
11  y→left := x; x→p := y
```

- the running-time of both is $\Theta(1)$
- only the pointers are modified

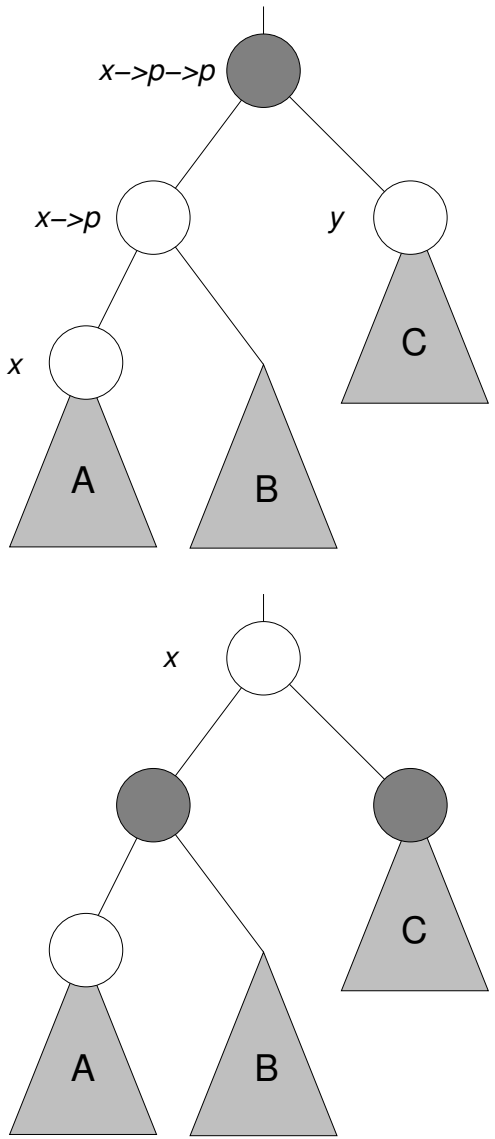
The basic idea of the addition

- first a new node is added in the same way as into a ordinary binary search tree
- then it is coloured red
- which basic properties of red-black trees could be violated if the addition is done this way?

1. is broken by the node added if its parent is red, otherwise it cannot be broken
2. doesn't get broken since the amounts and the locations of the black node beneath any node don't change, and there are no nodes beneath the new node
3. gets broken if the tree was originally empty

- let's fix the tree in the following way:
  - without ruining the property number 2 move the violation of property 1 upwards until it disappears
  - Finally fix property 3 by coloring the root black (this cannot break properties 1 and 2)
- violation of 1 = both the node and its parent are red
- moving is done by coloring nodes and making rotations

```
 RB-INSERT(T, x)
1   TREE-INSERT(T, x)
2   x→colour := RED
    (loop until the violation has disappeared or the root is reached)
3   while x ≠ T.root and x→p→colour = RED do
4       if x→p = x→p→p→left then
5           y := x→p→p→right
6           if y ≠ NIL and y→colour = RED then  (move the violation upwards)
7               x→p→colour := BLACK
8               y→colour := BLACK
9               x→p→p→colour := RED
10              x := x→p→p
11          else                                (moving isn't possible → fix the violation)
12              if x = x→p→right then
13                  x := x→p; LEFT-ROTATE(T, x)
14              x→p→colour := BLACK
15              x→p→p→colour := RED
16              RIGHT-ROTATE(T, x→p→p)
17      else
...     ▷ same as lines 5...16 except "left" and "right" have switched places
30  T.root→colour := BLACK                      (color the root black)
```



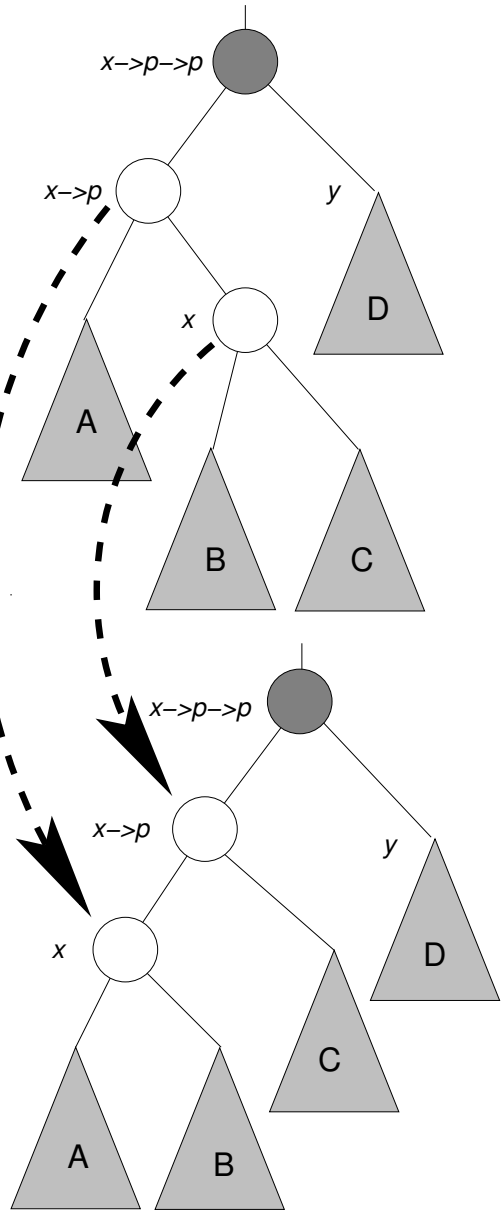
Moving the violation of property 1 upwards:

- the node x and its parent are both red
- also the node x's uncle/aunt is red and the grandparent is black.

$\Rightarrow$ the violation is moved upwards by coloring both the uncle of x and the parent black and the grandparent red.

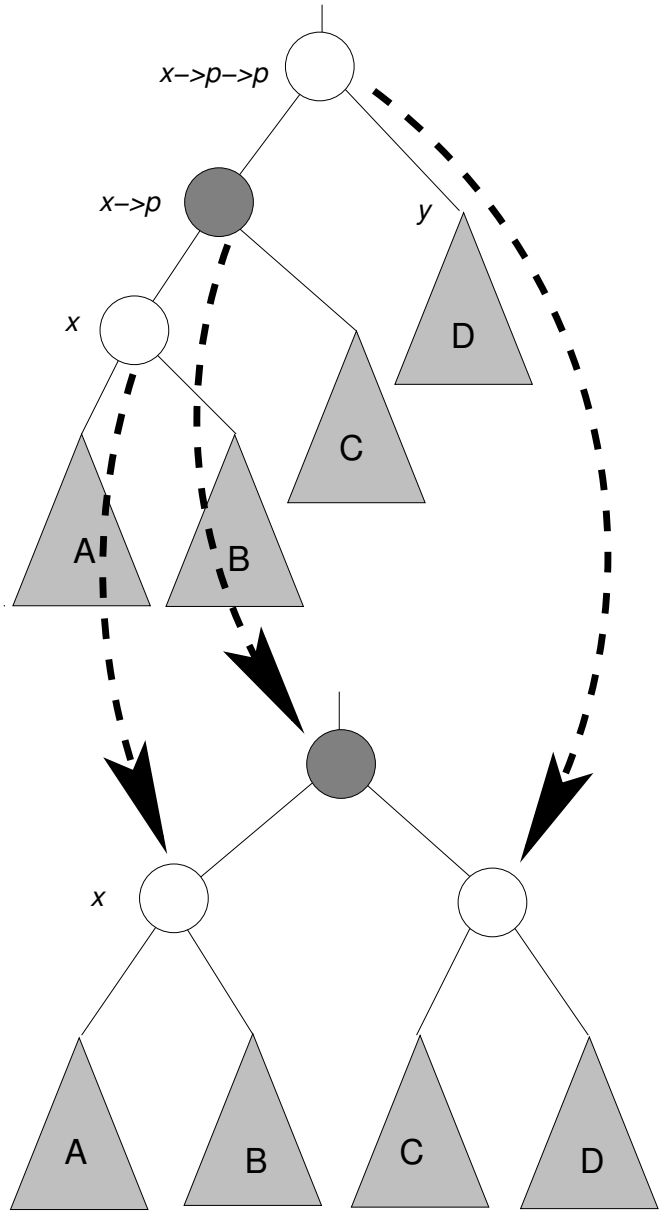
After the fixup:

- the property 1 may still be broken
  - the node x and its parent parent may both be red
- property 2 isn't broken
  - the number of black nodes on each path stays the same
- the property 3 may be violated
  - if we've reached the root, it may have been colored red



If there is no red uncle available, the violation cannot be moved upwards. A more complicated approach needs to be used instead:

- Make sure that x is the right child of its parent by making a rotation to the left



- then, colour the parent of x black and the grandparent red and make a rotation to the right
  - the grandparent is black for sure since otherwise there would have been two successive red nodes in the tree before the addition

After the fixup:

- there no longer are successive red nodes in the tree
- the fixup operations together don't break the property 2

  $\Rightarrow$ the tree is complete and execution the fixup algorithm can be stopped

General characteristics of the deletion algorithm

- first the node is deleted like from an ordinary binary search tree
  - w points to the node deleted
- if w was red or the tree was made entirely empty, the red-black properties are maintained

  $\Rightarrow$ nothing else needs to be done
- otherwise the tree is fixed with RB-DELETE-FIXUP starting from the possible child of w and its parent $w{\rightarrow}p$
  - TREE-DELETE ensures that w had atmost one child

```
RB-DELETE(T, z)
1   w := TREE-DELETE(T, z)
2   if w→colour = BLACK and T.root ≠ NIL then
3       if w→left ≠ NIL then
4           x := w→left
5       else
6           x := w→right
7       RB-DELETE-FIXUP(T, x, w→p)
8   return w
```

```
RB-DELETE-FIXUP(T, x, y)
1   while x ≠ T.root and (x = NIL or x→colour = BLACK) do
2       if x = y→left then
3           w := y→right
4           if w→colour = RED then
5               w→colour := BLACK; y→colour := RED
6               LEFT-ROTATE(T, y); w := y→right
7           if (w→left = NIL or w→left→colour = BLACK) and
            (w→right = NIL or w→right→colour = BLACK)
            then
8               w→colour := RED; x := y
9           else
10              if w→right = NIL or w→right→colour = BLACK then
11                  w→left→colour := BLACK
12                  w→colour := RED
13                  RIGHT-ROTATE(T, w); w := y→right
14              w→colour := y→colour; y→colour := BLACK
15              w→right→colour := BLACK; LEFT-ROTATE(T, y)
16              x := T.root
17      else
...         ▷ same as lines 3...16 except "left" and "right" have switched places
32      y := y→p
33  x→colour := BLACK
```

11.8 B-trees

B-trees are rapidly branching search trees that are designed for storing large dynamic sets on a disk

- the goal is to keep the number of search/write operations as small as possible
- all leaves have the same depth
- one node fills one disk unit as closely as possible

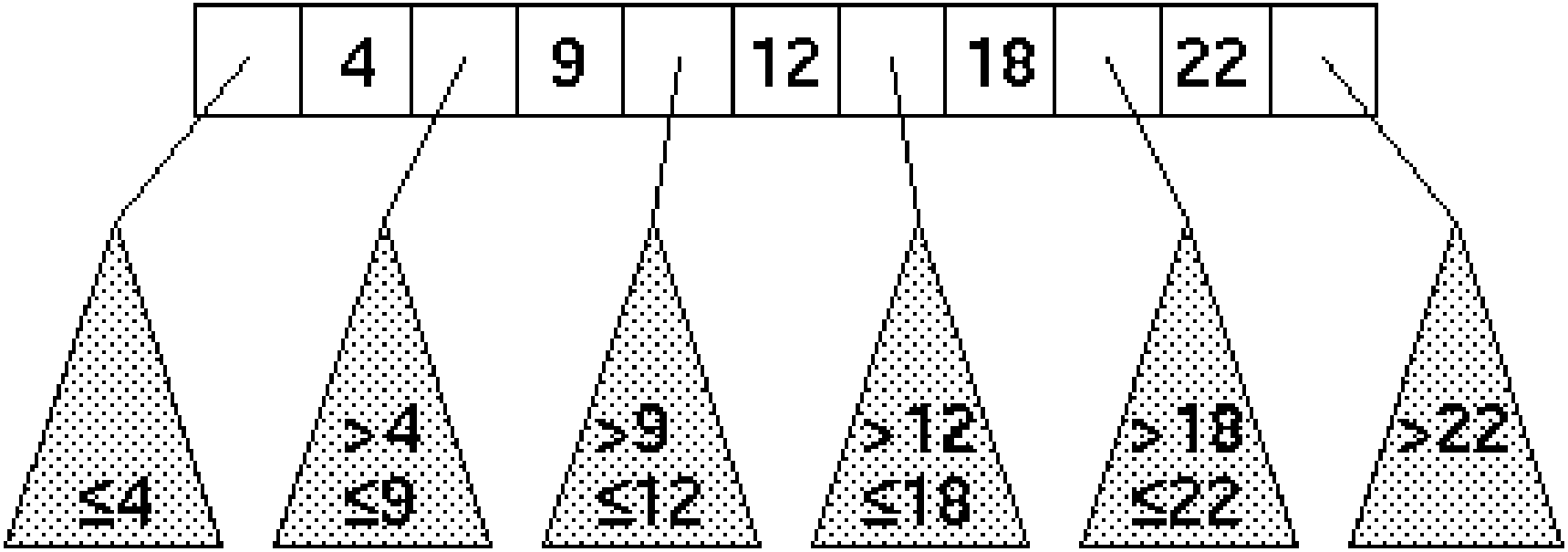
  $\Rightarrow$ B-tree often branches rapidly: each node has tens, hundreds or thousands of children

  $\Rightarrow$ B-trees are very shallow in practise
- the tree is kept balanced by alternating the amount of the node's children between $t, \ldots, 2t$ for some $t \in N, t \geq 2$
  - each internal node except the root always has at least $\frac{1}{2}$ children from the maximum amount

The picture shows how the keys of a B-tree divide the search area.

Searching in a B-tree is done in the same way as in an ordinary binary search tree.

- travel from the root towards the leaves
- in each node, choose the branch where the searched element must be in - there are just much more brances

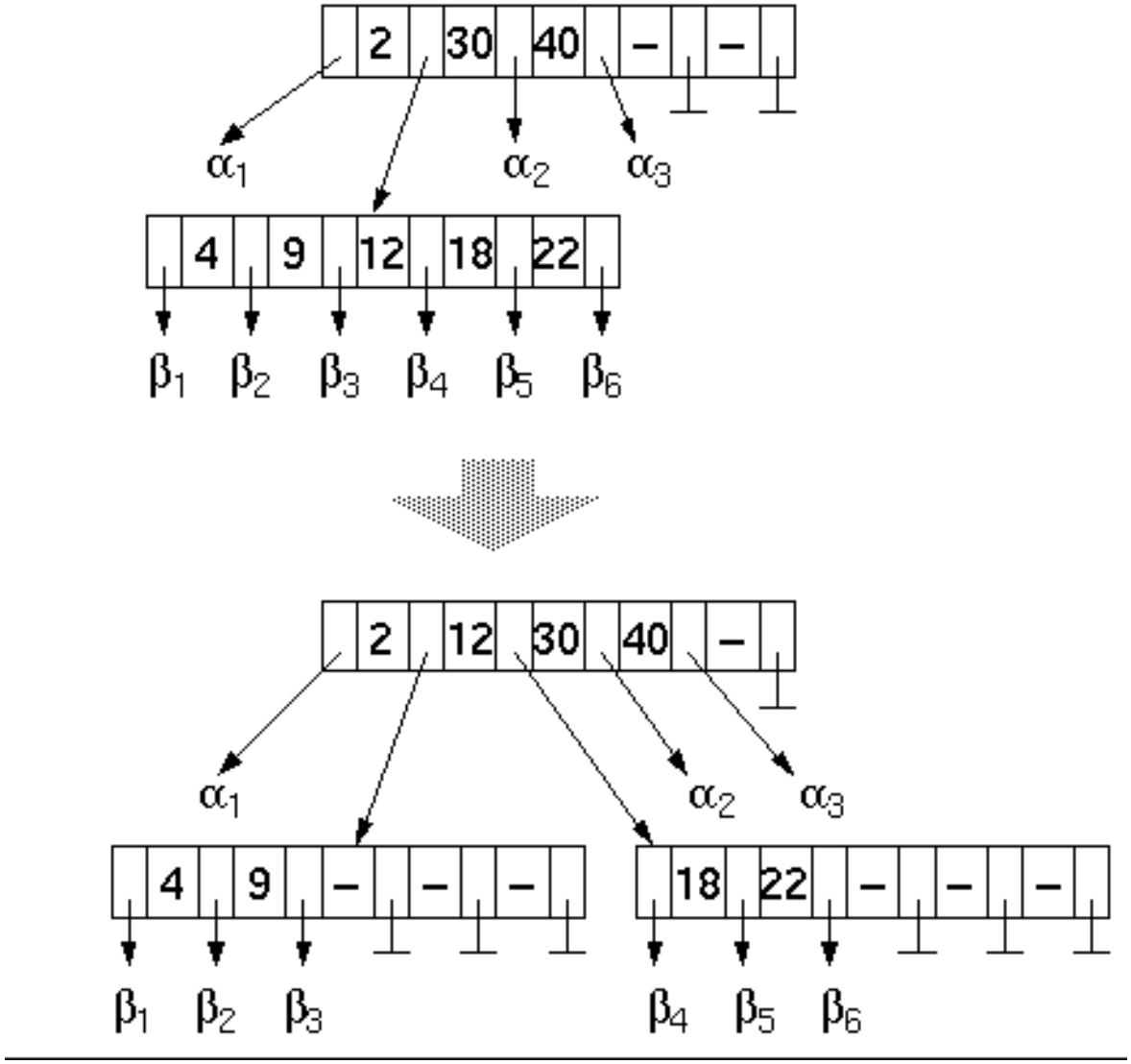


Inserting an element into a B-tree

- travel from the root to a leaf and on the way split each full node into half
  $\Rightarrow$ when a node is reached, its parent is not full
- the new key is added into a leaf
- if the root is split, a new root is created and the halves of the old root are made the children of the new root
  $\Rightarrow$ B-tree gets more height only by splitting roots
- a single pass down the tree is needed and no passes upwards

A node in a B-tree is split by making room for one more key in the parent and the median key in the node is then lifted into the parent.

The rest of the keys are split around the median key into a node with the smaller keys and a node with the larger keys.


2
30
40
–
–
α_1
α_2
α_3
4
9
12
18
22
β_1 β_2 β_3 β_4 β_5 β_6
2
12
30
40
–
α_1
α_2
α_3
4
9
–
–
–
18
22
–
–
–
β_1 β_2 β_3 β_4 β_5 β_6

Deleting a key from a B-tree is a similar operation to the addition.

- travel from the root to a leaf and always before entering a node make sure there is at least the mimimun amount + 1 keys in it
  - this quarantees that the amount of the keys is kept legal although one is removed
- once the searched key is found, it is deleted and if necessary the node in combined with either of its siblings
  - this can be done for sure, since the parent node has at least one extra key
- if the root of the end result has only one child, the root is removed and the child is turned into the new root

11.9 AVL-trees and Splay-trees

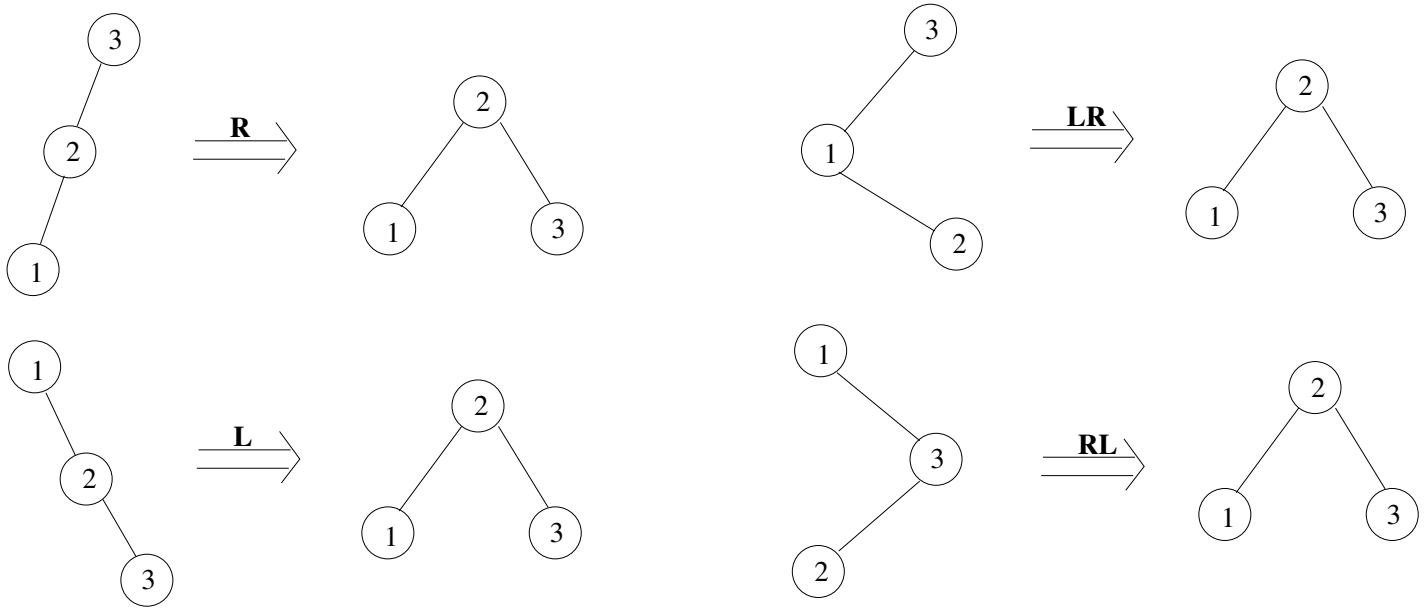
An *AVL tree* (after Adelson-Velsky, Landis) is a binary search tree where every node has a **balance factor** of either 0, +1, or -1.

- the factor is defined as the difference between the heights of the node's left and right subtree

In insertion of a new node makes and AVL tree unbalanced, rotations are used to transform the tree back to balance

There are four rotations:

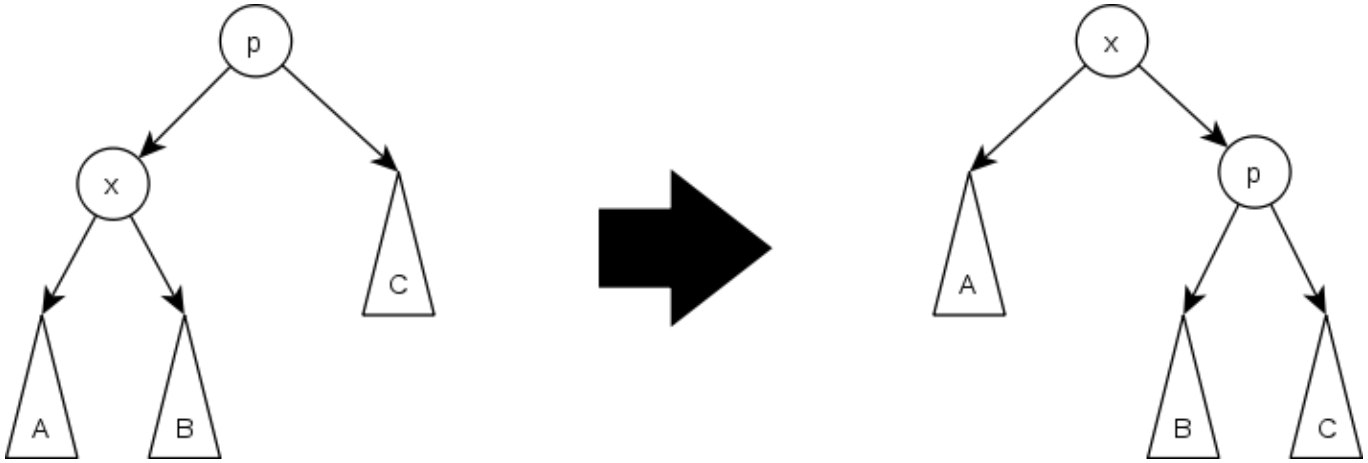
- Single right rotation
- Single left rotation
- Double left-right -rotation
- Double right-left -rotation



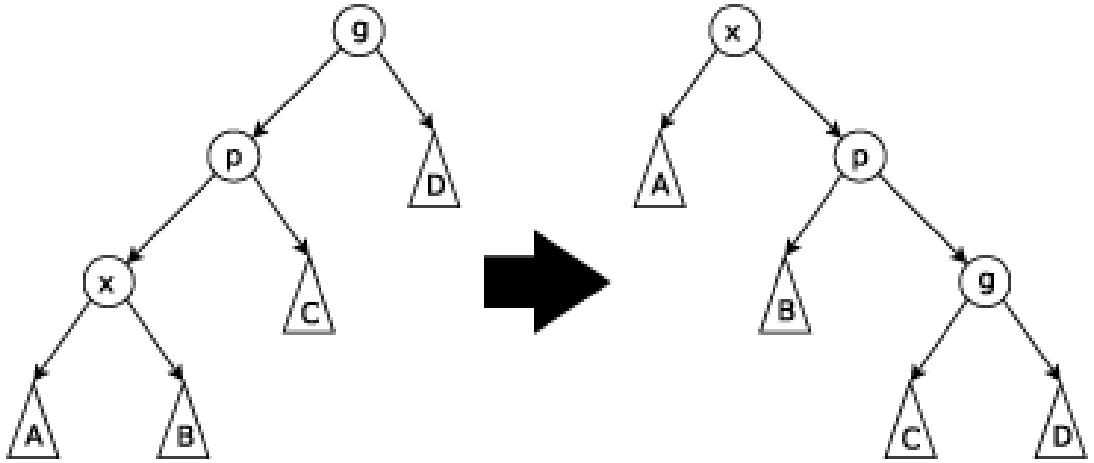
A *splay tree* is a binary search tree with the additional property that recently accessed elements are quick to access again.

A splay operation is performed on a node when accessed to move it to the root: a sequence of splay steps, each moving the node closer to the root.

- Zig Step:



- Zig-Zig Step:



- Zig-Zag Step: