Tree structures

AVL-tree and balancing

Agenda

- Balanced trees
- **AVL-tree**
  - Definition
  - Properties
  - Rotations
    - Left / Right
    - Single / Double
    - **SLR SRR DLR DRR**

- The meaning behind **BSTs** is that **search** should be as fast as possible
- The quickest searching is when the height is **log(N)** where N = the number of nodes in the tree

NB: check definitions of height and depth

- Some textbooks give the **height** of a tree as
  - **The maximum number of nodes on any path from the root to a leaf**
- Others define **height** as
  - The maximum number of **edges** on any path from the root to a leaf
- Some use both **height** and **depth** – **CHECK!!!**

Balancing: NB: check definitions of height and depth

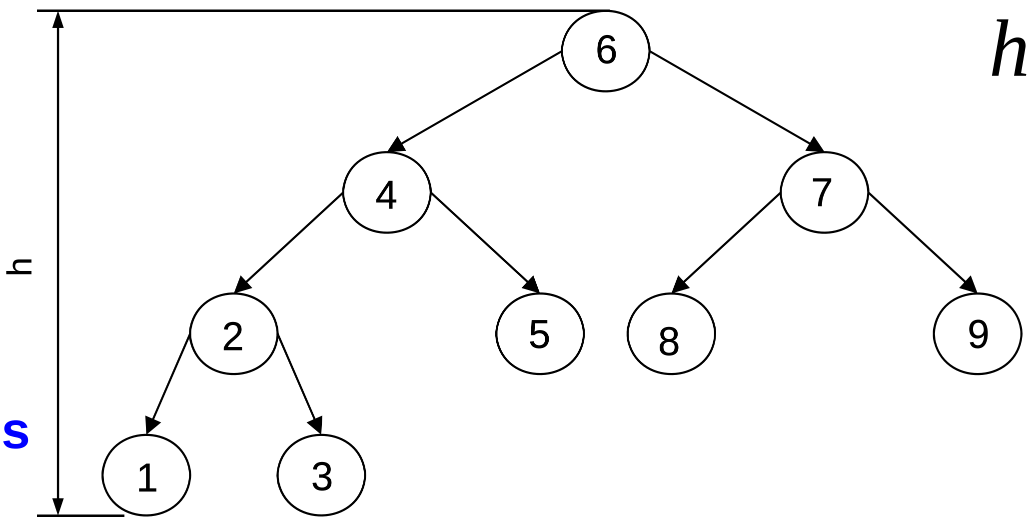
- By imposing a balance invariant on the tree, logarithmic **height** may be attained
- The "ultimate" balance invariant is achieved in the form of a **complete tree**
  - A complete tree is never higher than log(N) where N = the number of nodes
  - The great disadvantage of this is that is is very difficult to maintain the completeness invariant

4 nodes = 2.58

7 nodes = 3

9 nodes = 3.16

15 nodes = 4



$$h \geq \lceil \log(N+1) \rceil$$

AVL-tree

- **Adelson-Velskii and Landis** discovered a method of balancing a BST
- An AVL-tree is a BST where
  - For each node n,

|Height(Left(n)) - Height(Right(n))| < 2

must be satisfied

  - The height of one sub-tree may be no more than 1 unit compared with the other sub-tree

AVL

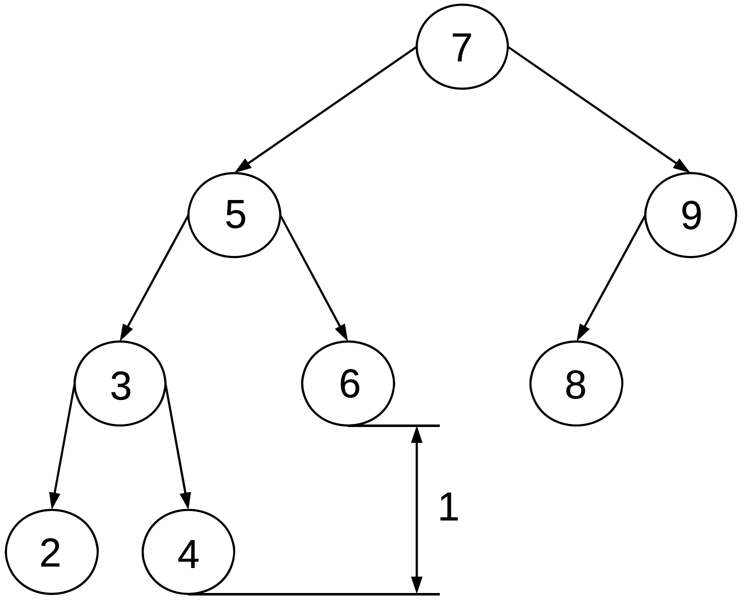
=

Binary Search Tree

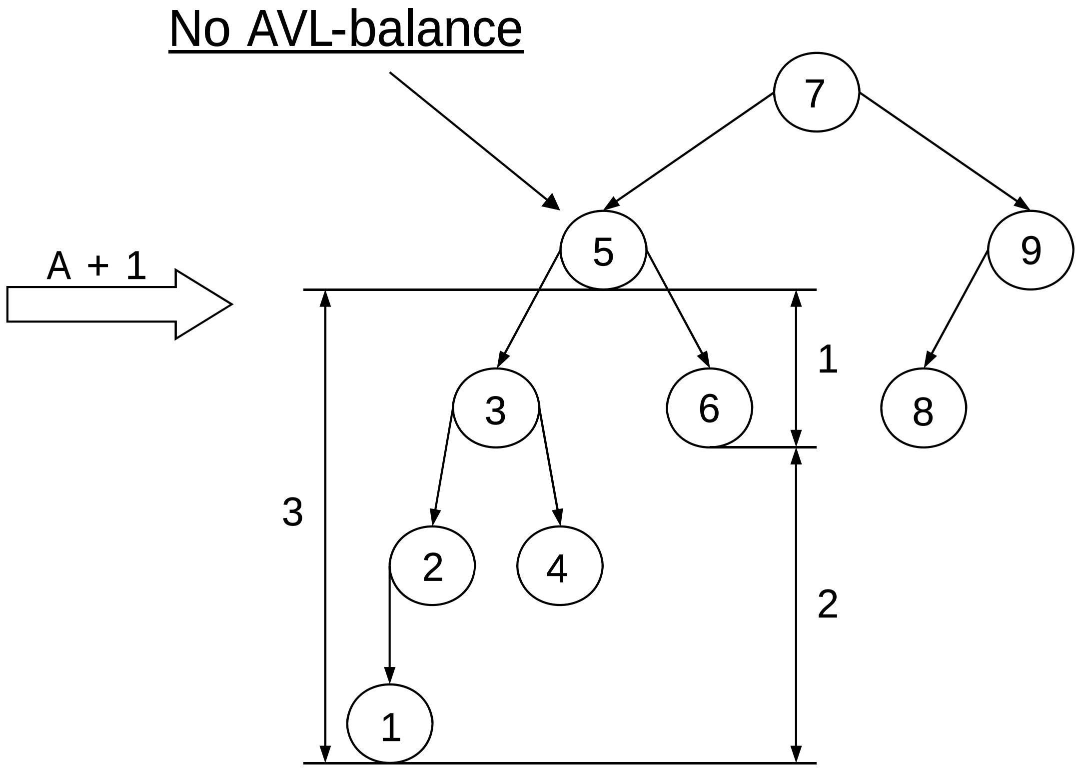
+

$$|\,Height\ (Left\ (n)) - Height\ (Right\ (n))\,| < 2$$

AVL-tree (contd.)



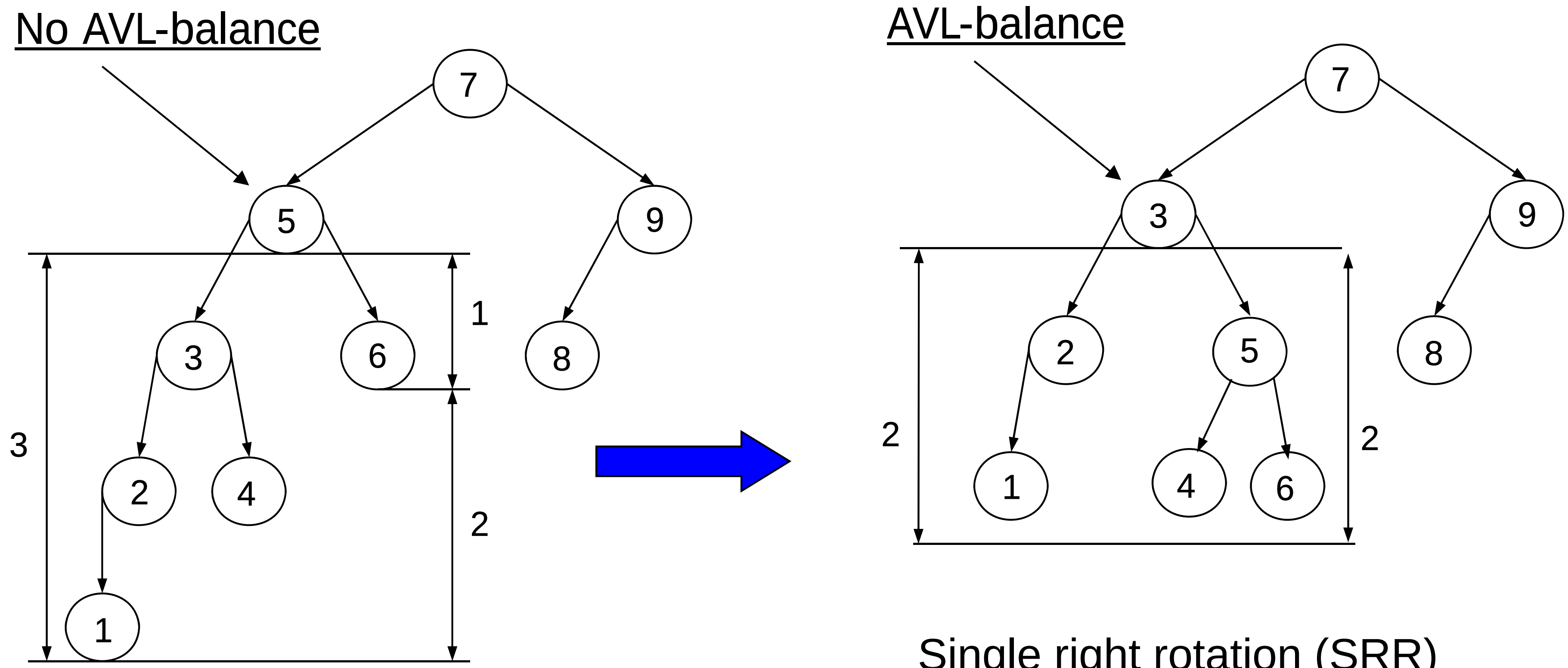
AVL-tree (sub-tree 5)
and tree at 7



BST-tree B (NOT AVL!)

$$|\,Height\ (Left\ ("5")) - Height\ (Right\ ("5"))\,| = 2$$

AVL-tree (contd.)



Single right rotation (SRR)

BST-tree B (NOT AVL!)

$$|\,Height\ (Left\ ("5")) - Height\ (Right\ ("5"))\,| = 2$$

AVL-tree (contd.)

- The invariant for an AVL tree requires balancing mechanisms
- These mechanisms are called **rotations**
- These mechanisms may be applied in 2 ways
  - Executed as part of operations such as **Add** and **Remove**
  - Executed as a **separate operation (Balance)** after operations such as Add and Remove
  - This latter method is preferred (& is simpler)

Rotation (correcting imbalance)

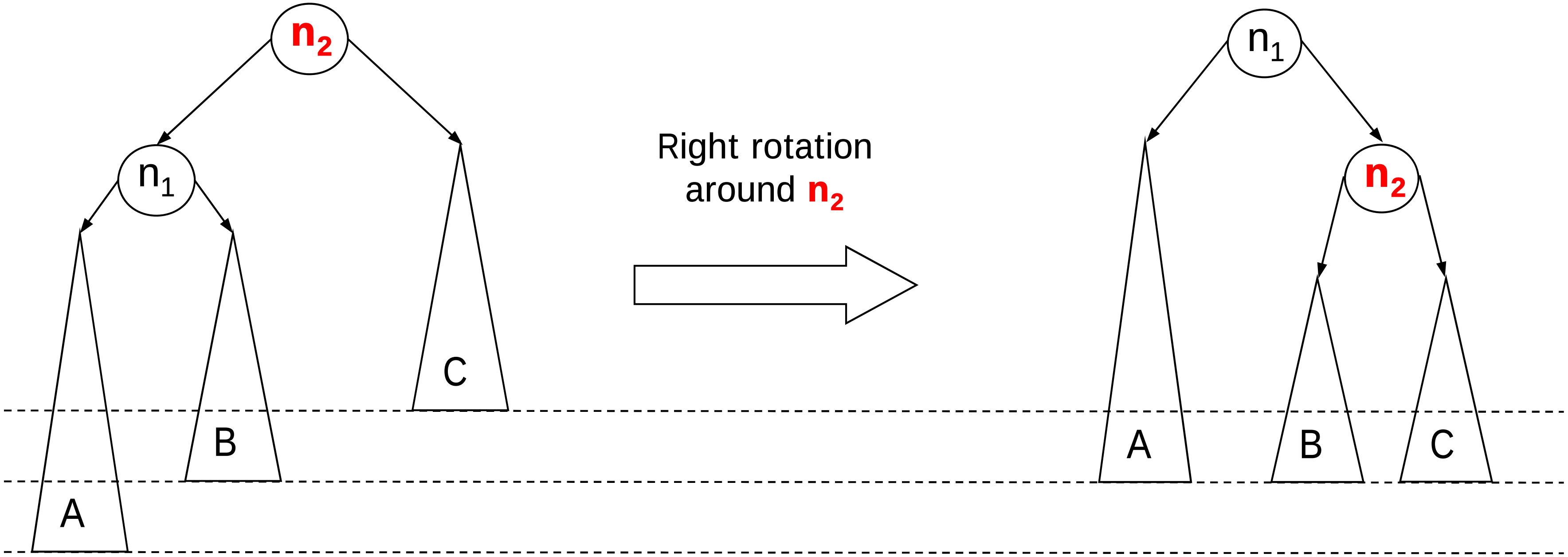
- **Moves the "centre of gravity" from one side of the (sub-)tree to another**
- In order to correct imbalances, there are **4 cases** to consider. If an imbalance occurs at X, the following may have taken place :
  1. An insertion in the **left sub-tree** of the **left child** of X requires a **simple right rotation** **(SRR – single right rotation)**
  2. An insertion in the **right sub-tree** of the **left child** of X requires a **left-right rotation**. (**DRR - double right rotation)**
  3. An insertion in the **left sub-tree** of the **right child** of X requires a **right-left rotation**. (**DLR - double left rotation)**
  4. An insertion in the **right sub-tree** of the **right child** of X requires a **simple left rotation** **(SLR – single left rotation)**

  After a rotation the BST-invariant still applies

  - ○ **Value(Left(n)) < Value(n) < Value(Right(n))**
  - ○ **DRR = SLR(LC(T)) + SRR(T);** **DLR = SRR(RC(T)) + SLR(T)**
  - ○ **SLR/DLR is mirror image of an SRR/DRR respectively**

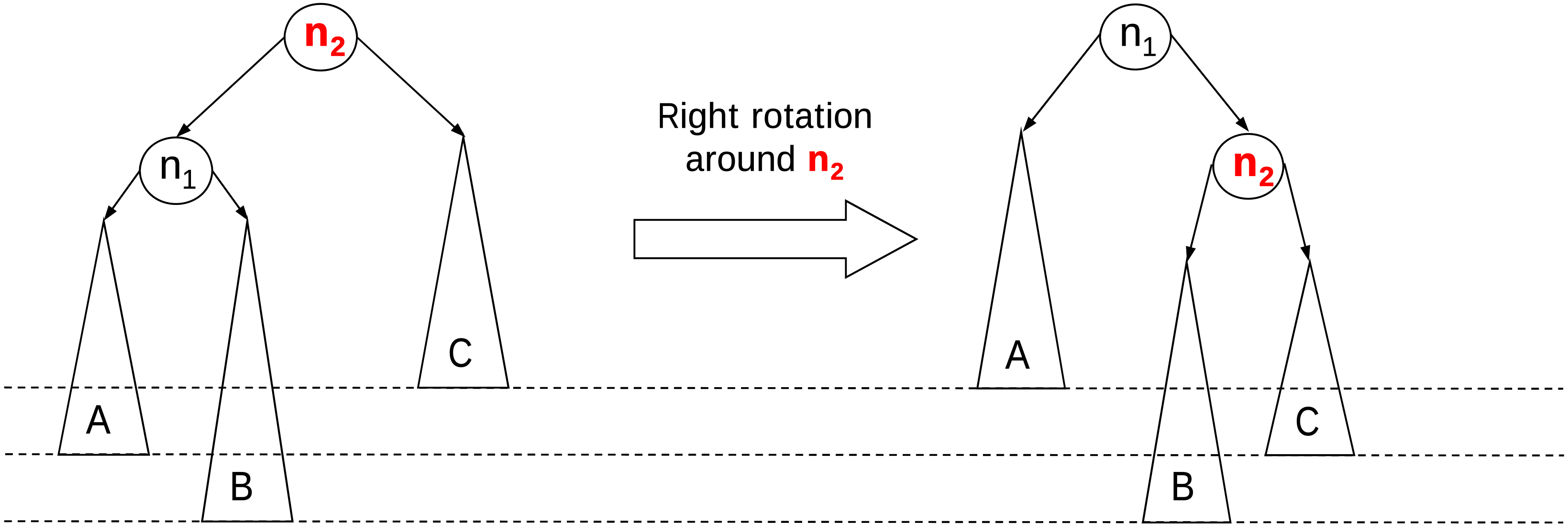
Rotation (contd.) (add “outside”)

- When the AVL-invariant is violated at n_2 and ”the centre of gravity” is displaced to the left, a right rotation around n_2 is required
- **Add to left child of left child / right child of right child (outside)**



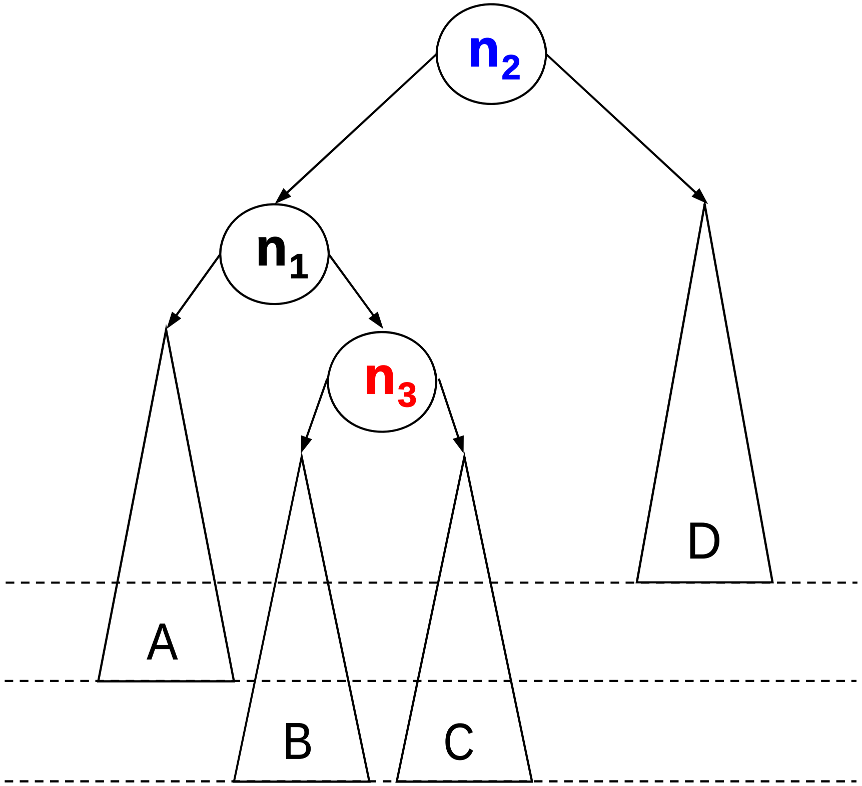
Rotation (contd.) (add “inside”)

- If the tree is not wholly displaced to the left, then a simple rotation will not work
- **Add to right child of left child / left child of right child (inside)**

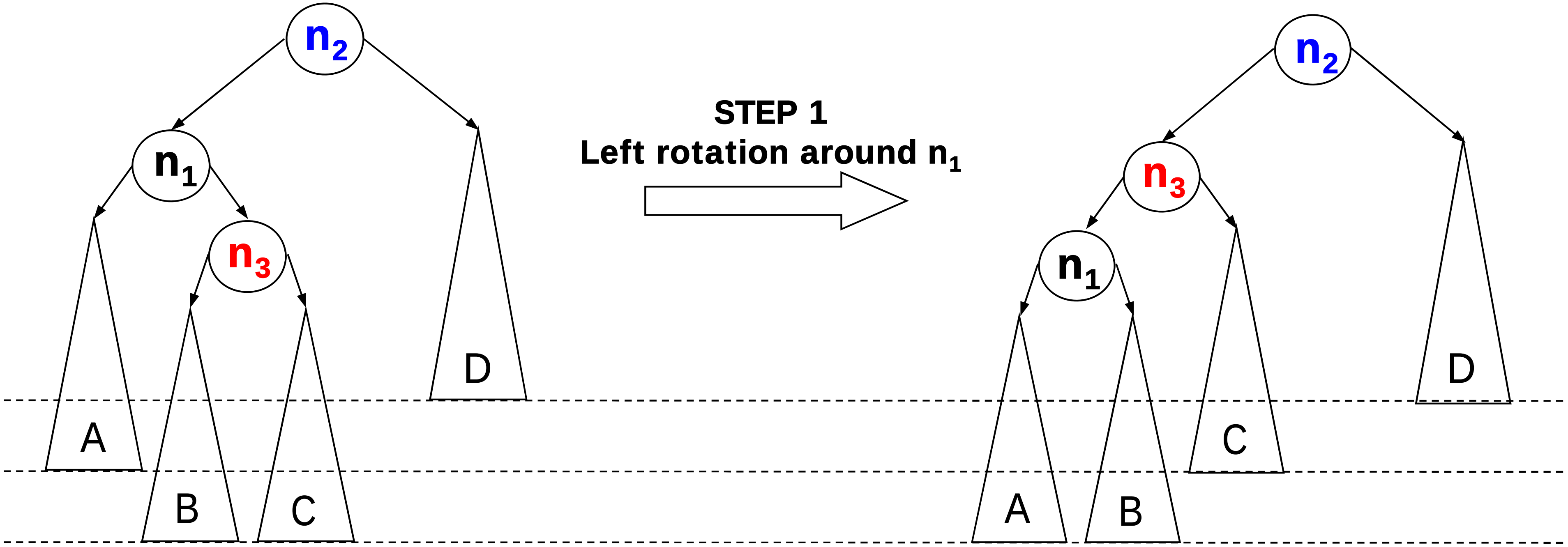


Rotation (contd.)

- In order to correct this problem, a third node must be taken into consideration - **n_3**
- Two rotations are required
  - **FIRST** a **left-rotation** around **n_1** in order to better place the “centre of gravity””
  - **THEN** a **right rotation** in the ”normal” order around the node (**n_2**) where the imbalance has occurred
  - NB the first rotation is always a single rotation.

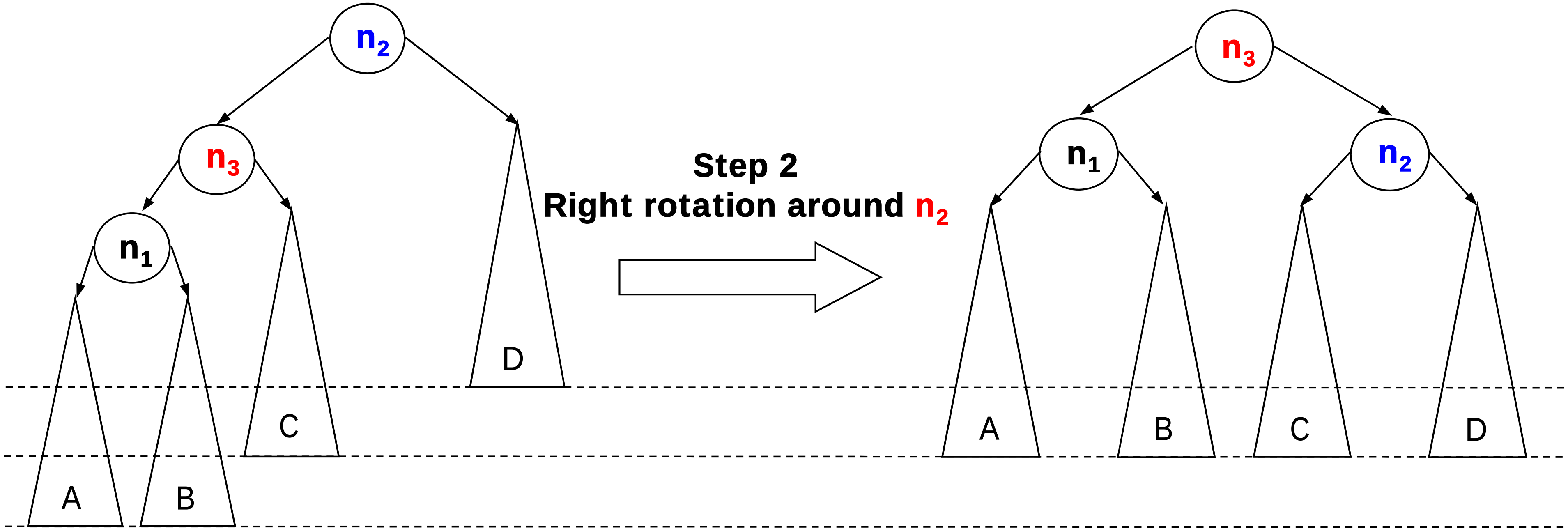


Rotation (contd.)



STEP 1
Left rotation around n_1

Rotation (contd.)



Step 2

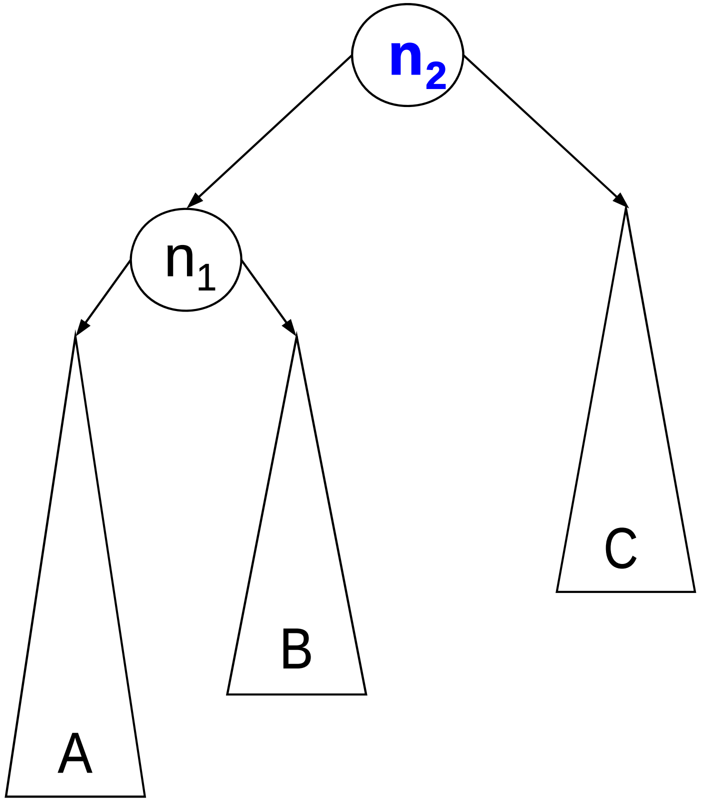
Right rotation around n_2

Rotation - Algorithms

- A **simple right rotation** can be implemented as follows

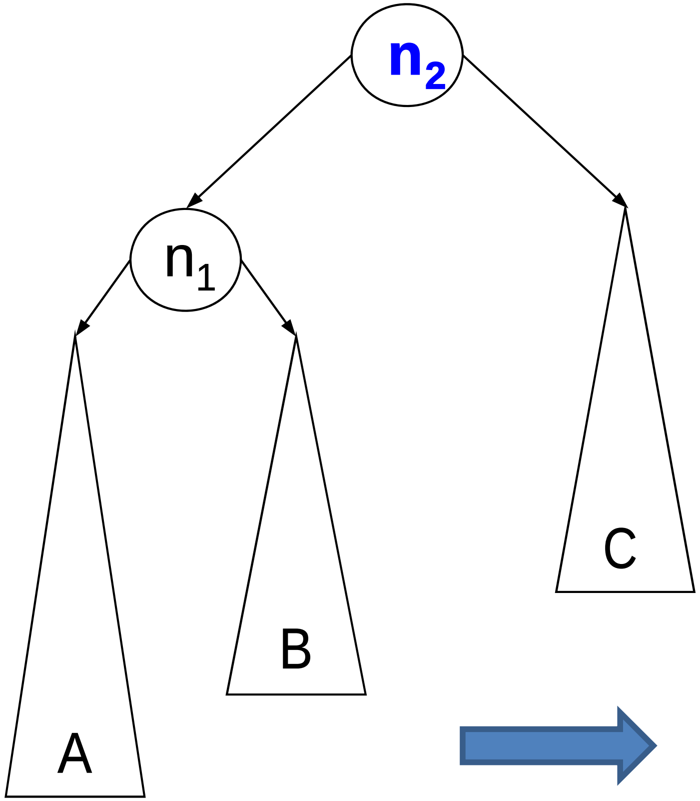
```
RotateRight(n2)
    n1          = n2.left
    n2.left     = n1.right
    n1.right    = n2
    return n1
    end RotateRight
```

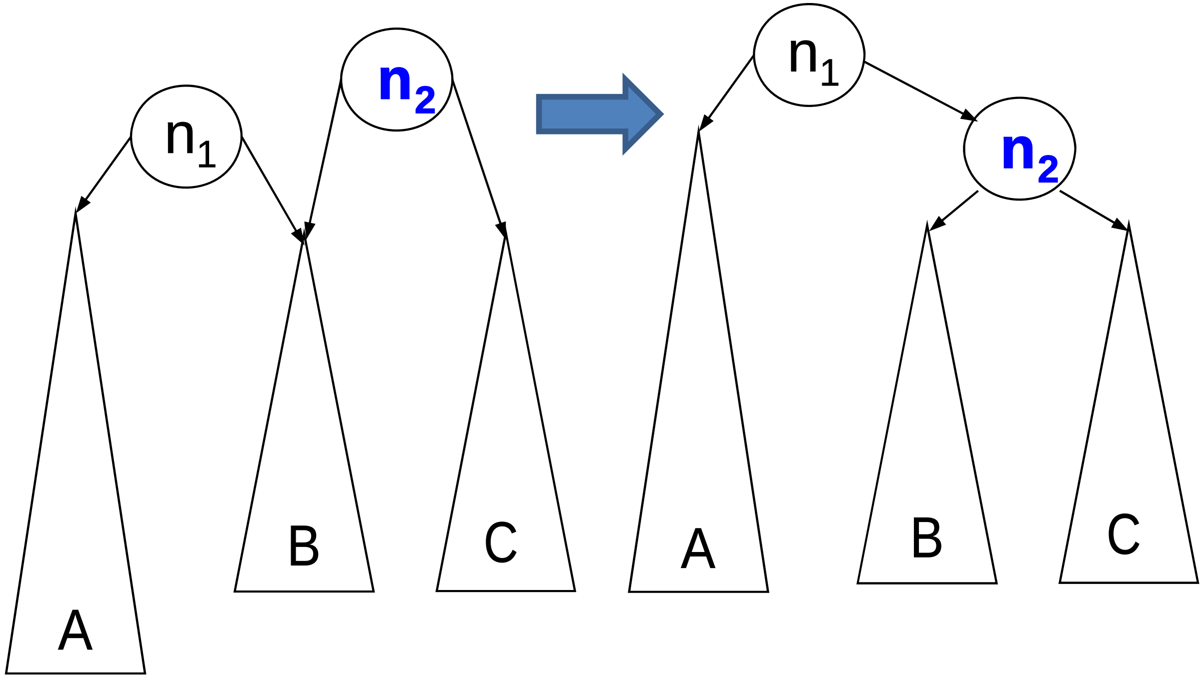
- **Similarly for a simple left rotation (mirror image)**



Rotation – Algorithms - SRR

```
n1        = n2.left
n2.left   = n1.right
n1.right = n2
return n1
```



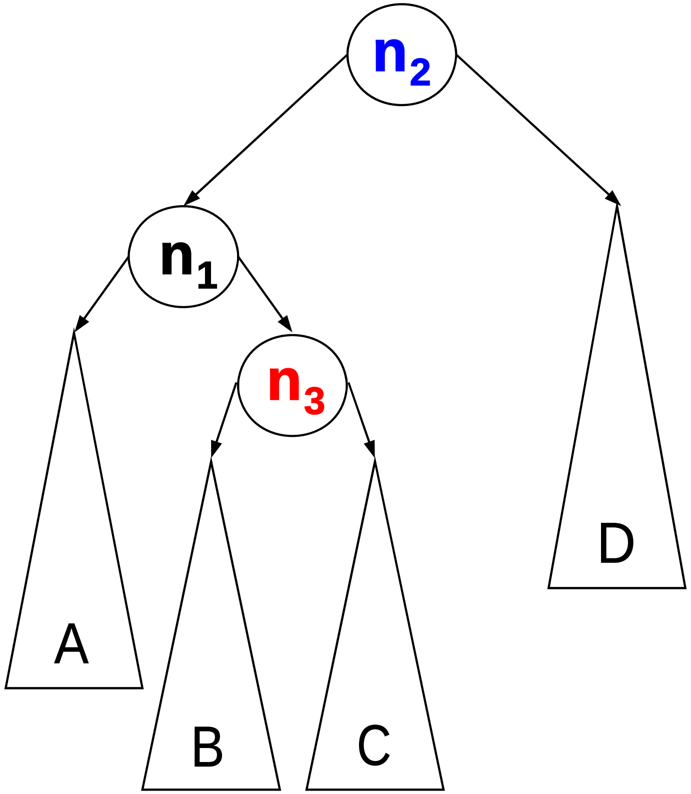


Rotation – Algorithms (contd.)

- A <u>double left-right rotation</u> DLR can be implemented as follows

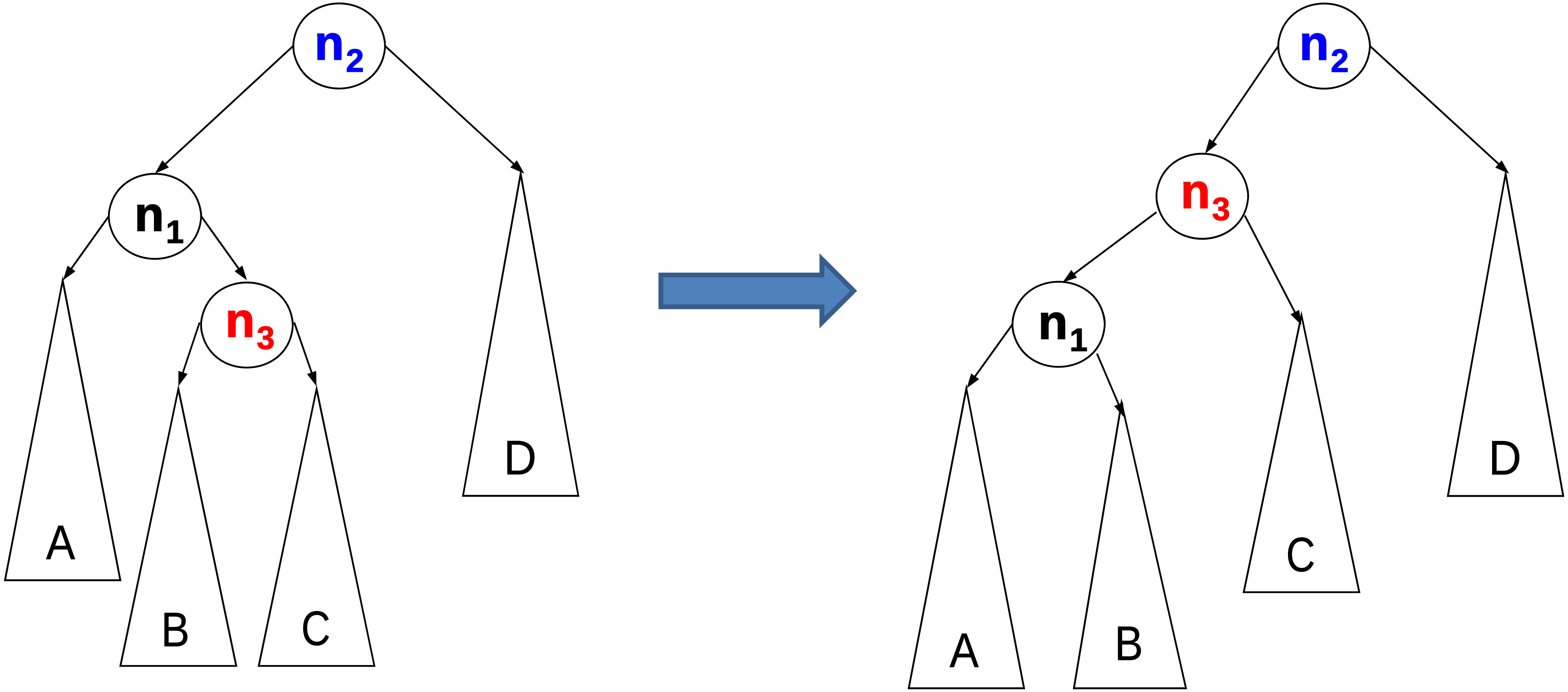
```
RotateDoubleLeftRight(n2)
    n2.left = RotateLeft(n2.left)
    return RotateRight(n2)
end RotateDoubleLeftRight
```

- **Similarly for a <u>double right left rotation</u> (mirror image)**



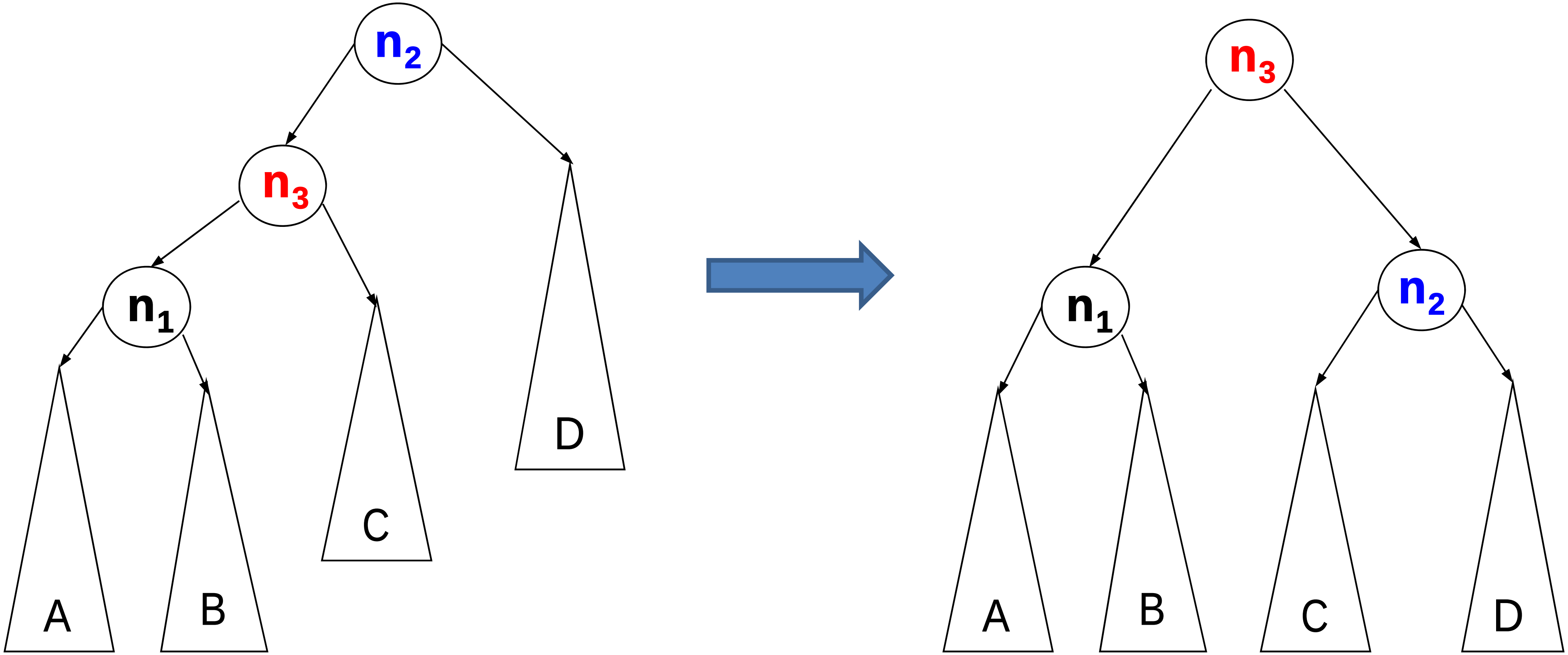
Rotation – Algorithms (contd.)

```
n2.left = RotateLeft(n2.left)
return RotateRight(n2)
```



Rotation – Algorithms (contd.)

```
n2.left = RotateLeft(n2.left)
return RotateRight(n2)
```



The “code”

- **SLR** (+ outside right)

```
RotateLeft(n2)
    n1          = n2.right
    n2.right    = n1.left
    n1.left     = n2
    return n1
    end RotateLeft
```

- **SRR** (+ outside left)

```
RotateRight(n2)
    n1          = n2.left
    n2.left     = n1.right
    n1.right    = n2
    return n1
    end RotateRight
```

- **DLR** (+ inside right)

```
RotateDoubleRightLeft(n2)
    n2.right = RotateRight(n2.right)
    return RotateLeft(n2)
end RotateDoubleRightLeft
```

- **DRR** (+ inside left)

```
RotateDoubleLeftRight(n2)
    n2.left = RotateLeft(n2.left)
    return RotateRight(n2)
end RotateDoubleLeftRight
```

SLR (+ outside right)

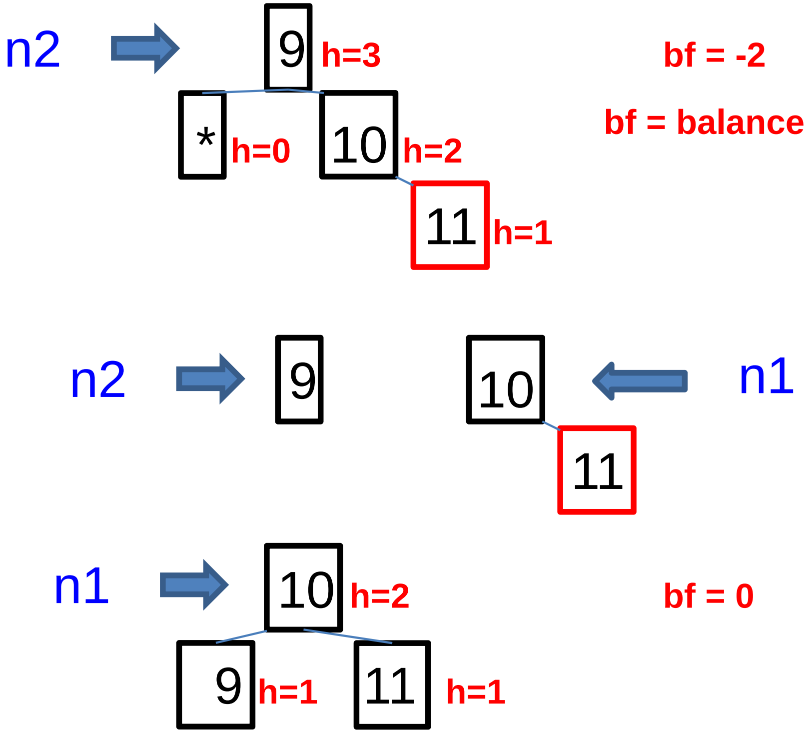
- **SLR (+ outside right)**

```
RotateLeft(n2)
    n1          = n2.right
    n2.right    = n1.left
    n1.left     = n2
    return n1
    end RotateLeft
```

n1 = (¤ 10 11)
n2.right = ¤
n1.left = 9

return n1 = (9, 10, 11)

- **Example: bf = H(LC) - H(RC)**



bf = -2

bf = balance factor

bf = 0

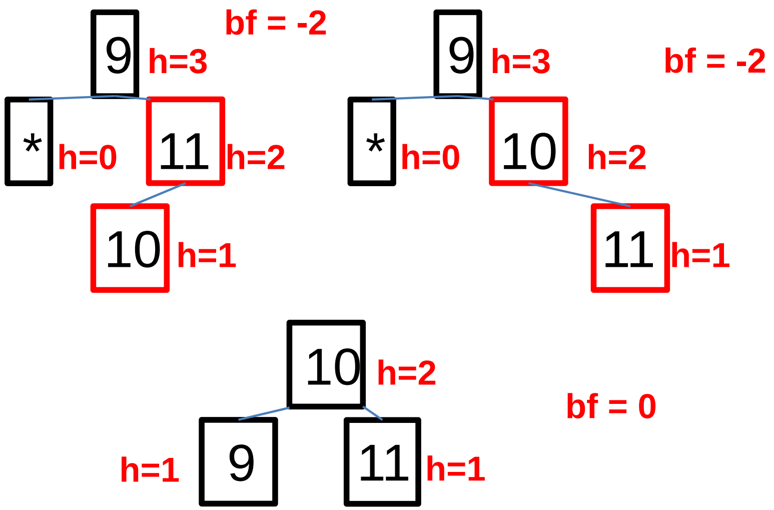
DLR (+ inside right)

- **DLR (+ inside right)**

```
RotateDoubleRightLeft(n2)                DLR
      n2.right = RotateRight(n2.right)
      return RotateLeft(n2)
end RotateDoubleRightLeft
```

```
RotateRight(n2)                          RR
      n1            = n2.left
      n2.left       = n1.right
      n1.right      = n2
      return n1
      end RotateRight
```

- **Example: bf = H(LC) - H(RC)**



n1 = 10

n2.left = ¤

n1.right = 11

return n1 = (¤, 10, 11)

(NB: in RR n2 = 11)