Algorithms and Data Structures

Lecture 7

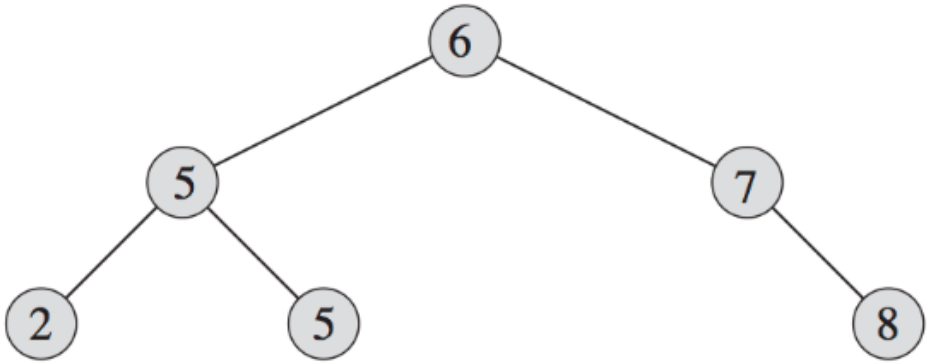
Binary Search Trees II

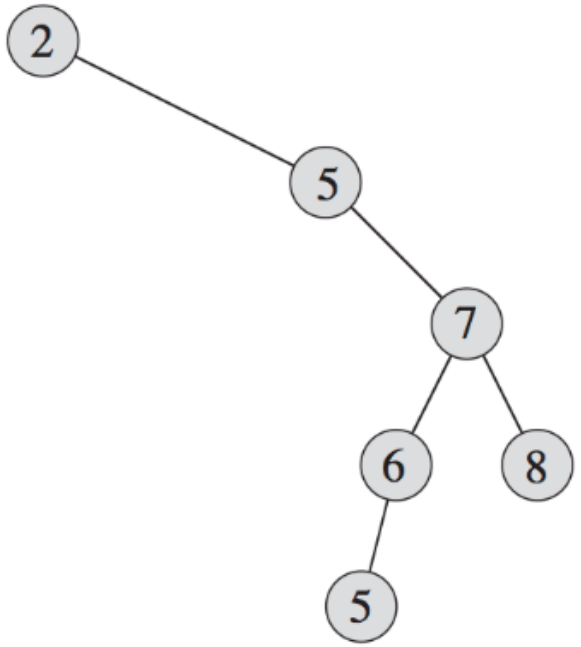
UNI FREIBURG

Fabian Kuhn

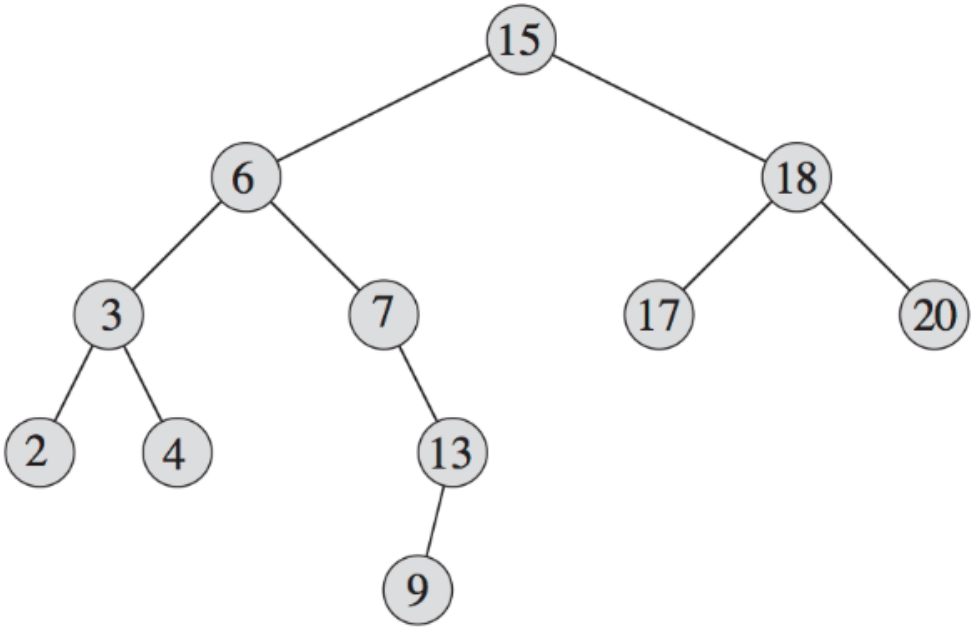
Algorithms and Complexity

Binary Search Trees





Source: [CLRS]



Depth of a Binary Search Tree

Worst-case running time of the operations
find, min, max, predecessor, successor, insert, delete:

$$O(\text{depth of tree})$$

- In the **best case**, the depth is $\log_2 n$
  - Definition depth: Length of longest path from the root to a leaf
- In the **worst case**, the depth is $n - 1$

- In the **average case**, the depth is $O(\log n)$
  - Average here mean a random insertion order

Is it possible to guarantee that the depth of a binary search tree is always $O(\log n)$?

“Typical” Case

UNI FREIBURG

Random binary search tree:

- n **keys are inserted in a random order**

Observation:

- With probability $1/3$, both subtrees of the root have $n/3$ nodes.

1. Drittel der Schlüssel	2. Drittel der Schlüssel	3. Drittel der Schlüssel

- The key of the root is the first inserted key
- With probability $1/3$ the key comes from the middle part.

“Typical” Case

UNI FREIBURG

Random binary search tree:

- **n keys are inserted in a random order**

Observation:

- With probability $1/3$, both subtrees of the root have $n/3$ nodes.
- Analogously, this holds for all subtrees
- Thus, on average in every 3rd from the root towards a leaf, the subtree gets smaller by a factor $2/3$.
- Decreasing by a factor $2/3$ can only happen $O(\log n)$ times.
- The depth of a random binary search tree is therefore $O(\log n)$
- Exact calculation gives:

  Expected depth of a random binary search tree: $\mathbf{4.311 \cdot \ln n}$

Enforce “Typical” Case?

UNI FREIBURG

“Typical” Case:

- If the keys are inserted in a random order, the resulting binary search tree has depth $O(\log n)$.
- All operations have running time $O(\log n)$.

Problem:

- A random insertion order is not necessarily the typical case!
- Presorted value can be equally typical.
  - This results in a very bad binary search tree.

Idea:

- Can we enforce a random insertion order?
- Keys are inserted in an arbitrary order, but the structure should always be as if they were inserted in random order!

Treap: «Enforce» Random Order

UNI FREIBURG

- Keys are inserted in an arbitrary order, but the structure should always be as if they were inserted in random order.

- For each key, when inserting they key, one determines a random value. The structure of the tree is then always reorganized such that it is the same as if the keys were inserted according to the sorted order of those random values.

- The necessary structure changes of an insert or delete operation can be done in time $O(\text{depth})$.

- With high probability, all operations then have running time $O(\log n)$.

Red-Black Trees

UNI FREIBURG

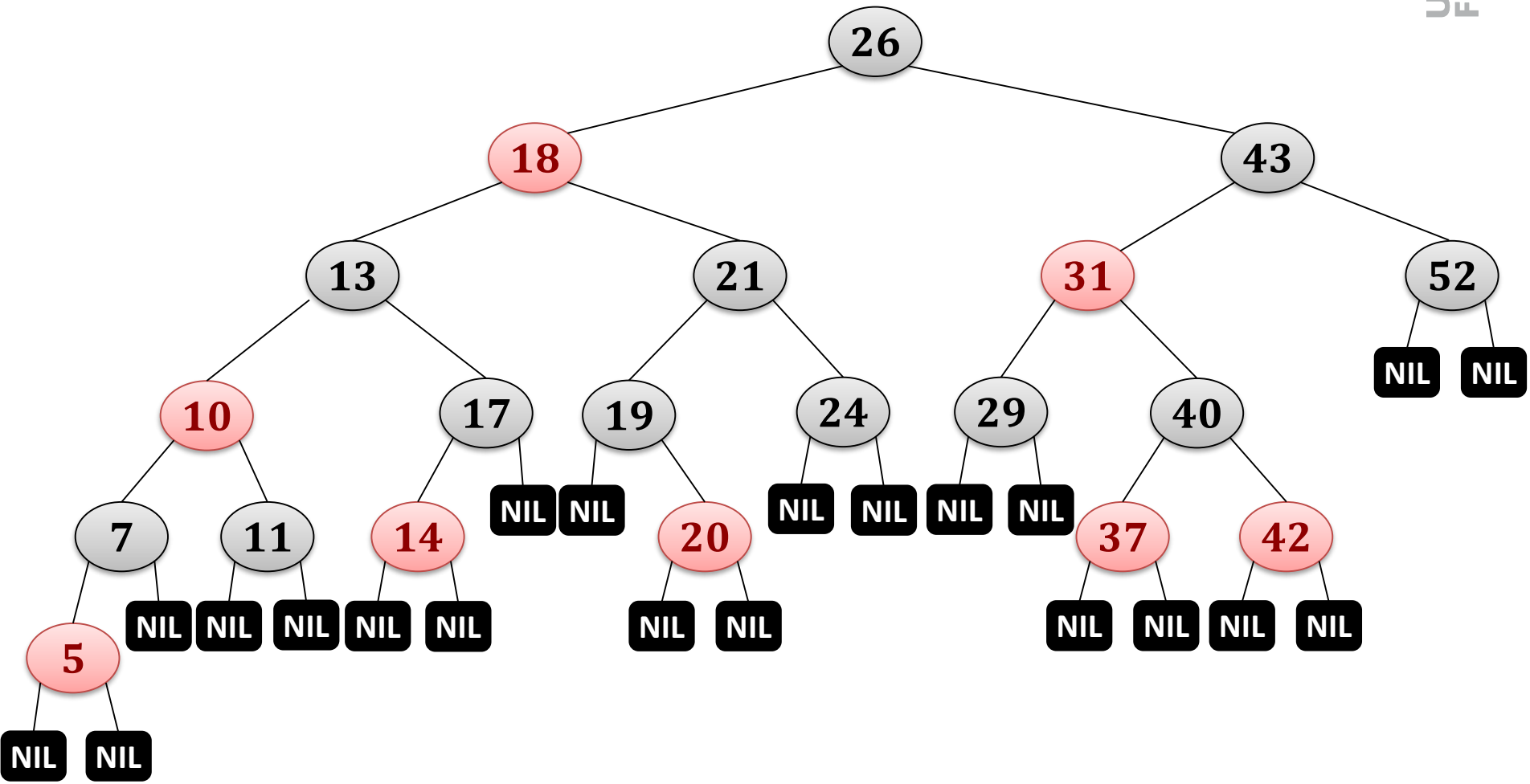
Goal: Binary search trees that are **always balanced**

- balanced, intuitively: each subtree, left & right $\approx$ equally large
- balanced, formally: subtree with k nodes has depth $O(\log k)$

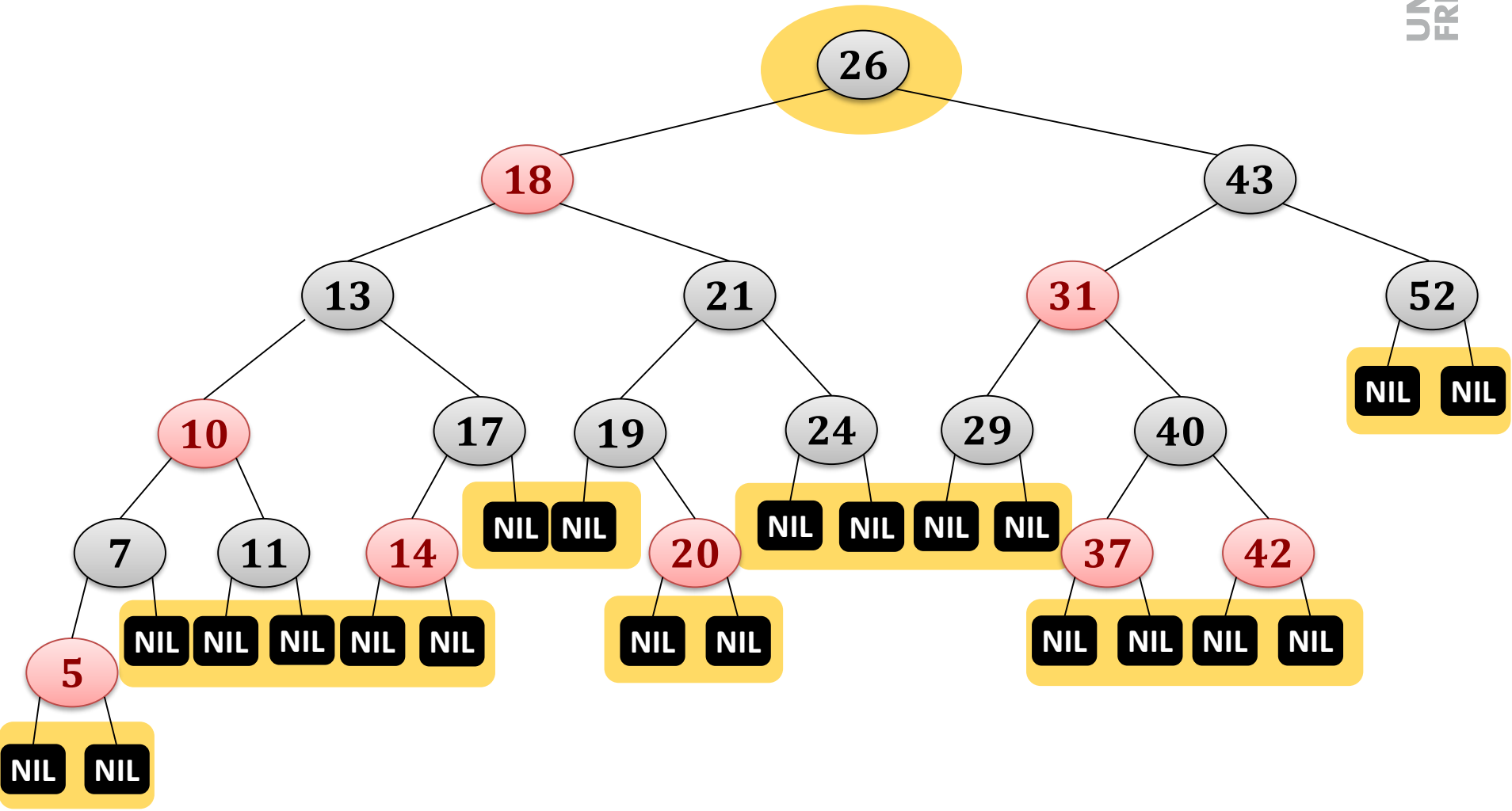
Red-black trees are binary search trees with the following properties:

1) All nodes are **red** or **black**.
2) The root is black.
3) The leaves (= NIL-nodes) are black.
4) Red nodes have two black children.
5) For each node v all (direct) paths from v to leaves (NIL) in the subtree of v have the same number of black nodes.

Red-Black Trees : Example

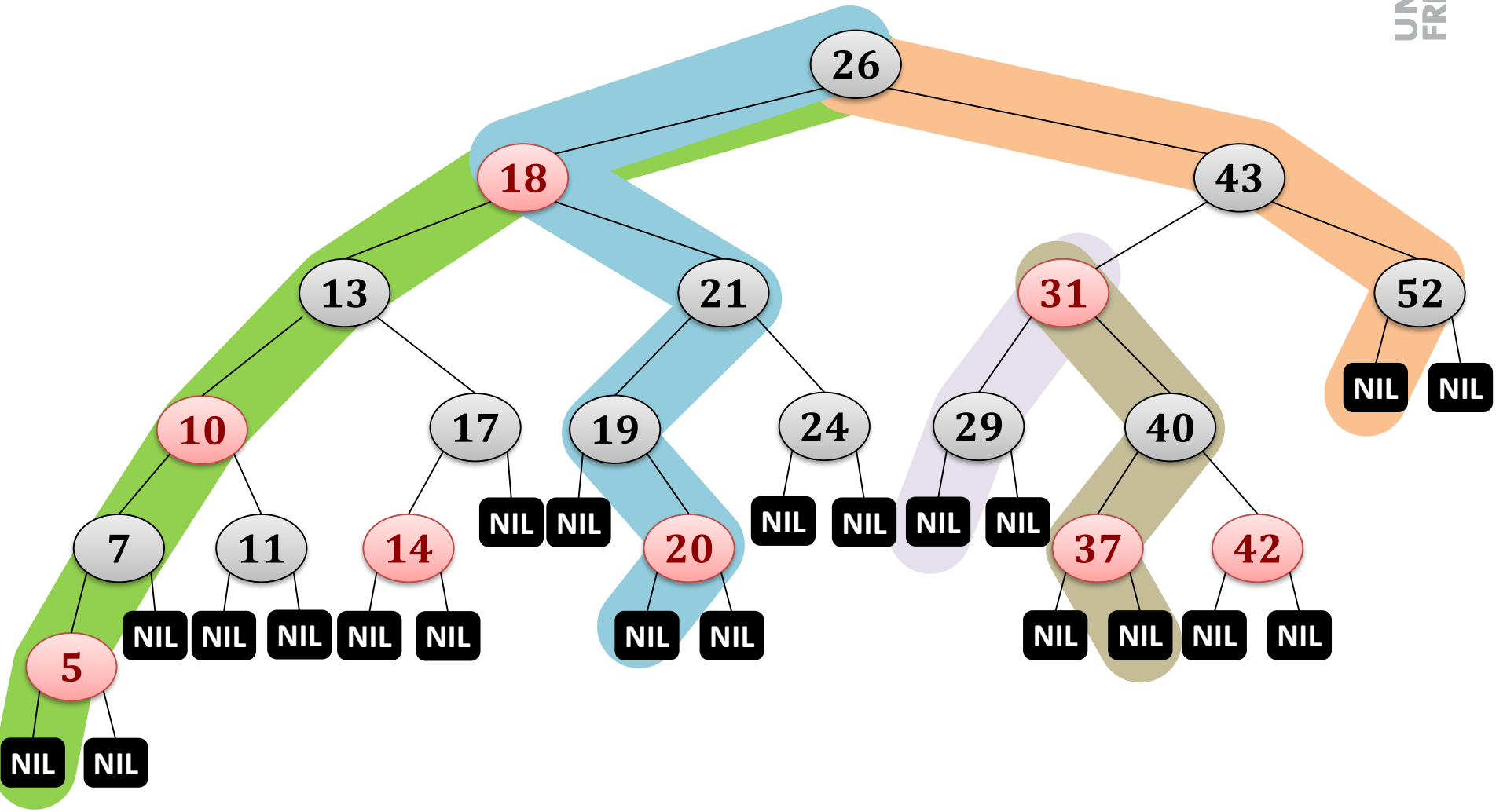


Red-Black Trees : Properties



- Root and leaves (NIL-nodes) are black.
- Red nodes have two black children.

Red-Black Trees : Properties



- In each subtree, all direct paths from the root to leaves of the subtree have the same number of black nodes.

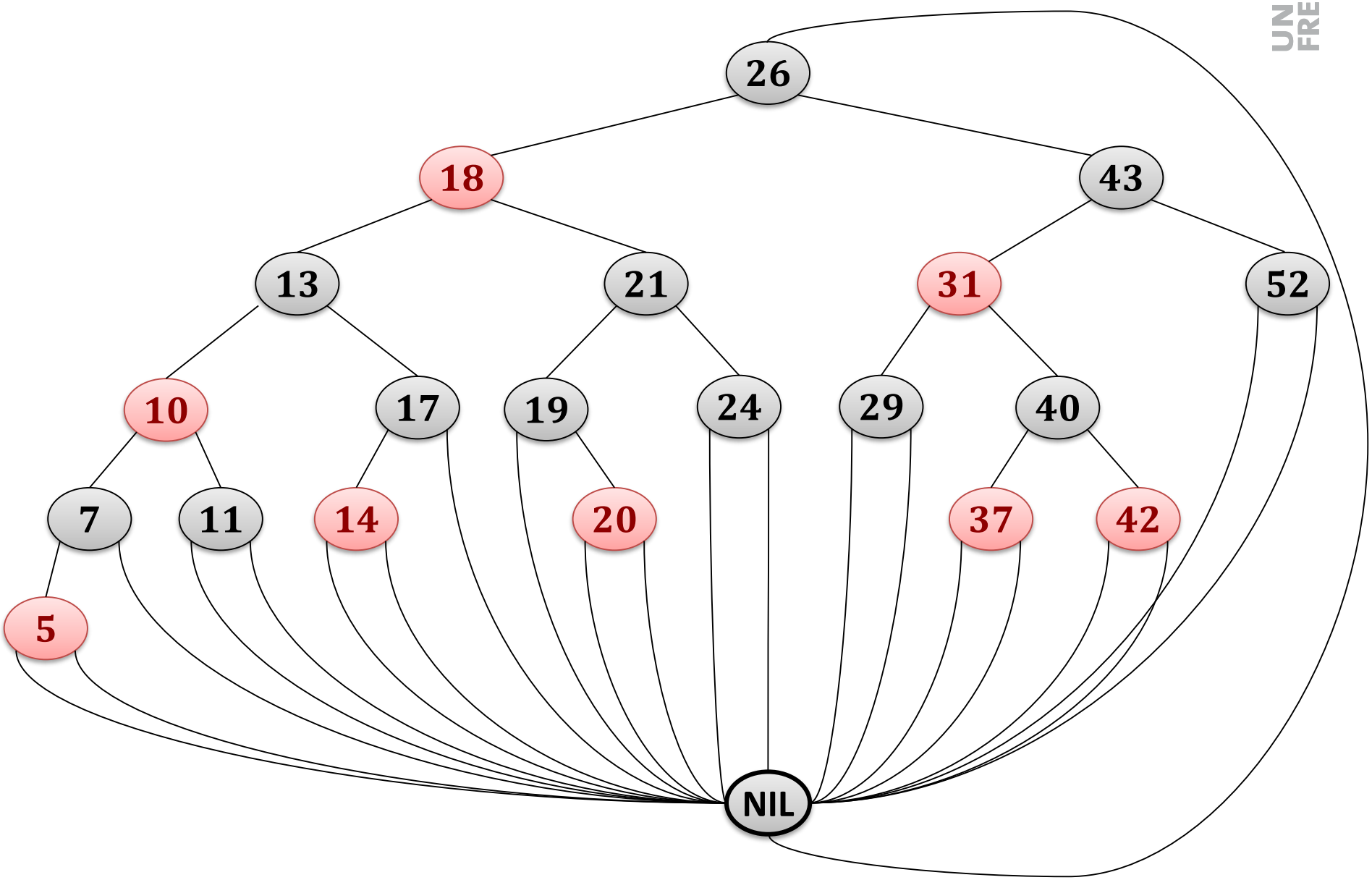
Sentinel

To simplify the code…

Sentinel Node: *NIL*

- Replaces all None/null pointers
- *NIL.key* is not defined
- *NIL.color = black*
  - The leaves of the tree are the NIL-nodes (they are all black)
  - Represents all leaves of the tree
- *NIL.left*, *NIL.right*, *NIL.parent* can be set arbitrarily
  - We have to make sure that they are never read and interpreted wrongly.
  - If it simplifies the code, one can assign values to *NIL.parent*, ...

Red-Black Trees : Sentinel

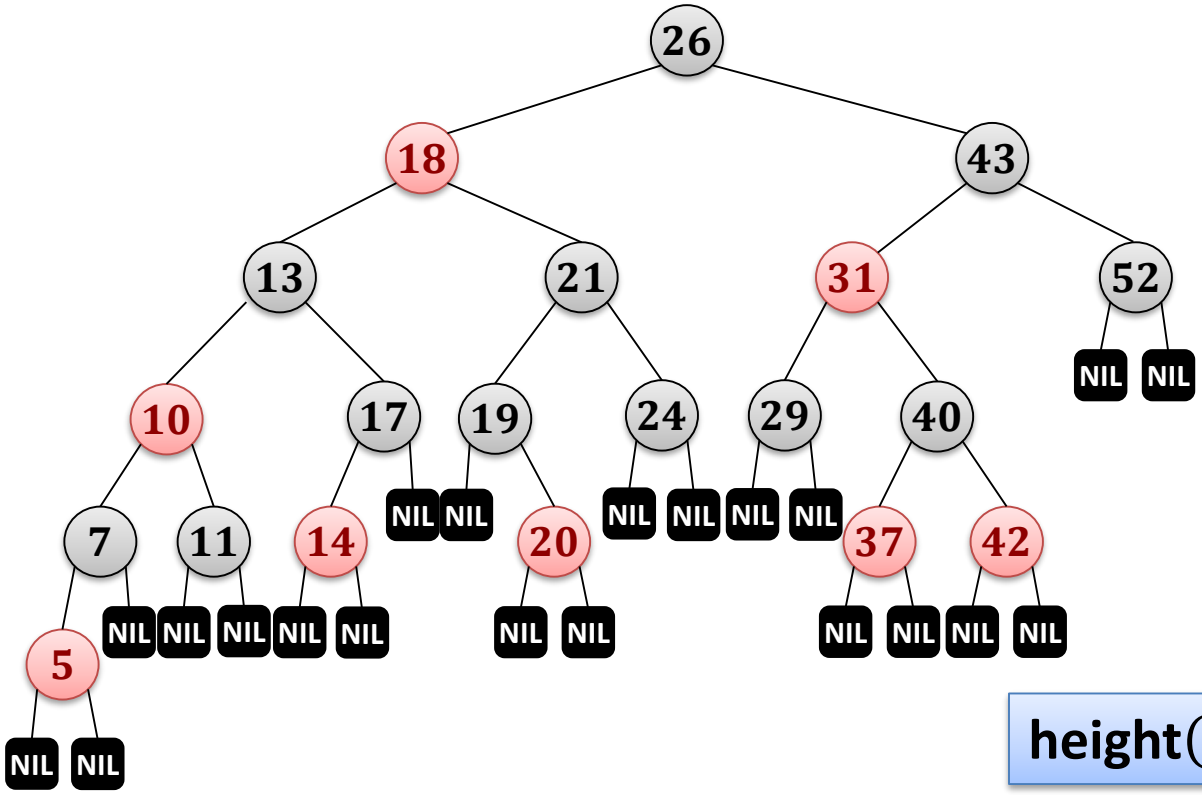


Height / Black-Height

Definition: The **height (H)** of a node v is the maximal length of a path from node v to a leaf node (NIL).

Definition: The **black-height (BH)** of a node v is the number of black nodes on every path from v to a leaf (NIL) in its subtree.

- The node v itself is **not** counted, the leaf (NIL, if $\neq v$) is however counted!



$$\text{height}(v) \leq 2 \cdot \text{black-height}(v)$$

Black-Height ↔ Number of Nodes

UNI FREIBURG

Lemma: The **number of inner nodes** in the subtree of a node v ($v \neq$ NIL) with **black-height $BH(v)$** is

$$\geq 2^{BH(v)} - 1$$

Proof by induction over the black-height $BH(v)$ of v:

- **Induction Base:** $BH(v) = 1$: #inner nodes $\geq 2^1 - 1 = 1$
  - v itself is an inner node!

- **Induction Step:**
  - Consider a node v with black-height $BH(v) > 1$
  - **Induction Hypothesis:**
    - Subtrees with black-height $s < BH(v)$ have $\geq 2^s - 1$ inner nodes
    - Subtrees with black-height $s = BH(v)$ have $\geq 2^{s-1} - 1$ inner nodes
      - Such subtrees contain subtrees with black-height $s - 1$

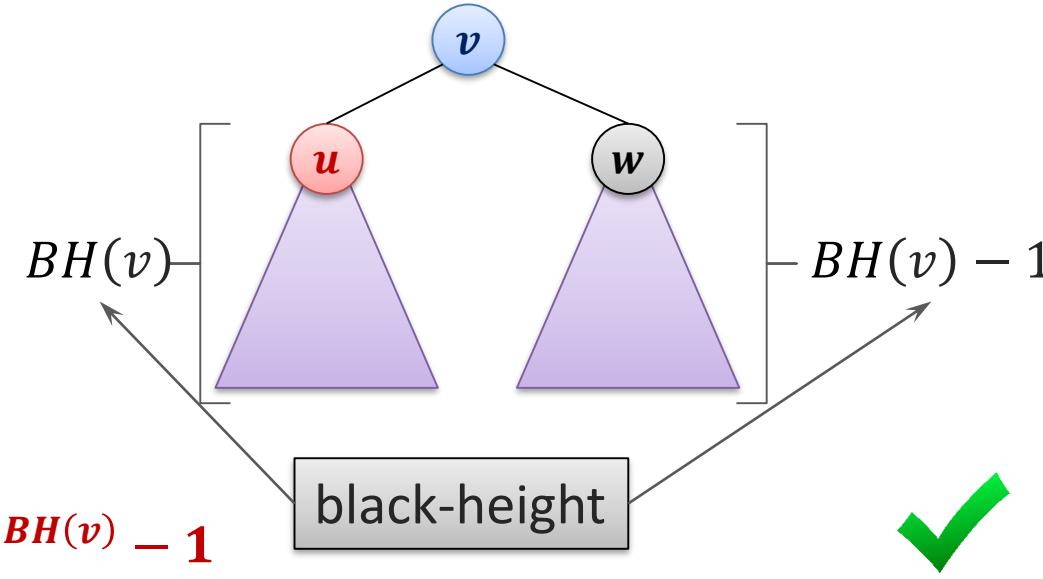
Black-Height ↔ Number of Nodes

The **number of inner nodes** in the subtree of a node v ($v \neq$ NIL) with **black-height $BH(v)$** is

$$\geq 2^{BH(v)} - 1$$

Proof by induction over the black-height $BH(v)$ of v:

- **Induction Step:**
  - Consider a node v with black-height $BH(v)$
  - Both subtrees have black-height $\geq BH(v) - 1$
  - Induction hypothesis: Both subtrees have $\geq 2^{BH(v)-1} - 1$ nodes.
  - **#nodes in subtree of v:** $\geq 2 \cdot \left(2^{BH(v)-1} - 1\right) + 1 = 2^{BH(v)} - 1$



Depth of a Red-Black Tree

UNI FREIBURG

Theorem:

The **depth** of a red-black tree is $\leq 2\log_2(n+1)$.

Proof:

- Number of inner nodes : n (all nodes except the NIL-nodes)
- From the lemma from before, we get

$$n \geq 2^{BH(root)} - 1$$

- When solving for $BH(root)$, one gets

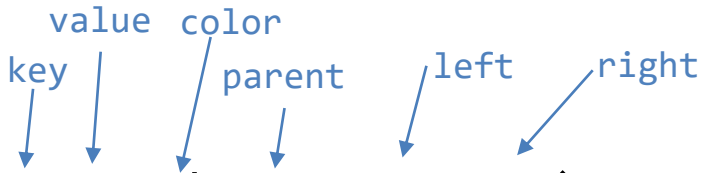
$$BH(root) \leq \log_2(n+1)$$

- The theorem now follows because

$$\text{height} \leq 2 \cdot \text{black–height}$$

Red-Black Trees : Insert

insert(x): First insert new node as usually, new node is **red**




```
if root == NIL then
    root = new Node(x,a,red,NIL,NIL,NIL)
else
    v = root;
    while v.key != x do
        if v.key > x then
            if v.left == NIL then
                w = new Node(x,a,red,v,NIL,NIL); v.left = w
            v = v.left
        else if v.key < x then
            if v.right == NIL then
                w = new Node(x,a,red,v,NIL,NIL); v.right = w
            v = v.right
    v.value = a
```

Red-Black Trees : Insert

Red-black tree properties after inserting

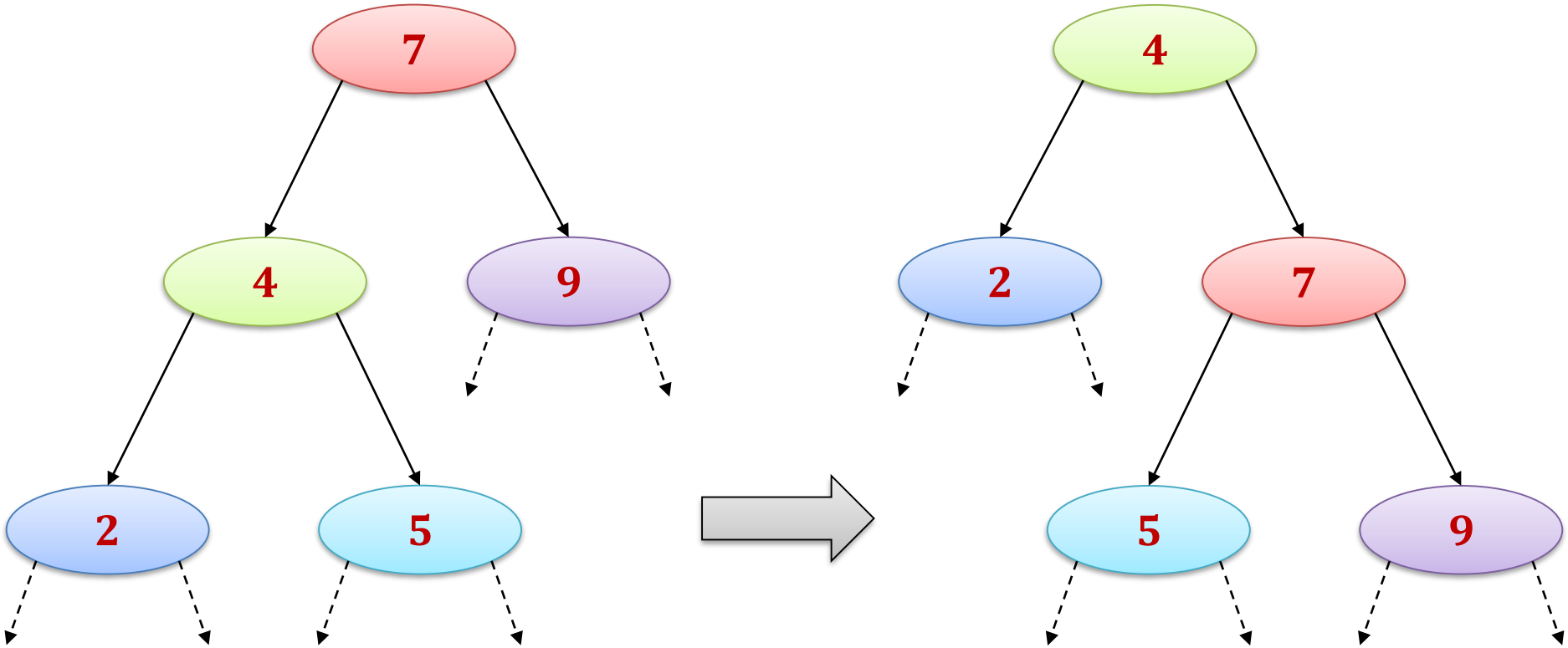
1. All nodes are red or black.
2. The root is black.
3. The leaves (NIL) are black.
4. Red nodes have two black children.
5. For every node v, all paths from v to leaves in the subtree of v have the same number of black nodes.

- Properties are all satisfied, except
  - Inserted nodes v is the root (Bedingung 2 nicht erfüllt)
  - The node $v.\mathrm{parent}$ is red (Bedingung 4 nicht erfüllt)
- If v is the root, we can just insert v as a black node.
- If $v.\mathrm{parent}$ is red, we have to adjust the tree.
  - such that 1, 3, and 5 remain satisfied and at the end 2 and 4 are satisfied

Rotations

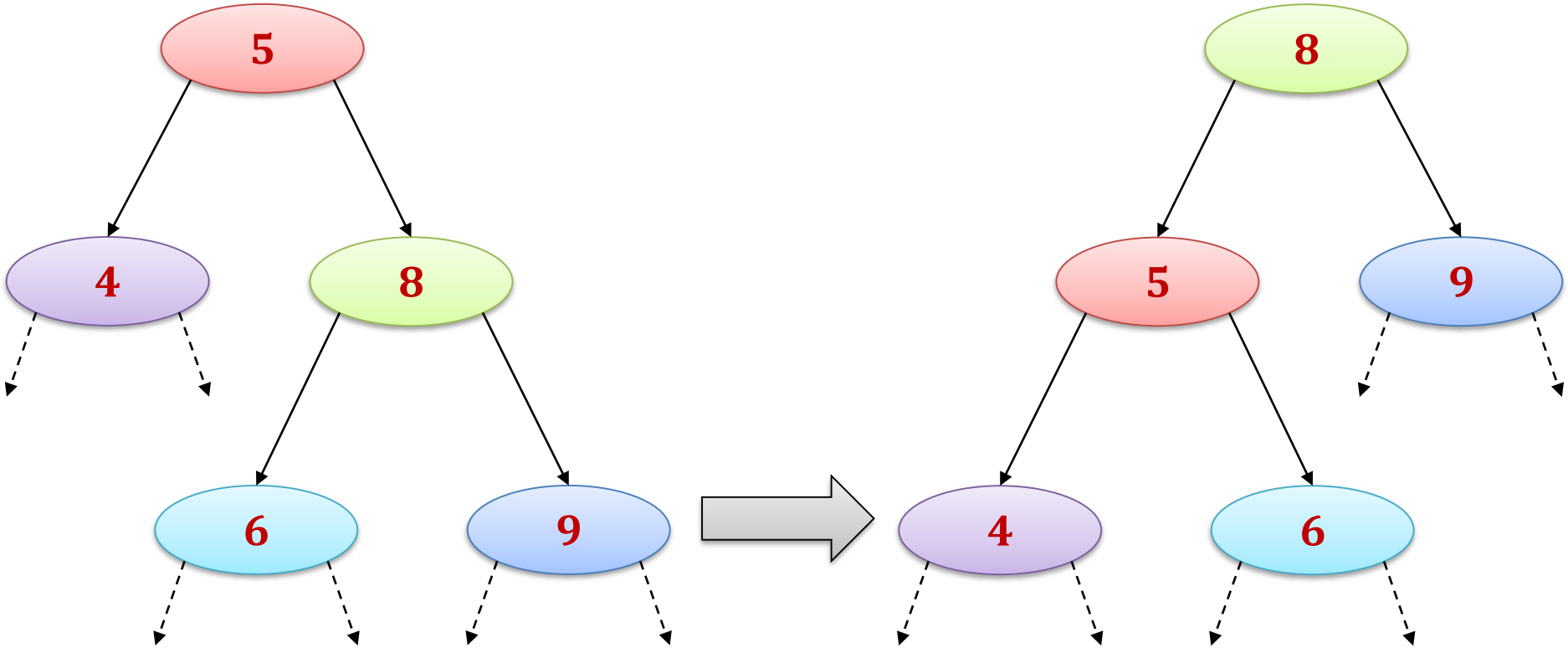
Right Rotation

- Operation to reorder a binary search tree locally
- Changes topology, preserves binary search tree property



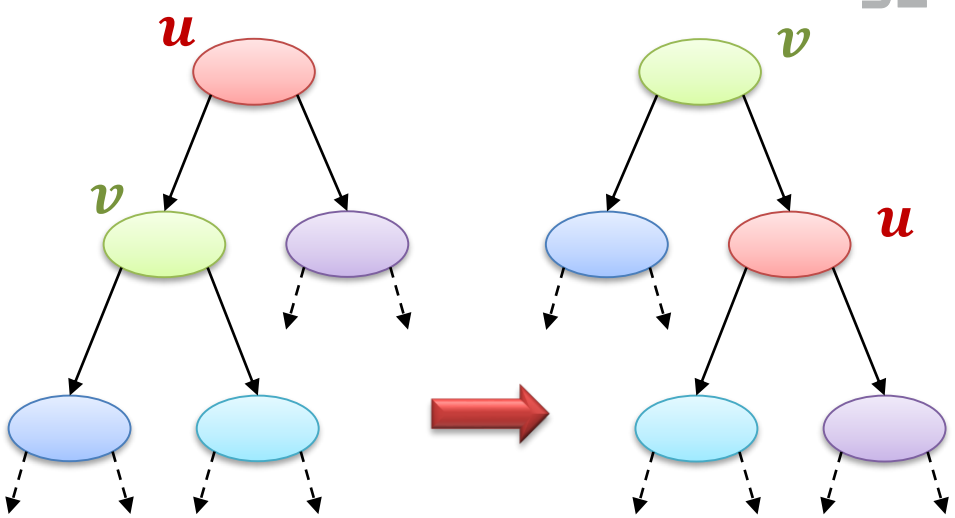
Rotations

Left Rotation



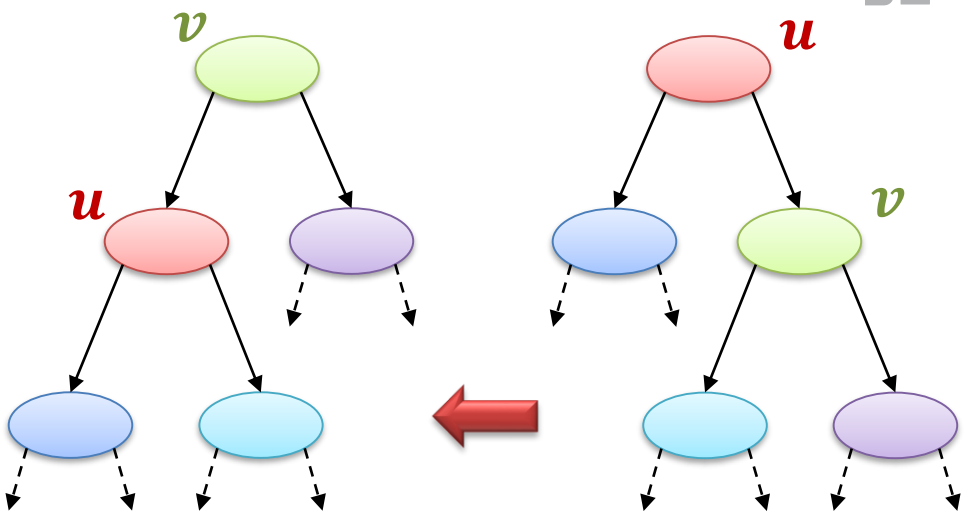
Right Rotation

```
right-rotate(u,v):
   u.left = v.right
   u.left.parent = u
   if u == root then
      root = v
   else
      if u == u.parent.left then
         u.parent.left = v
      else
         u.parent.right = v
   v.parent = u.parent
   v.right = u
   u.parent = v
```



Running Time: $O(1)$

Left Rotation




```
left-rotate(u,v):
   u.right = v.left
   u.right.parent = u
   if u == root then
      root = v
   else
      if u == u.parent.left then
         u.parent.left = v
      else
         u.parent.right = v
   v.parent = u.parent
   v.left = u
   u.parent = v
```

Running Time: $O(1)$

Correctness: Rotations

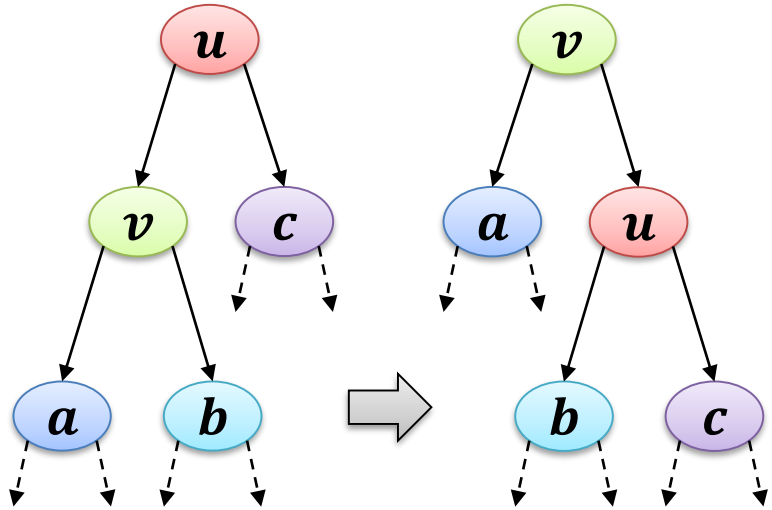
Lemma: Rotations preserve the “binary search tree” property.

Sorted order before rotation:

- $(a < v < b) < u < c$

Sorted order after rotation:

- $a < v < (b < u < c)$



Red-Black Trees : Insert

Red-black tree properties after inserting

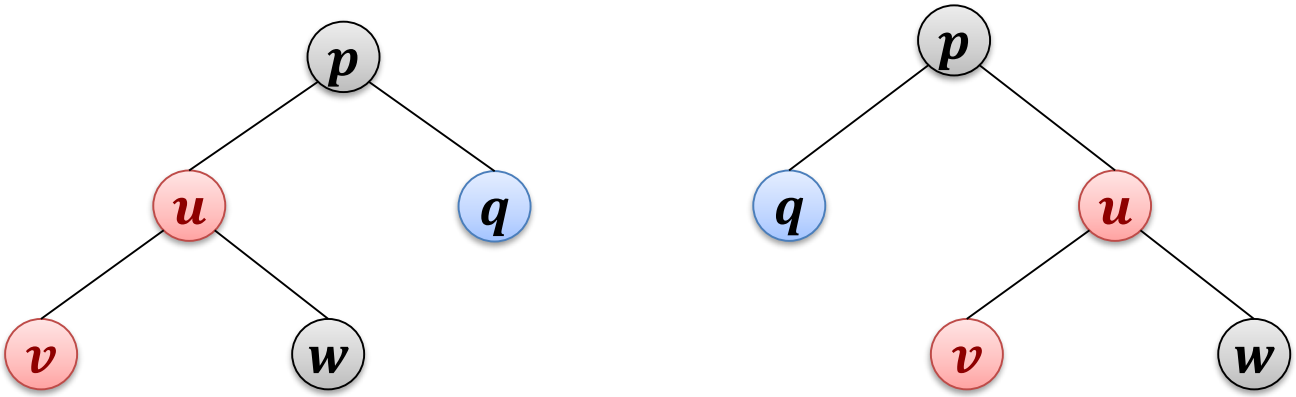
1. All nodes are red or black.
2. The root is black.
3. The leaves (NIL) are black.
4. Red nodes have two black children.
5. For every node v, all paths from v to leaves in the subtree of v have the same number of black nodes.

- Properties are all satisfied, except
  - Inserted nodes v is the root (Bedingung 2 nicht erfüllt)
  - The node $v.\mathrm{parent}$ is red (Bedingung 4 nicht erfüllt)
- If v is the root, we can just insert v as a black node.
- If $v.\mathrm{parent}$ is red, we have to adjust the tree.
  - such that 1, 3, and 5 remain satisfied and at the end 2 and 4 are satisfied

Red-Black Trees : Insert

Adjust tree after inserting:

- Assumptions:
  - v is red, v has two black children
  - $u := v.\text{parent}$ is red (otherwise, we are done)
  - v is the left child of u (other case symmetric)
  - Sibling node w of v (right child of u) is black
  - All red nodes except u have 2 black children

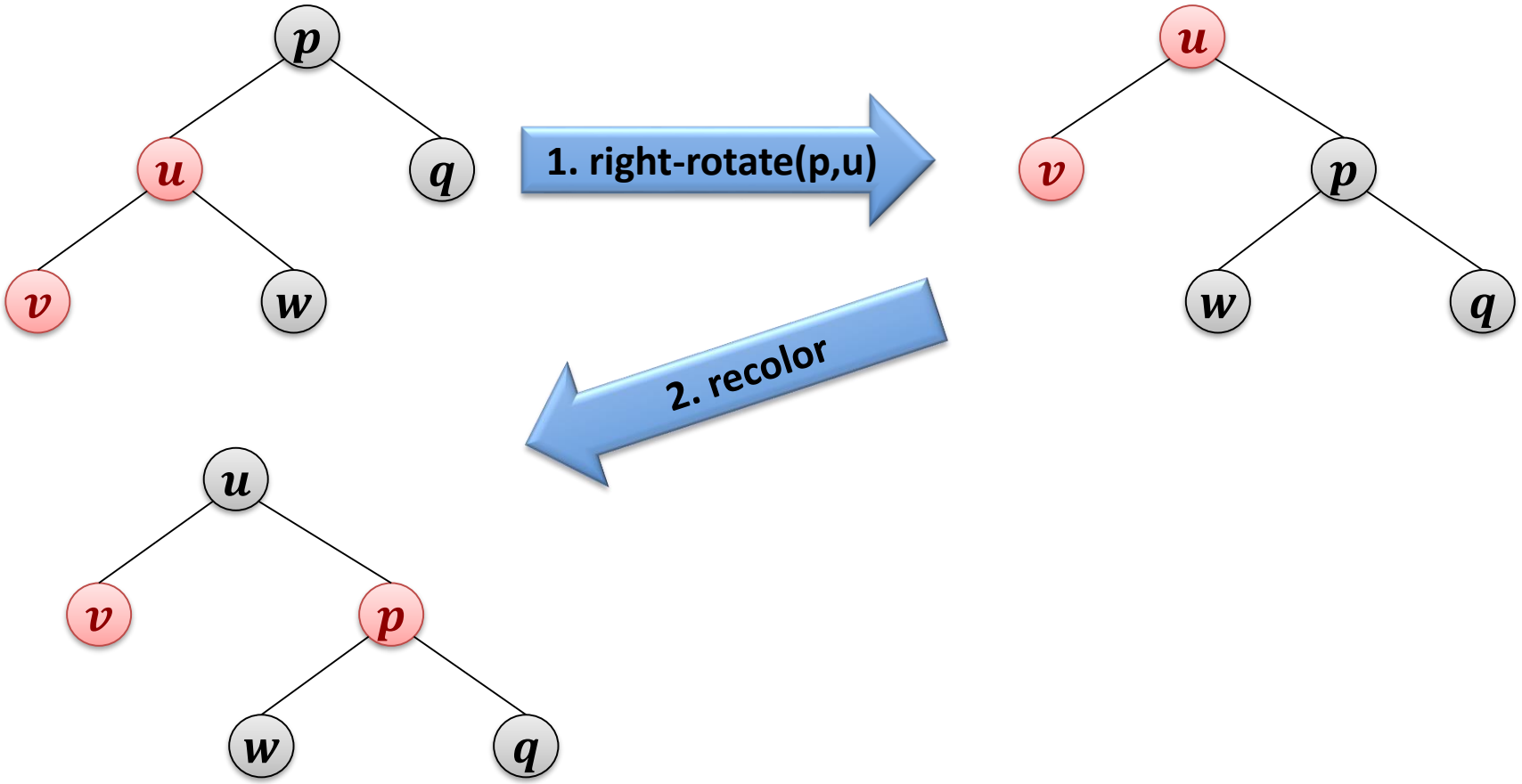


- Case distinction based on **color of q (sibling of u)** and based on **$u = p.\text{left}$ or $u = p.\text{right}$**

Red-Black Trees : Insert

Case 1: Sibling q of u is black

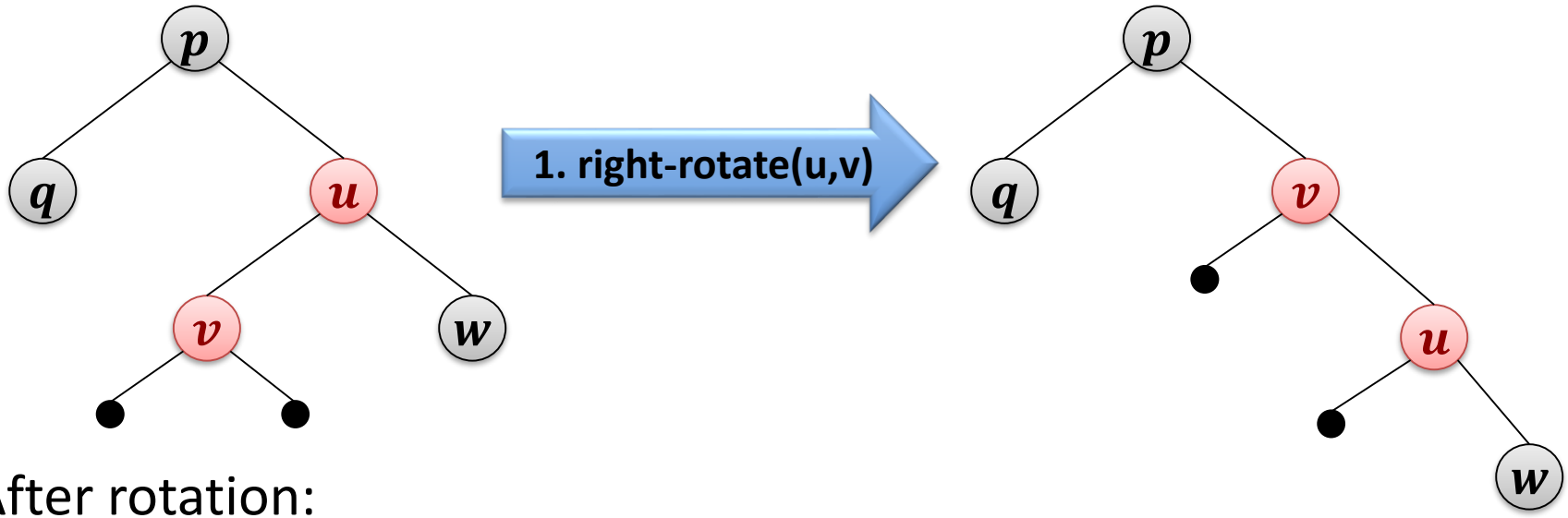
- **Case 1a: $u = p.\text{left}$**



Red-Black Trees : Insert

Case 1: Sibling q of u is black

- **Case 1b: $u = p.\text{right}$**



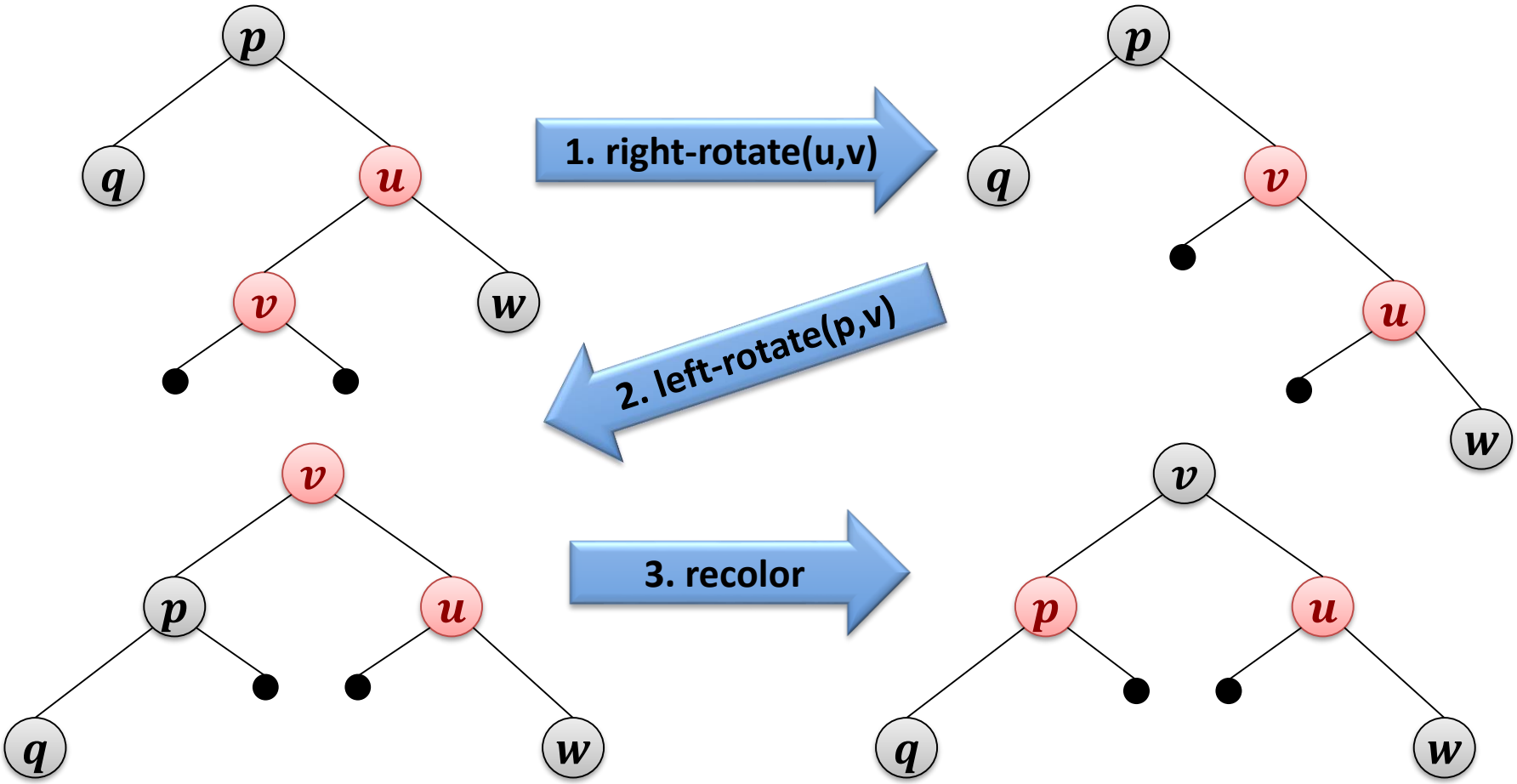
- After rotation:
  - symmetrical to Case 1a
  - u, v are red, sibling q is black
  - u is right child of v, v is right child of p
  - resolve by

    left-rotate(p,v) and recoloring

Red-Black Trees : Insert

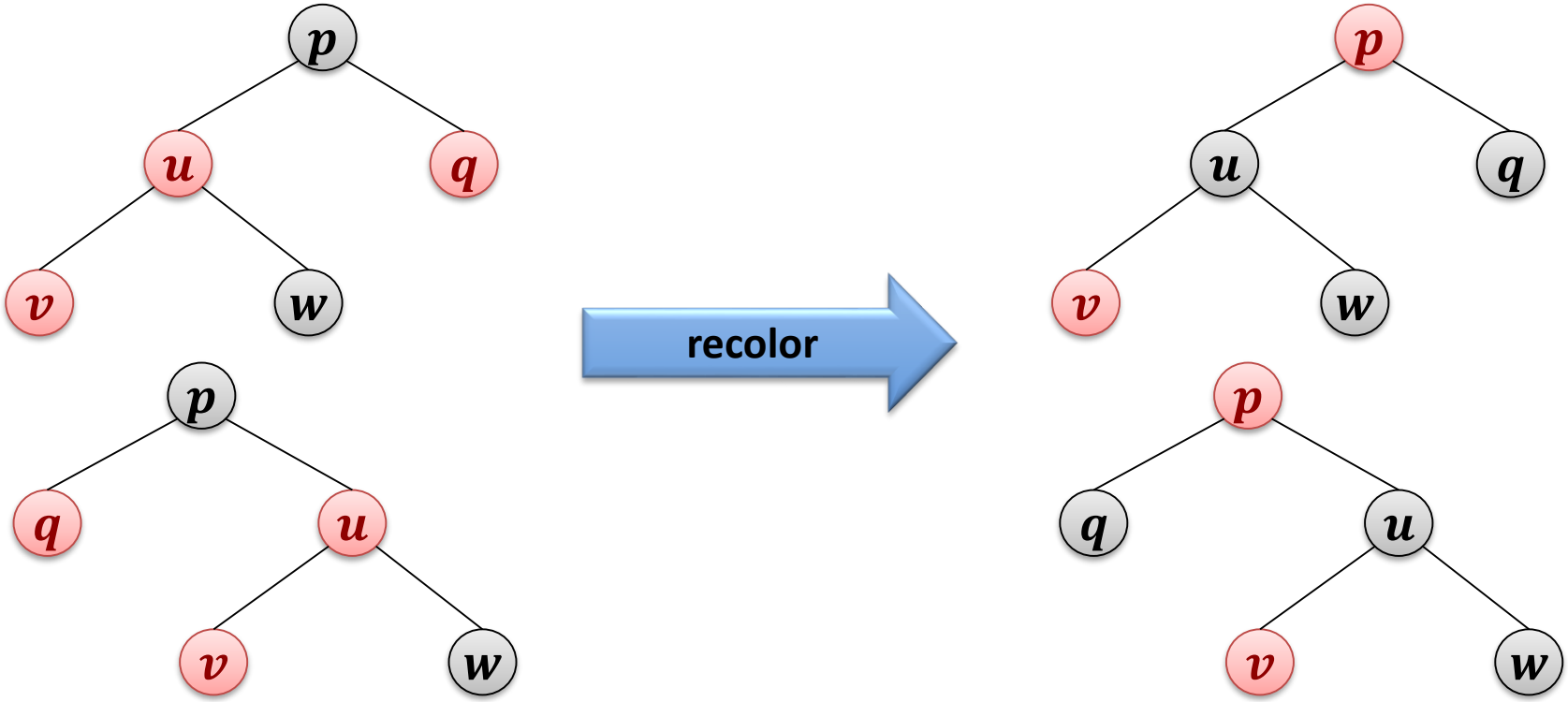
Case 1: Sibling q of u is black

- **Case 1b: $u = p.\text{right}$**



Red-Black Trees : Insert

Case 2: Sibling q of u is red



- If $p.\text{parent}$ is black, we are done
  - This is also the case if $p == \text{root}$ (we then do $\text{root.color} := \text{black}$)
- Otherwise, we are in the same case as in the beginning
  - but closer to the root!

Red-Black Trees : Insert

1. Insert new key normally.
   - Color of new node is red.
2. As long as we are in Case 2, recolor
   - Case 2: red node v with a red parent node u
   - Sibling of u is also red
3. As soon as we are not in Case 2
   - If it is a valid red-black tree, we are done.
   - If the root is red, the root has to be recolored black.
   - Otherwise, we are in Case 1a or 1b (or symmetrically) and, we can get a valid red-black tree with at most 2 rotations and recoloring of at most 2 nodes.

- **Running time:** $O(\text{tree depth}) = O(\log n)$

Red-Black Trees : Delete

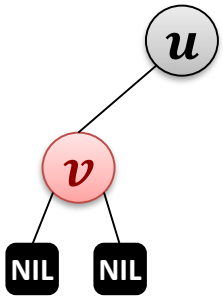
1. As usual, find a node v that can be deleted.
   - Node v has at most one non-NIL child!

Case Distinction (color of v and $v.\mathbf{parent}$)

Assumption: v is left child of u (other case symmetric)

Case 1: node v is red

- As v must have at least 1 NIL child, because of the red-black tree properties, v must have 2 NIL children.

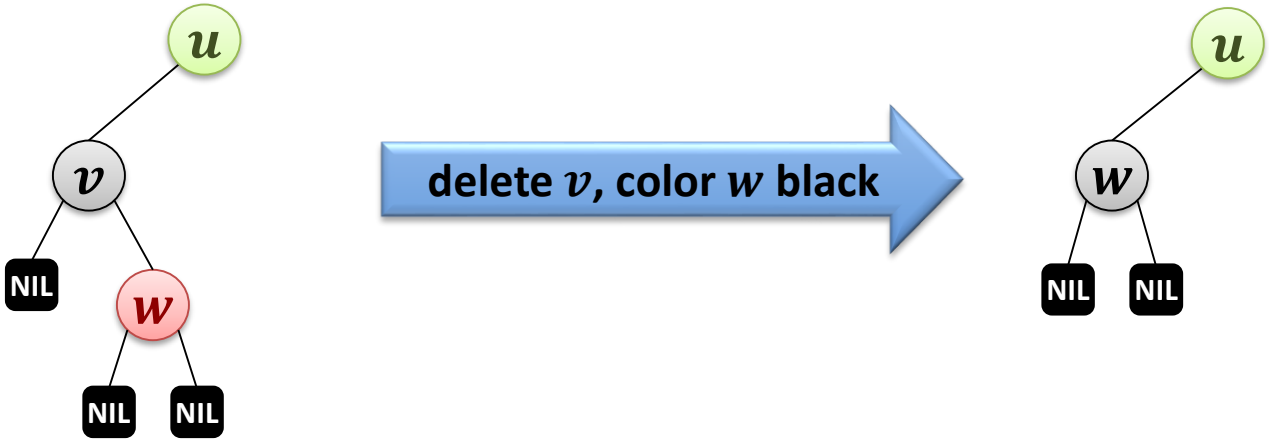


- v can just be deleted.
  - The tree remains a valid red-black tree.

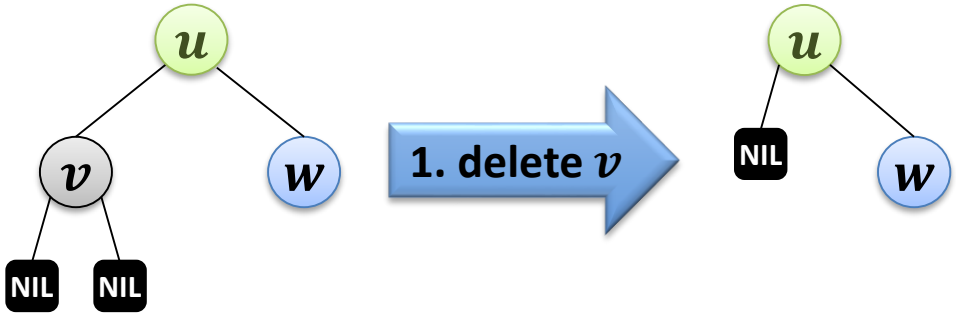
Red-Black Trees : Delete

Case 2: node v is black

- **Case 2a: v has one (red) none-NIL child w**



- **Case 2b: v has only NIL children**



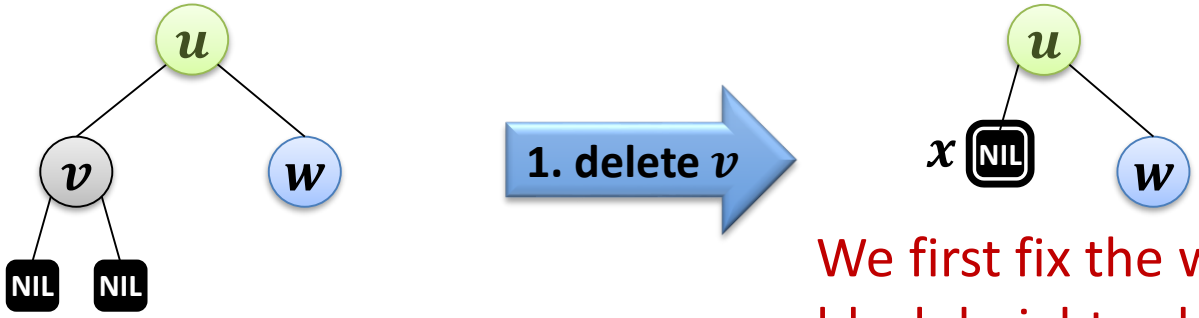
Node u now has black-height 1 to the left (instead of 2).

→ **We have to adjust the tree.**

Red-Black Trees : Delete

Problematic case:

- **Node v has only NIL children**



We first fix the wrong black-height coloring double black.

- **Goal**: We want to move the additional "black" up the tree until we can move it to a red node or until we reach the root (and we therefore do not have a problem any more).
- **Case distinction**: Color if w and of the children of w
  - Observation: w cannot be NIL (because of the black-height)!

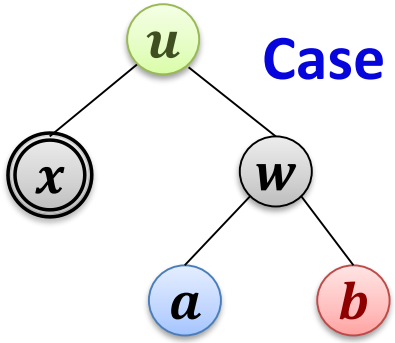
Red-Black Trees : Delete

Assumption:

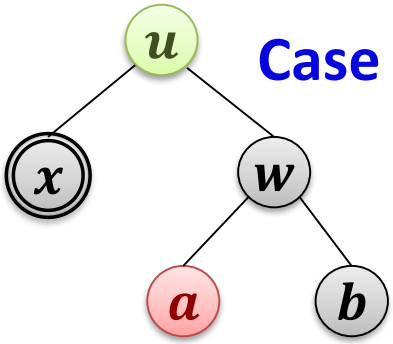
- Double black node x
- Parent u has arbitrary color (marked as green)
- x is left child of u (right child: symmetrically)
- Sibling of x (right child of u) is w

Case distinction:

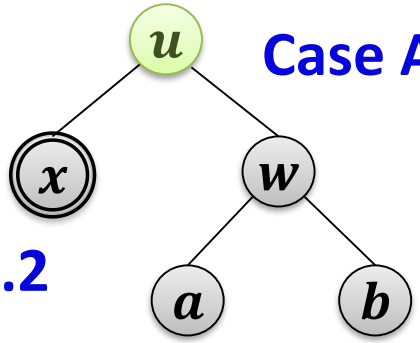
- **Case A: w is black, Case B: w is red**



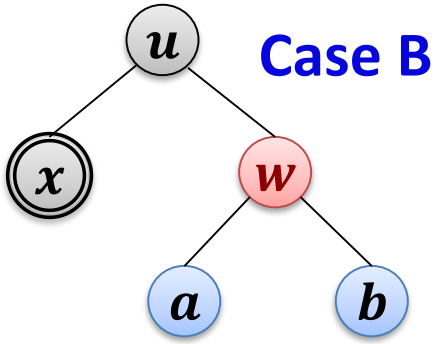
Case A.1



Case A.2



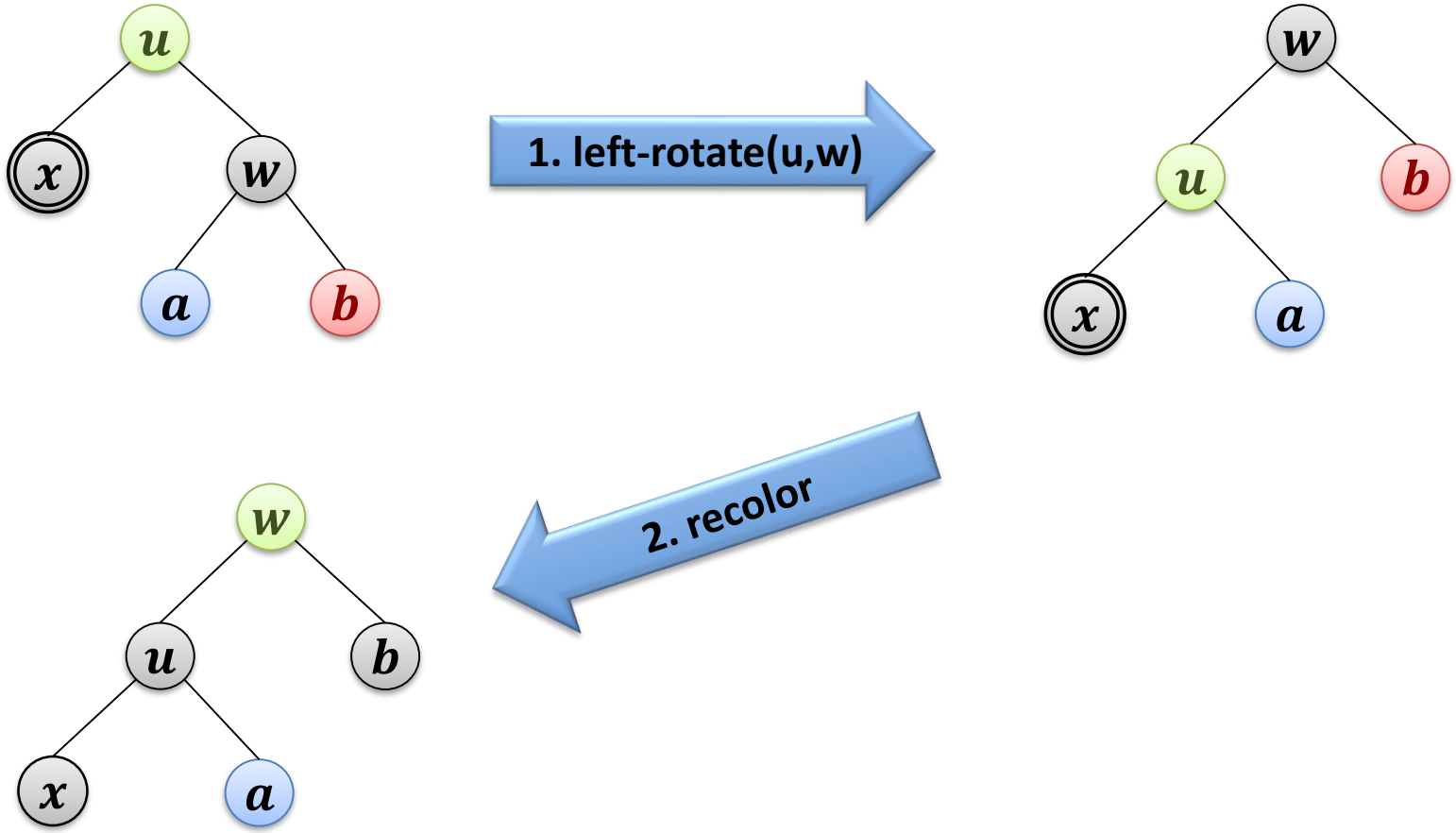
Case A.3



Case B

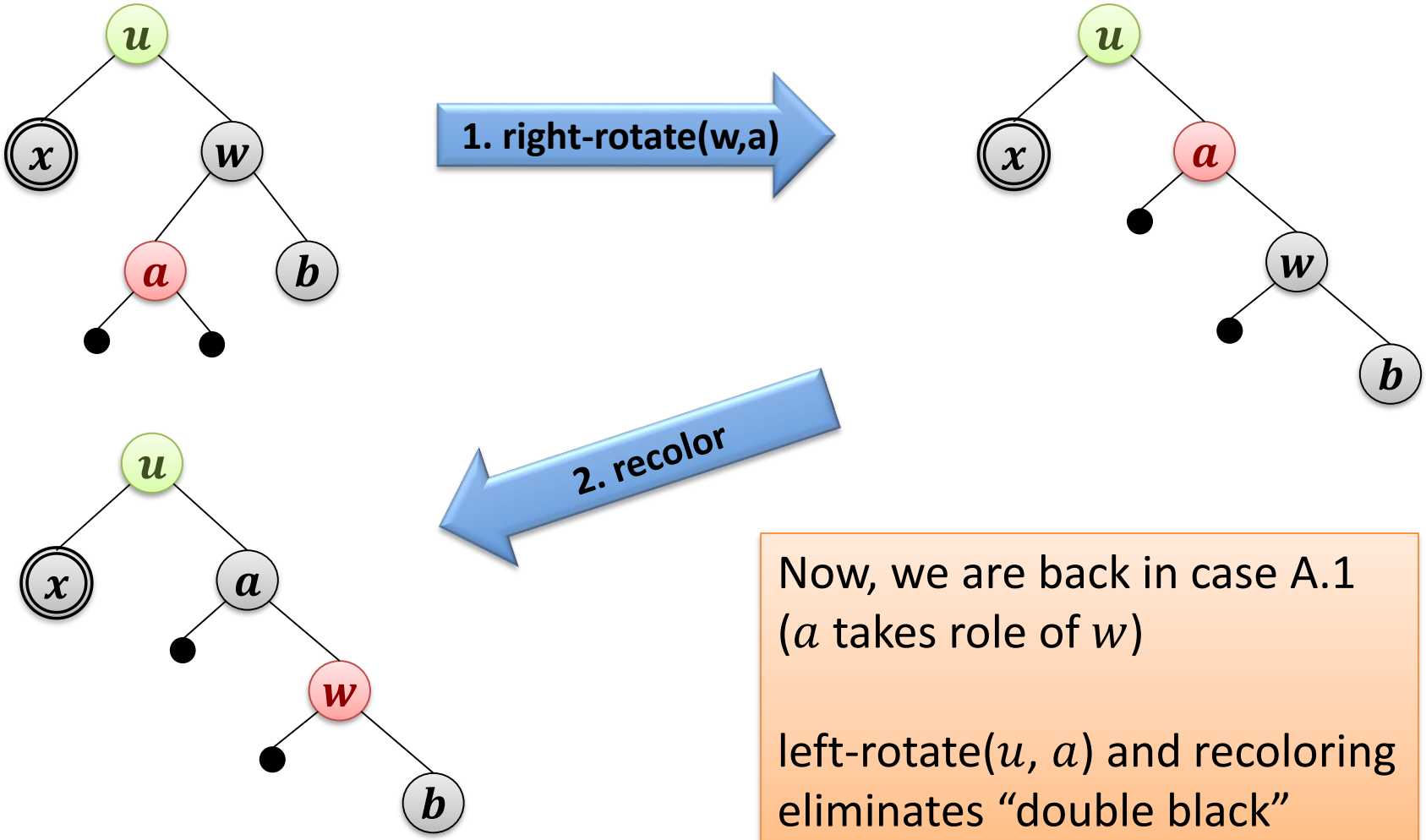
Red-Black Trees : Delete

Case A.1: w is black, right child of w is red



Red-Black Trees : Delete

Case A.2: w is black, left chilf of w is red, right child of w is black



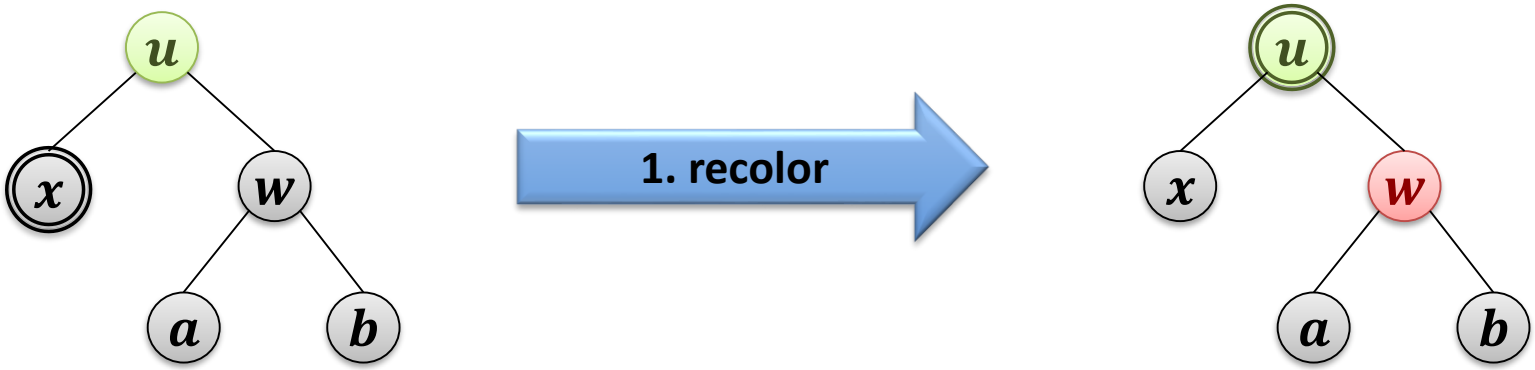
Now, we are back in case A.1 (a takes role of w)

left-rotate(u, a) and recoloring eliminates "double black"

Red-Black Trees : Delete

UNI FREIBURG

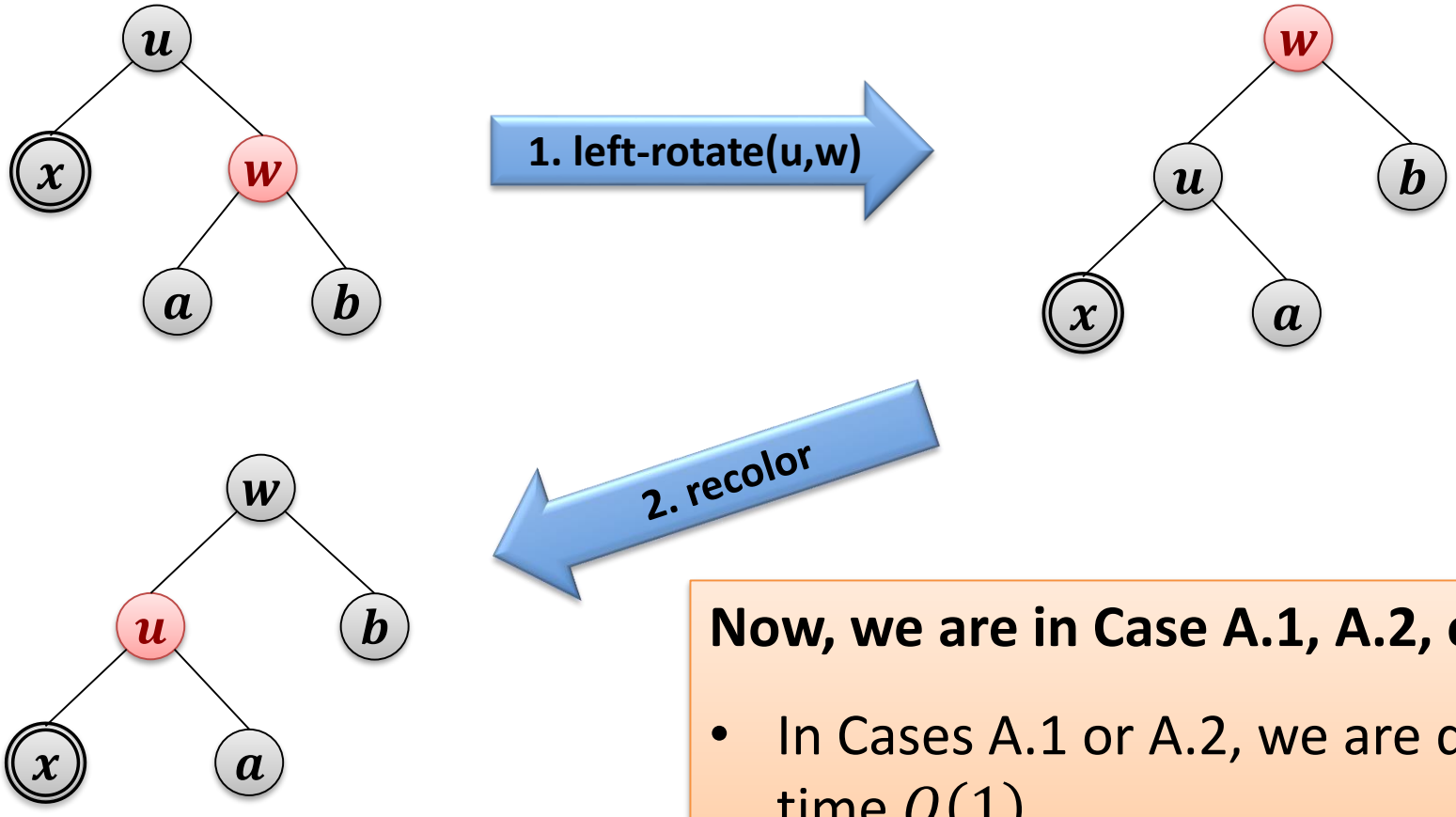
Case A.3: w is black, both children of w are black



- The additional "black" moves one level up.
- If u is red, u can now just be colored black.
- Otherwise, node u takes the role of x and it can again be in one of the Cases A.1, A.2, A.3, or B (see next slide).
- Case A.3 can happen at most $O(\log n)$ times.
  - If $u == \text{root}$, we can just remove the additional "black".
- In Cases A.1 and A.2, we are done after constant time.

Red-Black Trees : Delete

Case B: w is red



Now, we are in Case A.1, A.2, or A.3

- In Cases A.1 or A.2, we are done in time $O(1)$.
- In Case A.3, we are also done in time $O(1)$, because u is red!

Red-Black Trees : Delete

1. As usual
   - Find node v with at most 1 NIL child that can be deleted.
   - v is possibly predecessor/successor of node with the key to be deleted.
2. If v is black and has two NIL children, we have to adjust.
   - One then gets a black node x with an additional "black".
3. Possible cases: A.1, A.2, A.3, B
   - Case A.1: With 1 rotation and recoloring $O(1)$ nodes, we are done.
   - Case A.2: With 1 rotation and recoloring $O(1)$ nodes, we are in Case A.1.
   - Case A.3: If $x.\mathrm{parent}$ is red, we are done after recoloring $O(1)$ nodes, if $x.\mathrm{parent}$ is black, the additional "black" moves towards the root and we are again in Cases A.1, A.2, A.3, or B.
   - Case B : 1 rotation and recoloring $O(1)$ nodes yields A.1, A.2, or A.3, if A.3, then $x.\mathrm{parent}$ is red

- **Running time:** $O(\text{tree depth}) = O(\log n)$

Red-Black Trees : Summary

UNI FREIBURG

Red-Black Trees

- Binary search trees, that always have depth $O(\log n)$.
- In $O(\log n)$ time, one can insert at new (key, value)-pair or delete the (key, value) pair for a given key.

Dictionary Implementation

- **Worst-case running time $O(\log n)$** for the operations

  find, insert, delete, minimum, maximum, predecessor, successor

- Range query, where k (key, value)-pairs are returned requires running time $O(k + \log n)$.

Comparison to hash tables

- amortized $O(1)$ running time $\Leftrightarrow$ worst-case $O(\log n)$ running time
- Binary search trees support many additional operations.

AVL Trees

- Direct alternative to red-black trees
- AVL trees are binary search trees such that for every node v

$$|H(v.\text{left}) - H(v.\text{right})| \leq 1$$

- Instead of a color (red/black) each node stores the depth of its subtree.
- AVL also always have depth $O(\log n)$
  - even with a slightly better constant than red-black trees
- AVL condition can be maintained with $O(\log n)$ after every insert or delete operation
- Comparison to red-black trees
  - Find in AVL trees is slightly faster
  - Insert / delete in AVL trees is slightly slower

(a, b)-Trees

- Parameter $a \geq 2$ and $b \geq 2a - 1$
- Keys/values are only stored in the leaves
- All leaves are at the same depth
- If the root is not a leaf, it has between 2 and b children
- All other inner nodes have between a and b children
  - A (a, b)-tree is a search tree, but not a binary search tree!

Similar: B-Trees

- Here, the inner nodes also store keys
- For large a, b one needs more space than for binary search trees.
  - Because one has to provide space for blocks of size b
- The trees are however more shallow
- Particularly good, e.g., for file systems (where memory access is expensive)

Further Alternatives

AA-Trees:

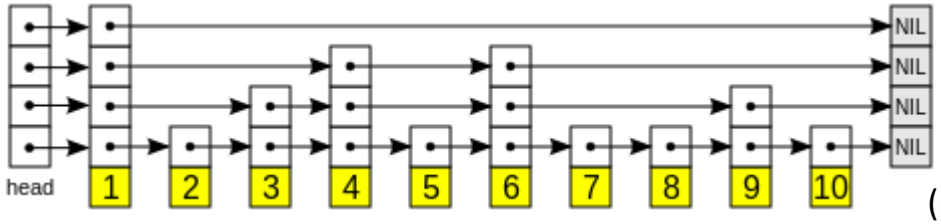
- similar to red-black trees (only right children can be red)

Splay Trees:

- Binary search trees with additional good properties
  - Elements that have been accessed recently are closer to the root
  - Good if several nodes can have the same key
  - However, not strictly balanced

Skip Lists:

- Linked lists with additional shortcuts
  - not a balanced search tree, but similar properties



(picture from wikipedia)