




*Sometimes, I wonder if non-tech friends wonder about my search history...*

how to remove a black child without affecting its red parents 

All Images Videos News More Settings Tools

About 7,350,000 results (0.50 seconds)

**how to kill child of fork? - Stack Overflow**  
<https://stackoverflow.com/questions/13273836/how-to-kill-child-of-fork> ▼  
3 answers  
Nov 8, 2012 - I have the following code which create a **child fork**. And I want to kill the **child** before it finish its execution in the **parent**. how to do it? #include ...

<b>Fork and child/parent processes</b>	1 answer	16 May 2017
pid from <b>fork()</b> never goes to <b>parent</b> , only <b>child</b>	1 answer	12 Aug 2016
How to use <b>fork()</b> to daemonize a <b>child</b> process ...	4 answers	27 Jan 2012
Are <b>child</b> processes created with <b>fork()</b> automatically ...	1 answer	18 Nov 2011

More results from stackoverflow.com



Universität  
Zürich<sup>UZH</sup>

Institut für Informatik

# Informatics II

## Tutorial Session 10

Wednesday, 4<sup>th</sup> of May 2022

Discussion of Exercises 8 and 9,  
Binary Search Trees, Red-Black Trees

14.00 – 15.45

BIN 0.B.06



## Agenda

- Review of Exercise 8
  - Binary Search Trees
- Review of Exercise 9
  - Red-Black Trees



**Universität  
Zürich<sup>UZH</sup>**

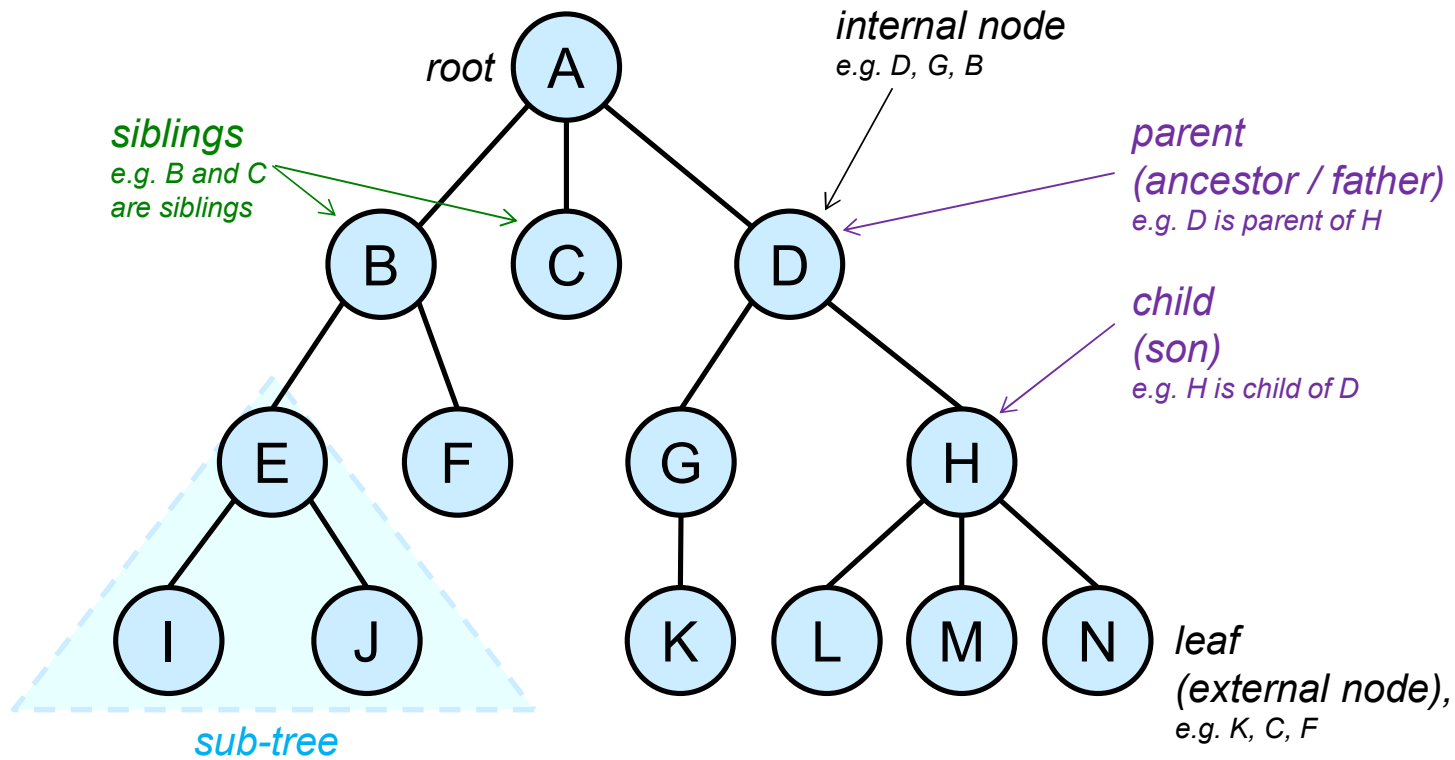
**Institut für Informatik**

# Trees

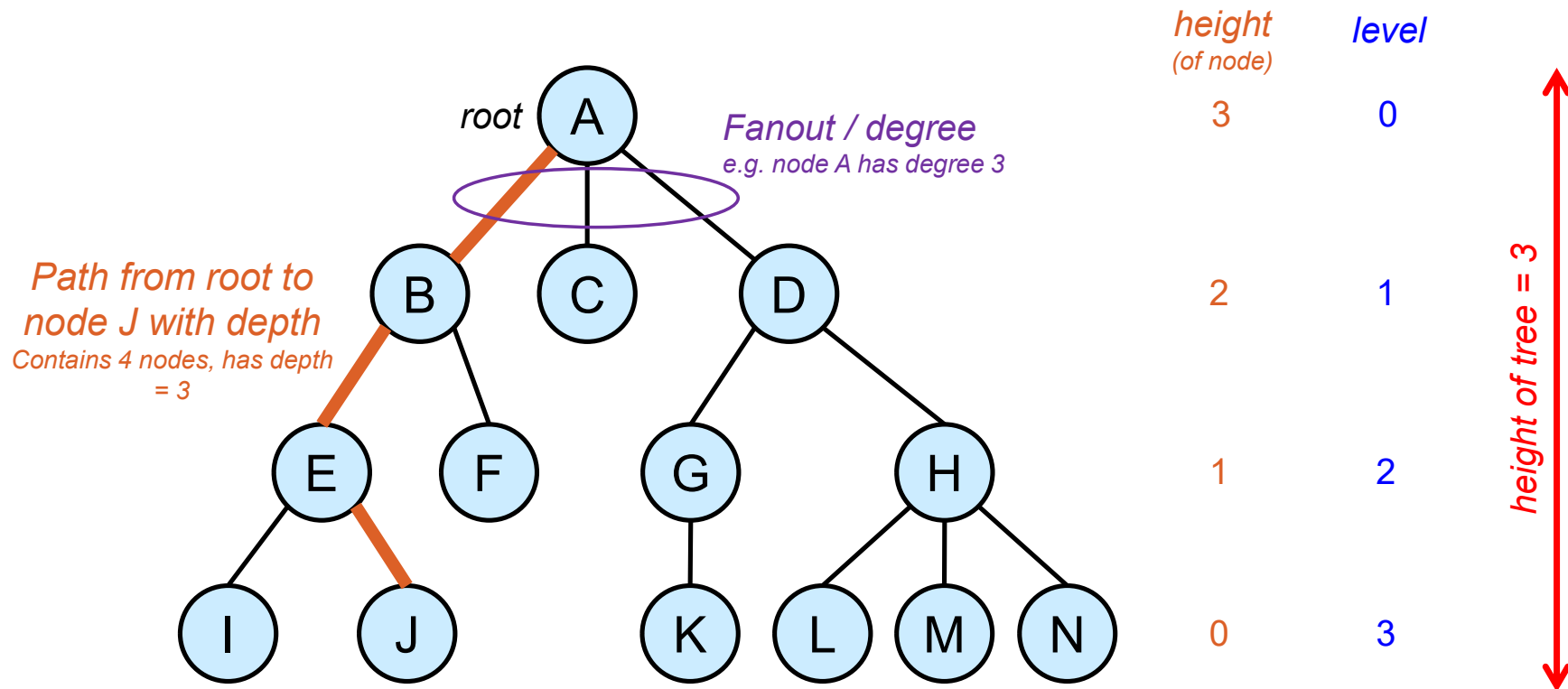
- Nomenclature
- Binary Trees
- Binary Tree Traversals
- Binary Search Trees



## Trees: Nomenclature



## Trees: Nomenclature



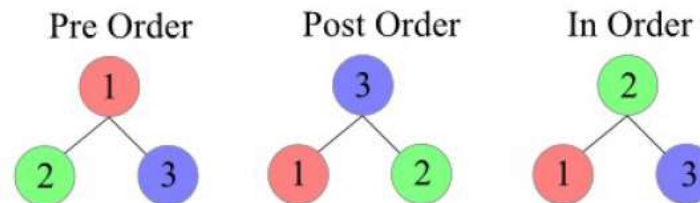


## Binary Tree Traversals: Introduction and Overview

Tree traversals are rules on how to visit each node of a binary tree exactly once. There are three famous kinds of depth-first traversals and one kind of breadth-first traversal (visiting neighbors first) which will be discussed in more detail later in the lecture.

Depth-first traversals:

- **Inorder:** left – root – right
- **Preorder:** root – left – right
- **Postorder:** left – right – root



Breadth-first traversal:

- **Levelorder:** from left to right, from root level to leaf level



## Binary Tree Traversals: Remarks

- Note that the **sequence** of nodes stemming from a particular traversal method in general **does not allow** to **unambiguously reconstruct** the original tree.
- An **exception** is the **preorder traversal** which allows to reconstruct the original tree in an unambiguous and efficient manner; therefore, the preorder traversal can be used (and considered as) as representational structure for trees, e.g. in order to store a tree in a file.





## Binary Tree Traversals

Recursively apply the principle stated before to all nodes of the tree by beginning at the root node.

For example: inorder traversal:                      Left – Root – Right

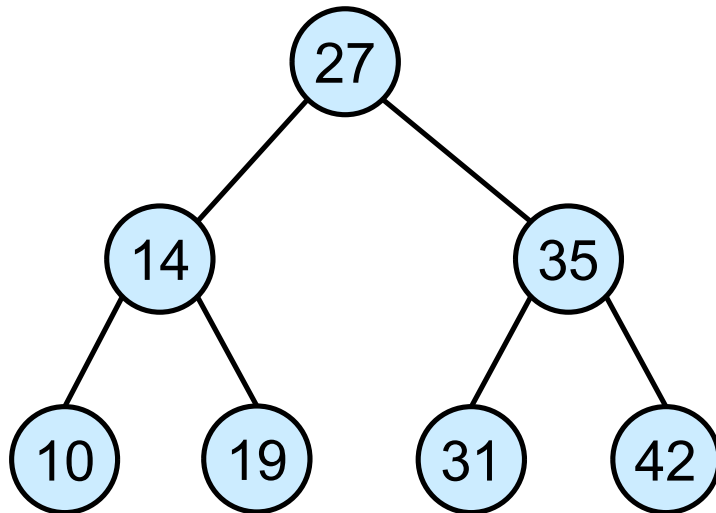
*Pseudo code:*

```
inorder(tree T) {  
    inorder(left_subtree(T));  
    visit(root);  
    inorder(right_subtree(T));  
}
```



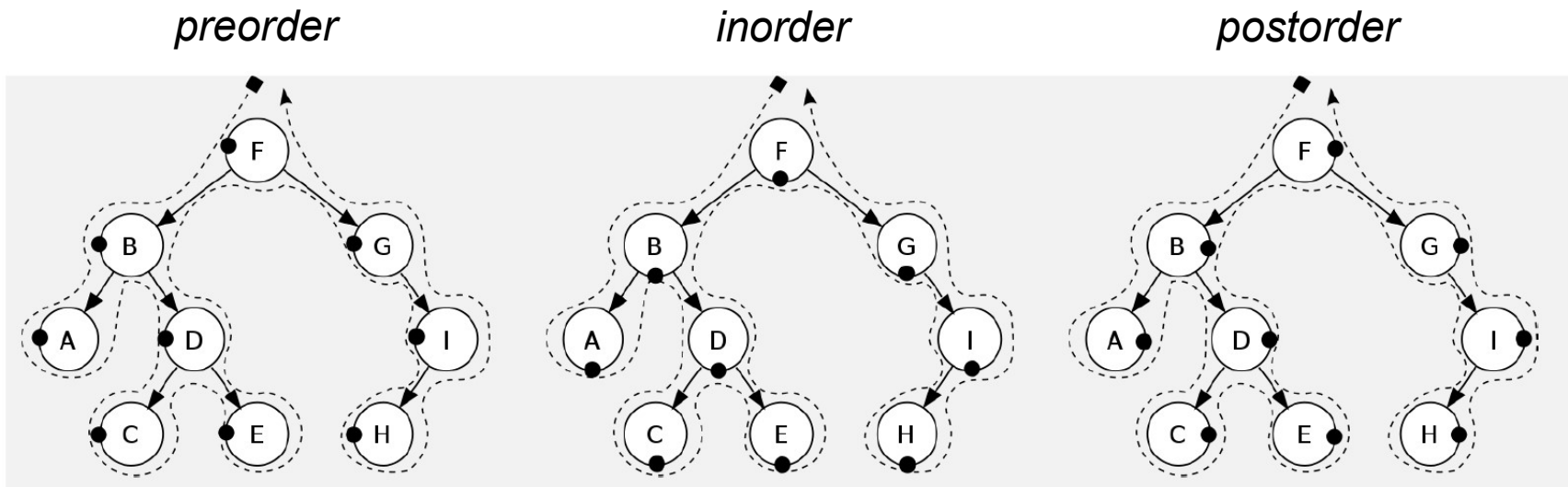
## Binary Tree Traversals: Example

*What are the inorder, preorder, postorder and levelorder traversals of the following binary tree?*



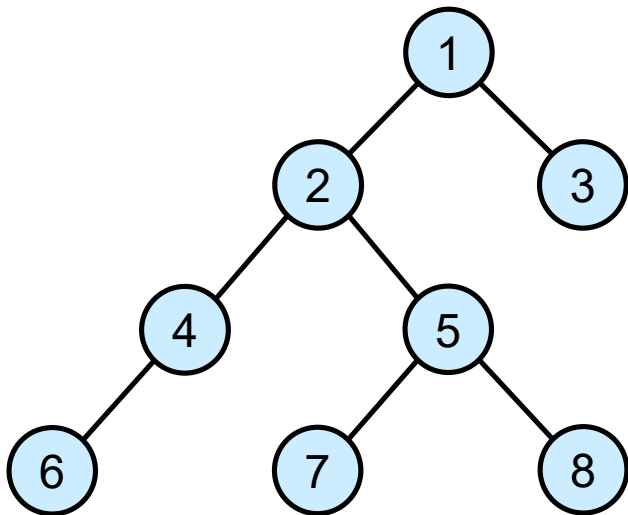
Inorder: 10, 14, 19, 27, 31, 35, 42  
Preorder: 27, 14, 10, 19, 35, 31, 42  
Postorder: 10, 19, 14, 31, 42, 35, 27  
Levelorder: 27, 14, 35, 10, 19, 31, 42

## Binary Tree Traversals: Visual Help



## Binary Tree Traversals: Iterative vs. Recursive Implementation

Instead of applying a recursive algorithm, the depth-first traversals (inorder, preorder, postorder) can also be implemented iteratively using a (explicit) stack. Similarly, a levelorder traversal can be implemented iteratively using a queue.



Step	0	1	2	3	4	5	6	7	8
			4	6		7			
Stack content	1	2 3	5 3	5 3	5 3	8 3	8 3	3	∅
Output of function «next»	-	1	2	4	6	5	7	8	3



## Tree Traversals: Applications

What are these tree traversal strategies useful for?

- Preorder: easy and efficient way to store binary search trees with the possibility to reconstruct them unambiguously
- Inorder: retrieving the node values of a binary search tree in ascending order
- Postorder: important for compiler design (in assembly: operand operand opcode)
- Levelorder: used in heapsort algorithm



## Additional Exercises: Tree Walks as a Tool

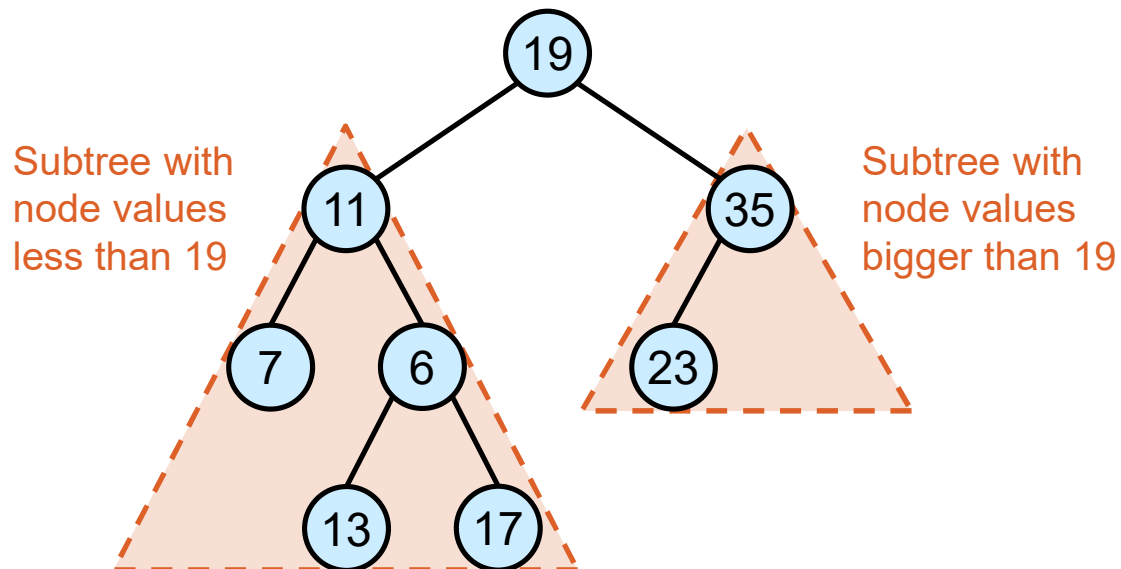
Write functions in C which calculate:

- the sum of the node values in a binary tree and
- the average of the node values in a binary tree.

## Binary Search Trees

A binary search tree (BST) is a binary tree where all descendent nodes to the left are smaller or equal than the ancestor node and all descendants to the right are bigger or equal than the ancestor node.

*Example:*



*Remark:* The above example assumes that there are no duplicates in the BST (which is a common and oftentimes applicable assumption in use cases where BST make sense as an ADT. If there can be duplicates, this comes with a bunch of problems / potential ambiguities that need to be dealt with.



## Binary Search Trees: Inserting Nodes / Building a Tree: Example

Insert the following key values into an initially empty binary search tree:

a) 67, 21, 57, 89, 12, 11, 7, 55

b) 42, 34, 29, 22, 17, 4





## Binary Search Trees: Degeneration

If keys are inserted in sorted order into a BST, the tree will **degenerate to a list**.

It is said: the tree is **unbalanced**.

We will see techniques later which will ensure that this does not happen (red-black trees).





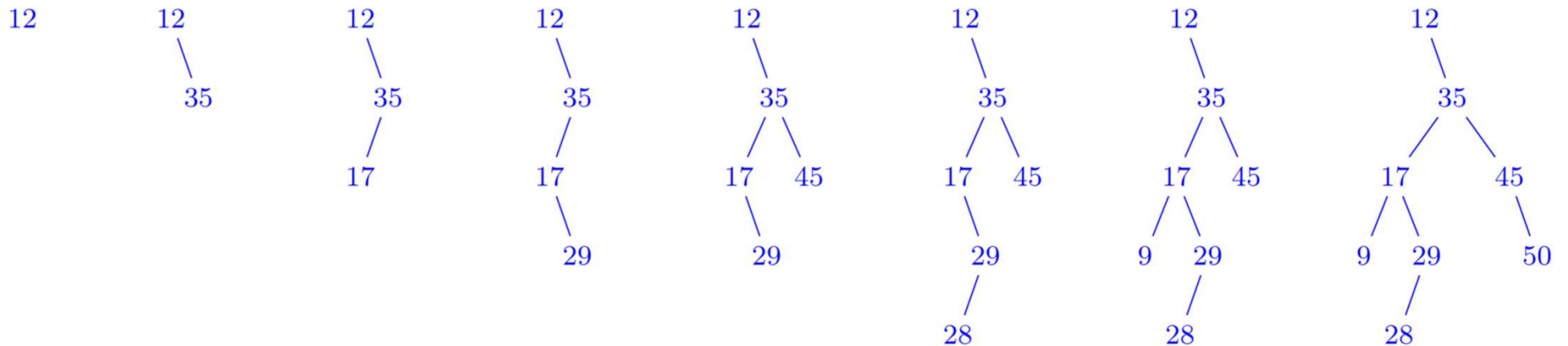
## **Additional Exercise: Degeneration Detector**

Describe conceptually (or using pseudo code) how you would write an algorithm which checks whether a binary search tree has degenerated to a linked list and returns true if this is the case.



## Exercise 8, Task 1a: Inserting Nodes into a BST

The following values are inserted in the given order into a previously empty BST: [12, 35, 17, 29, 45, 28, 9, 50]. Draw the tree structure when all values have been inserted.





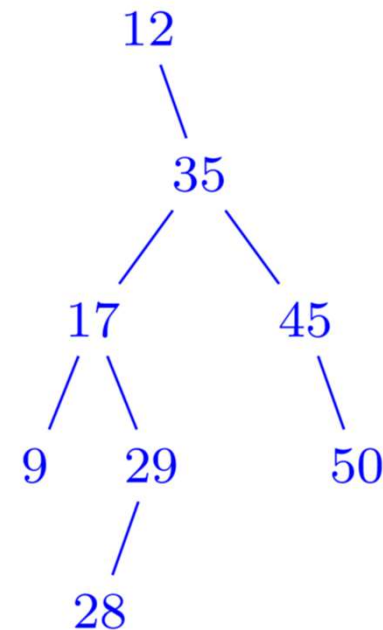
## Exercise 8, Task 1b: Tree Nomenclature

In the resulting tree from task 1a determine the following properties of the tree and justify your results:

- height of the tree
- depth of node with value 45

*Height of the tree: 4*

*Depth of node with value 45: 2*





## Exercise 8, Tasks 1c and 1d: Asymptotic Complexity of Operations

(c) Is it true that the time complexity of search in a binary search tree is  $O(1)$  in the best case? Identify when this best case is achieved.

**True**, complexity is  $O(1)$  if the node happens to be at the root.

(d) Is it true that the time complexity of search in a binary search tree is  $O(\log n)$  in the worst case where  $n$  is the number of nodes? Identify when this worst case is achieved.

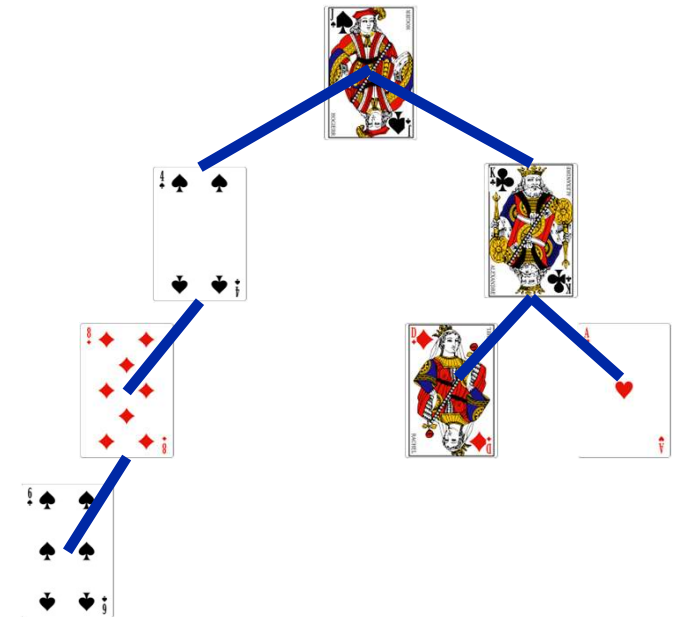
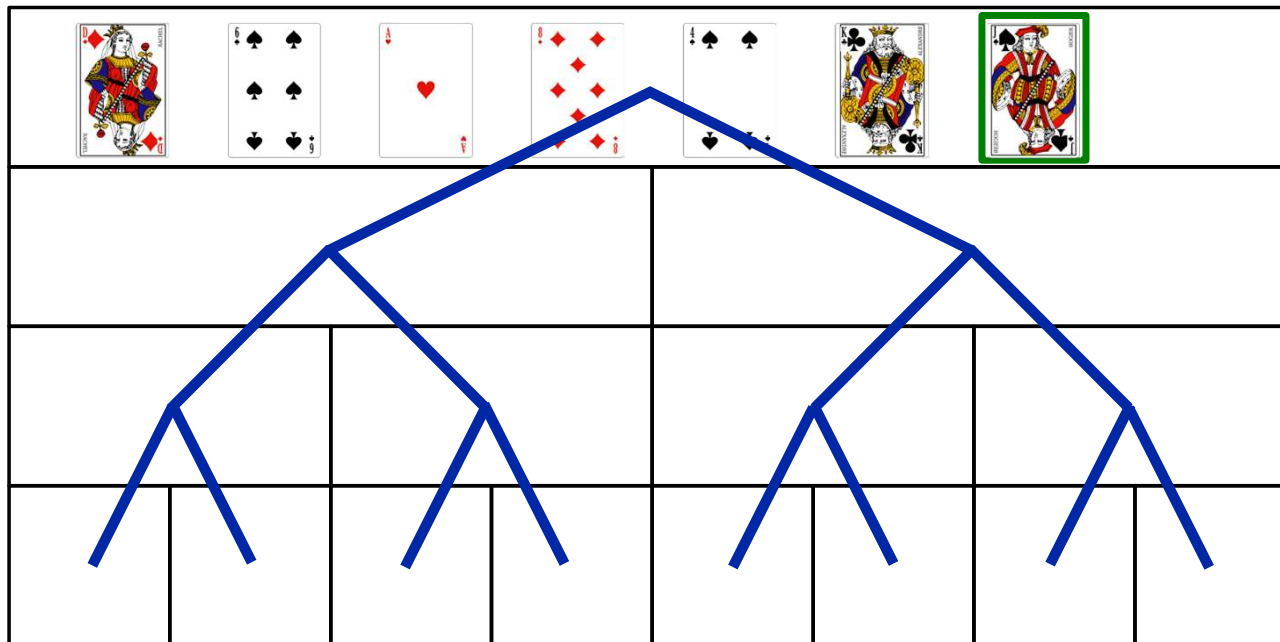
**False**, complexity will be  $O(n)$  if the tree is degenerate.



## **Binary Search Trees: Comprehension Question**

How could a binary search tree be used to sort an array of integers? What will be the asymptotic complexity of this method?

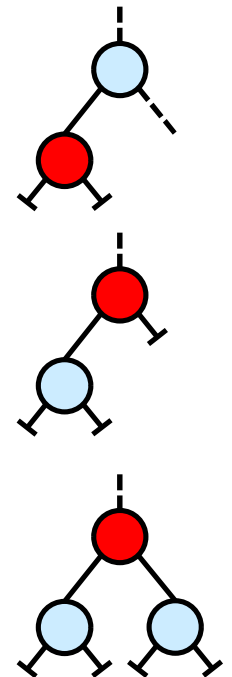
## Another Look at Quicksort



## Binary Search Trees: Removing a Node

Three different cases are to be discerned:

- **A:** Node to be removed is a **leaf** (external node): **just delete** the leaf (the appropriate node of the parent is set to null and the node is deallocated)
- **B:** Node to be removed is **not a leaf** (internal node)
  - **B1:** Node to be removed has only **one child**: The node's **parent can «adopt» the child**; the node itself is then deallocated
  - **B2:** Node to be removed has **two children**: find maximum in left subtree and replace the root of the subtree with this value, then remove the maximum node of the subtree; since the maximum in the subtree can be a leaf node or an internal node of the subtree, therefore the removal has to be done using the techniques from cases A or B1 respectively (note that the maximum/minimum node cannot have two children). Also note that this can also be done by taking the minimum in the right subtree.



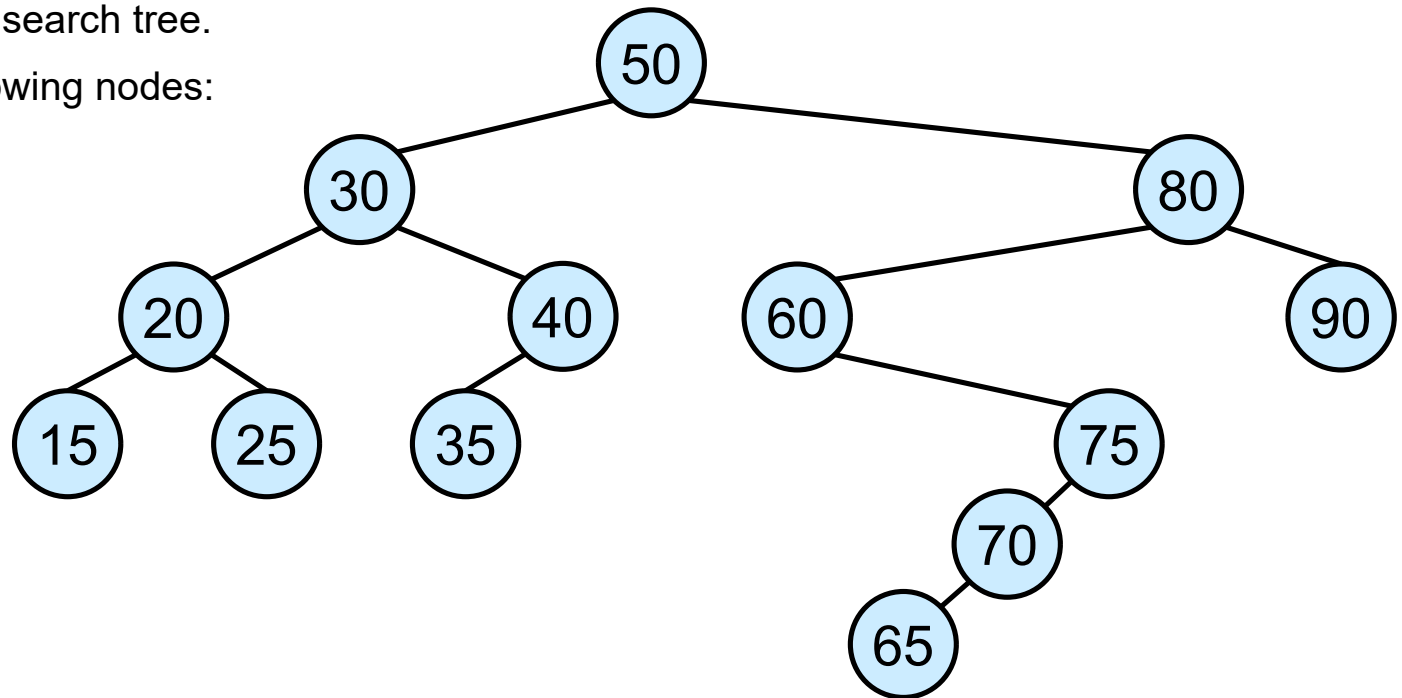


## Binary Search Tree: Node Deletion Example

Consider the adjacent binary search tree.

Consecutively delete the following nodes:

35, 70, 80, 50





## Additional Exercise Binary Search Trees: Asymptotic Complexity of Operations

Compose a table with the best case, average case and worst case asymptotic time complexities of the following operations applied to a binary search tree:

- insert a new node
- delete a node by its value
- search a node by its value



## Additional Exercise: Binary Search Trees Traversal

Draw the binary search tree which will have a postorder traversal sequence as follows:

9, 5, 12, 14, 19, 18, 27, 21, 20, 10

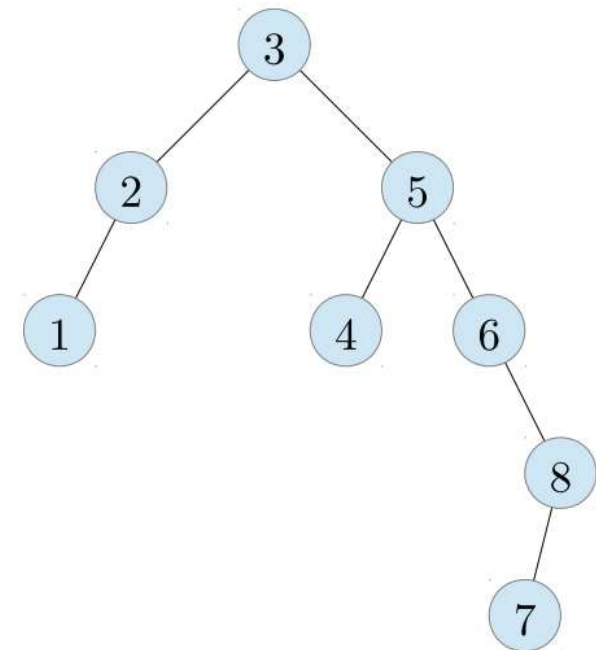


## Binary Search Trees / Tree Traversals: Additional Exercise

Draw a binary search tree which contains all integers from 1 to 8 and

- whose preorder traversal sequence starts with 3, 2, 1, 5, 4, and
- whose postorder traversal sequence ends with 7, 8, 6, 5, 3.

Is there more than one solution?

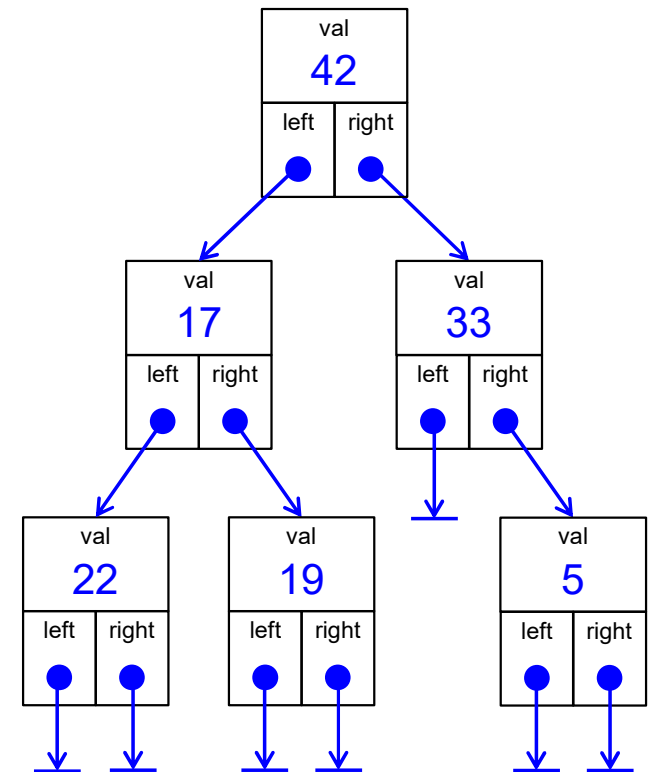


## Binary Trees: Implementation

The nodes of a binary tree can be implemented in C using a struct as follows:

```
struct TreeNode {  
    int val;  
    struct TreeNode* left;  
    struct TreeNode* right;  
};
```

Optionally, a parent pointer may be added which allows to get the parent node of a node directly. This has advantages in some situations but can also have disadvantages in others (for example, this makes removal operations more difficult because there are more pointers which have to be changed correctly; also this needs more memory, obviously).





## Exercise 8, Task 2: BST Implementation

Write a C program that contains the following functions:

- (a) `struct TreeNode* insert(struct TreeNode* root, int val)`
- (b) `struct TreeNode* search(struct TreeNode* root, int val)`
- (c) `struct TreeNode* delete(struct TreeNode* root, int val)`
- (d) `void printTree(struct TreeNode* root)`



## Exercise 8, Task 2a: BST Insertion

```
struct TreeNode* insert(struct TreeNode* root, int val) {  
    struct TreeNode* newTreeNode = NULL;  
    struct TreeNode* current = root;  
    if (root == NULL) {  
        newTreeNode = malloc(sizeof(struct TreeNode));  
        newTreeNode->val = val;  
        newTreeNode->left = NULL;  
        newTreeNode->right = NULL;  
        return newTreeNode;  
    }  
    if (val > root->val) {  
        root->right = insert(root->right, val);  
    } else {  
        root->left = insert(root->left, val);  
    }  
    return root;  
}
```



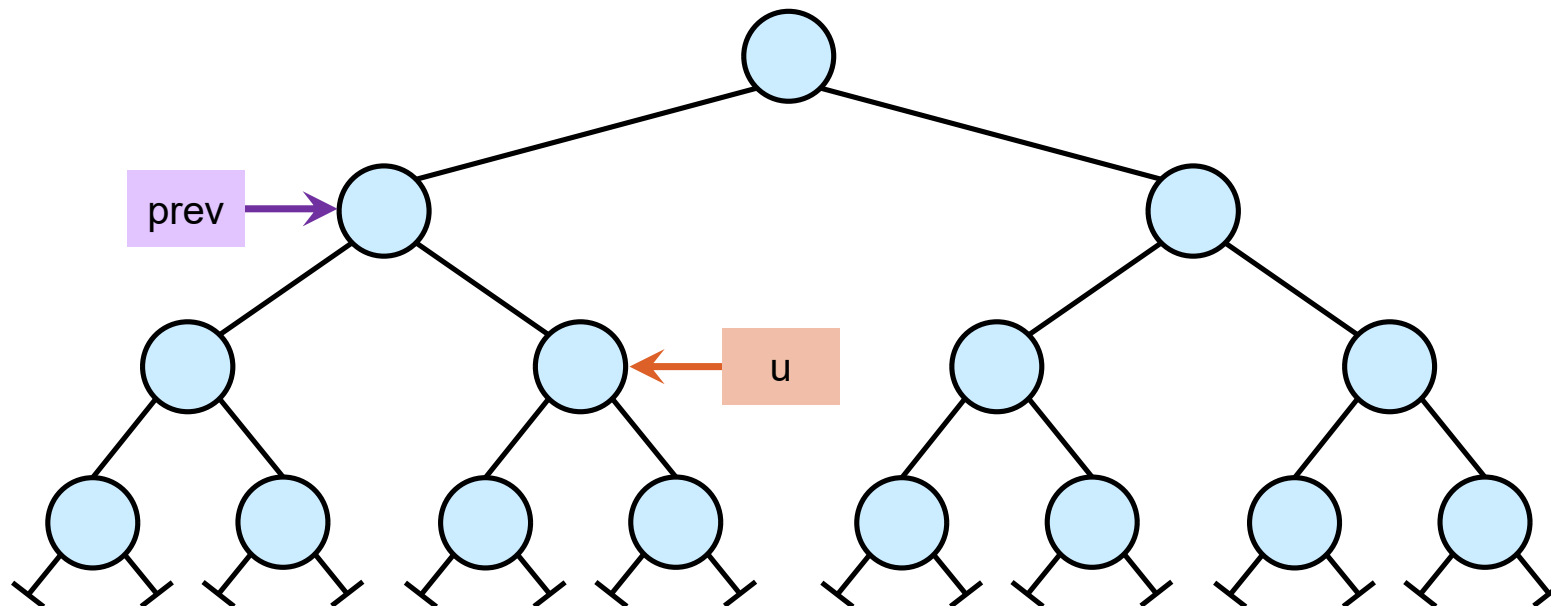
## Exercise 8, Task 2b: BST Search

```
struct TreeNode* search(struct TreeNode* root, int val) {  
    struct TreeNode* current = root;  
    while (current != NULL && current->val != val) {  
        if (val < current->val) {  
            current = current->left;  
        } else {  
            current = current->right;  
        }  
    }  
    return current;  
}
```



## Exercise 8, Task 2c: Implementation of Deletion Operation

Idea: «Laggy pointer»: Maintain two pointers **u** and **prev** which point to nodes of the tree such that **prev** always points to the parent of **u** (except in the very beginning, when **u** points to the root and **prev** to NULL). Advance both these pointers until **u** points to the node which shall be deleted.





## Exercise 8, Task 3: Balancing a Binary Search Tree

**Task 3. Balance Binary Search Tree** The time complexity for searching a key is  $O(\log n)$  for a balanced binary search tree. Hence, it is useful if we can transform a binary search tree to a balanced binary search tree. Write a function that converts binary search tree to a balanced binary search tree, with the following sub-tasks:

**Algo:** InorderTreeWalk(root)

```
arr ← [] ;  
if  $p \neq \text{NIL}$  then  
    InorderTreeWalk(root → left);  
    arr.push(root → val);  
    InorderTreeWalk(root → right);
```



## Exercise 8, Task 3: Balancing a Binary Search Tree

**Algo:** ConstructBSTFromArray(arr, start, end)

---

```
root ← NIL ;  
if start > end then  
    return NIL  
mid = (start + end) / 2 ;  
root ← arr[mid] ;  
root.left ← ConstructBSTFromArray(arr, start, mid - 1) ;  
root.right ← ConstructBSTFromArray(arr, mid + 1, end) ;  
return root
```

**Algo:** BalanceBST(root)

---

```
sortedArray ← InorderTreeWalk(root) ;  
balancedTree ← ConstructBSTFromArray(sortedArray, 0,  
    sortedArray.length - 1) ;
```



## Exercise 8, Task 4: Range Query by Trimming a Binary Search Tree

**Task 4. Range Query by Trimming Binary Search Tree** One important application of the binary search tree is for finding elements. Suppose you need to find all elements in the range  $[\text{low}, \text{high}]$ , write a pseudocode that returns the binary search tree with all its element lies in the range  $[\text{low}, \text{high}]$ . Write a pseudocode and C implementation for this task.

*Note:* You should not change the relative structure of the elements that will remain in the tree. For example, assume the original binary search tree is shown in Figure 1 and suppose you need to find all elements in the range  $[17, 45]$ , then the trimmed binary search tree should be shown in Figure 2.



**Universität  
Zürich<sup>UZH</sup>**

**Institut für Informatik**

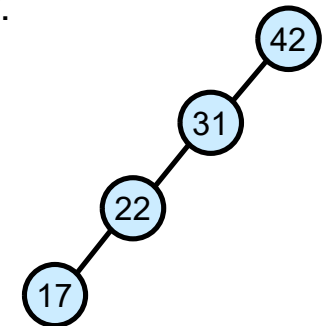
# Tree Balancing, Tree Rotations, Red-Black Trees

- Tree Balancing
- Tree Rotations
- Red-Black Trees



## Balancing Mechanisms for Binary Search Trees: Motivation

- Binary search trees may degenerate, in particular if nodes are inserted in sorted order.
- In this case, the asymptotic complexity of operations is suboptimal.
- We need a balancing mechanism to prevent this from happening.
- Examples for trees with such a mechanism: AVL trees, red-black trees, 2-3 trees, 2-3-4 trees, splay trees, B-trees, B<sup>+</sup>-trees, scapegoat trees, ...





## Red-Black Trees: General Remarks

- Red-black trees (RB trees) are a special kind of **binary search tree**.
- Every **node** is either **red or black**. (Typical implementation: Each node has an extra bit to save this.)
- All **leaves are black** and have **no values** attached to them; they are referred to as (black) «**nil pointers**» (this is also called the external node property).
- A **balancing mechanism prevents** the tree from **degenerating**.
  - Perfect balancing is neither aimed for nor (usually) achieved, though.
  - The longest path (root to deepest nil pointer) is no longer than twice the length of the shortest path (root to nearest nil pointer). The shortest path has all black nodes. The longest path exhibits an alternating sequence of red and black nodes.

### Remarks:

- Searching in a red-black tree is exactly the same as in an elementary binary search tree (just ignore colour). Read-only operations are also identical (but they benefit from the fact that red-black trees are much better balanced).
- There is a 1:1 correspondence between red-black trees and 2-3 trees.



## Red-Black Trees: Conditions

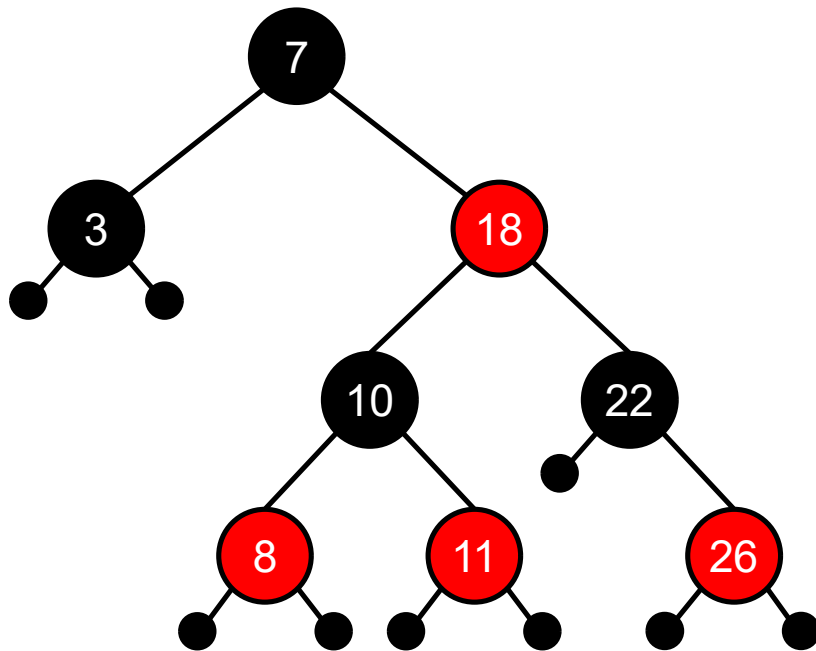
In a valid red-black tree, the following conditions must hold (additionally to BST, nodes red/black, external node property):

- **Root property:** The root node is always black.
- **Red property:** If a node is red, both its children are black. (Every red node has a black parent. On every path in the tree, there cannot be two red nodes consecutive.)
- **Depth property:** All leaves (black nil pointers) have the same black depth: For each node, all paths from the node to descendant leaves contain the same number of black nodes (count includes black nil pointers but not the root).

The latter two properties make sure that the tree remains balanced.



## Red-Black Trees: Example

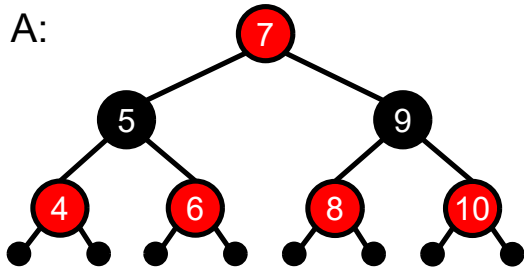


### *Check of RB conditions:*

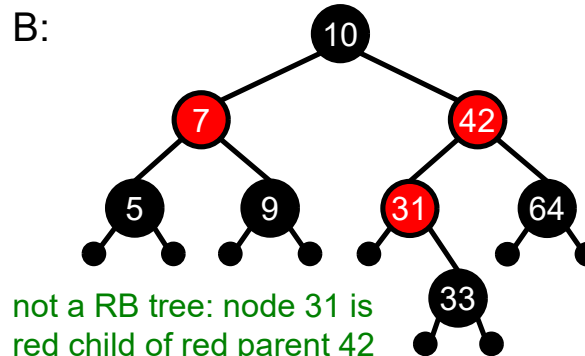
- It is a BST.
- All nodes are either red or black.
- All leaves are black and have no value (are black nil pointers).
- The root is black.
- The red node 18 has only black children, the same applies for 8, 11, 26.
- The black depth from 7 is always 2. The Black depth from 18 is always 2. The Black depth from all other nodes is always 1.

## Red-Black Trees: Conditions Exercise

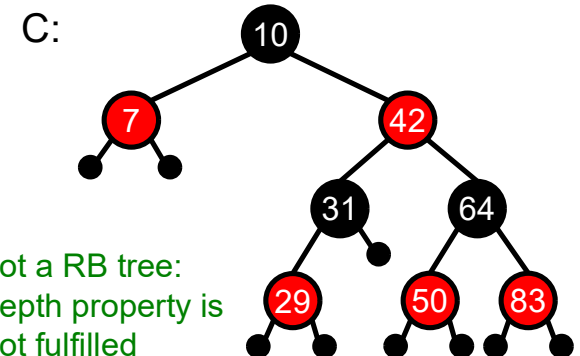
Are the following trees valid red-black trees?



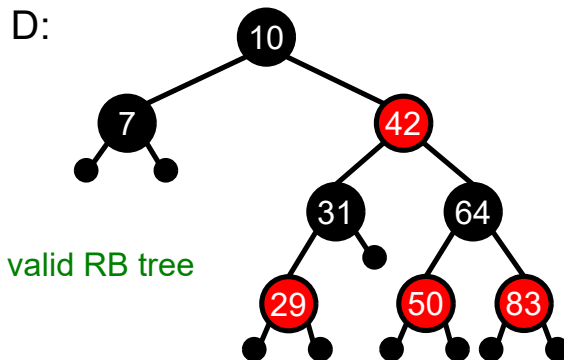
not a RB tree: root is red  
(could be fixed by coloring root black)



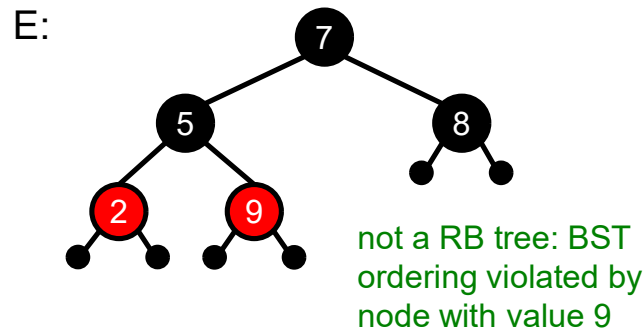
not a RB tree: node 31 is  
red child of red parent 42



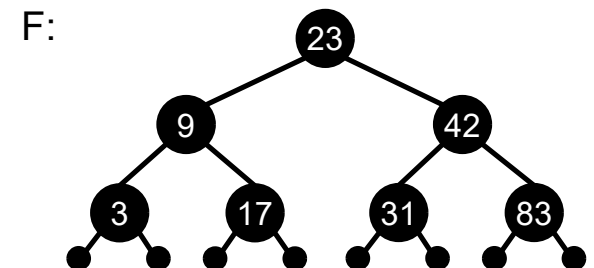
not a RB tree:  
depth property is  
not fulfilled



valid RB tree



not a RB tree: BST  
ordering violated by  
node with value 9



valid RB tree



## Tree Rotations: Introduction

A tree rotation is a way of **transforming an ordered tree** (e.g. binary search tree) into another ordered tree, thus an operation **preserving the sorted property**. Thus, these operations will change the structure of a binary search tree without affecting the order of its nodes / its inorder sequence.

Rotations are **used to rebalance** an ordered tree (e.g. red-black trees, but not only them).



## Tree Rotations: Introduction

There are two types of such rotations:

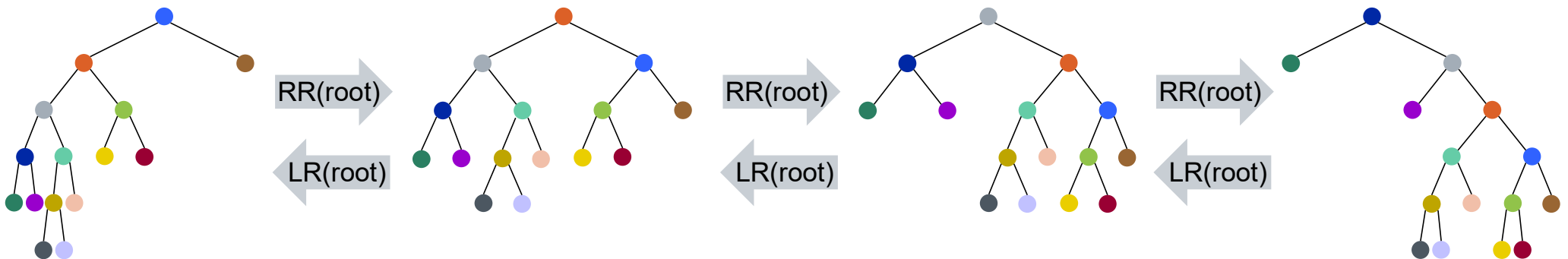
- left rotations (LR)
- right rotations (RR)

Rotations are applied on a certain node, i.e.  $LR(N)$ . A left rotation on node  $N$  will shift nodes from the right subtree of  $N$  to the left subtree of  $N$ , thus.

Left rotations and right rotations will cancel out each other (they are inverse operations).

## Tree Rotations: Effect Example

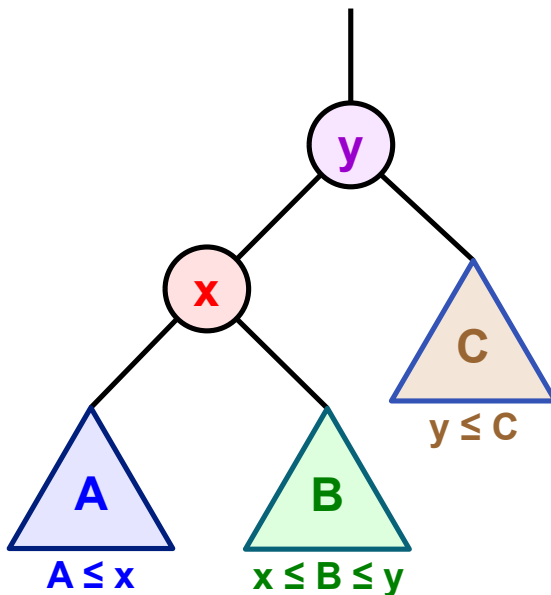
*Example:* Right rotations applied successively on the root of an initially left-heavy binary tree.  
(Opposite direction: left rotations on the root of an initially right-heavy tree.)



Note that the root is changed by each tree rotation applied on the (current) root.

## Tree Rotations: Preserving Sorted Property / Invariant

Consider the following binary search tree:



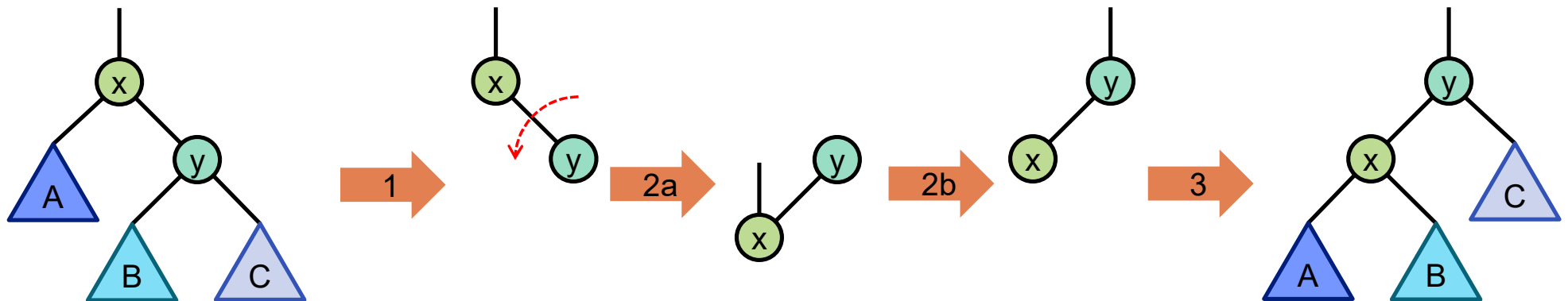
Note that the following conditions must hold (invariant of operation):

- all nodes in subtree A must be smaller than or equal to the value of x
- all nodes in subtree B must be bigger than or equal to x and smaller than or equal to y
- all nodes in subtree C must be bigger than y

nodes in A ≤ x ≤ nodes in B ≤ y ≤ nodes in C

## Tree Rotations: Left Rotation (LR)

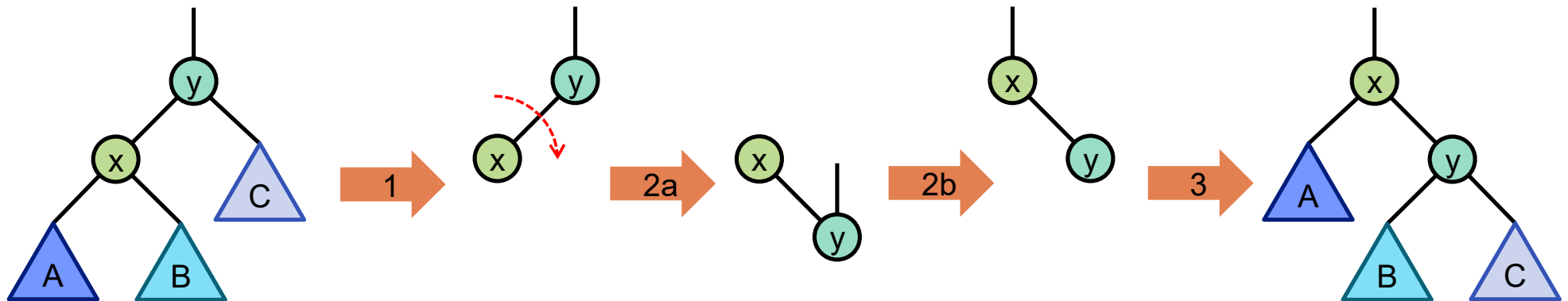
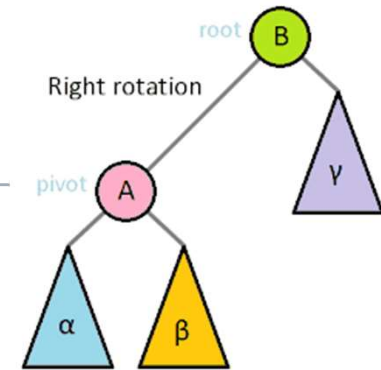
- 1) From the node on which the rotation is performed ( $x$ ), regard the *right* child ( $y$ ). Cut off all subtrees ( $A$ ,  $B$ ,  $C$ ) from the nodes involved ( $x$ ,  $y$ ).
- 2) a) Rotate the nodes  $x$  and  $y$  to the right around their common edge.  
b) Make the parent of  $x$  the parent of  $y$ . (This step can also be done in the end after step 3.)
- 3) Reattach the subtrees  $A$ ,  $B$ ,  $C$  according to the sorting condition as stated before:  
nodes in  $A \leq x \leq \text{nodes in } B \leq y \leq \text{nodes in } C$



## Tree Rotations: Right Rotation (RR)

A right rotation is simply the **opposite operation of a left rotation**.

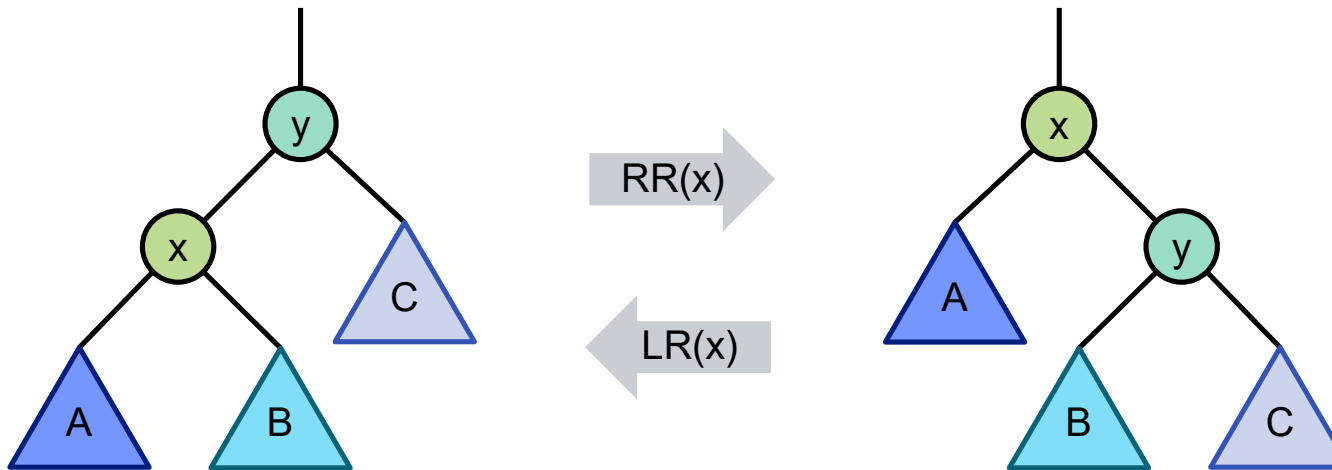
Note that in this case, the left child of the node on which the operation is performed is taken into consideration when determining the axis which is rotated.





## Tree Rotations: Check Invariant

An easy way to see (and check) that tree rotations will preserve the sorted condition of a BST, is to [look at](#) the results of an [inorder traversal](#) of the tree before and after the rotation:



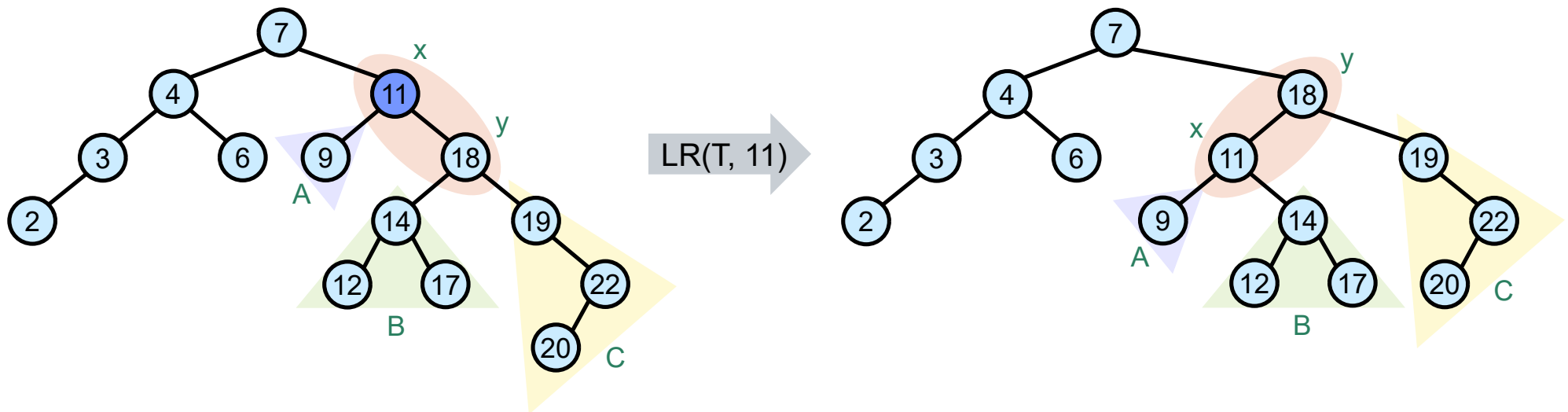
Inorder  
sequence:

$A x B y C$

$A x B y C$

## Tree Rotations: Example / Exercise

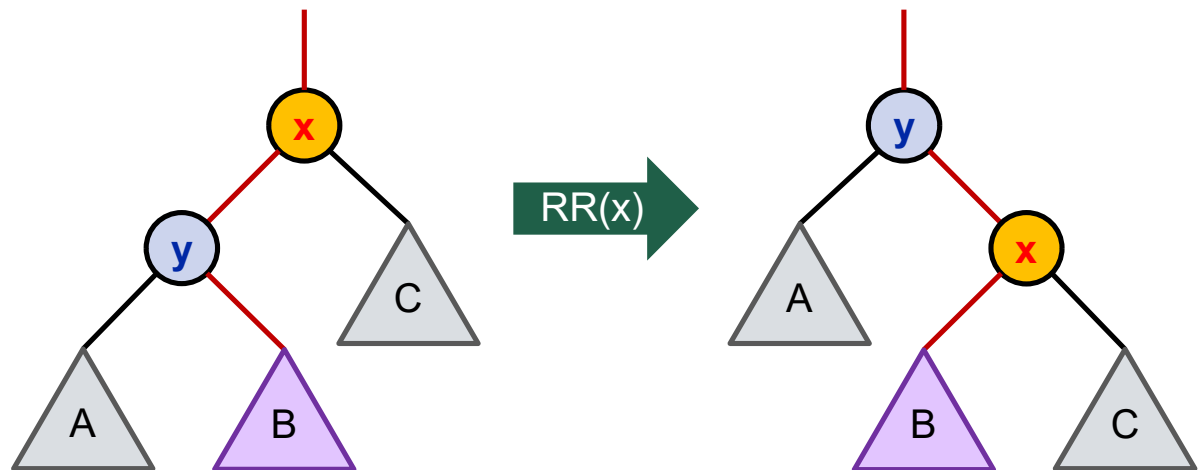
Consider the following binary search tree  $T$ . Perform a left rotation on the node with value 11.



## Right Rotation Implementation – Without Parent Pointers

*Pseudocode for right rotation:*

```
rightRotate(x) {
    y = x->left;
    x->left = y->right;
    y->right = x;
}
```



Note that the subtrees A and C remain unchanged and attached to their previous parents.

The above pseudocode only shows the basic idea and does not yet consider special cases. For example, if the node on which the rotation shall be performed (x) has no left child (i.e.  $y = x \rightarrow \text{left}$  does not exist), the rotation has no effect / cannot be done.

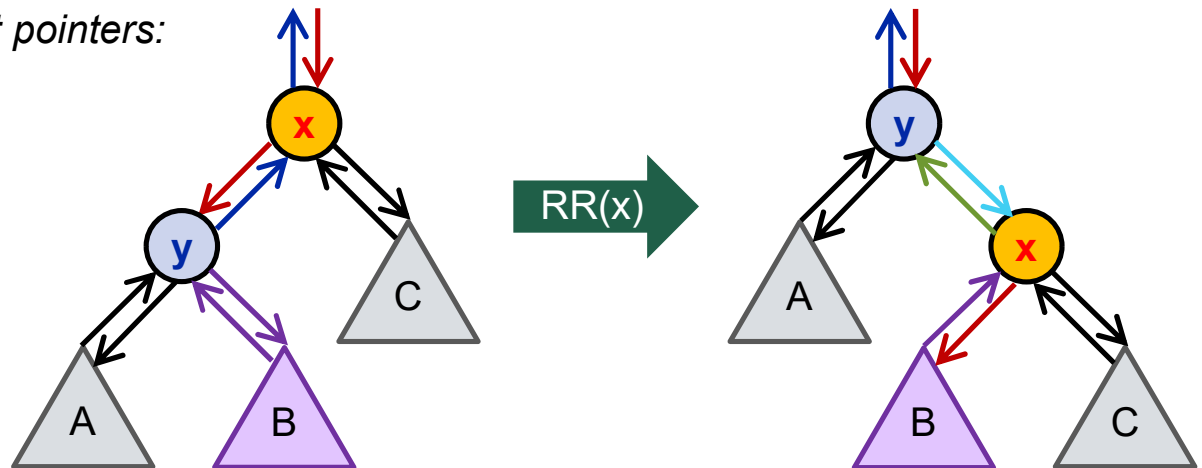
Also note that the asymptotic time complexity of tree rotations is  $O(1)$ .

## Right Rotation Implementation – With Parent Pointers

*Pseudocode for right rotation with parent pointers:*

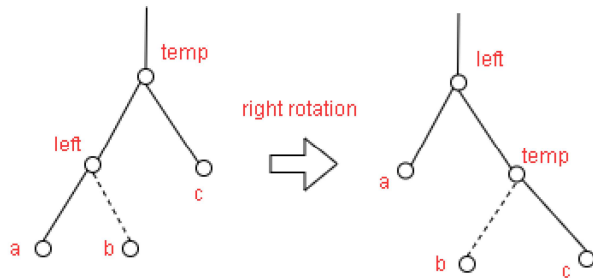
```

rightRotate(T, x) {
    y = x->left;
    x->left = y->right;
    y->parent = x->parent;
    if (y->right != NIL) {
        y->right->parent = x;
    }
    if (x->parent == NIL) {
        T.root = y;
    } else {
        if (x == x->parent->right) {
            x->parent->right = y;
        } else {
            x->parent->left = y;
        }
    }
    y->right = x;
    x->parent = y;
}
    
```



## Exercise 9, Task 4

1. Complete the code segment of `RightRotation(temp)` according to the diagram and instructions. Assume that the left child of node `temp` exists.



2. State what lines 16-21 have done.

Since node `temp` is replaced by node `left`, these several lines make node `left` the right or left child as node `temp` was.

```

1 struct node {
2     struct node* p; // parent
3     struct node* r; // right child
4     struct node* l; // left child
5 };
6
7
8 void rightrotate(struct node* temp){
9     struct node * g = temp->p;
10    struct node* left = temp->l;
11    (a)_____ // 'b' becomes left child of 'temp'
12    if(left->r) {
13        (b)_____ // 'temp' becomes parent of 'b'
14    }
15    (c)_____ // 'left' become the child of g
16    if (temp == g->l){
17        g->l = left;
18    }
19    else {
20        g->r = left;
21    }
22    (d)_____ // 'temp' becomes right child of 'left'
23    (e)_____ // 'left' becomes parent of 'temp'
24 }
```

a. `temp->l = left->r;`  
b. `left->r->p = temp;`  
c. `left->p = g;`  
d. `left->r = temp;`  
e. `temp->p = left;`

## Red-Black Trees: Insertion

Insertion works the same way as inserting into a binary search tree at first (i.e. smaller values go to the left, bigger values go the right).

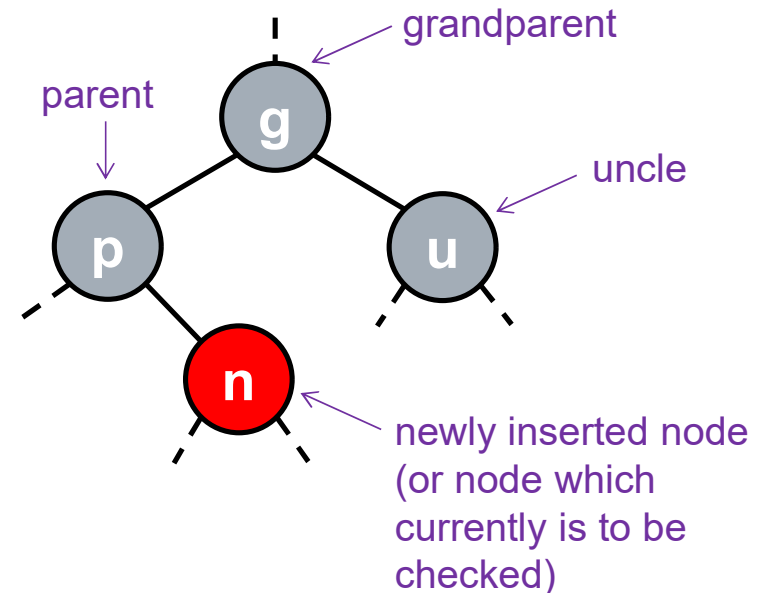
When a node is inserted, it is always **colored red initially**.

Next, the RB conditions are checked. If the tree which arose this way violates the RB conditions, one or both of the following operations is applied to restore the RB conditions:

- **node recolorings**
- **tree rotations**

Note that the new node can have children in general, because it is possible that changes propagate through the tree.

Also note that rotations do not change the black height of any node. (After a rotation, all subtrees are exactly the same relative to the root as they were before the rotation.)



## Red-Black Trees: Insertion Cases

There are **four different cases** which can occur:

- |              |              |   |  |
|--------------|--------------|---|--|
| new node has | black parent | { | <ul style="list-style-type: none"><li>– <b>Case 0: Black parent:</b> Parent of new node is black.<br/>→ Conditions cannot be violated if the tree was a valid RB tree before; <b>nothing to do.</b></li></ul>  |
|              | red parent   |   | { <ul style="list-style-type: none"><li>– <b>Case 1: Red uncle:</b> Red property violated by the new node and the uncle of the new node is red.<br/>→ Recolor the parent and uncle in black, recolor the grandparent in red; propagate upwards through the tree (in the end change root to black if necessary)</li><li>– <b>Case 2: Black uncle, triangle:</b> Red property violated and the uncle of the new node is black; grandparent, parent and new node form a triangle.<br/>→ Transform the tree to a case 3 configuration by rotating the parent and considering it as the new node.</li><li>– <b>Case 3: Black uncle, line:</b> Red property violated and the uncle of the new node is black; grandparent, parent and new node form a straight line.<br/>→ Recolor the parent black and the grandparent red, then rotate the grandparent.</li></ul> |



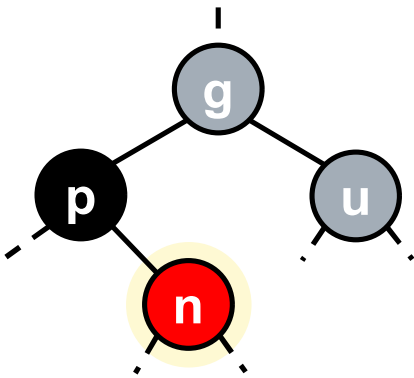
## Red-Black Trees: Insertion Cases: Remarks

- It could be argued that there is one additional case («[case E](#)»), namely the situation where the tree is empty before inserting the new node. In this case the new node can just be inserted as a black node as an exception (or it is inserted as a red node as usual and a check is performed in the end, whether the root is black and recolor it if necessary since this is always allowed).
- Regarding cases 2 and 3, take in mind that the uncle can also be a nil pointer – which is a black node, too.
- The cases 2 and 3 each consist of two equivalent mirror cases.



## Red-Black Trees: Insertion, Case 0 (Black Parent)

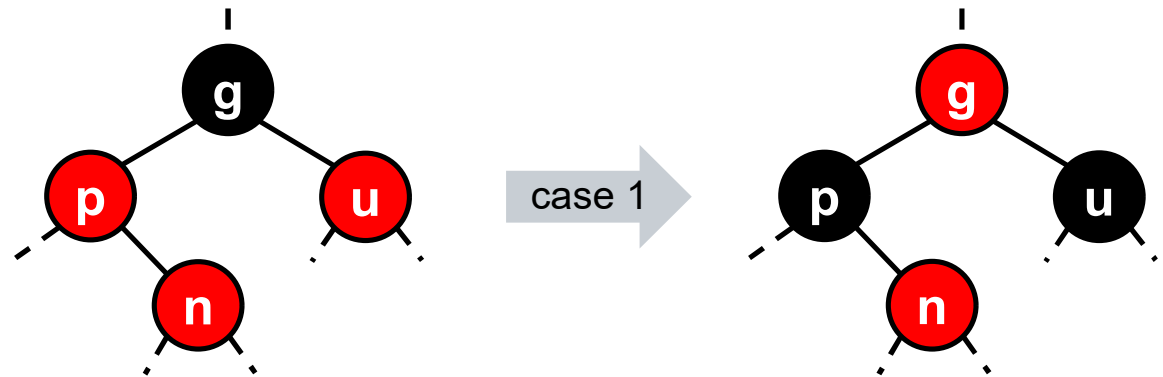
If the parent of the new node is black and the tree was a valid RB tree before insertion, no condition can be violated and there is nothing to do. (In this case, it doesn't matter what colors the grandparent and the uncle have and hence are shown in grey here.)



## Red-Black Trees: Insertion, Case 1 (Red Uncle)

If the parent of the newly inserted node is red, there is a violation of the red property (two consecutive red nodes). If, furthermore, the uncle of the new node is red, the tree can be restored to a valid RB tree by a set of recolorings as follows:

- 1) Recolor **parent** and **uncle** in **black**.
- 2) Recolor the **grandparent** in **red**.
- 3) If the grandparent's parent is red, there could be another violation of the red property. Therefore, this recoloring scheme could **propagate** upwards in the tree (while treating the grandparent as the newly inserted node, possibly until reaching the root).

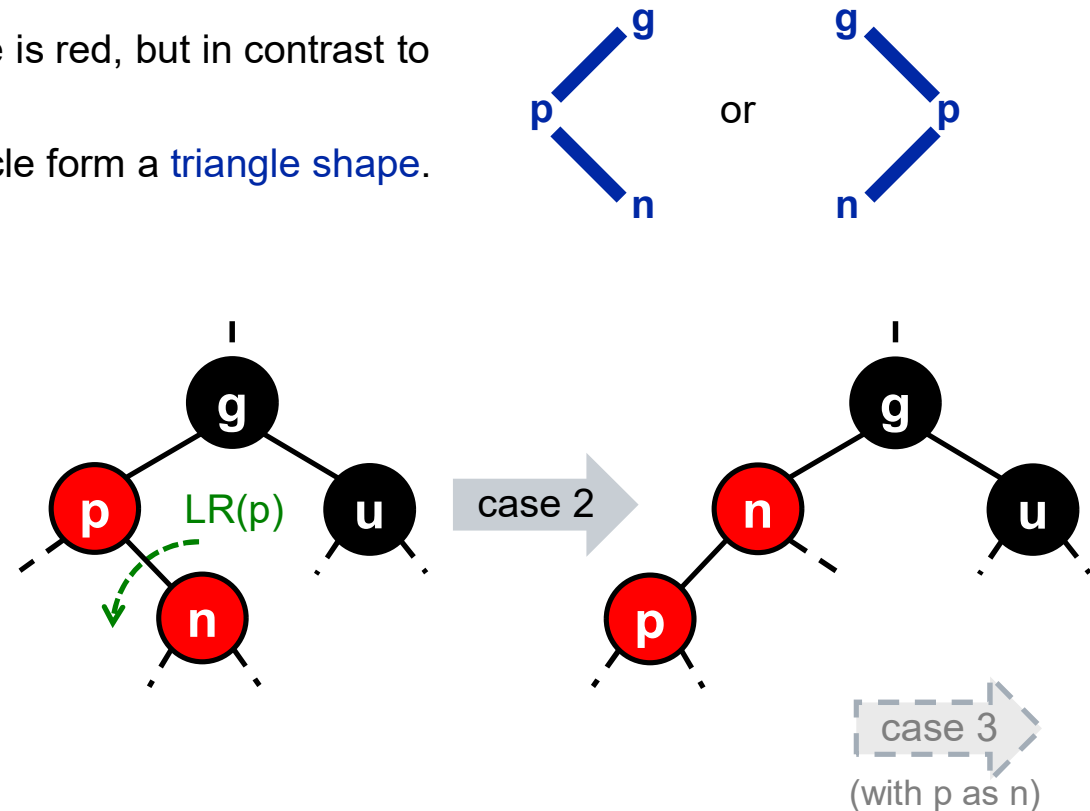


## Red-Black Trees: Insertion, Case 2 (Black Uncle): Triangle Case

In this situation, the parent of the newly inserted node is red, but in contrast to case 1, the uncle is black.

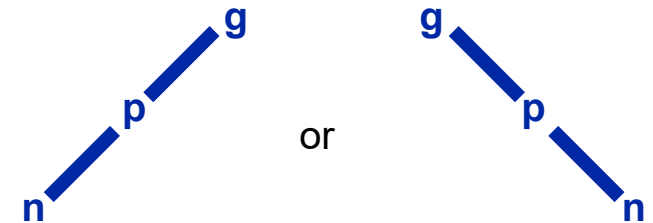
Furthermore, the grandparent, the parent and the uncle form a **triangle shape**.

- 1) Apply a **rotation on the parent** of the new node. A left rotation is applied when the triangle points to the left and a right rotation is applied when the triangle points to the right (i.e. the rotation should get grandparent, parent and new node into a line configuration).
- 2) Consider the **former parent as the newly inserted node**. (Do *not* swap them, but just shift the status of “new” to the former parent.)
- 3) The tree has now been transformed into a case 3 configuration (line case), thus continue with the **operations of case 3** from here.

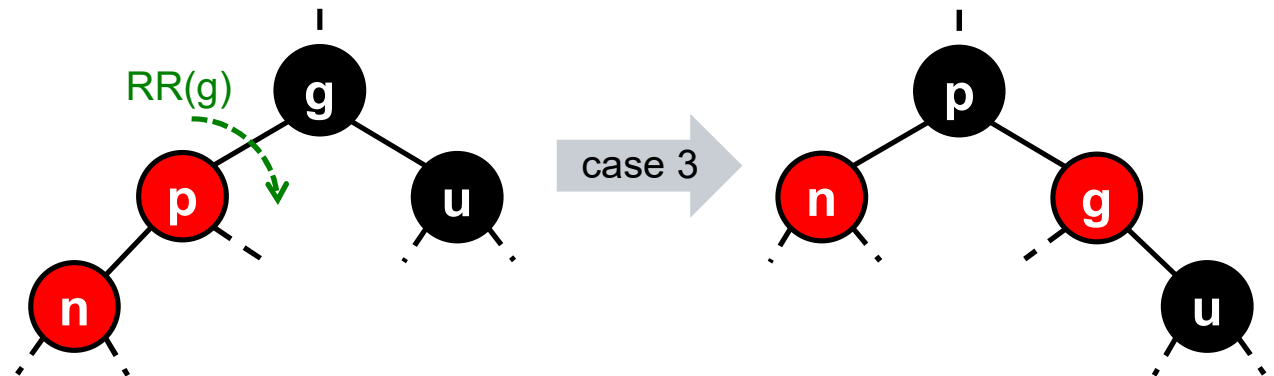


## Red-Black Trees: Insertion, Case 3 (Black Uncle): Line Case

Again, the parent of the newly inserted node is red and its uncle is black. Though, in this case, the new node, the parent and the grandparent form a **straight line**. This configuration can be the result of a previous case 2 transformation.

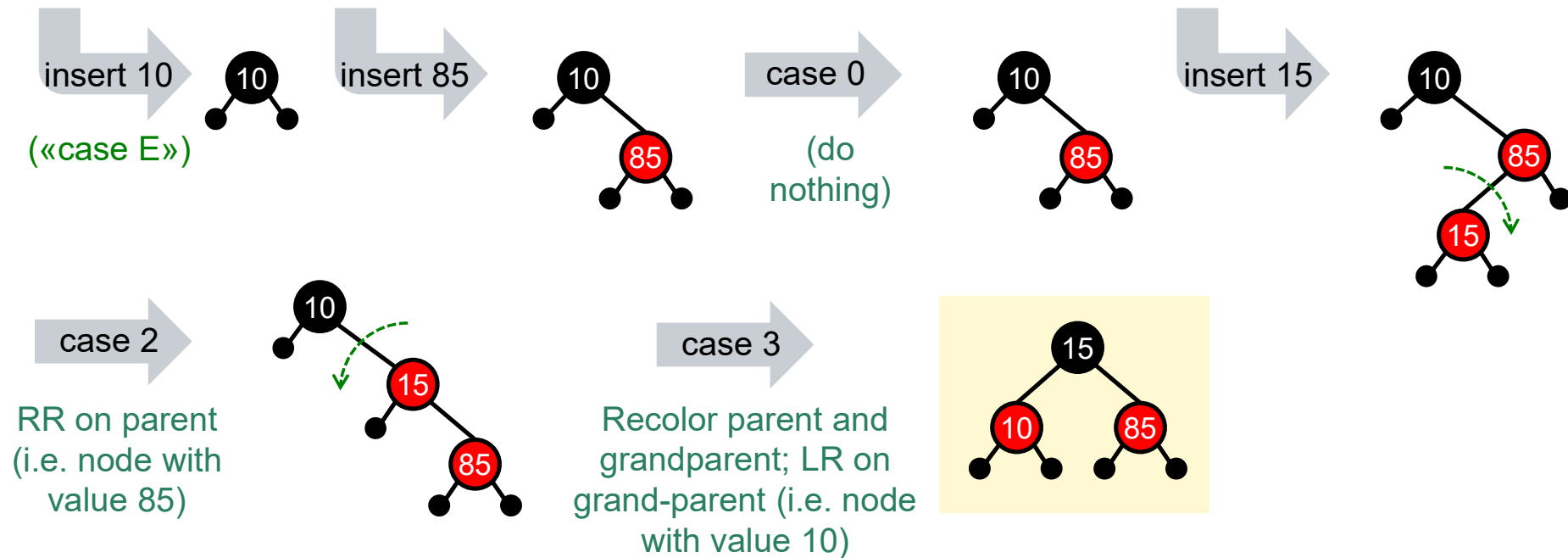


- 1) Recolor the **parent black** and recolor the **grandparent red**.
- 2) Apply a **rotation on the grandparent** of the new node towards the opposite site the tree is currently leaning. A right rotation is applied if the new node is a left child and a left rotation is applied when the new node is a right child of its parent.

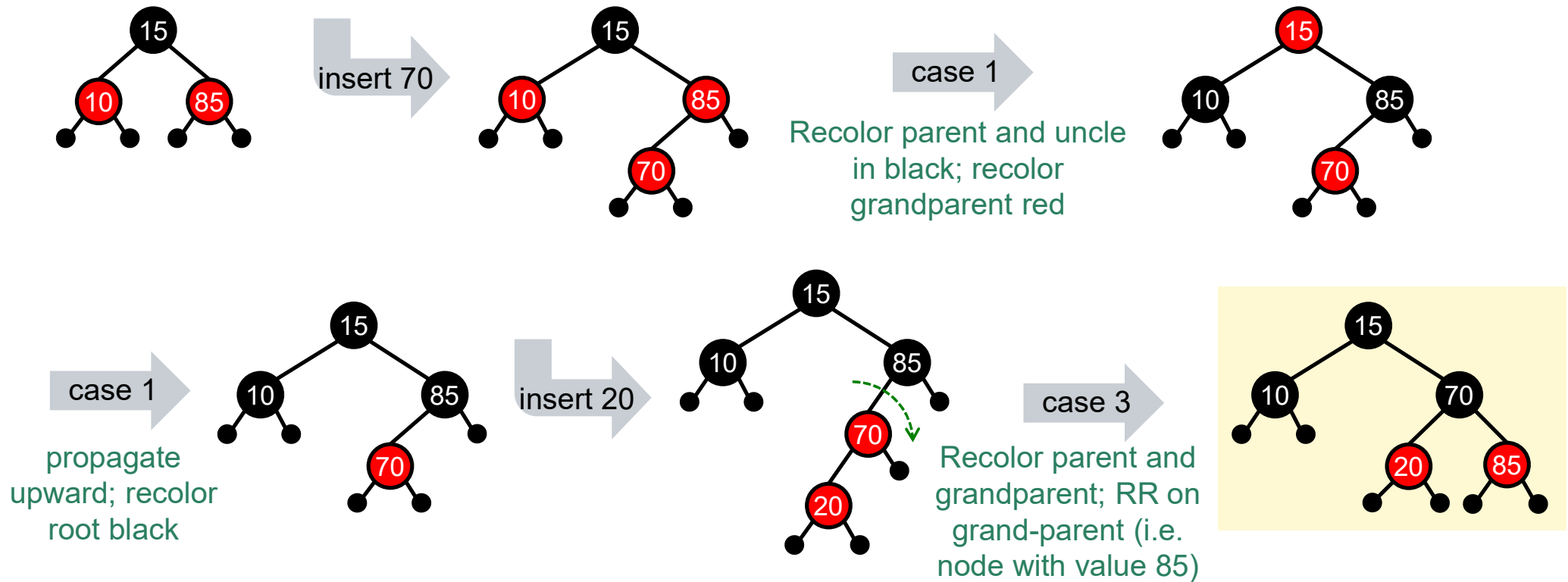


## Red-Black Trees: Insertion Example / Exercise

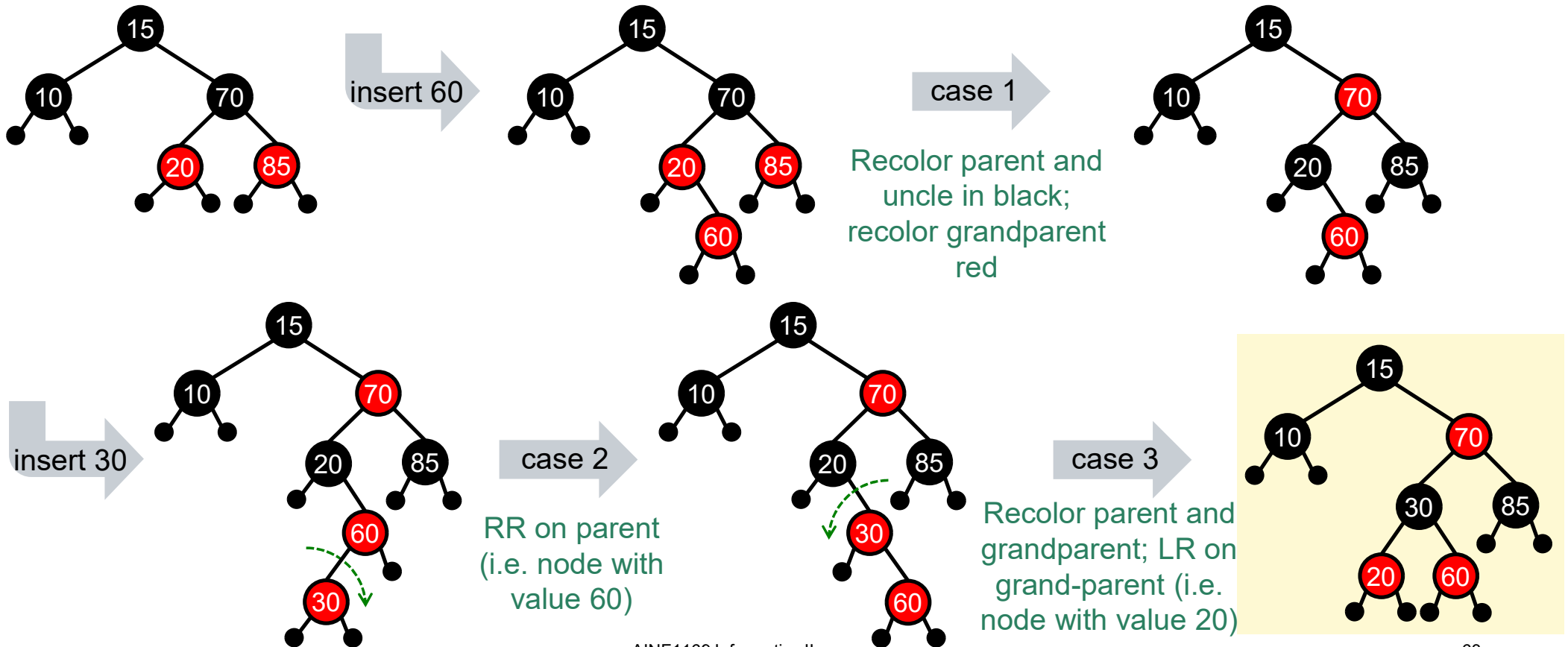
The values **10 85 15 ; 70 20 ; 60 30 ; 50 ;** are inserted in the given order into an empty red-black tree. Draw the red-black tree at the positions marked by a semicolon. (*Task 2.3 of Midterm 2 from FS 2014*)



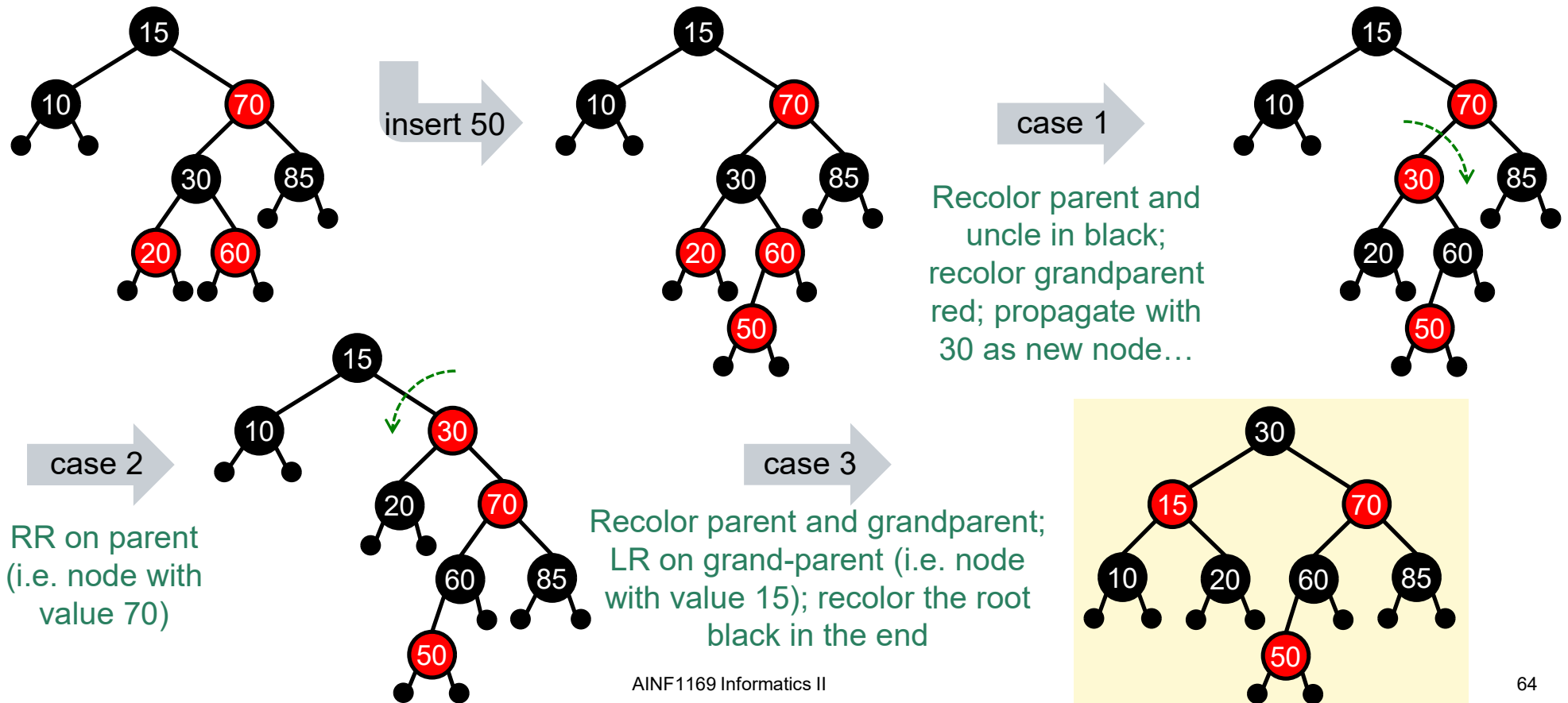
## Red-Black Trees: Insertion Example / Exercise



## Red-Black Trees: Insertion Example / Exercise



## Red-Black Trees: Insertion Example / Exercise







## Red-Black Trees: Deletions

Deleting a node in a red-black tree is done in a **two-stage operation**:

- ① **Preparing the node to be removed**: If the node to be deleted has two (non-nil) children, it is **overwritten** with the **value of the next smaller node** in the tree **without changing colors or structure** in the tree and the node from where this copy has been taken is considered to be the node to be deleted.
- ② **Restoring the RB conditions**: The deletion may have broken the black-depth property (the red property cannot be broken because of the way step ① is performed). The depth property is restored by one of five rules which may involve recolorings and rotations and which might be applied recursively.
  - While **insertions lead to a violation of the red property** (because inserting of the new node as red results in two consecutive red nodes) and has to be restored, the **depth property is violated by deletions** (because deleting of a black node results in a change of the black height in the respective subtree).
  - While for insertions, the color of the uncle is checked to decide which case has to be applied, for deletions the color of the **brother / sibling is checked to decide** which **case** has to be applied.



## Red-Black Trees: Deletions: Initial Preparatory Step

- a) If the node which shall be deleted has no (non-nil) children or if it has only one (non-nil) child, i.e. **if it has at least one nil child**:
  - There is **no preparatory work** necessary and the process can be continued with step ②, i.e. restoring the RB conditions.
- b) If the node which shall be deleted has **two (non-nil) children**, i.e. if it has no nil-children:
  - **Copy**: The value stored at the node which shall be deleted is overwritten with a copy of the largest value in its left subtree. Note that only the value is copied and the color and structure of the tree remain unchanged in this stage.
  - **Reset the «flag of death»**: The node from where this copy has been taken is considered to be the node which shall be deleted from now on and continue with step ②. This node is also called the **replacement node**.

Note that by applying the above technique, the node which is marked as the node to be deleted always will have either one non-nil child or no non-nil children. Also note that this initial preparatory step applies a similar approach as the removal of a node in a «normal» binary search tree with regard to the decision structure.



## Red-Black Trees: Deletion Cases

- **Case 0: Red node or red child:** Either the node to be deleted is red or its single child is red.
    - **nothing to do** (if deleted node was red) or **easy to fix** by recoloring the child of the node to be deleted in black
  - **Case 1: Red brother:** The depth property is violated and the node to be deleted has a red brother.
    - Recolor the brother in black and the parent in red; apply a rotation on the parent; continue with case 2, 3 or 4.
  - **Case 2: All black:** The depth property is violated and the node to be deleted has a black brother which has two black children (nephews of node to be deleted).
    - Recolor the brother in red. Consider the parent to be the node to be deleted and apply the operation recursively, until the parent is not red.
  - **Case 3: Black brother, left nephew red:** The node to be deleted has a black brother whose left child (the nephew of the node to be deleted) is red.
    - Recolor the left nephew in black. Recolor the brother in red. Apply a right rotation on the brother, then continue with case 4.
  - **Case 4: Black brother, right or both nephews red:** The node to be deleted has a black brother which has at least the right child red (the color of the other child of the brother does not matter).
    - Recolor the brother in the color of the parent (may be black or red), recolor the parent and the right nephew in black, then rotate the parent.
- either the node to be deleted or its replacement is red
- both the node to be deleted and its replacement are black



## Red-Black Trees: Deletion: Case 0: Red Node or Red Parent

If either the deleted node was red or it is black and has a single red child, there is either no violation of the depth property at all or the violation can easily be fixed.

There are two possible subcases:

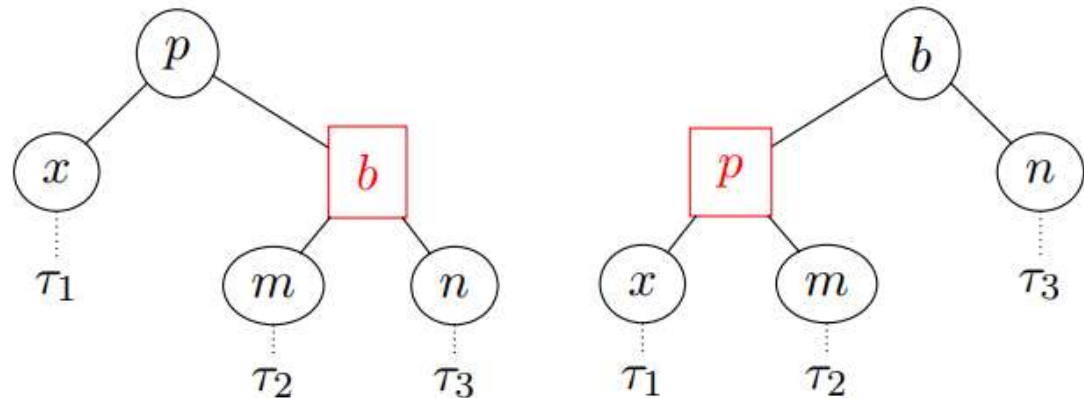
- **red node** (case 0A): the node which shall be deleted is red  
→ **nothing to fix**, since the depth property cannot be violated by removing a red tree (because we're only counting black nodes for this property)
- **red parent** (case 0B): the single child of the node which shall be deleted is red  
→ if applicable, set the color of the **child of the node which shall be deleted to black**; this makes up for the missing black node which would arise in black depth by deletion of a black node

## Red-Black Trees: Deletion: Case 1: Red Brother

If the brother is red and the node to be deleted is black, then removing the node would lead to a decreased black depth in the subtree of the deleted node.

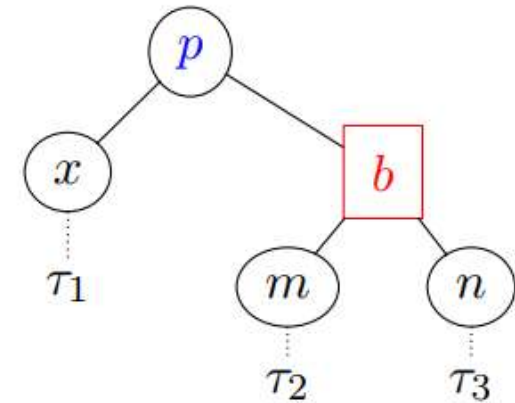
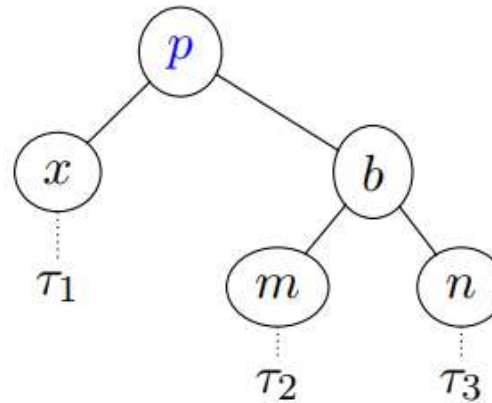
This can be fixed as follows:

- 1) Recolor the **brother** in **black**.
- 2) Recolor the **parent** in **red**.
- 3) Apply a **left rotation** on the **parent** of the node which shall be deleted.  
This converts the tree into a black brother case configuration.
- 4) Apply either case 2, 3 or 4 to fix the «double black» condition.



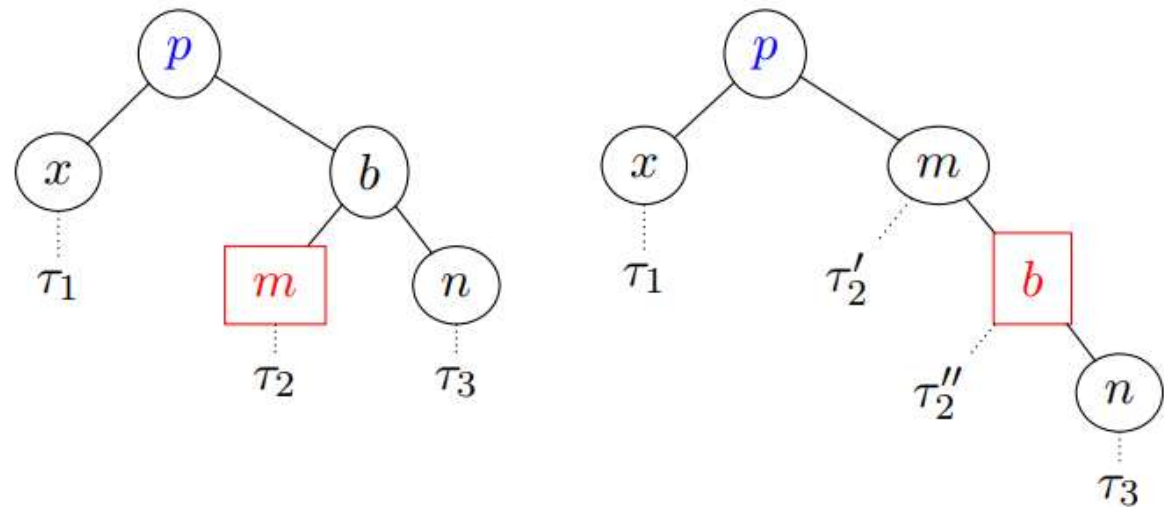
## Red-Black Trees: Deletion: Case 2: All Black

- 1) Recolor the **brother** in **red**.
- 2) Recursively call the delete procedure with the parent as the node which shall be deleted as long as the color of the parent is not red.
- 3) Set the color of the **parent** to **black** in the end.



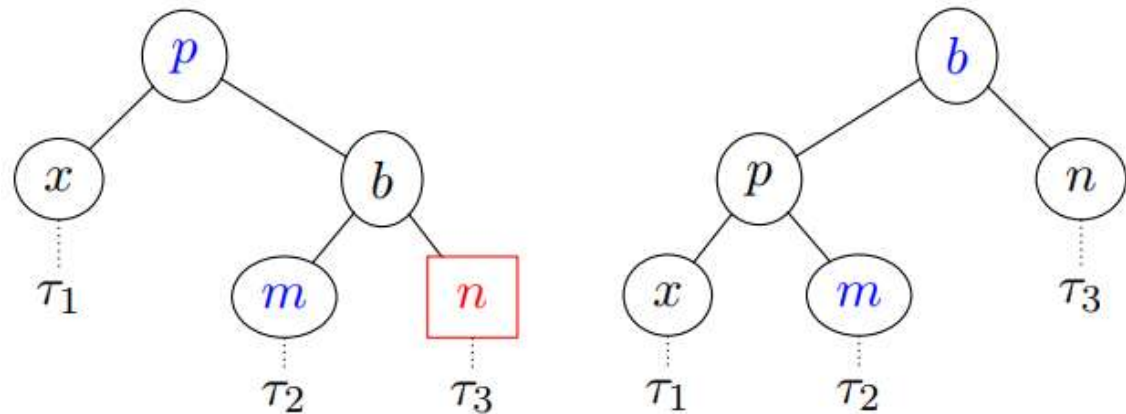
## Red-Black Trees: Deletion: Case 3: Black Brother, Left Nephew Red

- 1) Recolor the **left nephew** in **black**.
- 2) Recolor the **brother** in **red**.
- 3) Apply a **right rotation** on the **brother** of the node which shall be deleted. This converts the tree into case 4 configuration.
- 4) Continue with case 4.



## Red-Black Trees: Deletion: Case 4

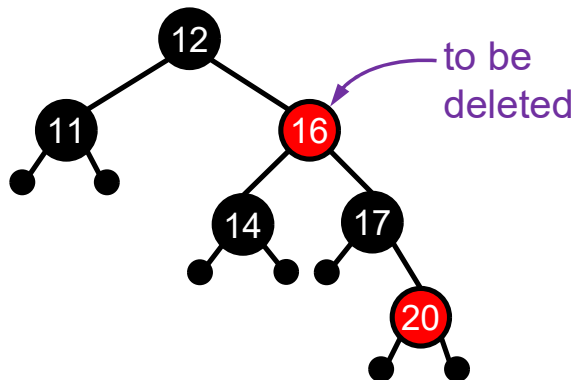
- 1) Recolor the **brother** in the same color as the **parent**.
- 2) Recolor the **parent** and the **nephew** in **black**.
- 3) Apply a **left rotation on the parent** of the node which shall be deleted.





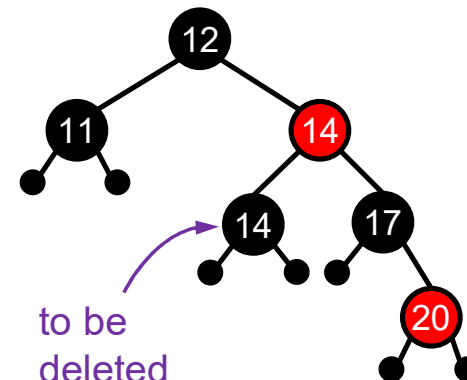
## Red-Black Trees: Deletion Exercise

Delete the node with key 16 from the red-black tree given below. Show all intermediate steps. (Task 3.1.1 of repetition exam from FS 2018)



BST delete /  
replace

- 1) Get largest value from left subtree of the node which shall be deleted.
- 2) Replace the value of the node which shall be deleted with this value.
- 3) Mark the node from which the value was copied as the new node which shall be deleted.



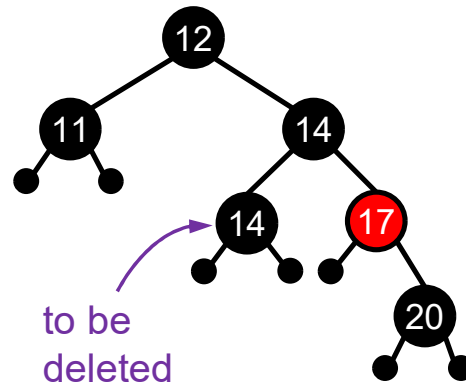
identify

Brother of node which shall be deleted after replacement step is black and has a red right child (= nephew).  
→ Case 4

## Red-Black Trees: Deletion Exercise

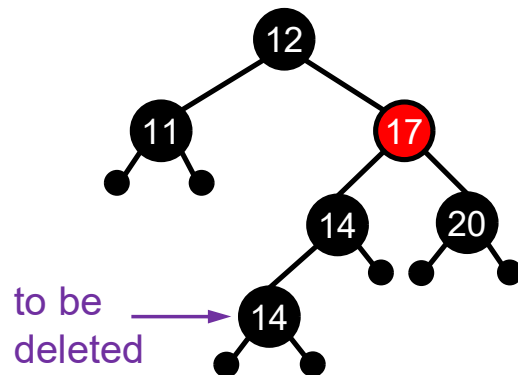
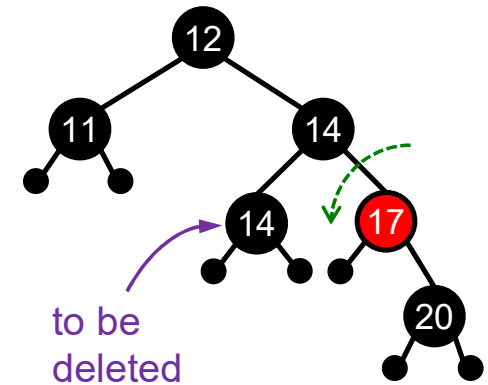
case 4  
(part 1)

- 1) Recolor the brother in the color of the parent.
- 2) Recolor the parent and the nephew in black.

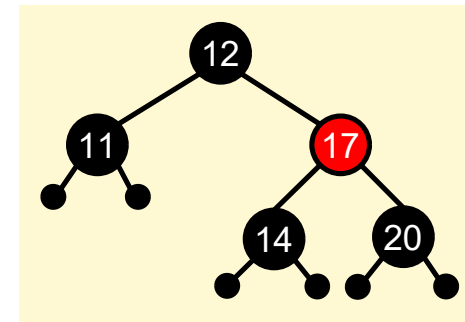


case 4  
(part 2)

Left rotation on  
parent of node  
which shall be  
deleted.

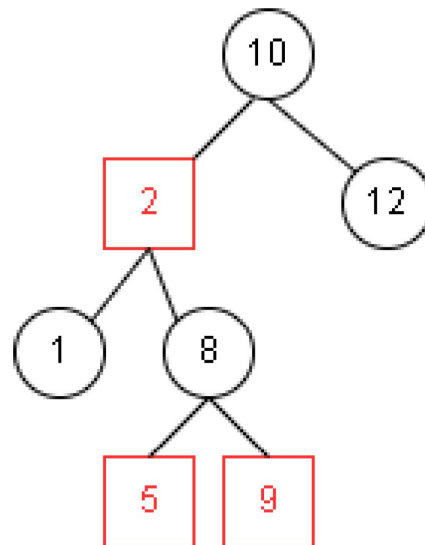


remove



## Exercise 9, Task 2

1. Consider the red-black tree shown below. State the operations that are required to insert 6 into the red-black tree.

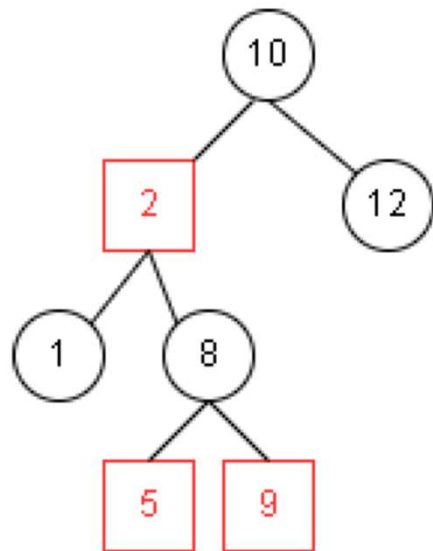


2. Show the red-black tree that results after each of the integer keys 2, 3, 6, 7, and 1 insertion into an empty red black tree step by step.



## Exercise 9, Task 3

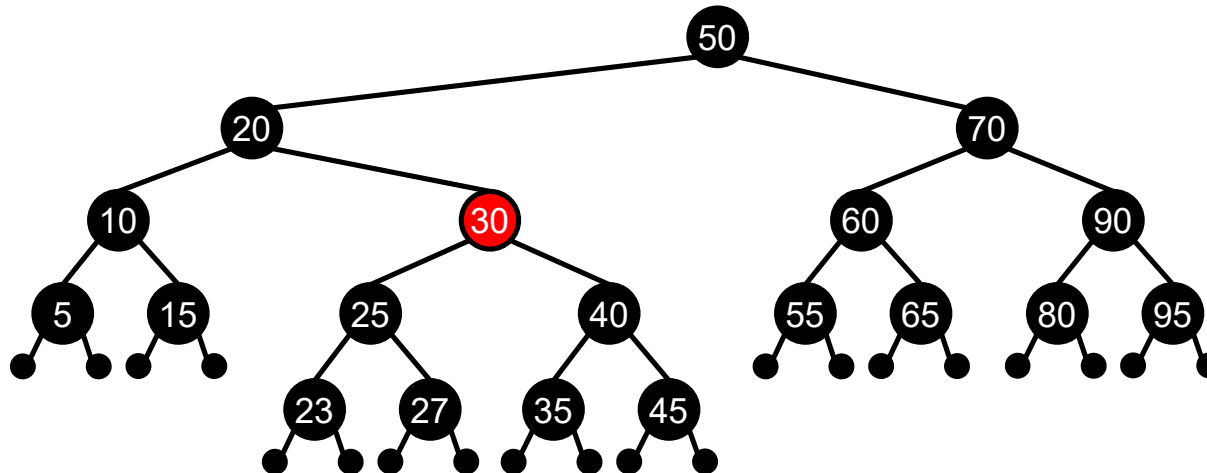
Delete 2 from the red black tree. Show state of tree after each step by drafts and explain the change.





## Additional Exercise: Red-Black Tree Deletion

Delete the node with key 20 from the red-black tree given below. Show all intermediate steps.





## Repetition on Red-Black Trees

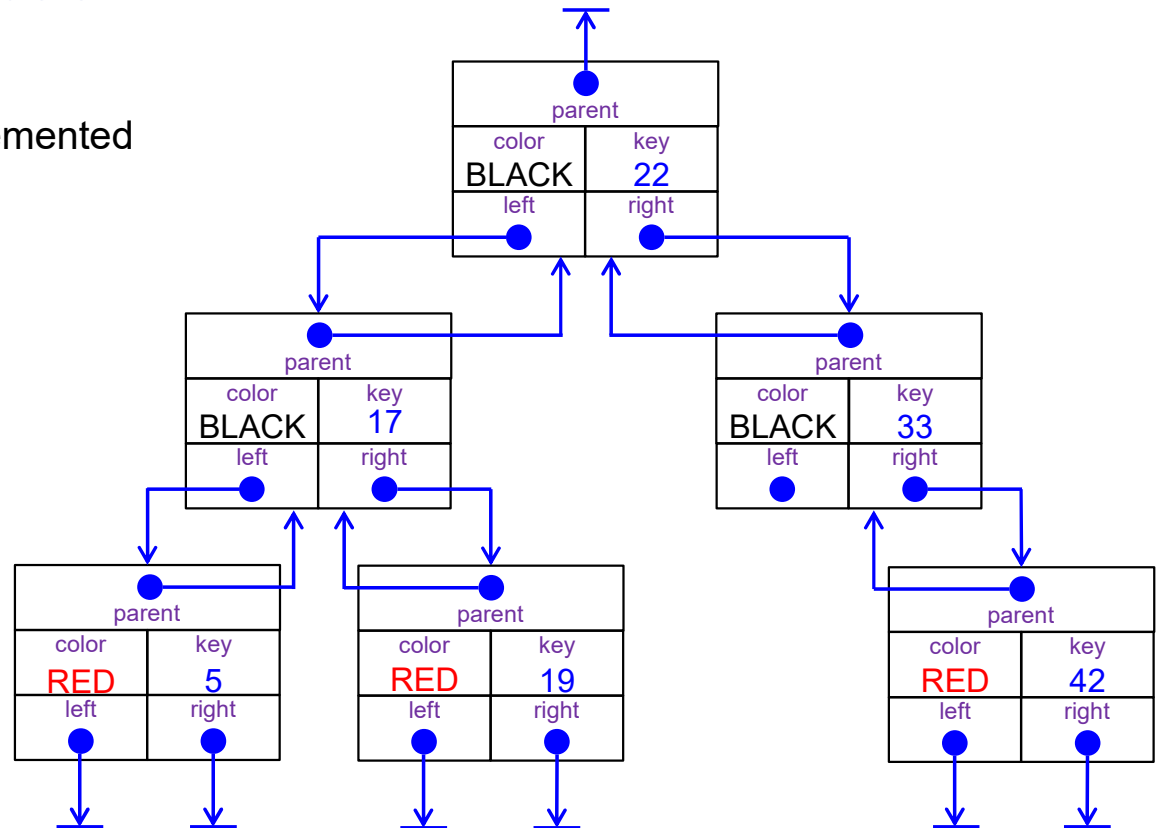
Potentially useful tools to understand and practice with red-black trees:

- Animation by David Galles: <https://www.cs.usfca.edu/~galles/visualization/RedBlack.html>
- Interactive animation by Tommi Kaikkonen: <https://tommikaikkonen.github.io/rbtree/>

## Red-Black Trees: Implementation

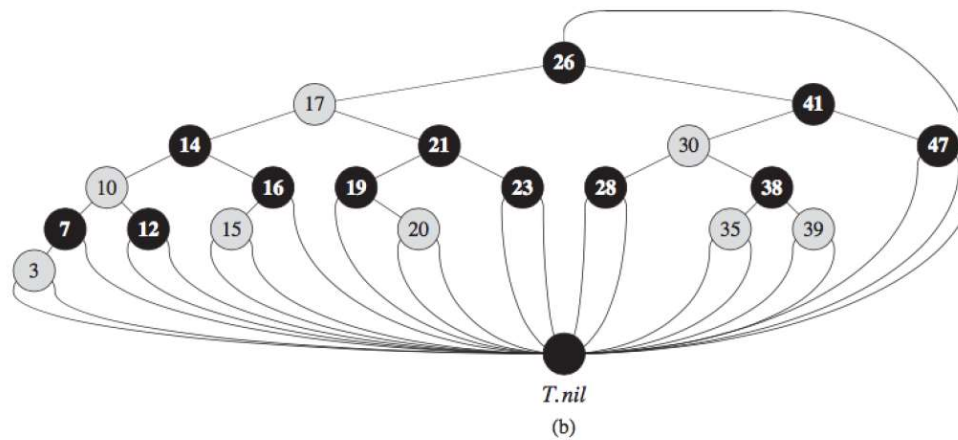
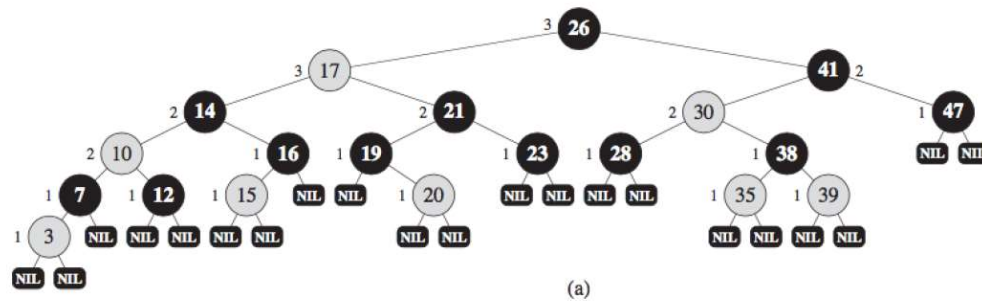
The nodes of a red-black tree can be implemented in C using a struct as follows:

```
struct rb_node {
    int key;
    int color;
    struct rb_node* left;
    struct rb_node* right;
    struct rb_node* parent;
};
```





## Red-Black Trees: Implementation: Sentinel Nodes





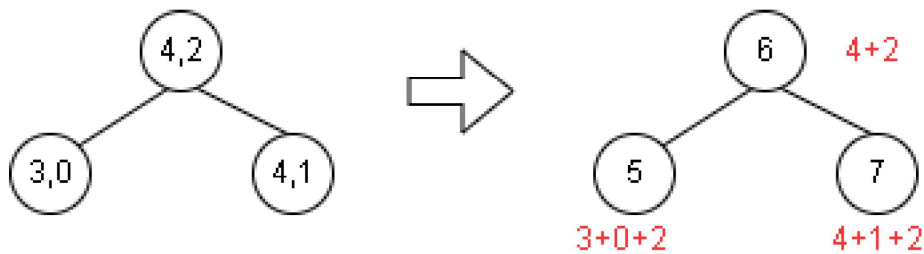


## **Red-Black Trees: Implementation: Comprehension Question**

Why does the implementation of red-black trees contain parent pointers, while this is (usually) not the case for Binary Search Trees?

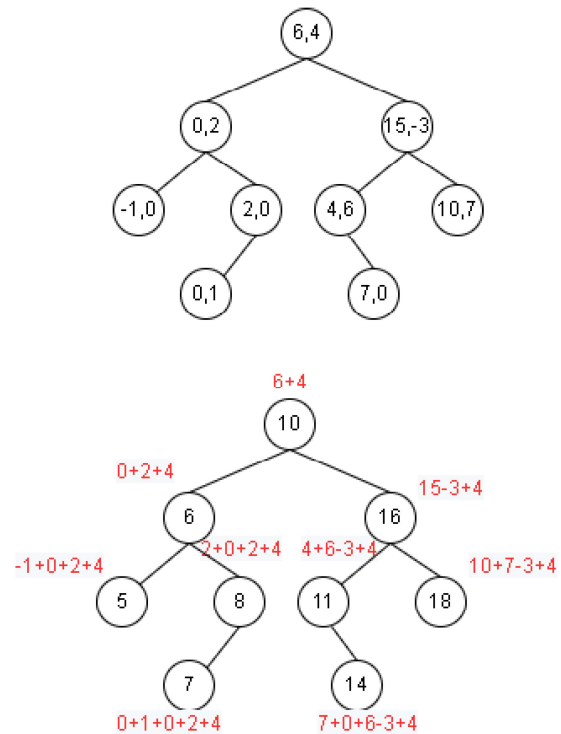
## Exercise 9, Task 5

Consider an enhanced binary search tree in which each node  $v$  contains a key and an value called **addend**. The addend of node  $v$  is implicitly added to keys of all nodes in the subtree rooted at  $v$  (including  $v$ ). Let  $(\text{key}, \text{addend})$  denote the contents of a node. See the following example that transforms an enhanced binary search tree to a binary search tree.

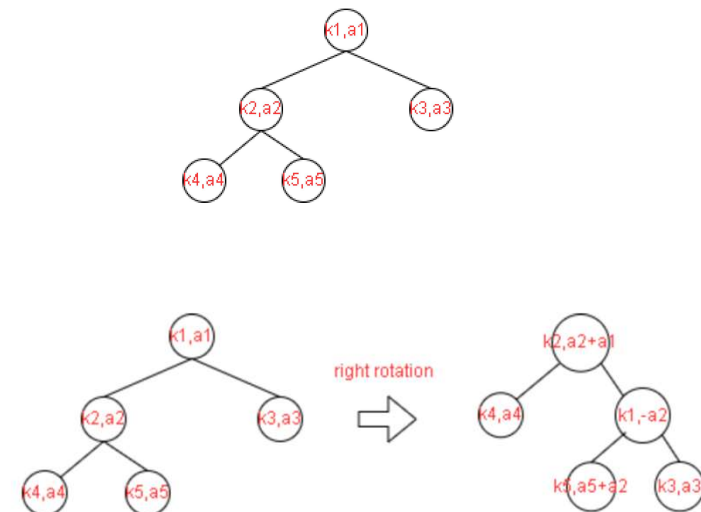


## Exercise 9, Task 5

1. Show the binary search tree of the enhanced binary search tree below?



2. Consider the enhanced binary search tree below. Show the state of the enhanced binary search tree after the right rotation of node k1.





**Universität  
Zürich<sup>UZH</sup>**

**Institut für Informatik**

## Wrap-Up

- Summary
- Feedback
- Outlook
- Questions



**Universität  
Zürich<sup>UZH</sup>**

**Institut für Informatik**

---

## **Wrap-Up**

- Summary



## Outlook on Next Thursday's Lab Session

*Next tutorial:* Wednesday, 11.05.2022, 14.00 h, BIN 0.B.06

*Topics:*

- Review of Exercise 10
- Hashing
- ...
- ... (your wishes)



**Universität  
Zürich<sup>UZH</sup>**

**Institut für Informatik**

---

## **Questions?**



**Universität  
Zürich<sup>UZH</sup>**

**Institut für Informatik**

---

*Thank you for your attention.*