BST and AVL

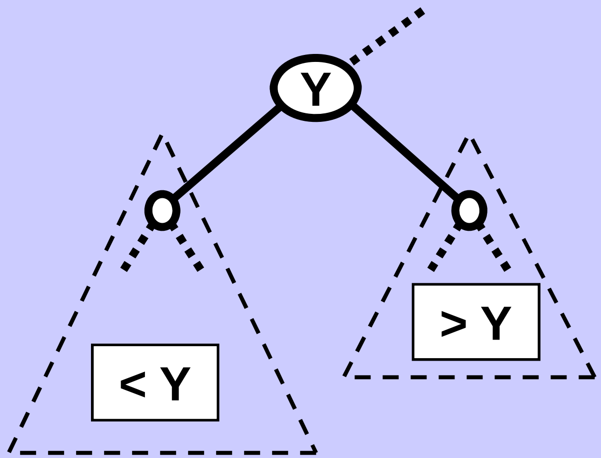
short illustrative repetition

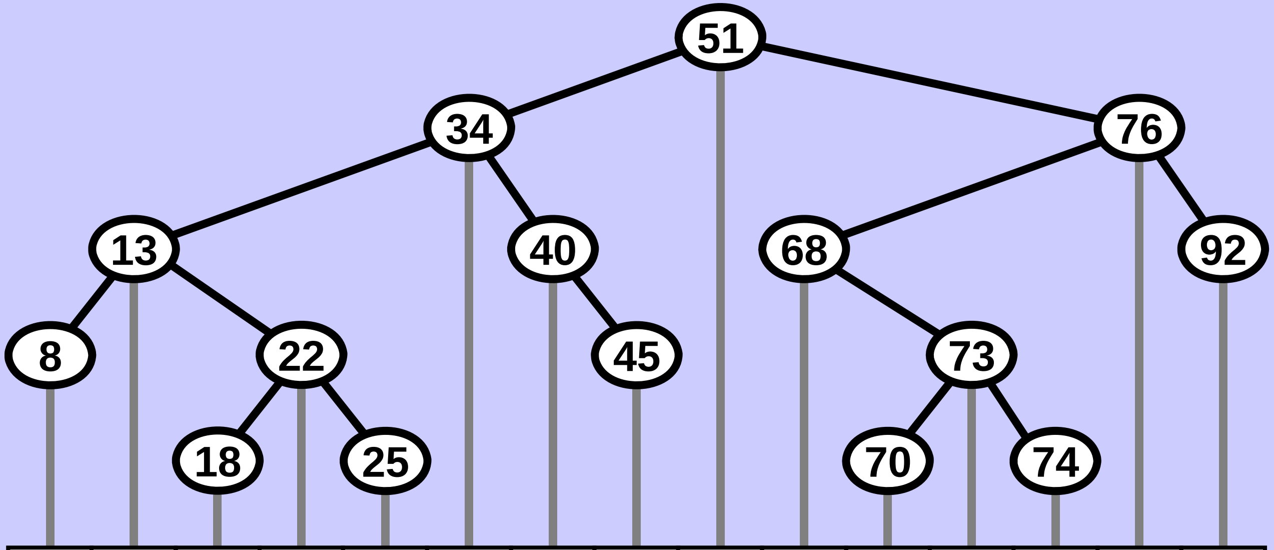
Binary search tree

For each node Y it holds:

Keys in the left subtree of Y are smaller than the key of Y.

Keys in the right subtree of Y are bigger than the key of Y.





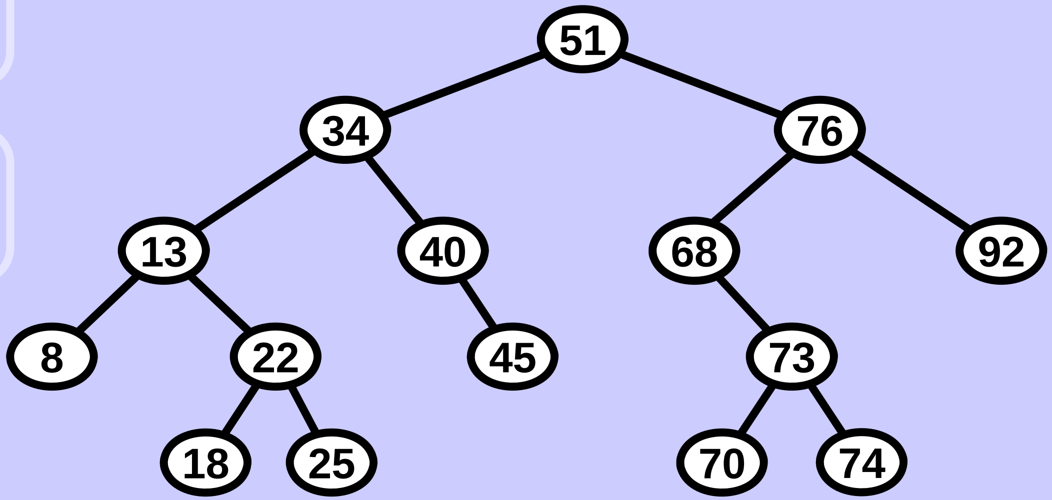
8	13	18	22	25	34	40	45	51	68	70	73	74	76	92

Binary search tree

BST may not be balanced and usually it is not.

BST may not be regular and usually it is not.

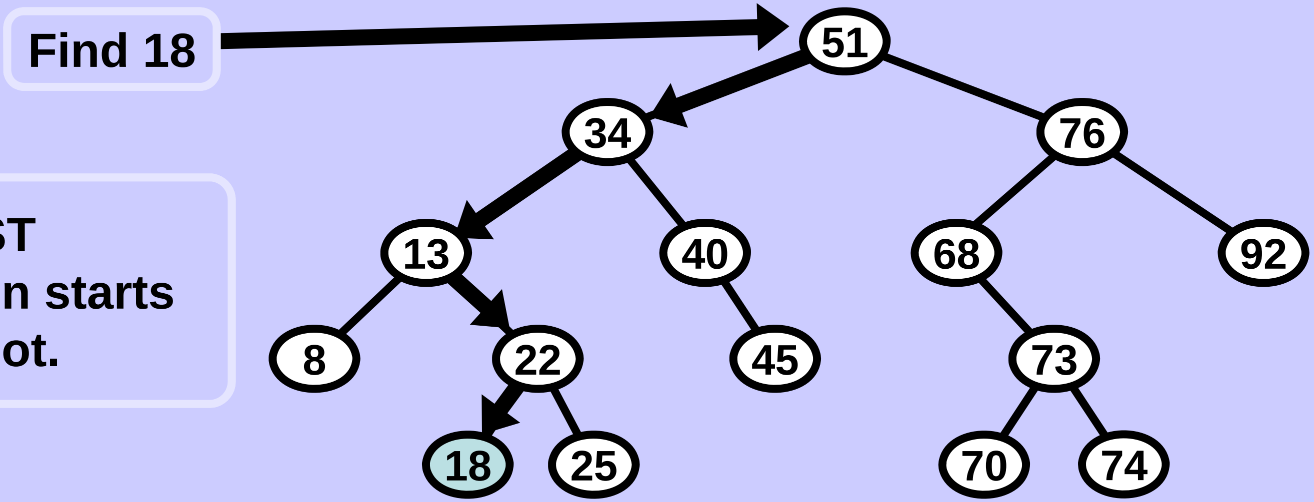
Apply the INORDER traversal to obtain sorted list of the keys of BST.



BST is flexible due to operations:

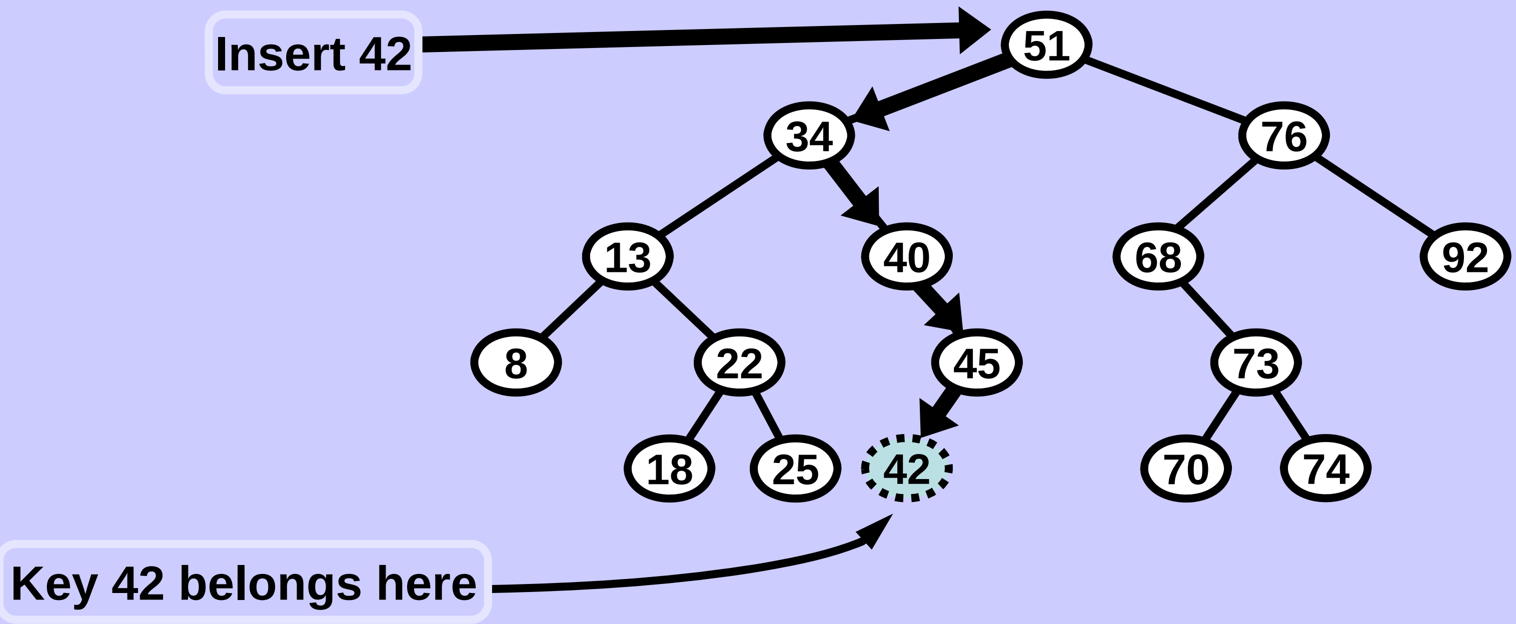
Find – return the pointer to the node with the given key (or null).
Insert – insert a node with the given key.
Delete – (find and) remove the node with the given key.

Operation Find in BST



Each BST operation starts in the root.

Operation Insert in BST

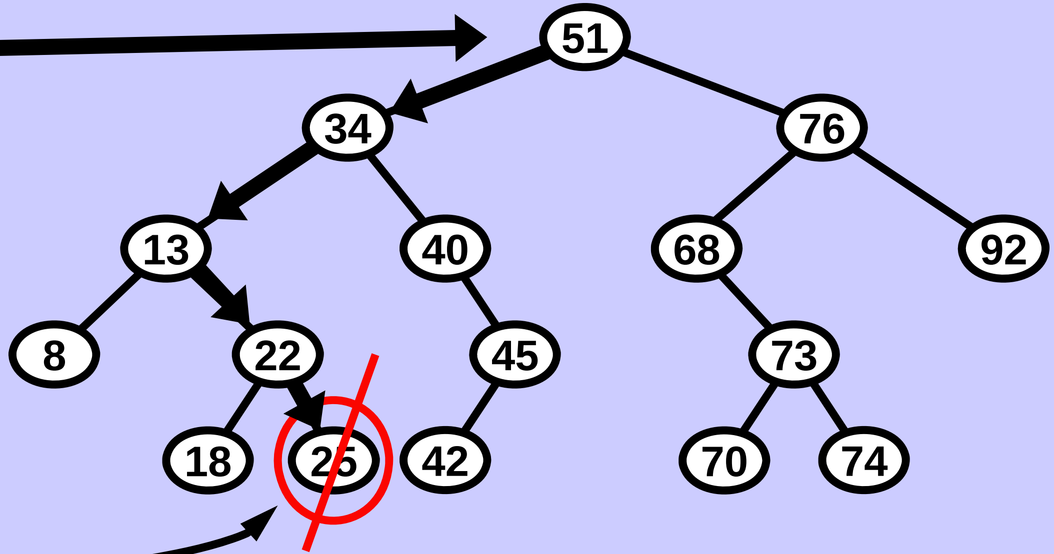


Insert
1. Find the place (like in Find) for the leaf where the key belongs.
2. Create this leaf and connect it to the tree.

Operation Delete in BST (I.)

Delete a node with 0 children (= leaf)

Delete 25



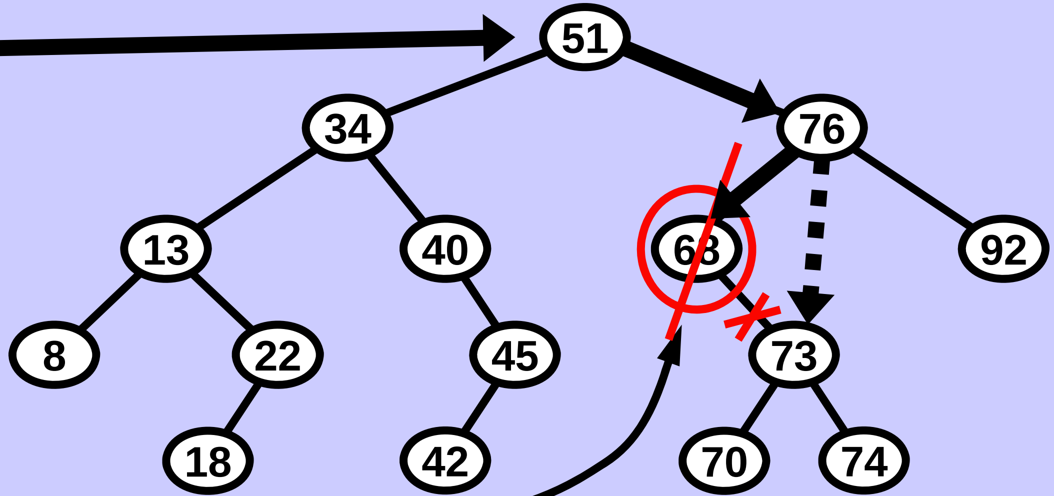
Leaf with key 25 disappears

Delete I. **Find the node (like in Find operation) with the given key and set the reference to it from its parent to null.**

Operation Delete in BST (II.)

Delete a node with 1 child.

Delete 68



Node with key 68 disappears

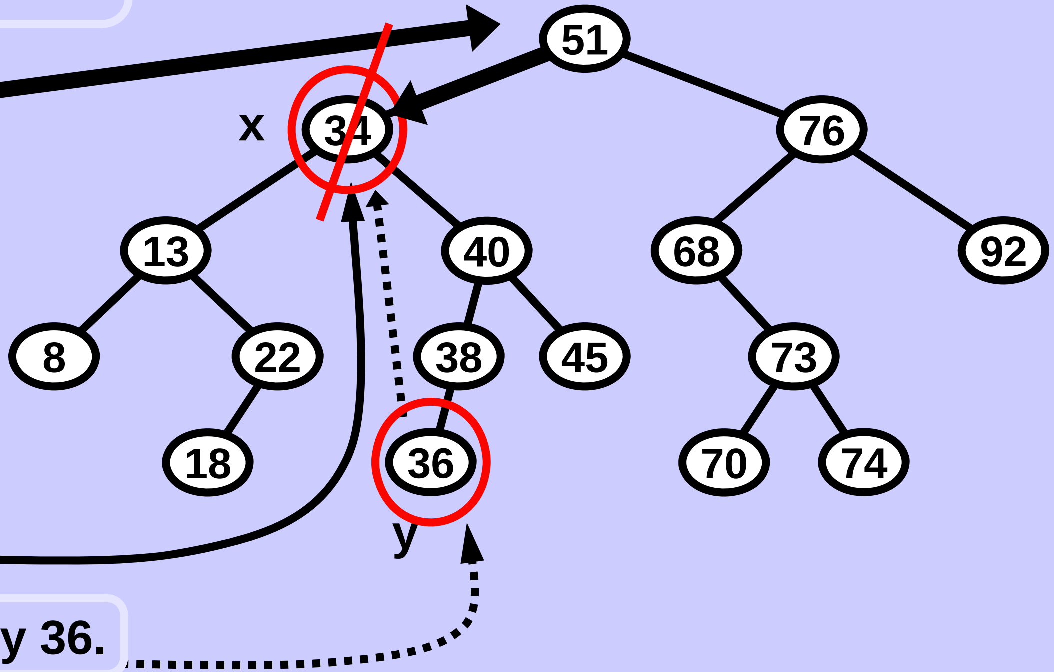
Change the 76 --> 68 reference to 76 --> 73 reference.

Delete II. Find the node (like in Find operation) with the given key and set the reference to it from its parent to its (single) child.

Operation Delete in BST (IIIa.)

Delete a node with 2 children.

Delete 34



Key 34 disappears.

And it is substituted by key 36.

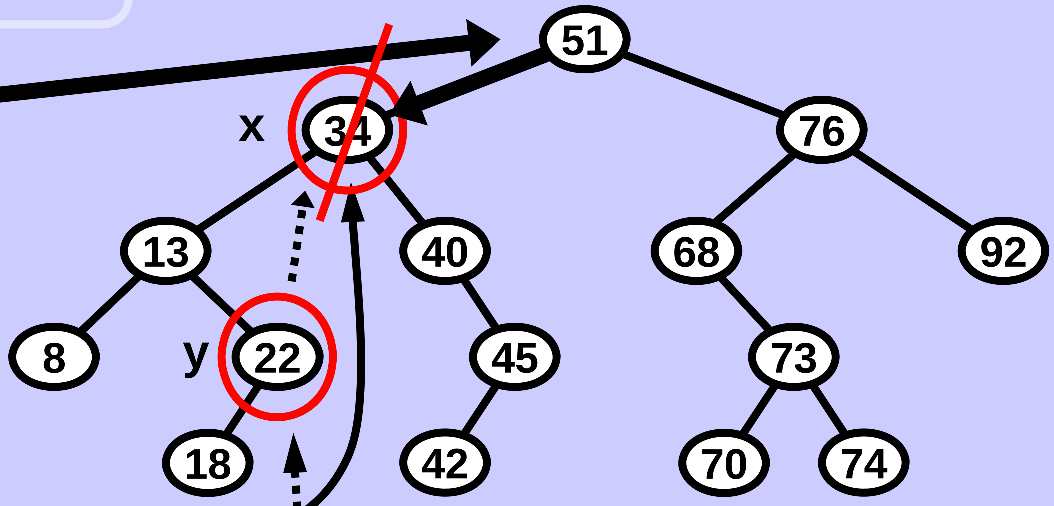
Delete IIIa.

1. Find the node (like in Find operation) with the given key and then find the ***leftmost*** (= smallest key) node y in the ***right*** subtree of x.
2. Point from y to children of x, <u>from parent of y point to the child of y instead of y</u>, from parent of x point to y.

Operation Delete in BST (IIIb.) is equivalent to Delete IIIa.

Delete a node with 2 children.

Delete 34



Key 34 disappears.

And it is substituted by key 22.

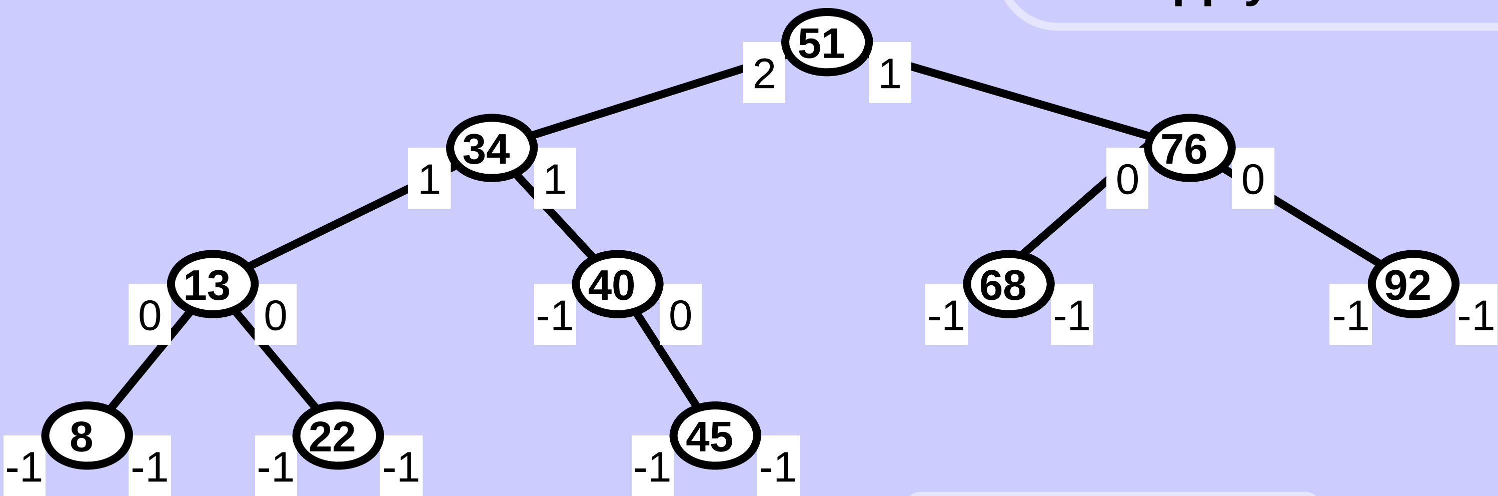
Delete IIIb.

1. Find the node (like in Find operation) with the given key and then find the ***rightmost*** (= smallest key) node y in the ***left*** subtree of x.
2. Point from y to children of x, <u>from parent of y point to the child of y instead of y</u>, from parent of x point to y.

AVL tree -- G.M. Adelson-Velskij & E.M. Landis, 1962

AVL tree is a BST with additional properties which keep it acceptably balanced.

Operations
Find, Insert, Delete
also apply to AVL tree.



There are two integers associated with each node:
Depth of the left and depth of the right subtree of the node.
Note: Depth of an empty tree is -1.

AVL rule:

The difference of the heights of the left and the right subtree may be only -1 or 0 or 1 in each node of the tree.

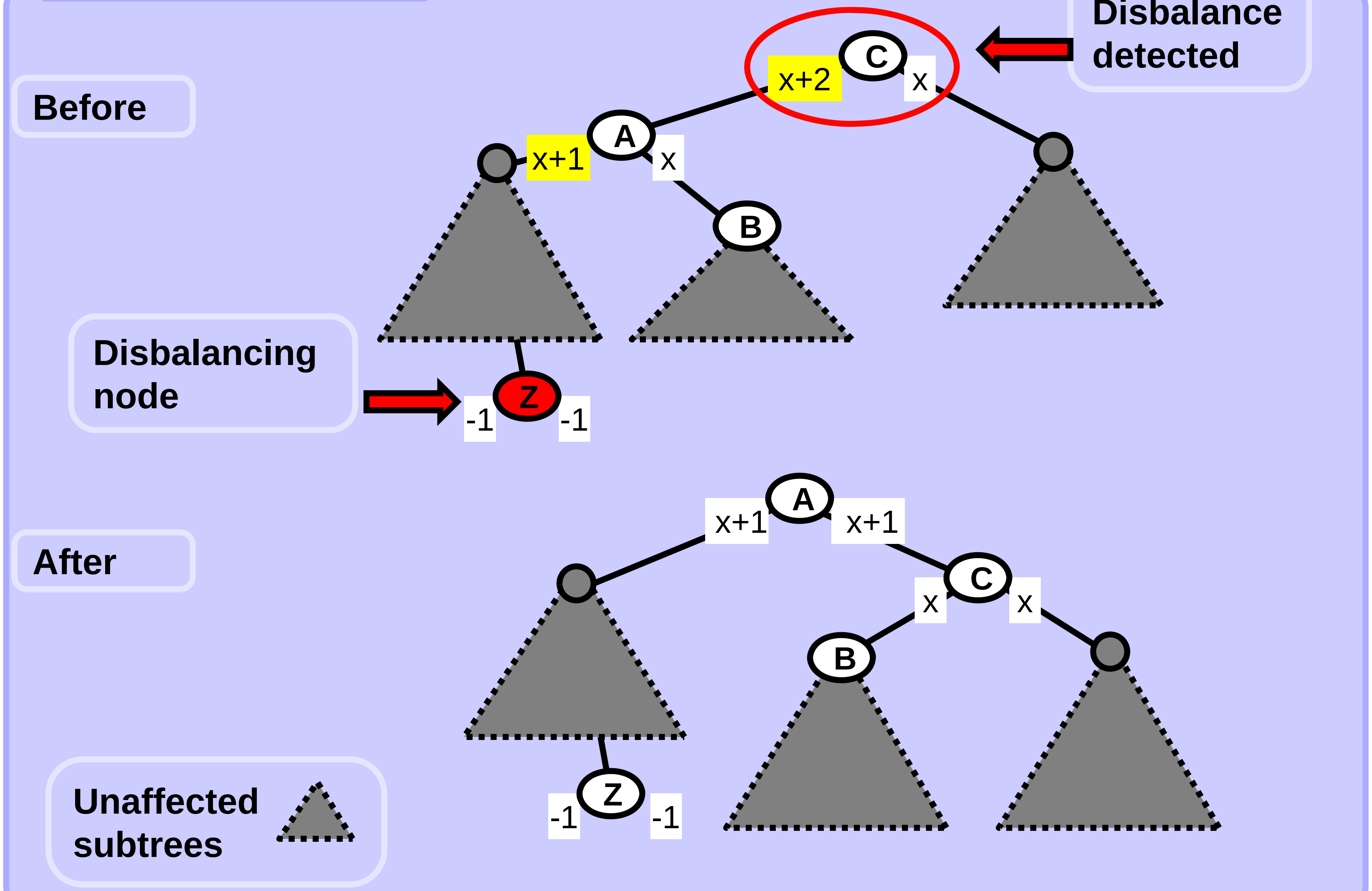
AVL tree -- G.M. Adelson-Velskij & E.M. Landis, 1962

Find -- same as in a BST

Insert -- first, insert as in a BST,
next, travel from the inserted node upwards
and update the node depths.
If disbalance occurs in any node along the path then
apply an appropriate rotation and stop.

Delete -- first, delete as in a BST,
next, travel from the deleted position upwards
and update the node depths.
If disbalance occurs in any node along the path then
apply an appropriate rotation.
Continue travelling along the path up to the root.

Rotation R in general

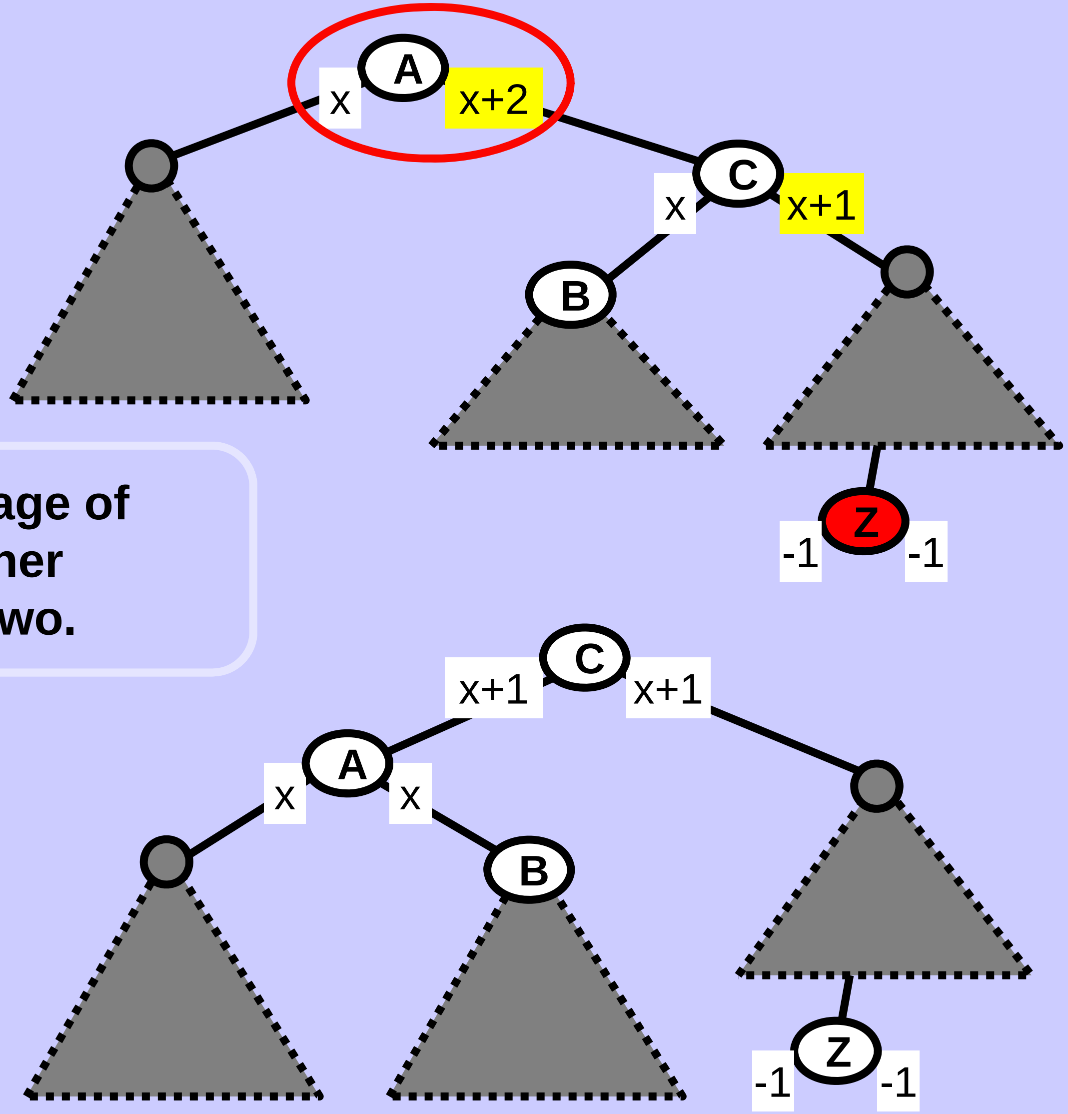


Rotation L in general

Before

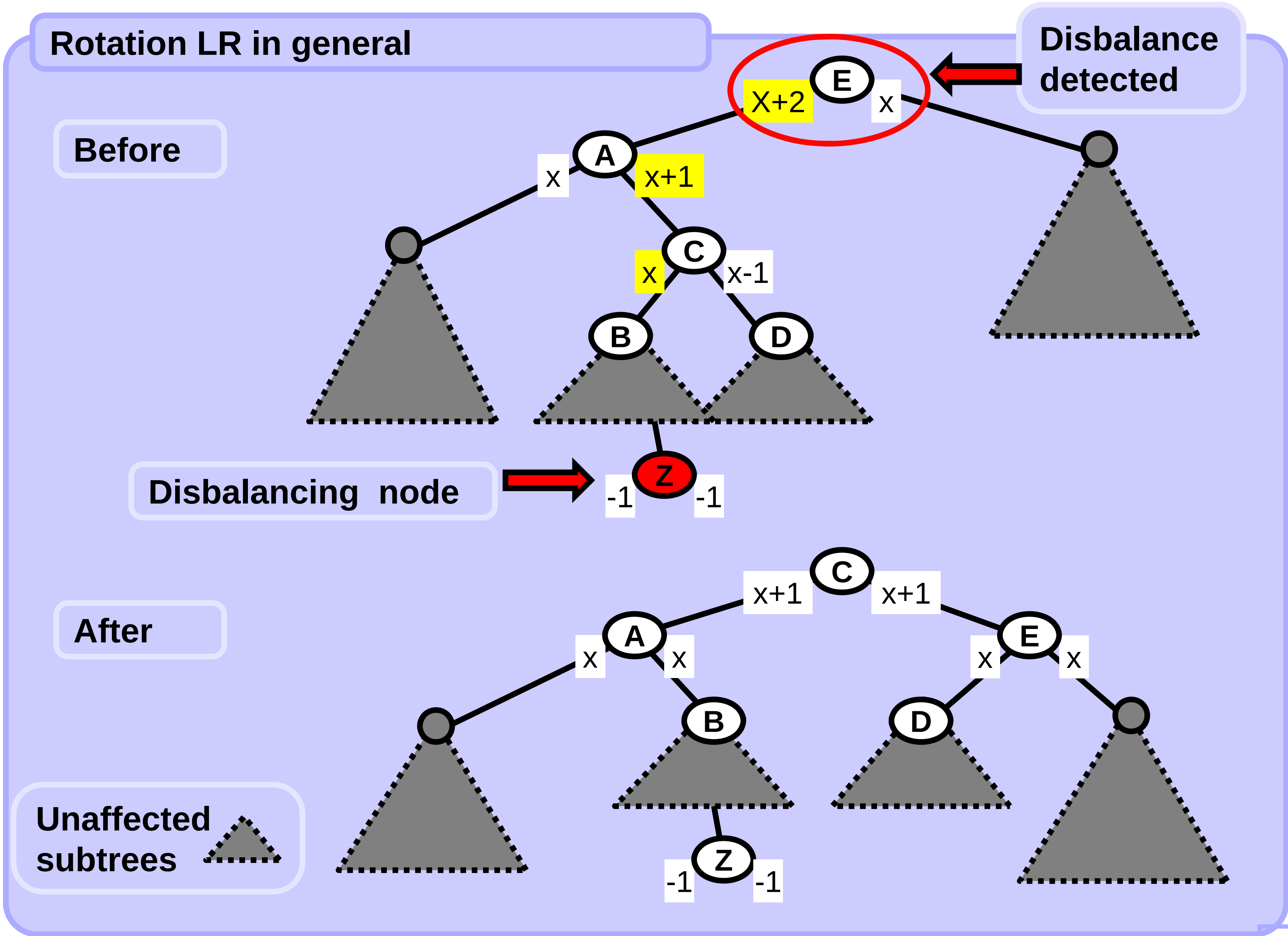
Rotation L is a mirror image of rotation R, there is no other difference between the two.

After



Unaffected subtrees

Rotation LR in general

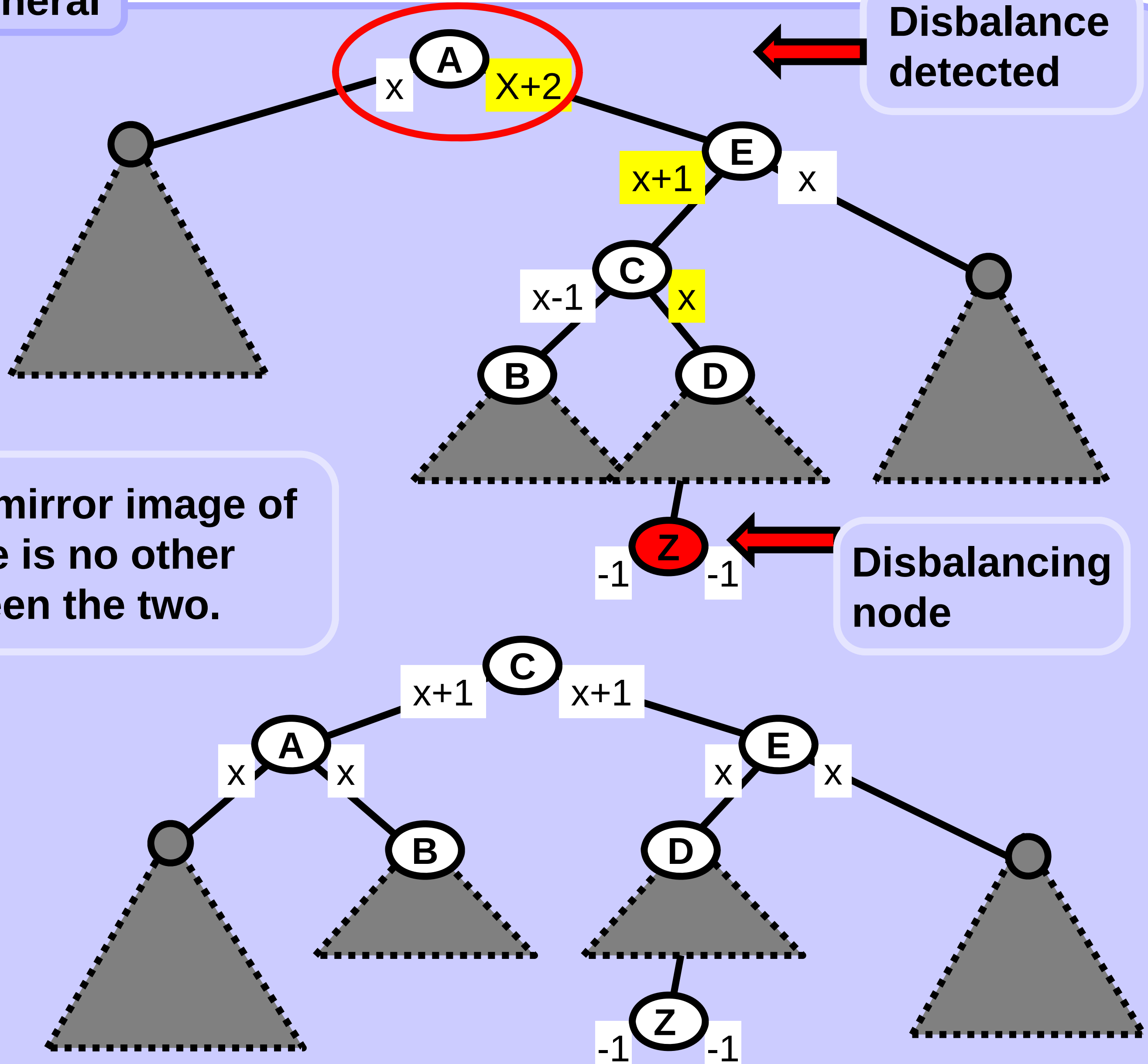


Rotation RL in general

Before

After

Rotation RL is a mirror image of rotation LR, there is no other difference between the two.



Unaffected subtrees

Rules for aplying rotations L, R, LR, RL in Insert operation

Travel from the inserted node up to the root
and update the subtree depths in each node along the path.

If a node is disbalanced and you came to it along two consecutive edges

* in the up and *right* direction
  perform rotation R in this node,
* in the up and *left* direction
  perform rotation L in this node,
* first in the in the up and *left* and then in the up and *right* direction
  perform rotation LR in this node,
* first in the in the up and *right* and then in the up and *left* direction
  perform rotation RL in this node,

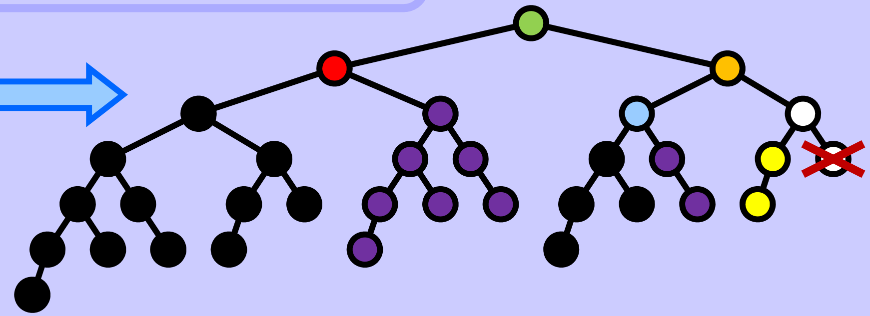
After one rotation in the Insert operation the AVL tree is balanced.

After one rotation in the Delete operation the AVL tree might still
not be balanced, all nodes on the path to the root have to be checked.

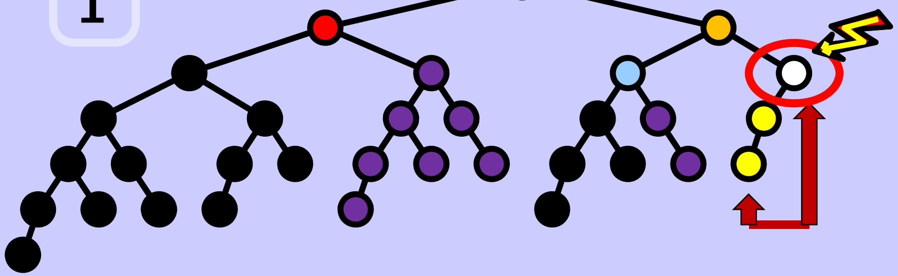
Necessity of multiple rotations in operation Delete.

**Example.
The AVL tree
is originally
balanced.**

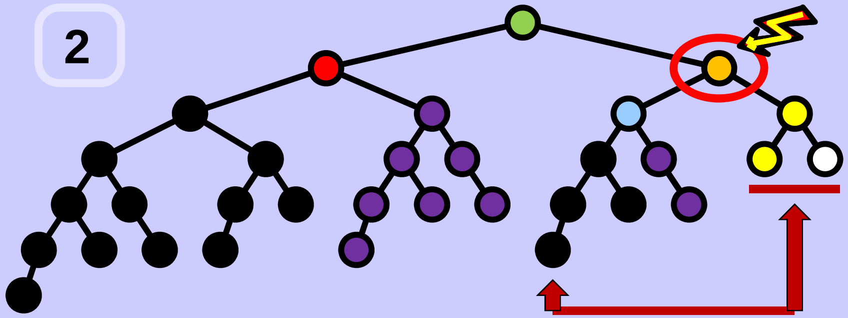
**Delete the
rightmost key.**



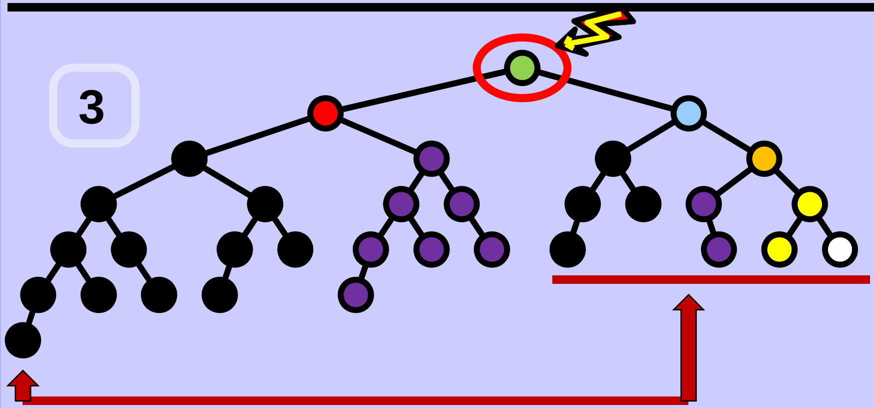
1



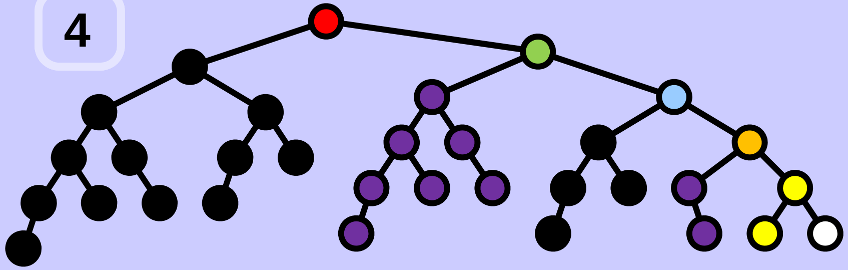
2



3



4



Balanced.

Asymptotic complexities of Find, Insert, Delete in BST and AVL

	BST with n nodes		AVL tree with n nodes
Operation	Balanced	Maybe not balanced	Balanced
Find	$O(\log(n))$	$O(n)$	$O(\log(n))$
Insert	$\Theta(\log(n))$	$O(n)$	$\Theta(\log(n))$
Delete	$O(\log(n))$	$O(n)$	$\Theta(\log(n))$