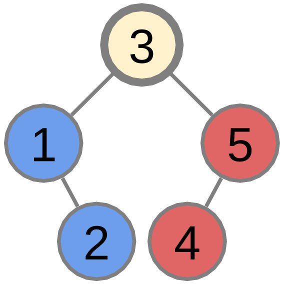
Maintaining Balance: Balanced BST

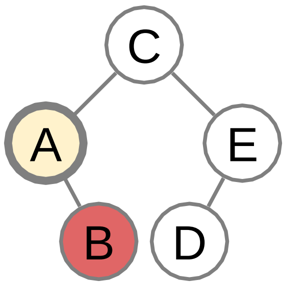
Lecture 20

by Marina Barsky

Recap: Definition

Binary search tree is a binary tree with the following property:
for each node with key ***x***, all the nodes in its **left subtree** have keys **smaller than *x***, and all the keys in its **right subtree** are **greater than *x***.





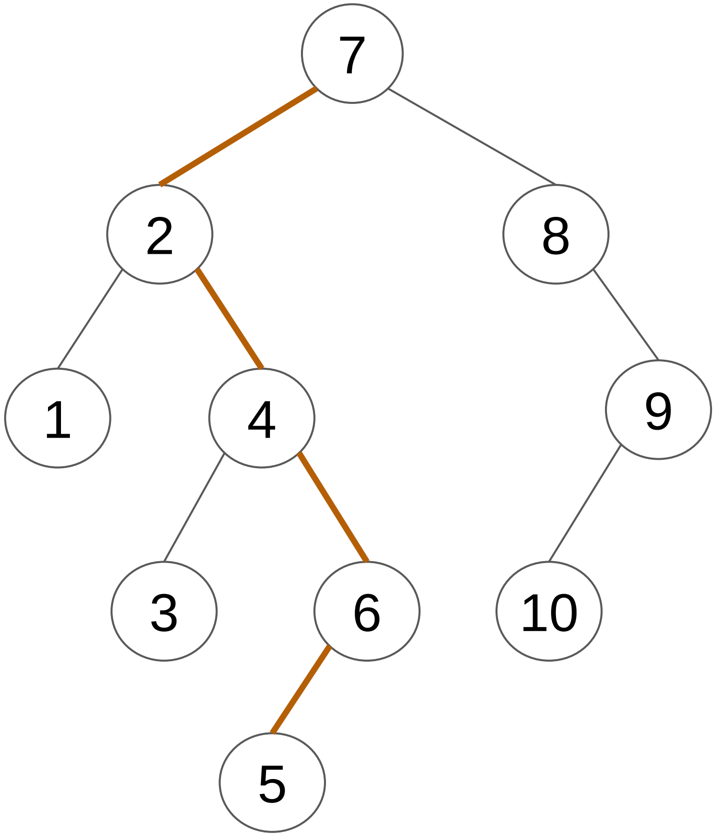
Recap: Operations on BST

- **Search (*k*)**
- **Successor (*k*)/Predecessor (*k*)**
- **Add (*k*)**
- **Remove (*k*)**

How fast is each operation?

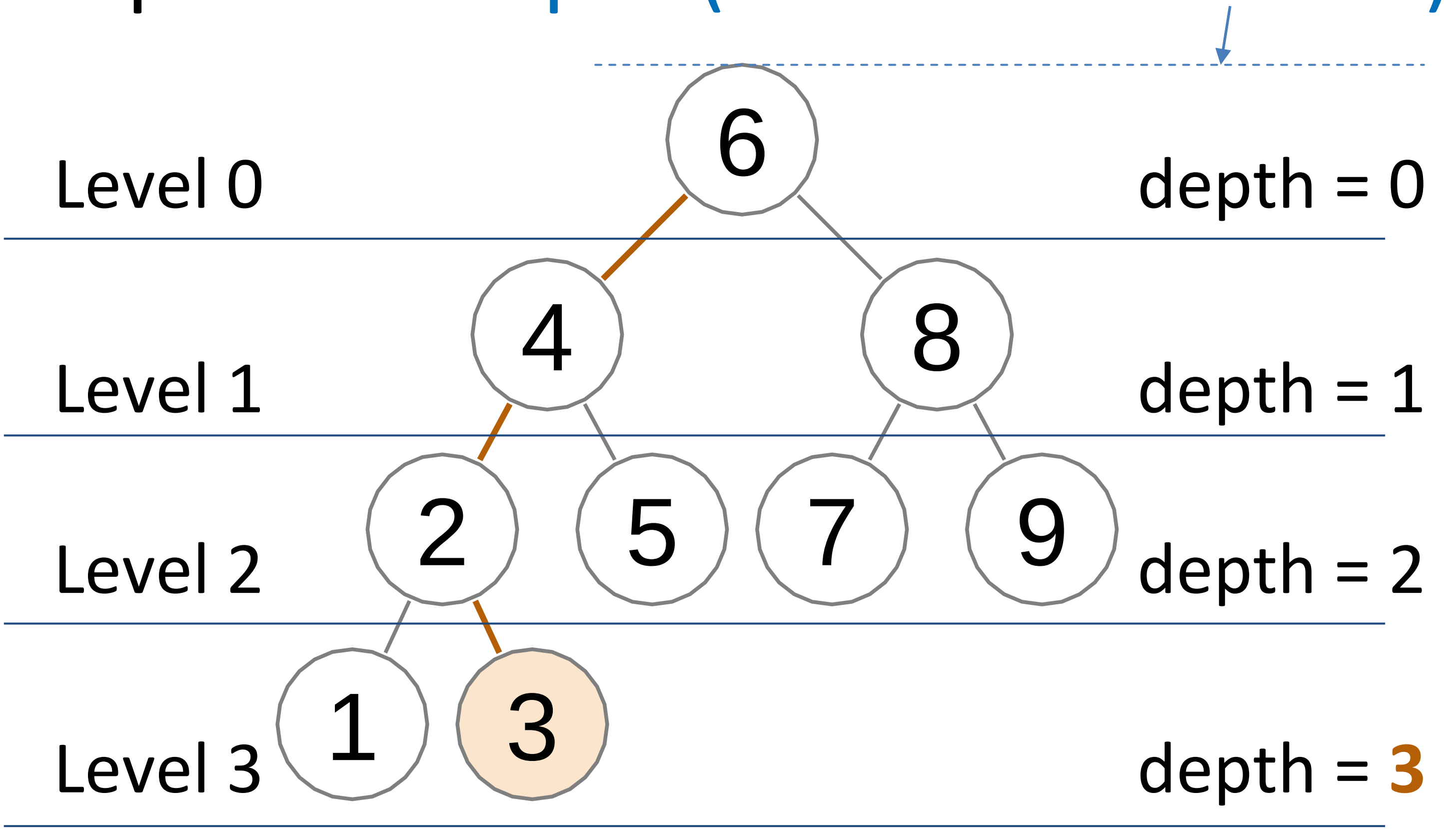
Example: search (*k*)

search (5)



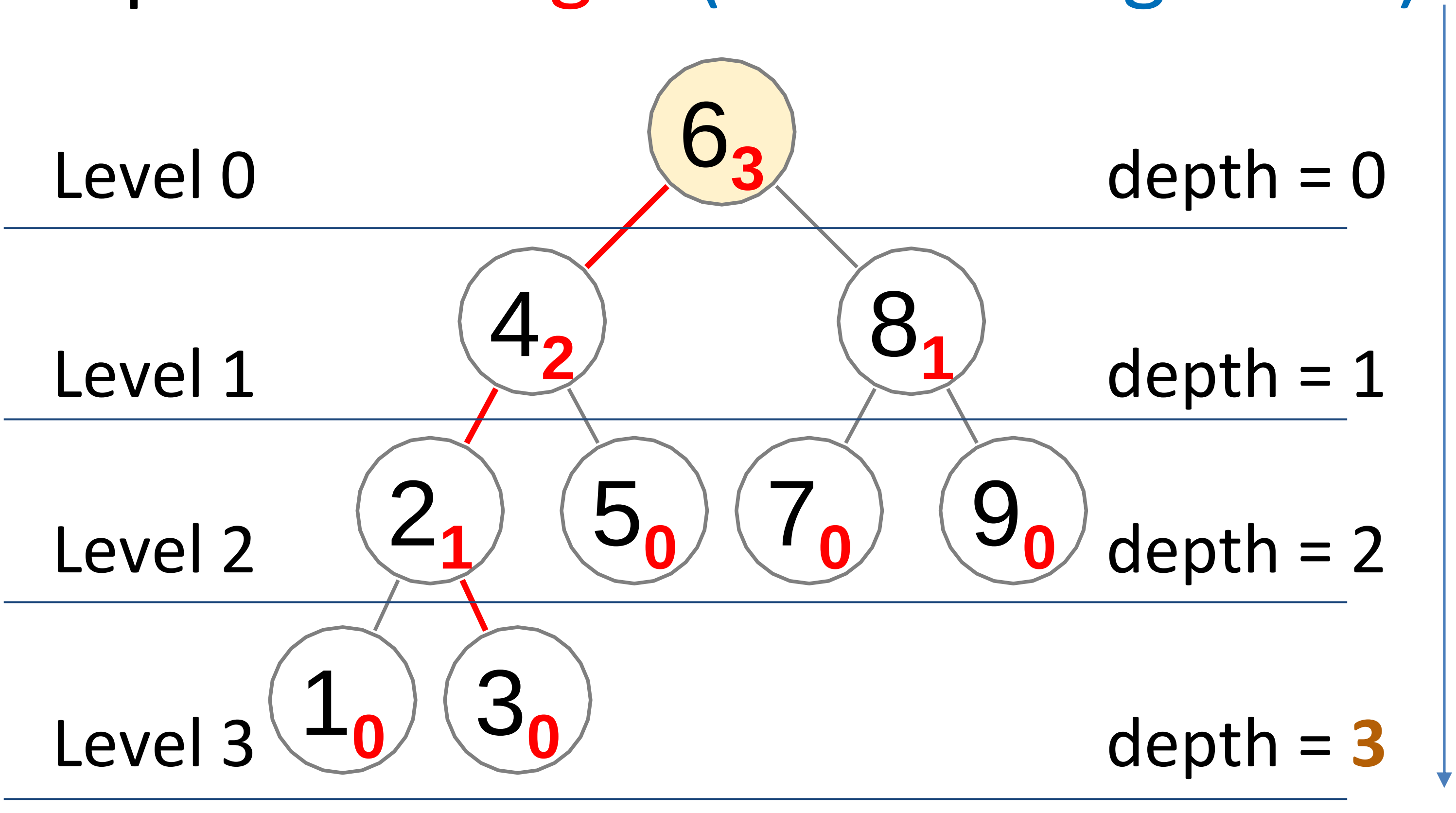
Total questions asked before we reach 5: **4**

Recap: node depth (below the surface)



Distance from the root:
how many edges to go from the root to a given node

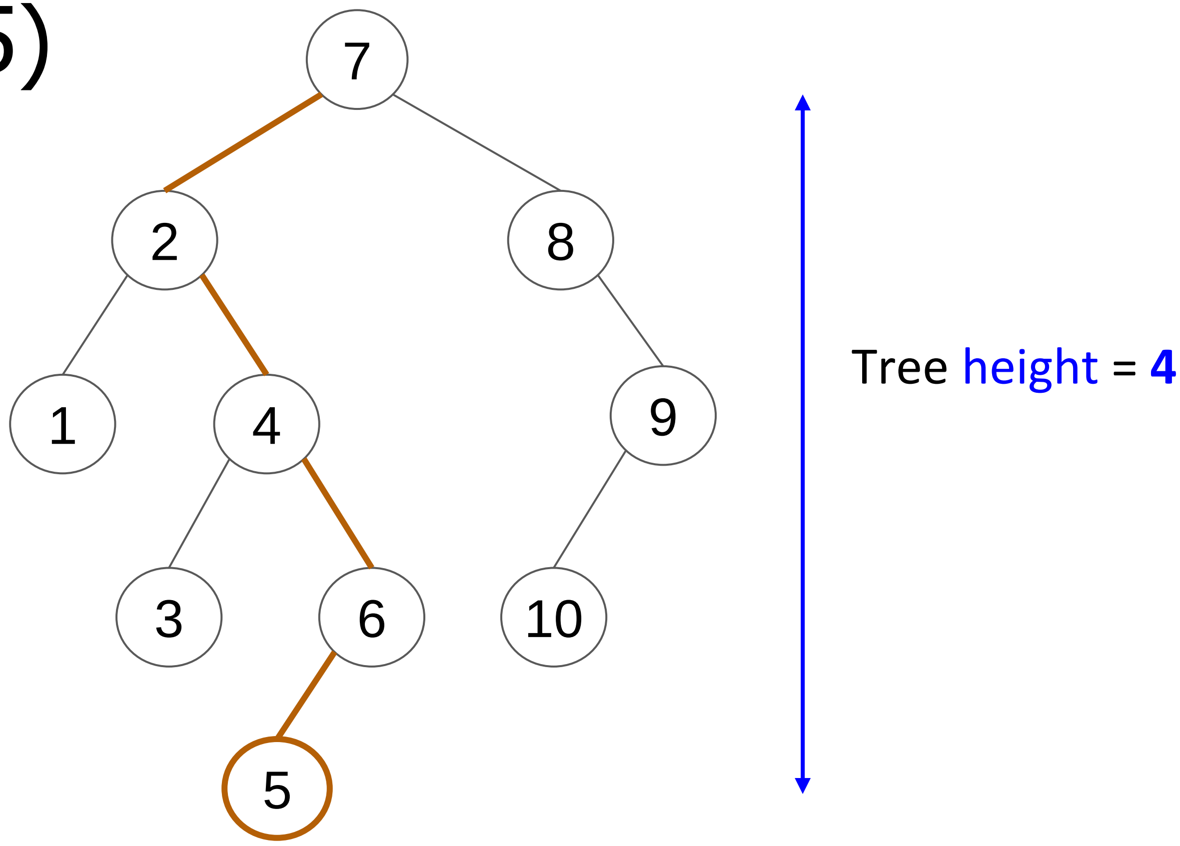
Recap: node height (above the ground)



Distance from the node to the bottom:
how many edges to go to the furthest leaf

Complexity: search (*k*)

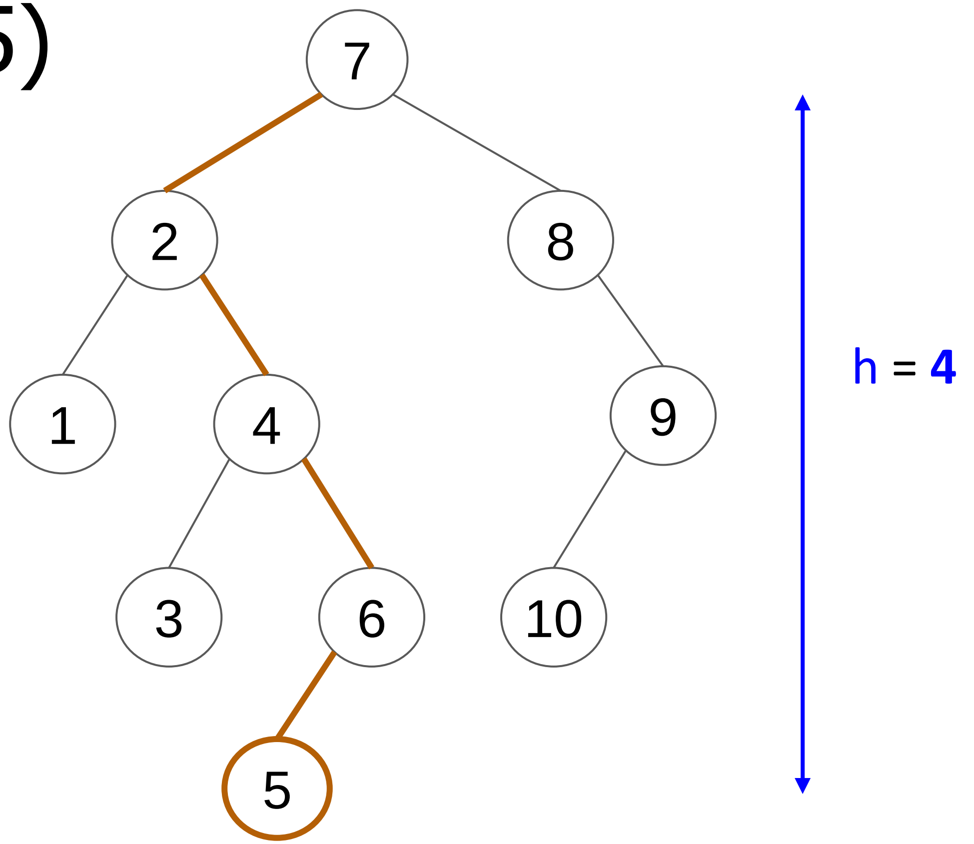
search (5)



- The number of operations is the depth of the node in question
- In the worst case it is bounded by the height of the tree

Complexity: search (*k*)

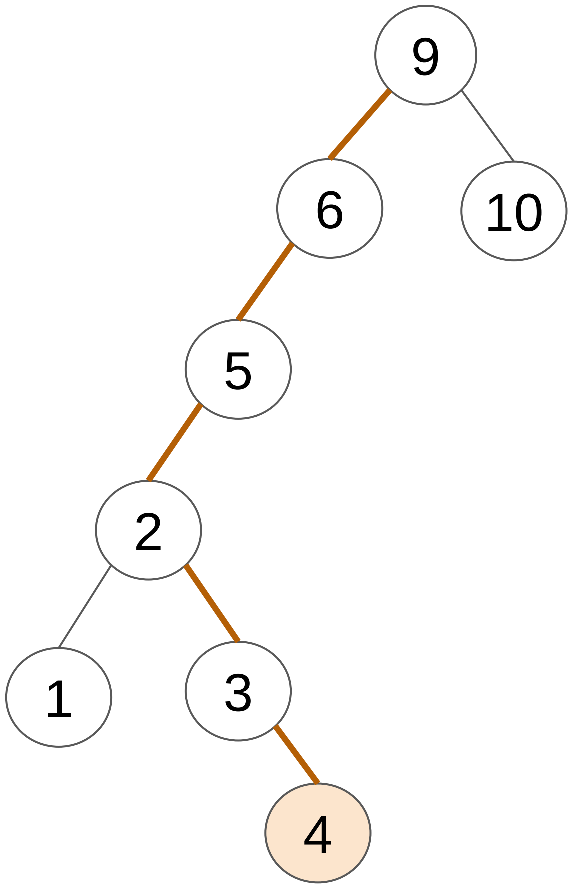
`search` (5)



- The complexity of all BST operations is O(*h*)
- What is the height of the tree in terms of *n* - number of nodes?

Complexity: O (*n*)

`search` (4)



The height can be as bad as O(***n***) !

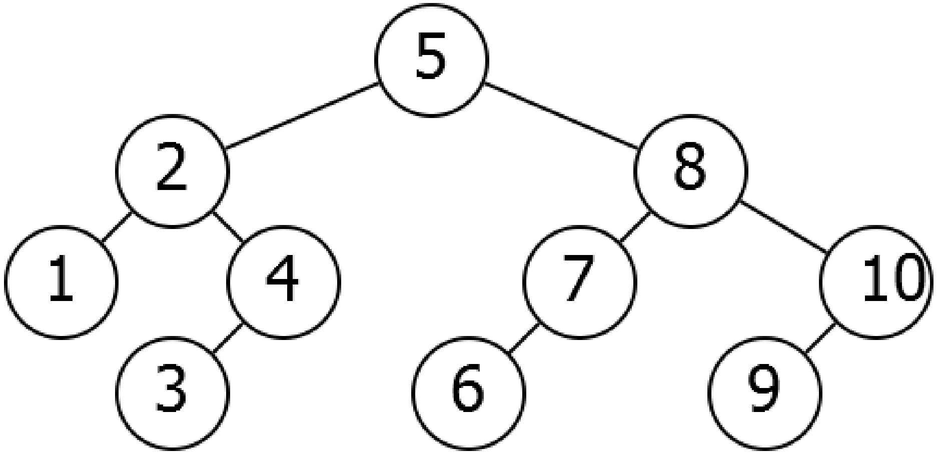
We could do O(n) before:

Sorted Array

- ➔ Range Search: $O(\log(n))$ ✓
- ➔ Nearest Neighbors: $O(\log(n))$ ✓
- ➔ Insert: $O(n)$ ×
- ➔ Delete: $O(n)$ ×

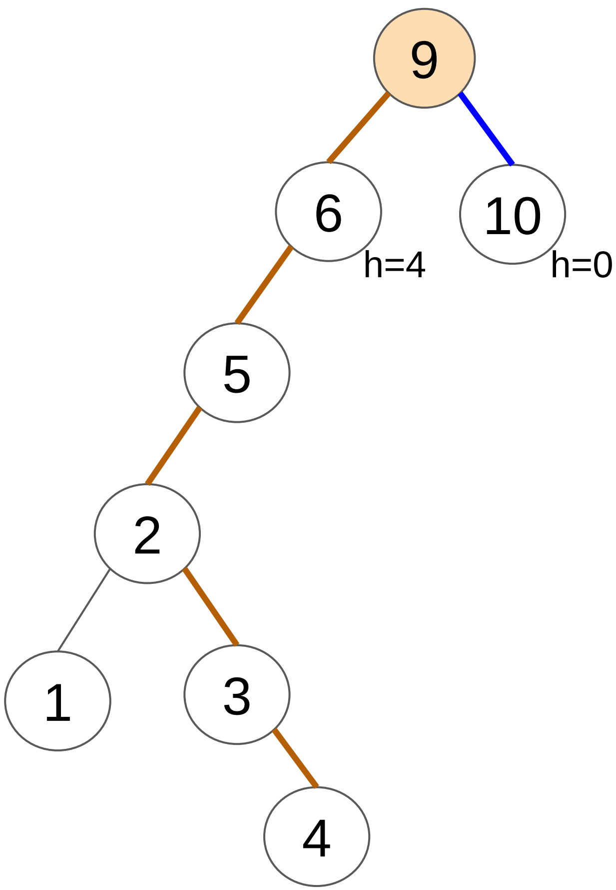
Linked List

- ➔ Range Search: $O(n)$ ×
- ➔ Nearest Neighbors: $O(n)$ ×
- ➔ Insert: $O(1)$ ✓
- ➔ Delete: $O(1)$ ✓



Height can be much smaller than O(n)

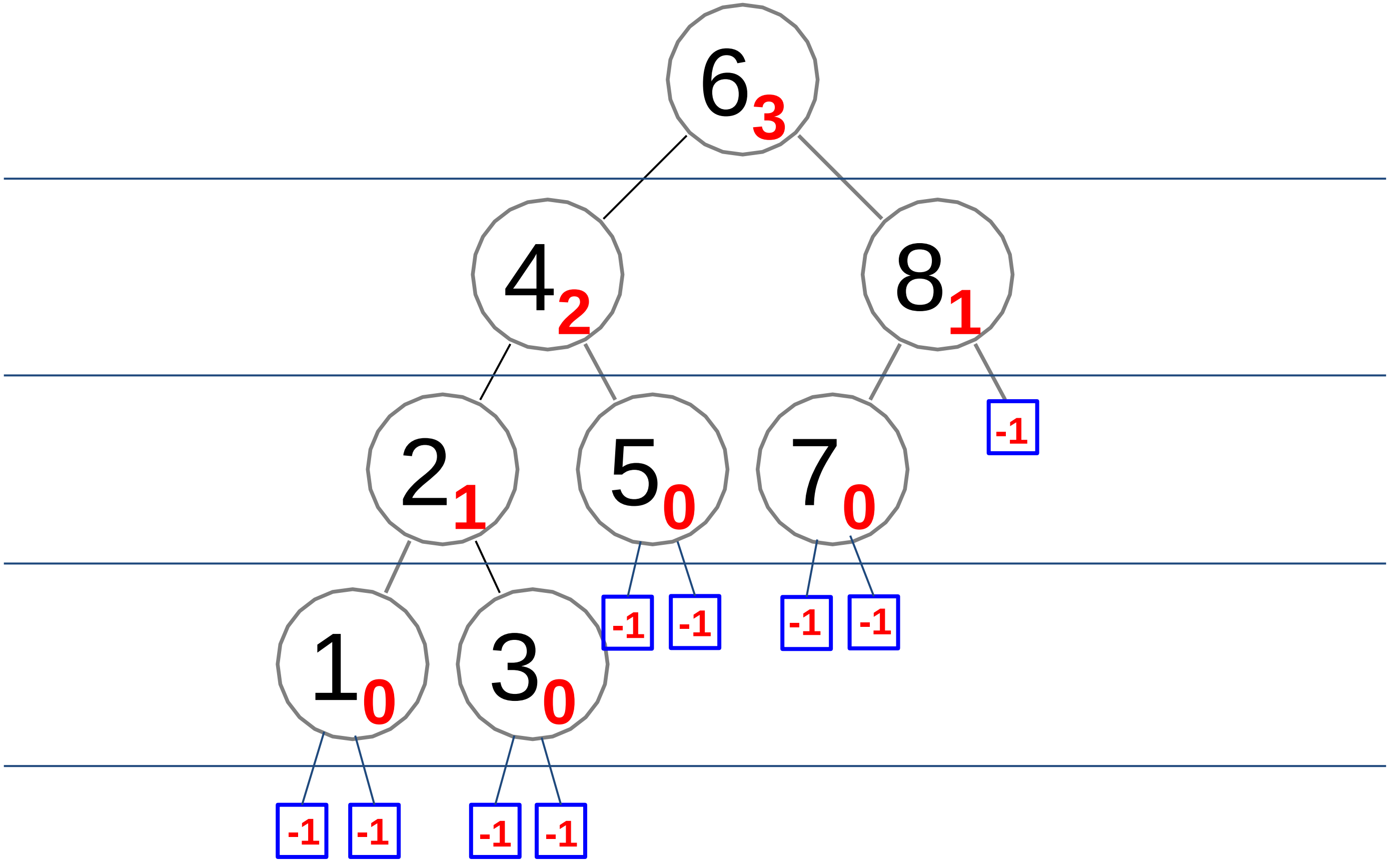
Keeping height low



- The worst-case running time of all the operations is proportional to the tree height *h*
- To achieve optimal performance we need to keep the height low
- One possible way: avoid disbalance in tree nodes

- The node is out of balance if the heights of its children differ by a lot

The height of Null nodes



To make it easier to compare balance of node's children - let's think of **each** BST node having **exactly 2 children**

If either left or right child is NULL - we consider it to be a special NULL node with height -1

Defining balance

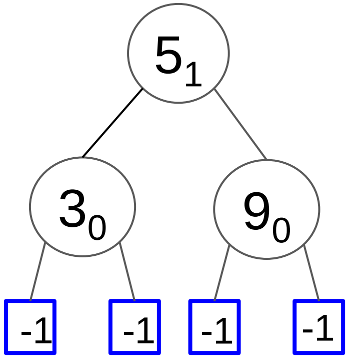
- One possible definition:

  For every internal node *v*, **the heights of the children** of *v* **may differ by at most 1**

- That is, if a node *v* has children, *x* and *y*, then $|h(x) - h(y)| \leq 1$.

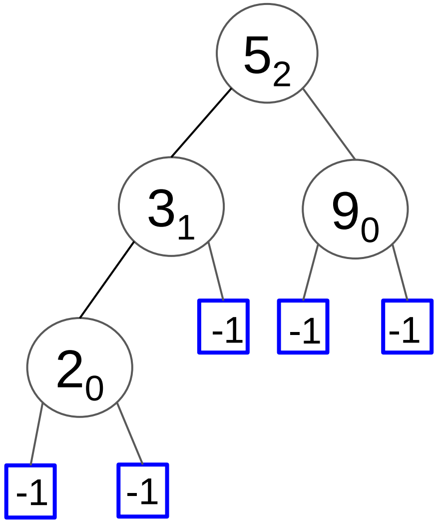
- That implies that we should track the current height for each node of the BBST

How the balance can be destroyed



We start with a perfectly balanced tree

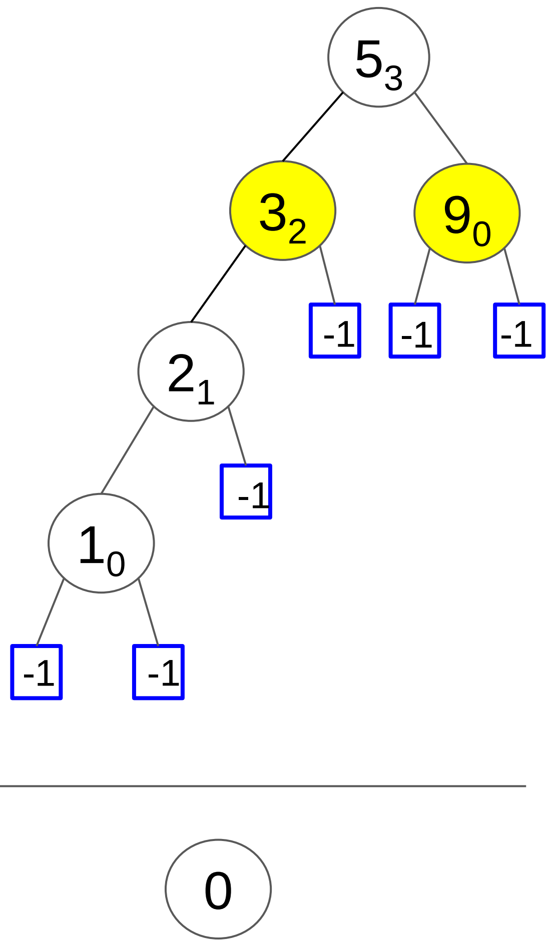
How the balance can be destroyed



We insert key 2

The tree is still balanced (check)

How the balance can be destroyed

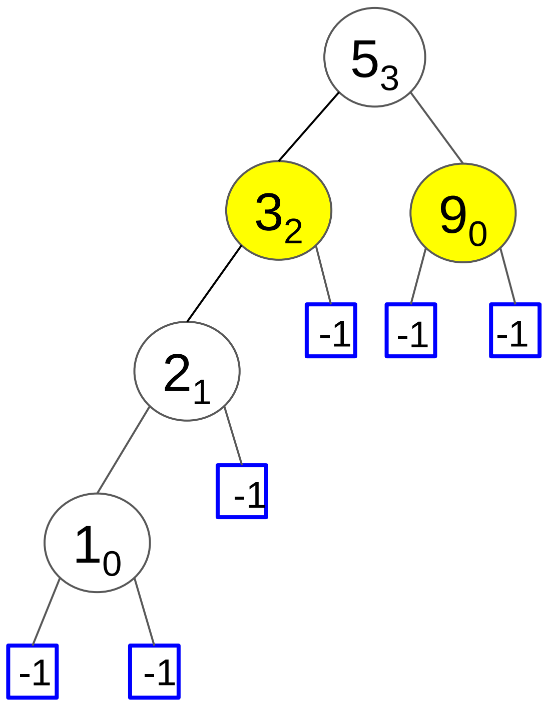


We insert key 1

The root has 2 children x and y and the height of the corresponding subtrees **differs by 2**

If we now add 0 - we will make it even more unbalanced

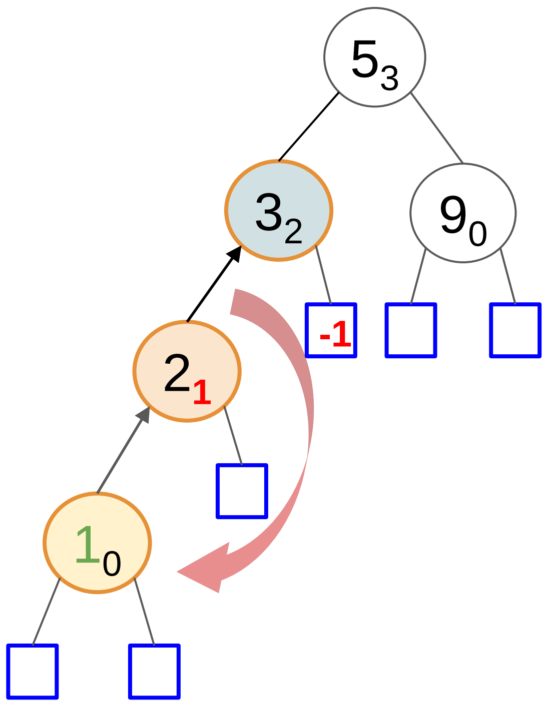
How the balance can be destroyed



We do not leave the tree like that - we rearrange the heavier branch that resulted from adding 1

If we rebalance on time, we will never need to deal with difference > 2

Rebalancing



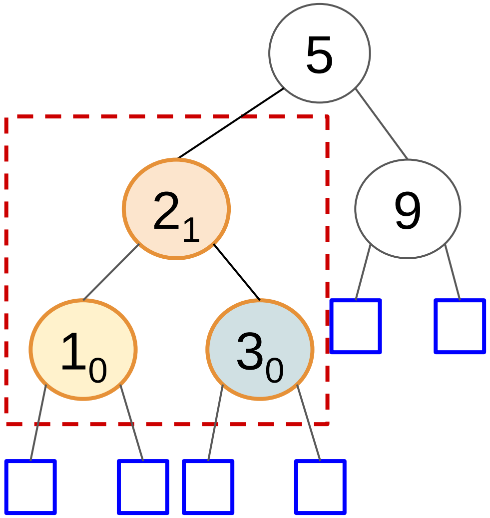
The imbalance in this case is caused by the newly added node **1** and is presented by the path 1, 2, 3 (3 being the first imbalanced node on this path)

We need to rearrange nodes 1,2,3

We can leave all of them in the same tree branch (all are < 5)

1<2<3: so if we pull 2 on top, then 1 will be its left child, and 3 its right child

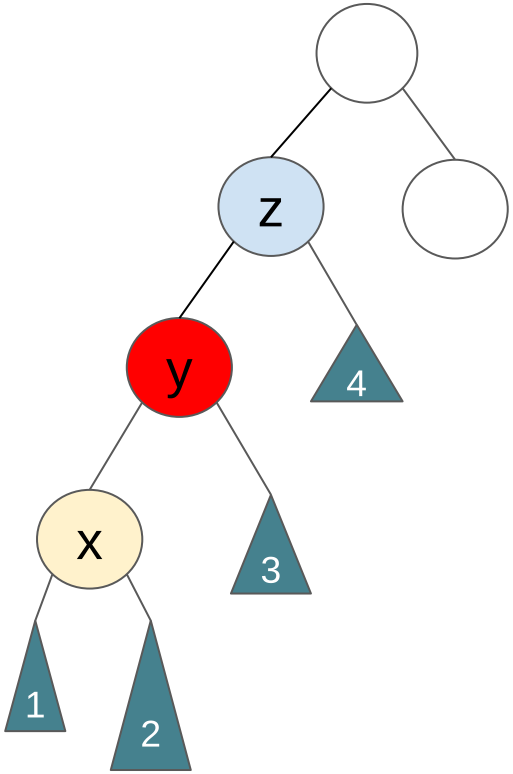
Rebalancing: rotation



This method of rearrangement is called a ***rotation***

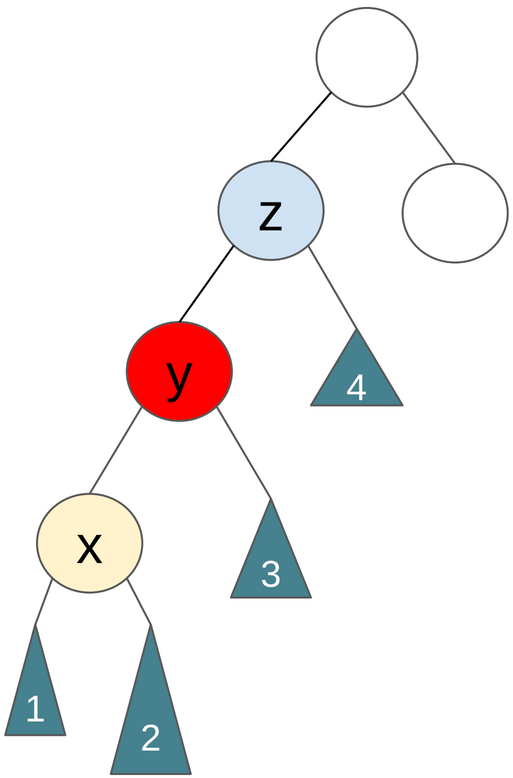
It is also called a **trinode restructuring**

Trinode restructuring: left-heavy subtree



The nodes x, y, z are in increasing order: x < y < z

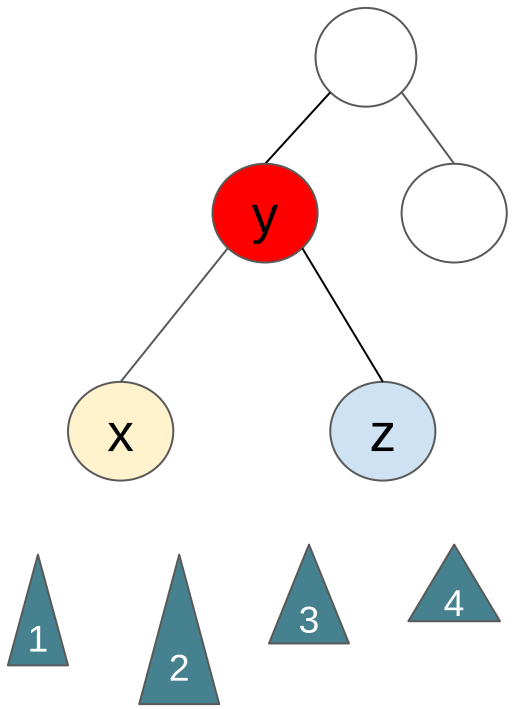
Trinode restructuring: left-heavy subtree



The nodes x, y, z are in increasing order: x < y < z

Pull y to the top and make x its left child and z its right child

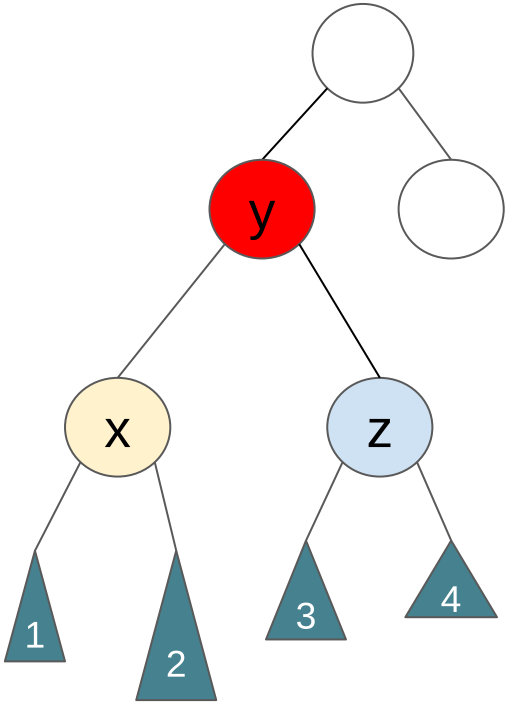
Trinode restructuring: left-heavy subtree



The nodes x, y, z are in increasing order

Pull y to the top and make x its left child and z its right child

General trinode restructuring: left-heavy subtree



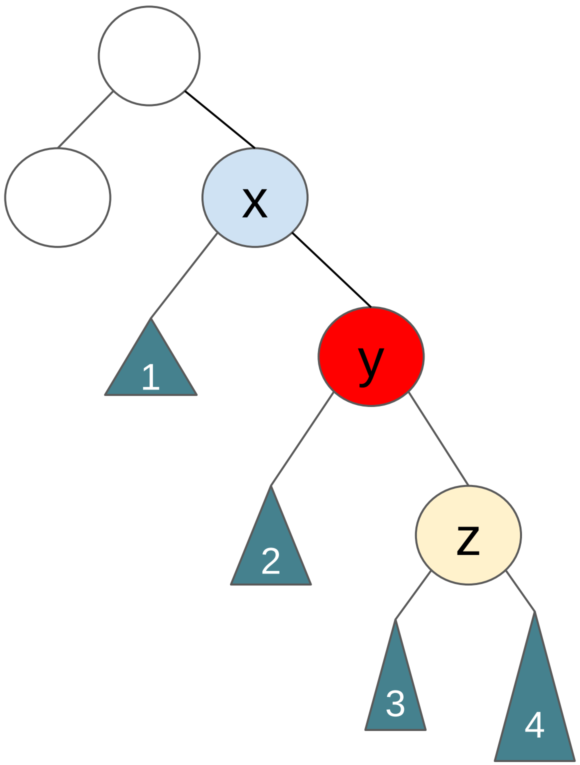
The nodes x, y, z are in increasing order

Pull y to the top and make x its left child and z its right child

Reattach all 4 children (some of them can be NULL) to x and z

The tree is now balanced

Trinode restructuring: right-heavy subtree the same idea

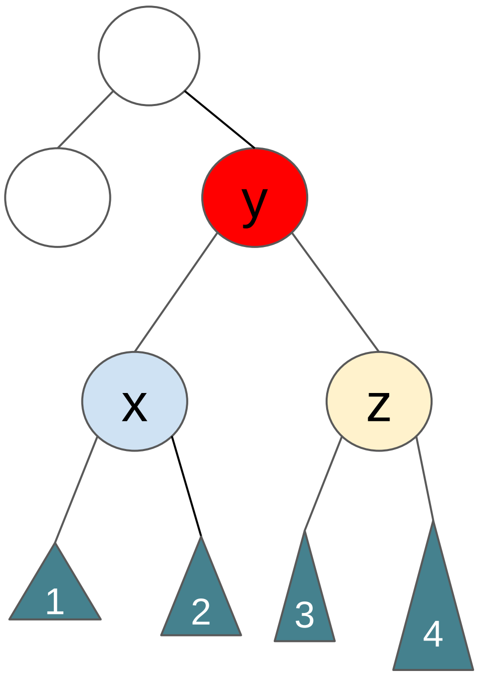


The nodes x, y, z are in increasing order

Pull y to the top and make x its left child and z its right child

Reattach all 4 children (some of them can be NULL) to x and z

Trinode restructuring: right-heavy subtree

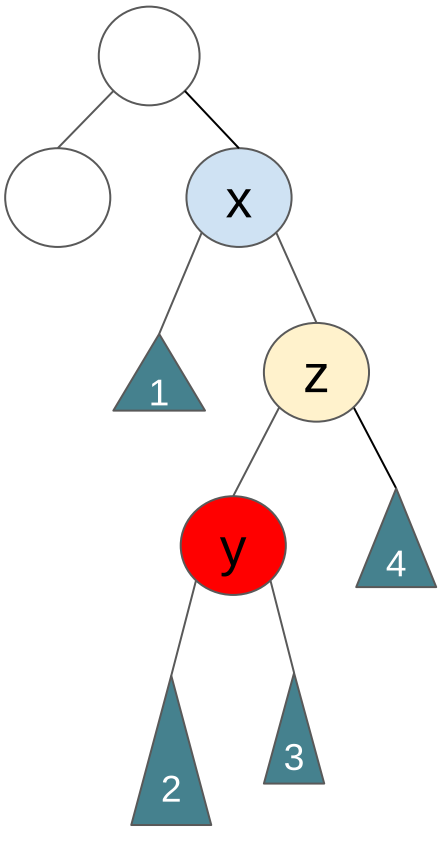


The nodes x, y, z are in increasing order

Pull y to the top and make x its left child and z its right child

Reattach all 4 children (some of them can be NULL) to x and z

Trinode restructuring: right-left-heavy subtree: the same idea

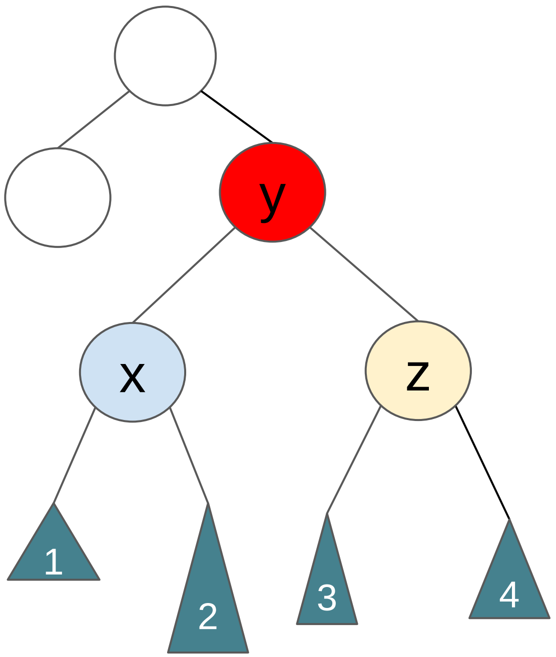


The nodes x, y, z are in increasing order

Pull y to the top and make x its left child and z its right child

Reattach all 4 children (some of them can be NULL) to x and z

Trinode restructuring: right-left-heavy subtree

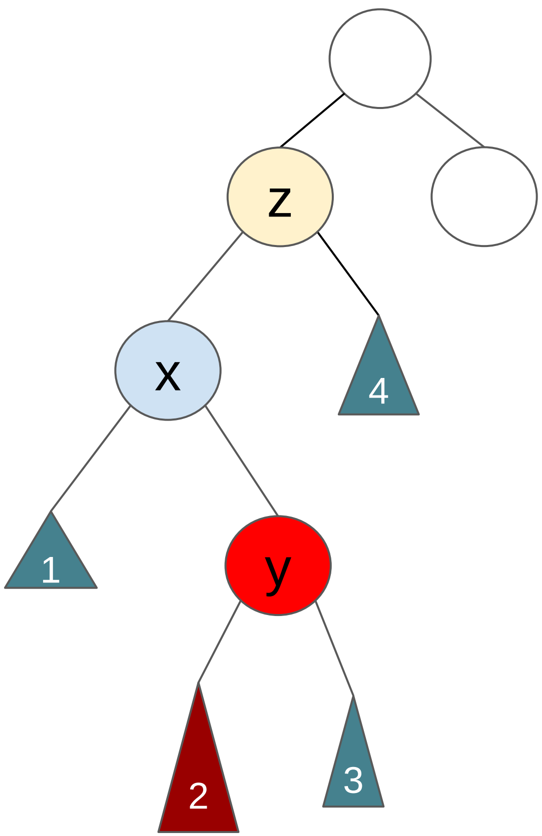


The nodes x, y, z are in increasing order

Pull y to the top and make x its left child and z its right child

Reattach all 4 children (some of them can be NULL) to x and z

Trinode restructuring: left-right-heavy subtree: the same idea

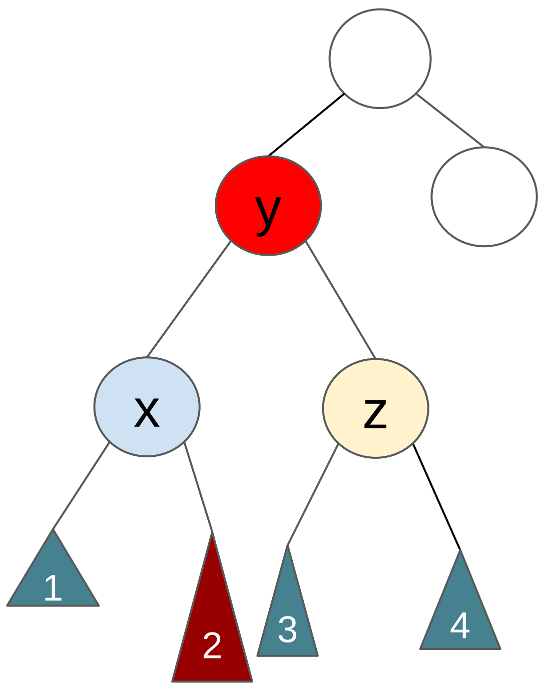


The nodes x, y, z are in increasing order

Pull y to the top and make x its left child and z its right child

Reattach all 4 children (some of them can be NULL) to x and z

Trinode restructuring: left-right-heavy subtree



The nodes x, y, z are in increasing order

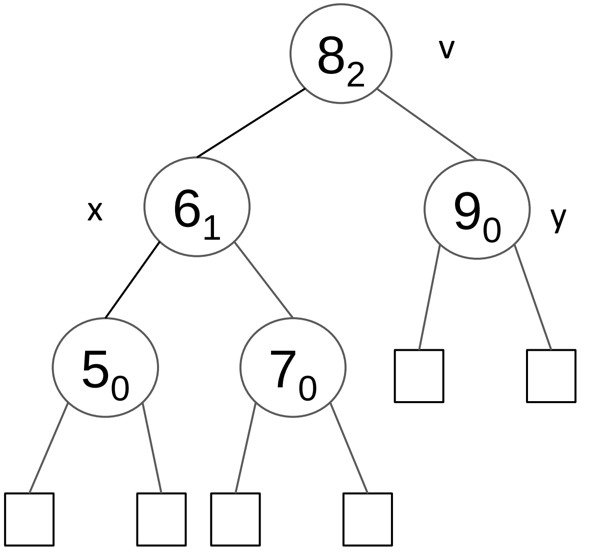
Pull y to the top and make x its left child and z its right child

Reattach all 4 children (some of them can be NULL) to x and z

AVL trees*

Definition

AVL tree is a Binary Search Tree with the following property: for every internal node *v* in AVL tree, the heights of the children of *v* differ by at most 1

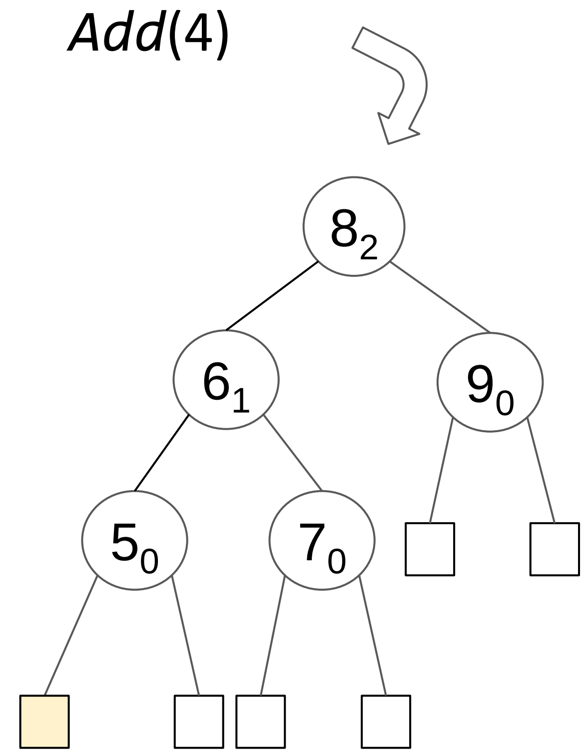


I.e. if the children of *v* are *x* and *y*, then $|h(x) - h(y)| \leq 1$

*Named after inventors **A**delson-**V**elsky and **L**andis

AVL tree: insertion

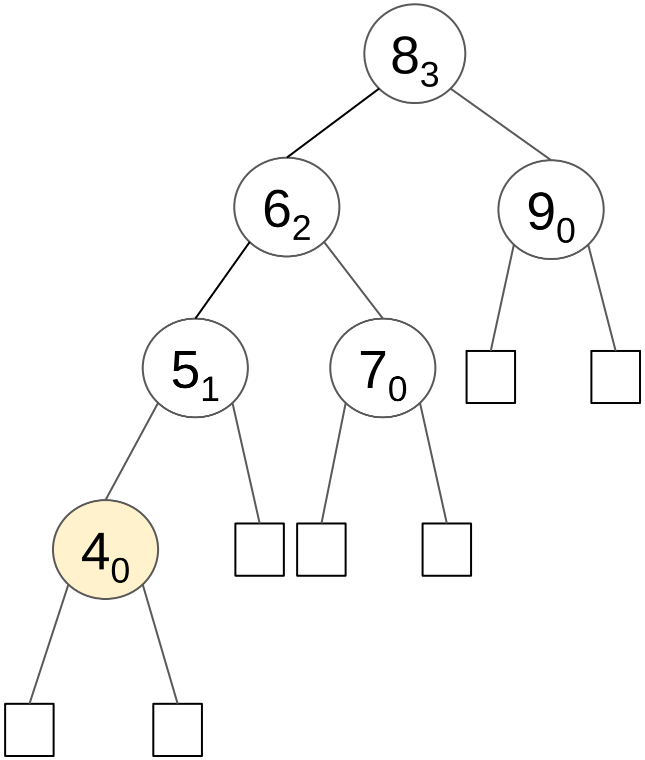
First, we perform regular insertion into BST and end up filling up one of the NULL nodes with the new value



AVL tree: insertion

External node becomes a new internal node

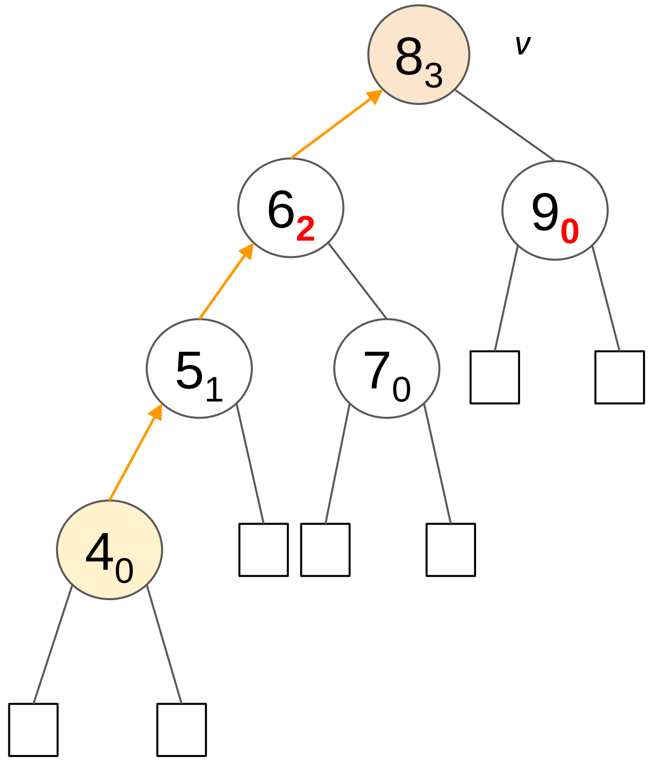
After the insertion, some internal nodes may become unbalanced



AVL tree: rebalancing after insertion

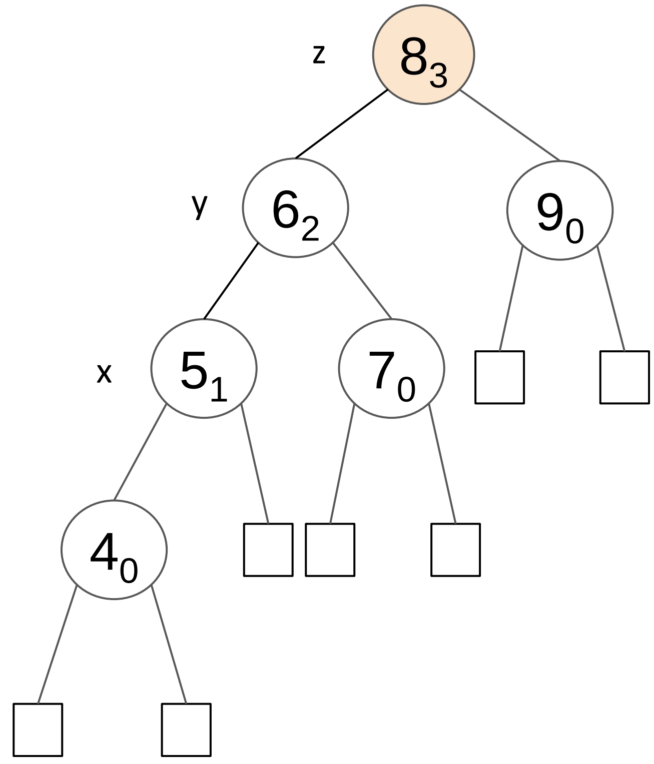
We can go up from the inserted node until we encounter the first unbalanced node v

Note that in order for a branch to become heavy, there must be at least 2 real nodes on this branch (think why one node is not enough)



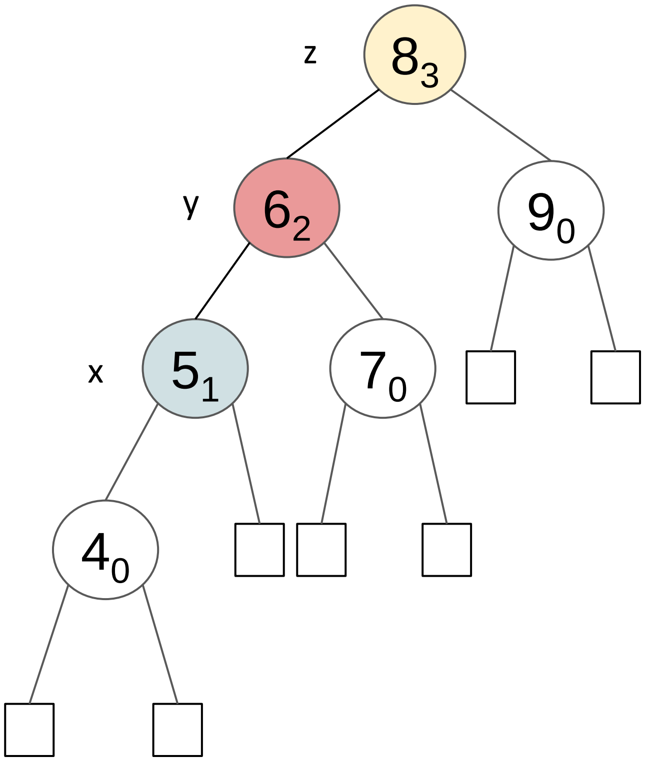
AVL tree: rebalancing after insertion

We keep track of the first unbalanced node *v* and the 2 nodes encountered before we reach *v*, and we name them according to their relative order as *x*, *y*, *z*



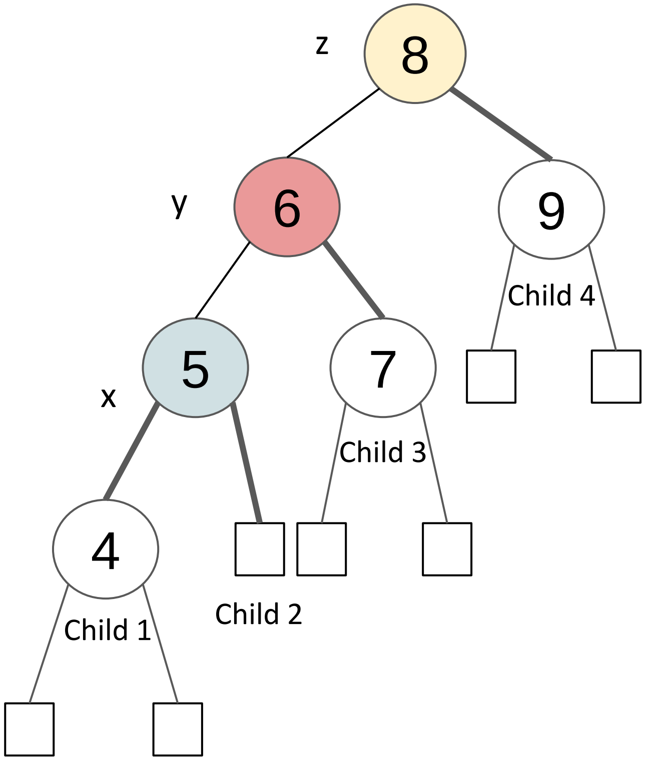
AVL tree: rebalancing after insertion

We then perform a rotation moving *y* on top of *x* and *z* - according to trinode restructuring rules



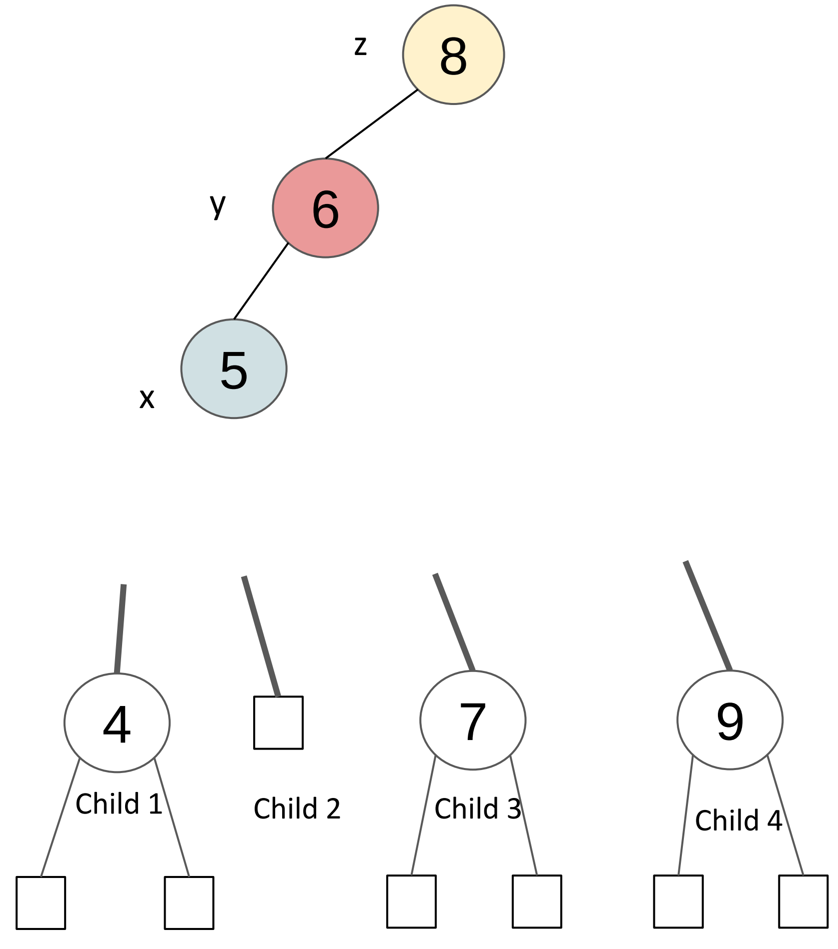
AVL tree: rebalancing after insertion

Detach 4 children of *x*, *y*, *z*



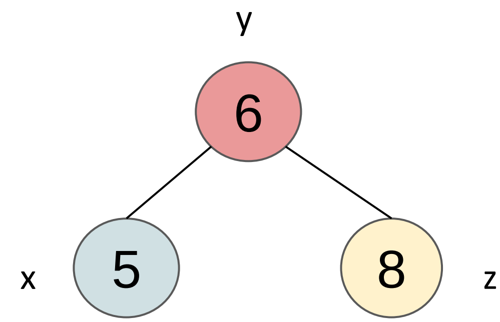
AVL tree: rebalancing after insertion

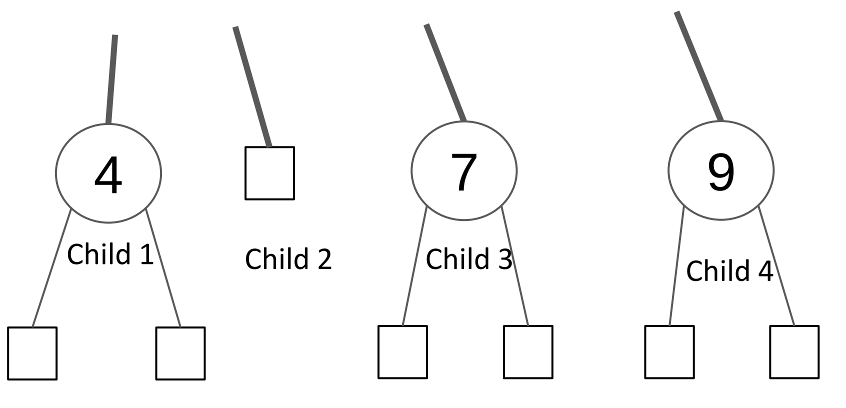
Detach 4 children of *x*, *y*, *z*



AVL tree: rebalancing after insertion

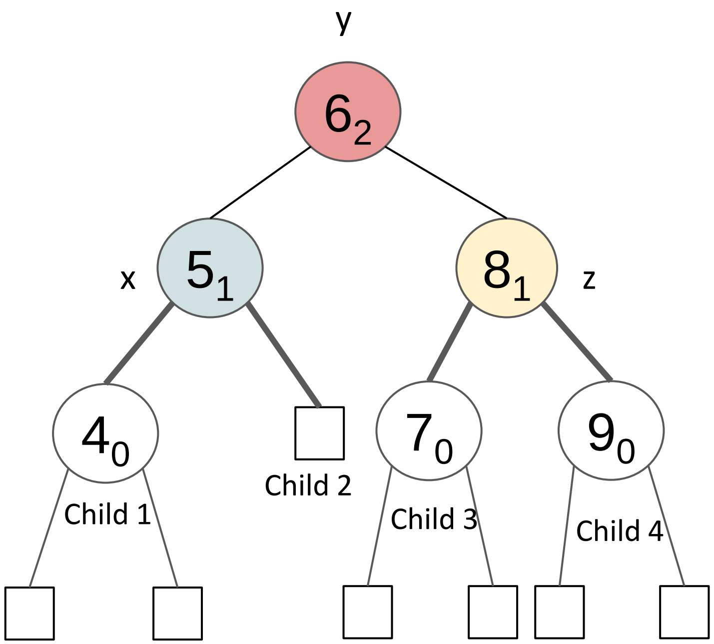
Perform rotation



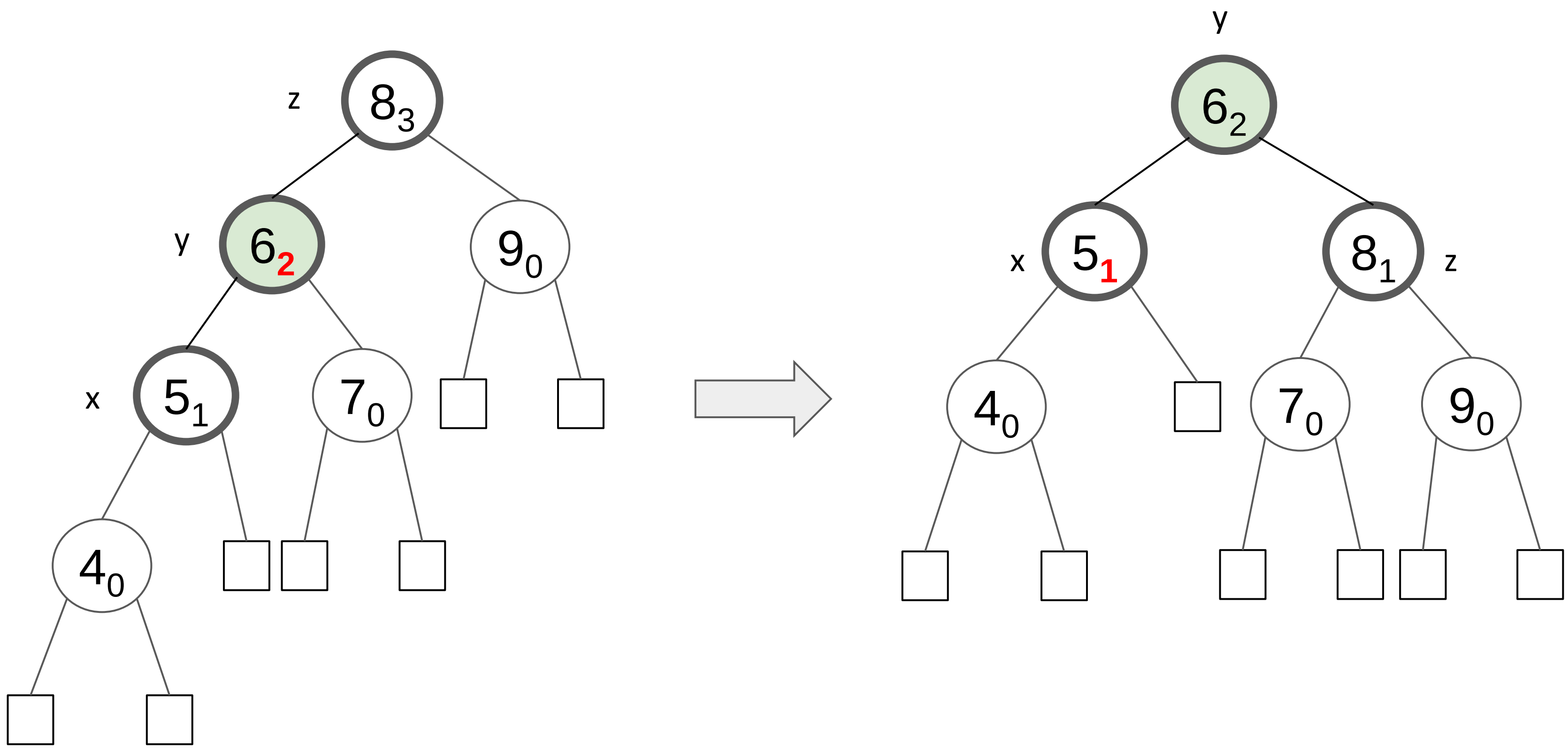


AVL tree: rebalancing after insertion

Reattach children

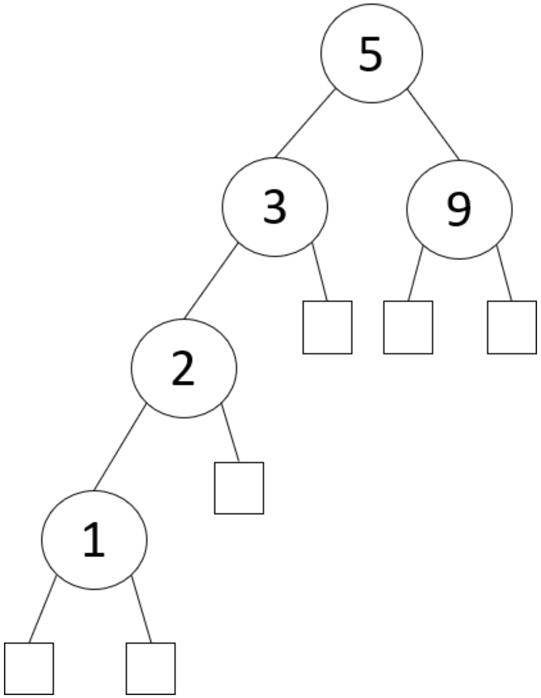


AVL tree: insertion summary



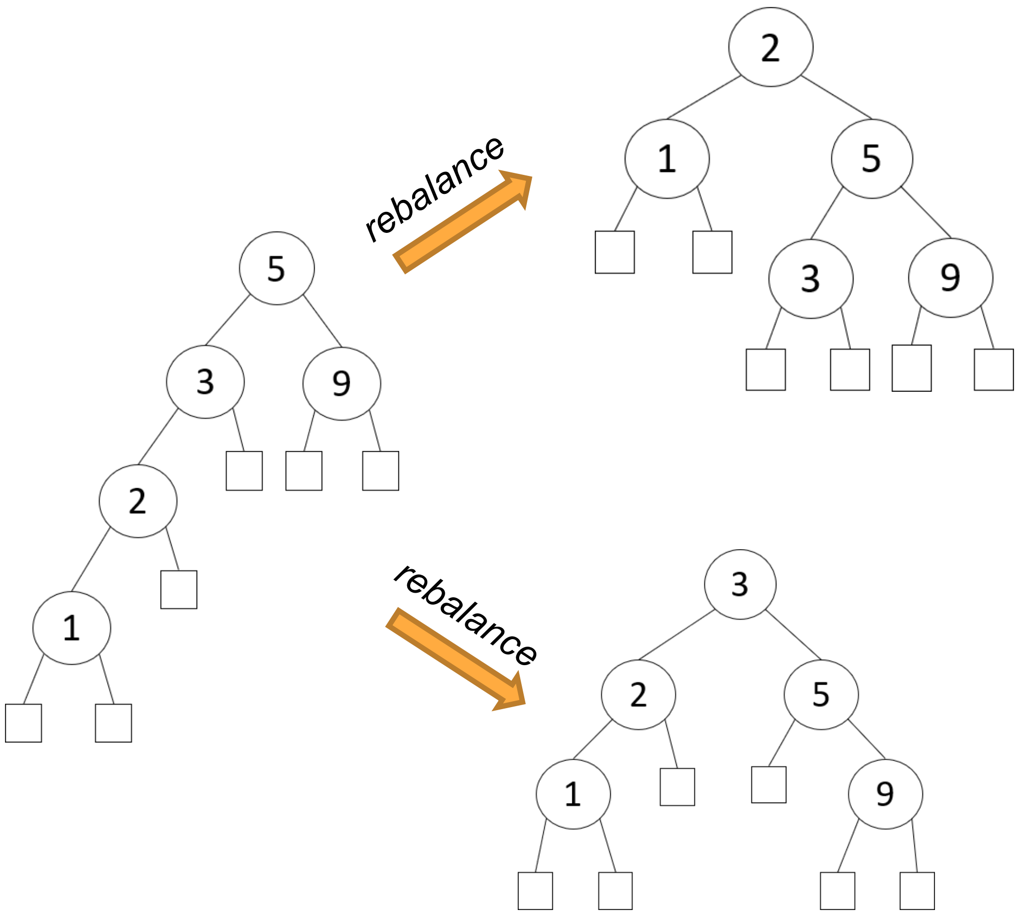
The rebalancing is local and involves only x, y, z - thus in constant time
The heavier subtree height is reduced by 1 - restoring AVL property for the parent node

Is this tree balanced?



A. The tree is balanced
B. The tree is unbalanced because of node 2
C. The tree is unbalanced because of node 3
D. The tree is unbalanced because of node 5
E. More than one unbalanced node

Which tree is the result of rebalancing the tree on the left?



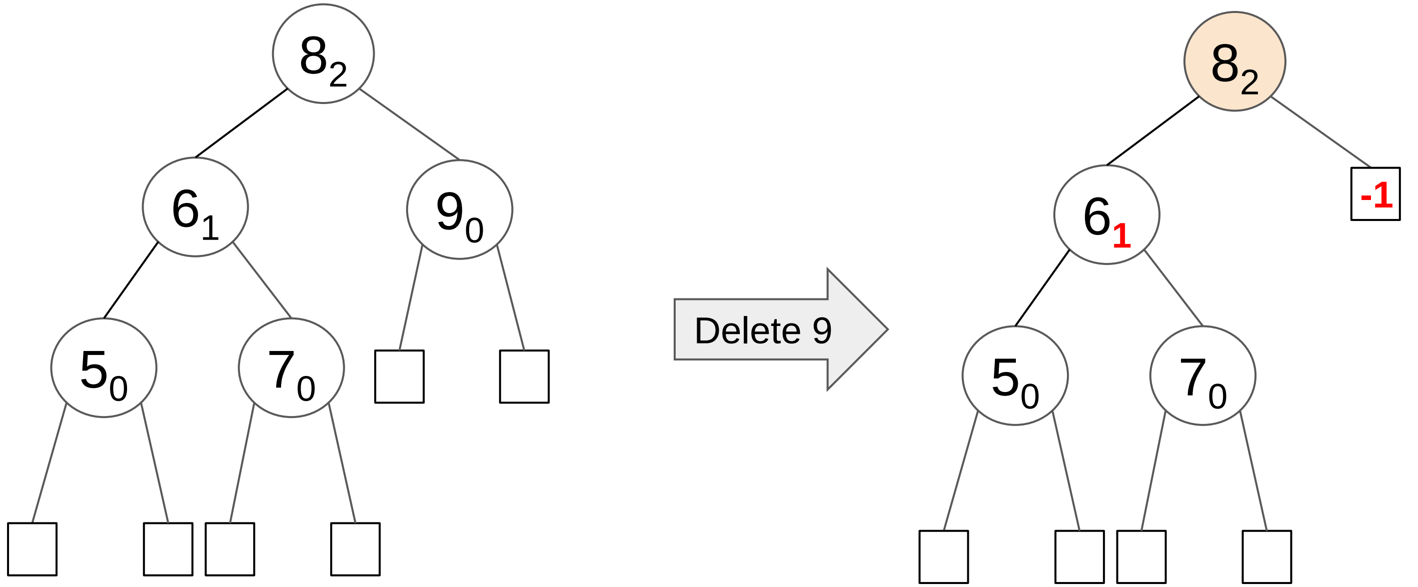
A

B

C. None is correct

D. Both are possible

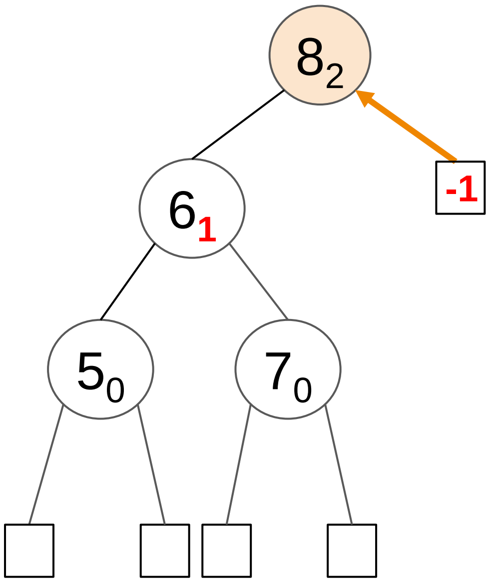
AVL tree: deletion - similar idea



By removing a node from AVL tree some nodes may become unbalanced
But this time the branch from which the node was removed becomes lighter than its sibling
We need to restructure the heavier sibling to reduce its height

AVL tree: rebalancing after deletion

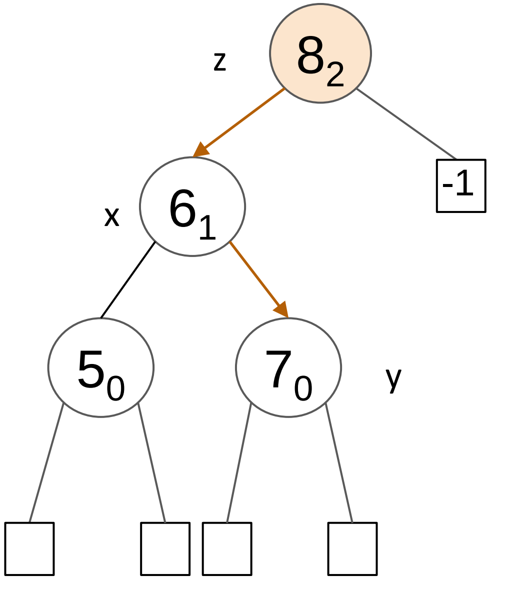
We move up the tree from the current NULL node until we encounter an internal node which is unbalanced



AVL tree: rebalancing after deletion

Then we move into the heavier subtree choosing the child with the larger height

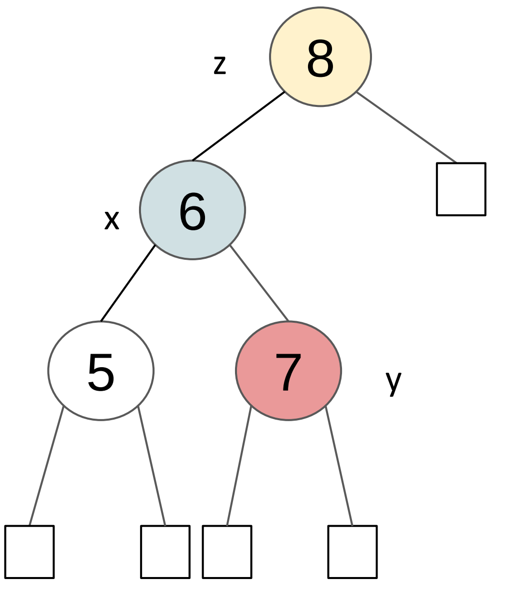
We produce 3 nodes x < y < z to be restructured



AVL tree: rebalancing after deletion

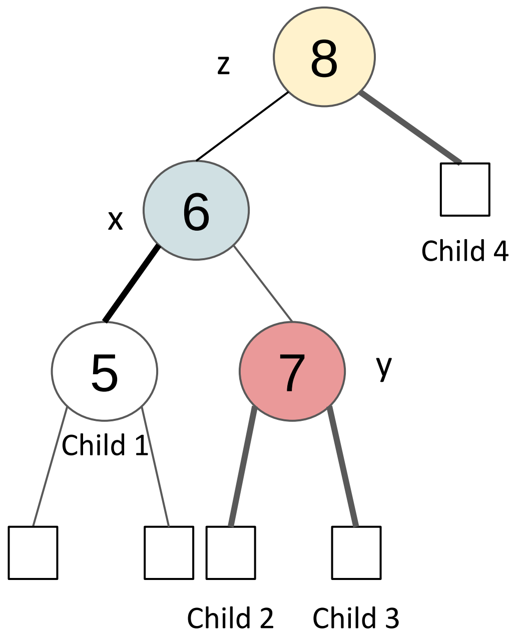
We perform rotation around y

This is accomplished with trinode restructuring as before



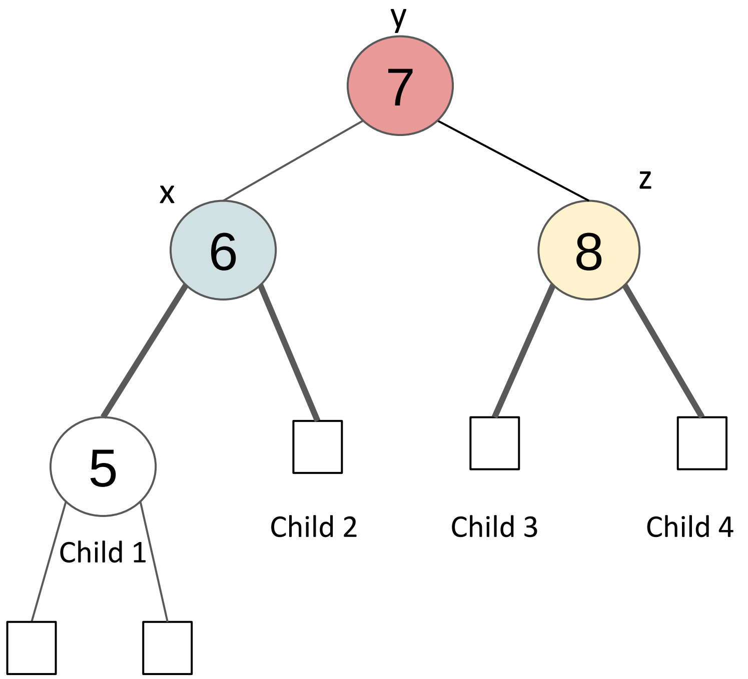
AVL tree: rebalancing after deletion

Trinode restructuring: detach children of x, y, z

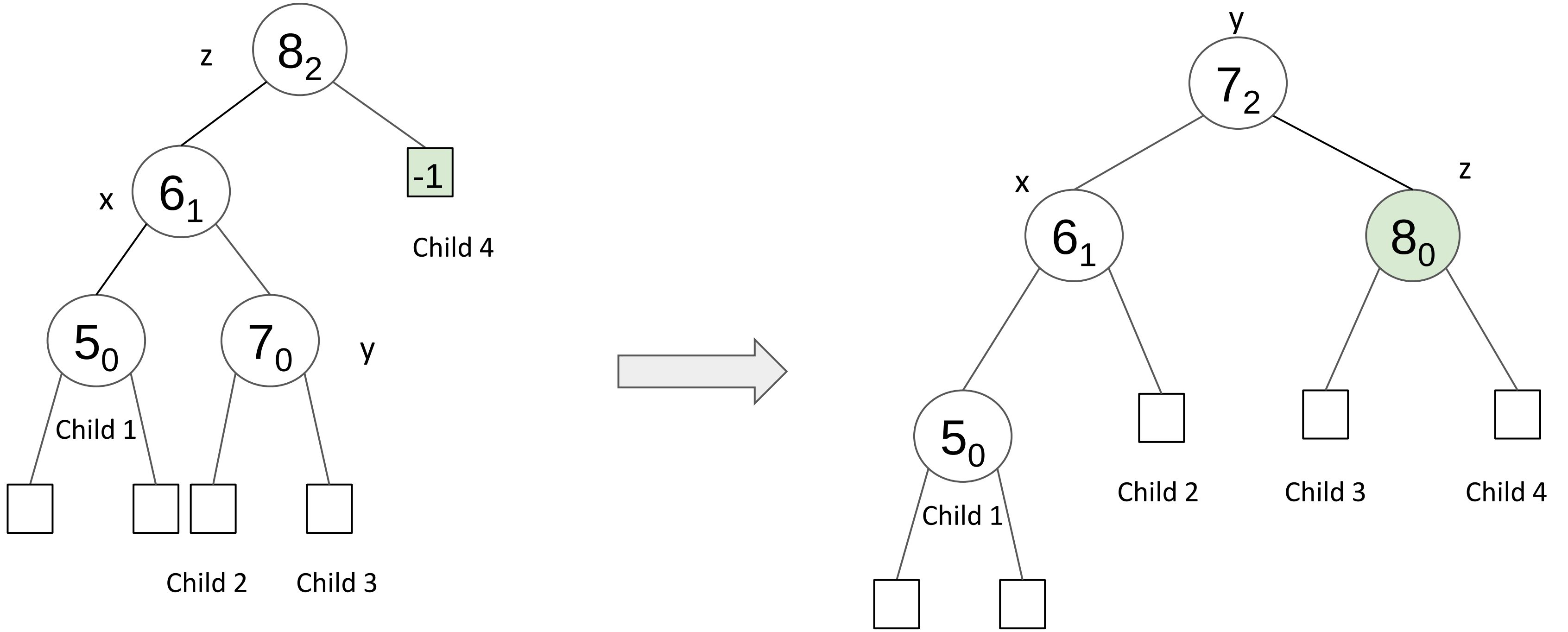


AVL tree: rebalancing after deletion

Move y on top and reattach 4 children



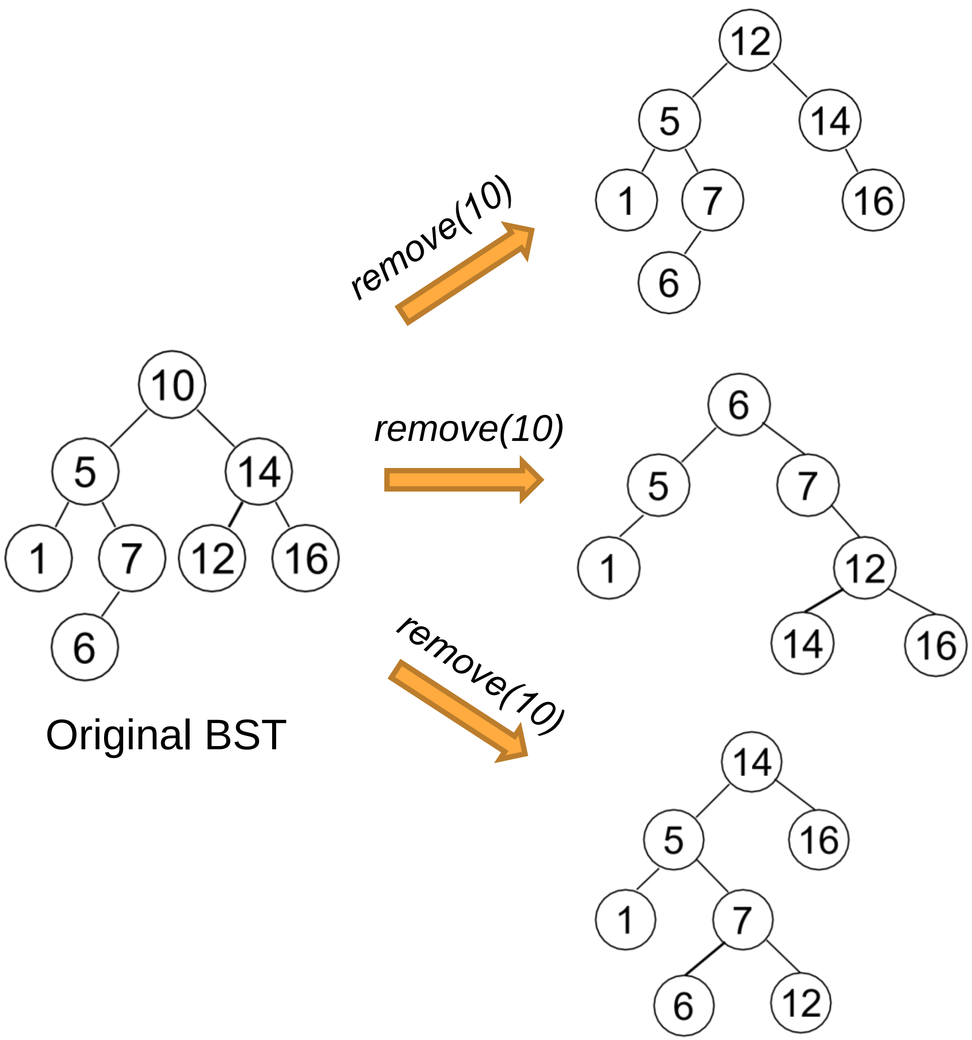
AVL tree: rebalancing after deletion



We fixed the imbalance in left subtree by increasing the height of the right child of the root by 1

Remove (10)

Which tree represents the result of deleting the node with key 10 from the tree below?



A

B

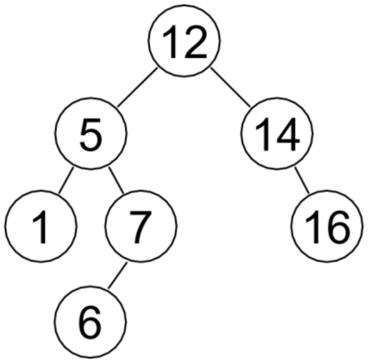
C

D. None of the above

E. More than one is correct

Is the result balanced?

Is the resulting BST balanced?



Resulting BST

A. The tree is balanced
B. Tree is unbalanced because node 5 is unbalanced
C. Tree is unbalanced because node 12 is unbalanced
D. Tree is unbalanced because node 14 is unbalanced
E. None of the above (something else?)