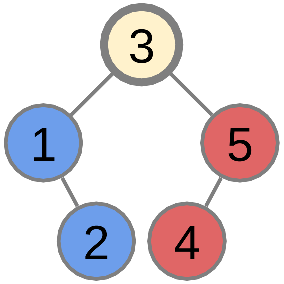
Maintaining Balance: Balanced Binary Search Trees (BBST)

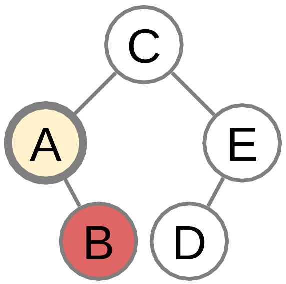
[Review 02.07]

By Marina Barsky

Recap: Definition

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





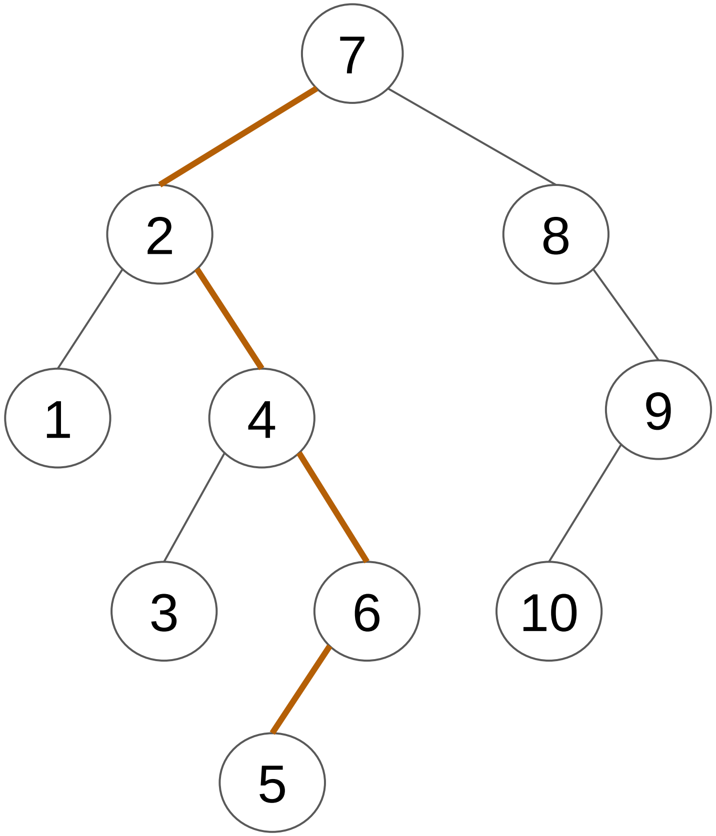
Recap: Operations on BST

- **Find (*k*)**
- **Successor (*k*)/Predecessor (*k*)**
- **Insert (*k*)**
- **Delete (*k*)**

How long does each operation take?

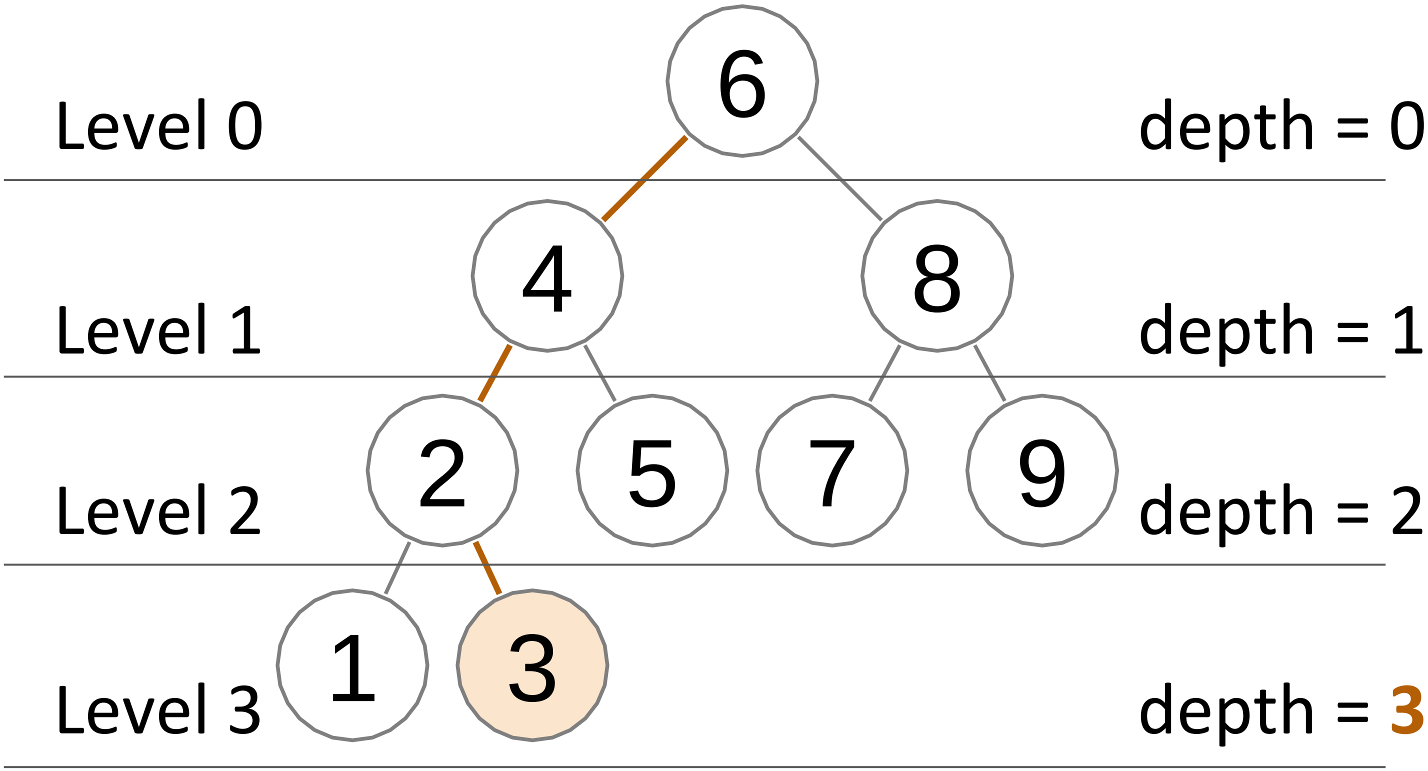
Example: find (*k*)

find (5)



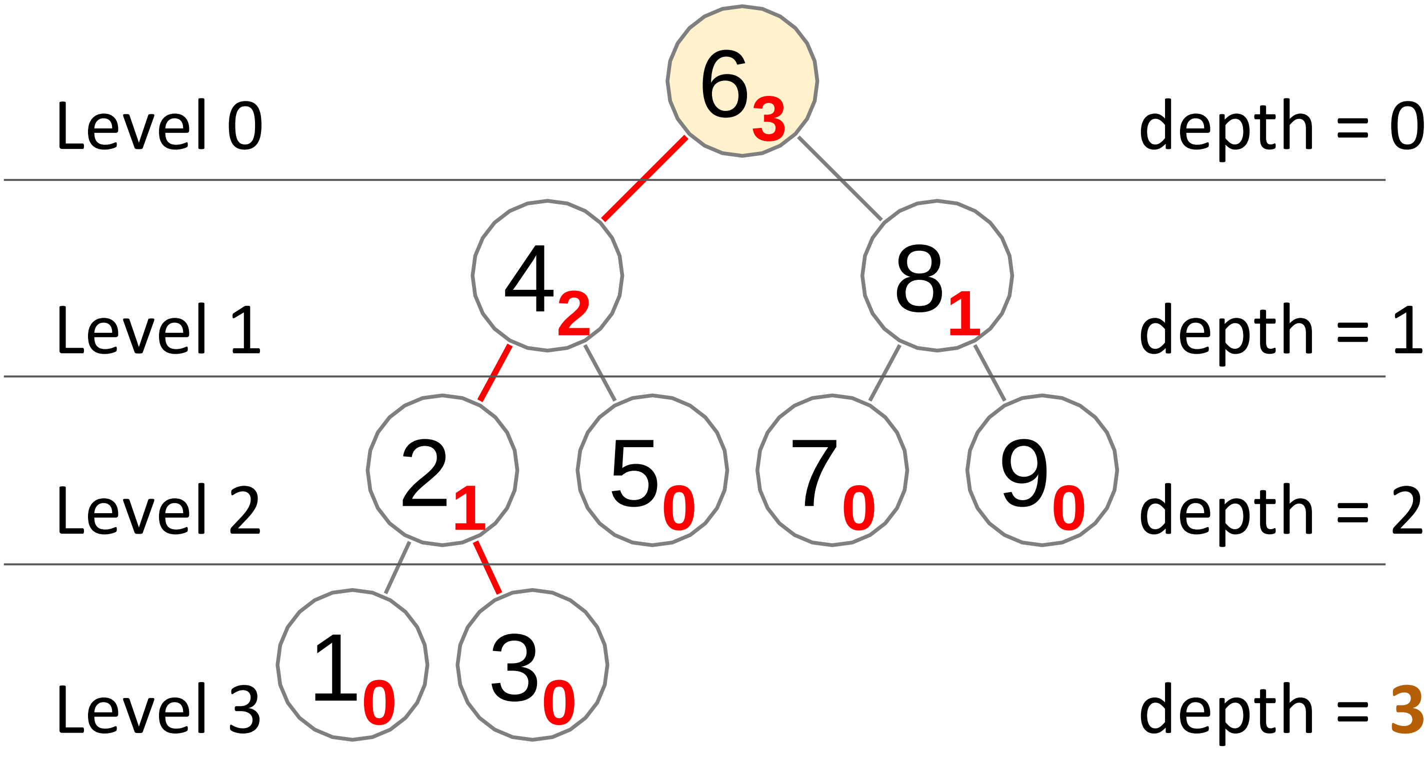
Total questions asked before we reach 5: **4**

Recap: node depth



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

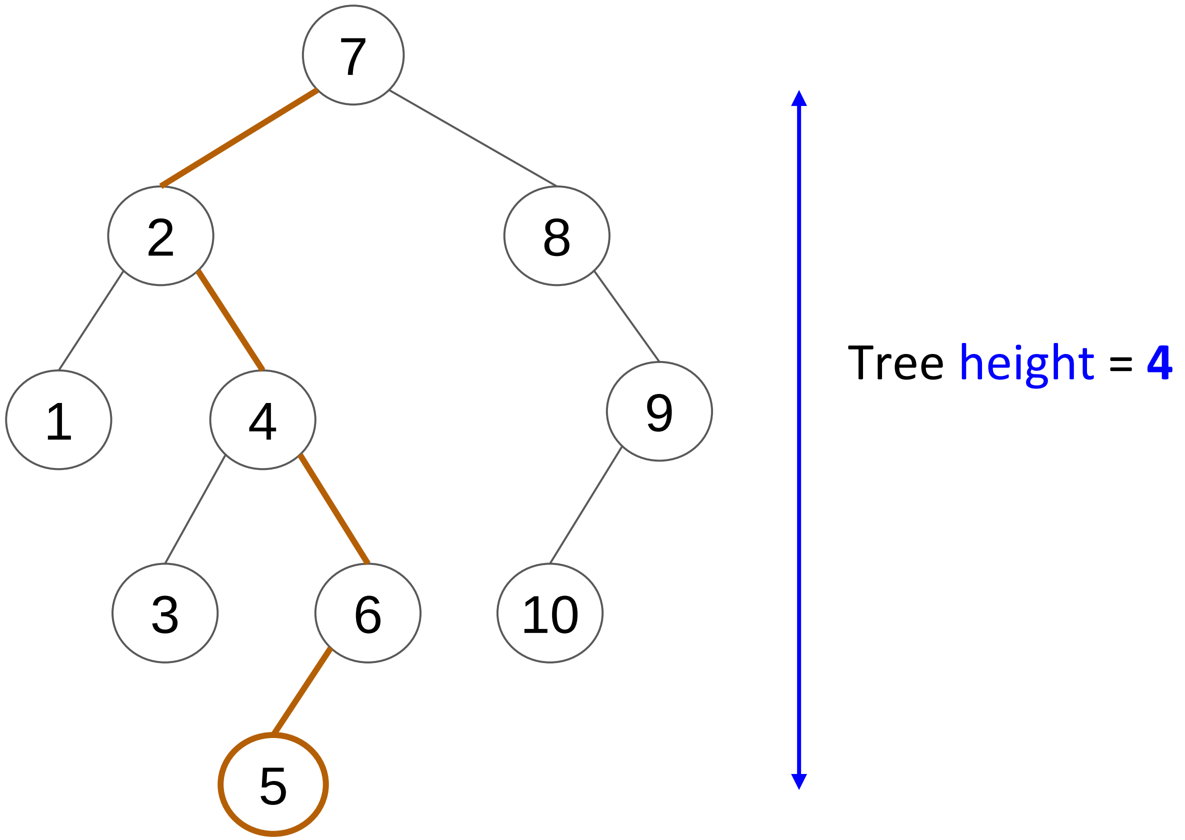
Recap: height of a subtree rooted at node *v*



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

Complexity: find (*k*)

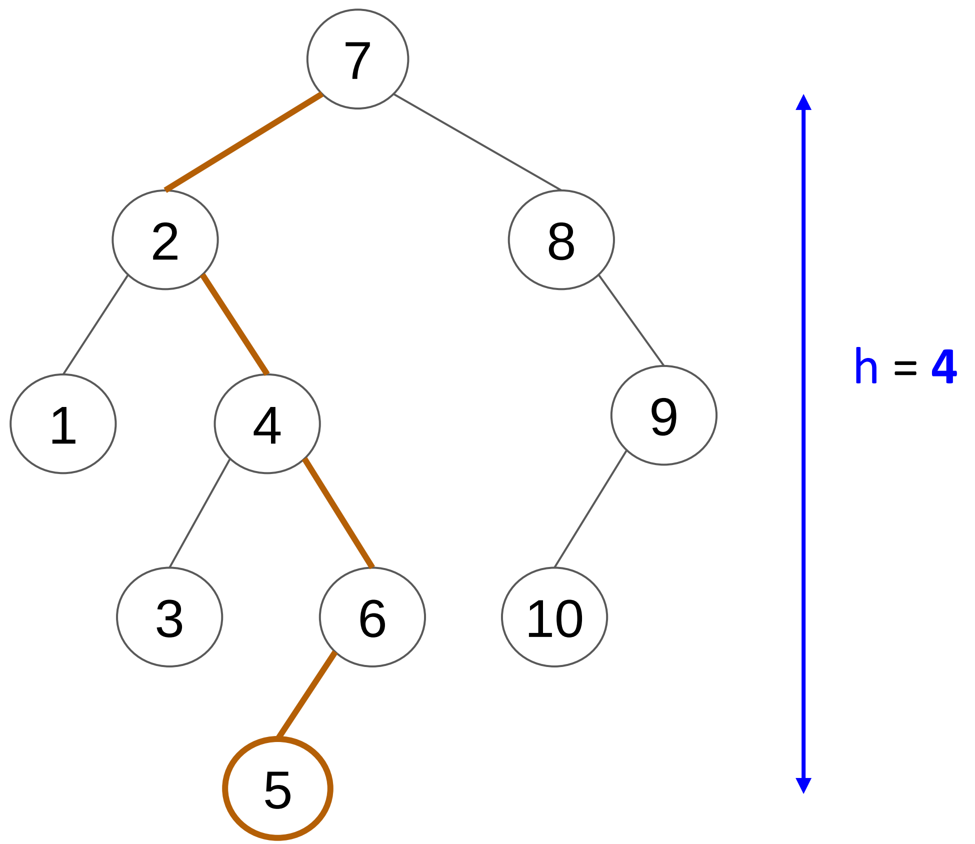
`find` (5)



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

Complexity: find (*k*)

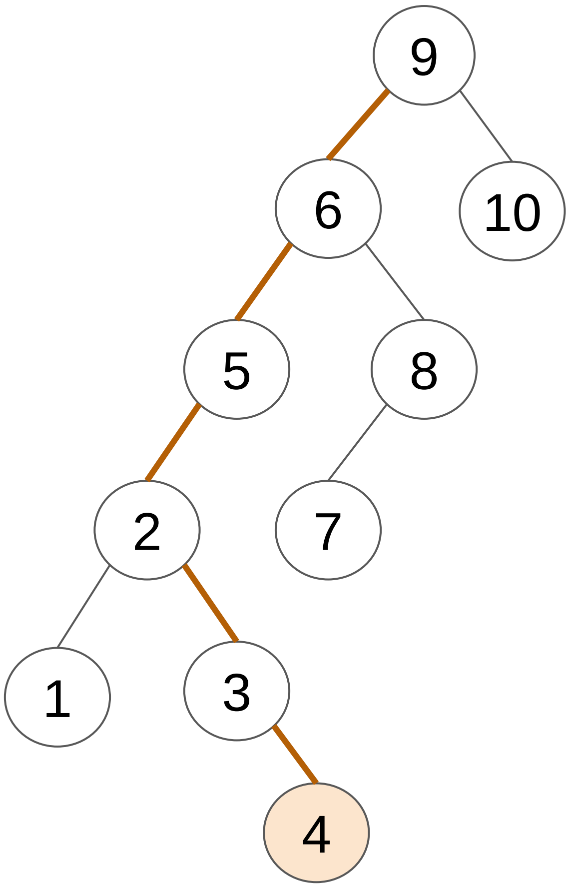
`find` (5)



- 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*)

`find` (4)



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

We could do O(n) before:

Sorted Array

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

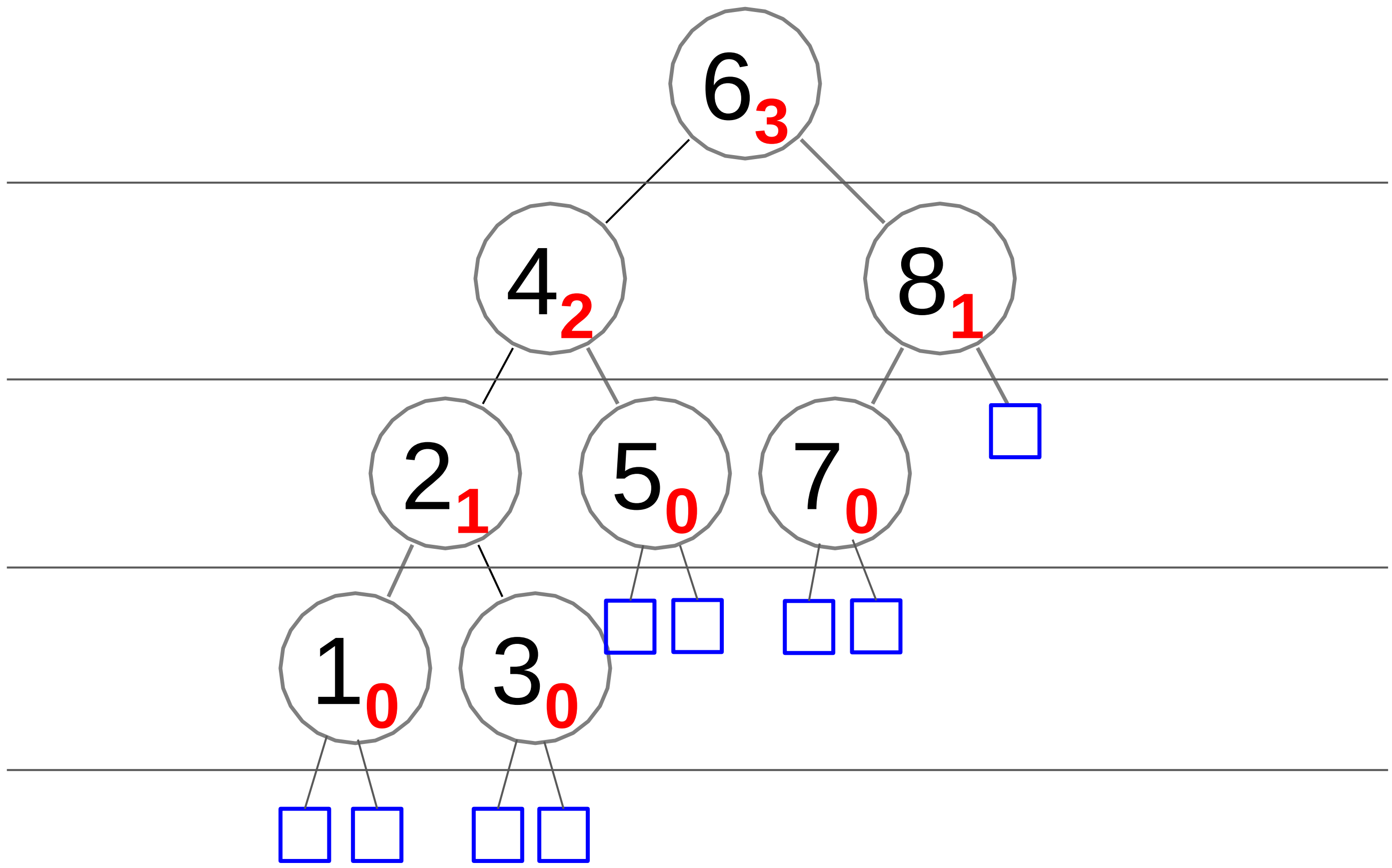
Sorted Linked List

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

In search for balance

- The Binary Search Tree of *n* nodes which has height O(log *n*) is called a ***Balanced Binary Search Tree*** (**BBST**)
- To achieve O(log *n*) time on all operations - we need to keep our tree **balanced**
- We will perform local restructuring of the nodes to always keep the height of the tree O(log *n*)

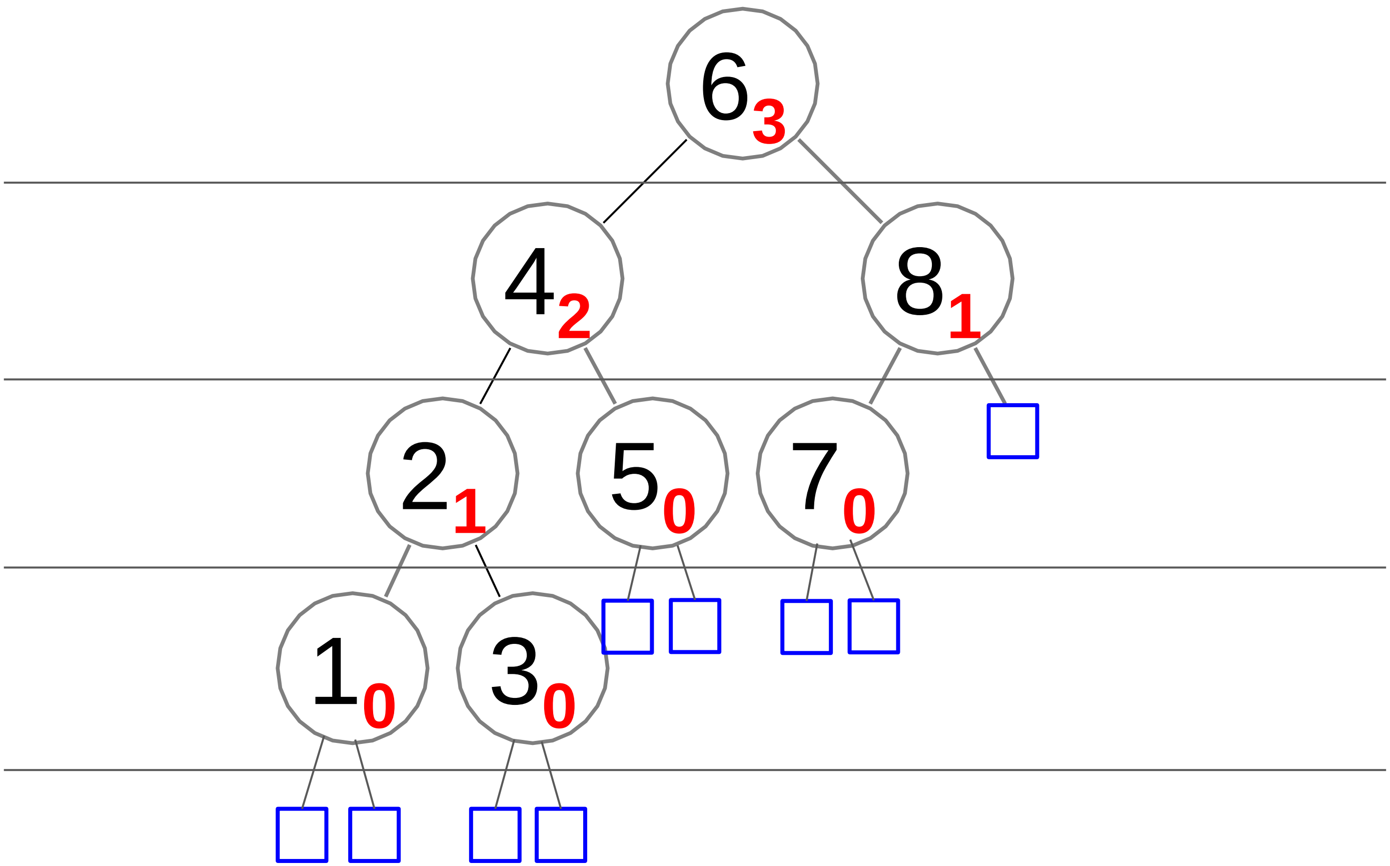
Completing the tree



To make it easier to argue about balance - let's make **each** BST node have **exactly 2 children**

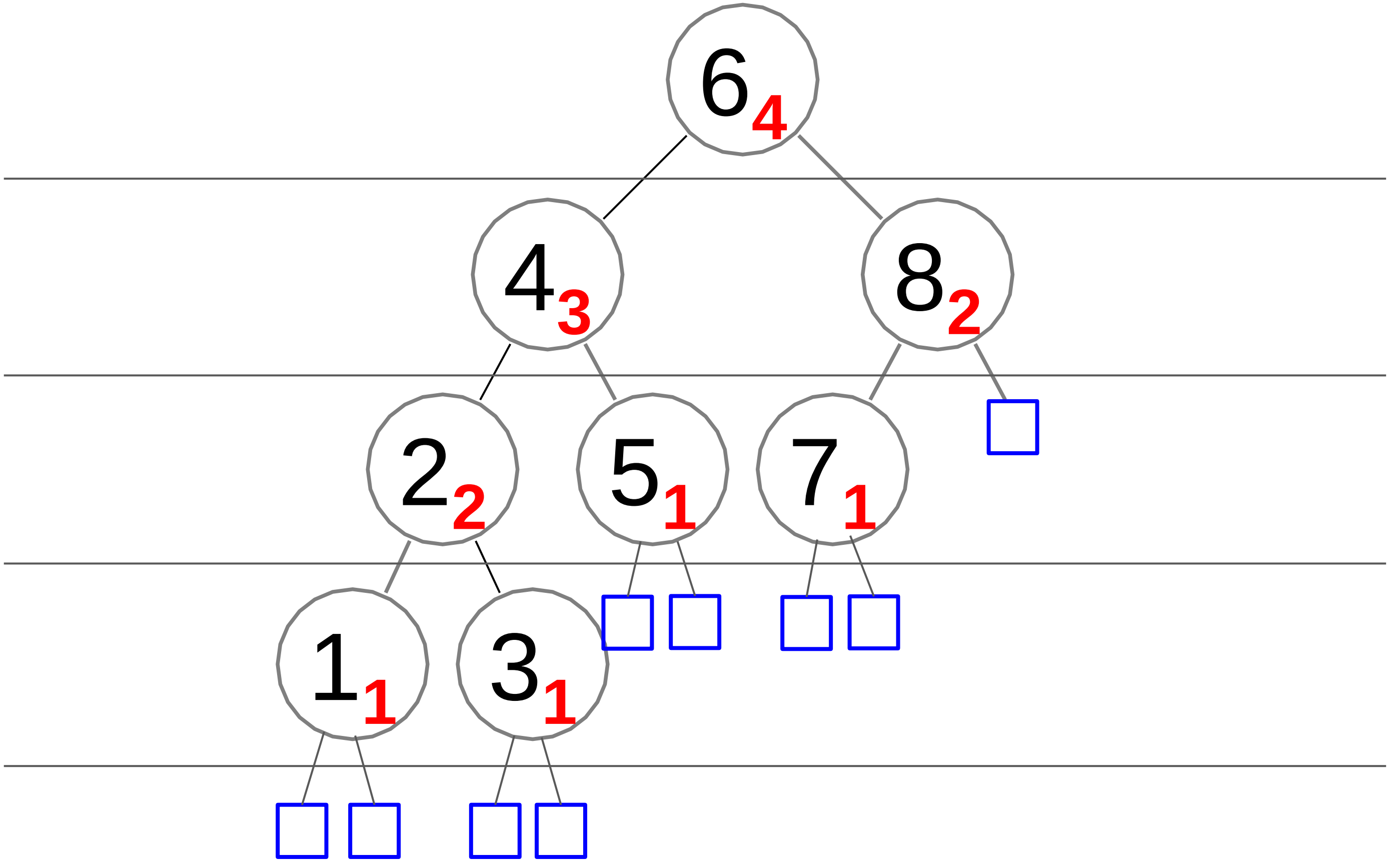
If there is no left/right child - we add a special NULL node

Redefining internal nodes



Now each node that stores a key becomes an ***internal*** node
And each ***external (leaf)*** node is a NULL node which does not store a key

The height also changes accordingly



Each external node has height 0

Defining balance

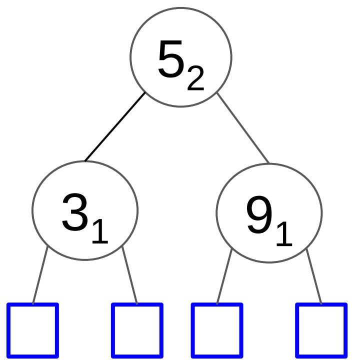
- One possible definition:

  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 $|h(x) - h(y)| \leq 1$.

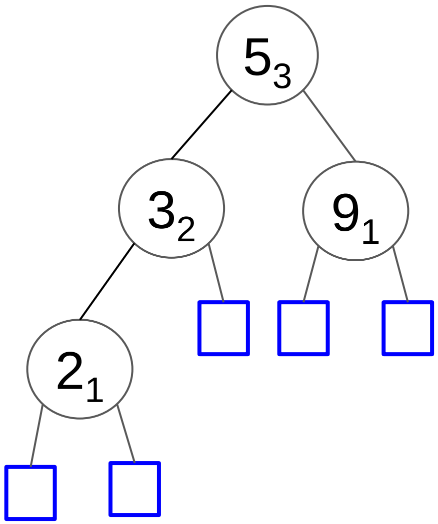
- That implies that we must keep track of the current height for each node of the BBST

How the balance can be destroyed



We start with a perfectly balanced tree

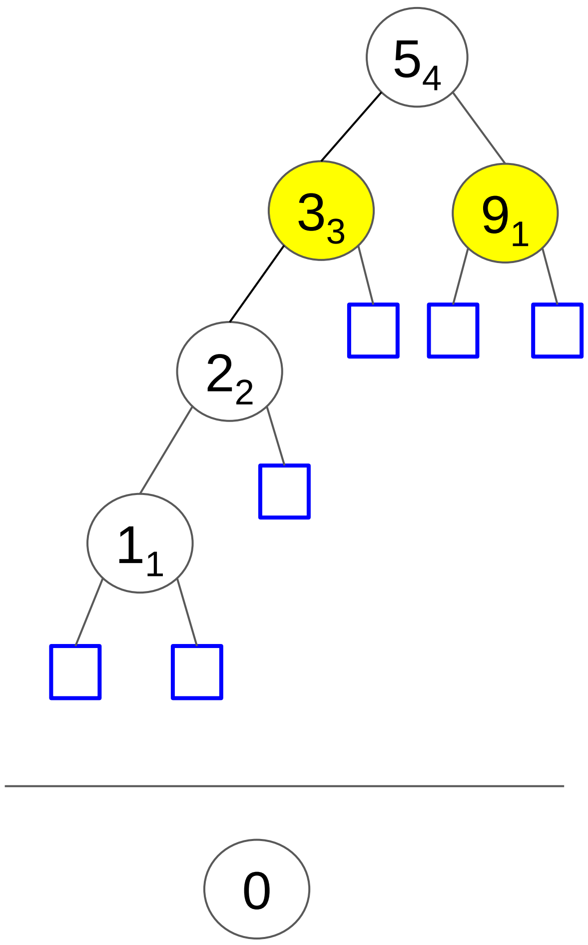
How the balance can be destroyed



We insert key 2

The tree is still balanced

How the balance can be destroyed

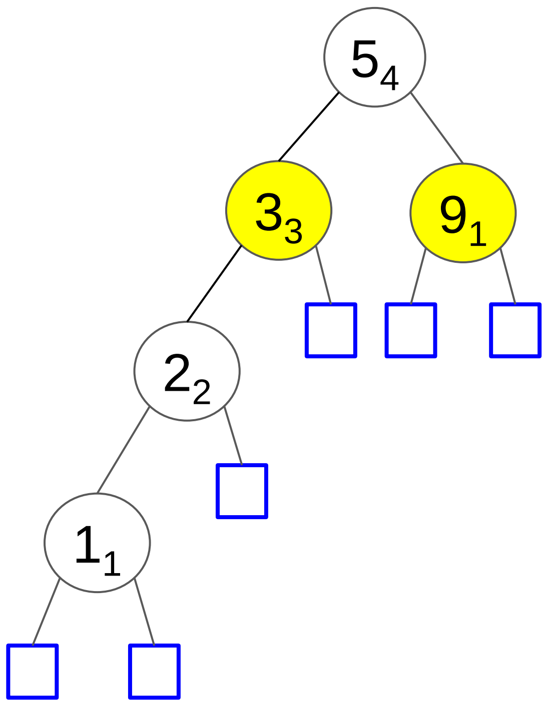


We insert key 1

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

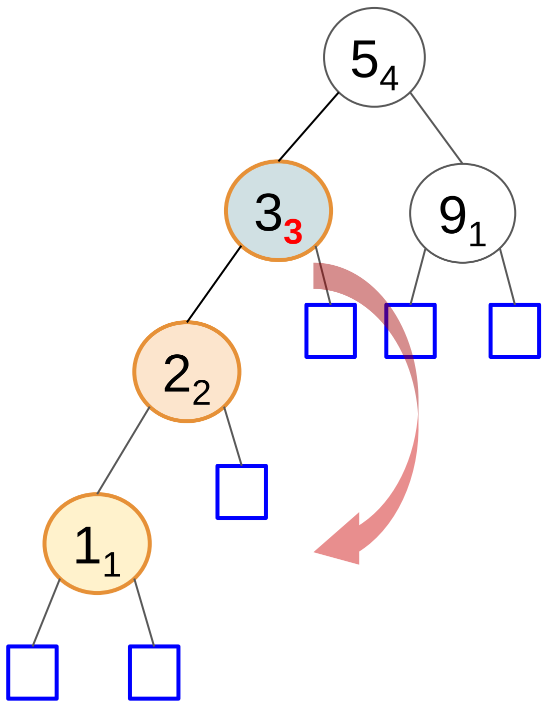
How the balance can be destroyed



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



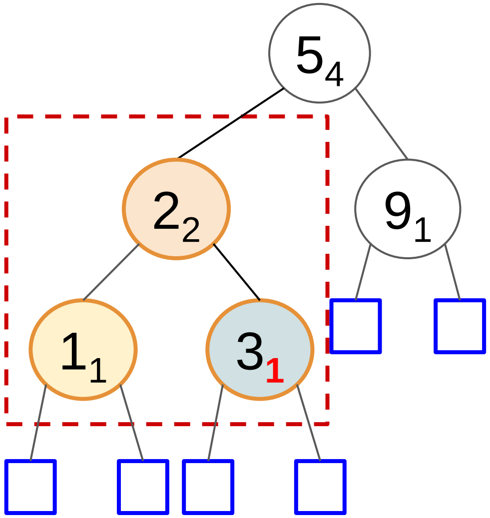
The imbalance in this case is caused by the newly added node **1** and is presented by the path 1, 2, 3 (3 being the first imbalanced node on this path)

We need to rearrange nodes 1,2,3

They all can be left 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

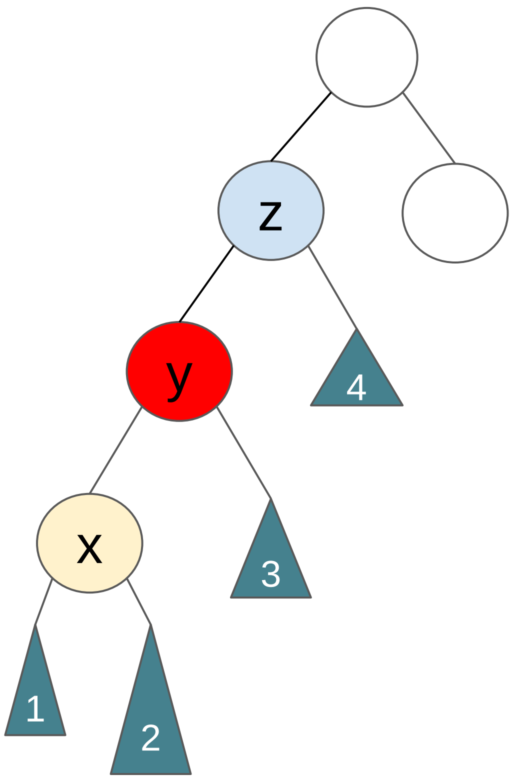
Rebalancing: rotation



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

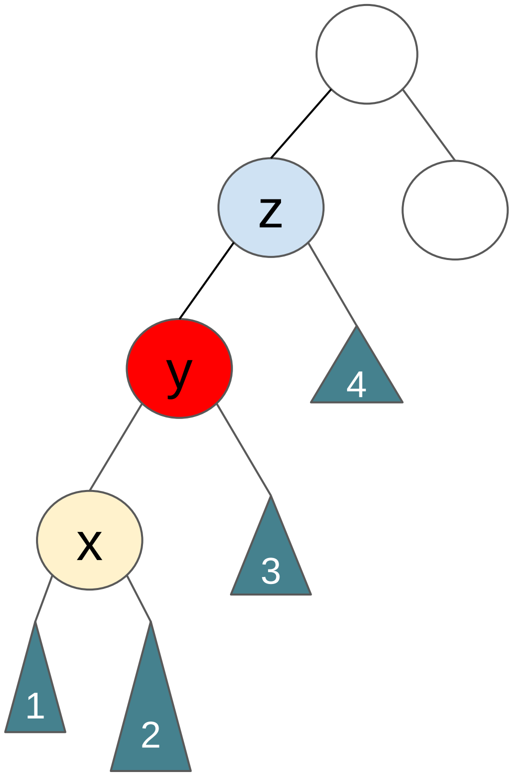
It is also called a **trinode restructuring**

Trinode restructuring: left-heavy subtree



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

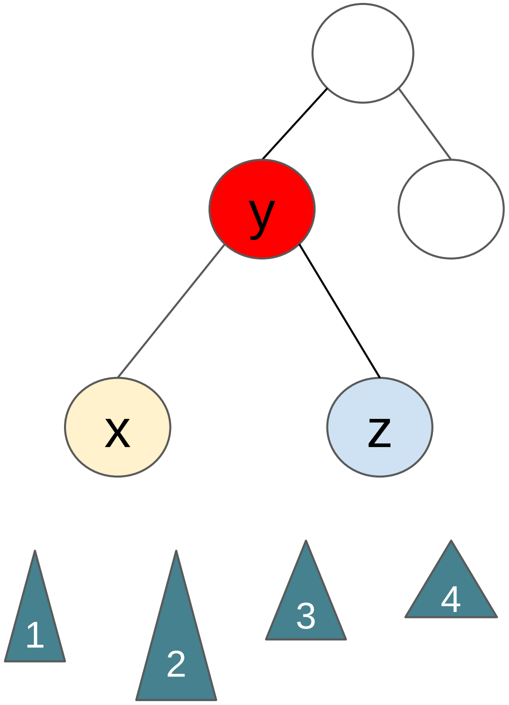
Trinode restructuring: left-heavy subtree



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

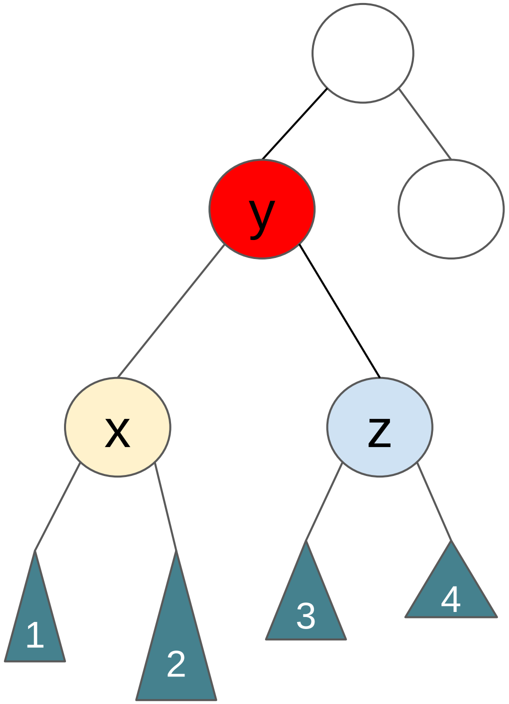
Trinode restructuring: left-heavy subtree



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

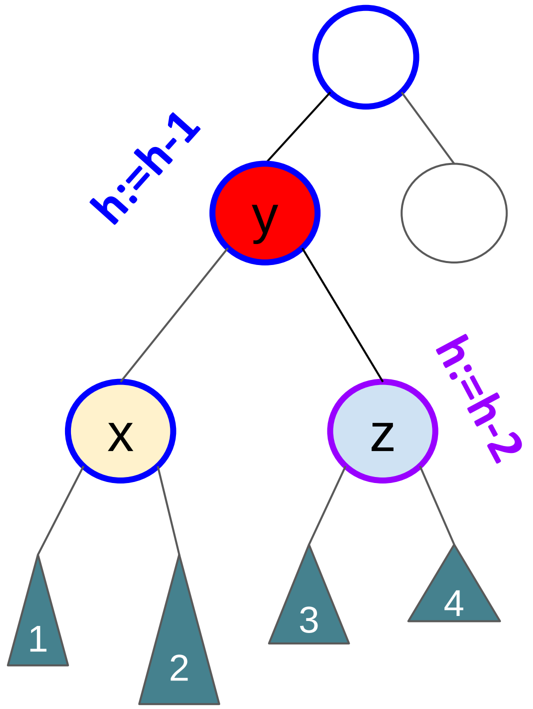


The nodes x, y, z are in increasing order

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

General trinode restructuring: left-heavy subtree



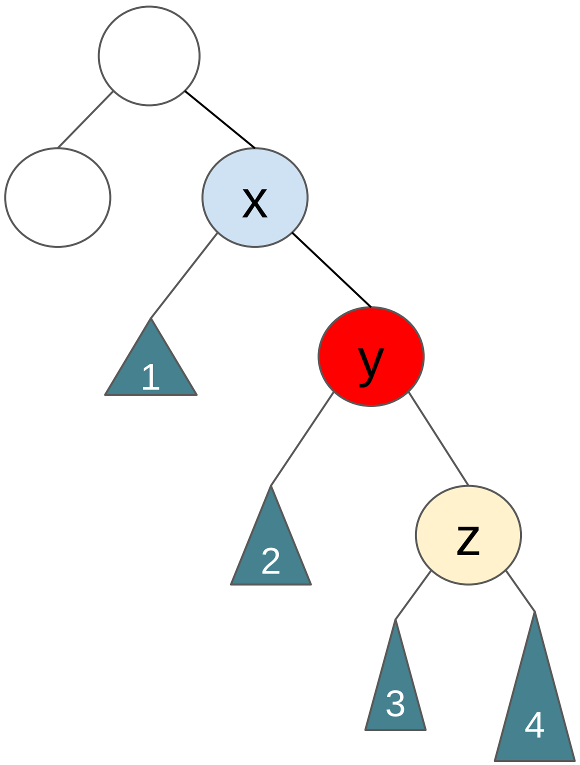
The nodes x, y, z are in increasing order

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

Update heights (locally: in constant time)

Trinode restructuring: right-heavy subtree the same idea



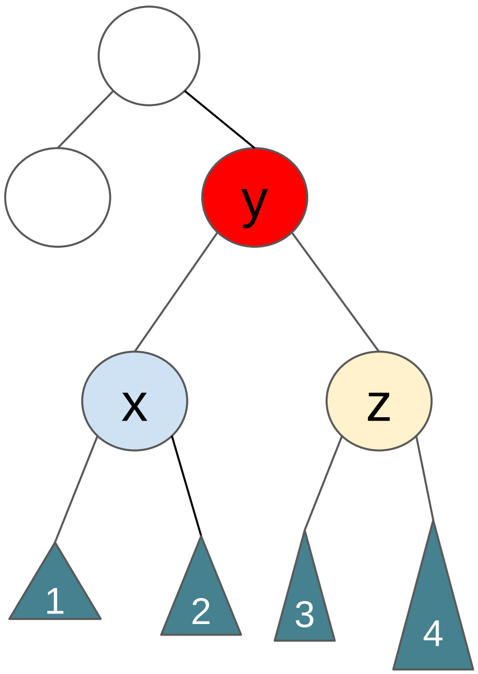
The nodes x, y, z are in increasing order

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

Update heights along this branch

Trinode restructuring: right-heavy subtree



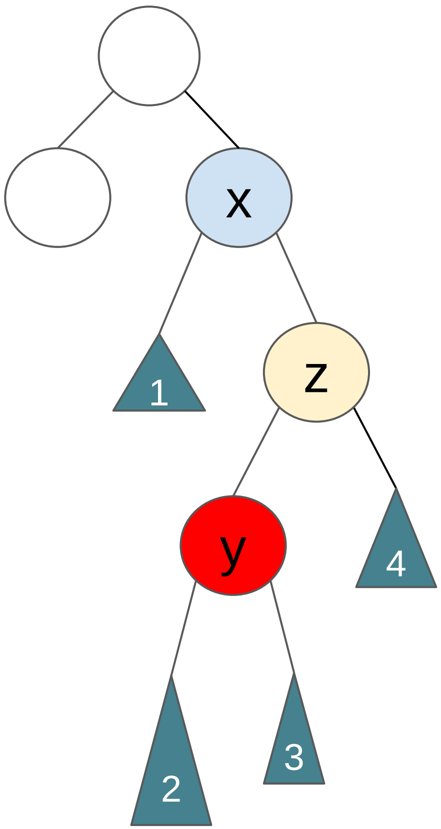
The nodes x, y, z are in increasing order

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

Update heights along this branch

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



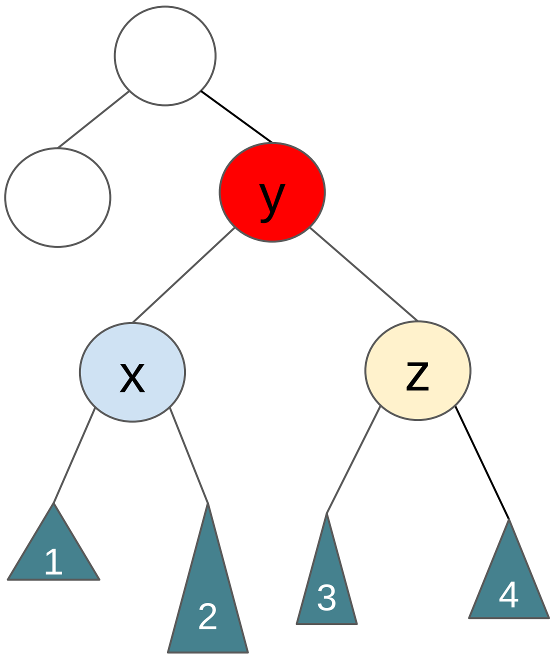
The nodes x, y, z are in increasing order

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

Update heights along this branch

Trinode restructuring: right-left-heavy subtree



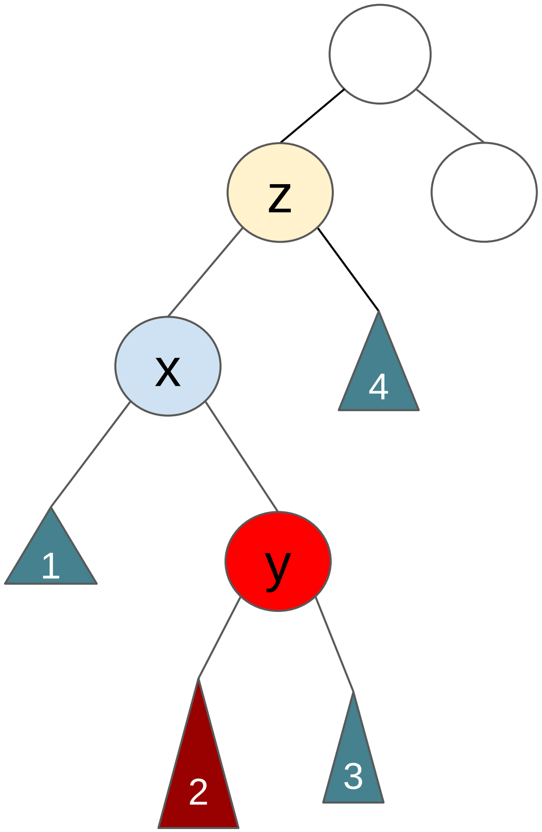
The nodes x, y, z are in increasing order

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

Update heights along this branch

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



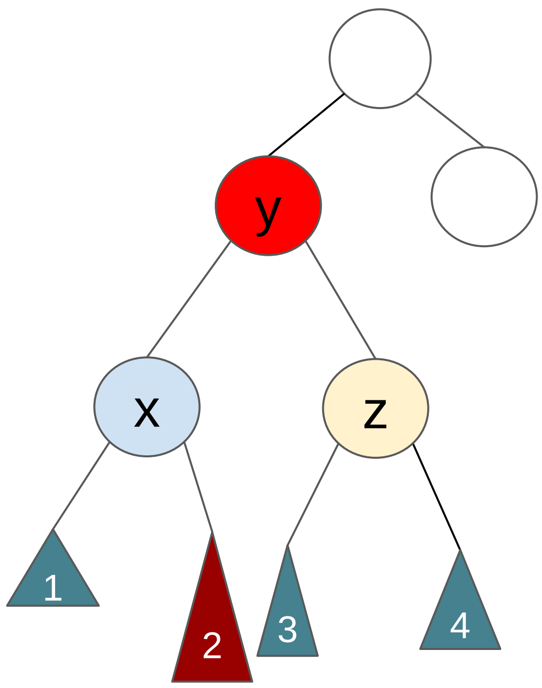
The nodes x, y, z are in increasing order

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

Update heights along this branch

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

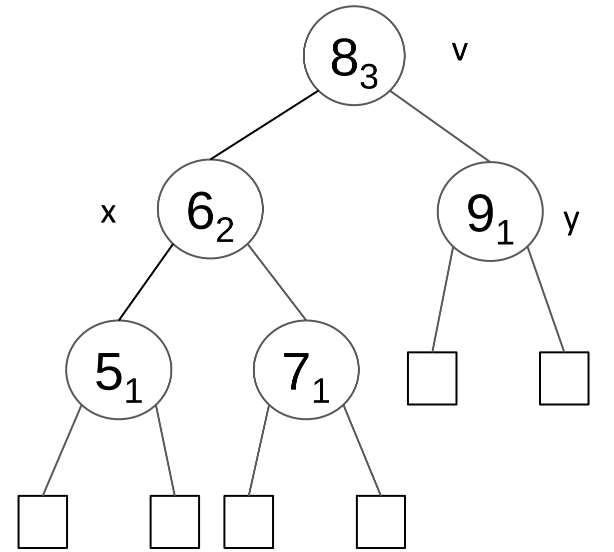
Reattach all 4 children (some of them can be NULL) to x and z

Update heights along this branch

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

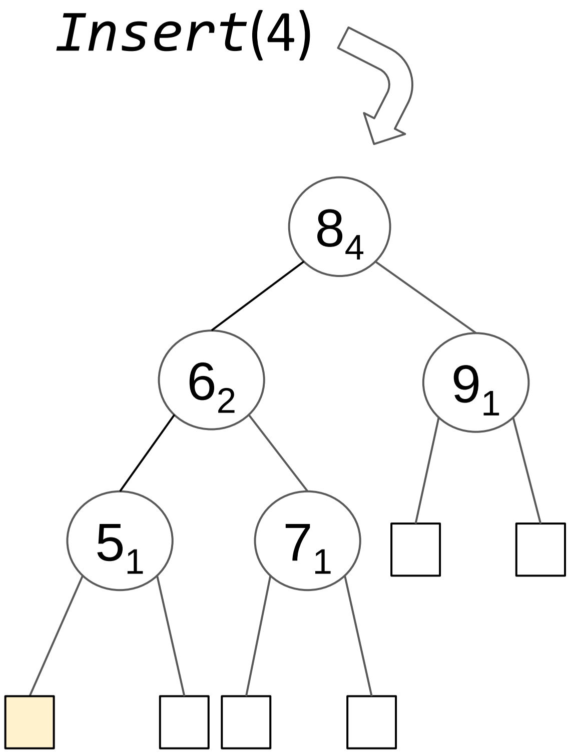


I.e. if the children of *v* are *x* and *y*, then $|h(x) - h(y)| \leq 1$

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

AVL tree: insertion

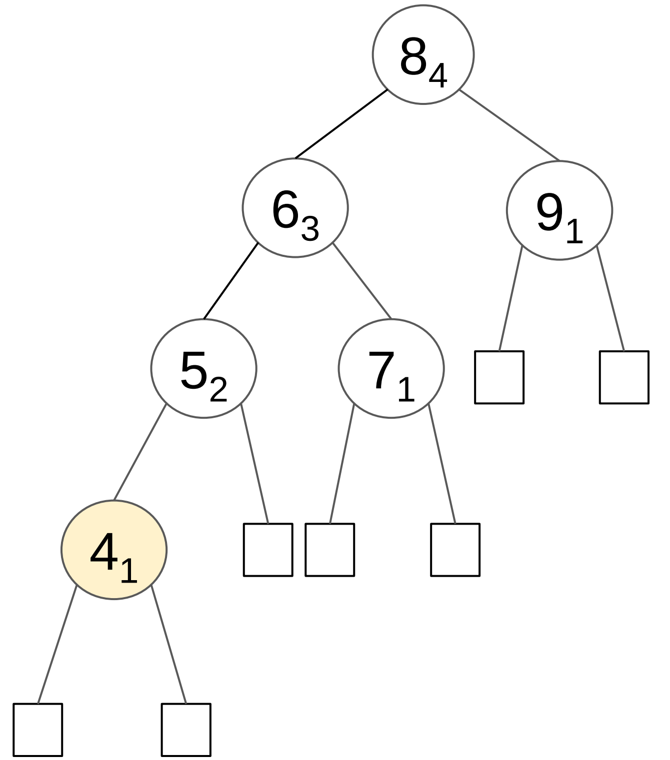
First, we perform regular insertion into BST and end up filling up one of the NULL nodes with the new value



AVL tree: insertion

External node becomes a new internal node

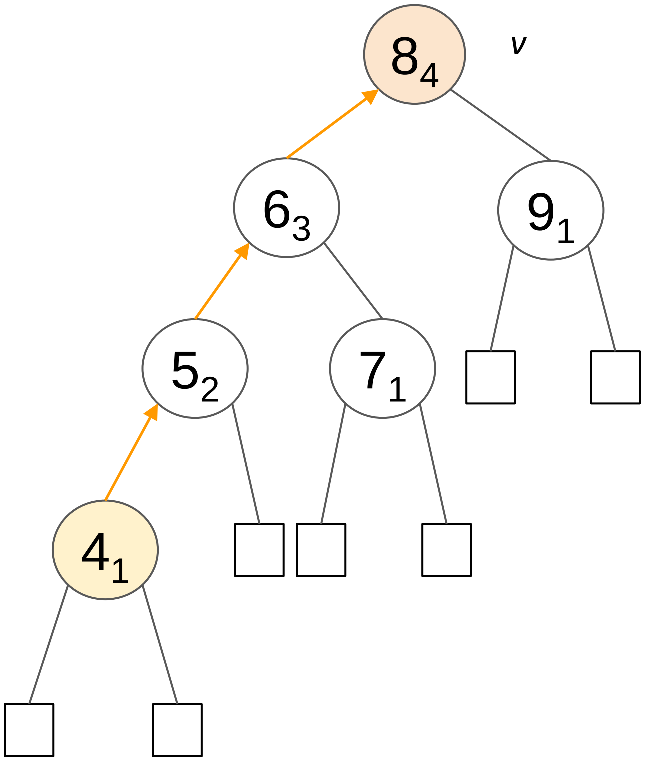
After the insertion, some internal nodes may become unbalanced



AVL tree: rebalancing after insertion

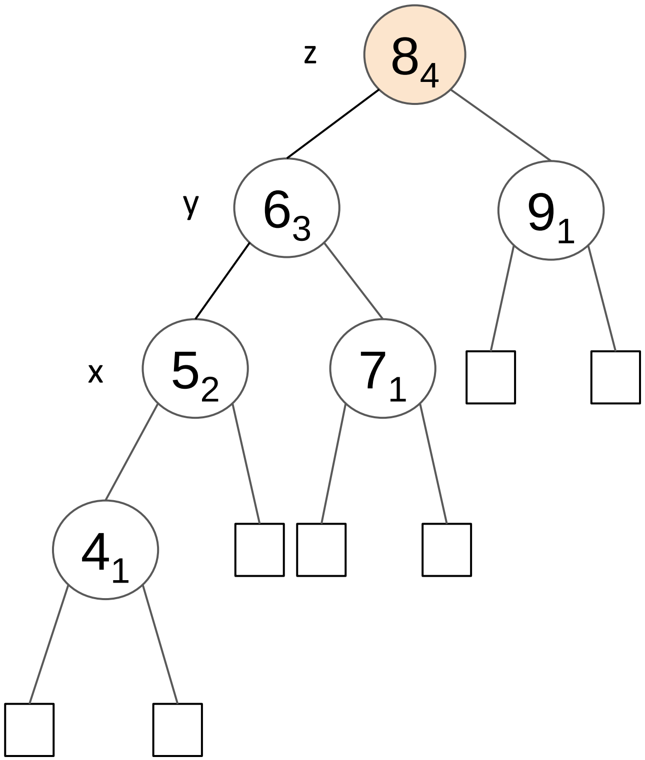
We go up from the inserted node until we encounter the first unbalanced node v

Note that in order for a branch to become unbalanced, there are at least 2 nodes below v



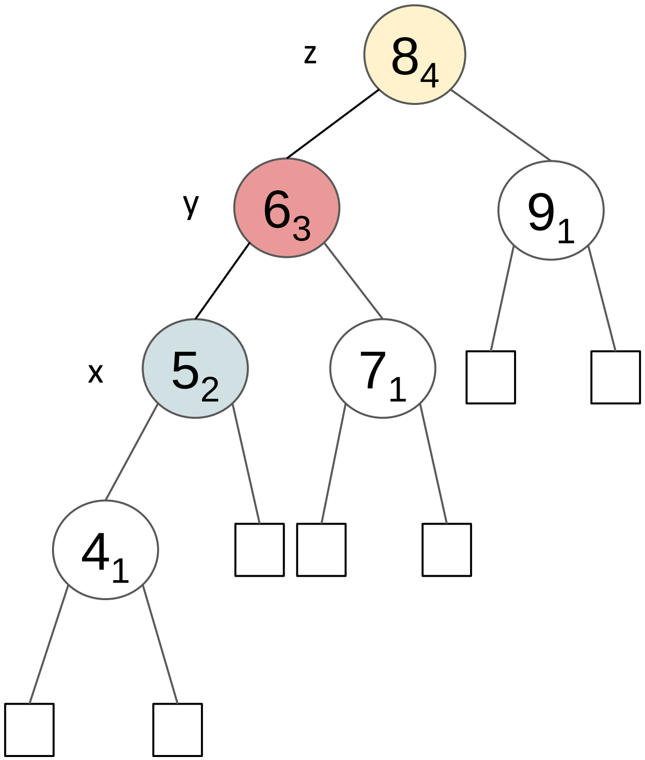
AVL tree: rebalancing after insertion

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



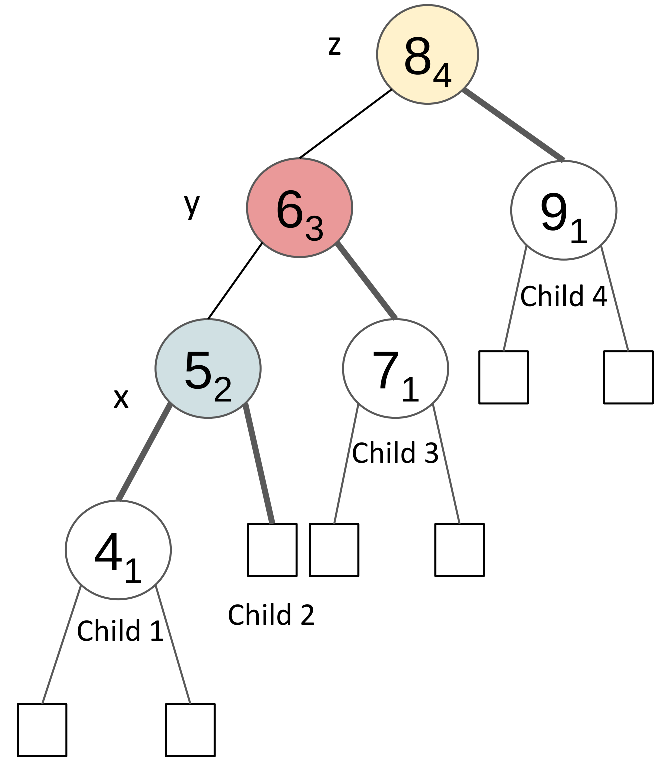
AVL tree: rebalancing after insertion

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



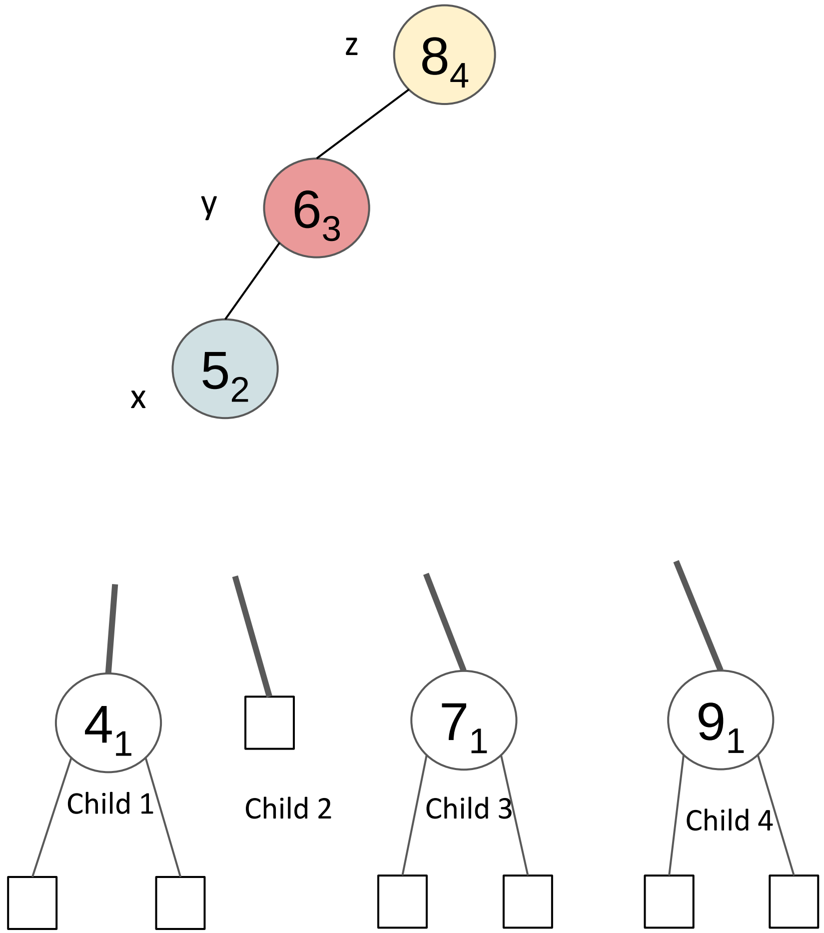
AVL tree: rebalancing after insertion

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



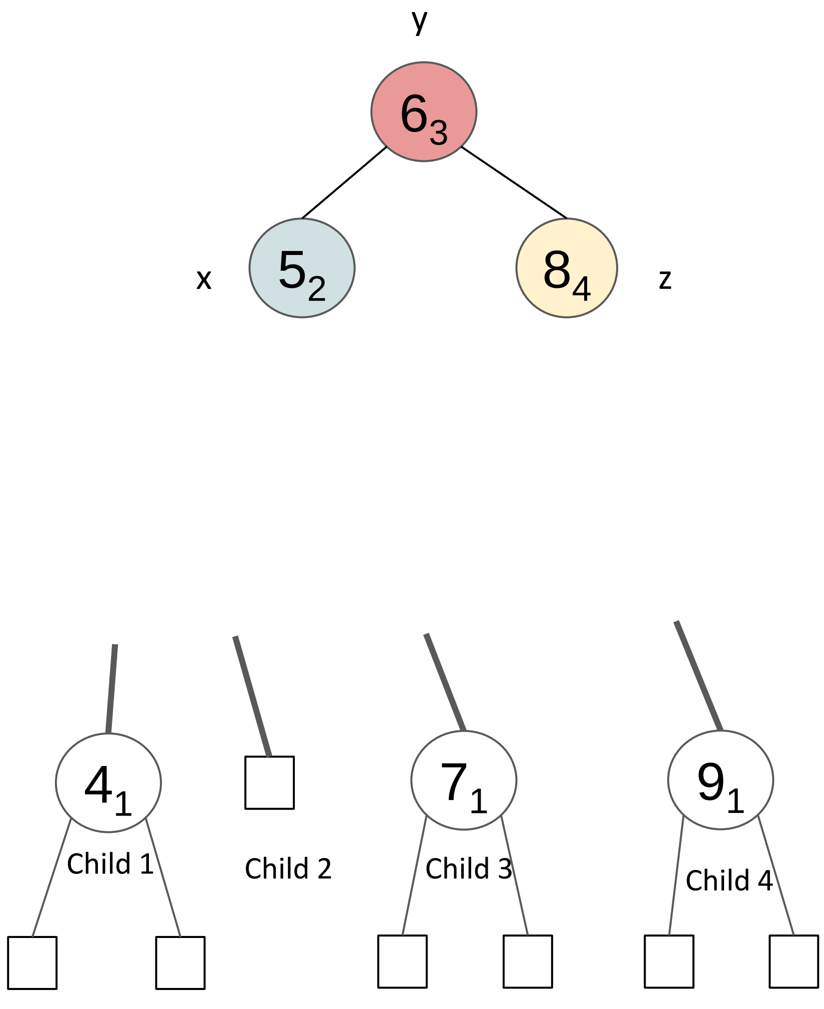
AVL tree: rebalancing after insertion

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



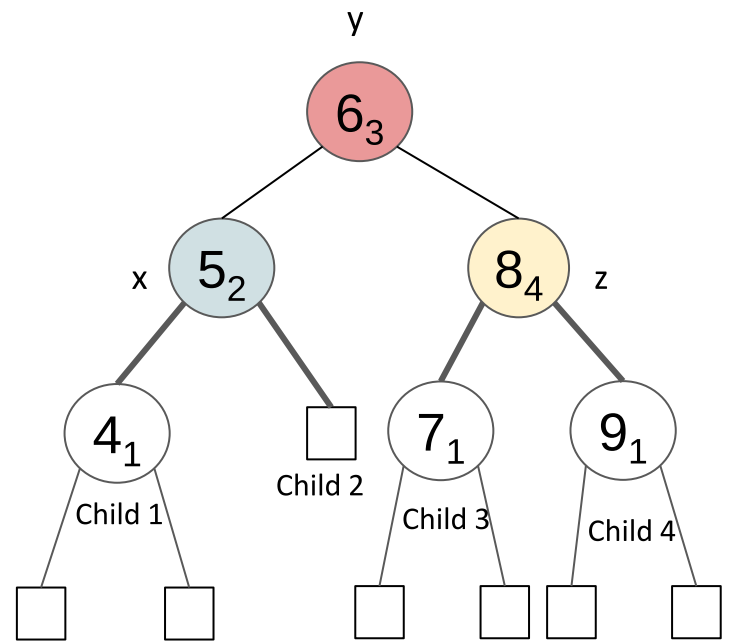
AVL tree: rebalancing after insertion

Perform rotation



AVL tree: rebalancing after insertion

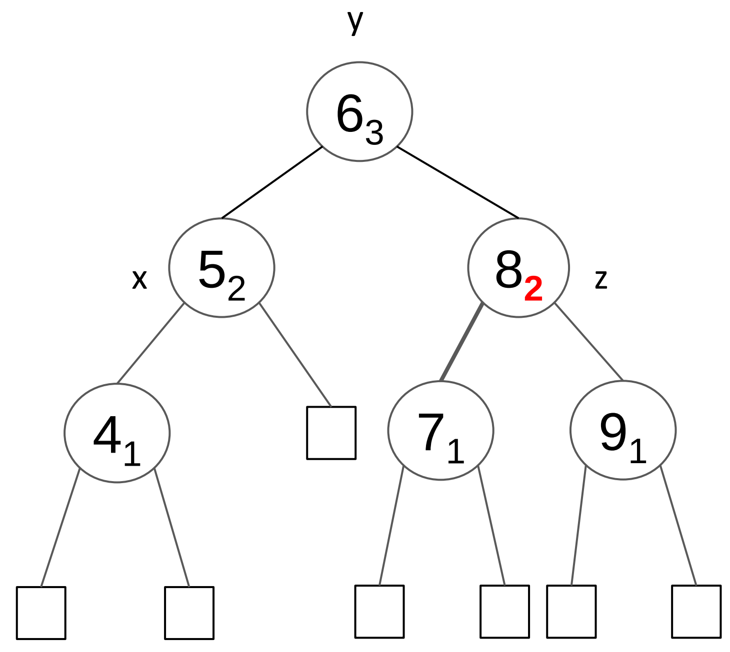
Reattach children



AVL tree: rebalancing after insertion

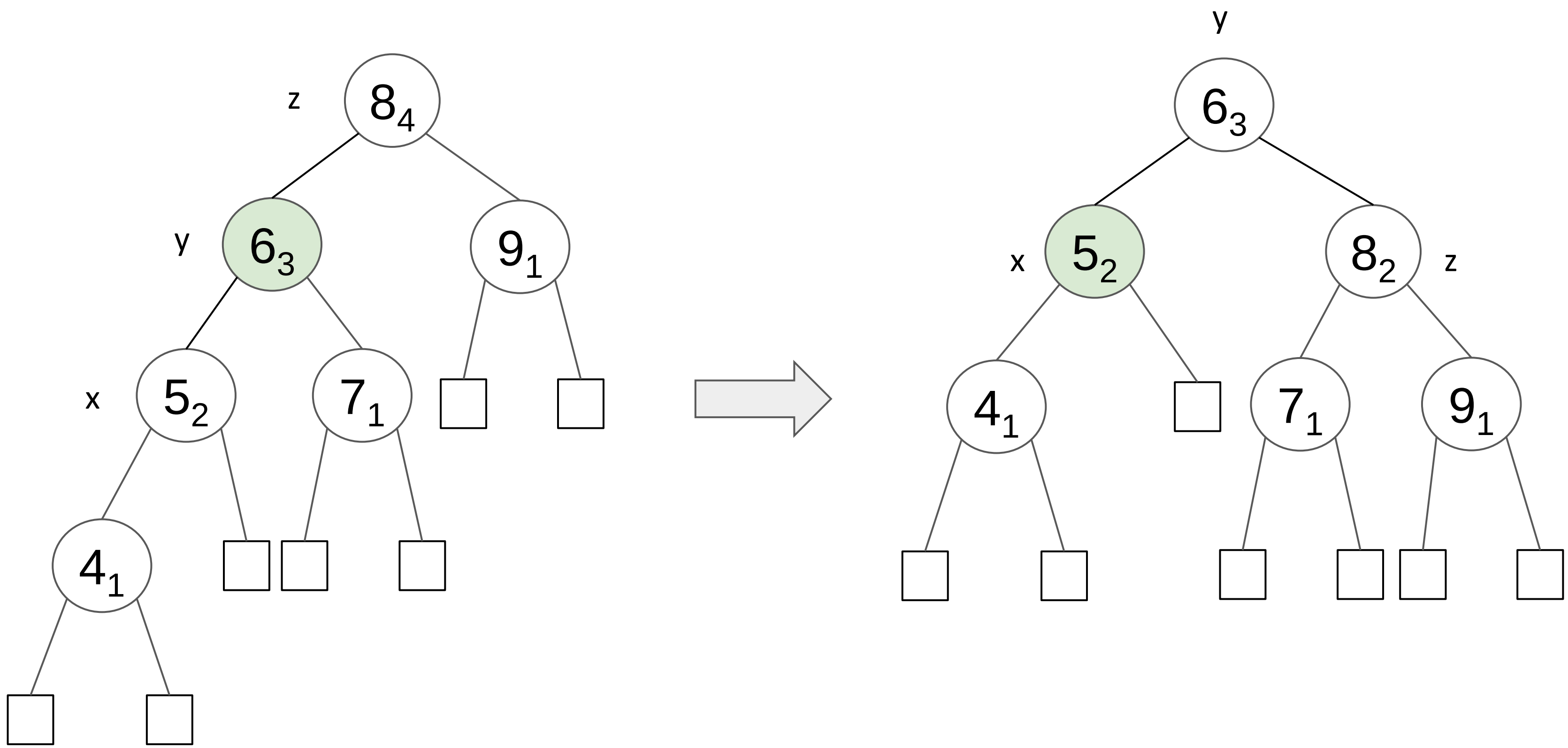
Update height of rebalanced nodes x, y or z

Note that the height of the children does not change and does not need to be updated



The entire time of insertion is still O(tree height)

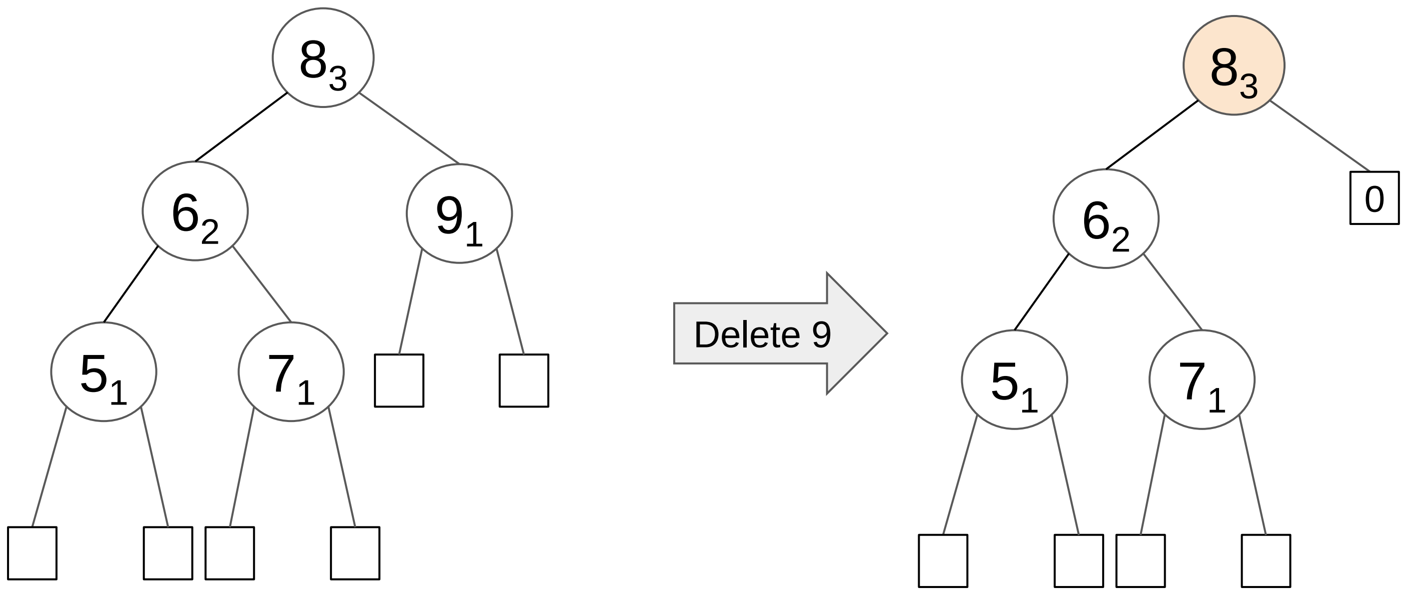
AVL tree: insertion summary



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

AVL tree: deletion - similar idea



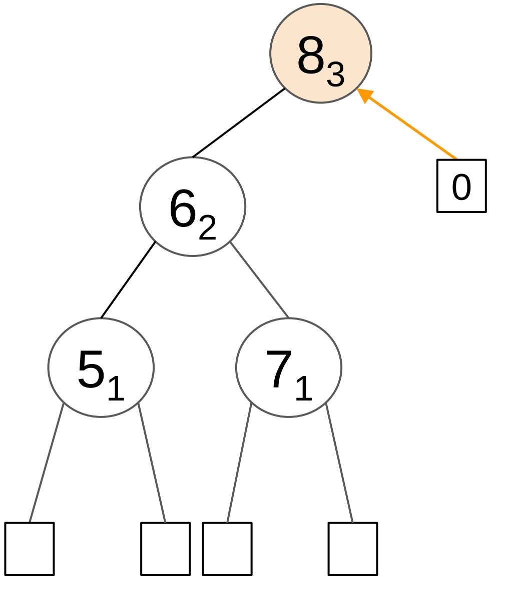
By removing a node from AVL tree some nodes may become unbalanced

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

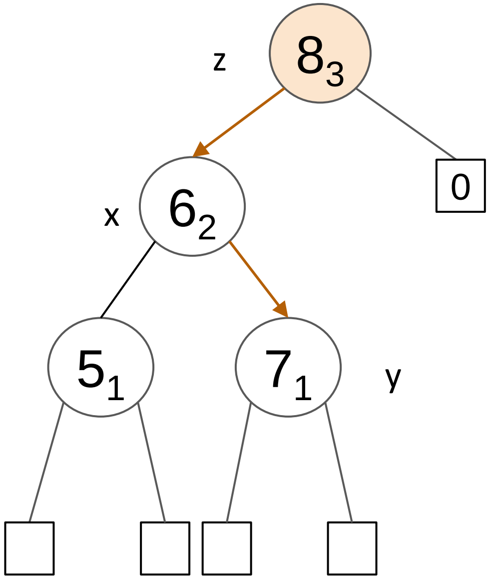
We move up the tree from the current NULL node until we encounter an internal node which is unbalanced



AVL tree: rebalancing after deletion

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

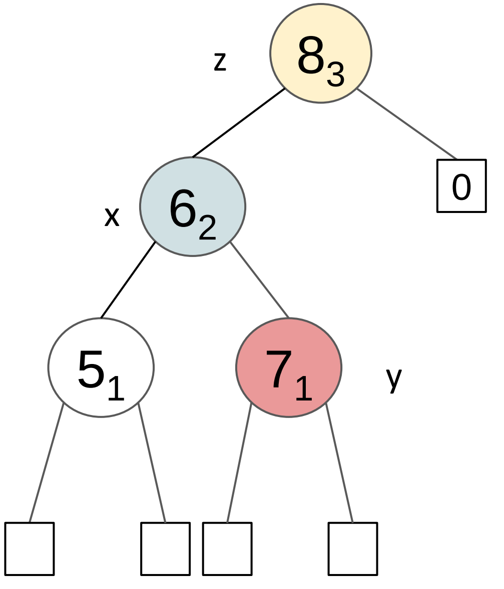
We produce 3 nodes x, y, z to be restructured



AVL tree: rebalancing after deletion

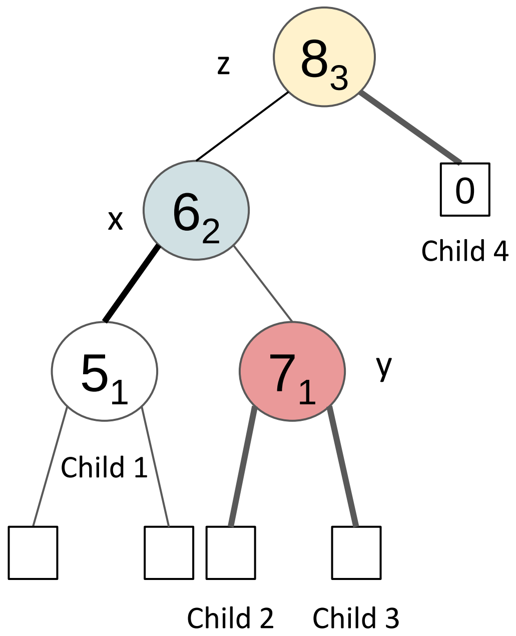
We perform rotation around y

This is accomplished with trinode restructuring as before



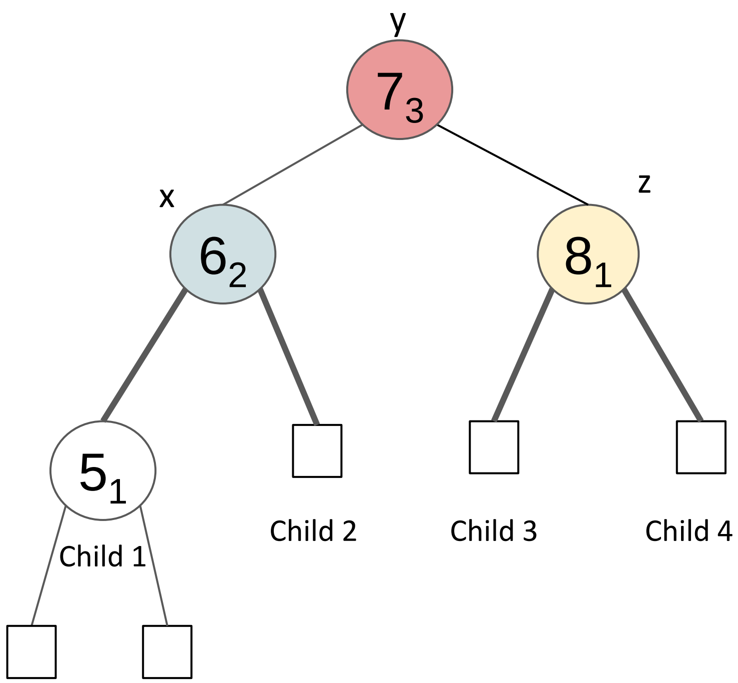
AVL tree: rebalancing after deletion

Trinode restructuring: detach children of x, y, z

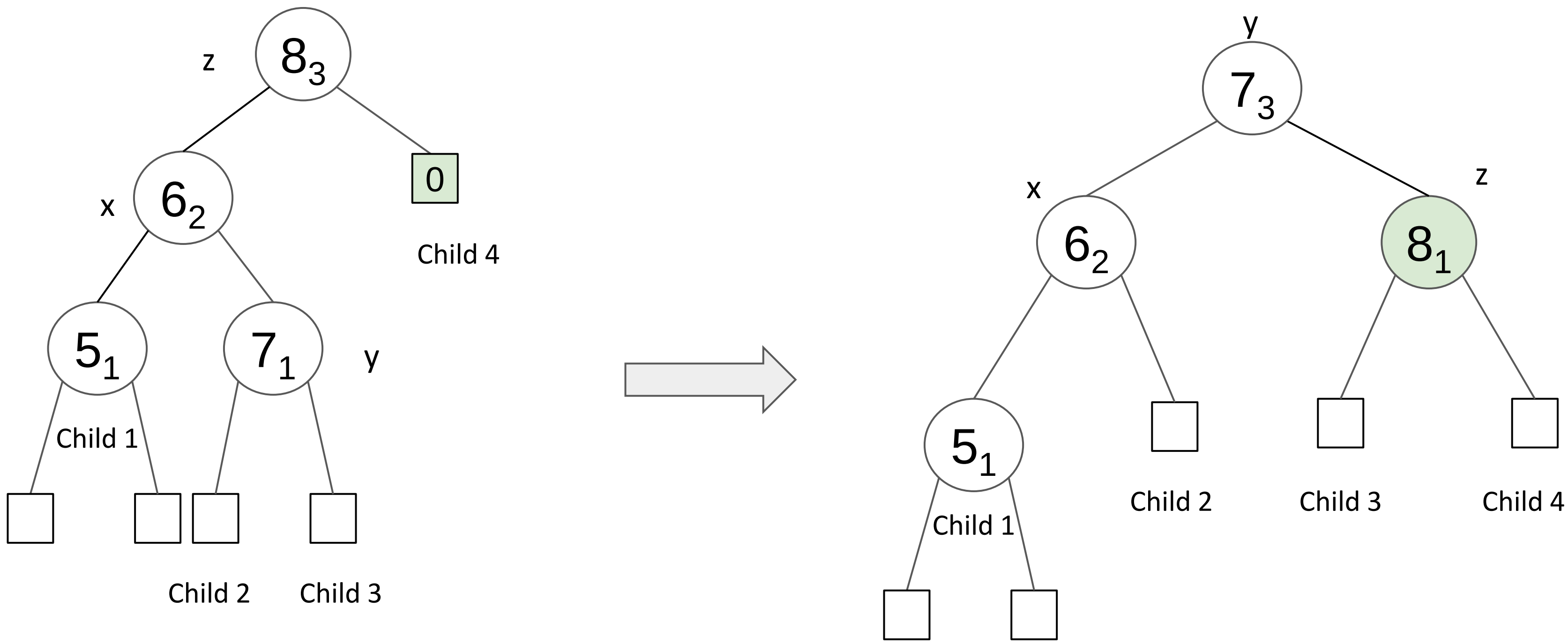


AVL tree: rebalancing after deletion

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

Theorem

AVL tree with n keys has height $O(\log n)$

For the proof refer to Chapter 4.2 of the provided book chapter

Many more Balanced Search Trees exist

Red-Black trees: wikipedia link

Splay trees: wikipedia link

B-trees: wikipedia link

...