Lecture 11: How to Balance a Tree

These slides include material originally prepared by Dr. Ron Cytron, Dr. Jeremy Buhler, and Dr. Steve Cole.

Announcements

- Exam 2 tomorrow
  - See Piazza/e-mail for details
  - *Must* go to your assigned room

- Exam 3: Cornerstone apps due end of this week

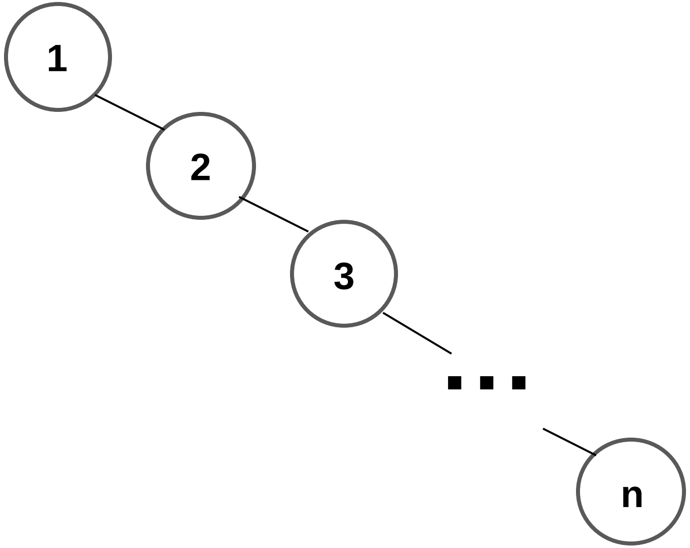
- Lab 11 out this week
  - Pre-lab due **Tue. 4/9**; rest due **Fri. 4/12**

Review: Worst-Case Costs for BST Operations

- Find – $\Theta(h)$ for tree of height h
- Min/Max – $\Theta(h)$ for tree of height h
- Insert – $\Theta(h)$ for tree of height h
- Iterate – $\Theta(h)$ for tree of height h
- Remove – $\Theta(h)$ for tree of height h

How Tall Can a BST with n Nodes Be?

- Insert keys 1..n in order



How Tall Can a BST with n Nodes Be?

- Insert keys 1..n in order

Tree height is worst-case Θ(n)

Can We Overcome Worst-Case Θ(n) Costs for Tree Operations?

What If Our Trees Were Never Too Tall?

- Defn: a binary tree with n nodes is said to be balanced if it has height O(log n).

- *Example:* a complete binary tree with 2^n-1 nodes has height n – 1, so is balanced.

- *In a balanced BST, all BST ops are worst case O(log n).*

Strategy for Balancing Trees

1. Define a structural property P that applies to only *some* BSTs

2. Prove that BSTs satisfying property P are balanced

3. Make sure a trivial BST (one node) satisfies P

4. Show how to insert, remove while maintaining P

From the End of Lecture 10, through Today

- **AVL tree** – heights of left, right subtrees of every node differ in height by at most 1

- Prove that AVL property implies balance

- Show how to maintain AVL property under insertion, deletion

- *(After that, a different approach to balanced trees!)*

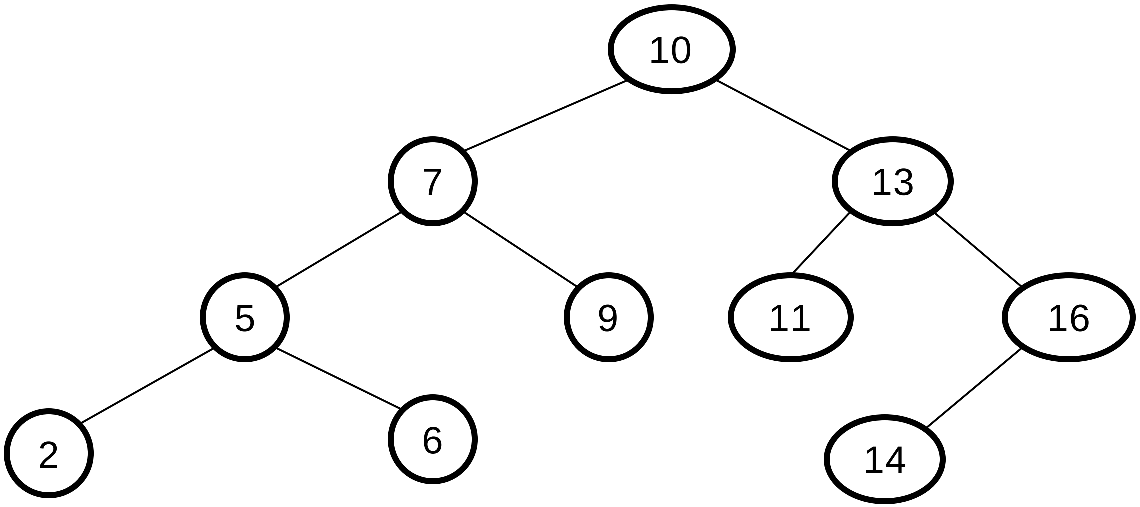
How do we maintain AVL property efficiently?

- Lecture 10 (AVL property)

  + Studio 10 (order stats in trees)

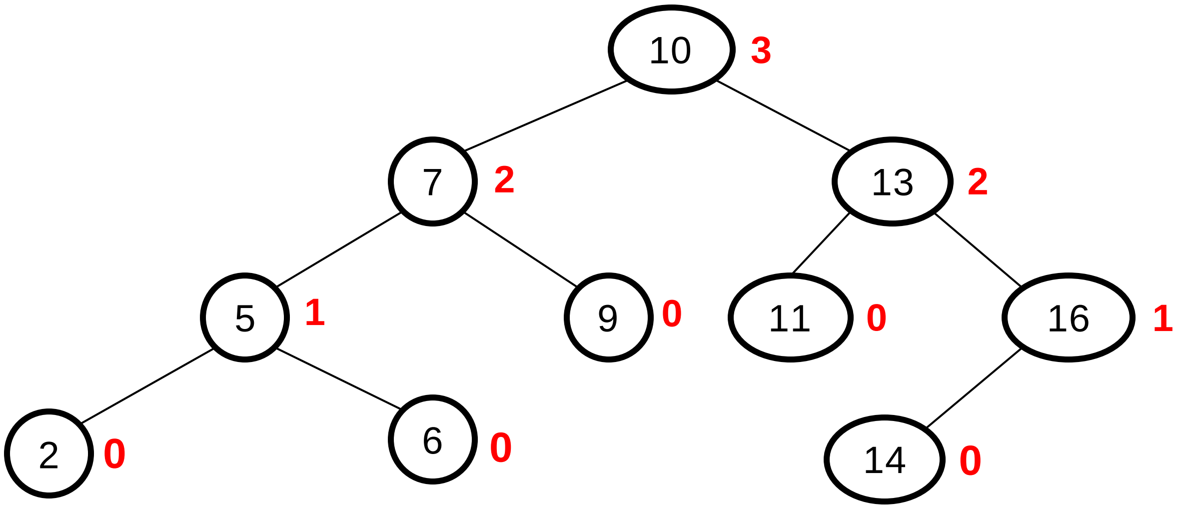
Checking the AVL Property

- To check AVL property for tree T, we will maintain height of each subtree of T in subtree's root.



Checking the AVL Property

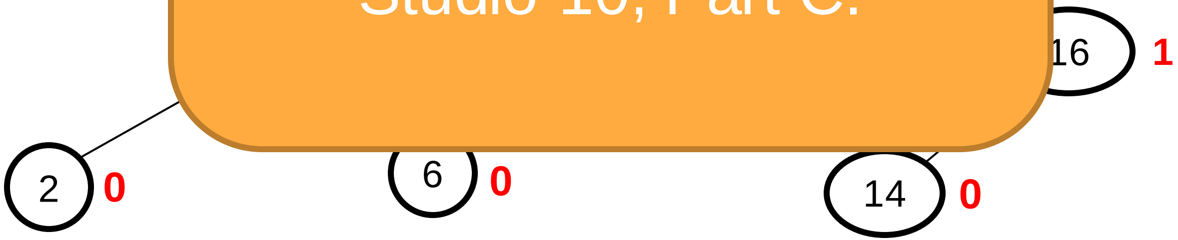
- To check AVL property for tree T, we will maintain height of each subtree of T in subtree's root.



Checking the AVL Property

- To check AVL property for tree T, we will maintain height of each sub

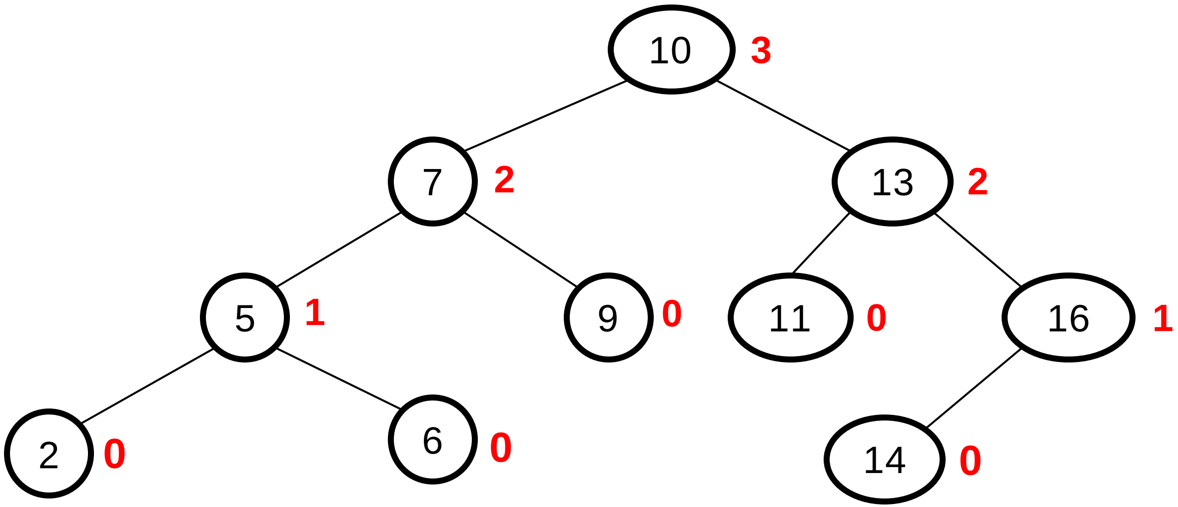
You studied how to maintain height (and size) under insertion, deletion in Studio 10, Part C.



Key Measurement: Balance Factor

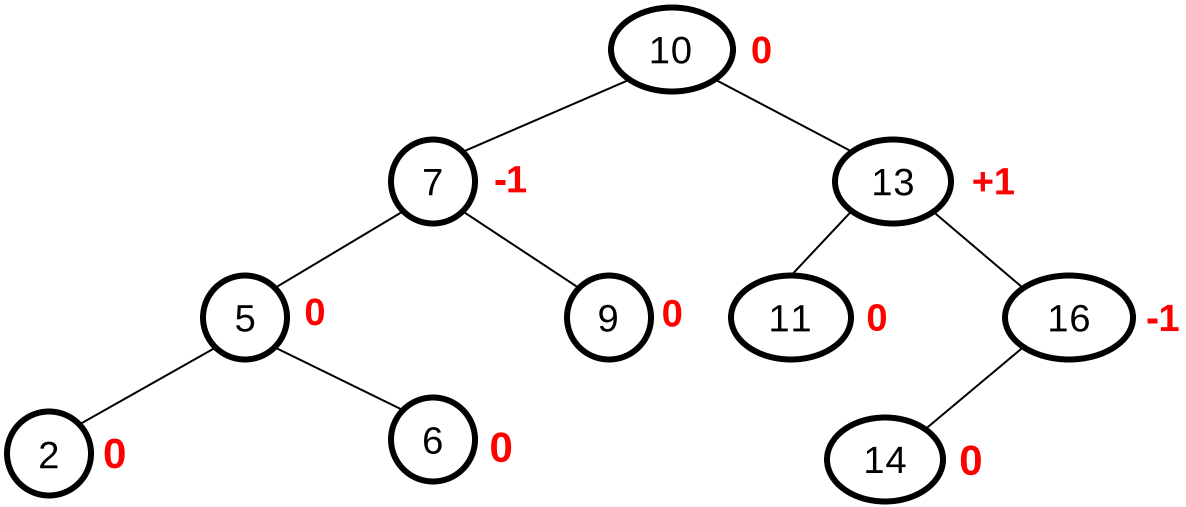
- For any node x, the **balance factor** of x is the difference

  (height of right subtree of x – height of left subtree of x)



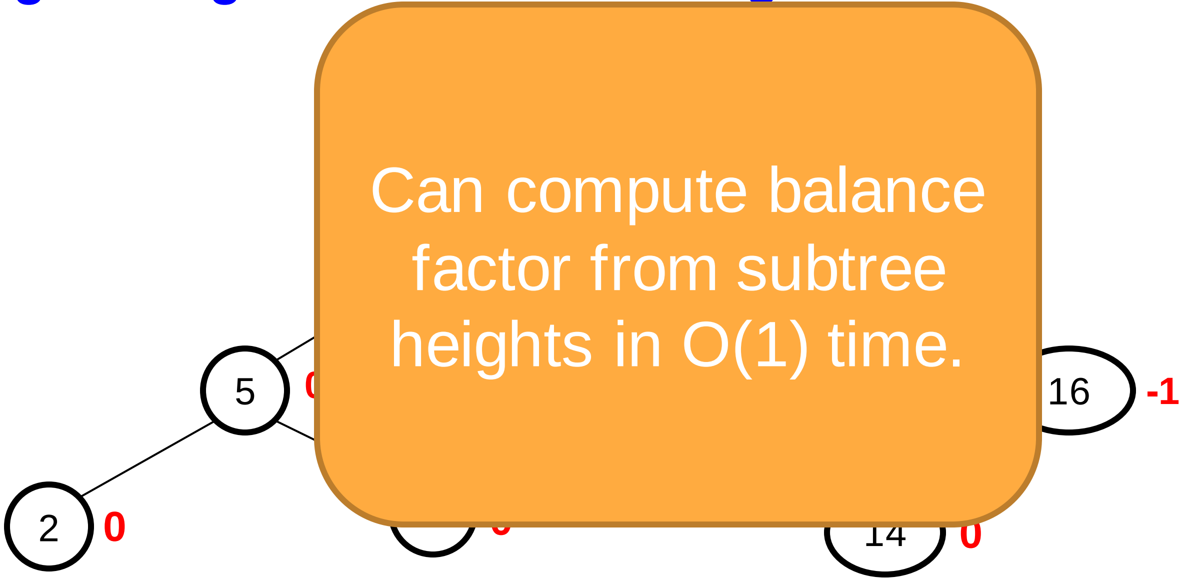
Key Measurement: Balance Factor

- For any node x, the **balance factor** of x is the difference
  (height of right subtree of x – height of left subtree of x)



Key Measurement: Balance Factor

- For any node x, the **balance factor** of x is the difference
  (height of right subtree of x – height of left subtree of x)



Balance Factor for AVL Trees

- Every node of an AVL tree has a balance factor of either **???, ???, or ???**

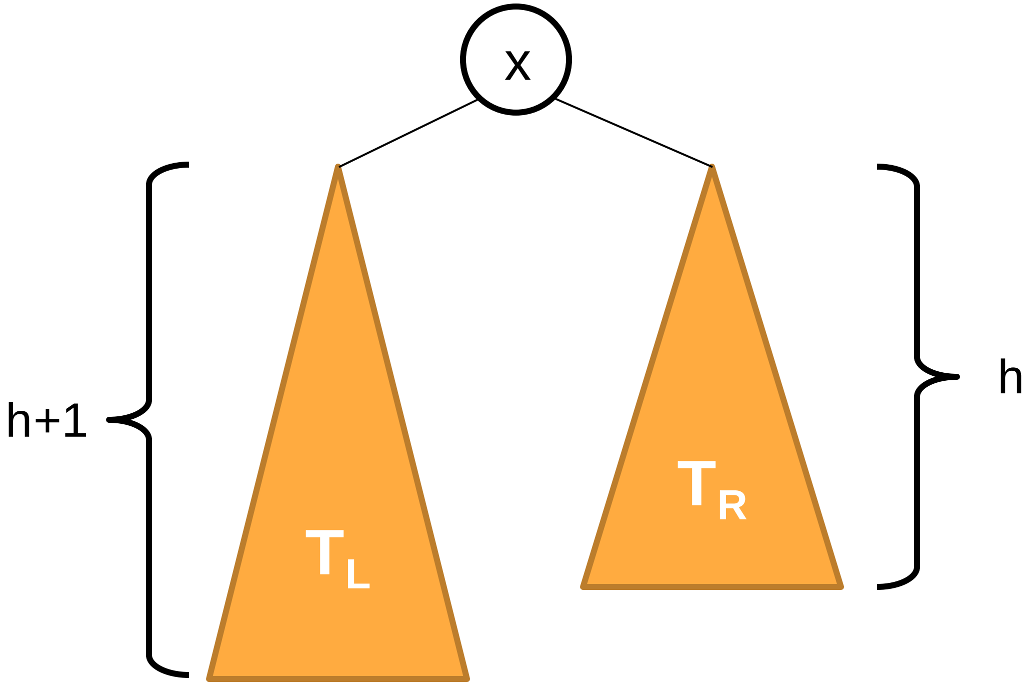
Balance Factor for AVL Trees

- Every node of an AVL tree has a balance factor of either
  -1, 0, or +1

- (Follows because subtree heights cannot differ by > 1.)

- **Question**: if we add or remove a node to/from an AVL tree, by how much could the balance factors of its nodes change?

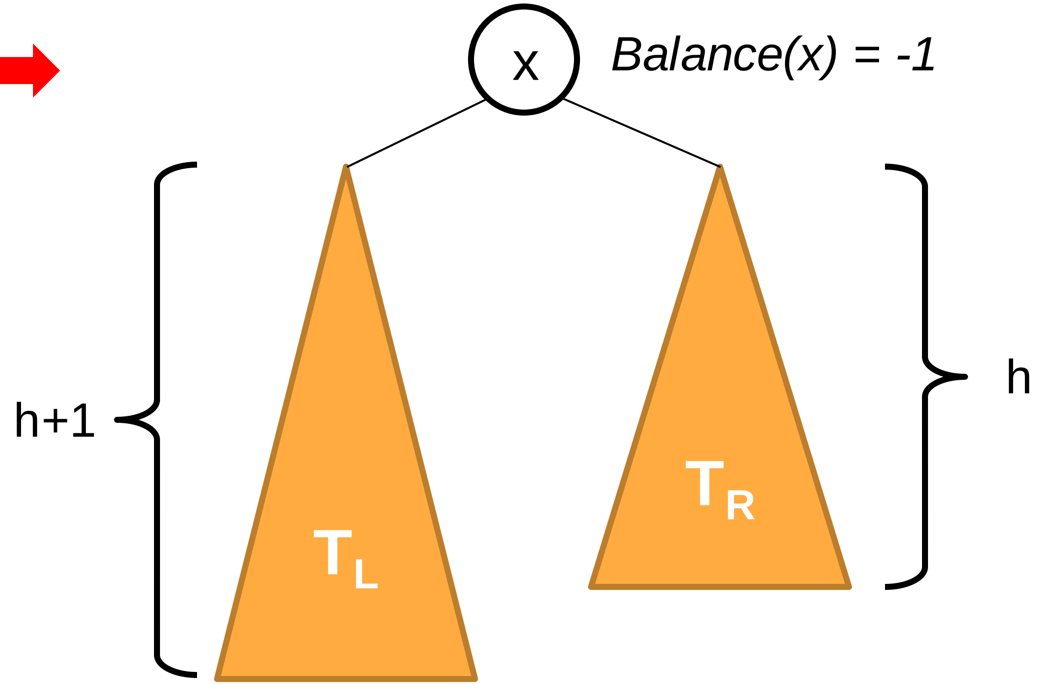
Inserting a Node into an AVL Tree



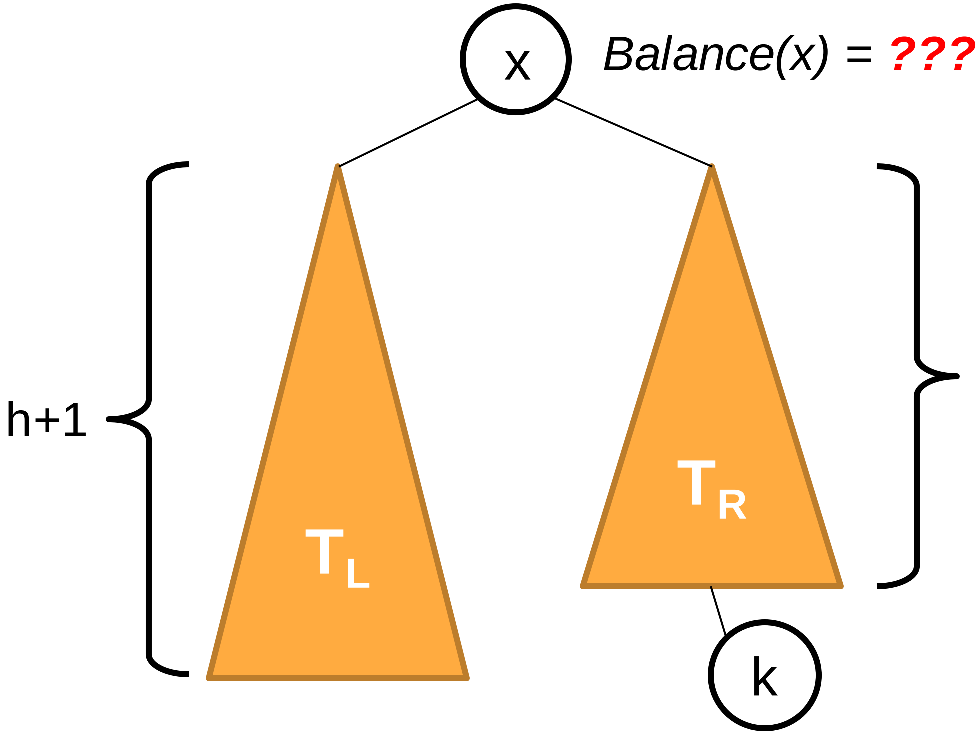
Before insertion, balance is 0 or +-1. Here, we show -1.

Inserting a Node into an AVL Tree

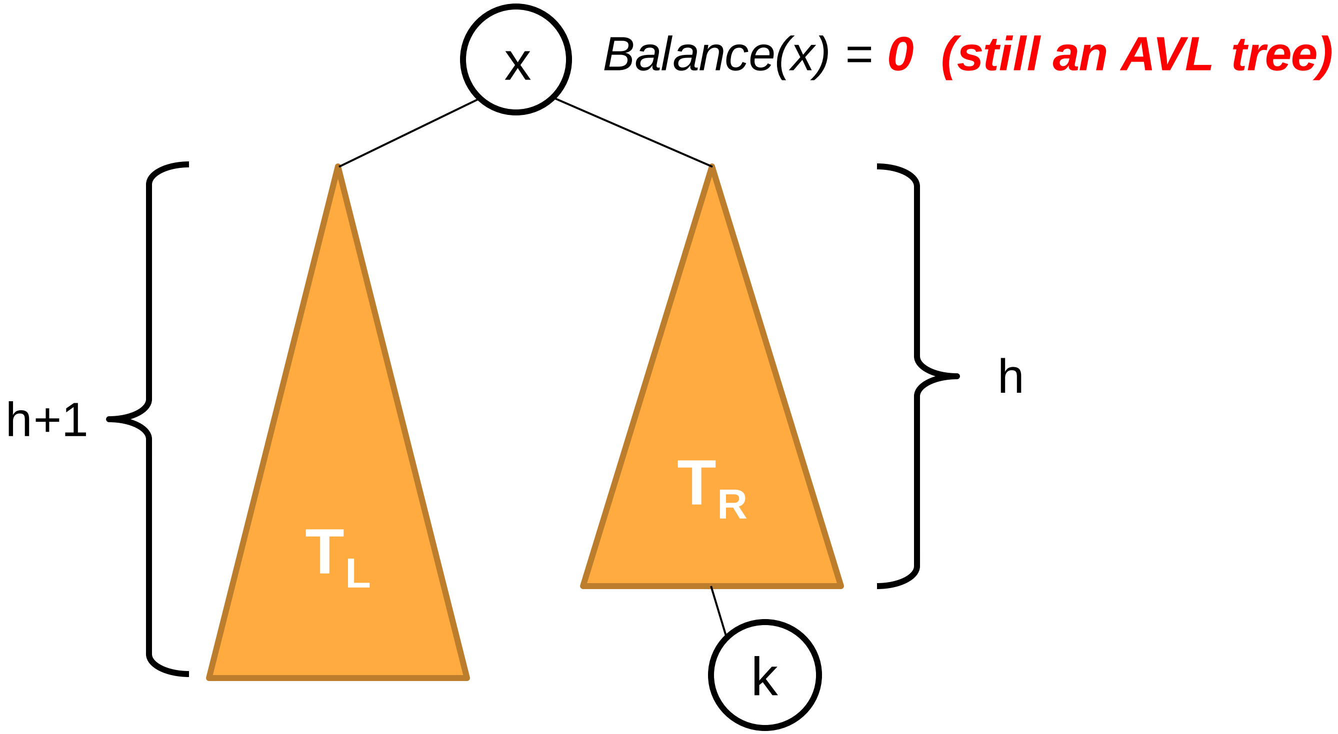
Insert k



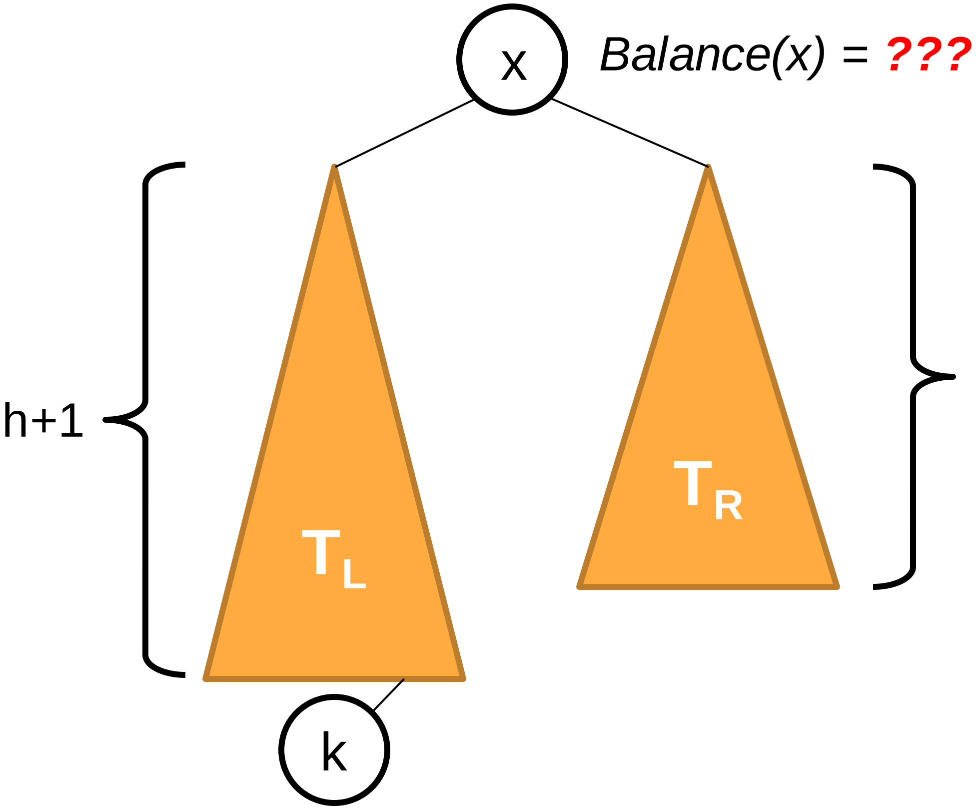
Inserting a Node into an AVL Tree



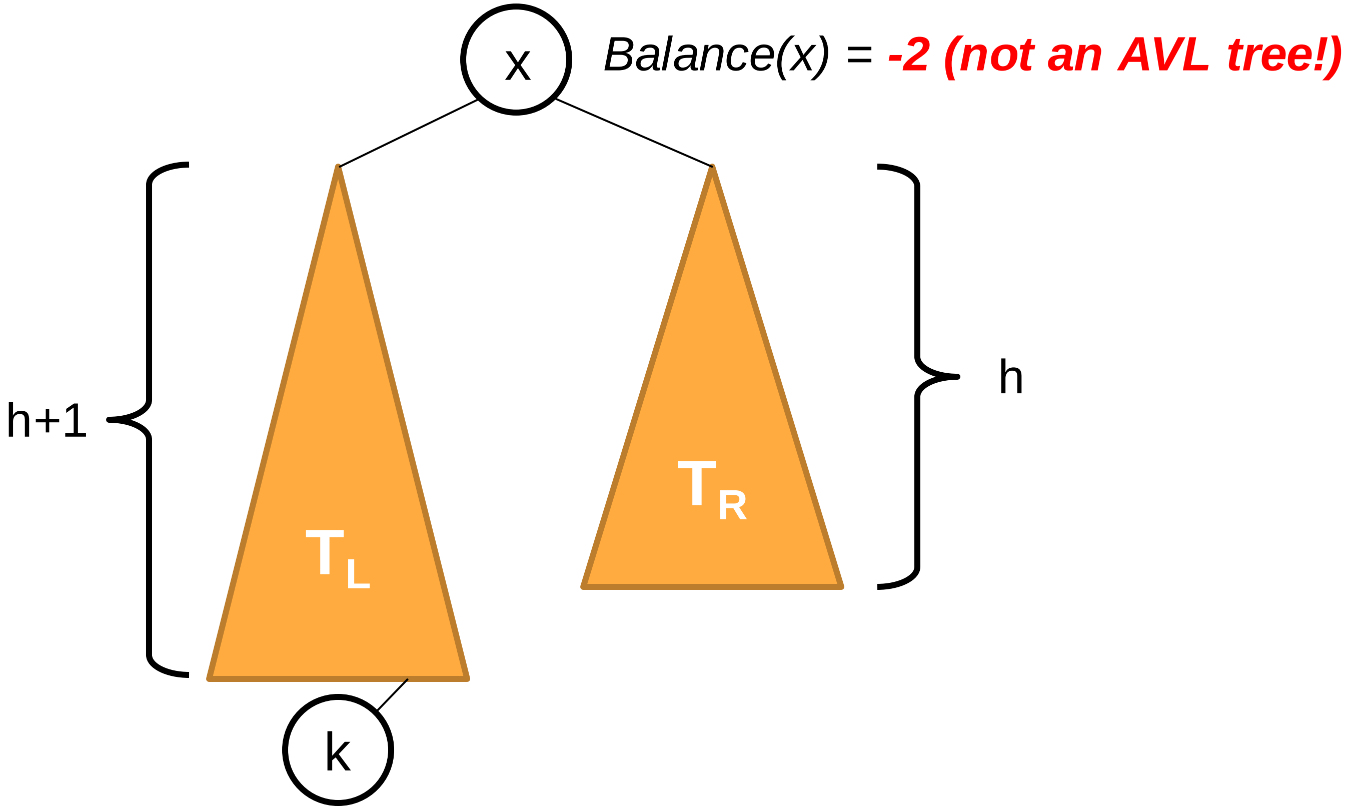
Inserting a Node into an AVL Tree



Inserting a Node into an AVL Tree



Inserting a Node into an AVL Tree



Insert and Remove can Unbalance the Tree

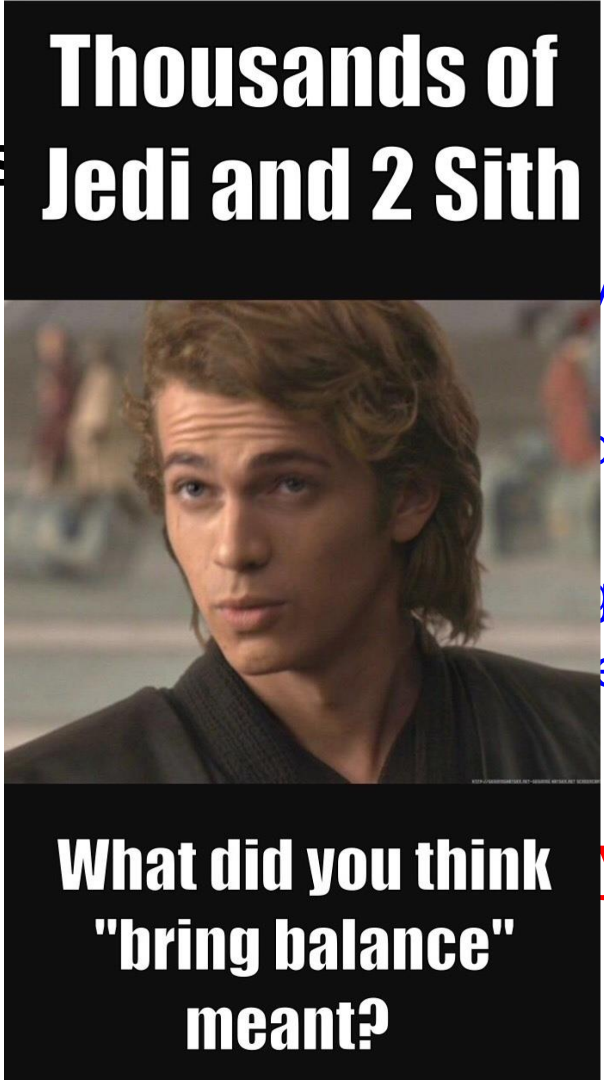
- Inserting node into an AVL tree can make the root's balance +2 or -2.
- Similarly, removing a node can make root's balance +2 or -2.
- (Why? Because inserting or removing one node changes height of **at most one** of root's subtrees by **up to ±1**.)
- Resulting tree may no longer be an AVL tree!

Insert and Remove can Unbalance the Tree

- Inserting node into an AVL tree can m … 2.
- Similarly, removing a node can ma
- (Why? Because inserting or remov
  most one of root's subtrees by **up**
- Resulting tree may no lo

Challenge: after insert or remove, restore balance to the tree…

Ins can Unbalance the Tree



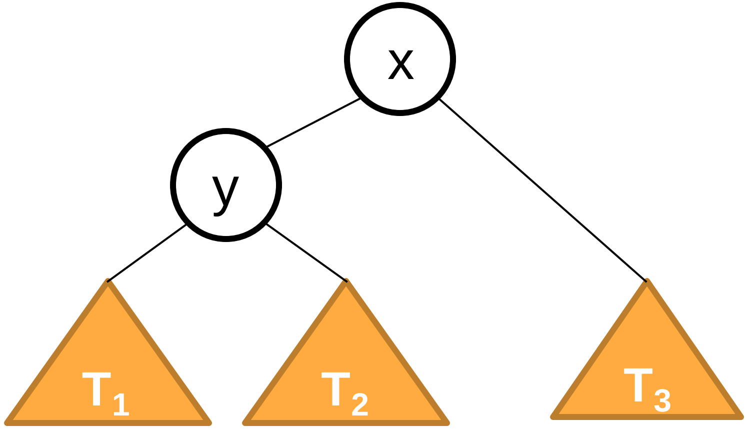
- VL tree can m 2.
- de can ma
- g or remov
  ees by **up**
- y no lo

Challenge: after insert or remove, restore balance to the tree…

while preserving BST property!

Important Tool: Tree Rotation

- A **tree rotation** (left or right) changes the root of the tree while maintaining the BST property.

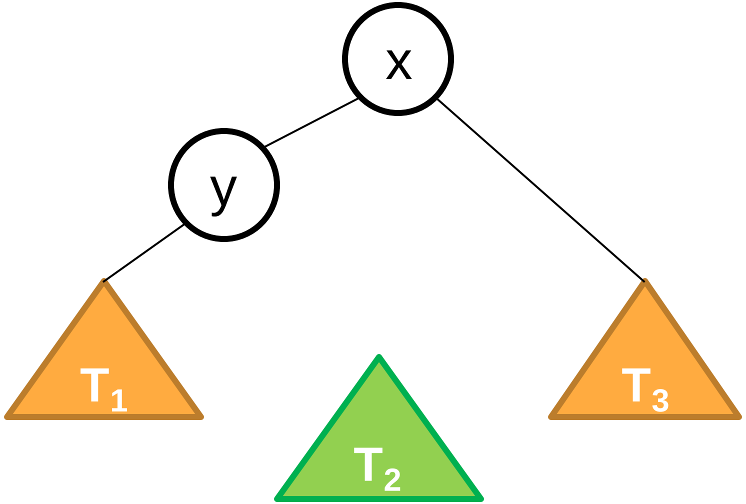


rotate right

Important Tool: Tree Rotation

- A **tree rotation** (left or right) changes the root of the tree while maintaining the BST property.

1. **Detach** right subtree from y, making it an "orphan"

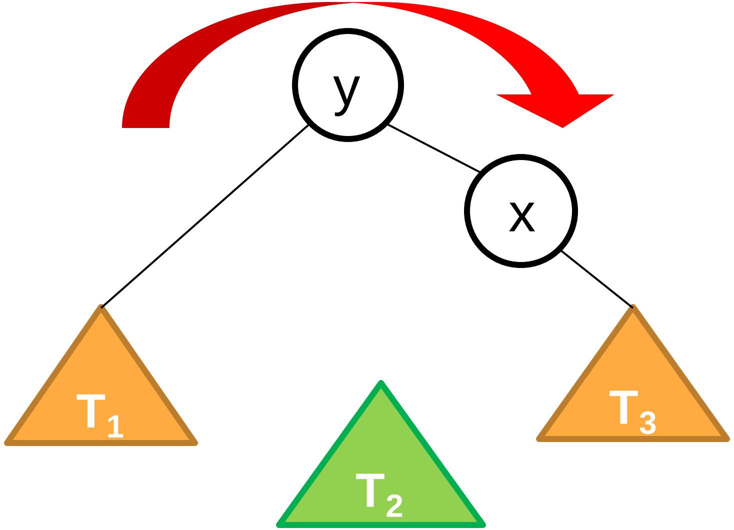


rotate right

Important Tool: Tree Rotation

- A **tree rotation** (left or right) changes the root of the tree while maintaining the BST property.

2. Make y **new root** of tree

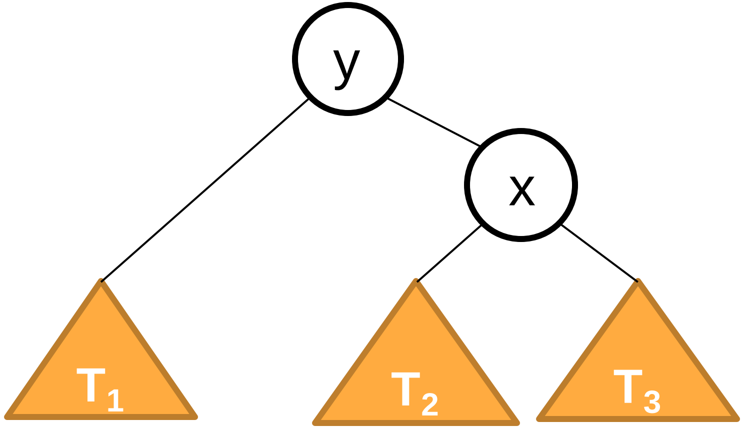


rotate right

Important Tool: Tree Rotation

- A **tree rotation** (left or right) changes the root of the tree while maintaining the BST property.

3. **Reattach**
"orphaned" subtree
as left subtree of x.



rotate right

Important Tool: Tree Rotation

- A **tree rotatio** root of the tree while mainta

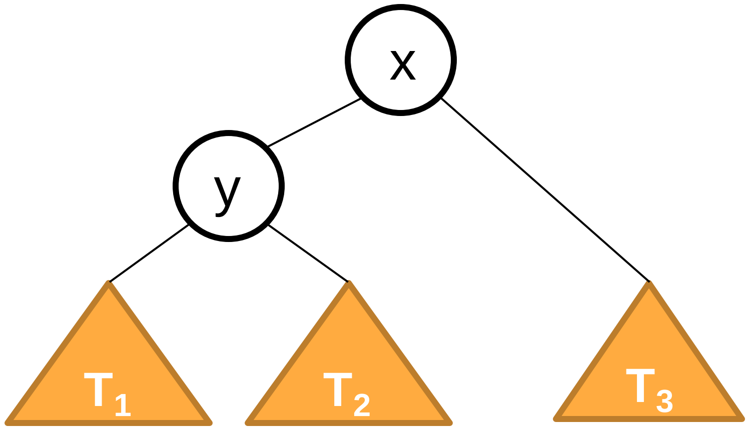
Left rotation is simply the reverse of these steps.

3. **Reattach** "orphaned" subtree as left subtree of x.

rotate right

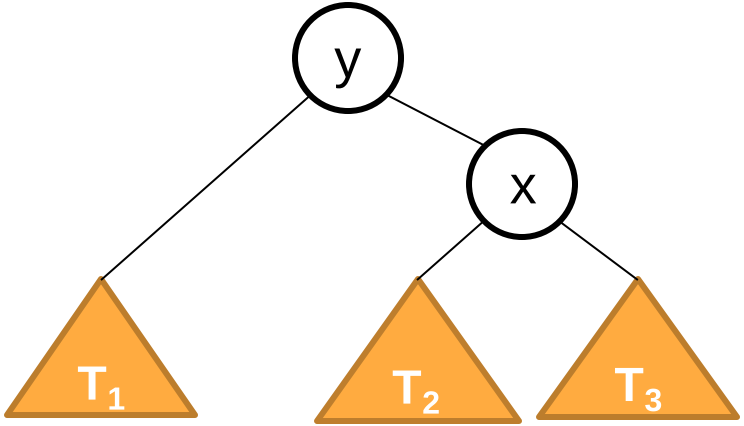
Does Rotation Preserve the BST Property?

- Before the rotation, BST property tells us that
  - $x \geq y$
  - $T_1 \leq y$
  - $T_3 \geq x$
  - $T_2 \geq y, \leq x$



Does Rotation Preserve the BST Property?

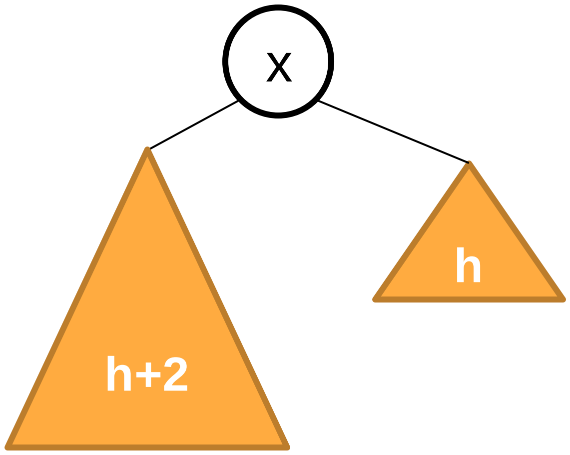
- Before the rotation, BST property tells us that
  - $x \geq y$
  - $T_1 \leq y$
  - $T_3 \geq x$
  - $T_2 \geq y, \leq x$



- These inequalities are consistent with final tree as well.

Correcting Balance via Rotation

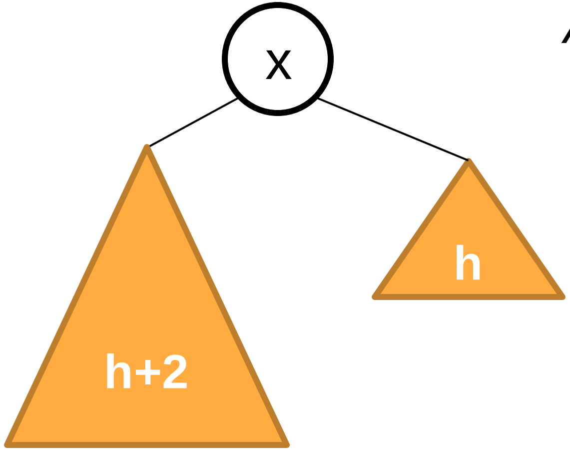
- Suppose after insertion, root has balance factor -2



- *(For +2, do the mirror image of what follows)*

Correcting Balance via Rotation

- Suppose after insertion, root has balance factor -2

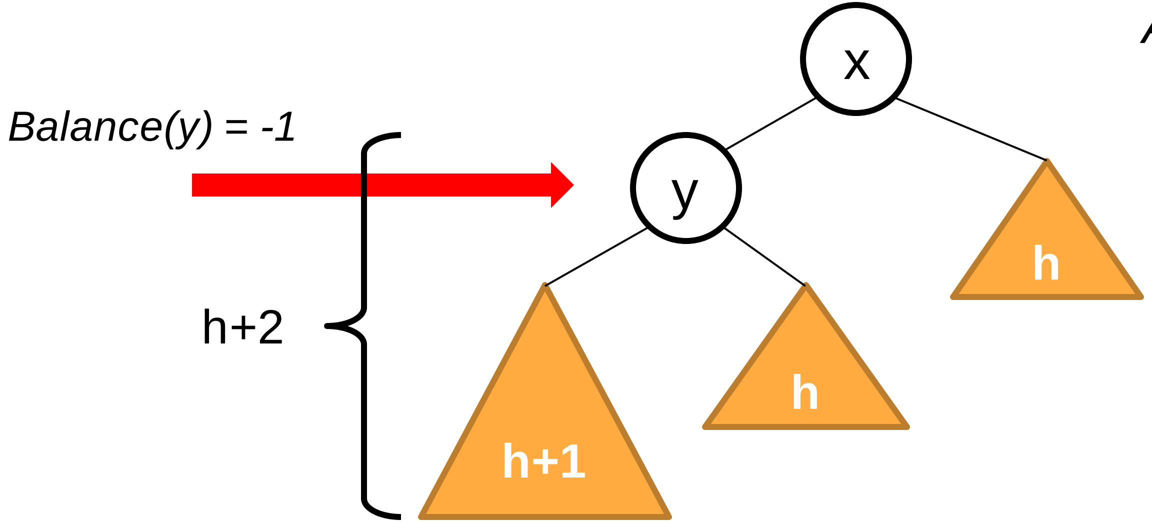


*Assume **both subtrees have AVL property**, so only violation is at root.*

- *(For +2, do the mirror image of what follows)*

Correcting Balance via Rotation (Case 1)

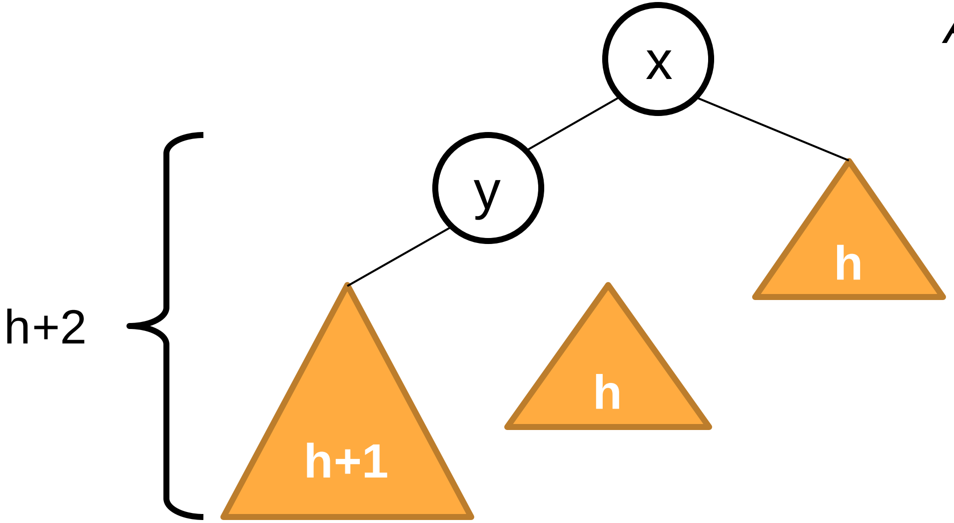
- Suppose after insertion, root has balance factor -2



*Assume **both subtrees have AVL property**, so only violation is at root.*

Correcting Balance via Rotation (Case 1)

- Suppose after insertion, root has balance factor -2

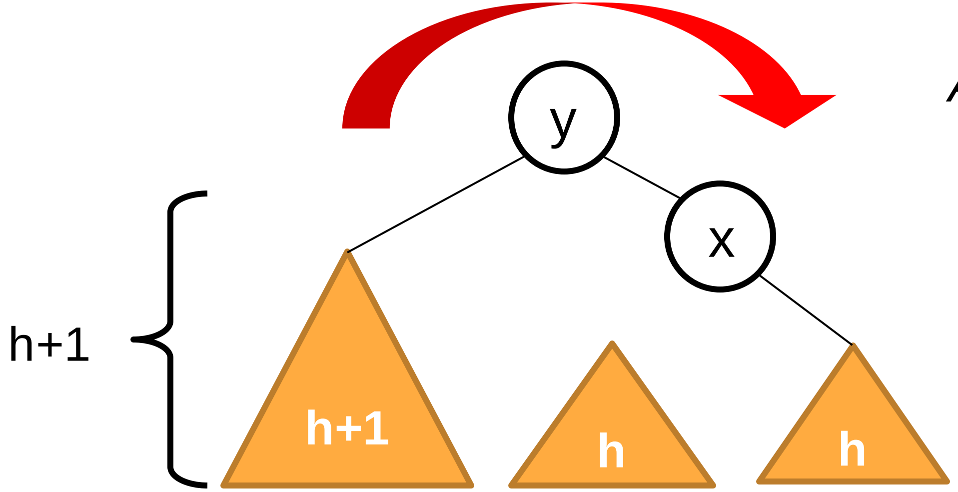


*Assume **both subtrees have AVL property**, so only violation is at root.*

rotate right

Correcting Balance via Rotation (Case 1)

- Suppose after insertion, root has balance factor -2

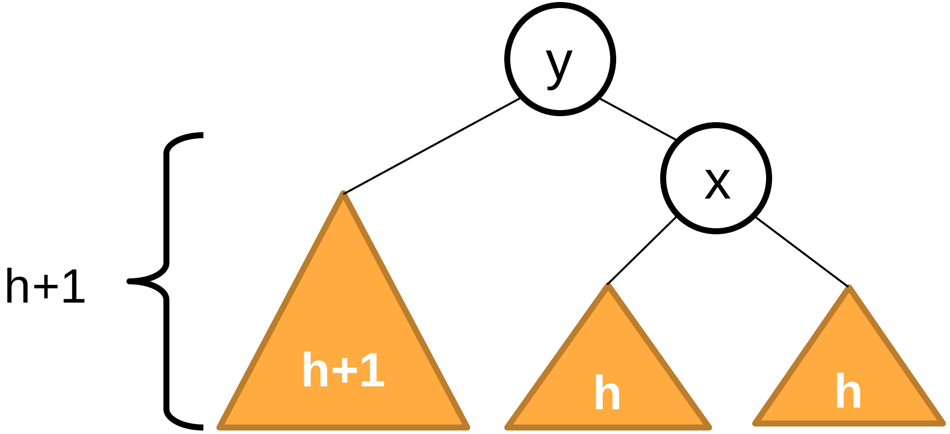


*Assume **both subtrees have AVL property**, so only violation is at root.*

rotate right

Correcting Balance via Rotation (Case 1)

- Suppose after insertion, root has balance factor -2



*After rotation, both subtrees have height **h+1** → root's balance factor is now **0**.*

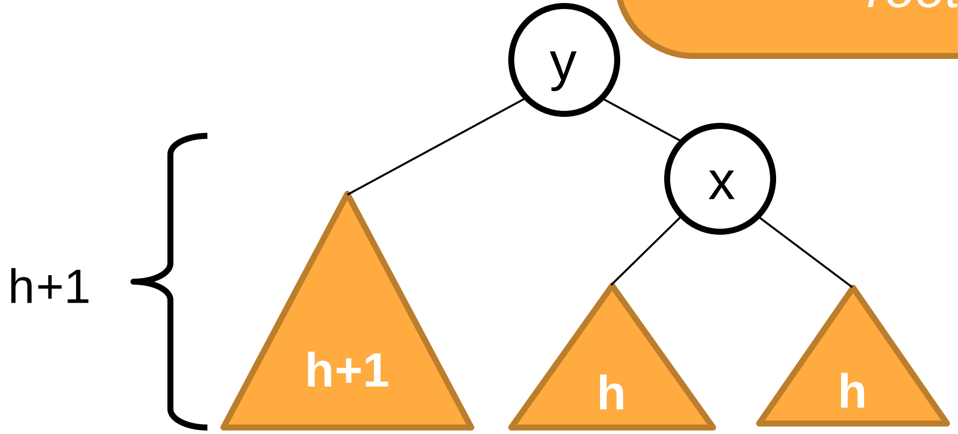
AVL PROPERTY RESTORED!

rotate right

Correcting Balance vi

- Suppose after insertion, root

Right rotation also restores AVL property if **both** subtrees of y have height h+1 before rebalancing, which could happen if we **remove** a node from x's right subtree. (*Final balance of root is then +1, not 0.*)



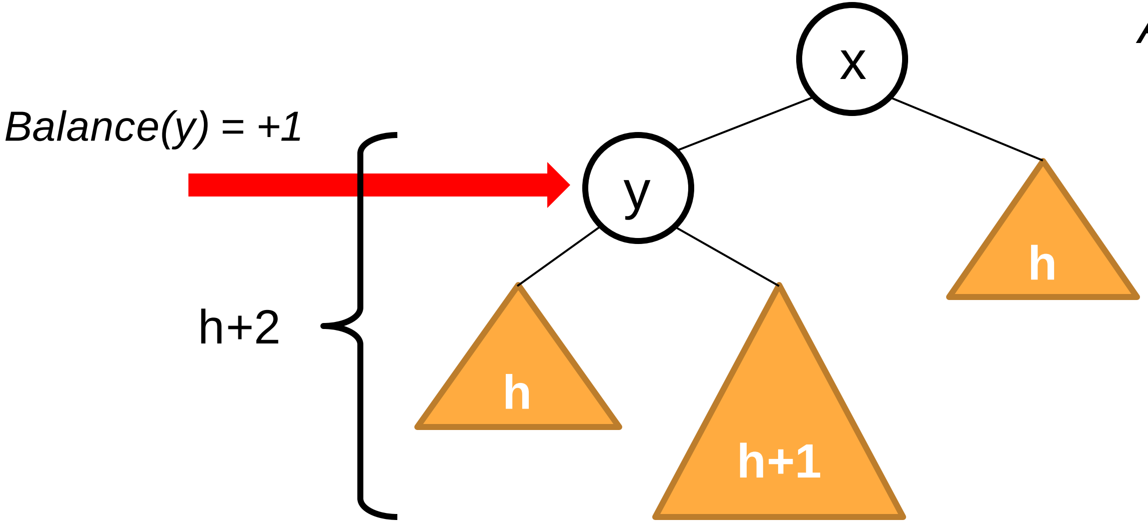
*have height **h+1** → root's balance factor is now **0**.*

AVL PROPERTY RESTORED!

rotate right

Correcting Balance via Rotation (Case 2)

- Suppose after insertion, root has balance factor -2



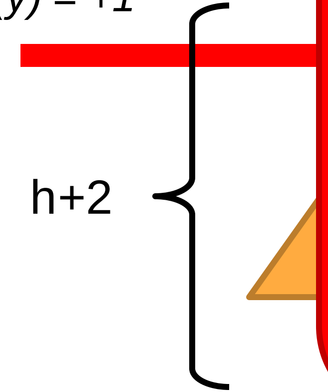
*Assume **both subtrees have AVL property**, so only violation is at root.*

Correcting Balance via Rotation (Case 2)

- Suppose after insertion, root has balance factor -2

*Assume **both subtrees have AVL property**, so only violation is at root.*

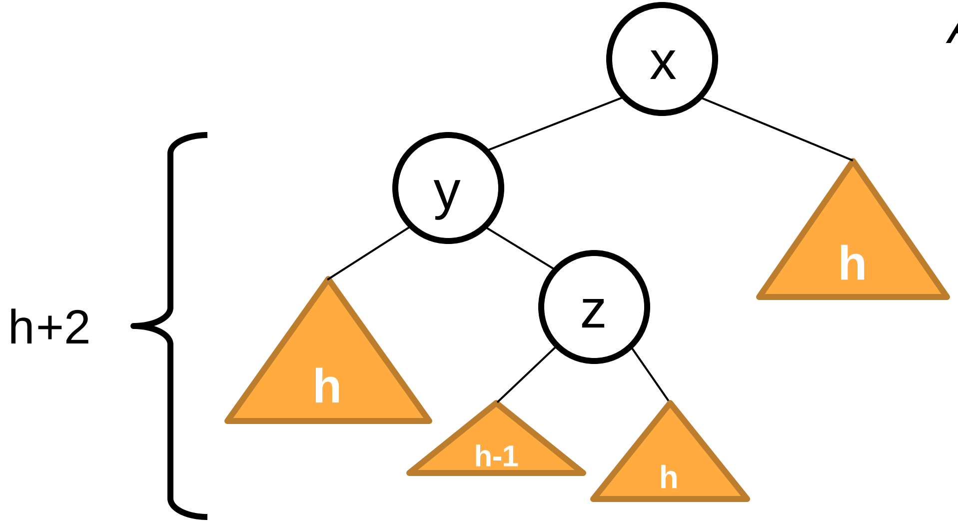
Balance(y) = +1



Right rotation alone **does not fix the problem** – makes balance +2. (Try it!)

Correcting Balance via Rotation (Case 2a)

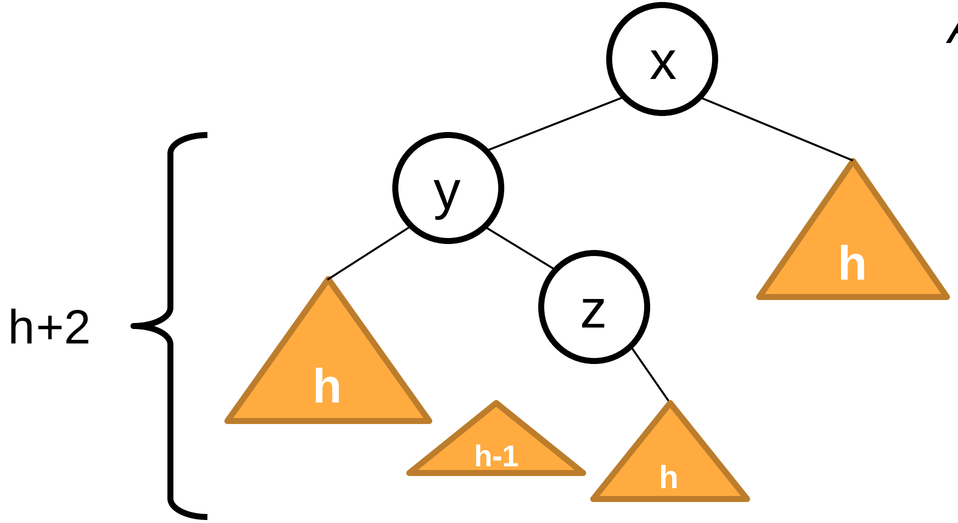
- Suppose after insertion, root has balance factor -2



*Assume **both subtrees have AVL property**, so only violation is at root.*

Correcting Balance via Rotation (Case 2a)

- Suppose after insertion, root has balance factor -2

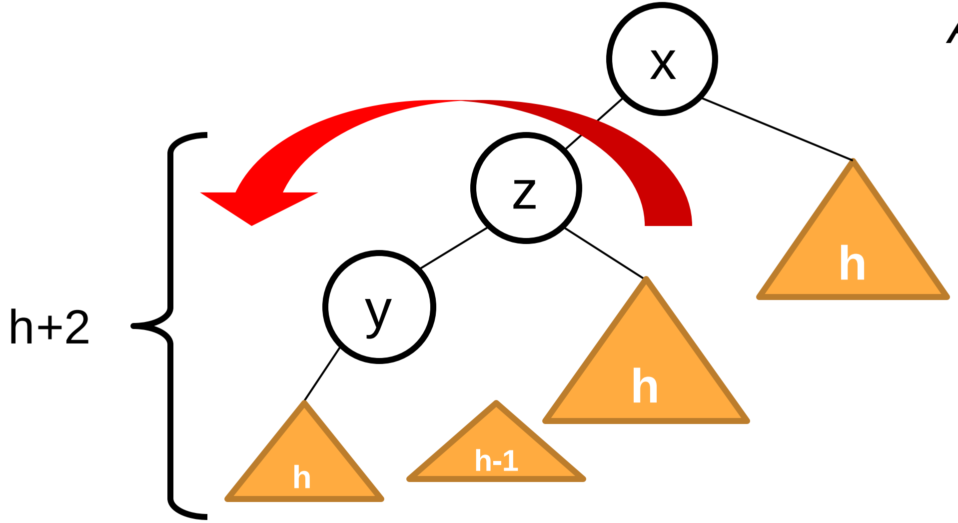


*Assume **both subtrees have AVL property**, so only violation is at root.*

*rotate **left***

Correcting Balance via Rotation (Case 2a)

- Suppose after insertion, root has balance factor -2

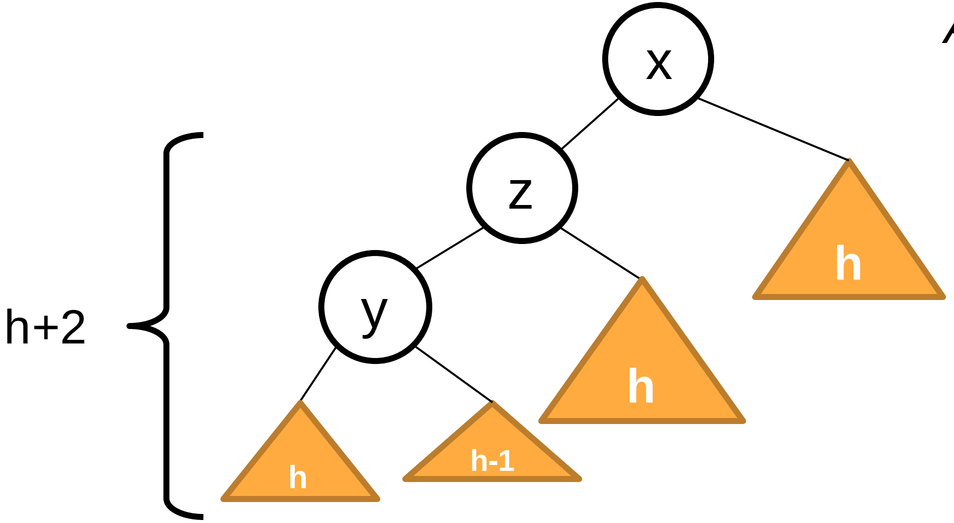


*Assume **both subtrees have AVL property**, so only violation is at root.*

*rotate **left***

Correcting Balance via Rotation (Case 2a)

- Suppose after insertion, root has balance factor -2

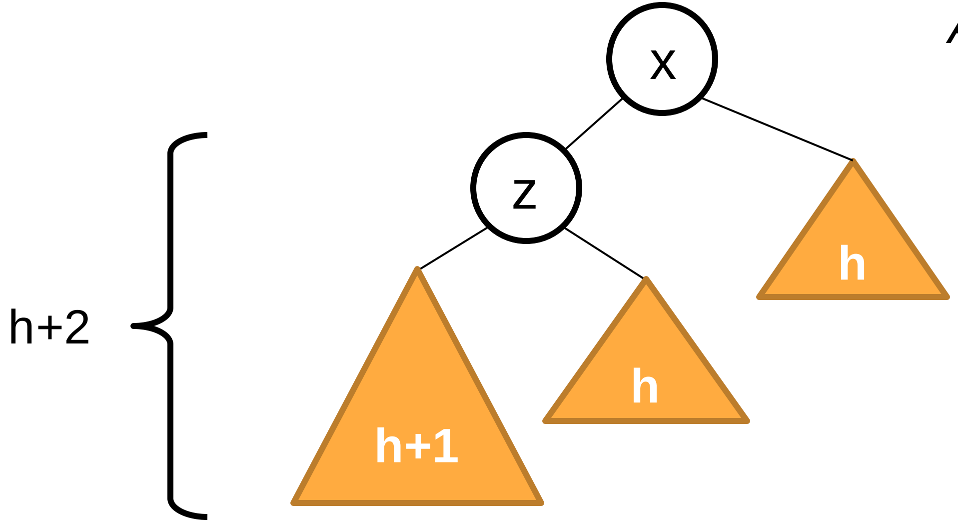


*Assume **both subtrees have AVL property**, so only violation is at root.*

*rotate **left***

Correcting Balance via Rotation (Case 2a)

- Suppose after insertion, root has balance factor -2

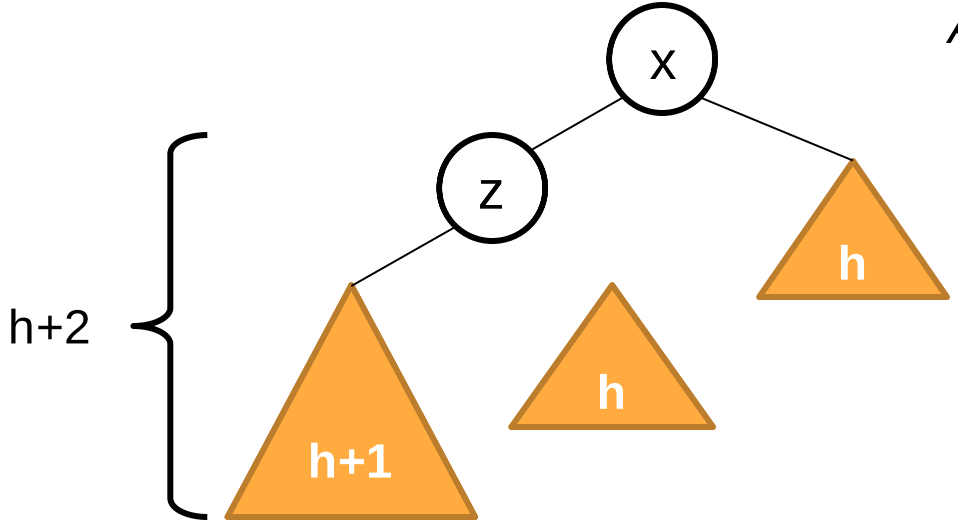


*Assume **both subtrees have AVL property**, so only violation is at root.*

NOW WE ARE IN CASE 1 AGAIN! Rotate x right to restore AVL property.

Correcting Balance via Rotation (Case 2a)

- Suppose after insertion, root has balance factor -2

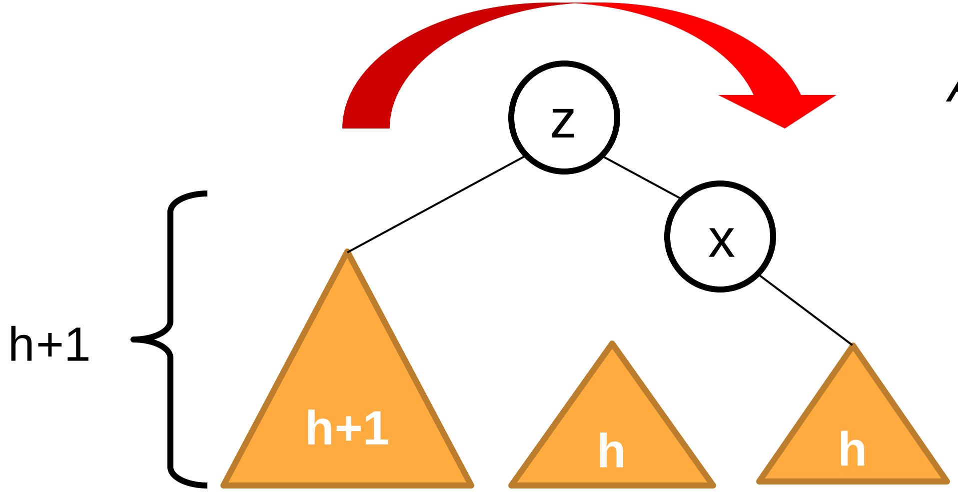


*Assume **both subtrees have AVL property**, so only violation is at root.*

rotate right

Correcting Balance via Rotation (Case 2a)

- Suppose after insertion, root has balance factor -2

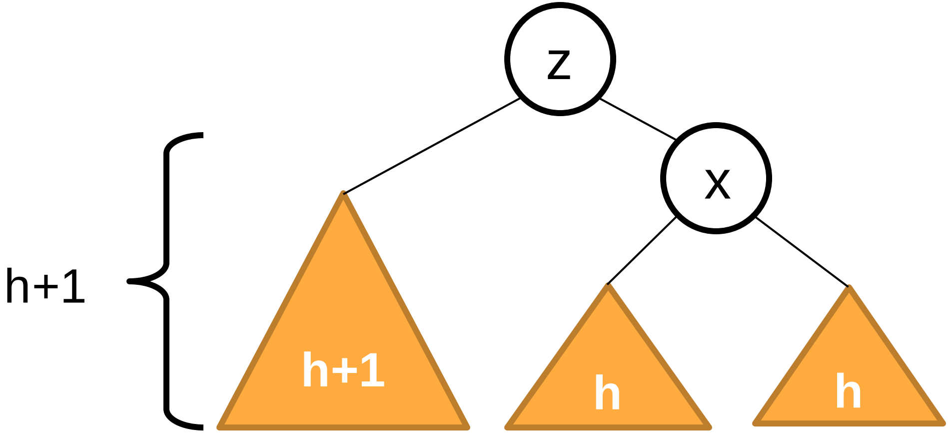


*Assume **both subtrees have AVL property**, so only violation is at root.*

rotate right

Correcting Balance via Rotation (Case 2a)

- Suppose after insertion, root has balance factor -2

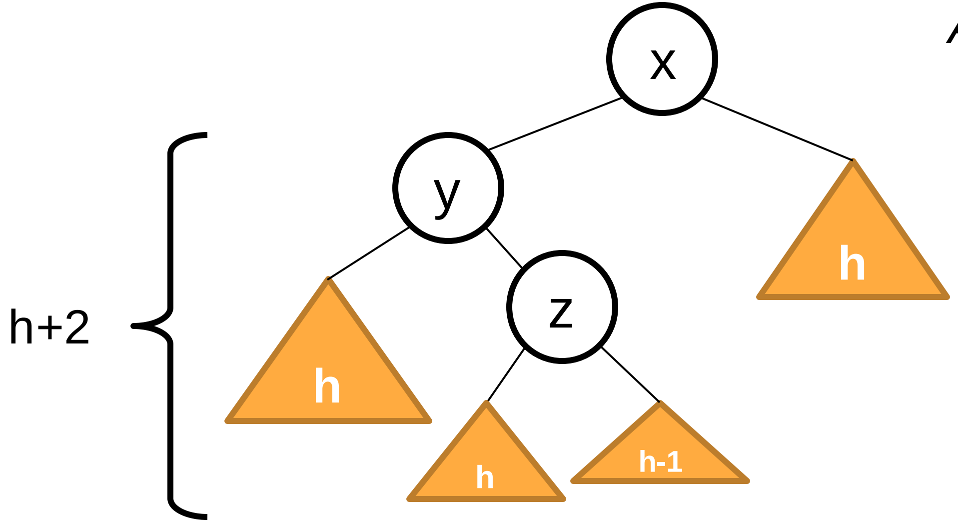


After rotation, both subtrees have height ***h+1*** *→ root's balance factor is now* ***0****.*

AVL PROPERTY RESTORED!

Correcting Balance via Rotation (Case 2b)

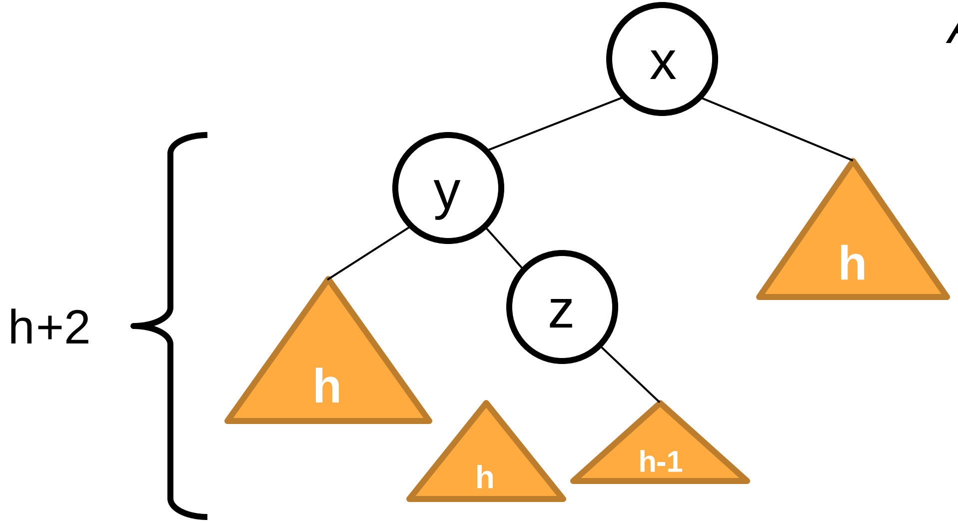
- Suppose after insertion, root has balance factor -2



*Assume **both subtrees have AVL property**, so only violation is at root.*

Correcting Balance via Rotation (Case 2b)

- Suppose after insertion, root has balance factor -2

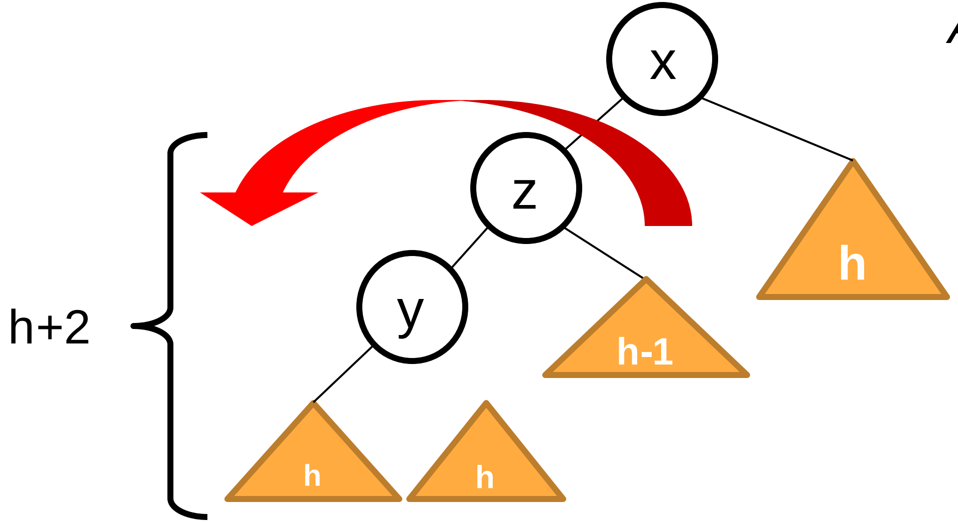


*Assume **both subtrees have AVL property**, so only violation is at root.*

*rotate **left***

Correcting Balance via Rotation (Case 2b)

- Suppose after insertion, root has balance factor -2

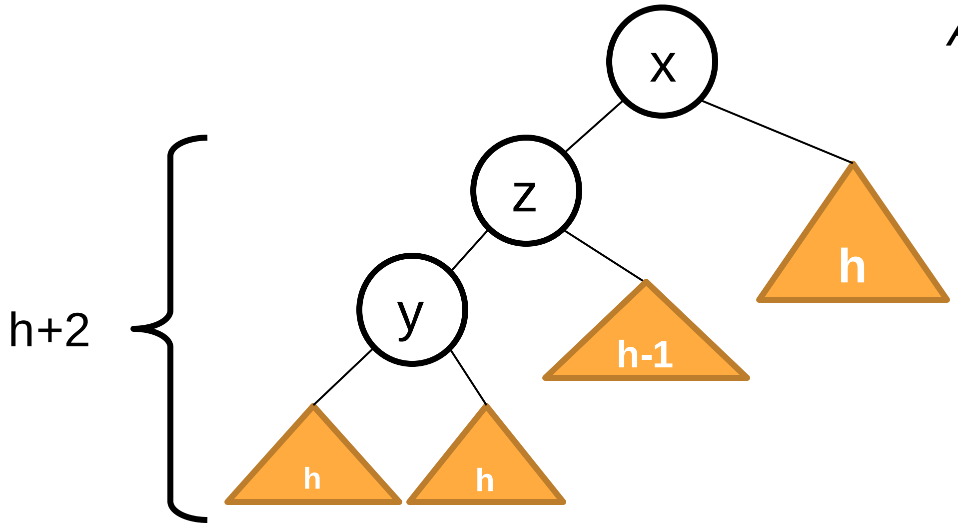


*Assume **both subtrees have AVL property**, so only violation is at root.*

*rotate **left***

Correcting Balance via Rotation (Case 2b)

- Suppose after insertion, root has balance factor -2

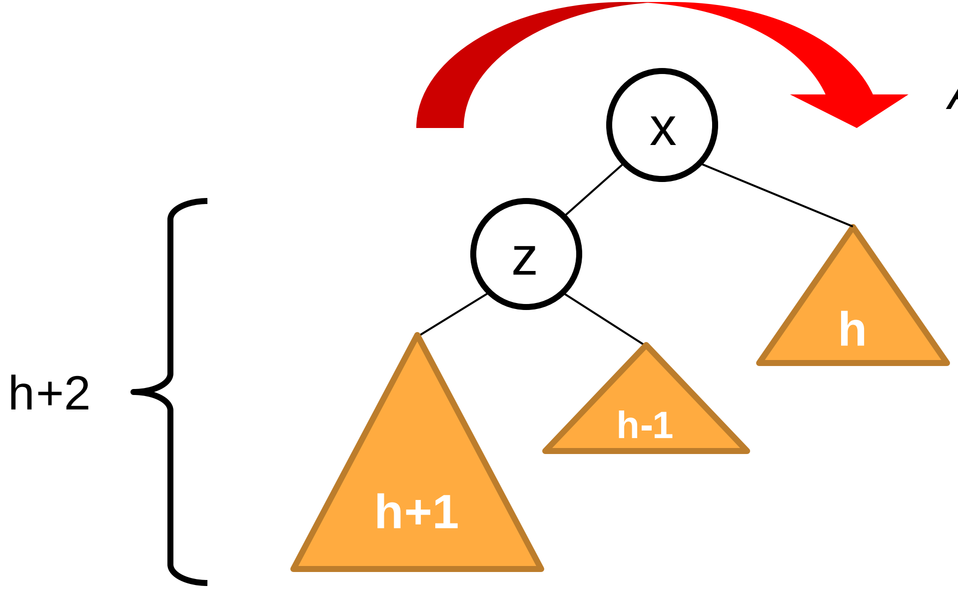


*Assume **both subtrees have AVL property**, so only violation is at root.*

*rotate **left***

Correcting Balance via Rotation (Case 2b)

- Suppose after insertion, root has balance factor -2



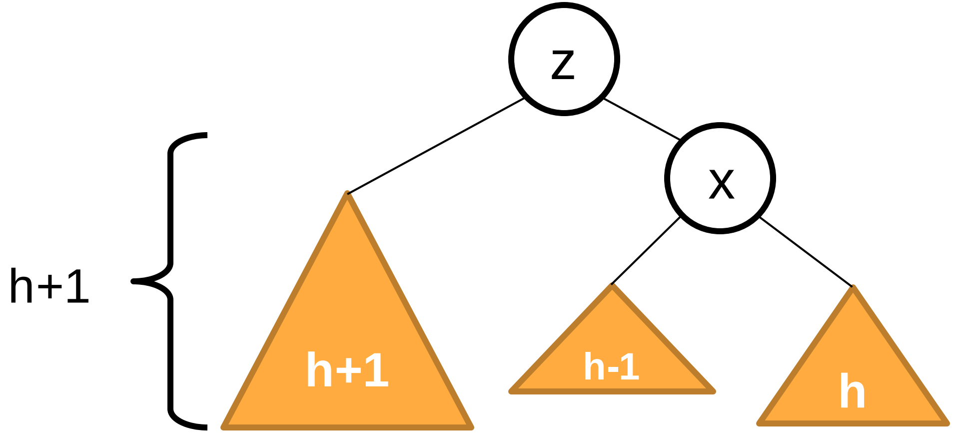
*Assume **both subtrees have AVL property**, so only violation is at root.*

This is not quite Case 1, but…

*rotate **right***

Correcting Balance via Rotation (Case 2b)

- Suppose after insertion, root has balance factor -2



After rotation, both subtrees have height ***h+1*** *→ root's balance factor is now* ***0****.*

A right rotation still restores the AVL property.

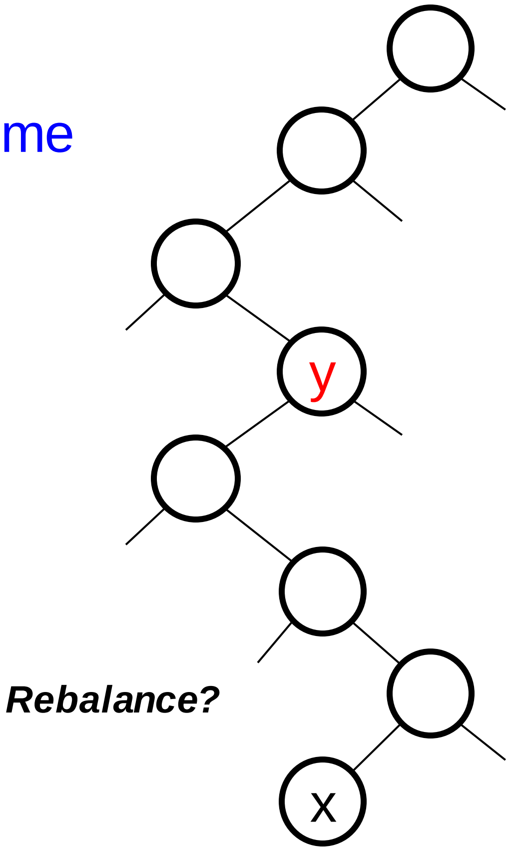
Summary of AVL Rebalancing Algorithm

- If root's balance factor is -2
-
- If root.left has balance factor +1 // CASE 2
- perform left rotate on root.left
- Perform right rotate on root // CASE 1
-
- Else if root's balance factor is +2, do opposite rotations, applying Case 2 to root.right
-
- *(If $-1 \leq$ balance factor ≤ 1, don't need to do anything)*

When Do We Rebalance?

- Inserting or removing a node x *may* unbalance some subtree rooted at *some* ancestor y of x.

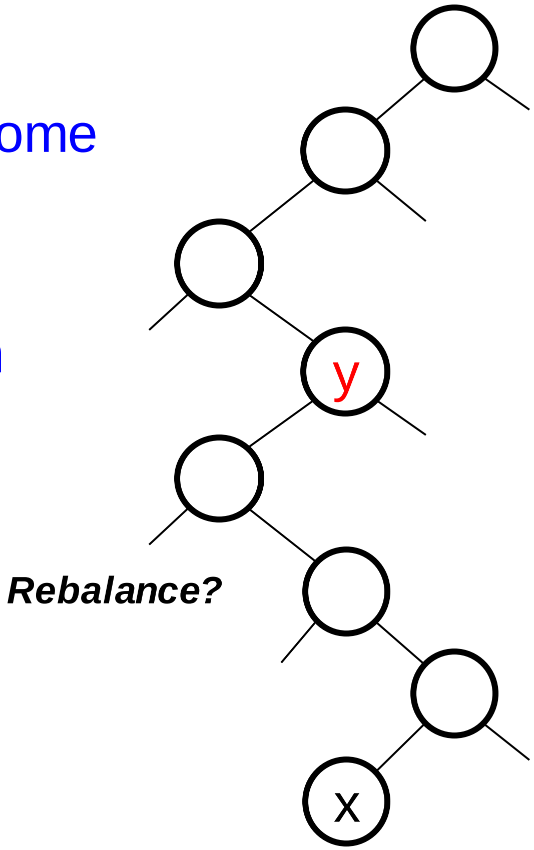
- To find y, try to rebalance subtree rooted at each ancestor of x moving up the tree, starting with its parent.



When Do We Rebalance?

- Inserting or removing a node x *may* unbalance some subtree rooted at *some* ancestor y of x.

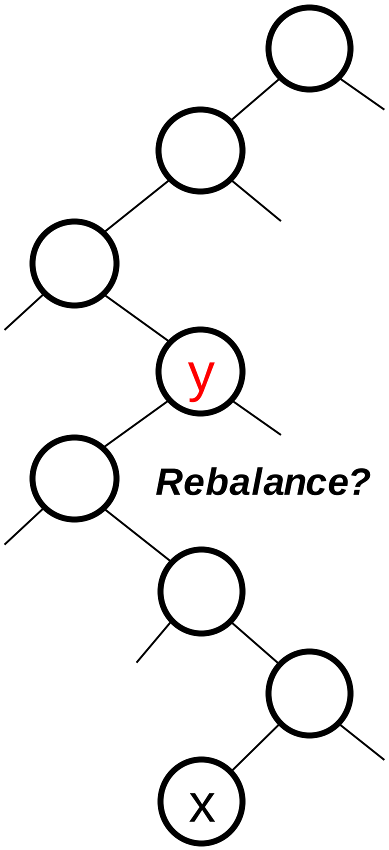
- To find y, try to rebalance subtree rooted at each ancestor of x moving up the tree, starting with its parent.



When Do We Rebalance?

- Inserting or removing a node x *may* unbalance some subtree rooted at *some* ancestor y of x.

- To find y, try to rebalance subtree rooted at each ancestor of x moving up the tree, starting with its parent.

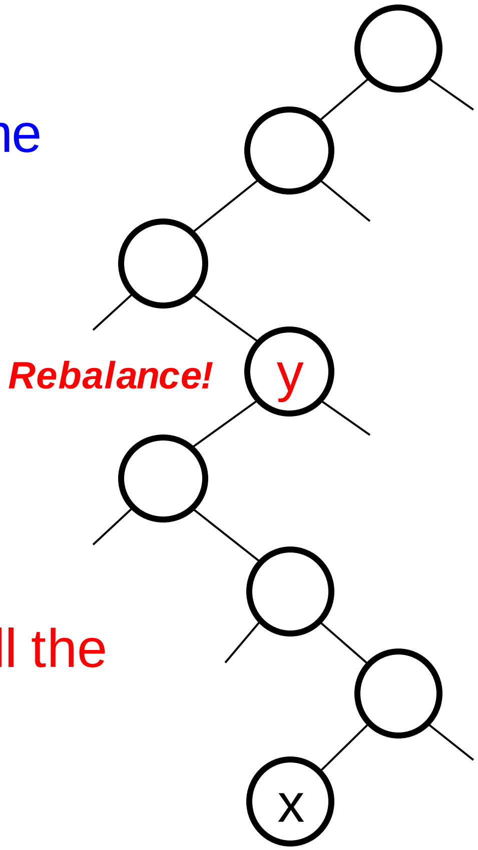


When Do We Rebalance?

- Inserting or removing a node x *may* unbalance some subtree rooted at *some* ancestor y of x.

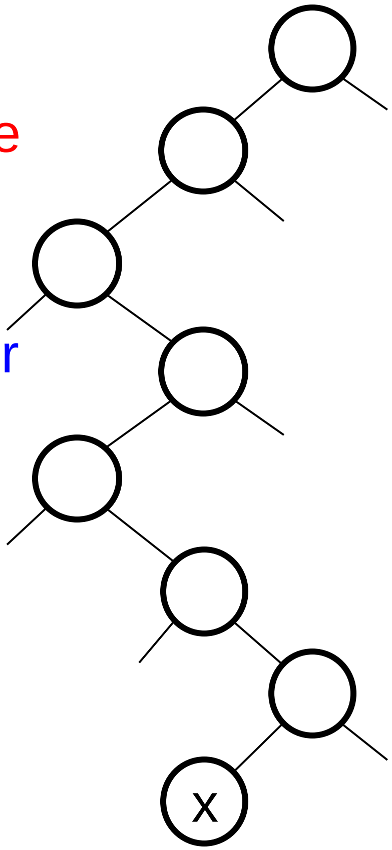
- To find y, try to rebalance subtree rooted at each ancestor of x moving up the tree, starting with its parent.

- **Question**: do we have to keep checking balance all the way to the root after rebalancing at y?



When Do We Rebalance?

- **Question**: do we have to keep checking balance all the way to the root after rebalancing at y?
- For **insertions**, can prove that we may safely stop after first rebalancing that changes tree.
- For **deletions**, we must continue to check and, if needed, rebalance all the way to the root.
- *(Asymptotically, no harm in continuing up to root in **both** cases – total cost is still only Θ(h))*



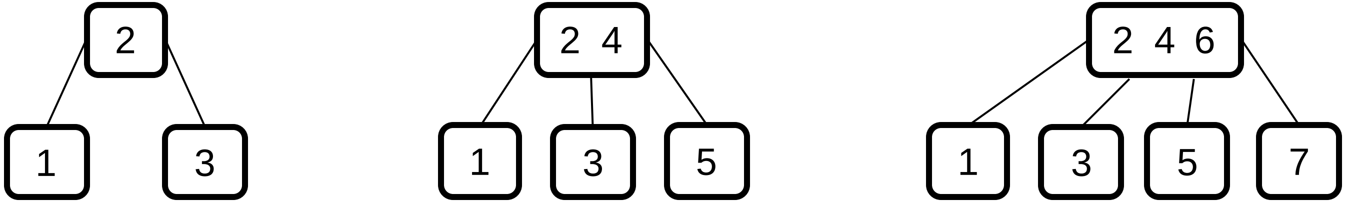
Cost of AVL Tree Maintenance

- As we saw in studio, maintaining height on insert/remove costs O(h)
- Rotation is O(1) operation, so check and rebalance is O(1) / level
- Hence, total cost of rebalance on insert/remove is O(h).
- Since h is Θ(log n) for an AVL tree, added cost is only O(log n).
- **→ All BST ops are now Θ(log n)**

There's More Than One Way To Balance a Tree…

An Alternative (Not Binary) Tree

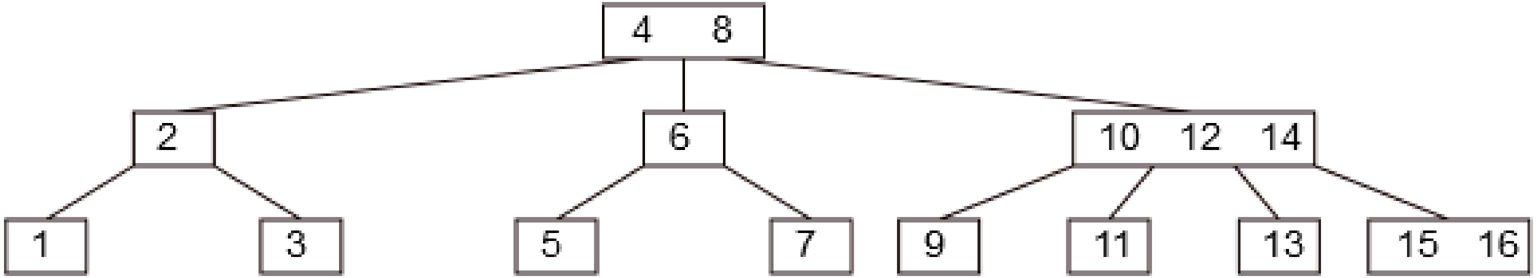
- We will allow each node of a tree to hold 1, 2, or 3 keys.
- A non-leaf node with t keys has t+1 children (2, 3, or 4 children).



- Natural analog of BST property holds between root and its subtrees.

2-3-4 Trees

- A **2-3-4 tree** is a tree in which each node holds 1, 2, or 3 keys as described…
- … and *every path from root to bottom of the tree has same height*.

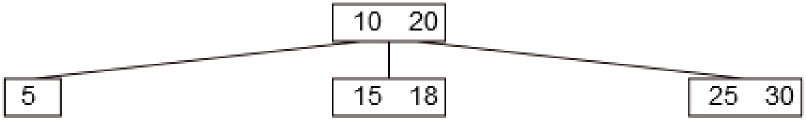


2-3-4 Trees Are Balanced

- **Claim**: A 2-3-4 tree of height h has at least $2^{h+1}-1$ keys
- **Pf**: if every node has 1 key (minimum possible), "same height" property implies that tree is a *complete binary tree* of height h. QED
- → Every 2-3-4 tree with n keys has height **O(log n).**

Maintaining 2-3-4 Tree Properties

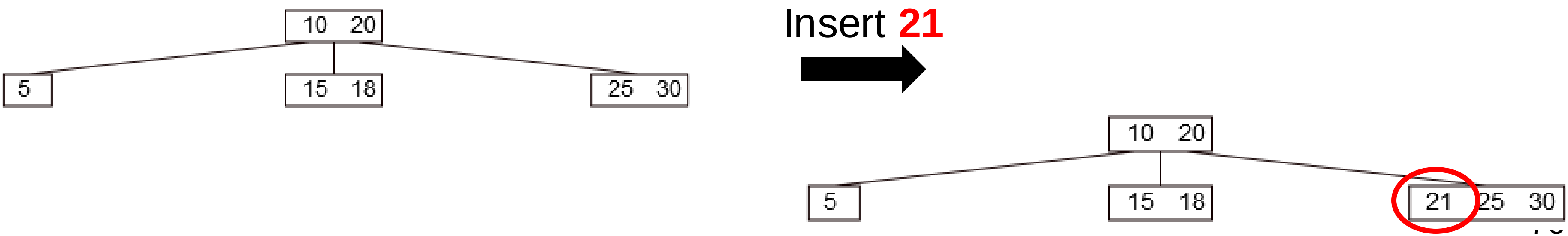
- As we perform insertions and deletions in a 2-3-4 tree…
- Must maintain that *every path from root to bottom has same height*
- This means we can't just create a new leaf for each insertion. We instead try to insert each new value into an existing leaf.



Insert **21**

Maintaining 2-3-4 Tree Properties

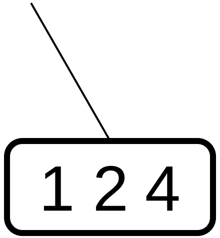
- As we perform insertions and deletions in a 2-3-4 tree…
- Must maintain that *every path from root to bottom has same height*
- This means we can't just create a new leaf for each insertion. We instead try to insert each new value into an existing leaf.



The Problem With Insertion

- What if the leaf we want to insert into is full (has three keys)?

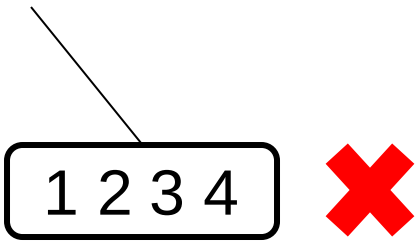
Insert 3



The Problem With Insertion

- What if the leaf we want to insert into is full (has three keys)?

Insert 3

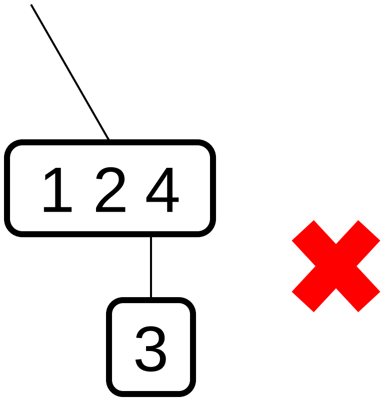


May not (permanently) overload a node.

The Problem With Insertion

- What if the leaf we want to insert into is full (has three keys)?

Insert 3

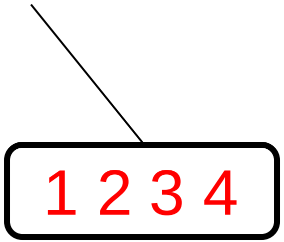


May not create one leaf deeper than the rest of the tree.

Solution: Split the Leaf

- Split overloaded node into 2 nodes; push median key up to parent

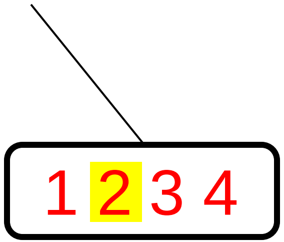
Insert 3



Solution: Split the Leaf

- Split overloaded node into 2 nodes; push median key up to parent

Insert 3

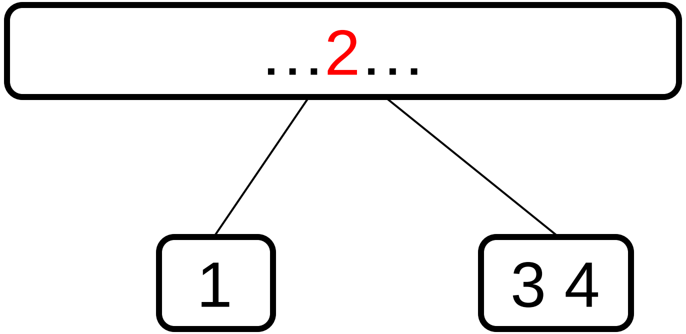


By "**median**", I mean key #2 out of 4 (in order) in the overloaded node.

Solution: Split the Leaf

- Split overloaded node into 2 nodes; push median key up to parent

Insert 3

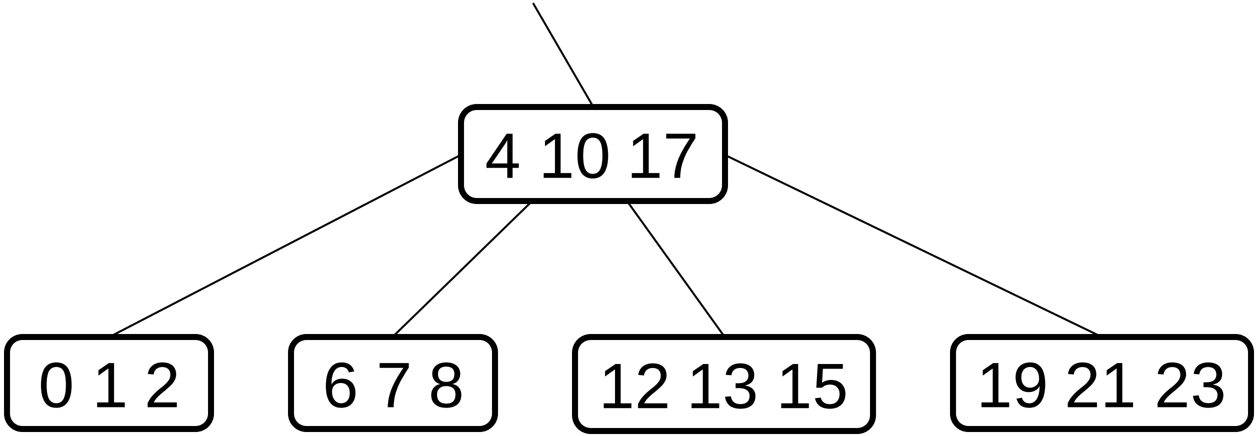


- (Moving a key to parent creates one more slot for a child pointer.)

What If the Parent Is Also Full?

- If moving a key to parent would overload it, recursively split parent!

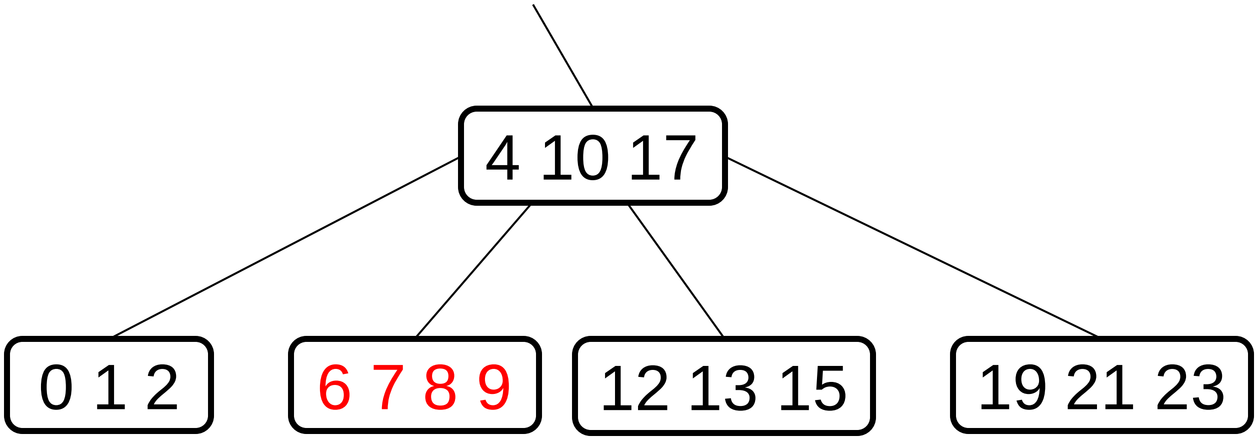
Insert 9



What If the Parent Is Also Full?

- If moving a key to parent would overload it, recursively split parent!

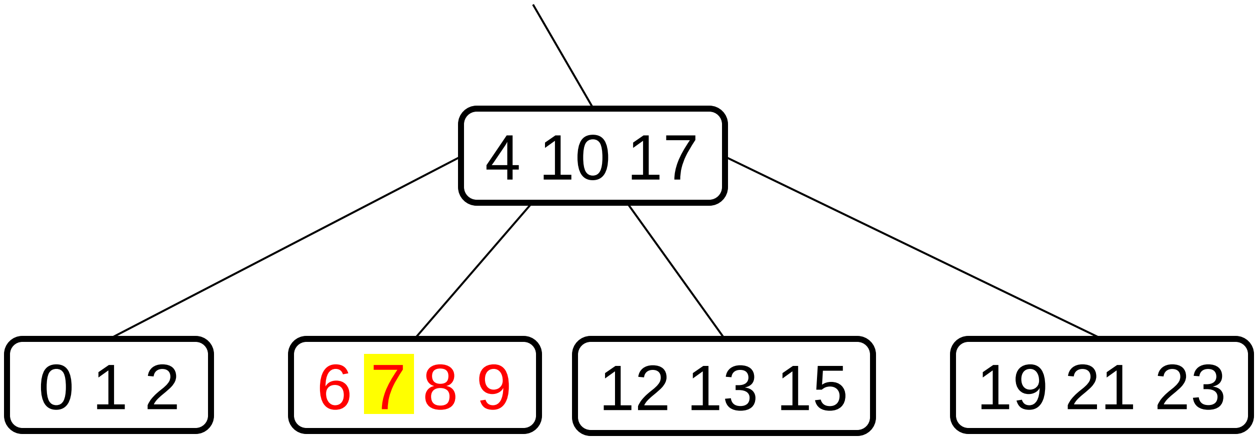
Insert 9



What If the Parent Is Also Full?

- If moving a key to parent would overload it, recursively split parent!

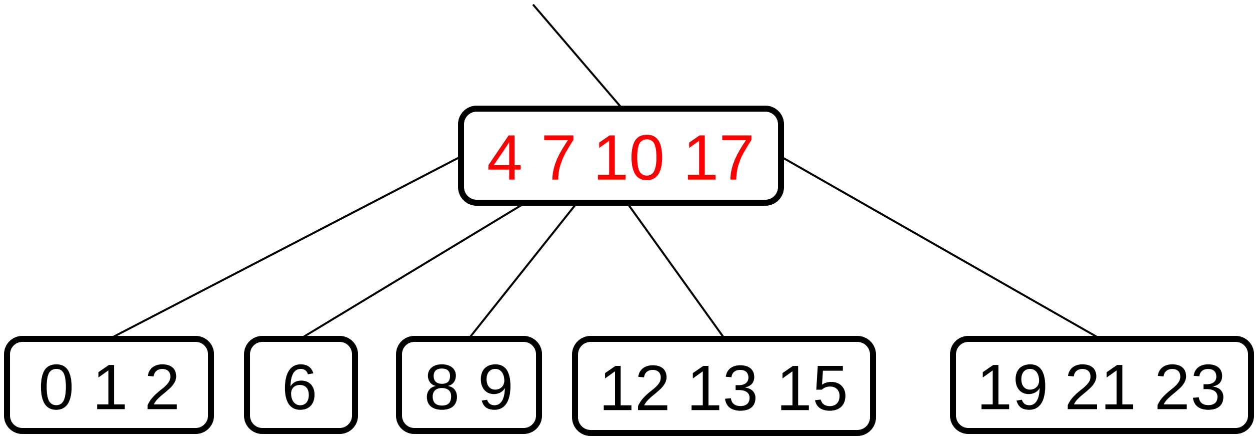
Insert 9



What If the Parent Is Also Full?

- If moving a key to parent would overload it, recursively split parent!

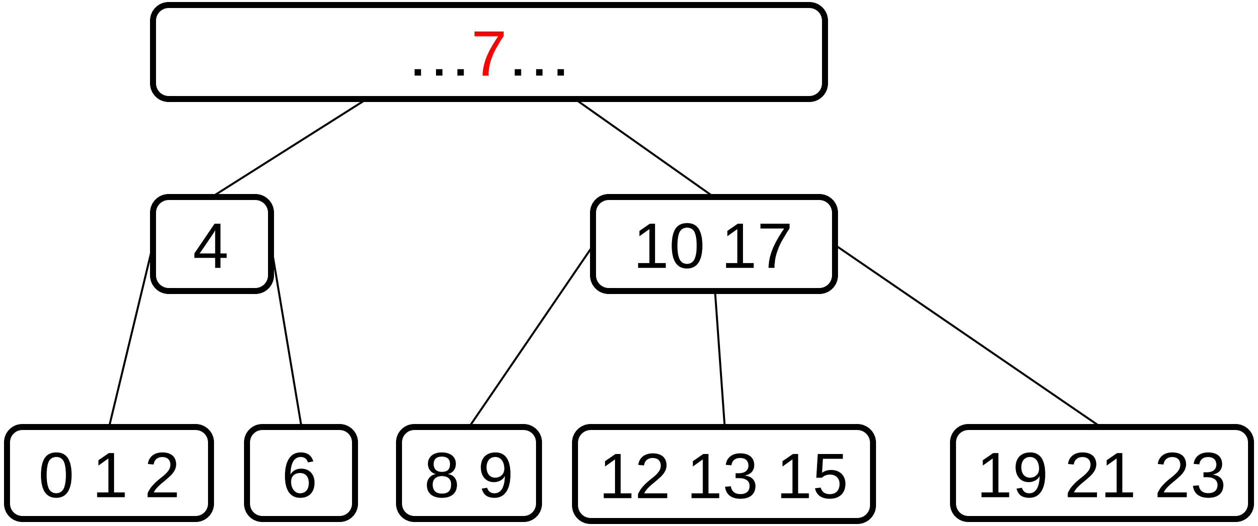
Insert 9



What If the Parent Is Also Full?

- If moving a key to parent would overload it, recursively split parent!

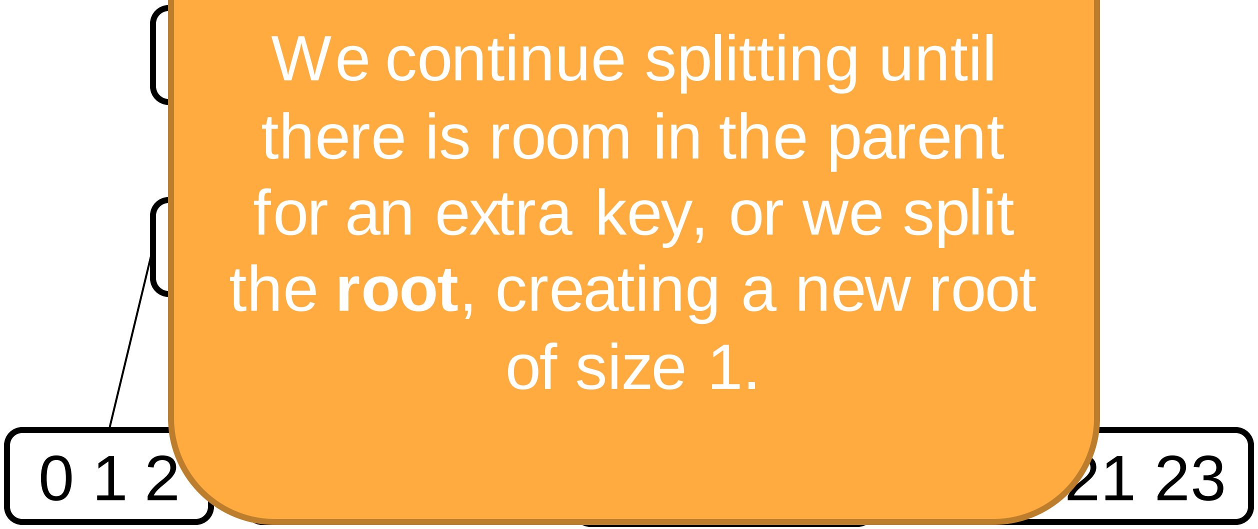
Insert 9



What If the Parent Is Also Full?

- If moving a key ly split parent!

Insert 9



We continue splitting until there is room in the parent for an extra key, or we split the **root**, creating a new root of size 1.

Cost of Insertion

- Splitting is an O(1) time operation.
- We might have to split at each level of tree.
- Hence, 2-3-4 tree insertion is still worst-case $\Theta(\log n)$

What About Deletion?

- Deletion from a 2-3-4 tree is more complex.
- Studio 11 works out some of the details.
- Still $\Theta(\log n)$.

What Good Is a 2-3-4 Tree?

- If your tree lives in external memory (the cloud?)…

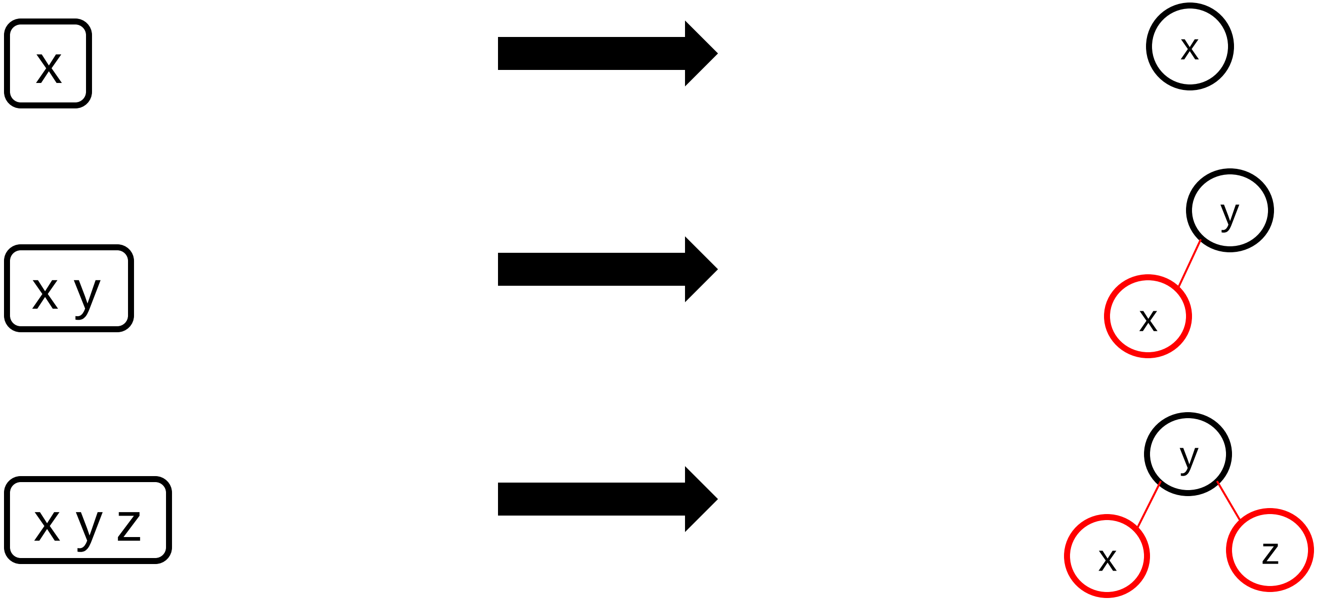
- Can generalize 2-3-4 trees to **B-trees**, which work the same but store hundreds or thousands of keys in each node.

- If you would prefer to use binary trees…

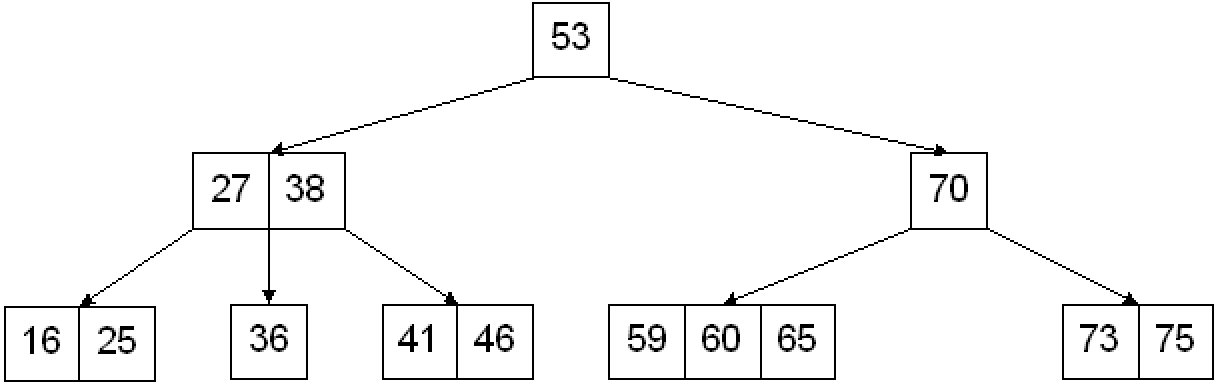
- *There's a trick to representing 2-3-4 trees as binary trees.*

Simulating a 2-3-4 Tree with a Binary Tree

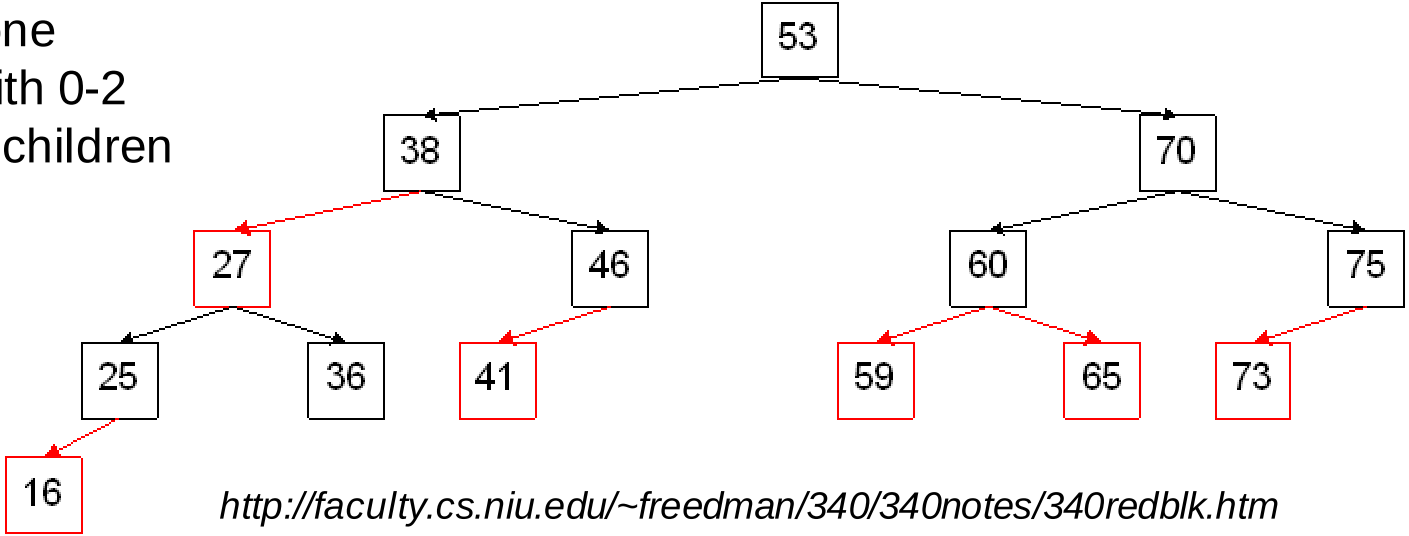
- *Idea:* Convert each node of a 2-3-4 tree to a little binary tree.



A Larger Example



Every node of 2-3-4 tree maps to one **black node** with 0-2 **red nodes** as children



http://faculty.cs.niu.edu/~freedman/340/340notes/340redblk.htm

General Construction: Red-Black Trees

- A red-black tree is a binary representation of a 2-3-4 tree.

- Hence, we know that
  - Same # of black nodes on path from root to every leaf
  - Cannot have a red child of a red node.

- → red-black trees have height O(log n).

More on Red-Black Trees

- Red-black tree properties can be efficiently maintained under insertion/removal of nodes.

- *(Algorithms are kind of gross – see your text)*

- Red-black trees are probably the *most widely used balanced binary tree structure*. For example, Java ordered sets use them.

Which Balanced Binary Tree Should You Use?

- AVL, red-black (=234), left-leaning red-black (=23), scapegoat tree, …
- AVL is simpler to code than other trees but rotates more often.
- Ongoing fights over which red-black-like variant is best.
- Best option: *use someone else's implementation*.
- RB trees commonly found in Java, C++, and other standard libraries.