Balanced Trees: AVL Tree

Fatemeh Panahi
Department of Computer Science
University of Toronto
CSC263-Fall 2017
Lecture 4

Announcements

- $A1$ Solutions are out, $A2$ handout will be posted on Friday.

- Tutorial on Friday:

  Textbook exercise for tutorial:
  12.2-6, 12.2-7, 12.2-8, 12-2 and problem 6-3.
  Work on these exercises before the tutorial.
  Some examples on AVL Tree and Augmentation.

Today

- Augmentation
- AVL Tree (Not covered in CLRS)
  - Height and Balance factor
  - Operations
    - Search
    - Insert
    - Delete

Augmented Data Structures:

existing data structure modified to store additional information and/or perform additional operations.

Balanced BST

AVL tree, Red-Black tree, 2-3 tree, AA tree, ...

Reflect on AVL tree

- We "augmented" BST by storing additional information (the balance factor) at each node.
- The additional information enabled us to do additional cool things with the BST (keep the tree balanced).
- And we can maintain this additional information efficiently in modifying operations (within $O(\log n)$ time, without affecting the running time of Insert or Delete).

Augmentation: General Procedure

1. Choose data structure to augment
2. Determine additional information
3. Check additional information can be maintained, during each original operation, hopefully efficiently.
4. Implement new operations.

AVL Trees

Why AVL tree?

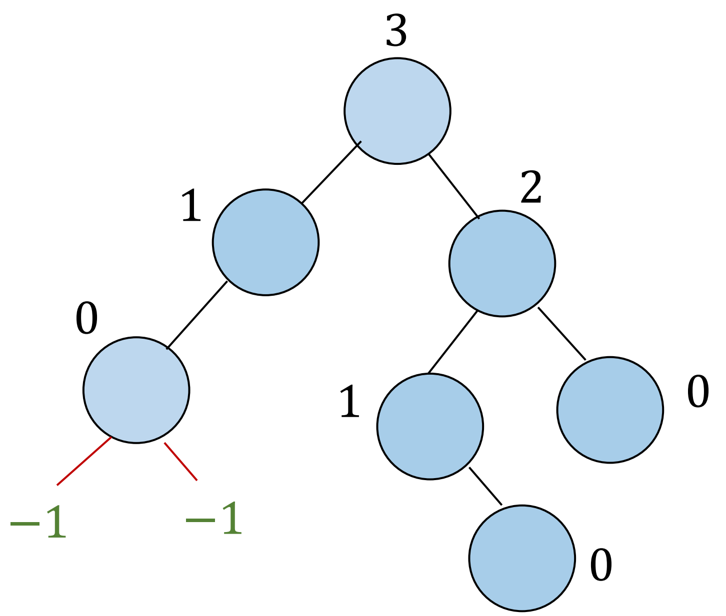
- First self-balancing BST to be invented.

Why is it called AVL?

- Invented by Georgy Adelson-Velsky and E. M. Landis in 1962.

Height of a BST Node

$$x.height \leftarrow 1 + max(x.left.height, x.right.height)$$



Trick: make NIL be an actual node with

. NIL.item = NIL,
. NIL.left = NIL,
. NIL.right = NIL,
. NIL.height = -1.

AVL Trees

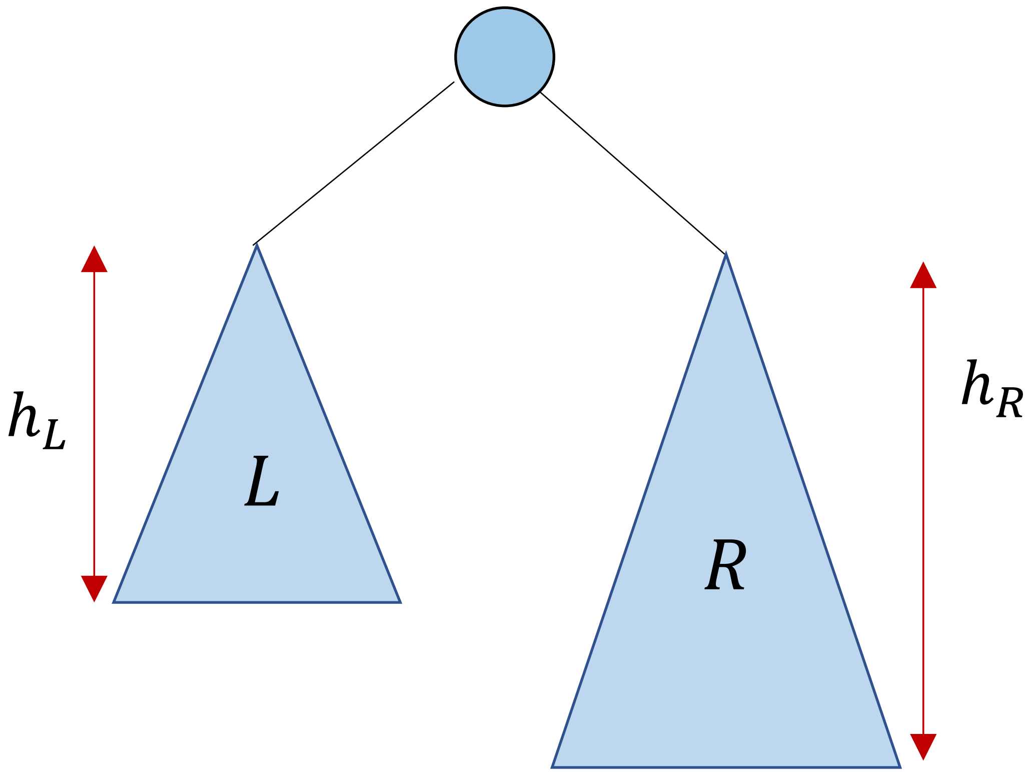
An extra attribute to each node in a BST: balance factor

$h_R(x)$: height of x's right subtree

$L_L(x)$: height of x's left subtree

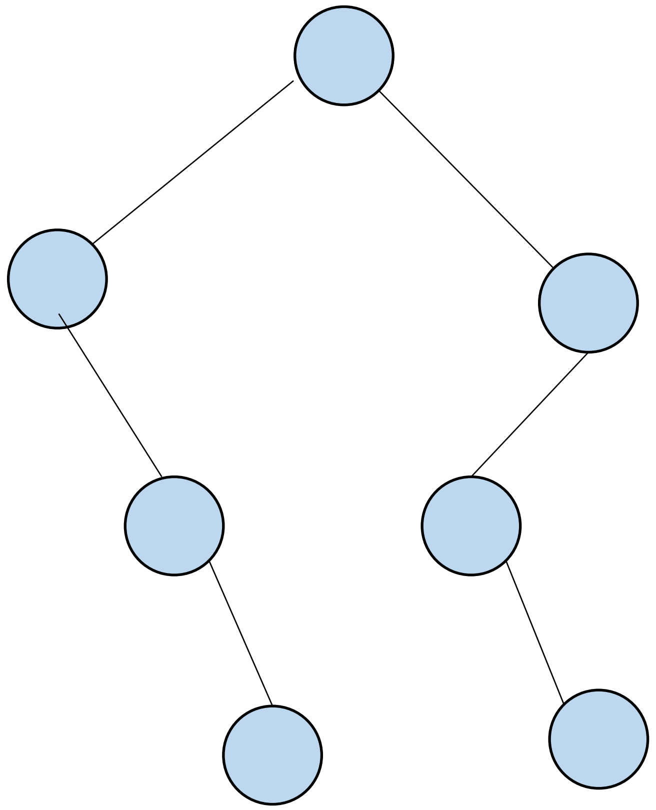
$BF(x) = h_R(x) - h_L(x)$

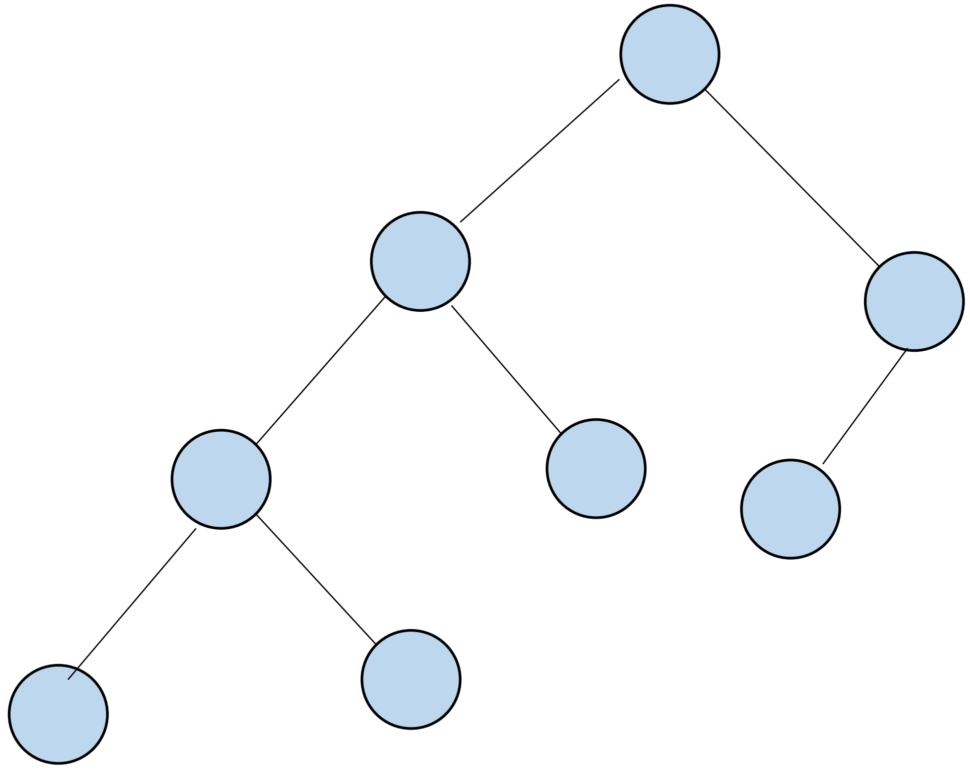
- $BF(x) = 0$: x is balanced
- $BF(x) = 1$: x is right-heavy
- $BF(x) = -1$: x is left-heavy
- $BF(x) > 1$ or < -1: x is imbalanced



AVL tree: definition

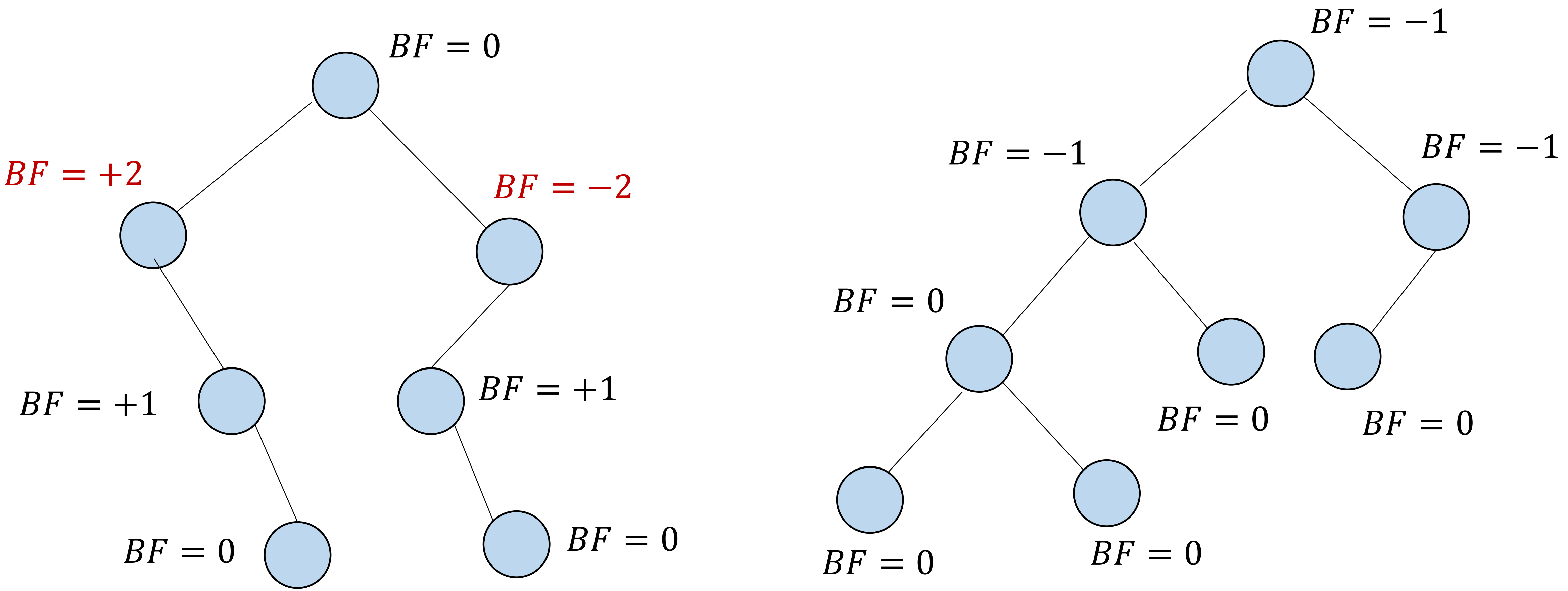
An AVL tree is a BST in which every node is balanced, right-heavy or left-heavy. i.e., the BF of every node must be 0, 1 or -1.





AVL tree: definition

An AVL tree is a BST in which every node is balanced, right-heavy or left-heavy. i.e., the BF of every node must be 0, 1 or -1.



AVL Trees - height

If there are n nodes, what is the maximum possible height?

$\Leftrightarrow$

If the height is h, what is the minimum possible number of nodes?

Let $M(h)$ be the minimum number of nodes in any AVL tree of height h. Then
$M(0) = 1, M(1) = 2$, and $M(h) = 1 + M(h-1) + M(h-2)$ for $h \geq 2$.

$fib(0) = 0, fib(1) = 1, fib(2) = 1, \ fib(3) = 2, \quad fib(h) = \frac{\phi^h - (1-\phi)^h}{\sqrt{5}} \quad \phi = \frac{1+\sqrt{5}}{2} \approx 1.61$

$M(h) = fib(h + 2) - 1$

AVL Trees - height

$fib(h) = \frac{\phi^h - (1-\phi)^h}{\sqrt{5}}$ $\quad \phi = \frac{1+\sqrt{5}}{2} \approx 1.61, 1 - \phi \approx -0.61$

$M(h) = fib(h + 2) - 1$

$$n \geq M(h) = \frac{(\phi^{h+2} - (1-\phi)^{h+2})}{\sqrt{5}} - 1 > \frac{\phi^{h+2}}{\sqrt{5}} - 1 - 1$$

$$\frac{\phi^{h+2}}{\sqrt{5}} - 2 < n \rightarrow \phi^{h+2} < \sqrt{5}(n+2) \rightarrow h + 2 < 1.44 \log n + 2 + constant$$

This implies $h \in O(\log n)$.

<u>Height of an AVL tree with n nodes is $O(\log n)$.</u>

Operations on AVL trees

- AVL-Search(root, k)
- AVL-Insert(root, x)
- AVL-Delete(root, x)

What should we consider for these operations:

- Before the operation, the BST is a valid AVL tree (precondition)
- After the operation, the BST must still be a valid AVL tree: so re-balancing may be needed.
- The balance factor attributes of some nodes need to be updated.

AVL-Search(root, k)

Search for key k in the AVL tree rooted at root

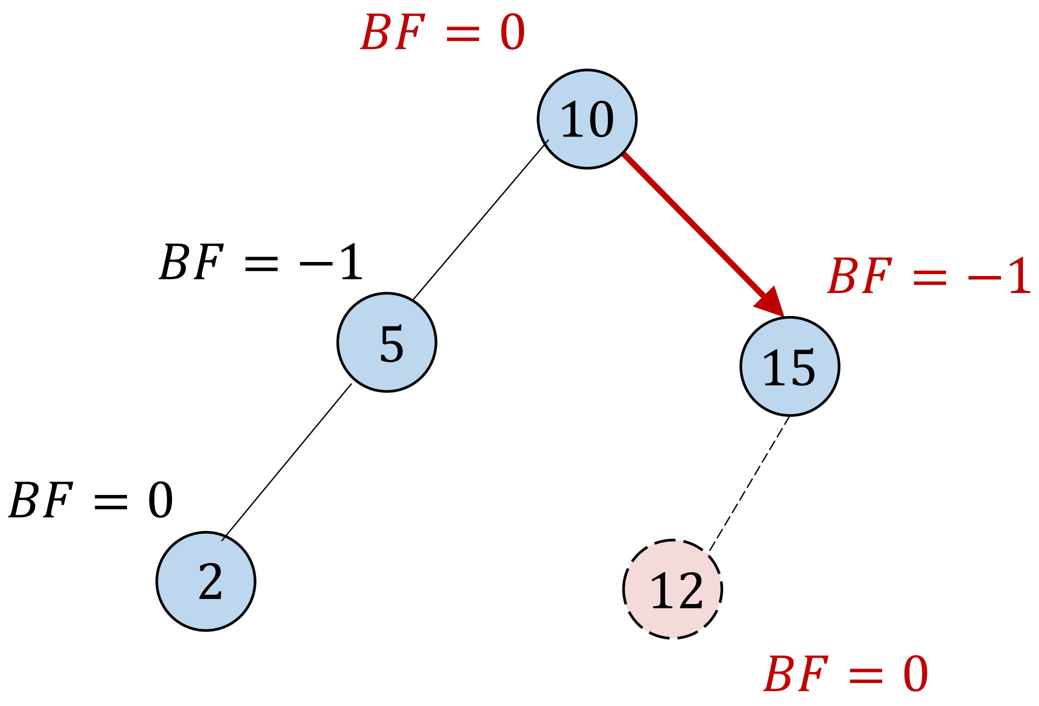
- First, do a TreeSearch(root, k) as in BST.
- Then, nothing else!

(The tree is still balanced since we didn’t change the tree)

AVL-Insert(root, x)

First, do a TreeInsert(root, x) as in BST, then update the balance factors

Example 1: Insert(12)

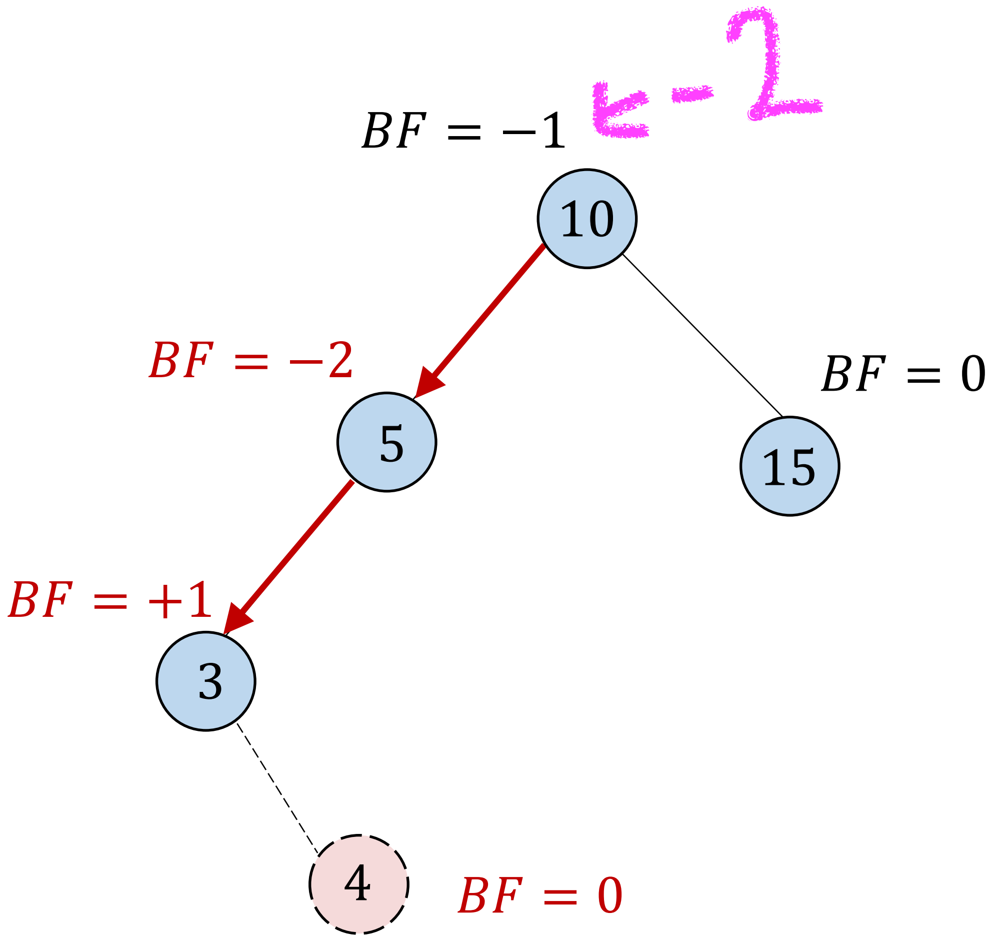


Still an AVL Tree,
Nothing has to change!

AVL-Insert(root, x)

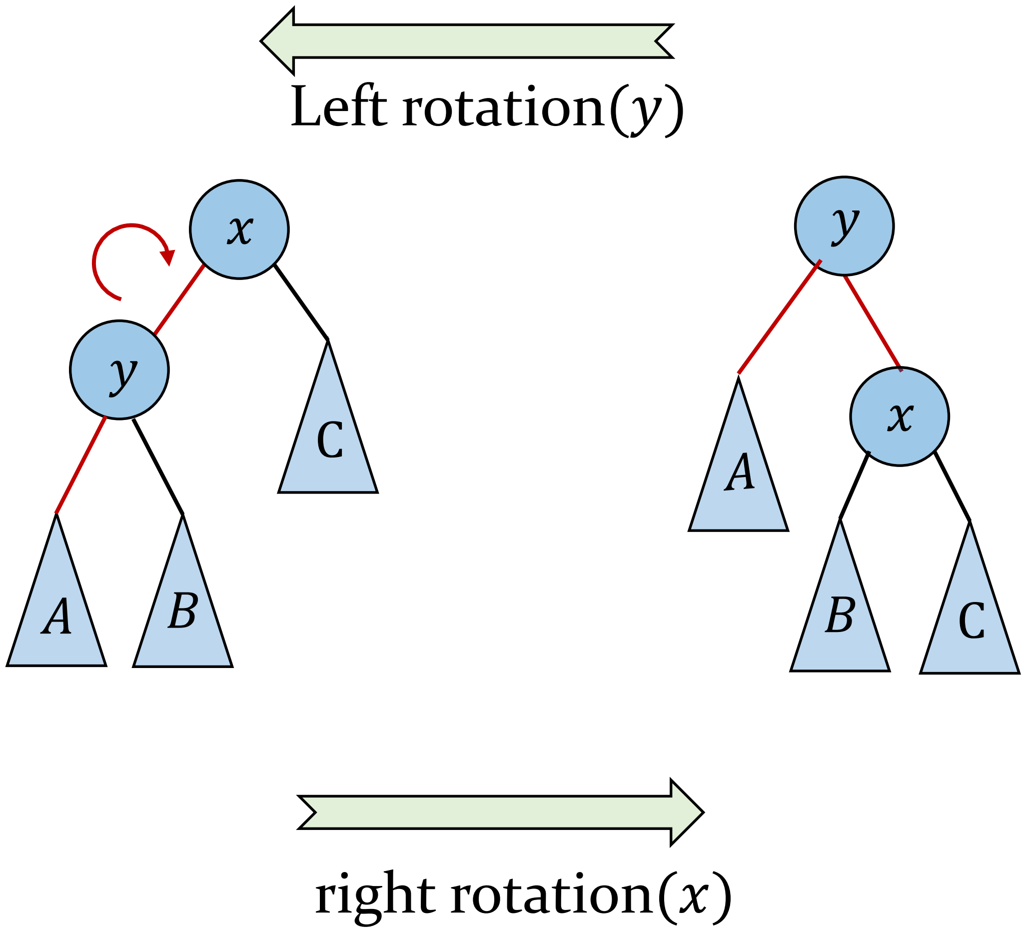
First, do a TreeInsert(root, x) as in BST, then update the balance factors

Example 1: Insert(4)



Not an AVL Tree anymore,
<u>Needs rebalancing!</u>

Basic move for rebalancing - Rotation



- Inorder walk of the tree is the same: $A\ y\ B\ x\ C$
- Three references to change: parent's left/right, y.left, x.right.
- Decreases depth of subtree A, increases depth of subtree C.
- Height is just updated accordingly as rotations happen and nobody outside the picture needs to be updated, because the height is the same as before and nobody above would notice a difference. So, only need O(1) time for updating heights.

Run time:

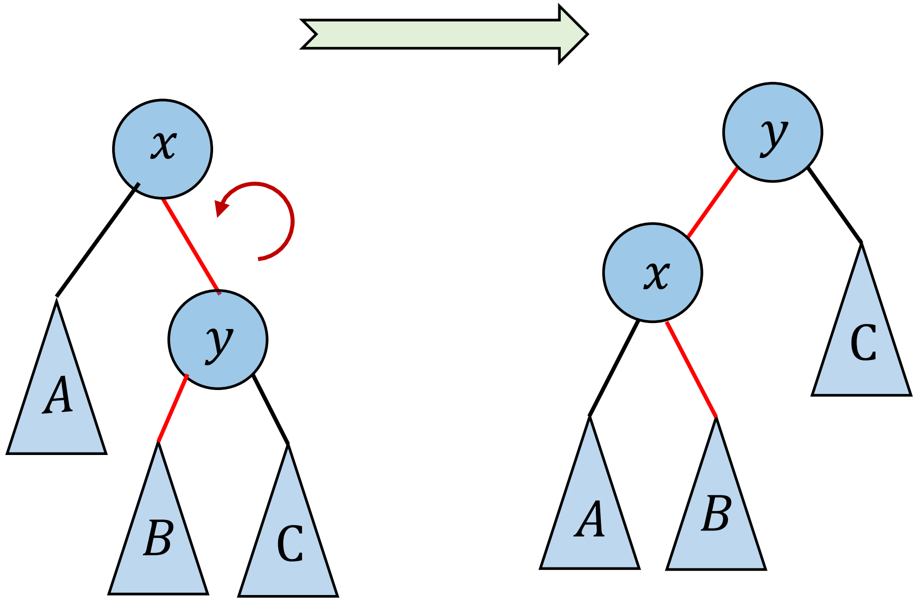
$\Theta(1)$

Rotation to the Left

$AVL\text{-}ROTATE\text{-}TO\text{-}THE\text{-}LEFT(x)$:

Rearrange references.
$y \leftarrow x.right$
$x.p \leftarrow y$
$x.right \leftarrow y.left$
$y.left \leftarrow x$
#Update heights.
$AVL\text{-}UPDATE\text{-}HEIGHT(x)$
$AVL\text{-}UPDATE\text{-}HEIGHT(y)$
Return new parent.
return y

Left rotation (x)



$AVL - UPDATE - HEIGHT(x)$:

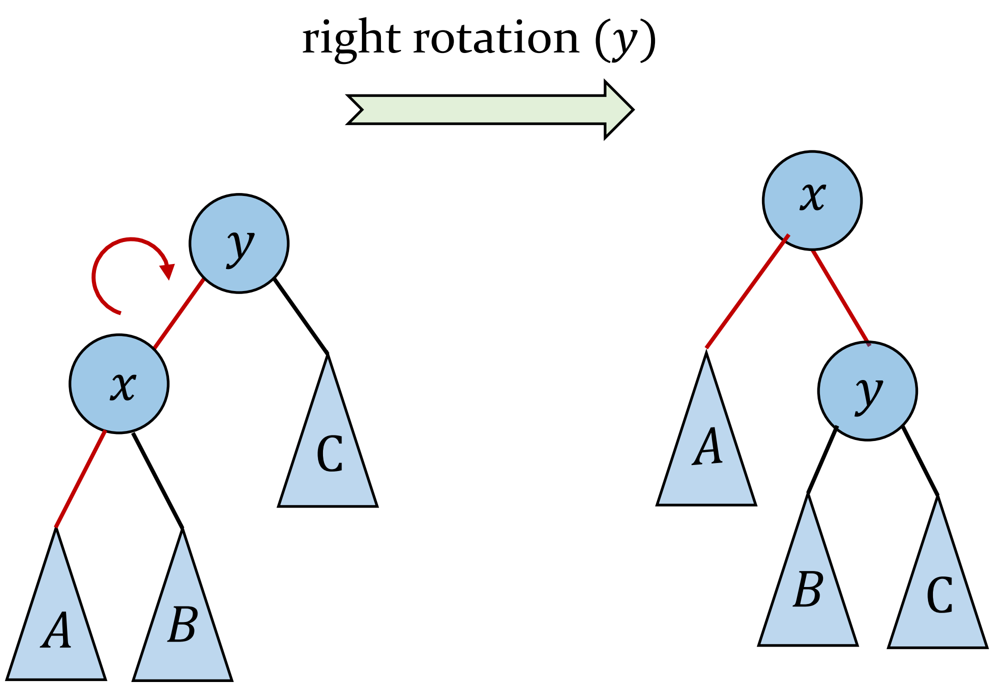
$$x.height \leftarrow 1 + max(x.left.height, x.right.height)$$

Rotation to the right

$AVL\text{-}ROTATE\text{-}TO\text{-}THE\text{-}RIGHT(y)$:

  # Rearrange references.
  $x \leftarrow y.left$
  $x \leftarrow y.p$
  $y.left \leftarrow x.right$
  $x.right \leftarrow$ y
  #Update heights.
  $AVL\text{-}UPDATE\text{-}HEIGHT(y)$
  $AVL\text{-}UPDATE\text{-}HEIGHT(x)$
  # Return new parent.
  return x

三个



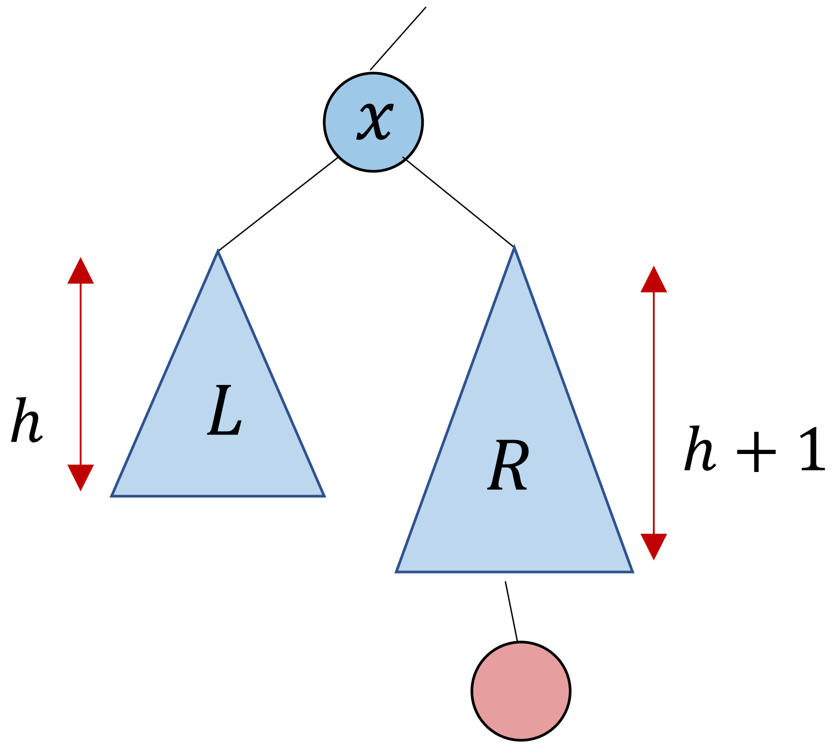
$AVL - UPDATE - HEIGHT(x)$:

$$x.height \leftarrow 1 + max(x.left.height, x.right.height)$$

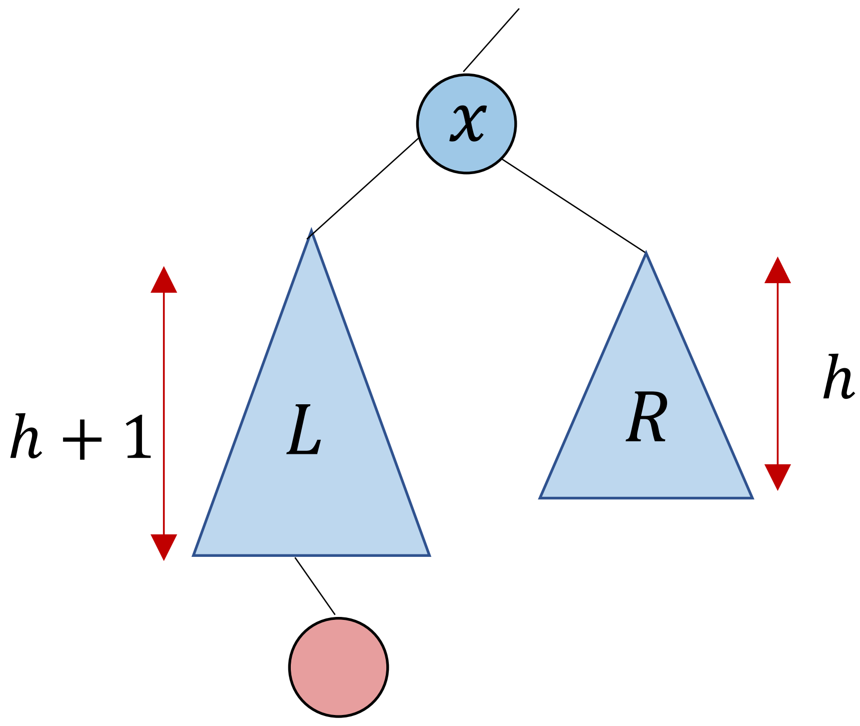
When do we need to rebalance?

Let x be the lowest ancestor of the new node who became imbalanced.

Case 1: the insertion increases the height of a node's right subtree, and that node was already right heavy.

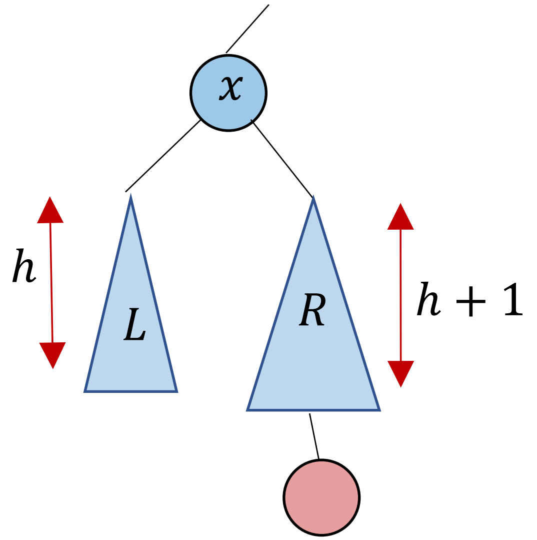


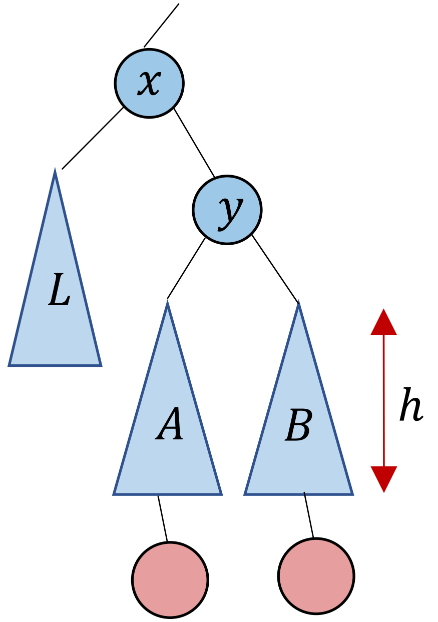
Case 2: the insertion increases the height of a node's left subtree, and that node was already left heavy.



Case 1

In order to rebalance, we need to increase the height of the left subtree and decrease the height of the right subtree, so we perform a left rotation





The height of subtrees A and B are both h.

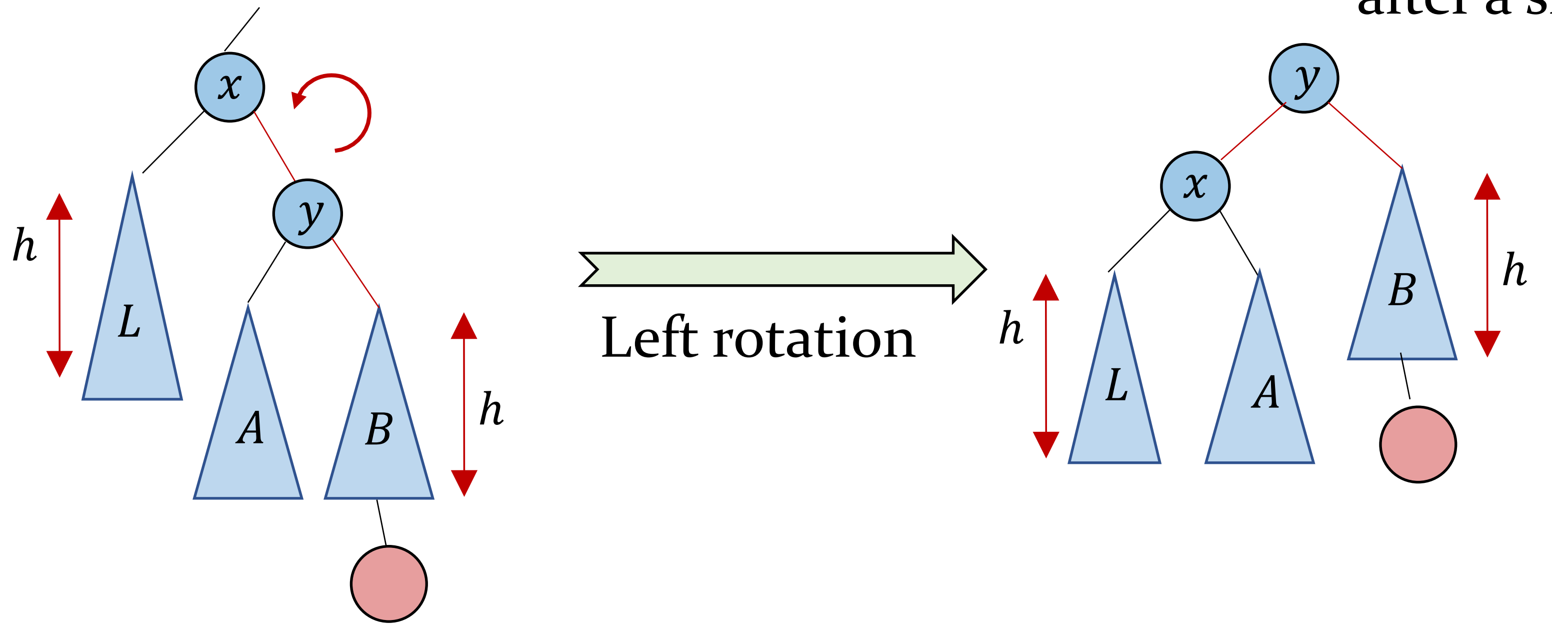
Is it possible that height of one is h and height of the other is $h-1$?

x is the lowest node that become imbalanced.

Case 1.1

Perform a left rotation

The tree becomes balanced after a single rotation.



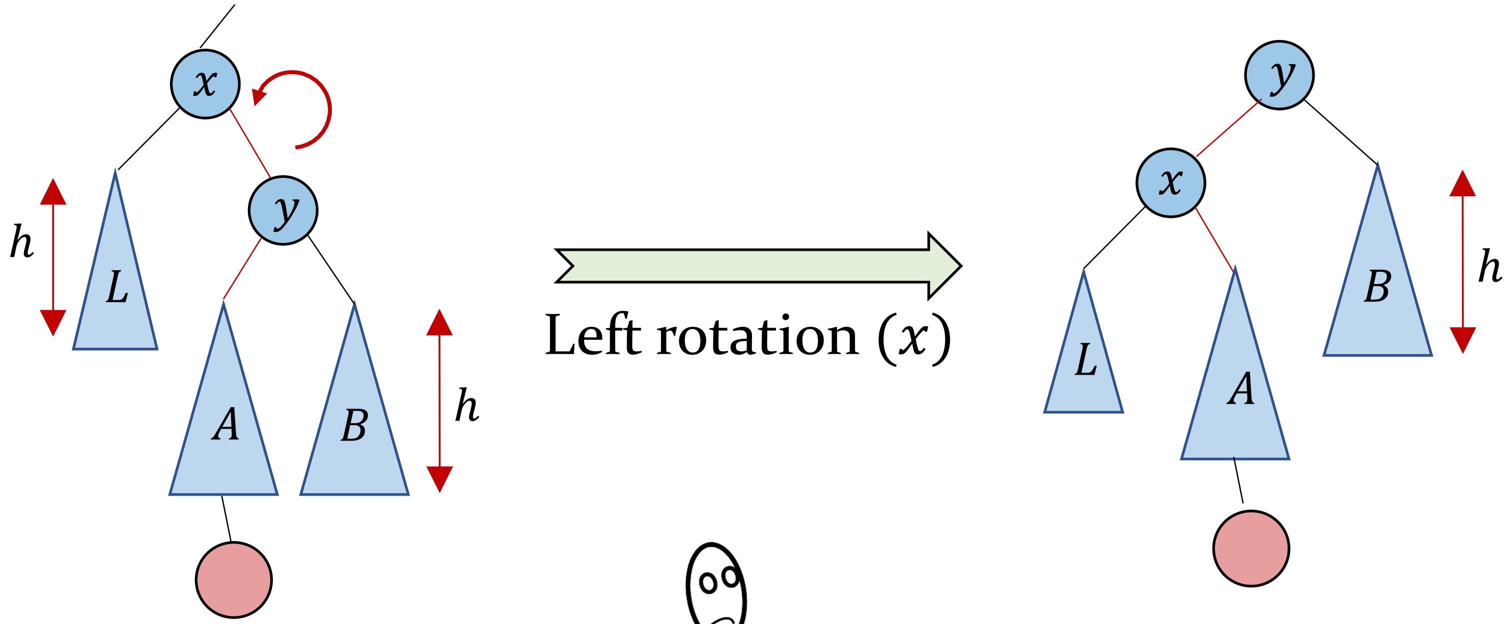
case 1.1

Remark: After the rotation, the height of the whole subtree in the picture does not change, everything happens in this picture stays in this picture, nobody above would notice.

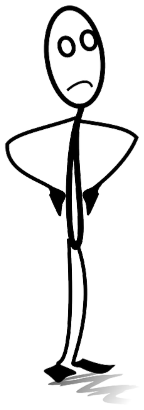
Case 1.2

Perform a left rotation

Still unbalanced!

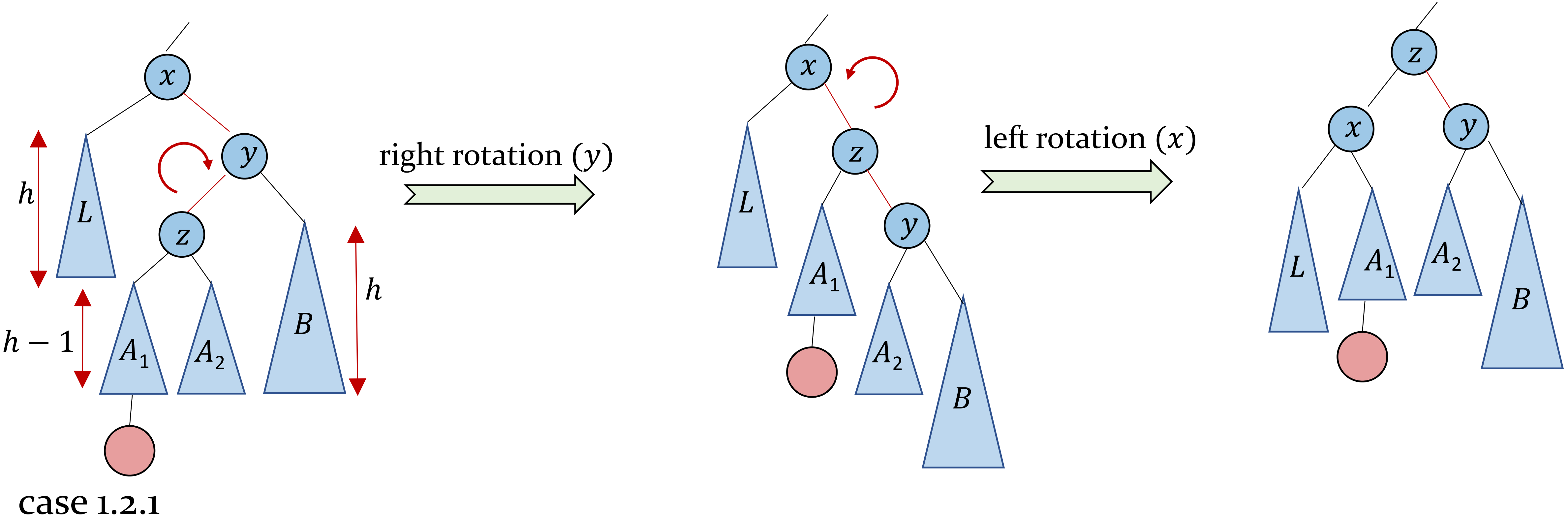


case 1.2



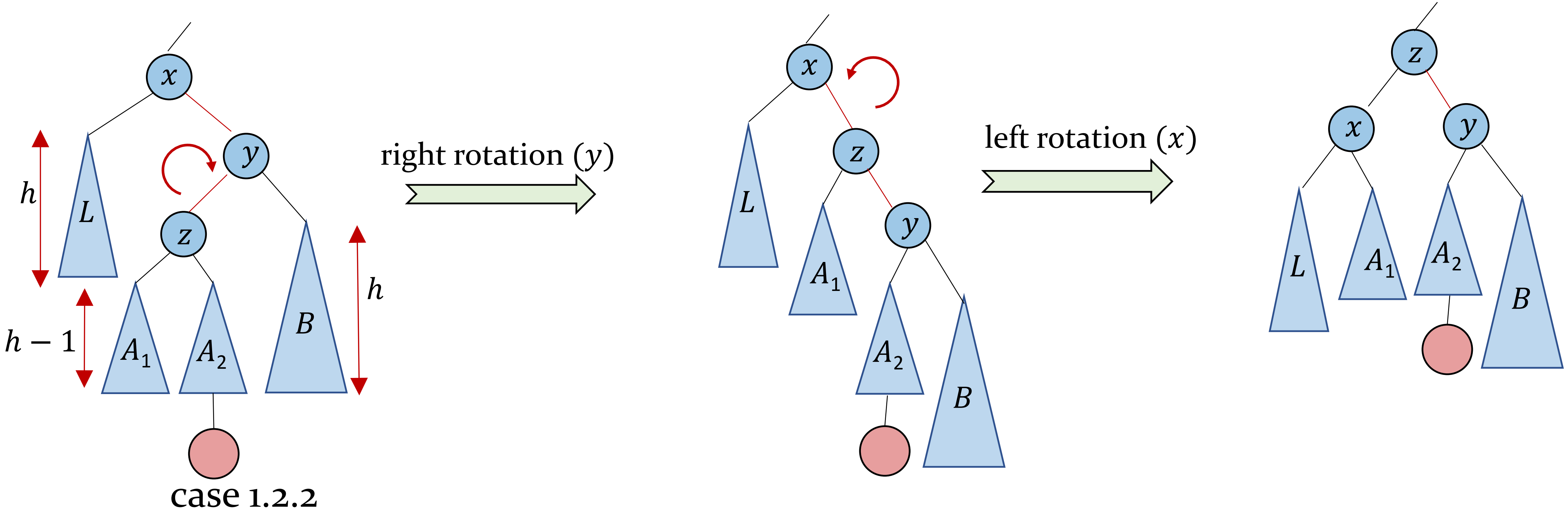
Case 1.2.1: more refined graph

Perform a left rotation



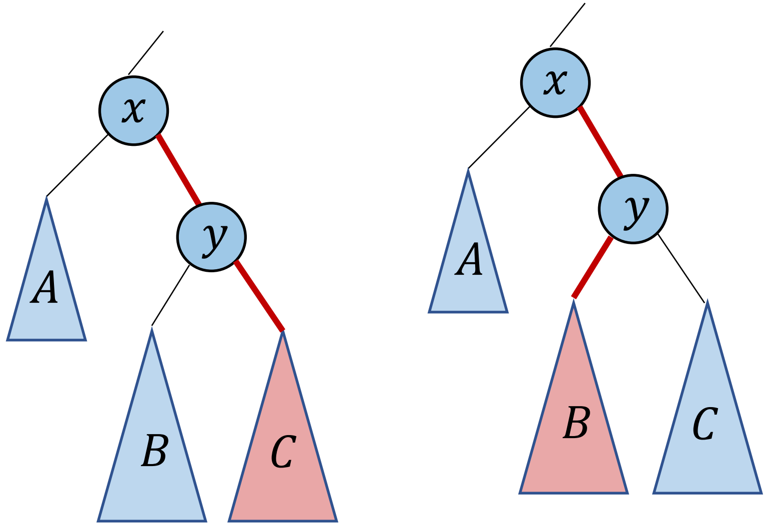
Case 1.2.2: more refined graph

Perform a left rotation



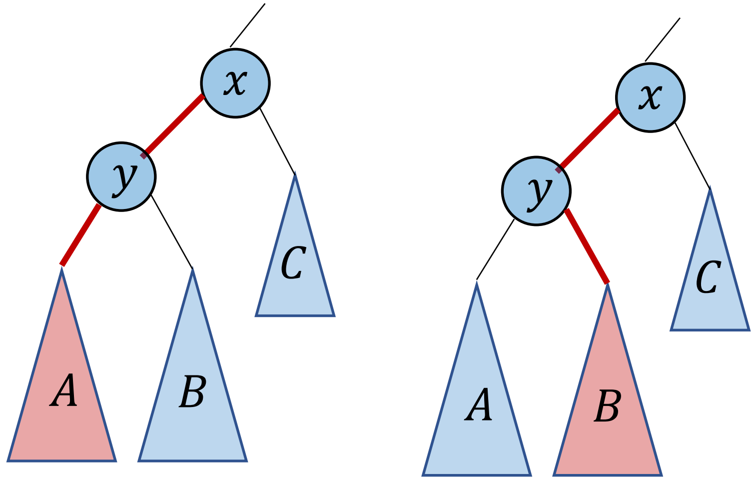
AVL-rebalancing

- Case 1:
  - Case 1.1: single left rotation
  - Case 1.2: double right-left rotation



$AVL\text{-}REBALANCE\text{-}TO\text{-}THE\text{-}LEFT(root)$
  if $root.right.left.height > root.right.right.height$
    $root.right \leftarrow AVL\text{-}ROTATE\text{-}TO\text{-}THE\text{-}RIGHT(root.right)$
  return $AVL\text{-}ROTATE\text{-}TO\text{-}THE\text{-}LEFT(root)$

- Case 2: (symmetric to Case 1)
  - Case 2.1: single right rotation
  - Case 2.2: double left-right rotation



$AVL\text{-}REBALANCE\text{-}TO\text{-}THE\text{-}RIGHT(root)$
  if $root.left.right.height > root.left.left.height$
    $root.left \leftarrow AVL\text{-}ROTATE\text{-}TO\text{-}THE\text{-}LEFT(root.left)$
  return $AVL\text{-}ROTATE\text{-}TO\text{-}THE\text{-}RIGHT(root)$

Some examples:

Insert 3, 2, 1, 4, 5, 6, 7, 16, 15, 14

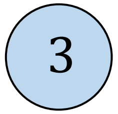


Fig 1

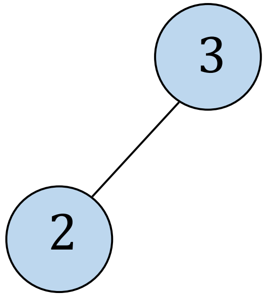


Fig 2

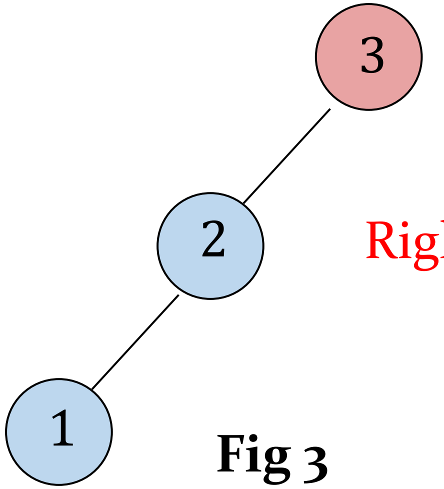


Fig 3

Right rotation

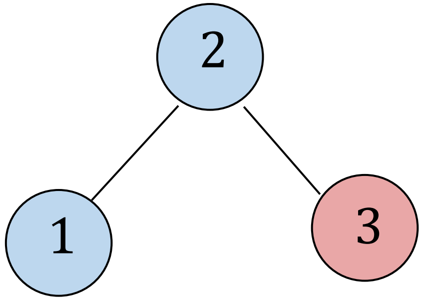


Fig 4

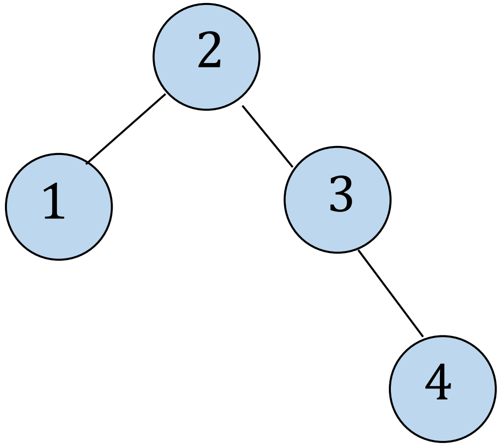


Fig 5

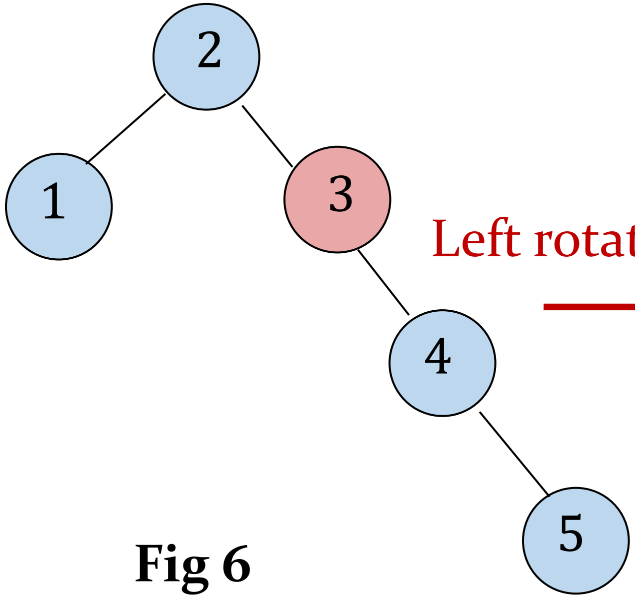


Fig 6

Left rotation

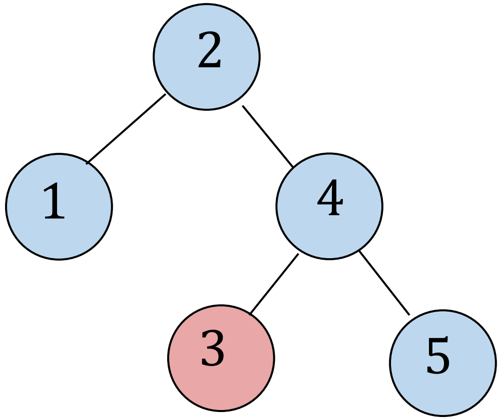


Fig 7

Example - continue

Insert: 3, 2, 1, 4, 5, 6, 7, 16, 15, 14

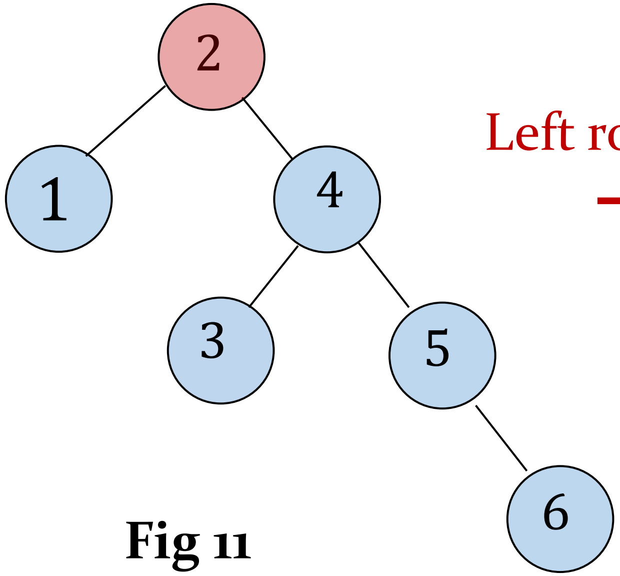


Fig 11

Left rotation(2)

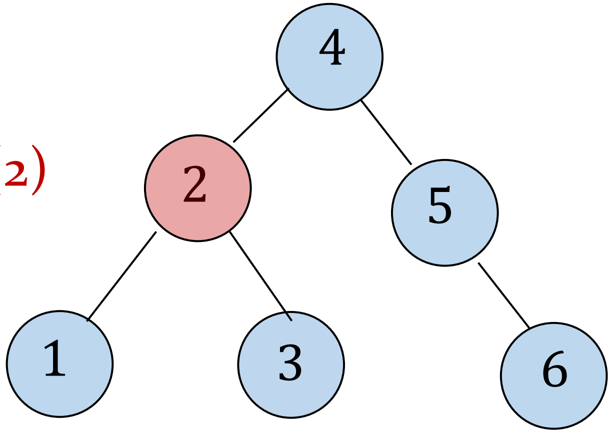


Fig 12

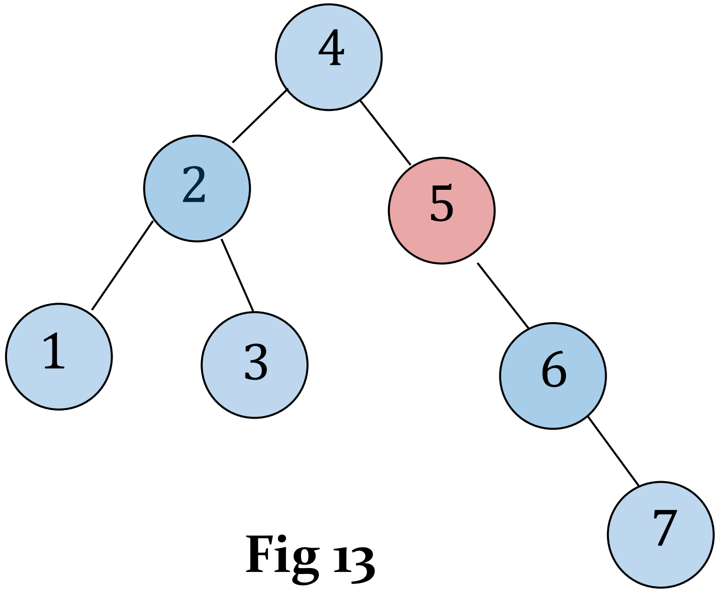


Fig 13

Example - continue

Insert: 3, 2, 1, 4, 5, 6, 7, 16, 15, 14

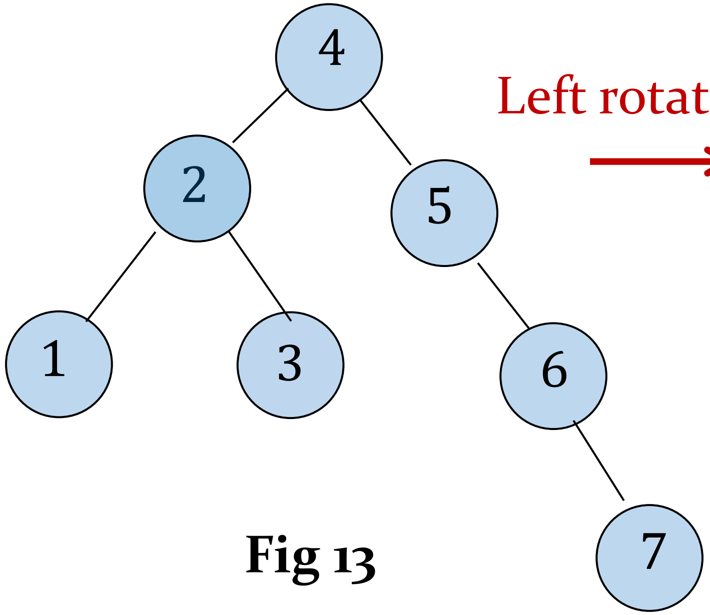


Fig 13

Left rotation(5)

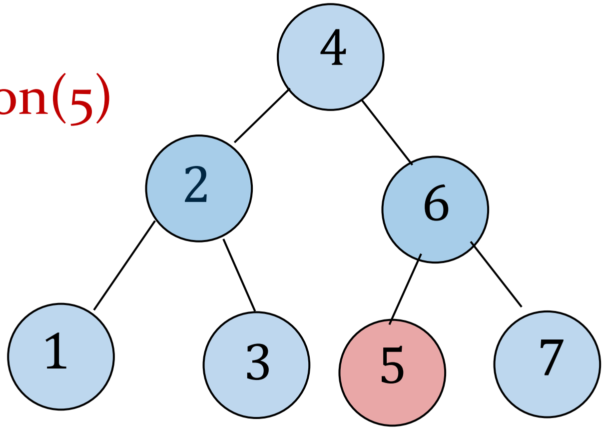


Fig 14

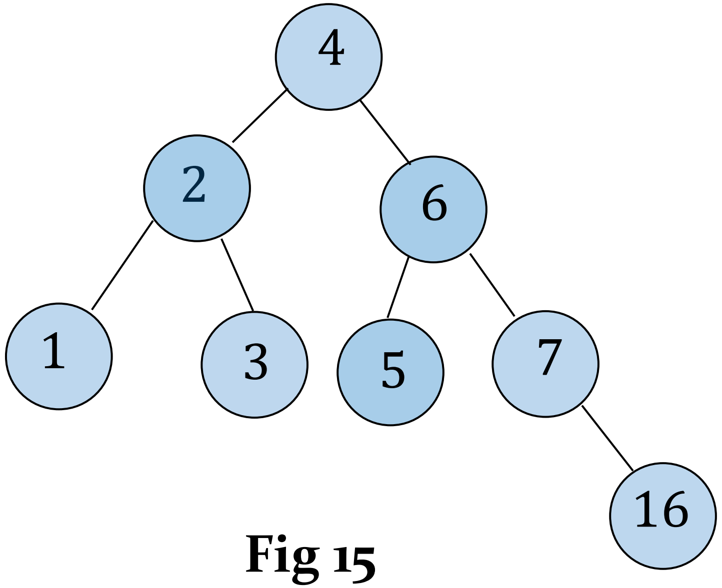


Fig 15

Example - continue

Insert: 3, 2, 1, 4, 5, 6, 7, 16, 15, 14

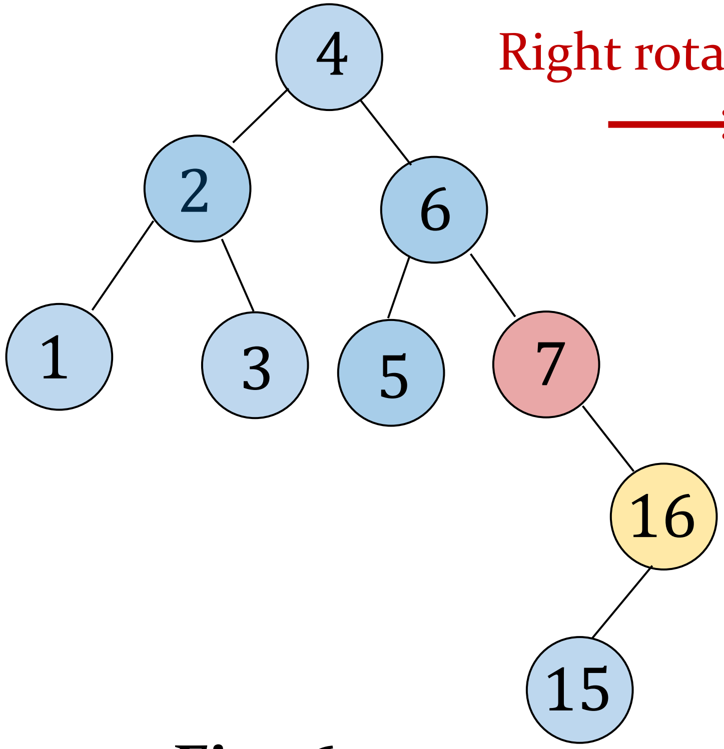


Fig 16

Left rotation(7)

Right rotation(16)

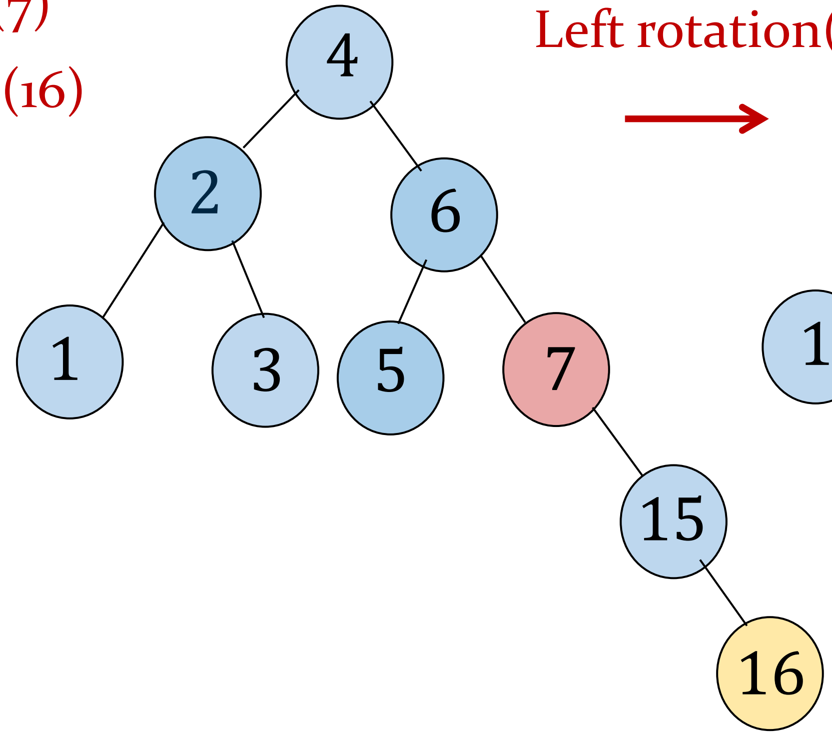


Fig 17

Left rotation(7)

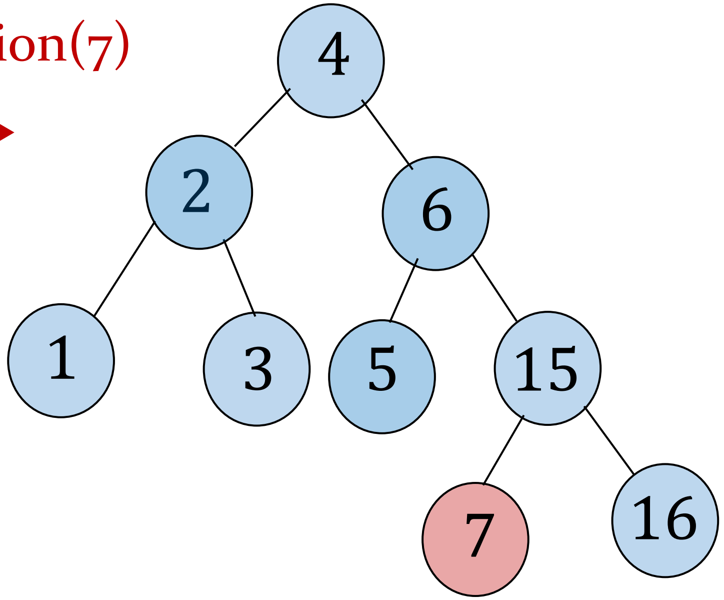
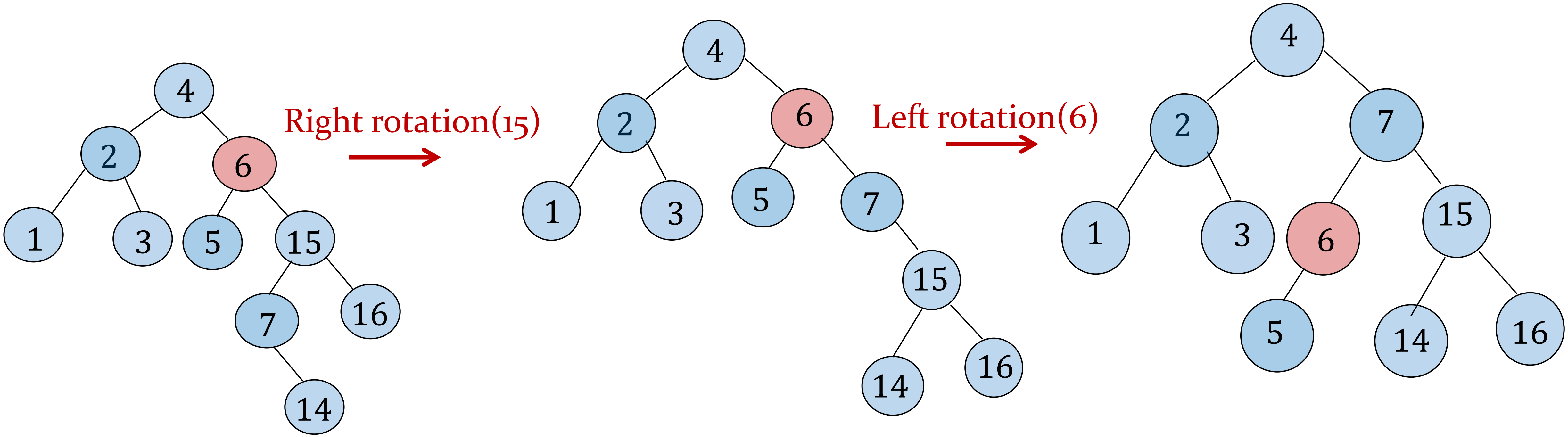


Fig 18

Example - continue

Insert: 3, 2, 1, 4, 5, 6, 7, 16, 15, 14



Example - continue

Insert: 3, 2, 1, 4, 5, 6, 7, 16, 15, 14

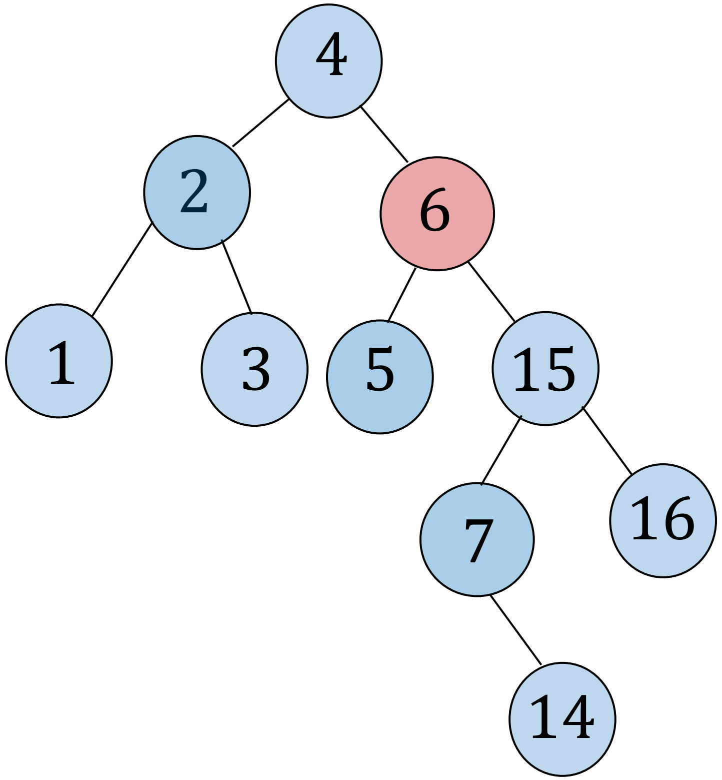


Fig 19

Left rotation(6)

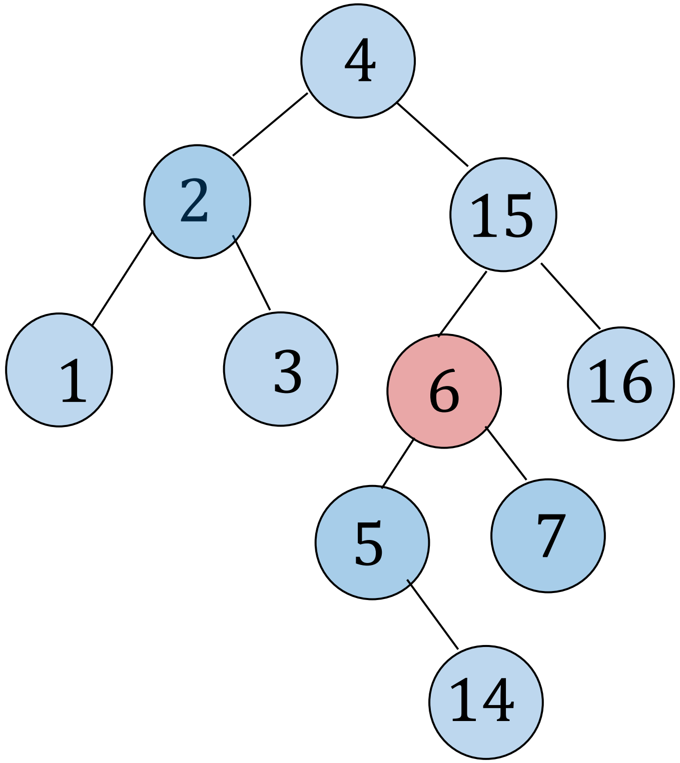
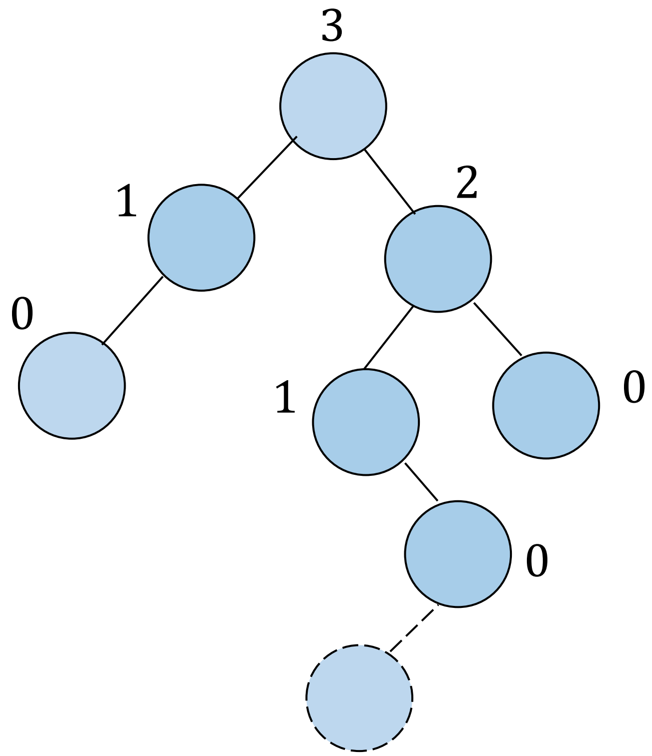


Fig 20

Update height

Remark: When a node is inserted only the height of its parent and ancestors will be updated.



AVL-Rebalance – pseudocode

$AVL - REBALANCE - TO - THE - LEFT(root)$:
    if $root.right.left.height > root.right.right.height$:
        root.right $\leftarrow AVL - ROTATE - TO - THE - RIGHT(root.right)$
return $AVL - ROTATE - TO - THE - LEFT(root)$

$AVL - REBALANCE - TO - THE - RIGHT(root)$:
    if $root.left.right.height > root.left.left.height$:
        $root.left \leftarrow AVL - ROTATE - TO - THE - LEFT(root.left)$
    return $AVL - ROTATE - TO - THE - RIGHT(root)$

AVL-Insert – pseudocode

```
AVL − INSERT(root, x):
    if root is NIL:                                   # found insertion point
      root ← TreeNode(x)                              # will be returned below
                                                      # assumption: TreeNode creates node with .height = 0
    elif x.key < root.item.key:
      root.left ← AVL − INSERT(root.left, x)
      if root.left.height > root.right.height + 1:
        root ← AVL − REBALANCE − TO − THE − RIGHT(root)
      else:                                           # no rebalancing, but height might have changed
        AVL − UPDATE − HEIGHT(root)
    elif x.key > root.item.key:
      root.right ← AVL − INSERT(root.right, x)
      if root.right.height > root.left.height + 1:
        root ← AVL − REBALANCE − TO − THE − LEFT(root)
      else:                                           # no rebalancing, but height might have changed
        AVL − UPDATE − HEIGHT(root)
    else:                                 # x.key = root.item.key Just replace root's item with x nothing else changes.
      root.item ← x
    return root
```

Running time of AVL-Insert

Just Tree-Insert plus some constant time for rotations and height updating.

Overall, worst case $O(h)$ since it's balanced, $O(\log n)$

AVL-Delete($root, x$)

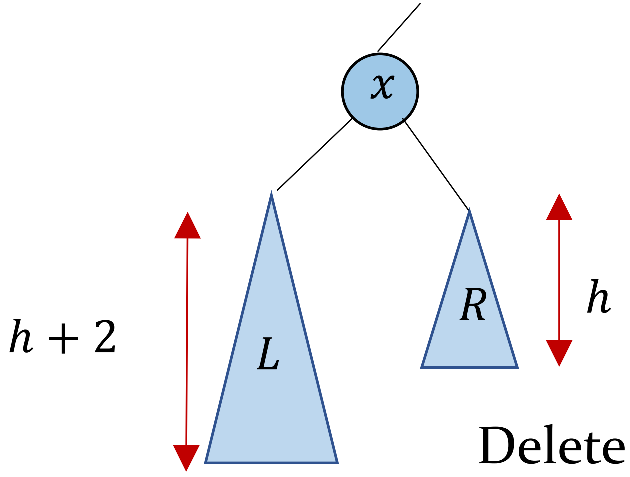
Delete node x from the AVL tree rooted at root

Cases that need rebalancing

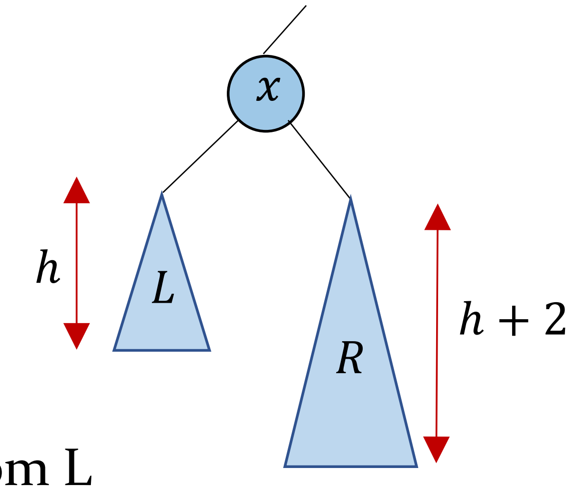
Height of the "whole subtree" rooted at x before deletion is h + 3

Case 1: the deletion reduces the height of a node's right subtree, and that node was left heavy.

Case 2: the insertion increases the height of a node's left subtree, and that node was already left heavy.



Delete from R



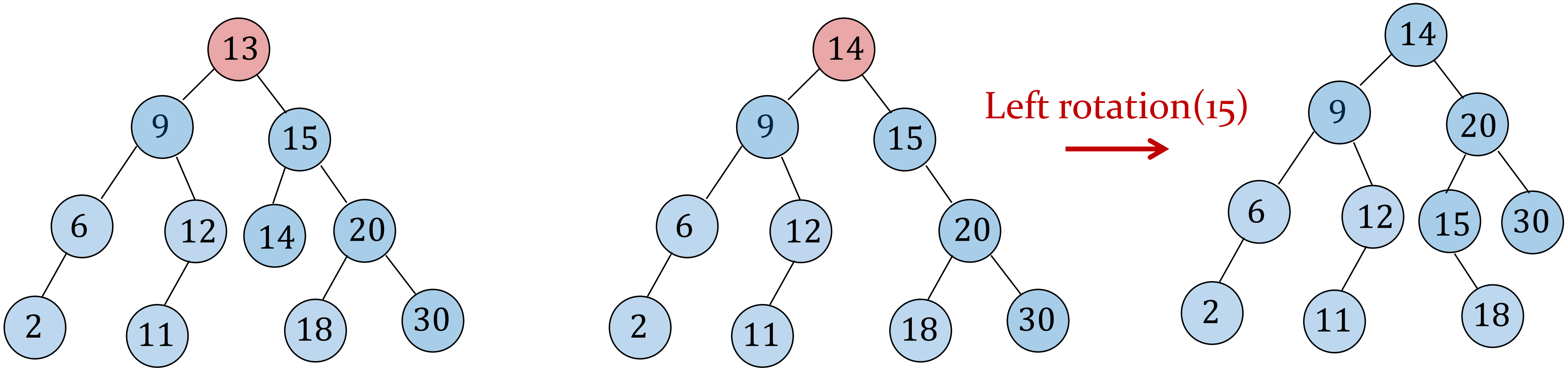
Delete from L

AVL-Delete: General idea

- First do a normal BST Tree-Delete
- The deletion may cause changes of subtree heights, and may cause certain nodes to lose AVL-ness (BF(x) is $0, 1$ or -1)
- Then rebalance by single or double rotations, similar to what we did for AVL-Insert.
- Then update height (BFs) of affected nodes.

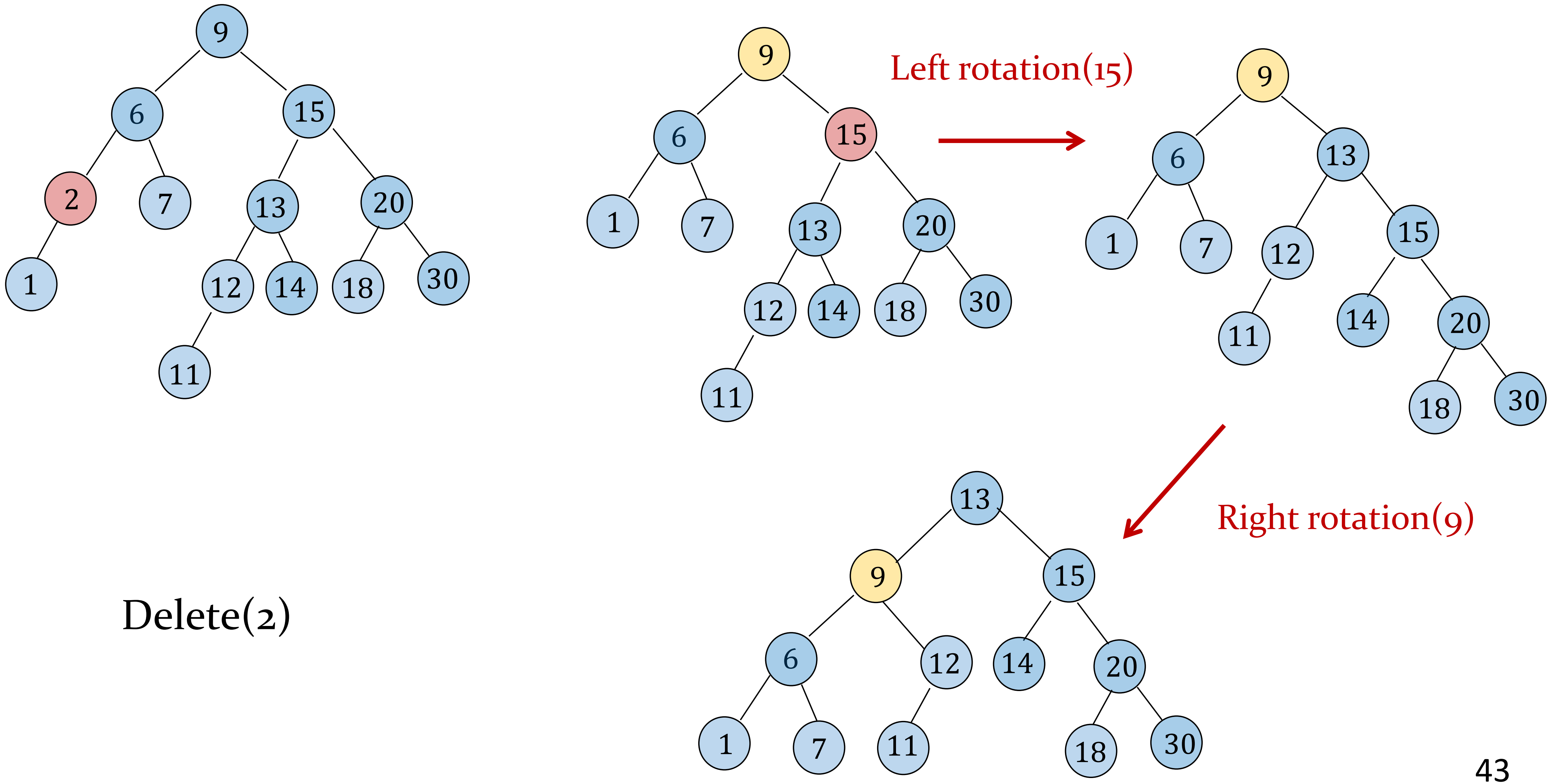
Homework: Write the pseudocode for AVL-Delete. (Bring it to the class next week: It is part of your participation)

AVL-Delete (Example)



Delete(13)

AVL-Delete (Example)



Can we search faster than O(log n)?

Next week we discuss Hash table

Example: Ordered Set

An ADT with the following operation

- Search(S, k) in O(log n)
- Insert(S, x) in O(log n)
- Delete(S, x) in O(log n)

  - Rank(k): return the rank of key k
  - Select(r): return the key with rank r

E.g., S = { 27, 56, 30, 3, 15 }
Rank(15) = 2 because 15 is the second smallest key
Select(4) = 30 because 30 is the 4th smallest key

Augmentation needed

Questions