Data Structures and Algorithms

Lecture 10: AVL Trees II

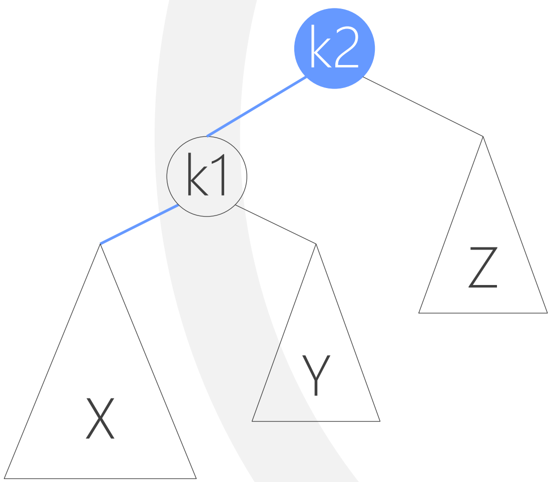
Department of Computer Science & Technology
United International College

Review of AVL Tree Insertion

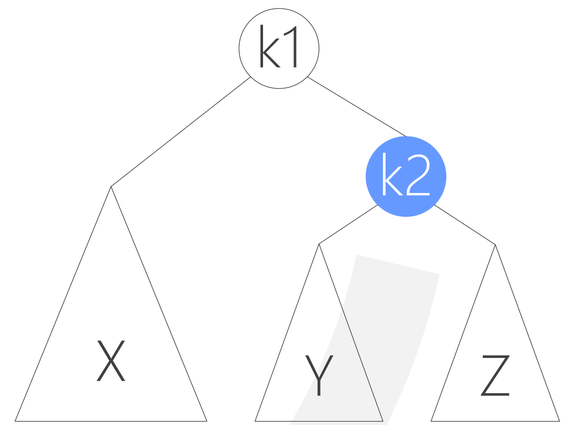
- The complete procedure of insertion
  1. Insert the new node to a proper position
  2. Starting from the new node, search upward for the first unbalanced node
     - Suppose that the height difference of a node's left and right sub-tree is d
       - $d <= 1$ → the node is balanced
       - $d = 0$ → the node is perfectly balanced
       - $d >= 2$ → the node is unbalanced
  3. Perform rotations on the unbalanced node (U)
     - Case 1: U is left heavy, its left child is left heavy
     - Case 2: U is left heavy, its left child is right heavy
     - Case 3: U is right heavy, its right child is left heavy
     - Case 4: U is right heavy, its right child is right heavy

Single Right Rotation to Fix Case 1 (left-left)

K2 is unbalanced

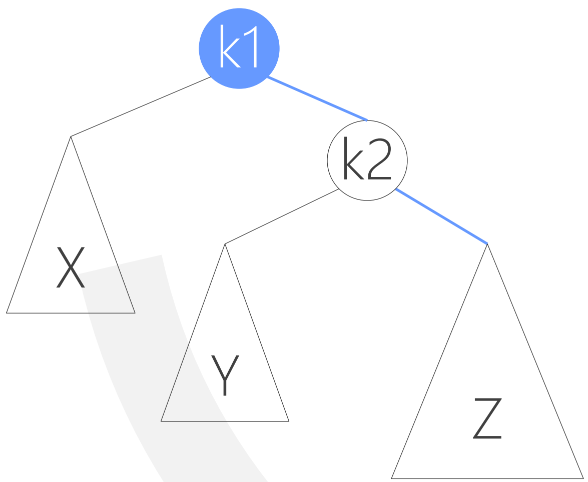


K1 is perfectly balanced

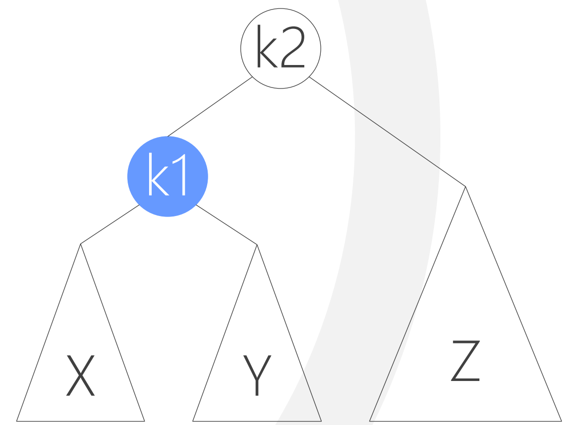


Single Left Rotation to Fix Case 4 (right-right)

K1 is unbalanced

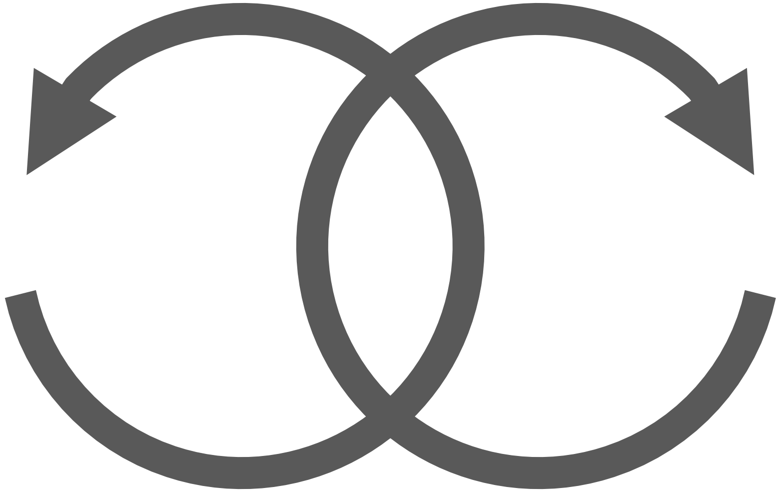


K2 is perfectly balanced



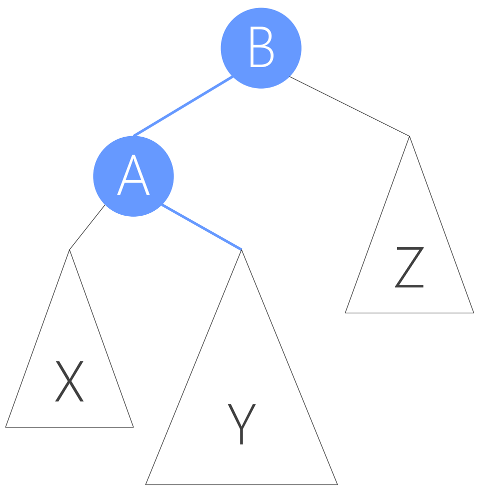
Double Rotation to Fix Case 2&3

- One single rotation to move the deepest sub-tree to the outer side
- Another single rotation to restore the balance

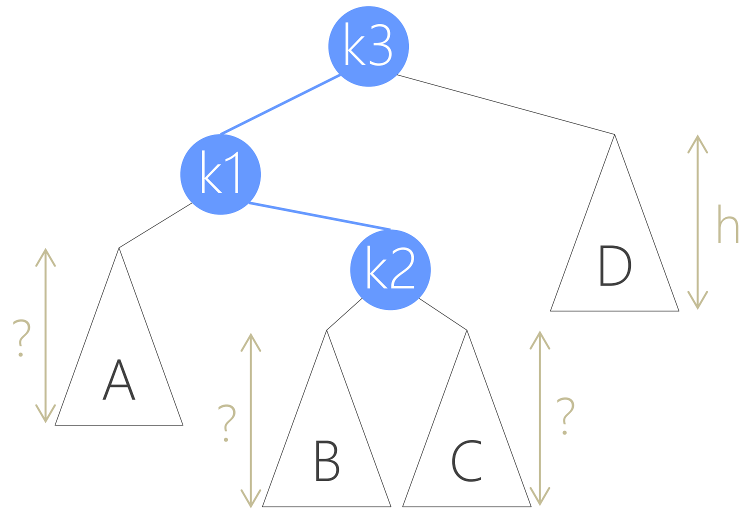


Case 2 (left-right)

B is unbalanced



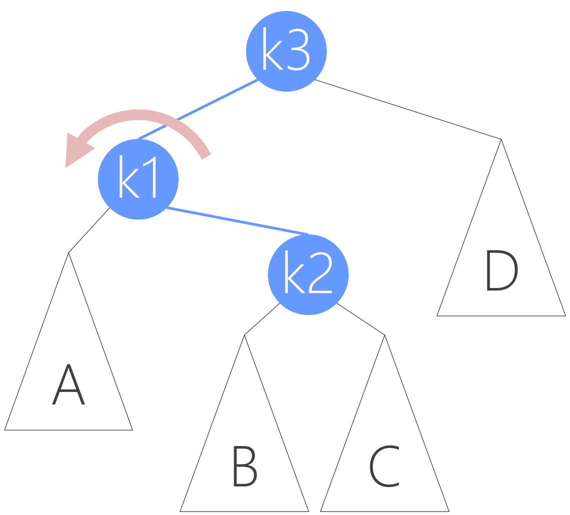
Label B's child



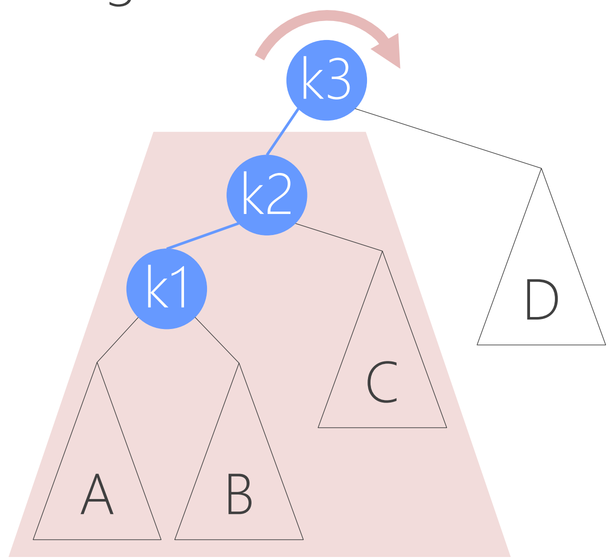
- If the height of sub-tree D is h
  - What is the possible height of A, B and C?

Double Rotation to Fix Case 2

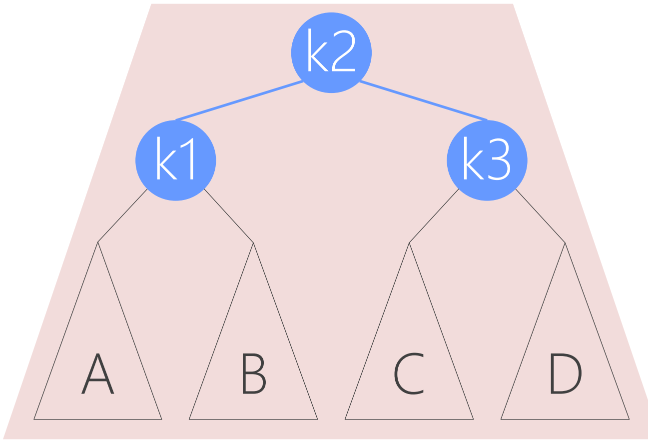
k3 is unbalanced



Single left rotation



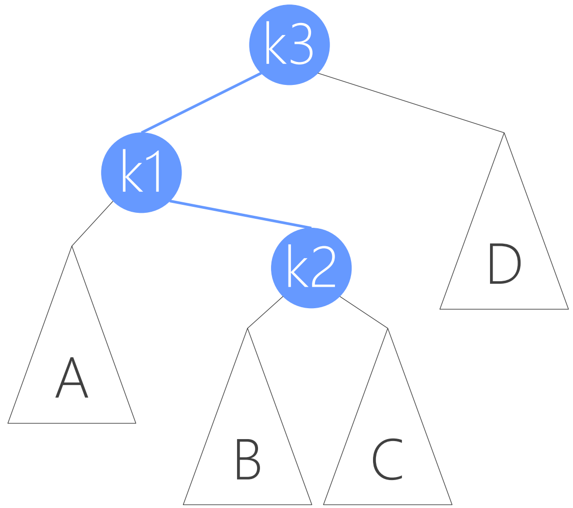
Single right rotation



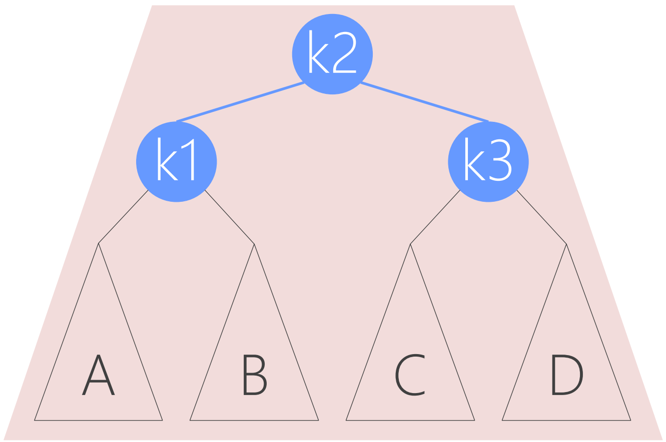
K2 is perfectly balanced

Direct Re-Arrangement

k3 is unbalanced



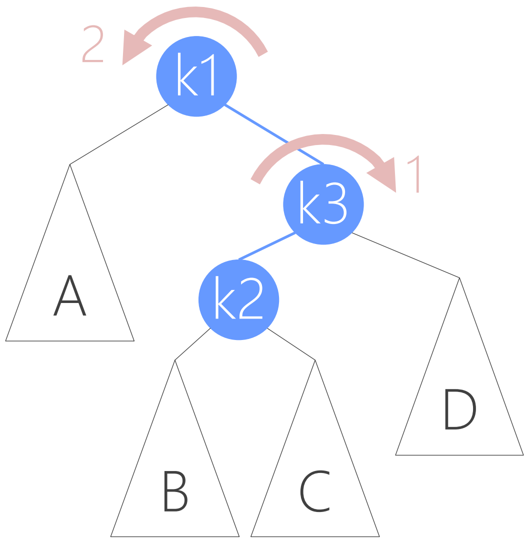
K2 is perfectly balanced



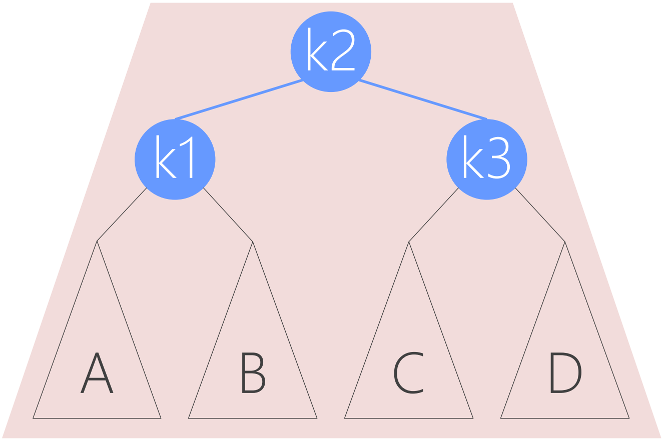
- Pre-condition: k3-k1-k2 forms a zig-zag shape
- Post-condition: k2 is the parent of k1 and k3

Case 3 (right-left)

k3 is unbalanced



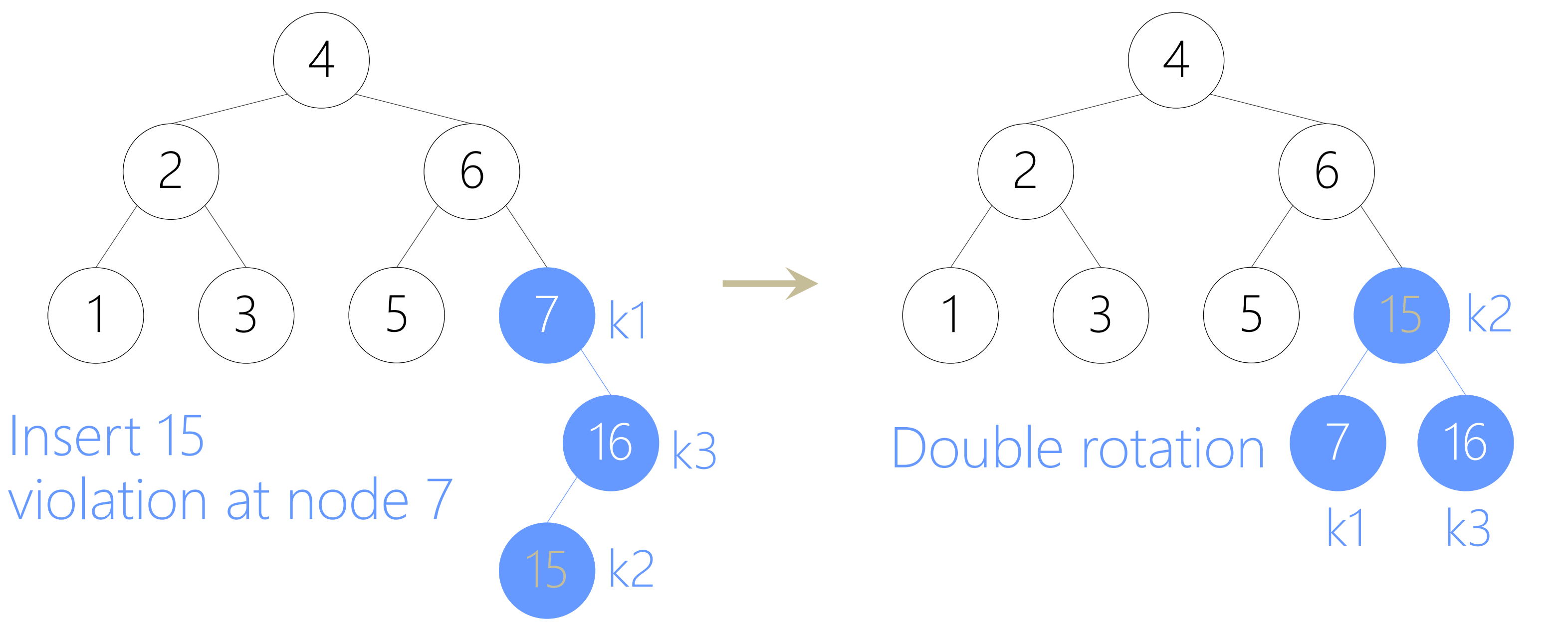
K2 is perfectly balanced

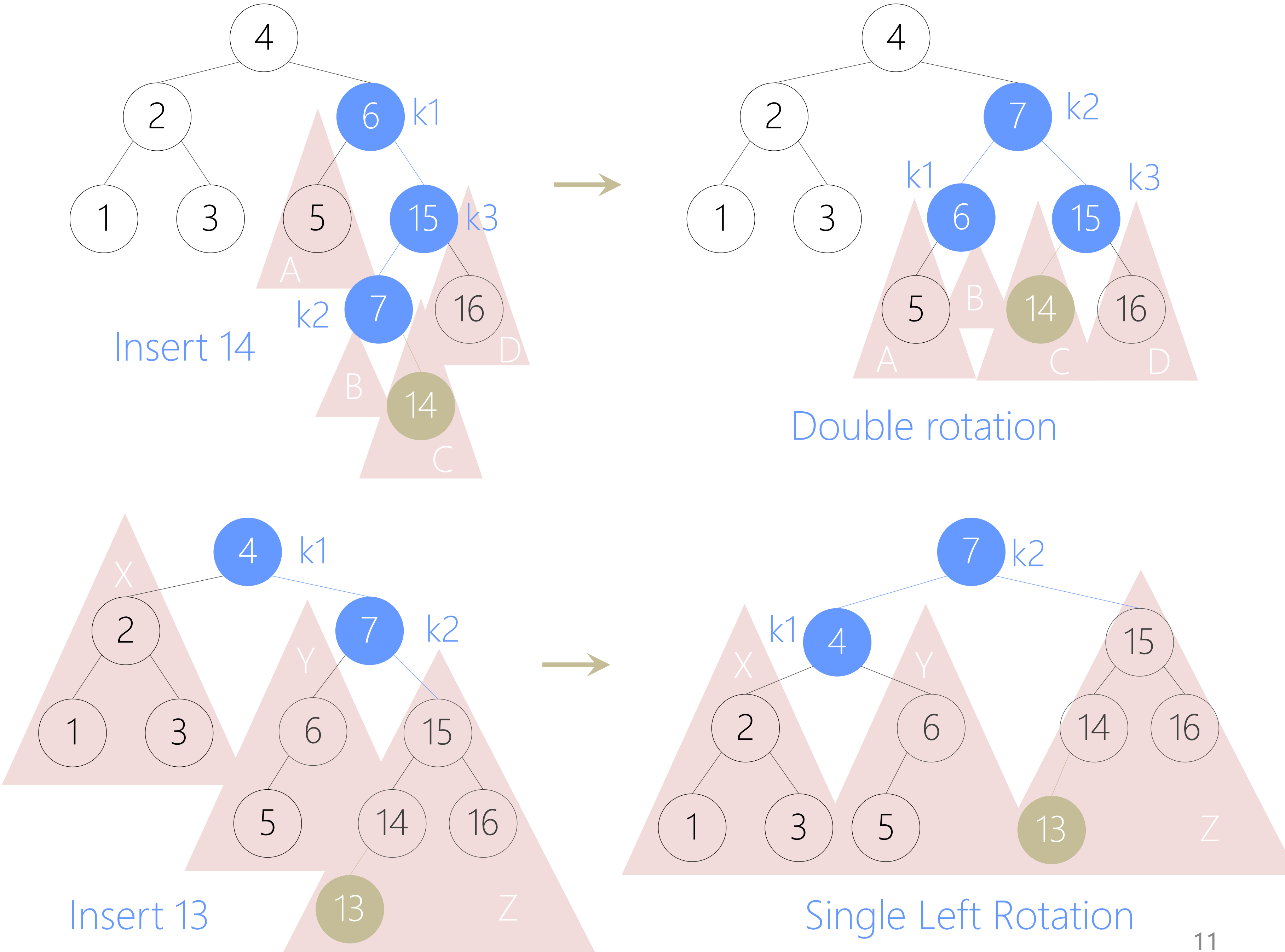


- Case 3 is symmetric to Case 2
- Pre-condition: k1-k3-k2 forms a zig-zag shape
- Post-condition: k2 is the parent of k1 and k3

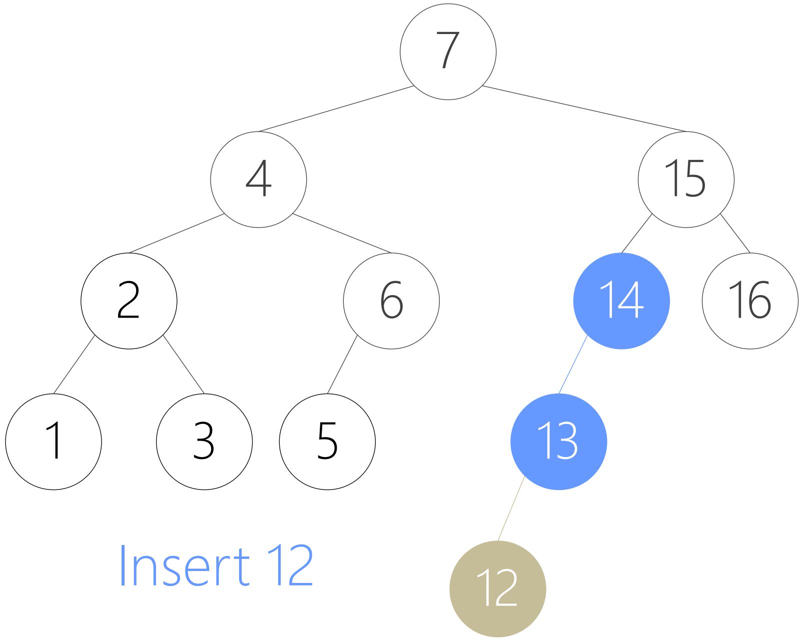
Example

- Continue our example
  - We've inserted 3, 2, 1, 4, 5, 6, 7, 16
  - We'll insert 15, 14, 13, 12, 11, 10, 8, 9

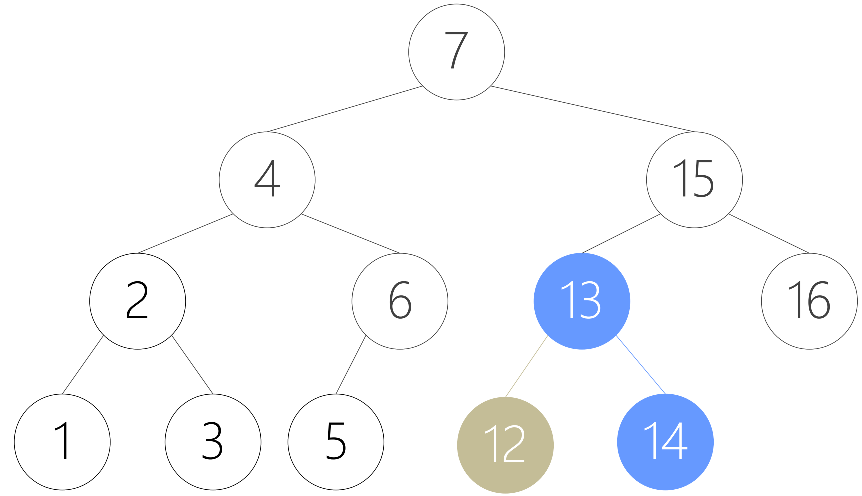




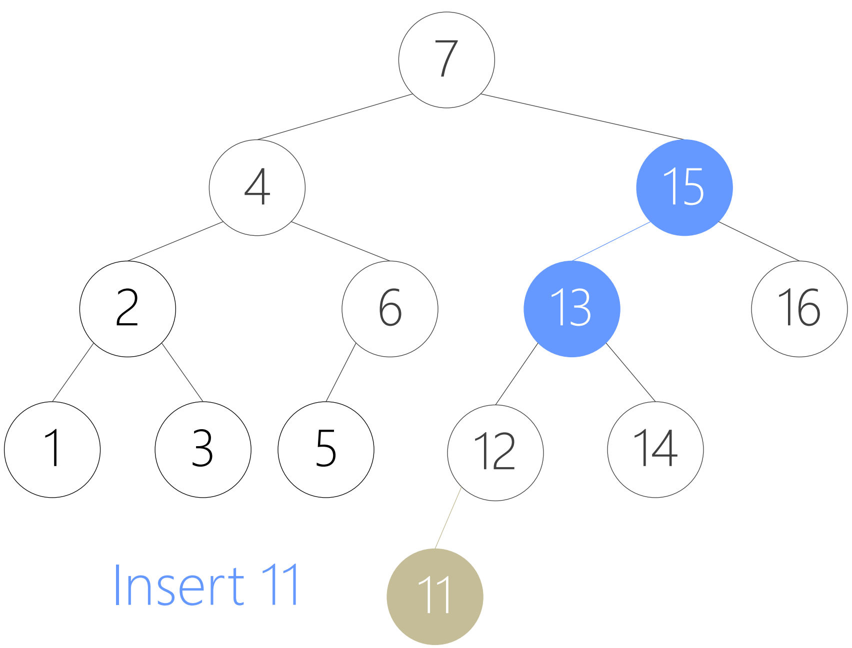
4
2
6
k1
1
3
5
15
k3
A
7
k2
16
D
Insert 14
B
14
C
4
2
7
k2
k1
k3
1
3
6
15
5
B
14
16
A
C
D
Double rotation
X
4
k1
2
7
k2
Y
1
3
6
15
5
14
16
13
Z
Insert 13
7
k2
k1
4
X
Y
15
2
6
14
16
1
3
5
13
Z
Single Left Rotation



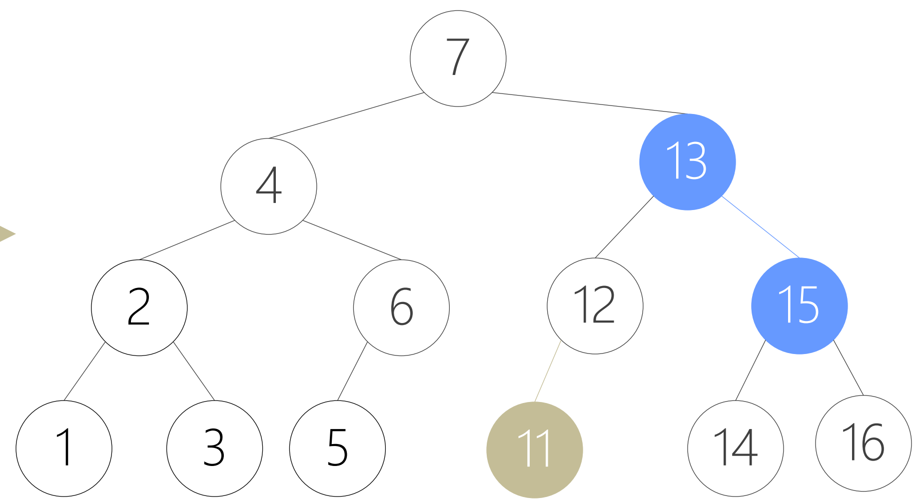
Insert 12



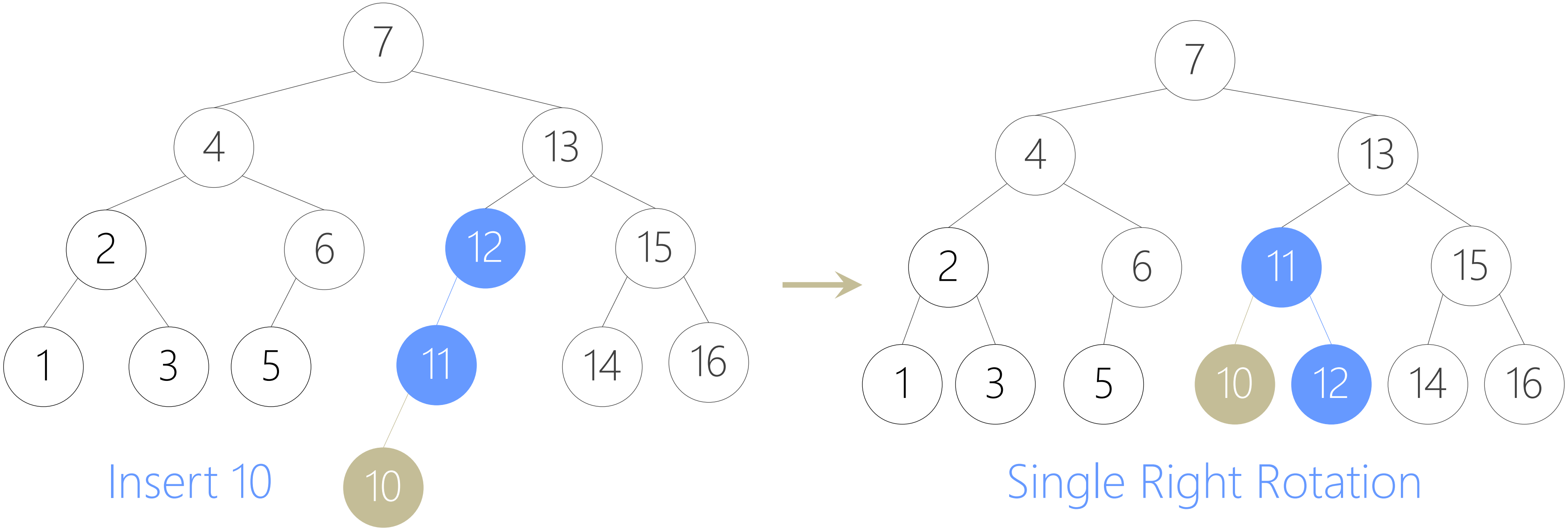
Single Right Rotation



Insert 11

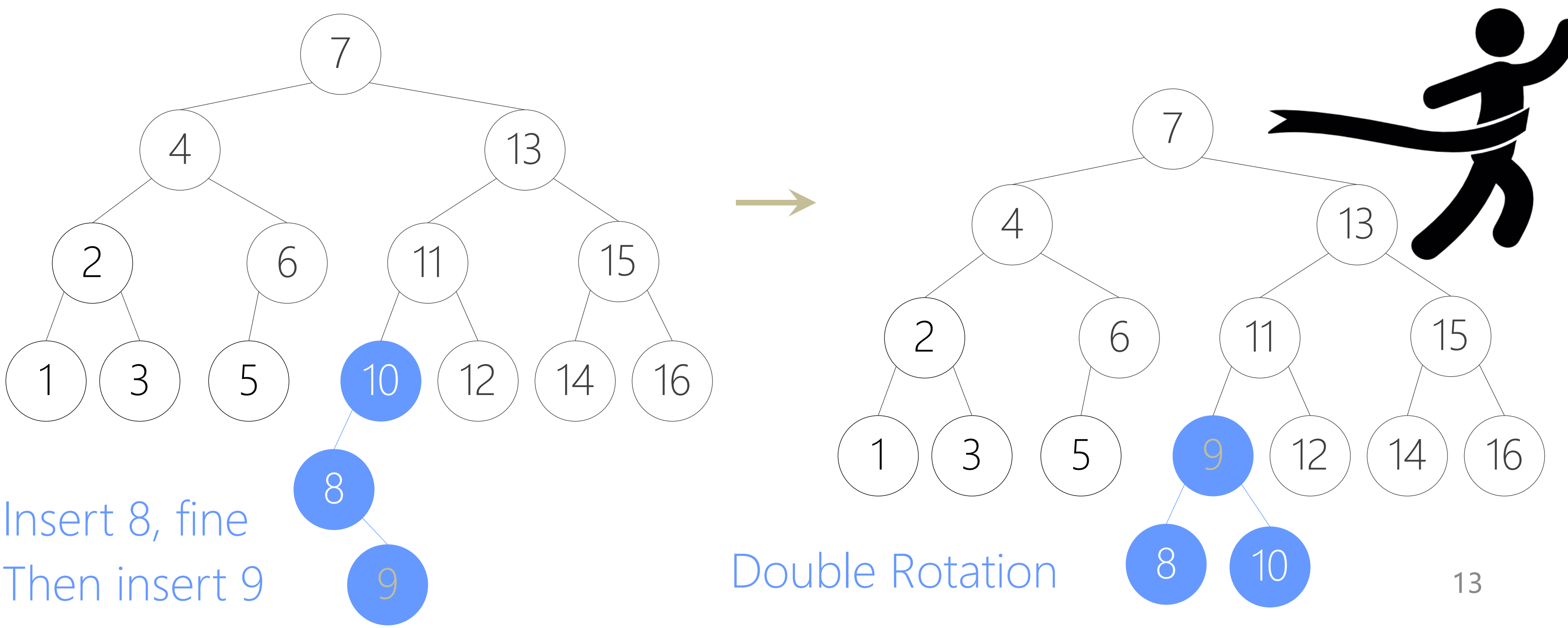


Single Right Rotation



Insert 10

Single Right Rotation



Insert 8, fine

Then insert 9

Double Rotation

Insertion Analysis

- Insert the new key as a new leaf: O(log(n))
- Then trace the path from the new leaf towards the root, for each node x encountered: O(log(n))
  - Check height difference: O(1)
  - If satisfies AVL property, proceed to next node: O(1)
  - If not, perform a rotation: O(1)
- The insertion stops when
  - A rotation is performed
  - Or, we've checked all nodes in the path
- Time complexity for insertion: O(log(n))

Check Height Difference

- Cost for checking height difference: O(1)
  - Keep "height" information on every tree node
    - The height of the sub-tree rooted at the node
    - height >= 0
  - Update "height" when a the sub-tree is altered
  - Compare the height of its sub-trees when you check the balance of a node

Node
- key: int - height: int - left: Node - right: Node

Pseudo Code for Insertion

Returns the (updated) root of the subtree after insertion

root's height is: (height of its deeper sub-tree) + 1

INSERT-NODE(root, x)

1. root = BST-INSERT-Node(root, x)
2. root.UPDATE-HEIGHT()
3. root = REBALANCE(root)
4. return root

Pseudo Code for Insertion

```
BST-INSERT-NODE(root, x)
1.  IF root = Null
2.      return CREATE-Node(x)
3.  IF x < root.key
4.      root.left = INSERT-NODE(root.left, x)
5.  ELSE IF x > root.key
6.      root.right = INSERT-NODE(root.right, x)
7.  return root
```

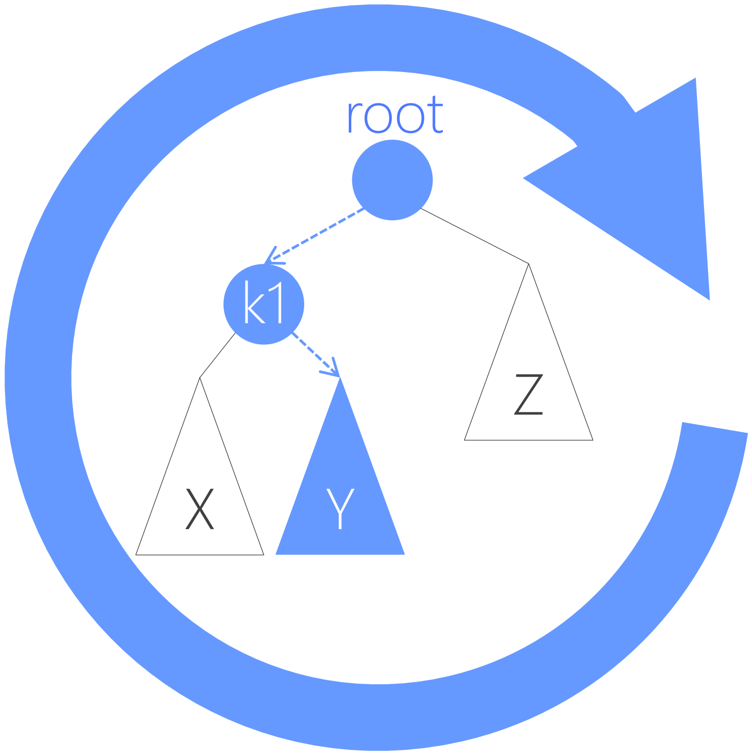
Returns the (updated) root of the subtree after insertion

Rebalance

Returns the (updated) root of the subtree after rebalance

```
REBALANCE(root)
1.  IF BALANCED(root)
2.      return root
3.  IF CASE1(root)  // left left
4.      root = RIGHT-ROTATE(root)
5.  IF CASE4(root) // right right
6.      root = LEFT-ROTATE(root)
7.  IF CASE2(root) // left right
8.      root.left = LEFT-ROTATE(root.left)
9.      root = RIGHT-ROTATE(root)
10. IF CASE3(root) // right left
11.     root.right = RIGHT-ROTATE(root.right)
12.     root = LEFT-ROTATE(root)
13. return root
```

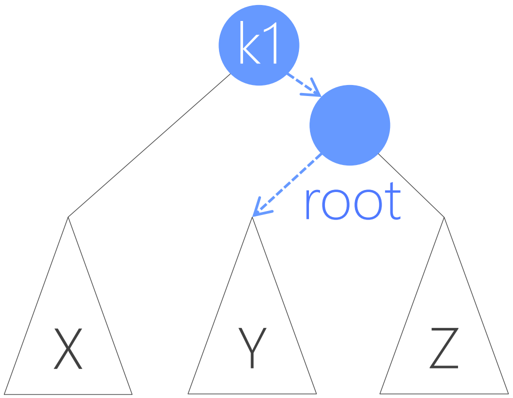
Rotation




```
RIGHT-ROTATE(root)
k1=root.left, Y=k1.right
k1.right=root
root.left=Y
UPDATE-HEIGHT(root)
UPDATE-HEIGHT(k1)
return k1
```

Returns the (updated) root of the subtree after rotation

Rotation



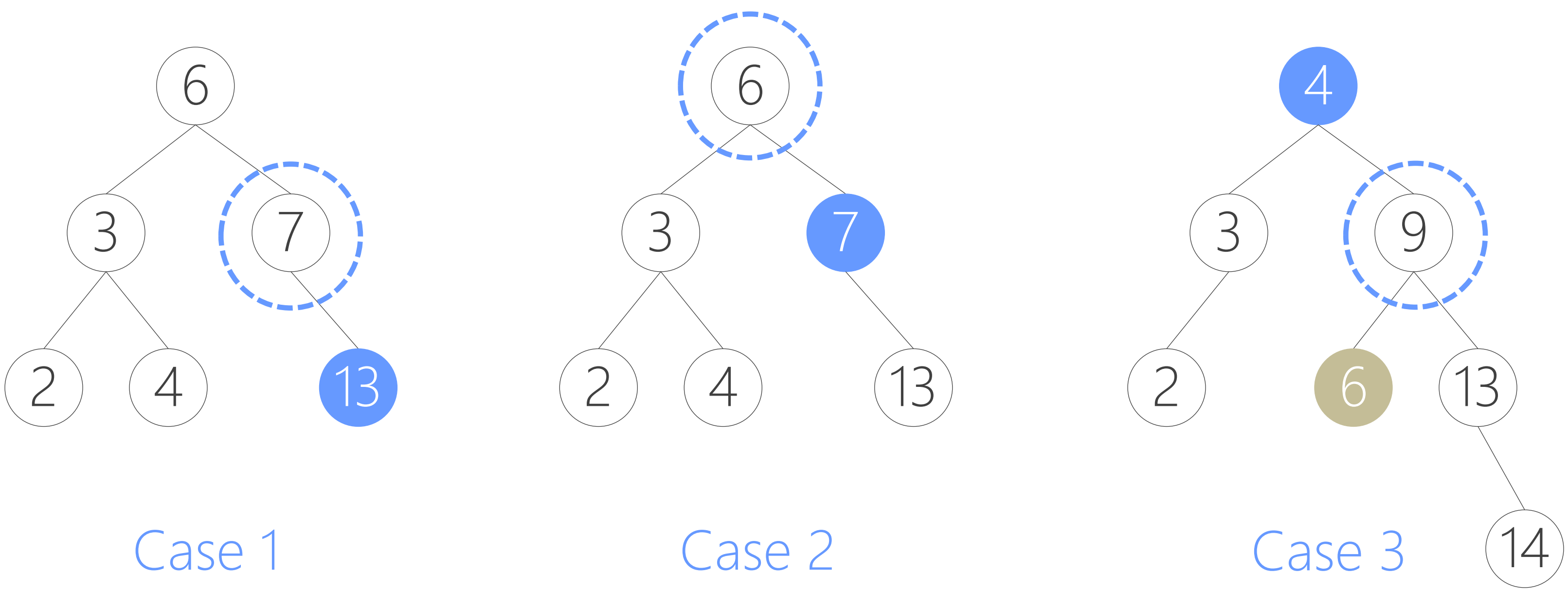
RIGHT-ROTATE(root)
1. k1=root.left, Y=k1.right
2. k1.right=root
3. root.left=Y
4. UPDATE-HEIGHT(root)
5. UPDATE-HEIGHT(k1)
6. return k1

Notes on Rotations

- Two pointers are modified in a rotation
  - The method returns the (updated) root of the subtree after rotation
- Sub-tree heights should be updated after a rotation
  - Always update the deeper node first!
- Left rotation and right rotation are symmetric

Deletion

1. Delete a node x as in an ordinary binary search tree
   – Note that the last (deepest) node in a tree deleted is a leaf or a node with one child



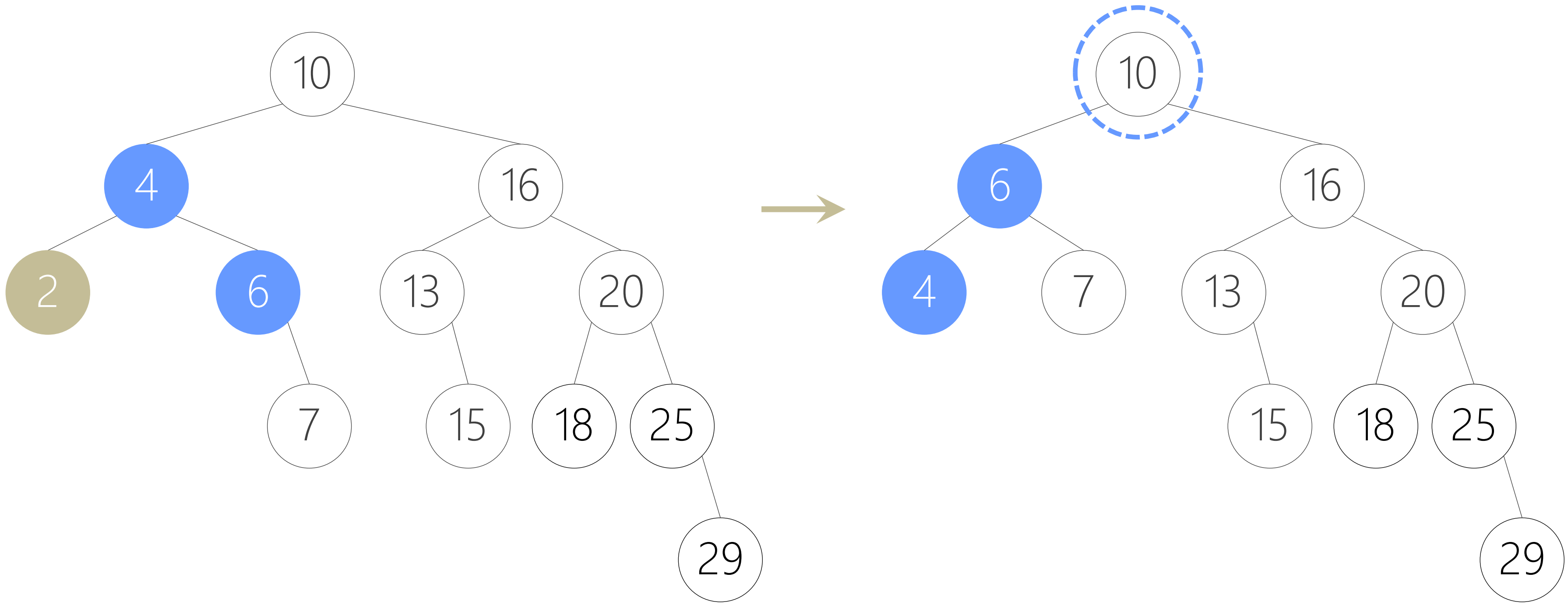
Deletion

1. Delete a node x as in an ordinary binary search tree
2. Then trace the path from the parent towards the root
3. For each node x encountered, check if it is balanced
   – Unbalanced: Perform appropriate rotations

   Continue to trace the path

   UNTIL WE REACH THE ROOT

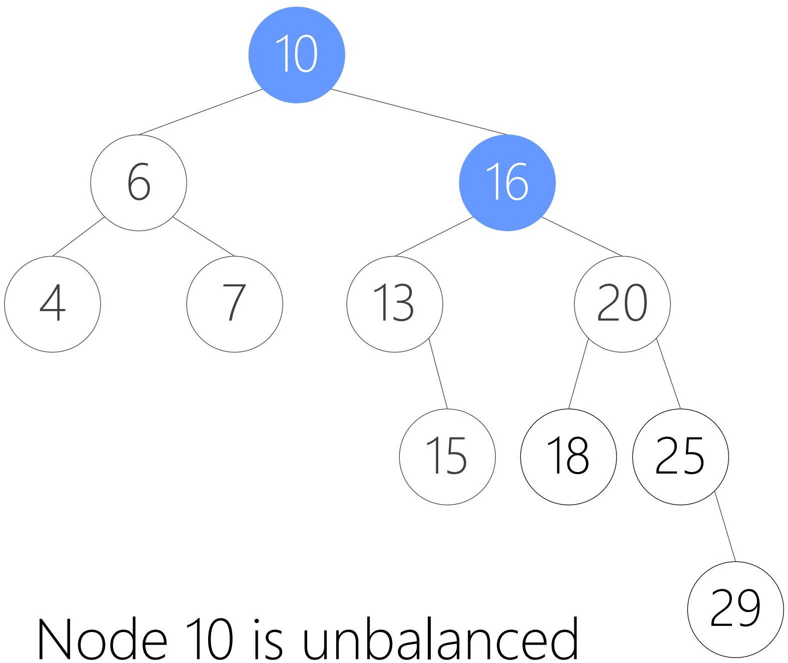
Delete Example



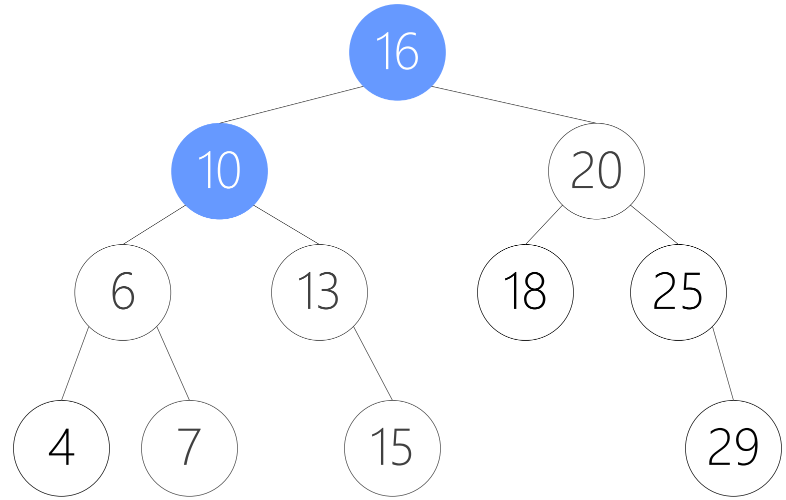
Delete 2, Node 4 is unbalanced
CASE 4: right-right

Single Left Rotation

Delete Example



Node 10 is unbalanced
CASE 4: right-right



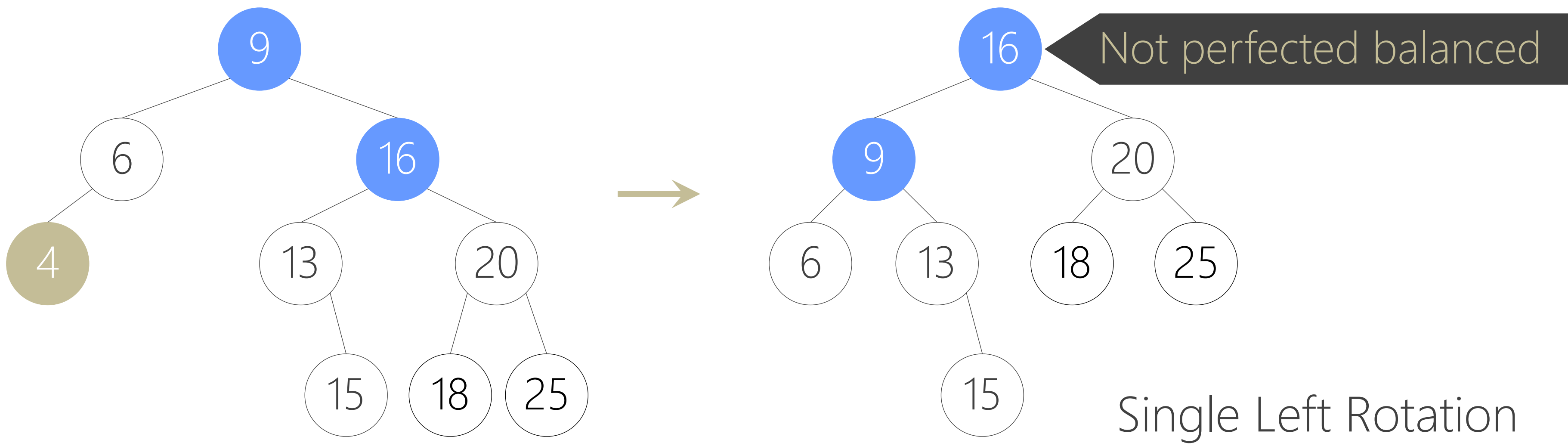
Single Left Rotation

For deletion, after rotation, we need to continue tracing upward to see if AVL-tree property is violated at other nodes.

Rotation in Deletion

- The rotation strategies (single or double) we learned for insertion can be reused
- Except for one new case: the heavy child is perfectly balanced
  - What kind of delete will cause this case?
  - A single rotation solves the problem

New Case Example



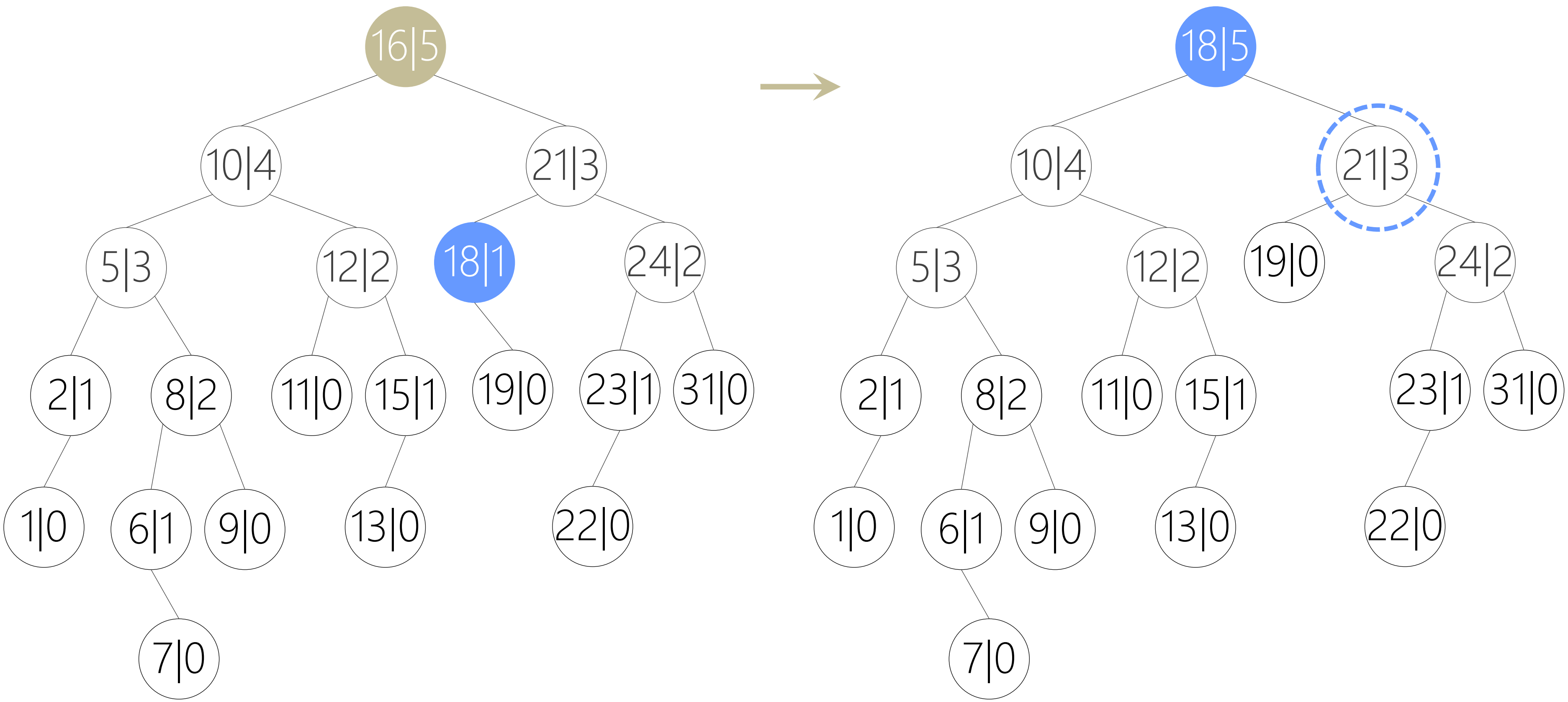
- Delete Node 4, Node 9 is unbalanced
- Node 9 is right heavy, and Node 16 is perfectly balanced
- Can treat it as Case 4 (right-right) or Case 3 (right-left)

Treat it as Case 4 since it's easier 

Review of the Delete Procedure

1. Delete Node from BST (recursive!)
2. Update Heights
3. Check Balance
    3.1 Violation?
        3.1.1 Determine Case
        3.1.2 Perform Rotations
4. Return Deleted Node

A Complete Delete Example

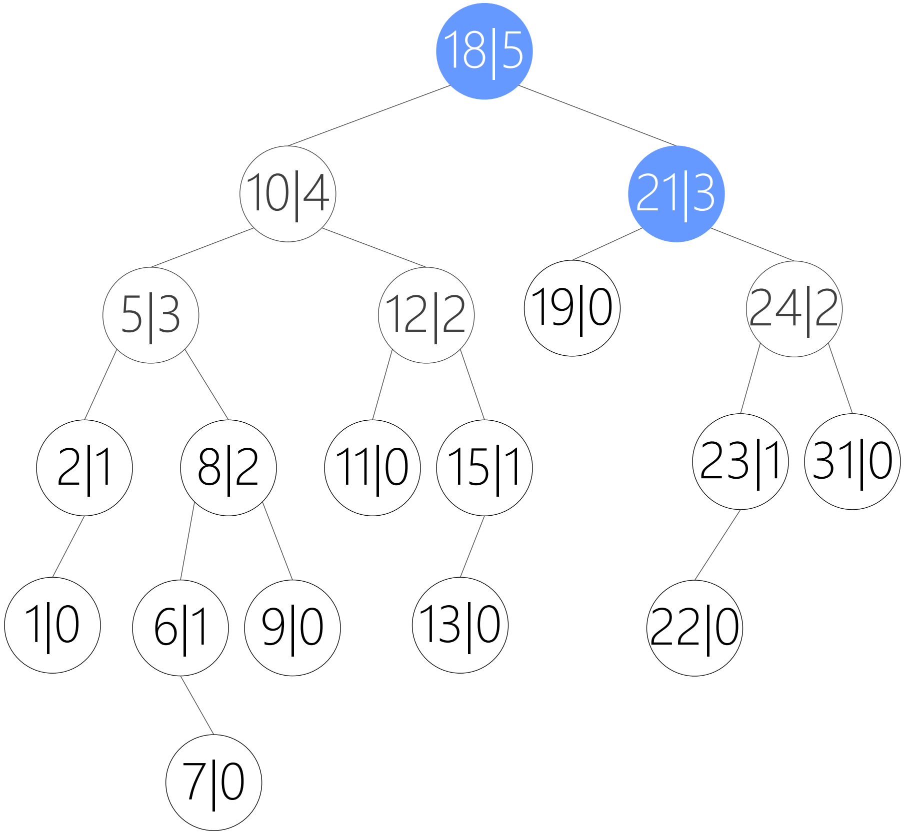


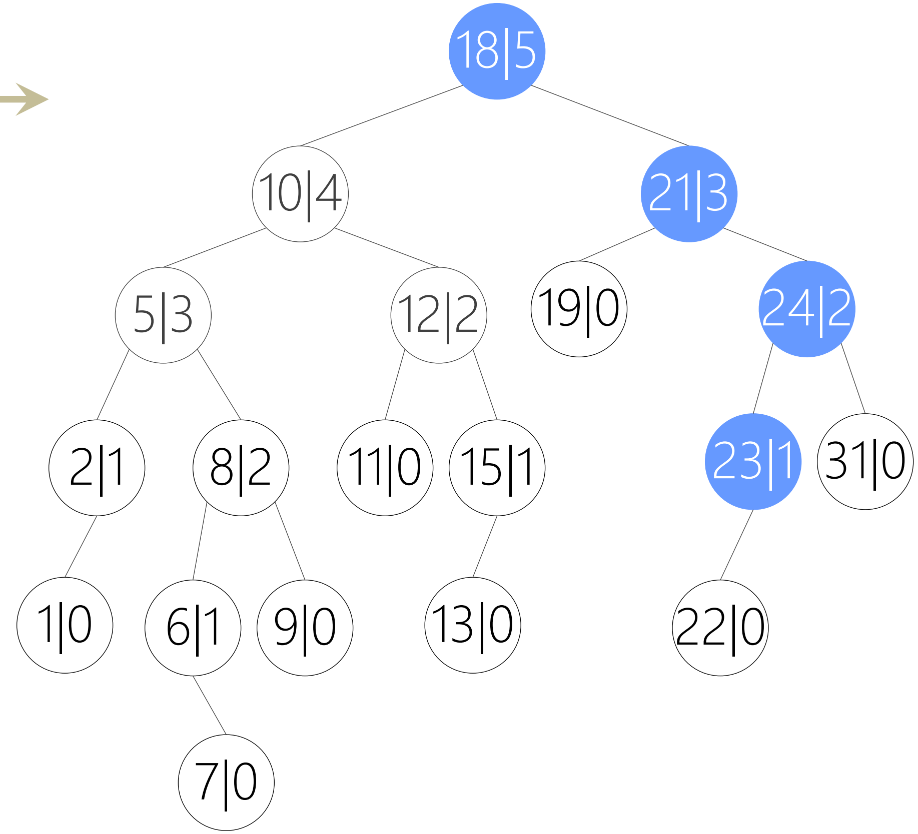
Delete Node 16

1

Replace root with node 18
Node 21's height is recomputed

A Complete Delete Example

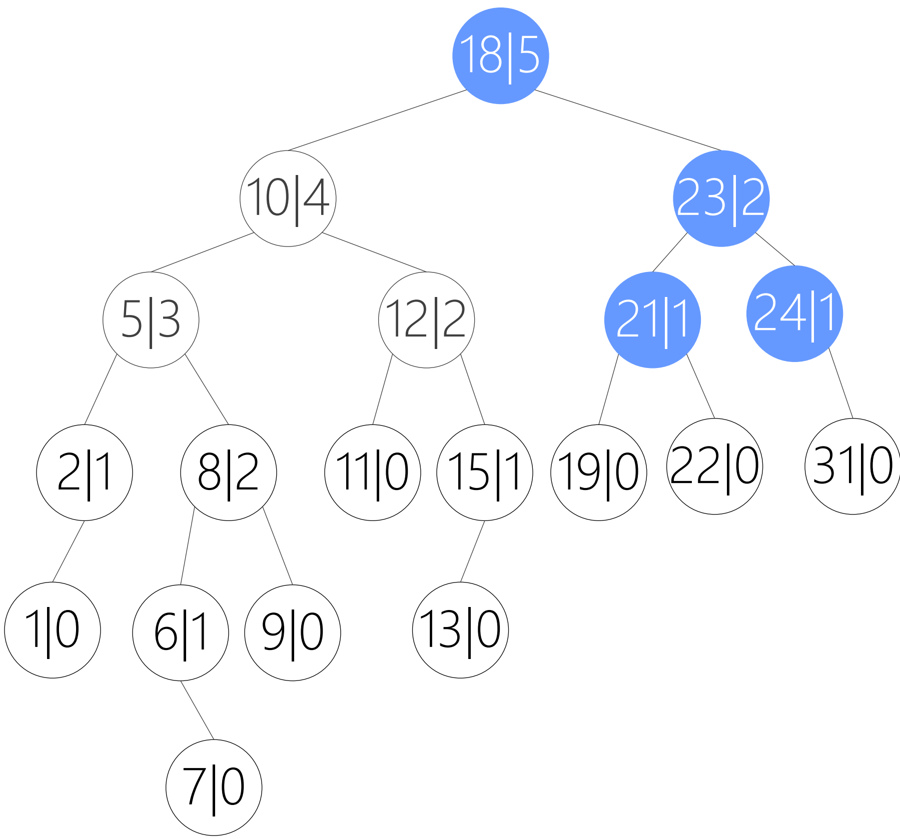




3.1
Node 21 is unbalanced

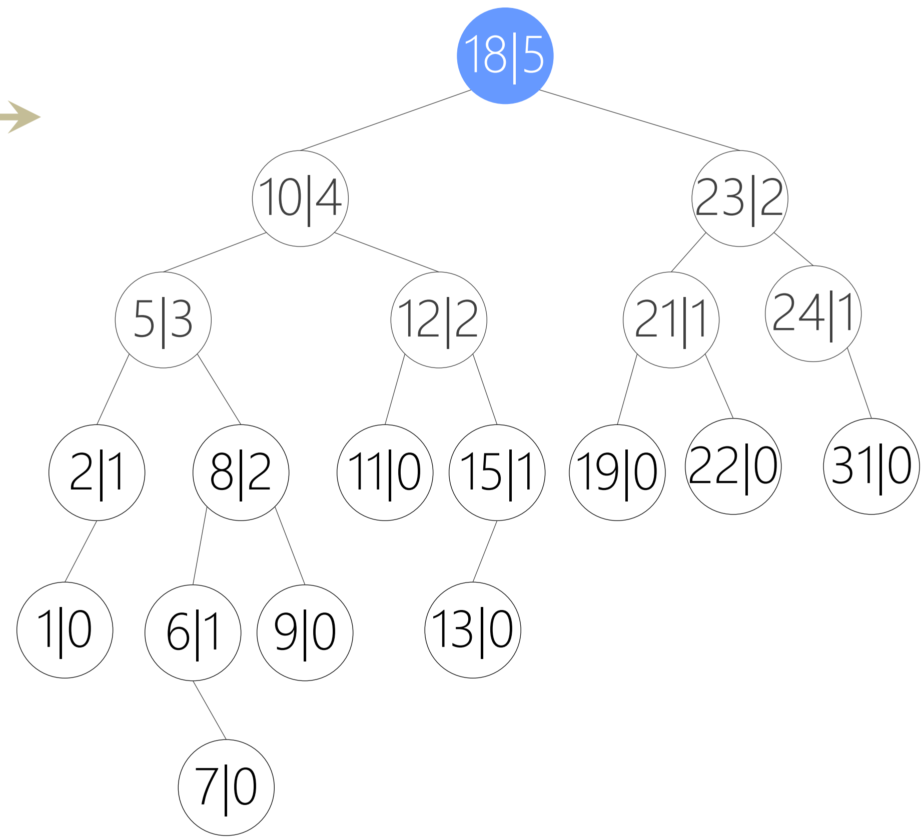
3.1.1
Case 3 Violation: right left

A Complete Delete Example



3.1.2

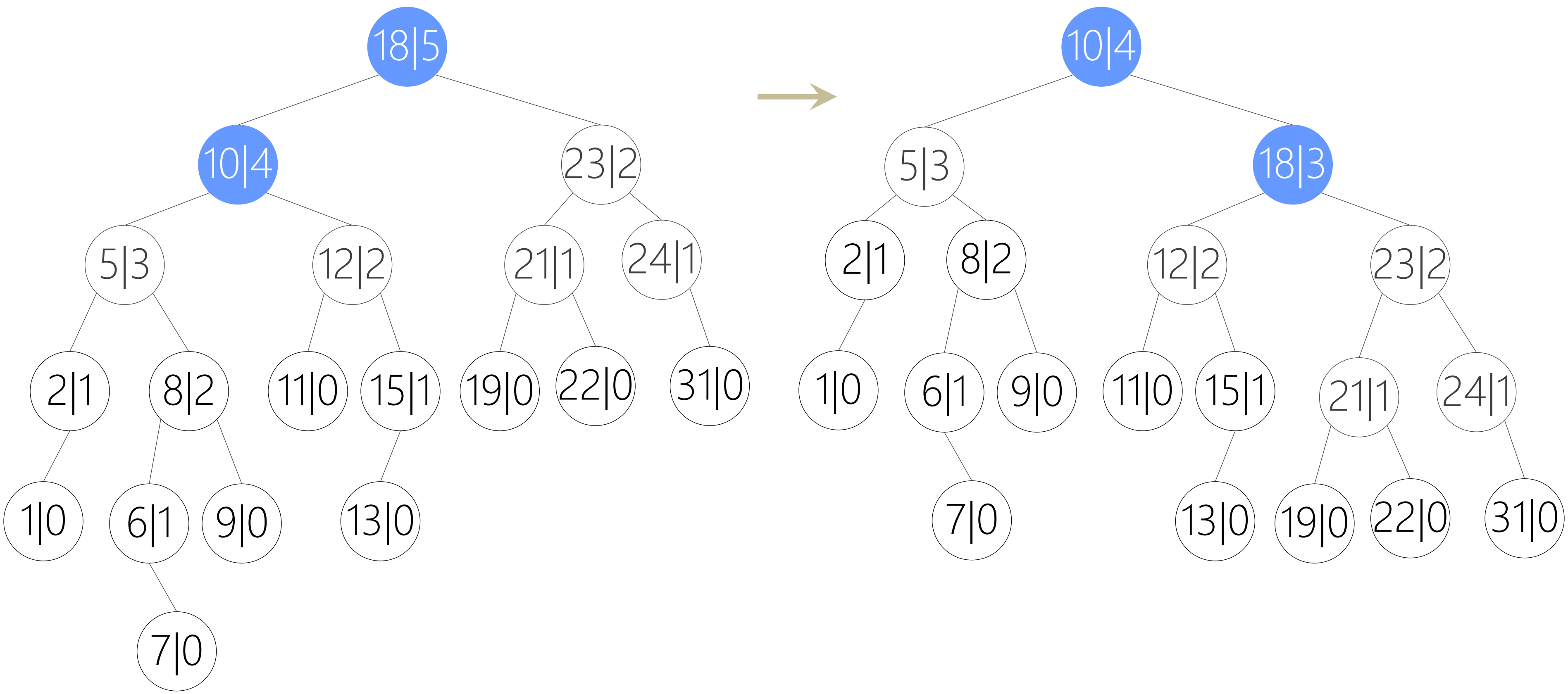
Perform a double rotation
Sub-tree height is updated



1

Node 18's height is updated

A Complete Delete Example



3.1

Node 18 is unbalanced

3.1.1

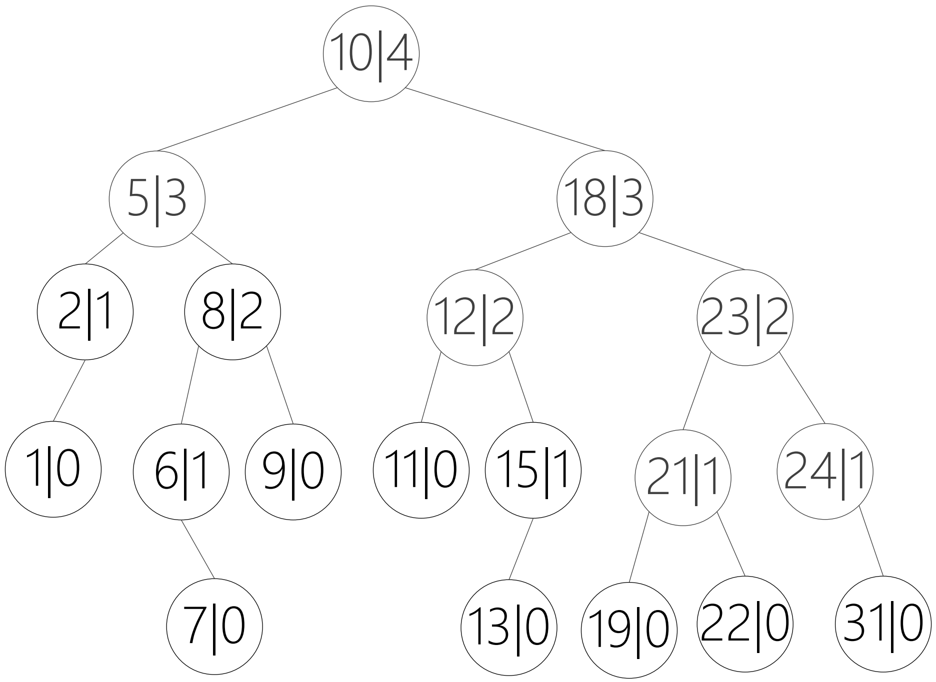
Case 1 Violation: left left

3.1.2

Perform a single right rotation

Sub-tree height is updated

A Complete Delete Example



4

Return Node 16.

Complete!

Pseudo Code for Deletion

Returns the (updated) root of the subtree after deletion

```
DELETE-NODE(root, x)
1.  root = BST-DELETE-NODE(root, x)
2.  IF root != Null
3.      root.UPDATE-HEIGHT()
4.      root =REBALANCE(root)
5.  return root
```

Pseudo Code for Deletion

Returns the (updated) root of the subtree after deletion

```
BST-DELETE-NODE(root, x)
1.  If root=Null
2.      return Null
3.  IF x<root.key
4.      root.left=DELETE-NODE(root.left, x)
5.  ELSE IF x>root.key
6.      root.right=DELETE-NODE(root.right, x)
7.  ELSE
8.      root=BST-DELETE-ROOT(root)
9.  Return root
```

DeleteRoot

```
BST-DELETE-ROOT(root)
// removes the root and returns the updated root
// of the subtree
IF root.left=Null
    return=root.right
IF root.right=Null
    return root.left
// root has two children
root.key=MIN-KEY(root.right)
root.right = BST-DELETE-NODE(root.right, root.key)
return root
```

Task

- Given Node.java, BinaryTreePrinter.java, BST.java and the skeleton of AVLTree.java, complete AVLTree.java
  - Node.java: implements the class for an AVL tree node
  - BinaryTreePrinter.java : implements a method to print out a binary tree
  - BST.java: implements the class for a binary search tree
  - AVLTree.java: implements the class for an AVL Tree
    - The class *AVLTree* is implemented as a subclass of *BST*
    - Read the skeleton and complete all the methods
    - This is the only file that you are going to modify
    - You may add (a lot of) auxiliary functions if there is a need
    - a main function is provided for testing purpose
- Submit AVLTree.java to iSpace.