Data Structures and Algorithms

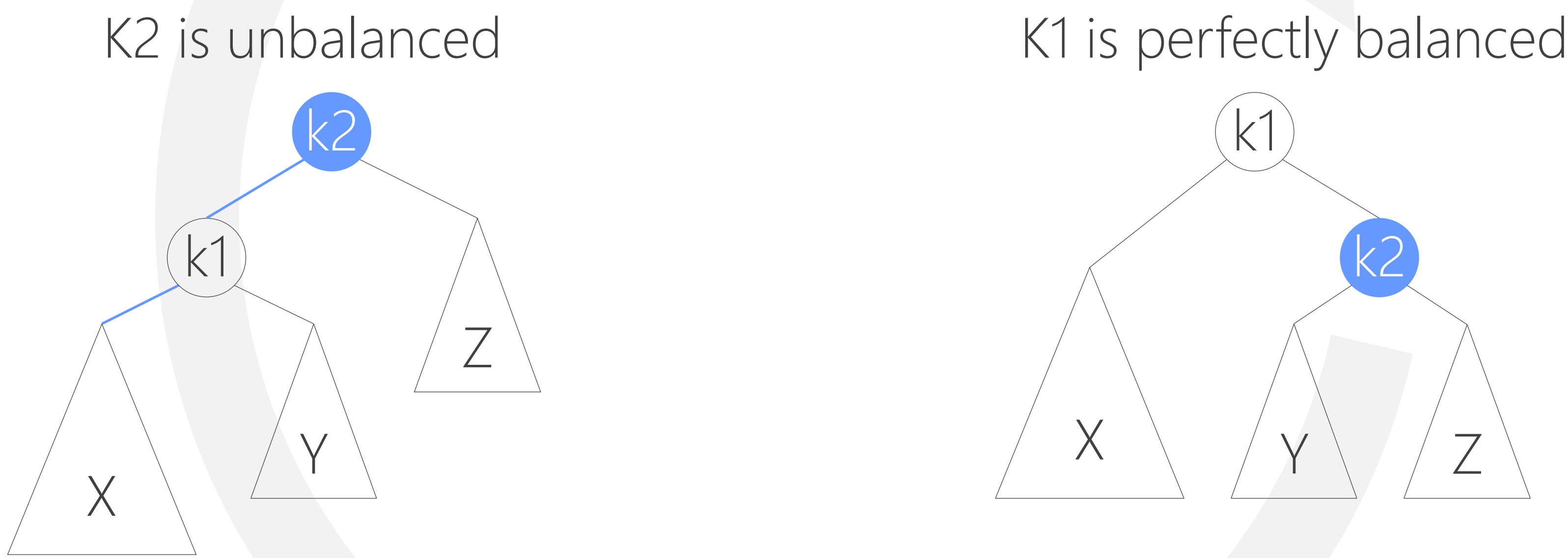
Lecture 10: AVL Trees II

Department of Computer Science & Technology
United International College

Review of AVL Tree Insertion

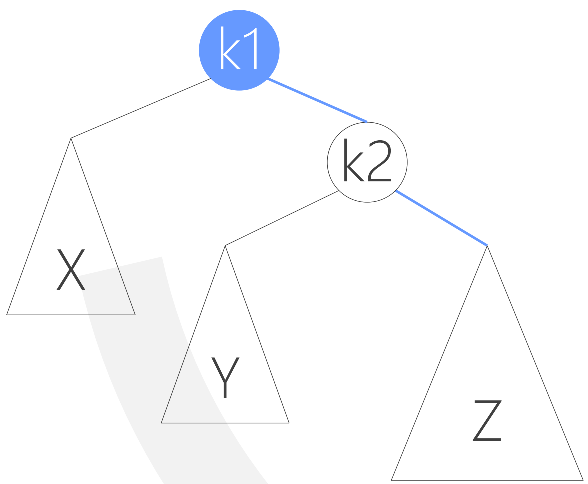
- The complete procedure of insertion
  1. Insert the new node to a proper position
  2. Starting from the new node, search upward for the first unbalanced node
     - Suppose that the height difference of a node's left and right sub-tree is *d*
       - *d*<=1 → the node is balanced
       - *d*=0 → 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)

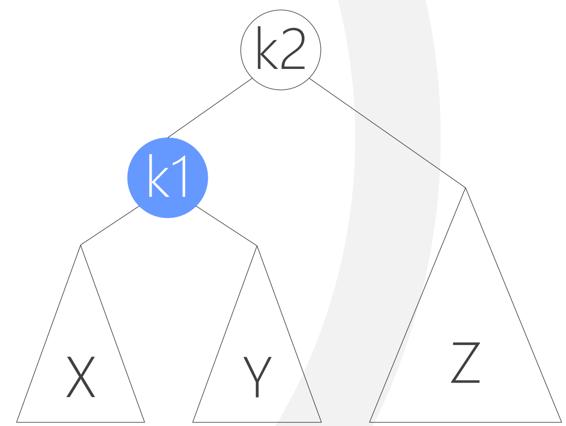


Single Left Rotation to Fix Case 4 (right-right)

K2 is unbalanced

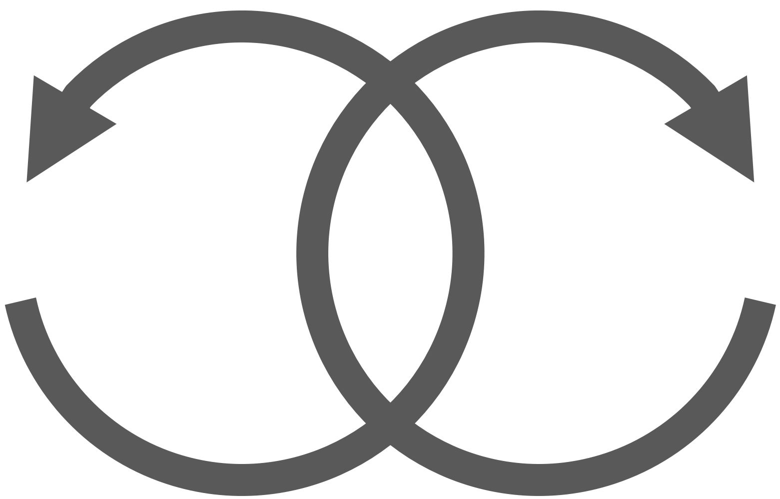


K2 is perfectly balanced



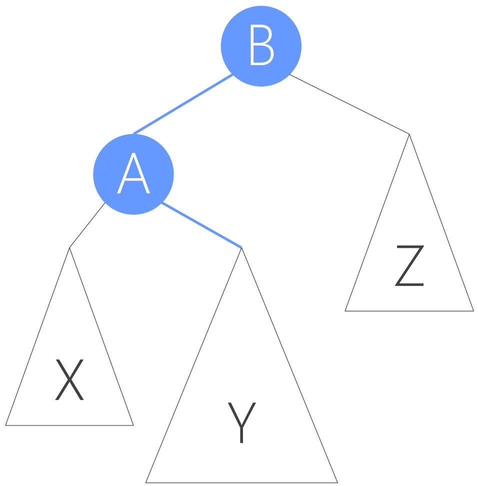
Double Rotation to Fix Case 2&3

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

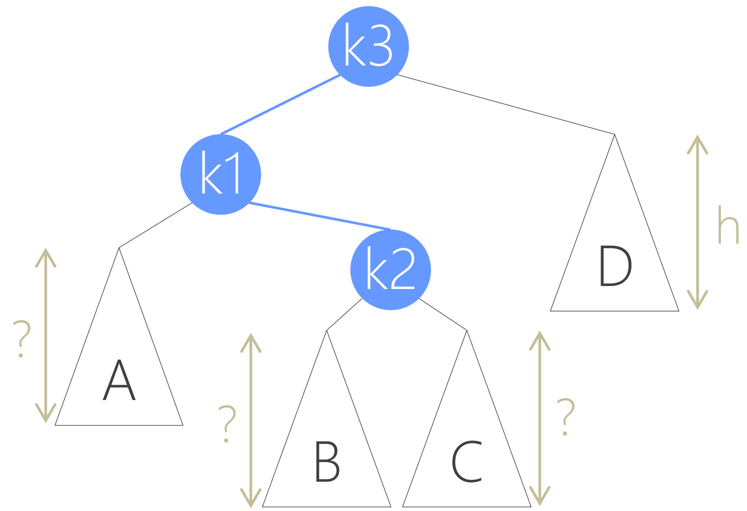


Case 2 (left-right)

B is unbalanced



Label B's child



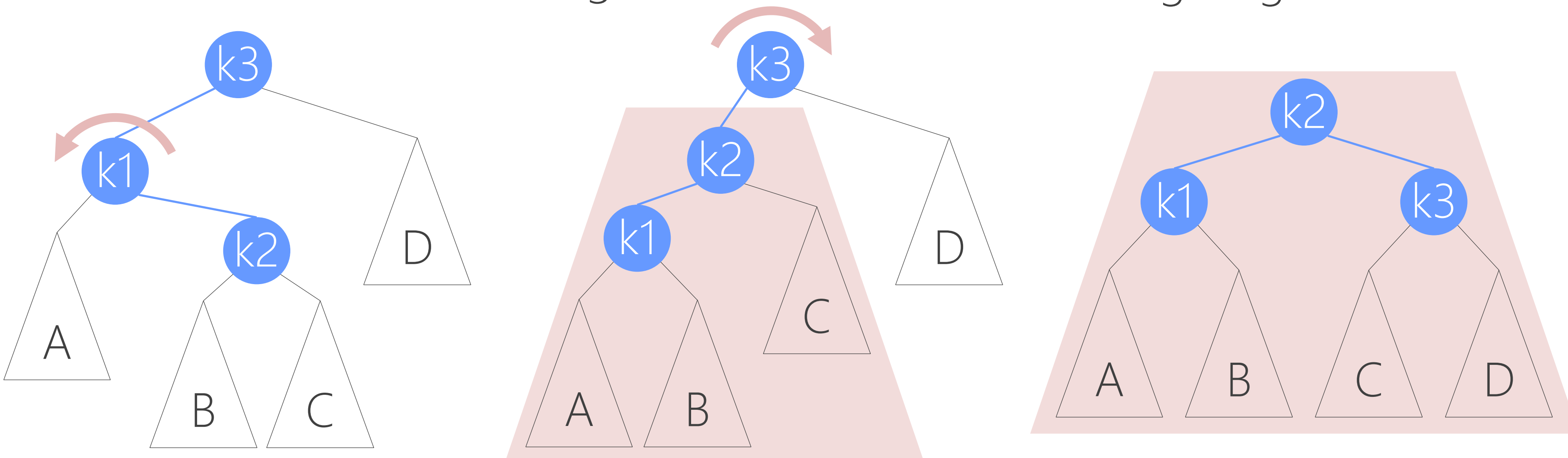
- If the height of sub-tree D is h
  - What is the possible height of A, B and C?

Double Rotation to Fix Case 2

k3 is unbalanced

Single left rotation

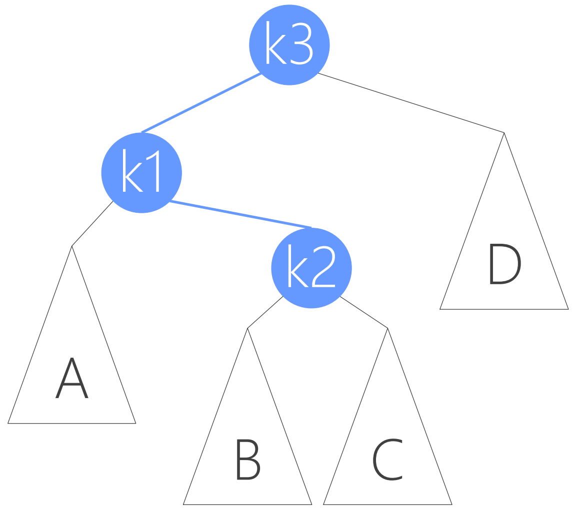
Single right rotation



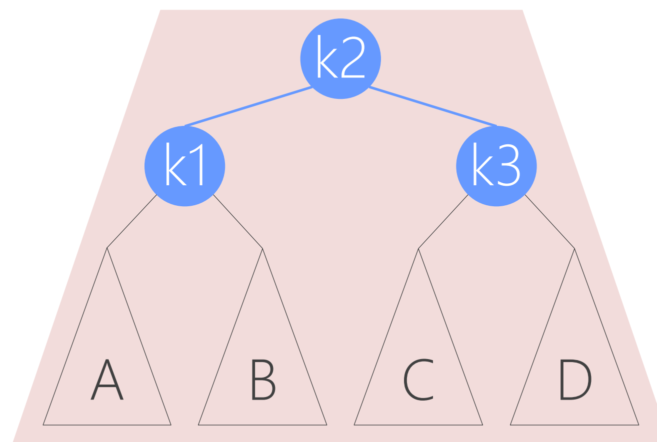
K2 is perfectly balanced

Direct Re-Arrangement

k3 is unbalanced



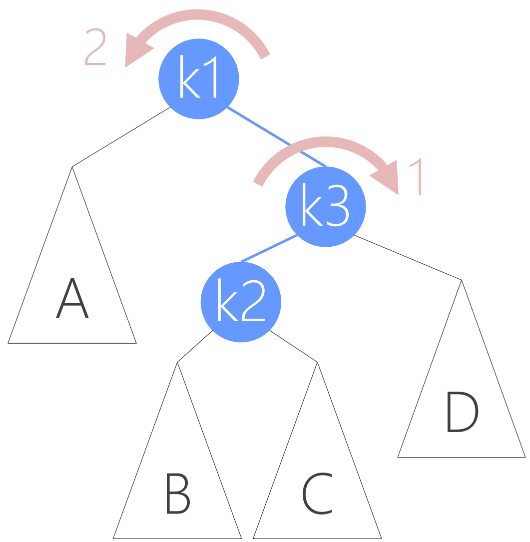
K2 is perfectly balanced



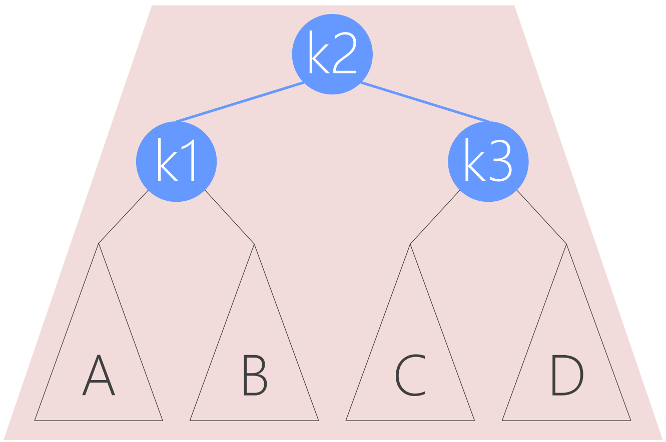
- Pre-condition: k3-k1-k2 forms a zig-zag shape
- Post-condition: k2 is the parent of k1 and k3

Case 3 (right-left)

k3 is unbalanced



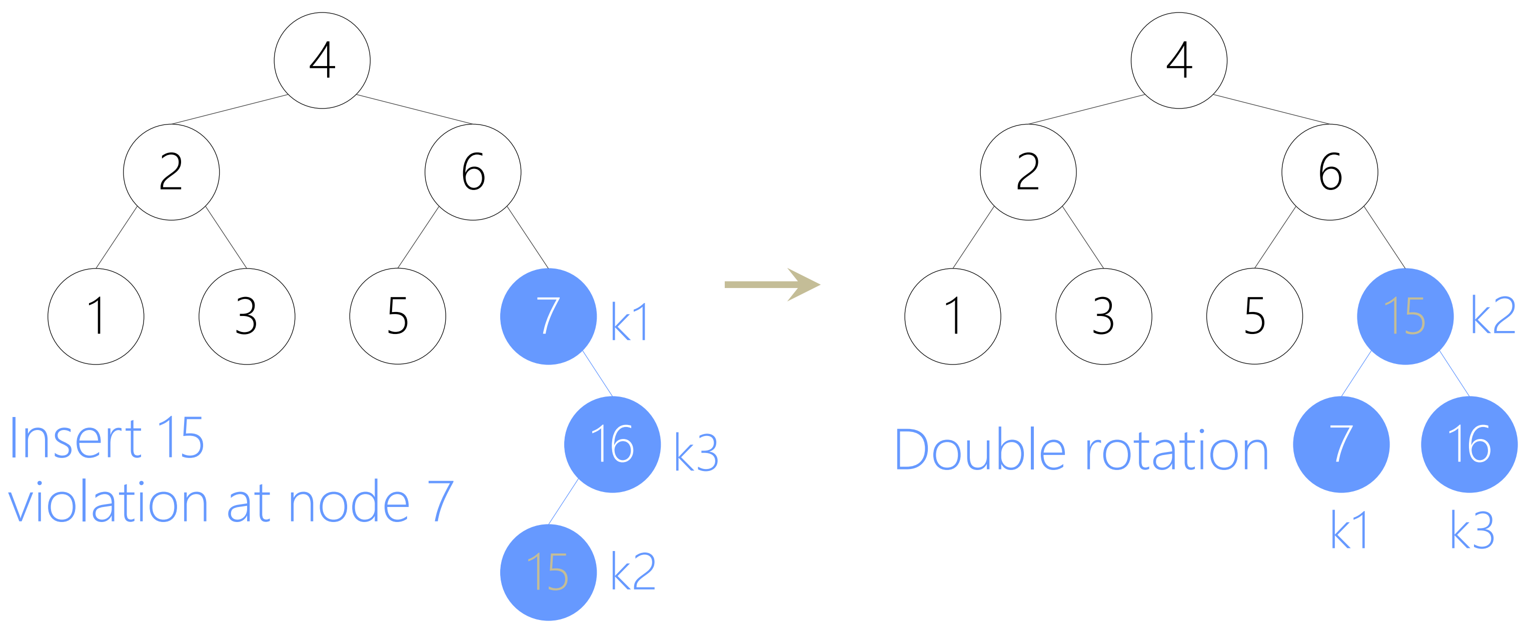
K2 is perfectly balanced

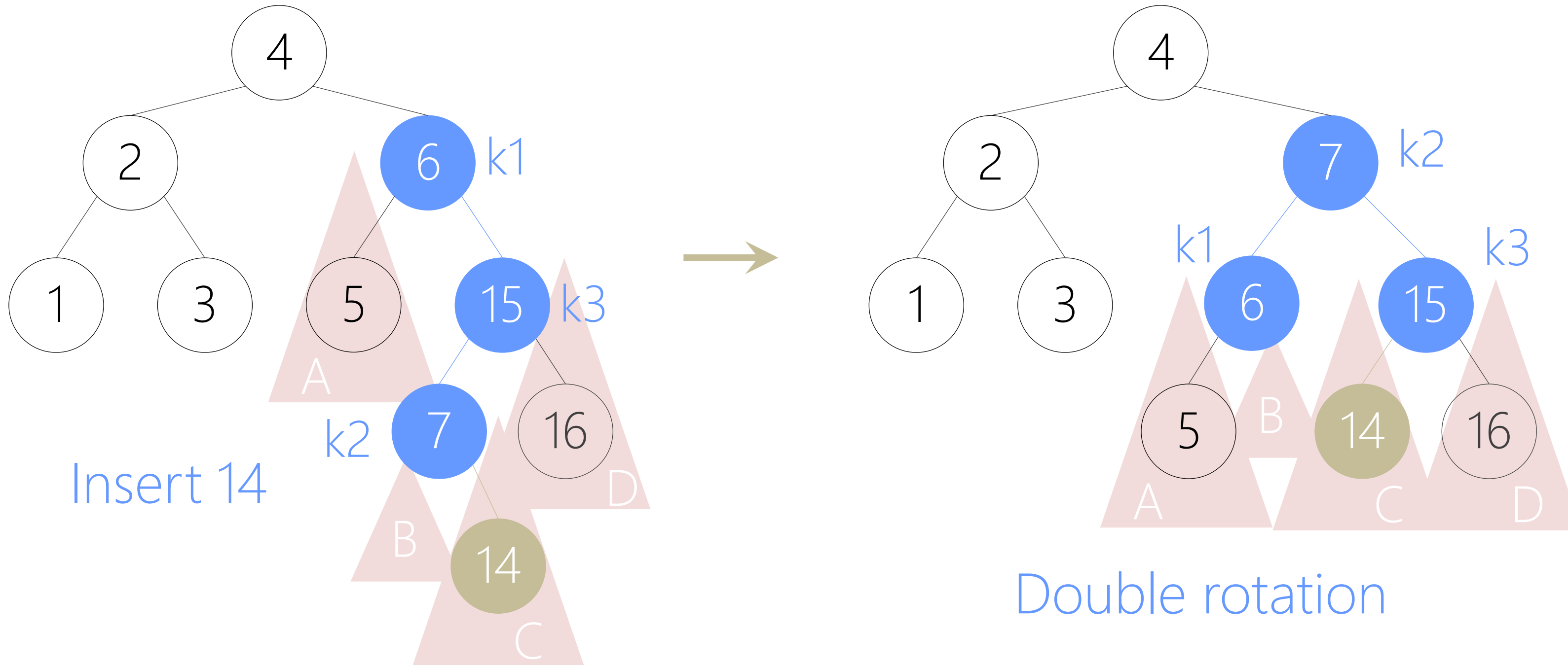


- Case 3 is symmetric to Case 2
- Pre-condition: k1-k3-k2 forms a zig-zag shape
- Post-condition: k2 is the parent of k1 and k3

Example

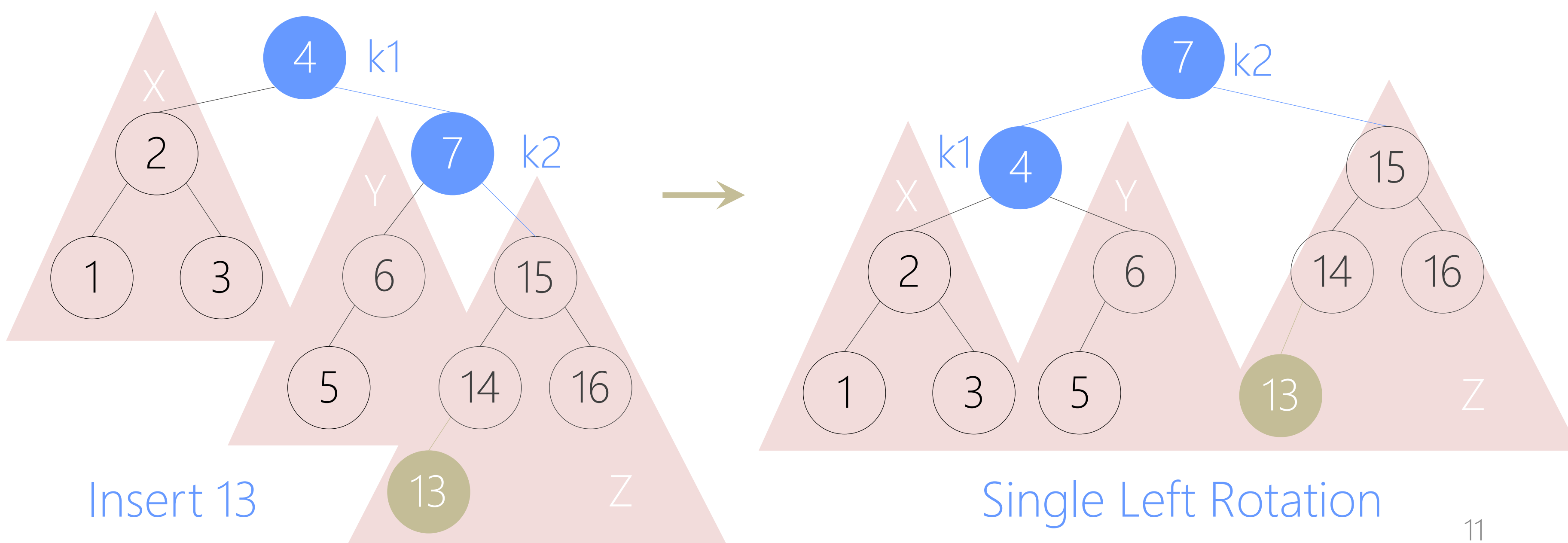
- Continue our example
  - We've inserted 3, 2, 1, 4, 5, 6, 7, 16
  - We'll insert 15, 14, 13, 12, 11, 10, 8, 9





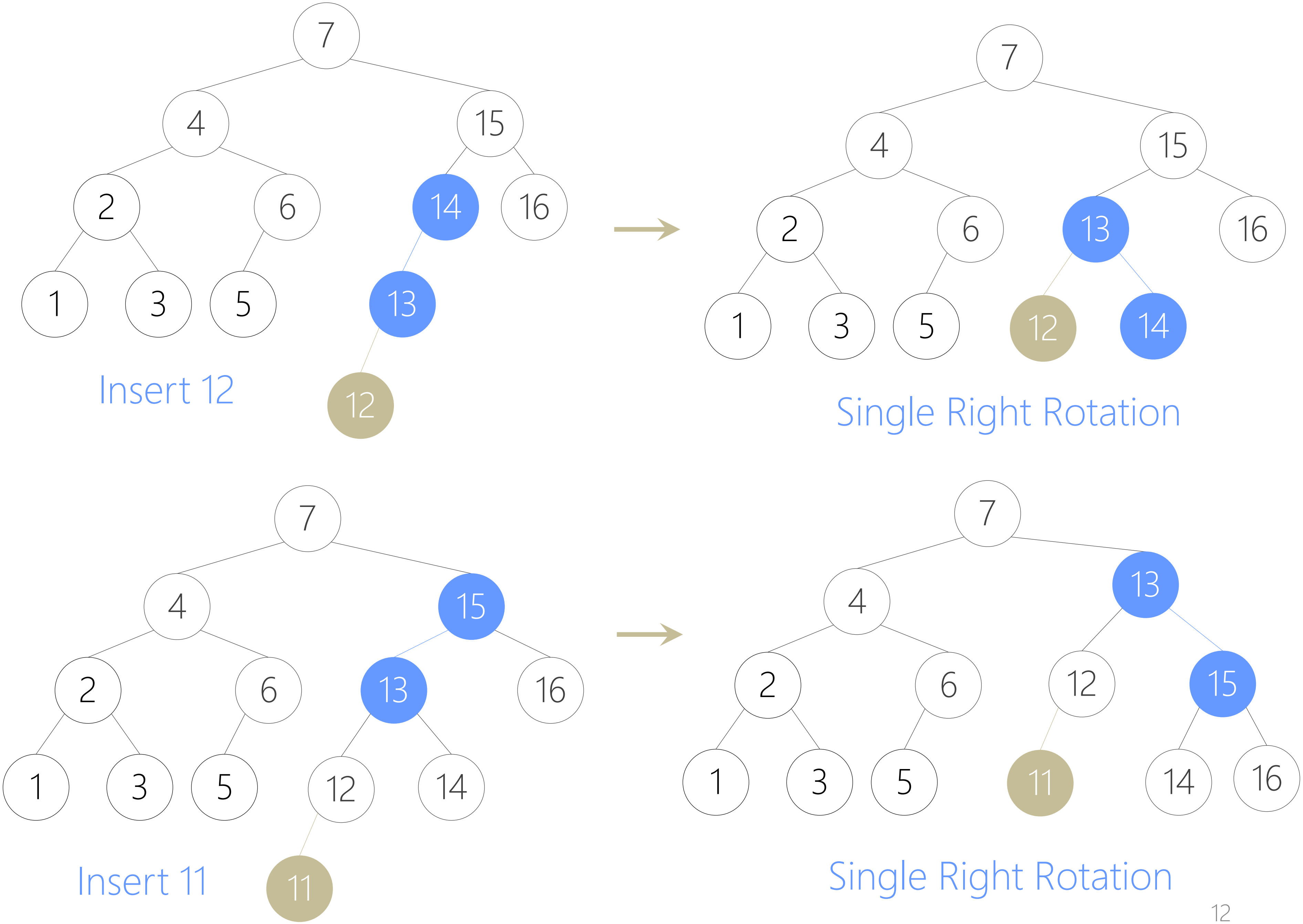
Insert 14

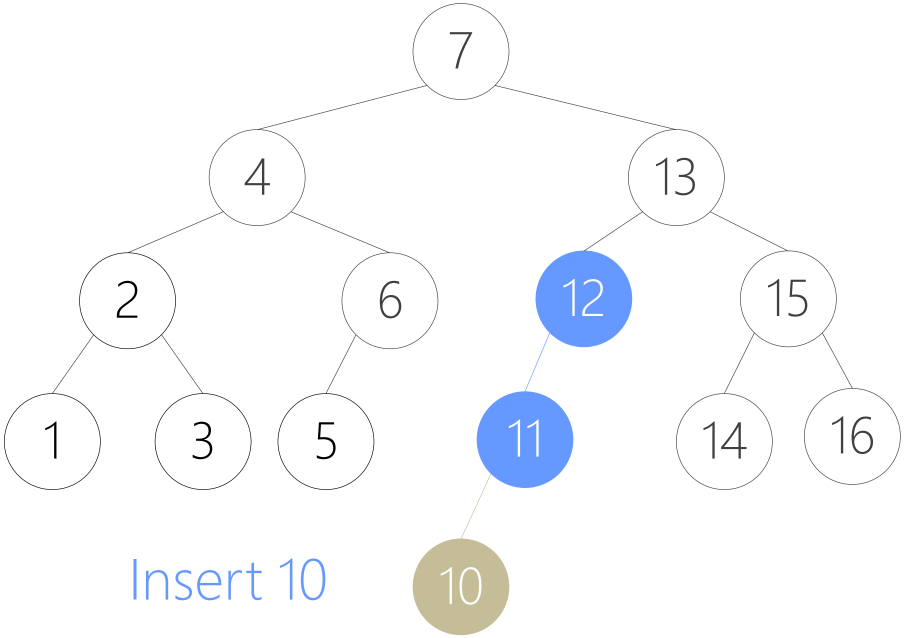
Double rotation



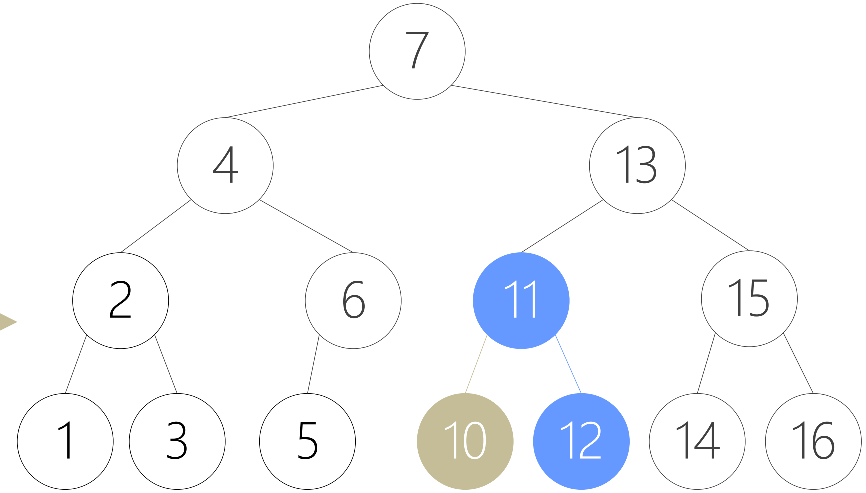
Insert 13

Single Left Rotation

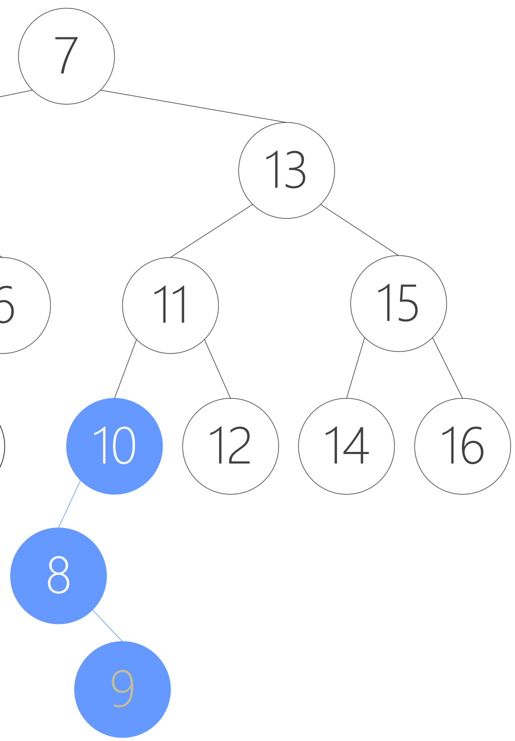

Insert 12
Single Right Rotation
Insert 11
Single Right Rotation



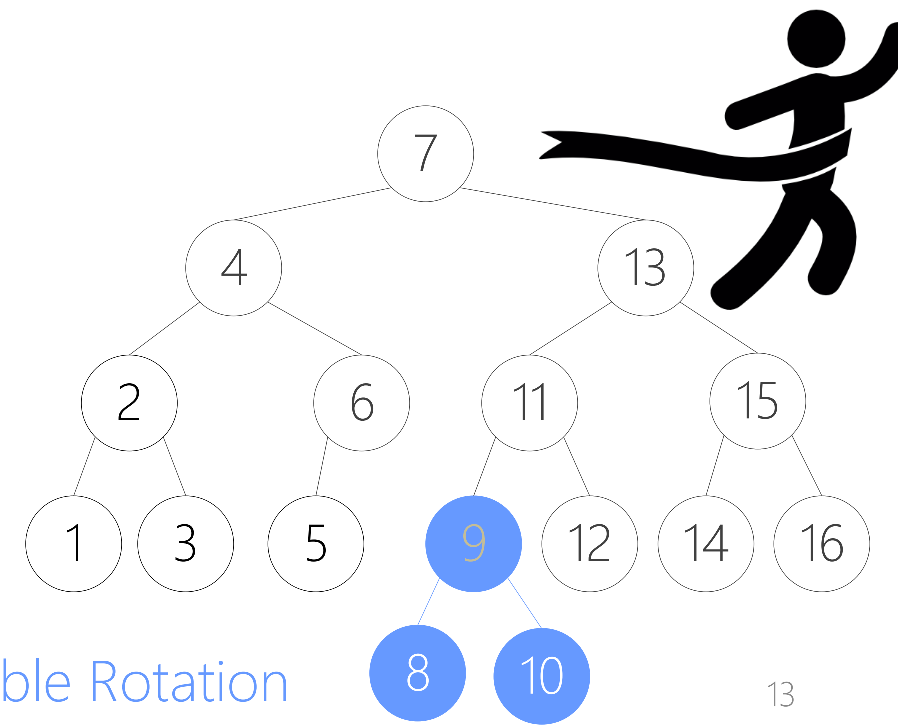
Insert 10



Single Right Rotation

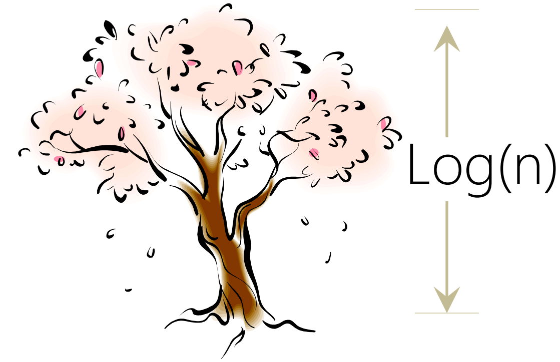


Insert 8, fine
Then insert 9



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

```
typedef struct AVLNode{
    object data;
    int height;
    AVLNode *left, *right;
}AVLNode;
```

Pseudo Code for Insertion

`root` should be a pointer to pointer

INSERT-NODE(root, x)
1. If root=Null
2. return root=CREATE-NODE(x)
3. IF root->key=x
4. return Null
5. IF root->key>x
6. newNode= INSERT-NODE(root->left, x)
7. ELSE
8. newNode= INSERT-NODE(root->right, x)
9. UPDATE-HEIGHT(root)
10. REBALANCE(root)
11. Return newNode

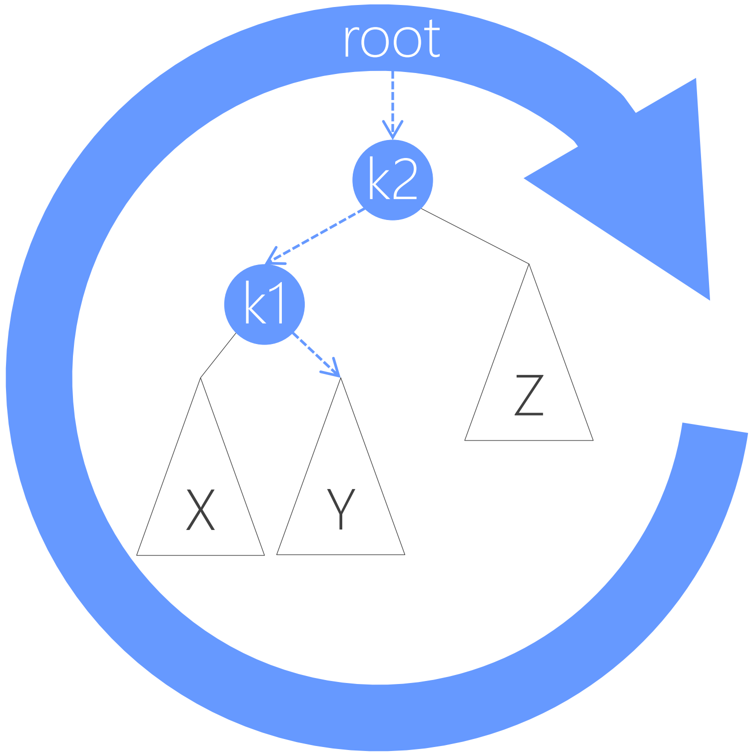
root's height is: (height of its deeper sub-tree) + 1

Rebalance

`root` should be a pointer to pointer

```
REBALANCE(root)
IF BALANCED(root)
    return
IF CASE1(root)  // left left
    RIGHT-ROTATE(root)
IF CASE4(root) // right right
    LEFT-ROTATE(root)
IF CASE2(root) // left right
    LEFT-ROTATE(root->left)
    RIGHT-ROTATE(root)
IF CASE3(root) // right left
    RIGHT-ROTATE(root->right)
    LEFT-ROTATE(root)
```

Rotation



RIGHT-ROTATE(root)

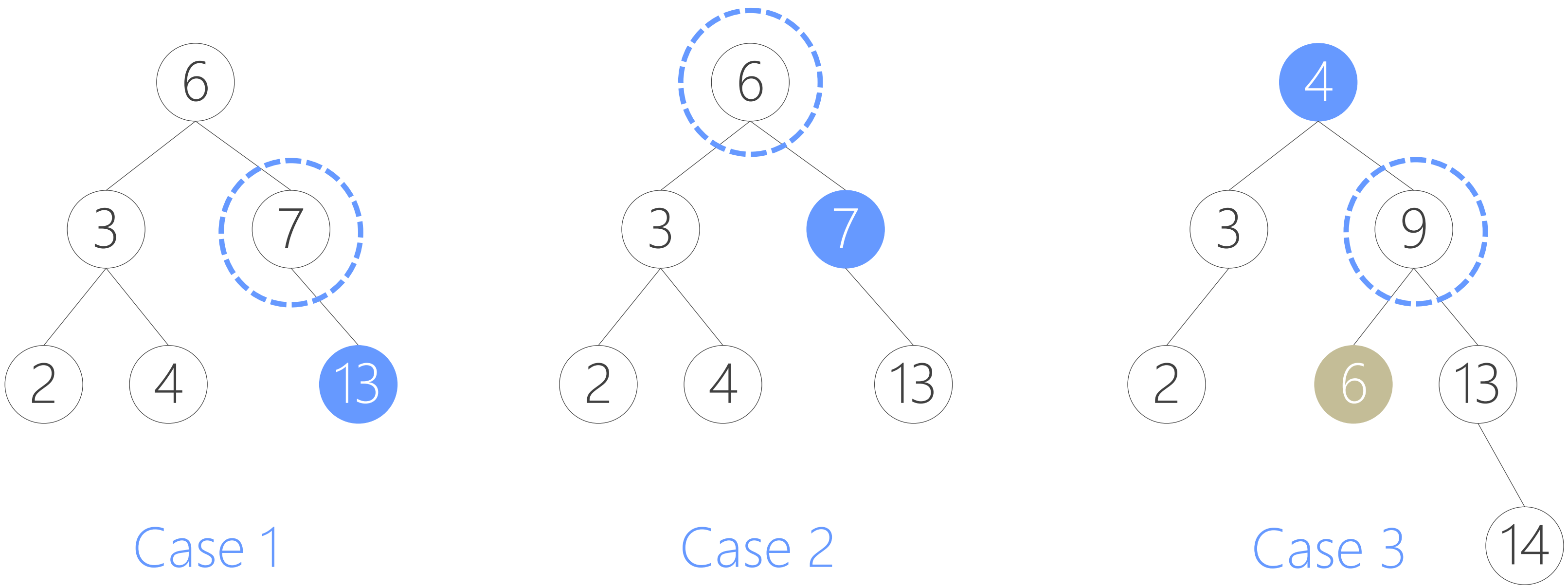
1. k2=root, k1=k2->left, Y=k1->right
2. root=k1
3. k1->right=k2
4. k2->left=Y
5. UPDATE-HEIGHT(k2)
6. UPDATE-HEIGHT(k1)

Notes on Rotations

- Three pointers are modified in a rotation
  - The `root` parameter should be a pointer to pointer
- 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



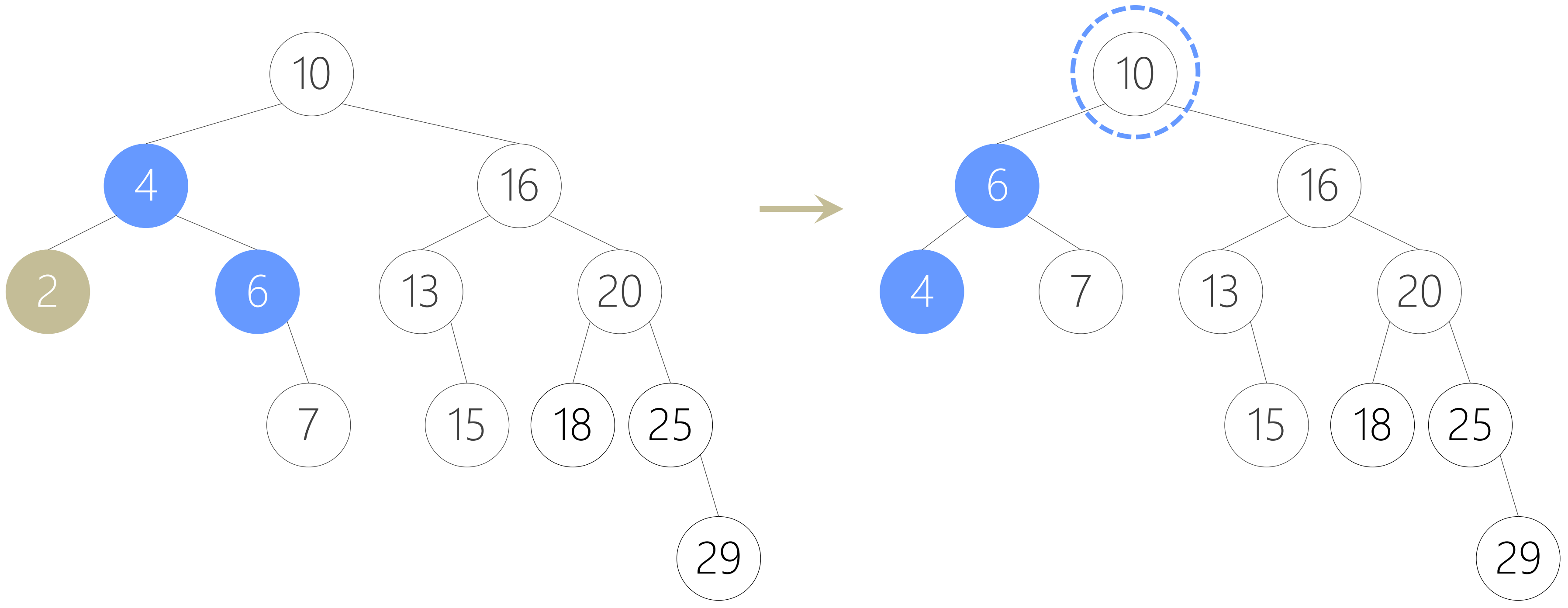
Deletion

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
   - Balanced: Proceed to parent(x)

   Continue to trace the path

   UNTIL WE REACH THE ROOT

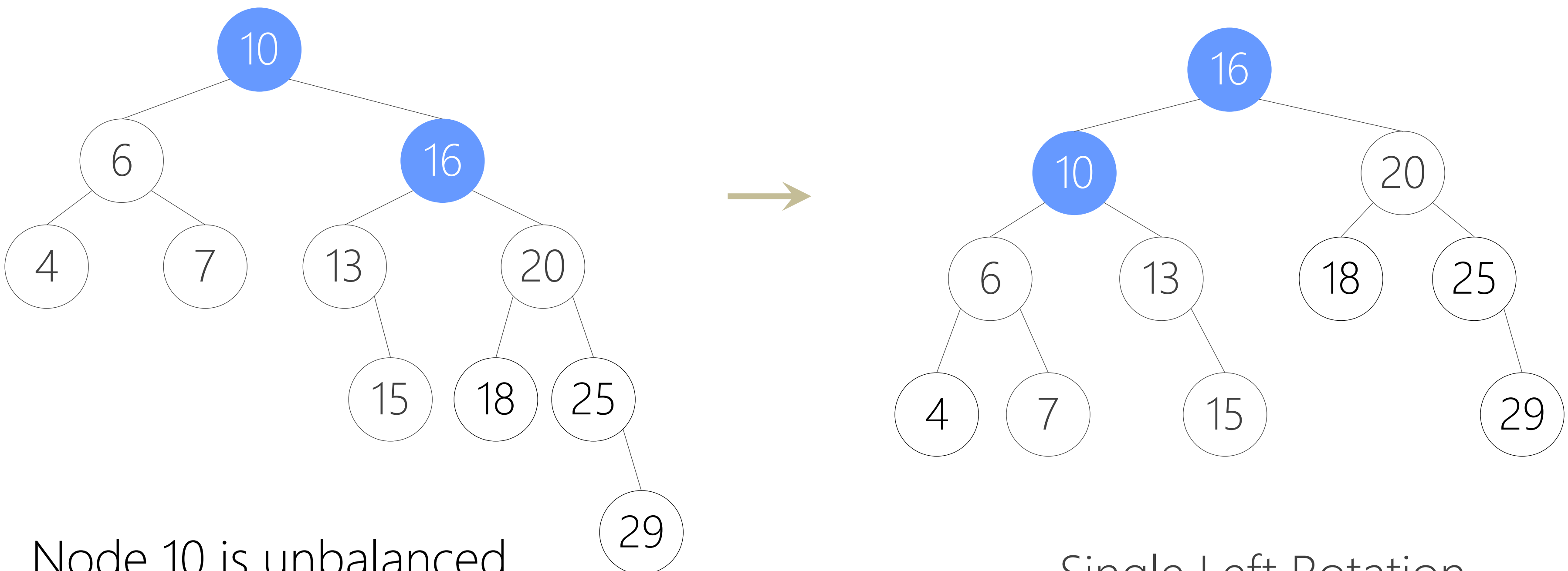
Delete Example



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

Single Left Rotation

Delete Example



Node 10 is unbalanced
CASE 4: right-right

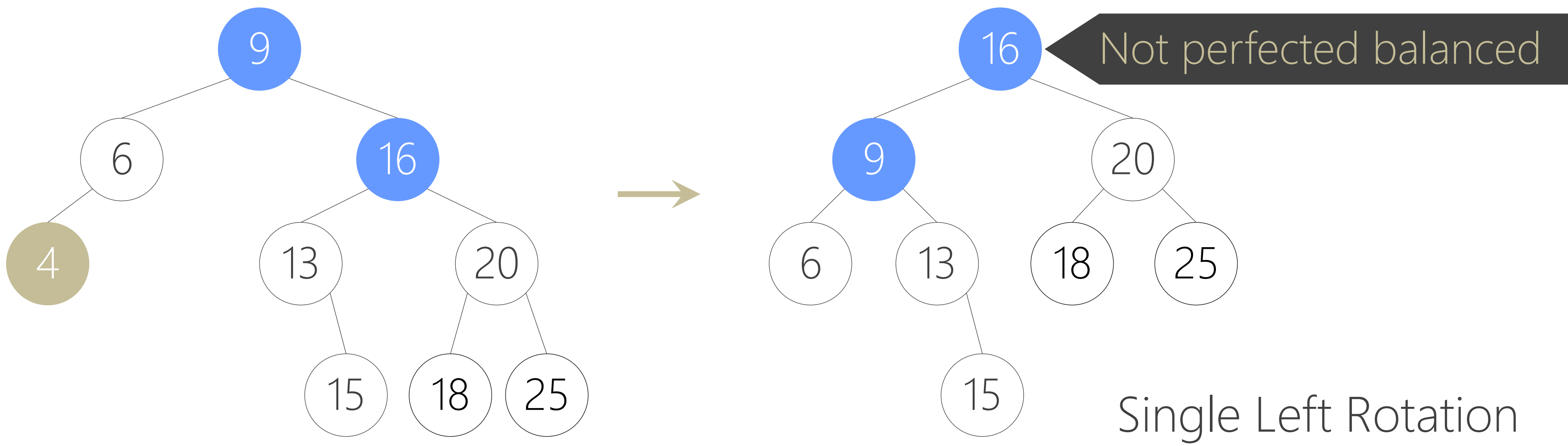
Single Left Rotation

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



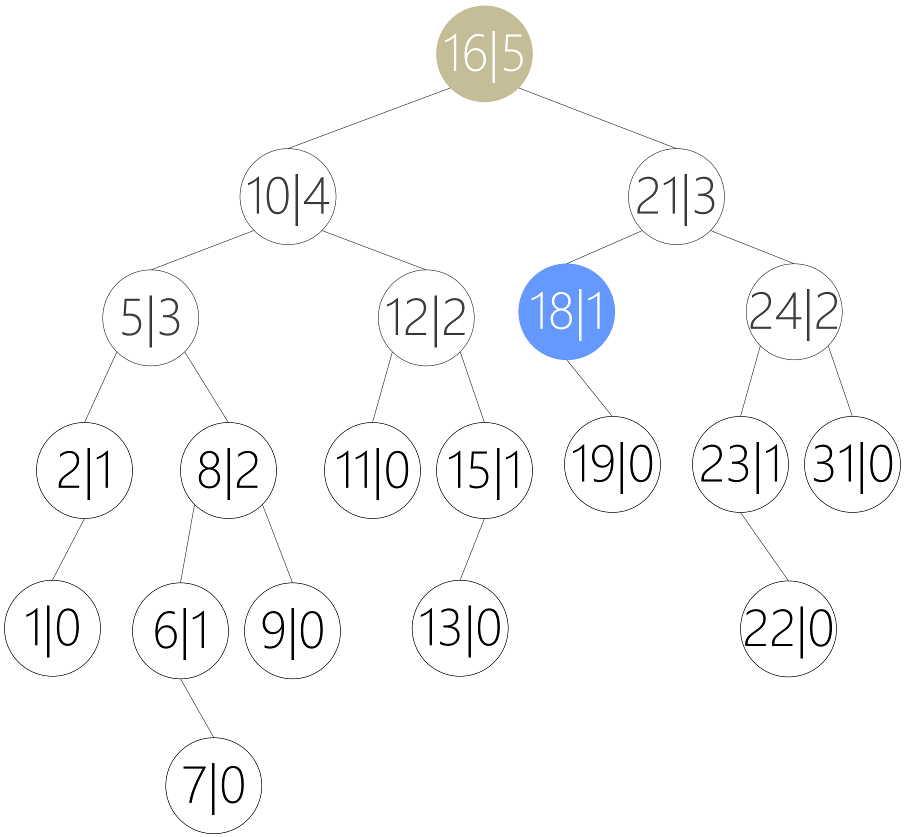
- Delete Node 4, Node 9 is unbalanced
- Node 9 is right heavy, and Node 16 is perfectly balanced
- Can treat it as Case 4 (right-right) or Case 3 (right-left)

Treat it as Case 4 since it's easier ☺

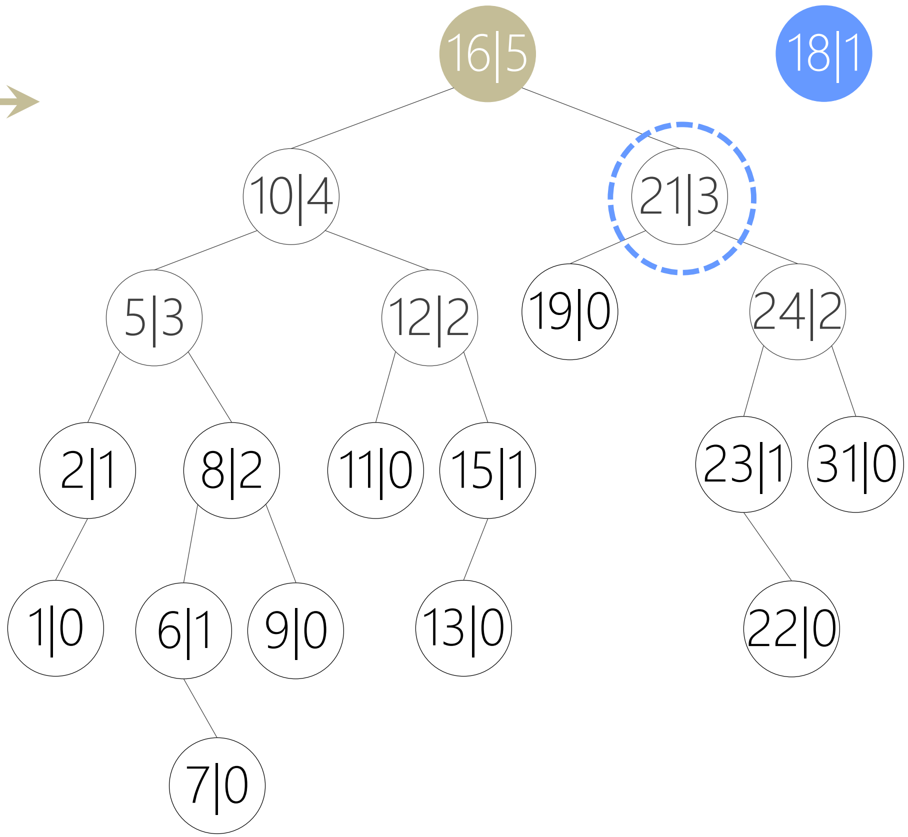
Review of the Delete Procedure

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



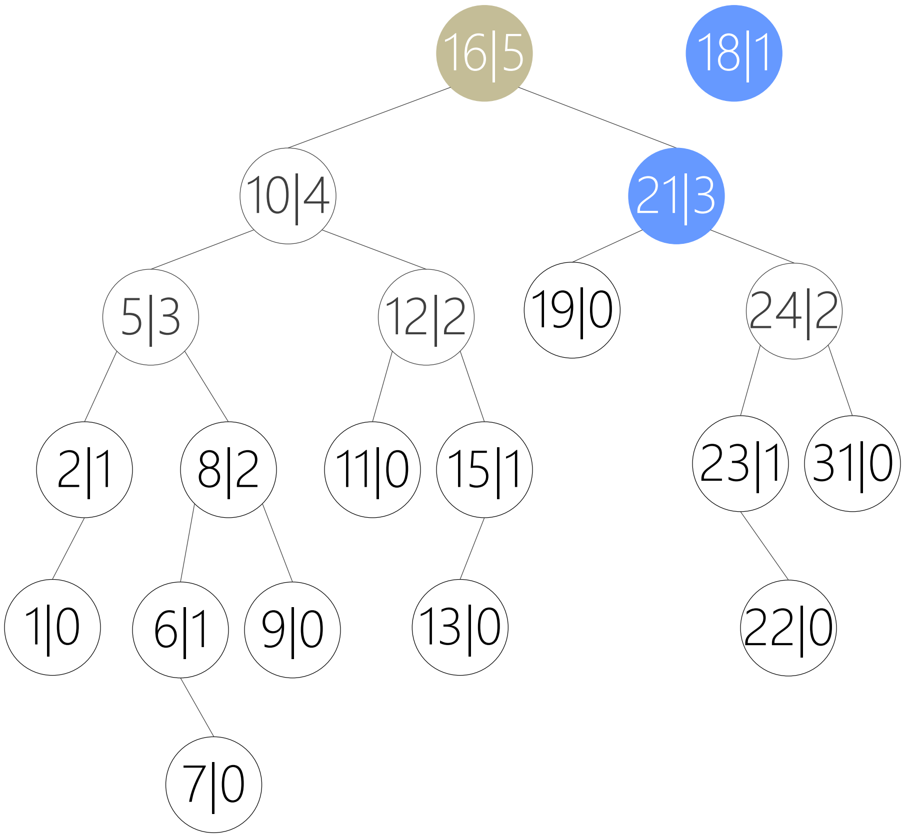
Delete Node 16



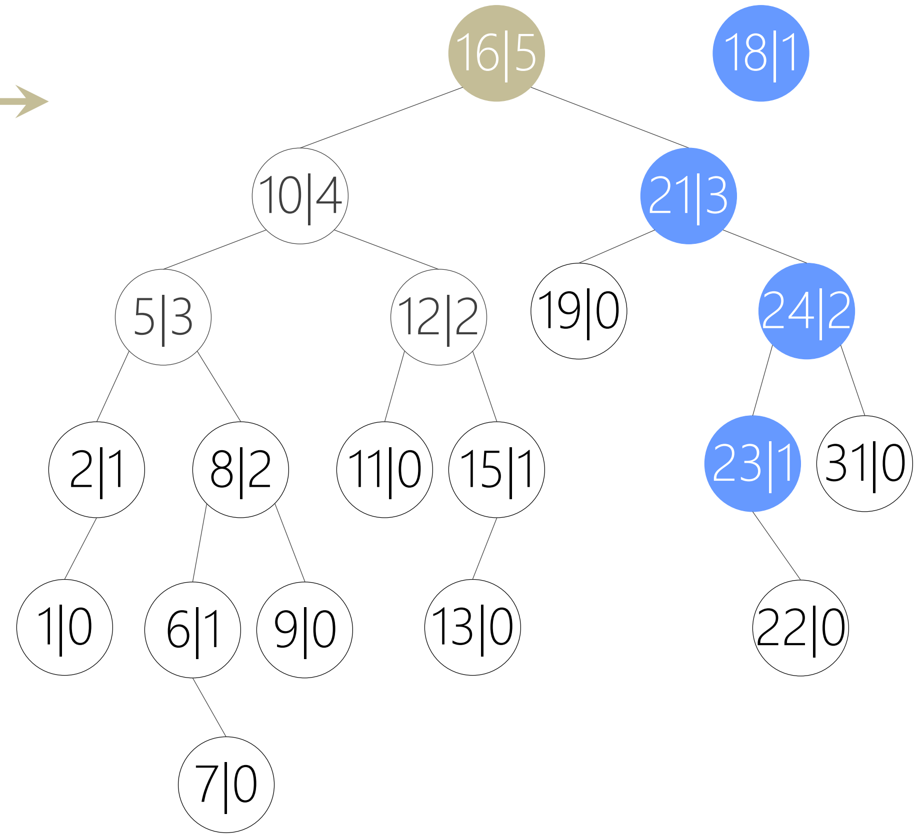
1

Delete Node 18
Node 21's height is recomputed

A Complete Delete Example

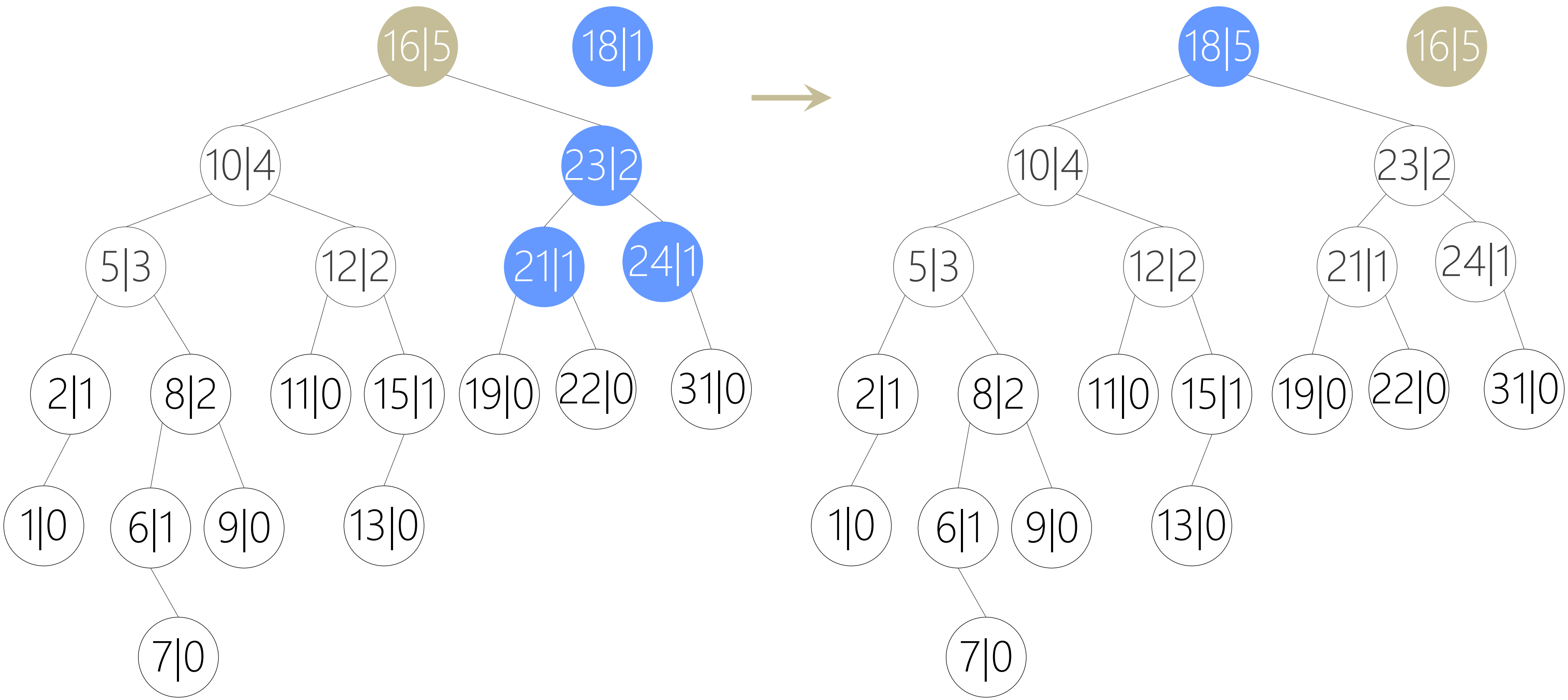


3.1
Node 21 is unbalanced



3.1.1
Case 3 Violation: right left

A Complete Delete Example



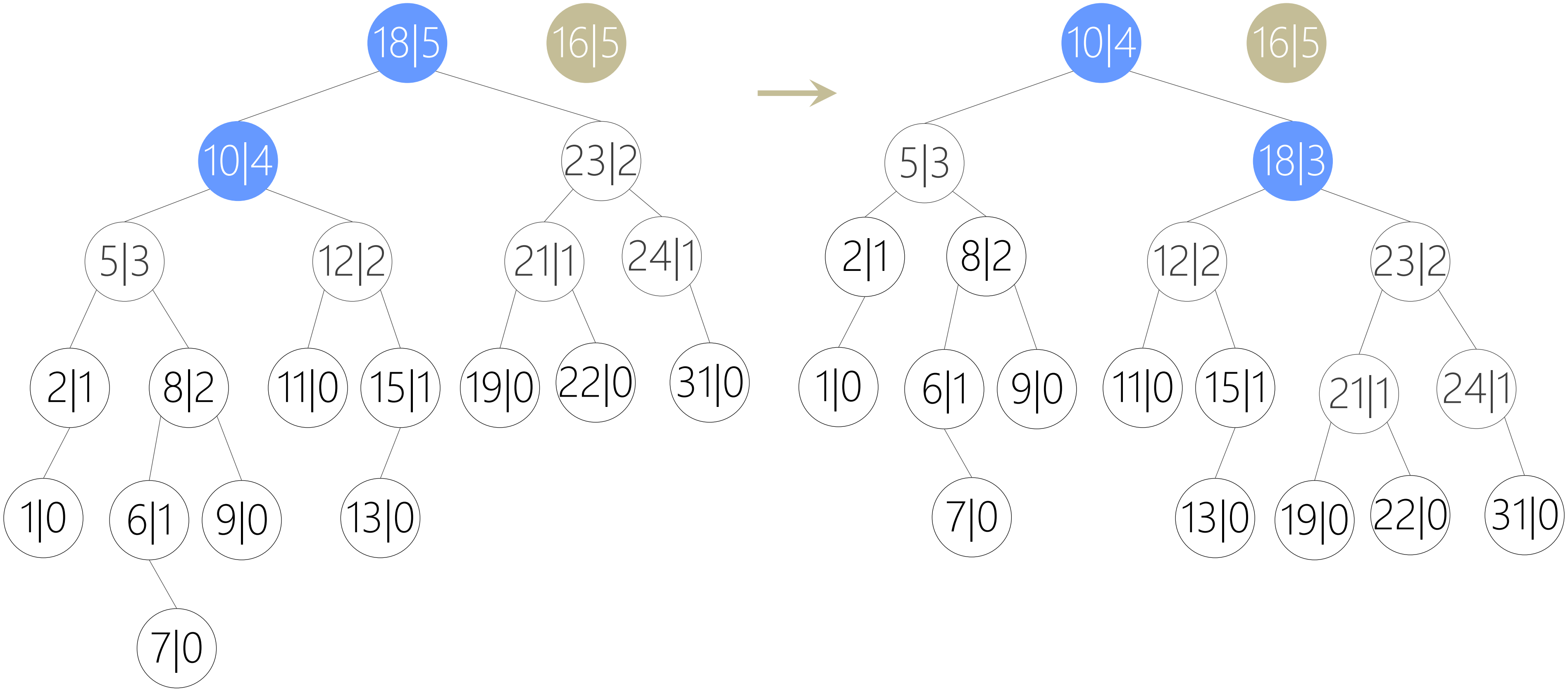
3.1.2

Perform a double rotation
Sub-tree height is updated

1

Replace Node 16 with Node 18
Node 18's height is updated

A Complete Delete Example



3.1

Node 18 is unbalanced

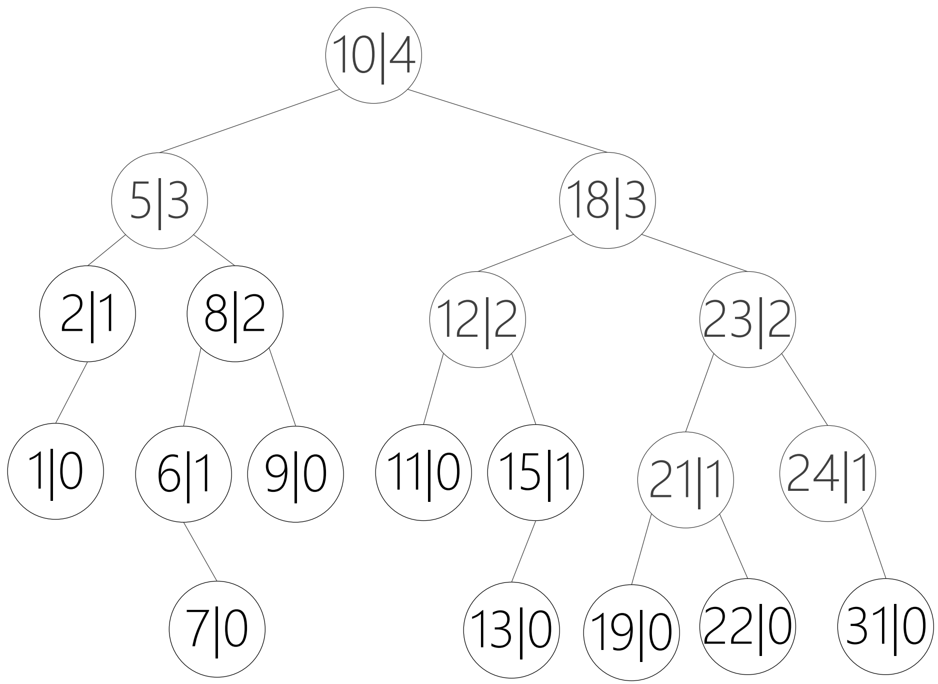
3.1.1

Case 1 Violation: left left

3.1.2

Perform a single right rotation
Sub-tree height is updated

A Complete Delete Example



16|5

4

Return Node 16.

Complete!

Pseudo Code for Deletion

`root` should be a pointer to pointer

```
DELETE-NODE(root, x)
If root=Null
    return Null
IF root->key>x
    matchNode=DELETE-NODE(root->left, x)
ELSE IF root->key<x
    matchNode=DELETE-NODE(root->right, x)
ELSE
    matchNode=DELETE-ROOT(root)
UPDATE-HEIGHT(root)
REBALANCE(root)
Return matchNode
```

DeleteRoot

`root` should be a pointer to pointer

```
DELETE-ROOT(root)
// remove and return the root
1.  currNode=root
2.  IF root->left=Null
3.      root=root->right
4.      return currNode
5.  IF root->right=Null
6.      root=root->left
7.      return currNode
8.  // root has two children
9.  minNode=DELETE-MIN(root->right)
10. minNode->left=root->left
11. minNode->right=root->right
12. root=minNode
13. Return currNode
```

DeleteMin

`root` should be a pointer to pointer

```
DELETE-MIN(root)
// remove and return the minimum node
// in the sub-tree lead by root
IF root->left=Null
    // root is the minimum node
    // and it has no left child
    minNode=root
    root=root->right
    return minNode
minNode = DELETE-MIN(root->left)
UPDATE-HEIGHT(root)
REBALANCE(root)
return minNode
```

Task

- Given AVL.h, printTree.cpp and main.cpp, complete AVL.cpp
  - AVL.h: the header file which defines the data and the methods of an AVL tree
  - printTree.cpp: implements the printTree method defined in AVL.h
  - AVL.cpp: implements the remaining methods defined in AVL.h
    - To be completed by you
    - This is the only file that you are going to modify
    - You may add (a lot of) auxiliary functions
    - findNode and destroyTree are the same as in a BST
  - main.cpp: a main function for testing purpose
- Submit AVL.cpp to iSpace.