Data Structures and Algorithms

Lecture 10 – The AVL Tree

Miao Zhang

Binary Search Tree – Best Case

- All BST operations are O(d), where d is tree depth
- minimum d is $d \quad \log_2 N$ for a binary tree with N nodes

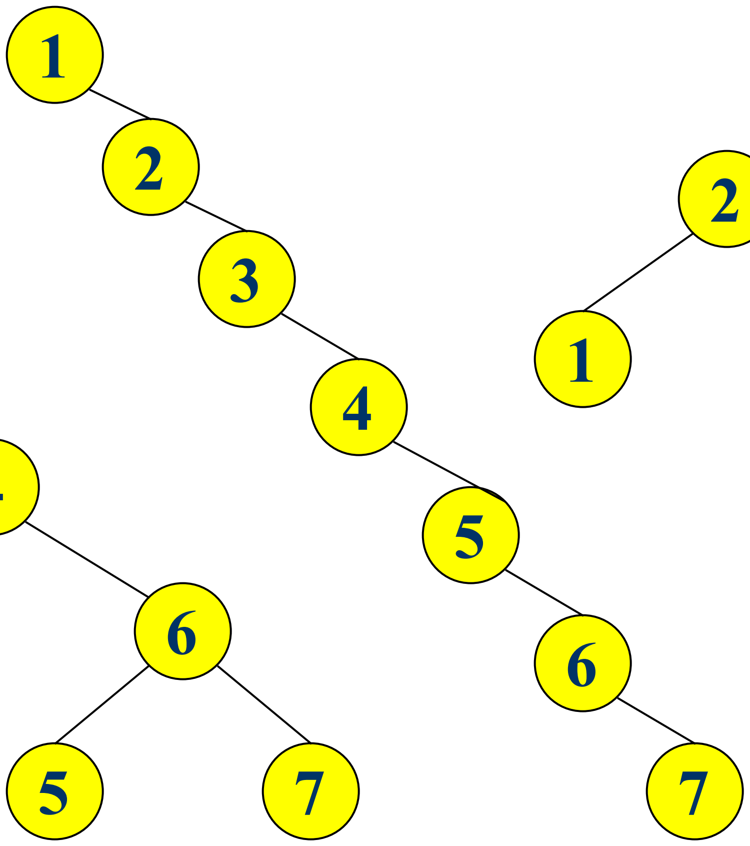
  - What is the best case tree?
  - What is the worst case tree?

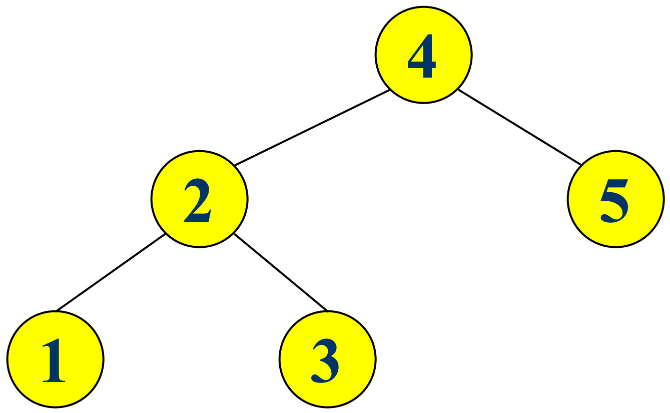
- So, the best case running time of BST operations is O(log N)

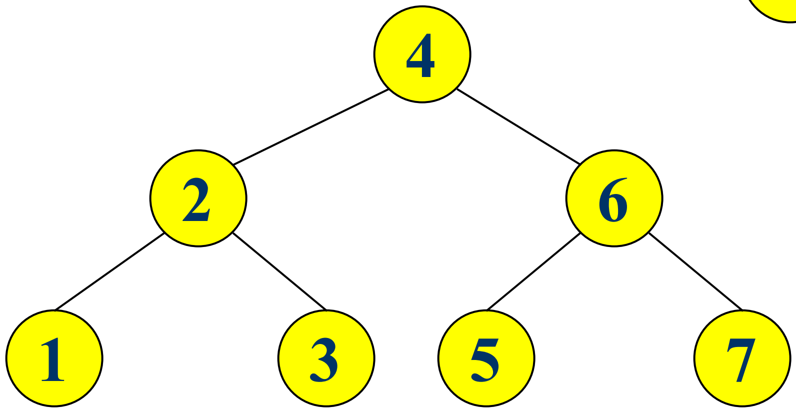
Binary Search Tree – Worst Case

- **Worst case running time is O(N)**
  - **What happens when you Insert elements in ascending order?**
    - **Insert: 2, 4, 6, 8, 10, 12 into an empty BST**
  - **Problem: Lack of "balance":**
    - **compare depths of left and right subtrees**
  - **Unbalanced degenerate tree**

Balanced and unbalanced BST







Is this "balanced"?

Approaches to balancing trees

- Don't balance
  - May end up with some nodes very deep
- Strict balance
  - The tree must always be balanced perfectly
- Pretty good balance
  - Only allow a little out of balance
- Adjust on access
  - Self-adjusting

Balancing Binary Search Trees

- Many algorithms exist for keeping binary search trees balanced

  - Adelson-Velskii and Landis (AVL) trees (height-balanced trees)

  - Splay trees and other self-adjusting trees

  - B-trees and other multiway search trees

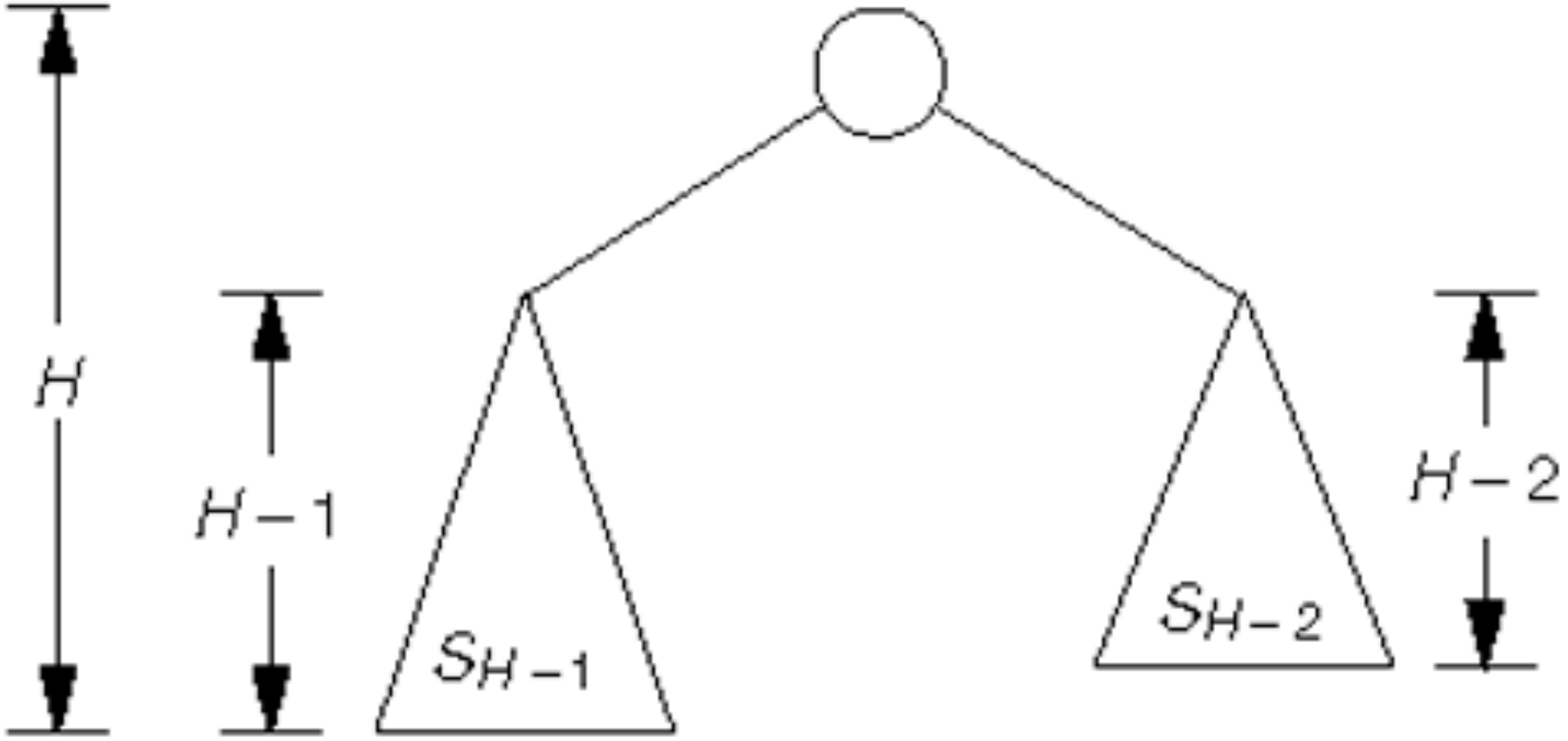
AVL Trees

- An AVL tree is a binary search tree with a *balance* condition.
- AVL tree *approximates* the ideal tree (completely balanced tree).
- AVL Tree maintains a height close to the minimum.

Definition:

An AVL tree is a binary search tree such that for any node in the tree, the height of the left and right subtrees can differ by at most 1.

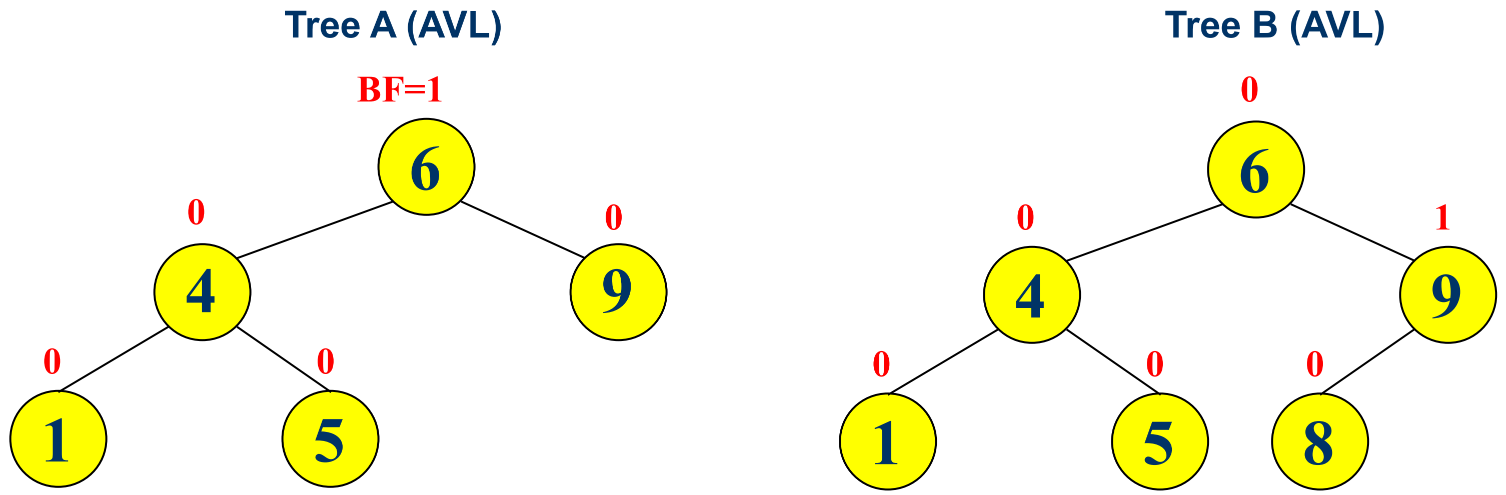
AVL Trees



AVL - Good but not Perfect Balance

- AVL trees are height-balanced binary search trees
- Balance factor of a node
  - height(left subtree) - height(right subtree)
- An AVL tree has balance factor calculated at every node
  - For every node, heights of left and right subtree can differ by no more than 1
  - Store current heights in each node

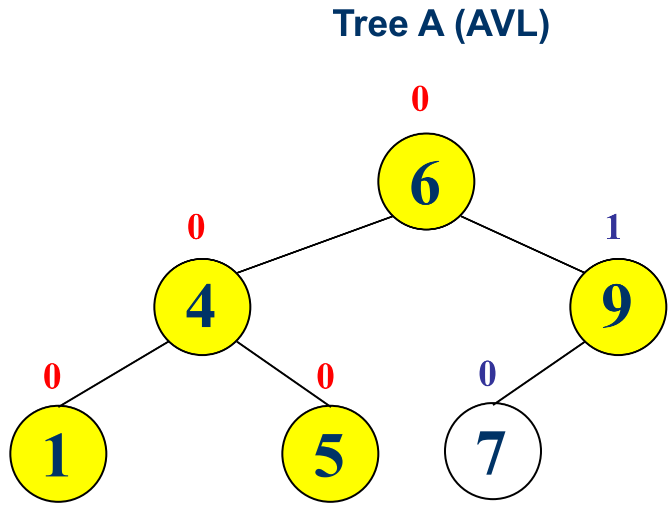
Node Heights

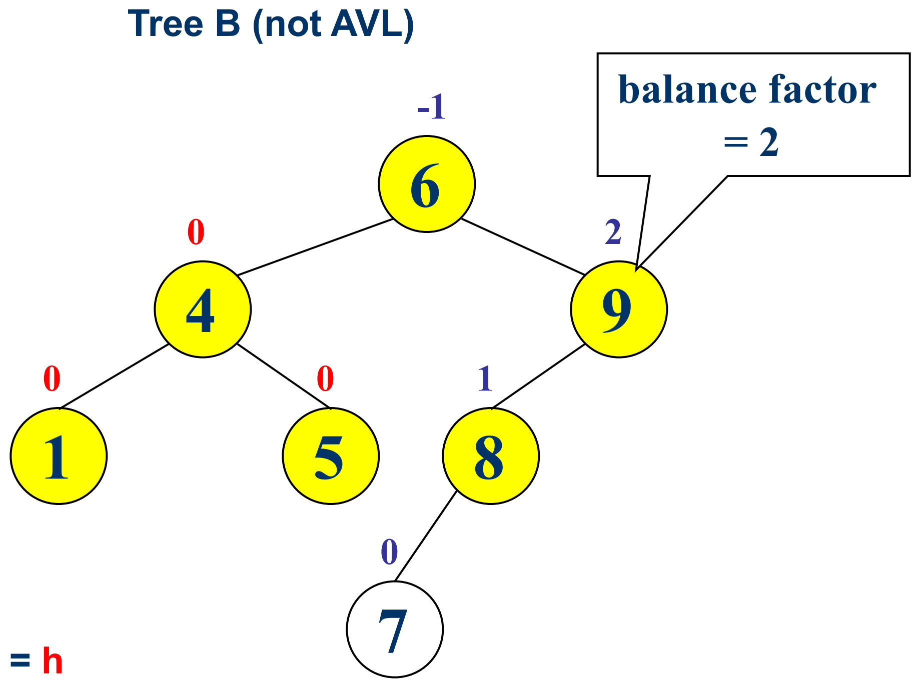


height of node = h

balance factor = h_{left}-h_{right}

Node Heights after Insert 7

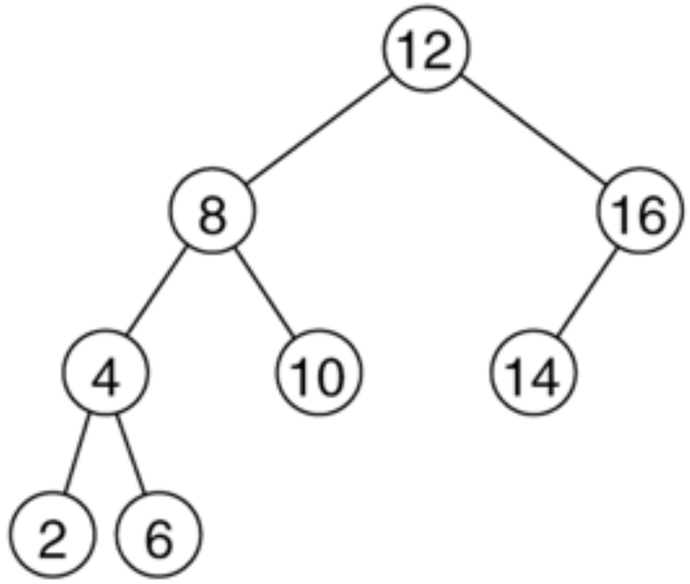




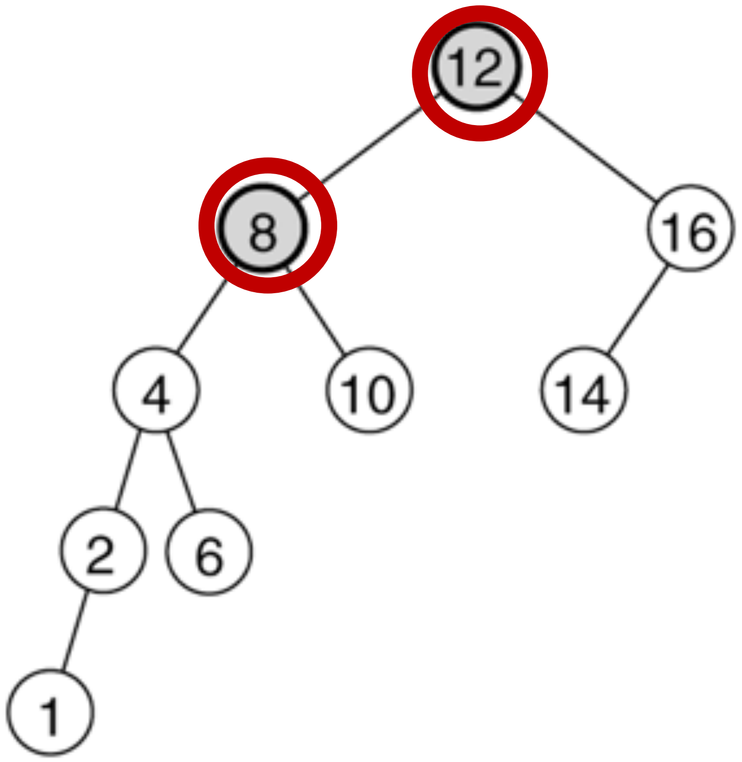
height of node = h
balance factor = $h_{left}-h_{right}$

Which One is an AVL Tree

(a) an AVL tree; (b) not an AVL tree (unbalanced nodes are darkened)



(a)



(b)

Properties

- The depth of a typical node in an AVL tree is very close to the optimal *log N*.
- Consequently, all searching operations in an AVL tree have logarithmic worst-case bounds.
- An update (insert or remove) in an AVL tree could destroy the balance. It must then be rebalanced before the operation can be considered complete.
- After an insertion, only nodes that are on the path from the insertion point to the root can have their balances altered.

Insert and Rotation in AVL Trees

- Insert operation may cause balance factor to become 2 or –2 for some node
  - only nodes on the path from insertion point to root node have possibly changed in height
  - So after the Insert, go back up to the root node by node, updating heights
  - If a new balance factor (the difference h_{left}-h_{right}) is 2 or –2, adjust tree by *rotation* around the node

Insertions in AVL Trees

Let the node that needs rebalancing be α.

There are 4 cases:

- **Outside Cases** (require single rotation) :
  1. Insertion into left subtree of left child of α.
  2. Insertion into right subtree of right child of α.

- **Inside Cases** (require double rotation) :
  3. Insertion into right subtree of left child of α.
  4. Insertion into left subtree of right child of α.

The rebalancing is performed through four separate rotation algorithms.

Balancing Operations: Rotations

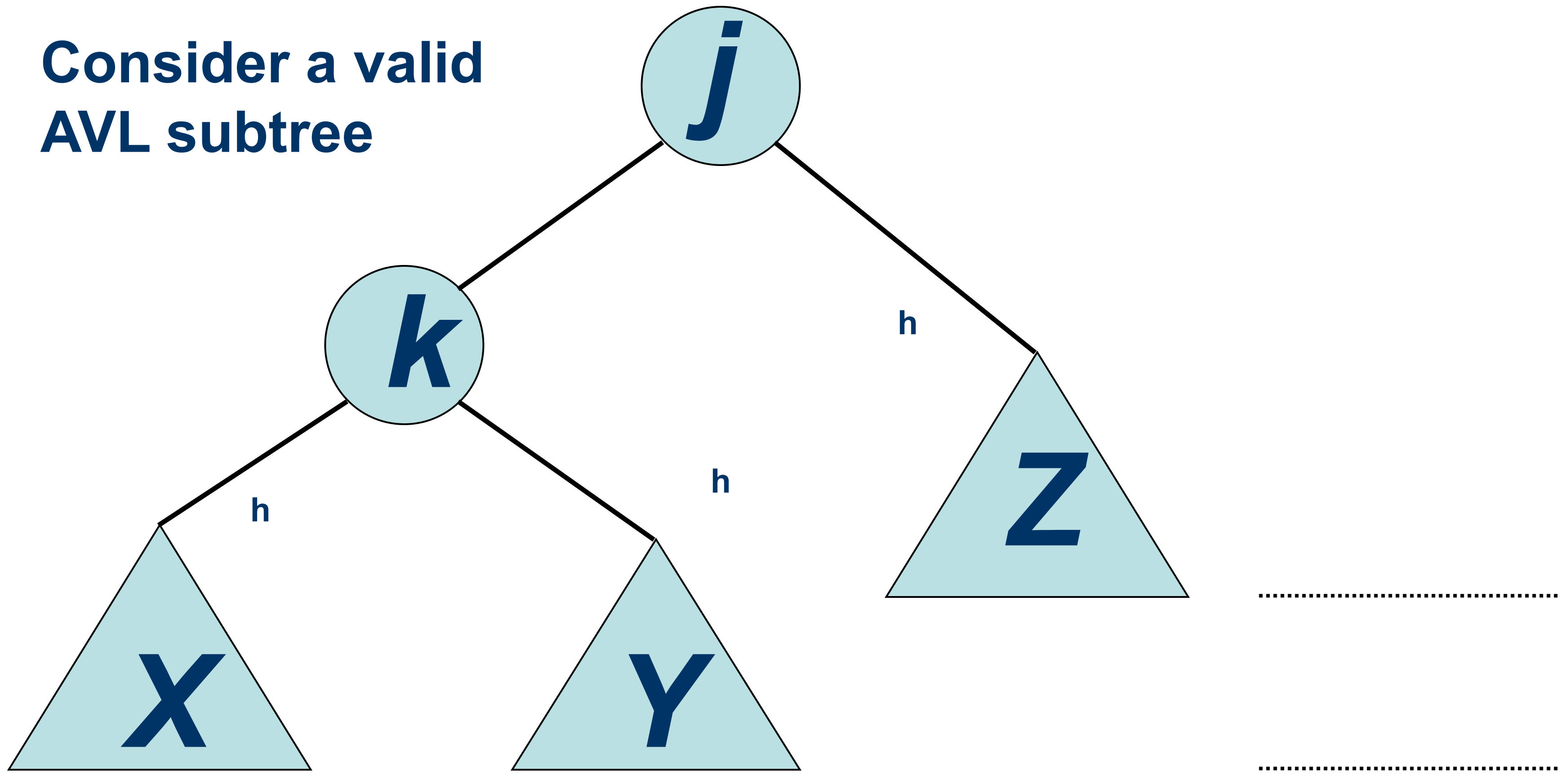
- Case 1 and case 2 are symmetric and require the same operation for balance.
  - Cases 1, 2 are handled by ***single rotation***.
- Case 3 and case 4 are symmetric and require the same operation for balance.
  - Cases 3, 4 are handled by ***double rotation***.

Single Rotation

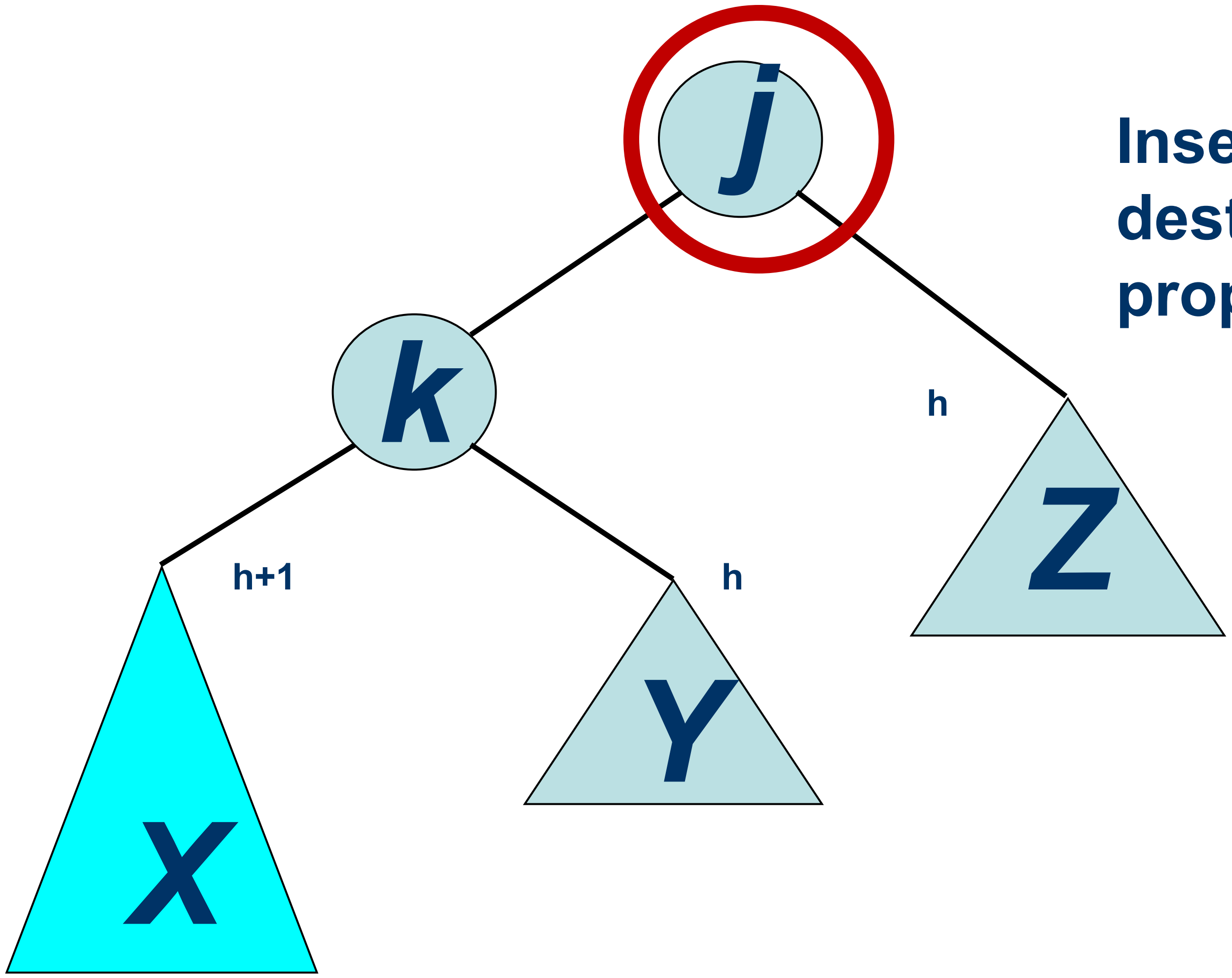
- A single rotation **switches the roles of the parent and child** while maintaining the search order.
- Single rotation handles the outside cases (i.e. 1 and 2).
- We rotate between a node and its child.
  - Child becomes parent. Parent becomes right child in case 1, left child in case 2.
- The result is a binary search tree that satisfies the AVL property.

AVL Insertion: Outside Case

Consider a valid AVL subtree



AVL Insertion: Outside Case



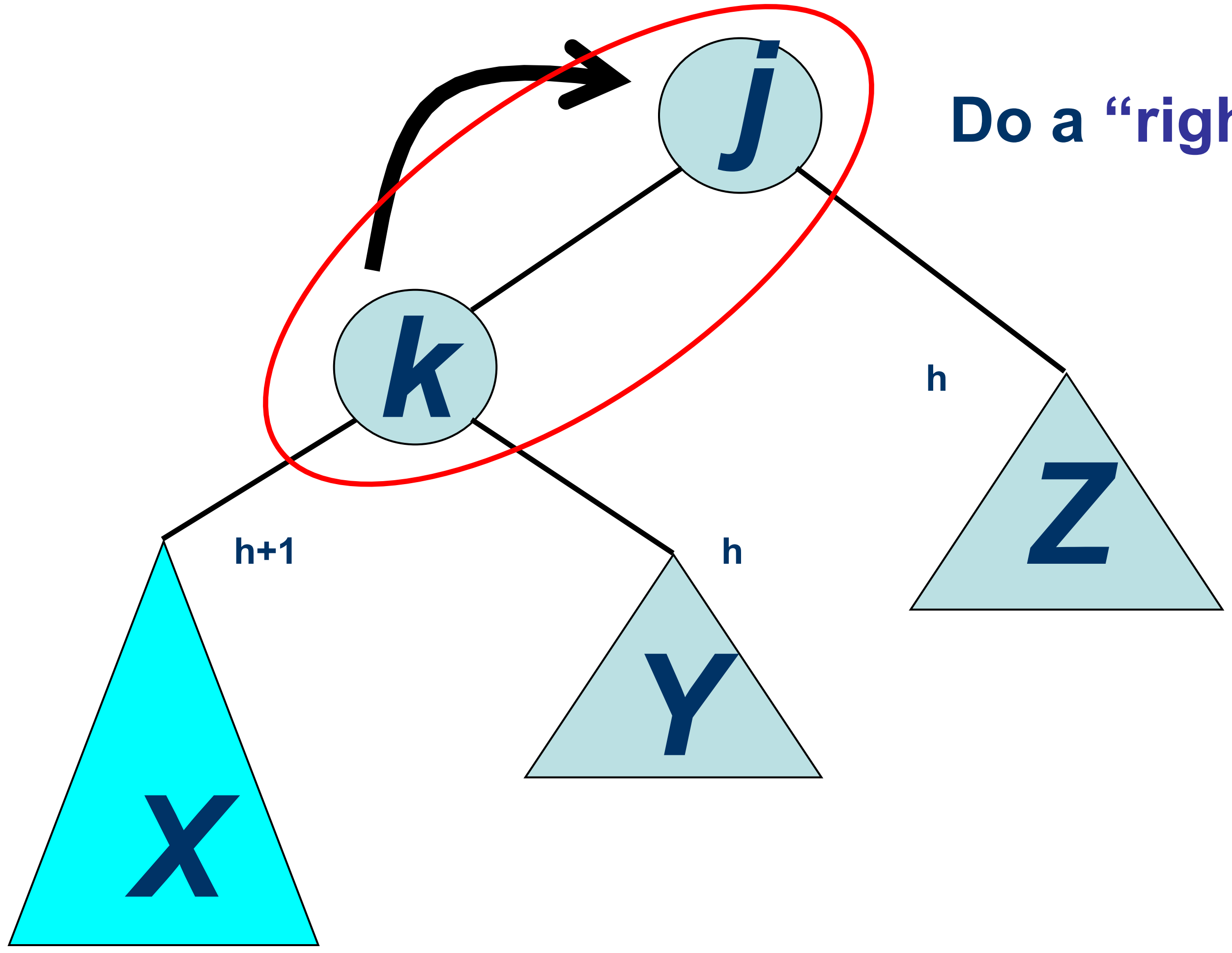
Inserting into X destroys the AVL property at node j

....

....

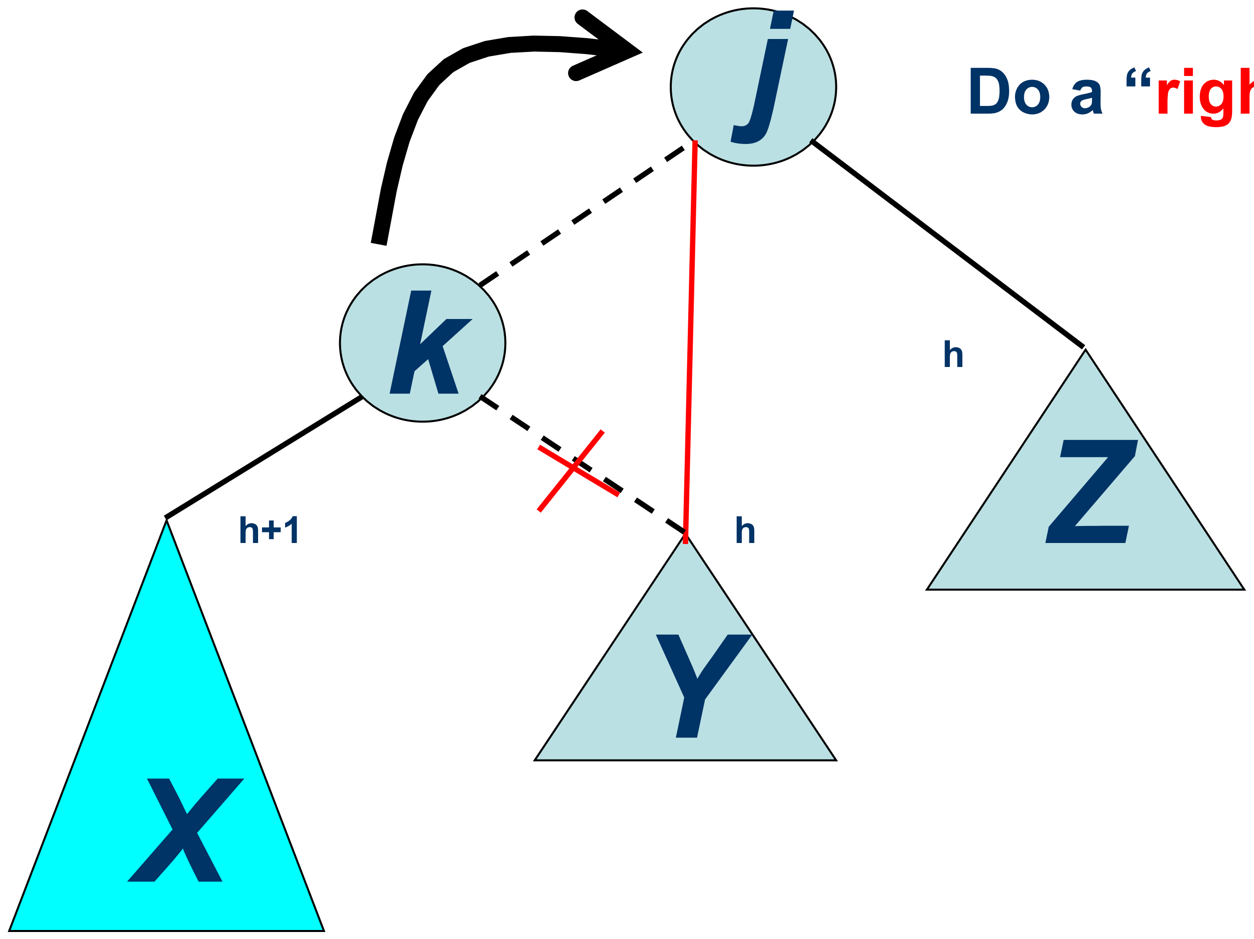
....

AVL Insertion: Outside Case



Do a "right rotation"

Single right rotation



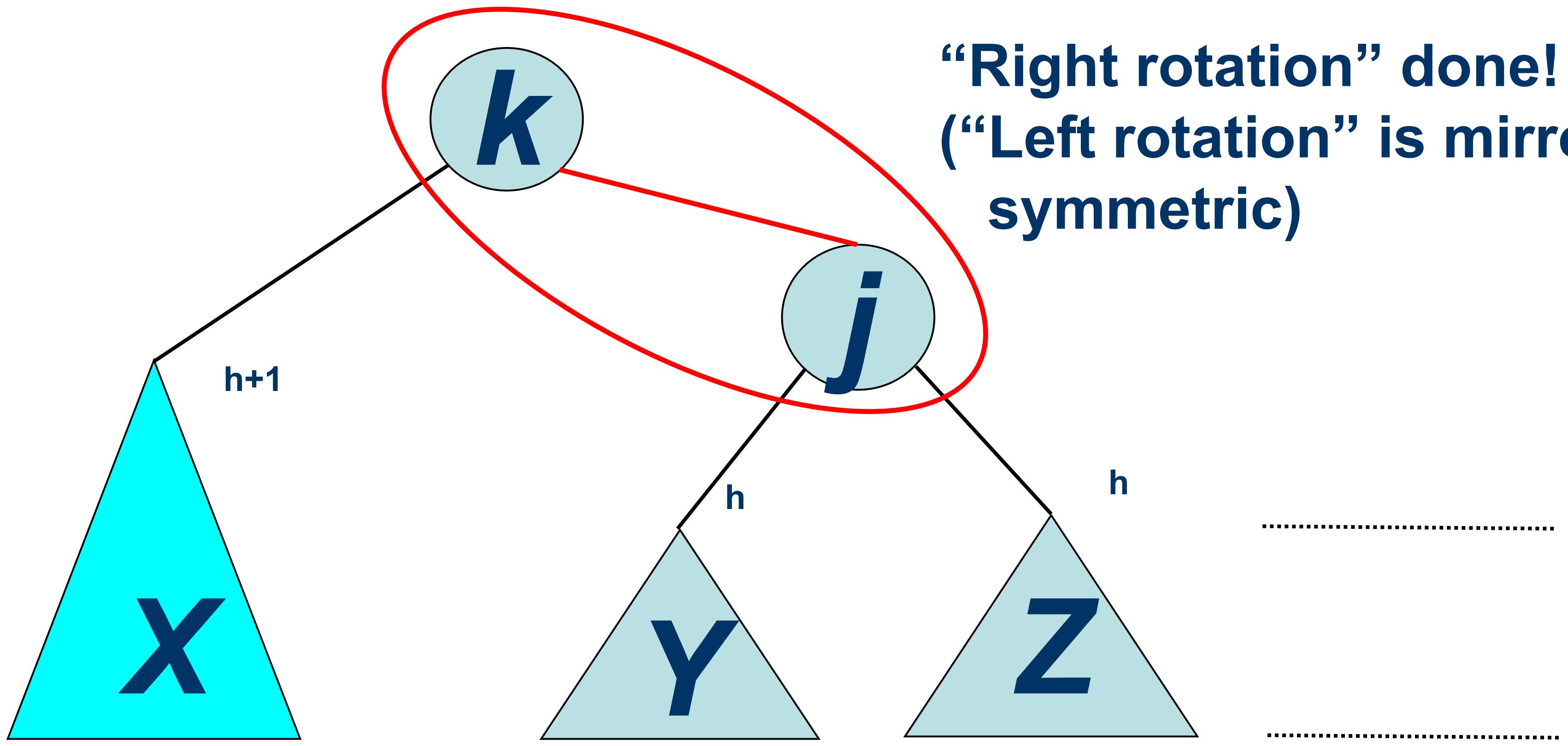
Do a "right rotation"

....

....

....

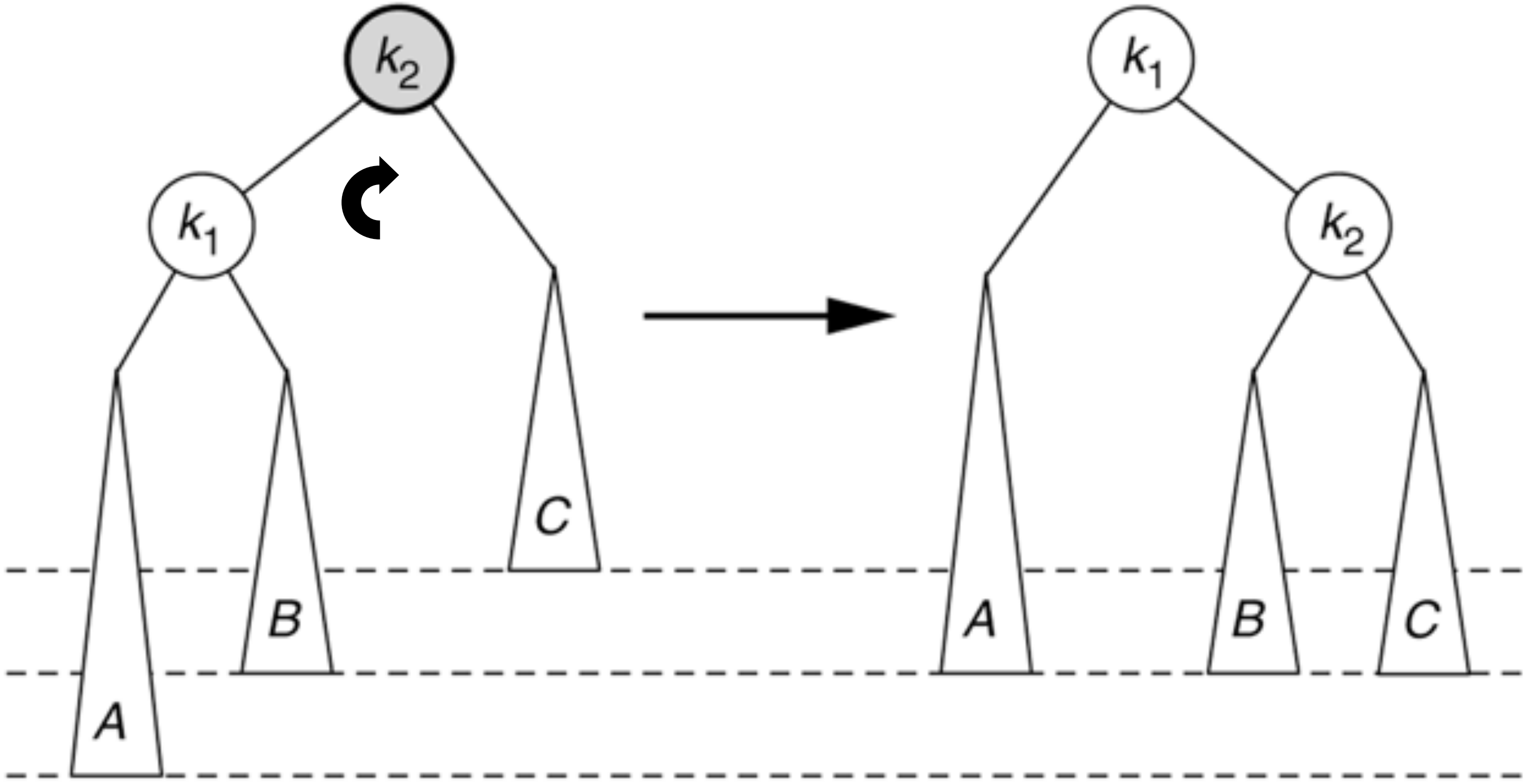
Outside Case Completed



“Right rotation” done!
(“Left rotation” is mirror symmetric)

AVL property has been restored!

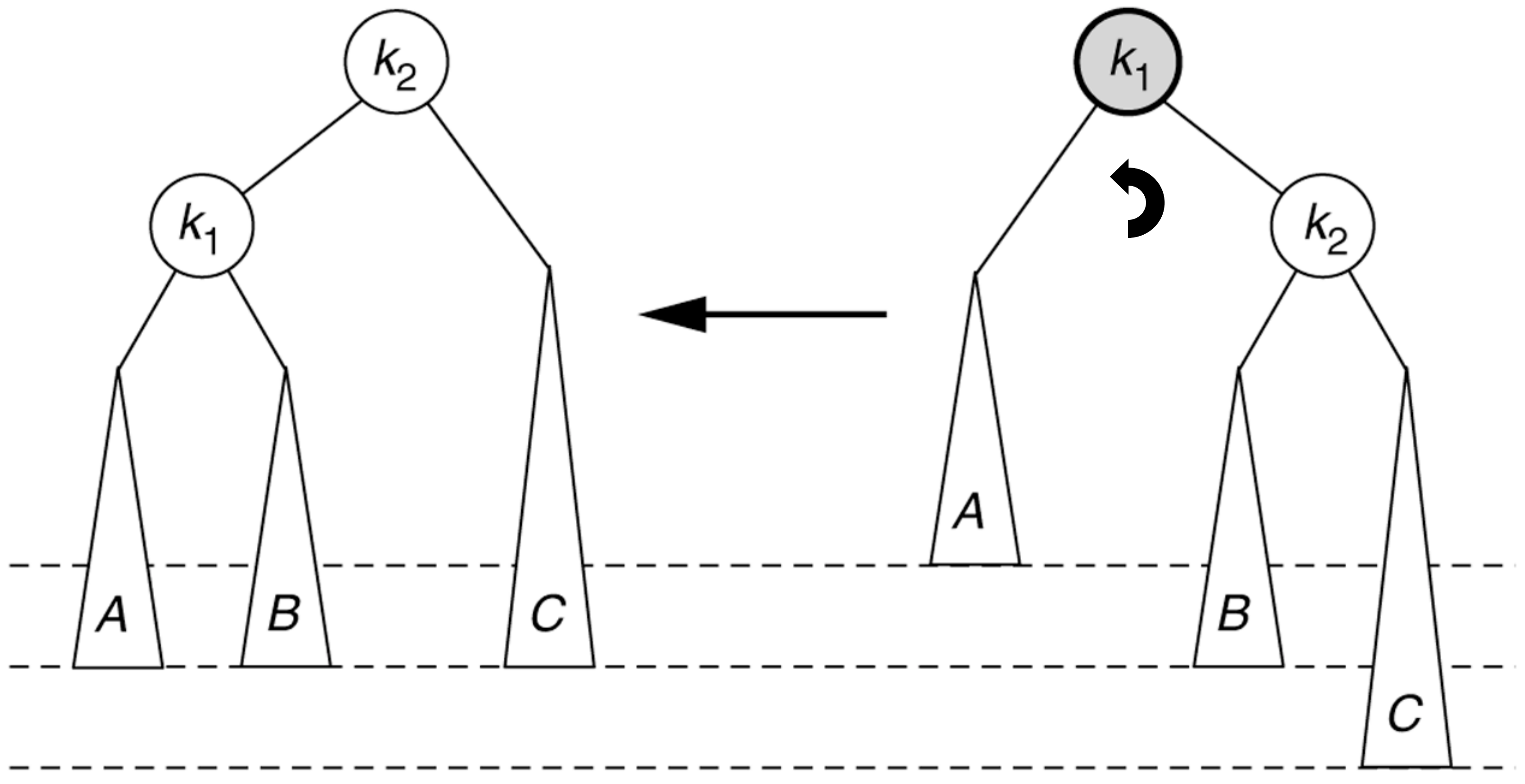
Single rotation to fix case 1: Rotate right



(a) Before rotation

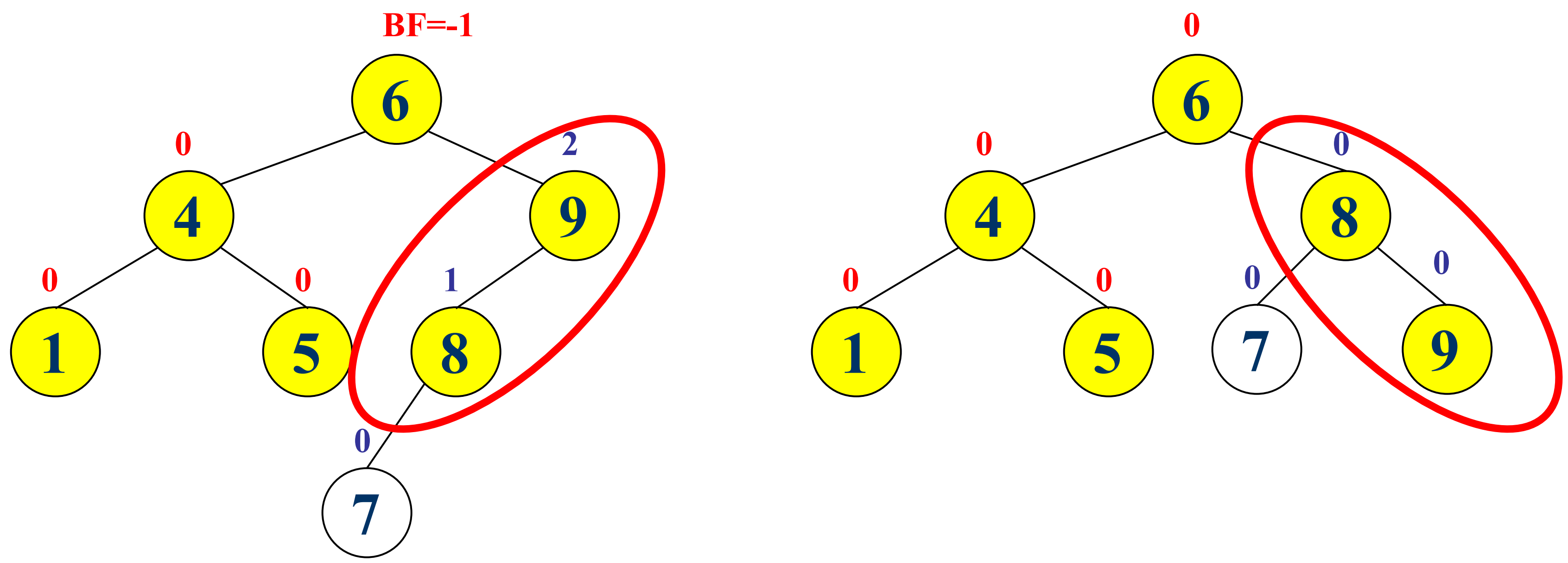
(b) After rotation

Single rotation to fix case 2: Rotate left

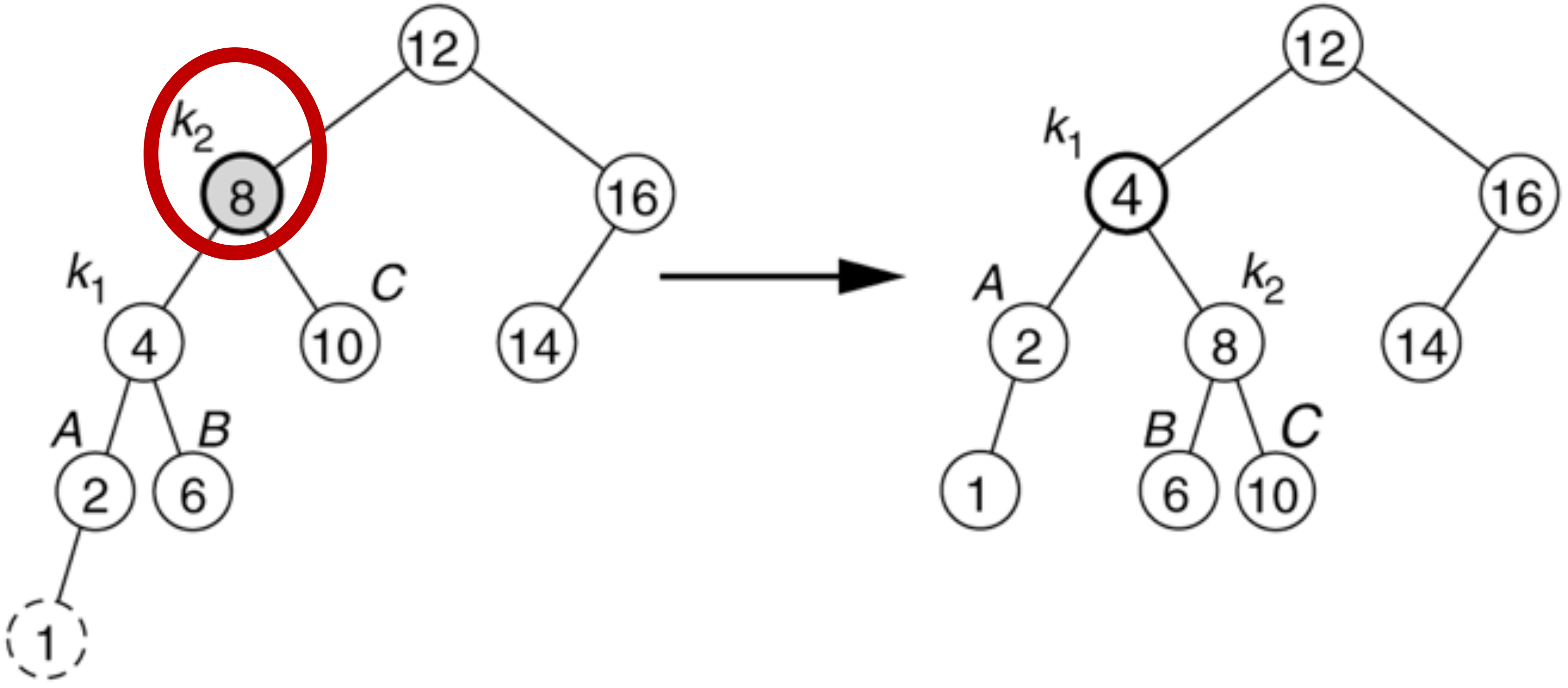


(a) After rotation (b) Before rotation

Single rotation fixes an AVL tree after insertion



Single rotation fixes an AVL tree after insertion of 1



(a) Before rotation

(b) After rotation

Analysis

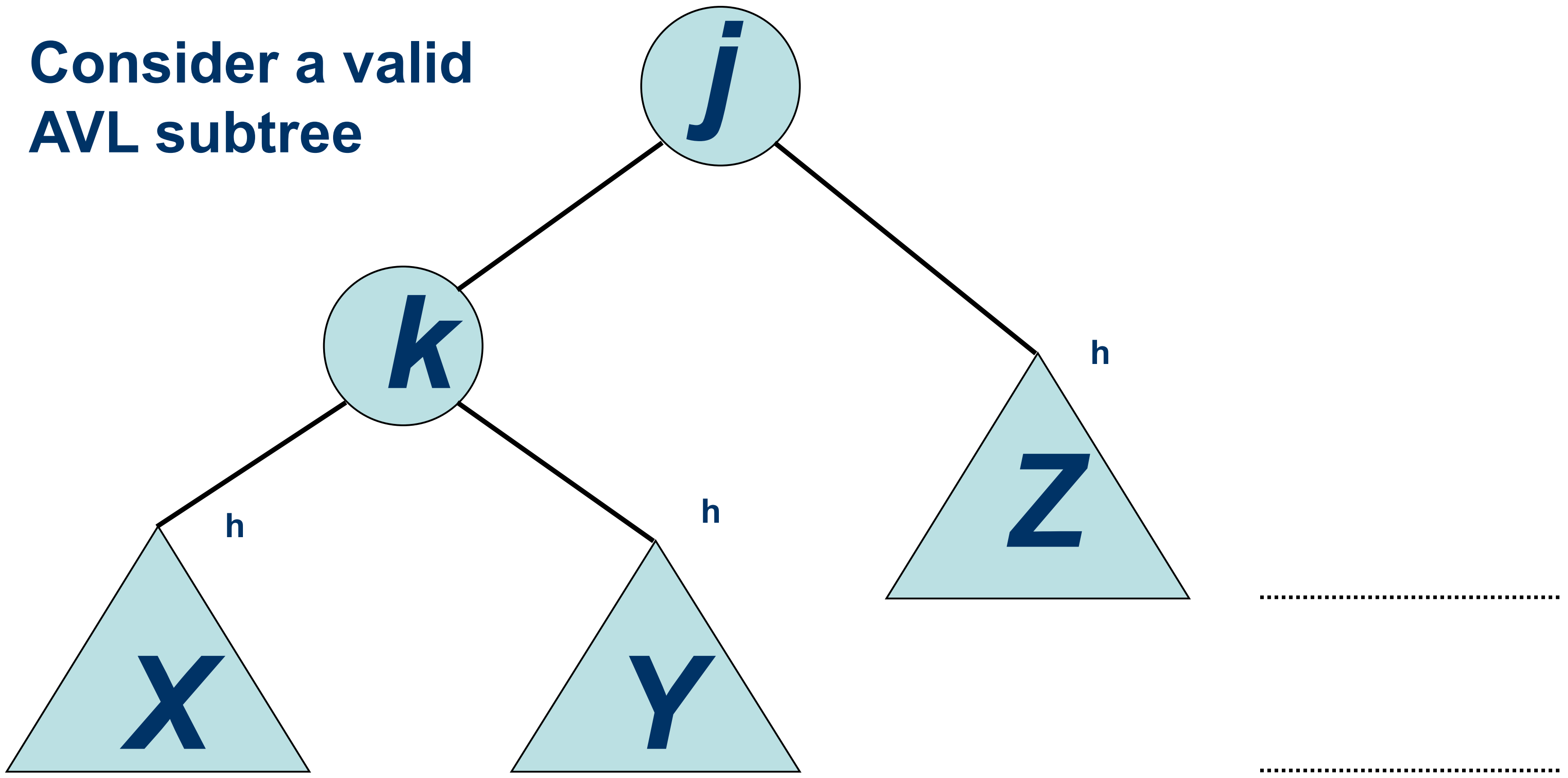
- One rotation suffices to fix cases 1 and 2.
- Single rotation preserves the original height:
  - The new height of the entire subtree is exactly the same as the height of the original subtree before the insertion.
- Therefore it is enough to do rotation only at the first node, where imbalance exists, on the path from inserted node to root.
- Thus the rotation takes O(1) time.
- Hence insertion is O(logN)

Double Rotation

- **Single rotation does not fix the inside cases (3 and 4).**
- **These cases require a *double* rotation, involving three nodes and four subtrees.**

AVL Insertion: Inside Case

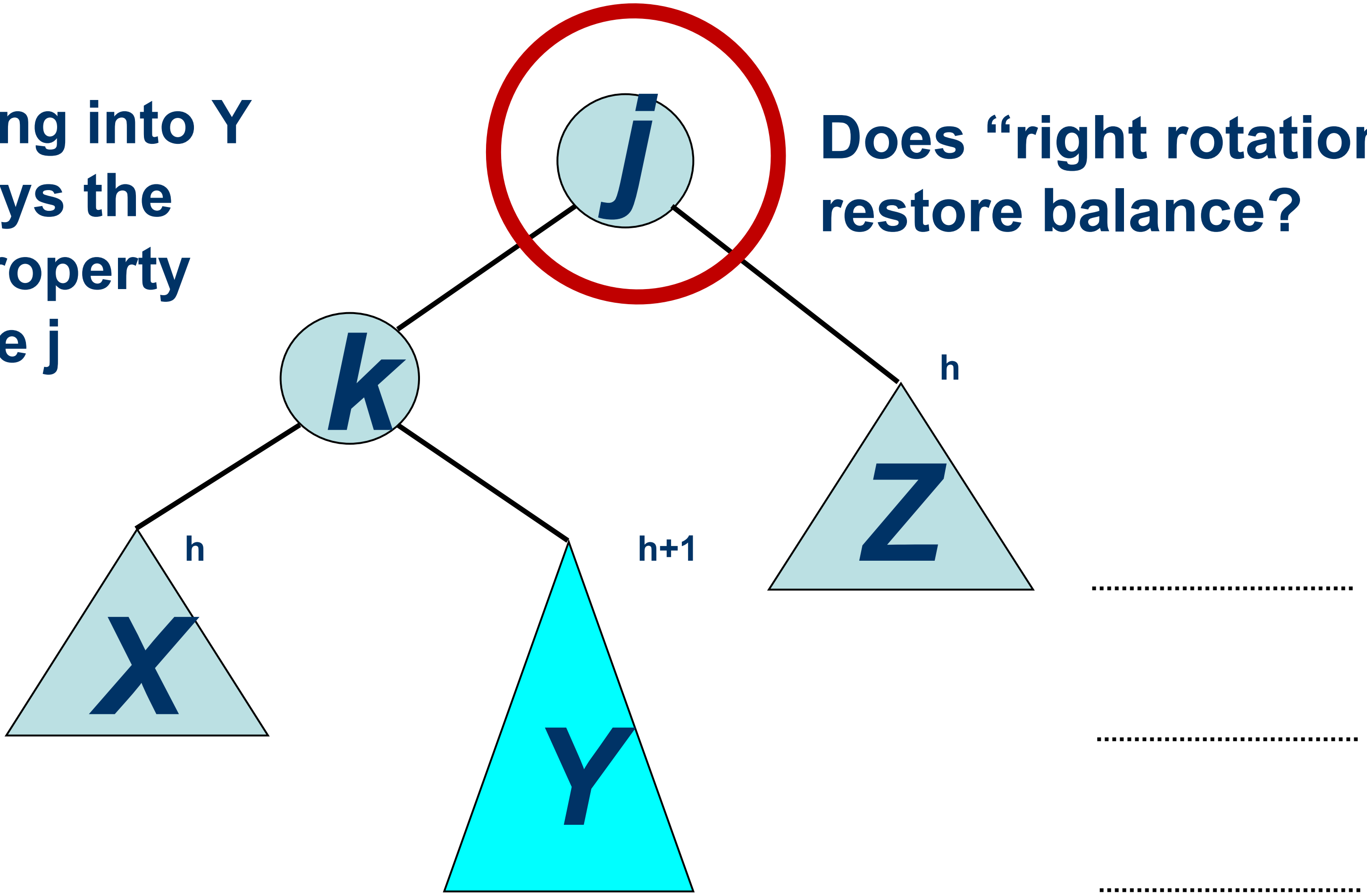
Consider a valid AVL subtree



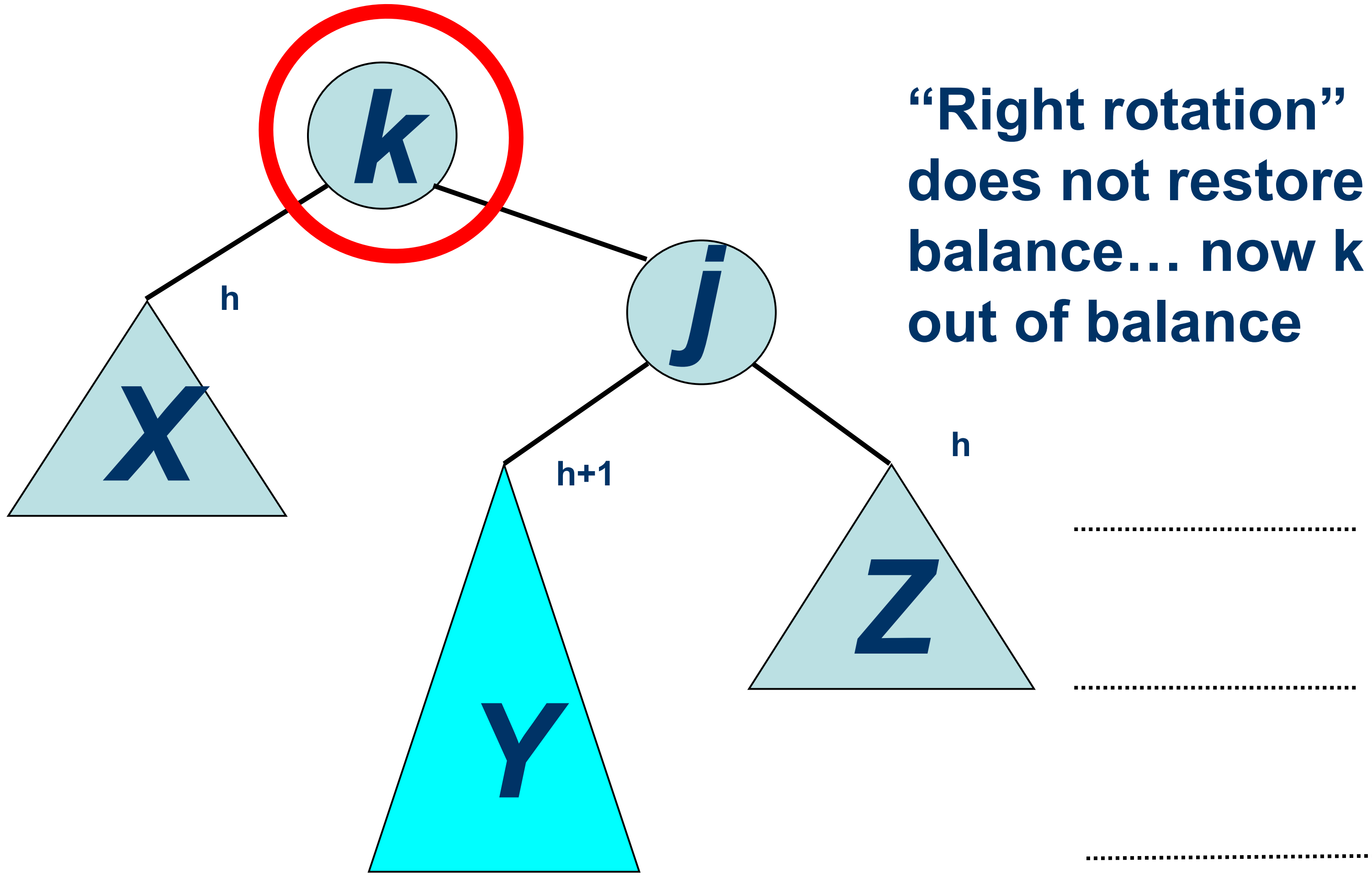
AVL Insertion: Inside Case

Inserting into Y destroys the AVL property at node j

Does "right rotation" restore balance?



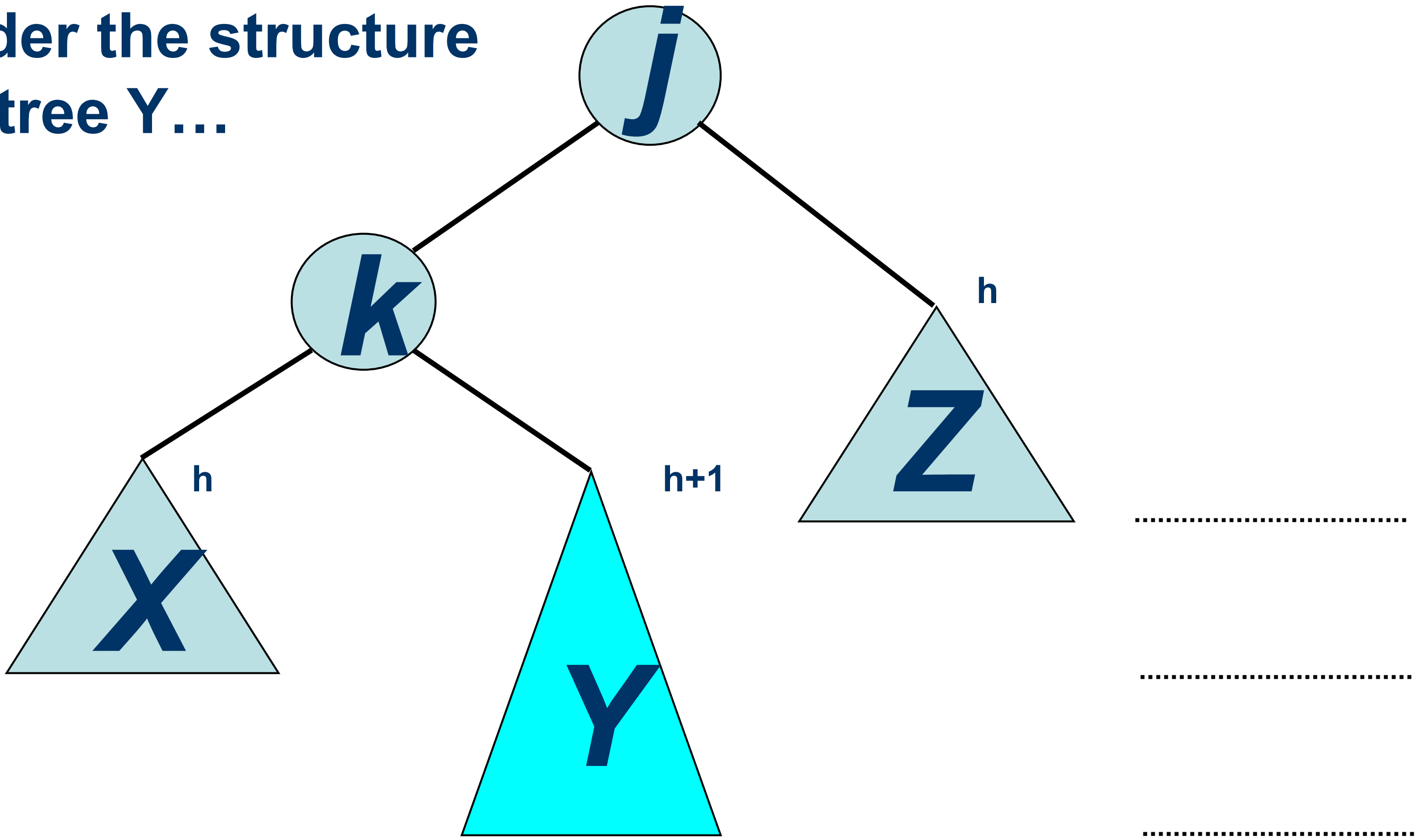
AVL Insertion: Inside Case



"Right rotation" does not restore balance… now k is out of balance

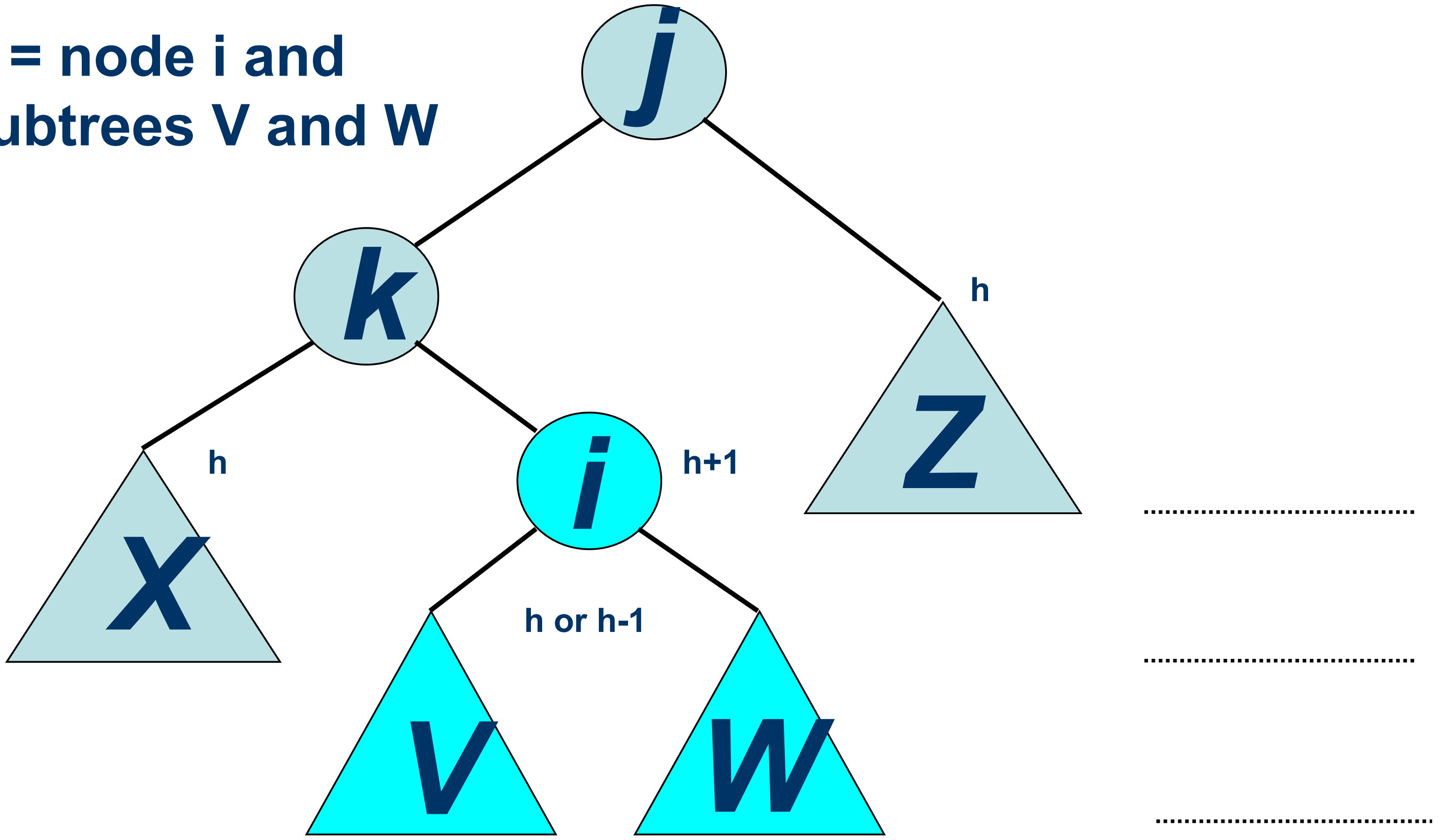
AVL Insertion: Inside Case

Consider the structure of subtree Y…

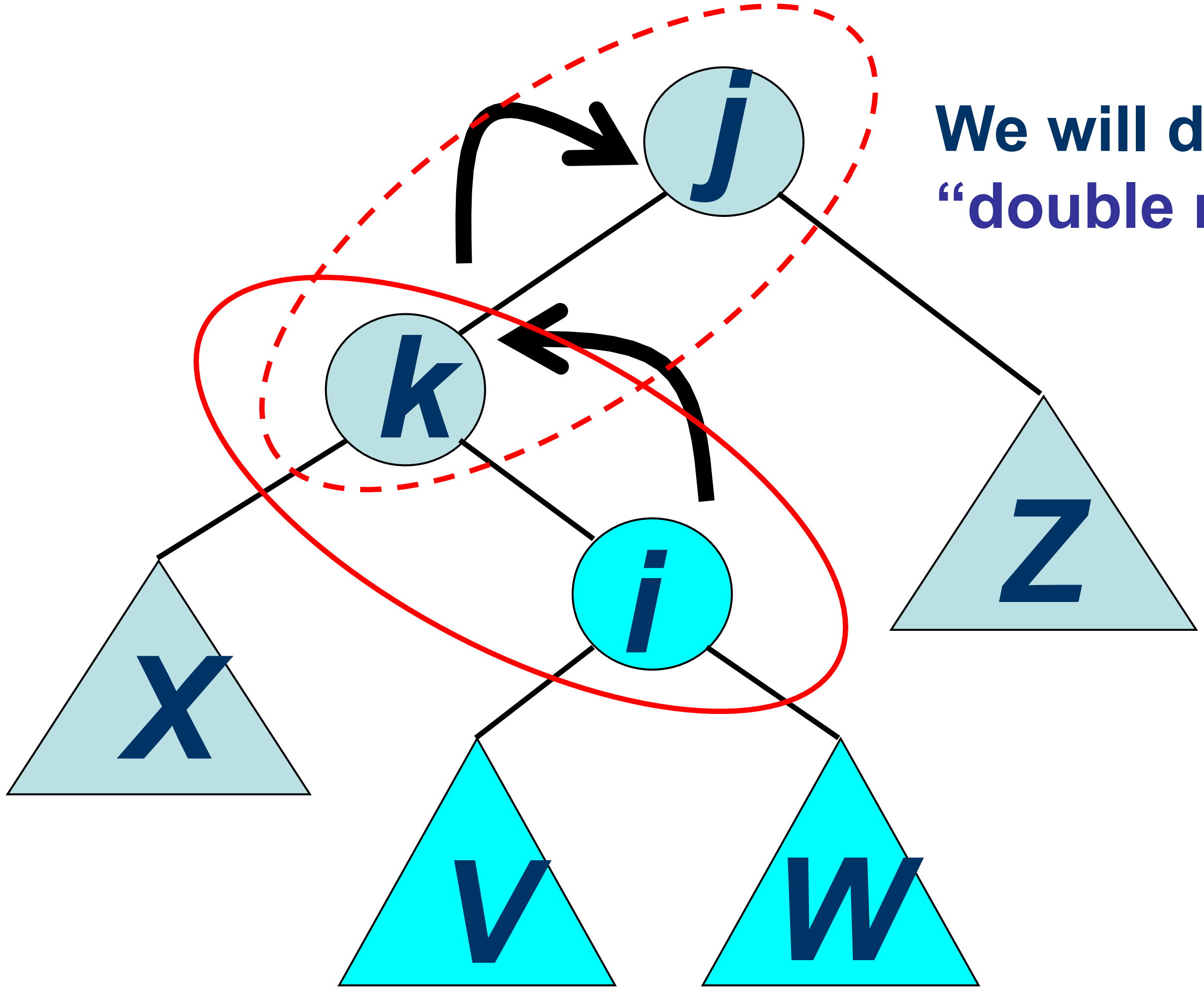


AVL Insertion: Inside Case

Y = node i and subtrees V and W

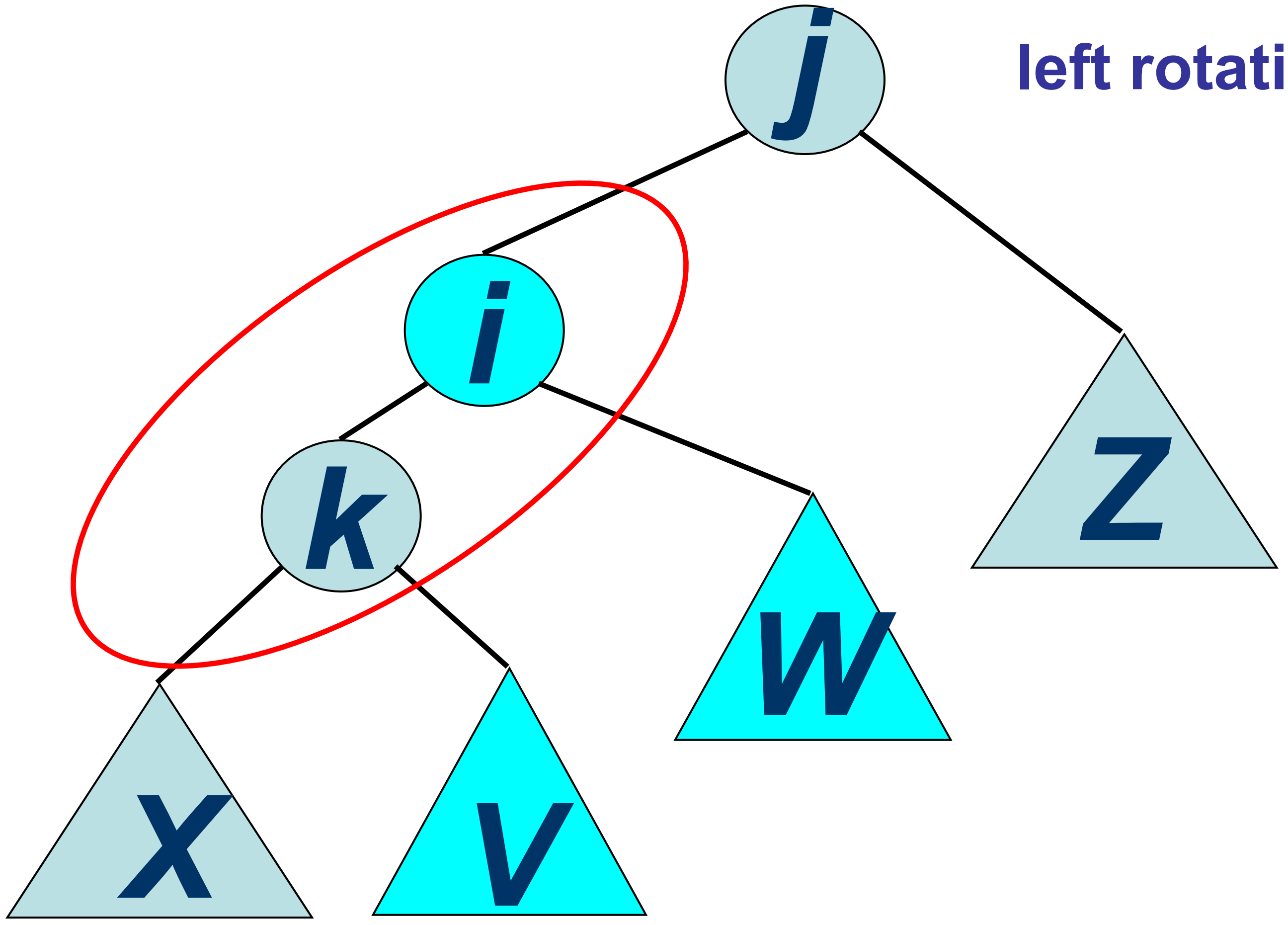


AVL Insertion: Inside Case



We will do a left-right "double rotation" . . .

Double rotation : first rotation



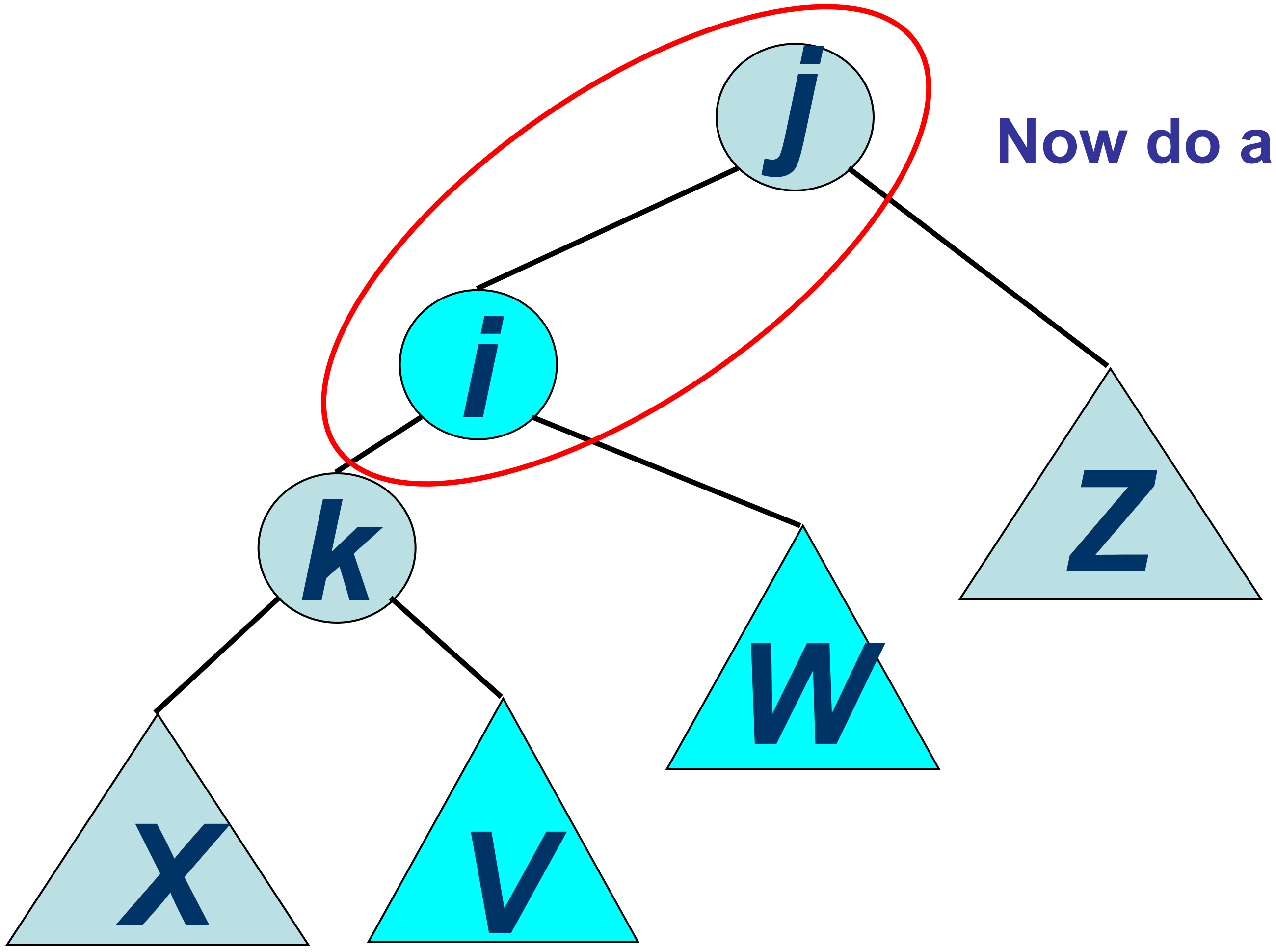
left rotation complete

....

....

....

Double rotation : second rotation

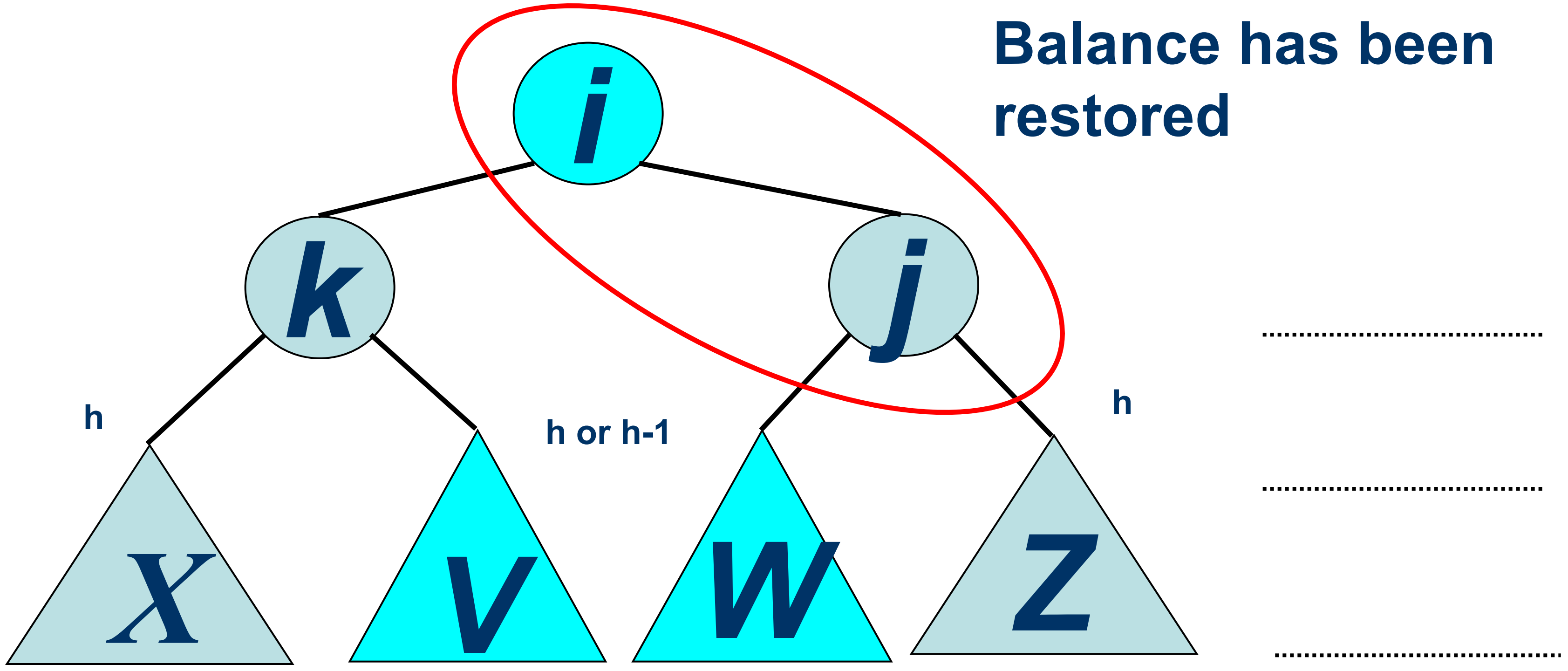


Now do a right rotation

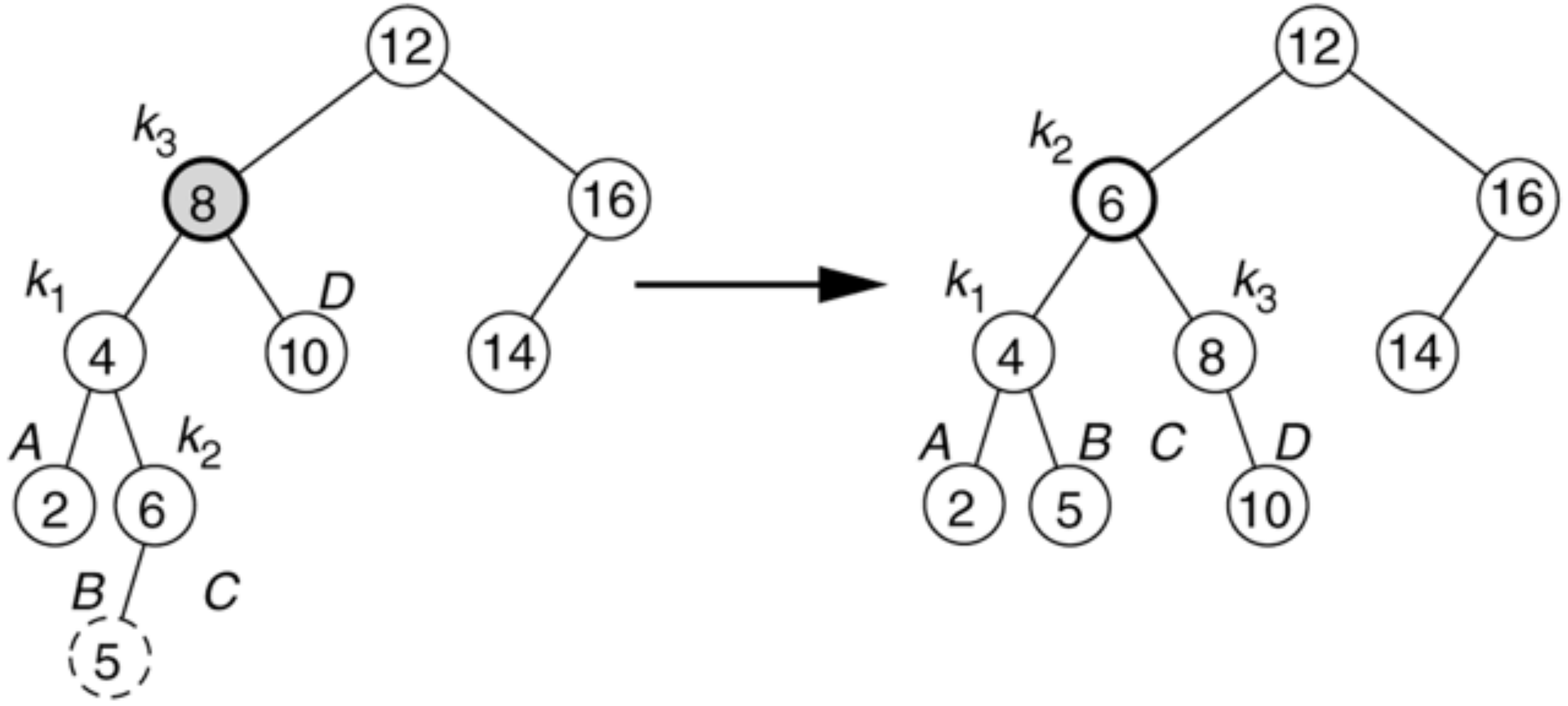
Double rotation : second rotation

right rotation complete

Balance has been restored

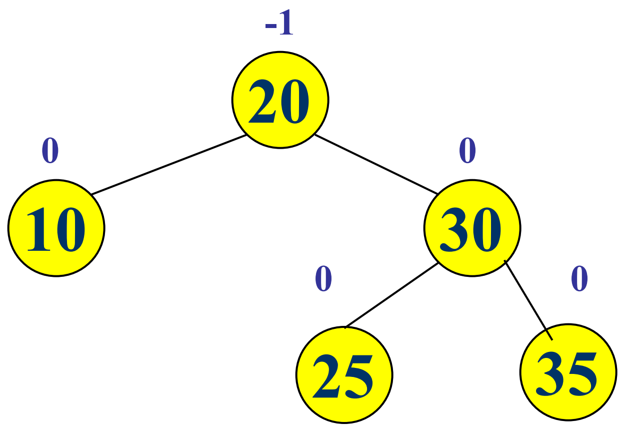


Double rotation fixes AVL tree after the insertion of 5



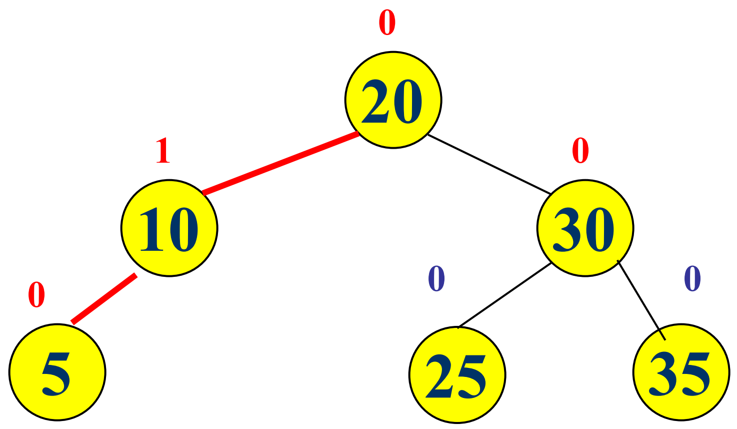
(a) Before rotation (b) After rotation

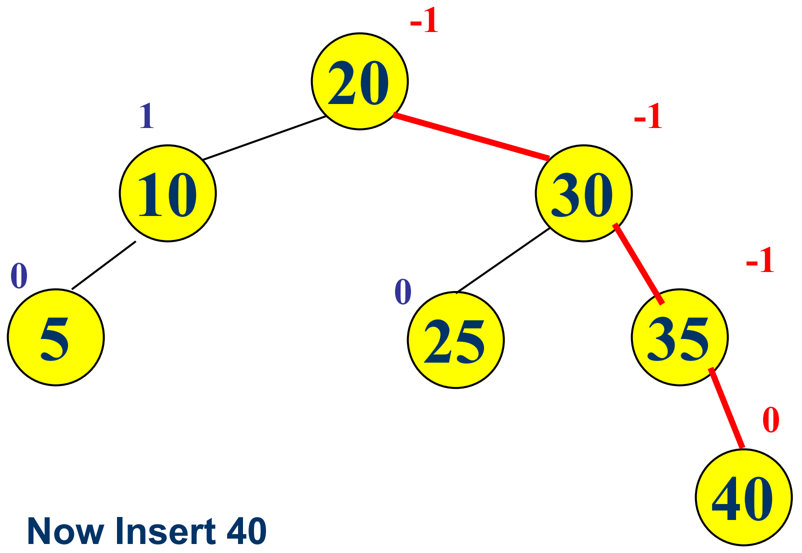
Example of Insertions in an AVL Tree



Insert 5, 40

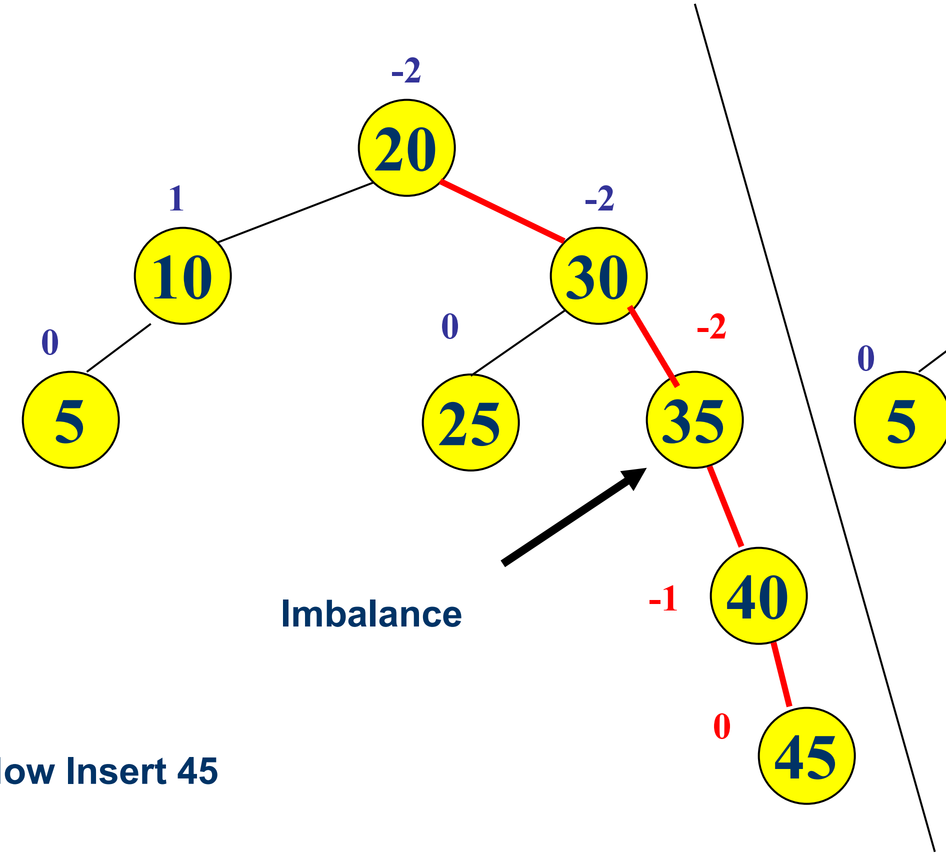
Example of Insertions in an AVL Tree



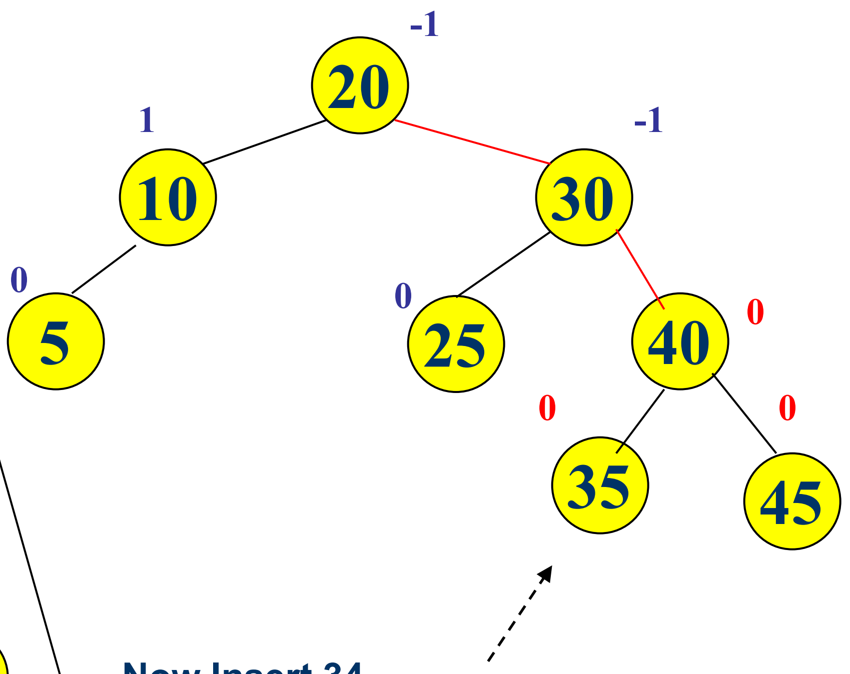


Now Insert 40

Single rotation (outside case)

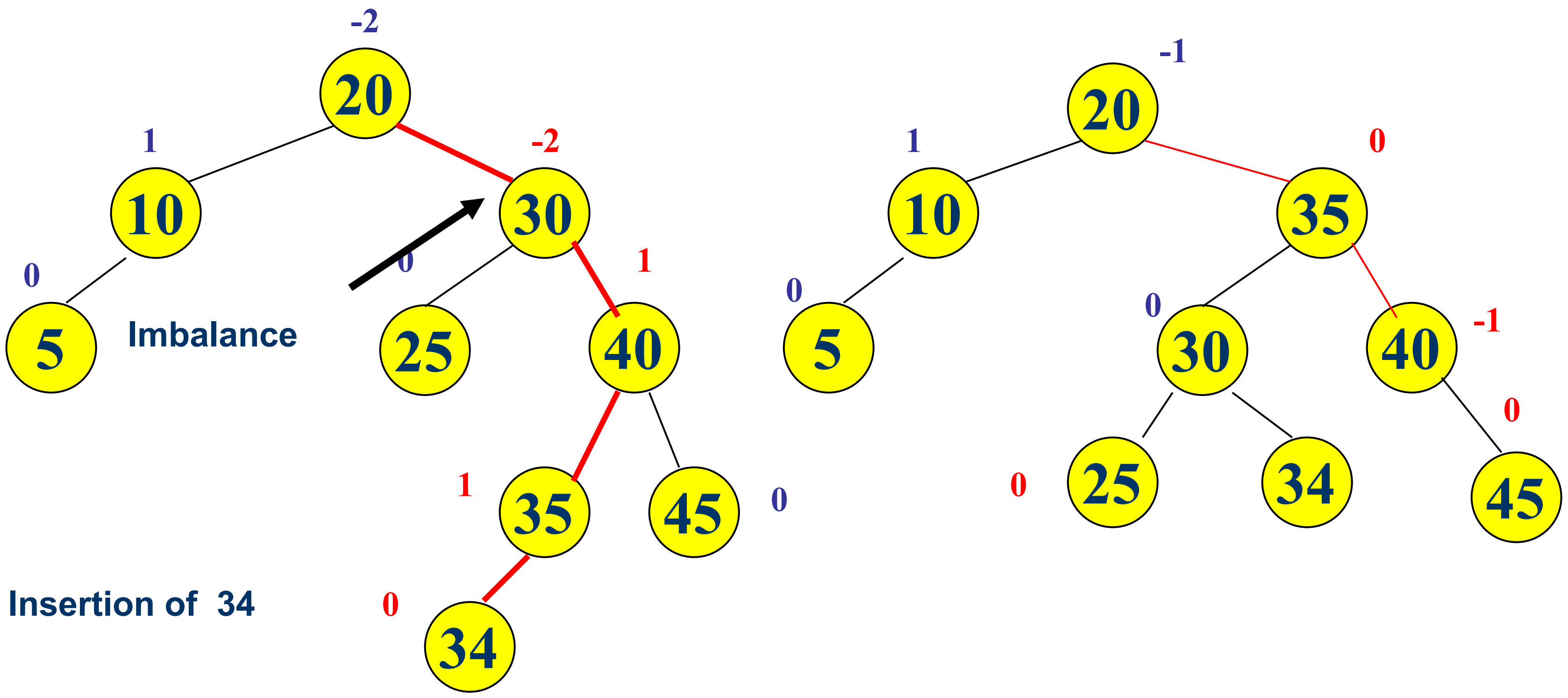


Now Insert 45

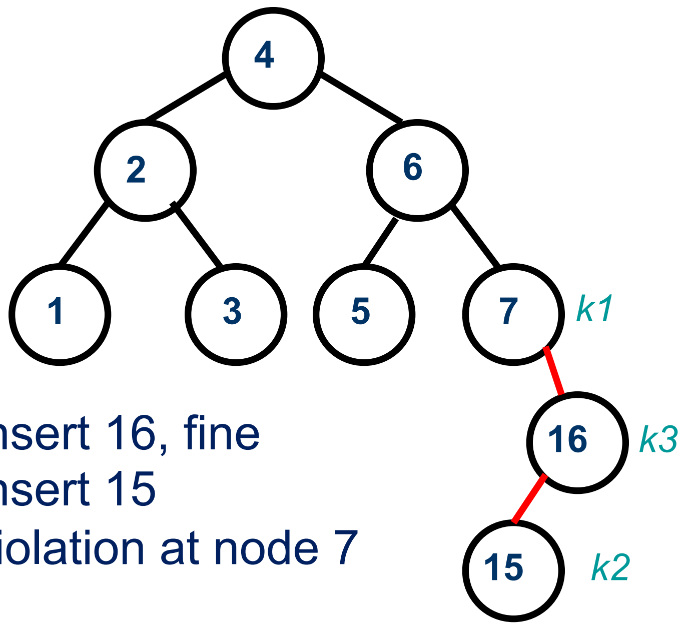


Now Insert 34

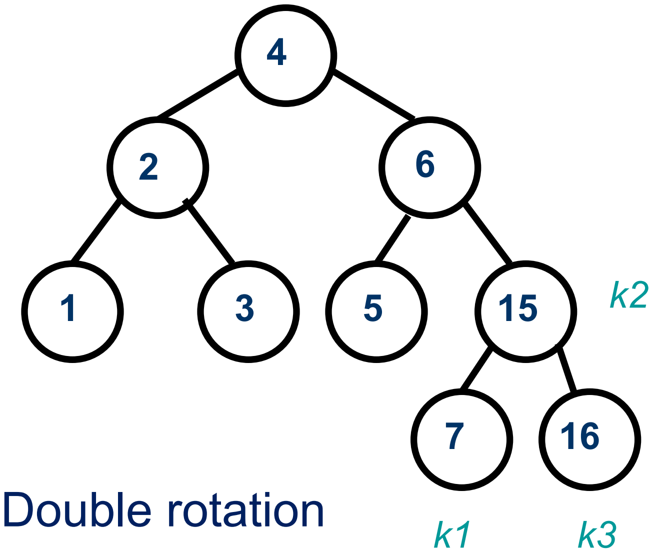
Double rotation (inside case)



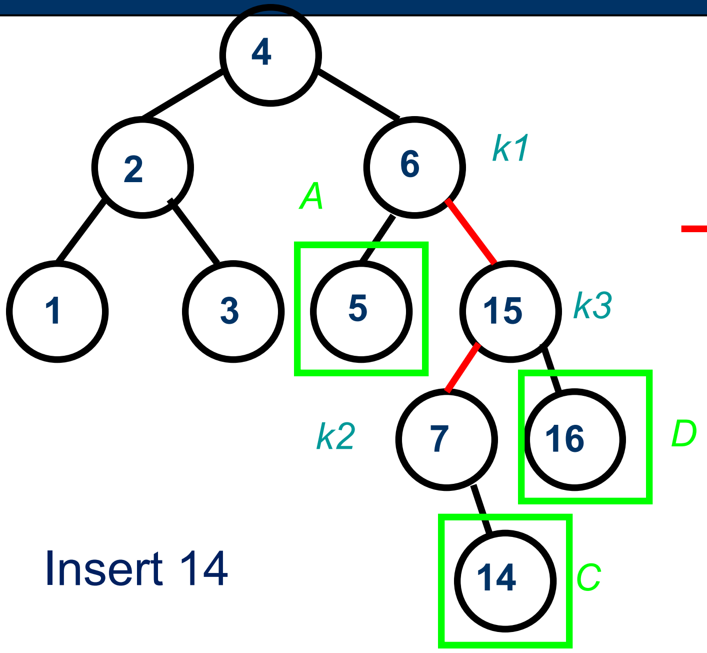
- Example
  Suppose that we've inserted 3, 2, 1, 4, 5, 6, 7, 16
  We'll insert 15, 14, 13, 12, 11, 10, 8, 9



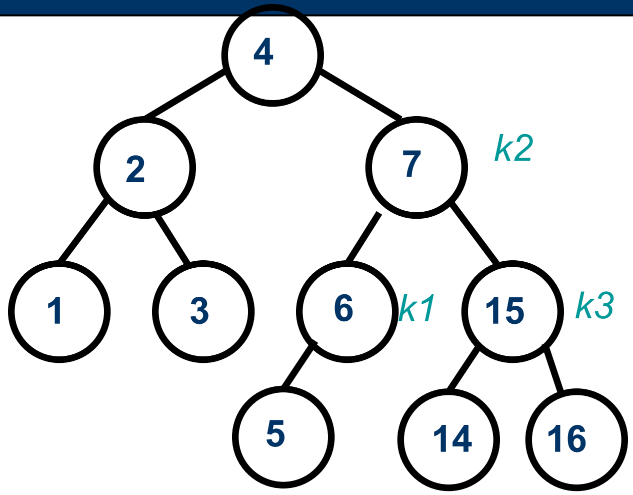
Insert 16, fine
Insert 15
violation at node 7



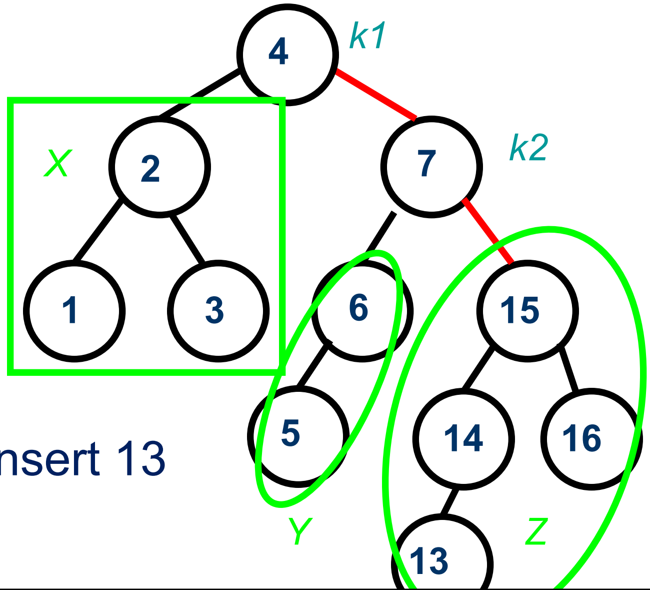
Double rotation



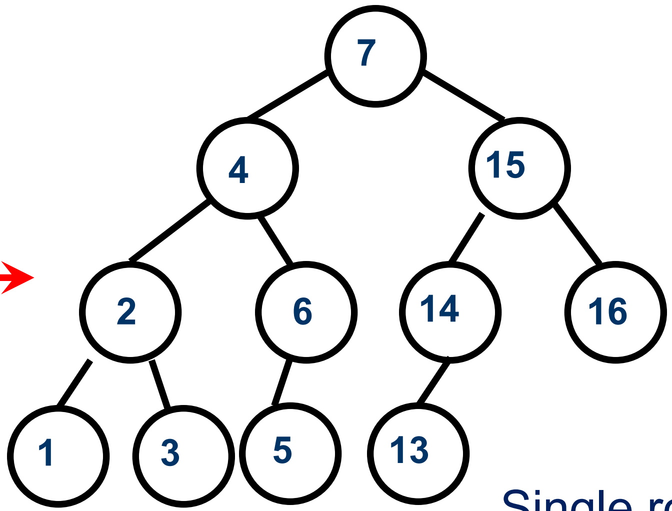
Insert 14



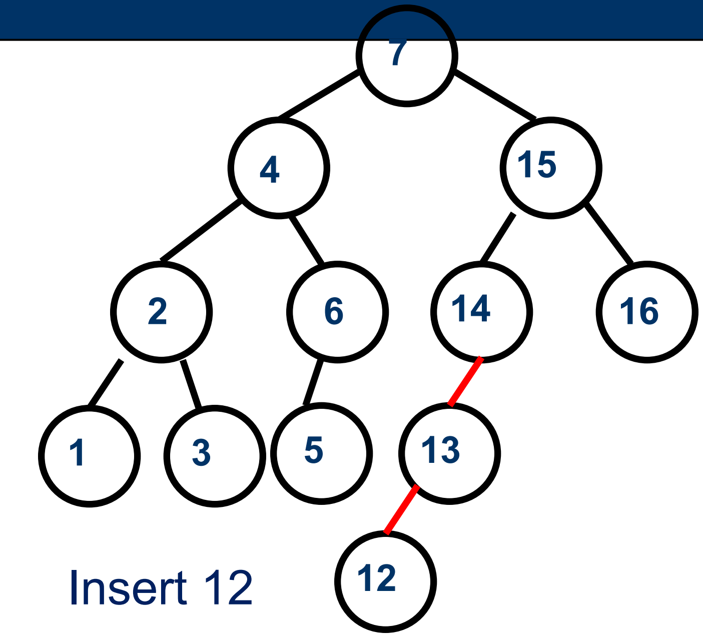
Double rotation



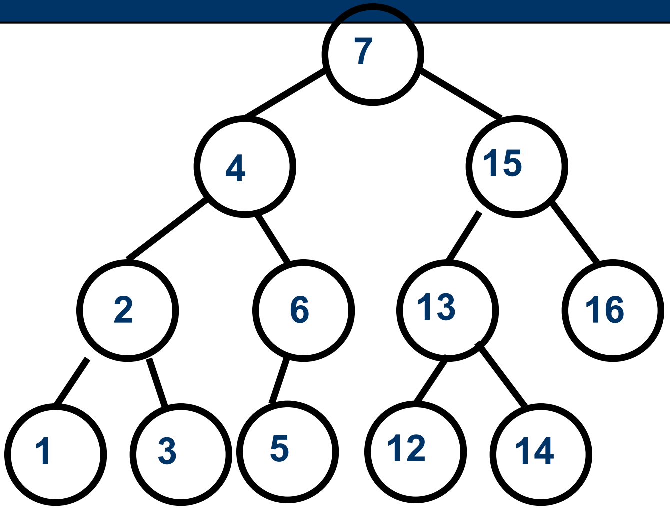
Insert 13



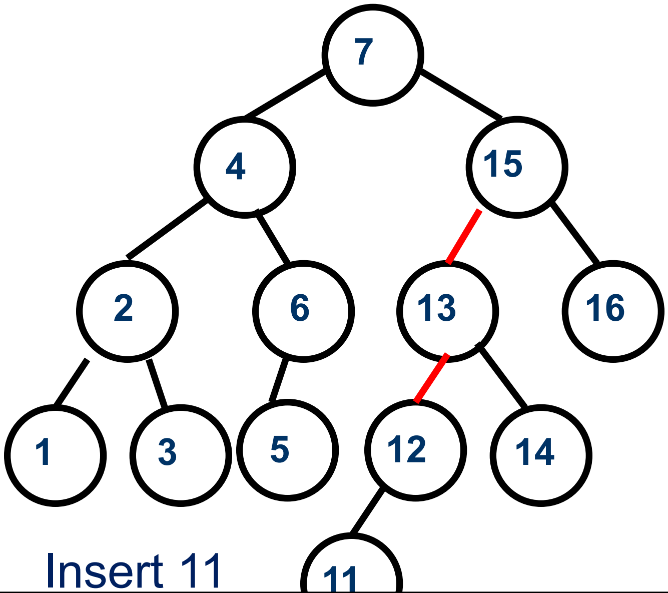
Single rotation



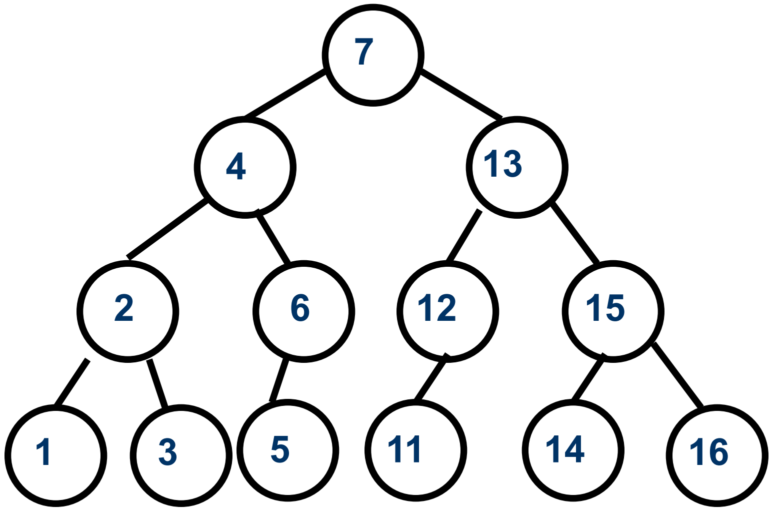
Insert 12



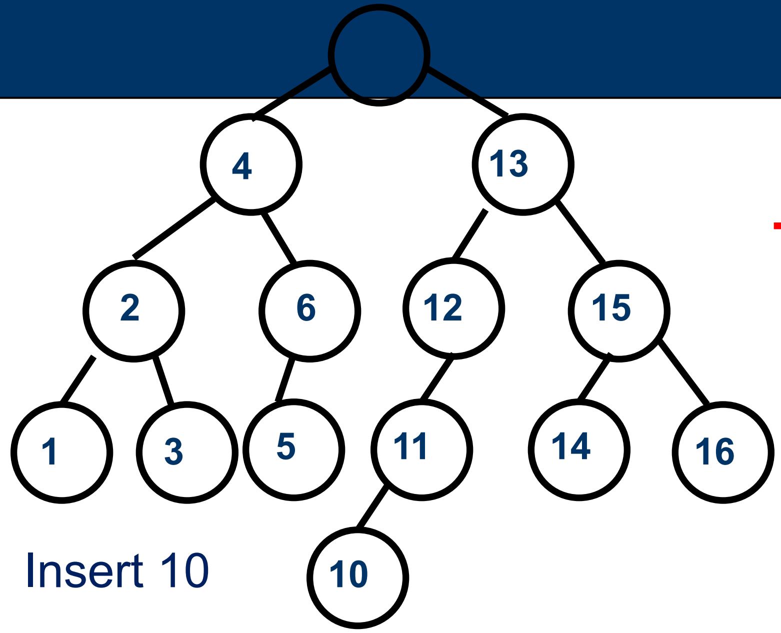
Single rotation



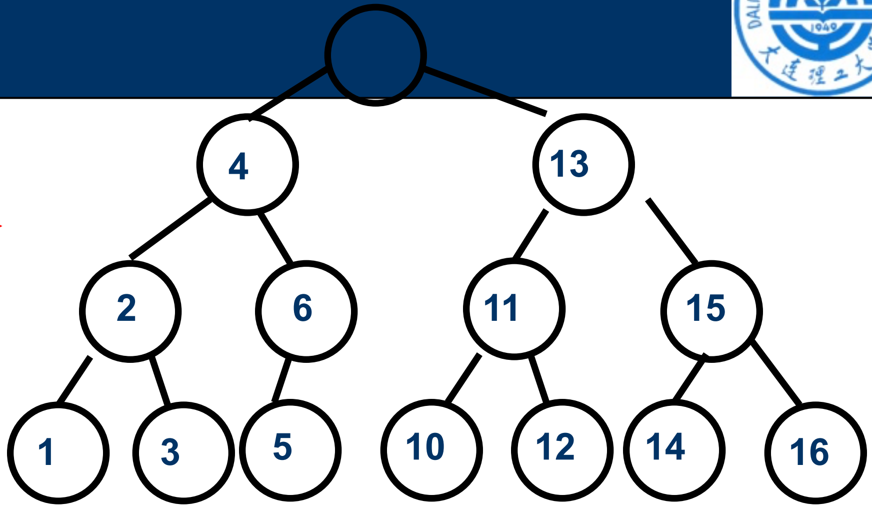
Insert 11



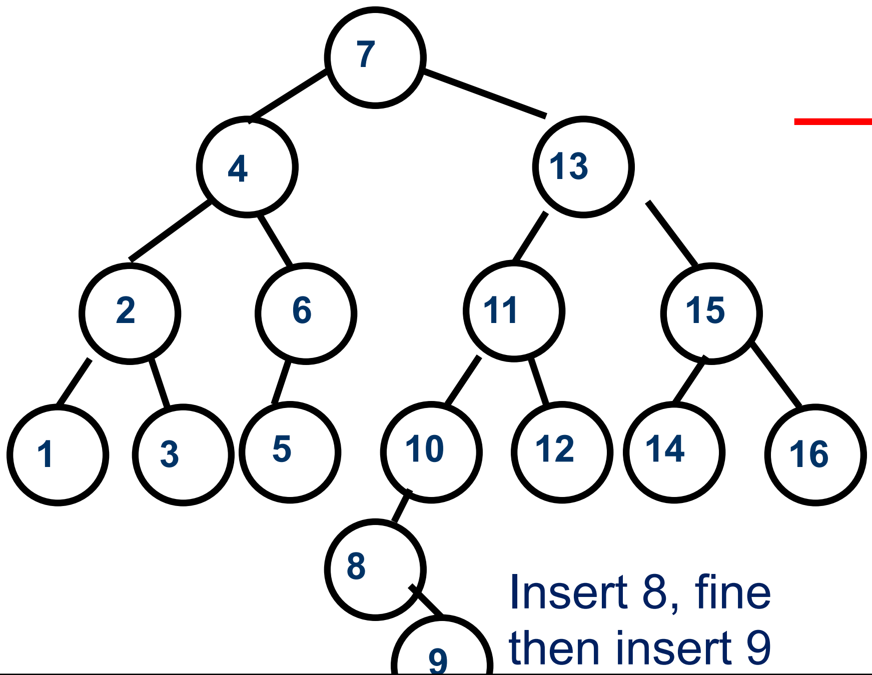
Single rotation



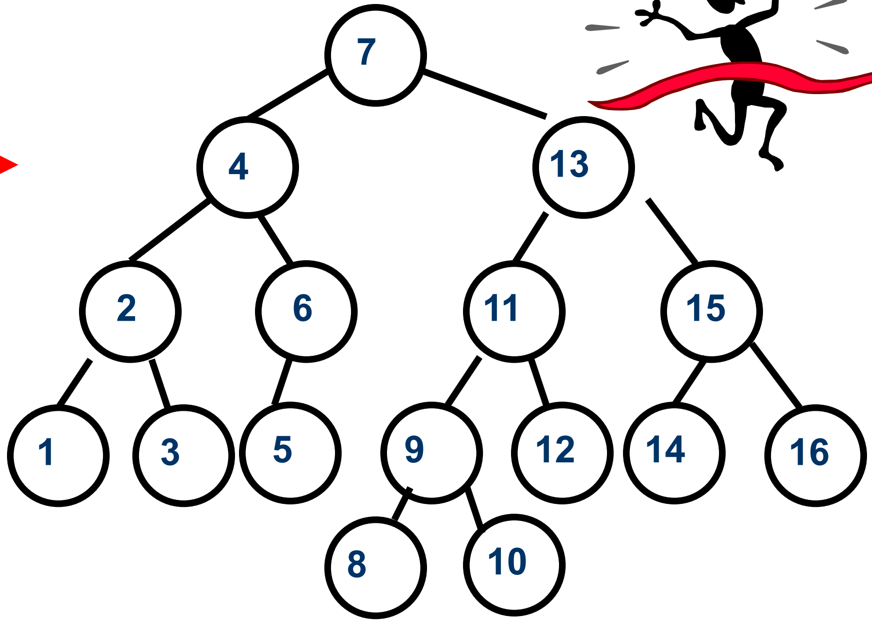
Insert 10



Single rotation



Insert 8, fine
then insert 9



Double rotation

Node declaration for AVL trees

```
template <class Comparable>
class AvlTree;

template <class Comparable>
class AvlNode
{
   Comparable element;
   AvlNode   *left;
   AvlNode   *right;
   int        height;

   AvlNode( const Comparable & theElement, AvlNode *lt,
            AvlNode *rt, int h = 0 )
     : element( theElement ), left( lt ), right( rt ),
                height( h ) { }
   friend class AvlTree<Comparable>;
};
```

Height

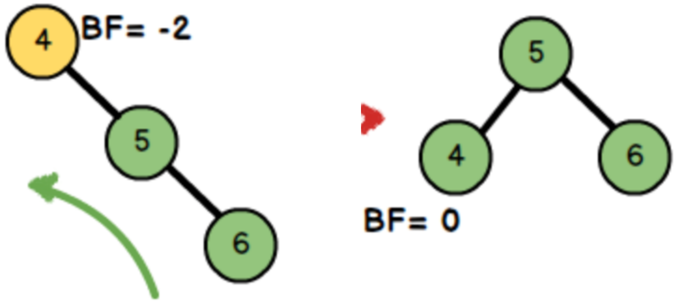
```
template class <Comparable>
int AvlTree<Comparable>::height( AvlNode<Comparable>
   *t) const
{
   return t == NULL ? -1 : t->height;
}
```

Insertion

```
/* Internal method to insert into a subtree.
 * x is the item to insert.
 * t is the node that roots the tree.
 */
template <class Comparable>
void AvlTree<Comparable>::insert( const Comparable & x, AvlNode<Comparable> * & t ) co
{
   if( t == NULL )  // find the position to insert
     t = new AvlNode<Comparable>( x, NULL, NULL );
   else if( x < t->element ) // the inserted value is less then the current node,
   {
     insert( x, t->left );// then insert it as the left child of the node
     if( height( t->left ) - height( t->right ) == 2 )//if insertion leads unbalance
       if( x < t->left->element )// case 2, insert as left child of the left subtre
           rotateWithLeftChild( t ); //do right rotation
       else
           doubleWithLeftChild( t );//case4 insert as the right child of the left subt
   }
   else if( t->element < x ) // the inserted value is less then the current node
   {
       insert( x, t->right ); //then insert it as the right child of the node
       if( height( t->right ) - height( t->left ) == 2 ) //if insertion causes unbalar
          if( t->right->element < x ) // case 1, insert as r child of the r subtree
             rotateWithRightChild( t );
          else
             doubleWithRightChild( t ); }//case 3, insert as the l child of the r subt
    else
       ;  // Duplicate; do nothing
    t->height = max( height( t->left ), height( t->right ) ) + 1;
}
```

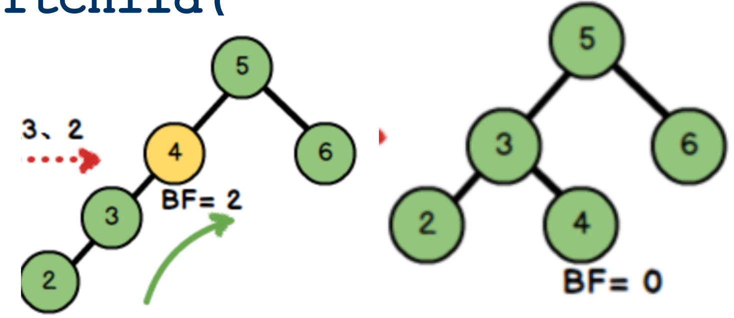
Single Left Rotation

```
/**
 * Rotate binary tree node with left child.
 * For AVL trees, this is a single rotation for case 1.
 * Update heights, then set new root.
 */
template <class Comparable>
void AvlTree<Comparable>::rotateWithLeft
  AvlNode<Comparable> * & k2 ) const
{
   AvlNode<Comparable> *k1 = k2->right;
   k2->right = k1->left;
   k1->left = k2;
   k2->height = max( height( k2->left ), height( k2->right ))+1;
   k1->height = max( height( k1->left ), k2->height )) + 1;
   k2 = k1;
}
```



Single right rotation

```
/**
 * Rotate binary tree node with left child.
 * For AVL trees, this is a single rotation for case 1.
 * Update heights, then set new root.
 */
template <class Comparable>
void AvlTree<Comparable>::rotateWithLeftChild(
  AvlNode<Comparable> * & k2 ) const
{
   AvlNode<Comparable> *k1 = k2->left;
   k2->left = k1->right;
   k1->right = k2;
   k2->height = max( height( k2->left ), height( k2->right ))+1;
   k1->height = max( height( k1->left ), k2->height )) + 1;
   k2 = k1;
}
```



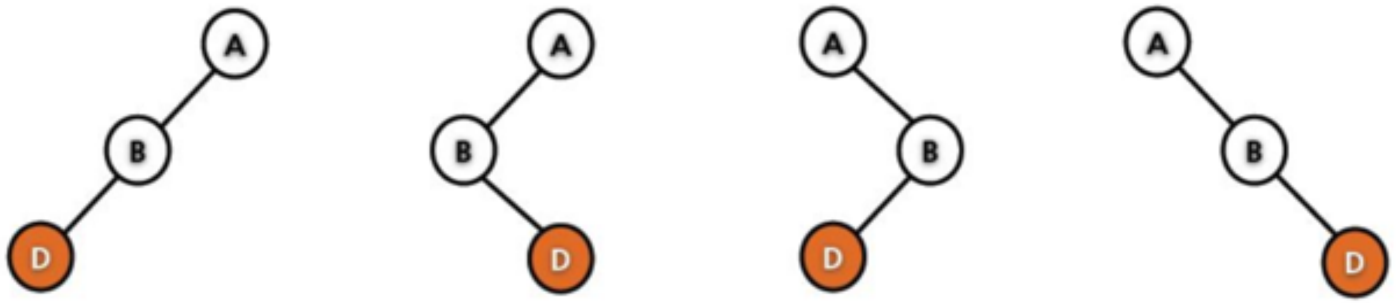
Double Rotation

```
/**
 * Double rotate binary tree node: first left child.
 * with its right child; then node k3 with new left child.
 * For AVL trees, this is a double rotation for case 2.
 * Update heights, then set new root.
 */
template <class Comparable>
void AvlTree<Comparable>::doubleWithLeftChild(
  AvlNode<Comparable> * & k3 ) const
{
   rotateWithRightChild( k3->left );
   rotateWithLeftChild( k3 );
}
```

Balance Factor

```
Balance Factor = The height of the left tree - The height of the right tree.
```

And then, The unbalanced state of the tree could be represented as the following picture.



1. **BF > 1**,The value of the new node is **smaller** than the value of the **left node**.
2. **BF > 1**, The value of the new node is **larger** than the value of the **left node**.
3. **BF < -1**, The value of the new node is **smaller** than the value of the **right node**.
4. **BF < -1**, The value of the new node is **larger** than the value of the **right node**.

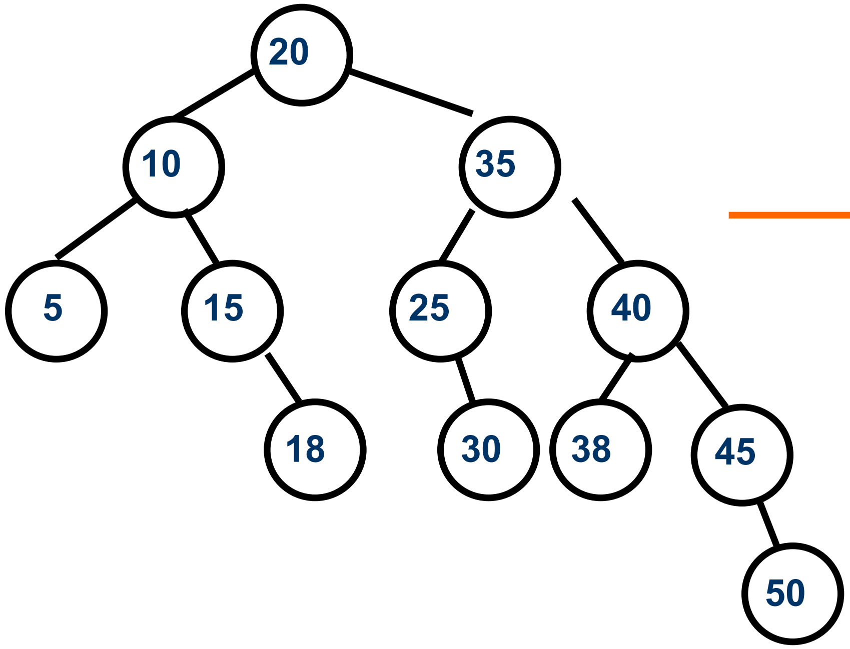
In each state of the tree, We have to conduct the rotation like this:

1. Right rotation (pivot is the B)
2. Left rotation (pivot is the D) and Right rotation.(pivot is the B)
3. Right rotation (pivot is the D) and Left rotation.(pivot is the B)
4. Left rotation (pivot is the B)

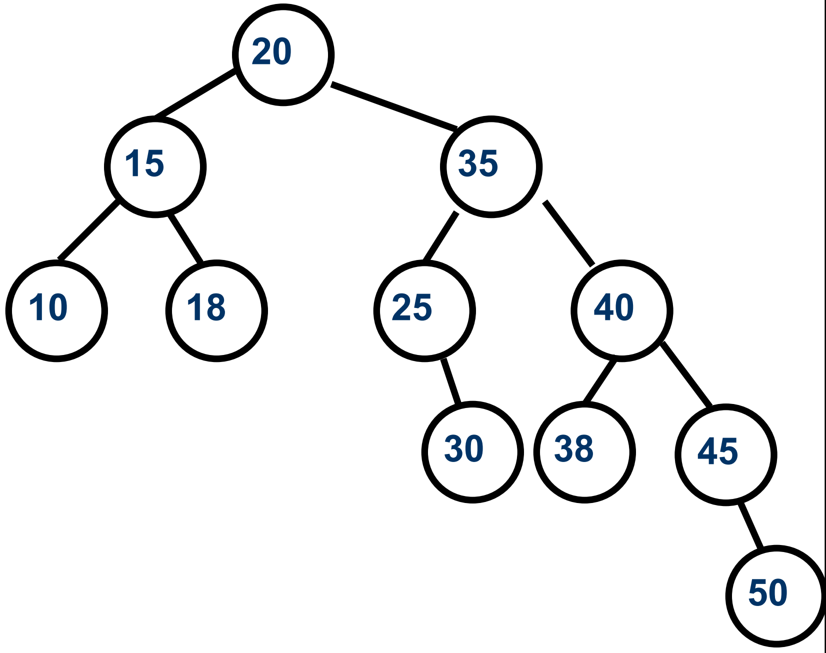
AVL Tree Deletion

- Similar but more complex than insertion
  - Rotations and double rotations needed to rebalance
  - Imbalance may propagate upward so that many rotations may be needed.

Deletion Example 1

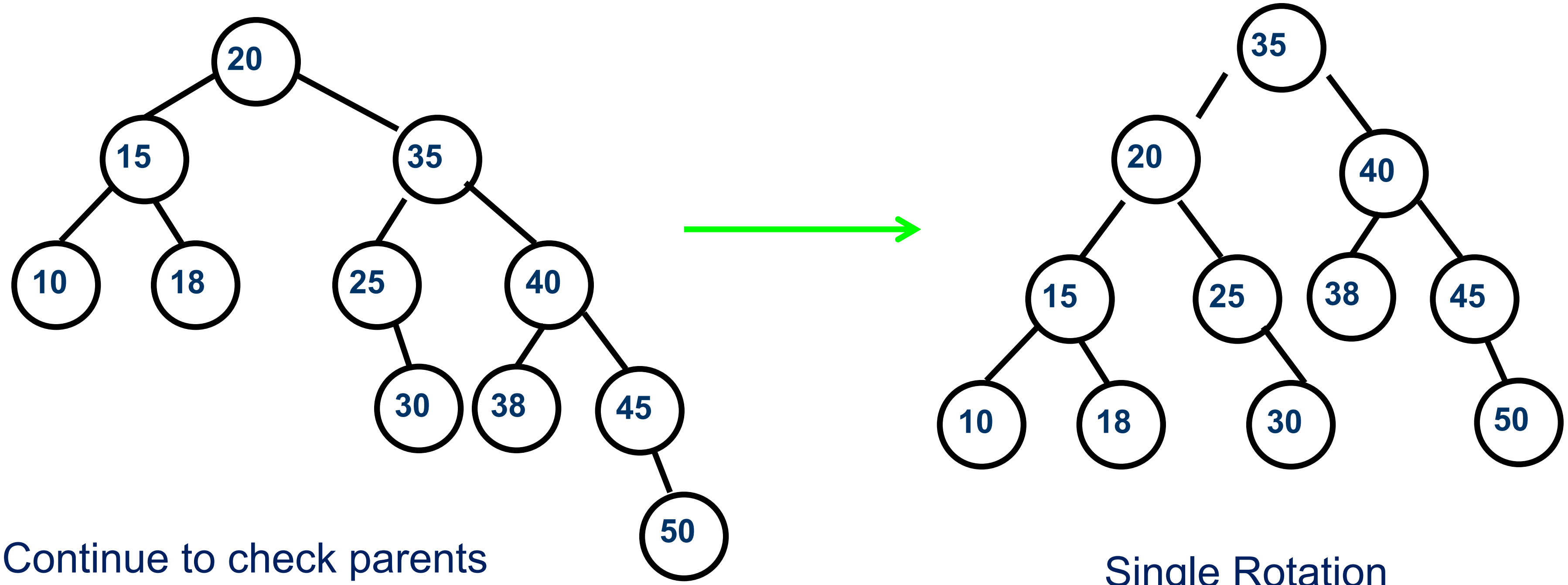


Delete 5, Node 10 is unbalanced



Single Rotation

Cont'd



Continue to check parents
Oops!! Node 20 is unbalanced!!

Single Rotation

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

Pros and Cons of AVL Trees

Arguments for AVL trees:

1. Search is O(log N) since AVL trees are always balanced.
2. Insertion and deletions are also O(logn)
3. The height balancing adds no more than a constant factor to the speed of insertion.

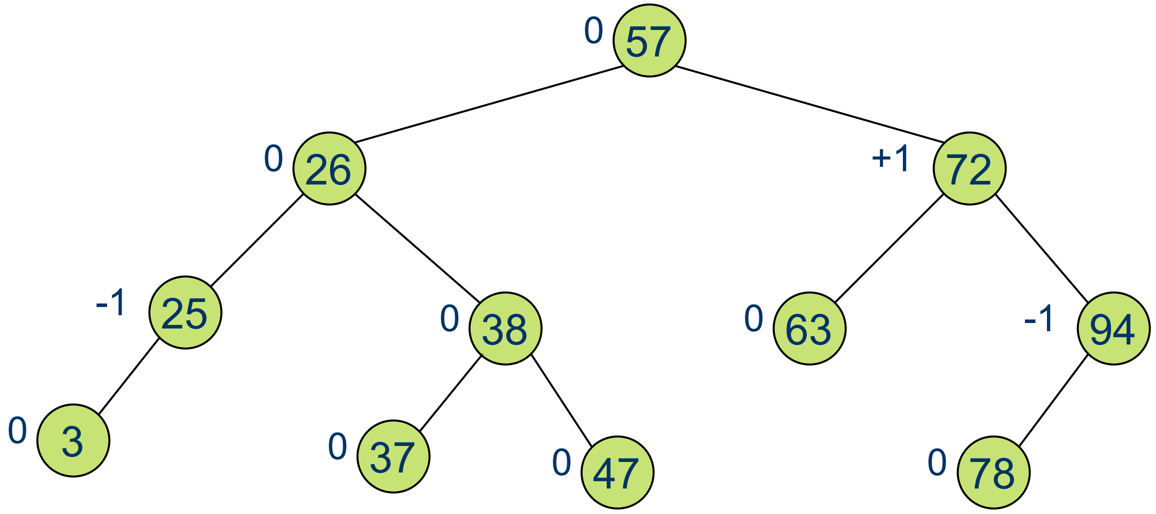
Arguments against using AVL trees:

1. Difficult to program & debug; more space for balance factor.
2. Asymptotically faster but rebalancing costs time.
3. Most large searches are done in database systems on disk and use other structures (e.g. B-trees).
4. May be OK to have O(N) for a single operation if total run time for many consecutive operations is fast (e.g. Splay trees).

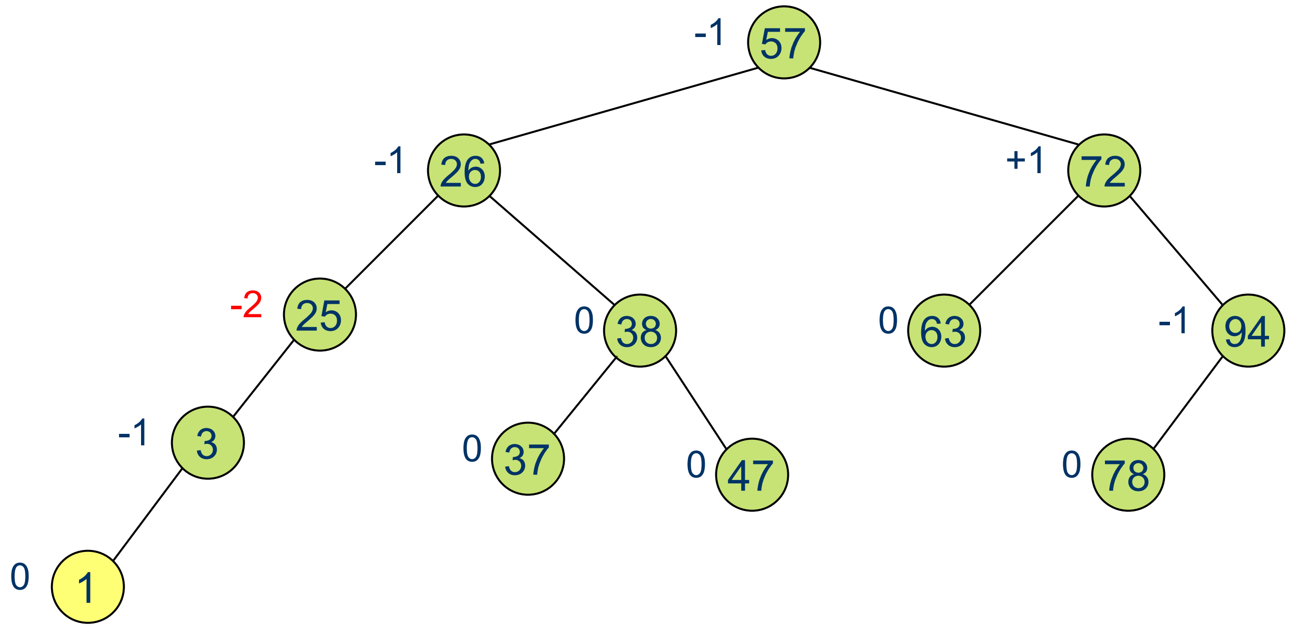
exercise

Note: Balance factor=h(right subtree)-h(left subtree)

Balance ok



insert 1,30, 32, 35



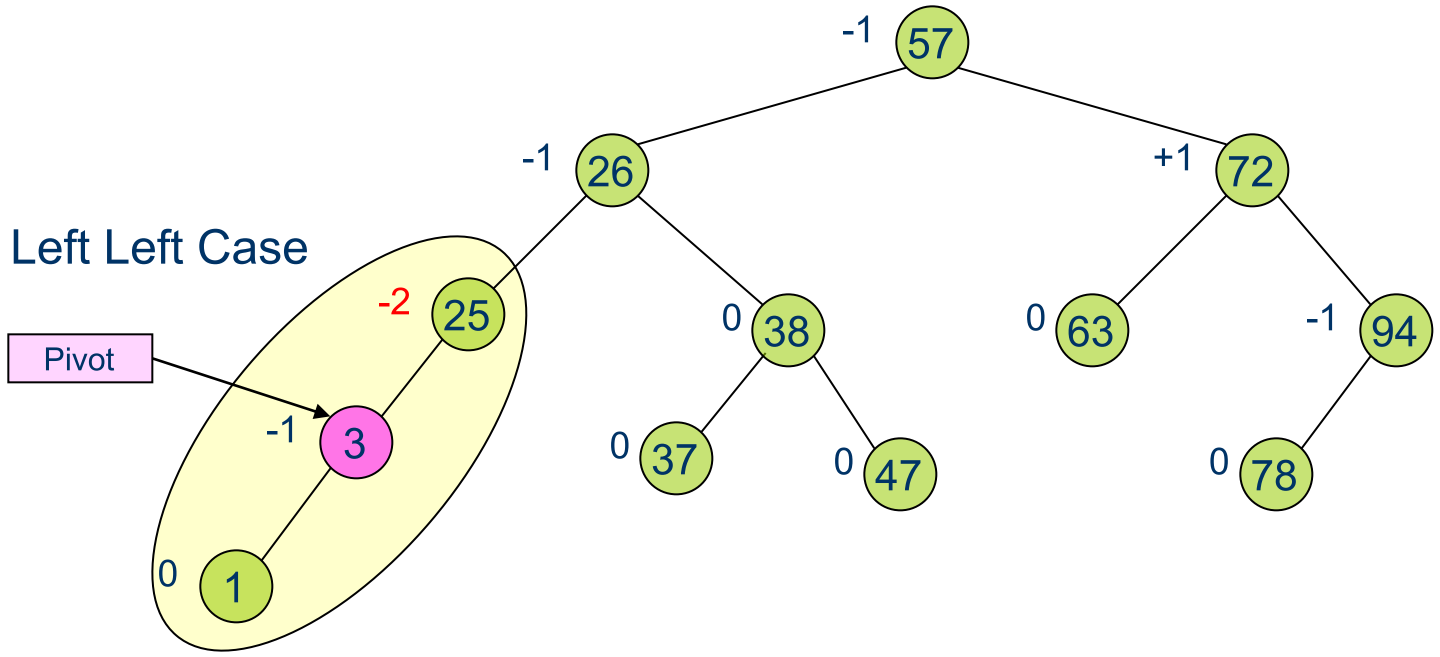
Balance not ok

(Balance factor of -2 is not allowed)

Next step: Find rebalancing case -->

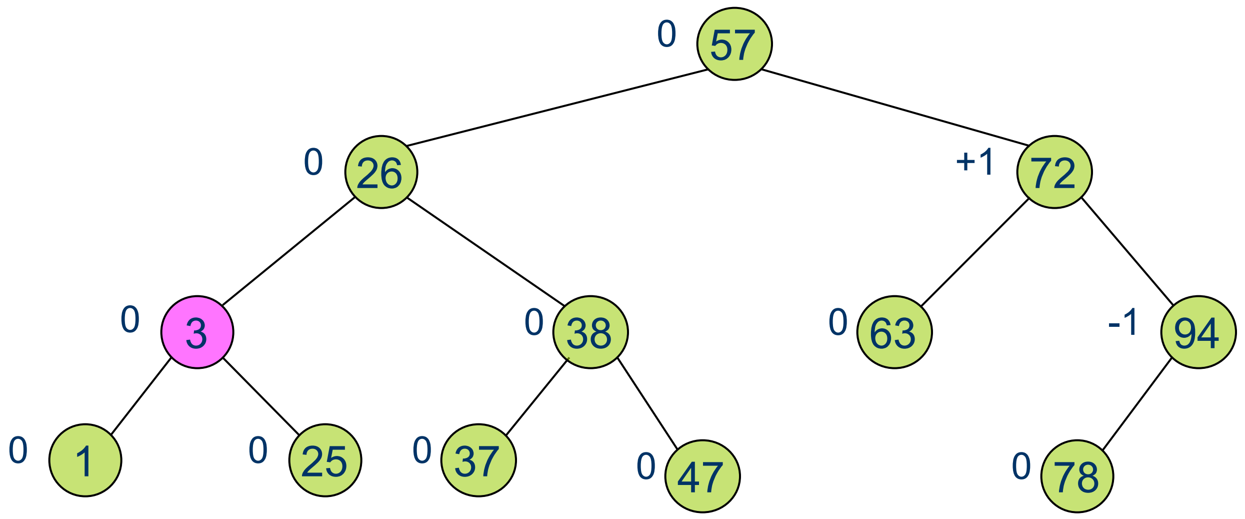
Balance not ok

Left Left Case



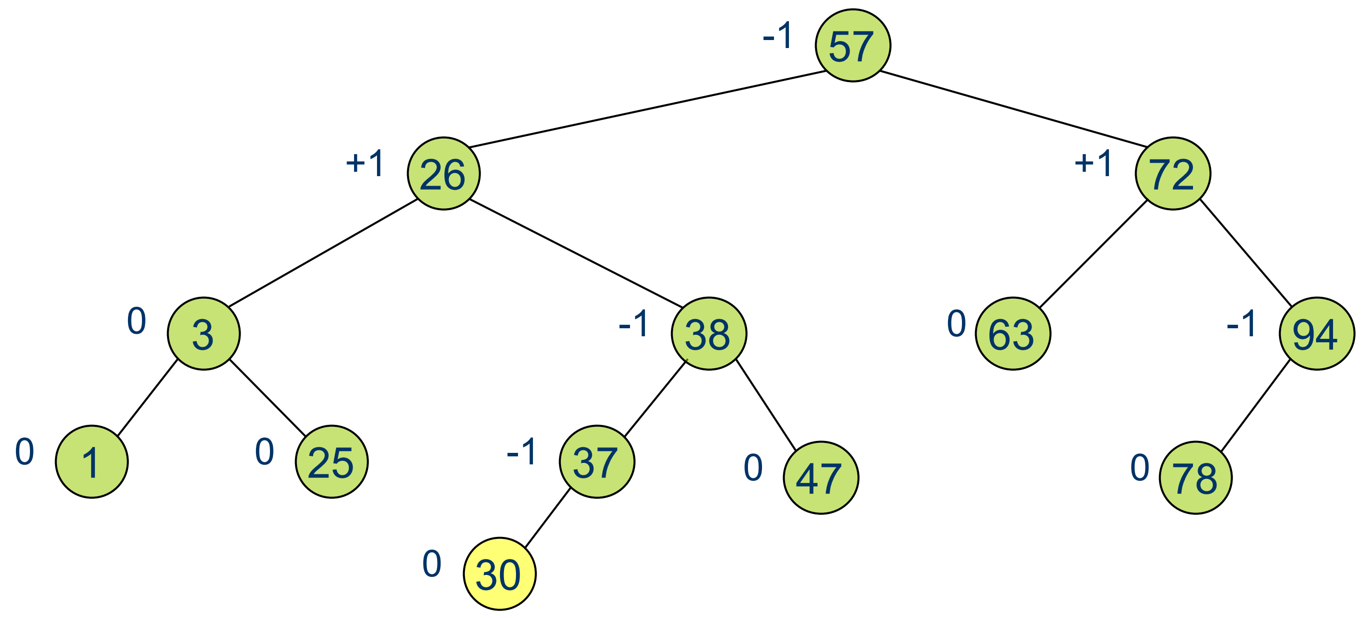
Next step: apply Left Left rebalancing -->

Balance ok



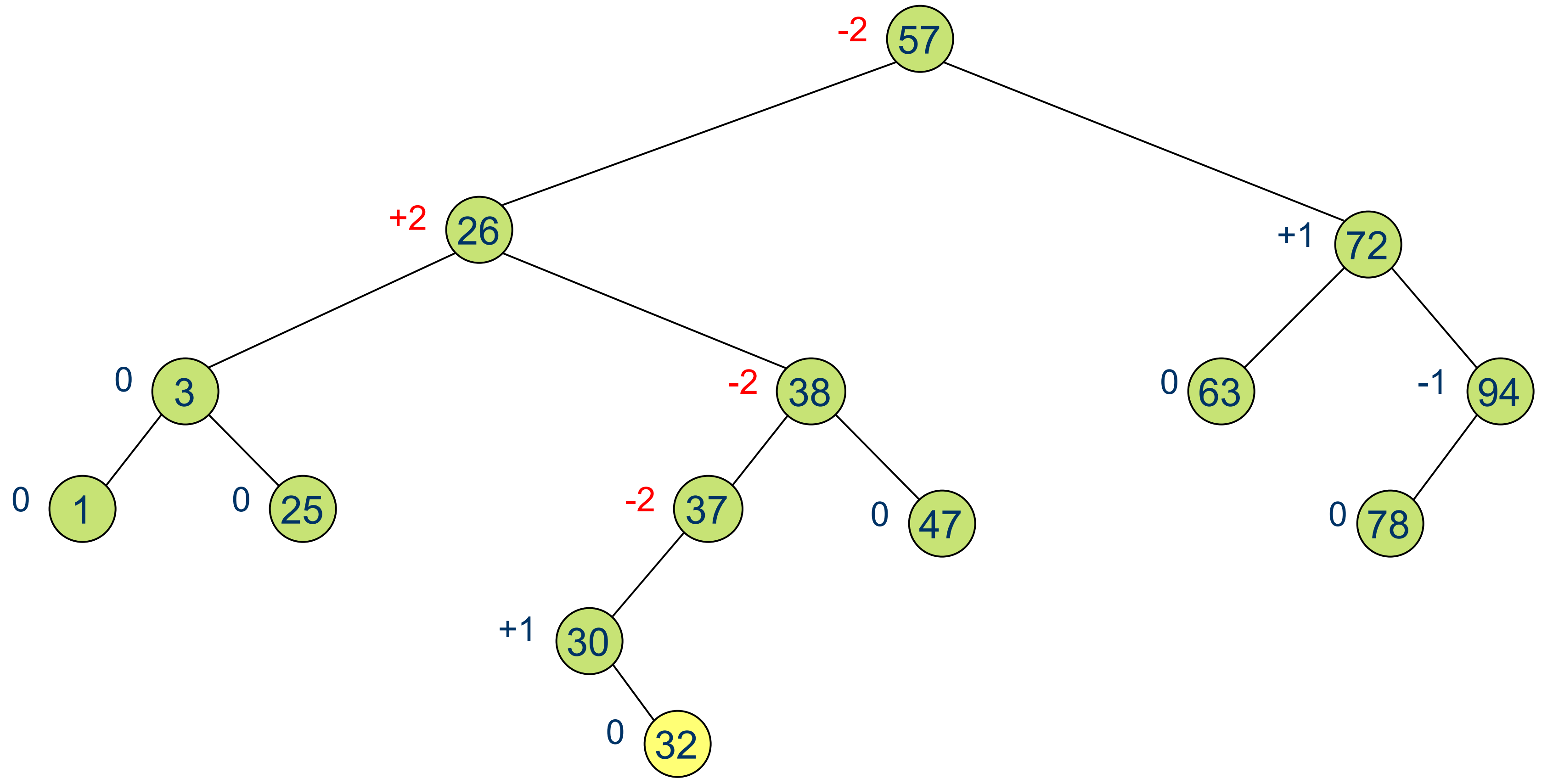
Next step: insert 30 -->

Balance ok



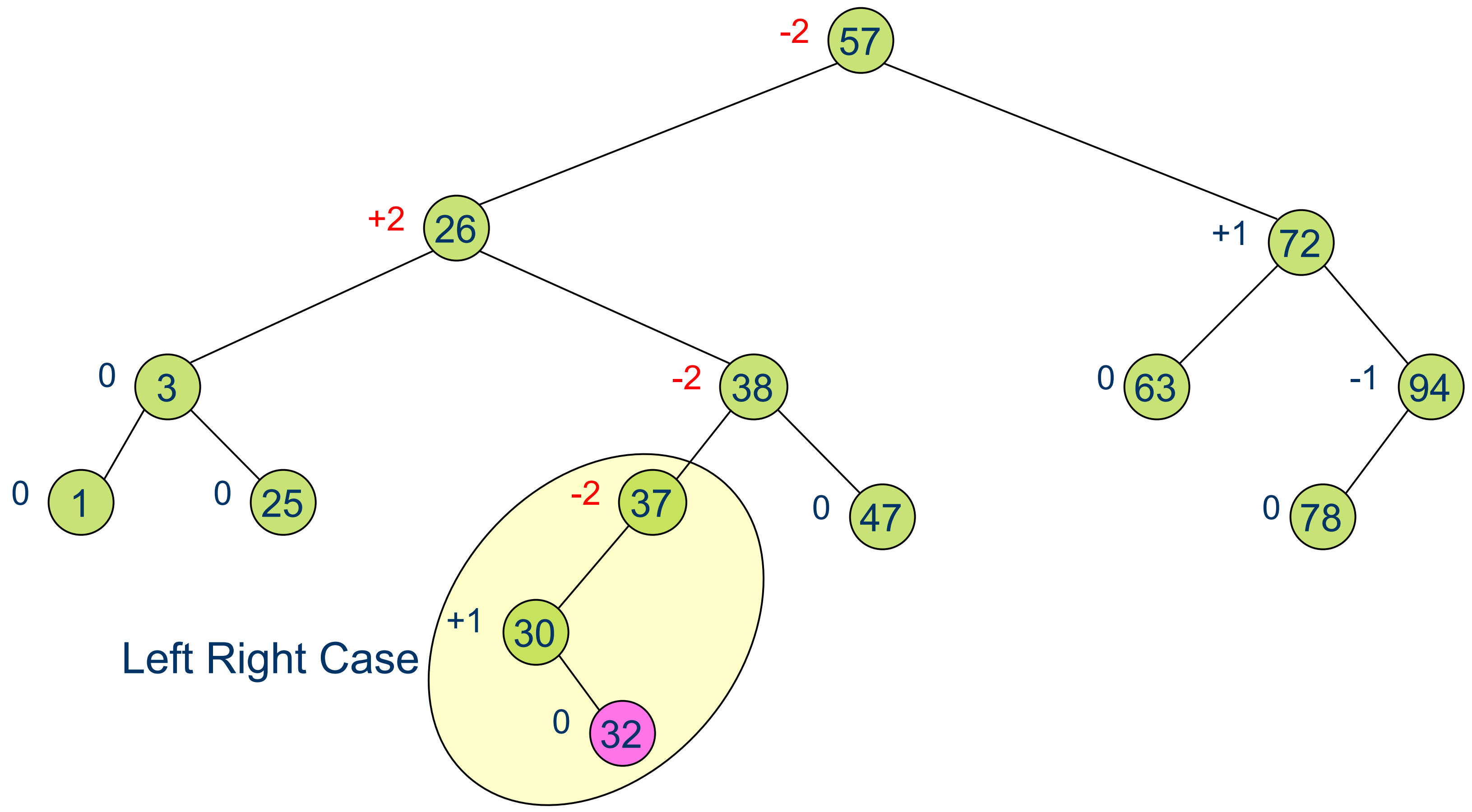
Next step: Insert 32 -->

Balance not ok



Next step: Find rebalancing case -->

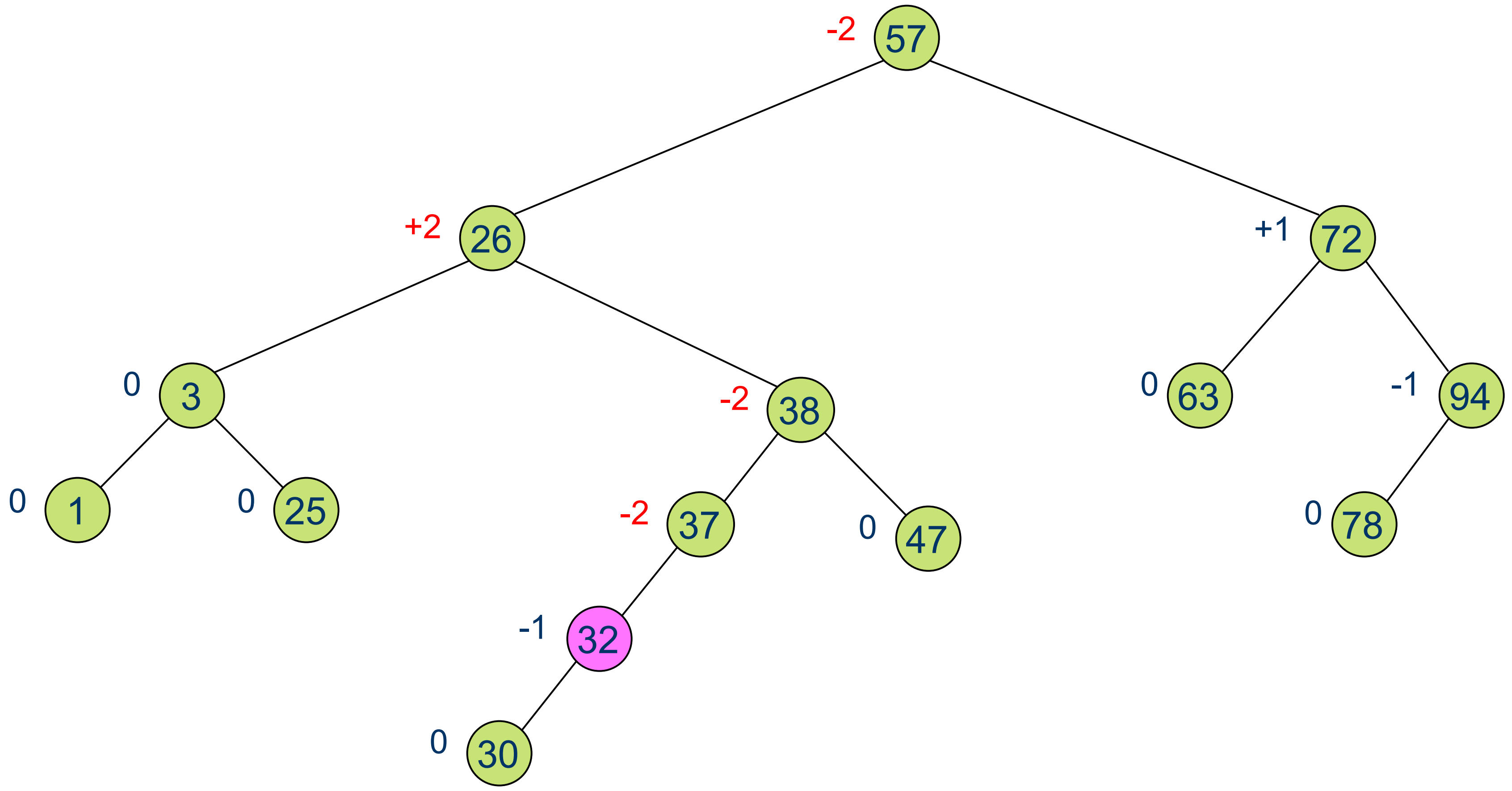
Balance not ok



Left Right Case

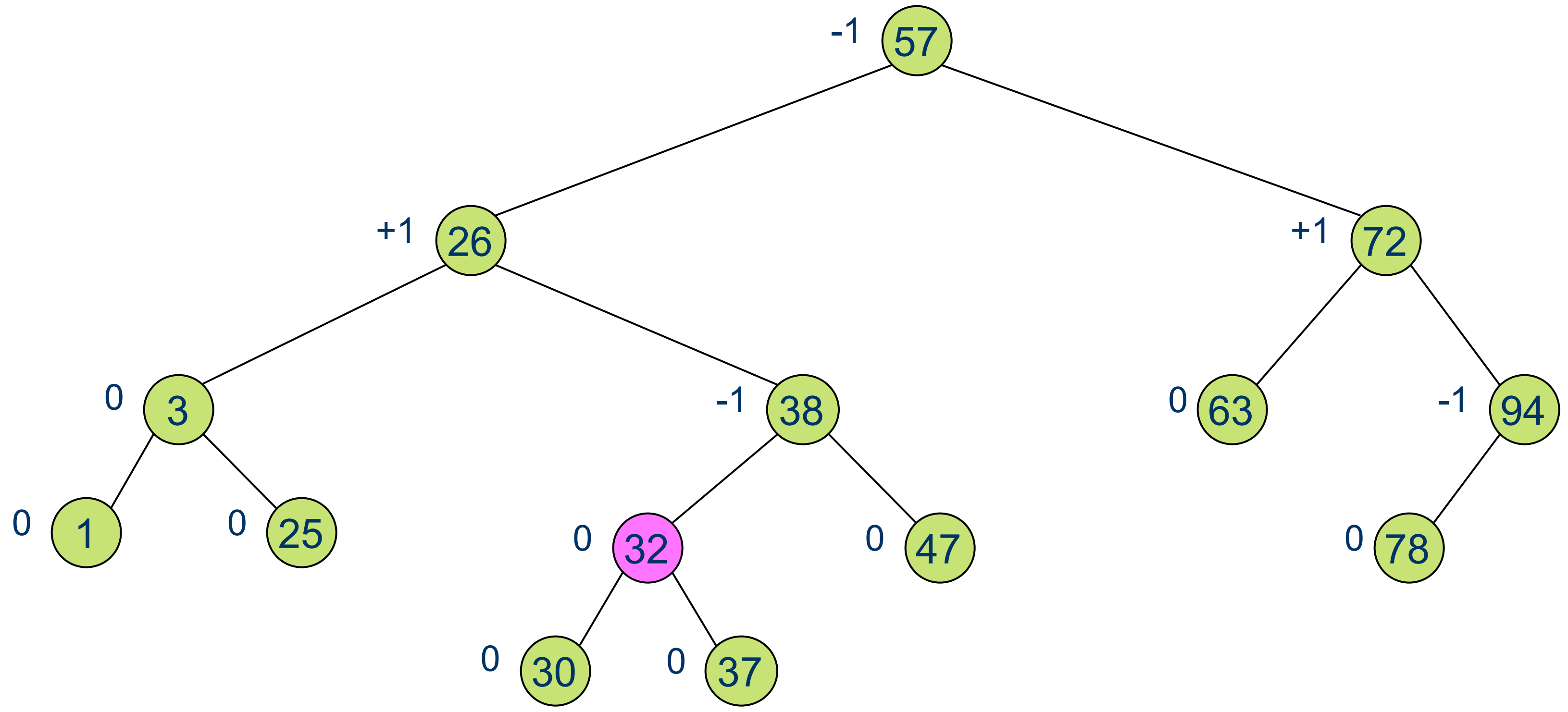
Next step: Rebalance (Step 1) -->

Balance not ok



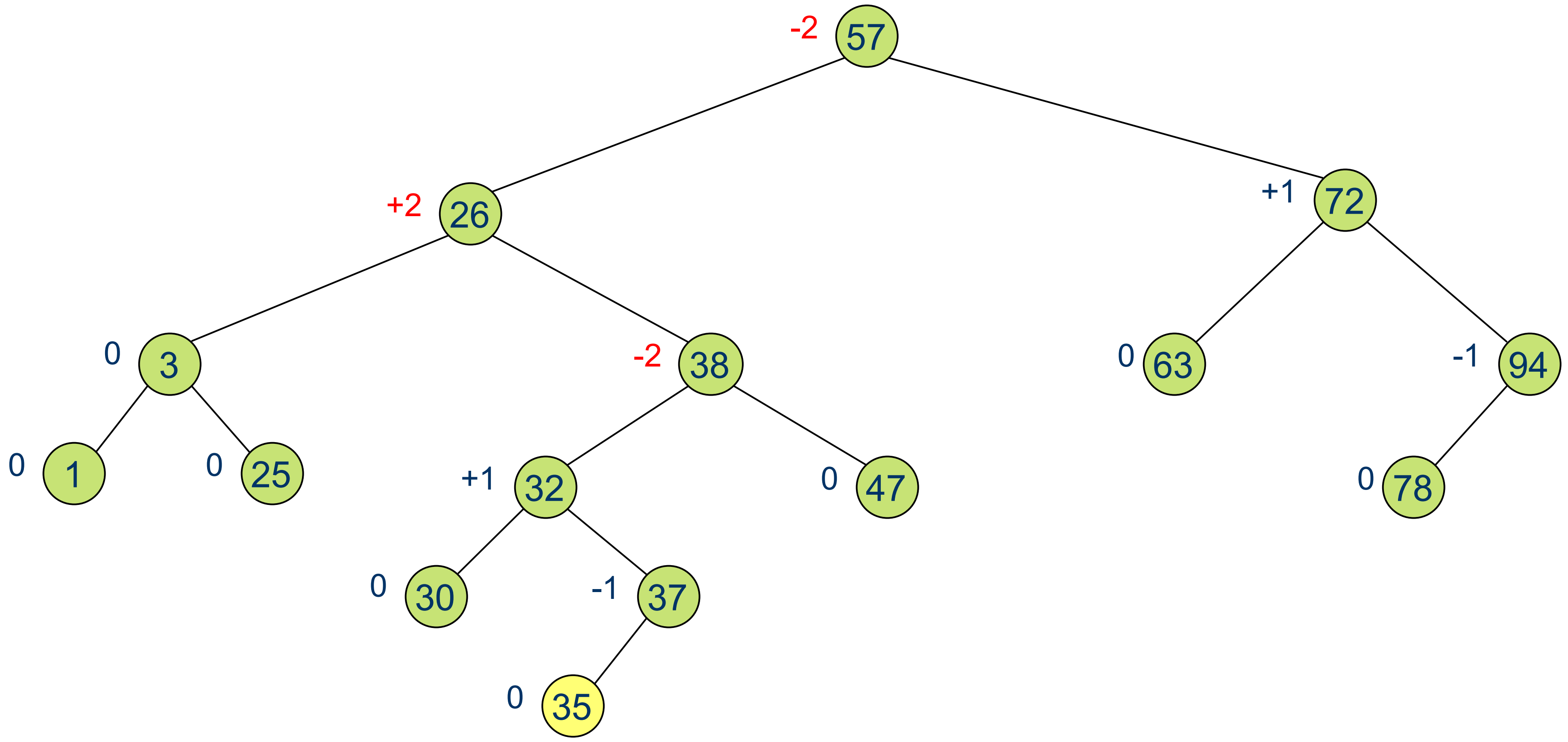
Next step: Rebalance (Step 2) -->

Balance ok



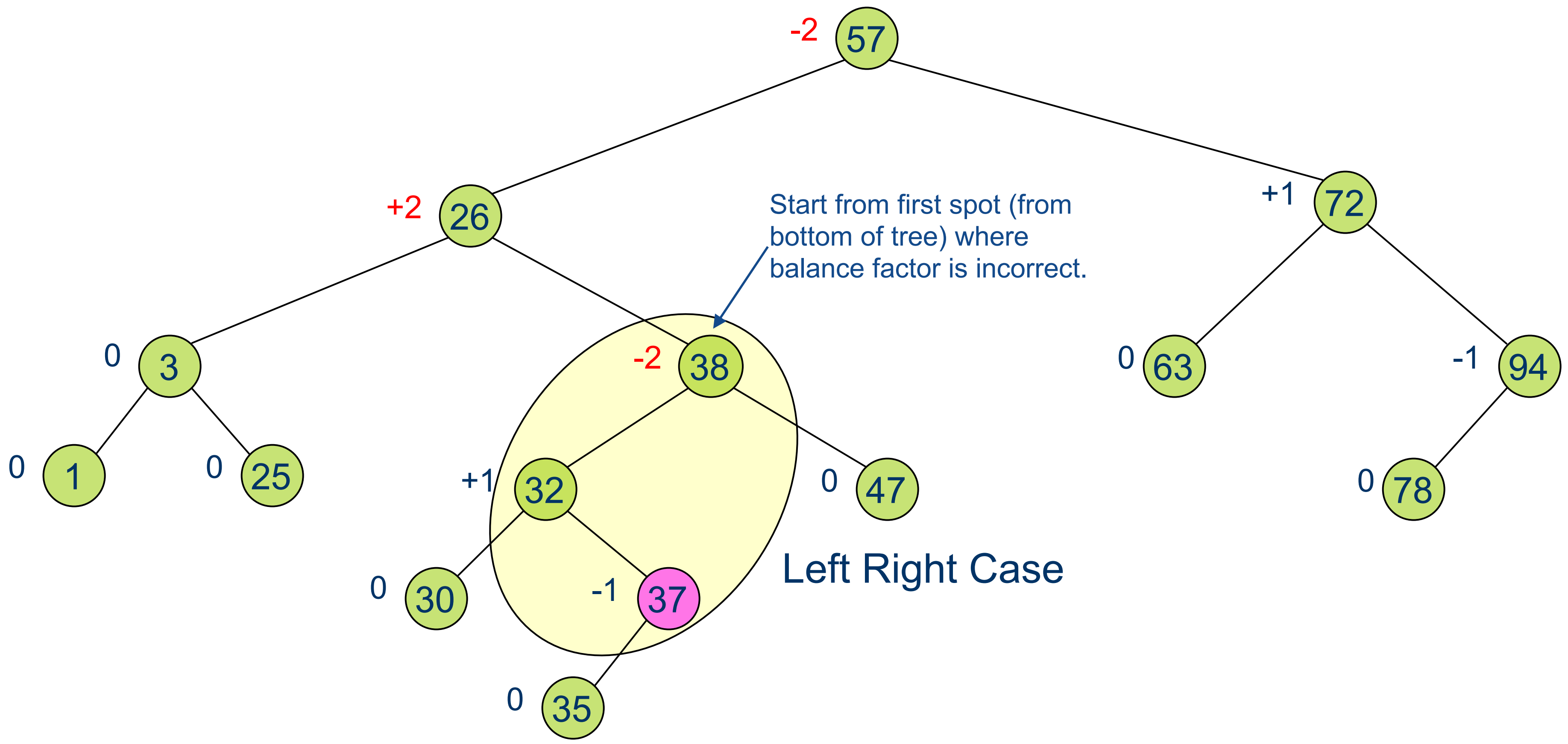
Next step: Insert 35 -->

Balance not ok



Next step: Find rebalancing case -->

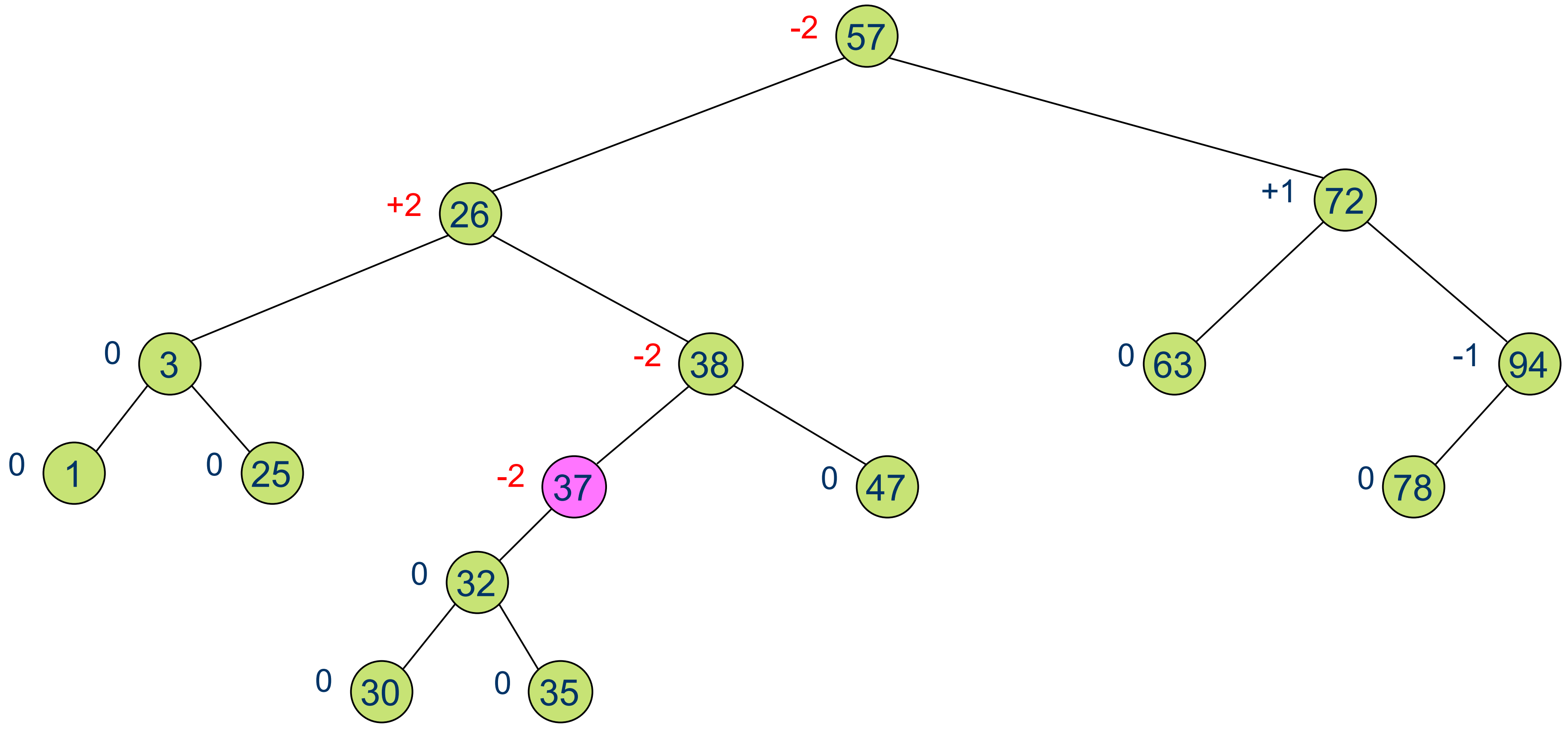
Balance not ok



-2
57
+2
26
+1
72
Start from first spot (from bottom of tree) where balance factor is incorrect.
0
3
-2
38
0
63
-1
94
0
1
0
25
+1
32
0
47
0
78
Left Right Case
0
30
-1
37
0
35

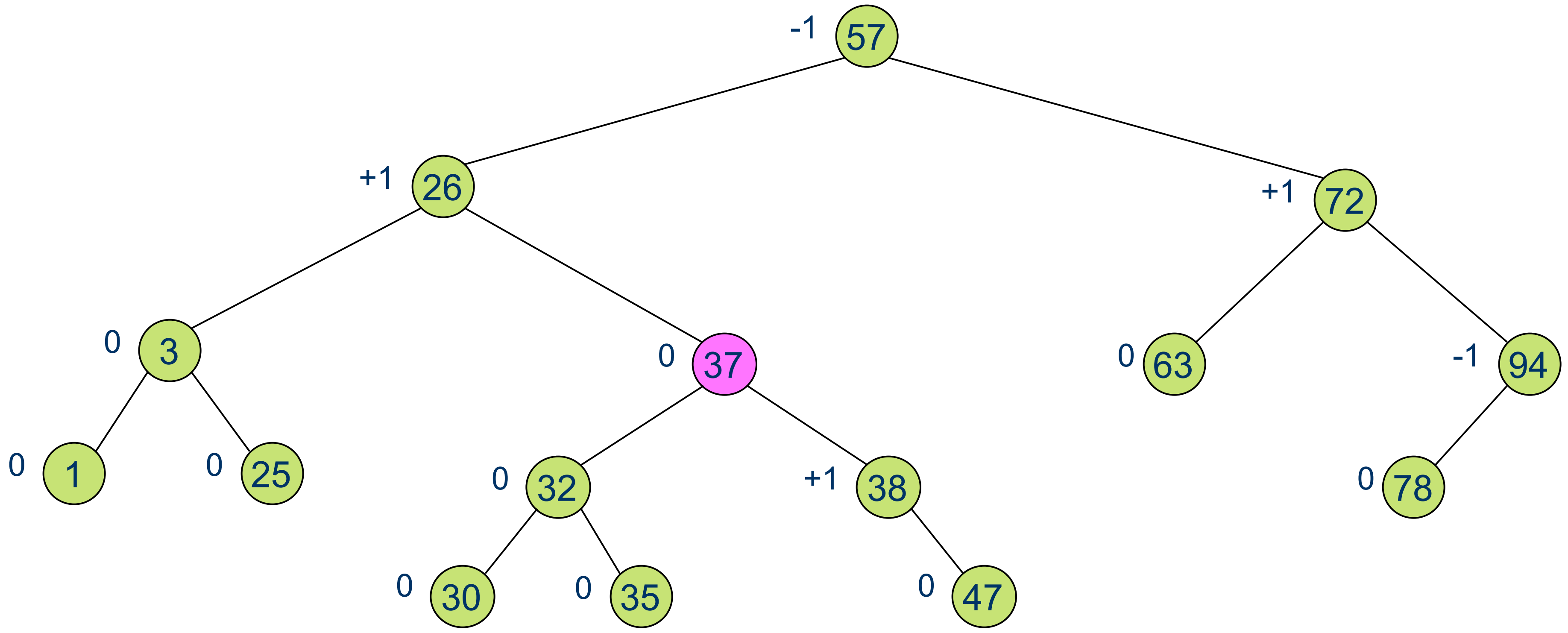
Next step: Rebalance (Step 1) -->

Balance not ok



Next step: Rebalance (Step 2) -->

Balance ok



Next step: Finished! -->

Homework

- Please refer to Icourse，Huawei Cloud.

- Due date for quiz: 23:30   2022/5/5
- Due date for homework: 23:30 2022/5/8
- Due data for online lab assignment: 2022/5/8   23：30
- Due data for offline lab assignment: 2022/5/8   18:00