AVL Trees

AVL Trees

- An AVL tree is a binary search tree with a *balance* condition.
- AVL is named for its inventors: **A**del'son-**V**el'skii and **L**andis
- 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.

Figure 19.21

Two binary search trees: (a) an AVL tree; (b) not an AVL tree (unbalanced nodes are darkened)

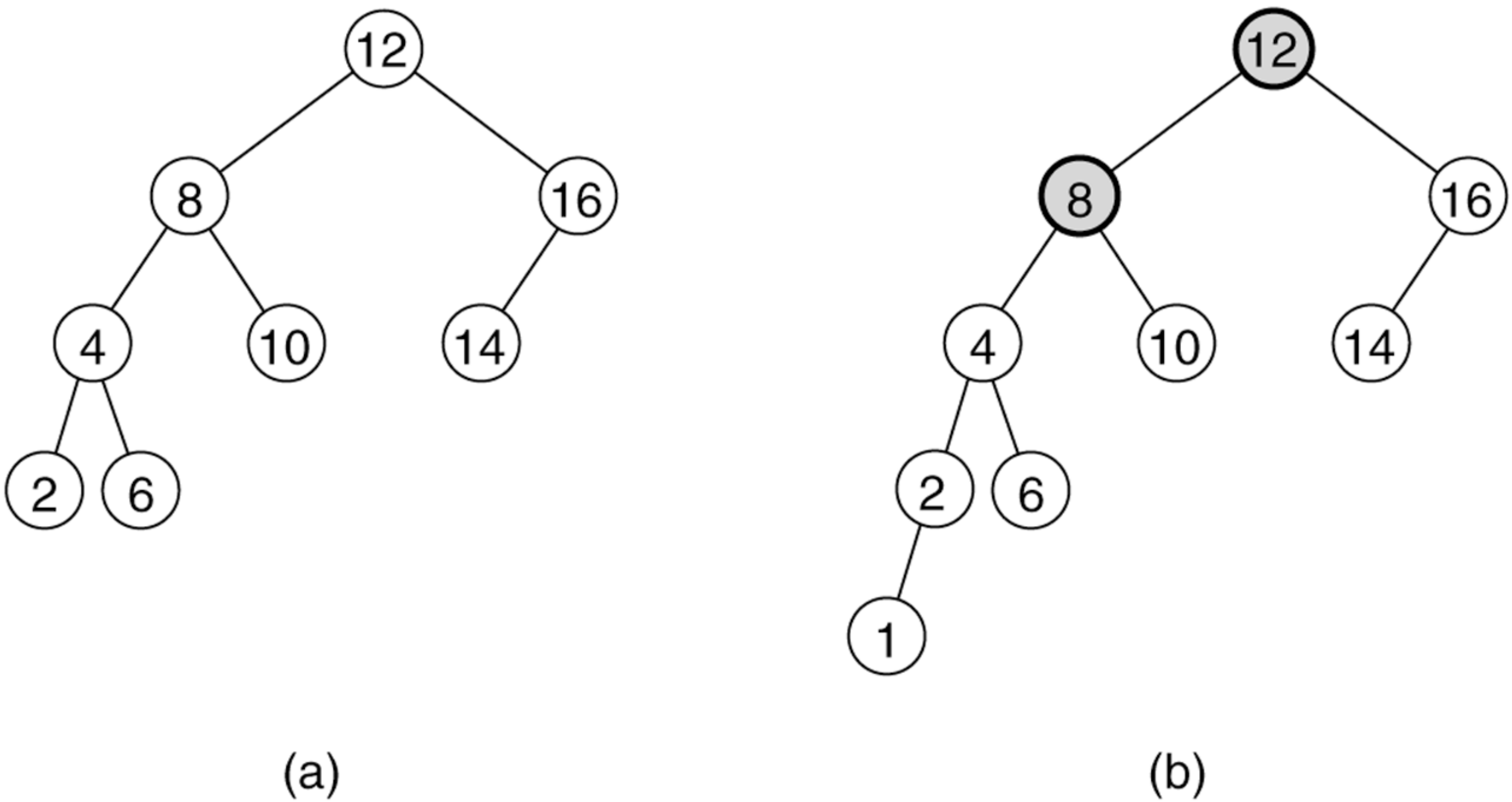
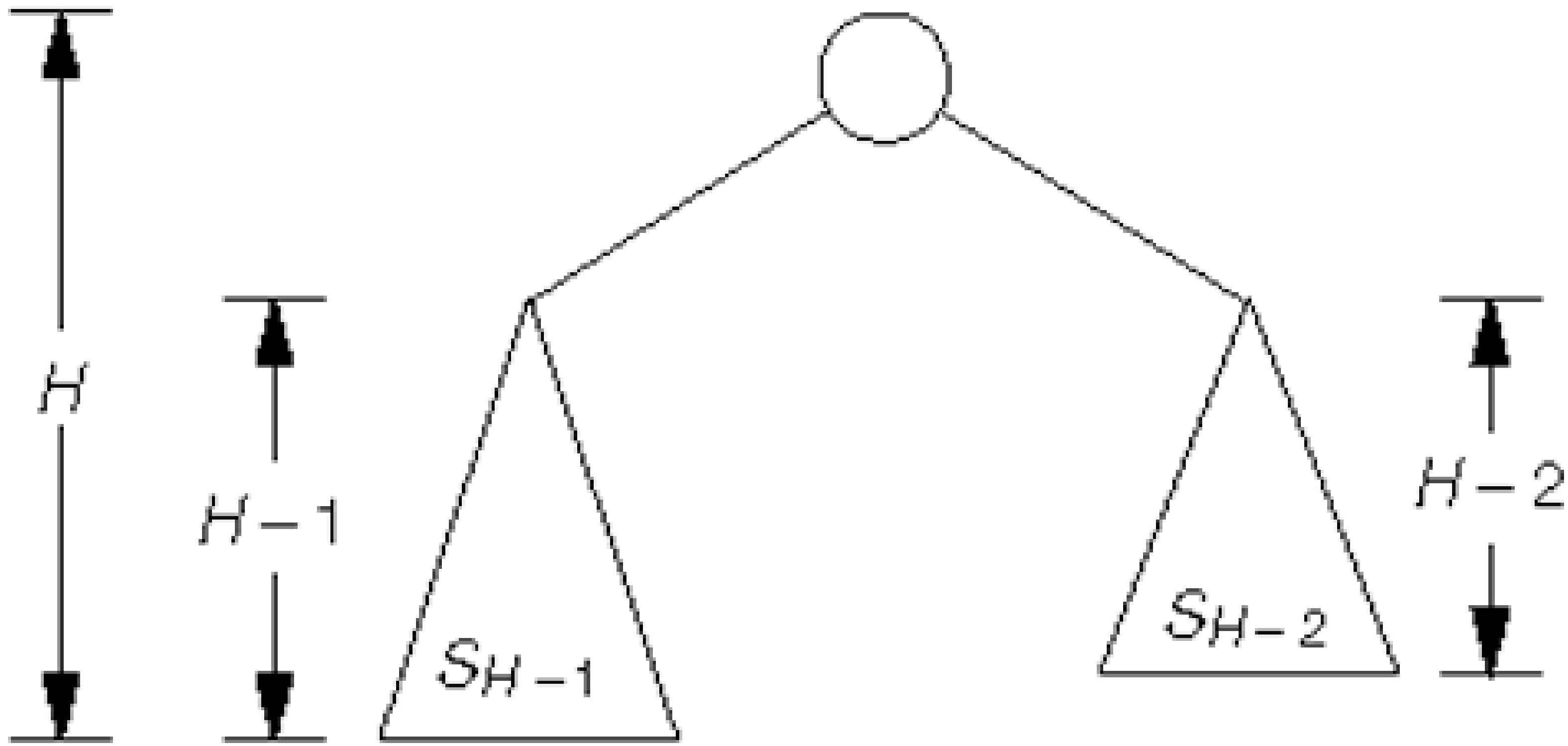


Figure 19.22

Minimum tree of height *H*



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.

Rebalancing

- Suppose the node to be rebalanced is X. There are 4 cases that we might have to fix (two are the mirror images of the other two):
  1. An insertion in the left subtree of the left child of X,
  2. An insertion in the right subtree of the left child of X,
  3. An insertion in the left subtree of the right child of X, or
  4. An insertion in the right subtree of the right child of X.
- Balance is restored by tree *rotations*.

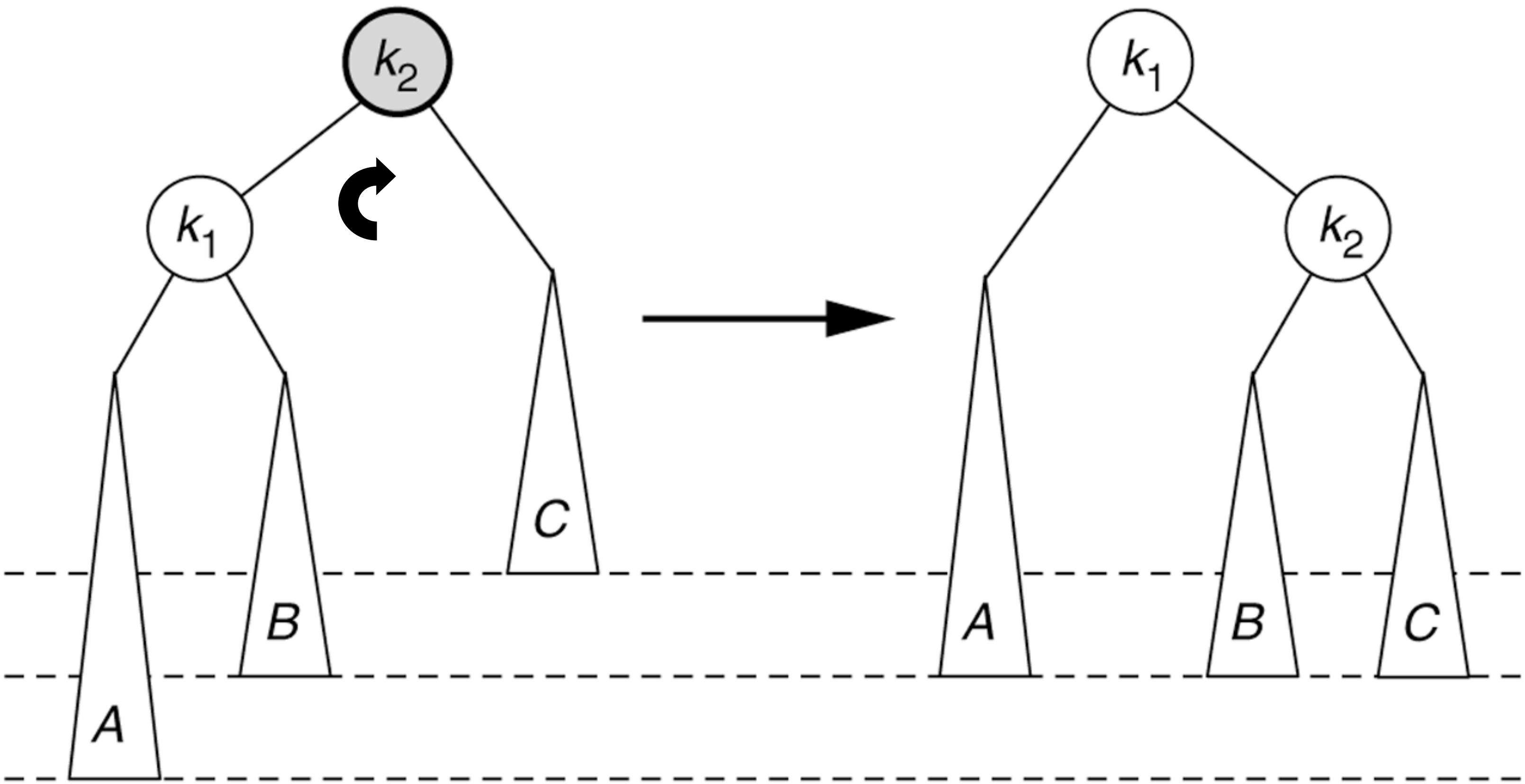
Balancing Operations: Rotations

- Case 1 and case 4 are symmetric and requires the same operation for balance.
  - Cases 1,4 are handled by *single rotation*.
- Case 2 and case 3 are symmetric and requires the same operation for balance.
  - Cases 2,3 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 4).
- We rotate between a node and its child.
  - Child becomes parent. Parent becomes right child in case 1, left child in case 4.
- The result is a binary search tree that satisfies the AVL property.

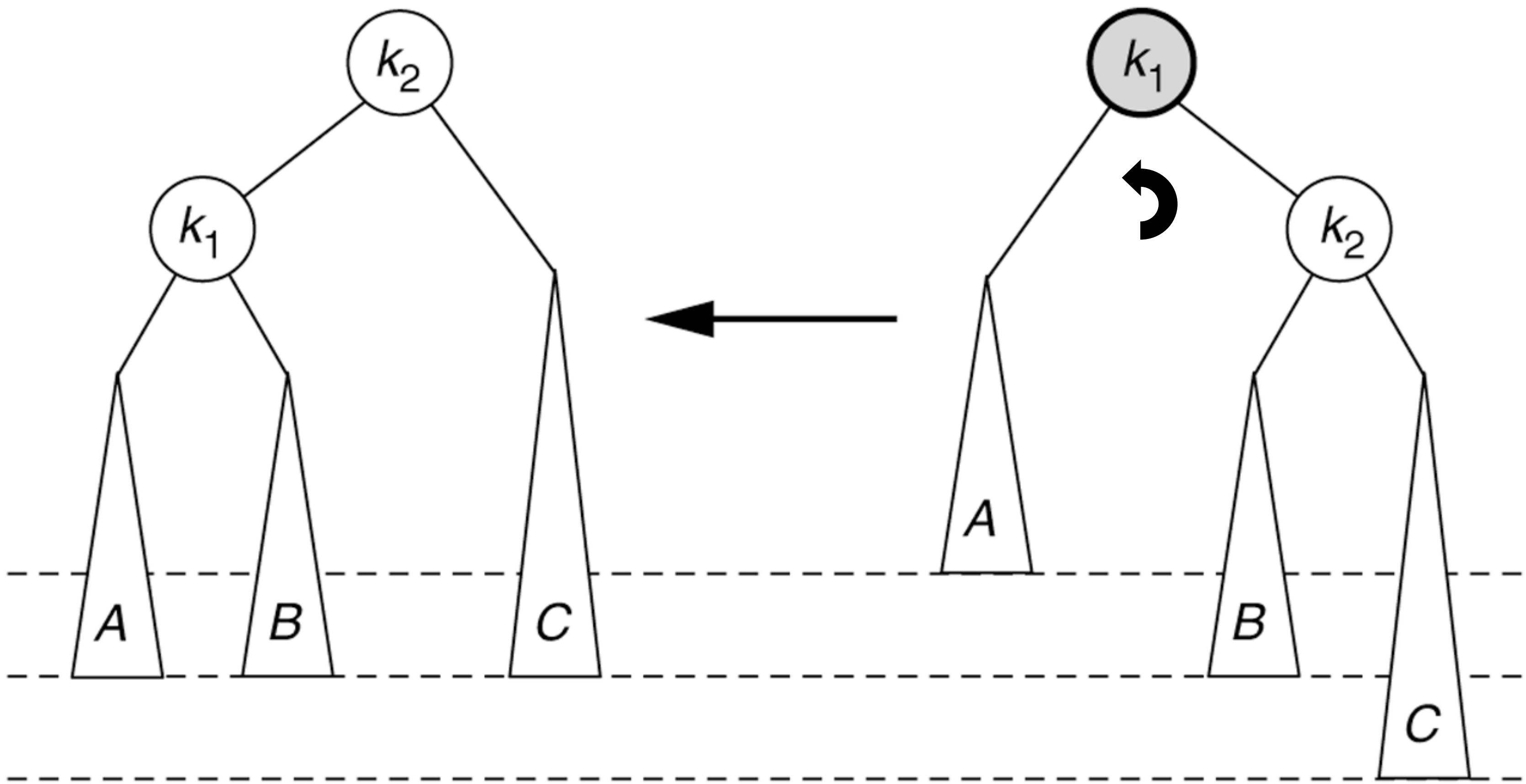
Figure 19.23
Single rotation to fix case 1: Rotate right



(a) Before rotation (b) After rotation

Figure 19.26

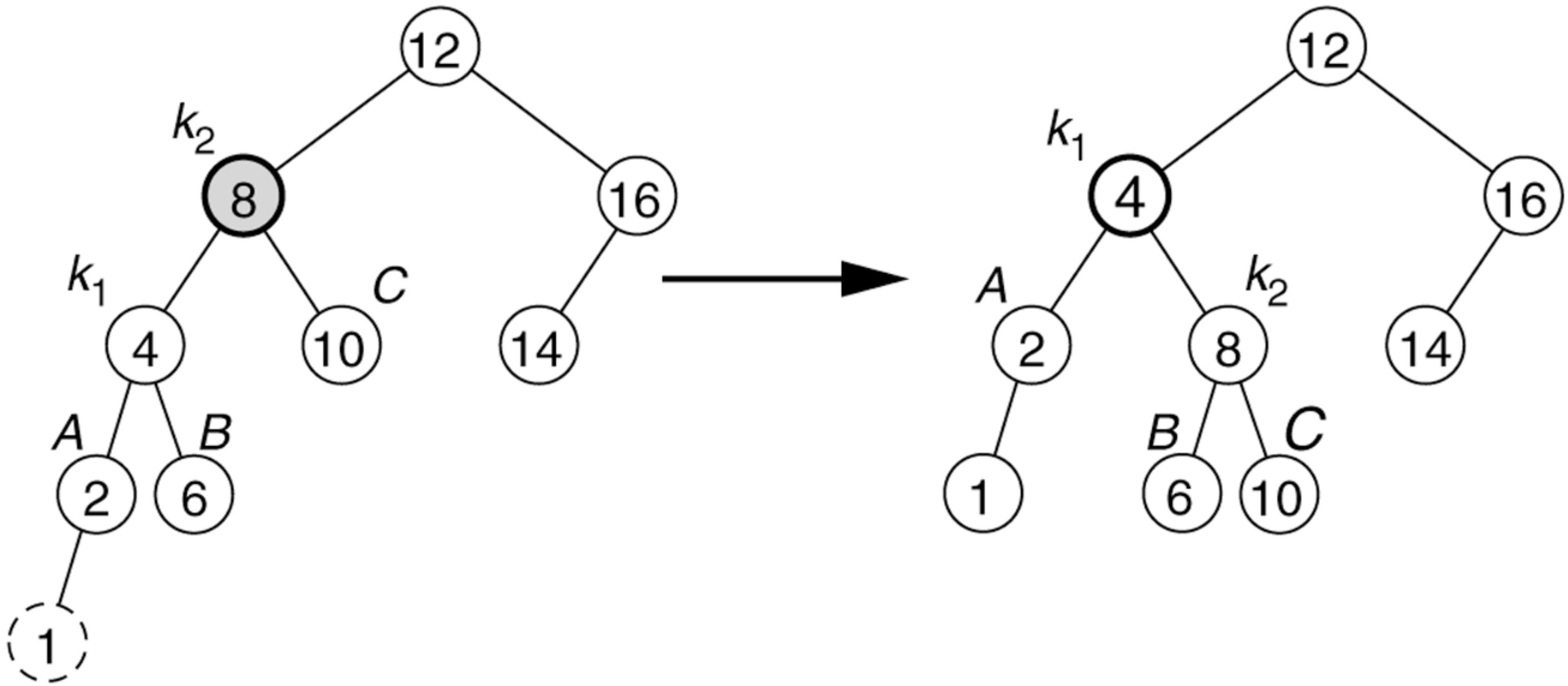
Symmetric single rotation to fix case 4 : Rotate left



(a) After rotation (b) Before rotation

Figure 19.25

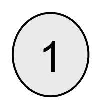
Single rotation fixes an AVL tree after insertion of 1.



(a) Before rotation (b) After rotation

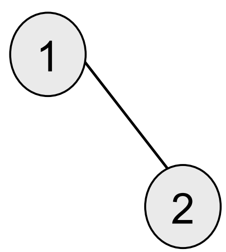
Example

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.
- Answer:



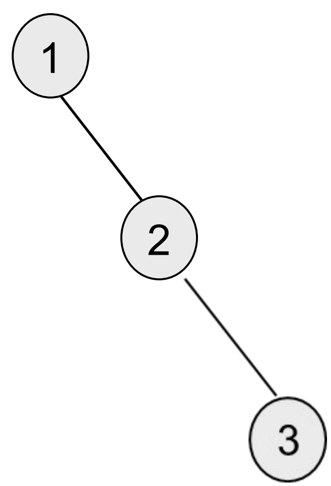
Example

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.
- Answer:



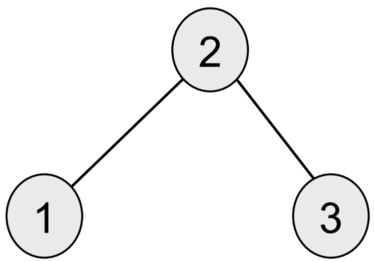
Example

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.
- Answer:



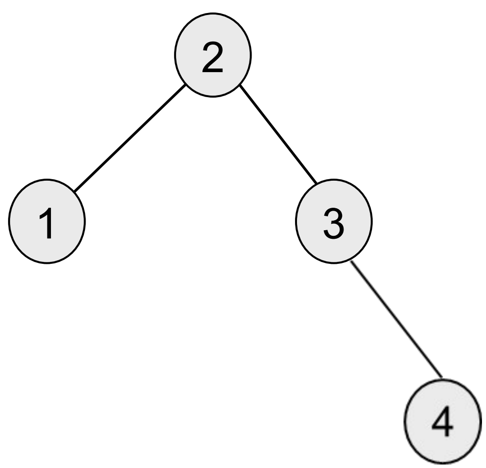
Example

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.



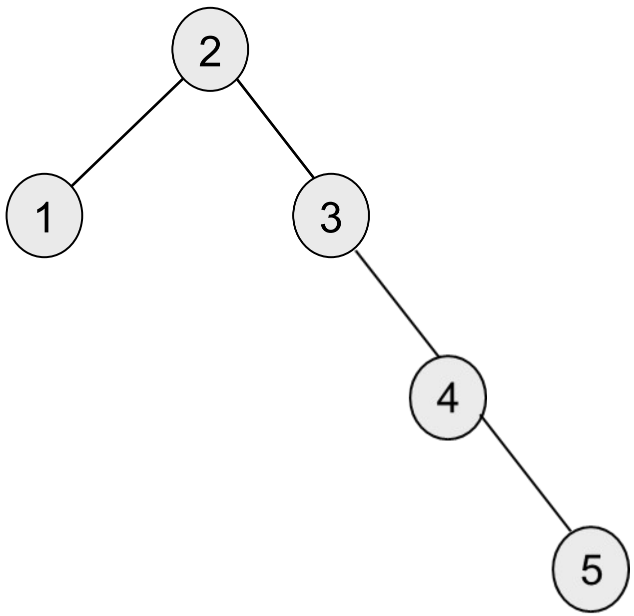
Example

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.
- Answer:



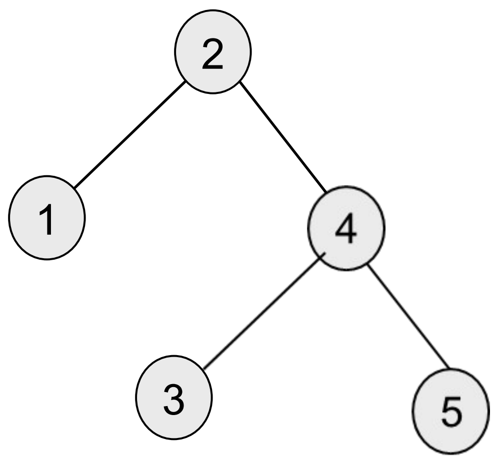
Example

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.



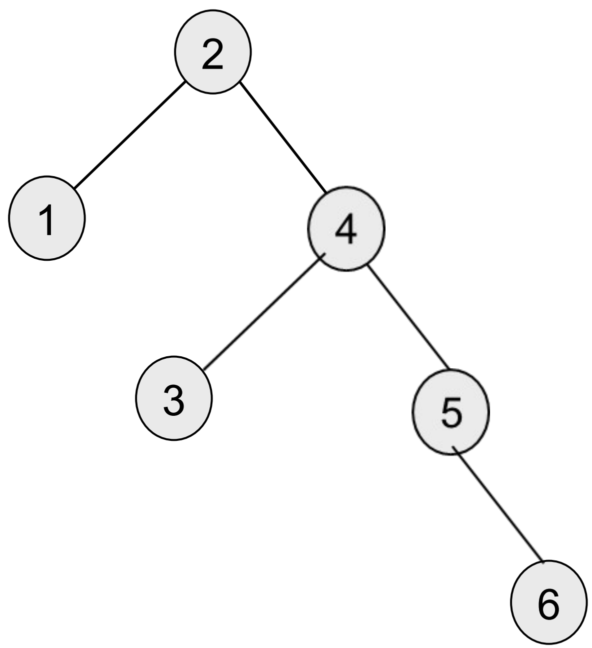
Example

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.



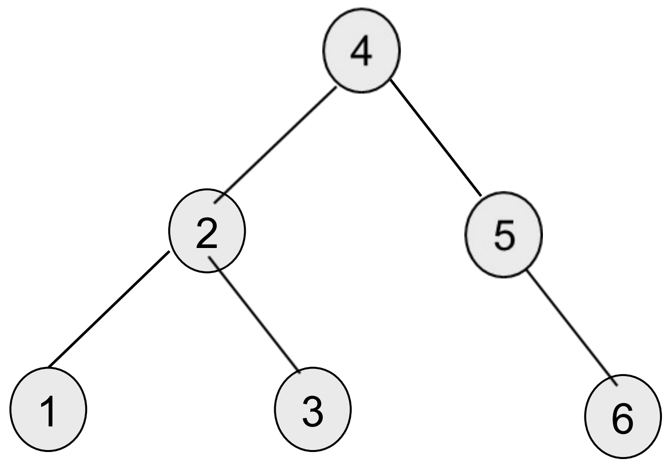
Example

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.



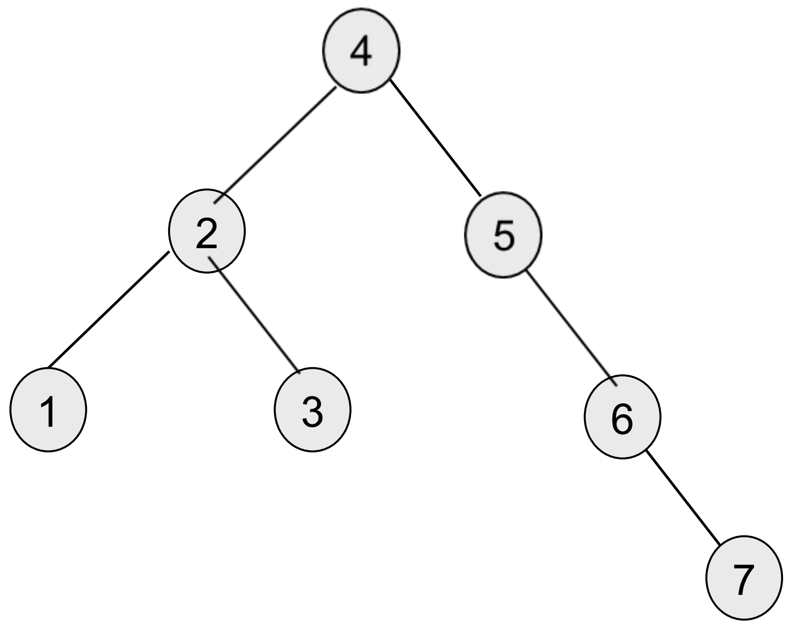
Example

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.



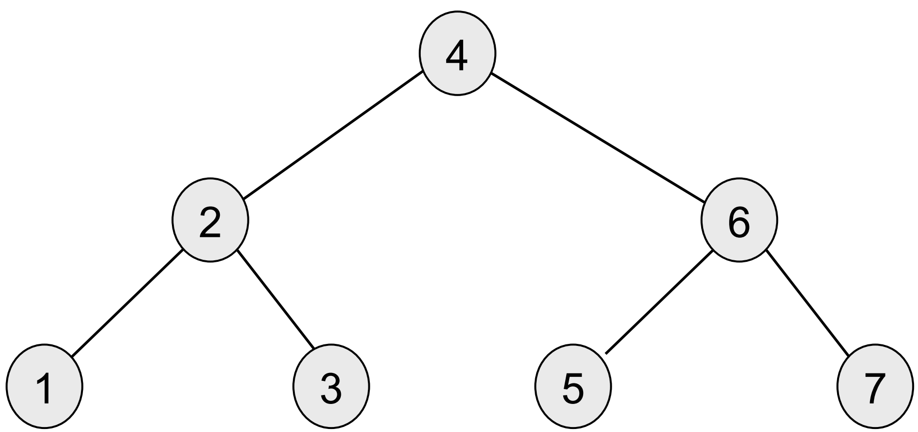
Example

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.



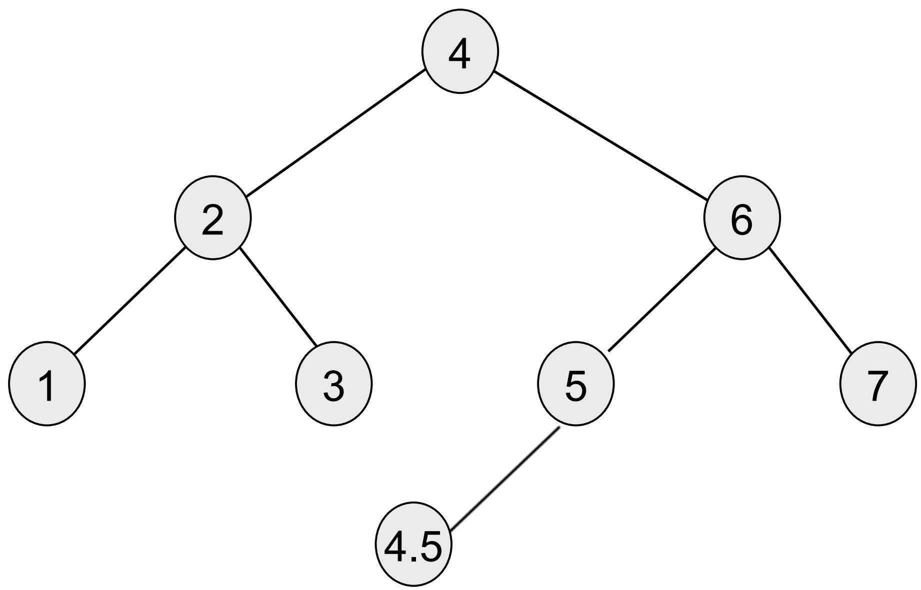
Example

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.
- Answer:



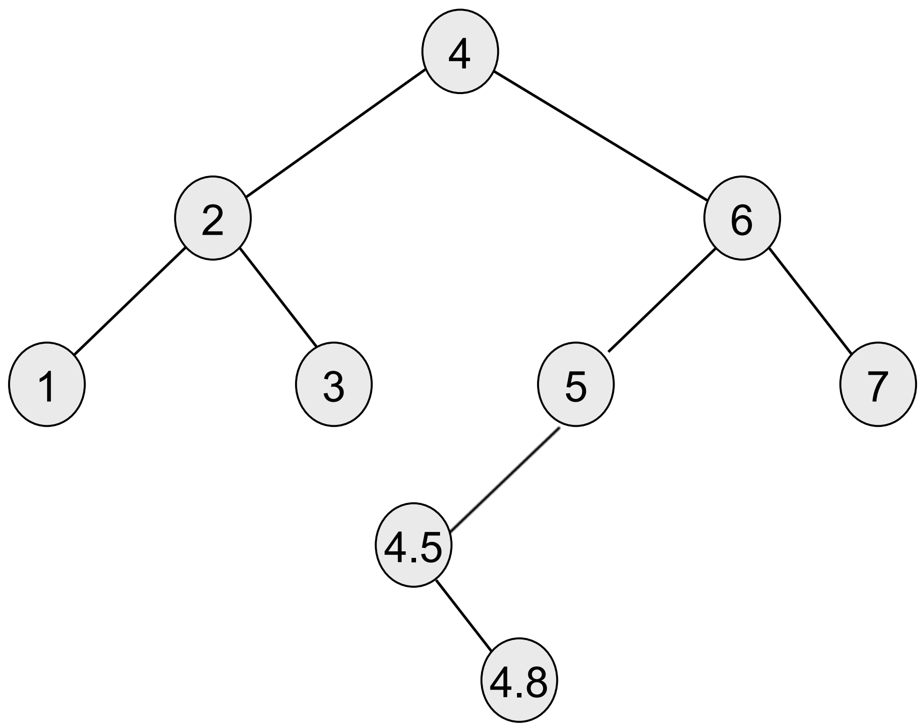
Example

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.



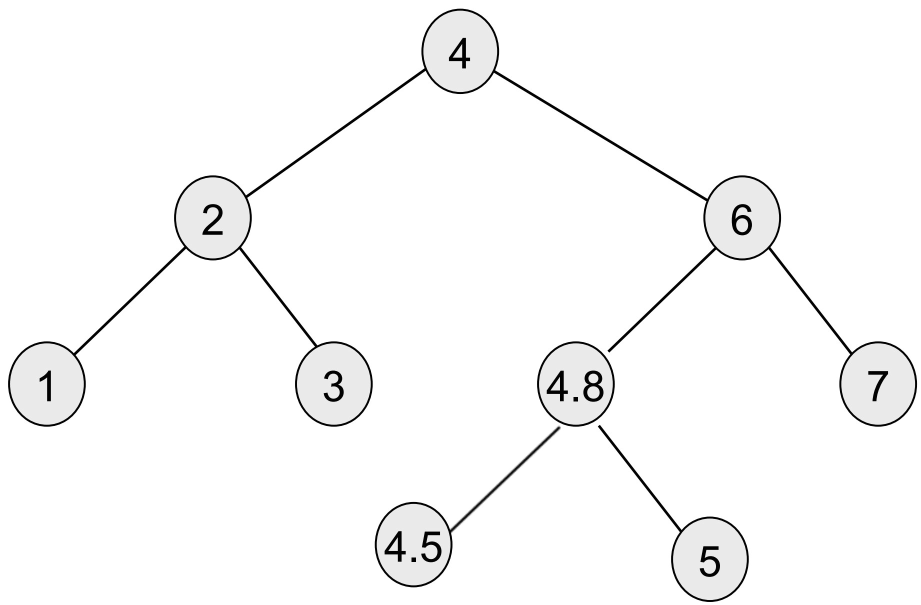
Example

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.



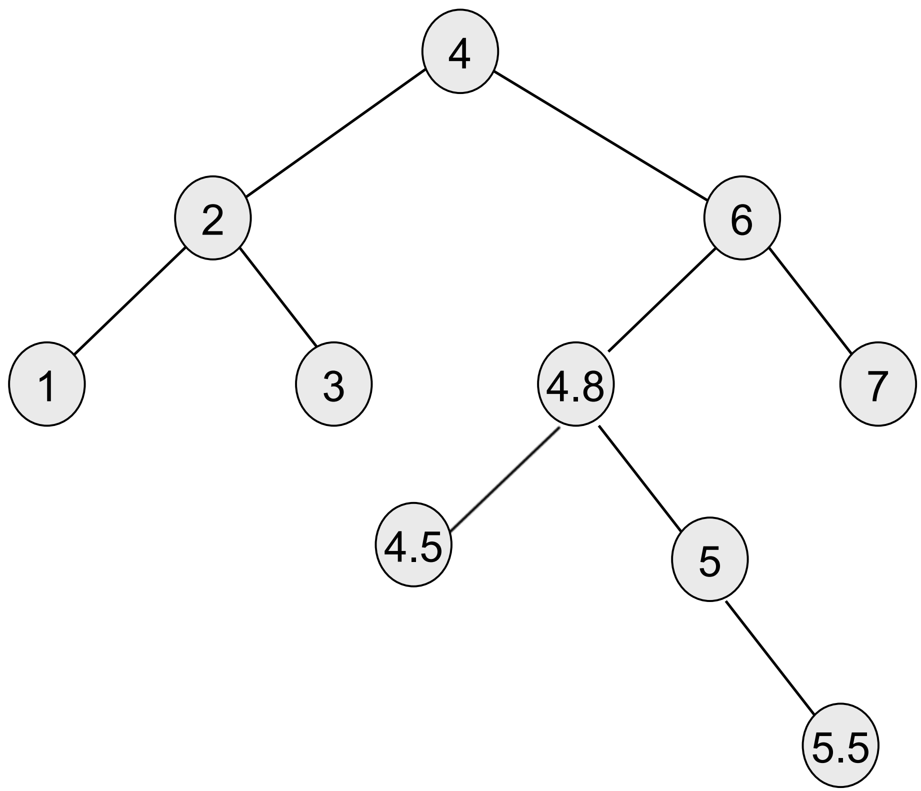
Example

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.



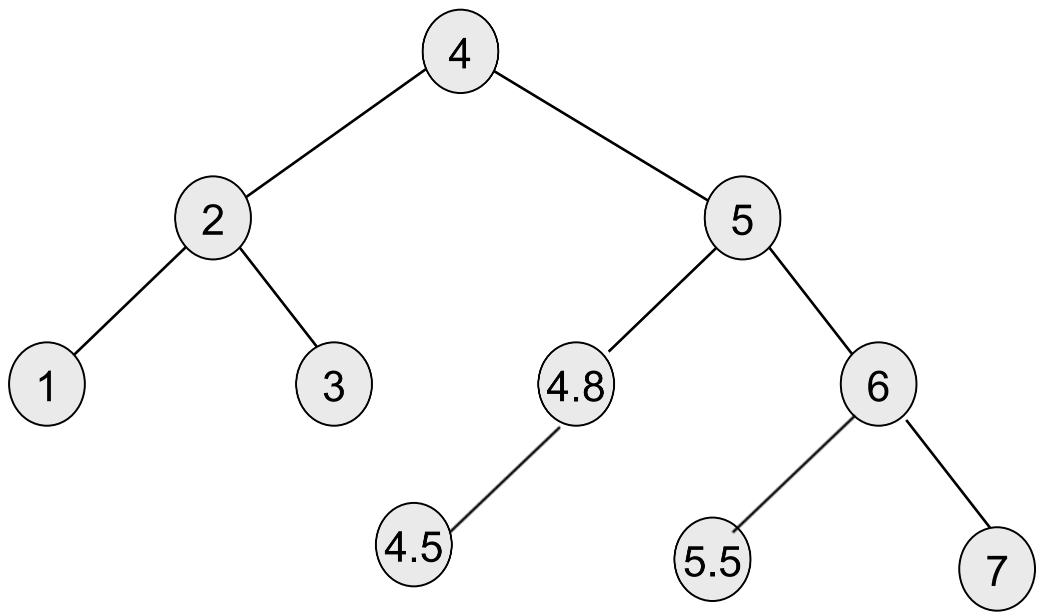
Example

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.



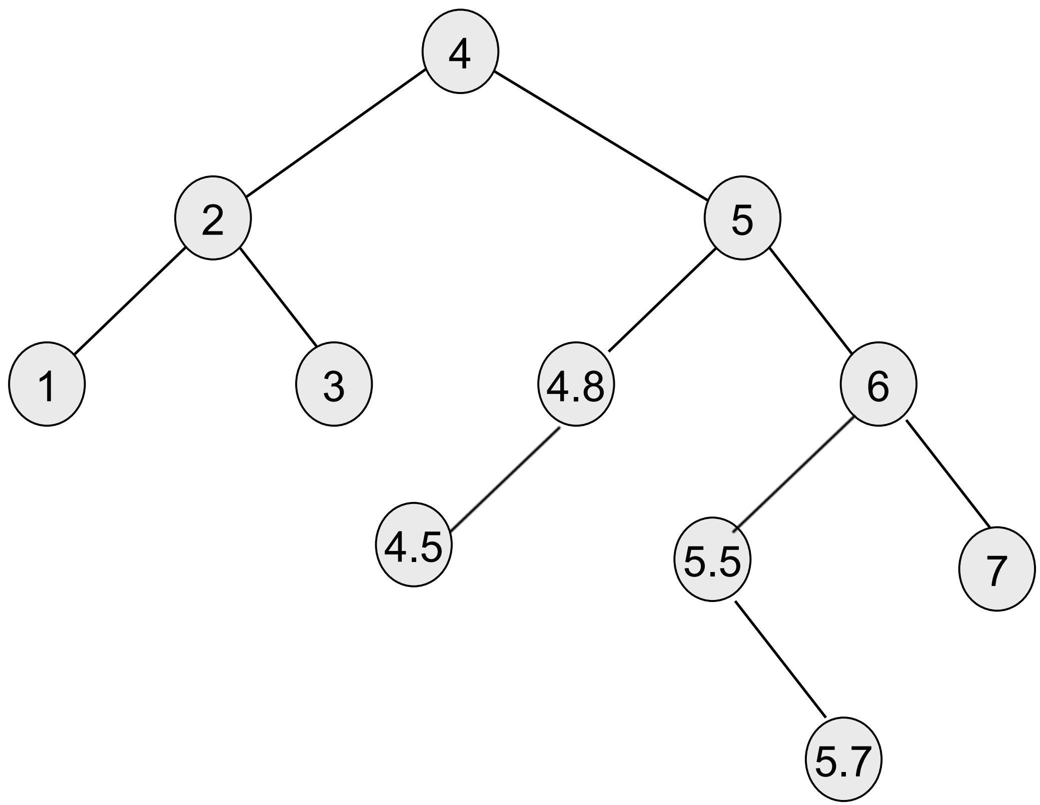
Example

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.



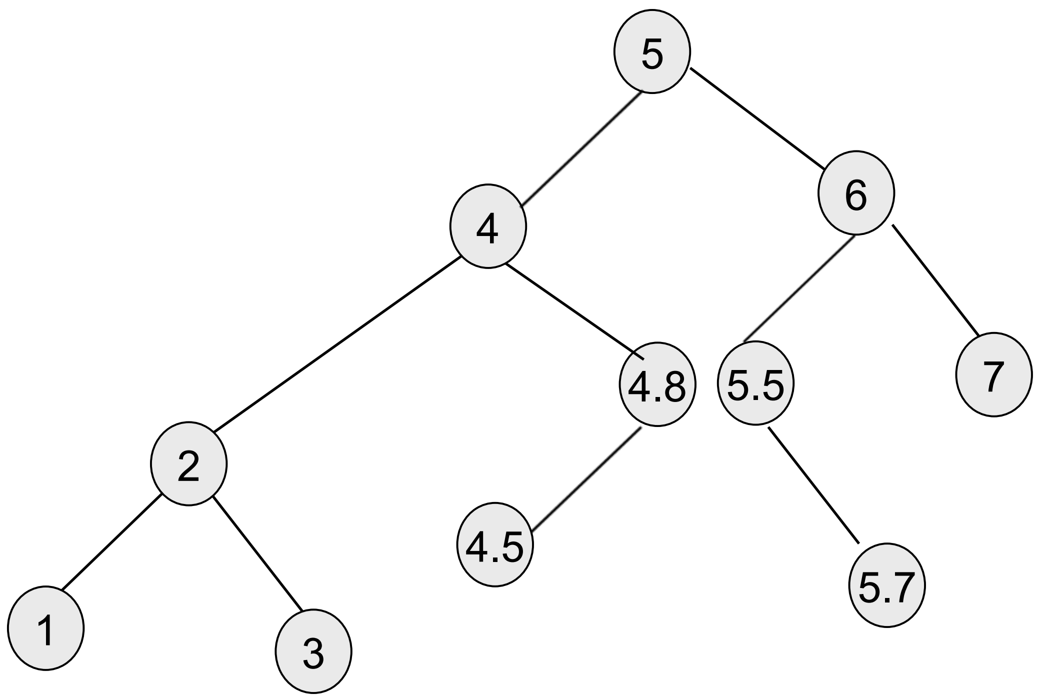
Example

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.



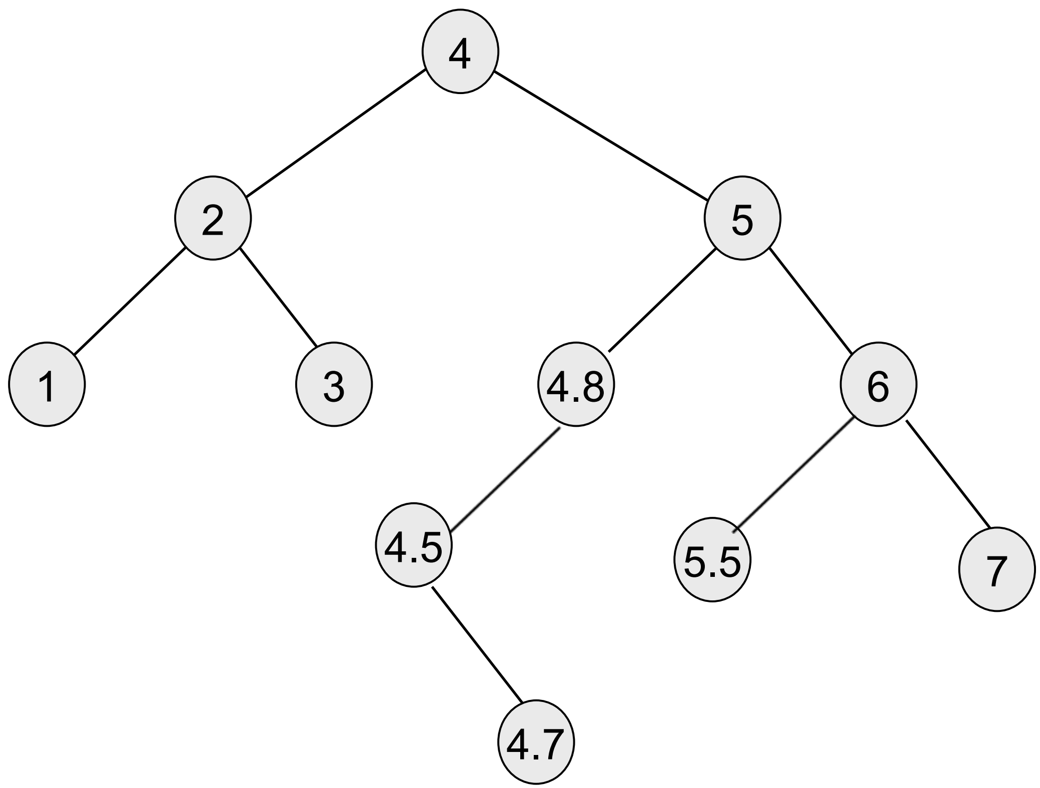
Example

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.



Example

- Start with an empty AVL tree and insert the items 3,2,1, and then 4 through 7 in sequential order.



Analysis

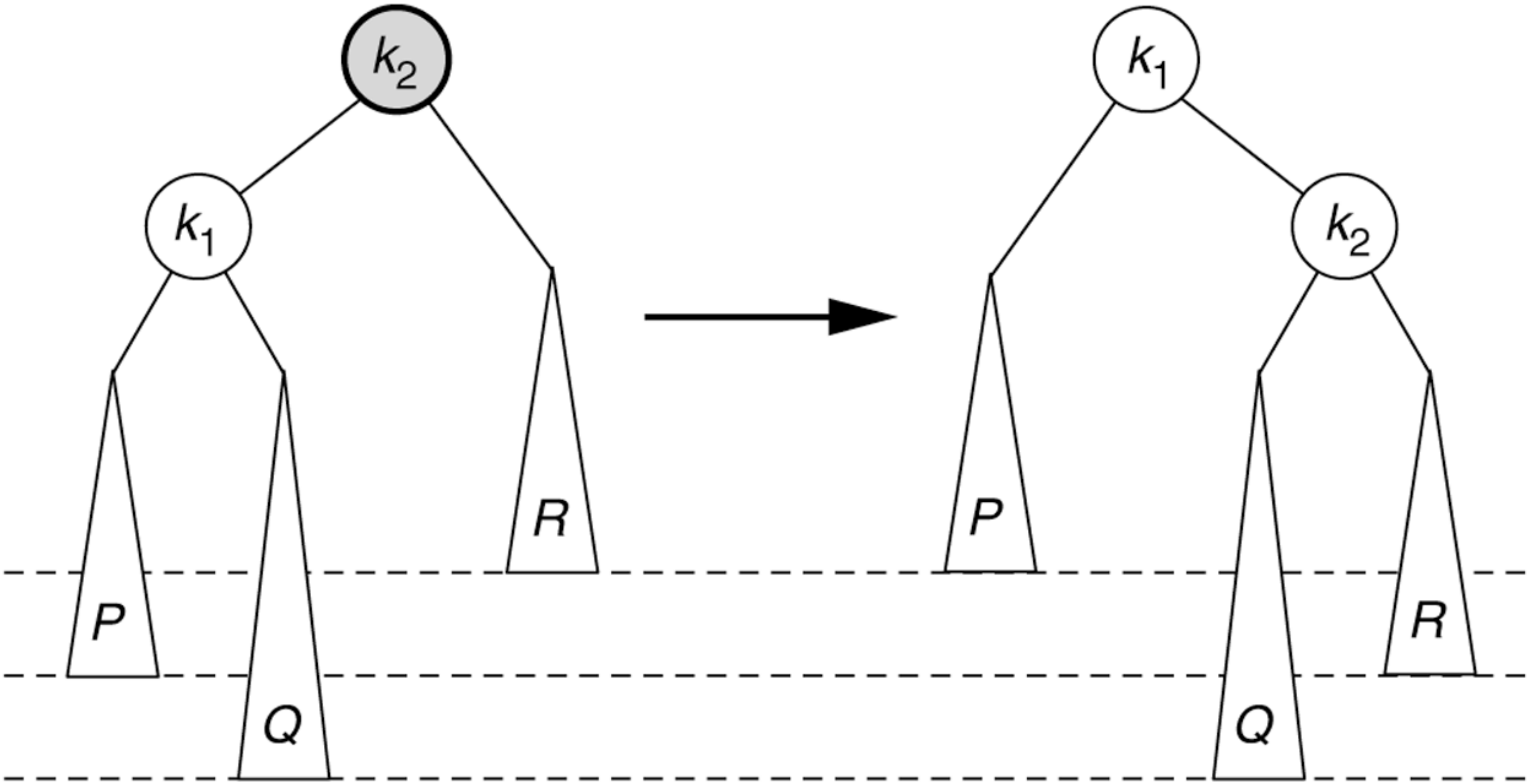
- One rotation suffices to fix cases 1 and 4.
- Single rotation preserves the original height:
  - The new height of the entire subtree is exactly the same as the height of the original subtree before the insertion.

- 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 (2 and 3).
- These cases require a *double* rotation, involving three nodes and four subtrees.

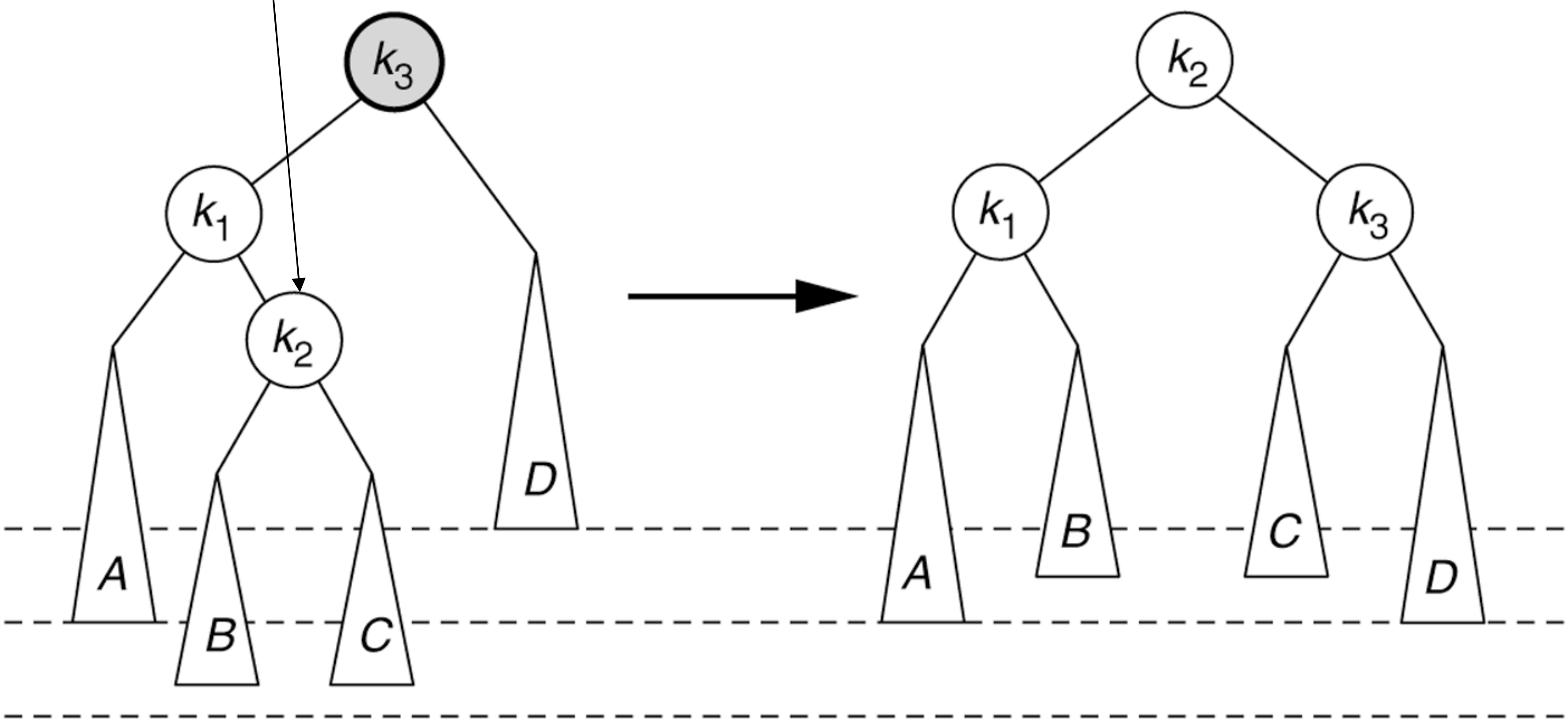
Figure 19.28
Single rotation **does not** fix case 2.



(a) Before rotation (b) After rotation

Left–right double rotation to fix case 2

Lift this up:
first rotate left between (k_1,k_2),
then rotate right betwen (k_3,k_2)



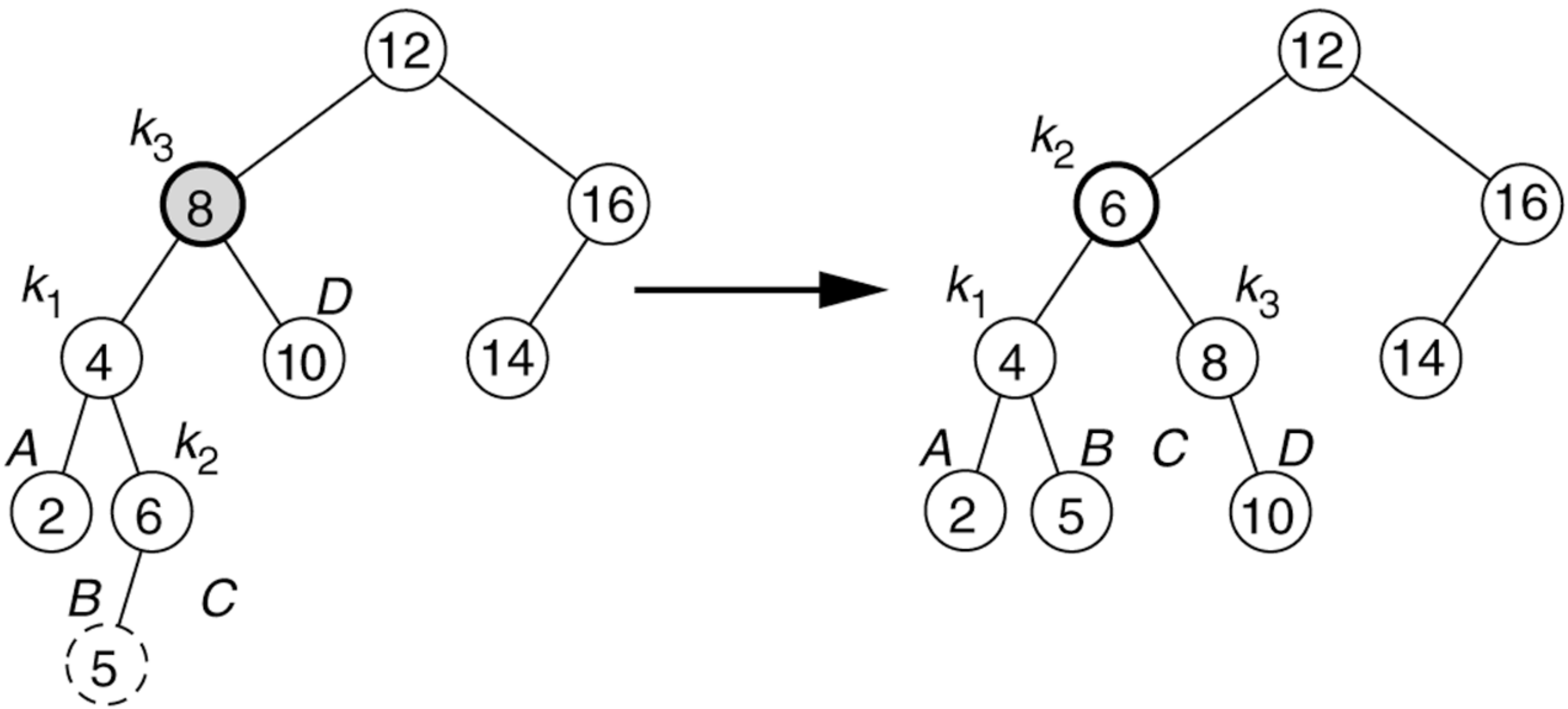
(a) Before rotation (b) After rotation

Left-Right Double Rotation

- A left-right double rotation is equivalent to a sequence of two single rotations:
  - 1st rotation on the original tree:
    a *left* rotation between X's left-child and grandchild
  - 2nd rotation on the new tree:
    a *right* rotation between X and its new left child.

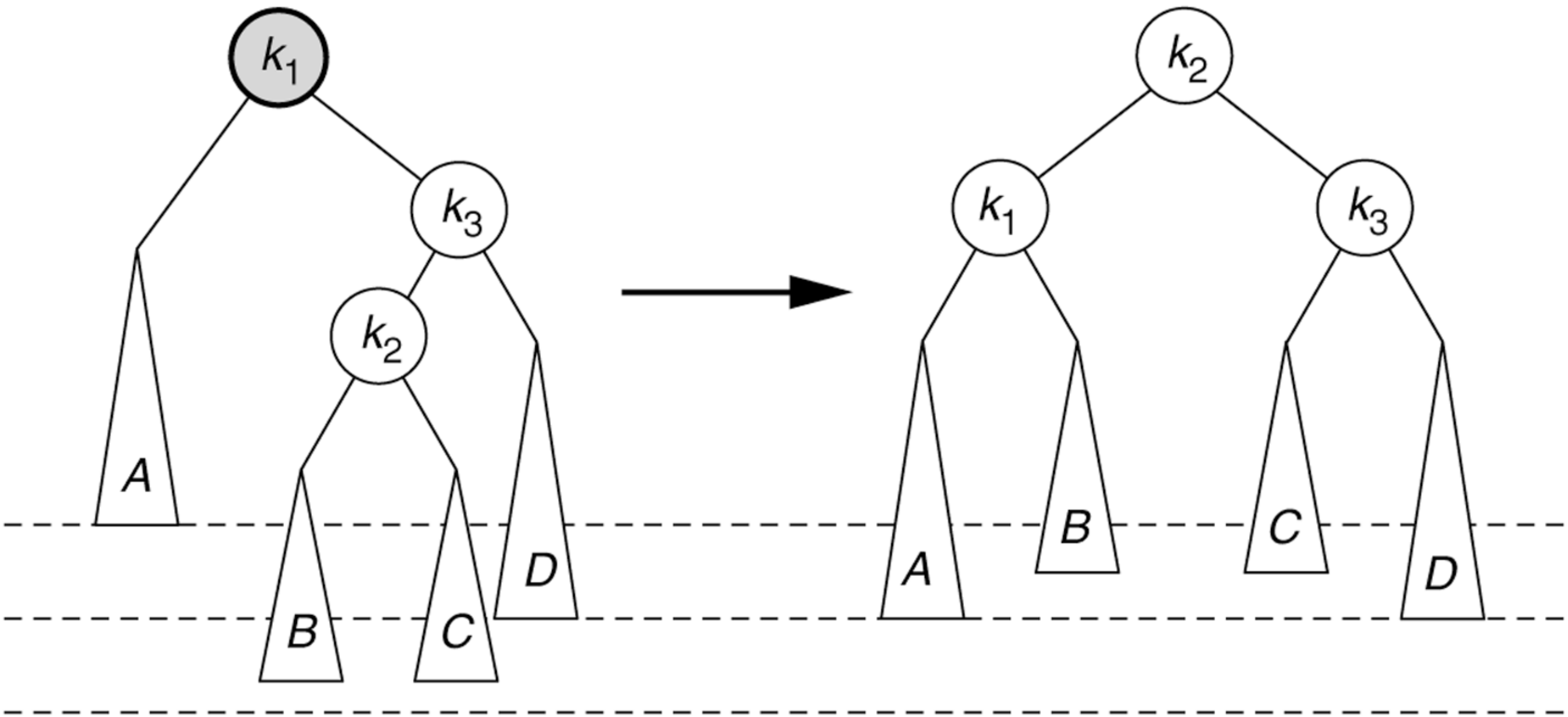
Figure 19.30

Double rotation fixes AVL tree after the insertion of 5.



(a) Before rotation (b) After rotation

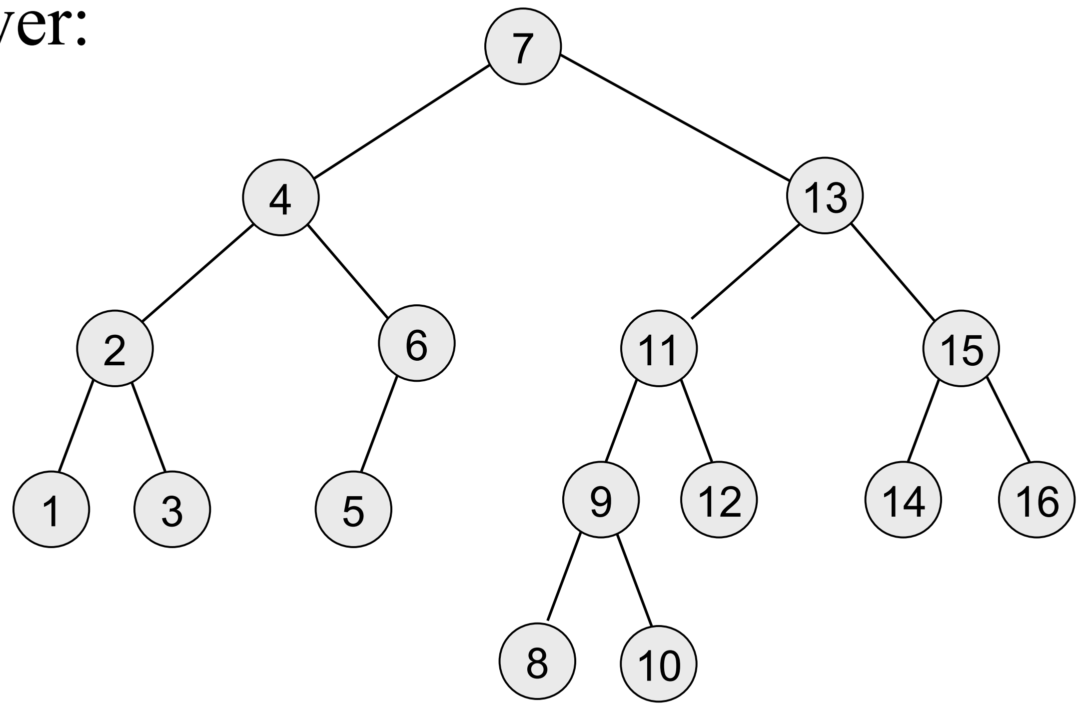
Right–Left double rotation to fix case 3.



(a) Before rotation (b) After rotation

Example

- Insert 16, 15, 14, 13, 12, 11, 10, and 8, and 9 to the previous tree obtained in the previous single rotation example.
- Answer:



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;
}
```

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<Compa
  rable> * & k3 ) const
{
   rotateWithRightChild( k3->left );
   rotateWithLeftChild( k3 );
}
```

```
/* 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
   ) const
{
   if( t == NULL )
     t = new AvlNode<Comparable>( x, NULL, NULL );
   else if( x < t->element )
   {
     insert( x, t->left );
     if( height( t->left ) - height( t->right ) == 2 )
       if( x < t->left->element )
           rotateWithLeftChild( t );
       else
           doubleWithLeftChild( t );
   }
   else if( t->element < x )
   {
       insert( x, t->right );
       if( height( t->right ) - height( t->left ) == 2 )
          if( t->right->element < x )
             rotateWithRightChild( t );
          else
             doubleWithRightChild( t );
    }
    else
       ;  // Duplicate; do nothing
    t->height = max( height( t->left ), height( t->right ) ) + 1;
}
```

AVL Tree -- Deletion

- Deletion is more complicated.
- We may need more than one rebalance on the path from deleted node to root.
- Deletion is O(logN)

Deletion of a Node

- Deletion of a node x from an AVL tree requires the same basic ideas, including single and double rotations, that are used for insertion.
- With each node of the AVL tree is associated a ***balance factor*** that is left high, equal or right high according, respectively, as the left subtree has height greater than, equal to, or less than that of the right subtree.

Method

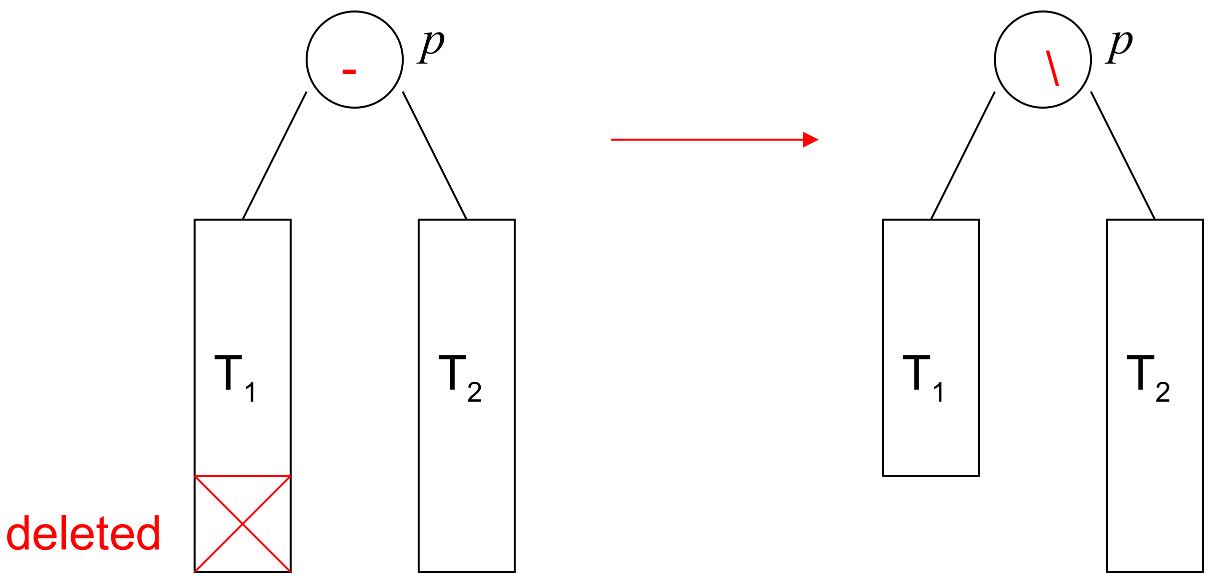
1. Reduce the problem to the case when the node *x* to be deleted has at most one child.
   – If *x* has two children replace it with its immediate predecessor *y* under inorder traversal (the immediate successor would be just as good)
   – Delete *y* from its original position, by proceeding as follows, using *y* in place of *x* in each of the following steps.

Method (cont.)

2. Delete the node *x* from the tree.
   - We'll trace the effects of this change on height through all the nodes on the path from *x* back to the root.
   - We use a Boolean variable `shorter` to show if the height of a subtree has been shortened.
   - The action to be taken at each node depends on
     - the value of `shorter`
     - balance factor of the node
     - sometimes the balance factor of a child of the node.
3. `shorter` is initially true. The following steps are to be done for each node *p* on the path from the parent of x to the root, provided shorter remains true. When shorter becomes false, the algorithm terminates.

Case 1

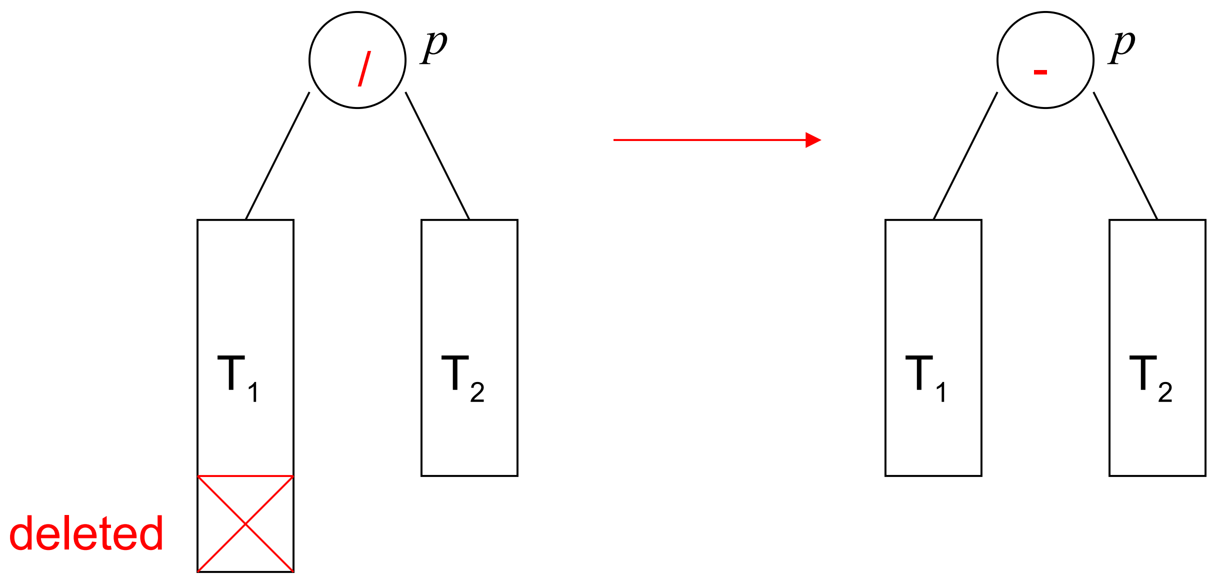
4. *Case* 1: The current node *p* has balance factor equal.
   – Change the balance factor of *p*.
   – shorter becomes false



- No rotations
- Height unchanged

Case 2

5. *Case* 2: The balance factor of *p* is not equal and the taller subtree was shortened.
   – Change the balance factor of *p* to equal
   – Leave shorter true.



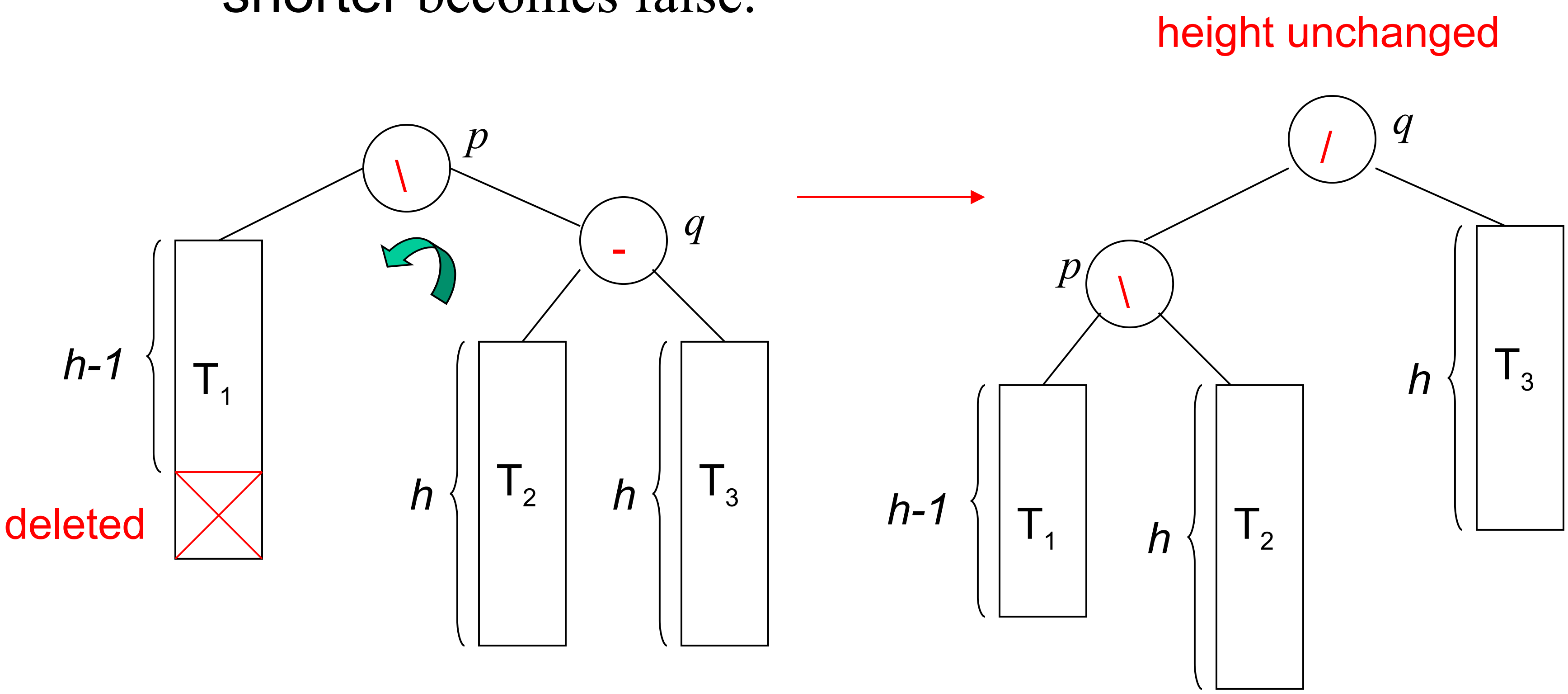
- No rotations
- Height reduced

Case 3

6. *Case* 3: The balance factor of p is not equal, and the shorter subtree was shortened.
   - Rotation is needed.
   - Let q be the root of the taller subtree of p. We have three cases according to the balance factor of q.

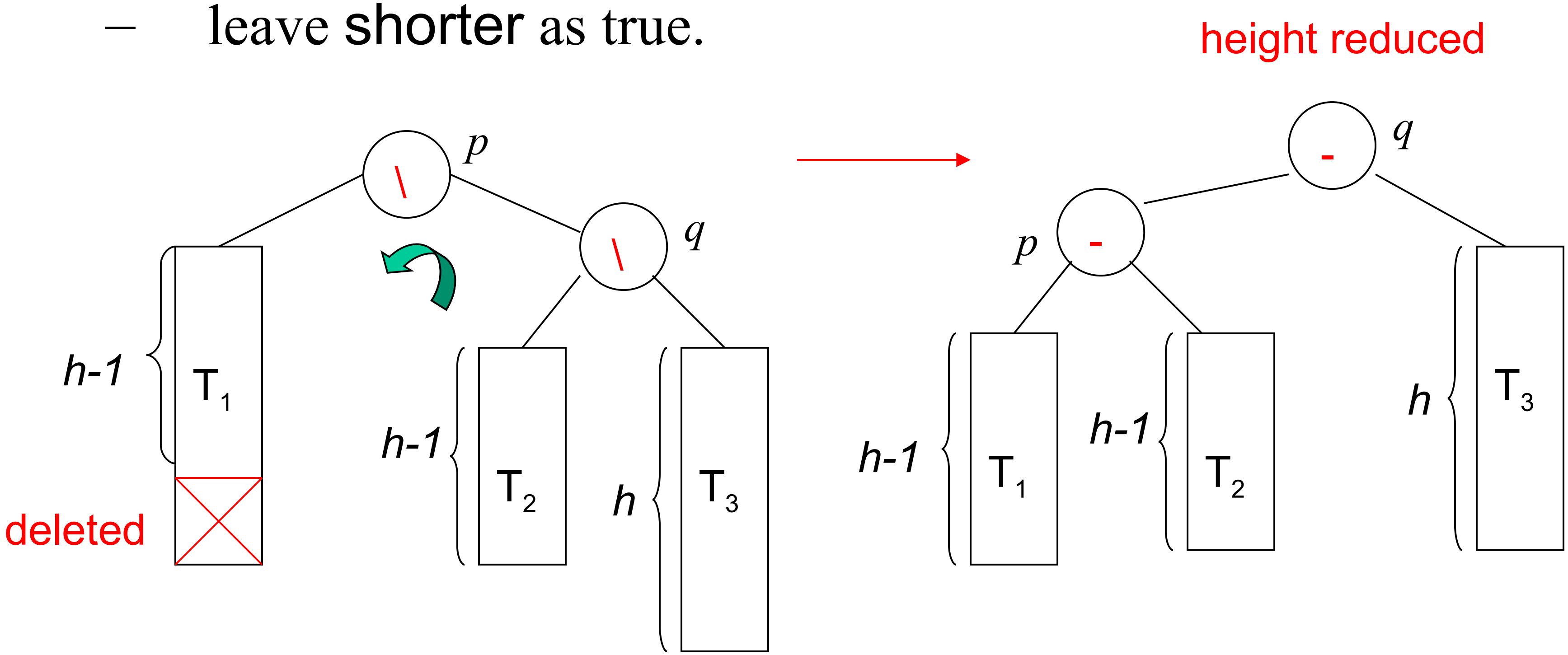
Case 3a

7. *Case 3a*: The balance factor of *q* is equal.
   - Apply a single rotation
   - shorter becomes false.



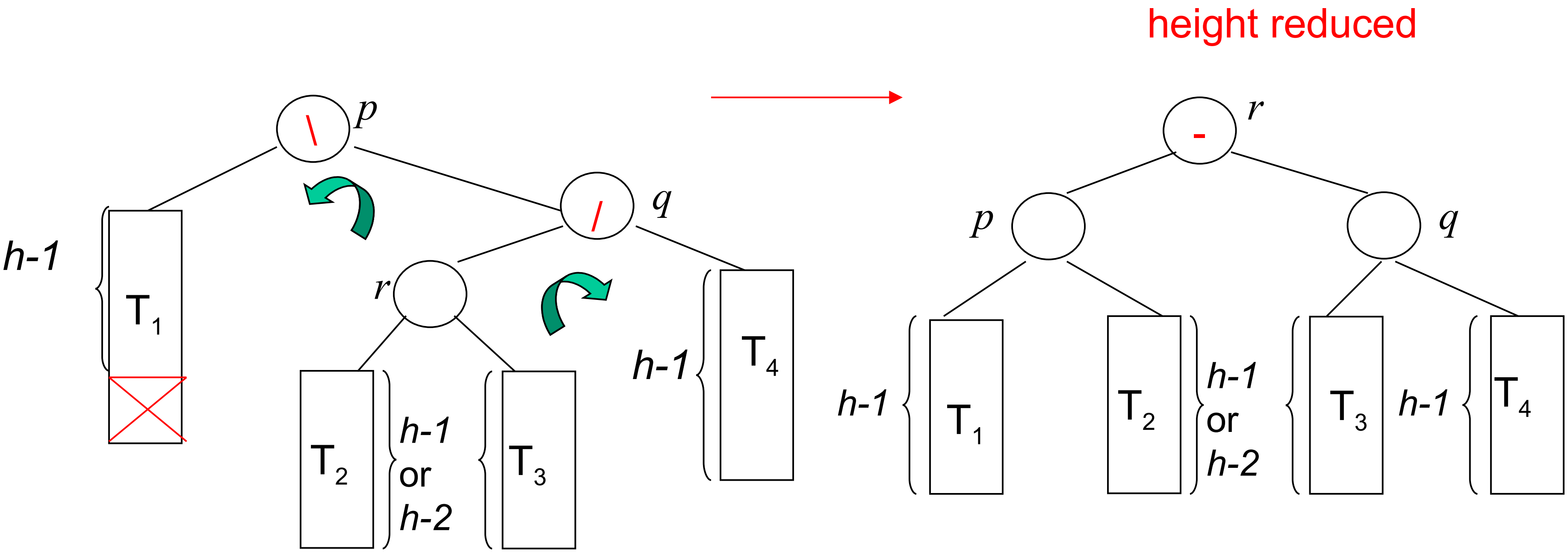
Case 3b

8. *Case 3b*: The balance factor of *q* is the same as that of p.
   – Apply a single rotation
   – Set the balance factors of *p* and *q* to equal
   – leave shorter as true.



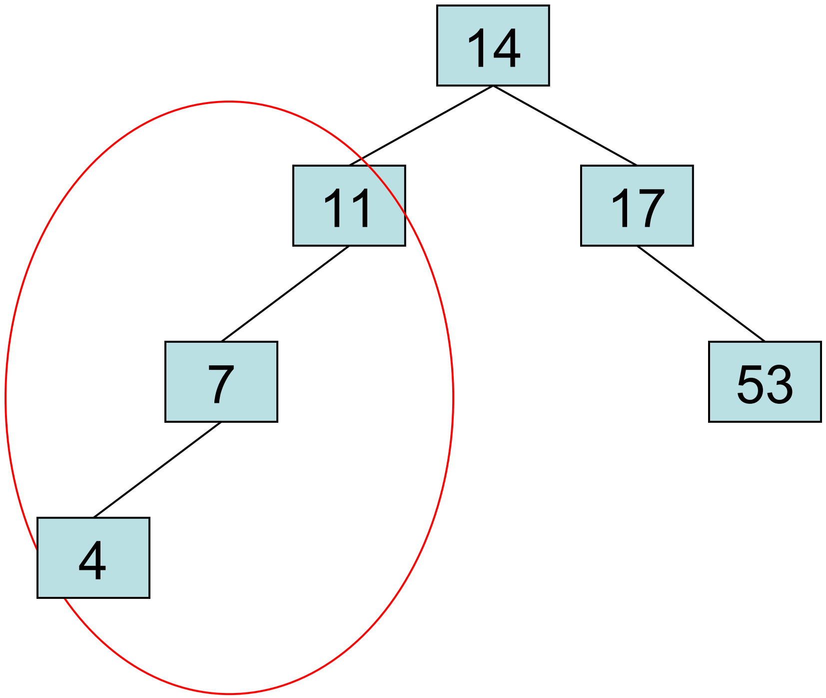
Case 3c

9. *Case* 3*c*: The balance factors of *p* and *q* are opposite.
   - Apply a double rotation
   - set the balance factors of the new root to equal
   - leave shorter as true.



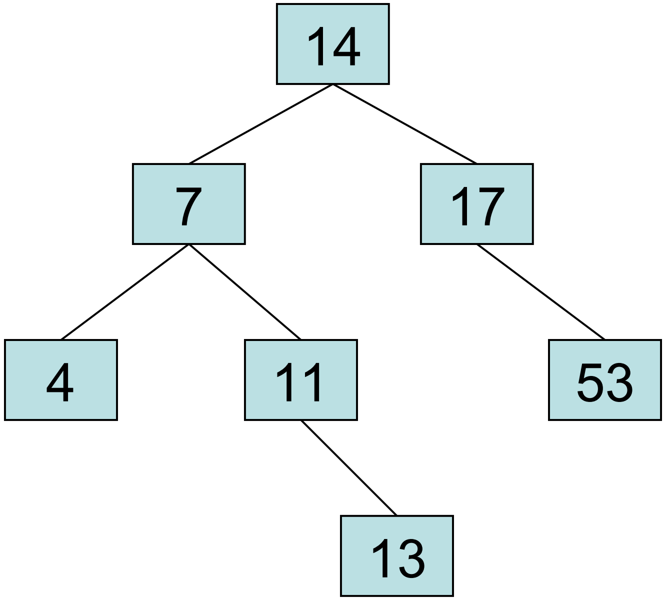
AVL Tree Example:

- **Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree**



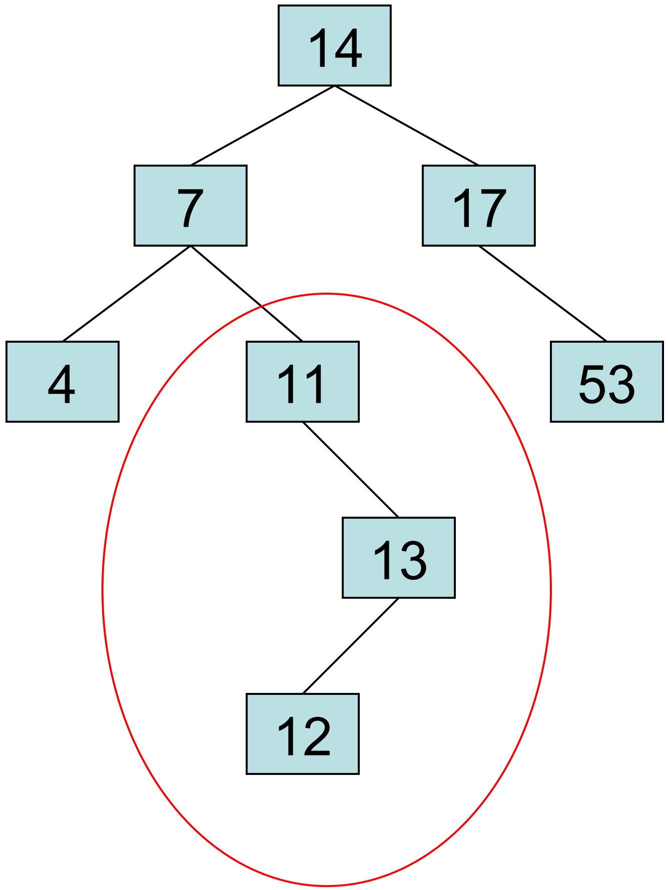
AVL Tree Example:

- **Insert 14, 17, 11, 7, 53, 4, 13 into an empty AVL tree**



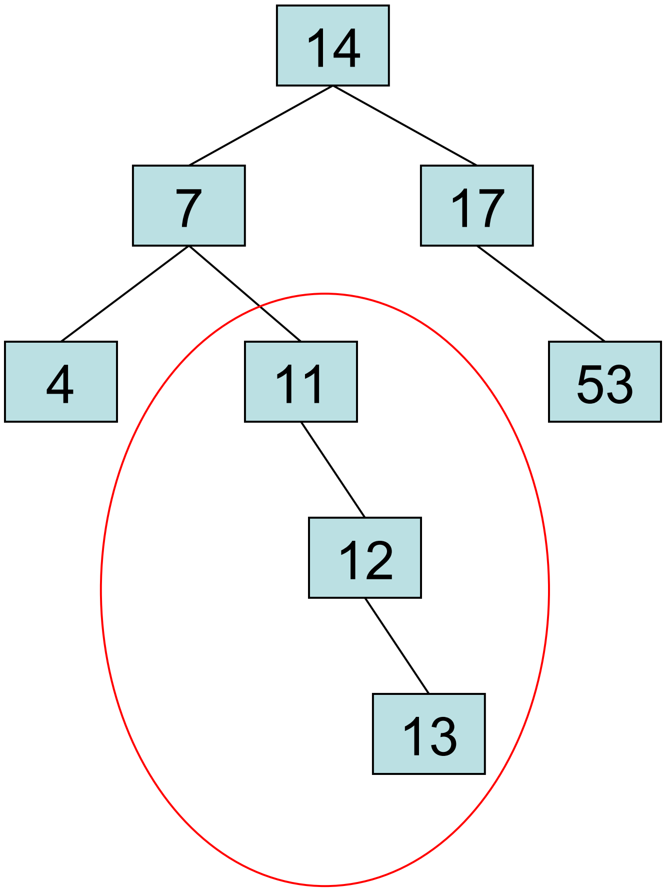
AVL Tree Example:

- **Now insert 12**



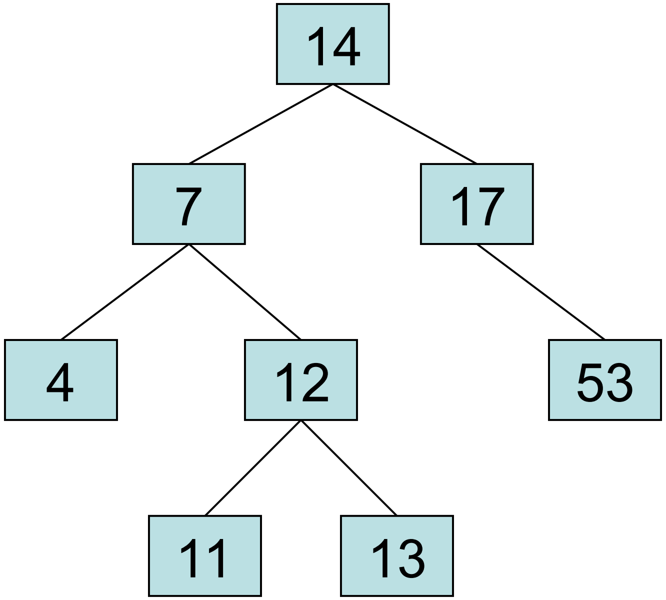
AVL Tree Example:

- **Now insert 12**



AVL Tree Example:

- **Now the AVL tree is balanced.**



- Node { int val;
- Node lchild;
- Node rchild;
- Node Parent;
- int ht;
- }
- insertAVL(root, Node n) {
- n = insertBST(root, n); flag = false;
- par1 = n.parent;
- while(par1 !=null || flag == false) {
- leftht = getleftht(par1);
- rtht = getrightht(par1);
- if (abs(leftht – rht) == 2) {
- avlBalance(par1,leftht, rtht);
- flag = true;
- }
- updateHt(par1, leftht, rtht);   par1 = par1.parent;
- }

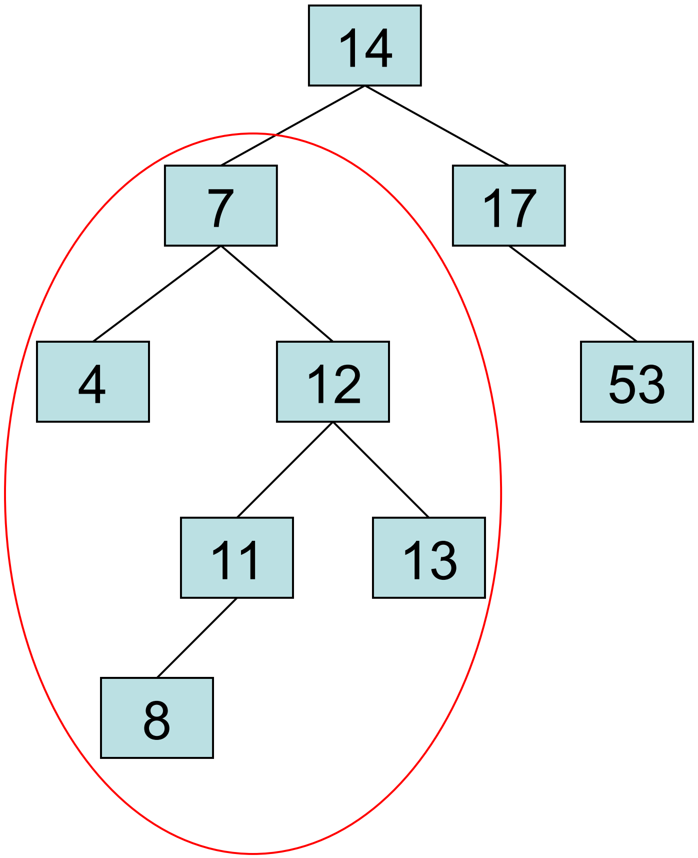
- avlBalance(Node n1, ltht, rtht) {
-     if (ltht > rtht) {
-         n2 = n1.left
-     }
-     else {
-         n2 = n1.right;
-     }
-     lltht3 = getleftht(n2);
-     rtht3 = getrightht(n2);
-     if (ltht3 > rtht3) {
-         n3 = n2.left;
-     }
-     else {
-         n3 = n2.right;
-     }
-

- If (n3 == n2.left && n2 == n1.left) {
-     balanceleftStline(n1,n2,n3);
- }
- If (n3 == n2.right && n2 == n1.right) {
-     balancerightStline(n1, n2, n3);
- }
- If (n3 == n2.left && n2 == n1.right) {
-     balancezigzagright(n1, n2, n3);
- }
- If (n3 == n2.right && n2 == n1.left) {
-     balancezigagleft(n1, n2, n3);
- }

- Balanceleftstline(n1, n2, n3) {
-     tmp1 = n1.lchild;
-     tmp2 = n1.parent;
-     n1.left = n2.right;
-     n1.parent = n2;
-     n2.parent = tmp2;
-     n2.right = n1;
-     tmp2.left = n2;
-     n2.right.parent = n1;
-

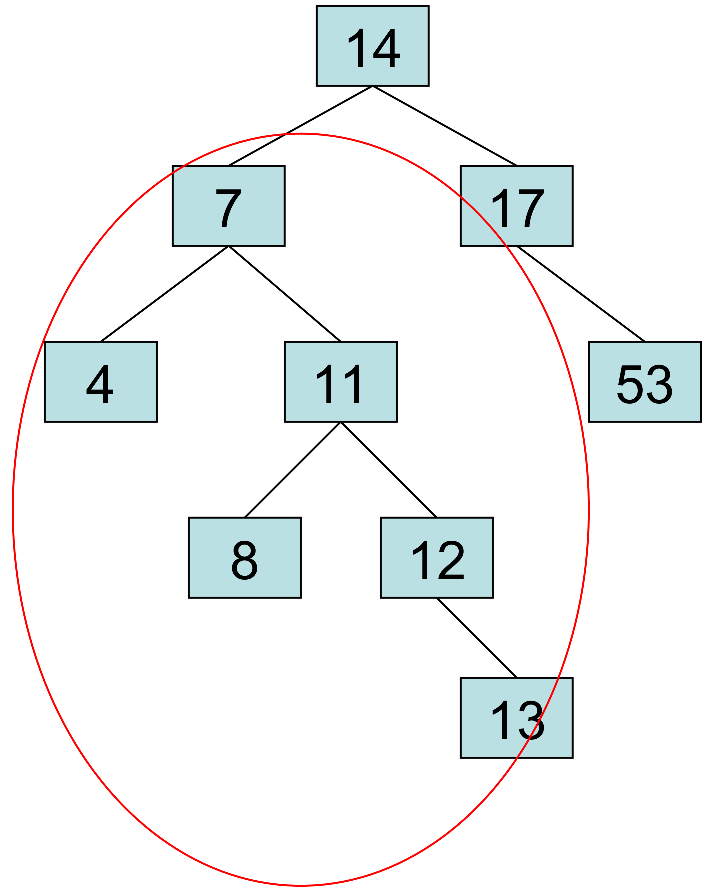
AVL Tree Example:

- **Now insert 8**



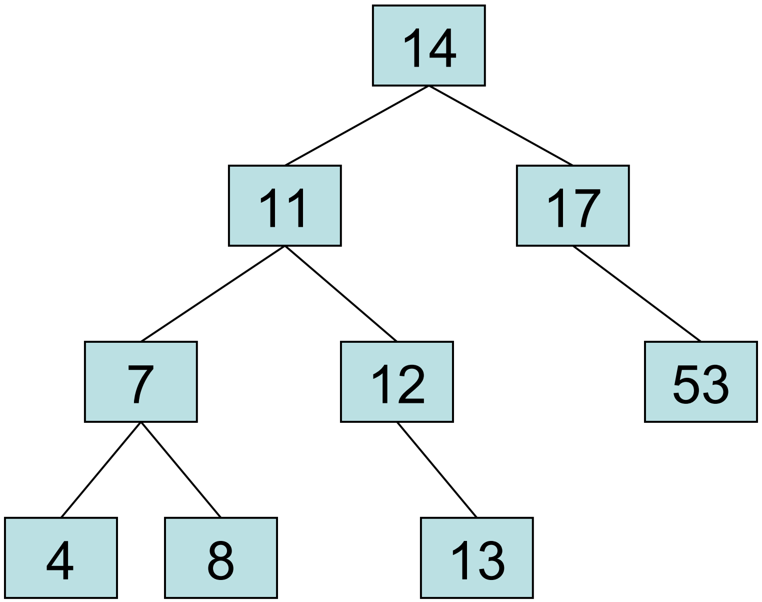
AVL Tree Example:

- **Now insert 8**



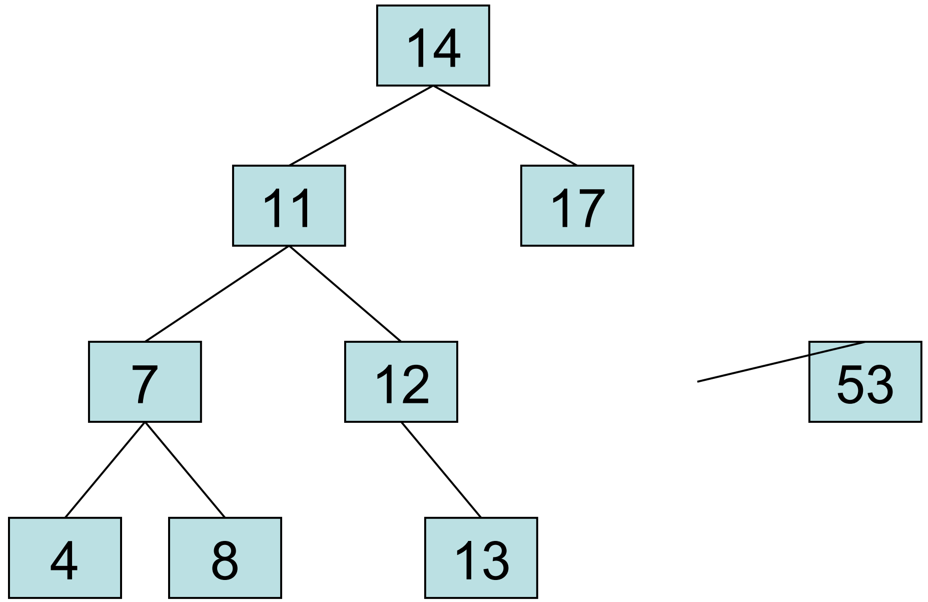
AVL Tree Example:

- **Now the AVL tree is balanced.**



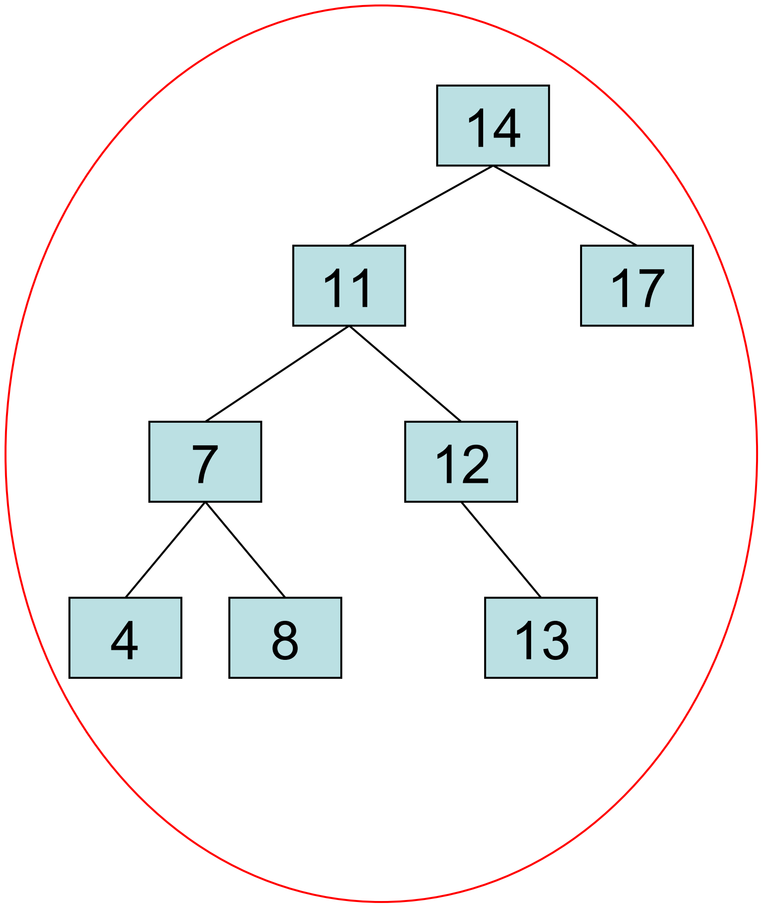
AVL Tree Example:

- **Now remove 53**



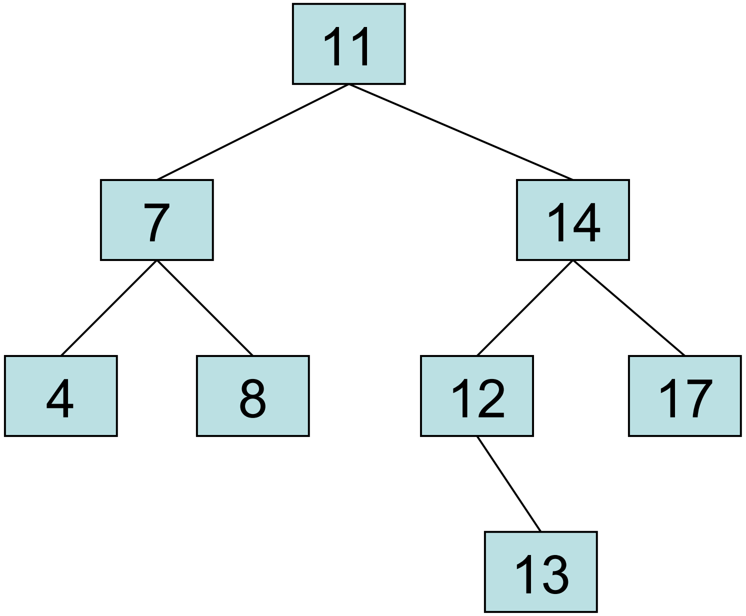
AVL Tree Example:

- **Now remove 53, unbalanced**



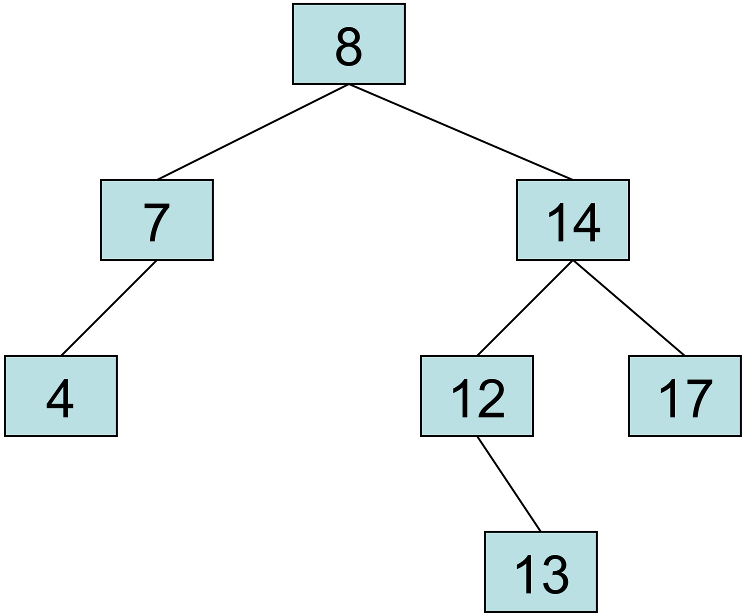
AVL Tree Example:

- **Balanced!   Remove 11**



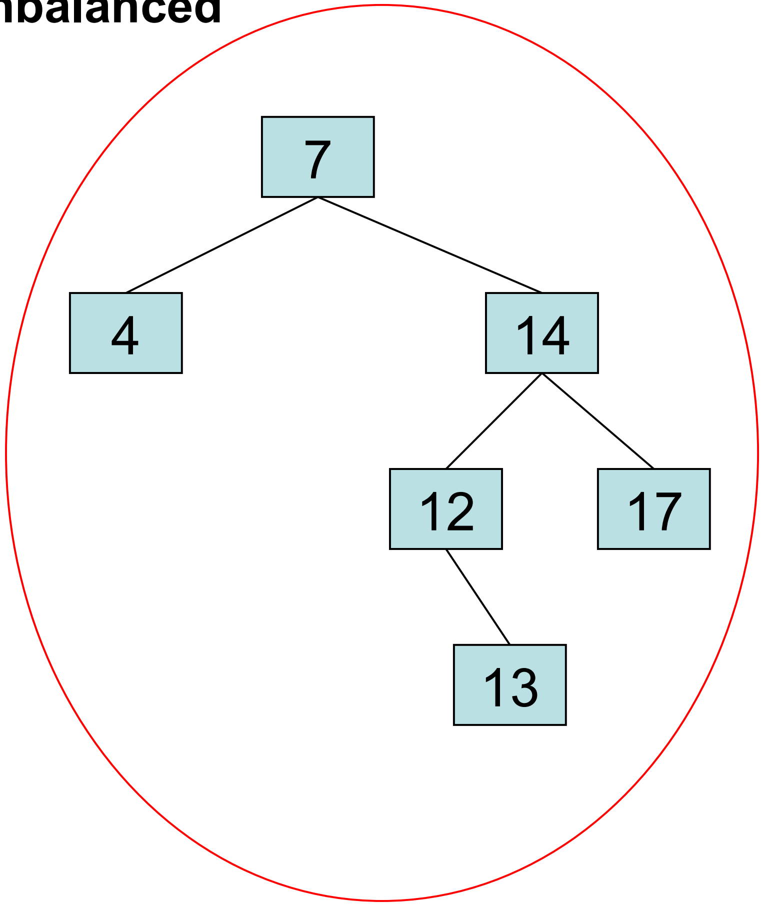
AVL Tree Example:

- **Remove 11, replace it with the largest in its left branch**



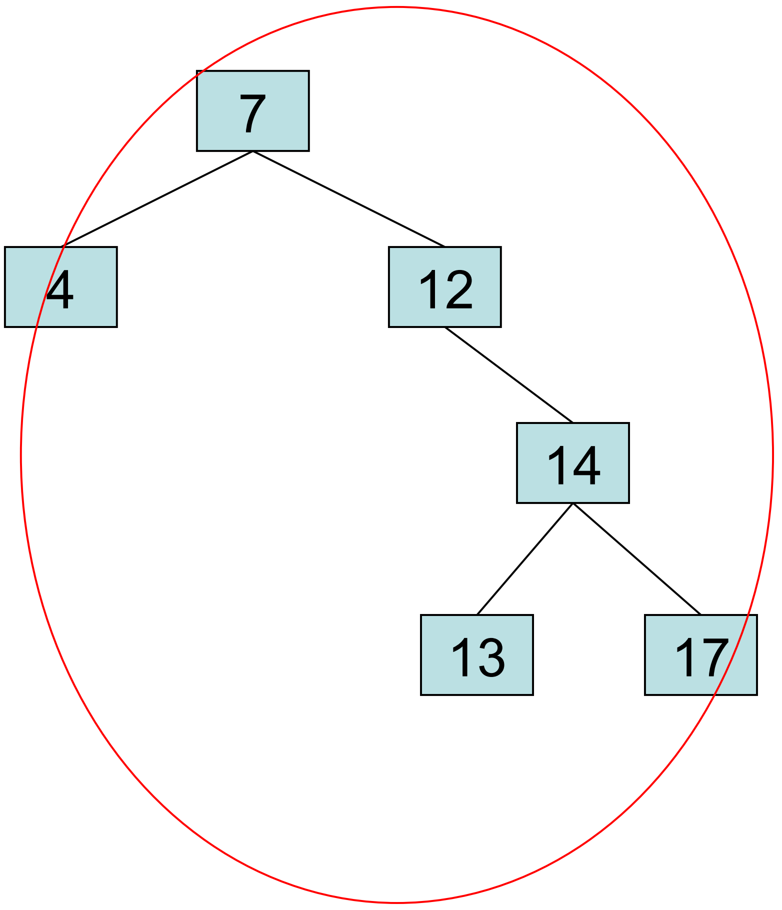
AVL Tree Example:

- **Remove 8, unbalanced**



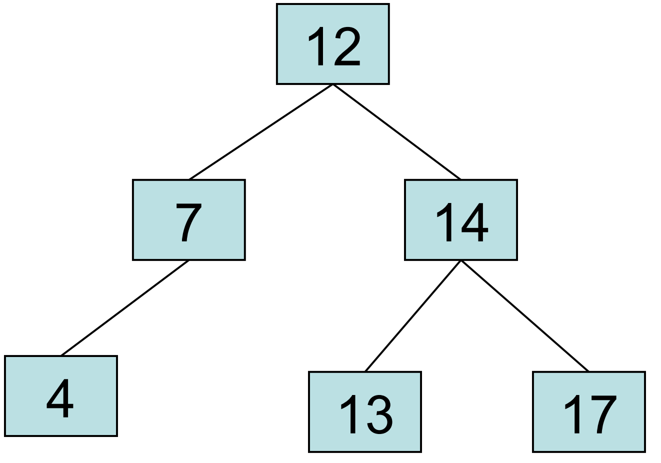
AVL Tree Example:

- **Remove 8, unbalanced**



AVL Tree Example:

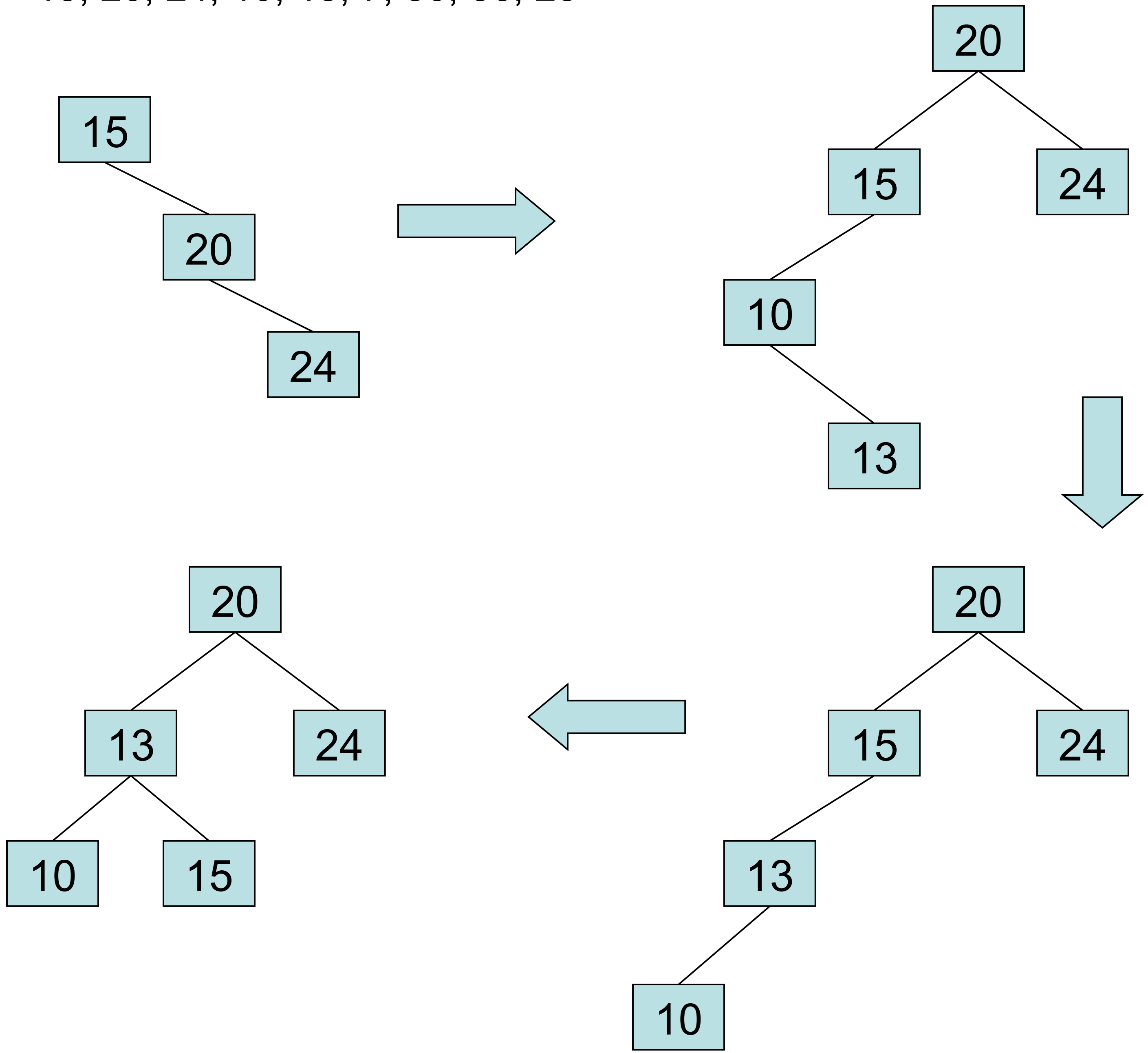
- **Balanced!!**



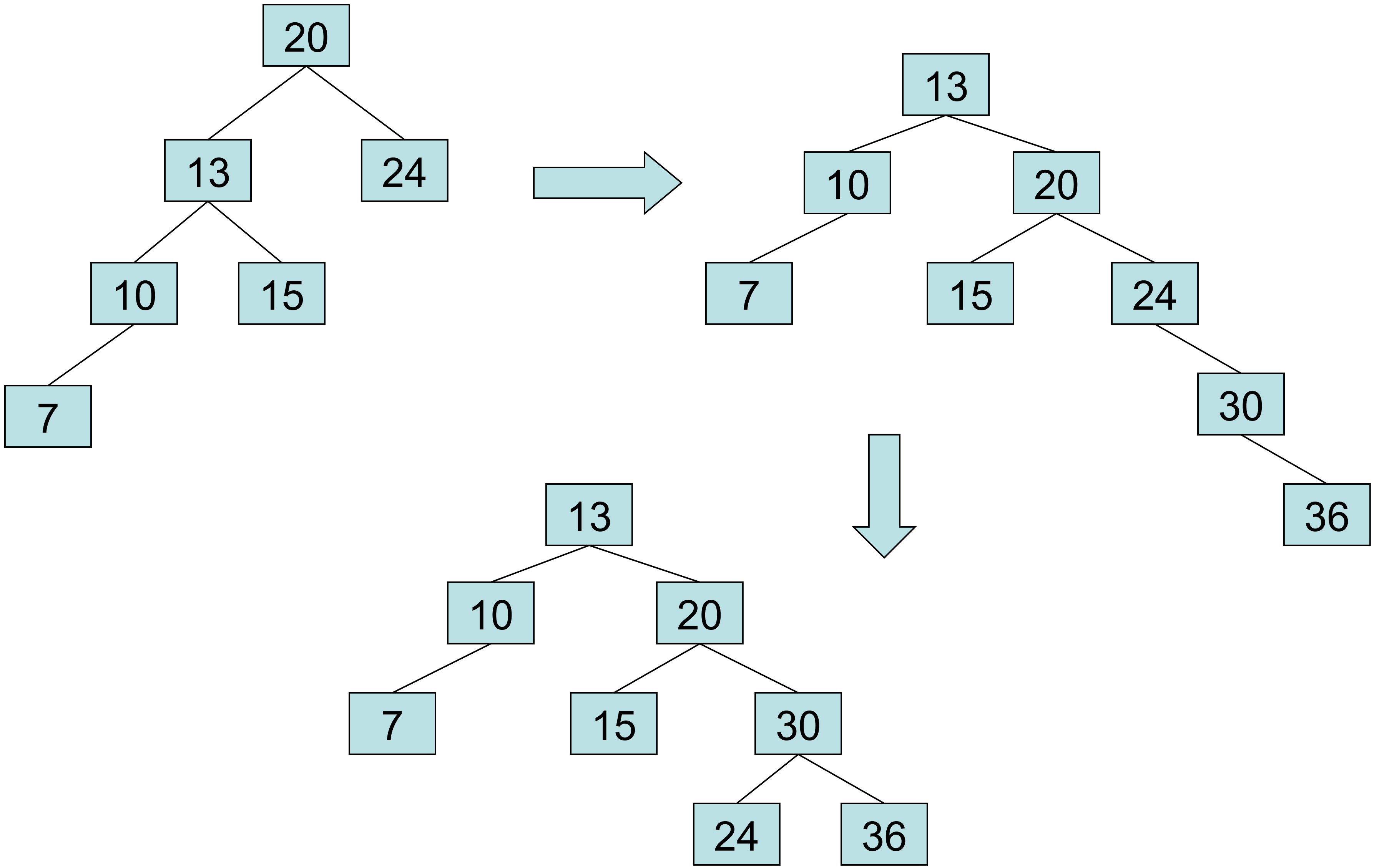
In Class Exercises

- Build an AVL tree with the following values:
  - 15, 20, 24, 10, 13, 7, 30, 36, 25

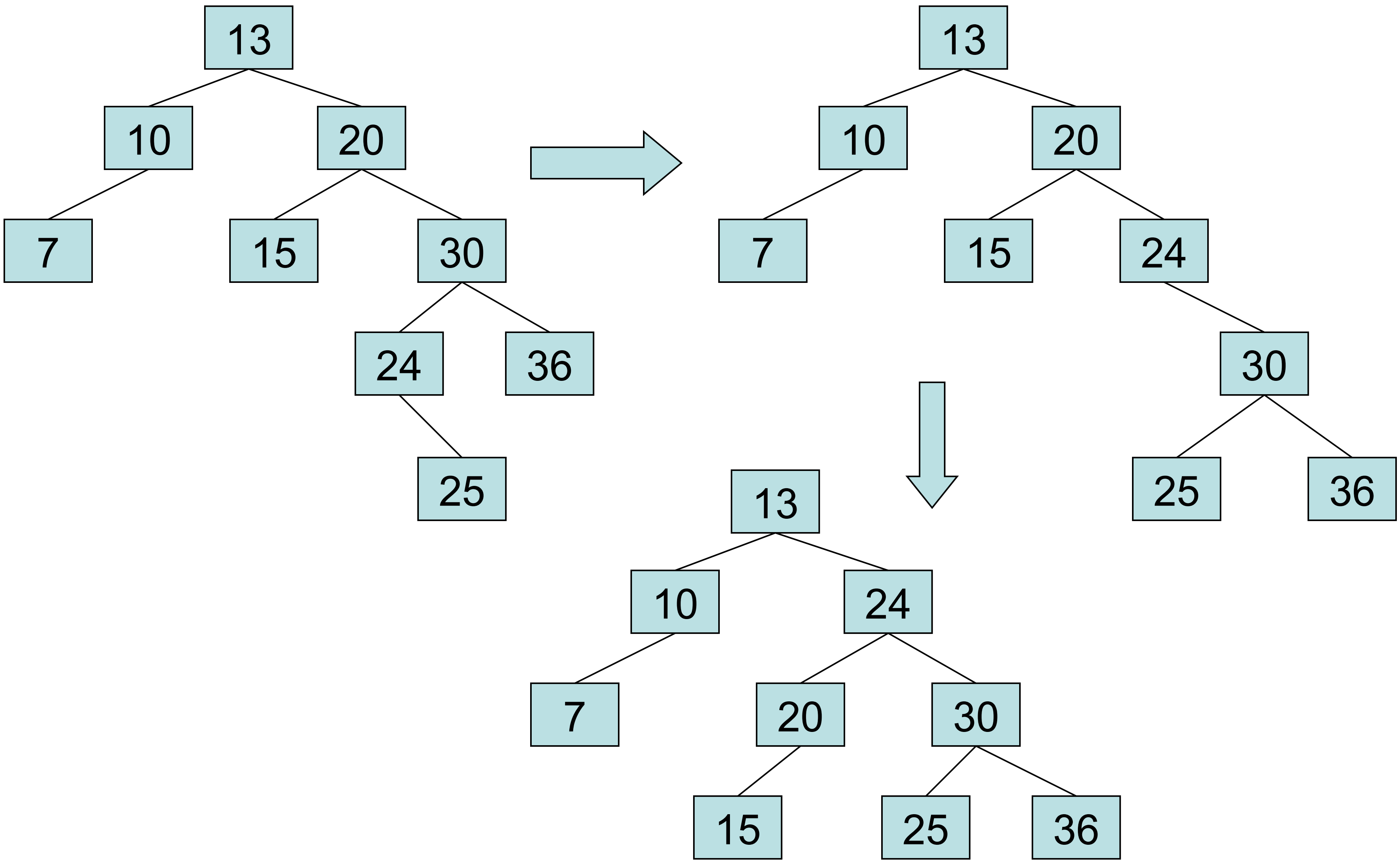
15, 20, 24, 10, 13, 7, 30, 36, 25



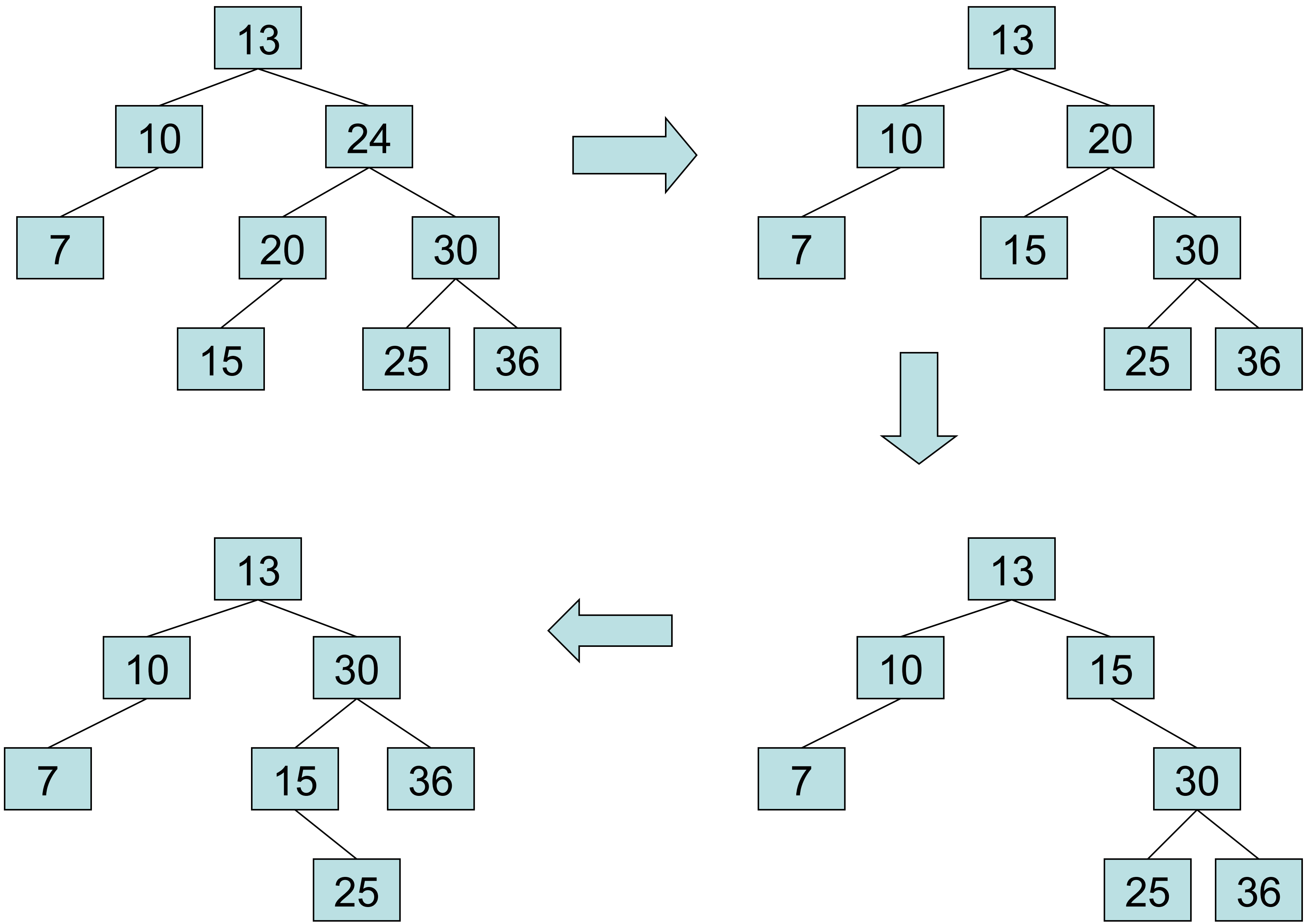
15, 20, 24, 10, 13, 7, 30, 36, 25



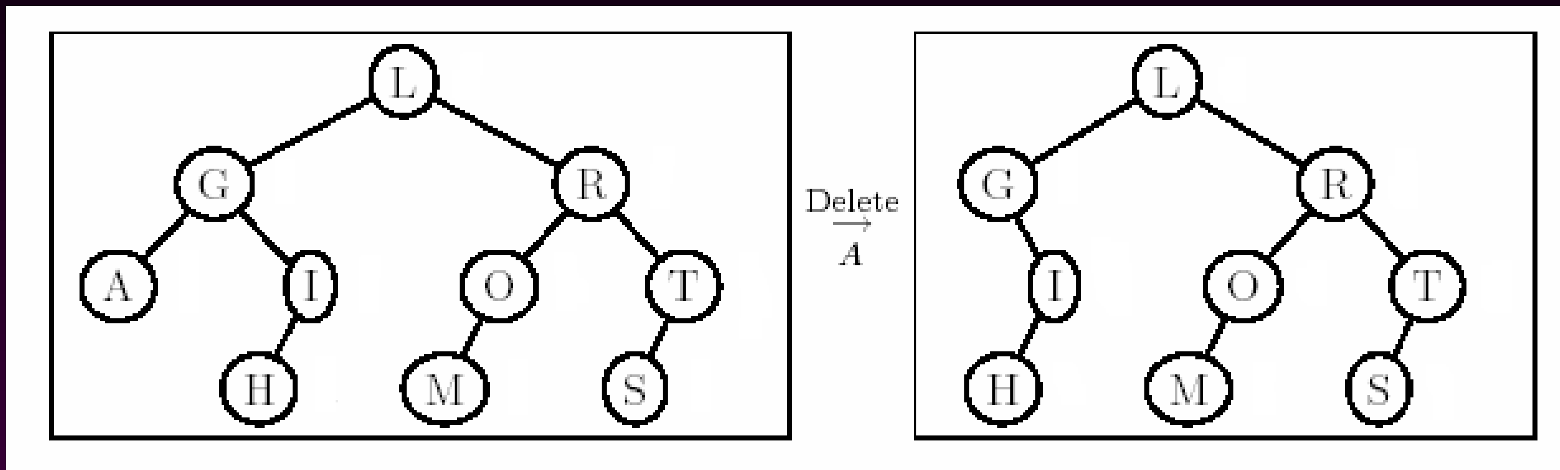
15, 20, 24, 10, 13, 7, 30, 36, 25

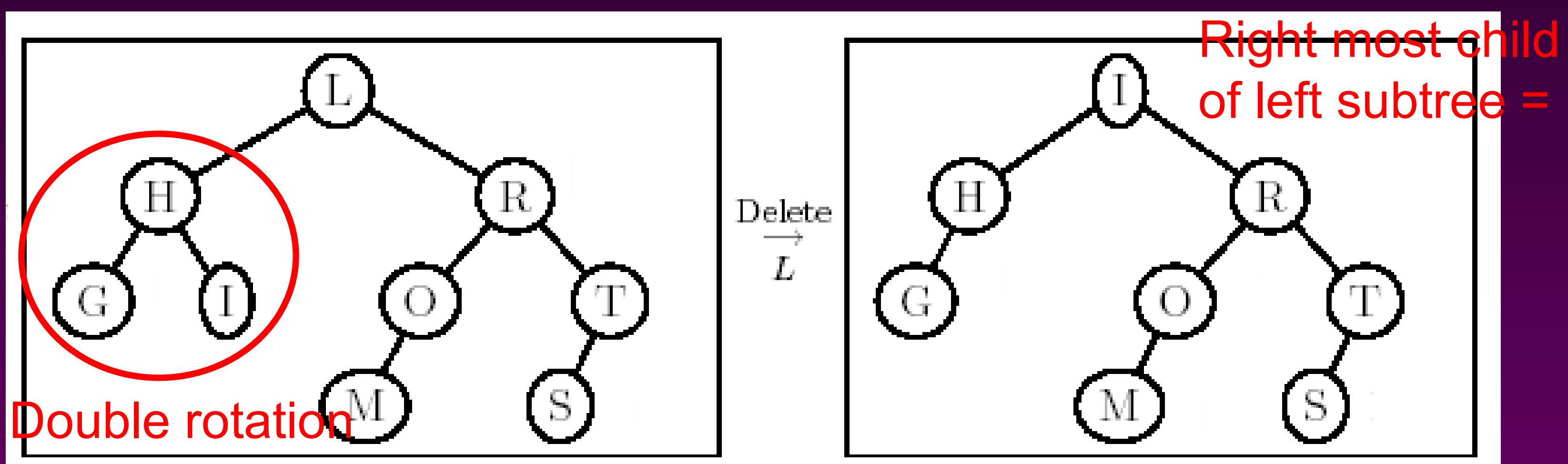


Remove 24 and 20 from the AVL tree.



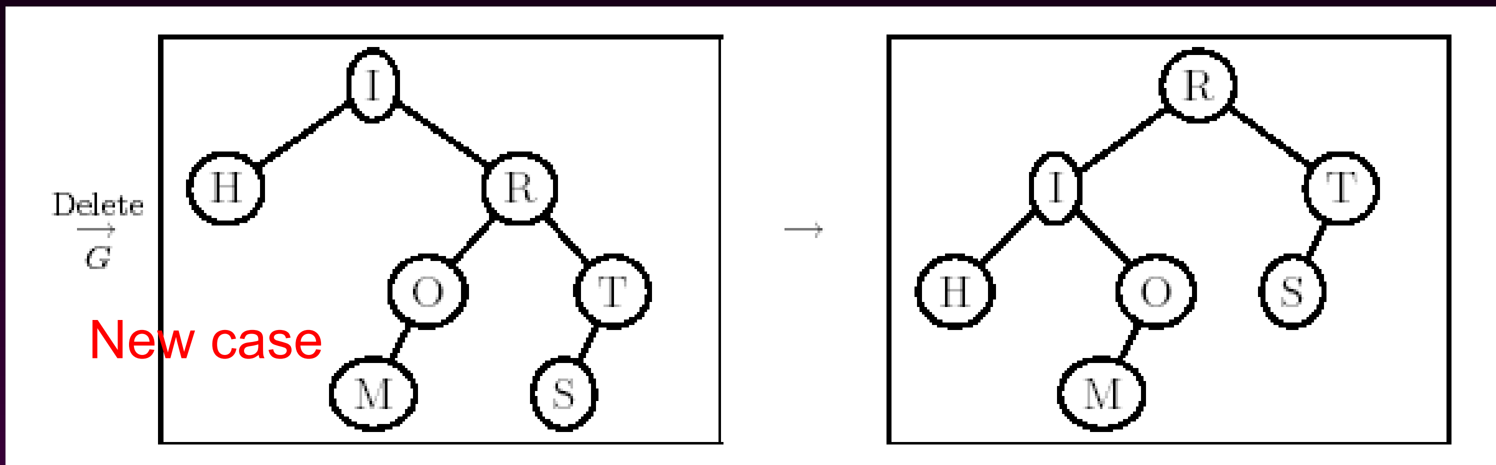
Deletion Example 2

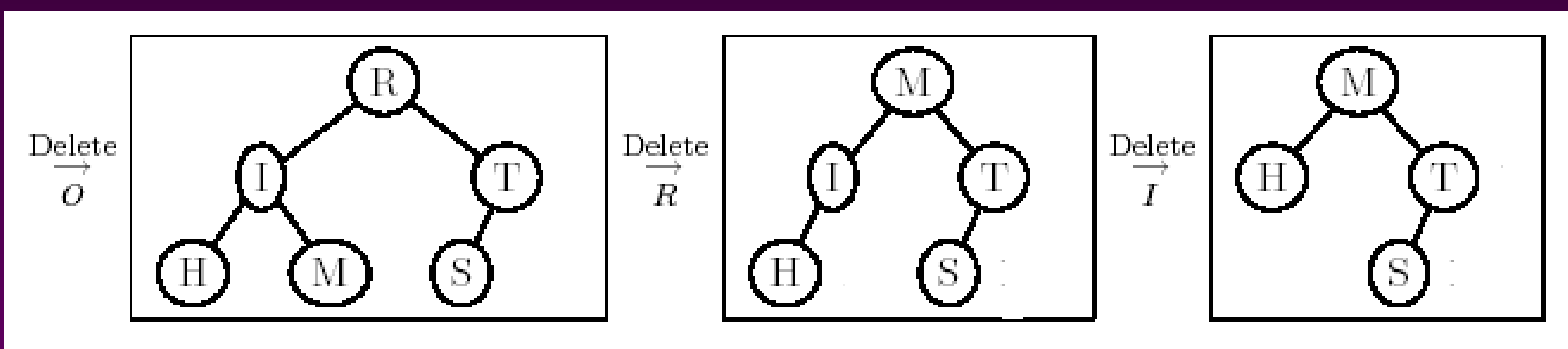




Ok here!

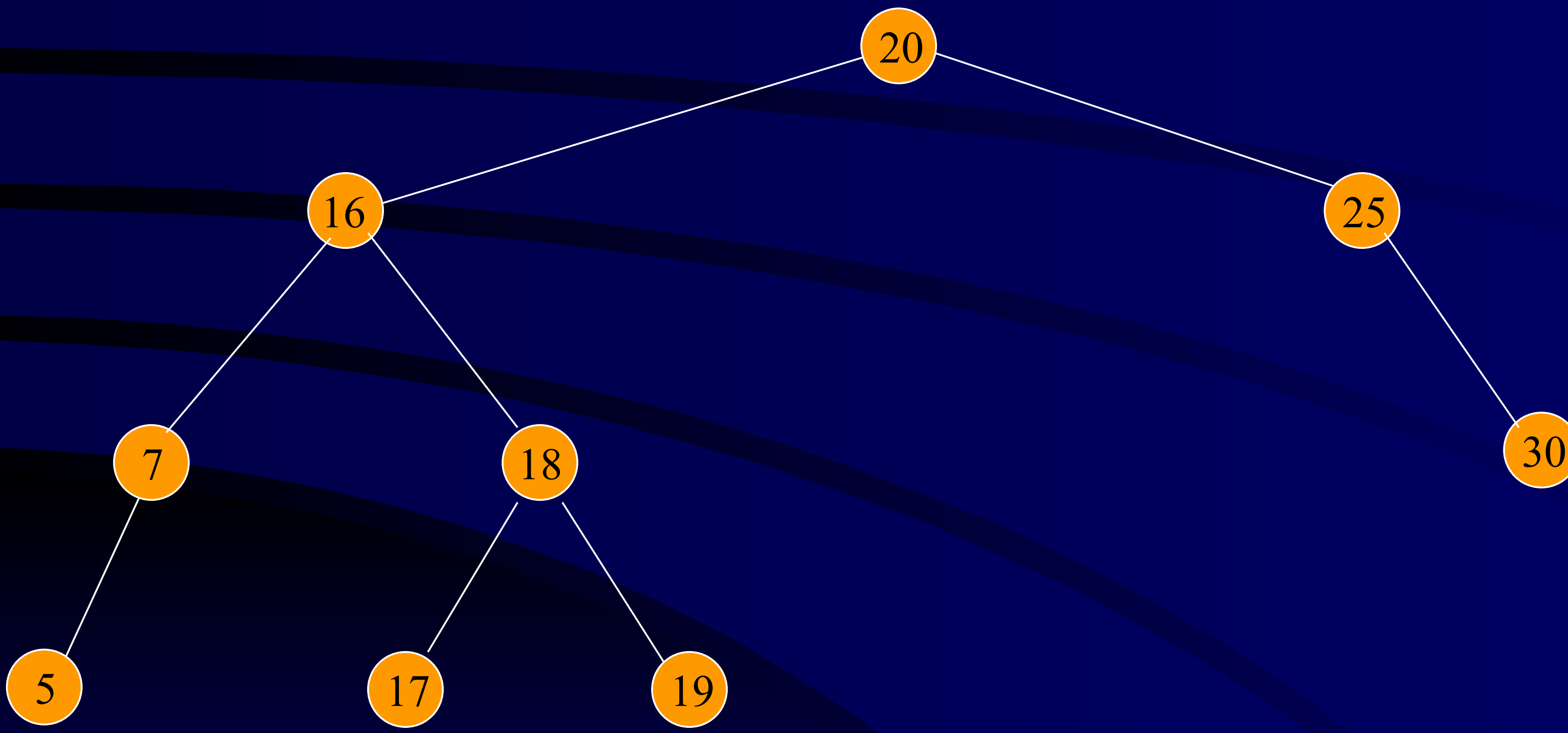
Example 2 Cont'd





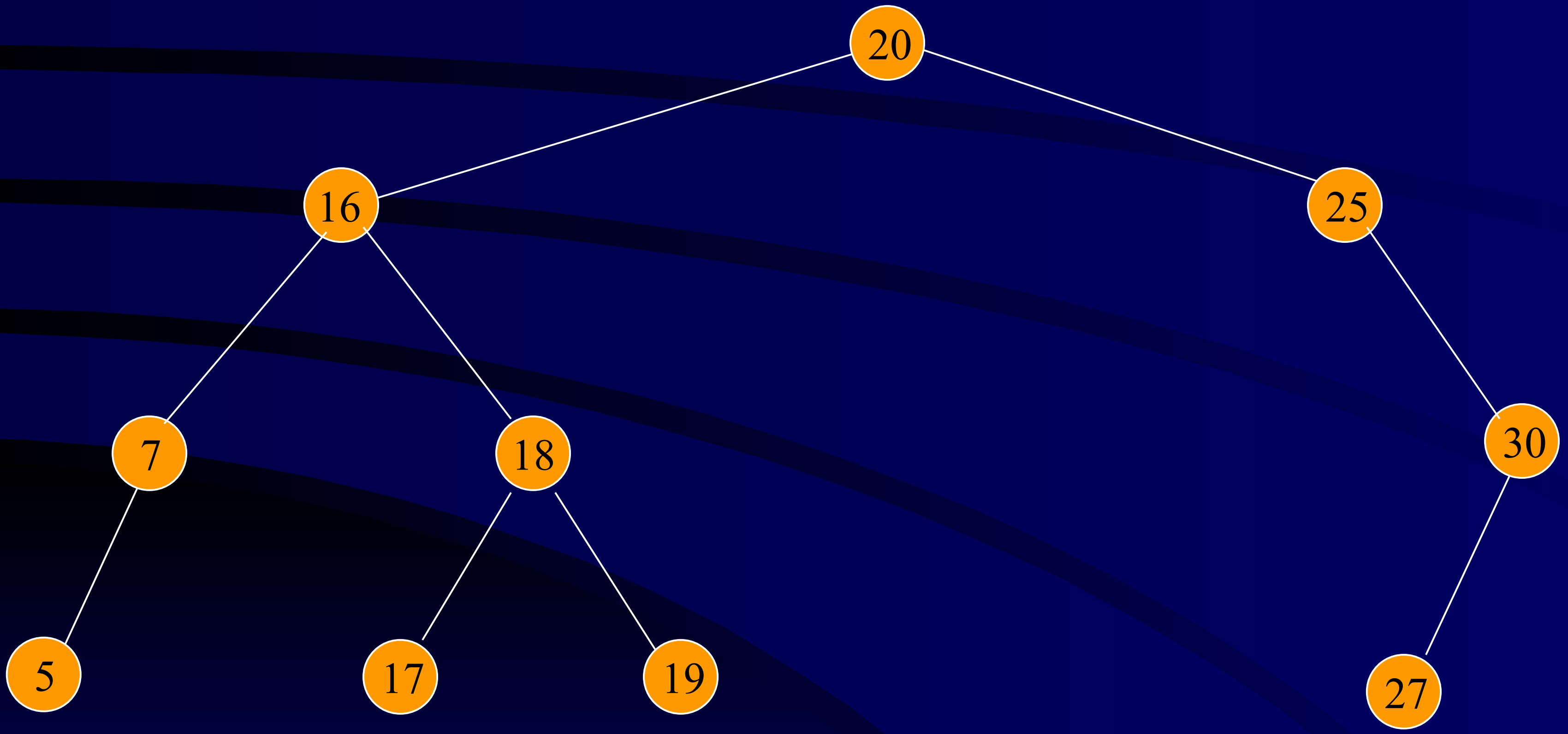
AVL Trees
(Adelson – Velskii – Landis)

AVL tree:



AVL Trees

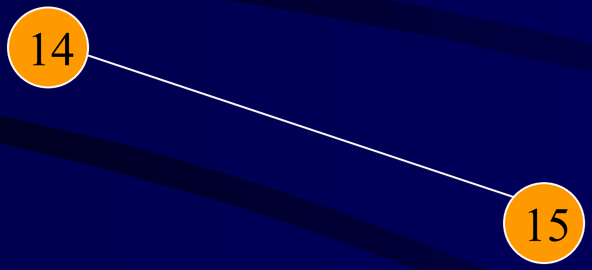
Not AVL tree:



AVL Tree Rotations

Single rotations: insert 14, 15, 16, 13, 12, 11, 10

- First insert 14 and 15:

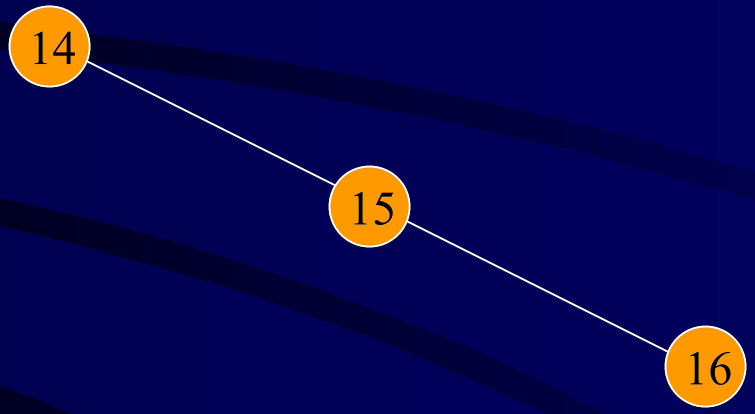


- Now insert 16.

AVL Tree Rotations

Single rotations:

- Inserting 16 causes AVL violation:

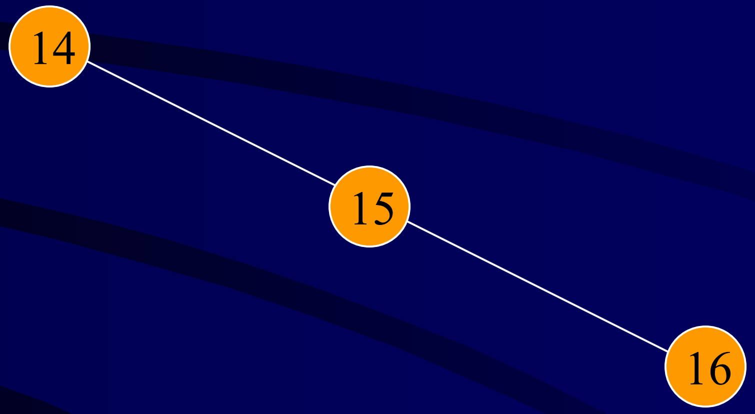


- Need to rotate.

AVL Tree Rotations

Single rotations:

- Inserting 16 causes AVL violation:

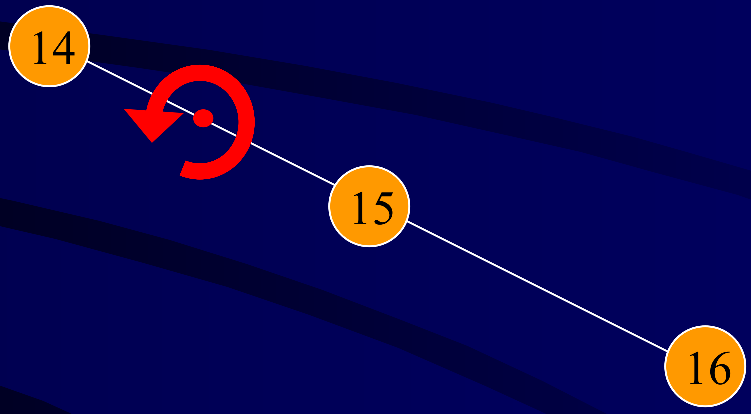


- Need to rotate.

AVL Tree Rotations

Single rotations:

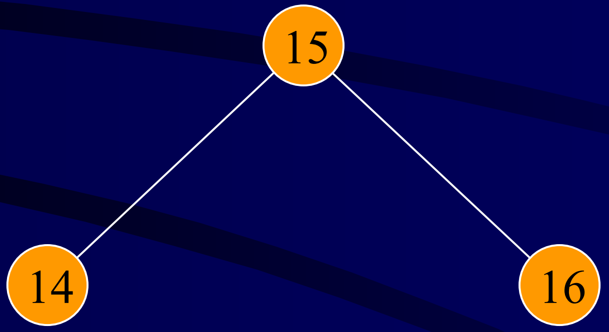
- Rotation type:



AVL Tree Rotations

Single rotations:

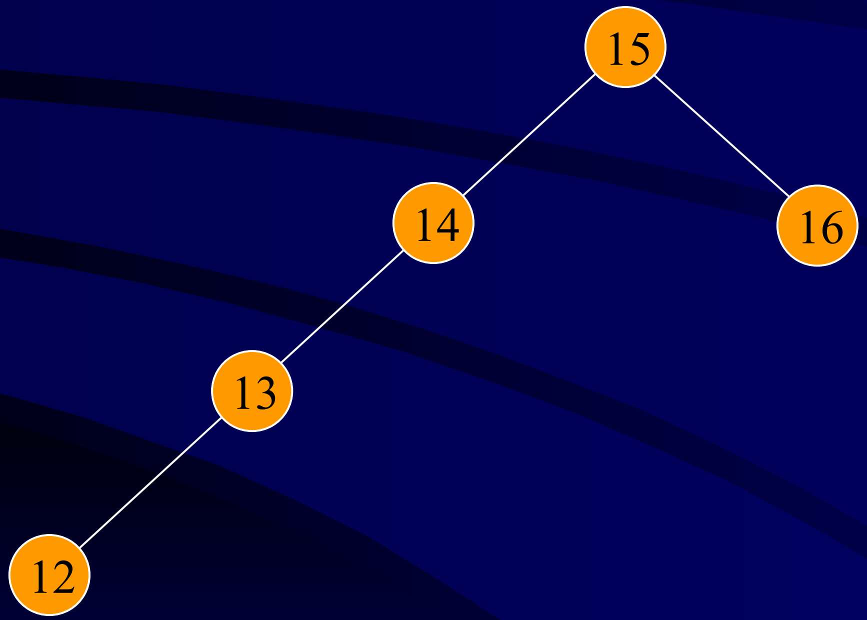
- Rotation restores AVL balance:



AVL Tree Rotations

Single rotations:

- Now insert 13 and 12:

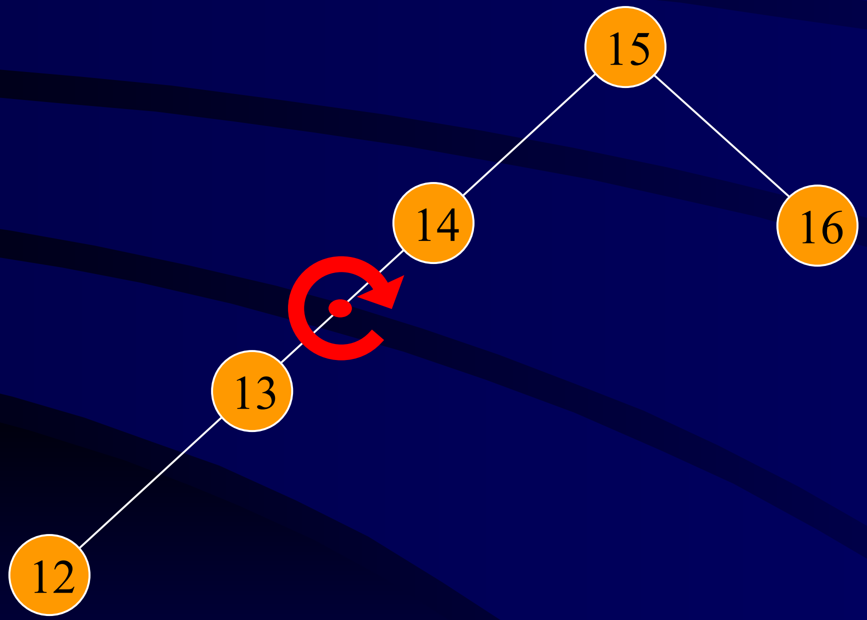


- AVL violation - need to rotate.

AVL Tree Rotations

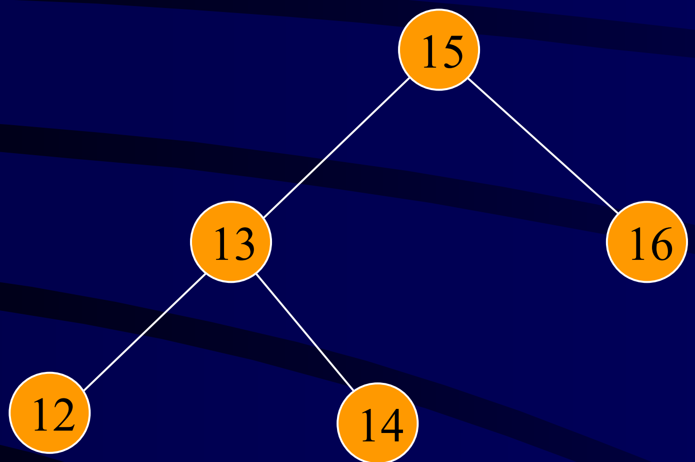
Single rotations:

- Rotation type:



AVL Tree Rotations

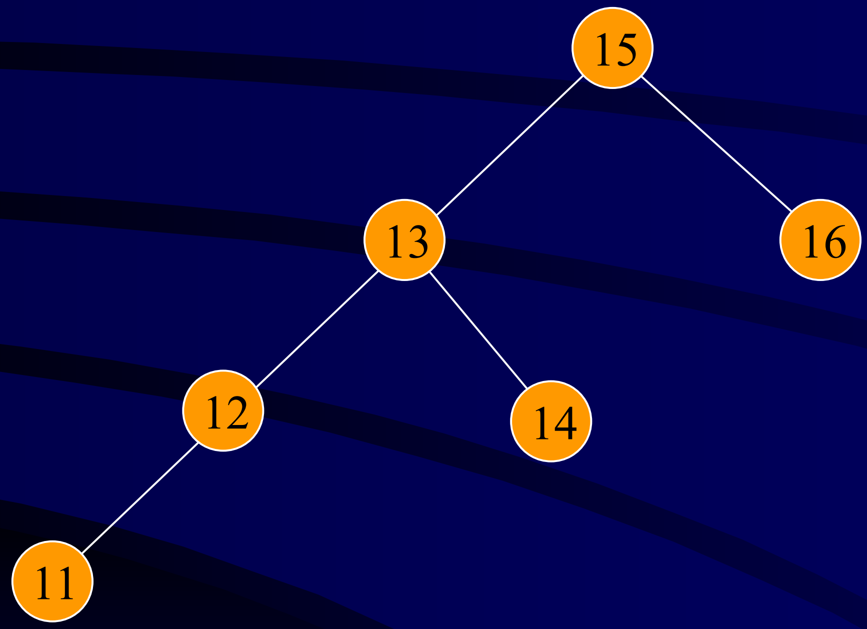
Single rotations:



- Now insert 11.

AVL Tree Rotations

Single rotations:

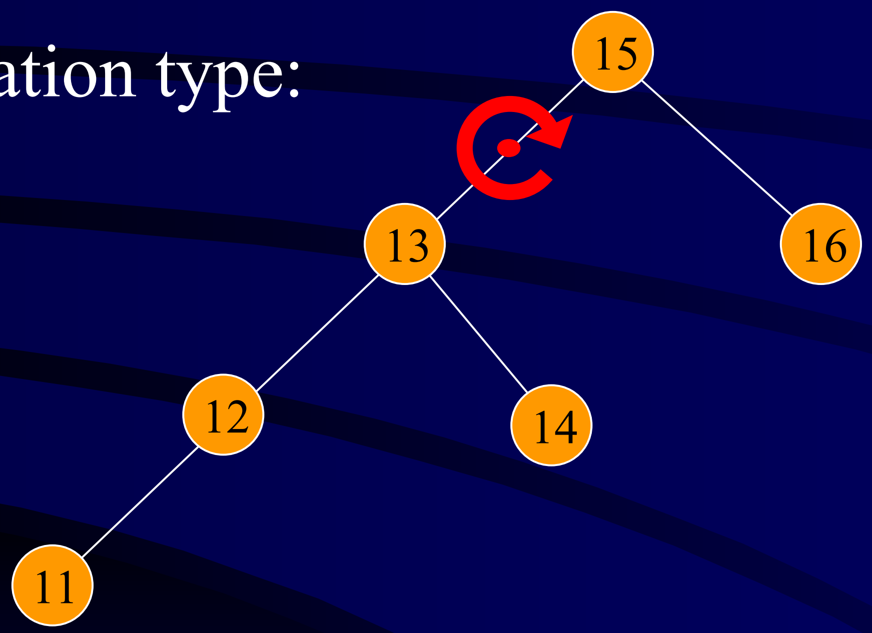


- AVL violation – need to rotate

AVL Tree Rotations

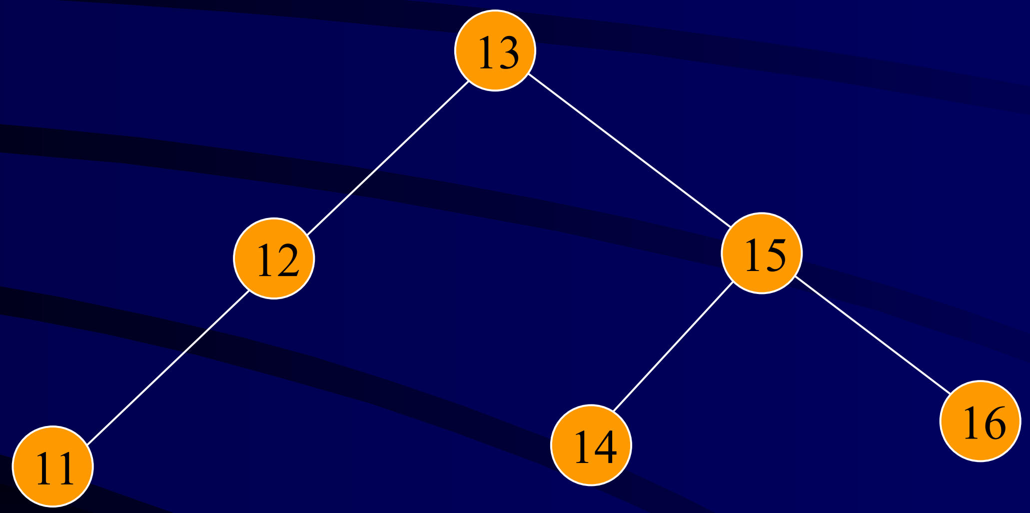
Single rotations:

- Rotation type:



AVL Tree Rotations

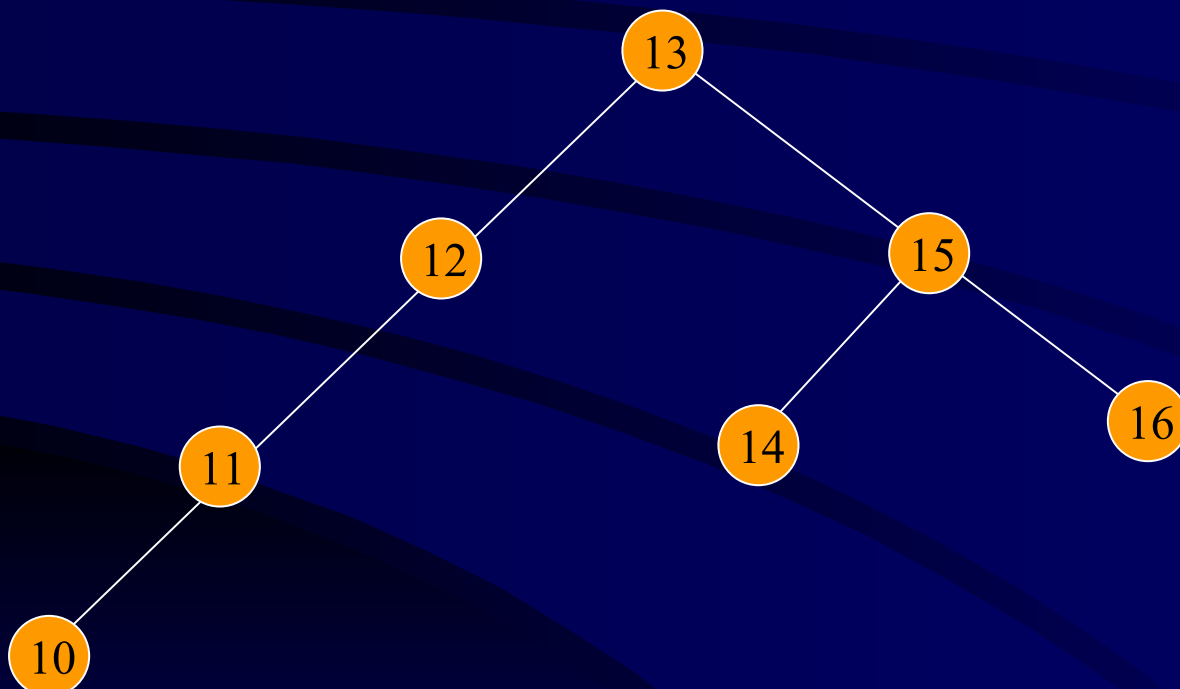
Single rotations:



- Now insert 10.

AVL Tree Rotations

Single rotations:

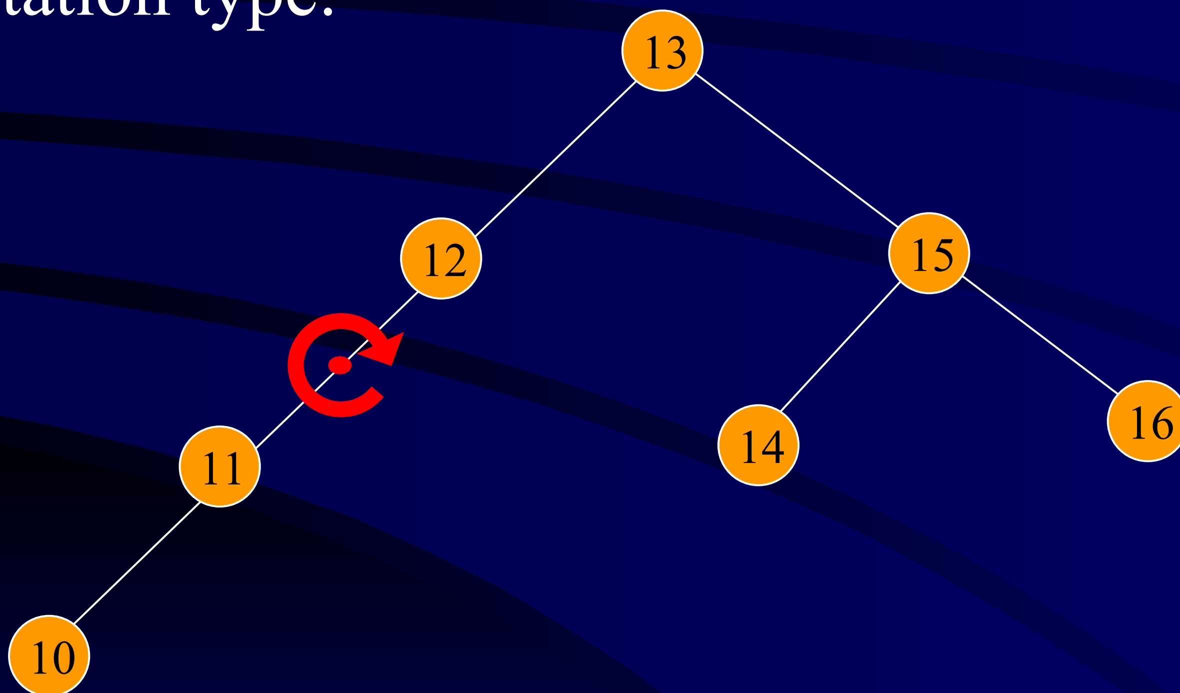


- AVL violation – need to rotate

AVL Tree Rotations

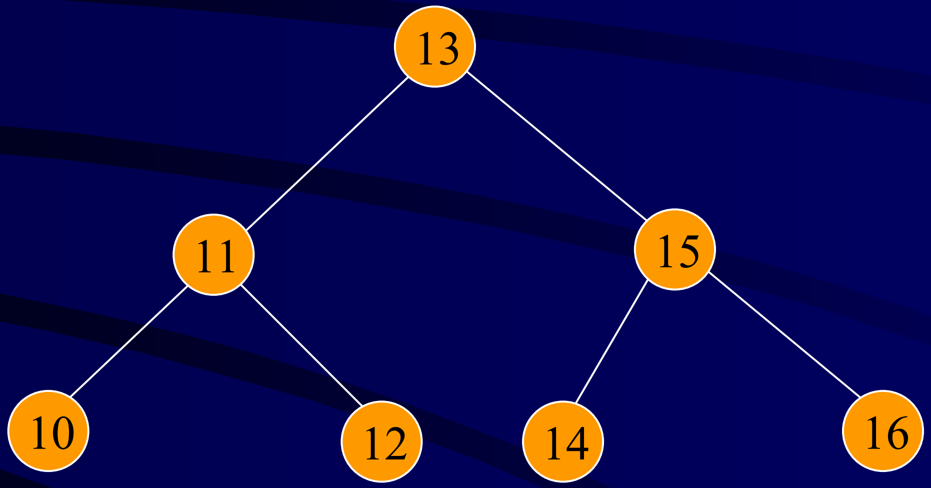
Single rotations:

- Rotation type:



AVL Tree Rotations

Single rotations:

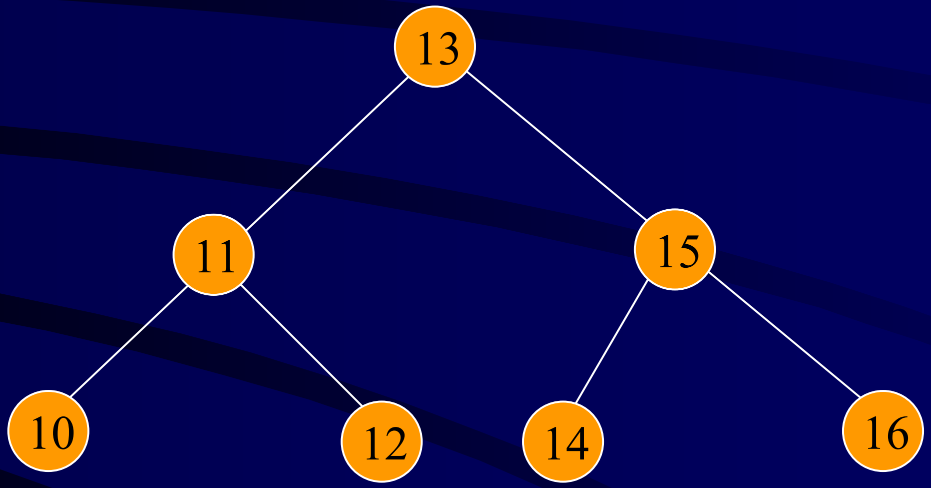


- AVL balance restored.

AVL Tree Rotations

Double rotations: insert 1, 2, 3, 4, 5, 7, 6, 9, 8

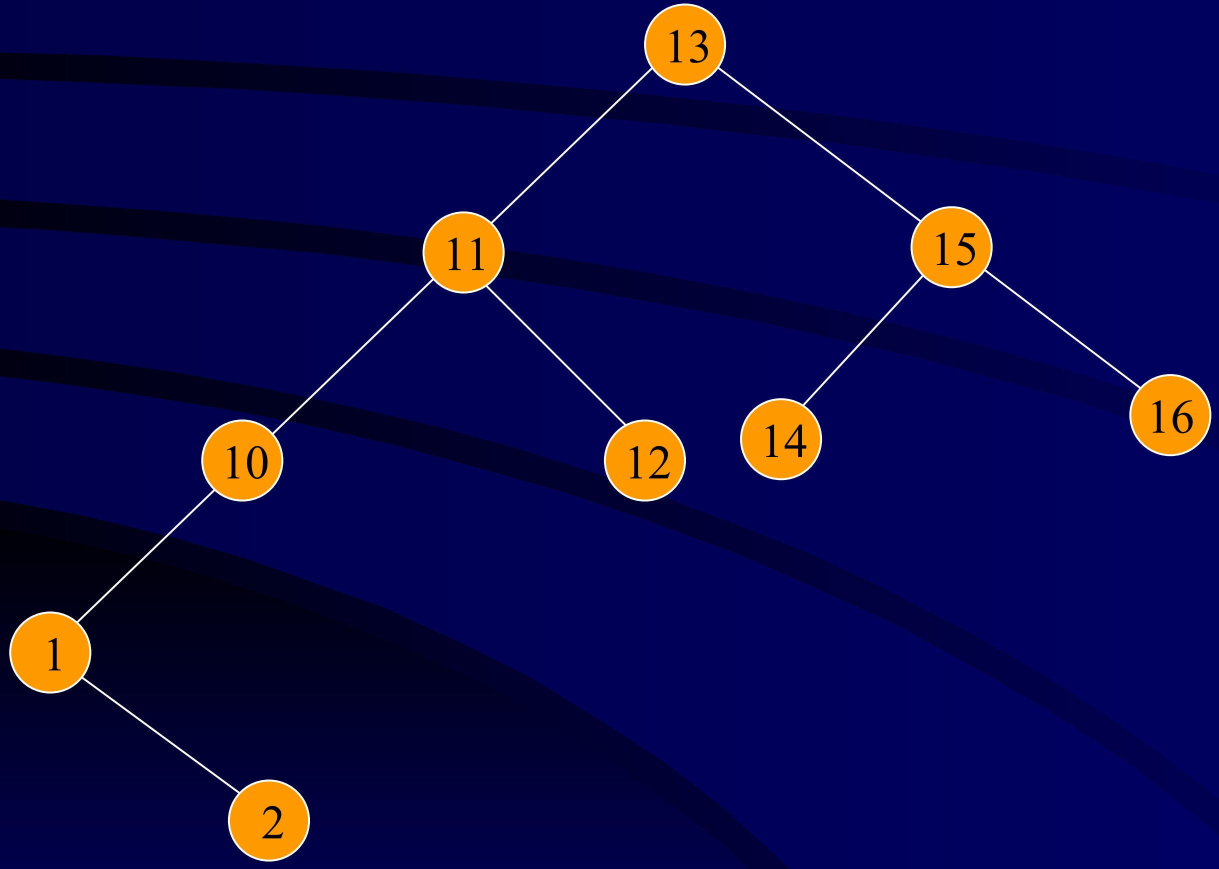
- First insert 1 and 2:



AVL Tree Rotations

Double rotations:

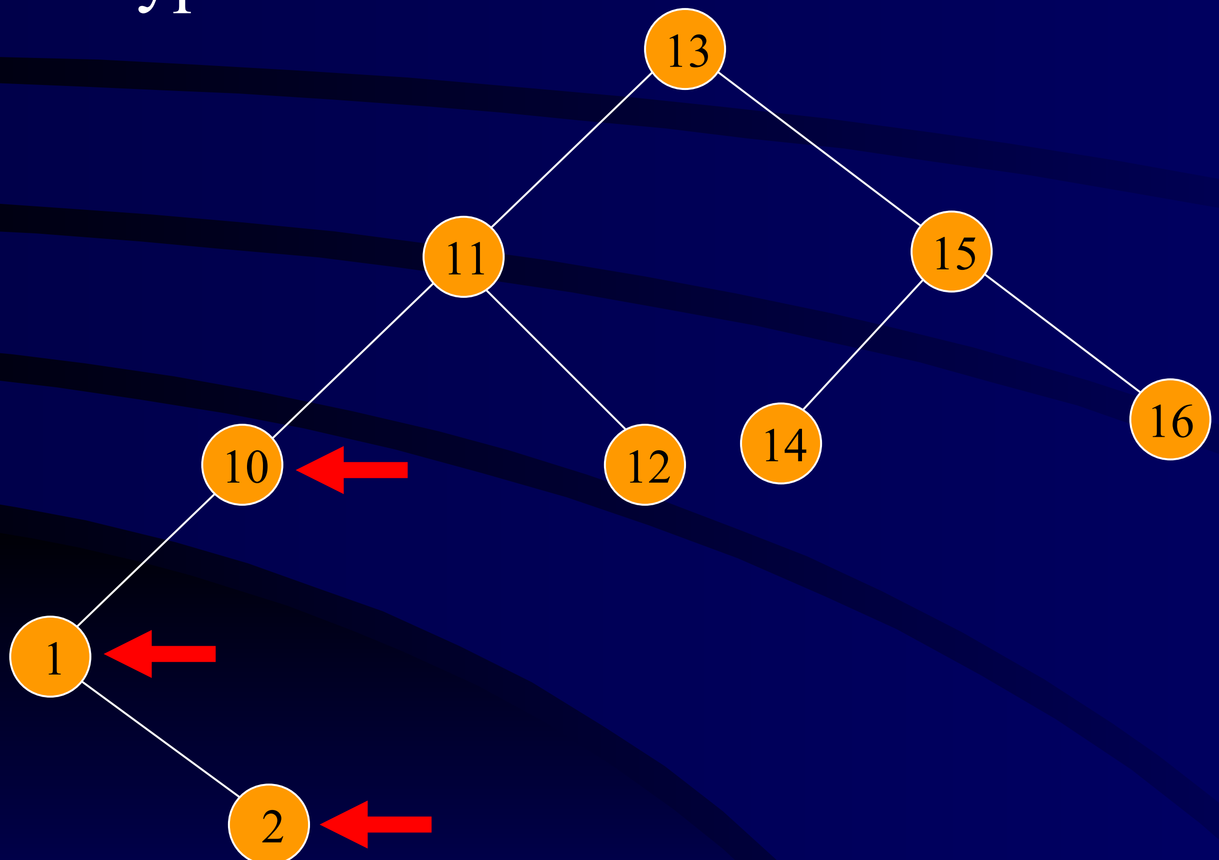
- AVL violation - rotate



AVL Tree Rotations

Double rotations:

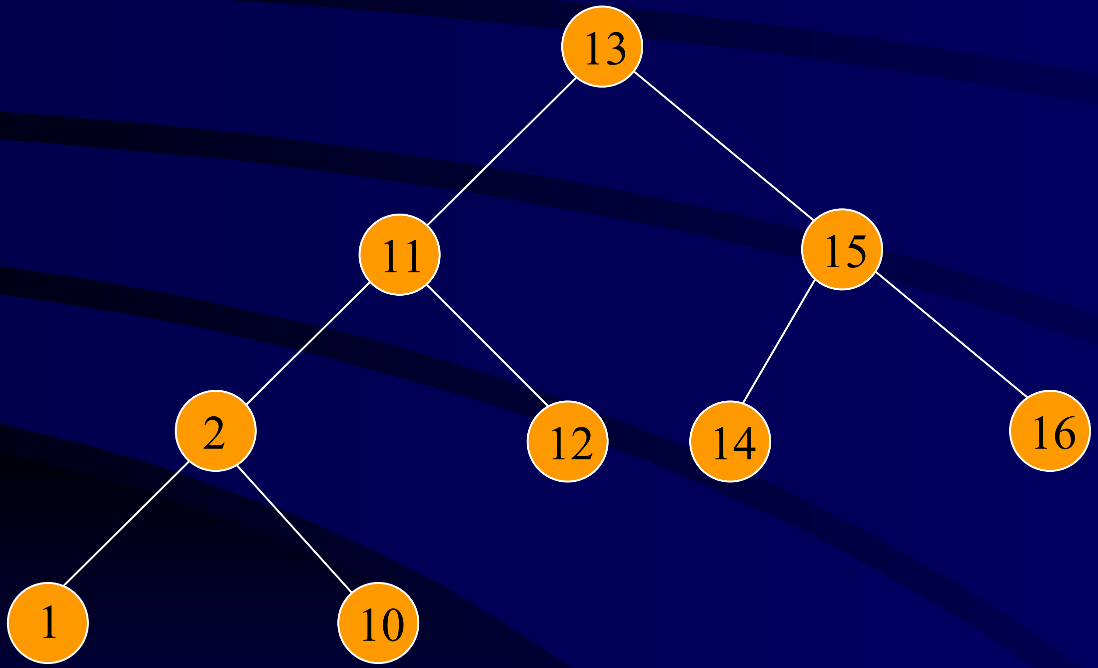
- Rotation type:



AVL Tree Rotations

Double rotations:

- AVL balance restored:

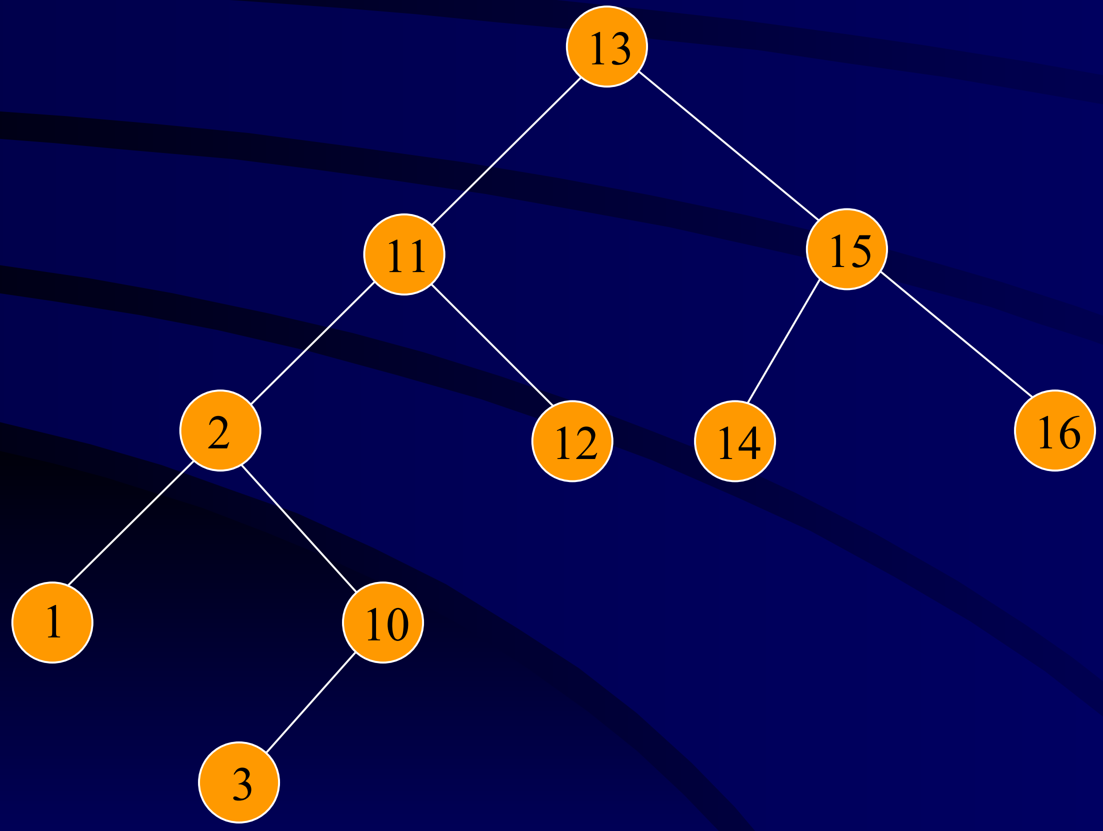


- Now insert 3.

AVL Tree Rotations

Double rotations:

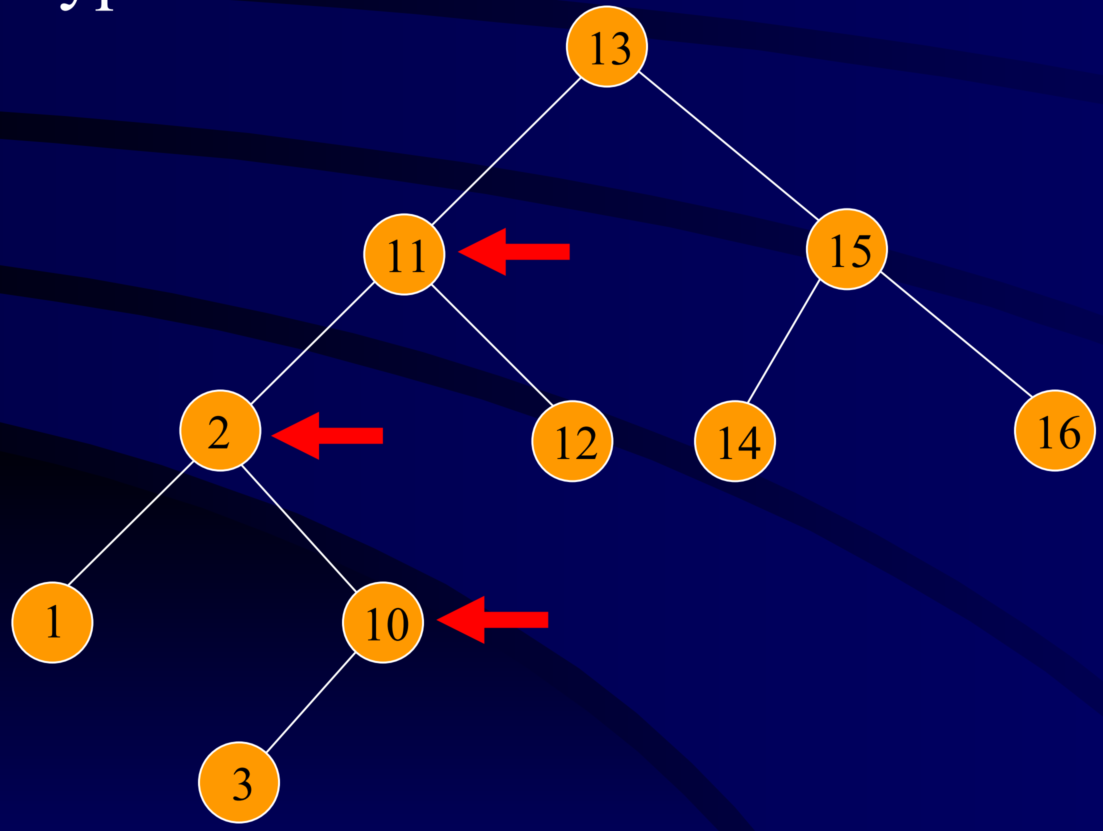
- AVL violation – rotate:



AVL Tree Rotations

Double rotations:

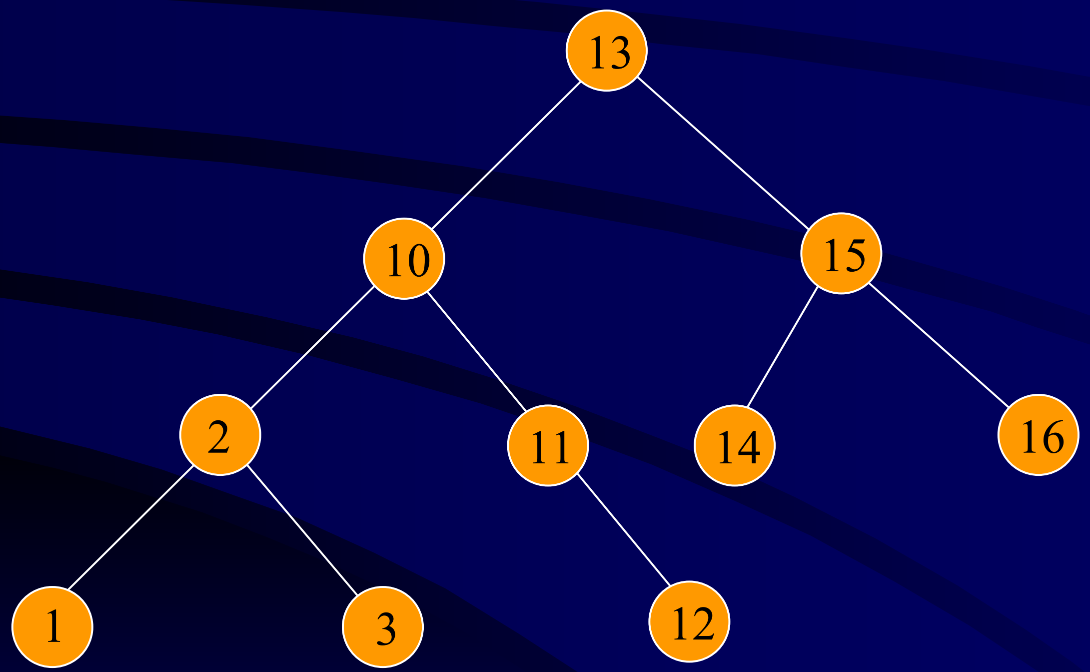
- Rotation type:



AVL Tree Rotations

Double rotations:

- AVL balance restored:

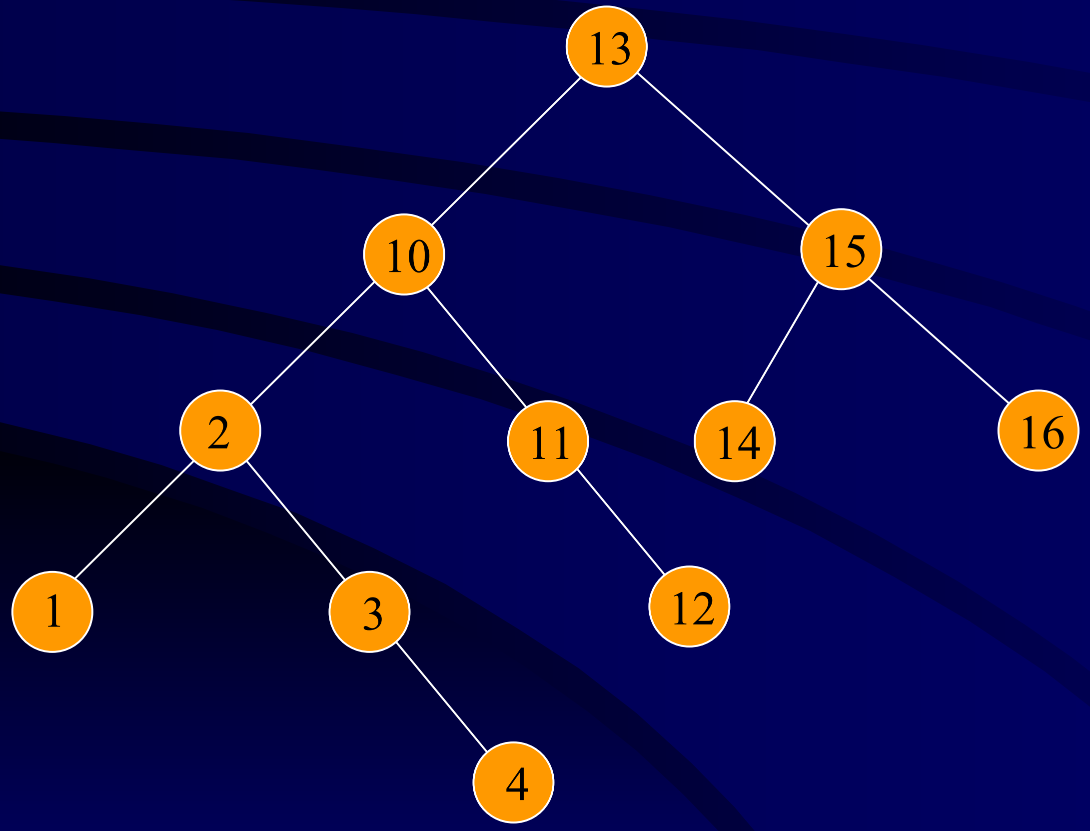


- Now insert 4.

AVL Tree Rotations

Double rotations:

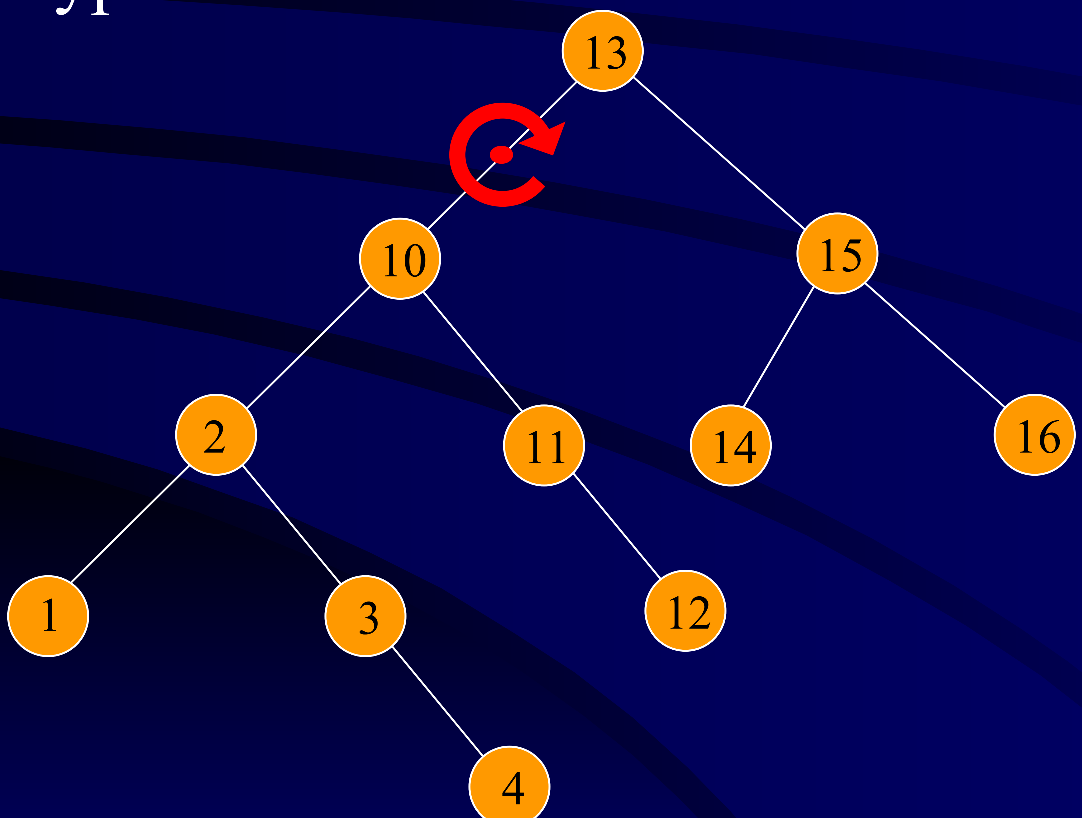
- AVL violation - rotate



AVL Tree Rotations

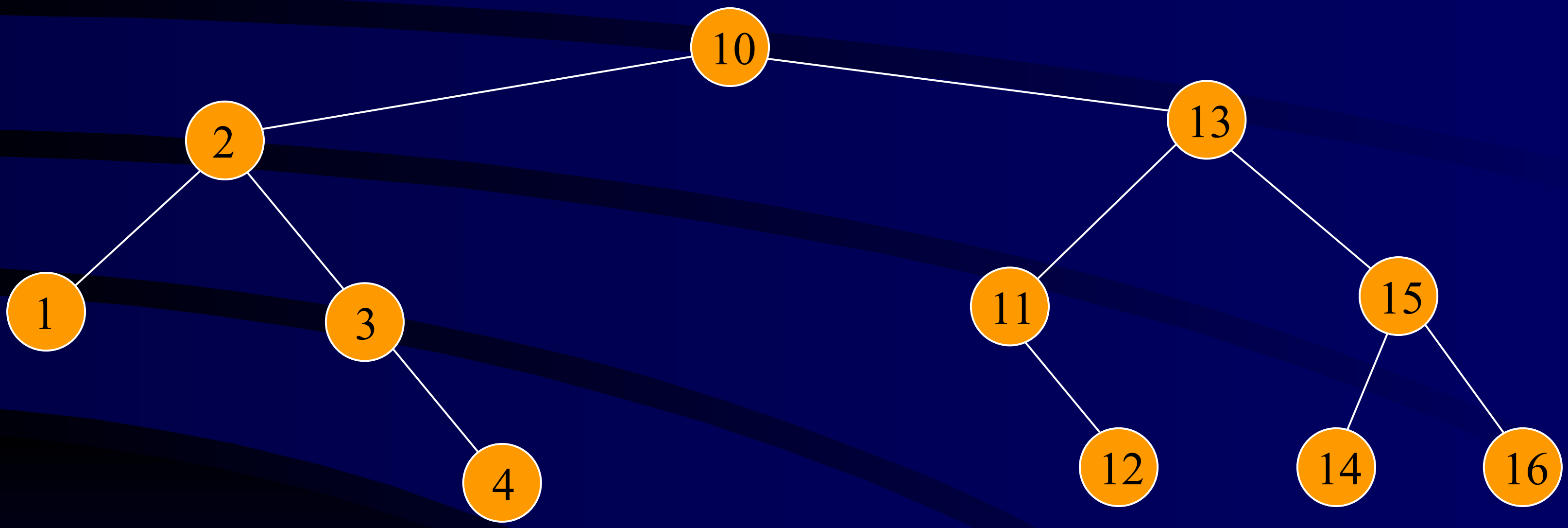
Double rotations:

- Rotation type:



AVL Tree Rotations

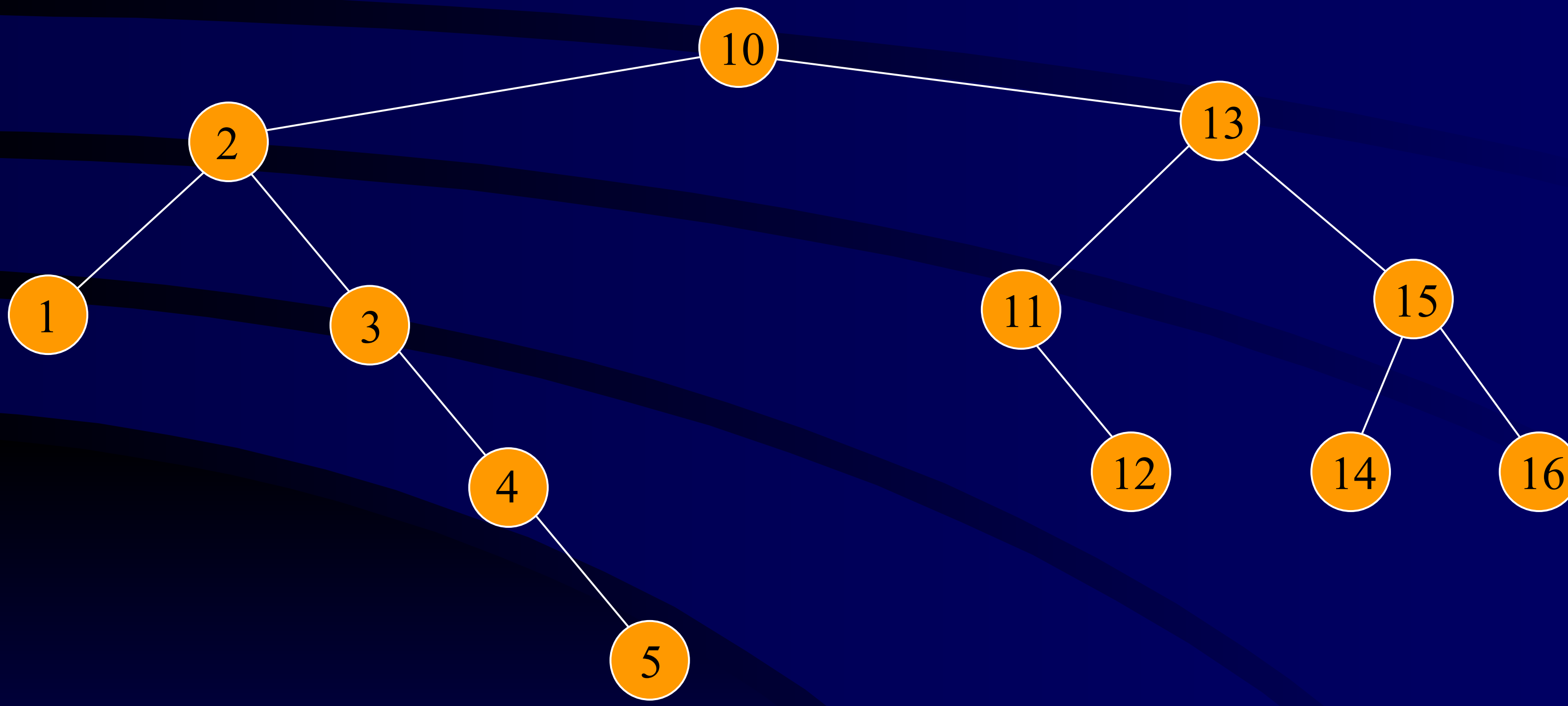
Double rotations:



- Now insert 5.

AVL Tree Rotations

Double rotations:

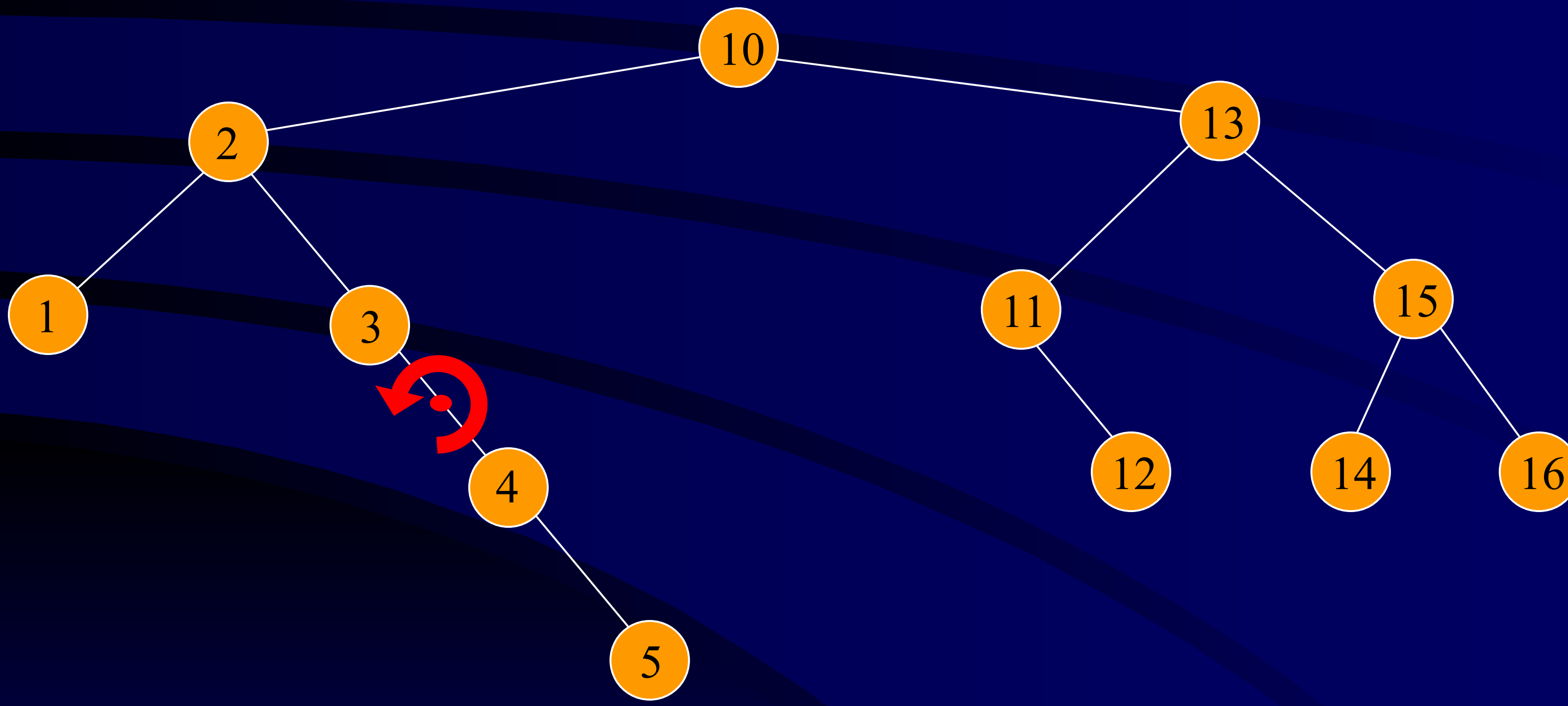


- AVL violation – rotate.

AVL Tree Rotations

Single rotations:

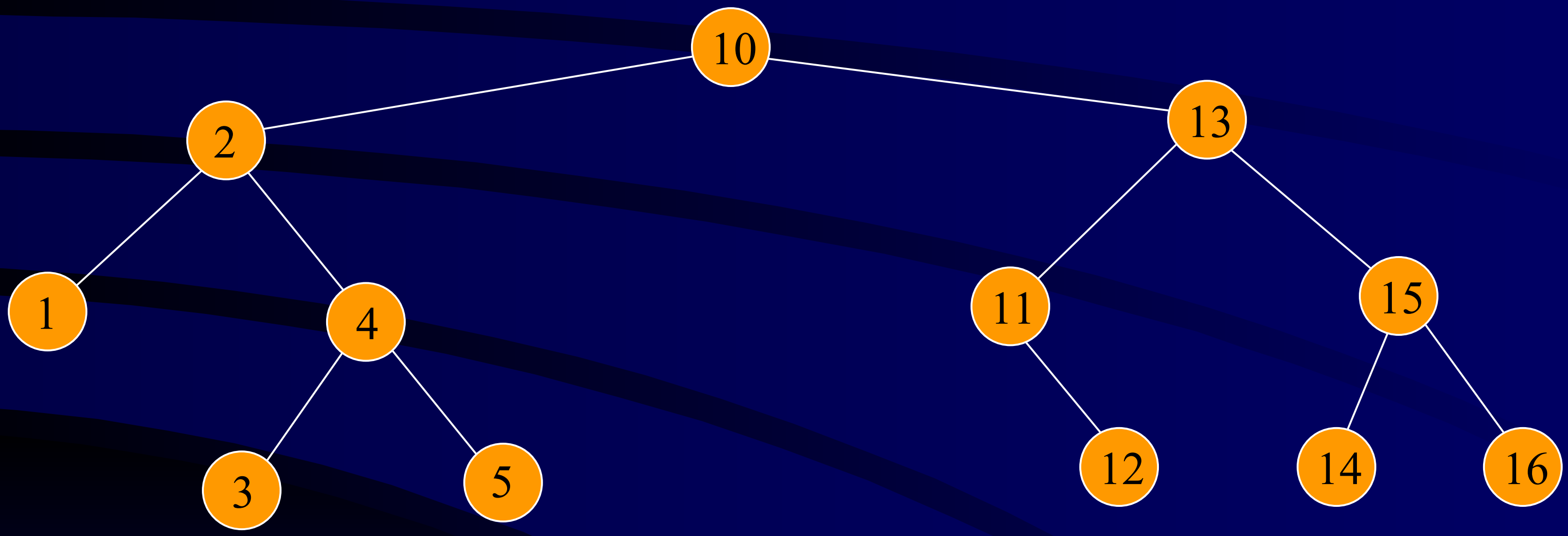
- Rotation type:



AVL Tree Rotations

Single rotations:

- AVL balance restored:

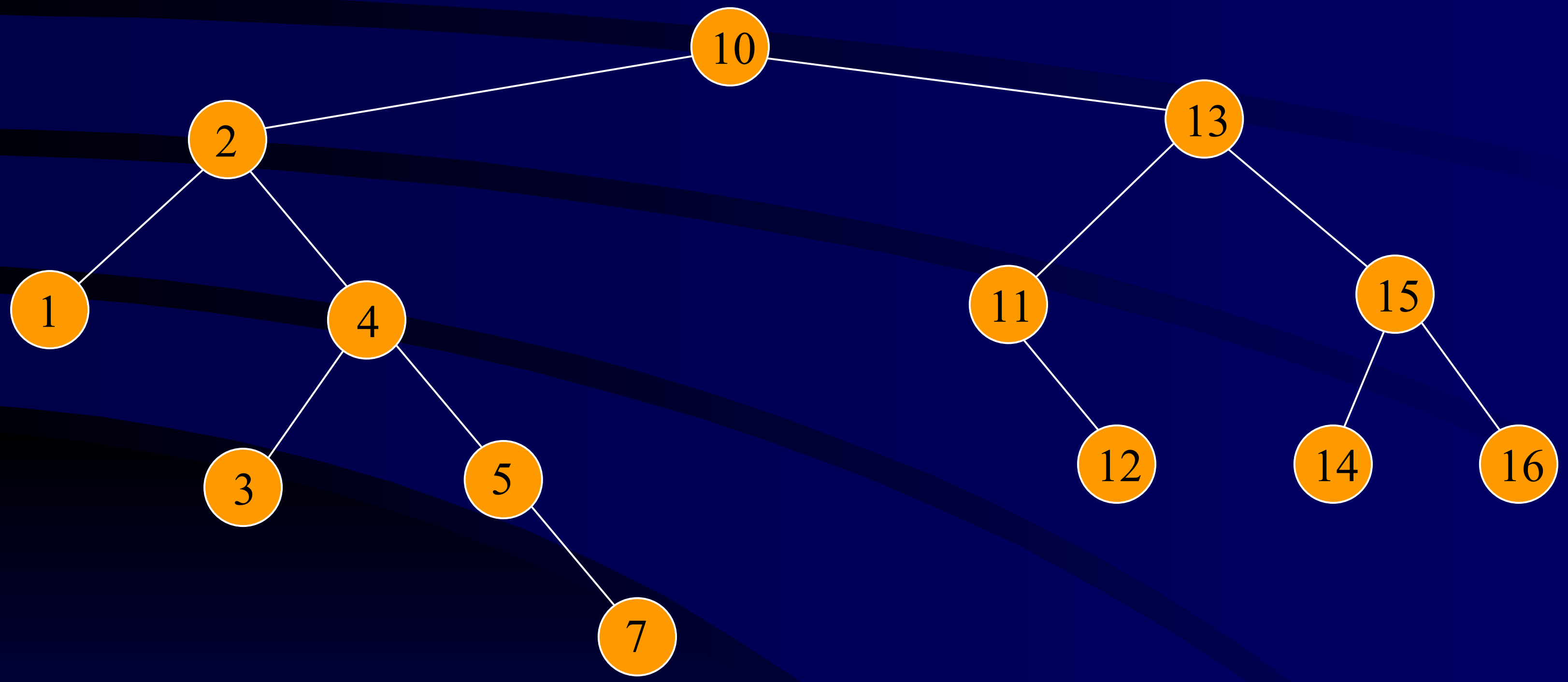


- Now insert 7.

AVL Tree Rotations

Single rotations:

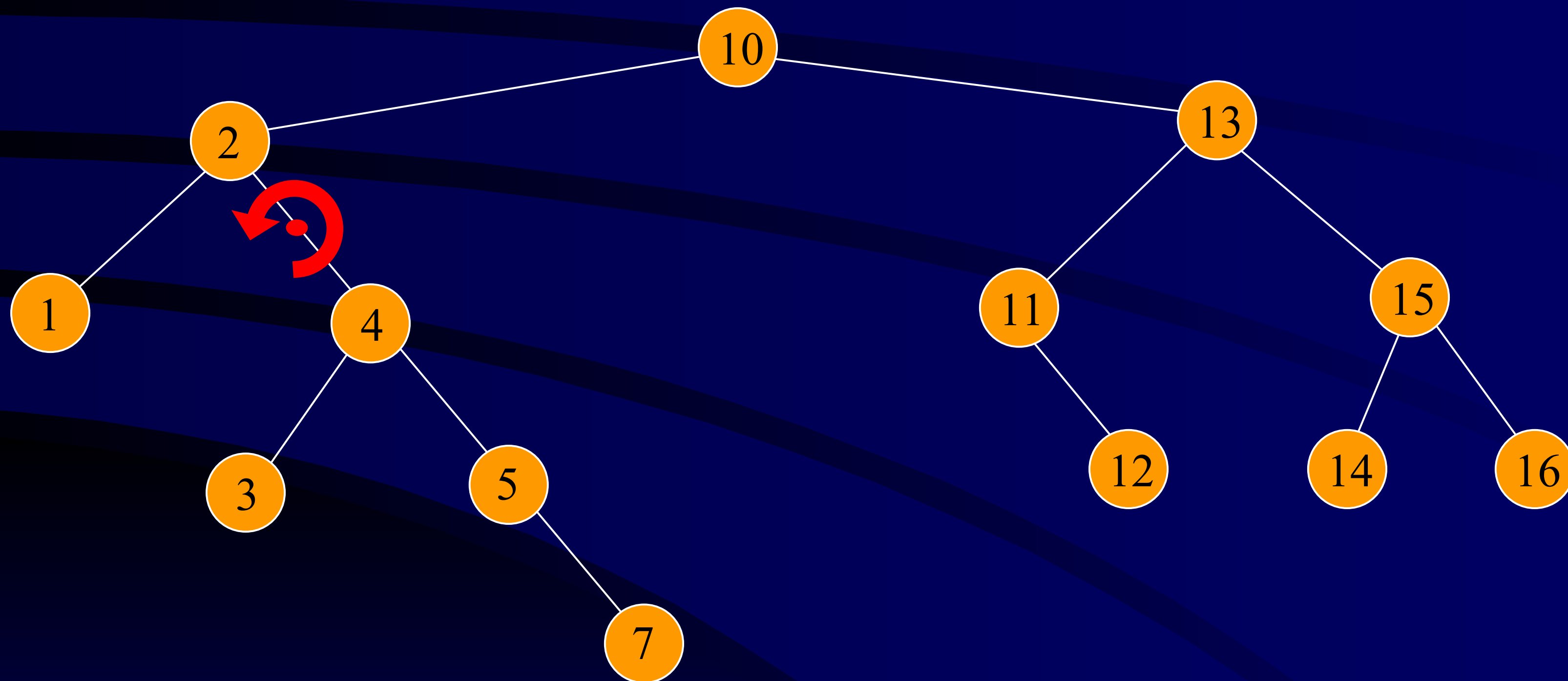
- AVL violation – rotate.



AVL Tree Rotations

Single rotations:

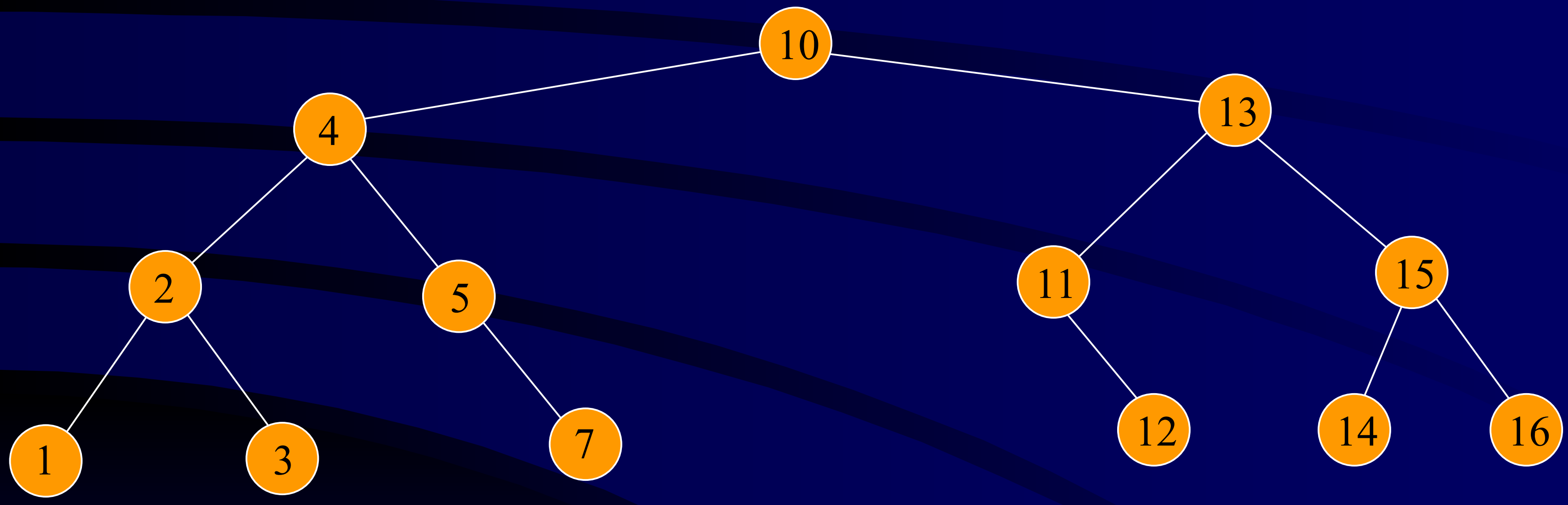
- Rotation type:



AVL Tree Rotations

Double rotations:

- AVL balance restored.

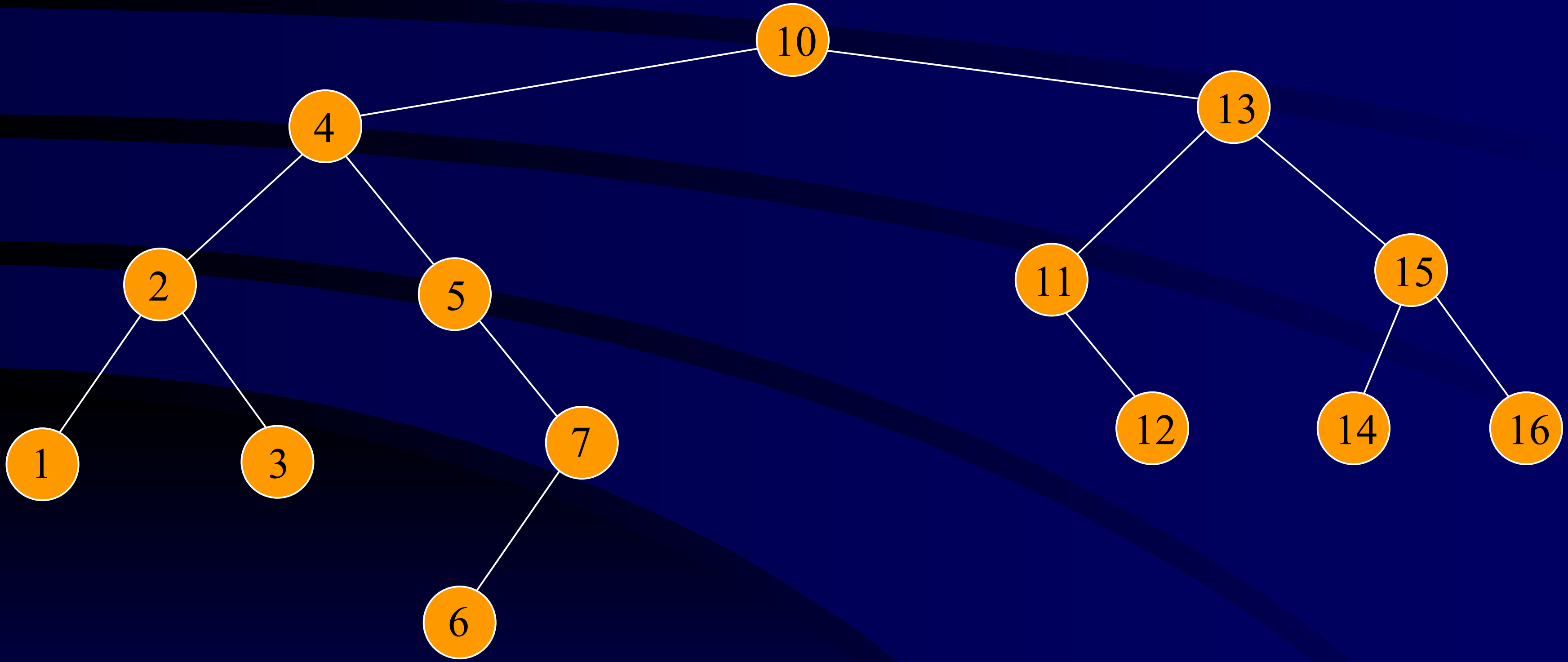


- Now insert 6.

AVL Tree Rotations

Double rotations:

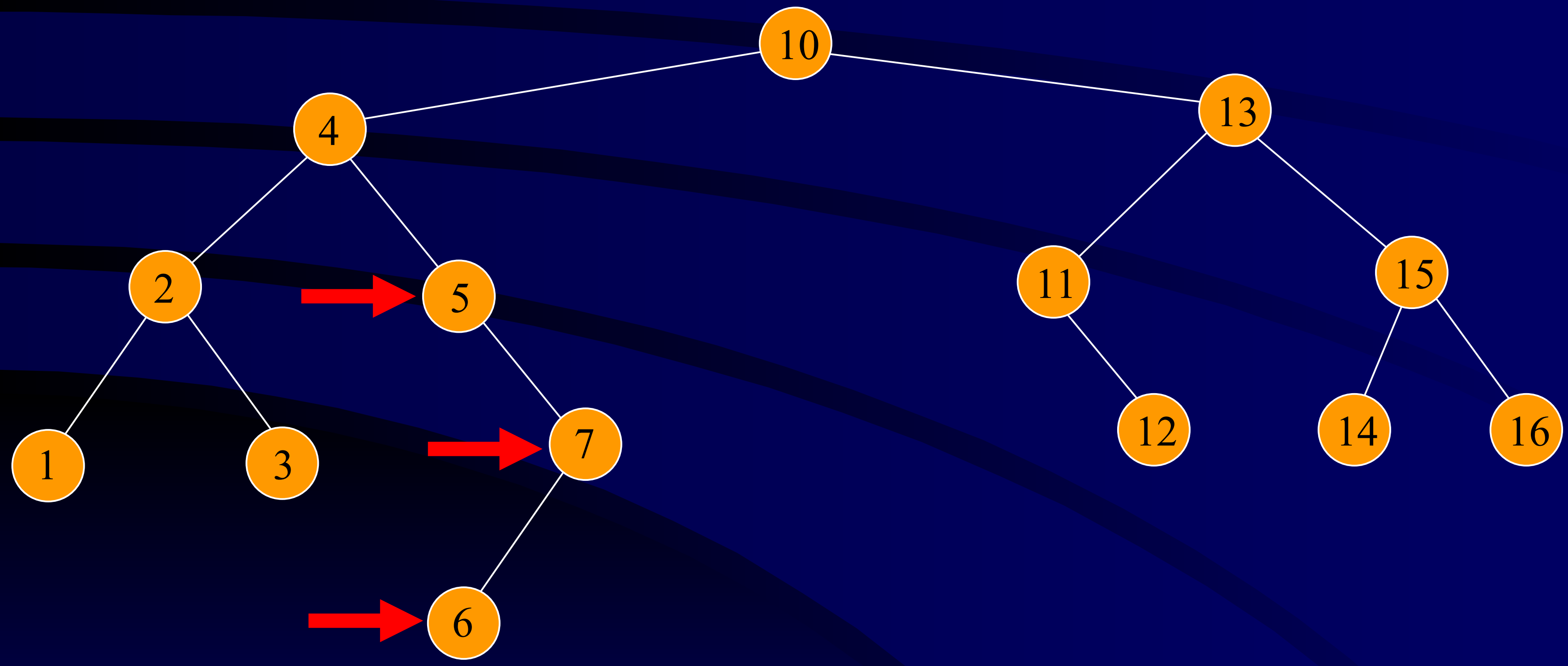
- AVL violation - rotate.



AVL Tree Rotations

Double rotations:

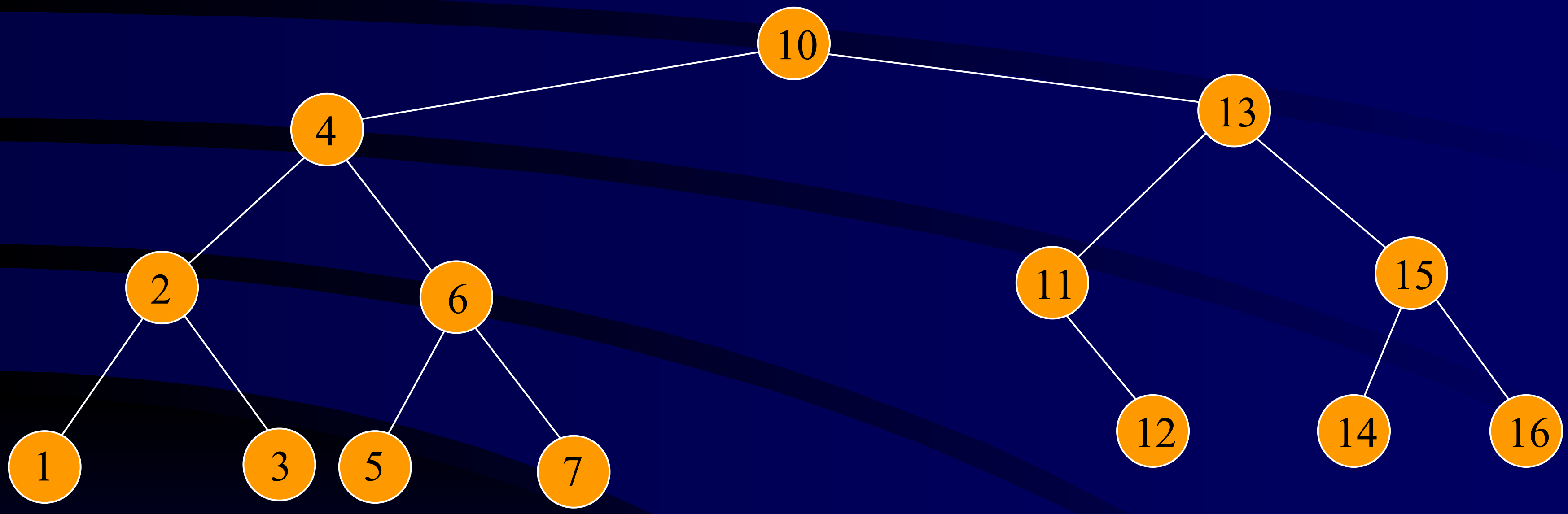
- Rotation type:



AVL Tree Rotations

Double rotations:

- AVL balance restored.

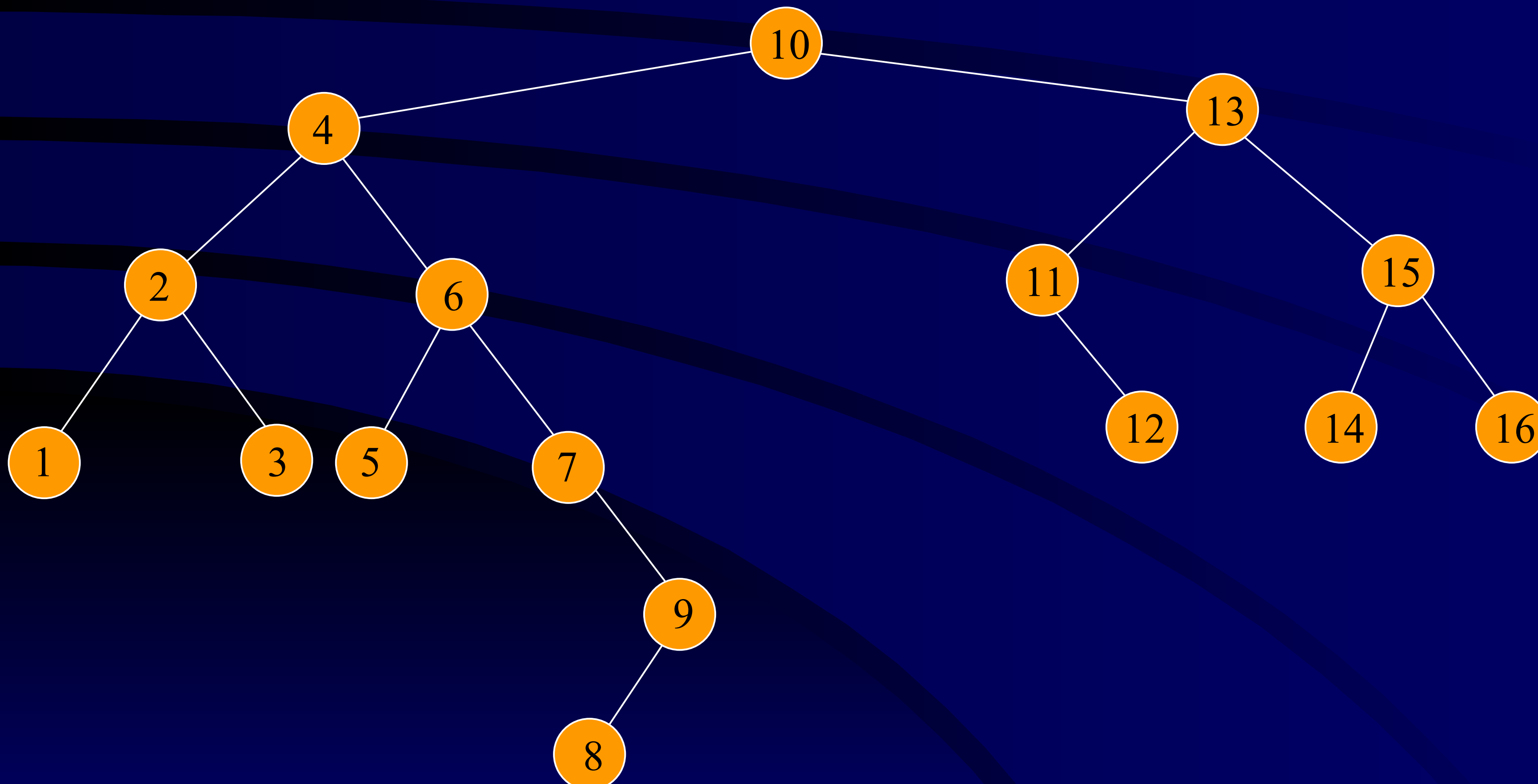


- Now insert 9 and 8.

AVL Tree Rotations

Double rotations:

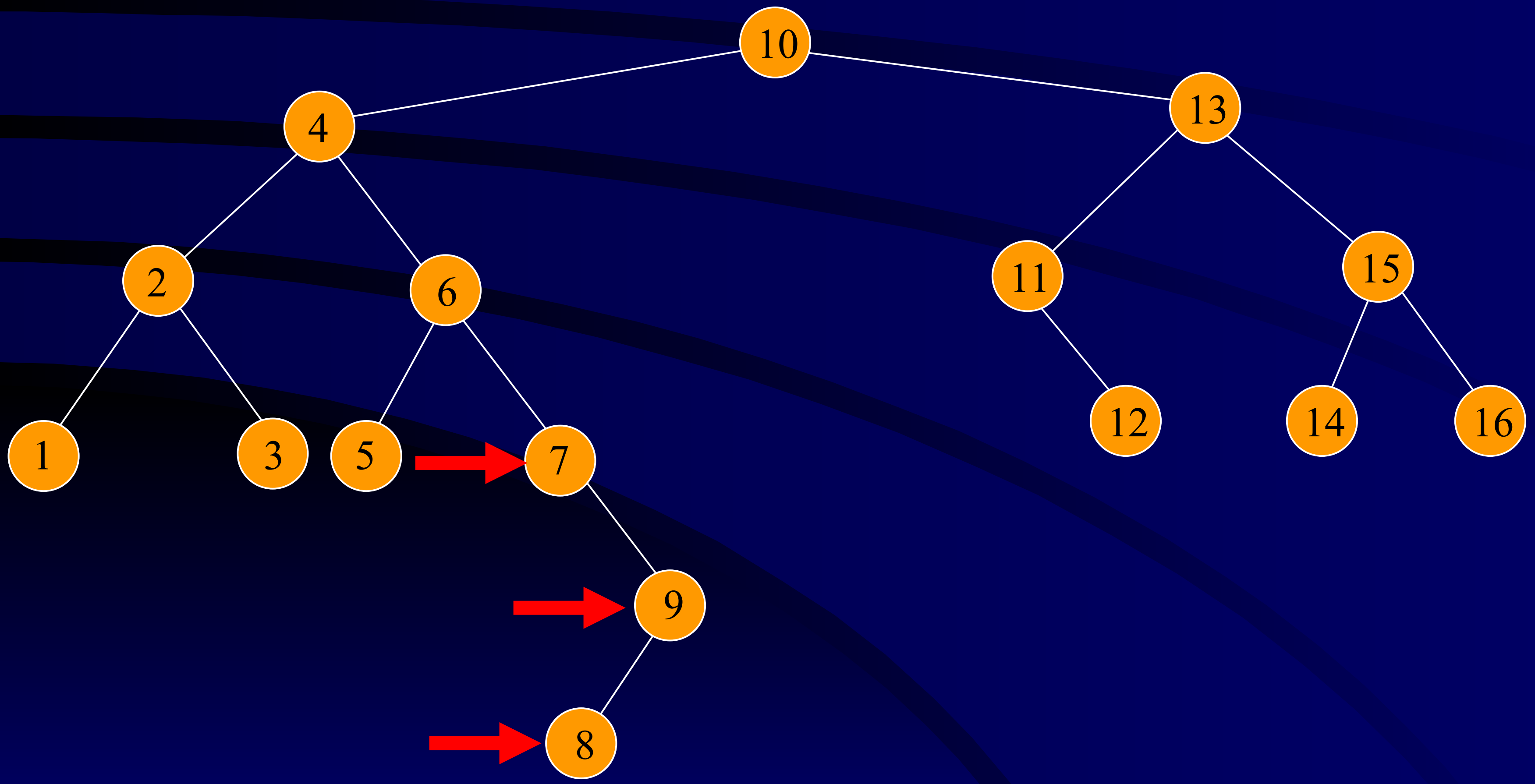
- AVL violation - rotate.



AVL Tree Rotations

Double rotations:

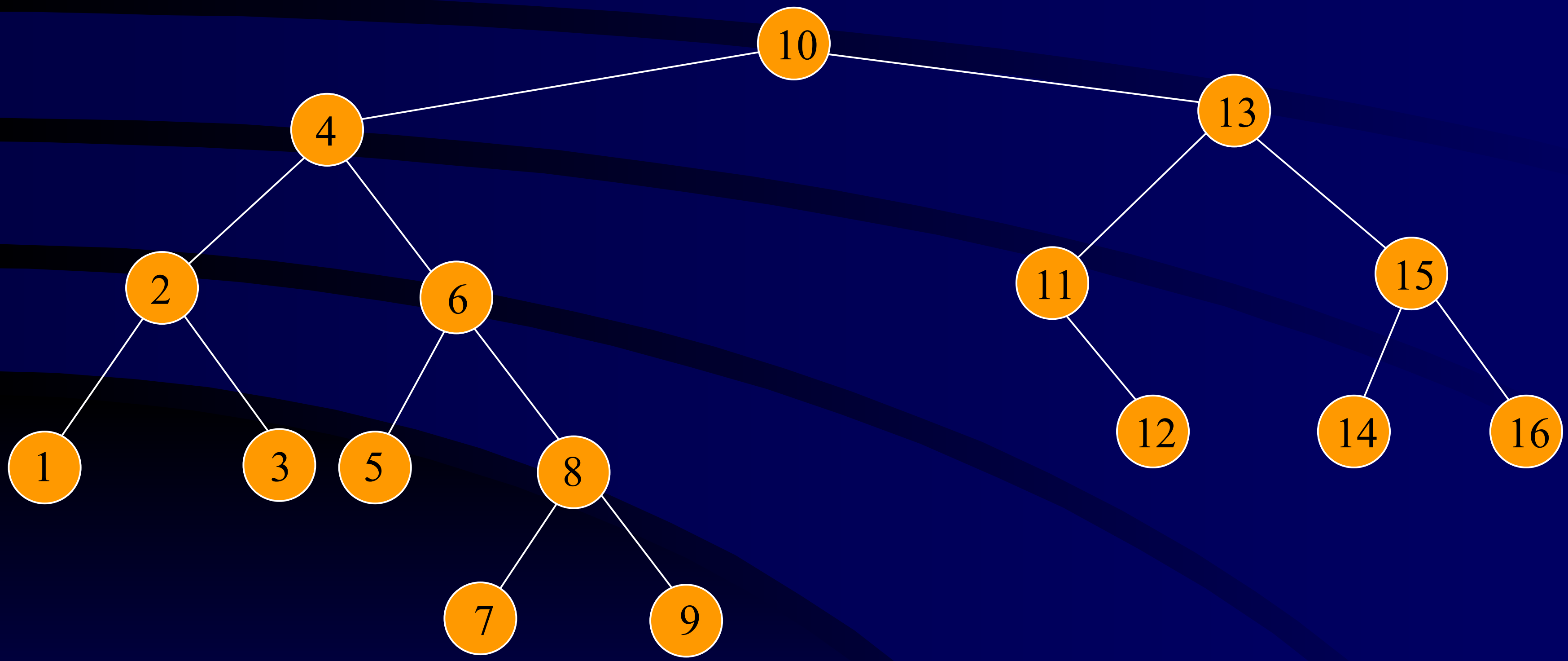
- Rotation type:



AVL Tree Rotations

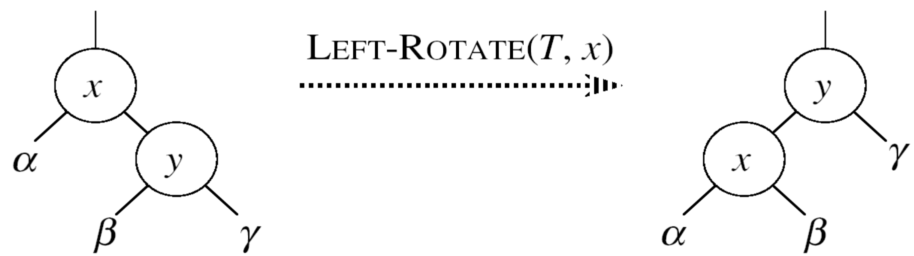
Final tree:

- Tree is almost perfectly balanced



LEFT-ROTATE(T, x)

```
y ← right[x]                    ► Set y
right[x] ← left[y]              ► y's left subtree becomes x's right subtree
if left[y] ≠ NIL
    then p[left[y]] ← x         ► Set the parent relation from left[y] to x
p[y] ← p[x]                     ► The parent of x becomes the parent of y
if p[x] = NIL
    then root[T] ← y
    else if x = left[p[x]]
            then left[p[x]] ← y
            else right[p[x]] ← y
left[y] ← x                     ► Put x on y's left
p[x] ← y                        ► y becomes x's parent
```



Example

Delete p.