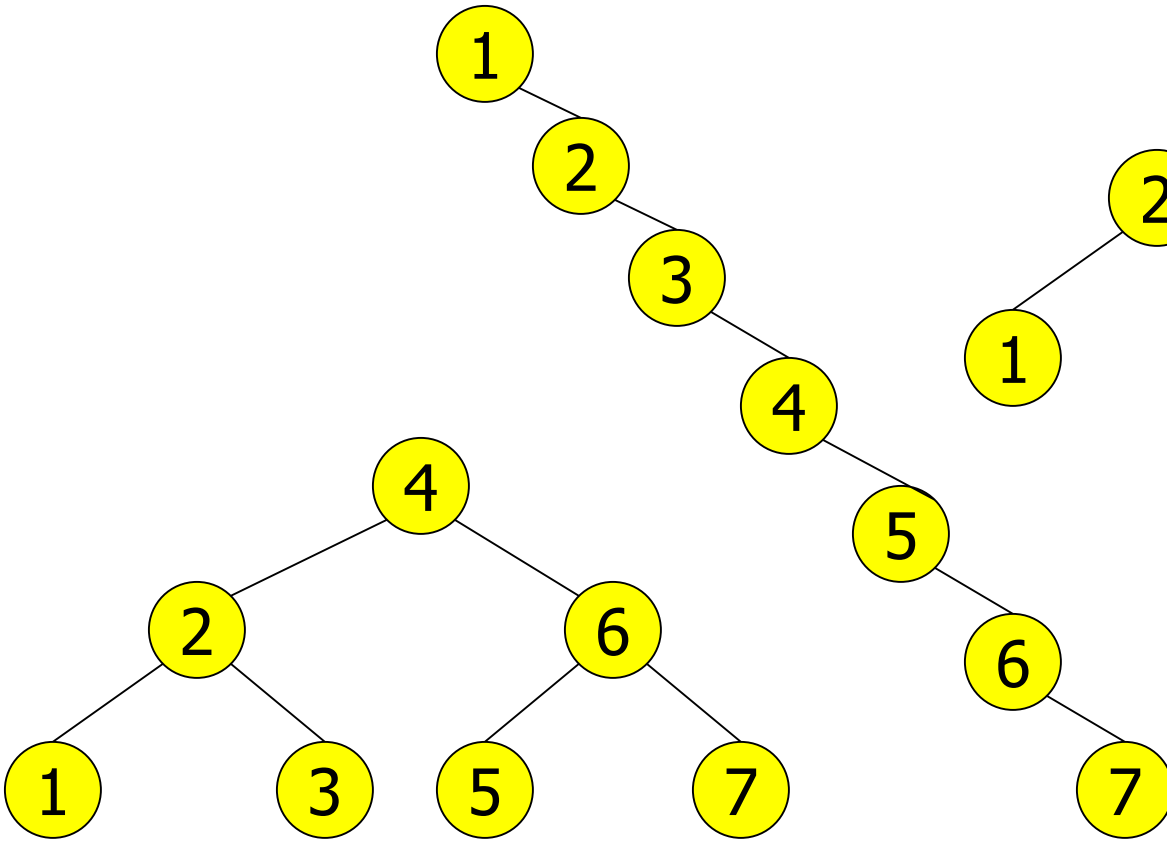
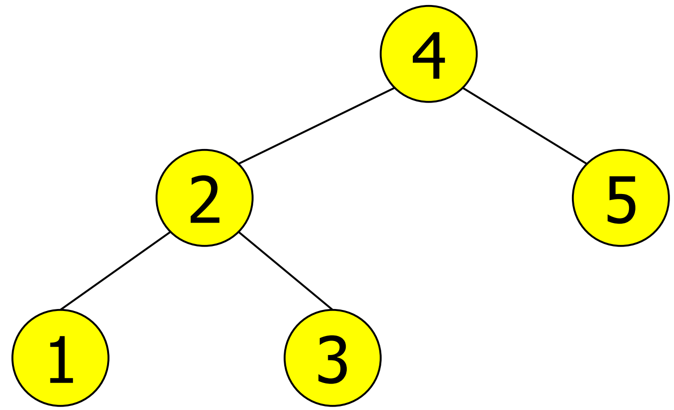
Data Structures

20. AVL Trees

Balanced and Unbalanced BST

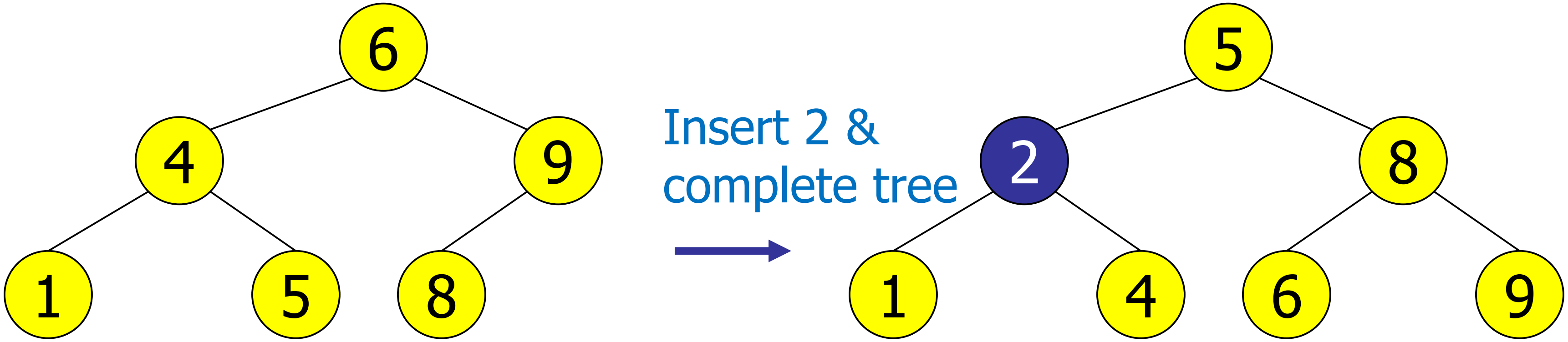




Is this “balanced”?

Balanced Tree

- Want a (almost) complete tree after every operation
  - Tree is complete except possibly in the lower levels towards right
- Maintenance of such a tree is expensive
  - For example, insert 2 in the tree on the left and then rebuild as a complete tree



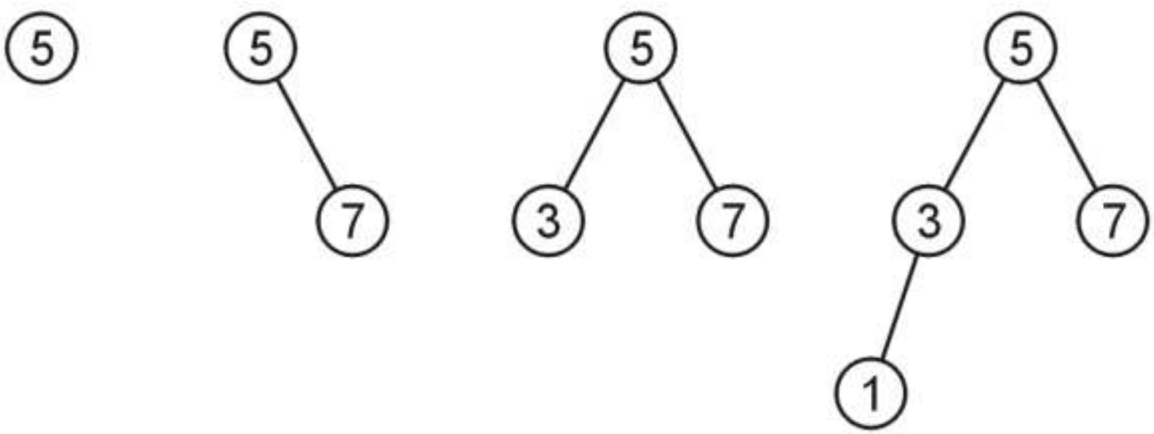
AVL Trees – Good but not Perfect Balance

- Named after Adelson-Velskii and Landis
- Balance is defined by comparing the height of the two sub-trees
- Recall:
  - An empty tree has height –1
  - A tree with a single node has height 0

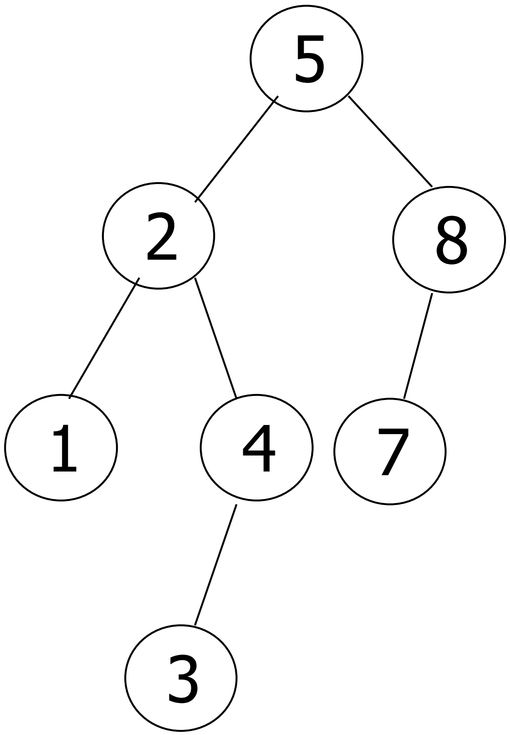
AVL Trees

- A **binary search tree** is said to be AVL **balanced** if:
  - The difference in the heights between the left and right sub-trees is at most 1, and
  - Both sub-trees are themselves AVL trees

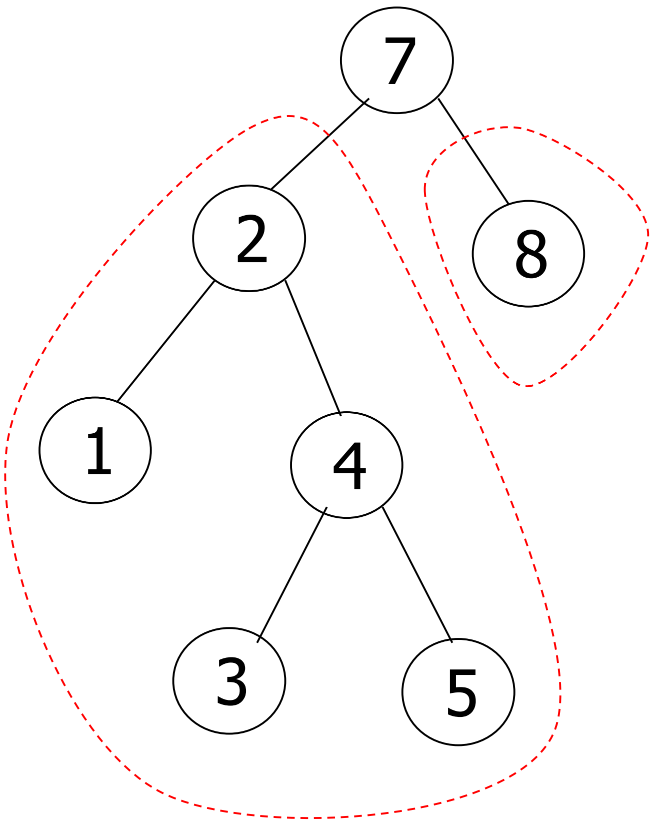
- AVL trees with 1, 2, 3 and 4 nodes



AVL Trees – Example



An AVL Tree



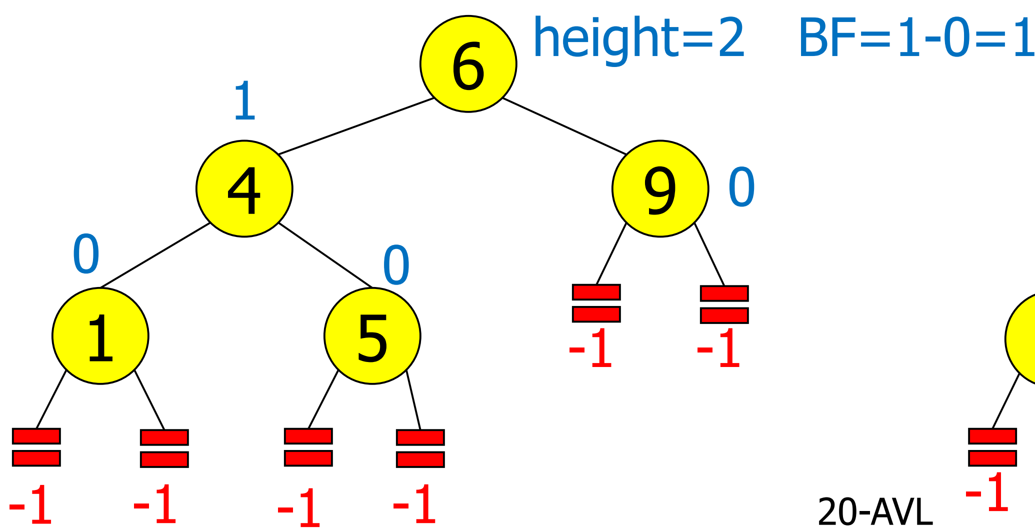
Not an AVL Tree

AVL Trees – Balance Factor

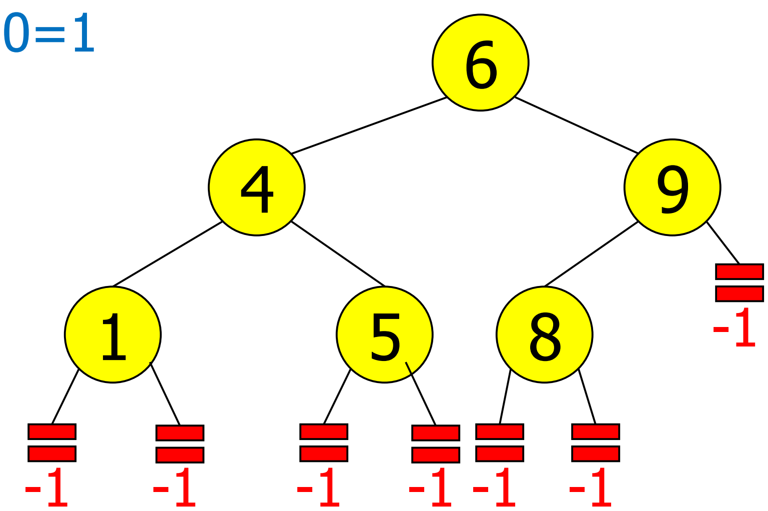
- An AVL tree has balance factor calculated at every node
  - Height of the left subtree minus the height of the right subtree
  - For an AVL tree, the balances of the nodes are always -1, 0 or 1

Height of node	=	h
Balance Factor (BF)	=	$h_{left} - h_{right}$
Empty height	=	-1

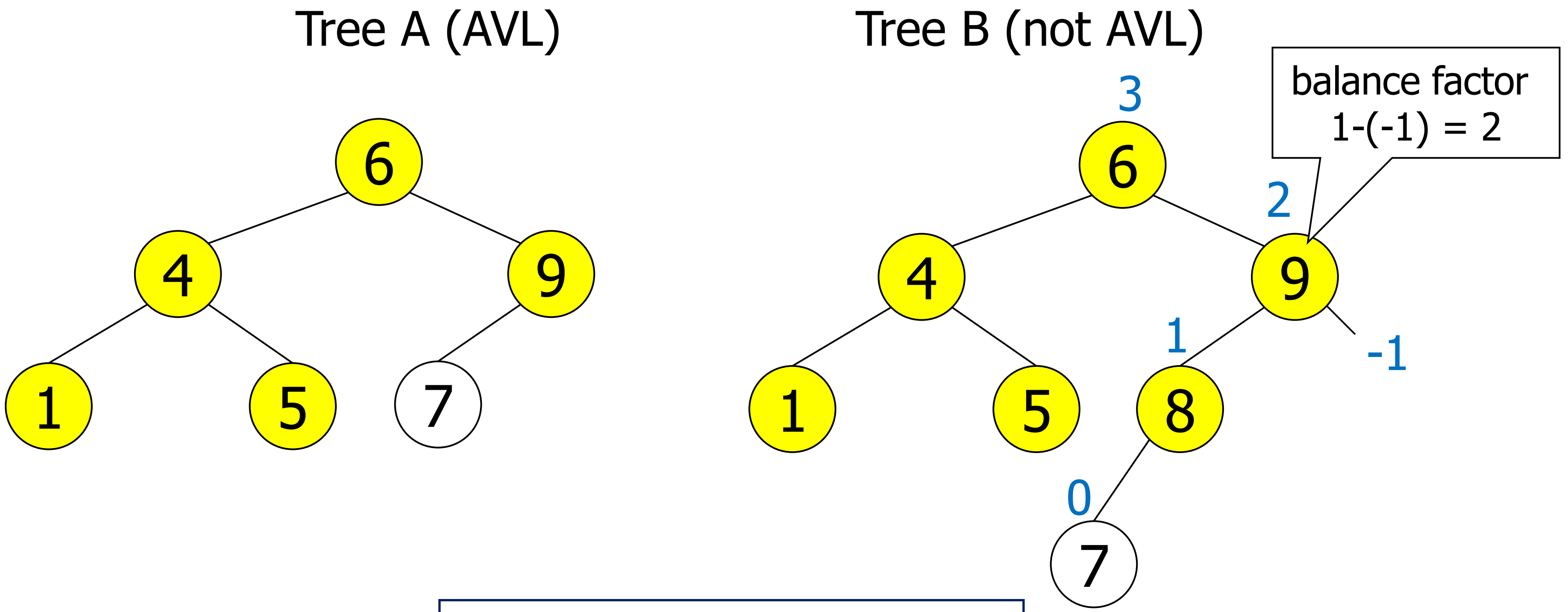
Tree A (AVL)



Tree B (AVL)



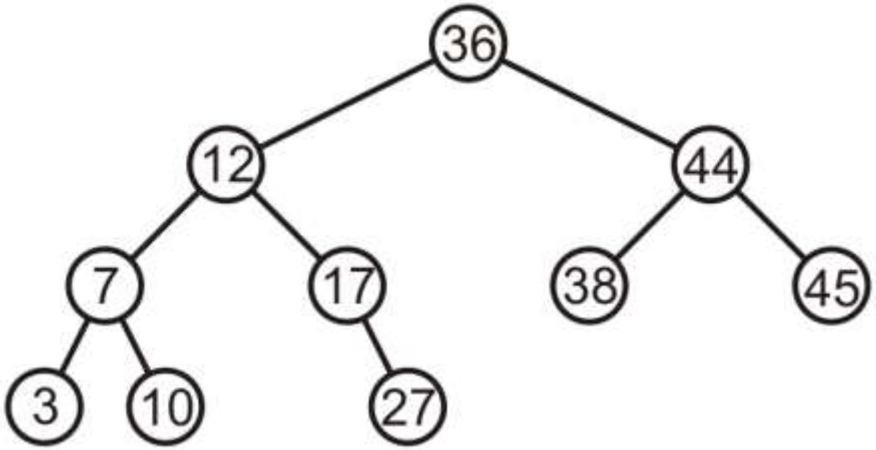
AVL Trees – Example



Height of node = h
Balance factor = h_{left}-h_{right}
Empty height = -1

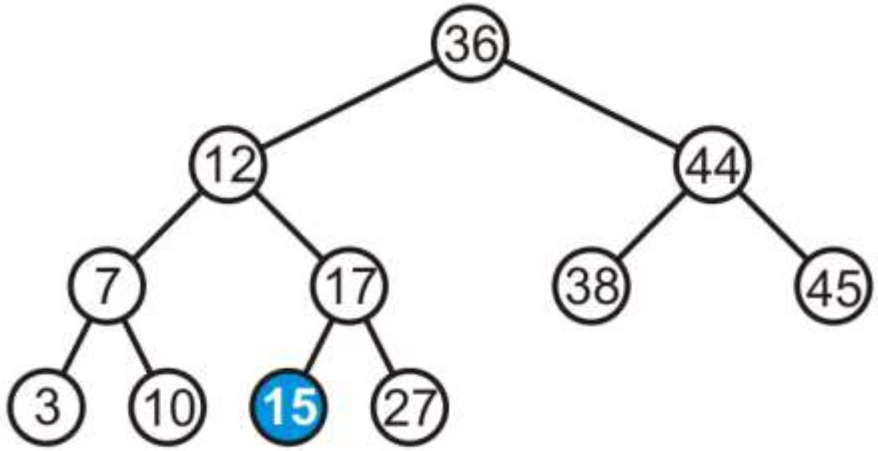
AVL Trees – Example

- Consider this AVL tree



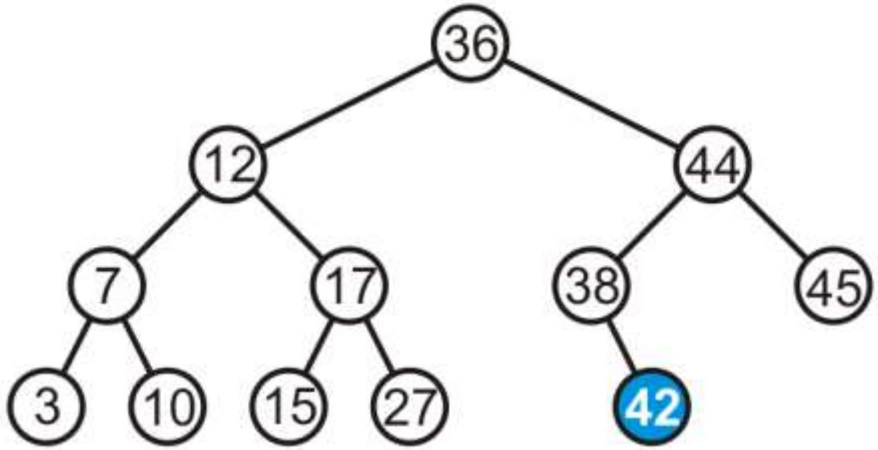
AVL Trees – Example

- Consider inserting 15 into this tree
  - In this case, the heights of none of the trees change
  - Tree remains balanced



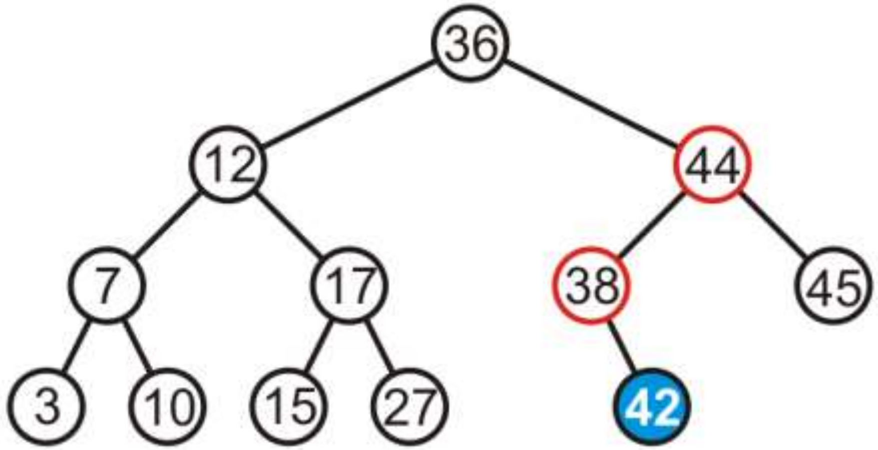
AVL Trees – Example

- Consider inserting 42 into this tree



AVL Trees – Example

- Consider inserting 42 into this tree
  - Height of two sub-trees rooted at 44 and 38 have increased by one
  - The tree is still balanced



AVL Trees

- To maintain the height balanced property of the AVL tree, it is necessary to
  - Perform a transformation on the tree, such that
  - In-order traversal of the transformed tree is the same as for the original tree (i.e., the new tree remains a binary search tree)

Transformation (Rotation) of AVL Trees

- Insert operations may cause balance factor to become 2 or –2 for some node

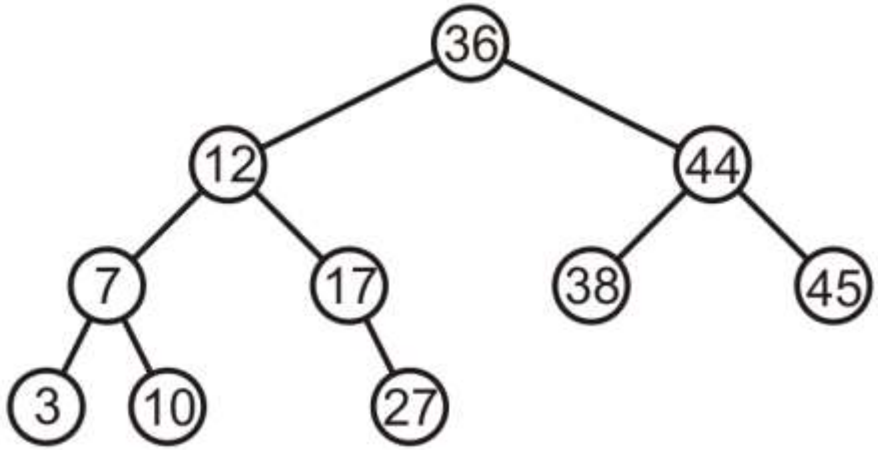
- Only nodes on the path from insertion point to root node have possibly change in height

- Follow the path up to the root, find the first node (i.e., deepest) whose new balance violates the AVL condition
  - Call this node "a"

- If a new balance factor (the difference h_{left}-h_{right}) is 2 or –2
  - Adjust tree by rotation around the node "a"

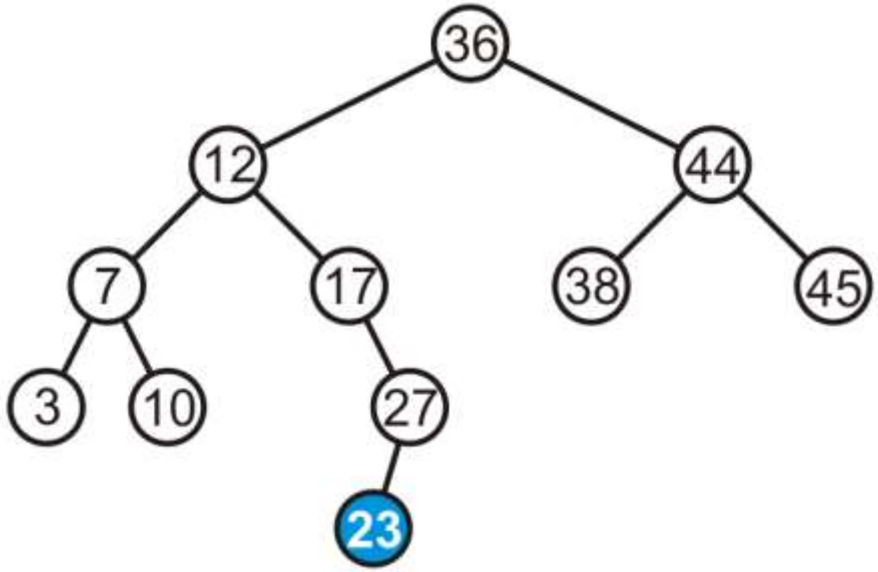
Balancing AVL Trees – Example

- If a tree is AVL balanced, for an insertion to cause an imbalance:
  - The heights of the sub-trees must differ by 1
  - The insertion must increase the height of the deeper sub-tree by 1



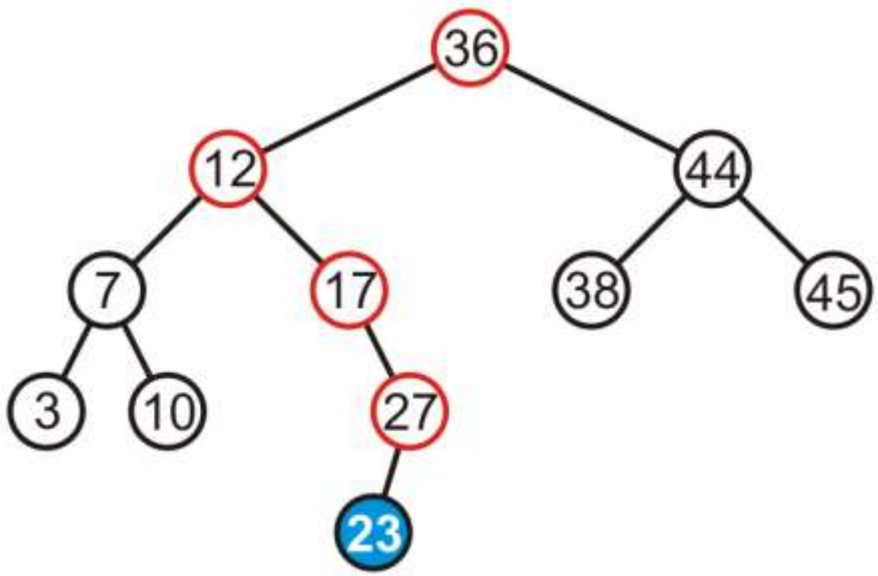
Balancing AVL Trees – Example

- Suppose we insert 23 into our initial tree



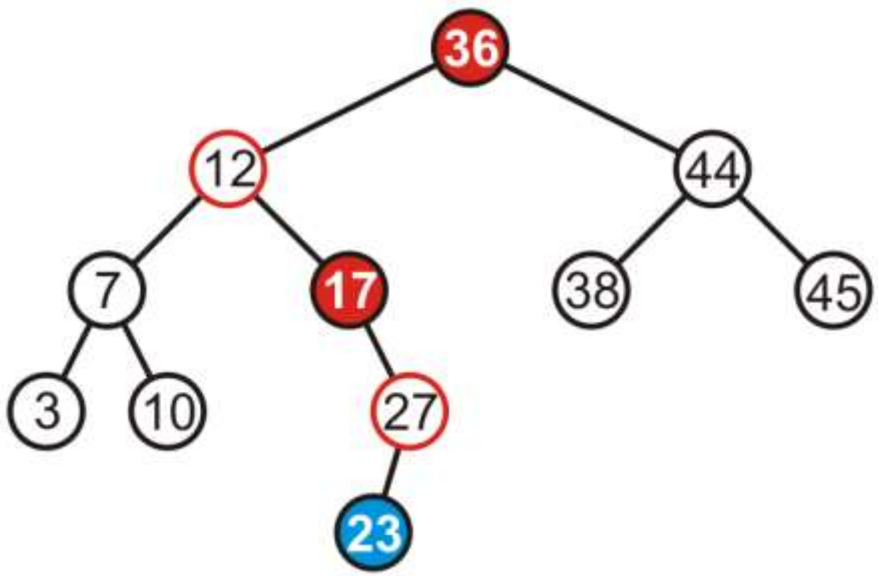
Balancing AVL Trees – Example

- The heights of each of the sub-trees from the insertion point to the root are increased by one



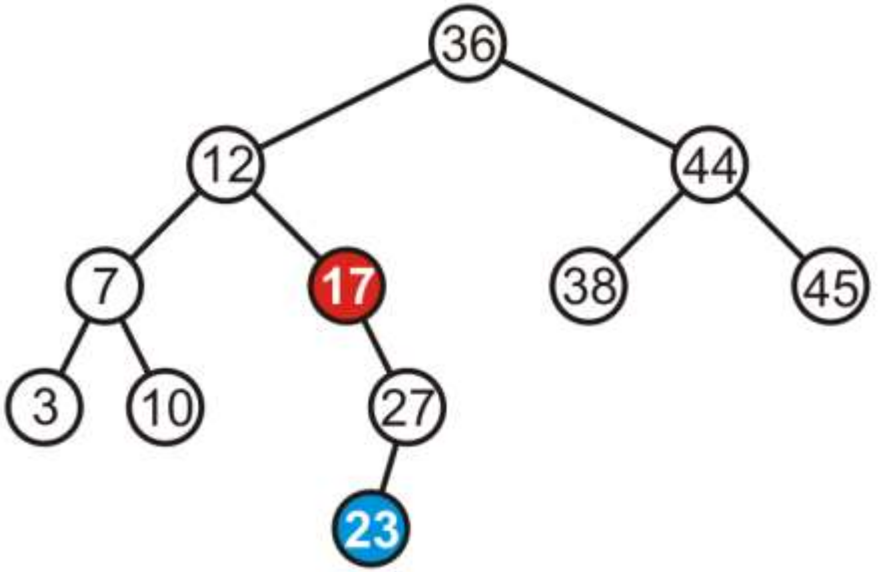
Balancing AVL Trees – Example

- Only two of the nodes are unbalanced, i.e., 17 and 36
  - Balance factor of 17 is -2
  - Balance factor of 36 is 2



Balancing AVL Trees – Example

- We only have to fix the imbalance at the lowest node



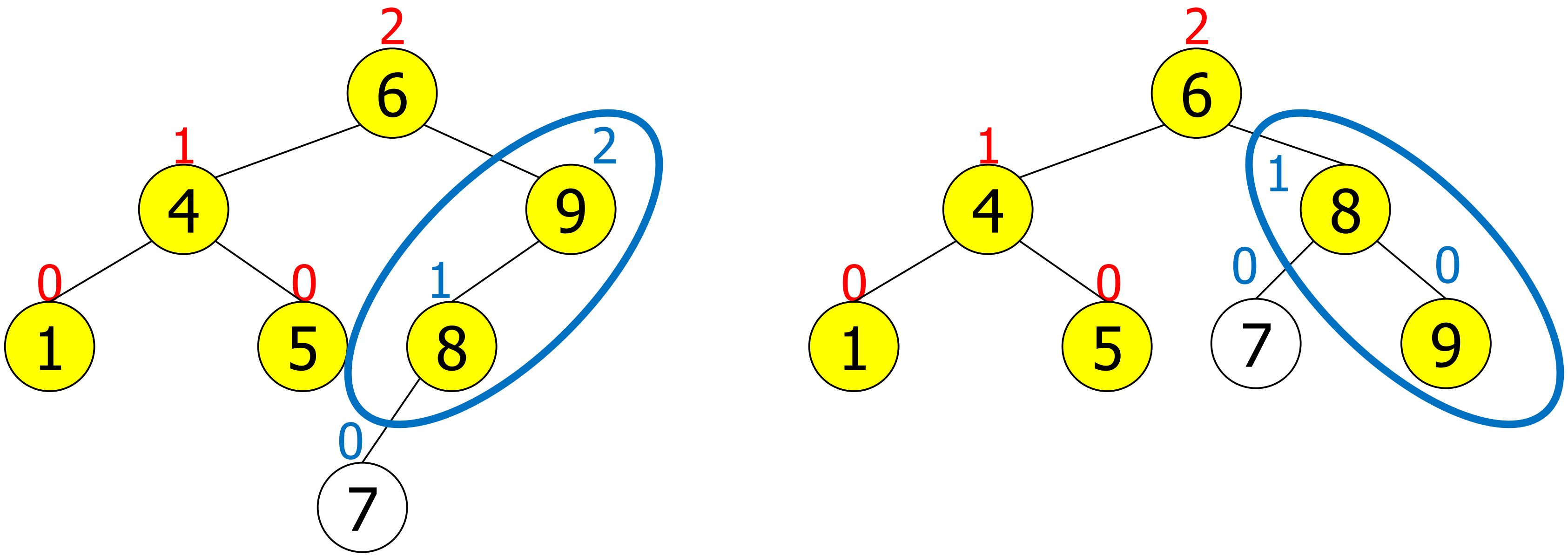
Fixing Imbalance By Rotation

- Let the node that needs rebalancing be “a”
- Imbalance during insertion may be handled using four cases

- Outside cases (Single Rotation)
  1. Right rotation (case RR)
  2. Left rotation (case LL)

- Inside cases (Double Rotation)
  3. Right-left rotation (case RL)
  4. Left-right rotation (case LR)

Single Rotation in an AVL Tree

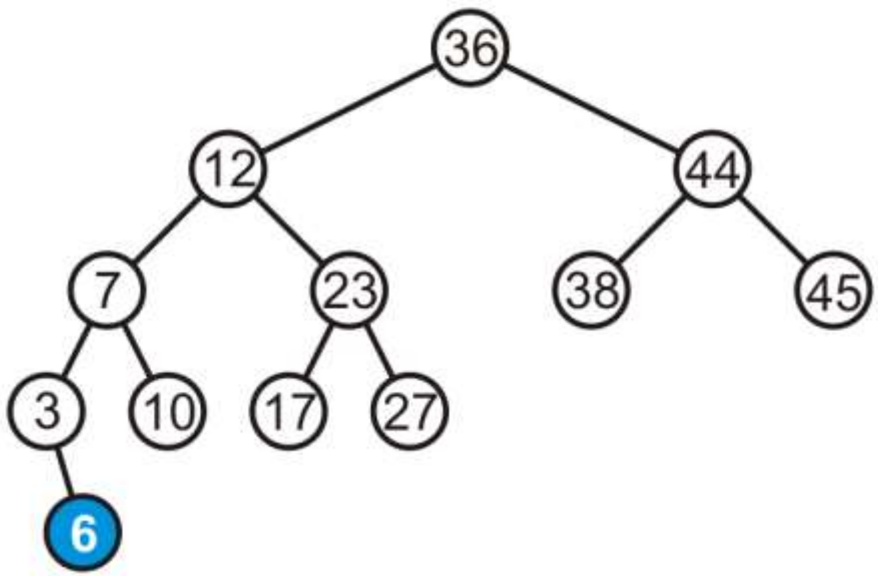


Right Rotation (RR) in an AVL Tree

- Node “b” becomes the new root
- Node “b” takes ownership of node “a”, as it's right child
- Node “a” takes ownership of node “b” right child (NULL if no child)
  - As left child of node “a”

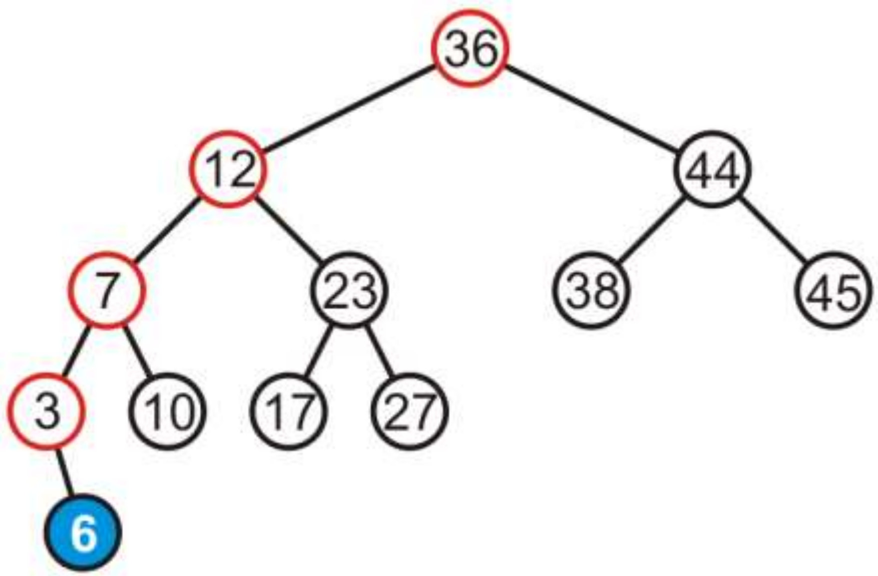
Right Rotation (RR) – Example

- Consider adding 6



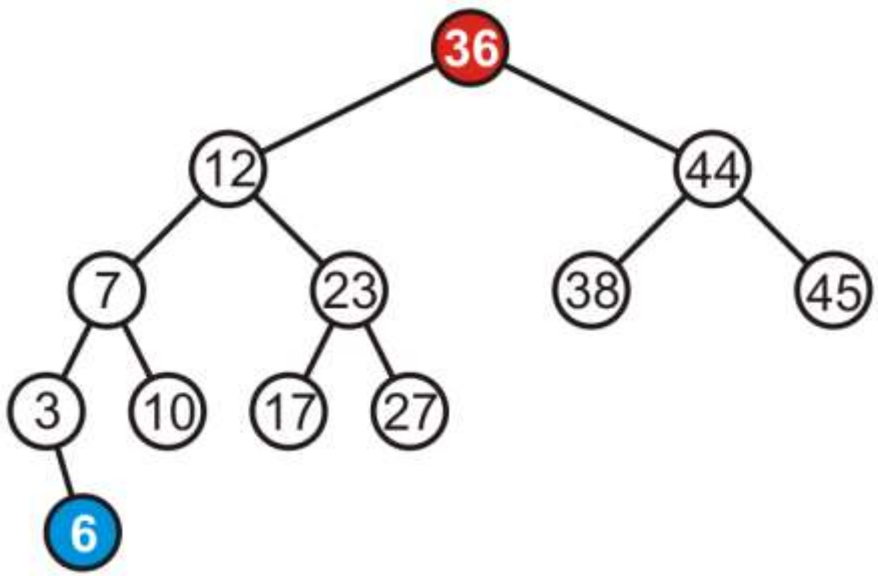
Right Rotation (RR) – Example

- Height of each of the trees in the path back to the root are increased by one



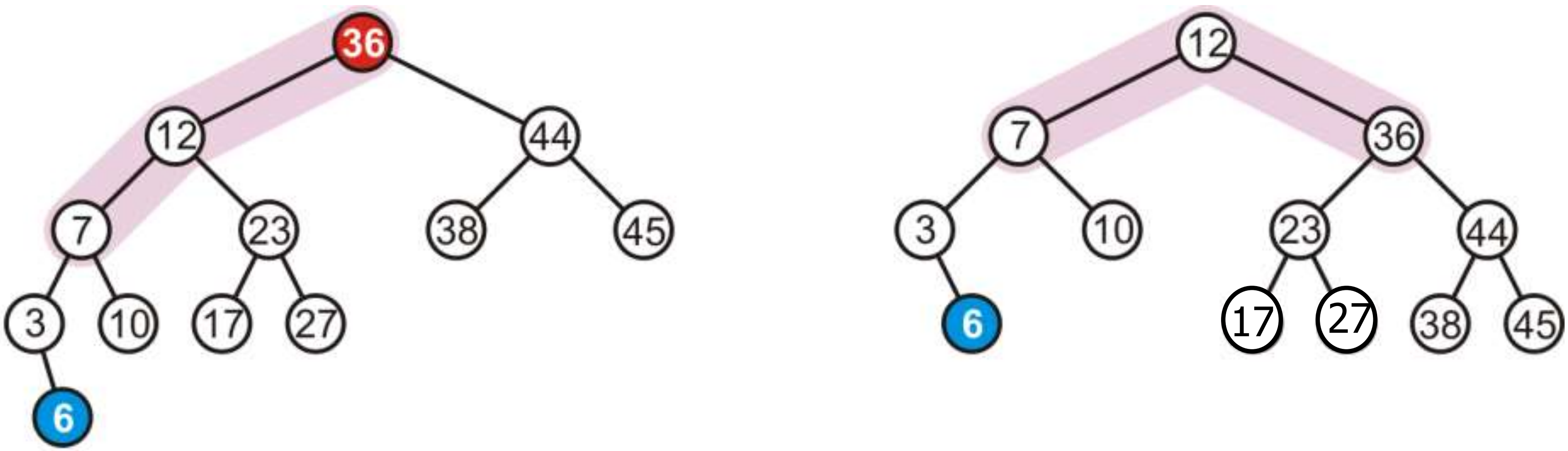
Right Rotation (RR) – Example

- Height of each of the trees in the path back to the root are increased by one
  - Only root node (i.e., 36) violates the balancing factor



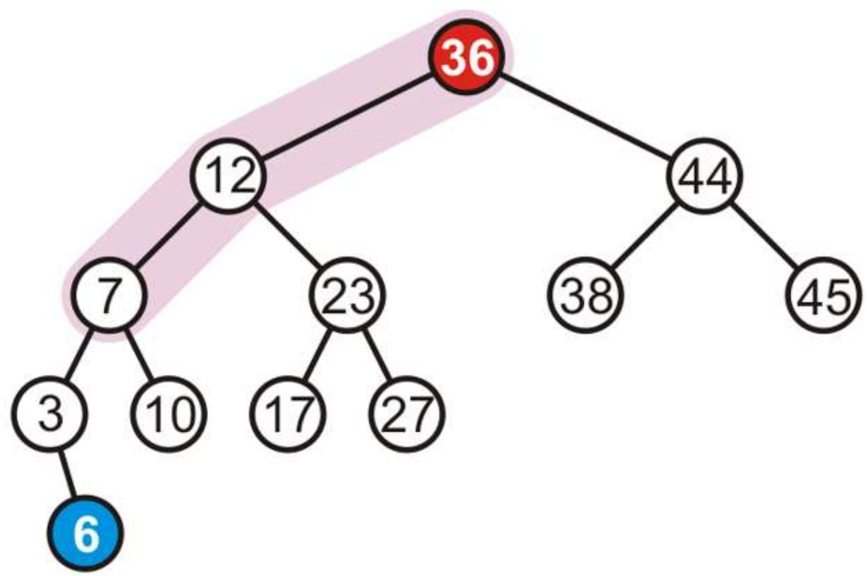
Right Rotation (RR) – Example

- To fix the imbalance, we perform right rotation of root (i.e., 36)



When to Perform Right Rotation (RR)?

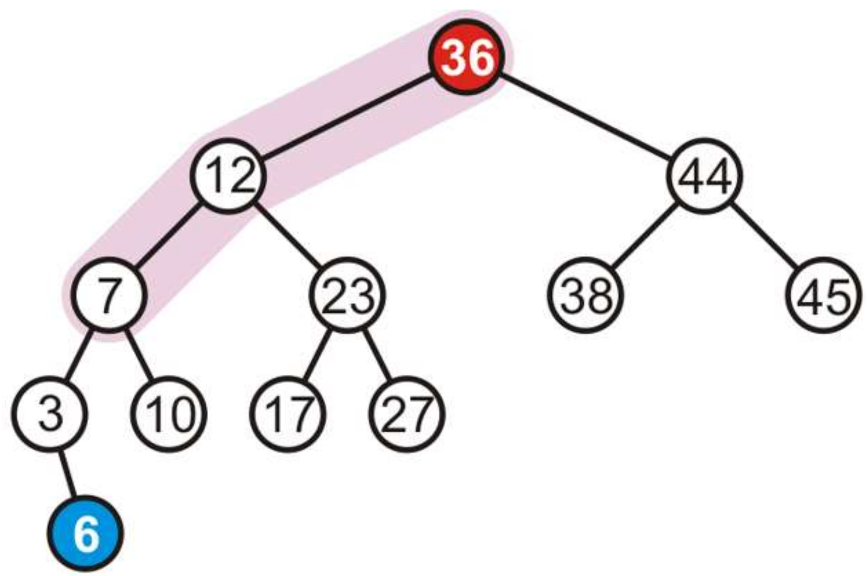
- Let the node that needs rebalancing be “a”
- Case RR
  - Insertion into left subtree of left child of node “a”
  - Left tree is heavy (i.e., $h_{left} > h_{right}$)



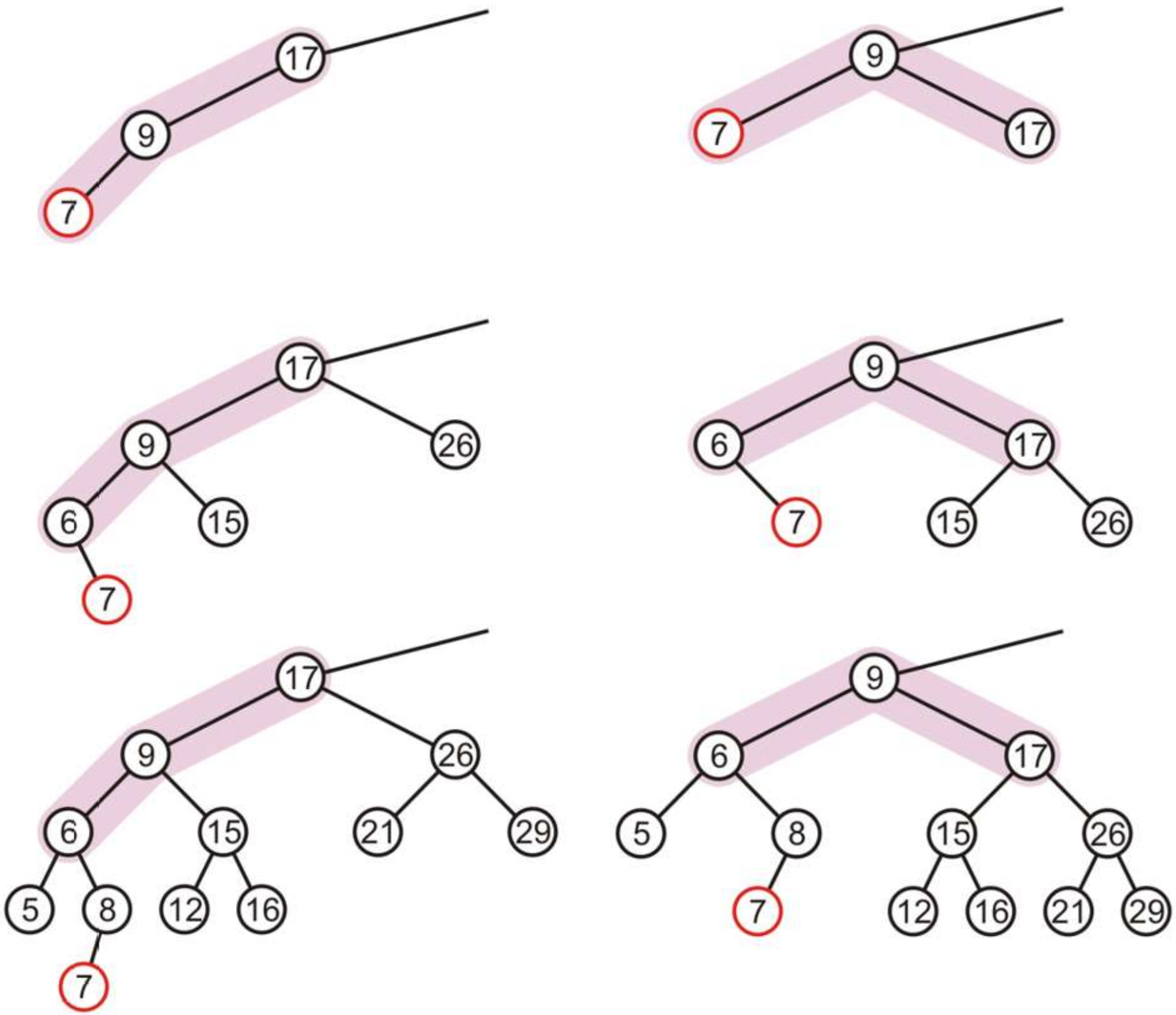
When to Perform Right Rotation (RR)

- Let the node that needs rebalancing be a

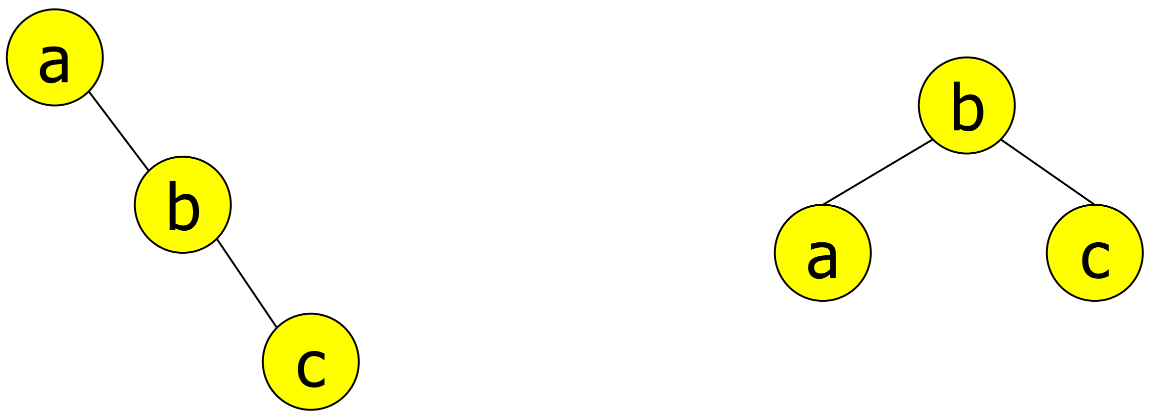
- Case RR
  - Insertion into left subtree of left child of node a (**RR**)
  - Left tree is heavy (i.e., $h_{left} > h_{right}$)



Right Rotation (RR) – Examples



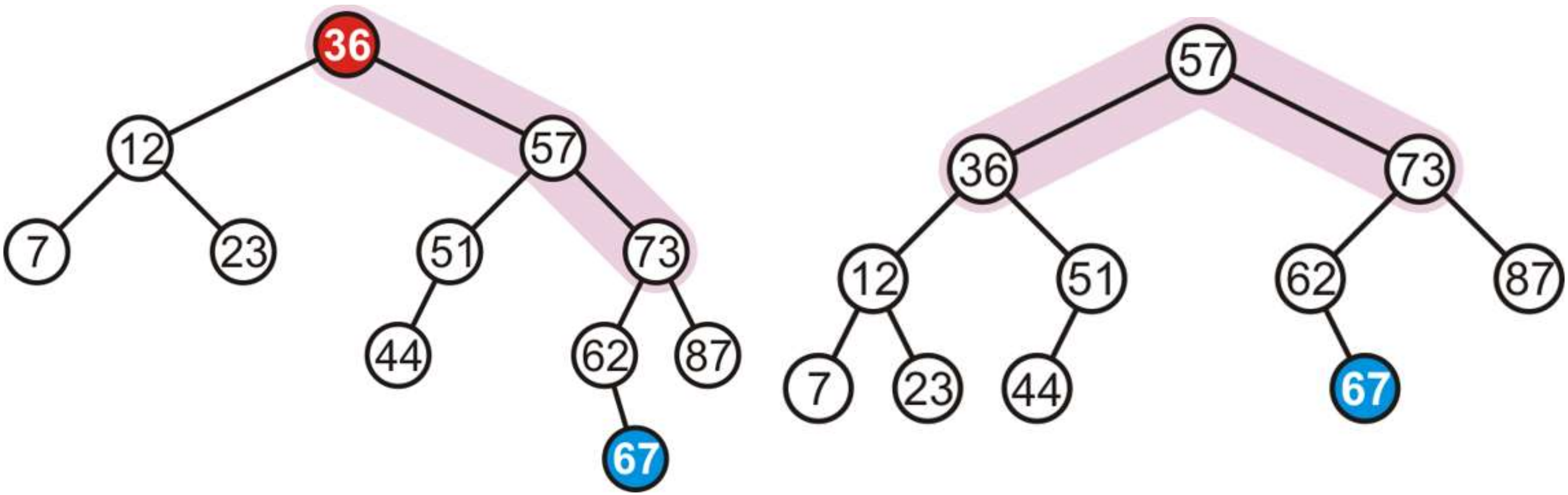
Left Rotation (LL) in an AVL Tree



- Node “b” becomes the new root
- Node “b” takes ownership of node “a” as its left child
- Node “a” takes ownership of node “b” left child (NULL if no child)
  - As right child of node “a”

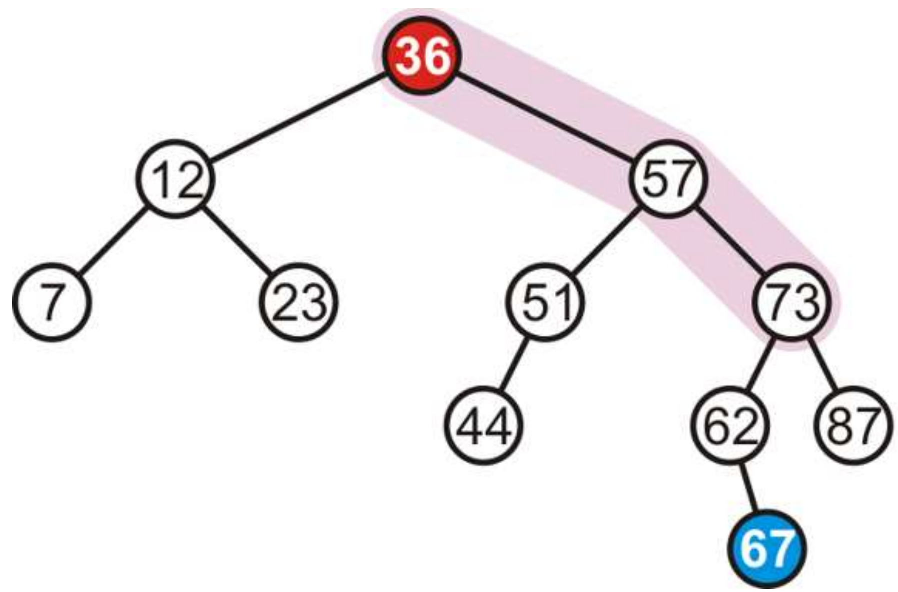
Left Rotation (LL) – Example

- Consider adding 67
  - To fix the imbalance, we perform left rotation of root (i.e., 36)



When to Perform Left Rotation (LL)

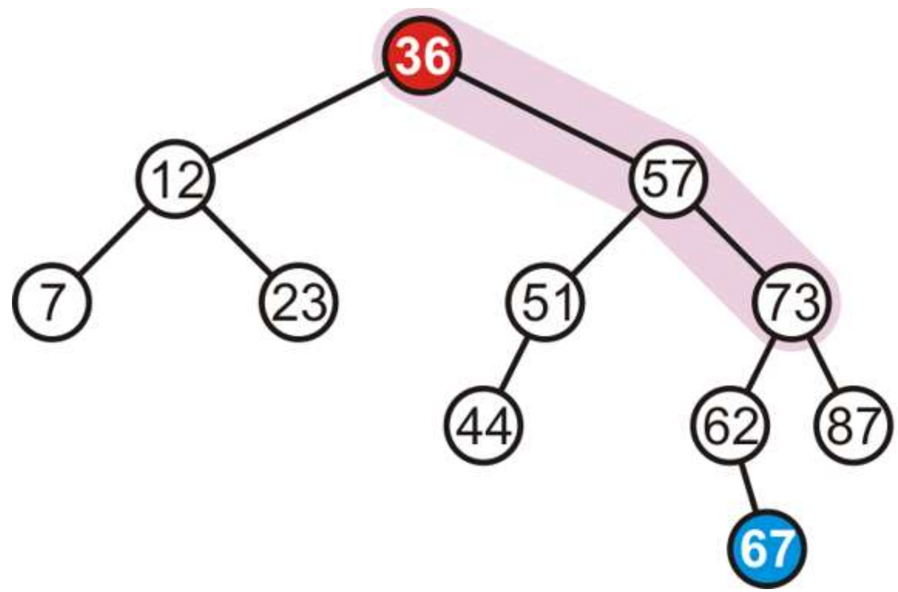
- Let the node that needs rebalancing be “a”
- Case LL
  - Insertion into right subtree of right child of node a
  - Right tree is heavy (i.e., $h_{left} < h_{right}$)



When to Perform Left Rotation (LL)

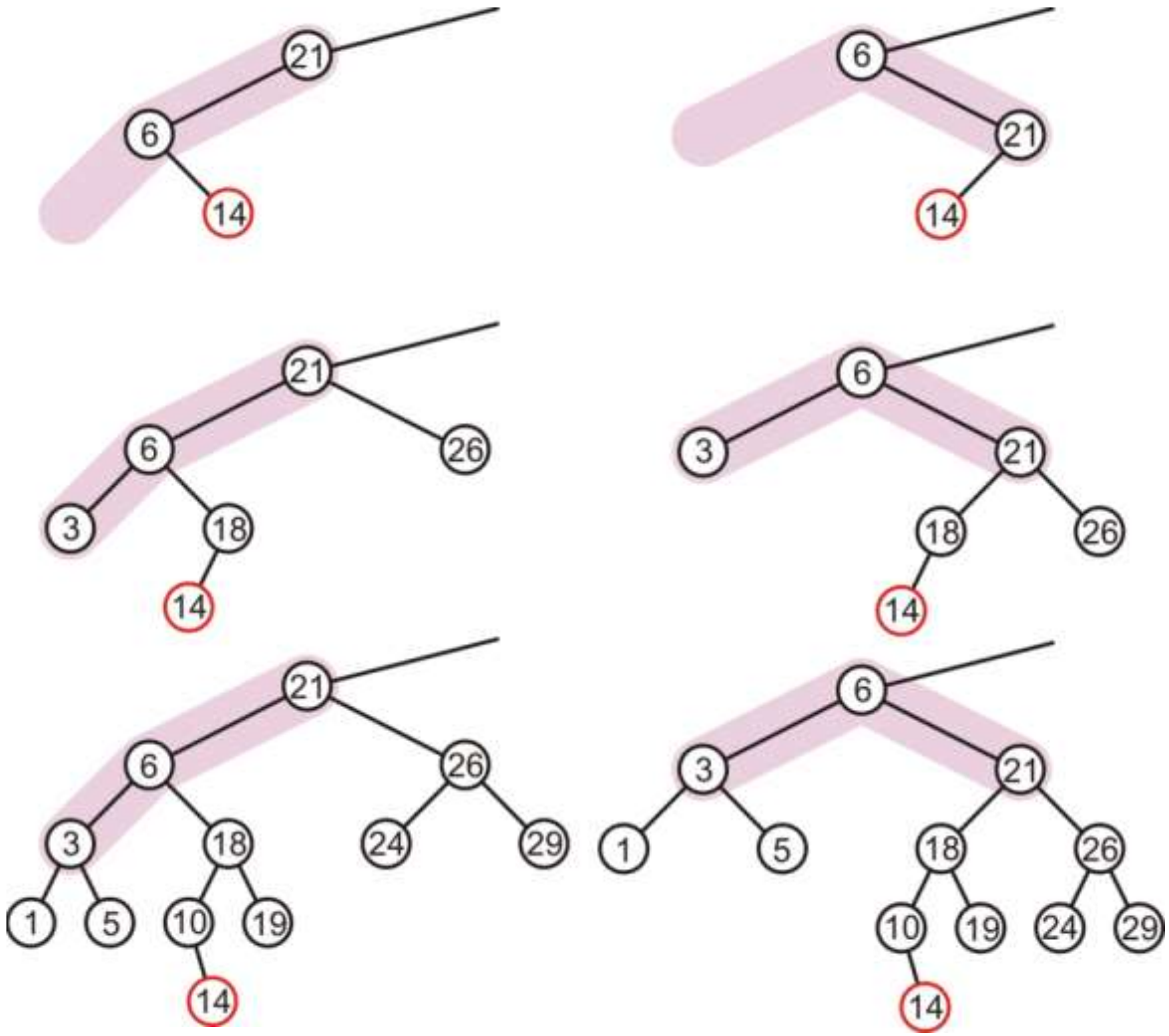
- Let the node that needs rebalancing be a

- Case LL
  - Insertion into right subtree of right child of node a (**LL**)
  - Right tree is heavy (i.e., $h_{left} < h_{right}$)

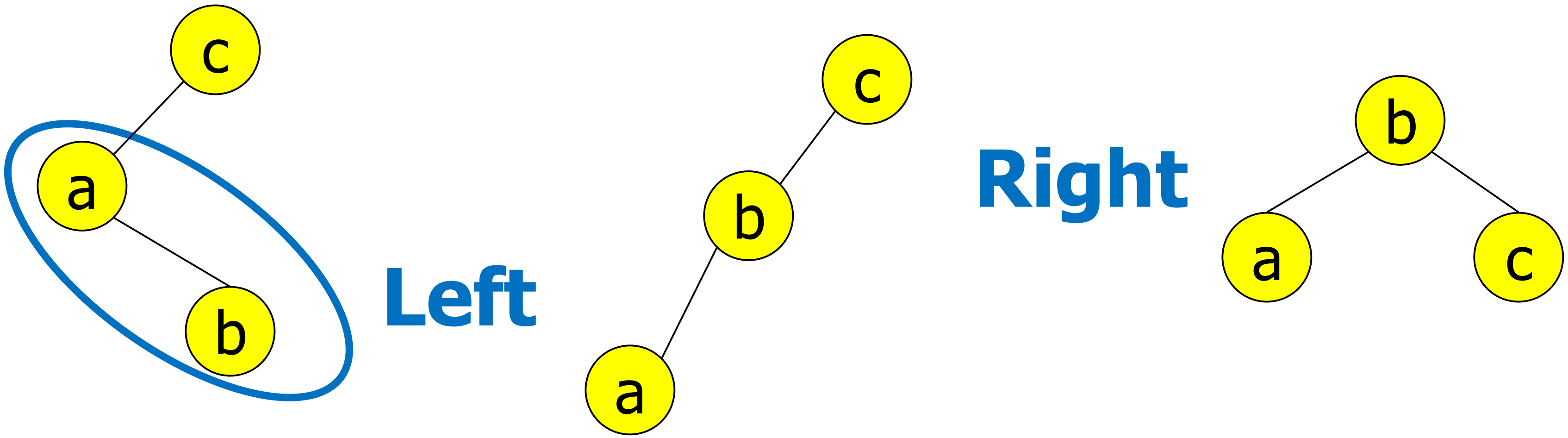


Single Rotation May Be Insufficient

- The imbalance is just shifted to the other side

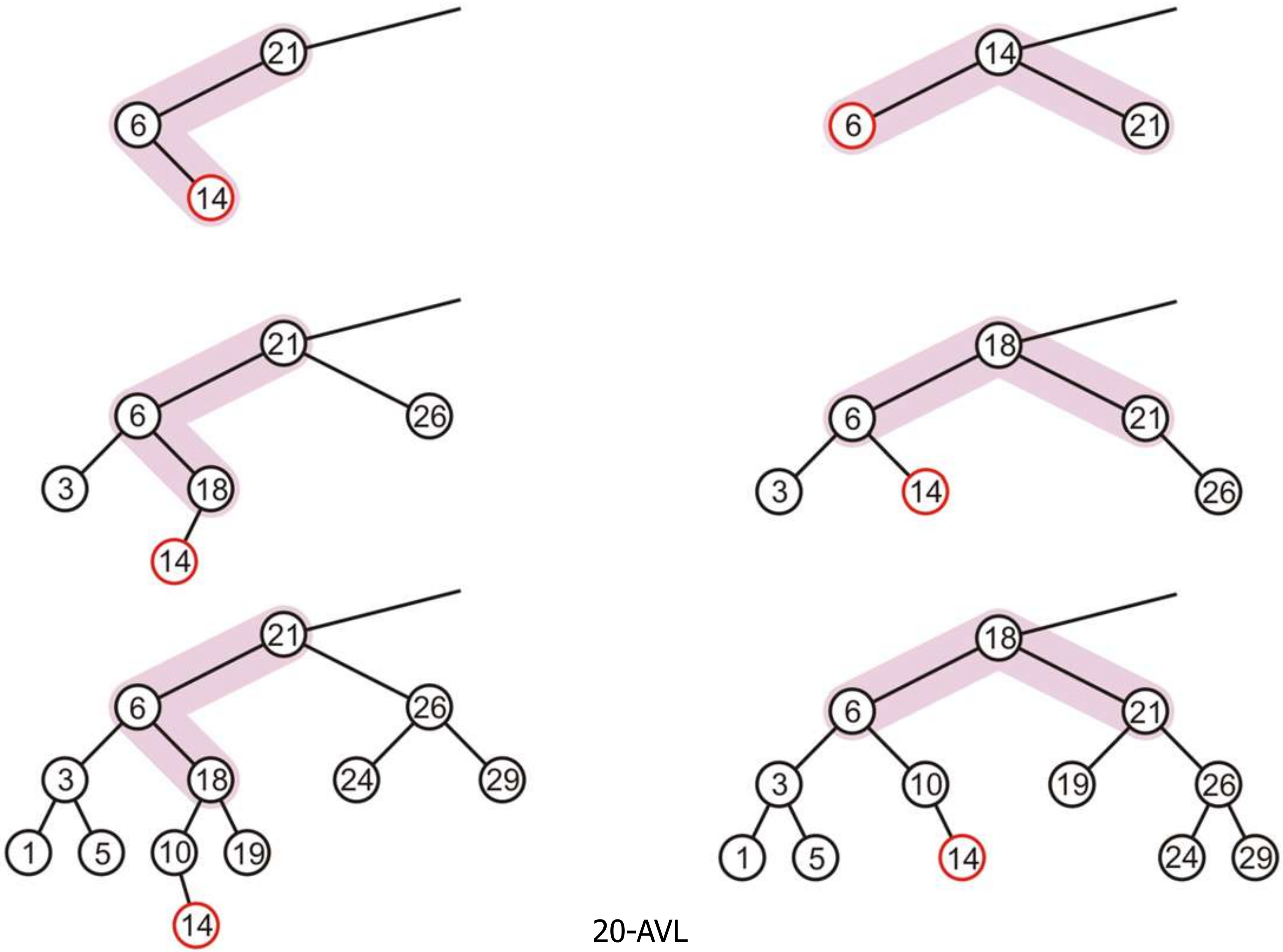


Right-Left Rotation (RL) or "Double Right"



- Perform a left rotation on the left subtree
- Node “b” becomes the new root
- Node “b” takes ownership of node “c” as its right child
- Node “c” takes ownership of node “b” right child
  - As its left child

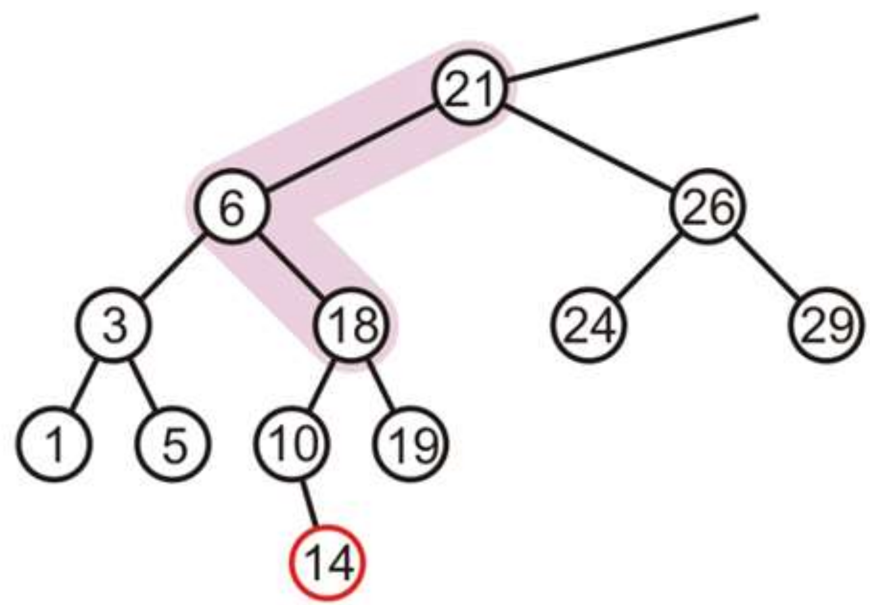
Right-Left Rotation (RL) or "Double Right" – Example



When to Perform Right-Left Rotation (RL)

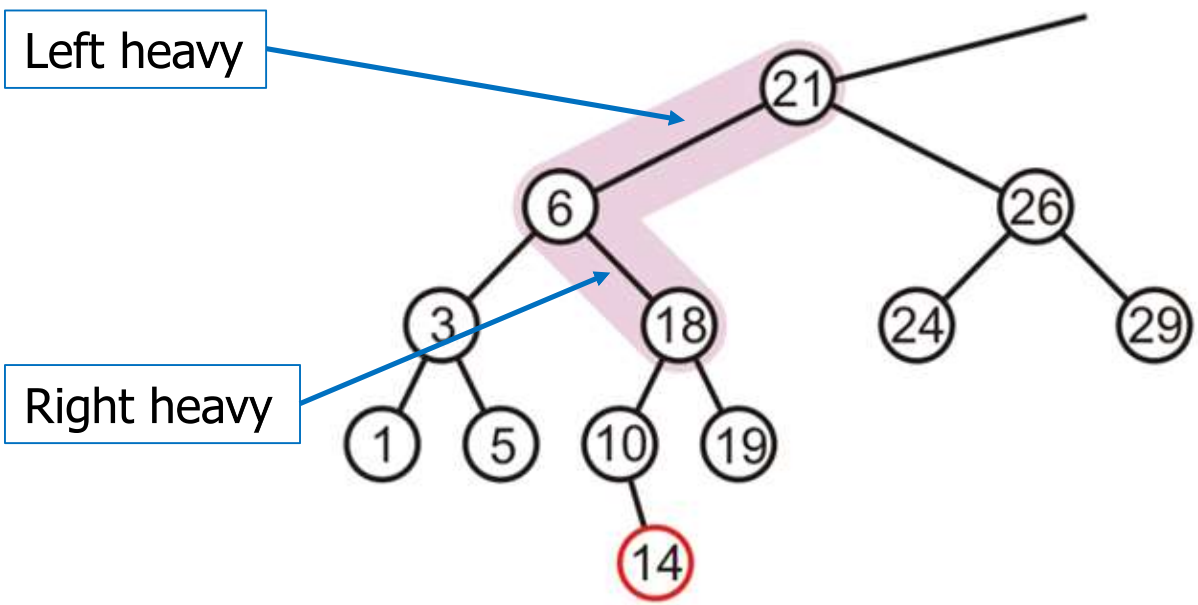
- Let the node that needs rebalancing be "a"

- Case RL
  - Insertion into right subtree of left child of node "a"

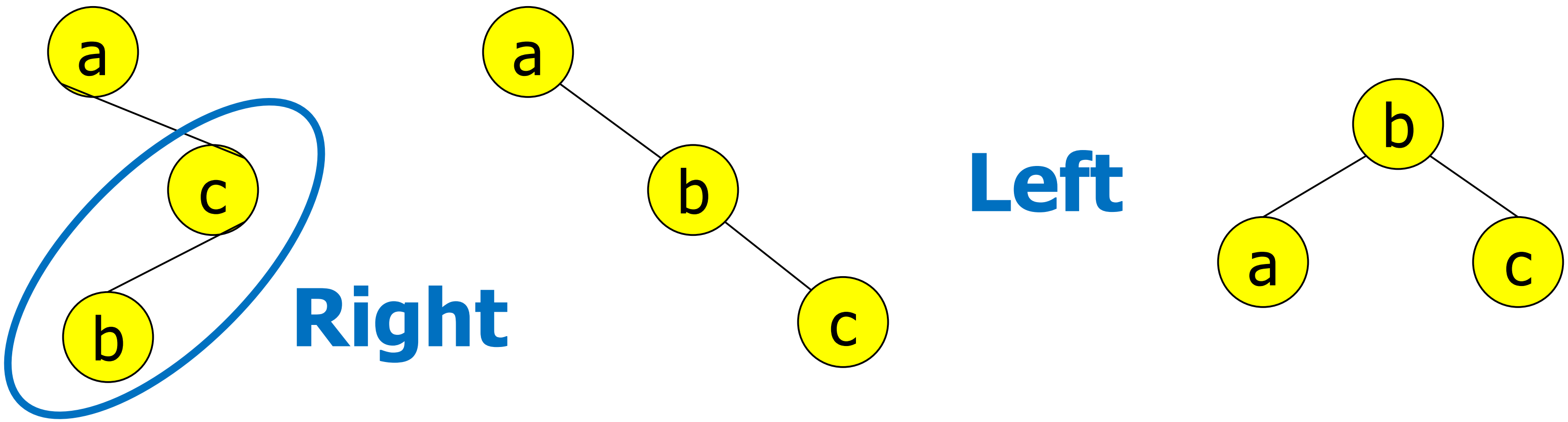


When to Perform Right-Left Rotation (RL)

- Let the node that needs rebalancing be a
- Case RL
  - Insertion into right subtree of left child of node a (**RL**)



Left-Right Rotation (LR) or "Double Left"

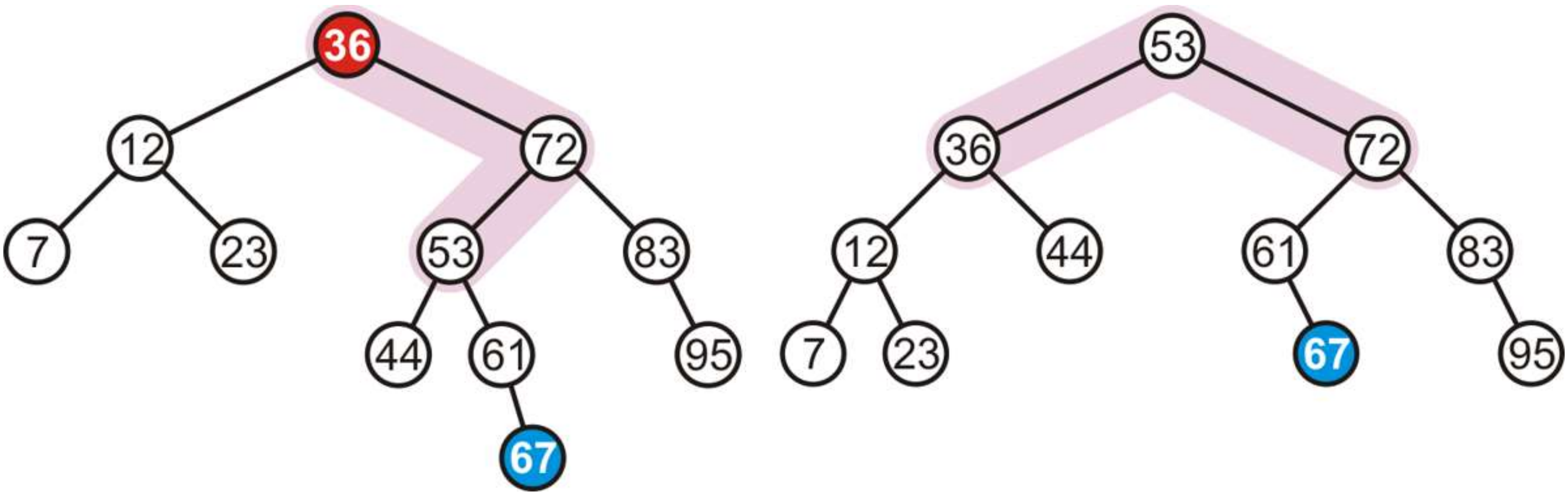


- Perform a right rotation on the right subtree

- Node “b” becomes the new root
- Node “b” takes ownership of node “a” as its left child
- Node “a” takes ownership of node “b” left child
  – As its right child

Left-Right Rotation (LR) or "Double Left" – Example

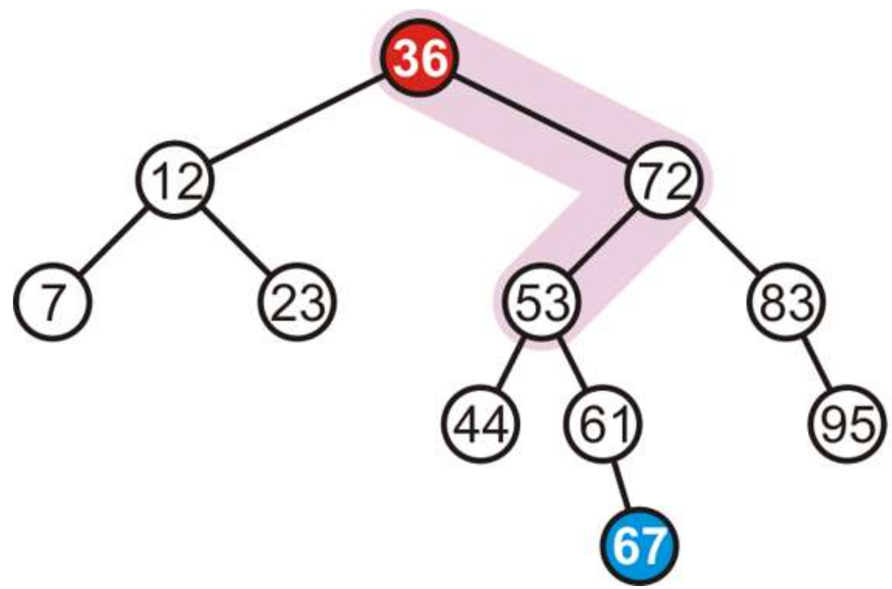
- Consider adding 67
  - To fix the imbalance, we perform left-right (LR) rotation of root



When to Perform Left-Right Rotation (LR)

- Let the node that needs rebalancing be a

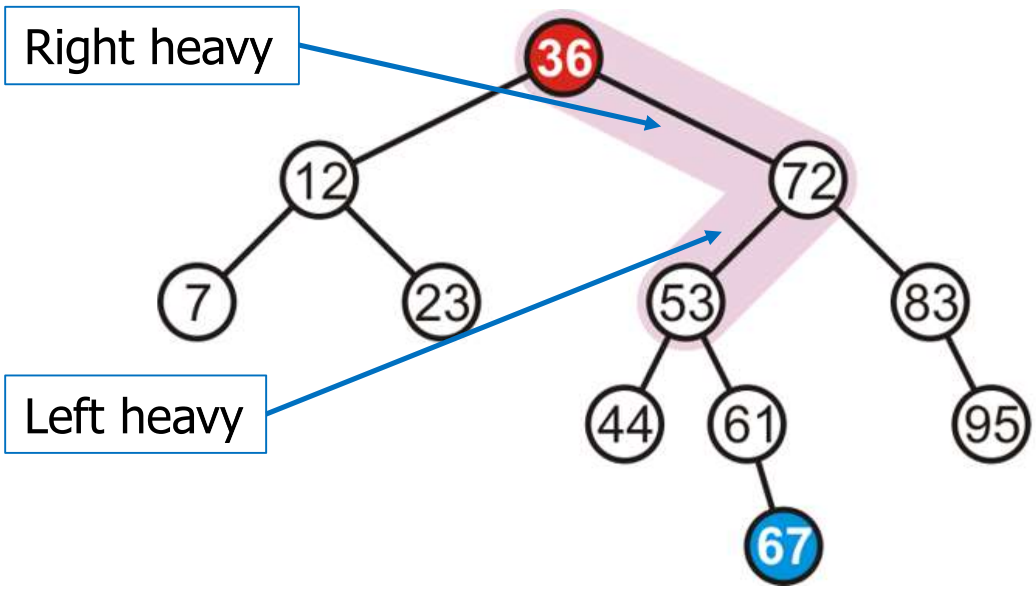
- Case LR
  - Insertion into left subtree of right child of node a



When to Perform Left-Right Rotation (LR)

- Let the node that needs rebalancing be a

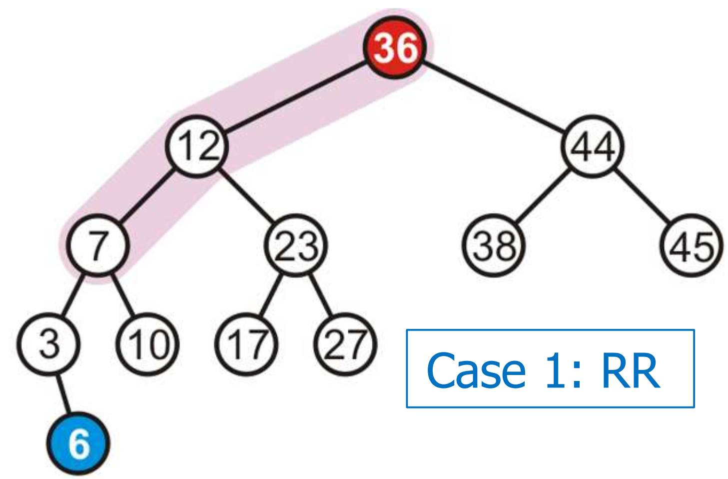
- Case LR
  - Insertion into left subtree of right child of node a (**LR**)



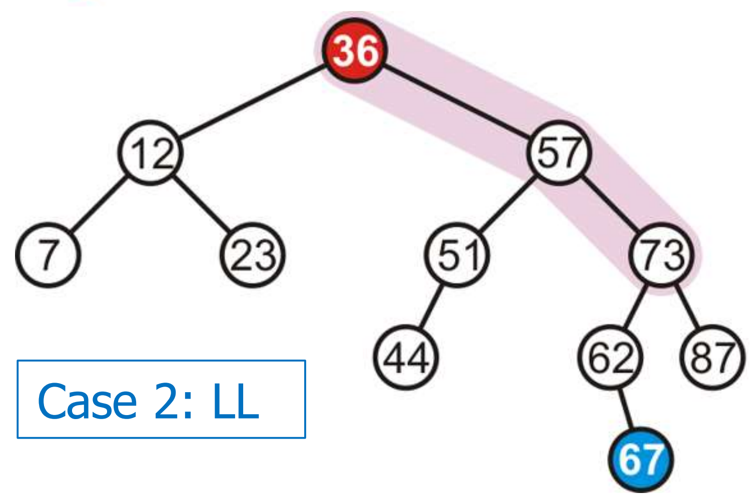
Summary: How And When To Rotate?

- Let the node that needs rebalancing be “a”
- Violation during insertion may occur in four cases

- Outside cases (Single Rotation)
  1. Insertion into left subtree of left child of node a (case RR)



  2. Insertion into right subtree of right child of node a (case LL)

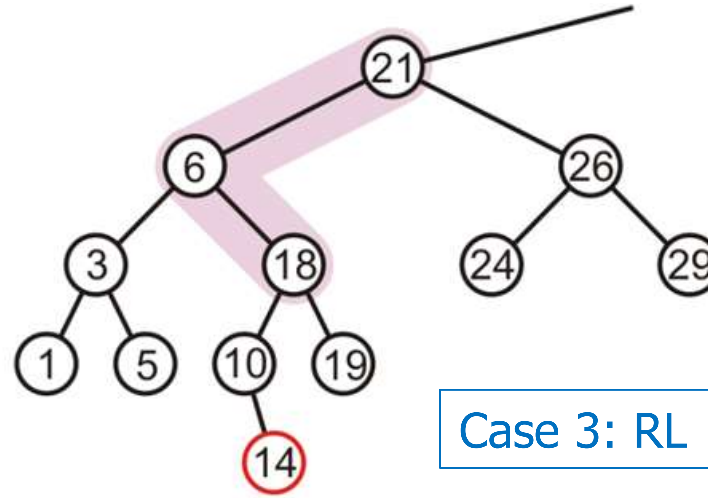


Summary: How And When To Rotate?

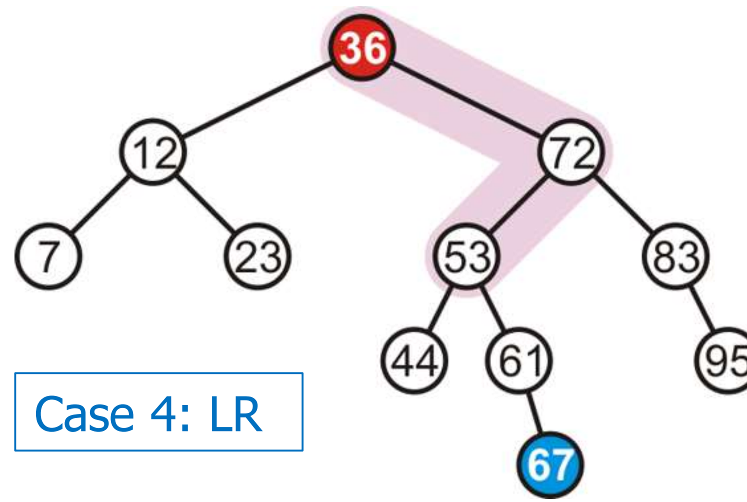
- Let the node that needs rebalancing be “a”
- Violation during insertion may occur in four cases

- Inside cases (Double Rotation)
  3. Insertion into right subtree of left child of a (case RL)

  4. Insertion into left subtree of right child of a (case LR)



Case 3: RL



Case 4: LR

Summary: How And When To Rotate?

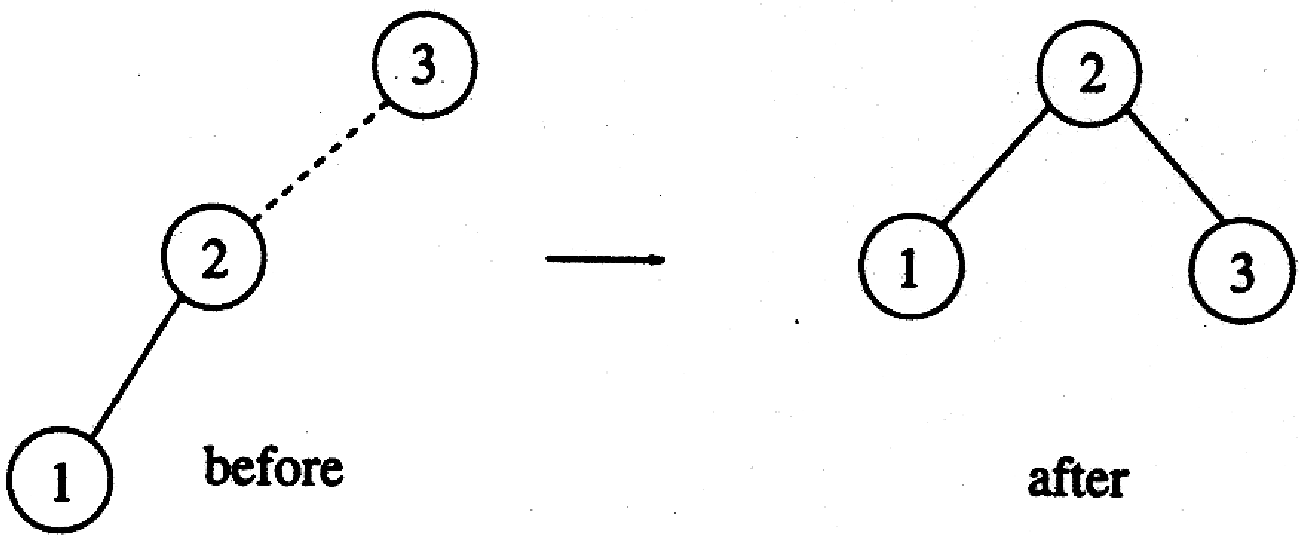
```
if tree is right heavy {
  if tree's right subtree is left heavy {
     Perform Left-Right rotation
  }
  else {
     Perform Single Left rotation
  }
}
…
```

```
…
else if tree is left heavy {
  if tree's left subtree is right heavy {
     Perform Right-Left rotation
  }
  else {
     Perform Single Right rotation
  }
}
```

AVL Tree – Complete Example

- Construct AVL Tree with the following input elements
  - 3, 2, 1, 4, 5, 6, 7

Insert 3, 2, 1

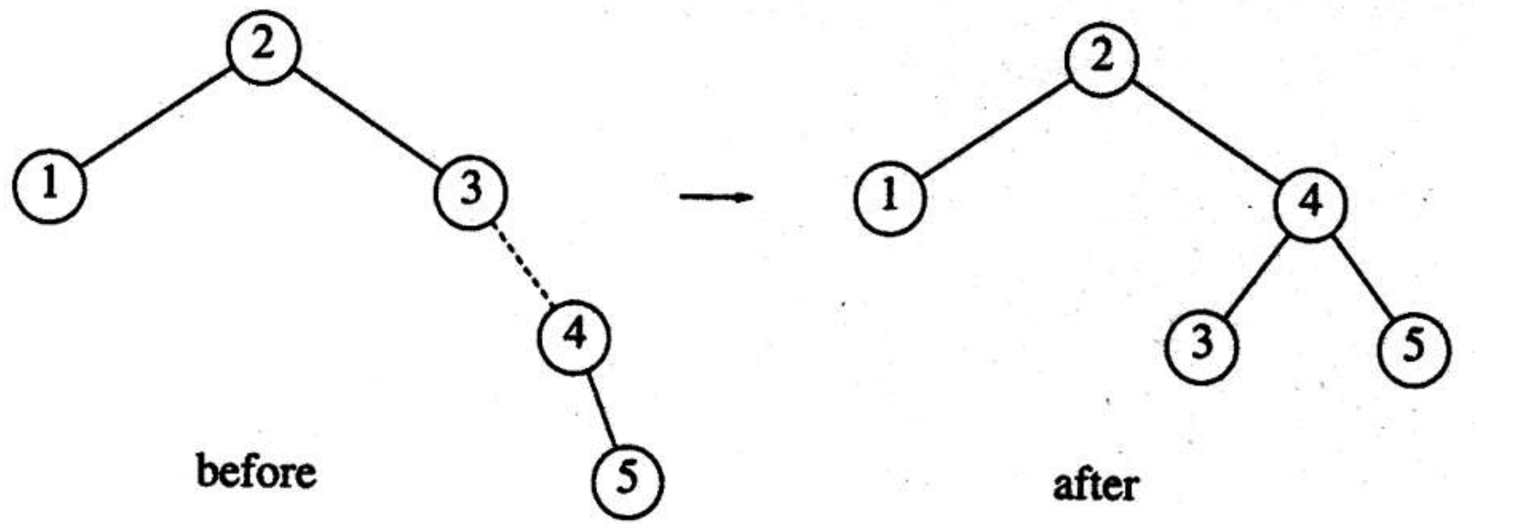


Insert 4, 5

AVL Tree – Complete Example

- Construct AVL Tree with the following input elements
  - 3, 2, 1, 4, 5, 6, 7

Insert 4, 5

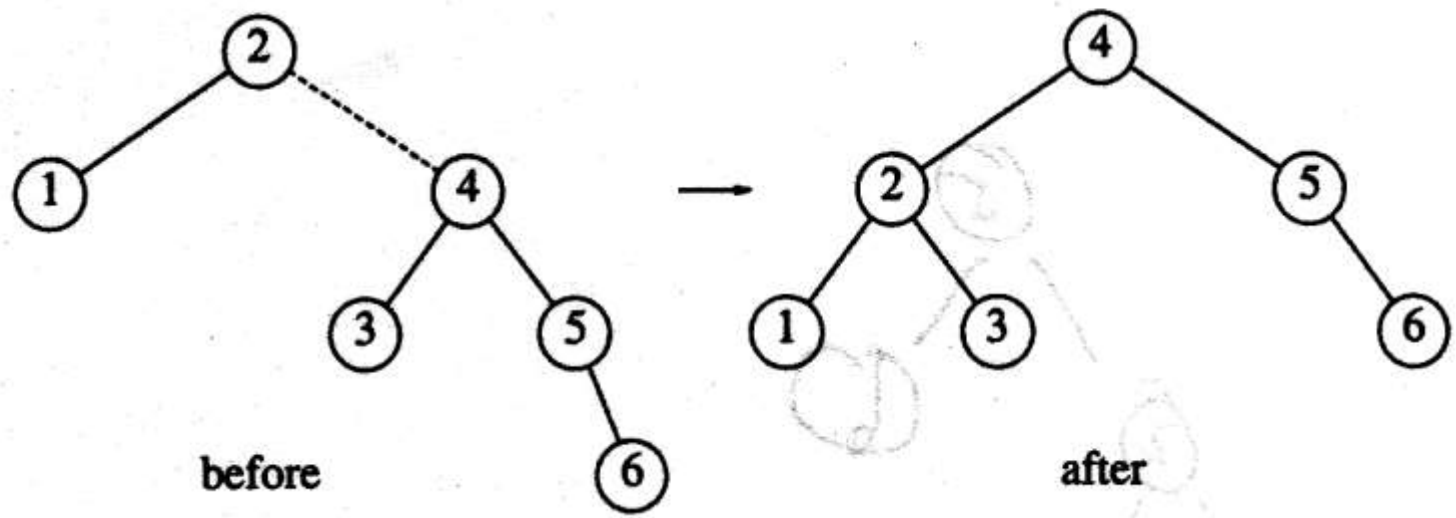


Insert 6

AVL Tree – Complete Example

- Construct AVL Tree with the following input elements
  - 3, 2, 1, 4, 5, 6, 7

Insert 6

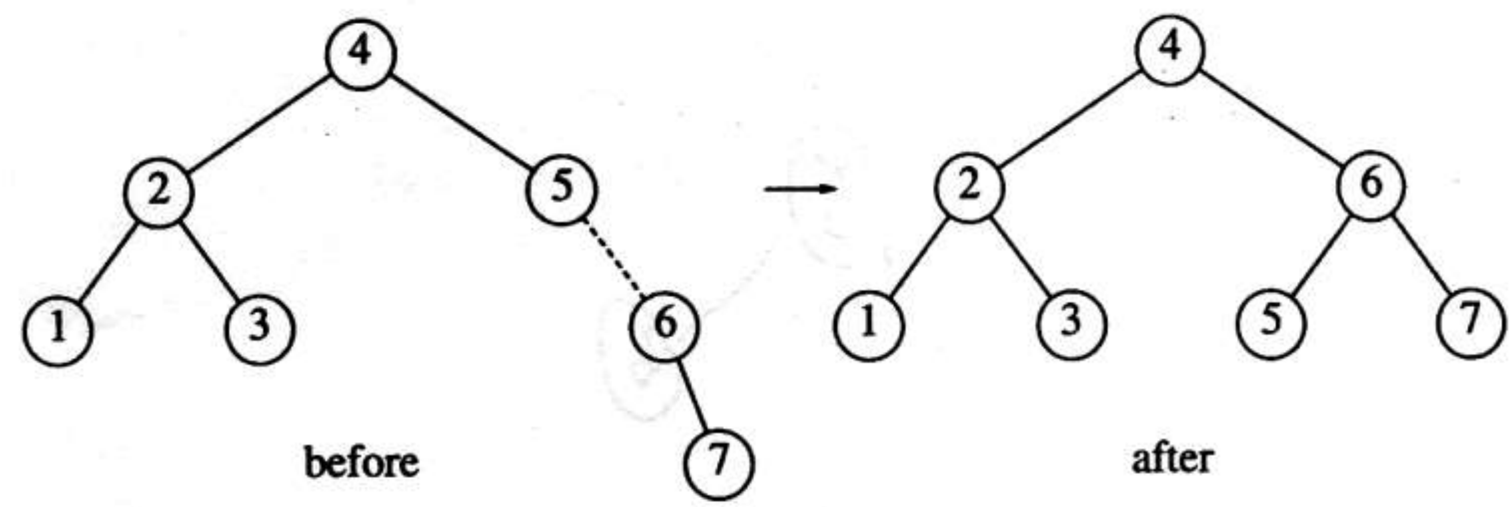


Insert 7

AVL Tree – Complete Example

- Construct AVL Tree with the following input elements
  - 3, 2, 1, 4, 5, 6, 7

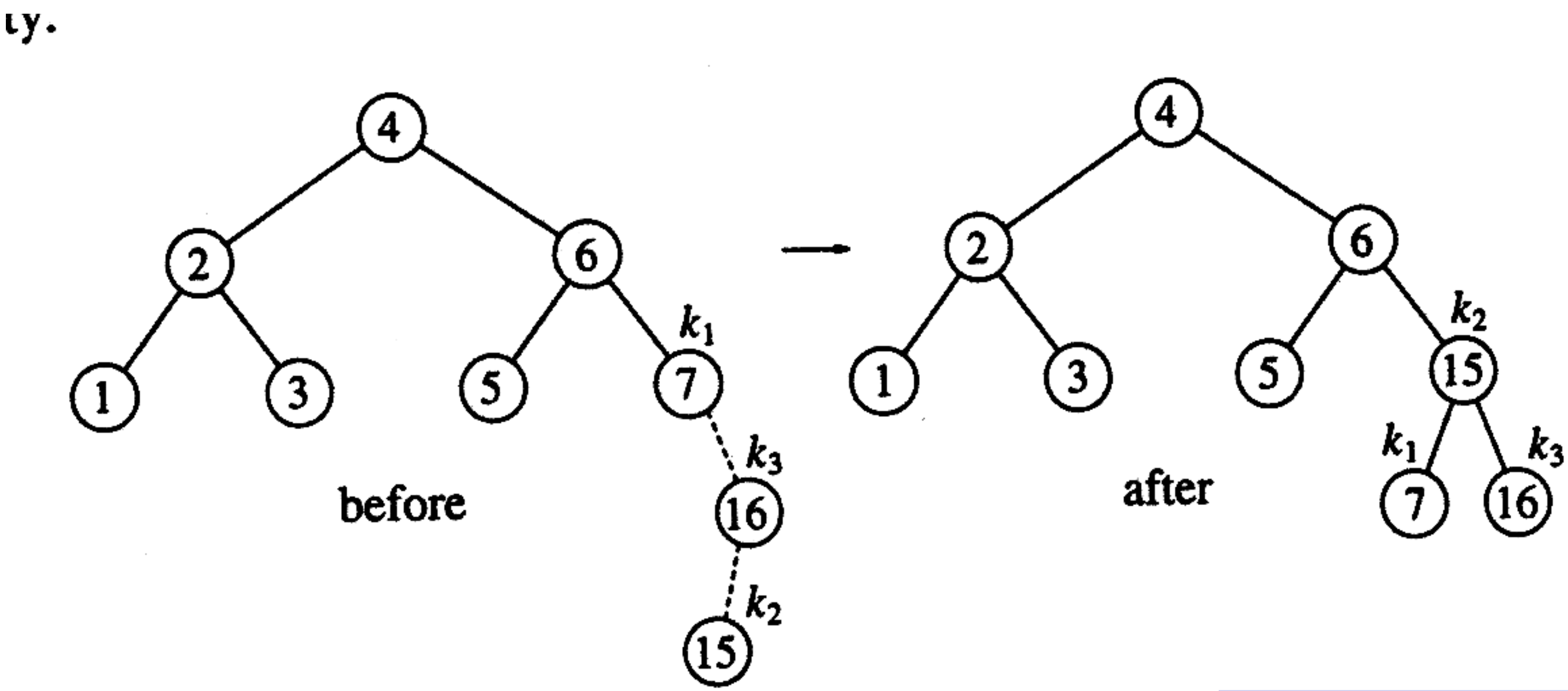
Insert 7



AVL Tree – Complete Example

- Suppose the following elements have to be inserted further
  - 16, 15, 14, 13, 12, 11, 10, 8

Insert 16, 15

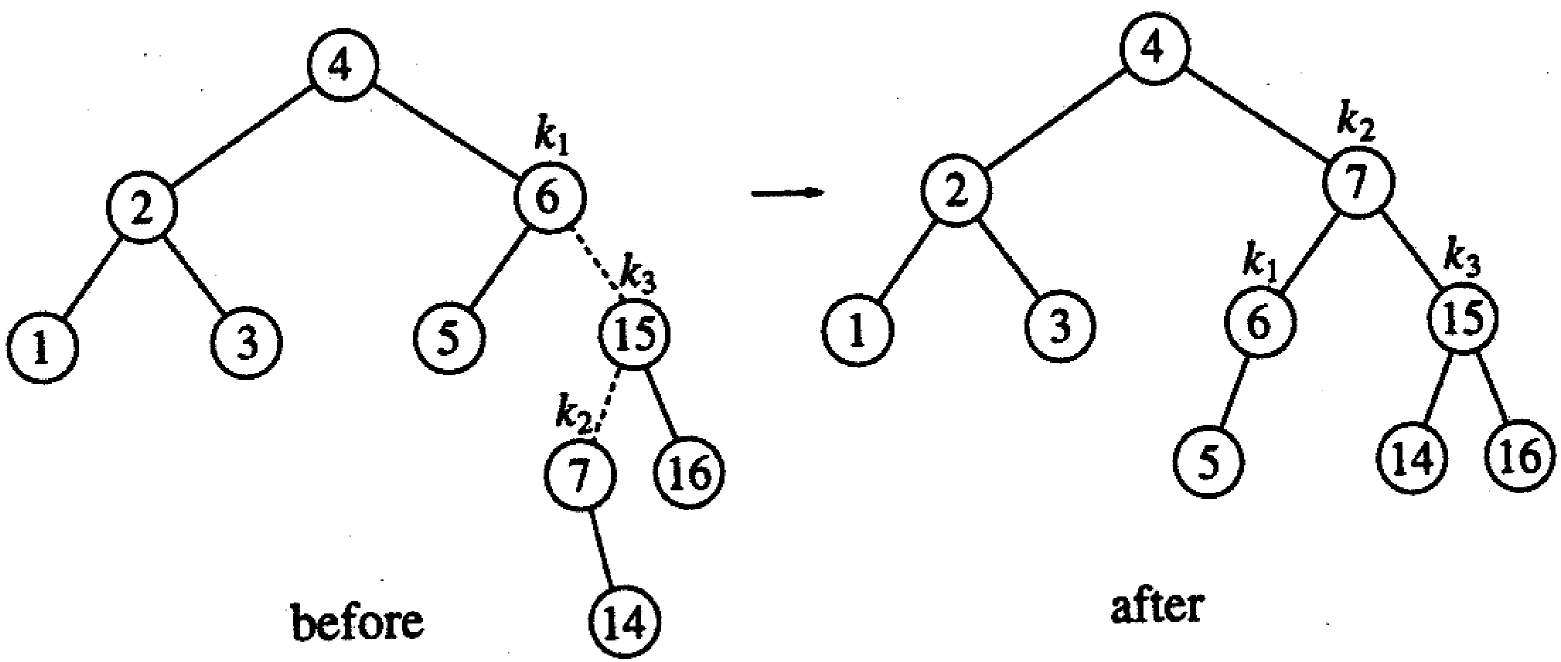


Insert 14

AVL Tree – Complete Example

- Suppose the following elements have to be inserted further
  - 16, 15, 14, 13, 12, 11, 10, 8

Insert 14

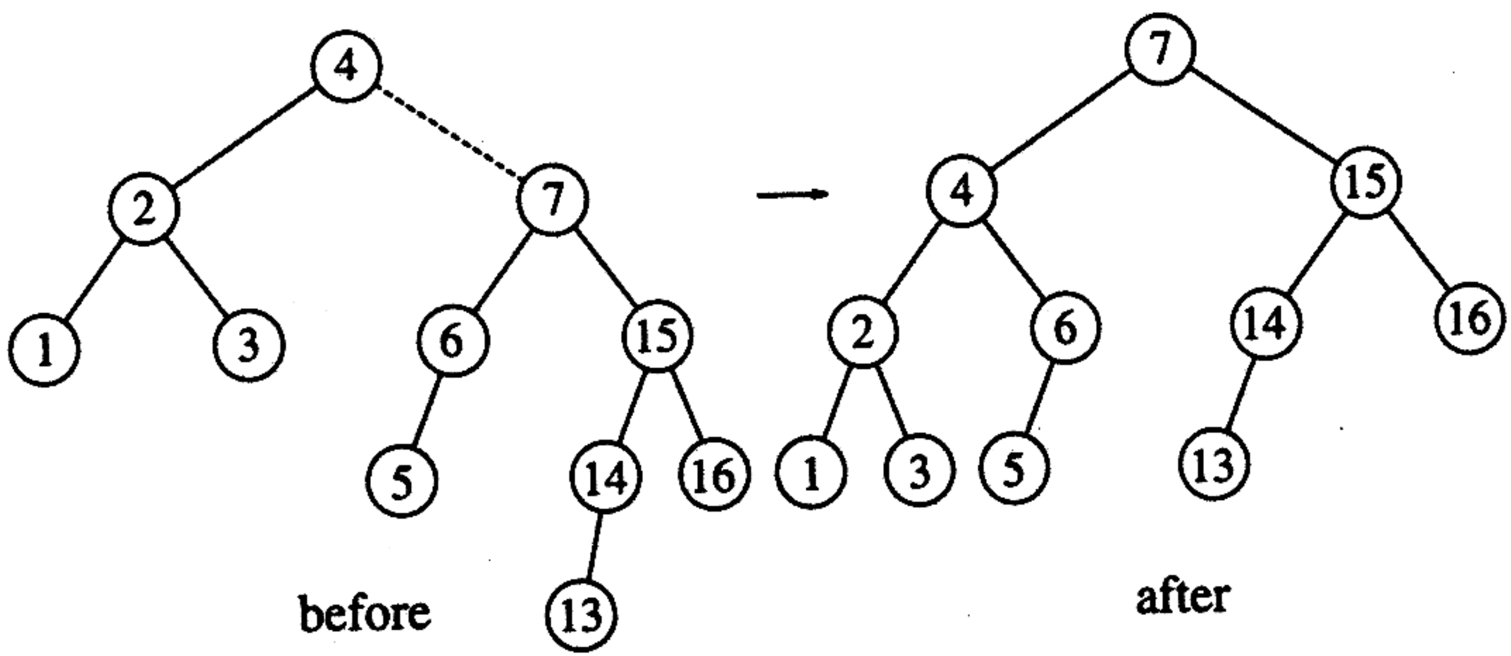


Insert 13

AVL Tree – Complete Example

- Suppose the following elements have to be inserted further
  - 16, 15, 14, 13, 12, 11, 10, 8

Insert 13

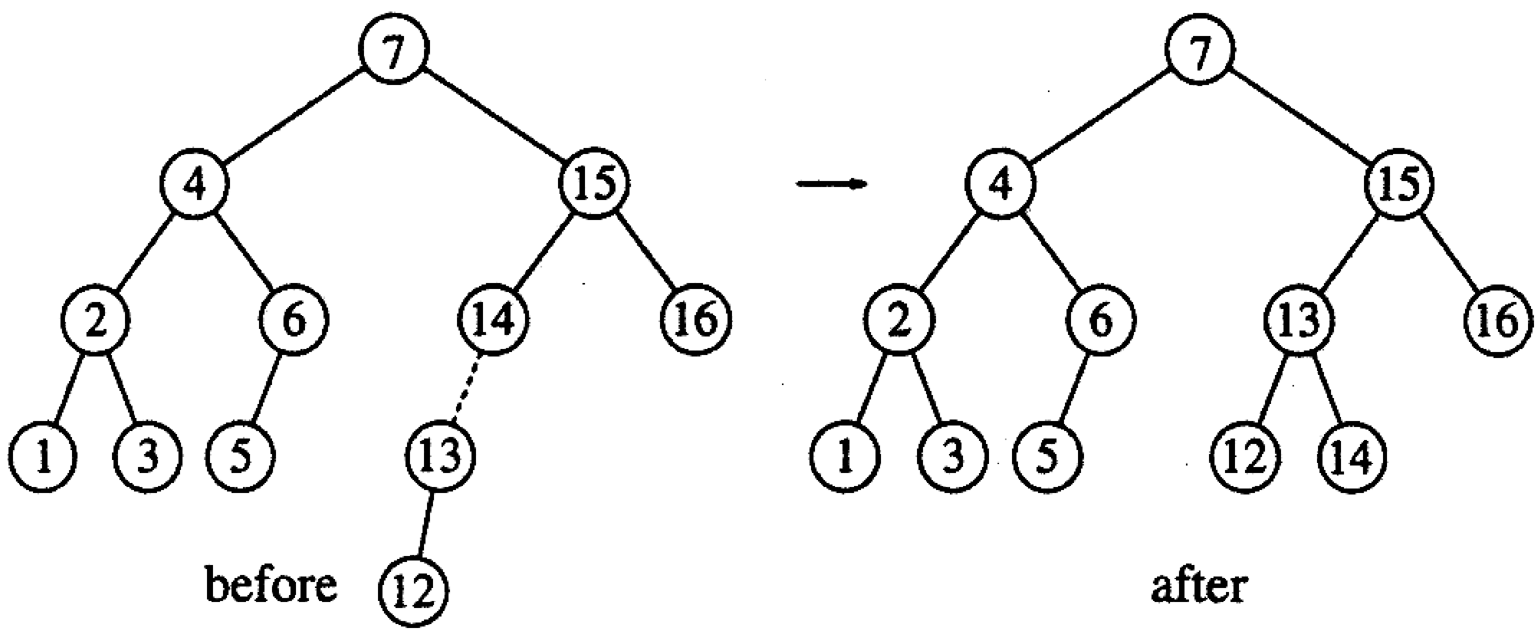


Insert 12

AVL Tree – Complete Example

- Suppose the following elements have to be inserted further
  - 16, 15, 14, 13, 12, 11, 10, 8

Insert 12

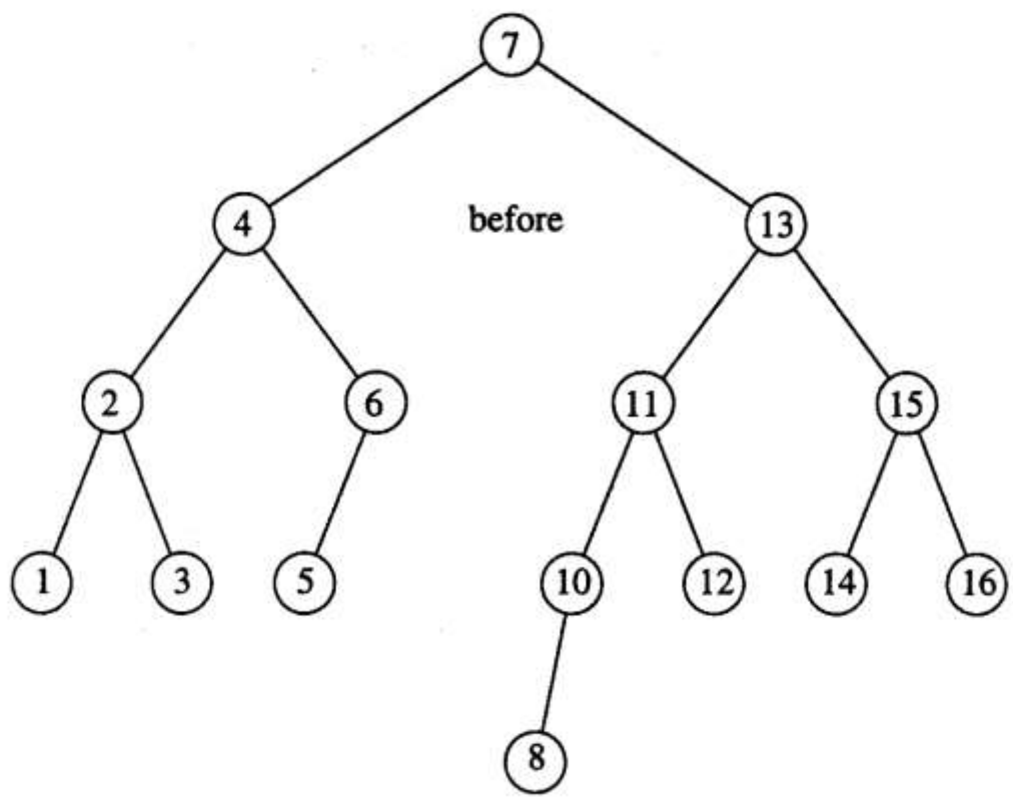


Insert 11, 10

AVL Tree – Complete Example

- Suppose the following elements have to be inserted further
  - 16, 15, 14, 13, 12, 11, 10, 8

Insert 11, 10 then 8



AVL Tree Implementation

AVL Trees: Implementation

```
struct AvlNode {
    ElementType Element;
    AvlTree     Left;
    AvlTree     Right;
    int         Height;
};
typedef struct AvlNode *Position;
typedef struct AvlNode *AvlTree;
AvlTree MakeEmpty( AvlTree T );
Position Find( ElementType X, AvlTree T );
Position FindMin( AvlTree T );
Position FindMax( AvlTree T );
AvlTree Insert( ElementType X, AvlTree T );
AvlTree Delete( ElementType X, AvlTree T );
ElementType Retrieve( Position P );
```

AVL Trees: Implementation

```
int Height(Position P )
{
    if( P == NULL )
        return -1;
    else
        return P->Height;
}
```

AVL Trees: Insert Function

```
AvlTree Insert( ElementType X, AvlTree T ) {
   if ( T == NULL ) { /* Create and return a one-node tree */
      T = new AvlNode;
      T->Element = X;
      T->Left = T->Right = NULL;
   }
   else if( X < T->Element ) {
      T->Left = Insert( X, T->Left );
      if( Height( T->Left ) - Height( T->Right ) == 2 )
         if( X < T->Left->Element )
            T = SingleRotateWithLeft( T ); // RR rotation
         else
            T = DoubleRotateWithLeft( T );  // RL rotation
   }
   else if( X > T->Element ) {
      T->Right = Insert( X, T->Right );
      if( Height( T->Right ) - Height( T->Left ) == 2 )
         if( X > T->Right->Element )
            T = SingleRotateWithRight( T );  // LL rotation
         else
            T = DoubleRotateWithRight( T );  // LR rotation
   }  /* Else X is in the tree already; we'll do nothing */
   T->Height = Max( Height( T->Left ), Height( T->Right ) ) + 1;
   return T;
}
```

AVL Trees: Insert Function

```
AvlTree Insert( ElementType X, AvlTree T ) {
   if ( T == NULL ) { /* Create and return a one-node tree */
      T = new AvlNode;
      T->Element = X;
      T->Left = T->Right = NULL;
   }
   else if( X < T->Element ) {
      T->Left = Insert( X, T->Left );
      if( Height( T->Left ) - Height( T->Right ) == 2 )
         if( X < T->Left->Element )
            T = SingleRotateWithLeft( T ); // RR rotation
```


```
if ( T == NULL ) { /* Create and return a one-node tree */
   T = new AvlNode;
   T->Element = X;
   T->Left = T->Right = NULL;
}
```


```
            T = SingleRotateWithRight( T );  // LL rotation
         else
            T = DoubleRotateWithRight( T );  // LR rotation
   }  /* Else X is in the tree already; we'll do nothing */
   T->Height = Max( Height( T->Left ), Height( T->Right ) ) + 1;
   return T;
}
```

AVL Trees: Insert Function

```
AvlTree Insert( ElementType X, AvlTree T ) {
   if ( T == NULL ) { /* Create and return a one-node tree */
      T = new AvlNode;
      T->Element = X;
      T->Left = T->Right = NULL;
   }
   else if( X < T->Element ) {
      T->Left = Insert( X, T->Left );
      if( Height( T->Left ) - Height( T->Right ) == 2 )
         if( X < T->Left->Element )
            T = SingleRotateWithLeft( T ); // RR rotation
         else
            T = DoubleRotateWithLeft( T );  // RL rotation
   }
}
```


```
else if( X < T->Element ) {
   T->Left = Insert( X, T->Left );
   if( Height( T->Left ) - Height( T->Right ) == 2 )
      if( X < T->Left->Element )
         T = SingleRotateWithLeft( T );  // RR rotation
      else
         T = DoubleRotateWithLeft( T );  // RL rotation
}
```

AVL Trees: Insert Function

```
AvlTree Insert( ElementType X, AvlTree T ) {
```

```
else if( X > T->Element ) {
   T->Right = Insert( X, T->Right );
   if( Height( T->Right ) - Height( T->Left ) == 2 )
      if( X > T->Right->Element )
         T = SingleRotateWithRight( T );  // LL rotation
      else
         T = DoubleRotateWithRight( T );  // LR rotation
}
/* Else X is in the tree already; we'll do nothing */
```

```
  }
```


```
  else if( X > T->Element ) {
     T->Right = Insert( X, T->Right );
     if( Height( T->Right ) - Height( T->Left ) == 2 )
        if( X > T->Right->Element )
           T = SingleRotateWithRight( T ); // LL rotation
        else
           T = DoubleRotateWithRight( T );  // LR rotation
  }  /* Else X is in the tree already; we'll do nothing */
```


```
  T->Height = Max( Height( T->Left ), Height( T->Right ) ) + 1;
  return T;
}
```

AVL Trees: Insert Function

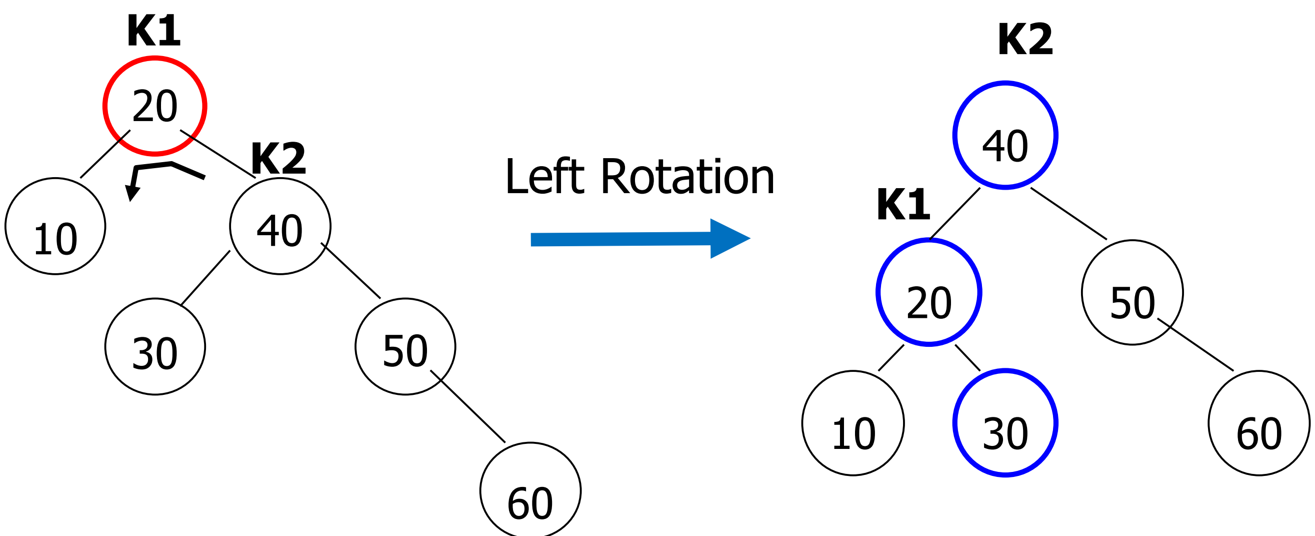
```
AvlTree Insert( ElementType X, AvlTree T ) {
   if ( T == NULL ) { /* Create and return a one-node tree */
      T = new AvlNode;
      T->Element = X;
      T->Left = T->Right = NULL;
   }
   else if( X < T->Element ) {
      T->Left = Insert( X, T->Left );
      if( Height( T->Left ) - Height( T->Right ) == 2 )
         if( X < T->Left->Element )
            T = SingleRotateWithLeft( T ); // RR rotation
         else
            T = DoubleRotateWithLeft( T ); // RL rotation
   }
   else if( X > T->Element ) {
      T->Right = Insert( X, T->Right );
      if( Height( T->Right ) - Height( T->Left ) == 2 )
```

```
T->Height = Max( Height( T->Left ), Height( T->Right ) ) + 1;
return T;
```

```
            T = DoubleRotateWithRight( T ); // LR rotation
   } /* Else X is in the tree already; we'll do nothing */
   T->Height = Max( Height( T->Left ), Height( T->Right ) ) + 1;
   return T;
}
```

AVL Trees: LL Rotation

```
Position SingleRotateWithRight( Position K1 ) {
    Position K2;
    K2 = K1->Right;  // K1: node whose balance factor is violated
    K1->Right = K2->Left;
    K2->Left = K1;
    K1->Height = Max( Height(K1->Left), Height(K1->Right) ) + 1;
    K2->Height = Max( Height(K2->Right), K1->Height ) + 1;
    return K2;  /* New root */
}
```



AVL Trees: RR Rotation

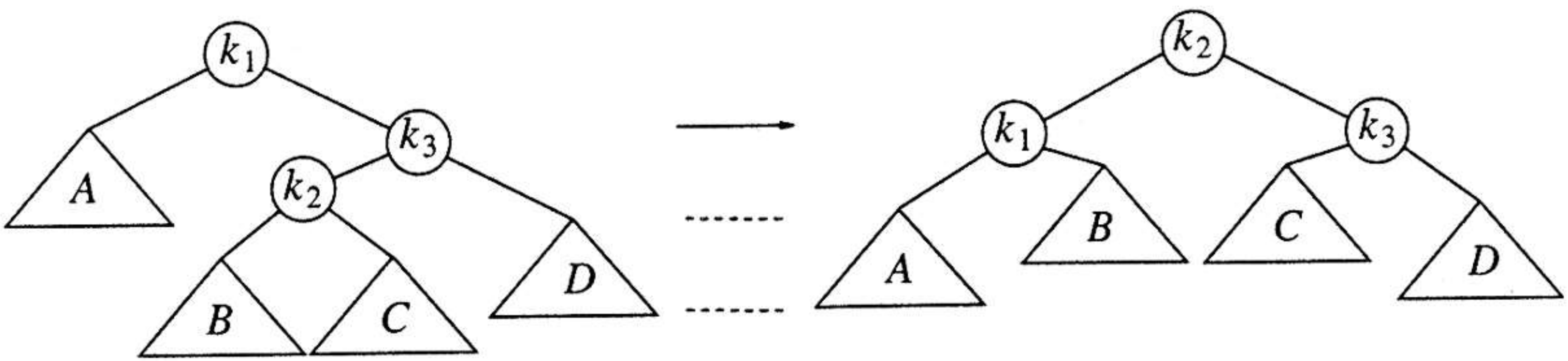
```
Position SingleRotateWithLeft( Position K1 ) {
   Position K2;
   K2 = K1->Left; // K1: node whose balance factor is violated
   K1->Left = K2->Right;
   K2->Right = K1;
   K1->Height = Max( Height(K1->Left), Height(K1->Right) ) + 1;
   K2->Height = Max( Height(K2->Left), K1->Height ) + 1;
   return K2;  /* New root */
}
```

AVL Trees: LR Rotation

```
Position DoubleRotateWithRight( Position K1)
{
    /* RR rotation between K3 and K2 */
    K1->Right = SingleRotateWithLeft( K1->Right );
    /* LL rotation between K1 and K2 */
    return SingleRotateWithRight( K1 );
}
```

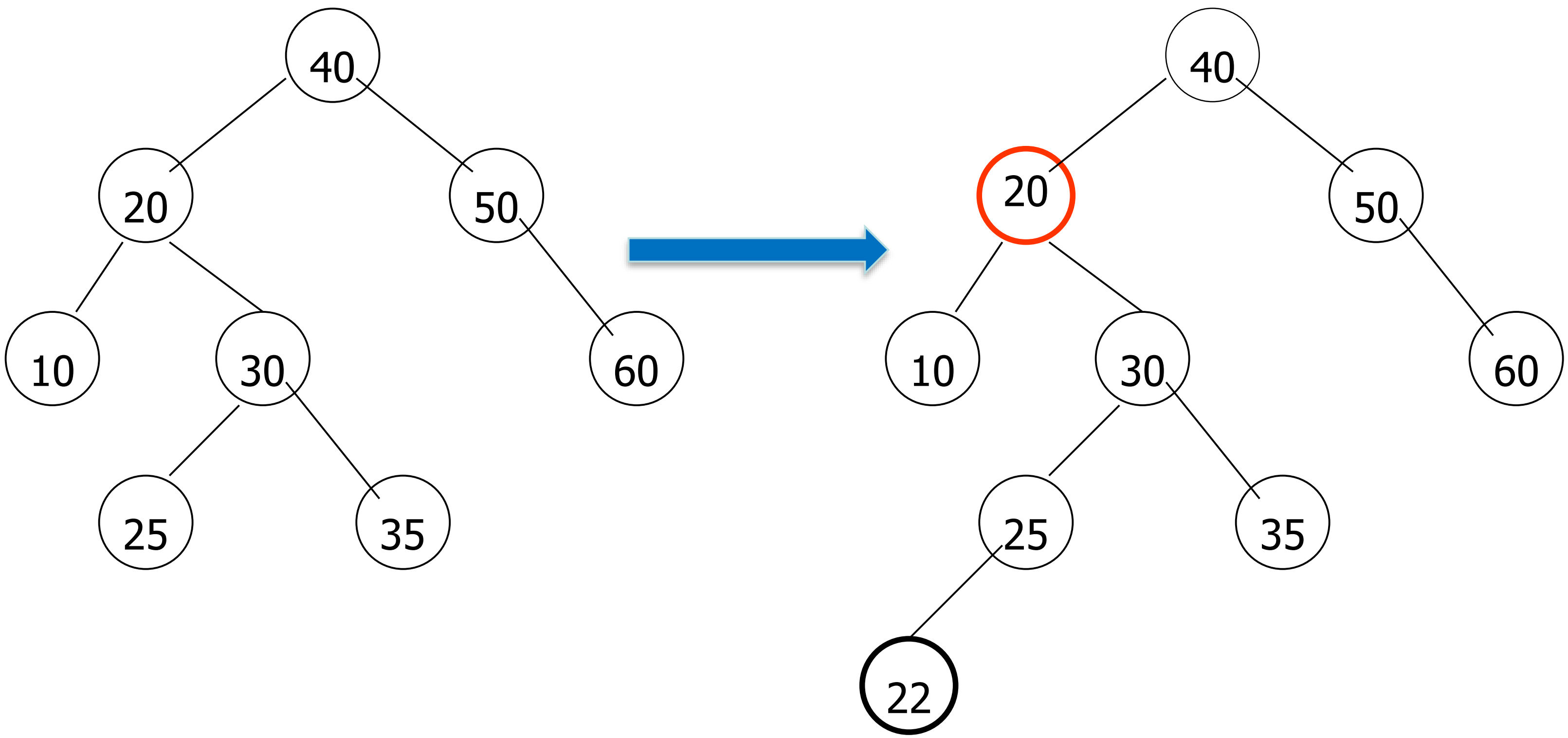
Single Right Rotation at K3

Single Left rotation at K1

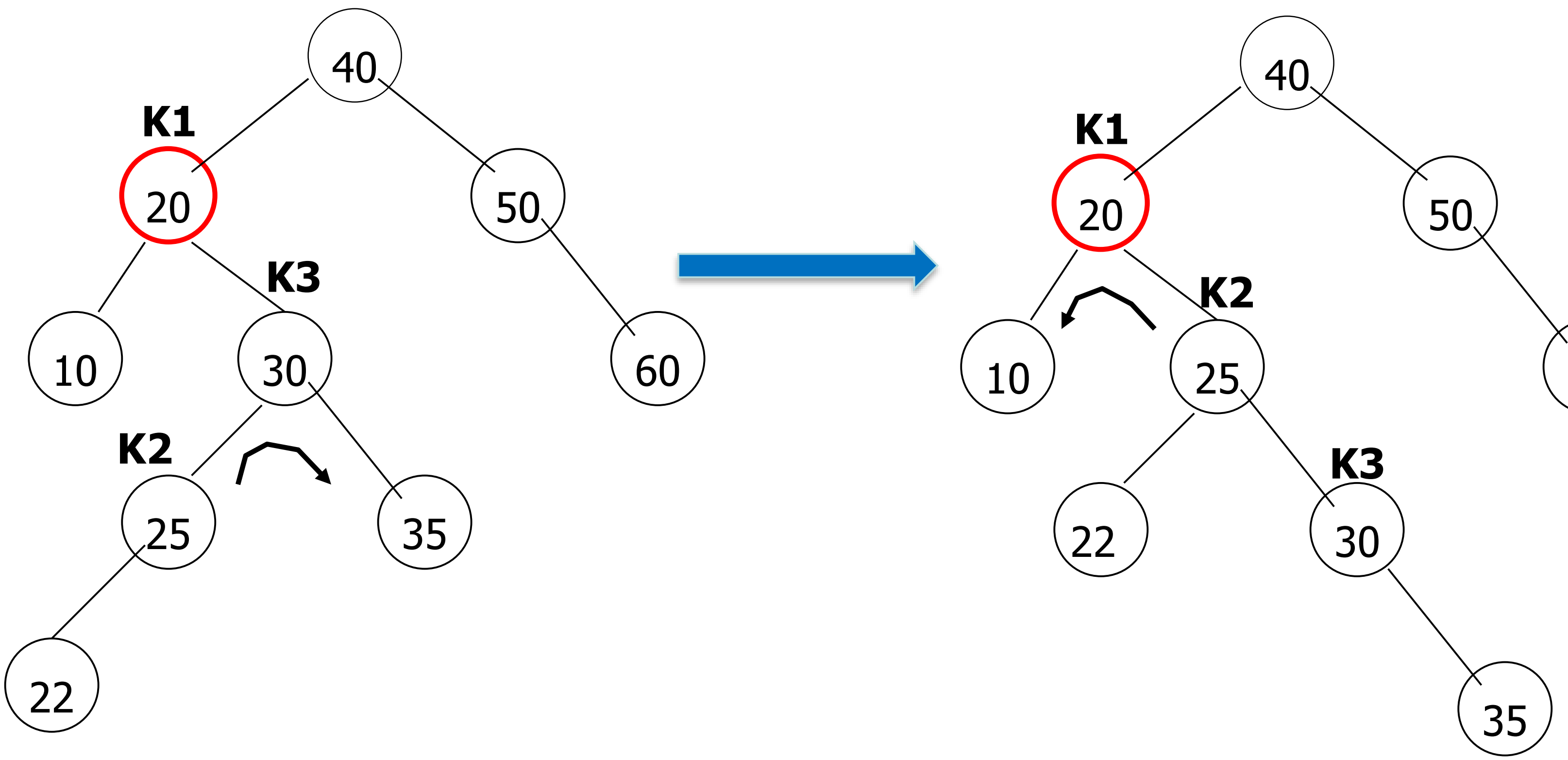


LR Rotation

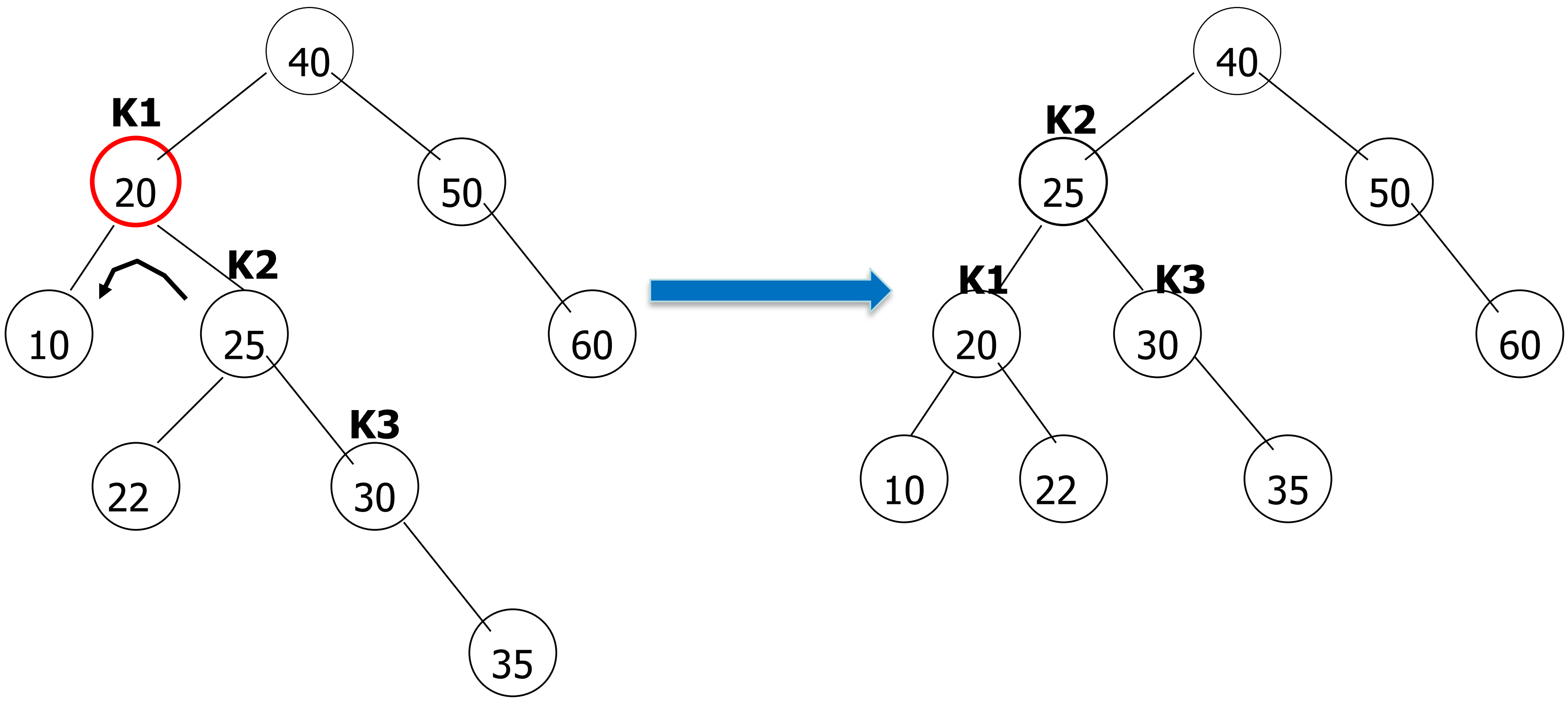
- Adding node 22
  - Requires double rotation!



LR Rotation



LR Rotation

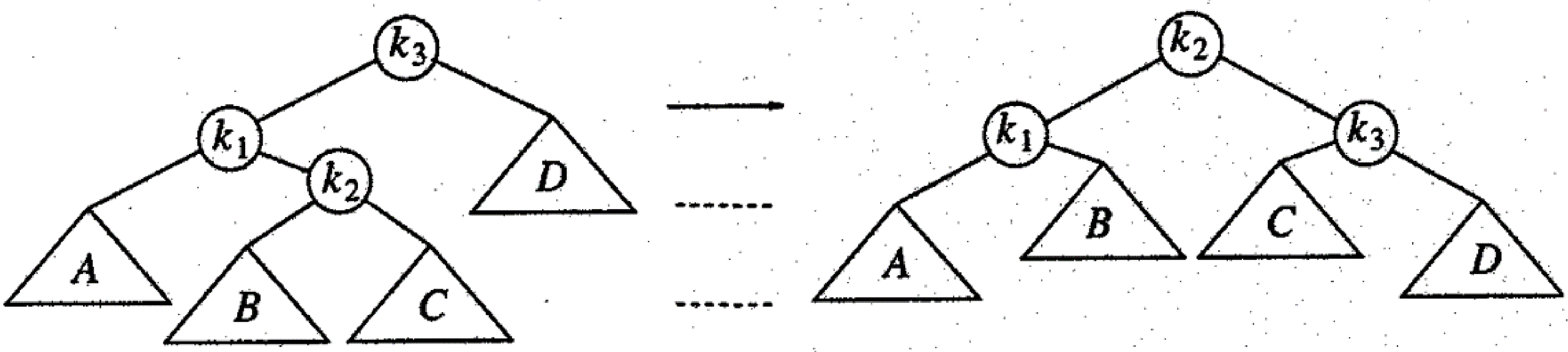


AVL Trees: RL Rotation

```
Position DoubleRotateWithLeft( Position K3 )
{
     /* LL rotation between K1 and K2 */
     K3->Left = SingleRotateWithRight( K3->Left );
     /* RR rotation between K3 and K2 */
     return SingleRotateWithLeft( K3 );
}
```

Single Left Rotation at K1

Single Right rotation at K3



AVL Tree Deletion

AVL Tree: Deletion

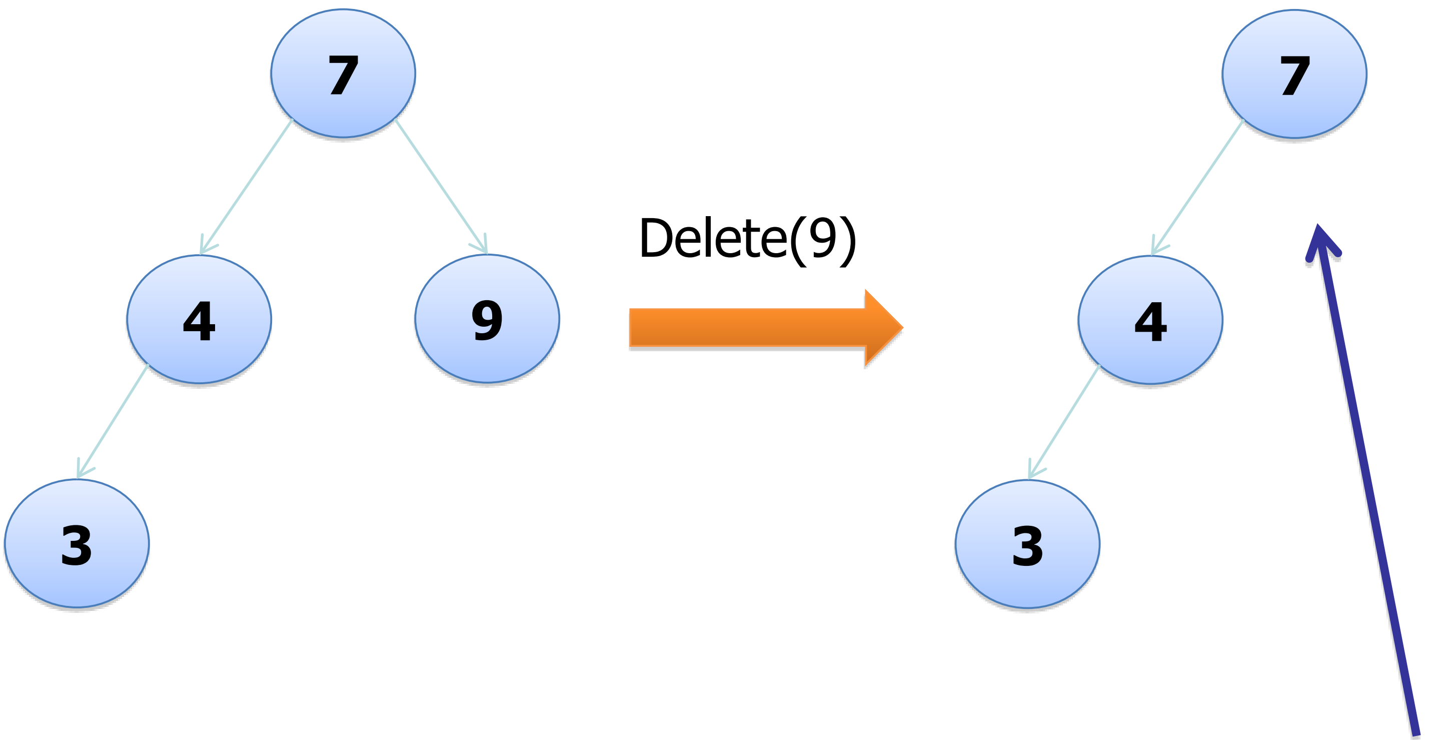
- Goal: To preserve the height balance property of BST after deletion

- Step 1: Perform BST delete
  - Maintains the BST property
  - May break the balance factors of ancestors!

- Step 2: Fix the AVL tree balance constraint
  - Perform transformation on the tree by means of rotation such that
    - ➢ BST property is maintained
    - ➢ Transformation fixes any balance factors that are < -1 or > 1

AVL Tree: Deletion

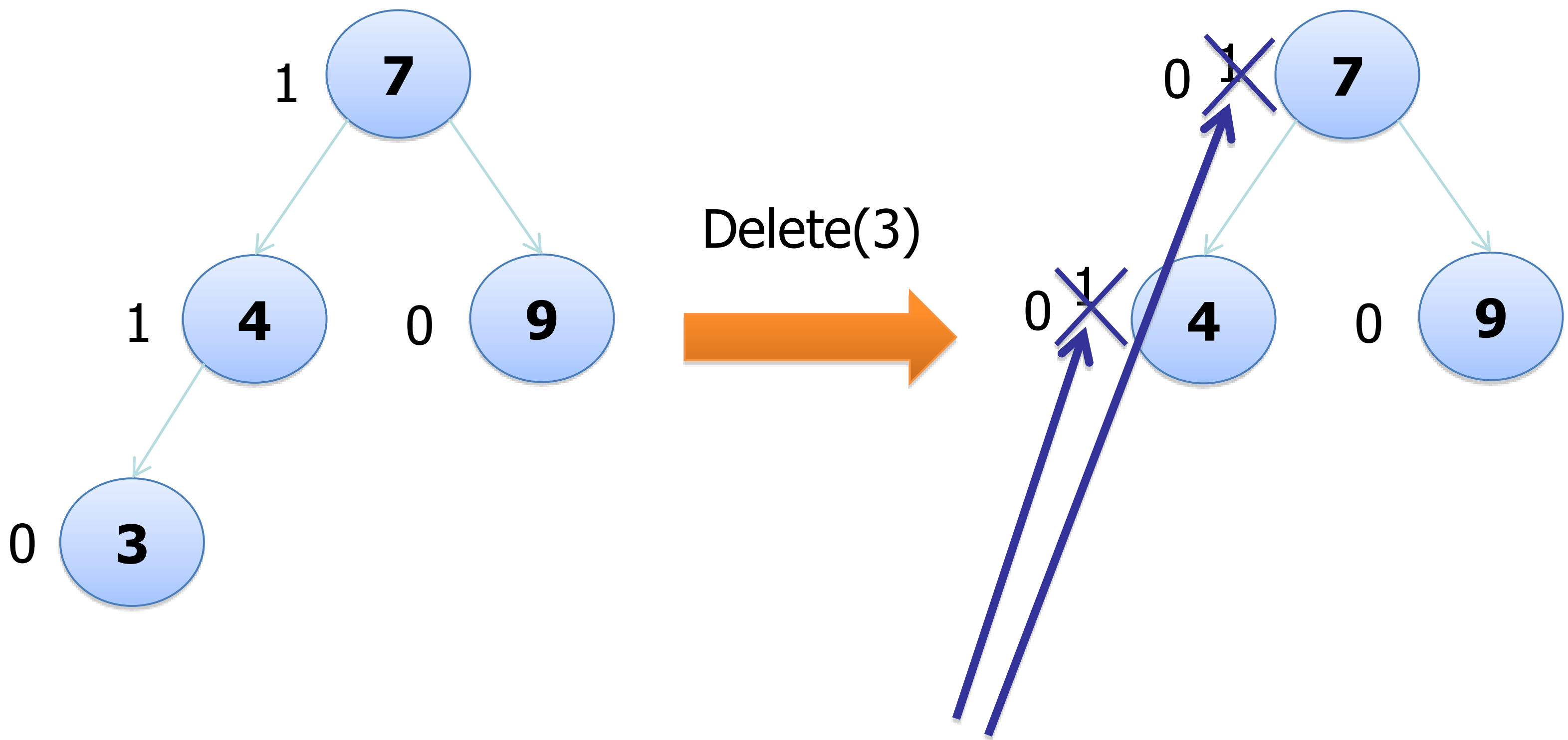
- BST deletion breaks the invariants of AVL tree



NOT an AVL tree!

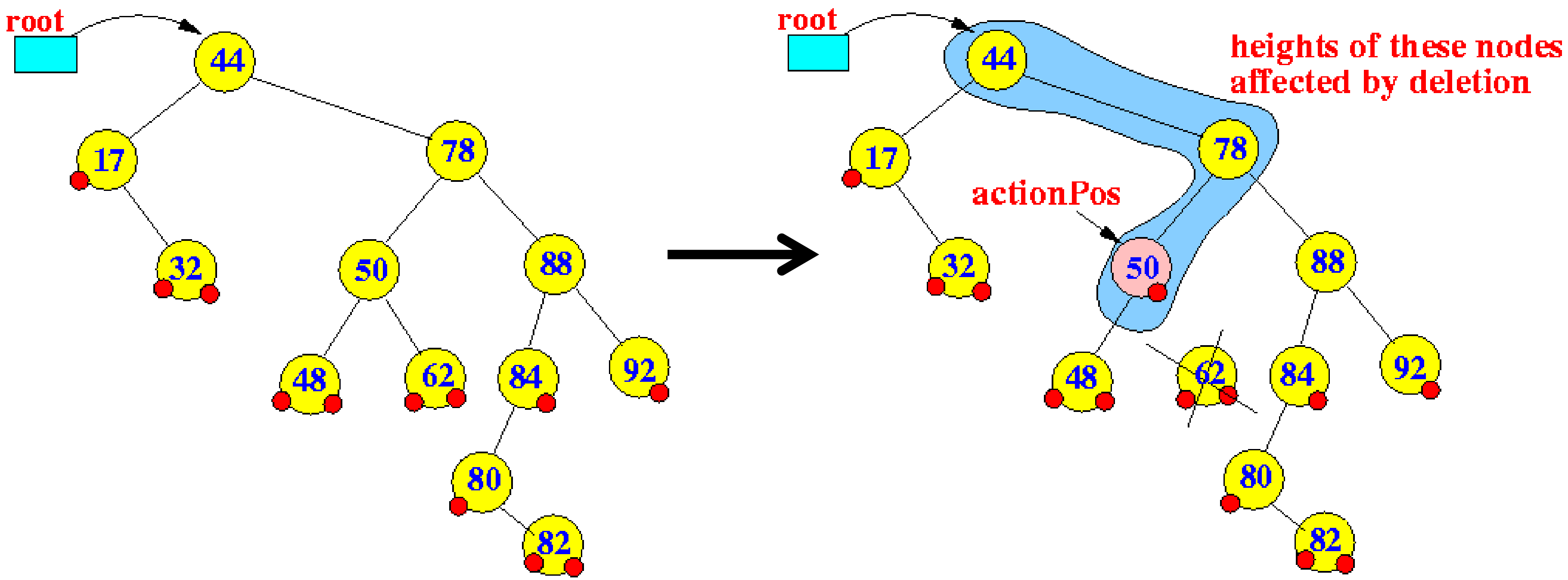
AVL Tree: Deletion

- BST deletion breaks the balance factors of ancestors



AVL Tree: BST Deletion

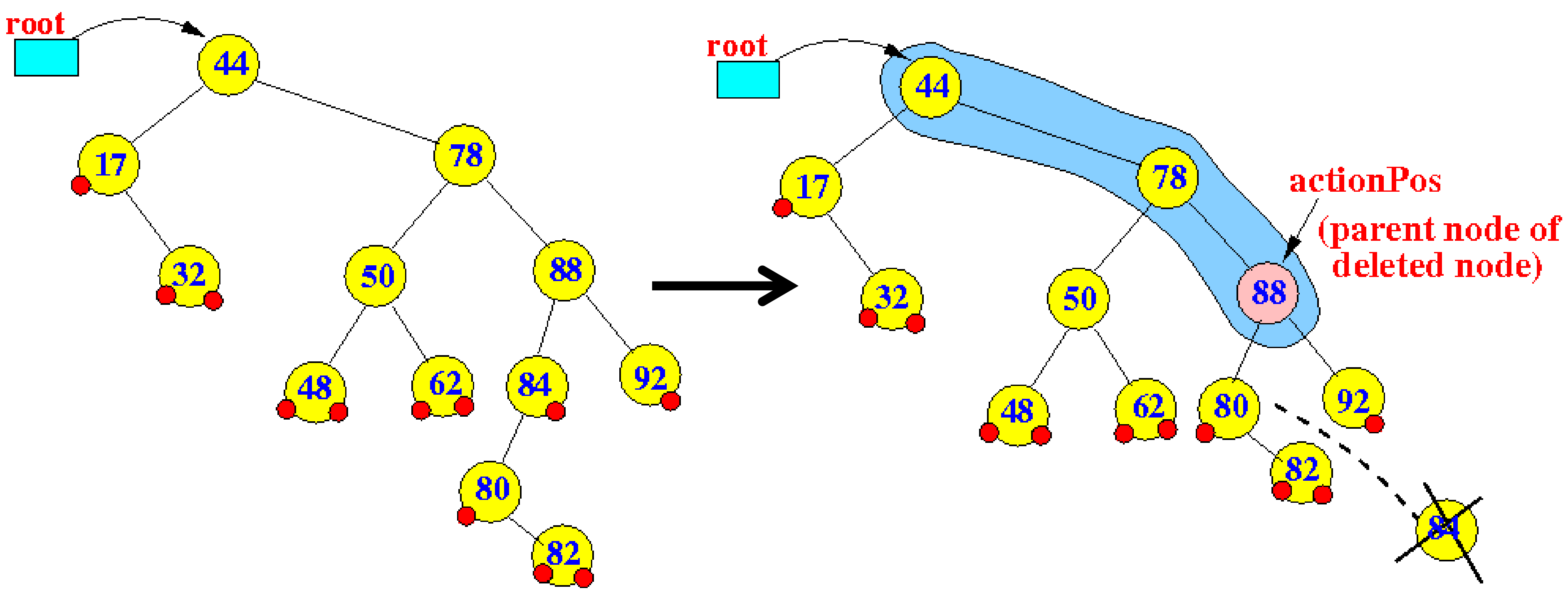
- Case 1: Node to be deleted has degree 0 (i.e., leaf node)
  - Consider deleting node containing 62



- Action position: Reference to parent node from which a node has been physically removed
  - First node whose height may be changed by deletion

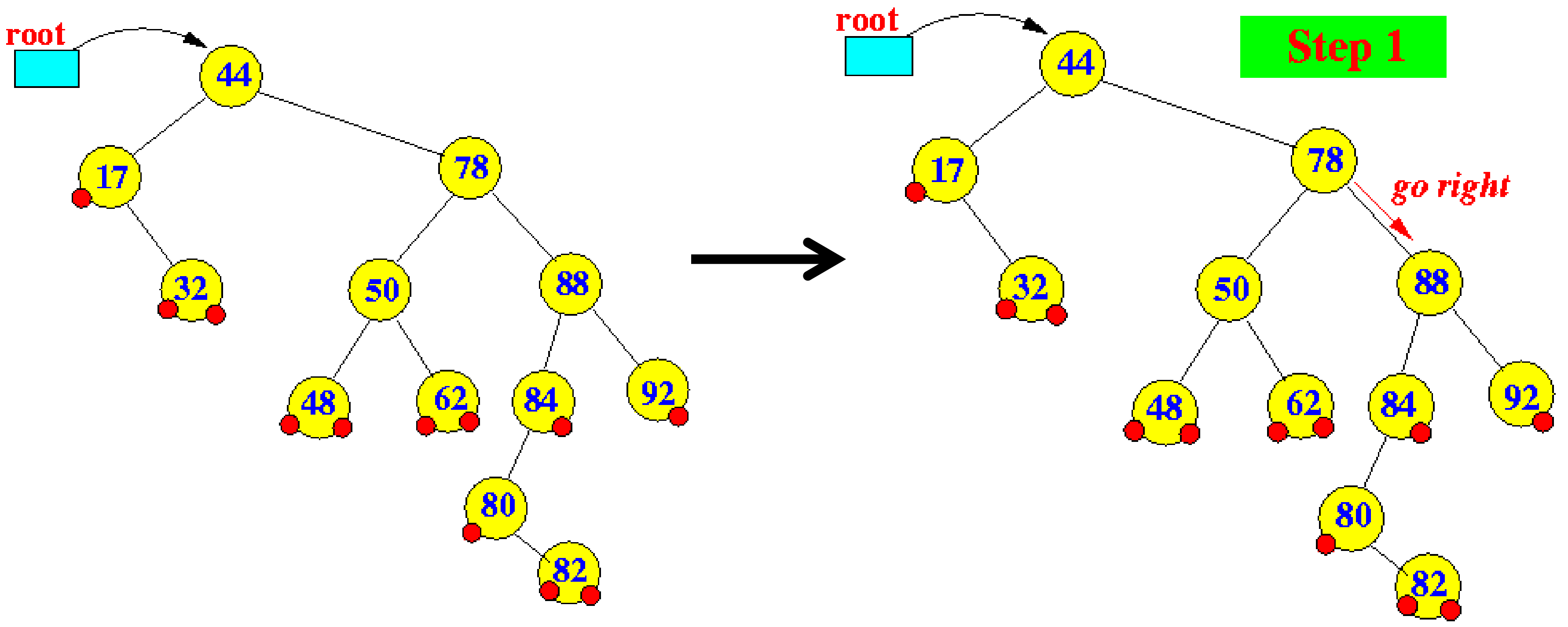
AVL Tree: BST Deletion

- Case 2: Node to be deleted has degree 1 (i.e., node with one child)
  - Consider deleting node containing 84



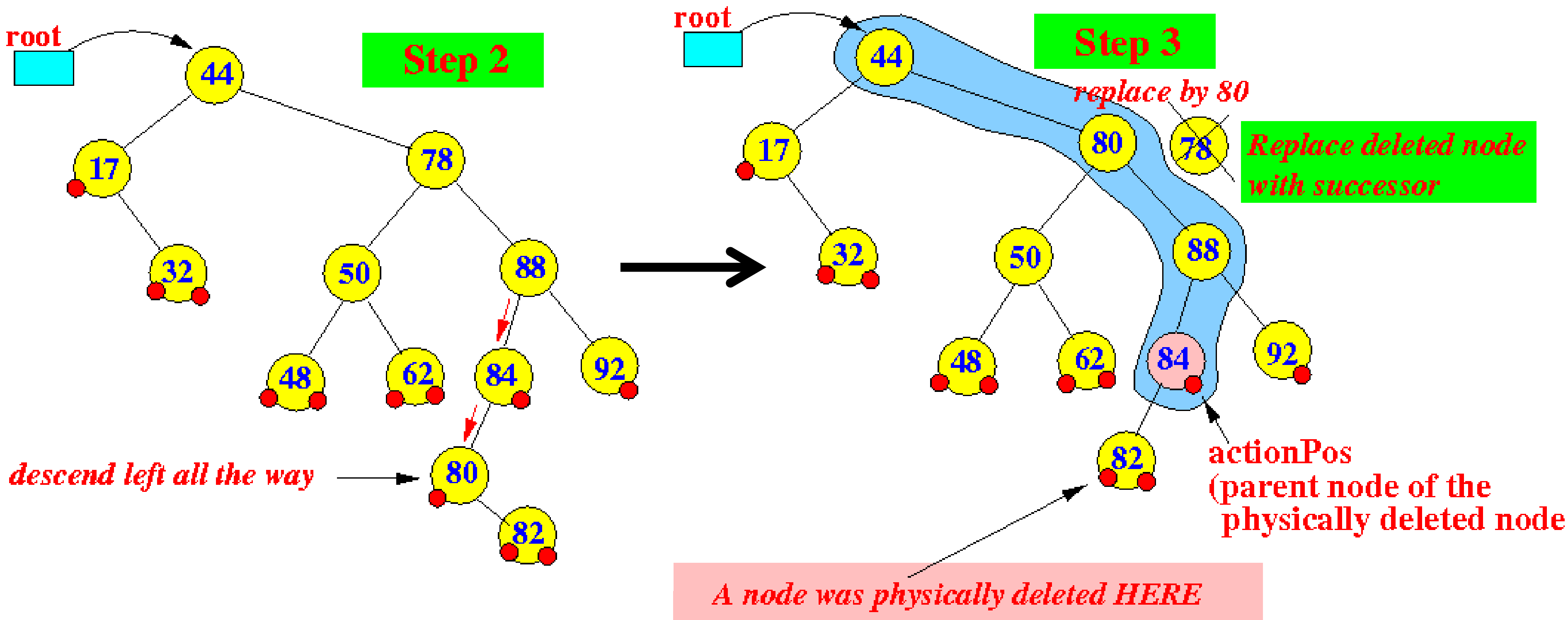
AVL Tree: BST Deletion

- Case 3: Node p to be deleted has two children (i.e., degree 2)
  - Replace node p with the minimum object in the right subtree
  - Delete that object from the right subtree
  - Consider deleting node containing 78



AVL Tree: BST Deletion

- Case 3: Node p to be deleted has two children (i.e., degree 2)
  - Replace node p with the minimum object in the right subtree
  - Delete that object from the right subtree
  - Consider deleting node containing 78



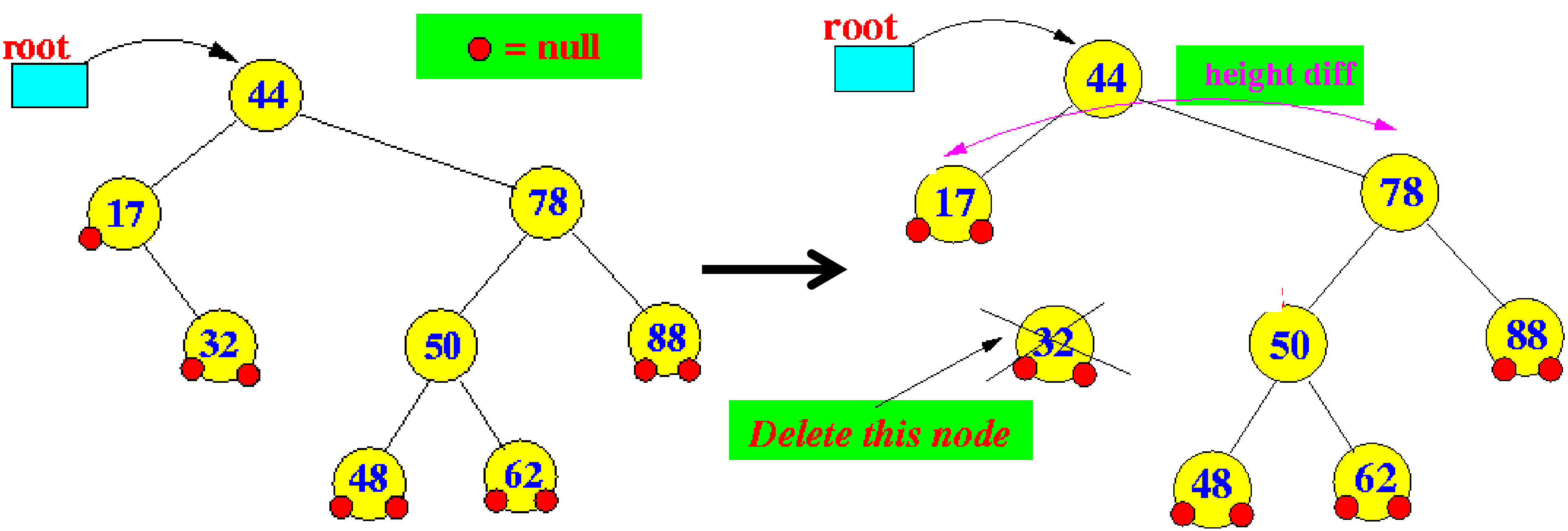
AVL Tree Deletion

- After removing a child, delete must check for imbalance
  - Similar to insert operation

- Rotations can be used to re-balance an out-of-balanced AVL tree
  - LL, RR, LR and RL rotations

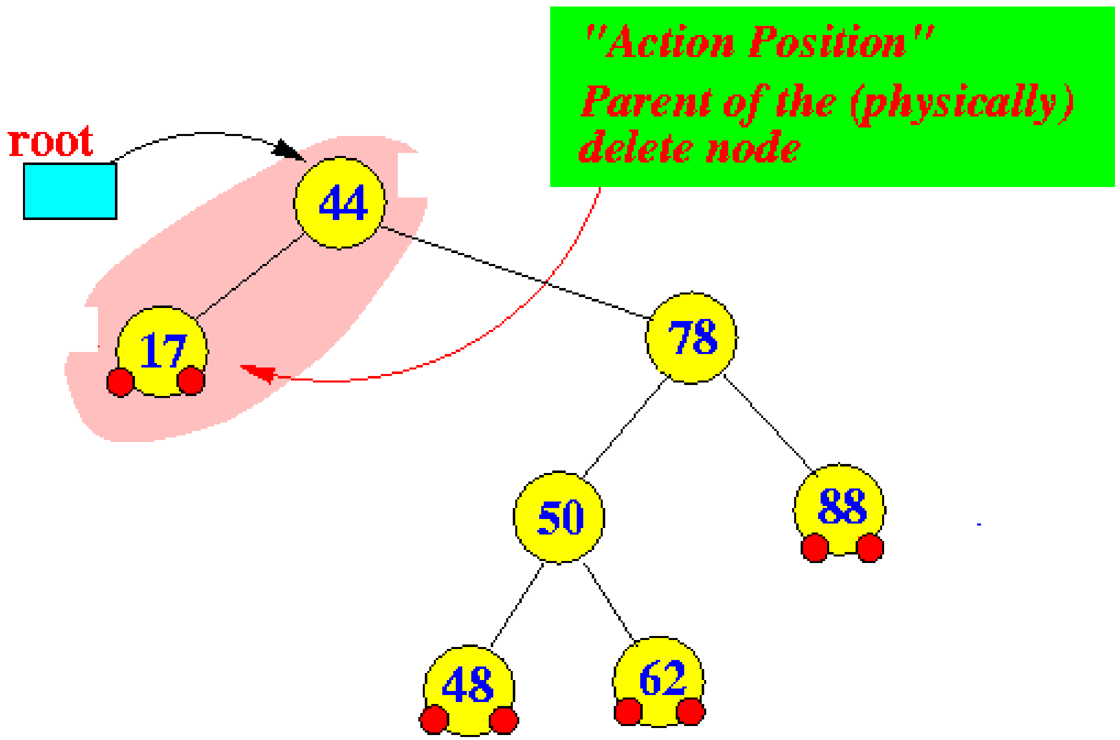
AVL Tree Deletion: Example

- Deleting a node from an ALV tree can cause imbalance
  - Consider deleting node 32



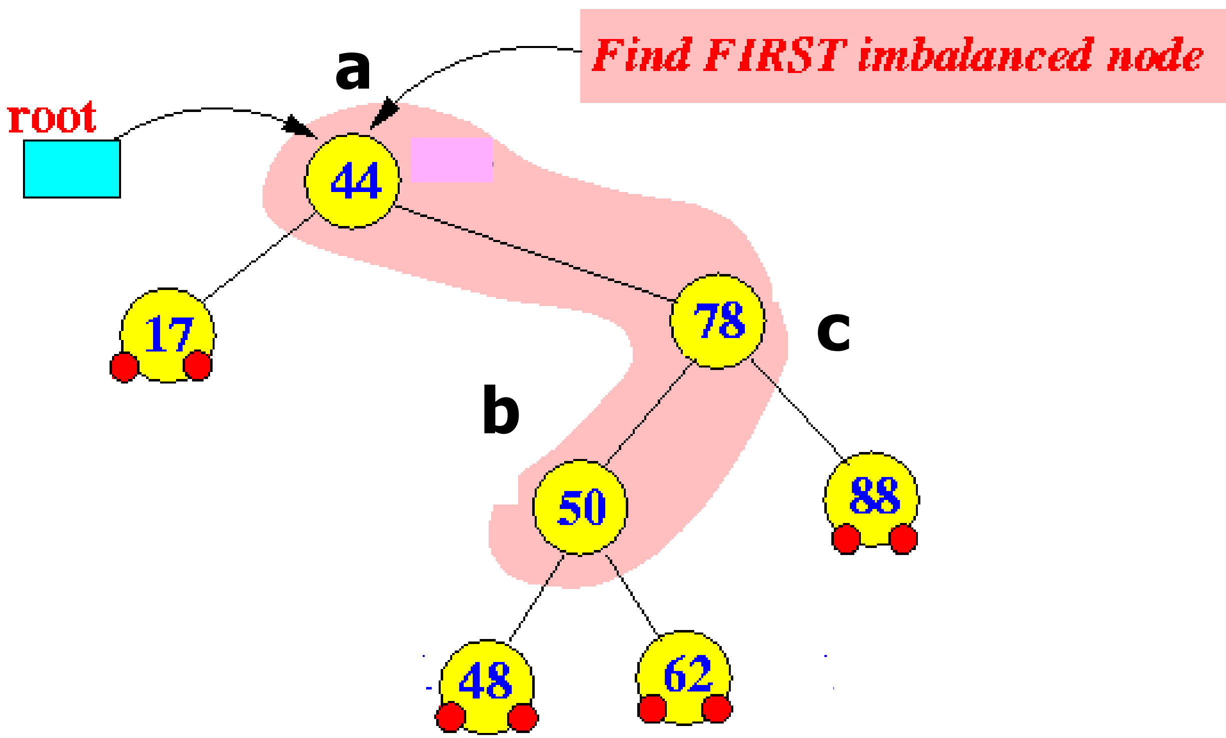
AVL Tree Deletion: Example

- The balance factor changes at only nodes between the root and the parent node of the physically deleted node
  - Starting at the action position find the first imbalanced node



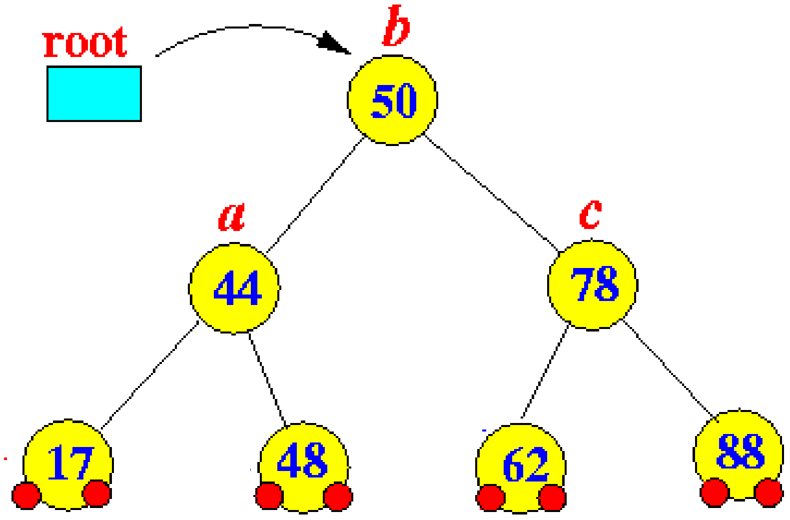
AVL Tree Deletion: Example

- Perform rotation using shaded nodes
  - Node a is the first imbalanced node from the action position
  - Node c is the child node of node a that has the higher height
  - Node b is the child node of node b that has the higher height



AVL Tree Deletion: Example

- The tree after LR rotation



AVL Tree Deletion: Multiple Imbalance

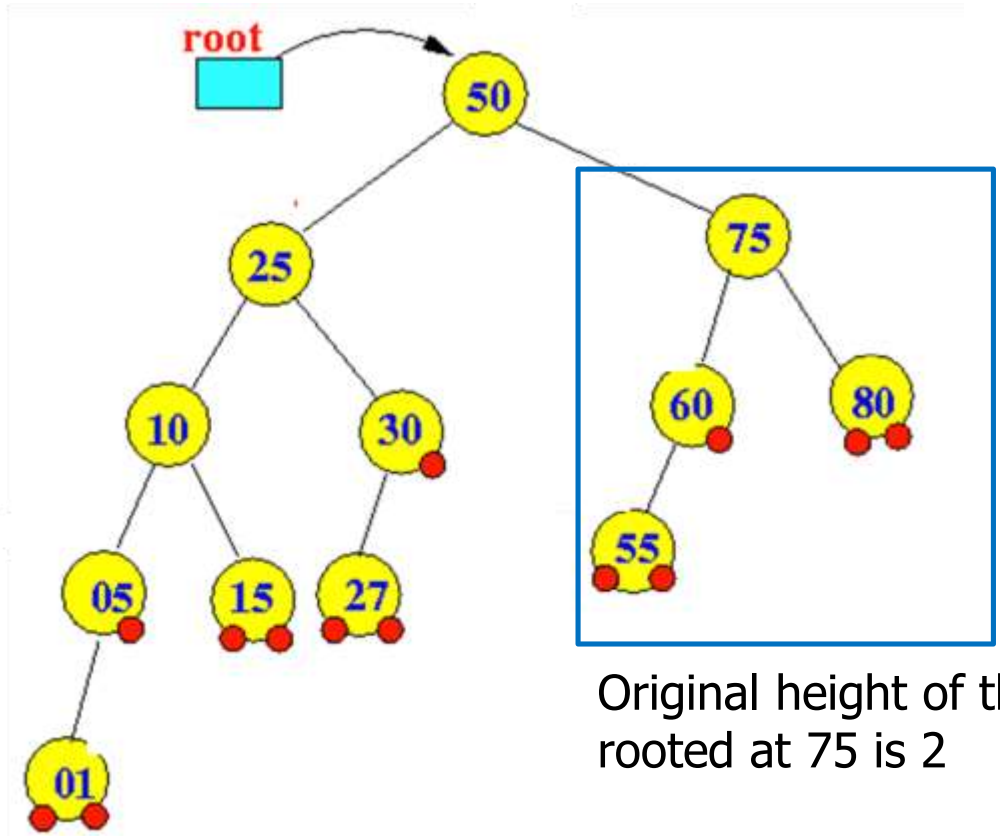
- The imbalance at the first imbalance node due to a deletion operation can be restored using rotation
  - Resulting subtree does not have the same height as the original subtree !!!
  - Nodes that are further up the tree may require re-balancing

- Deleting a node may cause more than one AVL imbalance !!!

- Unfortunately, delete may cause `O(h)` imbalances
  - Insertions will only cause one imbalance that must be fixed

AVL Tree Deletion: Example

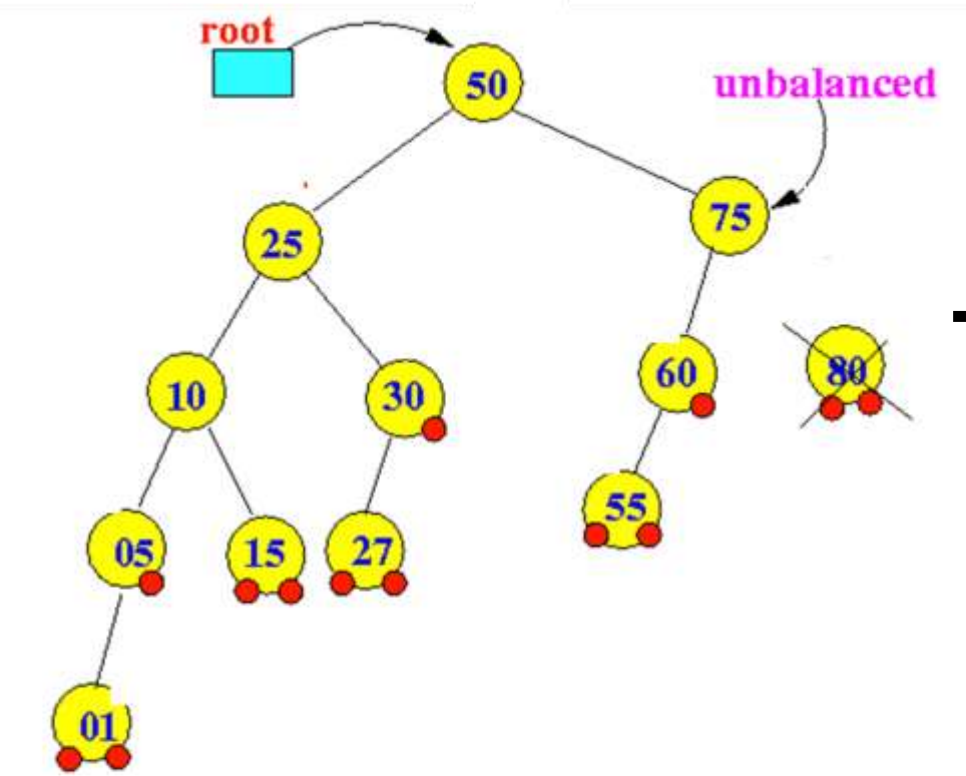
- Consider the following AVL tree

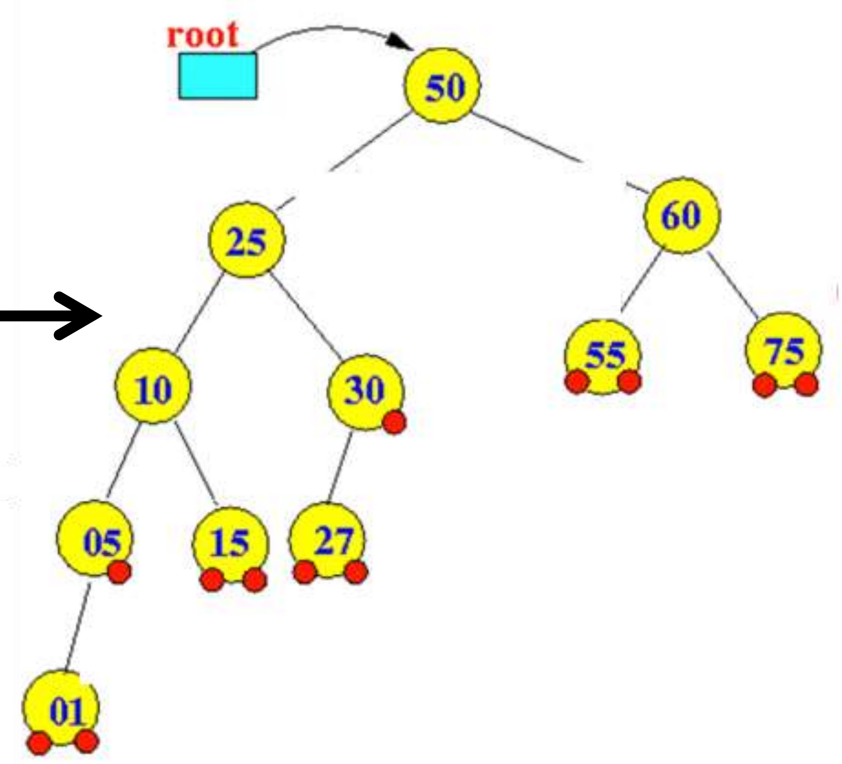


Original height of the subtree rooted at 75 is 2

AVL Tree Deletion: Example

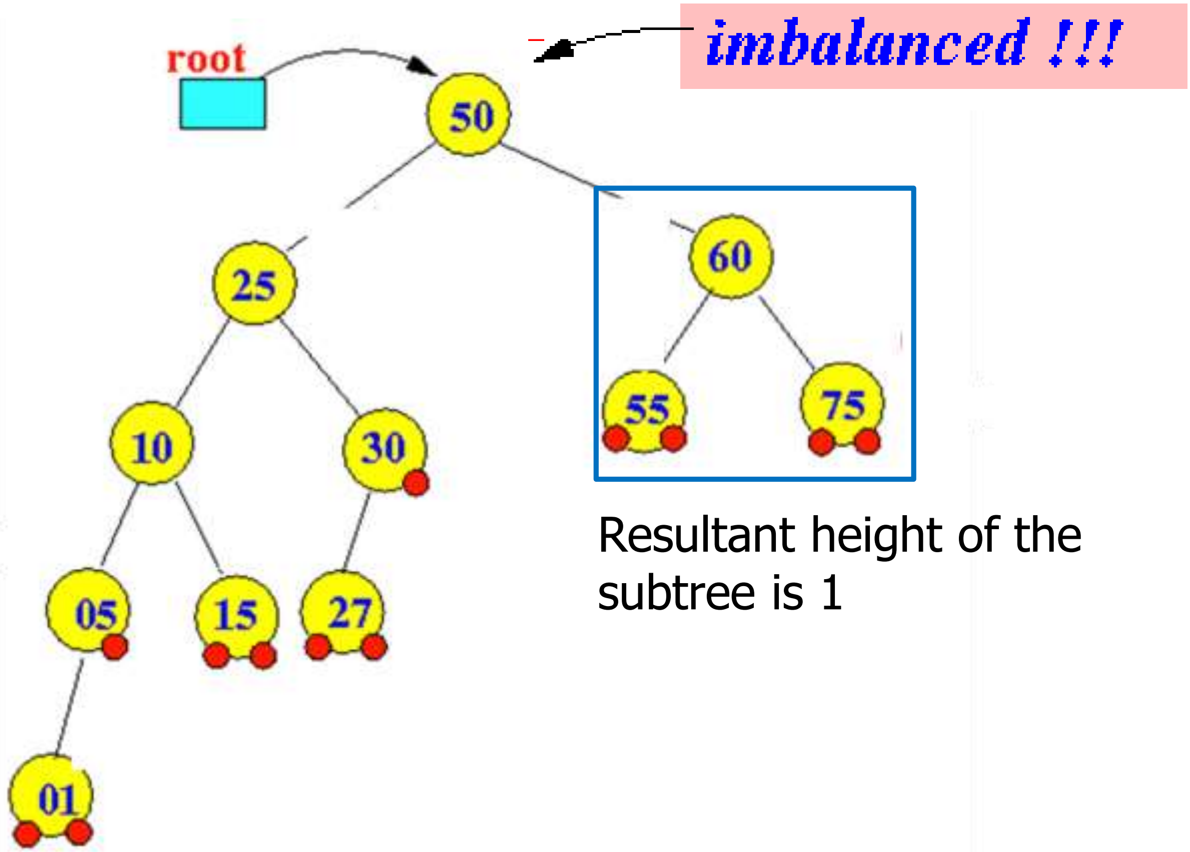
- Node with value 80 is deleted
  - The imbalance is on left-left subtree
  - Imbalance can be fixed using RR rotation





AVL Tree Deletion: Example

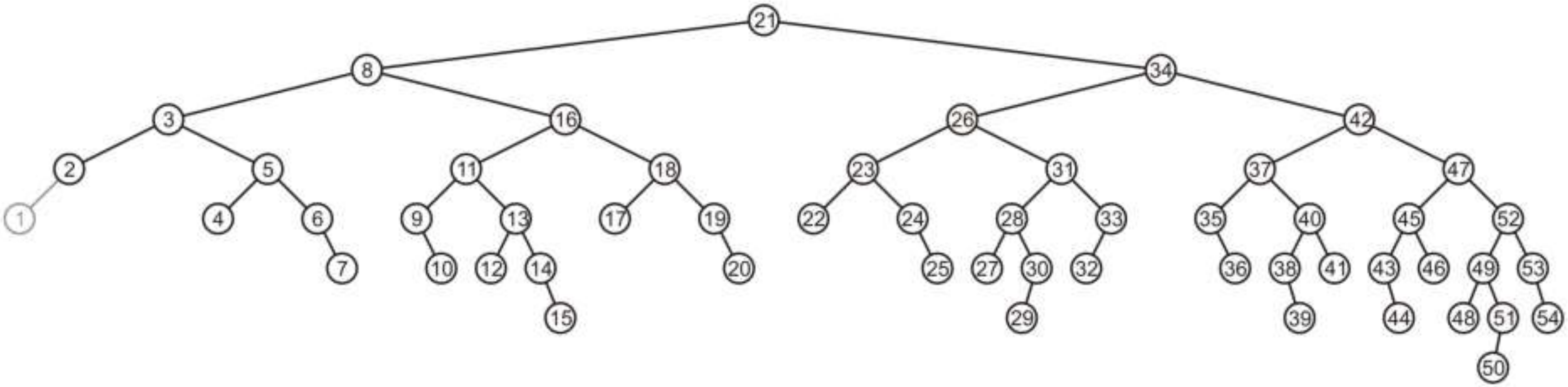
- Node 50 requires re-balancing
  - The imbalance is on left-left subtree
  - Imbalance can be fixed using RR rotation – Home work!!



Resultant height of the subtree is 1

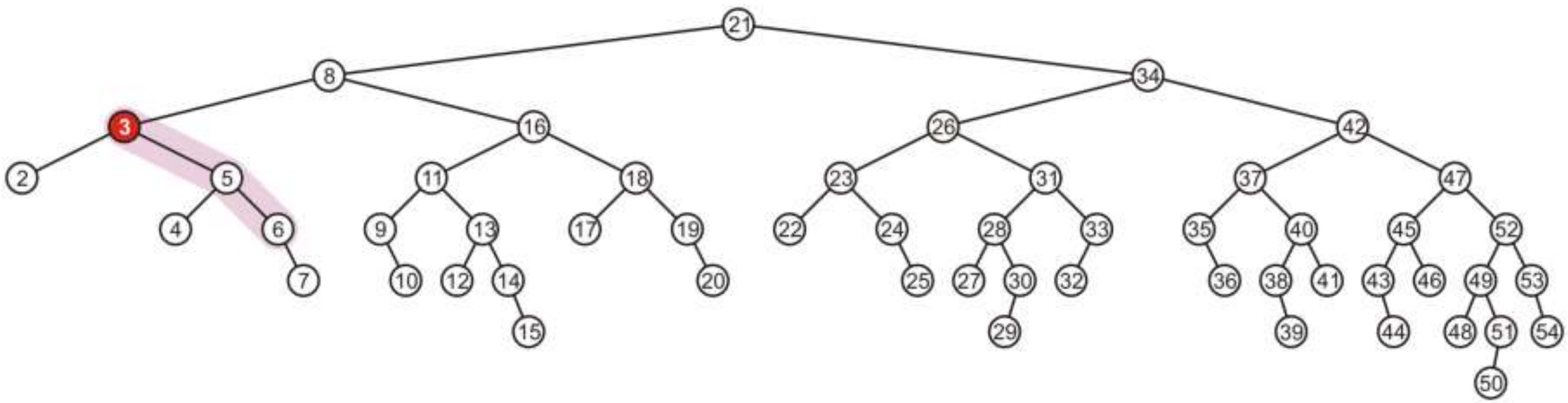
AVL Tree Deletion: Example

- Consider the following AVL tree
  - Suppose node with value 1 is deleted



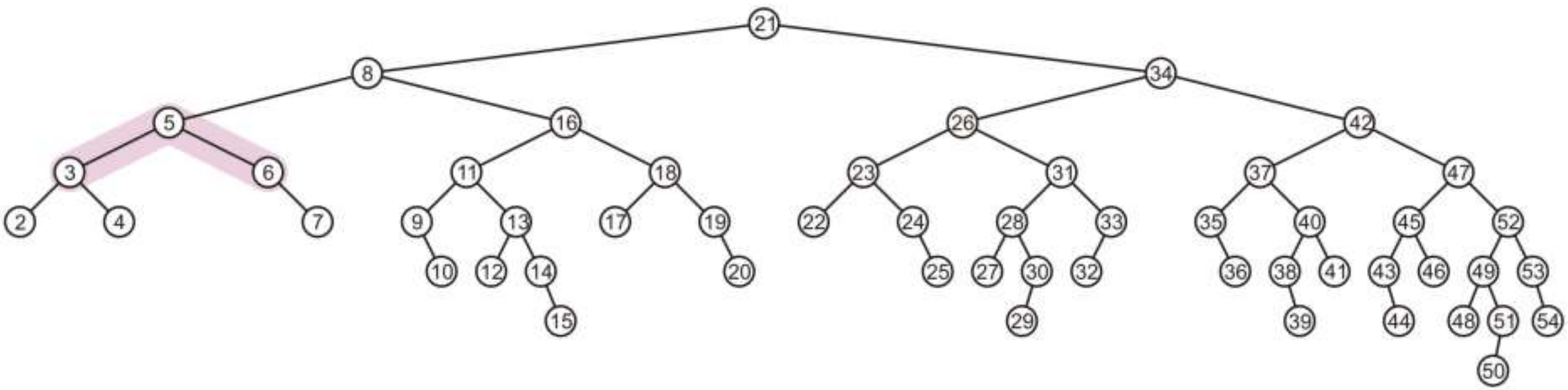
AVL Tree Deletion: Example

- While its previous parent, 2, is not unbalanced, its grandparent 3 is
  - The imbalance is in the right-right subtree



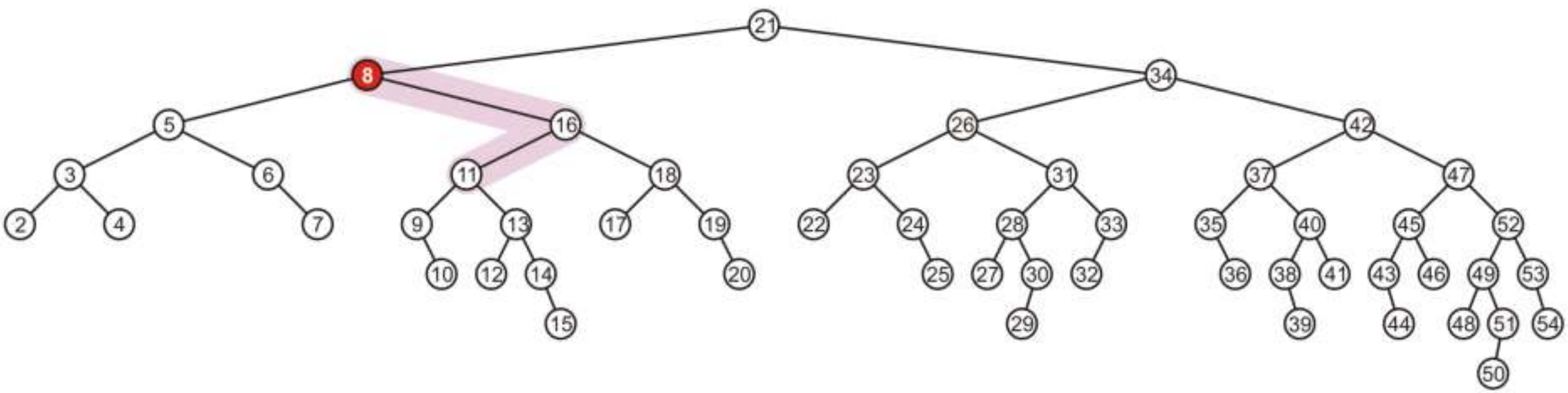
AVL Tree Deletion: Example

- While its previous parent, 2, is not unbalanced, its grandparent 3 is
  - The imbalance is in the right-right subtree
  - Imbalance can be fixed using LL rotation



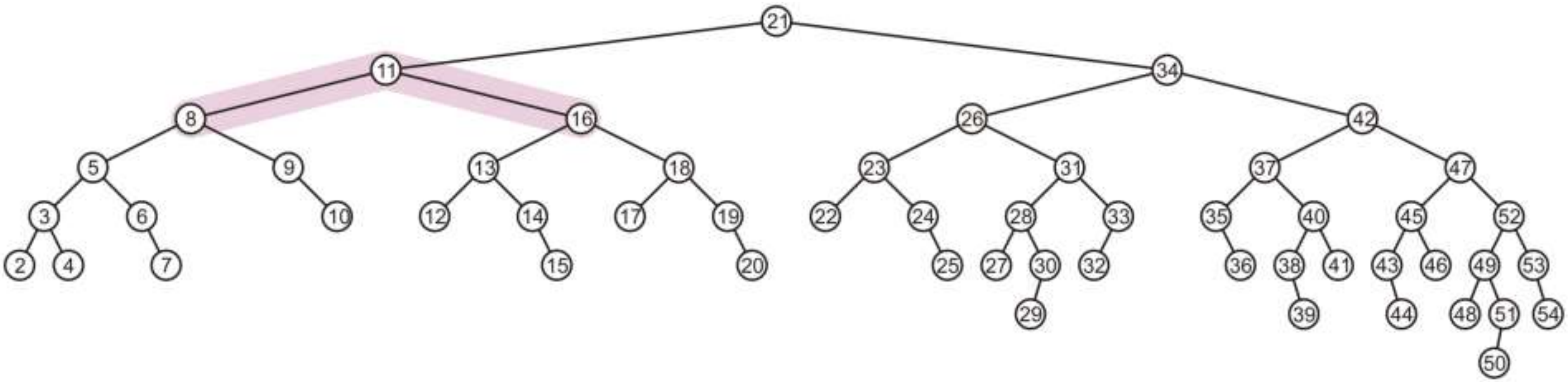
AVL Tree Deletion: Example

- The subtrees of node 5 is now balanced
- Recursing to the root, however, 8 is also unbalanced
  - The imbalance is in right-left subtree



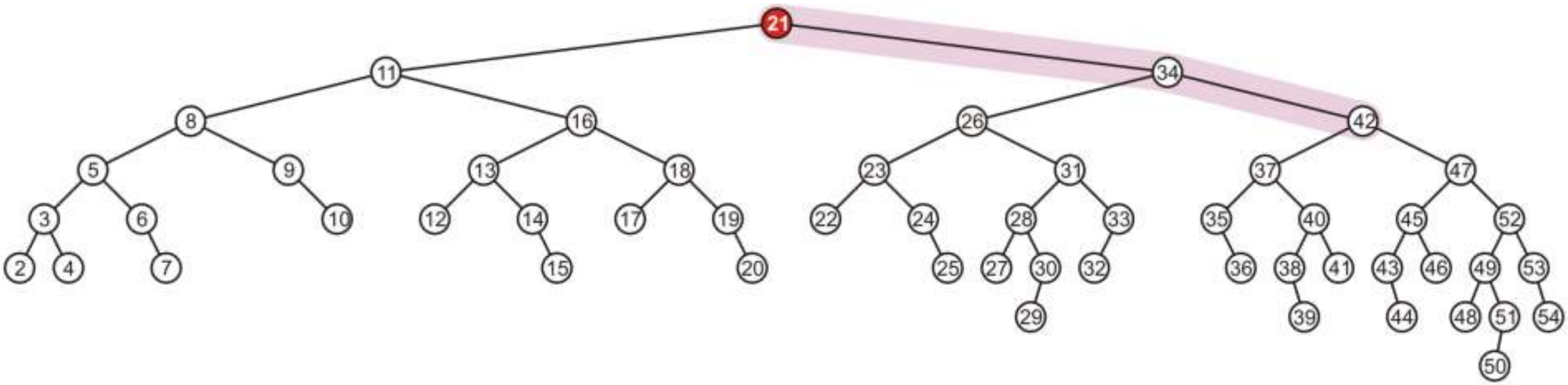
AVL Tree Deletion: Example

- The node with value 8 is unbalanced
  - The imbalance is in right-left subtree
  - LR rotation can fix imbalance



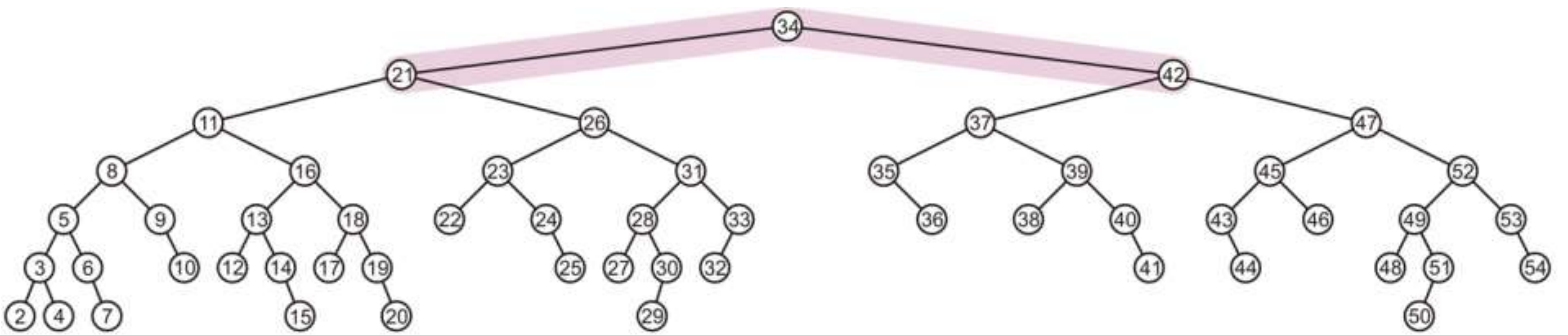
AVL Tree Deletion: Example

- Root 21 is still imbalanced
  - The imbalance is in right-right subtree



AVL Tree Deletion: Example

- Root 21 is still imbalanced
  - The imbalance is in right-right subtree
  - LL rotation can fix the imbalance



Any Question So Far?