AVL Trees

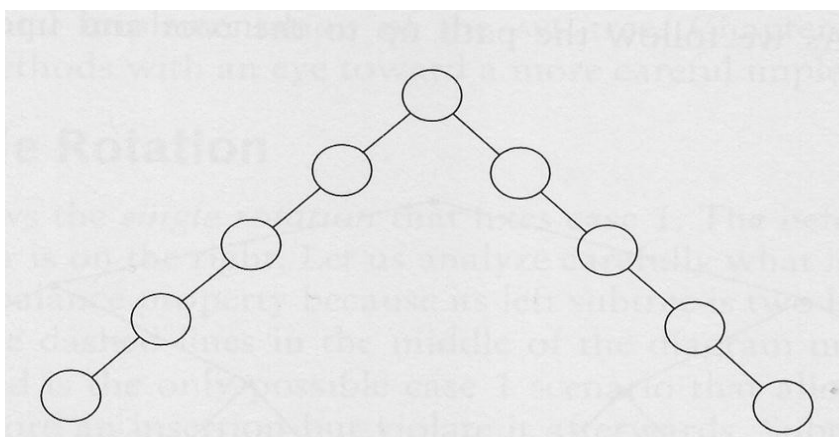
- AVL (Adelson-Velskii and Landis)

  G. Adelson-Velskii, E. M. Landis (1962). "An algorithm for the organization of information". *Proceedings of the USSR Academy of Sciences (in Russian)* 146: 263–266.

- AVL Tree = BST with a balance condition.
- Balance condition => ensures that the tree depth is *O(log N).*
- Idea 1:

  The left and right subtrees must have the same height.

  Problem: does not force the tree to be shallow



AVL Trees

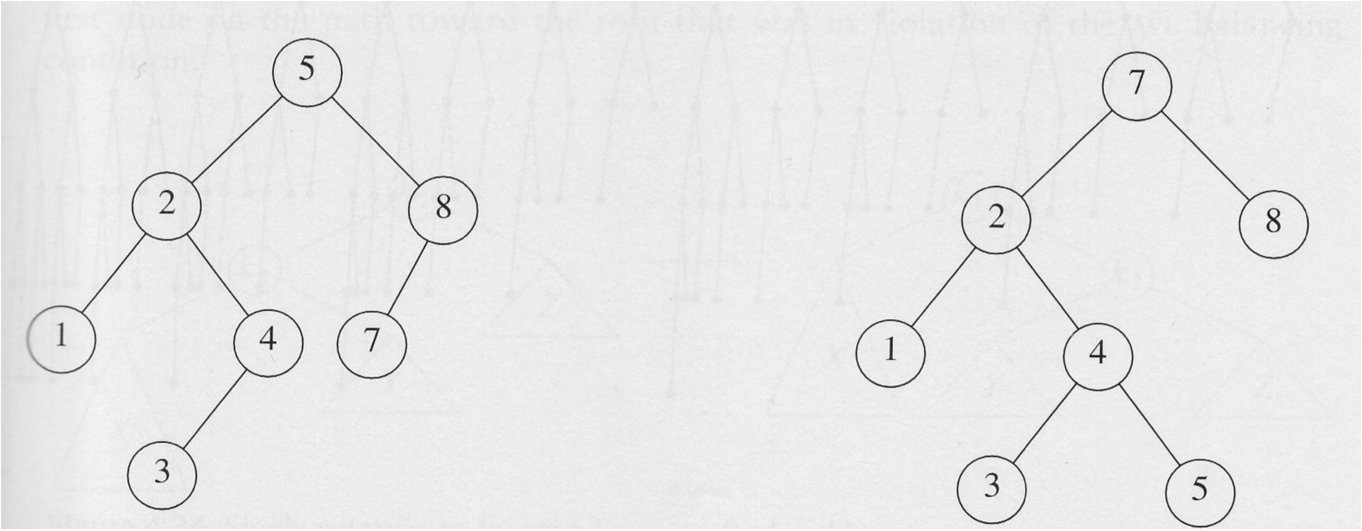
- Idea 2:

  Every node must have left and right subtrees of the same height.

  Problem: only perfectly balanced trees would satisfy it => too rigid to be useful.

- Idea 3: (AVL Tree)

  A BST in which for every node the height of the left and right subtrees can differ by at most 1.



AVL Trees

- Height information must be kept in the node structure.
- Upper bound on the height of an AVL tree of N nodes => 1.44 log(N+2) – 0.328 =O(log N)
- Tree operations => O(log N)
- Insertion => could violate the AVL property.

  The AVL tree property needs to be restored
- Can be done by a simple operation

  => rotation

Insertion

- AVL property violation for a given node x might occur in four cases:
  1. An insertion into the left subtree of the left child of x.
  2. An insertion into the right subtree of the left child of x.
  3. An insertion into the left subtree of the right child of x.
  4. An insertion into the right subtree of the right child of x.
- 1 – 4, and 2 - 3 are mirror image symmetries

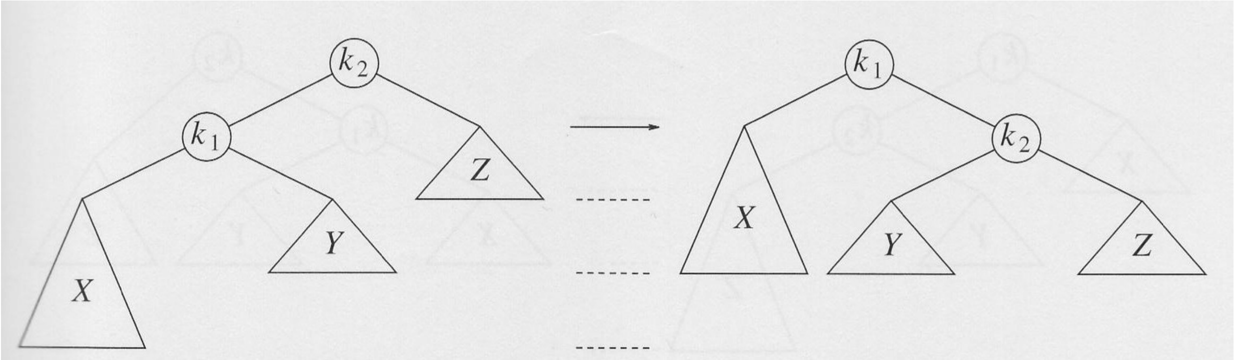
  => only two basic cases
- 1 and 4: insertion occurs on the “outside”

  => single rotation
- 2 and 3: insertion occurs on the “inside”

  => double rotation

Single Rotation

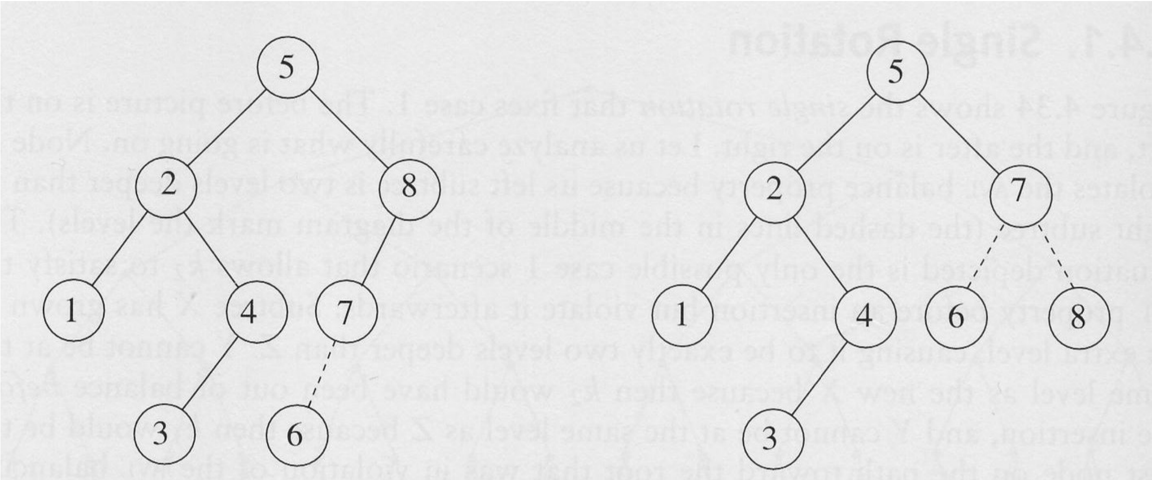
Case 1:



The height of the entire subtree is the same as before insertion

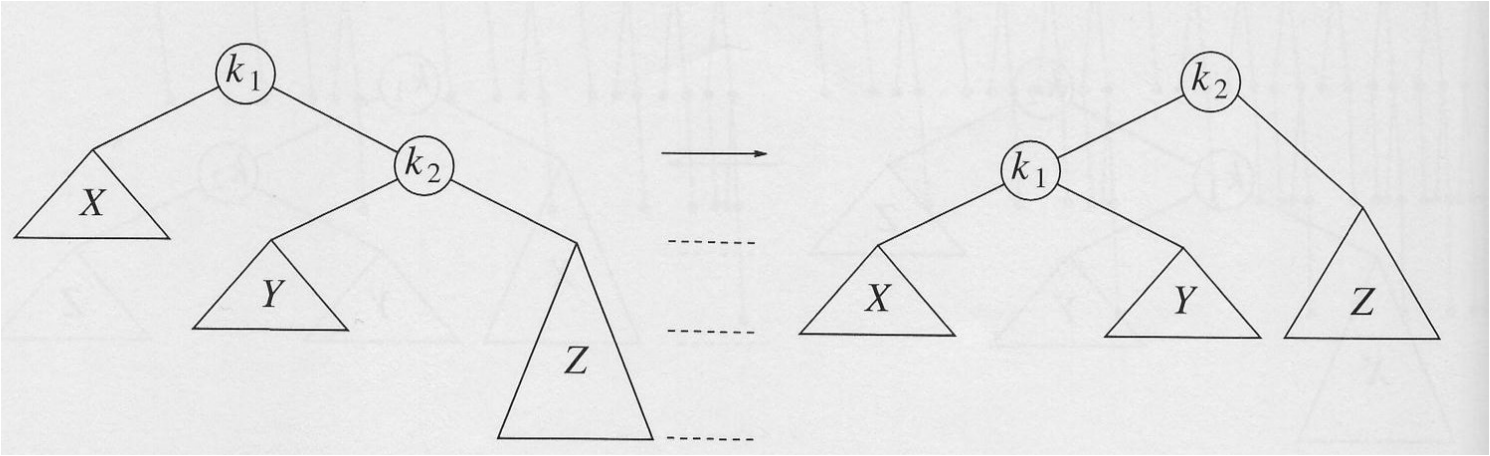
=> No further updates of the height or rotations on the path to the root are needed.

Example:



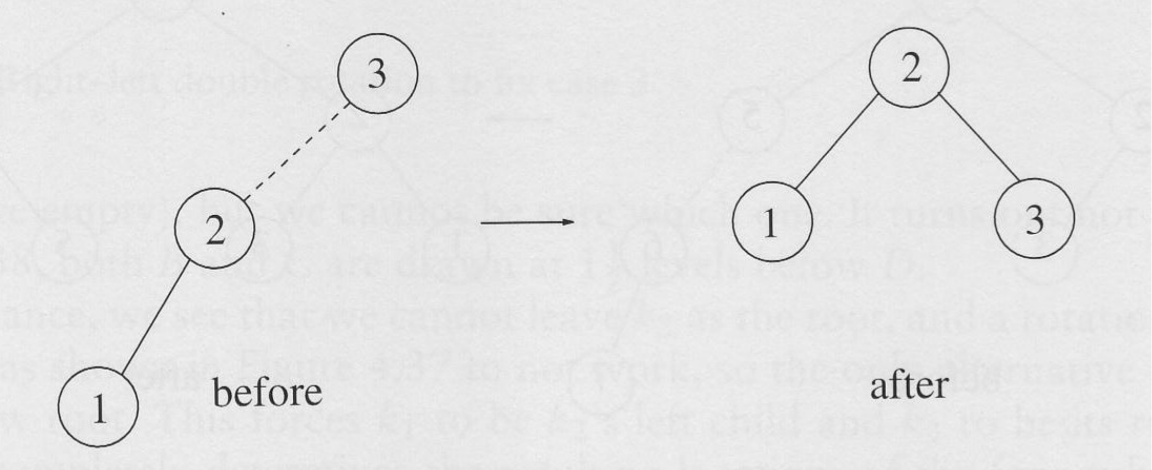
Single Rotation

- Case 4:

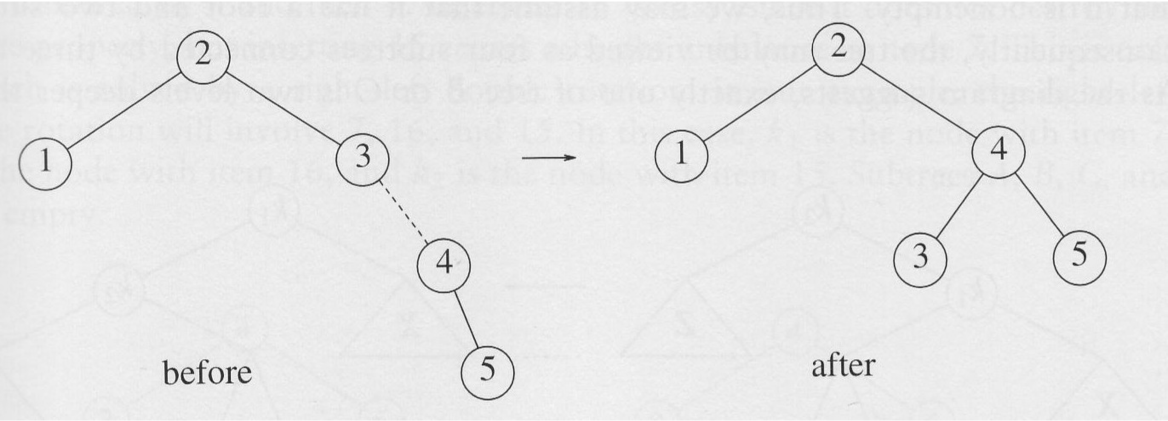


Single Rotation: Example

Insert: 3, 2, 1

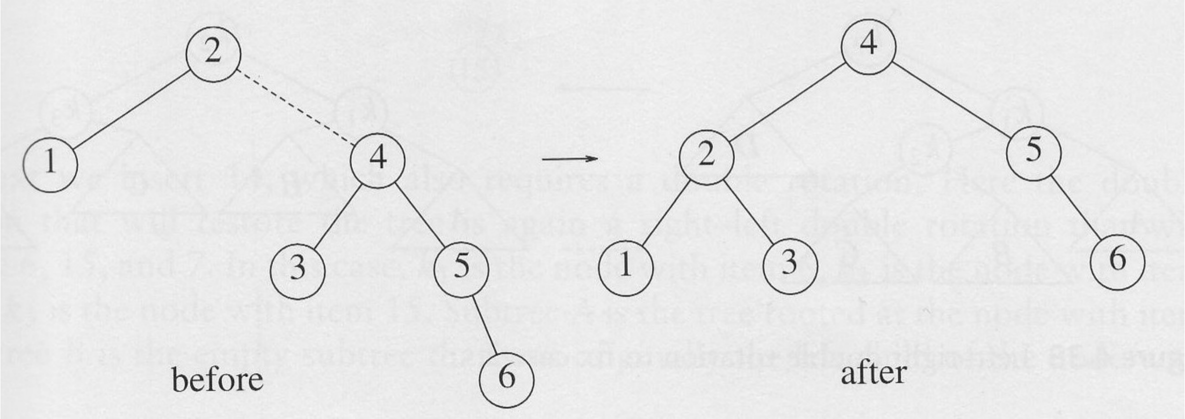


Insert: 4, 5

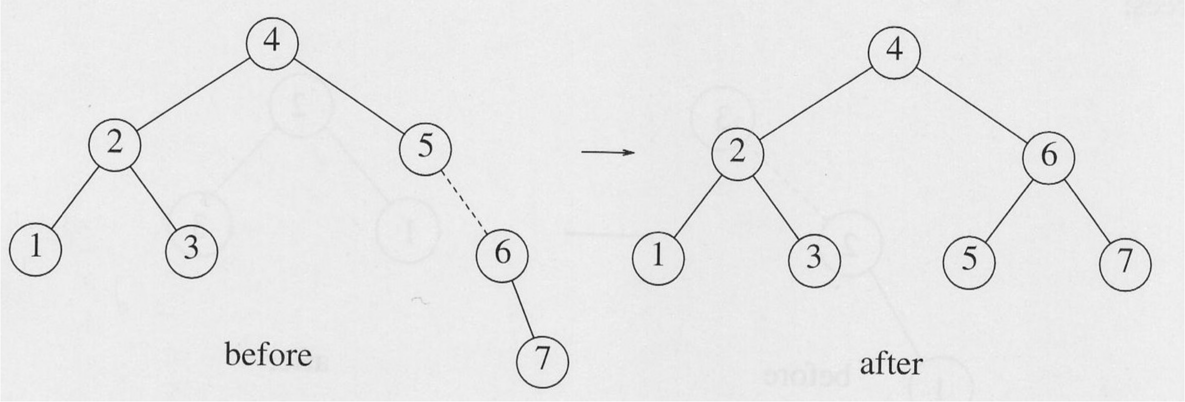


Single Rotation: Example

Insert: 6

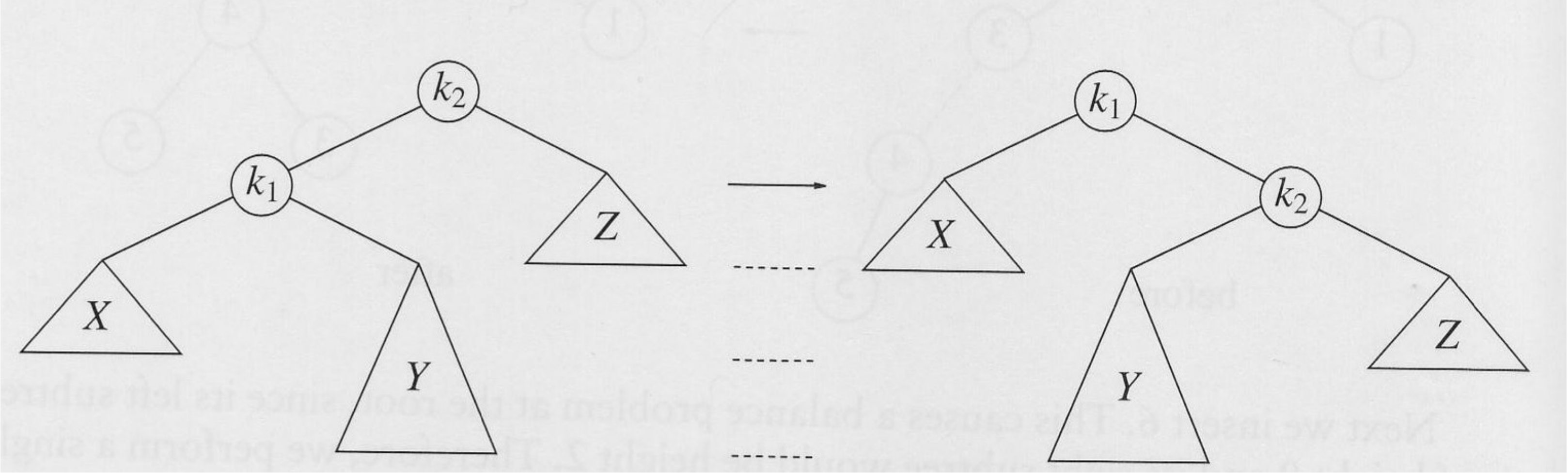


Insert: 7



Double Rotation

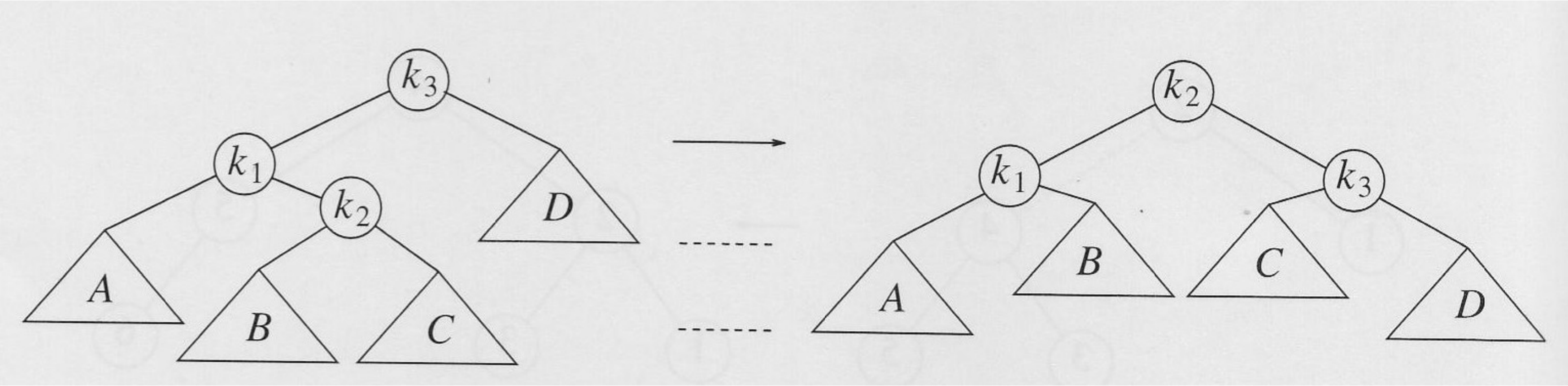
Single rotation fails to fix case 2 (left child – right subtree:



Double rotation solves the problem.

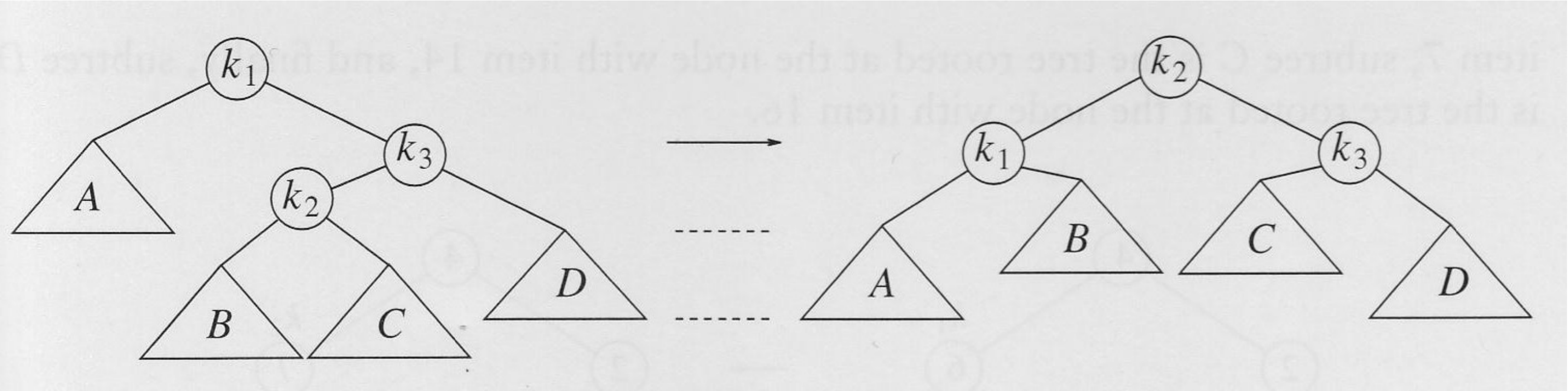
Left-Right Double Rotation

- Fixes case 2.
- First rotate between x's child and grandchild and then between x and its new child.



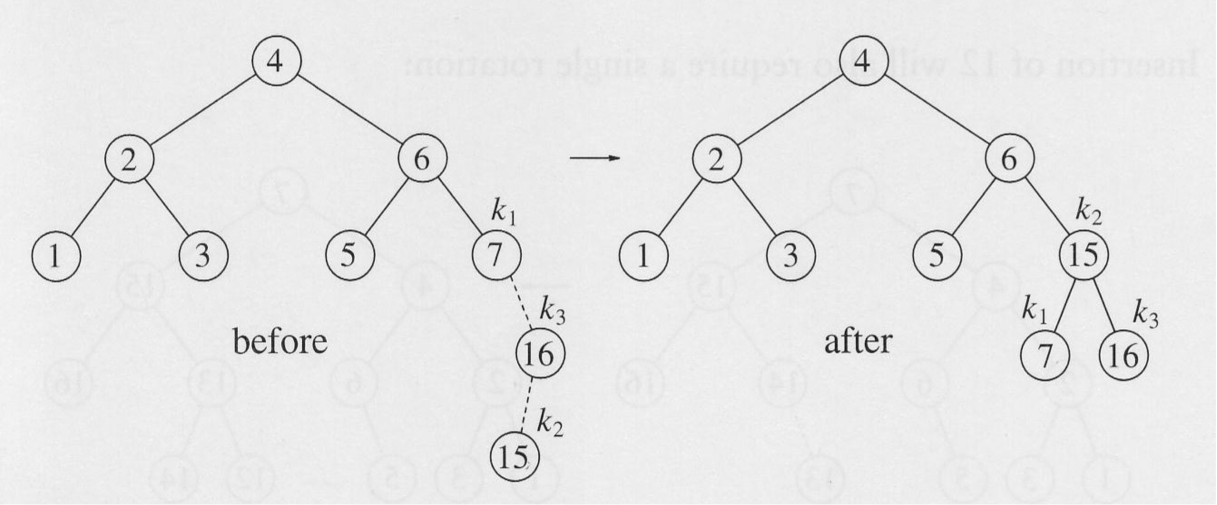
Right-Left Double Rotation

- Fixes case 3.
- First rotate between x's child and grandchild and then between x and its new child.

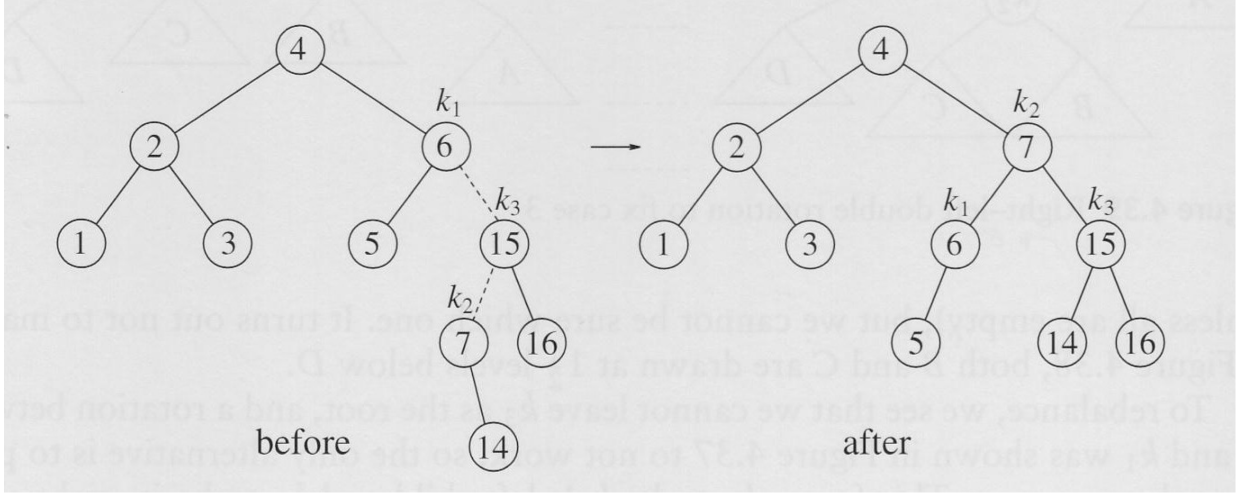


Double Rotation: Example

Insert: 16, 15 => case 3

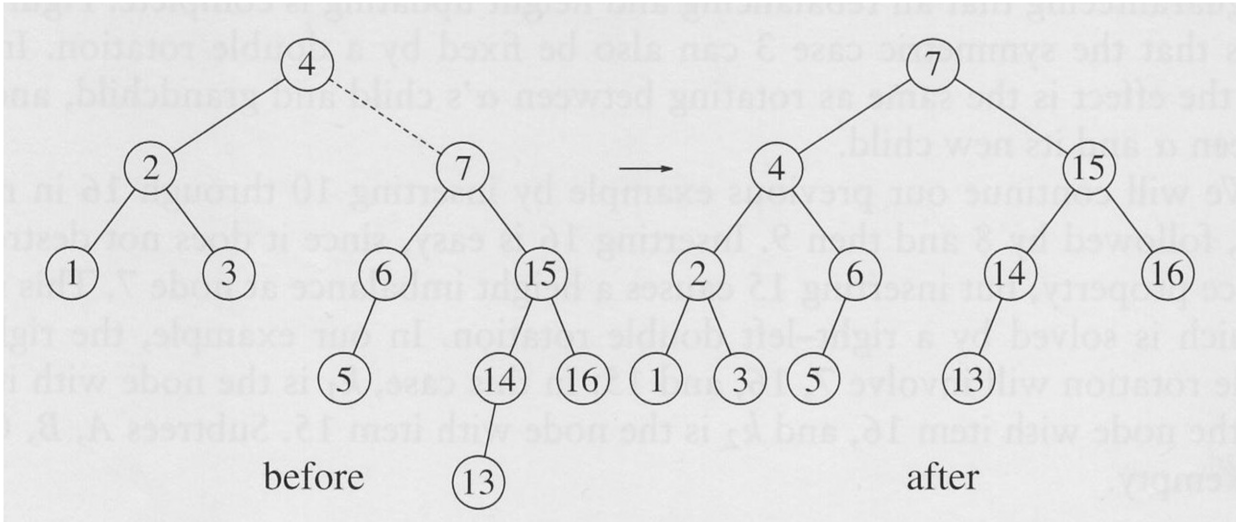


Insert: 14 => case 3

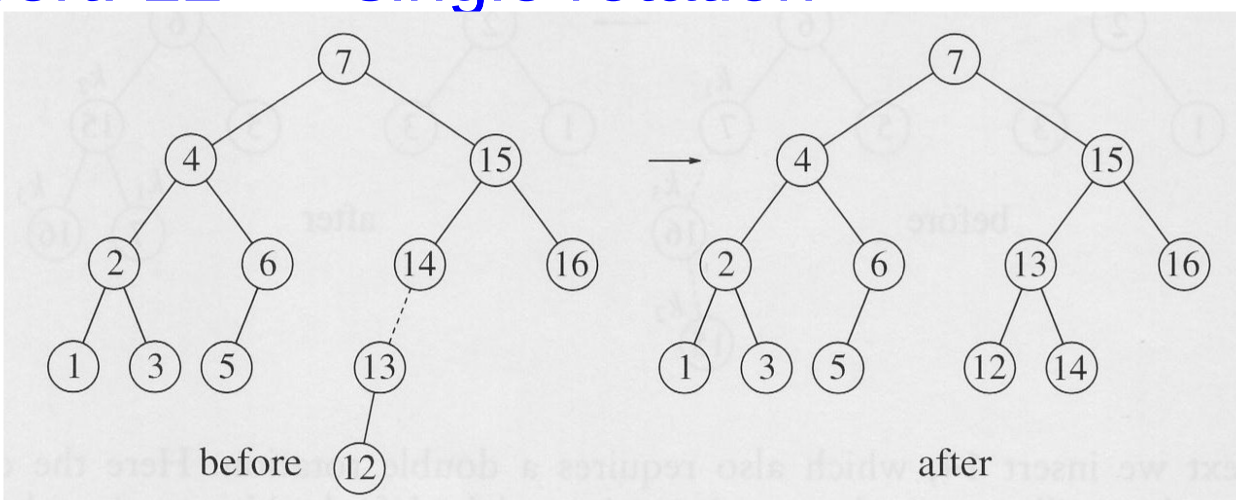


Double Rotation: Example

Insert: 13 => imbalance at the root => single rotation



Insert: 12 => single rotation



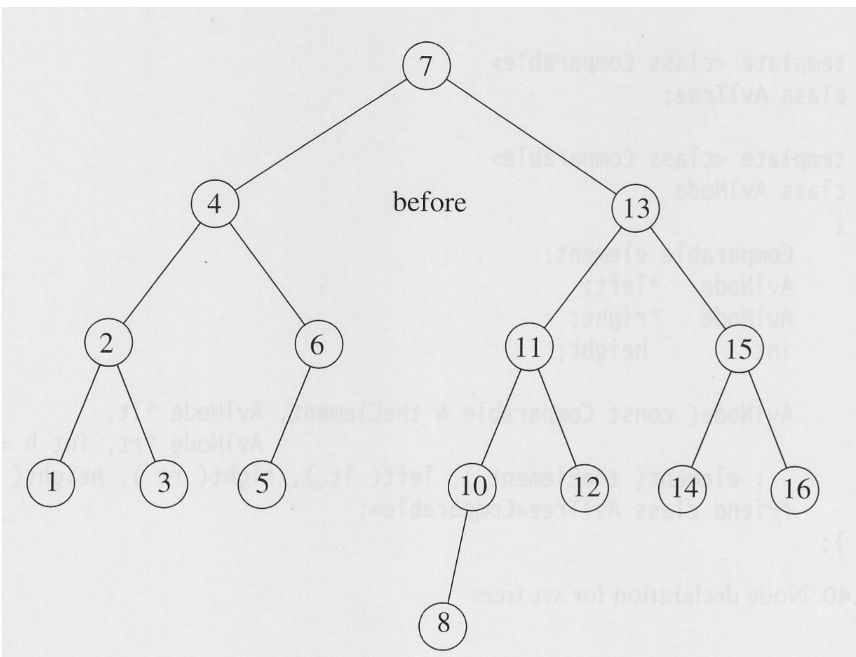
Double Rotation: Example

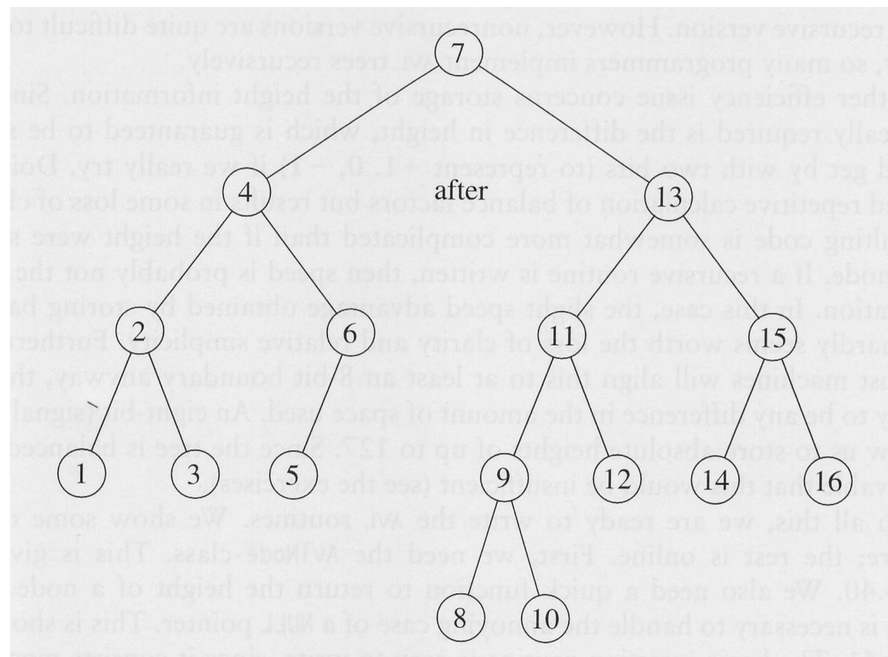
Insert: 11 => single rotation

Insert: 10 => single rotation

Insert: 8 => no rotation

Insert: 9 => node 10 unbalanced, case 2, left-right rotation





AVLNode Class

```
Struct 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

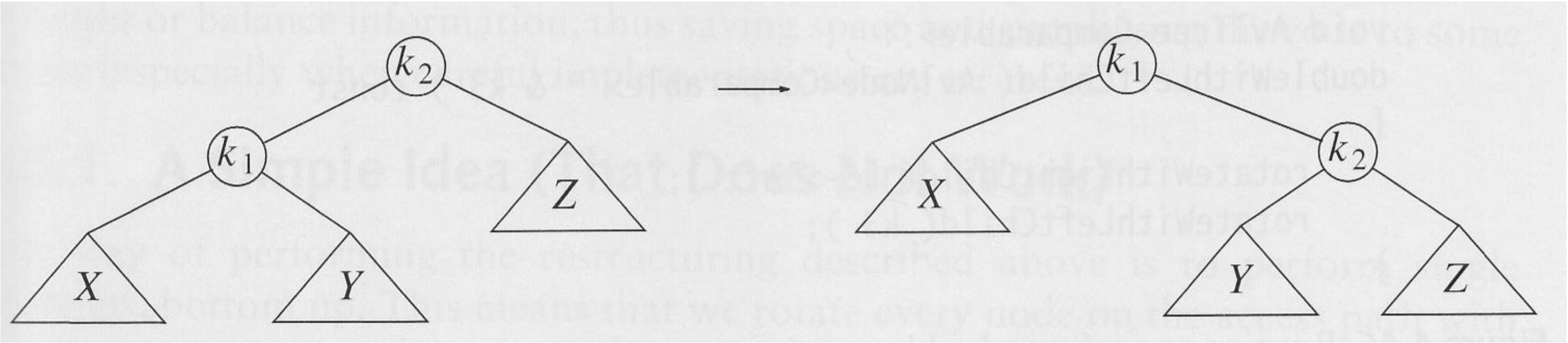
```
/**
* Return the height of node t or -1 if NULL.
*/

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

insert

```
void insert( const Comparable & x, AvlNode * & 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 )   //check if left subtree
                    rotateWithLeftChild( t );  //case 1
              else
                    doubleWithLeftChild( t );  //case 2
  }
  else if( t->element < x )
  {
        insert( x, t->right );
        if( height( t->right ) - height( t->left ) == 2 )
              if( t->right->element < x )          //check if right subtree
                    rotateWithRightChild( t );   //case 4
              else
                    doubleWithRightChild( t );  //case 3
  }
  else
        ; // Duplicate; do nothing
  t->height = max( height( t->left ), height( t->right ) ) + 1;
}
```

Single rotation: Case 1

```
void rotateWithLeftChild( AvlNode * & k2 ) const
{
 AvlNode *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: Case 2

```
/**
* 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.
*/
void doubleWithLeftChild( AvlNode * & k3 ) const
{
    rotateWithRightChild( k3->left );
    rotateWithLeftChild( k3 );
}
```