AVL Trees

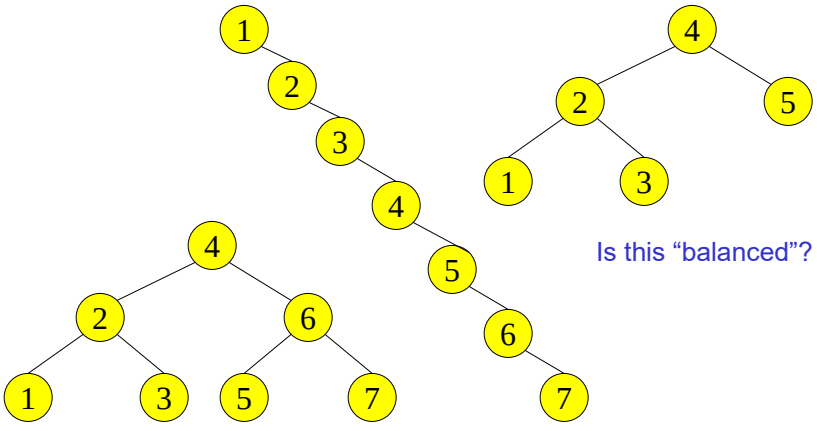
Binary Search Tree Best Time

- All BST operations are O(h), where d is tree height.
- maximum h is $h = \lfloor \log_2 N \rfloor$ for a binary tree with N nodes
  - What is the best case tree?
  - What is the worst case tree?
- So, best case running time of BST operations is O(log N)

Binary Search Tree
Worst Time

- 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 heights of left and right subtree
  - › Unbalanced degenerate tree

Balanced and unbalanced BST

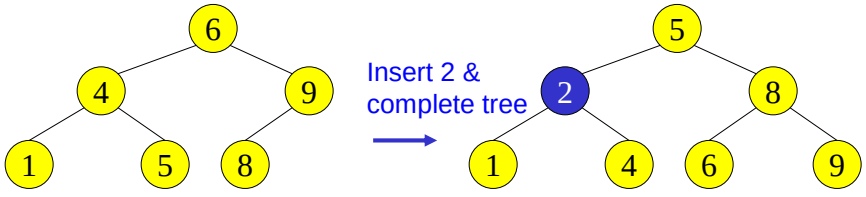


Balancing Binary Search Trees

- 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

Perfect Balance

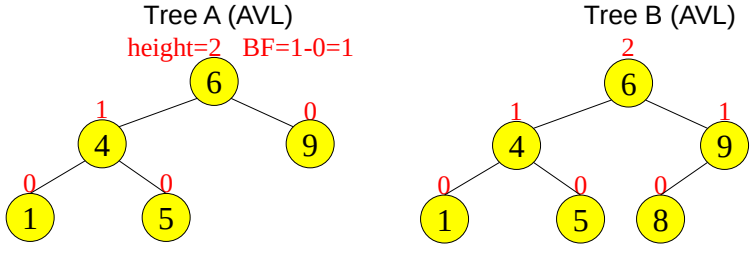
- Want a complete tree after every operation
  - › tree is full except possibly in the lower right
- This is expensive
  - › For example, insert 2 in the tree on the left and then rebuild as a complete tree



AVL - Good but not Perfect Balance

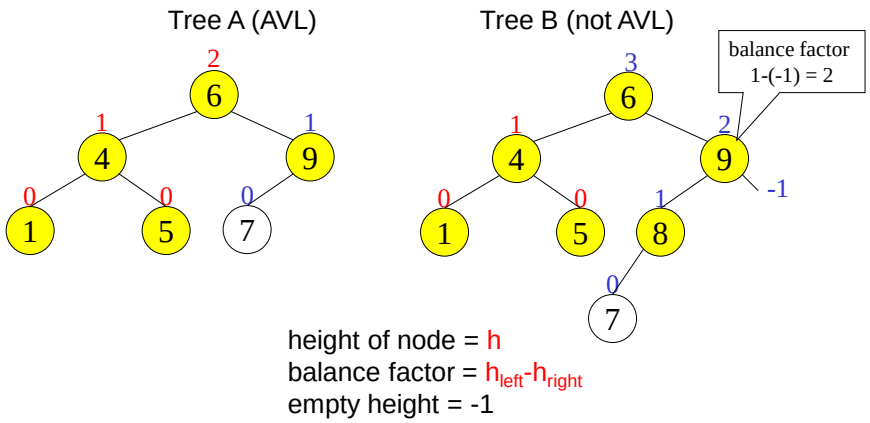
- AVL trees are height-balanced binary search trees
- Balance factor of a node
  - › height(left subtree) - height(right subtree)
- An AVL tree has balance factor calculated at every node
  - › For every node, heights of left and right subtree can differ by no more than 1
  - › Store current heights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

Node Heights



height of node = h
balance factor = h_{left}-h_{right}
empty height = -1

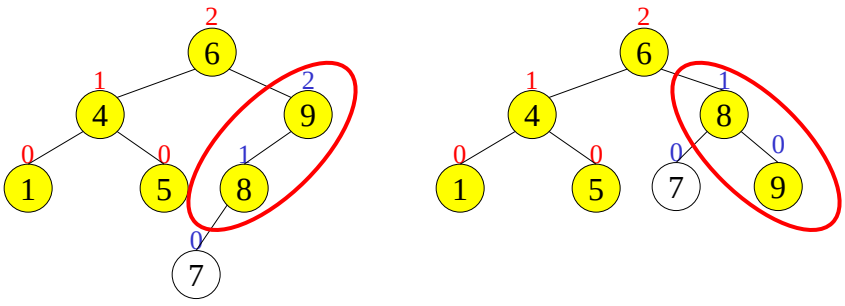
Node Heights after Insert 7



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

Single Rotation in an AVL Tree



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.

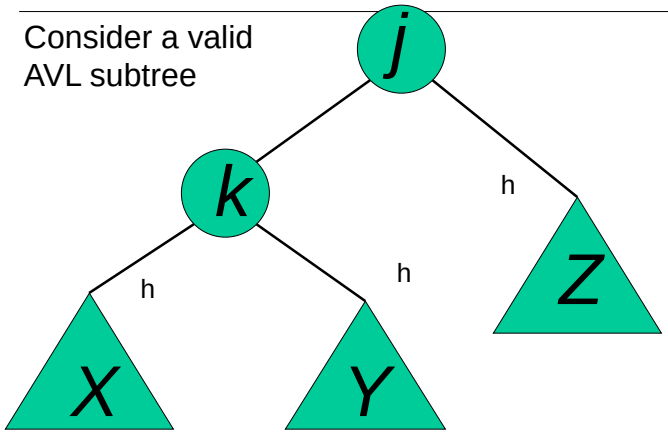
Inside Cases (require double rotation) :

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.

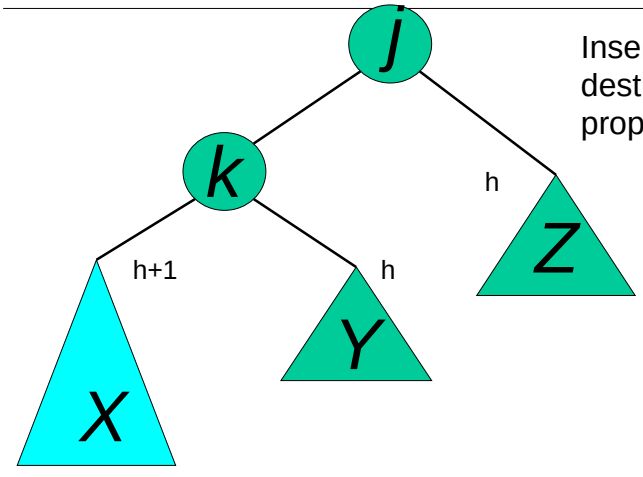
AVL Insertion: Outside Case

Consider a valid AVL subtree

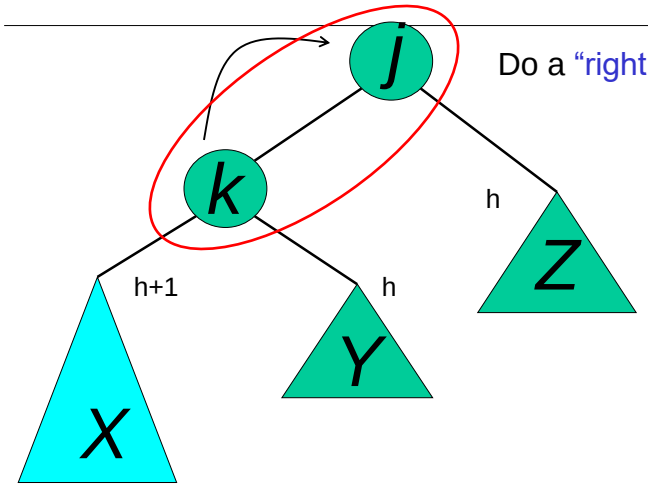


AVL Insertion: Outside Case

Inserting into X destroys the AVL property at node j

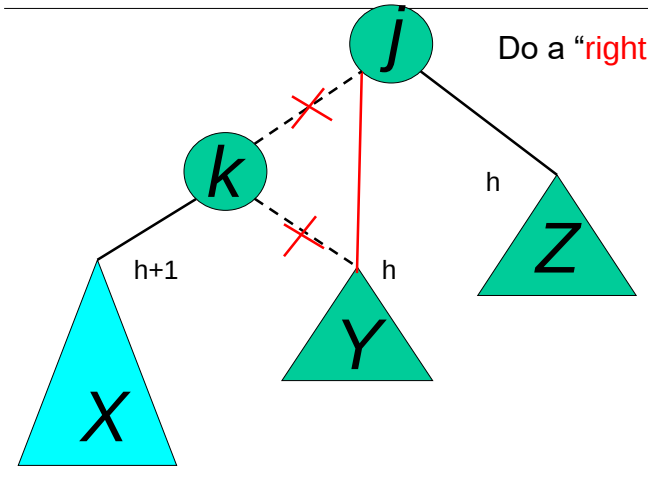


AVL Insertion: Outside Case



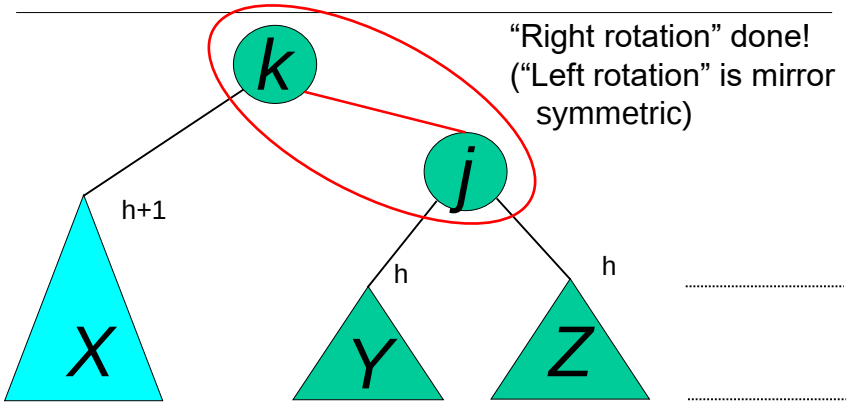
Do a "right rotation"

Single right rotation



Do a "right rotation"

Outside Case Completed

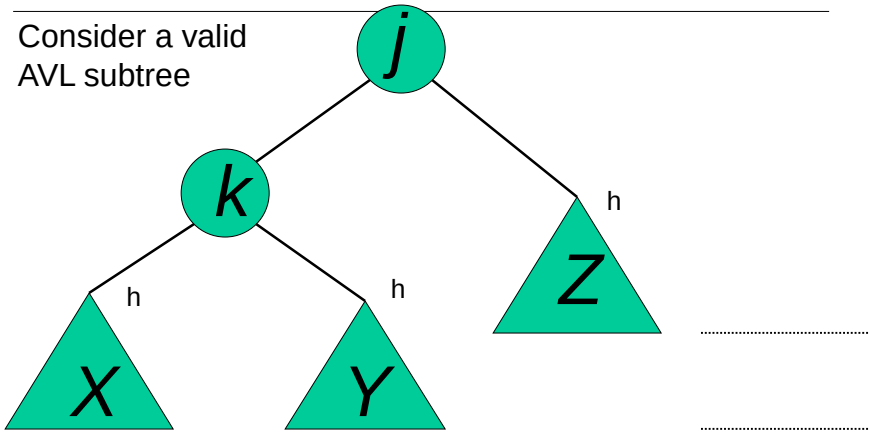


"Right rotation" done!
("Left rotation" is mirror symmetric)

AVL property has been restored!

AVL Insertion: Inside Case

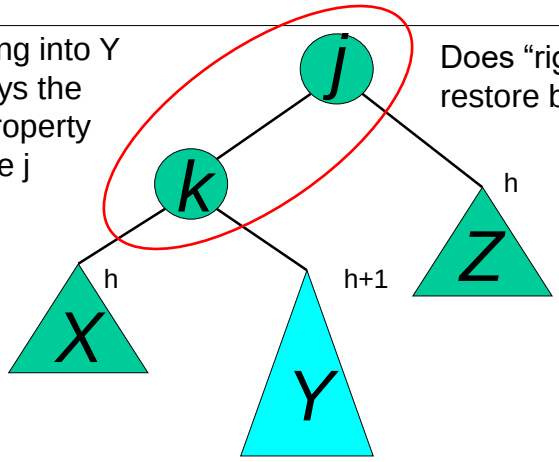
Consider a valid AVL subtree



AVL Insertion: Inside Case

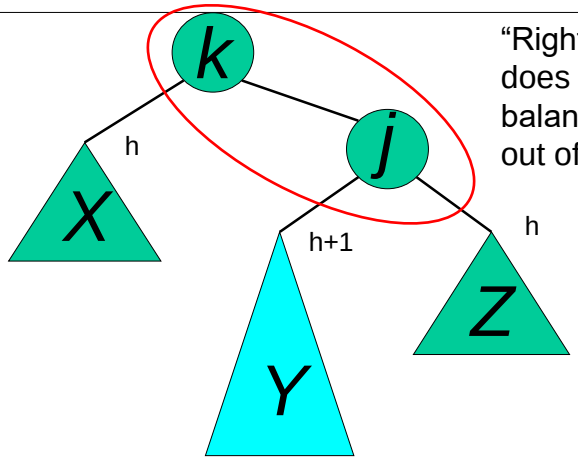
Inserting into Y destroys the AVL property at node j

Does “right rotation” restore balance?



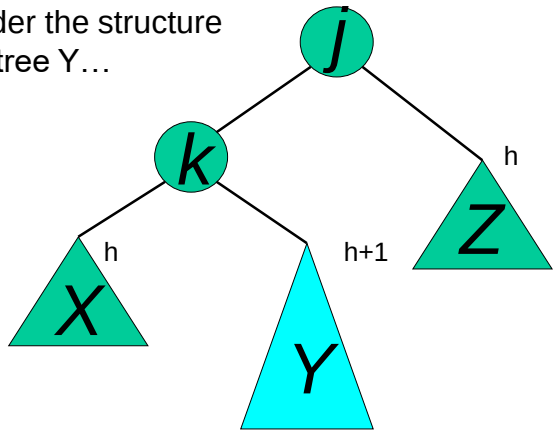
AVL Insertion: Inside Case

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



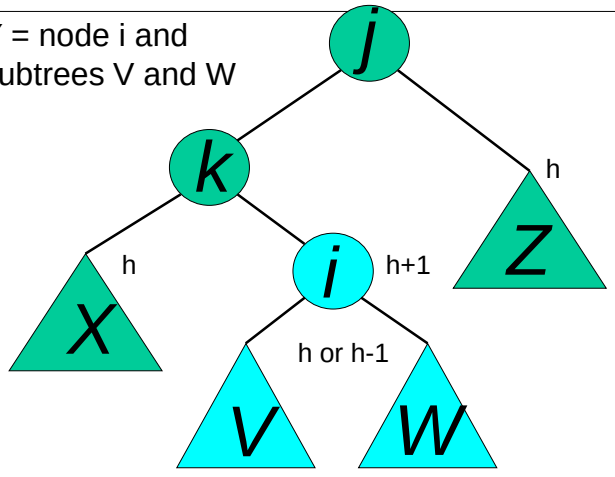
AVL Insertion: Inside Case

Consider the structure of subtree Y…

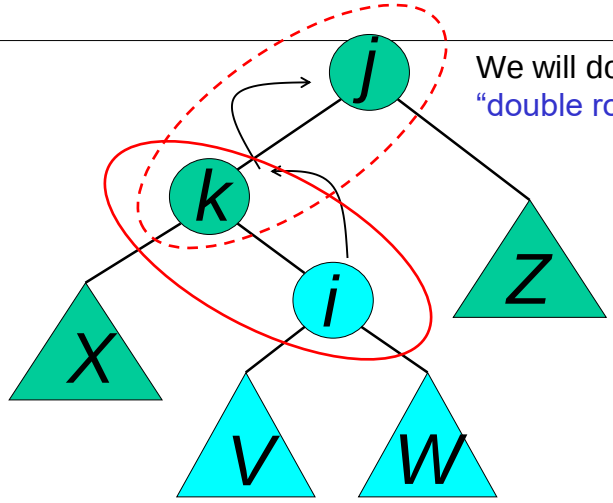


AVL Insertion: Inside Case

Y = node i and subtrees V and W

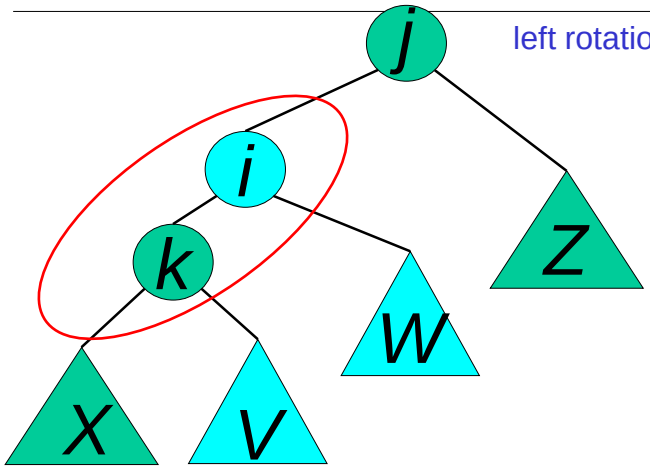


AVL Insertion: Inside Case



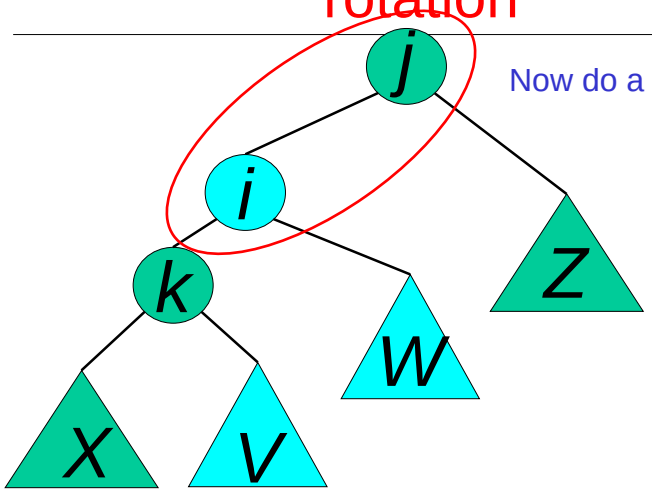
We will do a left-right "double rotation" . . .

Double rotation : first rotation



left rotation complete

Double rotation : second rotation

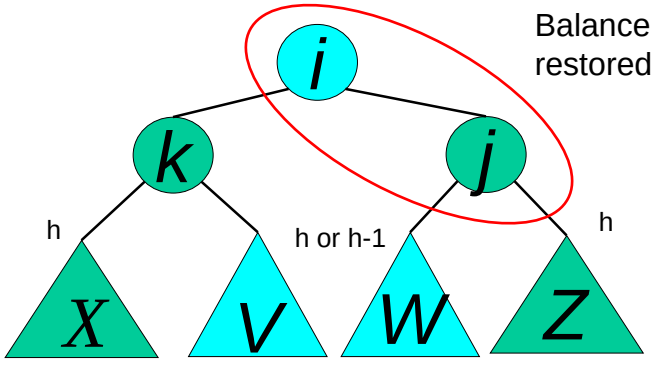


Now do a right rotation

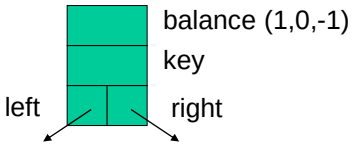
Double rotation : second rotation

right rotation complete

Balance has been restored



Implementation



No need to keep the height; just the difference in height, i.e. the balance factor; this has to be modified on the path of insertion even if you don't perform rotations

Once you have performed a rotation (single or double) you won't need to go back up the tree


```
struct AvlNode;
typedef struct AvlNode *Position;
typedef struct AvlNode *AvlTree;

/* Place in the implementation file */
struct AvlNode
{
    ElementType Element;
    AvlTree  Left;
    AvlTree  Right;
    int      Height;
};
```

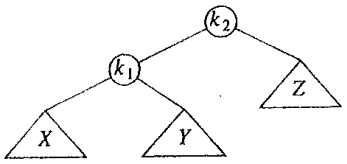
```
static int
Height( Position P )
{
    if( P == NULL )
        return -1;
    else
        return P->Height;
}
```

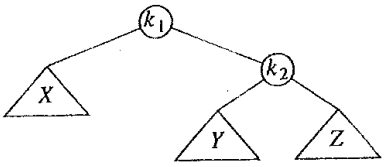
```
private static class AvlNode<AnyType>
{
        // Constructors
    AvlNode( AnyType theElement )
      { this( theElement, null, null ); }

    AvlNode( AnyType theElement, AvlNode<AnyType> lt, AvlNode<AnyType> rt )
      { element = theElement; left = lt; right = rt; height = 0; }

    AnyType           element;      // The data in the node
    AvlNode<AnyType>  left;         // Left child
    AvlNode<AnyType>  right;        // Right child
    int               height;       // Height
}
```

```
private int height( AvlNode<AnyType> t )
{
    return t == null ? -1 : t.height;
}
```


Single Rotation

```
static Position
SingleRotateWithLeft( Position K2 )
{
    Position K1;

    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;

    return K1;  /* New root */
}
```



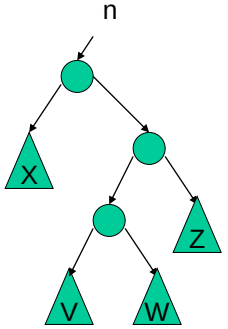


```
private AvlNode<AnyType> rotateWithLeftChild( AvlNode<AnyType> k2 )
{
    AvlNode<AnyType> k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;
    k2.height = Math.max( height( k2.left ), height( k2.right ) ) + 1;
    k1.height = Math.max( height( k1.left ), k2.height ) + 1;
    return k1;
}
```


Double Rotation

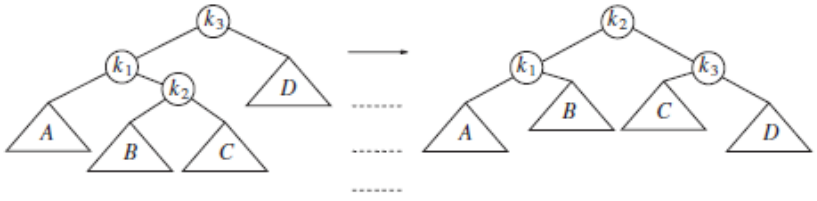
- Implement Double Rotation in two lines.

```
DoubleRotateFromRight(n : reference node pointer) {
????
}
```



Double Rotation

```
static Position
DoubleRotateWithLeft( Position K3 )
{
    /* Rotate between K1 and K2 */
    K3->Left = SingleRotateWithRight( K3->Left );

    /* Rotate between K3 and K2 */
    return SingleRotateWithLeft( K3 );
}
```




```
private AvlNode<AnyType> doubleWithLeftChild( AvlNode<AnyType> k3 )
{
    k3.left = rotateWithRightChild( k3.left );
    return rotateWithLeftChild( k3 );
}
```

Insertion in AVL Trees

- Insert at the leaf (as for all BST)
  - › 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

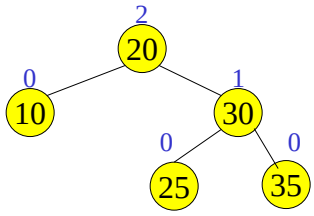

```
AvlTree
Insert( ElementType X, AvlTree T )
{
    if( T == NULL )
    {
        /* Create and return a one-node tree */
        T = malloc( sizeof( struct AvlNode ) );
        if( T == NULL )
            FatalError( "Out of space!!!" );
        else
        {
            T->Element = X; T->Height = 0;
            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 );
            else
                T = DoubleRotateWithLeft( 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 );
            else
                T = DoubleRotateWithRight( T );
    }
    /* Else X is in the tree already; we'll do nothing */

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

```
private AvlNode<AnyType> insert( AnyType x, AvlNode<AnyType> t )
{
    if( t == null )
        return new AvlNode<>( x, null, null );

    int compareResult = x.compareTo( t.element );

    if( compareResult < 0 )
        t.left = insert( x, t.left );
    else if( compareResult > 0 )
        t.right = insert( x, t.right );
    else
        ;  // Duplicate; do nothing
    return balance( t );
}
```



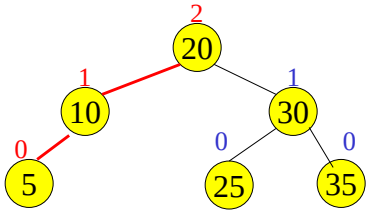
```
private static final int ALLOWED_IMBALANCE = 1;

// Assume t is either balanced or within one of being balanced
private AvlNode<AnyType> balance( AvlNode<AnyType> t )
{
    if( t == null )
        return t;

    if( height( t.left ) - height( t.right ) > ALLOWED_IMBALANCE )
        if( height( t.left.left ) >= height( t.left.right ) )
            t = rotateWithLeftChild( t );
        else
            t = doubleWithLeftChild( t );
    else
    if( height( t.right ) - height( t.left ) > ALLOWED_IMBALANCE )
        if( height( t.right.right ) >= height( t.right.left ) )
            t = rotateWithRightChild( t );
        else
            t = doubleWithRightChild( t );

    t.height = Math.max( height( t.left ), height( t.right ) ) + 1;
    return t;
}
```

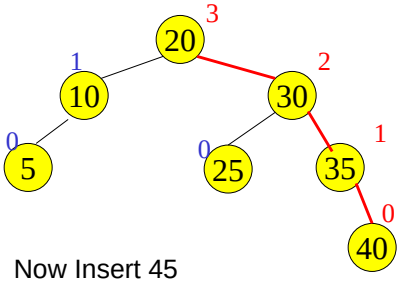
Example of Insertions in an AVL Tree



Insert 5, 40

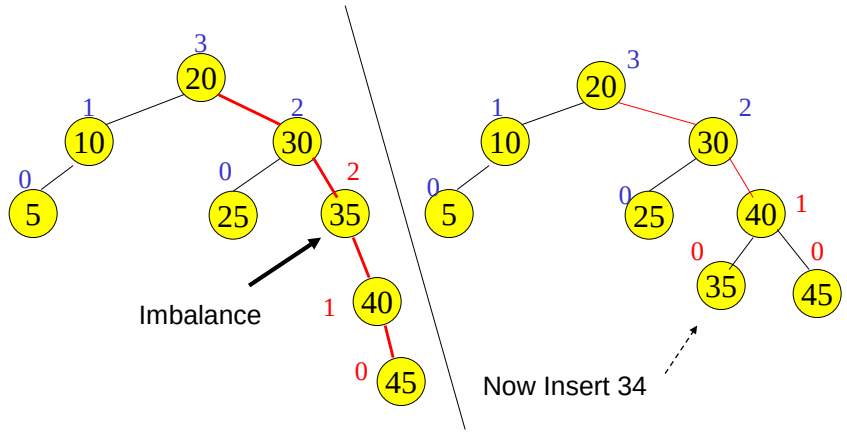
Example of Insertions in an AVL Tree



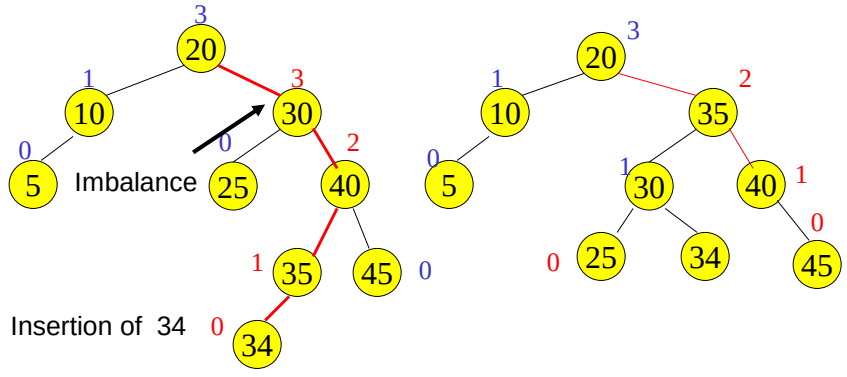


Now Insert 45

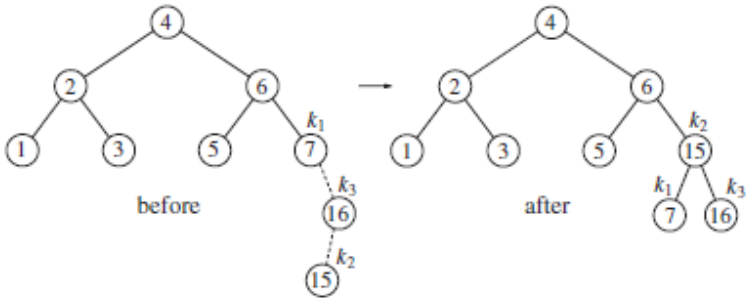
Single rotation (outside case)



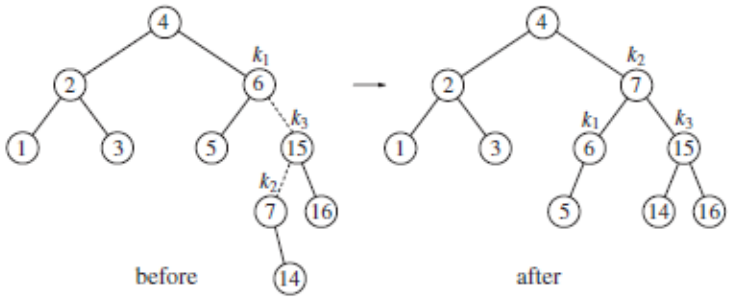
Double rotation (inside case)



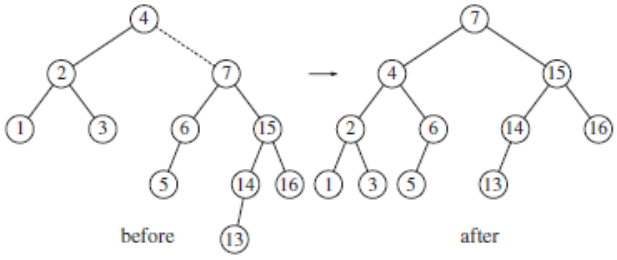
- Insert 15



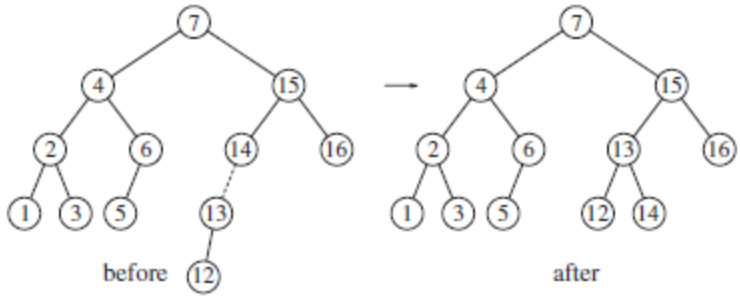
- Insert 14



- Insert 13



- Insert 12



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.


```
void remove( const Comparable & x, AvlNode * & t )
{
    if( t == nullptr )
        return;   // Item not found; do nothing

    if( x < t->element )
        remove( x, t->left );
    else if( t->element < x )
        remove( x, t->right );
    else if( t->left != nullptr && t->right != nullptr )  // Two children
    {
        t->element = findMin( t->right  )->element;
        remove( t->element, t->right );
    }
    else
    {
        AvlNode *oldNode = t;
        t = ( t->left != nullptr ) ? t->left : t->right;
        delete oldNode;
    }

     balance( t );
}
```

```
private AvlNode<AnyType> remove( AnyType x, AvlNode<AnyType> t )
{
    if( t == null )
        return t;    // Item not found; do nothing

    int compareResult = x.compareTo( t.element );

    if( compareResult < 0 )
        t.left = remove( x, t.left );
    else if( compareResult > 0 )
        t.right = remove( x, t.right );
    else if( t.left != null && t.right != null ) // Two children
    {
        t.element = findMin( t.right ).element;
        t.right = remove( t.element, t.right );
    }
    else
        t = ( t.left != null ) ? t.left : t.right;
    return balance( t );
}
```


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).