# Balanced Trees

*15-121*

*Data Structures*

Ananda Gunawardena

# A Good Tree

In a "good" BST we have

depth of T = O( log n )

n = num of nodes in the tree

- Theorem: If the tree is constructed from *n* inputs given in random order, then we can expect the depth of the tree to be $\log_2 n$. (no proof given)

- But if the input is already (nearly, reverse,…) sorted we are in trouble.

# Forcing good behavior

- We can show that for any n inputs, there always is a BST containing these elements of logarithmic depth.
- But if we just insert the standard way, we may build a very unbalanced, deep tree.
- Can we somehow force the tree to remain shallow?
    - At low cost?

# Balanced Trees

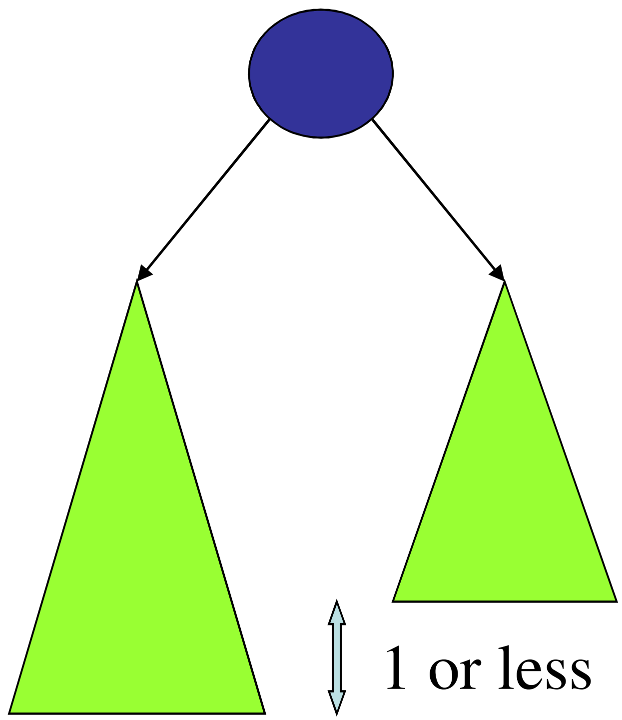
- A balanced tree is where equal (“almost”) number of nodes exists in left and right sub trees of each node
  - However in practice this is hard to enforce
- We can expect the trees to remain shallow by “randomizing” a data set before inserting to a tree
  - Need O(n) operations to randomize the data set
- A relaxed balanced condition can be used in building a “good” tree
  - AVL trees

# AVL Trees

- AVL trees requires a relaxed balanced condition
- Define “balanced factor” of a node to be the absolute difference in heights between left and right sub trees
- AVL tree is defined as a tree such that, for each node, balanced factor ≤ 1

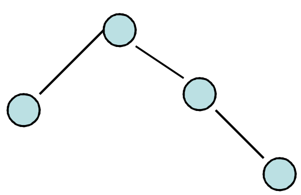
# AVL-Trees

G. M. Adelson-Velskii and E. M. Landis, 1962

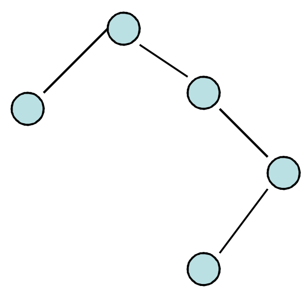


# AVL-Trees

An AVL-tree is a BST with the property that at every node the difference in the depth of the left and right subtree is at most 1.



OK



not OK

# Bad News

Insertion in an AVL-tree is more complicated: inserting a new element as a leaf may break the balance of the tree.

But we can't just place the new element somewhere else, we have to maintain the BST property.

**Solution**: insert in standard place, but then rebalance the tree.

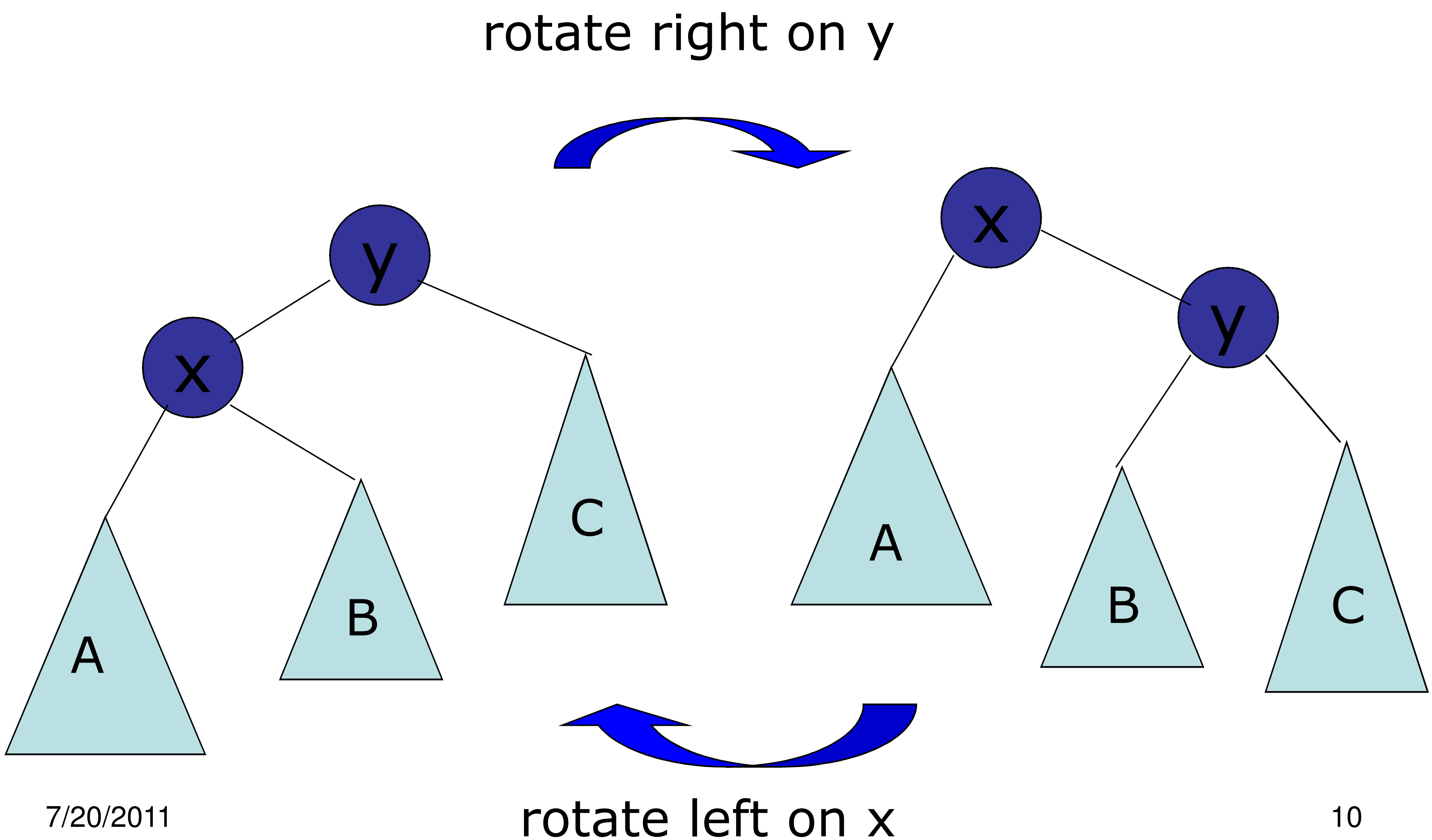
# But How?

The magic concept is rotation. A rotation rearranges a BST, but preserve the BST property.

Can be done out in O(1) steps.

To fix up an AVL tree, we have to perform several rotations, but only along a branch: total damage is O(log #nodes).

# But How?



# So?

Rotation does not change the flattening, so we still have a BST.

But the depth of the leaves change by -1 in A, stay the same in B, and change by +1 in C.

# Well, not quite

Unfortunately, there are other cases to consider, depending on where exactly the balance condition is violated.

To fix some of them requires a double rotation.

There is a nice demo at:

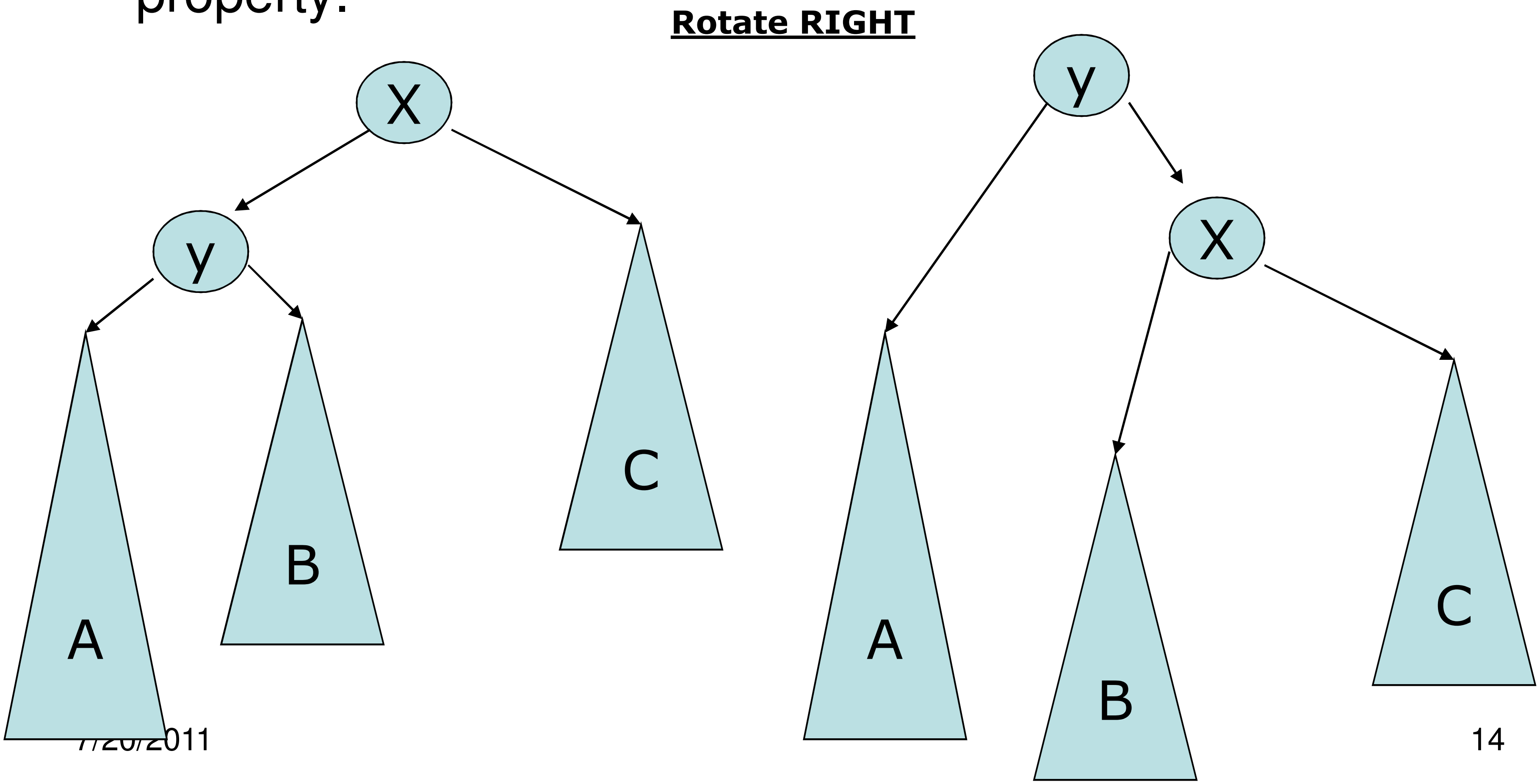
http://www.site.uottawa.ca/~stan/csi2514/applets/avl/BT.html

# Tree Balance Rotations

- Note that after an insertion only the nodes in the path from root to the node have their balance information altered
- Find the node that violates the AVL condition and rebalance the tree.
- Assume that X is the node that has violated the AVL condition
  - I.e The left and right sub trees of X is differ by 2 in height.

# Four Cases – Case I

Case I – The left subtree of the left child of X violates the property.



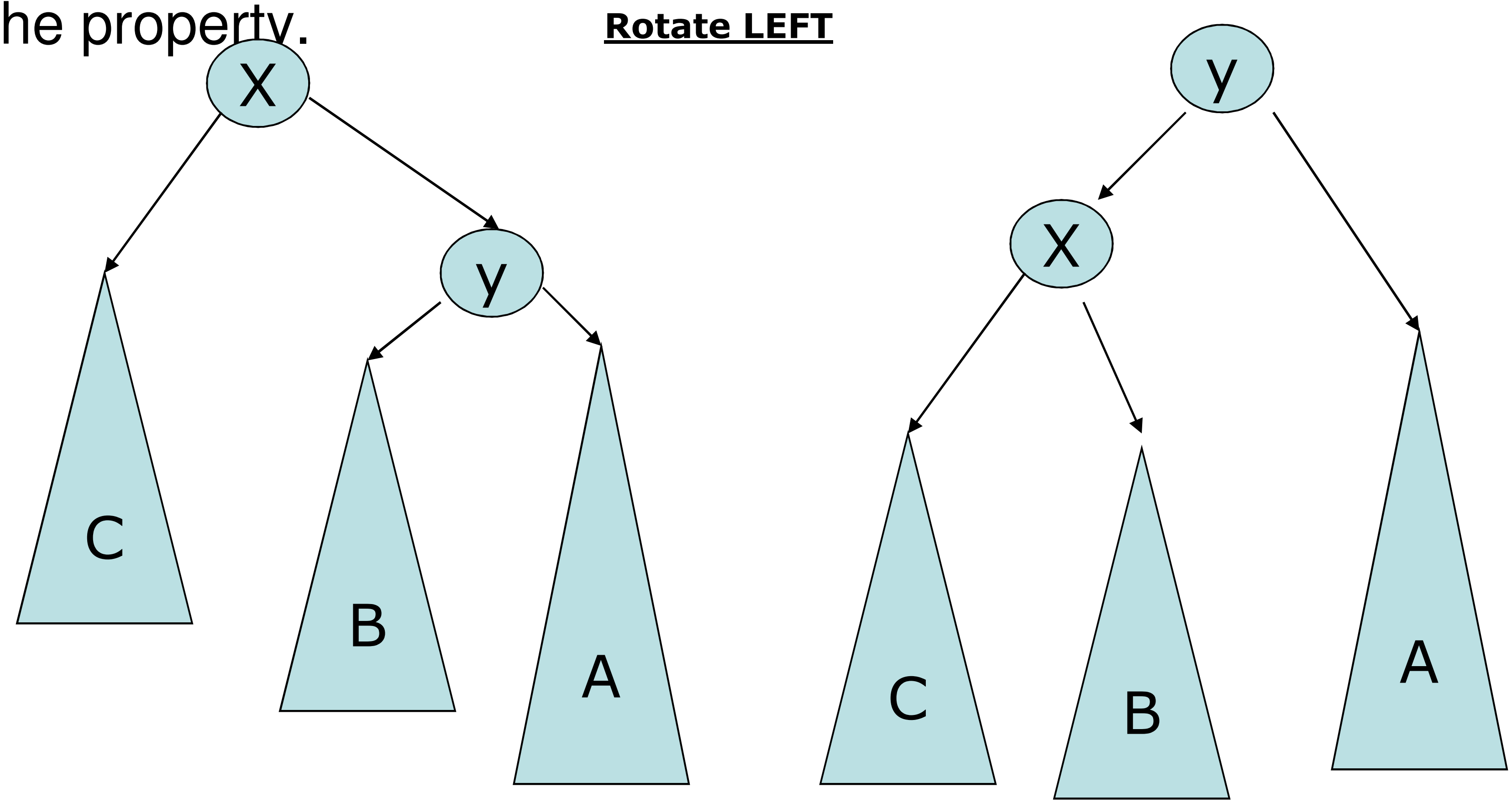
# Code for Case 1

/** * Rotate binary tree node with left child. * For AVL trees, this is a single rotation for case 1. */

```
static <AnyType> BinaryNode<AnyType> rotateWithLeftChild(
  BinaryNode<AnyType> k2 ) {
      BinaryNode<AnyType> k1 = k2.left;
      k2.left = k1.right;
      k1.right = k2; return k1;
  }
```

# Four Cases – Case 2

Case 2 – The right subtree of the right child of X violates the property.

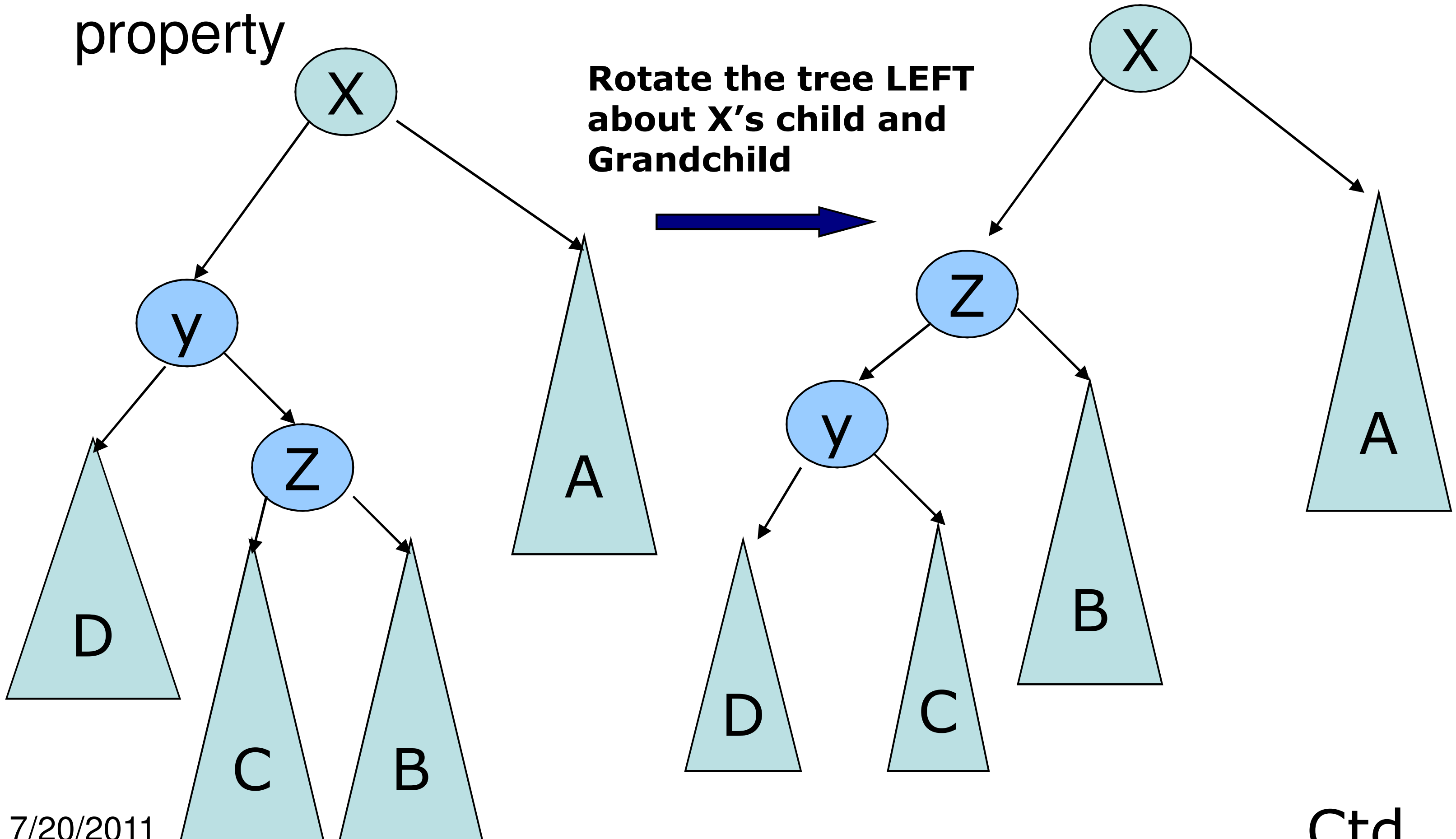


# Code for Case 2

```
/** * Rotate binary tree node with right child. * For
  AVL trees, this is a single rotation for case 4. */
 static <AnyType> BinaryNode<AnyType>
  rotateWithRightChild( BinaryNode<AnyType> k1
  )
 { BinaryNode<AnyType>
    k2 = k1.right; k1.right = k2.left;
    k2.left = k1; return k2;
}
```

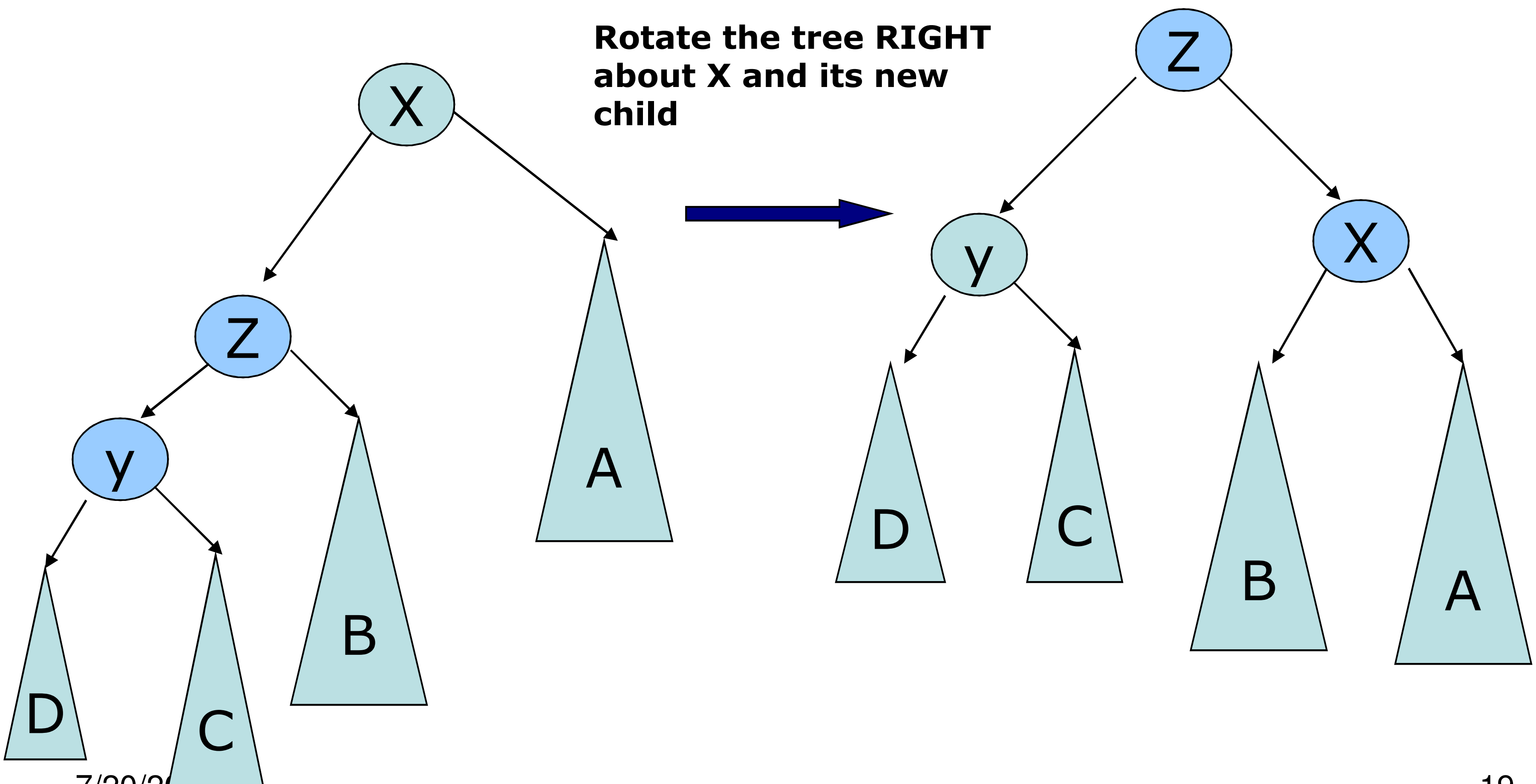
# Four Cases – Case 3

Case 3 – The right subtree of the left child of X violates the property




Ctd..

# Four Cases – Case 3 ctd..

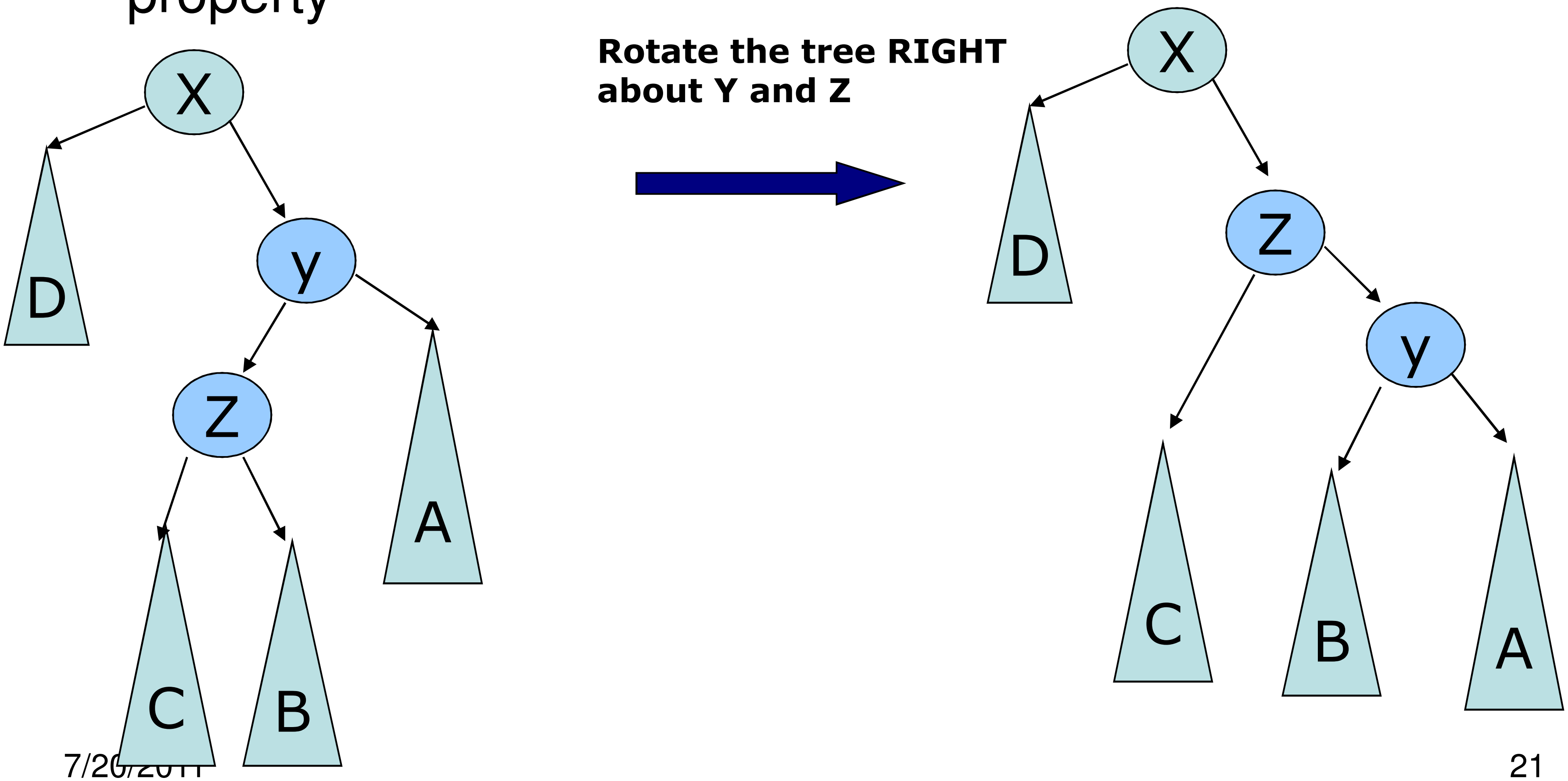


# Code for Case 3

```
/** * Double rotate binary tree node: first right child
  * with its left child; then node k1 with new right
  child. * For AVL trees, this is a double rotation
  for case 3. */
 static <AnyType> BinaryNode<AnyType>
  doubleRotateWithRightChild(
      BinaryNode<AnyType> k1 )
 {
   k1.right = rotateWithLeftChild( k1.right ); return
   rotateWithRightChild( k1 );
 }
```

# Four Cases – Case 4

Case 4 – The left subtree of the right child of X violates the property

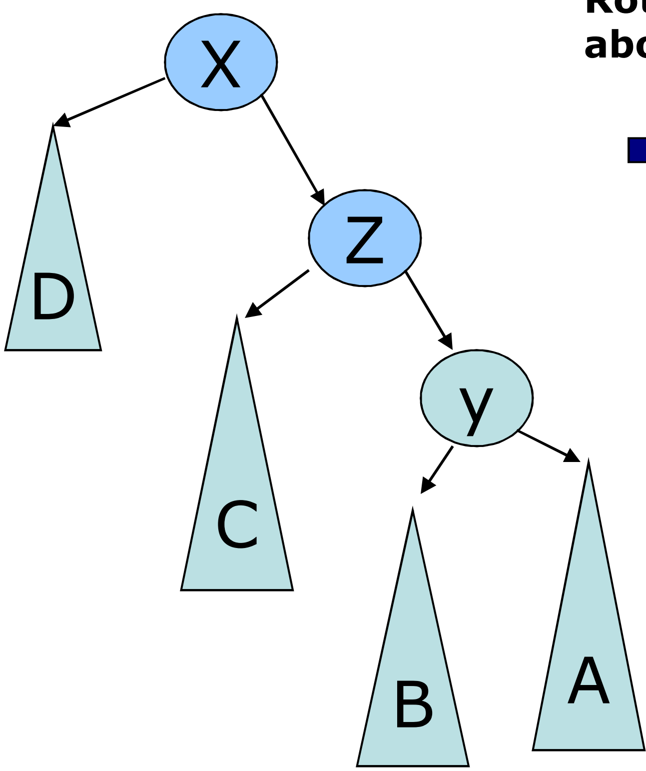




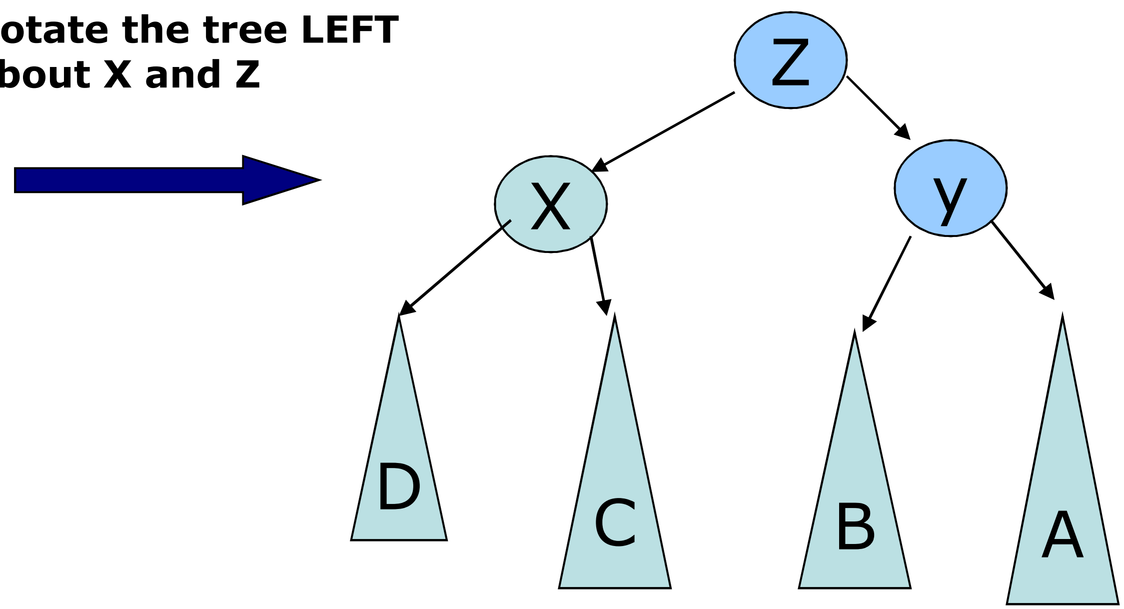



Ctd..

# Four Cases – Case 4 ctd..




**Rotate the tree LEFT about X and Z**



# Code for Case 4

```
/** 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.
*/
  static <AnyType> BinaryNode<AnyType>
       doubleRotateWithLeftChild( BinaryNode<AnyType>
   k3 )
   { k3.left = rotateWithRightChild( k3.left );
      return rotateWithLeftChild( k3 );
    }
```

# AVL Tree Examples

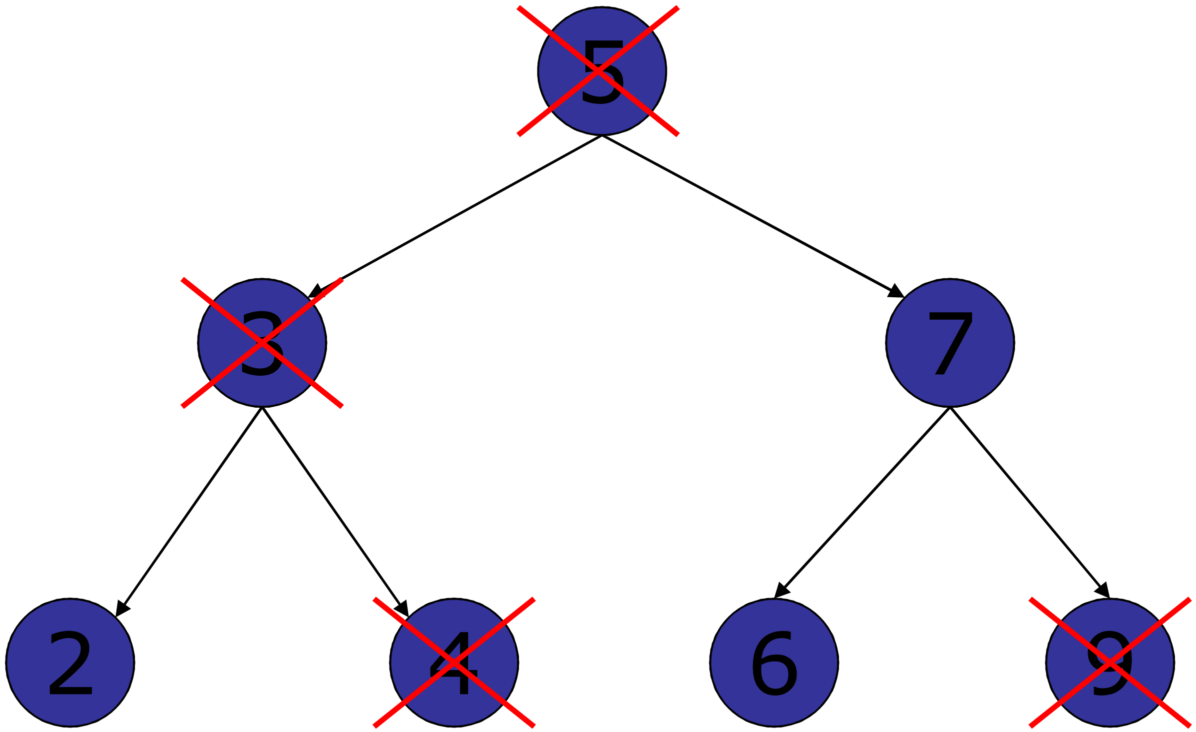
- Insert 12, 8, 7, 14, 18, 10, 20 with AVL rotations

# Implementing AVL trees

- The main complications are insertion and deletion of nodes.

- For deletion:
  - Don't actually delete nodes.
  - Just mark nodes as deleted.
  - Called *lazy deletion.*

# Lazy deletion




On average, deleting even half of the nodes by marking leads to depth penalty of only 1.

# AVL Summary

- Insert a new node in left or right subtree.
- Test the height information along the path of the insertion. If not changed we are done
- Otherwise do a single or double rotation based on the four cases
- Question: What is the best thing to do about height info?