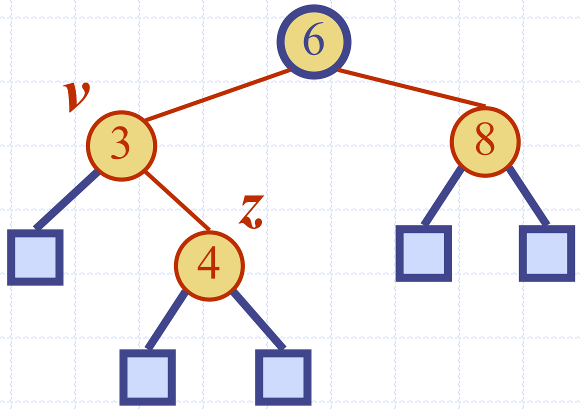
# AVL Trees



# Performance of BST

| Operation | Running Time |
|---|---|
| k in T | $O(h)$ |
| T[k], T[k] = v | $O(h)$ |
| T.delete(p), del T[k] | $O(h)$ |
| T.find_position(k) | $O(h)$ |
| T.first(), T.last(), T.find_min(), T.find_max() | $O(h)$ |
| T.before(p), T.after(p) | $O(h)$ |
| T.find_lt(k), T.find_le(k), T.find_gt(k), T.find_ge(k) | $O(h)$ |
| T.find_range(start, stop) | $O(s+h)$ |
| iter(T), reversed(T) | $O(n)$ |

- Space usage is $O(n)$, where $n$ is the number of items stored in the map.

# Binary Search Tree - Best Time

- All BST operations are O(h), where h is tree height.
- In the best case, T has height

$$h = \mathrm{Ceil}(\log(n+1)) - 1$$

- So, best case running time of BST operations is O(log n)

# Binary Search Tree - Worst Time

- Worst case running time is O(N)
- Insertion of elements in monotonous order

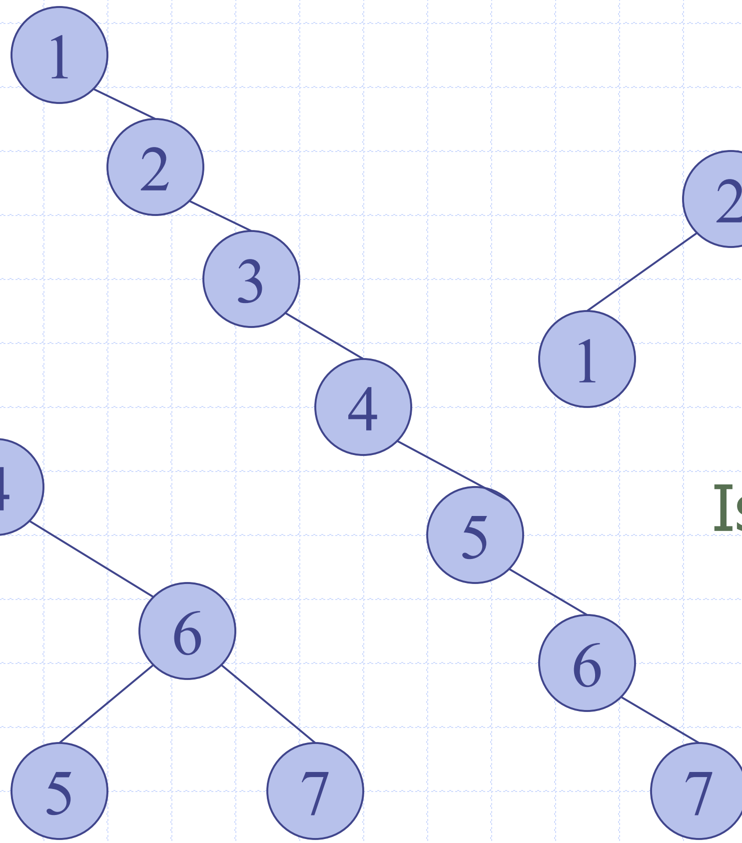
    Insert: 2, 4, 6, 8, 10, 12 into an empty BST

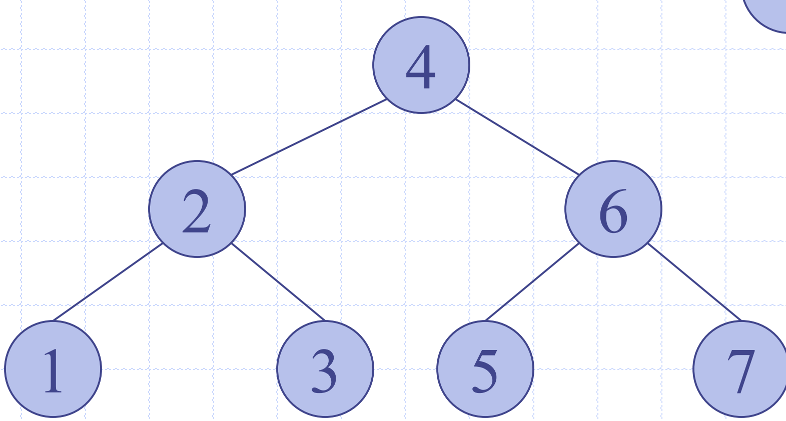
    - Problem: Lack of balance

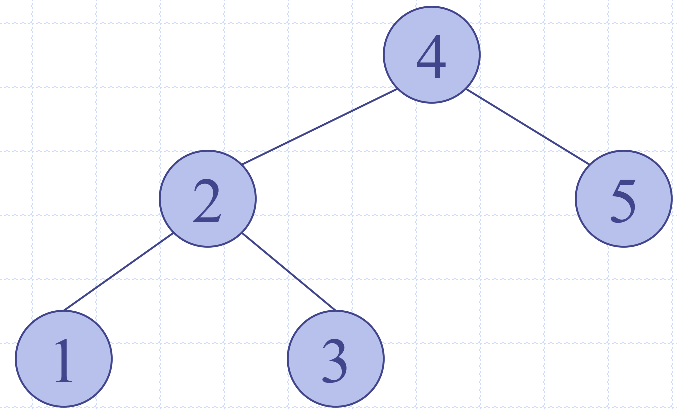
    compare depths of left and right subtree

    - Unbalanced degenerate tree

# Balanced vs Unbalanced BST




Is this “balanced”?

# Approaches to balancing trees

- Don't balance
  - May end up with some nodes very deep
- Strict balance
  - The tree must always be balanced perfectly
- Pretty good balance
  - Only allow a little out of balance
- Adjust on access
  - *Self-adjusting*

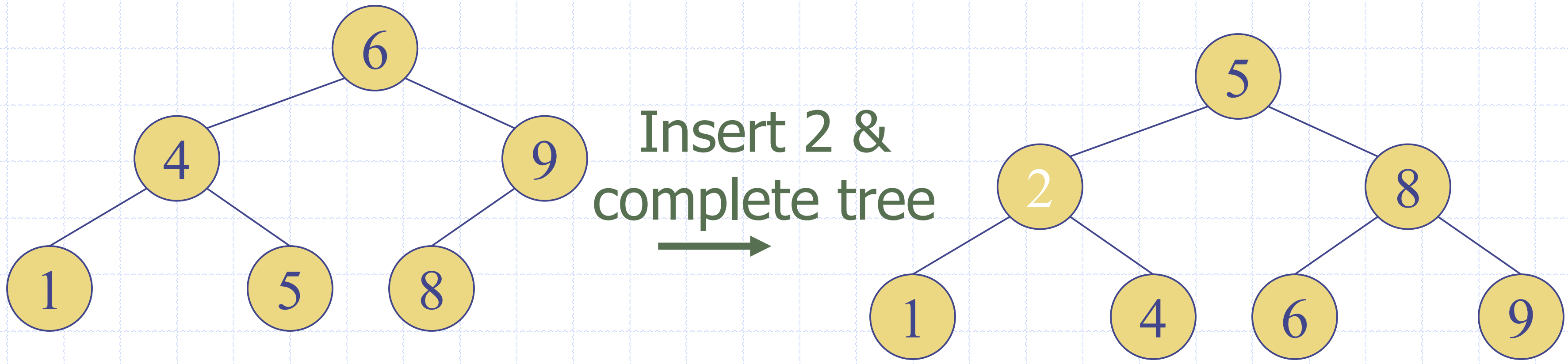
# Balancing Binary Search Trees

Many algorithms exist for keeping BSTs balanced

- Adelson-Velskii and Landis (AVL) trees (height-balanced trees)
- Splay trees and other self-adjusting trees
- Red-Black trees
- B-trees and other multiway search trees

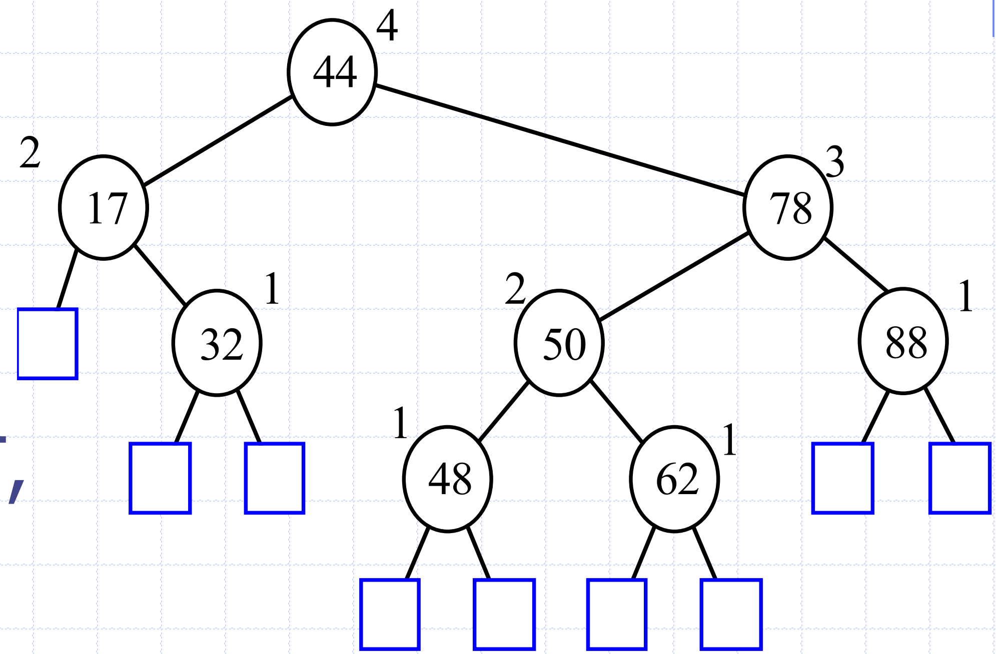
# Perfect Balance

- 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 Tree Definition

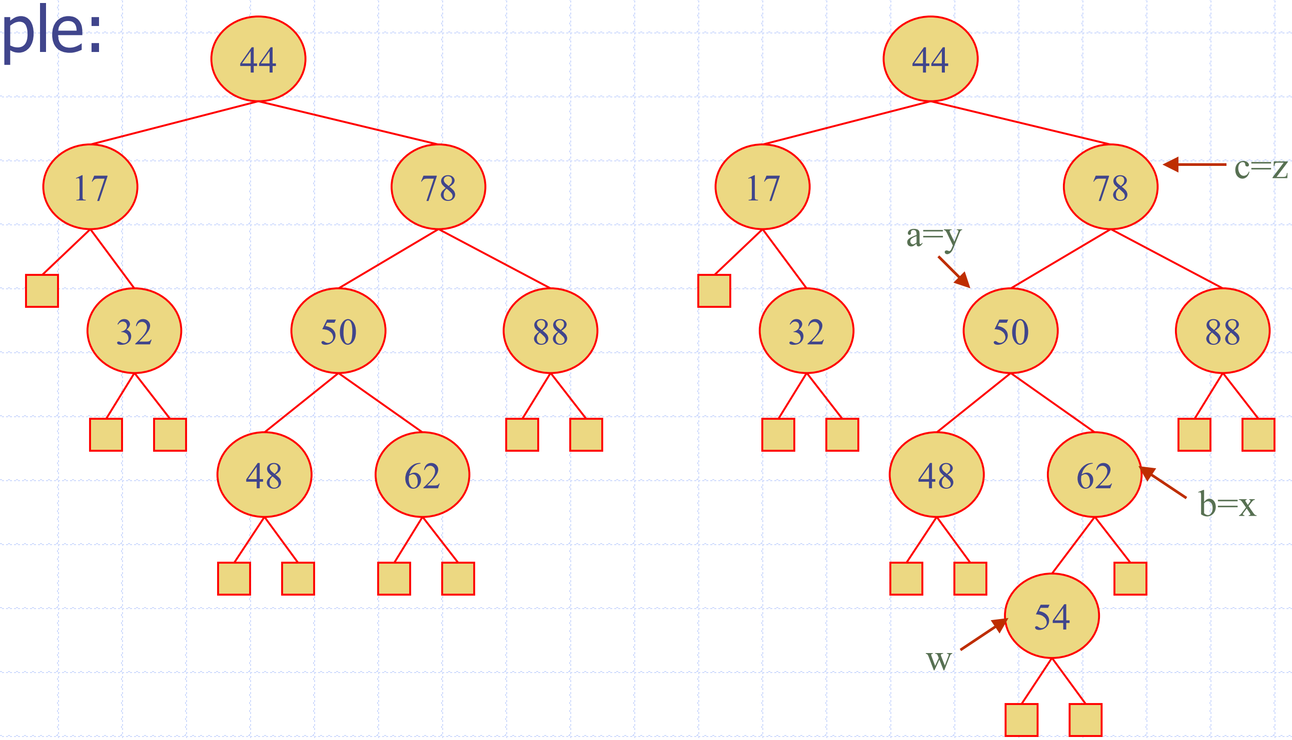
- AVL trees are balanced
- An AVL Tree is a **binary search tree** such that for every internal node v of T, the **heights of the children of v can differ by at most 1**




An example of an AVL tree where the heights are shown next to the nodes:

# Insertion

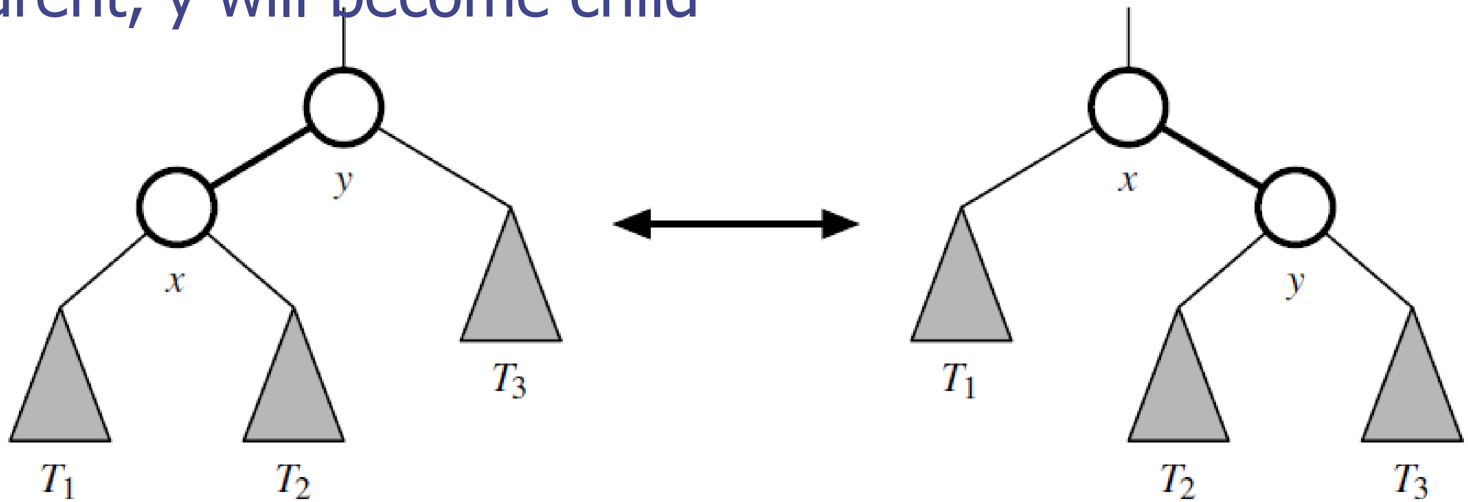
- Insertion is as in a binary search tree
- Always done by expanding an external node.
- Example:




before insertion after insertion

# Rotation

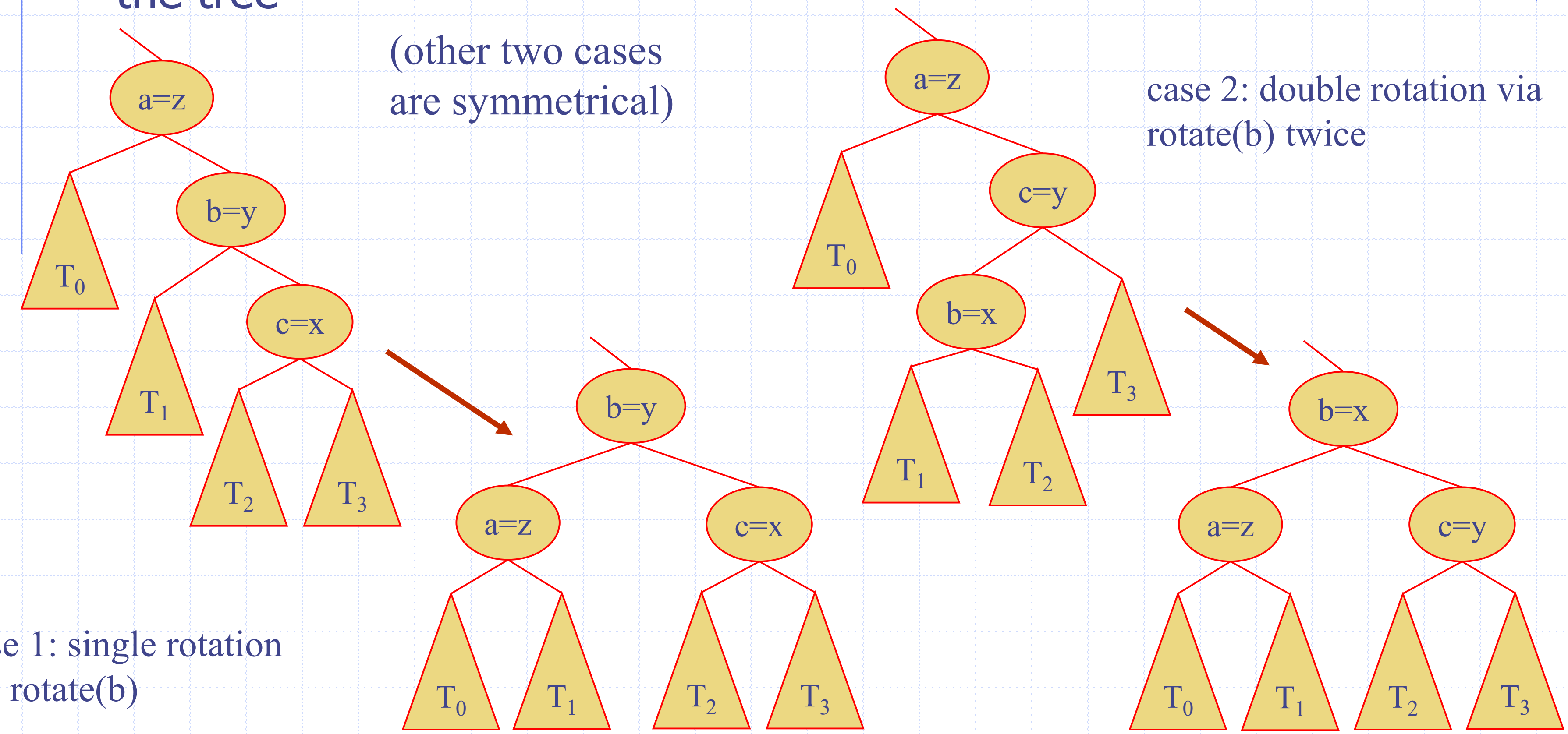
Rotate(x): x will become parent, y will become child




Rotate(y): y will become parent, x will become child

# Trinode Restructuring

- let (*a*,*b*,*c*) be an inorder listing of *x*, *y*, *z*
- perform the rotations needed to make *b* the topmost node of the tree



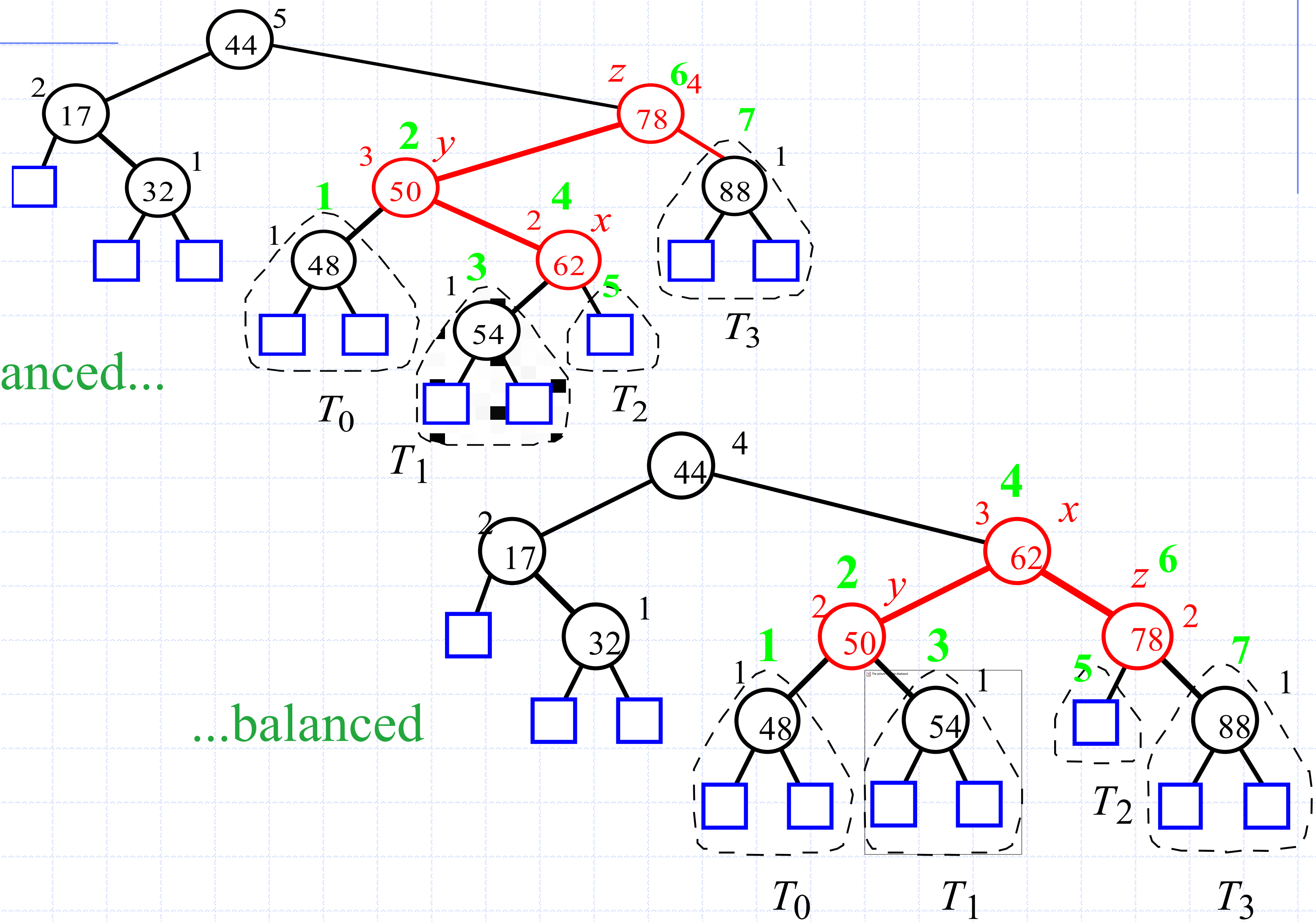

(other two cases are symmetrical)

case 1: single rotation via rotate(b)

case 2: double rotation via rotate(b) twice

# Insertion Example, continued

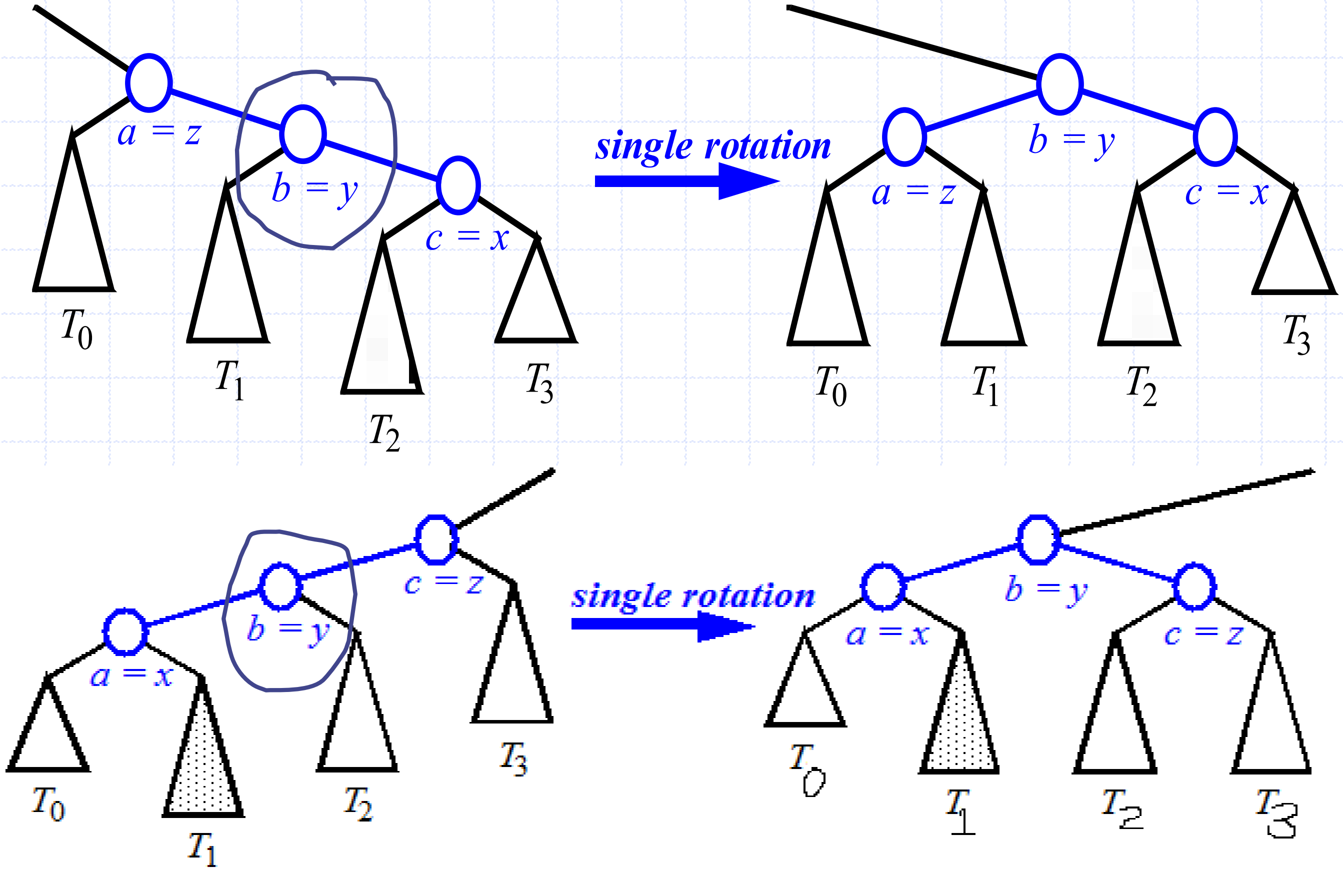
unbalanced...




...balanced

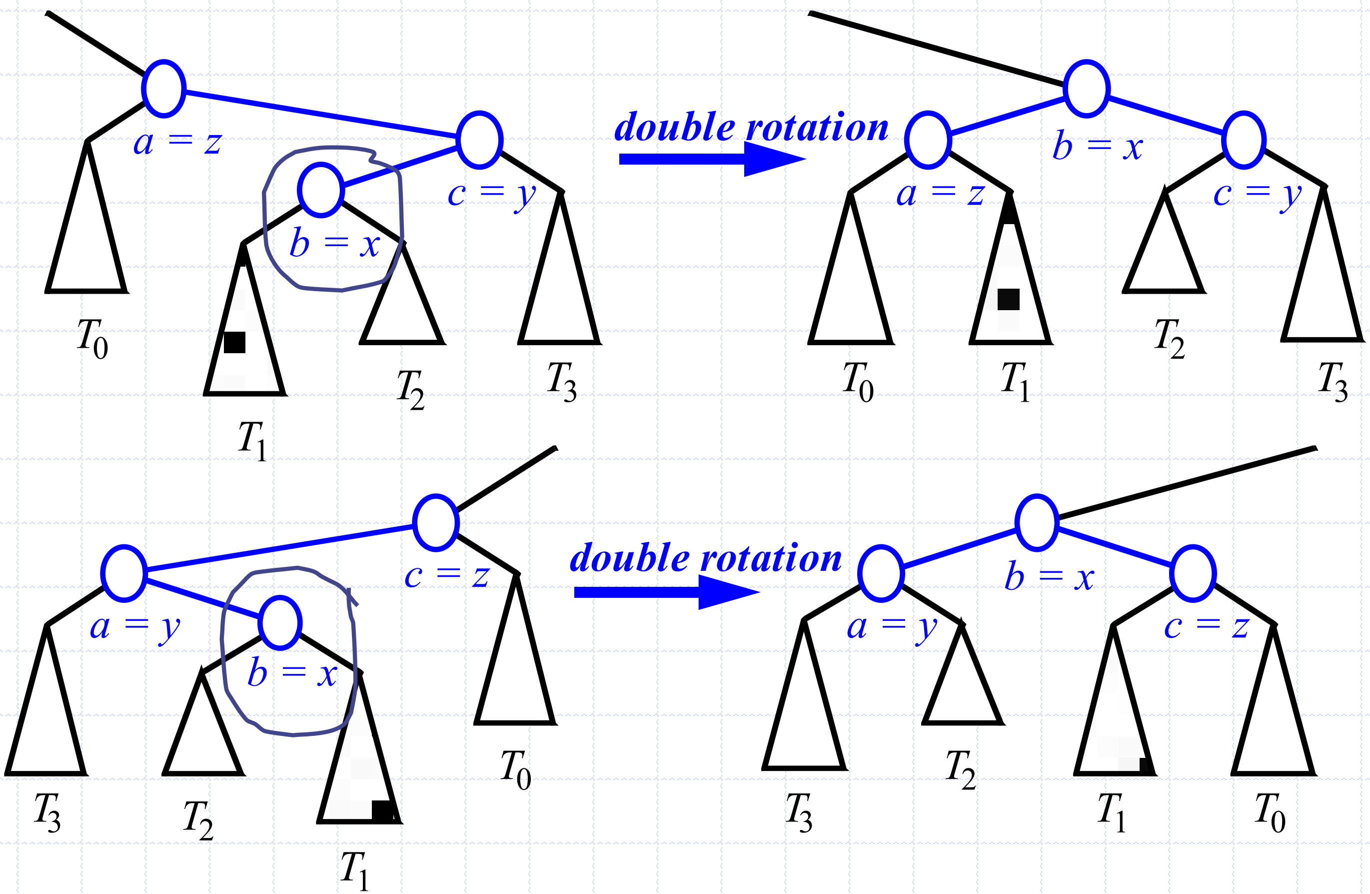
# Restructuring (as Single Rotations)

- Single Rotations: a < b < c => rotate (b)



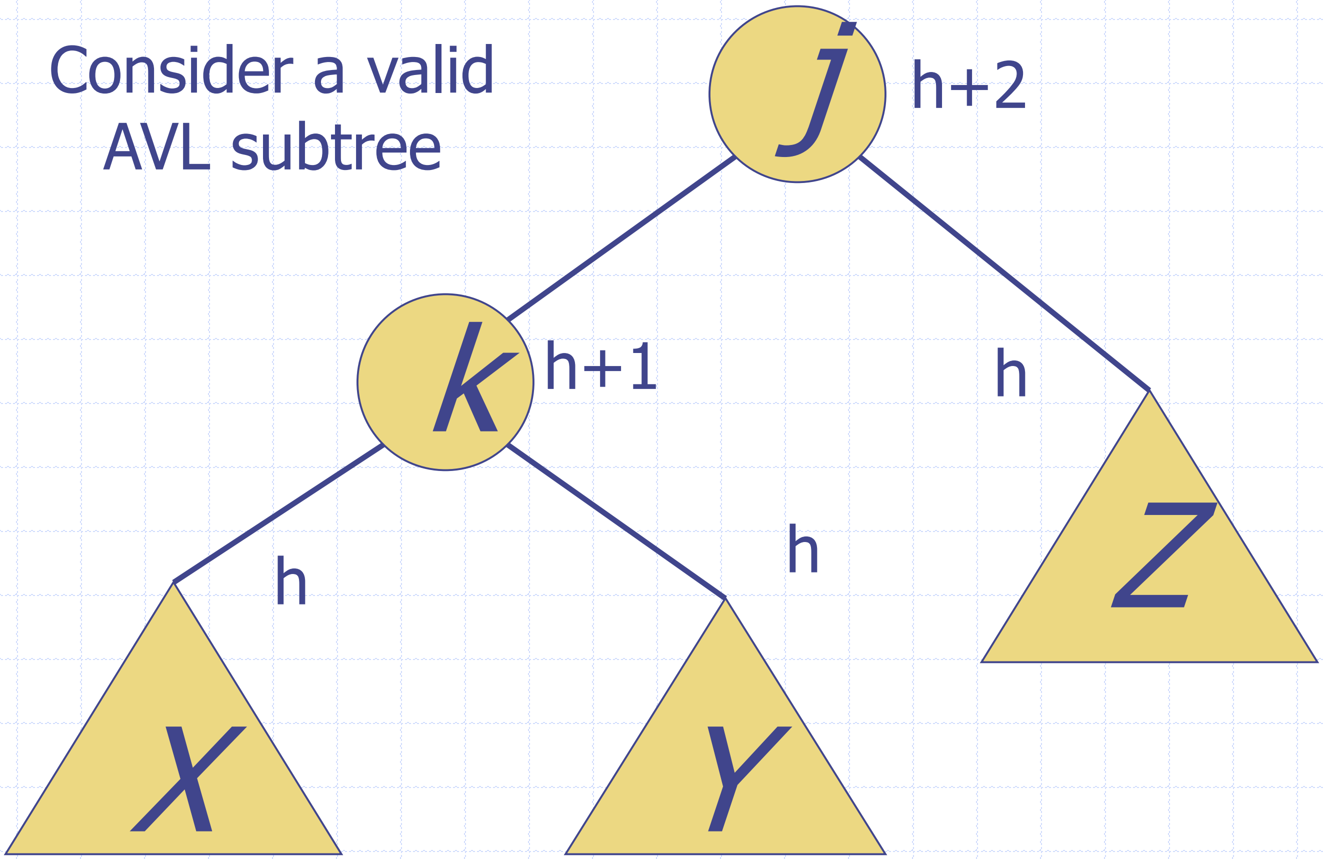
# Restructuring (as Double Rotations)

- double rotations: a < b < c => rotate(b) twice

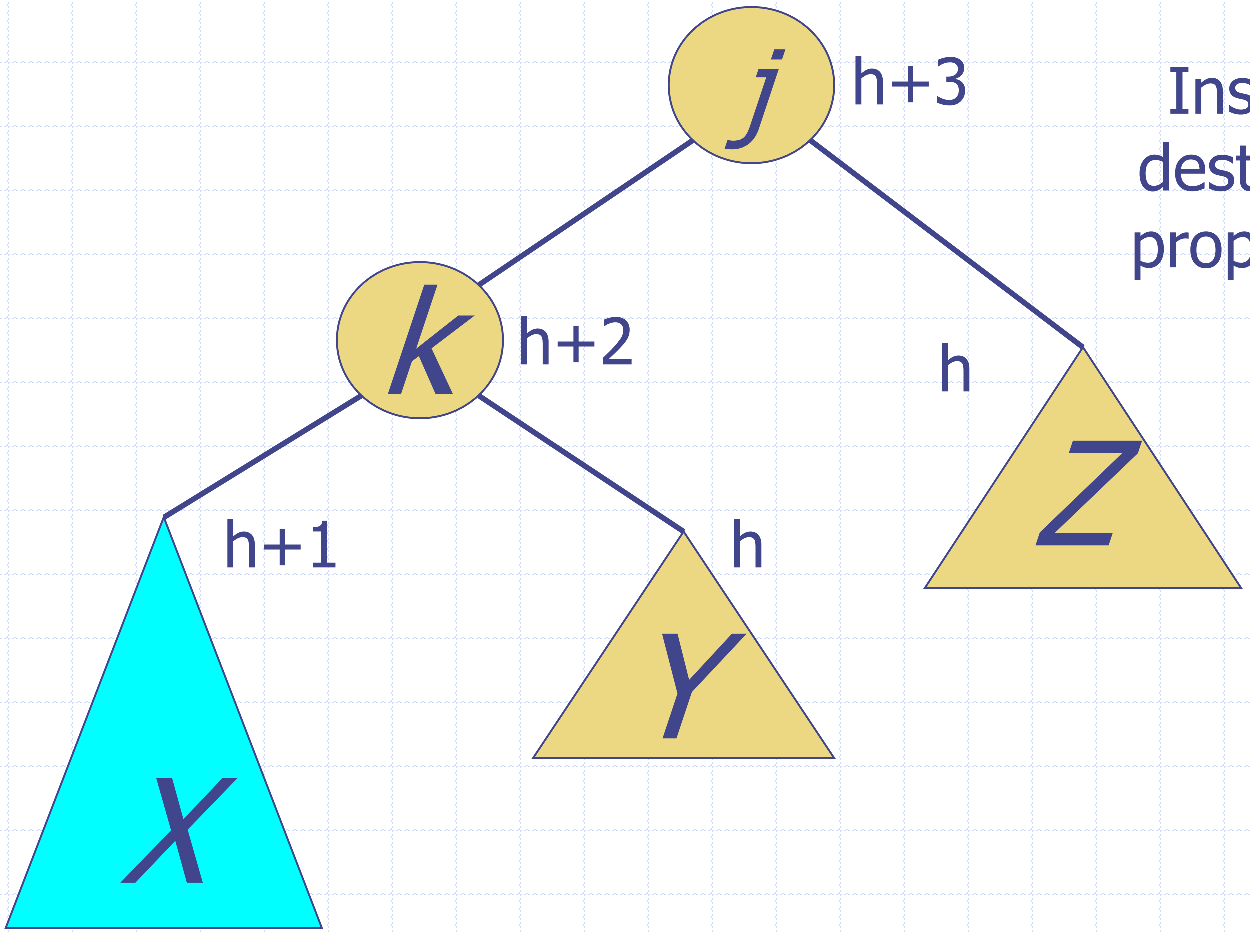


# AVL Insertion: Outside Case

Consider a valid
AVL subtree

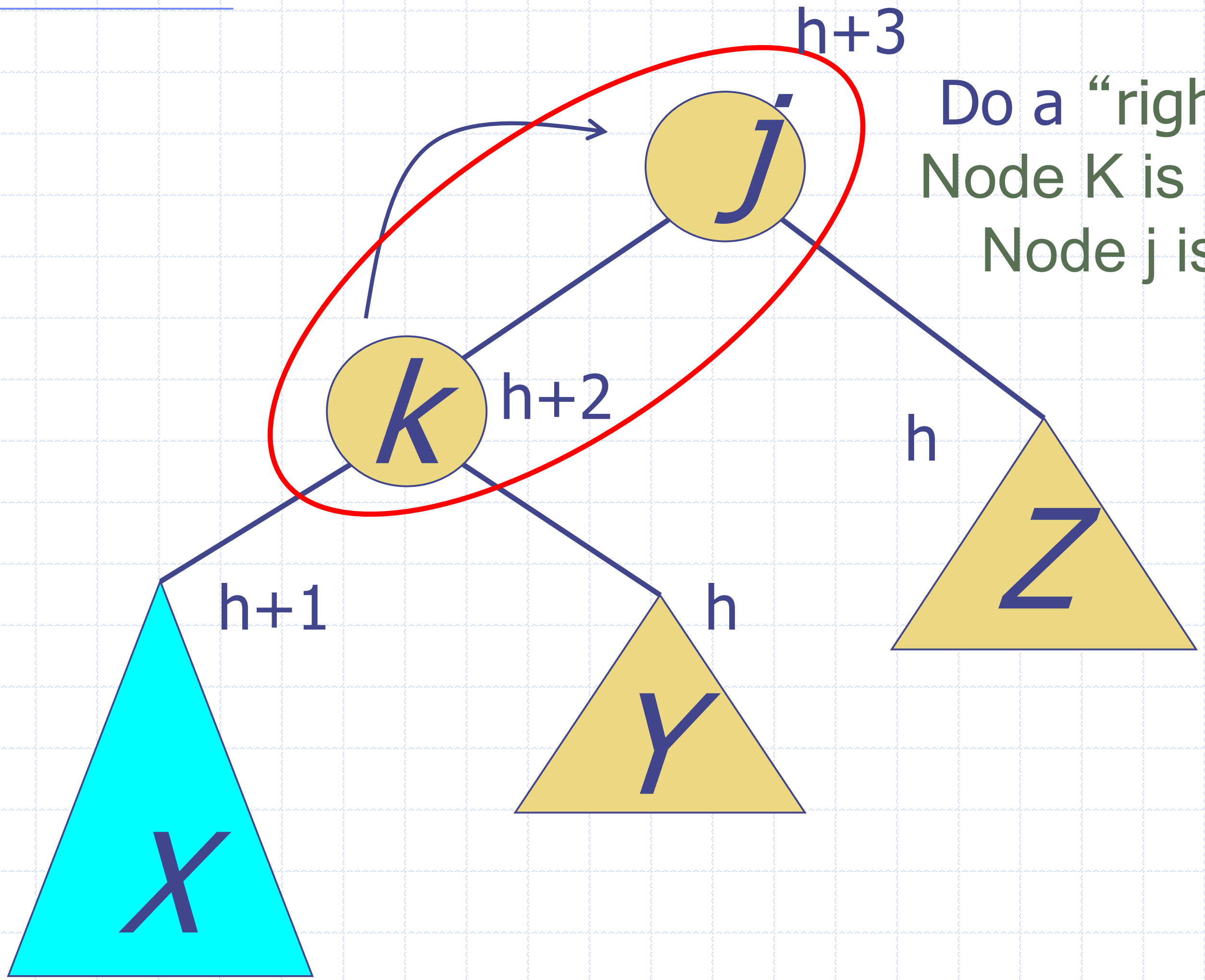


# AVL Insertion: Outside Case



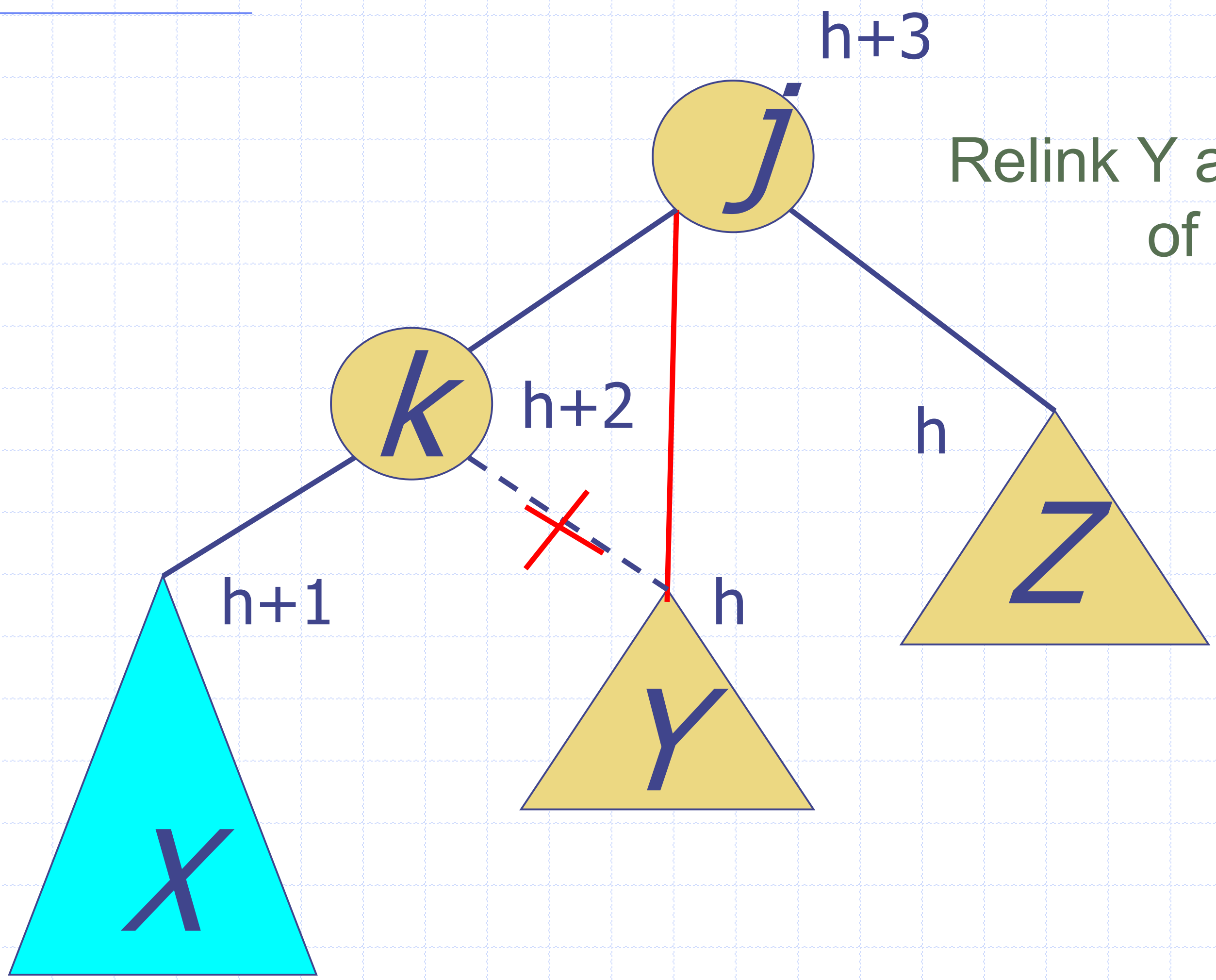

Inserting into X destroys the AVL property at node j

# AVL Insertion: Outside Case



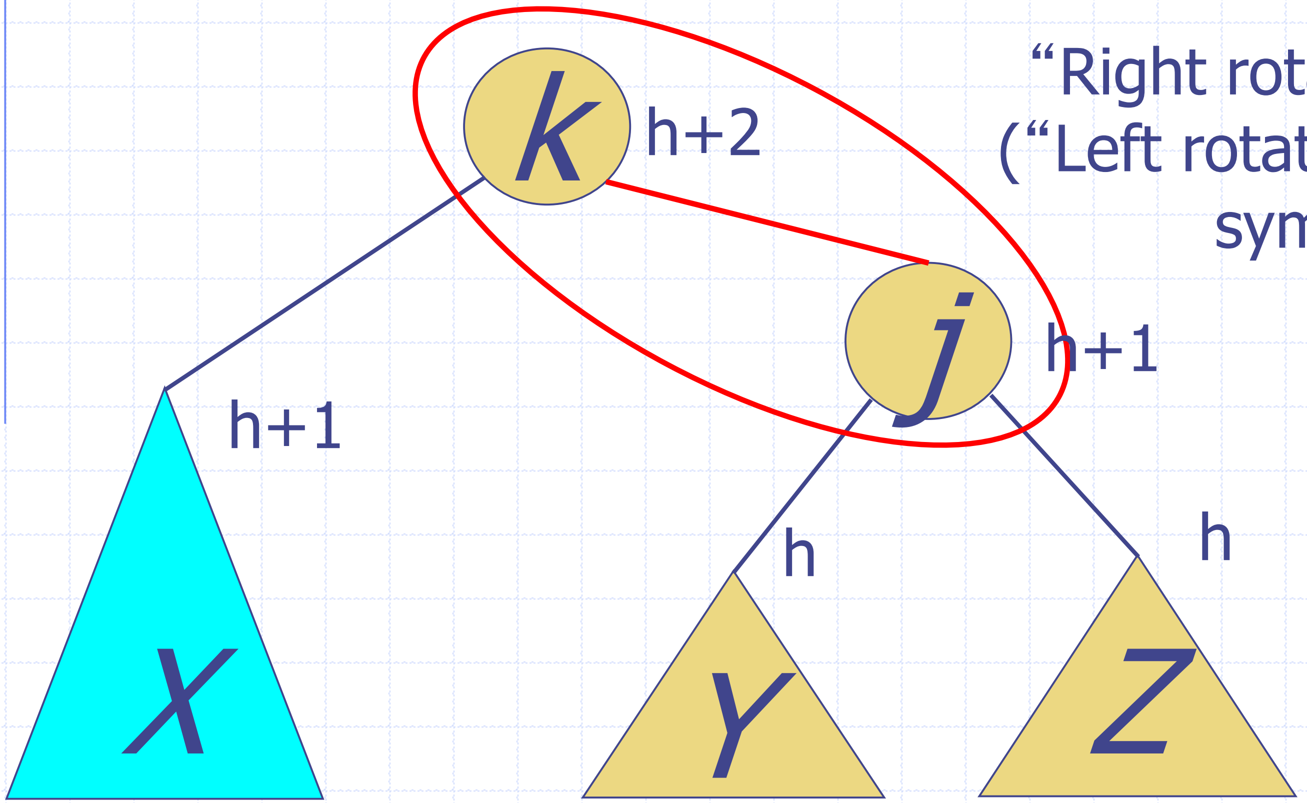

Do a "right rotation":
Node K is promoted &
Node j is demoted

# Single Right Rotation




Relink Y as left subtree of node j

# Outside Case Completed

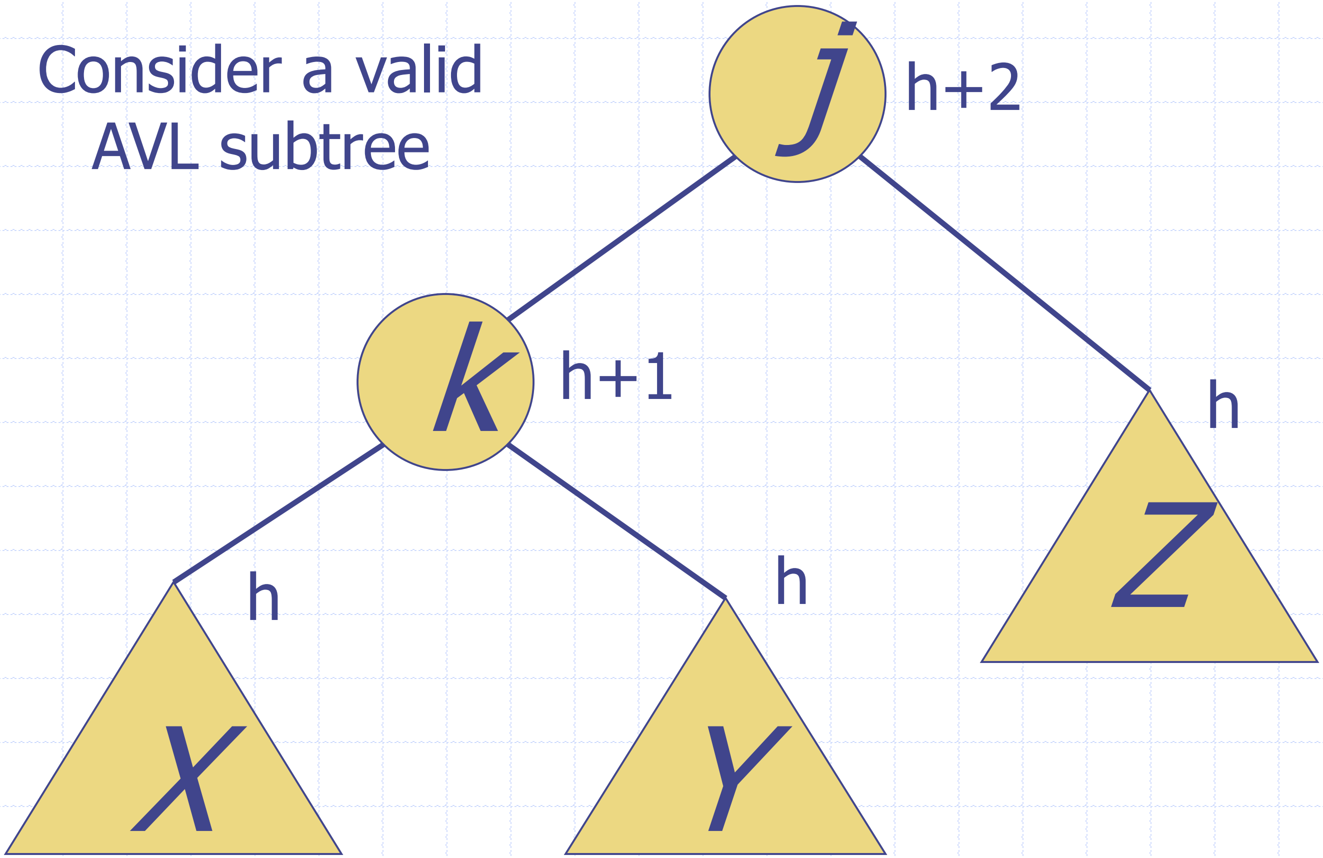



“Right rotation” done!
(“Left rotation” is mirror symmetric)

*AVL property has been restored!*

# AVL Insertion: Inside Case

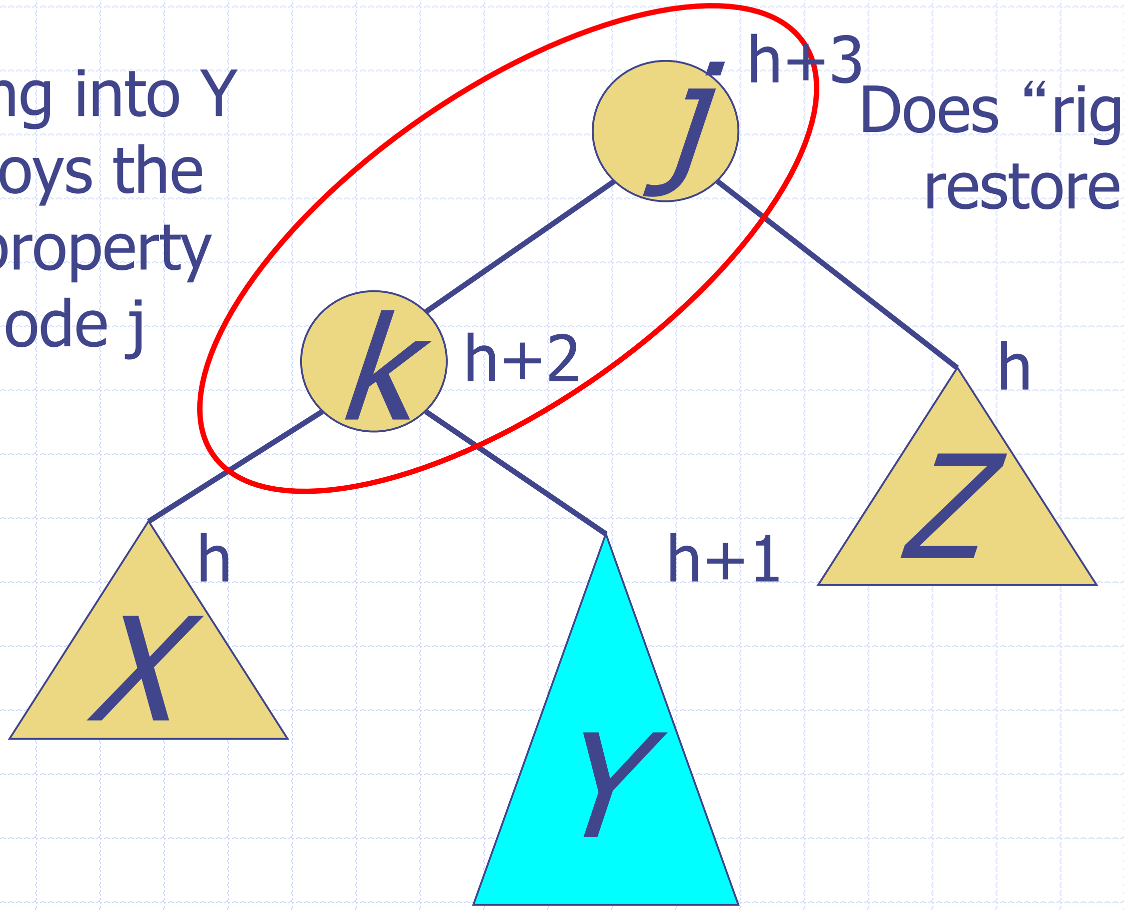
Consider a valid
AVL subtree



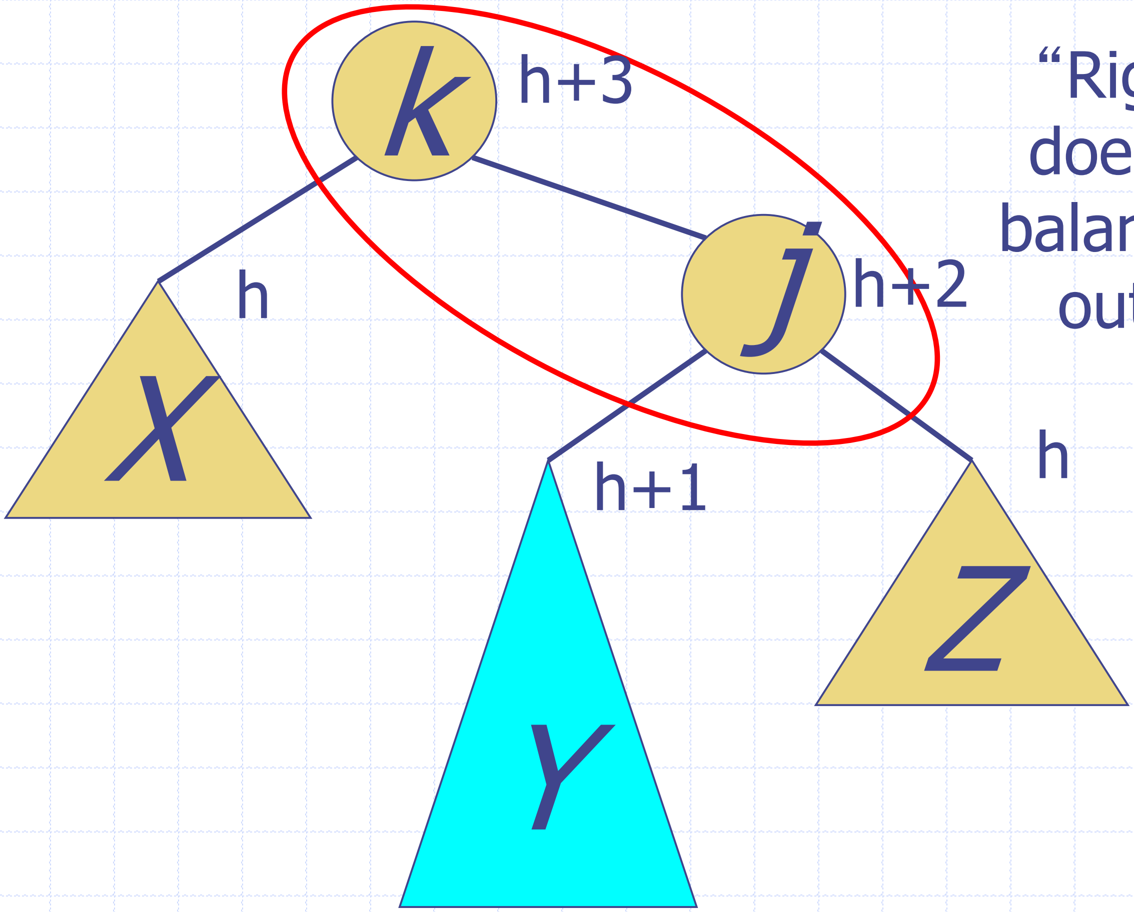
# AVL Insertion: Inside Case

Inserting into Y destroys the AVL property at node j

Does "right rotation" restore balance?



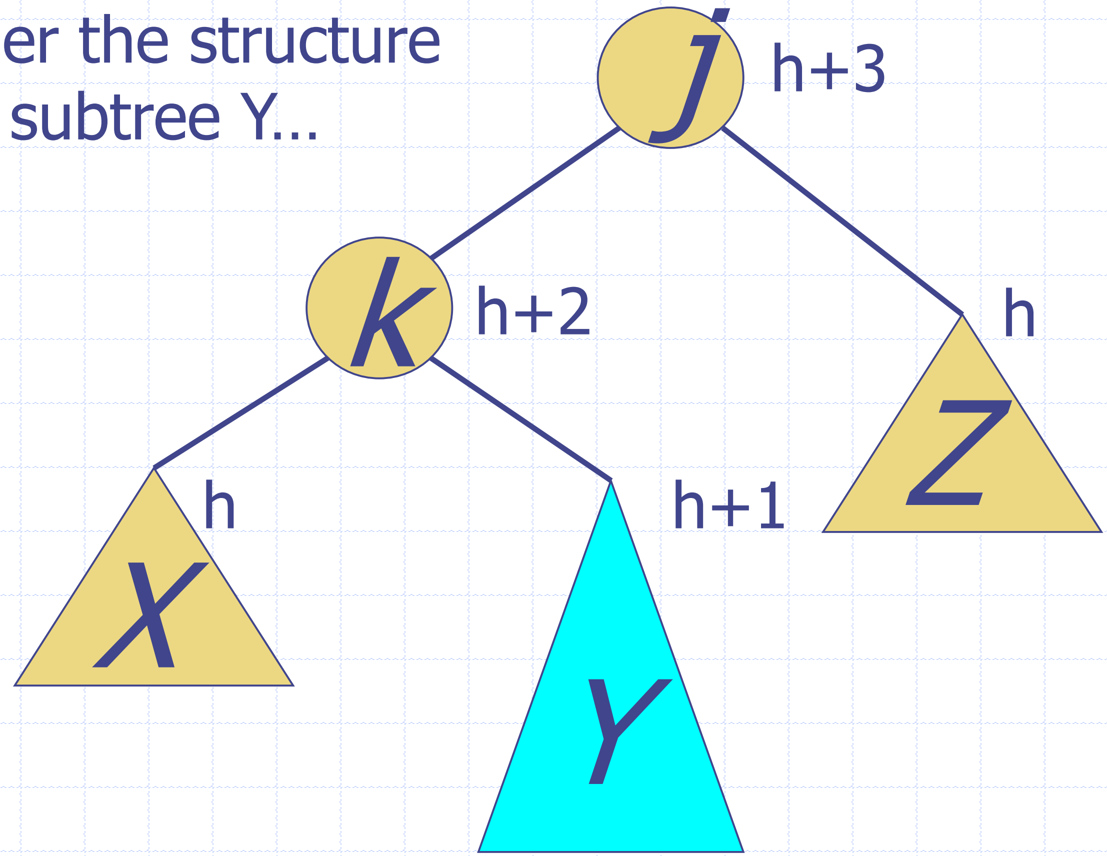
# AVL Insertion: Inside Case




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

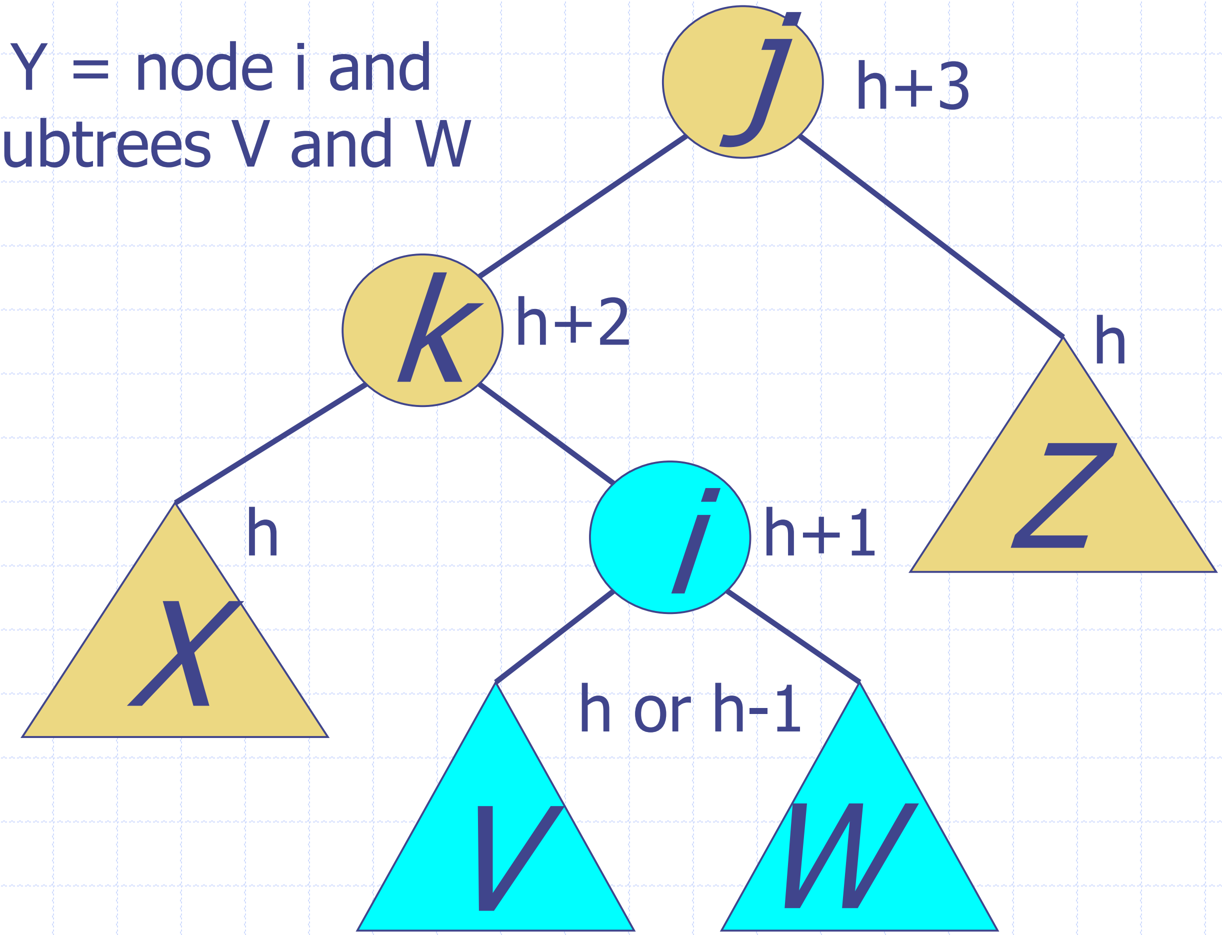
# AVL Insertion: Inside Case

Consider the structure
of subtree Y…

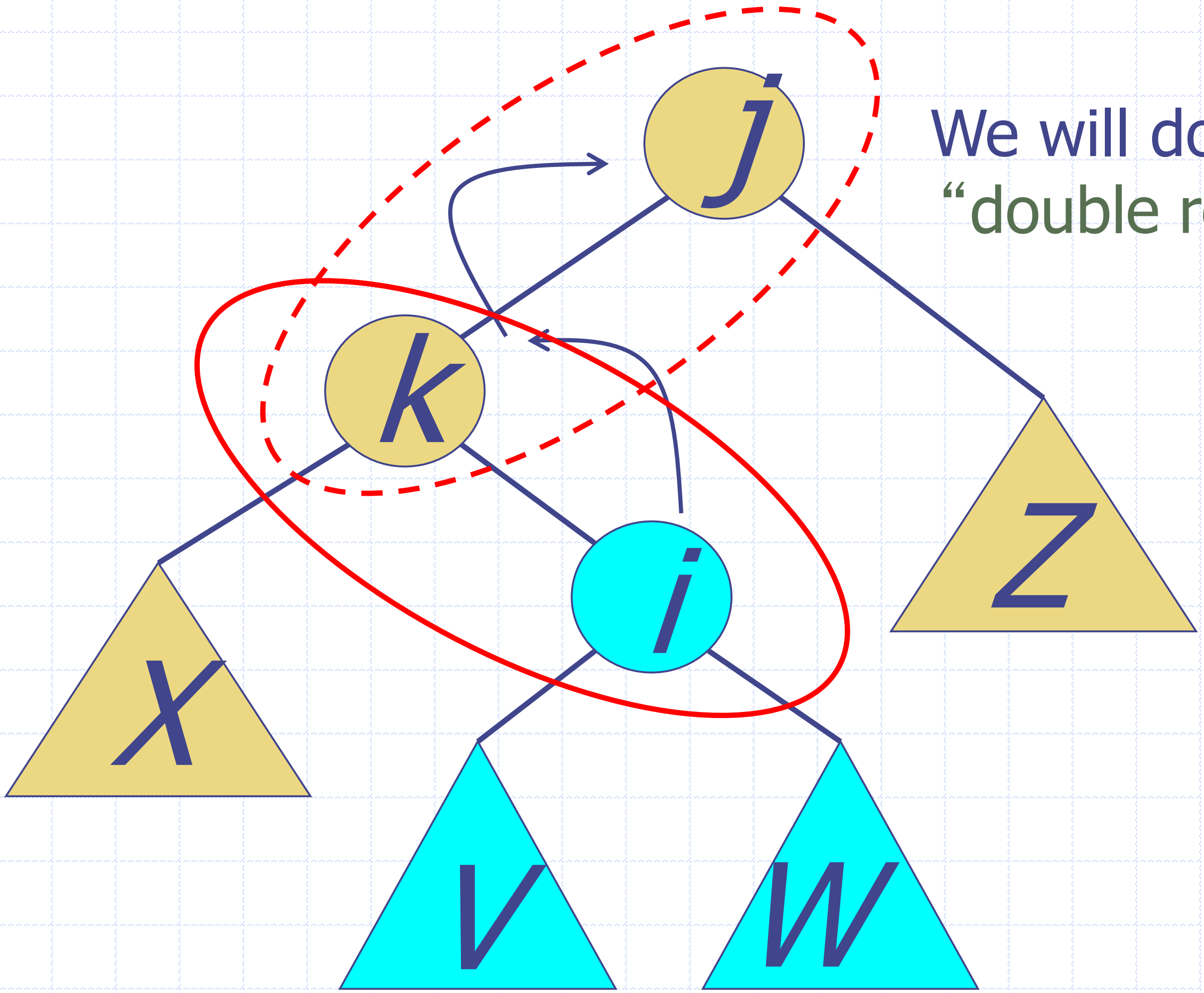


# AVL Insertion: Inside Case

Y = node i and
subtrees V and W

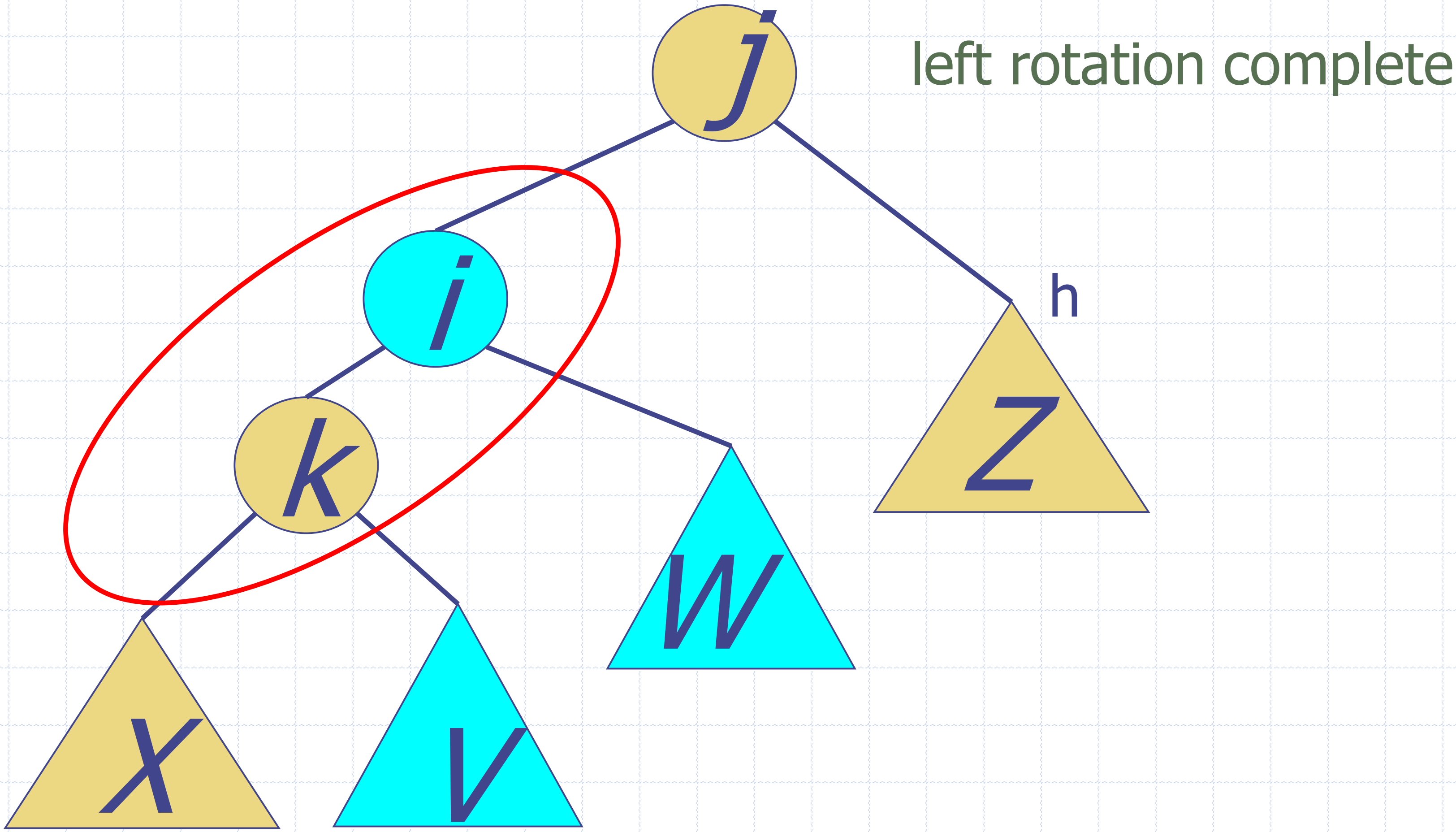


# AVL Insertion: Inside Case

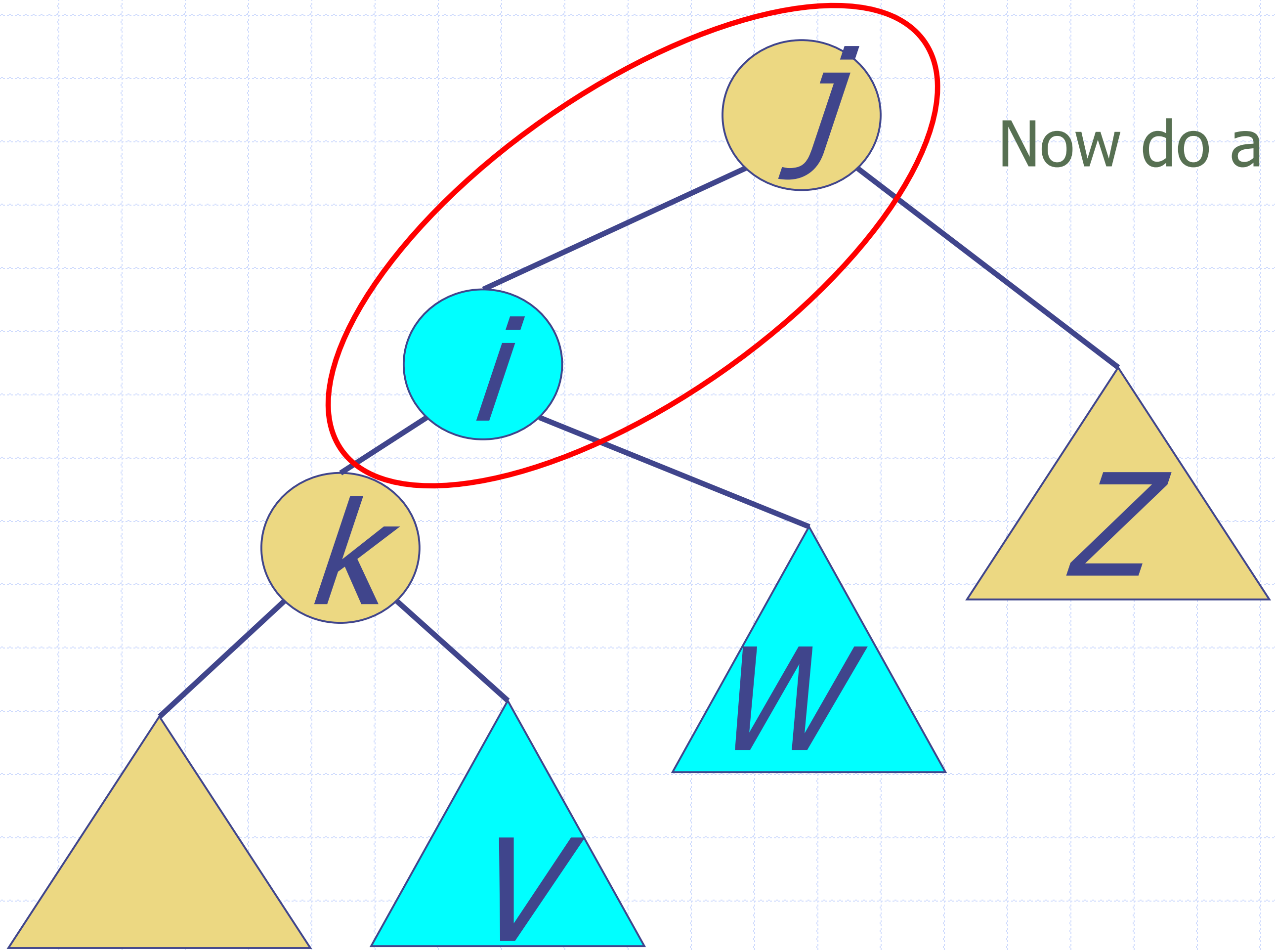



We will do a left-right "double rotation" . . .

# Double Rotation: First Rotation



# Double Rotation: Second Rotation

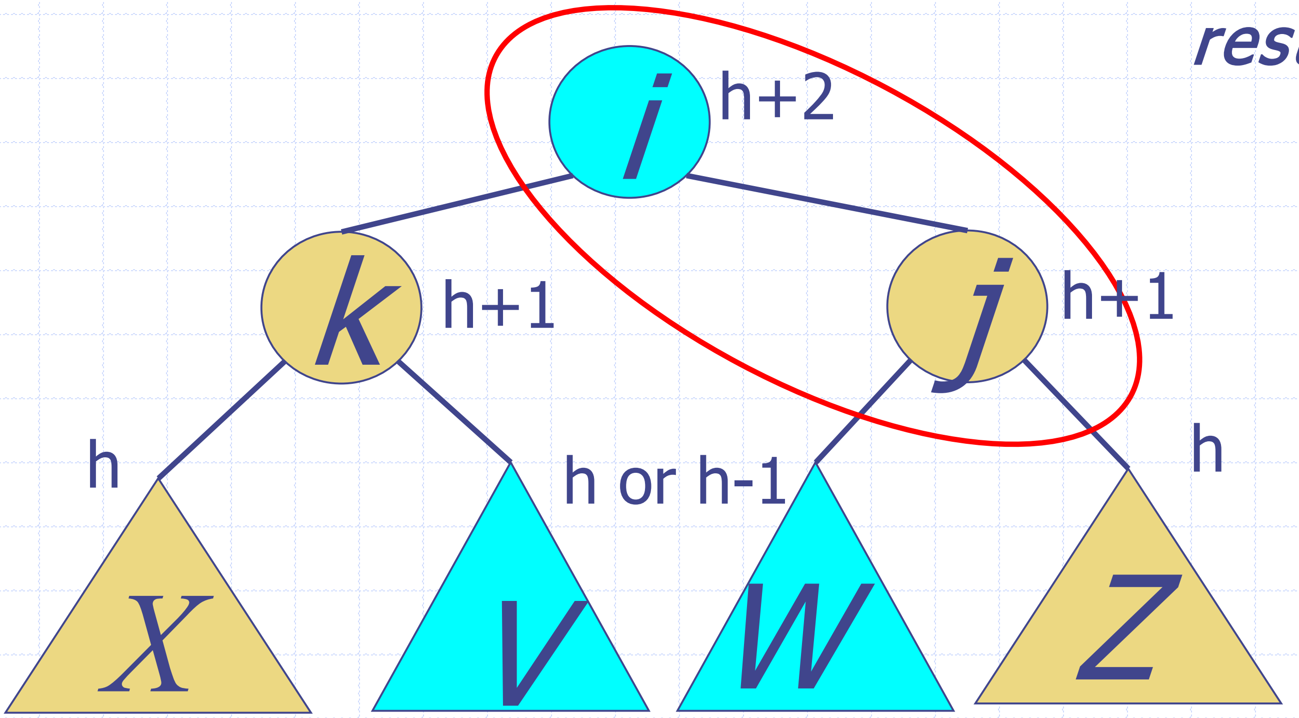



Now do a right rotation

# Double Rotation: Second Rotation

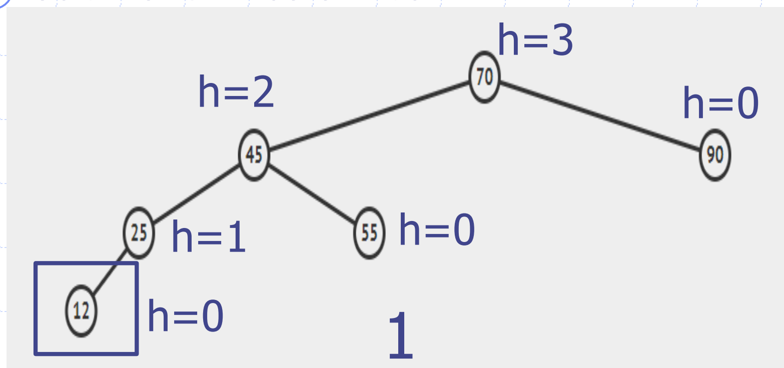
right rotation complete

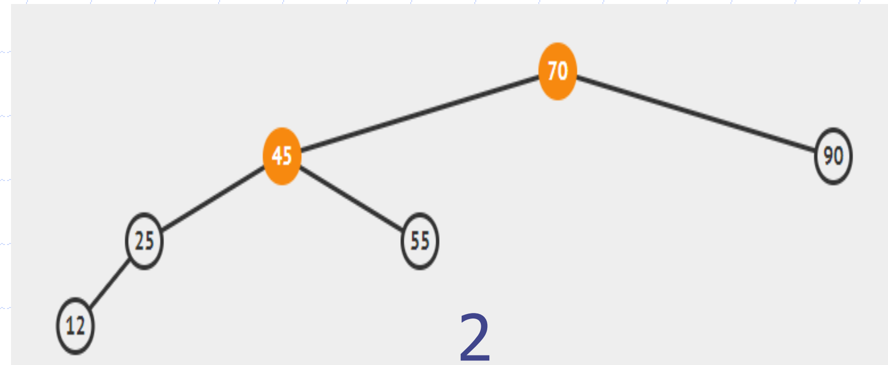
***Balance has been restored***



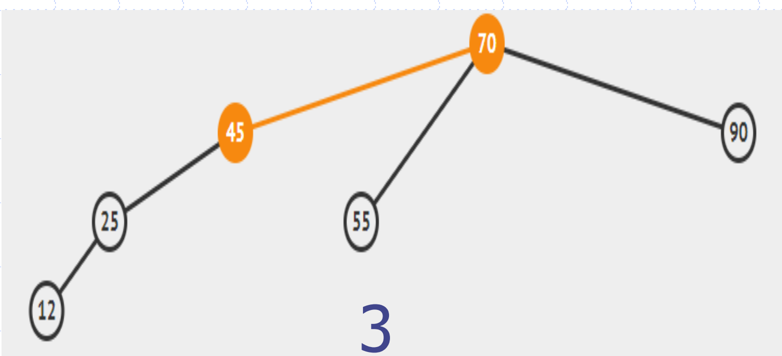
# Inserted 12 (Single rotation)

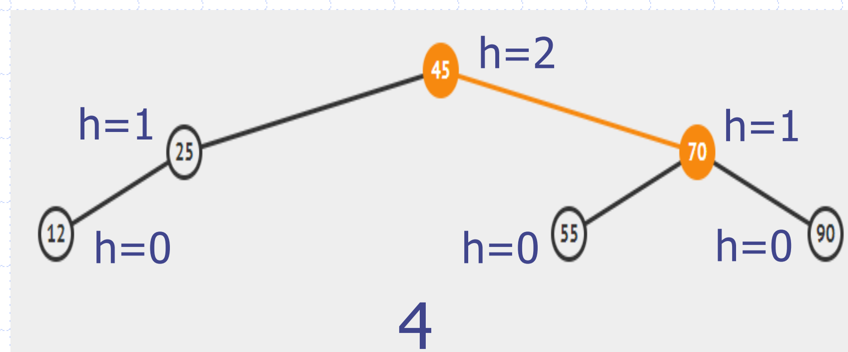
Node 70 is imbalanced




From 70, make two steps deeper (choose the deeper path) and make the tri-nodes in-order: 25, 45, 70 with a straight line. So rotate (promote) node 45 once

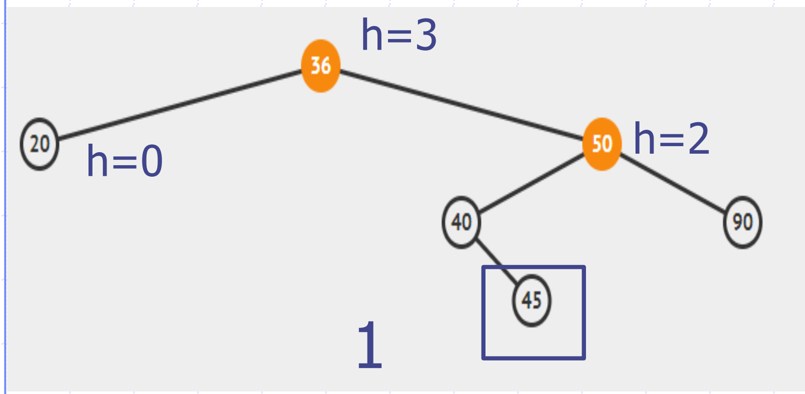


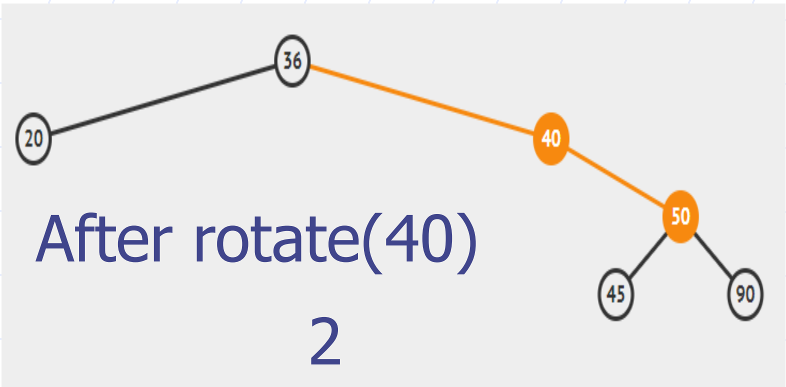




After rotate(45)

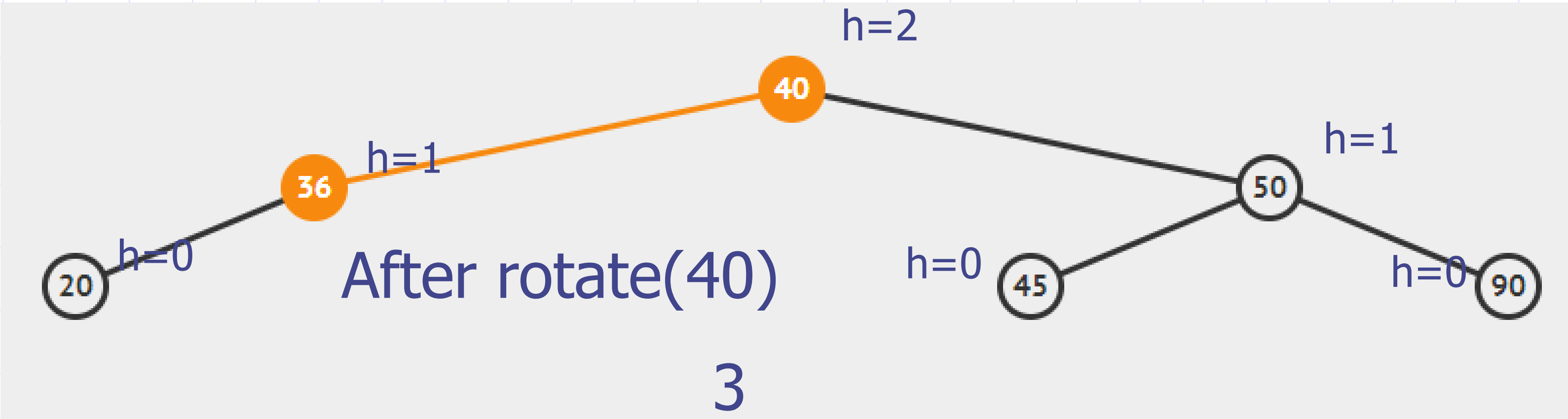
# Inserted 45 (Double Rotation) Rotate right, then left

Node 36 is imbalanced



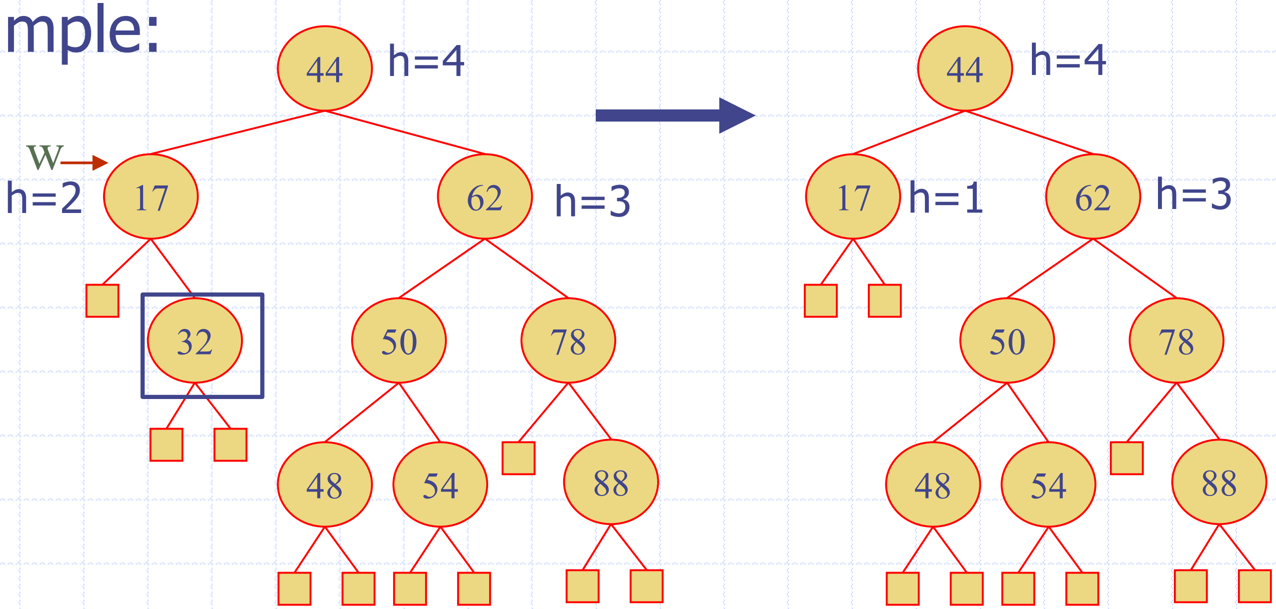




From 36, make two steps deeper (choose the deeper path) and make the tri-nodes in-order: 36, 40, 50 with a non-straight line. So rotate (promote) node 40 twice.



# Removal (Delete 32)

- Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, w, may cause an imbalance.
- Example:

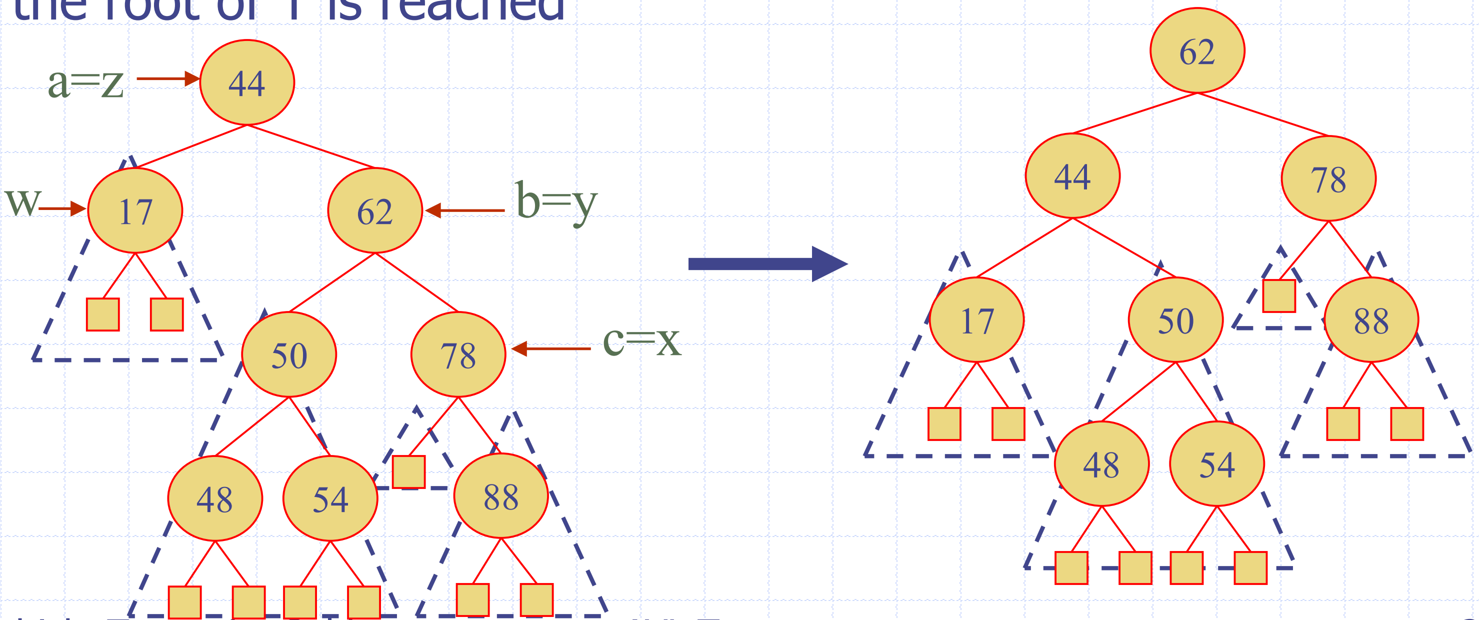



before deletion of 32

after deletion

# Rebalancing after a Removal

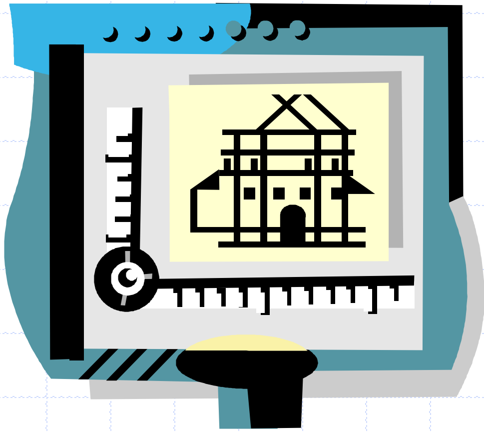
- Let z be the first unbalanced node encountered while travelling up the tree from w. Also, let y be the child of z with the larger height, and let x be the child of y with the larger height
- We perform restructure(x) to restore balance at z
- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached



# Summary

- Find the unbalanced node from bottom up, say rooted at node z
- Take the deeper path from z with 2 steps, forming a path z->y->x
- If z->y->x is a straight line, rotate(y) once
- Otherwise, rotate(x) twice

# AVL Tree Performance



- A single restructure takes O(1) time
  - using a linked-structure binary tree
- Searching takes O(log n) time
  - height of tree is O(log n), no restructures needed
- Insertion takes O(log n) time
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)
- Removal takes O(log n) time
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)

# Implementing _Rotate function in Python

```
def _rotate(self, p):
  """Rotate Position p above its parent.

  Switches between these configurations, depending on whether p==a or p==b.

        b                  a
       / \                / \
      a   t2            t0   b
     / \                    / \
    t0  t1                 t1  t2

  Caller should ensure that p is not the root.
  """
  """Rotate Position p above its parent."""
  x = p._node
  y = x._parent                               # we assume this exists
  z = y._parent                               # grandparent (possibly None)
  if z is None:
    self._root = x                            # x becomes root
    x._parent = None
  else:
    self._relink(z, x, y == z._left)          # x becomes a direct child of z
  # now rotate x and y, including transfer of middle subtree
  if x == y._left:
    self._relink(y, x._right, True)           # x._right becomes left child of y
    self._relink(x, y, False)                 # y becomes right child of x
  else:
    self._relink(y, x._left, False)           # x._left becomes right child of y
    self._relink(x, y, True)                  # y becomes left child of x
```

```
def _relink(self, parent, child, make_left_child):
  """Relink parent node with child node (we allow child to be None)."""
  if make_left_child:                         # make it a left child
    parent._left = child
  else:                                       # make it a right child
    parent._right = child
  if child is not None:                       # make child point to parent
    child._parent = parent
```

# Implementing _restructure function in Python

```python
def _restructure(self, x):
  """Perform a trinode restructure among Position x, its parent, and its grandparent.

  Return the Position that becomes root of the restructured subtree.

  Assumes the nodes are in one of the following configurations:

      z=a                 z=c           z=a               z=c
     /   \               /   \         /   \             /   \
   t0    y=b           y=b   t3      t0    y=c         y=a   t3
        /   \         /   \               /   \       /   \
      t1    x=c     x=a   t2            x=b   t3    t0    x=b
           /   \   /   \               /   \             /   \
          t2   t3 t0   t1             t1   t2           t1   t2

  The subtree will be restructured so that the node with key b becomes its root.

            b
          /   \
        a       c
       / \     / \
      t0  t1  t2  t3

  Caller should ensure that x has a grandparent.
  """
  """Perform trinode restructure of Position x with parent/grandparent."""
  y = self.parent(x)
  z = self.parent(y)
  if (x == self.right(y)) == (y == self.right(z)):   # matching alignments
    self._rotate(y)                                  # single rotation (of y)
    return y                                         # y is new subtree root
  else:                                              # opposite alignments
    self._rotate(x)                                  # double rotation (of x)
    self._rotate(x)
    return x                                         # x is new subtree root
```

# Implementing _relink function in Python

```python
def _relink(self, parent, child, make_left_child):
  """Relink parent node with child node (we allow child to be None)."""
  if make_left_child:                                  # make it a left child
    parent._left = child
  else:                                                # make it a right child
    parent._right = child
  if child is not None:                                # make child point to parent
    child._parent = parent
```

# Python Implementation

```
class AVLTreeMap(TreeMap):
  """Sorted map implementation using an AVL tree."""

  #-------------------------- nested _Node class --------------------------
  class _Node(TreeMap._Node):
    """Node class for AVL maintains height value for balancing."""
    __slots__ = '_height'         # additional data member to store height

    def __init__(self, element, parent=None, left=None, right=None):
      super().__init__(element, parent, left, right)
      self._height = 0            # will be recomputed during balancing

    def left_height(self):
      return self._left._height if self._left is not None else 0

    def right_height(self):
      return self._right._height if self._right is not None else 0
```

# Python Implementation, Part 2

```python
#------------------------ positional-based utility methods ------------------------
def _recompute_height(self, p):
  p._node._height = 1 + max(p._node.left_height(), p._node.right_height())

def _isbalanced(self, p):
  return abs(p._node.left_height( ) − p._node.right_height()) <= 1

def _tall_child(self, p, favorleft=False): # parameter controls tiebreaker
  if p._node.left_height( ) + (1 if favorleft else 0) > p._node.right_height():
    return self.left(p)
  else:
    return self.right(p)

def _tall_grandchild(self, p):
  child = self._tall_child(p)
  # if child is on left, favor left grandchild; else favor right grandchild
  alignment = (child == self.left(p))
  return self._tall_child(child, alignment)
```

Find a deep path: z -> y -> x

# Python Implementation, end

```
def _rebalance(self, p):
  while p is not None:
    old_height = p._node._height          # trivially 0 if new node
    if not self._isbalanced(p):           # imbalance detected!
      # perform trinode restructuring, setting p to resulting root,
      # and recompute new local heights after the restructuring
      p = self._restructure(self._tall_grandchild(p))
      self._recompute_height(self.left(p))
      self._recompute_height(self.right(p))
    self._recompute_height(p)             # adjust for recent changes
    if p._node._height == old_height:     # has height changed?
      p = None                            # no further changes needed
    else:
      p = self.parent(p)                  # repeat with parent

#------------------------- override balancing hooks -------------------------
def _rebalance_insert(self, p):
  self._rebalance(p)

def _rebalance_delete(self, p):
  self._rebalance(p)
```

Left and right subtree got changed. So recompute heights for them

Break the while loop

Continue to maintain balance for parent(p) since height(p) was changed